

Real-Time Road Traffic Management and Update System

Applying SOLID Principles and Design Patterns



Matthias Gaillard and Sara Pereira

Table des matières

Introduction	3
Context	3
Objectives	3
System Overview	4
General Description	4
Architecture.....	4
Server package.....	4
Client package	8
Concurrency Management	9
Design Decisions.....	10
Application of SOLID Principles	10
Single Responsibility	10
Open/Closed	10
Liskov Substitution	10
Interface Segregation.....	10
Dependency Inversion	10
Design Patterns	11
Chain of Responsibility	11
State	11
Memento	11
Abstract	11
Template.....	11
Class diagram.....	12
Implementation Details	13
Results and Evaluation	14
Connection Screen	14
Registration Screen.....	14
End users screen	15
Server screen	15
Administrator users screen	16
Server screen	16
Conclusion	17

Introduction

Context

In today's cities, effective traffic management is crucial. People's daily travel is impacted by traffic congestion and road problems. These issues can be resolved with the aid of a system that offers real-time travel time updates and recommends the most efficient routes.

Such a system is simulated in this project. Roads are shown as connections (edges) and intersections as points (nodes) in a graph representation of the network. Travel time can be updated by admin users according to actual circumstances, such as traffic jams or accidents. Additionally, the system uses the Dijkstra algorithm to determine the shortest path between two points.

The project adheres to SOLID principles to facilitate expansion and maintenance. Its client-server architecture enables simultaneous connections from numerous users. This maintains data consistency and guarantees real-time updates. The system is comparable to what is used in GPS navigation.

Objectives

1. Show the road network as a graph with roads as connections (edges) and intersections as points (nodes), with travel times allocated to each road.
2. Give admin users the option to dynamically modify travel times in response to actual occurrences such as traffic bottlenecks, collisions, or road closures.
3. Determine the shortest path and shortest path value between two points in the network using an algorithm like Dijkstra.
4. Make sure the code is simple to scale, expand, maintain and uses SOLID principles.
5. Put in place a client-server architecture in which clients submit updates and queries to the server, which maintains the graph and handles updates.
6. Manage several users simultaneously by making sure the system maintains data consistency and controls concurrency.

System Overview

General Description

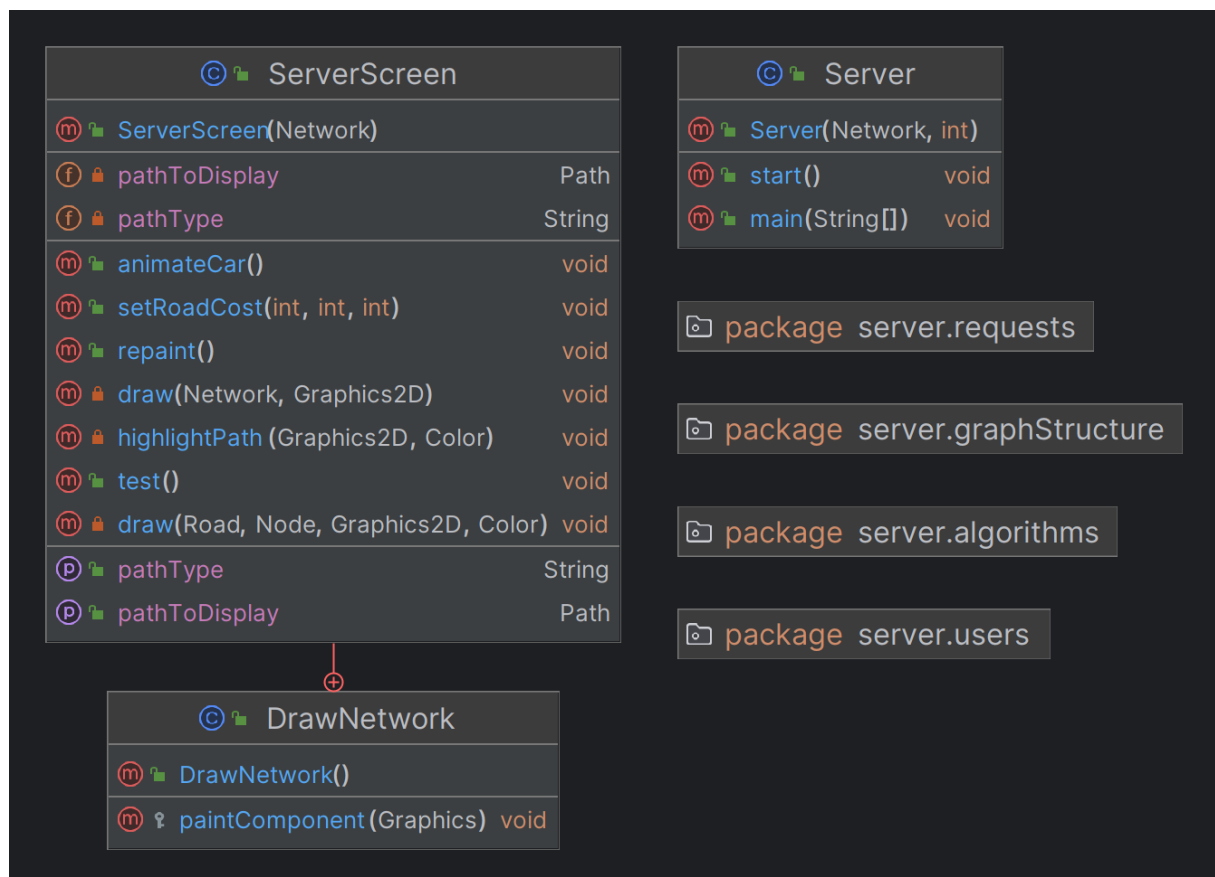
The client-server architecture of the system is intended to replicate real-time traffic management. Roads are represented as edges with travel times assigned to them, and intersections or road points are represented as nodes in the graph model of the road network.

Clients can request the shortest path and shortest path value between two points or update travel times by connecting to the server. To guarantee that the graph always shows the most recent road conditions, the server processes these updates in real-time. Additionally, it determines the shortest path based on the most recent travel times using effective algorithms like Dijkstra's.

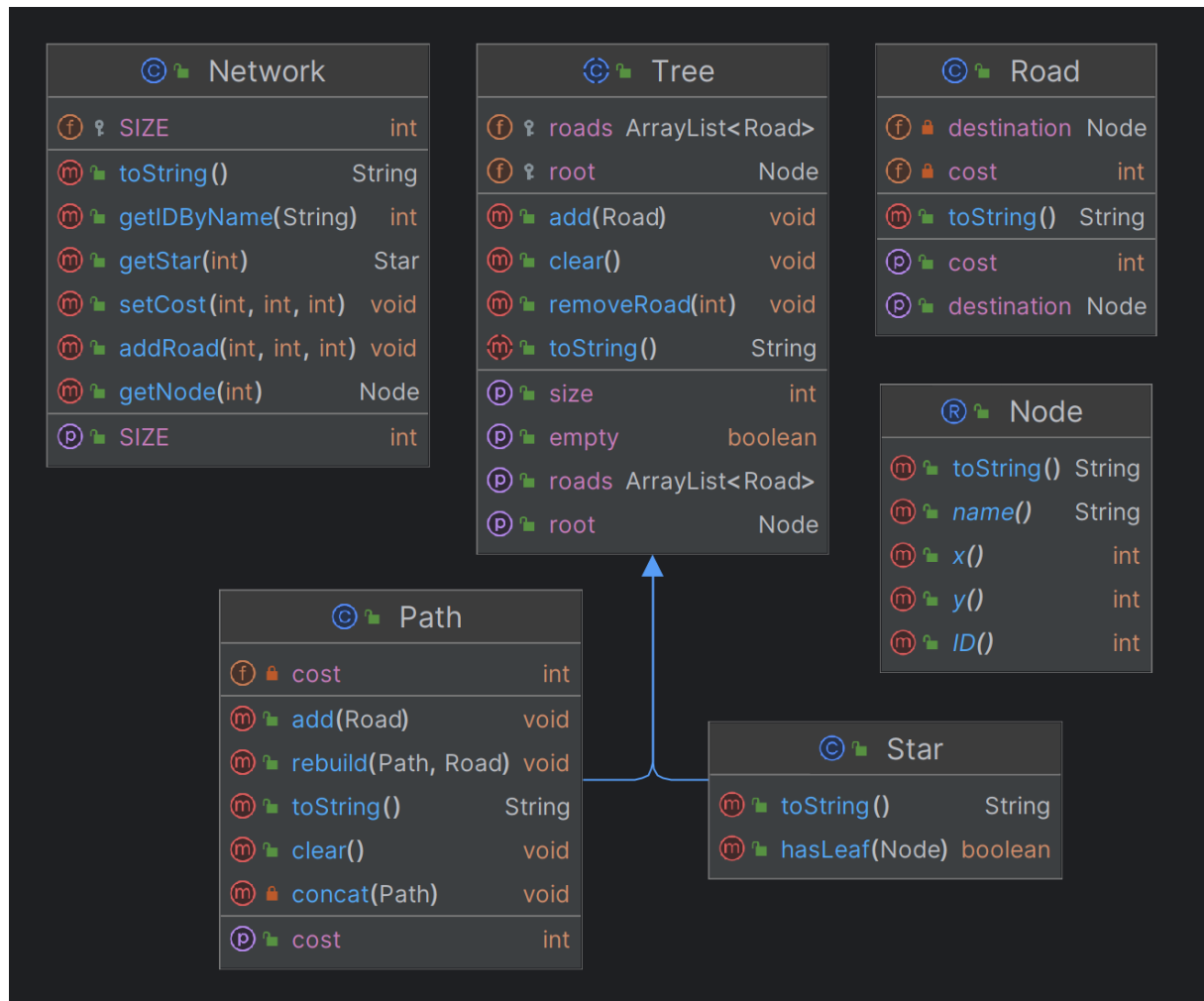
Architecture

For effective traffic management, the system uses a client-server architecture. In the following sections, we will explain the structure of the different classes used in our project.

Server package



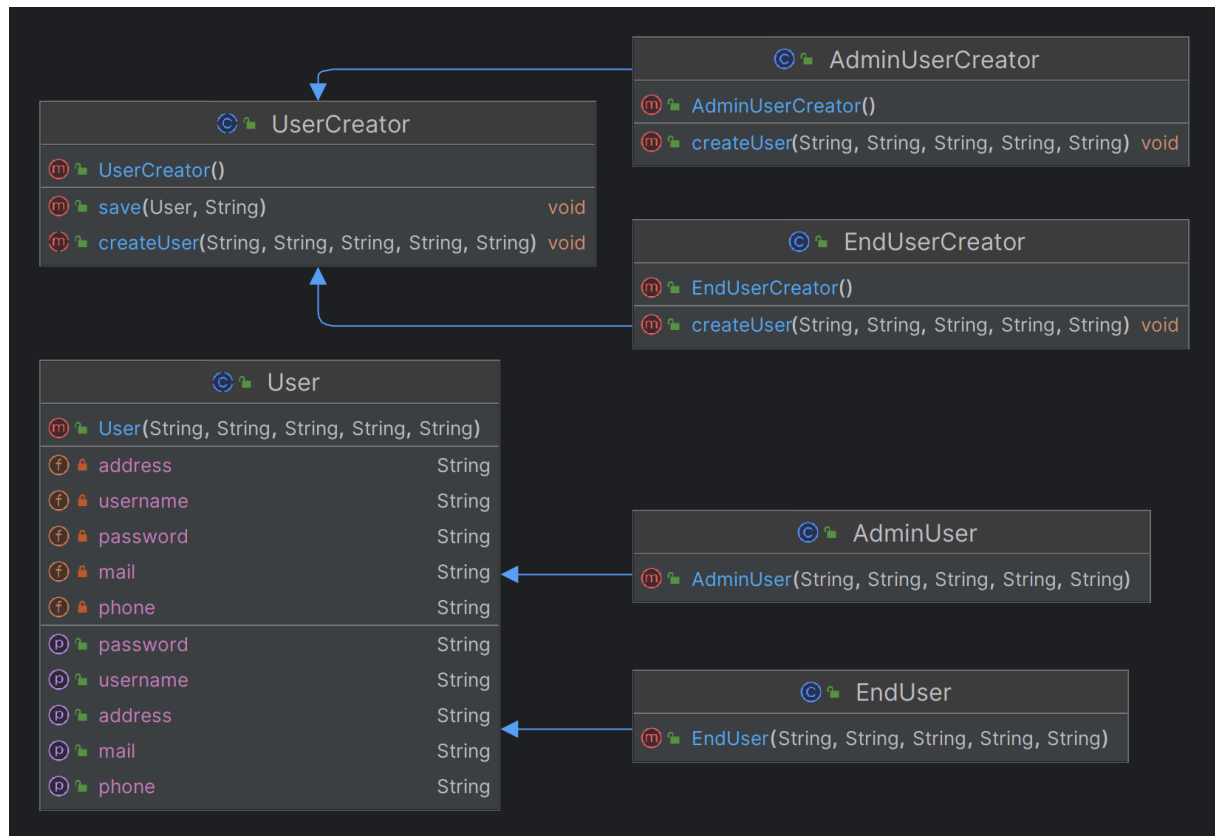
This package is where most of the code is located. Its classes will oversee a lot of functionalities. The main class here is the `Server` class, which will need to be run when we want to use the application. When run, it displays the current network in a JFrame, using the `ServerScreen` class as its content panel.



This package contains all necessary classes to store our graph data structure :

- The smallest element of our graph is the vertex, represented by the `Node` record (name, id and coordinates)
- An edge between nodes is represented by the `Road` class. It contains the destination node and the road cost.
- `Tree` is an abstract data structure to store a collection of roads with a source node. This class has two subclasses :
 - `Star` is a `Tree` such that each road is incident with the source node. It is used in the graph structure.
 - `Path` is a `Tree` such that each road is incident to the previous one, and the first one is incident to the source node. This structure is used to compute and store shortest paths using Dijkstra algorithm.
- `Network` uses an array of `Star` to describe the neighborhood of each `Node`. With only that, we can store every information needed about the graph.

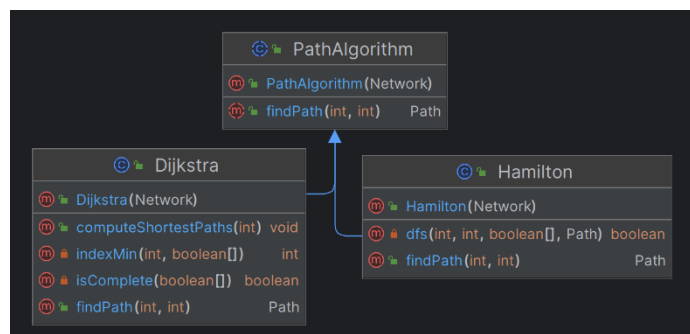
Users



In this package, we will define the structure of an account in the `User` class. Normal users are represented with the `EndUser` class and have restraint rights. Administrative users are represented with the `AdminUser` class and have more rights than end users. We additionally have different creator classes to create the users and store them in JSON files.

Algorithms

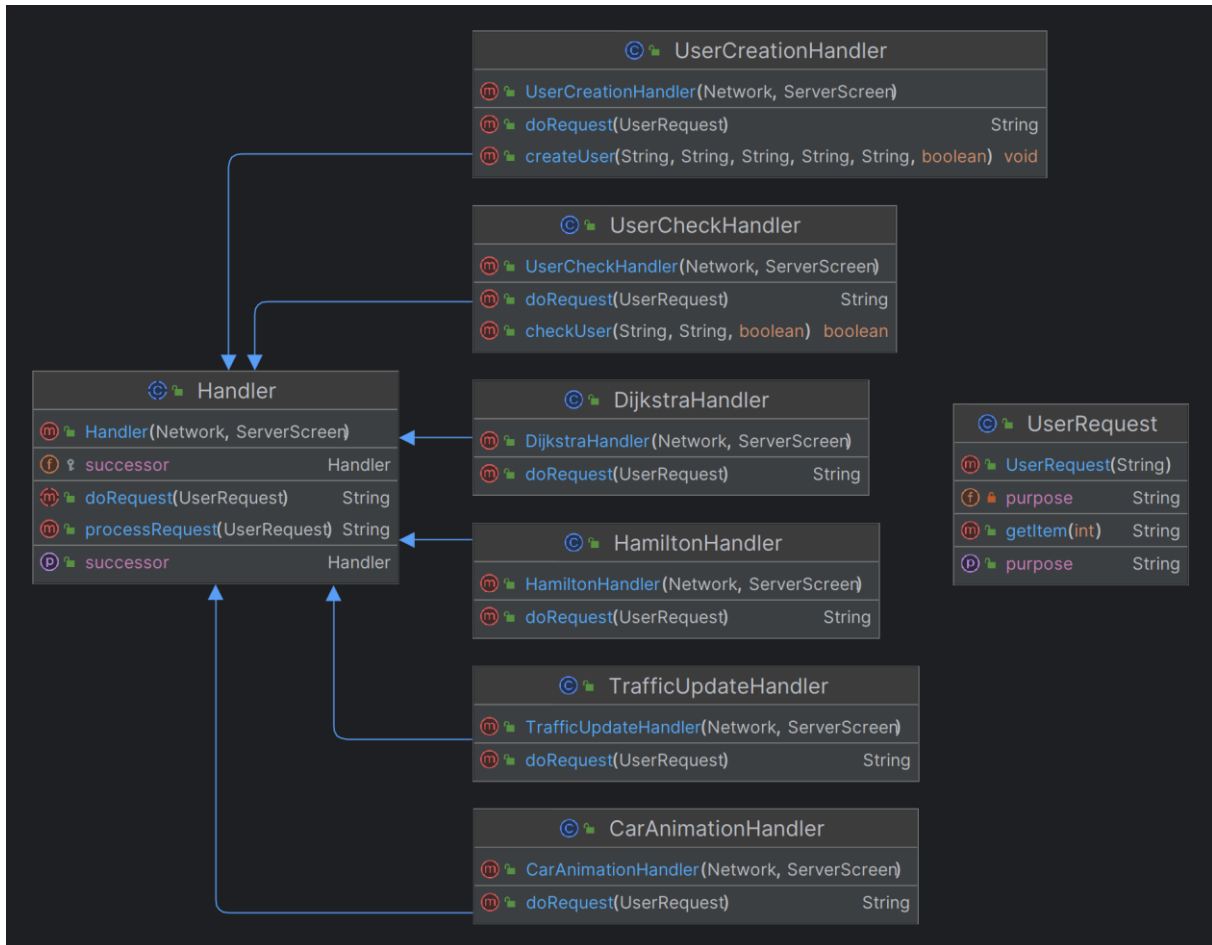
This package contains a super class `PathAlgorithm`, which will take a `Network` as parameter in its constructor and has a method to search for paths. The actual method is implemented in its subclasses which are specific algorithms. In our case, we implemented the following ones :



`Dijkstra` subclass will compute shortest path calculations

`Hamilton` subclass will search for a Hamiltonian path between two nodes. (That is, a path going through each vertex exactly once.) It could be useful for busy people who want to do a tour of a country, visiting every location in the process but having no time to lose !

Requests



In this package, we implemented the management of all the types of requests the client can ask the server :

`UserCreationHandler` will handle the creation of new end or admin users inside a JSON file.

`UserCheckHandler` class will check inside a JSON if there already exists an element with a given username and password.

`DijkstraHandler` will search the shortest path between two given nodes and draw it in blue.

`HamiltonHandler` will search a Hamiltonian path between two given nodes and draw it in orange.

`TrafficUpdateHandler` will update a given road cost to a new cost and show the change in the `ServerScreen`.

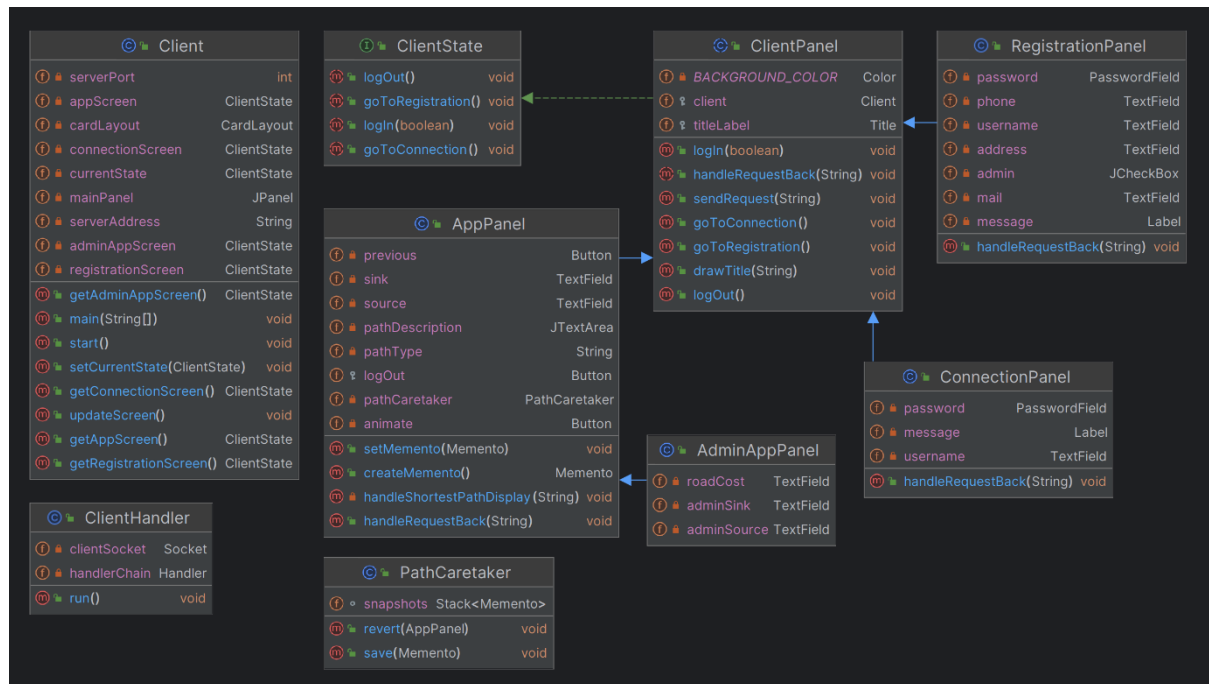
`CarAnimationHandler` will display a little animation of a car going through a searched path in the network.

The concrete request is represented by the `UserRequest` class, which contains a `purpose` (type) and a list of items which are the parameters of the request (nodes, new cost to update...)

Client package

The `Client` class is the second and last runnable class of the project. After we ran the server, we can run it to display the graphic interface the client can use to send queries. The `Client` class is itself a `JFrame` that can display several panels according to the situation.

Panels



In this package, we have the different screens that can be displayed to the client :

`ClientState` is an interface that allows us to switch from one state (panel) to another. It is part of the state pattern.

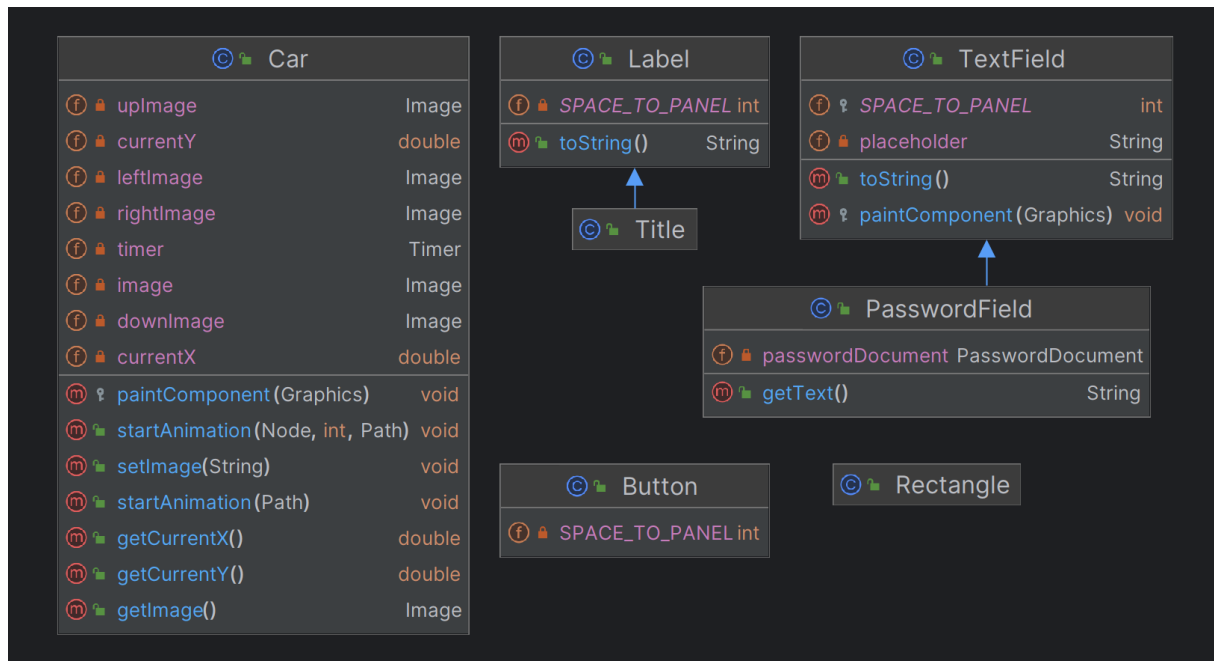
`ConnectionPanel` displays a connection screen. If the user has already an account, he can authenticate himself and use the application. Otherwise, he needs to sign up.

RegistrationPanel displays a sign-up screen. The user can create his account by filling the different text fields. If he wants to have all rights, he can click the “admin” checkbox.

AppPanel displays the application interface to the client. He can ask the server the shortest path between two nodes by filling the form.

`AdminAppPanel` inherits `AppPanel` and displays a form to update a road cost to a new one.

PathCaretaker is used in AppPanel and will store information about previous searched paths by the user, to restore them later.



In this package, we have classes for all small elements in our panels, especially customized JComponents who come from JavaSwing.

- Button is a customized JButton
- Label and Title are customized JLabel
- TextField is a customized JTextField
- PasswordField is a customized JPasswordField
- Rectangle is a customized JPanel
- Car is a JPanel containing an image of a car.

Concurrency Management

The server effectively handles and processes several client requests at once by utilizing Java's concurrency tools, such as threads. This guarantees continuous communication by enabling multiple clients to send requests simultaneously without blocking one another. By putting these tools into practice, the server can manage multiple connections at once, handle many requests, optimize resource usage, and make it work better, becoming scalable and responsive.

`ClientHandler` class will manage the different clients by assigning them a separate thread. It ensures that their requests are processed concurrently without blocking other connections.

`ClientPanel` and its subclasses will send client requests to the server. They will initialize communication and ensure that messages are dispatched correctly for processing.

Design Decisions

Application of SOLID Principles

Single Responsibility

This principle is respected because each class has its own job. Indeed, the only class that can store and modify graph data is the `Network` class. `Dijkstra` class handles path research. And for another example, `Handler` class was specifically made to handle requests.

Open/Closed

The `Dijkstra` subclass and `PathAlgorithm` superclass can be used to observe this principle. It is possible to add new algorithms, such as A^* , as subclasses without altering the current classes. Scalability is encouraged by this modular architecture. Besides, this principle is also respected by the `Handler` and `ClientPanel` superclasses. By adding subclasses that redefine or extend behavior, they enable the system to be expanded with new request types or client panels without changing the current implementation.

Liskov Substitution

Any of its subclasses, such as `Dijkstra`, can be used in place of the `PathAlgorithm` without affecting how the program behaves. This enables the system to be expanded in the future using various pathfinding techniques. The principle also applies to other inheritance relations we have in our project. For instance, an `AppPanel` or `AdminAppPanel` object, which are subclasses of a general `ClientPanel` superclass, can be used by the `Client` as `ClientPanel` objects.

Interface Segregation

We almost did not use a single interface in our code, so the principle is not applicable in theory. However, we do have multiple graphic interfaces represented by the `ClientPanel` classes. Instead of having one interface for using the GPS, we decided to split it into two different classes, `AppPanel` for end users and `AdminAppPanel` for admin users, since these types of users have different needs. We thus have sort of an interface segregation.

Dependency Inversion

Rather than using tangible implementations, high-level components rely on abstractions. For instance, it is simpler to add new request types because the server communicates with the `Handler` abstract class rather than concrete handler subclasses. The same can be said for the `ClientPanel` class (and its subclasses), who implements the `ClientState` interface to change its visual content.

Design Patterns

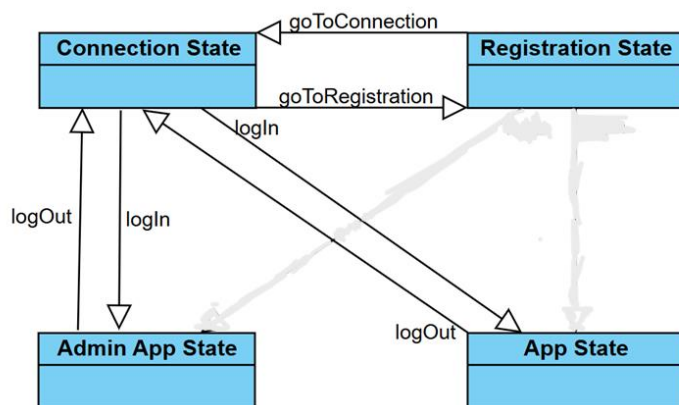
Chain of Responsibility

We [previously](#) described the multiple handlers in the requests package. These classes, in addition to the `UserRequest` class, implement this pattern. The `Server` has a chain of six `Handler`, one for each subclass. Here are the different steps for a query to be executed :

- The client prepares a request in `String` format and gives it to the `ClientHandler` class.
- It will transform it into a `UserRequest` format and send it to the `Server`.
- The `Server` will transfer to its first `Handler` (`UserCreationHandler`).
- It will check if the request is really about user creation by looking at its purpose.
 - If it is, it does the request and returns whatever needed result to the client.
 - Otherwise, it sends the request to its successor `UserCheckHandler`, and the chain continues until the right `Handler` is found.

State

The state pattern is used to manage the current client panel state. Every state represents a distinct phase of the client's engagement with the system (with its respective `JFrame`). We can move from one panel to another using methods from the `ClientState` interface, according to the state diagram on the right. If needed, more information about these classes can be found [here](#).



Memento

We can click on the "Previous" button to restore the last path that we looked for. The originator `AppPanel` class has a `Memento` inner class and can save instances of it inside the `PathCaretaker` class. The stored elements are the source and sink (destination) node, as well as the type of path (Dijkstra or Hamiltonian). When we restore the last saved `Memento`, the text fields are filled with the corresponding source and sink node names, and a Dijkstra or Hamilton request is sent to the `Server`. This way, the path is also visually restored in the `ServerScreen`.

Abstract Factory

We used this pattern to create the two types of users we need : end users and admin users. We thus have an abstract factory `UserCreator` whose concrete subclasses are `EndUserCreator` and `AdminUserCreator`.

Template

The `processRequest` method in the `Handler` superclass defines the common structure for handling requests. By redefining the `doRequest` abstract method called in `processRequest`, the subclasses override steps while maintaining the general structure that the superclass provides. This enables customization in each subclass while assuring that the core logic stays consistent, allowing the system to adhere to the template pattern.

Class diagram

Here is the overall UML class diagram for the 44 classes of our project. You may have a look at the [Architecture](#) section for more details.

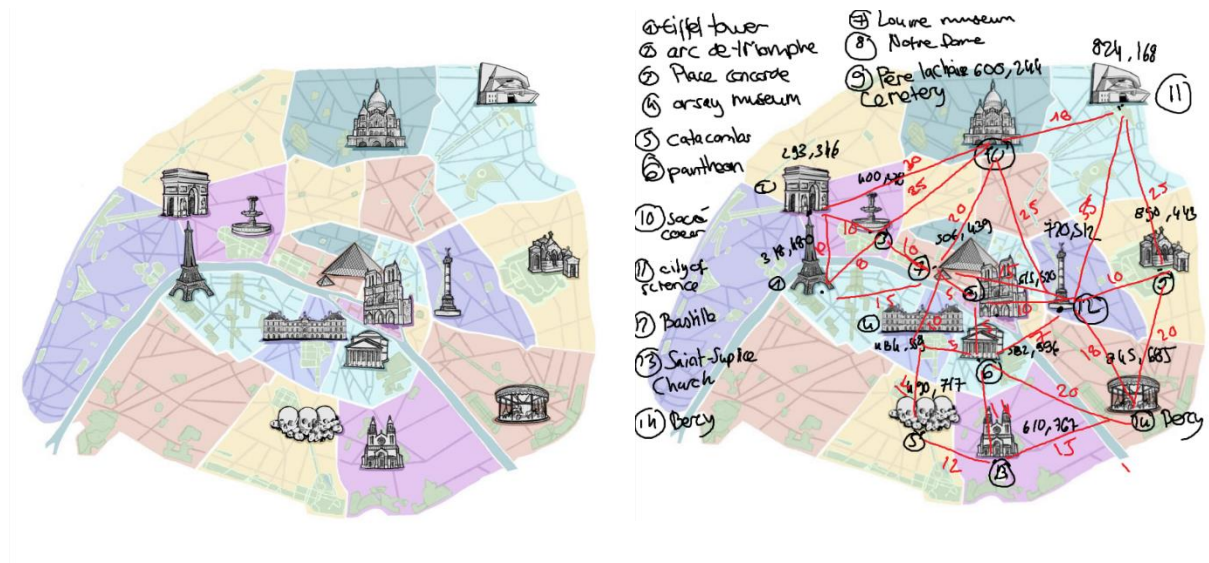


Implementation Details

We used JavaSwing to have a graphic interface for our application, both for the client (to make queries) and for the server (to dynamically display the graph)

Java sockets are used by the system to facilitate client-server communication. Dijkstra's algorithm is used to calculate the shortest paths, and in order to replicate real-world circumstances, network data is dynamically loaded from hardcoded data. Every user request is handled instantly.

We made the network based on the following maps :



We were able to determine the coordinates of each node thanks to this map. Knowing the graph and which cities are connected to which cities was also helpful.

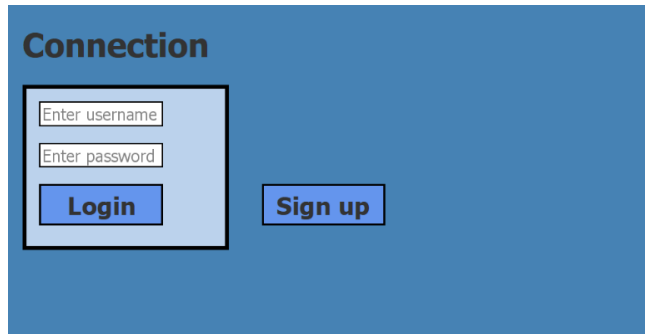
Note : we removed the road from Notre Dame to Bastille for better graph readability.

Results and Evaluation

Connection Screen

If you already have an account, you can log in on the GPS application's first screen.

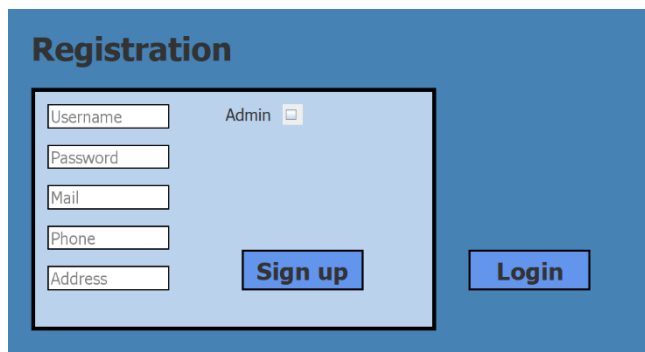
End users can find the shortest path between two locations, while **admin users** are able to do the same, but also change the cost between nodes.

The image shows a 'Connection' screen with a blue background. It features a light blue rectangular box containing two input fields: 'Enter username' and 'Enter password'. Below these fields are two buttons: 'Login' and 'Sign up'.

You will be successfully logged in if your username and password are correct and you will be able to use the GPS. If not, you will see an error message informing you that something went wrong.

If you do not have an account, by clicking on the sign-up button, it will open the registration screen and give you the possibility to create an account.

Registration Screen

The image shows a 'Registration' screen with a blue background. It features a light blue rectangular box containing several input fields: 'Username', 'Password', 'Mail', 'Phone', and 'Address'. To the right of the 'Username' field is an 'Admin' checkbox. Below the input fields are two buttons: 'Sign up' and 'Login'.

You can create an account if you do not already have one by entering details like your address, phone, number, email and address.

Additionally, there is a checkbox that allows you to select whether you want to be an admin user. If you select it, you will become an admin.

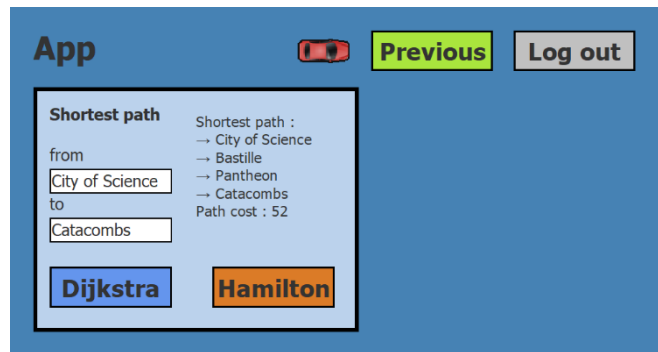
Clicking **Sign up** will create your account. It will show you an error message if your username is already used. Clicking **Login** will redirect you to the connection screen.

When you create your account, it will automatically open the GPS main screen.

End users screen

This screen will show up when you log in as an end user.

By entering the source in the first node input and the destination in the second node input, an end user can find the shortest path between two locations. The shortest path will appear to the right of the title when you click **Dijkstra**. A path that visits every point on the map between the two locations you have chosen will be shown (if it exists) if you click on **Hamilton**. As I will explain later, the graph will also be displayed on the server screen.



Additionally, there are **three buttons** :

- **Car button** : Makes the car move as it takes the path shown on the graph.
- **Previous** : Brings up the previous search's path. (Dijkstra or Hamilton)
- **Log out** : Redirects you to the login screen.

Server screen

As explained previously, various actions will be shown on the server screen, a separate window, when you click on the buttons I described.

- When **Dijkstra** is clicked, the shortest path will show up in blue.
- When you click on the **Car**, a car will travel the path.
- Lastly, as previously mentioned, clicking on **Hamilton** will show a path that passes by each location on the map.

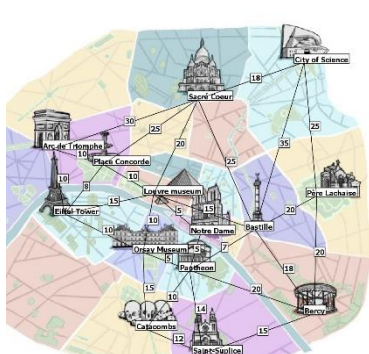


Figure 3 : The graph

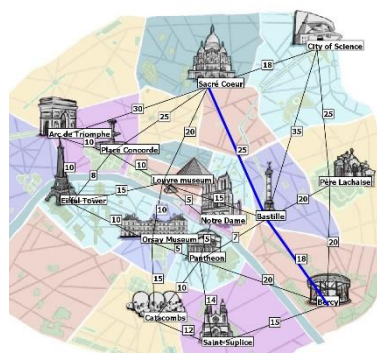


Figure 2 : Dijkstra path example

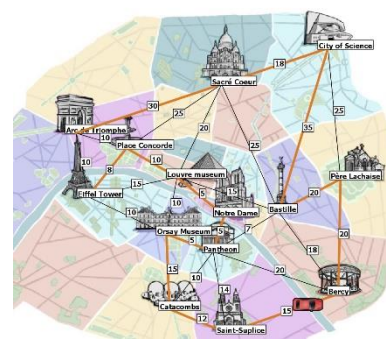
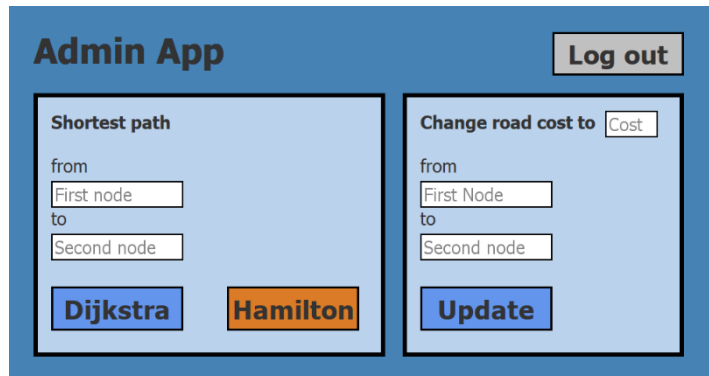


Figure 1 : Hamilton path example

Administrator users screen

When you log in as an administrator, this screen will show up.

You can alter the road cost, which is an extra feature, but otherwise everything is the same as for the end user. The road you wish to alter does not need to exist in the first place. In that case, a new road will be created and displayed. Beware, the graph might become complicated to read in the server screen...



Put the new cost in the new cost input field, the source in the first node input, and the destination in the second node input. The road cost will be updated upon selecting **Update**, and the graph will show the modifications.

Server screen

As said before, in this instance, we can modify the road cost by selecting Update. The new road cost will be instantly reflected in the graph.

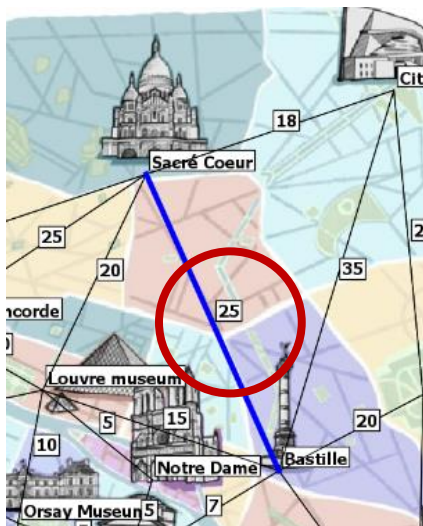


Figure 4 : Before the update

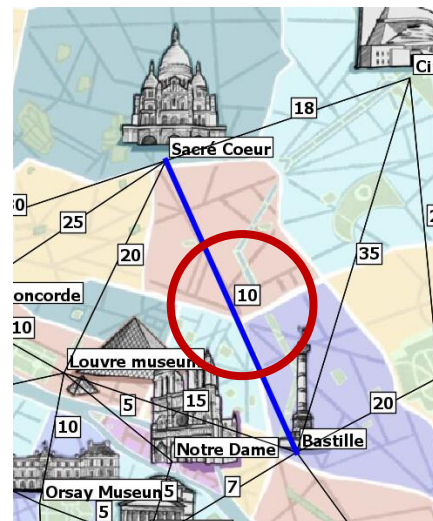


Figure 5 : After the update

Conclusion

To sum up, this project effectively illustrates how to implement a real-time traffic management system that follows SOLID guidelines and integrates essential design patterns. The system's capacity to effectively handle multiple concurrent user requests is guaranteed by the implementation of a client-server architecture. Dijkstra's algorithm and the graph-based depiction of road networks allow for precise and dynamic shortest path computations.

The system maintains its flexibility, extensibility, and maintainability through the implementation of design principles like Dependency Inversion, Open/Closed, and Single Responsibility. Design patterns like Chain of Responsibility, State, and Memento are integrated to increase the system's scalability and resilience.

This project has useful ramifications for GPS navigation and traffic control in the real world. Future improvements, like adding more path-finding algorithms or new UI features, are made possible by its modular architecture.