**Traffic Management Application**

**Abstract requirements definition**

The context of the project is traffic management in a city (>100.000 inhabitants).

You are going to design an application that provides an innovative approach to one of the many problems in the traffic management. You choose the topic of your specific project, for example managing the time schedule of public transportation.

While you are free to choose your specific topic, your project shall fulfill the following requirements:

* The estimated effort for the implementation of the project is 20 man-years and the delay is less than a year, i.e. you can work with at least 20 people in parallel.
* You need a more complex software architecture than a monolithic program.
* At least some part of the project requires a Human-Machine Interface that can be implemented with a touch-screen GUI interface following the guidelines on utility, usability, UX and QoE.
* At least one of project's entities are controlled by two input sources (i.e. the arrival time of the bus depends both on the traffic simulation (source 1) and the preplanned schedule (source 2)).
* Your project is suitable for implementing one set of design patterns. The sets are detailed below.

Choose one of the following sets of design patterns. You have to implement *each* design pattern *at least once* in your project.

* Set 1: Composite, Memento, Builder, Adapter, Observer
* Set 2: Visitor, Abstract Factory, Interpreter, Singleton, Interceptor
* Set 3: Chain of Responsibility, Decorator, Factory, Proxy, Flyweight
* Set 4: Active Object, Monitor Object, Composite, Decorator, Strategy

Interpretation the abstract requirements

**3. A concise explanation of the particular software architecture (i.e. n-tier architecture, storage location, the set of design patterns chosen)**

**Software Architecture**

In this project we will use 3-tier application architecture which is a modular [client-server](https://searchnetworking.techtarget.com/definition/client-server) architecture that consists of a presentation tier, an application tier and a data tier. The data tier stores information, the application tier handles logic and the presentation tier is a graphical user interface ([GUI](https://searchwindevelopment.techtarget.com/definition/GUI)) that communicates with the other two tiers. The three tiers are logical, not physical, and may or may not run on the same physical server.

**Presentation tier** - This tier, which is built with [VUE](https://searchmicroservices.techtarget.com/definition/HTML5), cascading style sheets ([CSS](https://www.theserverside.com/definition/cascading-style-sheet-CSS)) and Nativescript, is deployed to a computing device through a web-based application. The presentation tier communicates with the other tiers through application program interface ([API](https://searchmicroservices.techtarget.com/definition/application-program-interface-API)) calls.

**Application tier**- The application tier, which may also be referred to as the logic tier, is written in a programming language such as [Java](https://www.theserverside.com/definition/JavaScript) and contains the [business logic](https://whatis.techtarget.com/definition/business-logic) that supports the application’s core functions. The underlying application tier can either be hosted on distributed servers in the cloud or on a dedicated in-house server, depending on how much processing power the application requires.

**Data tier**- The data tier consists of a database and a program for managing read and write access to a database. This tier may also be referred to as the storage tier and can be hosted on-premises or in the cloud. Popular database systems for managing read/write access include [MySQL](https://searchoracle.techtarget.com/definition/MySQL), PostgreSQL, Microsoft SQL Server and [MongoDB](https://searchdatamanagement.techtarget.com/definition/MongoDB).

**4. A detailed explanation of the usefulness of each design pattern**

In this project we will use set 2: Visitor, Abstract Factory, Interpreter, Singleton, Interceptor.

**Abstract Factory:** **Provide an interface for creating families of related or dependent objects without specifying their concrete classes**

* a system should be independent of how its products are created, composed and represented.
* a system should be configured with one of multiple families of products.
* a family of related product objects is designed to be used together and you need to enforce this constraint.
* you want to provide a class library of products and you want to reveal just their interfaces not their implementations.

The **Abstract Factory (*AbstractFactoryTrafficParticipant* )** interface declares a set of methods for creating each of the abstract products.

* It declares an interface for operations that create abstract products such as *PhysicalParticipantFactory* and *LogicalParticipantFactory*.

**Concrete Factories(***PhysicalParticipantFactory*, *LogicalParticipantFactory***)** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.

* It implements the operations to create concrete product objects such as Privileged and NotPrivileged.

**Abstract Products(***Car***,** *Motorcycle, Truck, Bus***):** declare interfaces for a set of distinct but related products which make up a product family.

* It declares an interface for a type of Privileged and NotPrivileged object.

**Concrete Products**(*PhysicalPrivilegedCar, LogicalPrivilegedCar, PhysicalNotPrivilegedCar, LogicalNotPrivilegedCar and etc.*): are various implementations of abstract products, grouped by variants. Each abstract product must be implemented in all given variants.

* It defines defines privilegad and not privileged objects to be created by the corresponding concrete factory
* It extends the *PhysicalParticipantFactory* and *LogicalParticipantFactory* abstract class.

## **Advantages of Abstract Factory Design Pattern**

There are following benefits of using abstract factory design pattern.

We can be sure that the products you’re getting from a factory are compatible with each other.

* We avoid tight coupling between concrete products and another modules.
* Single Responsibility Principle: We can extract the product creation code into one place, making the code easier to support.
* Open/Closed Principle: We can introduce new variants of products without breaking existing usages.

In this project, we had a problem with dividing privileged (polices, ambulance) and not privileged(normal driver, bus) vehicles. Polices and the normal driver has the almost same physical and logic properties but police cars have additional features (lights, labels) and privileged role in the roads. For instead of the doubling physical and logic features I used there abstract factory with divided physical and logic factory which have privileged and not privileged version.

Overall developers can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

**Visitor pattern: is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.**

The ***Visitor******(****ShortestRouteVisitor****)*** interface declares a set of visiting methods that can take concrete elements of an object structure as arguments. These methods may have the same names if the program is written in a language that supports overloading, but the type of their parameters must be different.

Each ***Concrete Visitor (****NotPrivilegedShortestRoute****,*** *PrivilegedShortestRoute****)*** implements several versions of the same behaviors, tailored for different concrete element classes.

The ***Element******(****LogicalParticipantFactory****)*** abstract class declares a method for “accepting” visitors. This method should have one parameter declared with the type of the visitor interface.

Each ***Concrete Element (*** *LogicalPrivilegedCar, LogicalNotPrivilegedCar, LogicalPrivilegedMotorcycle, LogicalNotPrivilegedMotorcycle, and etc.****)***must implement the acceptance method. The purpose of this method is to redirect the call to the proper visitor’s method corresponding to the current element class. Be aware that even if a base element class implements this method, all subclasses must still override this method in their own classes and call the appropriate method on the visitor object.

The ***Client*** usually represents a collection or some other complex object. Usually, clients aren’t aware of all the concrete element classes because they work with objects from that collection via some abstract interface.

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. The point of the visitor pattern is to replace the "huge switch" with polymorphism by handling the visitor invocation on the individual subclass.

**Interpreter pattern:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

**Singleton pattern:** is one of the Gangs of Four Design patterns and is available in the Creational Design Pattern category. To apply Singleton pattern, one can opt different approaches but most of them have following common concepts.

* Private constructor to restrict instantiation of the class from different classes.
* Private static variable of the similar class that is the only instance of the class.
* Public static system that returns the instance of the class, which is the global access point for external world and acquire the instance of the singleton class.

Singleton pattern is one of its kinds of a design solution where an application require having one and only one instance of any class, in every possible circumstance. This type of design pattern comes under creational pattern and provides an ideal way to make an object. An important aspect is that it contains a single class that helps in creating an object and help in getting access to its only object of the class.

In my project I used Singleton to implement Road signs which will created just one time and used whole project like global. Road signs is not changeable that is why Singleton useful in this situation.

**Interceptor:** When a request enters a Web application, it often must pass several entrance tests prior to the main processing stage. For example,

* Has the client been authenticated?
* Does the client have a valid session?
* Is the client's IP address from a trusted network?
* Does the request path violate any constraints?
* What encoding does the client use to send the data?
* Do we support the browser type of the client?

Some of these checks are tests, resulting in a yes or no answer that determines whether processing will continue. Other checks manipulate the incoming data stream into a form suitable for processing.

The classic solution consists of a series of conditional checks, with any failed check aborting the request. Nested if/else statements are a standard strategy, but this solution leads to code fragility and a copy-and-paste style of programming, because the flow of the filtering and the action of the filters is compiled into the application.

**Create pluggable filters to process common services in a standard manner without requiring changes to core request processing code. The filters intercept incoming requests and outgoing responses, allowing preprocessing and post-processing. We are able to add and remove these filters unobtrusively, without requiring changes to our existing code.**

We are able, in effect, to decorate our main processing with a variety of common services, such as security, logging, debugging, and so forth. These filters are components that are independent of the main application code, and they may be added or removed declaratively. For example, a

user trying to login 100 times in 1 minute it might be a fraud and after several times the server will be overloaded and the server will shutdown. When a client requests to log in, the filters in the chain are each processed in order before the authentication process I can count the authentication request from the same IP and should block this IP after several time.

* **Filter** - Filter which will performs certain task prior or after execution of request by request handler.
* **Filter Chain(***CheckDDOSAttackFilter***,***AuthenticationFilter,DebuggingFilter* **)** - Filter Chain carries multiple filters and help to execute them in defined order on target.
* **Target** - Target object is the request handler
* **Filter Manager** - Filter Manager manages the filters and Filter Chain.
* **Client** - Client is the object who sends request to the Target object.

**5. The proposed work for each of the 20 people that will work in your project,i.e. the proof that your software design allows for the separation of concerns approach.**

Project will divide to 3 main modules: **Data tier**, **Application tier**, **Presentation tier**

**Data tier**: 5 developers

In the Data tier 2 of the developer will create architecture of database and 3 of them will create procedures, functions, indexing etc.

In this module, the database will design based on the below information:  
User, Vehicles (Logical Participant, Physical Participant)

|  |  |
| --- | --- |
| **Logical Participant** | **Physical Participant** |
| Car | Car |
| Motorcycle | Motorcycle |
| Bus | Bus |

**Application tier**: 10 developers

* Physical and Logical interpretation of traffic participants – 3 developers
* Shortest Routes – 2 developers
* Authentication and security – 3 developers
* Testing – 2 developers

In the Application tier software will make some logic calculations, insert, update (find best route for current situation) etc. In this project we will have complicated design pattern usage, security level and several machine learning algorithms which is make this part more special and difficult.

**Presentation tier** : 4 developers

In the Presentation tier 3 of the developer will work on UX and 1 of them for QoE.

**6.The identification of at least three use-cases for your scenario that demonstrate the usage of your software architecture framework (i.e. the design 2 patterns usefulness) with a short explanation.**

Urbanization and developing technology engine vehicles count rising day by day. That is why each country trying to develop traffic lights, road signs, building new bridges and so on. Also one of the main requirement of drivers to go from A to B as soon as possible and using minimal energy. My project based to give each user most optimal route for current position and time. Additionally give access to road polices change light timing, ramp meter, speed limit and notify these changes to drivers simultaneously. I am going to develop the Optimization of Traffic application.

There are 6 main functions:

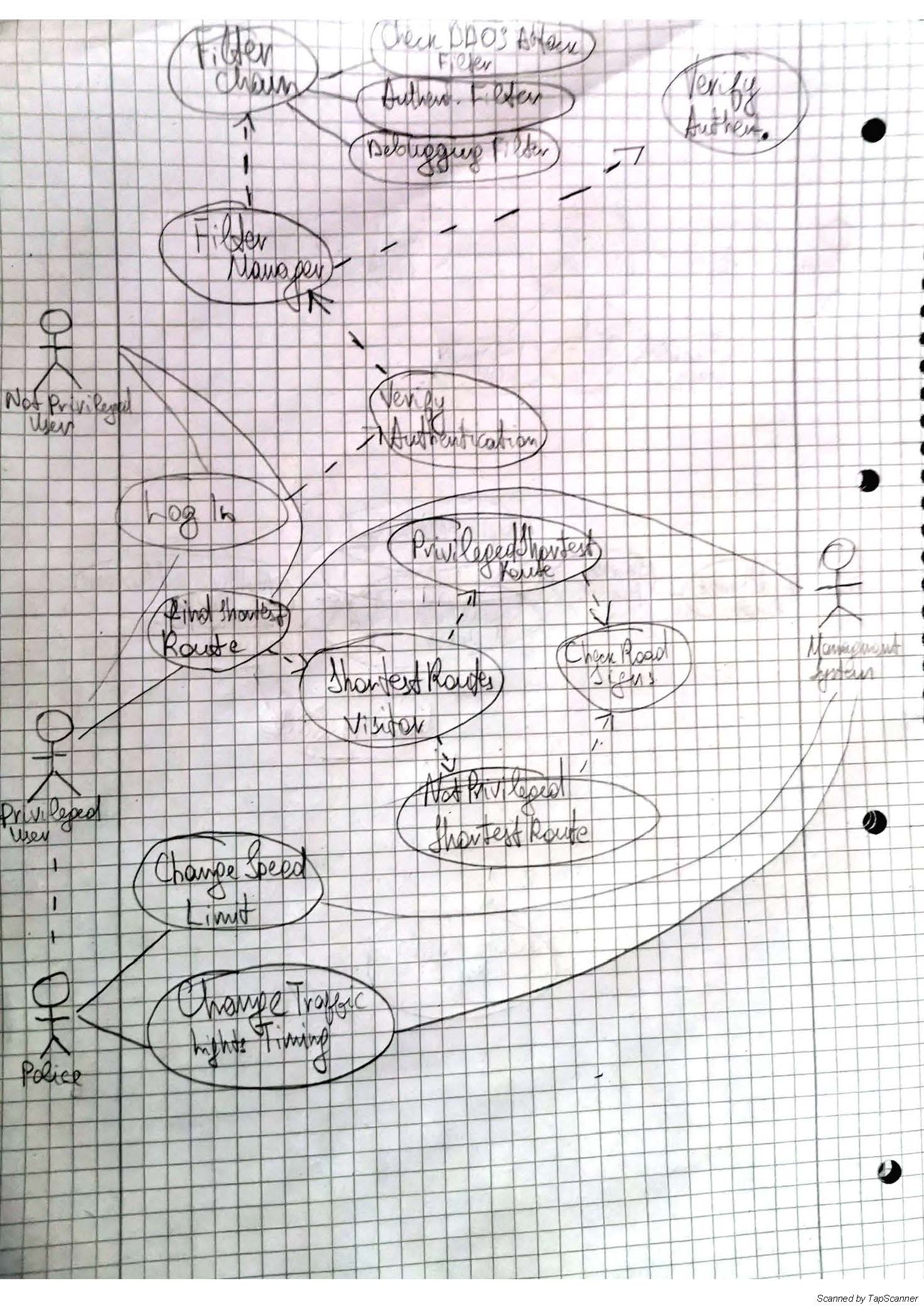
This permission will be only the police

1. Control the speed limits and notifying this drivers.
2. Control the traffic lights timing.
3. Control the ramp meter.

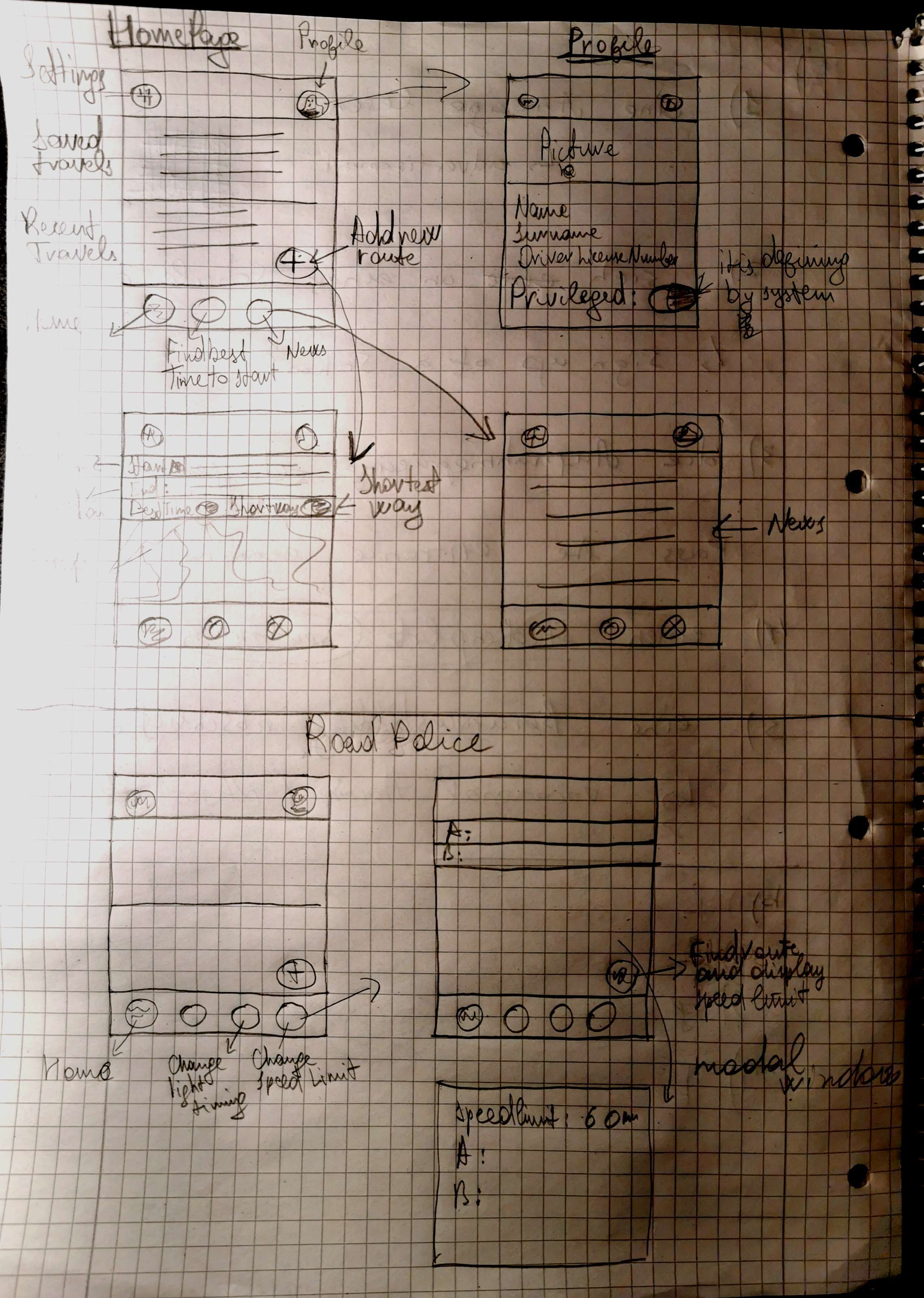
4. Some of the privileged vehicles will have: Emergence button.

5. Finding shortest routes.

6. Finding best time to start travel.

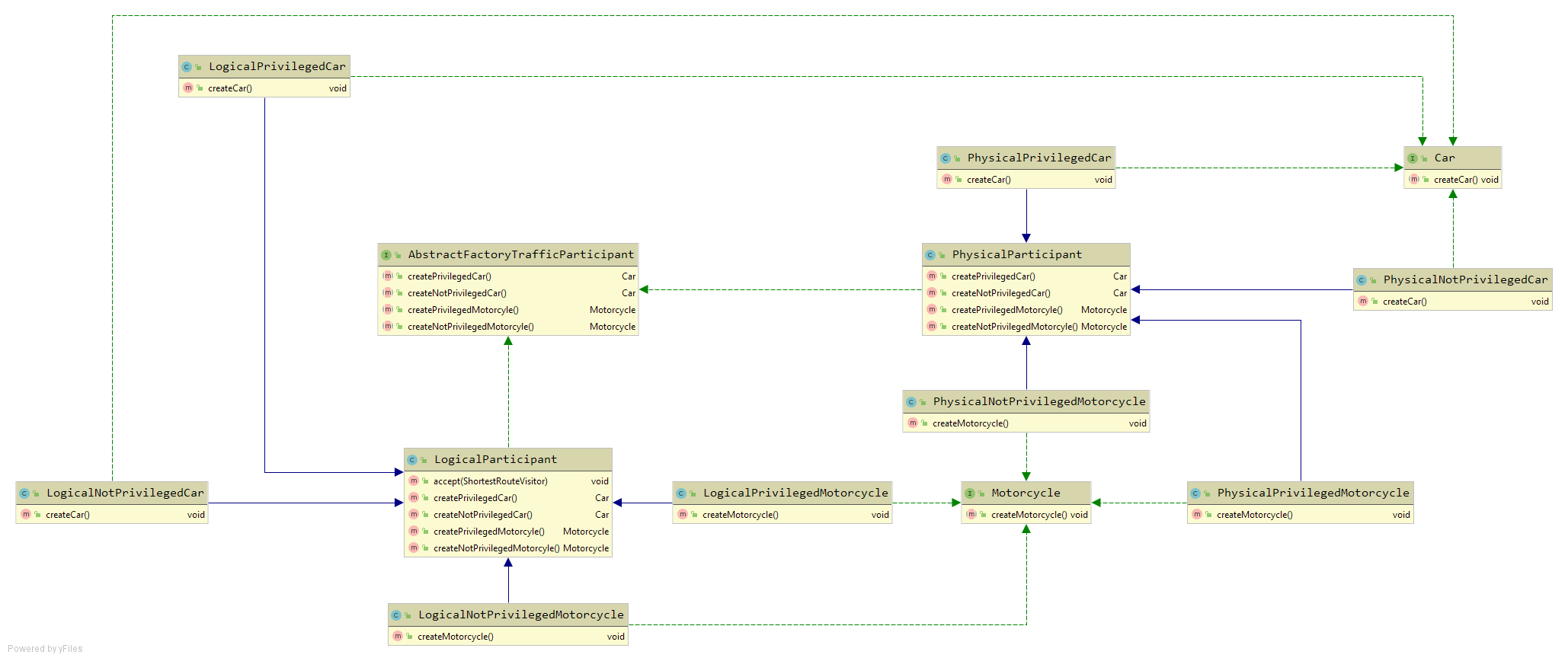


**An explanation how you designed your GUI and where and why you wouldtake into account UX and QoE**

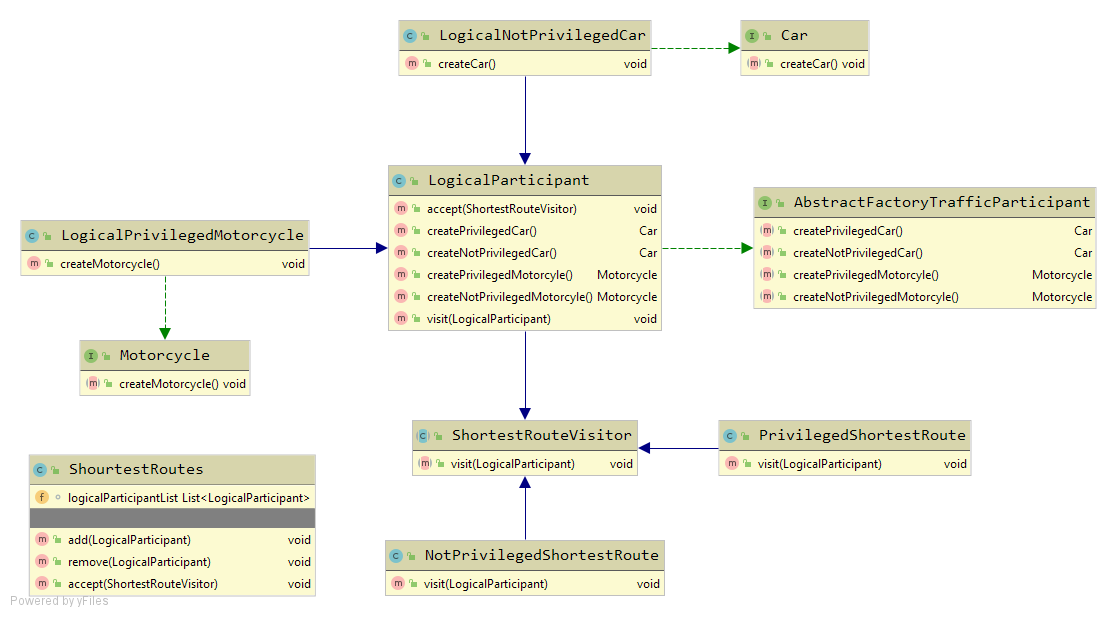


**7. A class diagram for your code (also clearly indicating where design patterns, data storage etc. is implemented)**

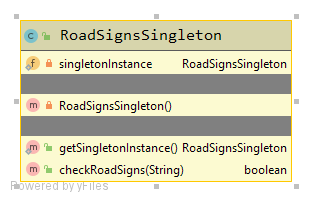
**Abstract Factory**



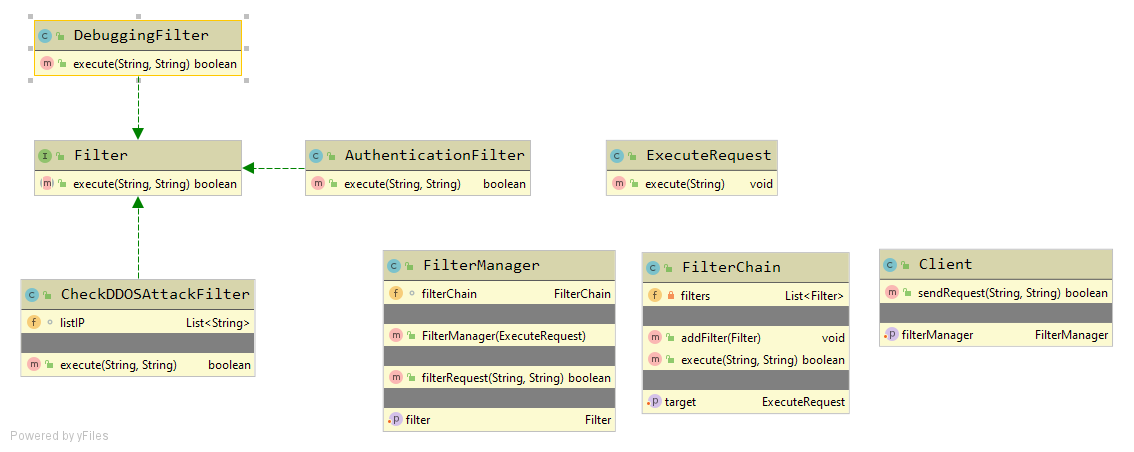
**Visitor Pattern**



**Singleton Pattern**



**Interceptor Pattern**



**8. A short personal summary of your work focusing on difficulties and improvements for the project**

When I started this project I focused on combine design pattern and create more powerful software. But in the process I had several problem that is why I needed to change my thinking from implementation on design pattern in application to solving problem with design pattern. I saw that specially when I write the code for abstract factory and visitor combination. Visitor pattern not suitable for the some level hierarchy. Then I divided these pattern and created separately implementation. I think for like these combination I need more experience with design patterns.

Then I had difficulties to show use-cases with software architecture which forced to me learn more about demonstration of the action flow.