# **Damose: Software Design Decision Summary**

Francesco Lucarelli

Student id: **2097837**

Date: [Date]

### **Features Implemented**

* Basic level (18-23 / 30)
* 1 programmer
  + Offline operation, with GTFS static data.
  + Display and search for stops, which shows the next lines that will stop you and the corresponding arrival times.
  + Display and search lines, which shows the current stop for each vehicle of the line.
  + Prediction of the arrival time of a one-stop line based on the static schedule.
  + Map of view of the position of the vehicles on the basis of the static schedule (non-interactive and without real-time updates), which shows the number / code of the line and the direction of the vehicle.
  + Differentiated management of the different types of vehicles (bus, tram, etc.).
  + Project development via git.
  + Interactive (zoom in/out) static map
* Intermediate level (24-27/30)
* 1 programmer

Real-time updating of bus locations (if online), both on the map and in the search results..

* Possibility of saving favourites (routes and stops).
* Display and search lines, which shows the current stop for each vehicle of the line.
* Prediction of the arrival time using real-time data
* Automatic switch between online and offline.
* Dependency management via maven or gradle.

### 

## 

## **2. Object-Oriented Design (OOD) Decisions**

### **2.1 Class Design & Responsibilities**

* Identify 3-5 Core Classes:
  + **Main:**
    - Primary Responsibility: manage the main GUI, search logic and display of routes, bus stops and bus locations on the map.
    - Justification:This class was created to centralise the start-up and configuration of the application, separating the bootstrap logic from the rest of the functionality. It encapsulates the initialisation behaviour, facilitating the management and maintenance of the application start-up.
  + **GTFSFetcher:**
    - Primary Responsibility: is only responsible for downloading and parsing GTFS-RT data, separating the network logic from the rest of the application.
    - Justification: This class was created to isolate GTFS data fetching logic, separating data acquisition operations from the rest of the application. It encapsulates update and validation behaviour, facilitating the maintenance and extension of functionality related to accessing RT GTFS data.
  + **GlobalParameters:**
    - Primary Responsibility: centralizes global parameters and constants, so if you need to change a value (e.g., URL, size, colors) you only need to change it in one place.
    - Justification: This class was created to avoid duplication and spreading of configuration settings in the code. It encapsulates global configuration data, facilitating the maintenance, modification and readability of application settings.
  + **BusWaypoint, Route, Trip, Stop, StopTime:**
    - Primary Responsibility: represent domain entities (buses, routes, stops, trips) and allow the data to be modeled clearly.
    - Justification:These classes were created to model the fundamental entities of the GTFS domain in a clear and structured manner, separating responsibilities and improving the readability, maintainability and extensibility of the code relating to the management of public transport data.
  + **CustomWaypointRenderer:**
    - Primary Responsibility: Just takes care of the custom display of markers on the map
    - Justification: This class was created to separate the waypoint rendering logic from the rest of the application, facilitating customisation and maintenance of the graphic visualisation. It encapsulates the drawing and interaction rules, improving the modularity and reusability of the code related to the representation of data on the map.
  + **StaticGTFSDownloader:**
    - Primary Responsibility: It is used to download and save static GTFS data (such as routes, stops, and trips) locally from the remoamobilità open data URL. It also handles integrity checking via the MD5 file. This process ensures that the application has access to the latest (static) data.
    - Justification:This class was created to isolate the logic of downloading and verifying static GTFS data, separating these operations from the rest of the application. It encapsulates specific data acquisition, saving and validation behaviours, facilitating the maintenance and extension of functionalities related to the updating of GTFS data.
  + **Favourites:**
    - Primary Responsibility: Manages the storage, retrieval and modification of user favourites, such as stops, or selected routes. Provides methods for adding, removing and checking the presence of items among the favourites, ensuring quick access to the information most used by the user.
    - Justification: This class was created to isolate the logic related to favourites, separating the management of user preferences from the rest of the application. It encapsulates specific data (list of favourites) and behaviours (add, remove, check), facilitating the maintenance and extension of functionality related to favourites.

### **2.2 Encapsulation**

* How is data hidden/protected within your classes?
  + [e.g., All instance variables are declared as private/protected.]
* How is access to data controlled?
  + [e.g., Through public getter and setter methods for controlled modification.]
* Benefits Achieved:
  + [e.g., Prevents direct manipulation of internal state, reduces coupling, makes code easier to maintain.]

### **2.3 Inheritance (If Applicable)**

* Identify the main Base Class(es) and Derived Class(es):
  + [e.g., Base Class: Animal, Derived Classes: Dog, Cat]
* Justify their use over composition:
  + [Why was inheritance the right choice here? Did it promote code reuse for common behavior or attributes?]

### **2.4 Polymorphism (If Applicable)**

* Provide the main uses of polymorphism in your code:
  + [e.g., A method like 'display()' defined in a base class and overridden in derived classes; or method overloading.]

### **2.5 Abstraction (If Applicable)**

* Explain how you used abstract classes or interfaces:
  + [e.g., An interface 'IDataSource' defines methods like 'read()' and 'write()' without specifying implementation details.]
* Why were certain details hidden?
  + [e.g., To focus on essential functionalities and allow different concrete implementations (e.g., FileDataSource, DatabaseDataSource).]

### **2.6 Design Patterns (If Applicable)**

* Mention which design patterns were applied:
  + [e.g., Singleton, Factory, Observer, Strategy]
* Explain *why* each pattern was chosen:
  + [e.g., Singleton for ensuring only one instance of a Logger class; Factory for creating different types of reports based on user input.]

## **3. Architectural & Project Management Considerations**

### **3.1 Scalability**

* How does your design account for potential future growth?
  + [e.g., Modular design allows adding new features without rebuilding the whole system; separation of data logic from UI logic; considerations for handling more users or larger data volumes (even if not implemented).]

### **3.2 Maintainability**

* Describe aspects of your code that promote maintainability:
  + [e.g., Clear variable/method naming conventions, code comments, consistent formatting, small and focused methods, clear separation of concerns.]
* How easy would it be for a new developer to understand and modify your code?
  + [Self-assessment: What makes it easy/hard? What documentation exists?]

### **3.3 Testability**

* How does your design facilitate testing?
  + [e.g., Classes have clear responsibilities, making unit testing easier; dependencies are managed (e.g., no direct database calls in UI classes).]
* What types of tests were considered (even if not written)?
  + [e.g., Unit tests for individual methods, integration tests for interactions between components.]

## **Conclusion**