# **Damose: Software Design Decision Summary**

Student Name:

Student id:

Date: [Date]

### **[if team project] Team**

* Student 1
* …

### **[if team project] Team Roles & Responsibilities**

* Backend Developer(s):
  + Responsibilities:
  + Main classes: …

### **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.).
* 2 – 4 programmers; as above, more
  + …

### 

## 

## **1. [if custom project] Project Overview**

* Project Name: [Brief, descriptive name of your project]
* Core Problem Solved: [Clearly state the problem your software addresses. What is its main purpose?]

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

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

* Identify 3-5 Core Classes:
  + Class Name 1:
    - Primary Responsibility: [e.g., Managing user authentication and profile data.]
    - Justification: [Why was this a distinct class? What specific data/behavior does it encapsulate?]
  + …

### **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**