# **Damose: sintesi della decisione di progettazione del software**

Francesco Lucarelli

Matricola: **2097837**

Data: [data]

### **Caratteristiche implementate**

* Livello di base (18-23/ 30)
* 1 programmatore
  + Funzionamento offline, con dati statici GTFS.
  + Visualizzazione e ricerca di fermate, che mostra le prossime linee che si ferma e gli orari di arrivo corrispondenti.
  + Visualizza e cerca le linee, che mostra la fermata corrente per ogni veicolo della linea.
  + Previsione del l'ora di arrivo di una linea a una fermata in base al programma statico.
  + Mappa di visualizzazione della posizione dei veicoli sulla base del programma statico (non interattivo e senza aggiornamenti in tempo reale), che mostra il numero/codice della linea e la direzione del veicolo.
  + Gestione differenziata dei diversi tipi di veicoli (autobus, tram, ecc.).
  + Sviluppo del progetto tramite git.
  + Mappa interattiva (zoom in/out) statica
* Livello intermedio (24-27/30)
* 1 programmatore

Aggiornamento in tempo reale delle posizioni degli autobus (se online), sia sulla mappa che nei risultati di ricerca..

* Possibilità di salvare preferiti (linee e/o fermate).
* Visualizza e cerca le linee, che mostra la fermata corrente per ogni veicolo della linea.
* Previsione del tempo di arrivo utilizzando dati in tempo reale
* Switch automatico tra online e offline.
* Gestione delle dipendenze tramite maven o gradle.

### 

## 

## **2. Decisioni in materia di progettazione orientata agli oggetti (OOD)**

### **2.1 Progettazione della classe e responsabilità**

* Individuare 3-5 classi di base:
  + **Main:**
    - Responsabilità primaria: gestire la GUI principale, la logica di ricerca e la visualizzazione di percorsi, fermate degli autobus e le posizioni degli autobus sulla mappa.
    - Giustificazione: Questa classe è stata creata per centralizzare l'avvio e la configurazione dell'applicazione, separando la logica di bootstrap dal resto della funzionalità. Incapsula il comportamento di inizializzazione, facilitando la gestione e la manutenzione del l'avvio del l'applicazione.
  + **GTFSFetcher:**
    - Responsabilità primaria: è responsabile solo per il download e l'analisi dei dati GTFS-RT, separando la logica di rete dal resto dell'applicazione.
    - Giustificazione: questa classe è stata creata per isolare la logica di recupero dei dati GTFS, separando le operazioni di acquisizione dati dal resto dell'applicazione. Racchiude il comportamento di aggiornamento e convalida, facilitando la manutenzione e l'estensione delle funzionalità relative all'accesso ai dati GTFS RT.
  + **GlobalParameters:**
    - Responsabilità primaria: centralizza i parametri e le costanti globali, quindi se hai bisogno di cambiare un valore (ad esempio, URL, dimensioni, colori) devi solo cambiarlo in un posto.
    - Giustificazione: Questa classe è stata creata per evitare la duplicazione e la diffusione delle impostazioni di configurazione nel codice. Incapsula i dati di configurazione globale, facilitando la manutenzione, la modifica e la leggibilità delle impostazioni del l'applicazione.
  + **Bus Waypoint, percorso, viaggio, fermata, tempo:**
    - Responsabilità primaria: rappresentano le entità di dominio (autobus, rotte, fermate, viaggi) e consentono di modellare chiaramente i dati.
    - Giustificazione: Queste classi sono state create per modellare le entità fondamentali del dominio GTFS in modo chiaro e strutturato, separando le responsabilità e migliorando la leggibilità, la manutenibilità e l'estensibilità del codice relativo alla gestione dei dati di trasporto pubblico.
  + **CustomWaypointRenderer:**
    - Responsabilità primaria: si occupa solo della visualizzazione personalizzata dei marcatori sulla mappa
    - Giustificazione: Questa classe è stata creata per separare la logica di rendering dei waypoint dal resto dell'applicazione, facilitando la personalizzazione e il mantenimento della visualizzazione grafica. Racchiude le regole di disegno e interazione, migliorando la modularità e la riutilizzabilità del codice relativo alla rappresentazione dei dati sulla mappa.
  + **StaticGTFSDownloader:**
    - Responsabilità primaria: viene utilizzato per scaricare e salvare dati GTFS statici (quali percorsi, fermate e viaggi) localmente dalla remoamobilità open data URL. Gestisce anche il controllo di integrità tramite il file MD5. Questo processo assicura che l'applicazione abbia accesso ai dati più recenti (statici).
    - Giustificazione: Questa classe è stata creata per isolare la logica del download e della verifica dei dati statici GTFS, separando queste operazioni dal resto dell'applicazione. Racchiude comportamenti specifici di acquisizione, salvataggio e convalida dei dati, facilitando la manutenzione e l'estensione delle funzionalità relative all'aggiornamento dei dati GTFS.
  + **Favoriti:**
    - Responsabilità primaria: gestisce l'archiviazione, il recupero e la modifica dei preferiti degli utenti, come le fermate o i percorsi selezionati. Fornisce metodi per l'aggiunta, la rimozione e il controllo della presenza di elementi tra i favoriti, garantendo un rapido accesso alle informazioni più utilizzate dall'utente.
    - Giustificazione: Questa classe è stata creata per isolare la logica relativa ai preferiti, separando la gestione delle preferenze dell'utente dal resto dell'applicazione. Incapsula dati specifici (elenco dei preferiti) e comportamenti (aggiungere, rimuovere, controllare), facilitando il mantenimento e l'estensione delle funzionalità relative ai preferiti.

### **2.2 Incapsulamento**

* Come sono i dati nascosti/protetti all'interno delle vostre classi?
  + [per esempio, tutte le variabili di istanza sono dichiarate private/protette. ]
* Come viene controllato l'accesso ai dati?
  + [per esempio, attraverso metodi pubblici getter e setter per la modifica controllata. ]
* Benefici conseguiti:
  + [per esempio, impedisce la manipolazione diretta dello stato interno, riduce l'accoppiamento, rende il codice più facile da mantenere. ]

### **2.3 Successione (se applicabile)**

* Identificare le principali classi di base e le classi derivate:
  + [per esempio, classe di base: animale, classi derivate: cane, gatto]
* Giustificarne l'uso sulla composizione:
  + [Perché l'eredità era la scelta giusta qui? Ha promosso il riutilizzo del codice per comportamenti o attributi comuni? ]

### **2.4 Polimorfismo (se applicabile)**

* Fornire gli usi principali del polimorfismo nel codice:
  + [es., un metodo come 'display()' definito in una classe di base e sovrascritto nelle classi derivate; o sovraccarico del metodo. ]

### **2.5 Astrazione (se applicabile)**

* Spiegare come sono state utilizzate le classi o le interfacce astratte:
  + [per esempio, un'interfaccia 'IDataSource' definisce metodi come 'read()' e 'write()' senza specificare dettagli di implementazione. ]
* Perché alcuni dettagli sono stati nascosti?
  + [es., concentrarsi sulle funzionalità essenziali e consentire diverse implementazioni concrete (ad esempio, FileDataSource, DatabaseDataSource). ]

### **2.6 Schemi di progettazione (se applicabile)**

* Indicare quali modelli di progetto sono stati applicati:
  + [es., Singleton, fabbrica, osservatore, strategia]
* Spiegare *il modello* scelto:
  + [per esempio, Singleton per garantire solo un'istanza di una classe Logger; Fabbrica per la creazione di diversi tipi di rapporti in base all'input dell'utente. ]

## **3. Considerazioni architettoniche e di gestione del progetto**

### **3.1 Scalabilità**

* In che modo il vostro progetto tiene conto della potenziale crescita futura?
  + [ad esempio, il design modulare consente di aggiungere nuove funzionalità senza ricostruire l'intero sistema; separazione della logica dei dati dalla logica del l'interfaccia utente; considerazioni per gestire più utenti o volumi di dati più grandi (anche se non implementati). ]

### **3.2 Manutenibilità**

* Descrivi gli aspetti del tuo codice che promuovono la manutenzione:
  + [es., chiare convenzioni di denominazione variabile/metodo, commenti di codice, formattazione coerente, metodi piccoli e mirati, chiara separazione delle preoccupazioni. ]
* Quanto sarebbe facile per un nuovo sviluppatore capire e modificare il tuo codice?
  + [Autovalutazione: cosa rende facile/difficile? Quale documentazione esiste? ]

### **3.3 Possibilità di verifica**

* In che modo il vostro progetto facilita le prove?
  + [es., le classi hanno responsabilità chiare, rendendo più facile la prova di unità; le dipendenze sono gestite (ad esempio nessuna chiamata di database diretto nelle classi UI). ]
* Quali tipi di test sono stati presi in considerazione (anche se non scritti)?
  + [es., test unitari per i singoli metodi, test di integrazione per le interazioni tra componenti. ]

## **Conclusione**