- # Damose: sintesi delle scelte di progettazione
- Francesco Lucarelli
- Matricola: **2097837**
- Data: [14/06/25]
- **Caratteristiche implementate**
- 1 programmatore

Funzionamento offline, con dati statici GTFS.

- Ricerca e visualizzazione delle fermate: mostra le prossime linee che si fermano alla fermata selezionata e gli orari di arrivo corrispondenti in base ai dati statici caricati.
- Ricerca e visualizzazione delle linee: mostra la fermata corrente per ogni veicolo della linea scelta.
- Controllo dell'aggiornamento dei dati statici, con download automatico in caso di dati vecchi in locale.
- Previsione dell'ora di arrivo di una linea ad una data 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.) sia in forma tabellare che sulla mappa con l'utilizzo di differenti icone.
- Sviluppo del progetto tramite git.
- Mappa interattiva (con funzioni di zoom in/out e centratura) statica

Funzionamento realtime.

aggiornamento in tempo reale delle posizioni dei veicoli (se online), sia sulla mappa che nella tabella dei risultati di ricerca.

- Possibilità di salvare preferiti (linee e/o fermate).
- Ricerca e visualizzazione delle linee: mostra la posizione corrente per ogni veicolo della linea.
- Selezionando una fermata: visualizza i veicoli che servono la fermata scelta, indicando per ciascuno di essi i tempi stimati di arrivo alla fermata utilizzando dati in tempo reale (confronto tra lo stopSequence del mezzo e quello della fermata)

- Switch automatico tra modalità online e offline in base alla presenza di connessione internet
- Gestione delle dipendenze tramite 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, caricare i dati statici , i preferiti, gestire gli eventi in corrispondenza dell'interazione dell'utente con l'interfaccia.
    - 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, incentra su di essa la gestione dell'interfaccia utente, facilitando la gestione e la manutenzione dell'applicazione.

  - **GTFS Fetcher:**
    - 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 in caso di necessità di cambiare un valore (ad esempio, URL, dimensioni, colori) basta cambiarlo solo in questa classe.
    - 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 dell'applicazione.

- **Bus Waypoint, Route, Trip, Stop, StopTime:**
  - Responsabilità primaria: queste classi rappresentano le entità basilari del programma (punti sulla mappa visuale, linee, fermate, orari di fermata) e consentono di modellare chiaramente i dati e di caricarli in memoria dai file.
  - Giustificazione: Queste classi sono state create per modellare le entità fondamentali dell'ambiente GTFS in modo chiaro e strutturato, separando le relative rappresentazioni degli oggetti e migliorando la leggibilità, la manutenibilità e l'estensibilità del codice relativo alla gestione dei dati di trasporto pubblico.

- **CustomWaypointRenderer:**
  - Responsabilità primaria: si occupa 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 corse, fermate e linee) localmente dagli open data di romamobilità. Gestisce anche il controllo di integrità tramite il file MD5. Questo processo assicura che l'applicazione abbia accesso sempre 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.

- **Favourites:**
  - Responsabilità primaria: gestisce l'archiviazione, il recupero e la modifica dei preferiti da parte degli utenti, come le fermate o le linee selezionate. 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, e la loro disponibilità anche al riavvio dell'applicazione grazie al salvataggio su file.

- 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?
  Le variabili di utilizzo di ciascuna classe sono state dichiarate private e ove necessario sono stati forniti i relativi metodi di getter e/o setter
- Come viene controllato l'accesso ai dati?
  attraverso metodi pubblici getter e setter per la modifica controllata.
- Benefici conseguiti:
  Si impedisce che le variabili di stato delle varie classi possano essere modificate, in maniera incontrollata da codice esterno alla classe. In particolare i metodi setter sono stati resi disponibili solo se strettamente necessario.

- **2.4 Polimorfismo**

- Fornire gli usi principali del polimorfismo nel codice:
  In taluni casi si è fatto uso del sovraccarico dei metodi di una classe, ad esempio nella classe Stoptimes, vi sono due implementazione del metodo: getStoptimeFromStopId(String stopId): ritorna la lista di tutti gli stopTimes relativi allo stopId - getStoptimeFromStopId(String stopId,long minutesRange): ritorna la lista dei soli stopTimes relativi allo stopId con arrivo previsto allo stopId entro i prossimi minutesRange

- # 3. Aspetti architettonici e di gestione del progetto

- **3.1 Scalabilità**

- In che modo il vostro progetto tiene conto della potenziale crescita futura?
  La separazione spinta delle funzionalità in classi differenti costituisce un design modulare che consente di aggiungere nuove funzionalità senza ricostruire l'intero sistema; si è cercato di separare il più possibile la logica dei dati e degli algoritmi dalla logica del l'interfaccia utente; il programma è facilmente estensibile ad una versione multiutente, semplicemente

associando il salvataggio dei preferiti ad uno specifico userId e gestendo le funzionalità di login.

- **3.2 Manutenibilità**

- Descrivi gli aspetti del tuo codice che promuovono la manutenzione:
  sono state utilizzare delle chiare convenzioni di denominazione variabile/metodo, il codice è ricco di commenti sia per distinguere lo scopo dei vari blocchi, o metodi, sia in particolari righe ove si è ritenuto necessario marcarne il funzionamento.
- Quanto sarebbe facile per un nuovo sviluppatore capire e modificare il tuo codice?
  Tramite la lettura della javadoc è possibile avere un'idea della struttura delle classi di progetto e dei loro scopi di funzionamento. Per ogni metodo è descritto lo scopo, nel vari punti del codice ove si è ritenuto sottolineare il funzionamento sono stati aggiunti commenti sulla riga.

- **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)?
  In fase di sviluppo si è proceduto sin da subito al testing del codice scritto, a tal proposito si trovano sviluppati dei metodi print() con lo scopo della visualizzazione immediata dei dati da testare in console, spesso in molti punti sono stati effettuati dei println in console del contenuti di variabili per verificare in modo immediato se assumevano valori conformi al corretto funzionamento (per migliore leggibilità questi print sono stati poi tolti, man mano che ci si accertava del corretto funzionamento).

● Conclusione

- E' stato molto entusiasmante mettermi alla prova con questo progetto, averlo affrontato da solo mi ha permesso di iniziare sin da subito e dedicargli tutto il tempo a me necessario per il suo sviluppo, test e debug. Uno dei principali scogli da superare nello sviluppo del progetto inizialmente è stata la corretta comprensione della struttura dei file GTFS e del significato dei dati in essi contenuti. Mi è venuto piuttosto naturale replicare la stessa separazione strutturale dei file con delle rispettive classi, in modo da avere una rappresentazione in memoria dei dati messi a disposizione su file. In termini di efficienza, in generale l'implementazione scelta per la rappresentazione dei dati

in primo momento si è rivelata sempre efficiente, ed eccezione dei dati del file stoptimes, che considerata la sua notevole dimensione mi ha costretto a rivederne la rappresentazione risolvendo il problema con l'utilizzo della classe Map, anziché di una classica ArrayList. Anche lo sviluppo dell'interfaccia grafica ha avuto diversi momenti di revisione, spesso le scelte più immediate in termini di codice, non si trasformavano in un interfaccia utente altrettanto chiara e semplice da utilizzare, in questi casi è stato utile pensare prima al risultato visuale e funzionale che si voleva ottenere e poi passare al codice.