Weather App

Relazione per il progetto di
Programmazione ad Oggetti

Anno Accademico 2024/2025

Componenti del gruppo:

Giacomo Casadei

Lorenzo Magni

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software si pone come obiettivo quello di costruire una applicazione per le previsioni del meteo, che permetta la visualizzazione, riferita ad una località selezionata dall'utente, dei principali dati meteorologici, quali temperatura, temperatura percepita, temperature minima e massima del giorno, previsione per le ore successive e per la settimana e fasi lunari. Inoltre, l'applicazione mira a fornire una modalità "VIAGGIO" che possa permettere ad un utente di inserire luogo, data e ora di partenza, destinazione finale in modo da calcolare un percorso corredato di dati meteorologici. Verranno fornite informazioni sulle condizioni meteo lungo il tragitto, con la possibilità di proporre percorsi alternativi.

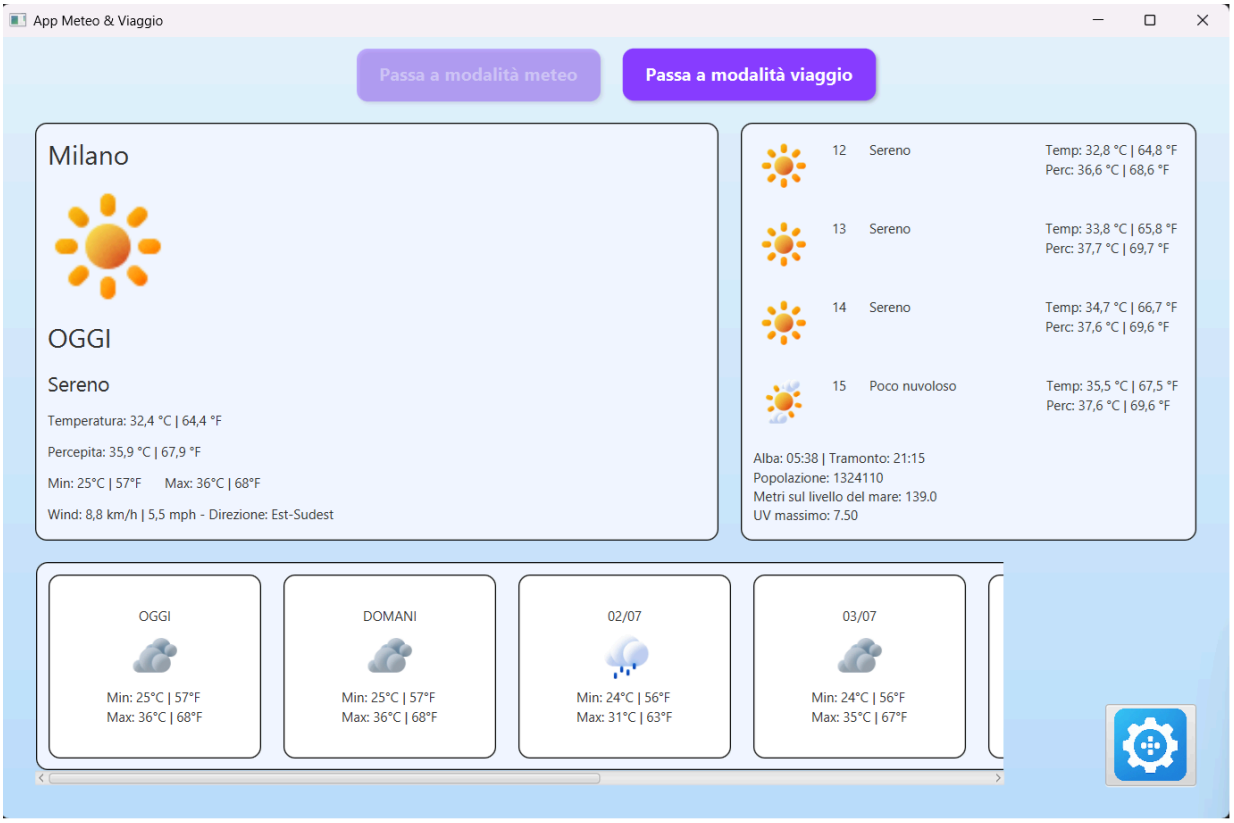

Figura 1.1: Schermata principale app

Requisiti funzionali

1. **Schermata di caricamento**
    - All'avvio mostra subito lo splash mentre si carica e parsa il CSV delle città.
2. **Scelta della città**
    - Se non esiste città nel file di configurazione:
        - Chiede se usare la geolocalizzazione via IP oppure selezionare manualmente la città.
        - In caso di geolocalizzazione, salva la città e mostra le info meteo.
        - Altrimenti richiede all'utente una città.
    - Se la città è già salvata, all'avvio mostra subito le sue informazioni.
3. **Gestione delle informazioni meteo**
    - Nella schermata principale visualizza i dati meteo aggiornati automaticamente ogni 20 minuti.
    - Possibilità di:
        - Cambiare città (aggiorna istantaneamente la schermata principale).
        - Consultare info aggiuntive (fasi lunari, grafici settimanali di temperature massime e minime).
4. **Travel Mode**
    - Permette di selezionare città, data e ora di partenza e destinazione con autocompletamento.
    - Integra dati di navigazione forniti da API esterne e dati meteo per eseguire un'analisi dei percorsi (considerando distanza e condizioni meteo).
    - Visualizzazione di percorsi alternativi con indicazioni di strada da percorrere, tempo di percorrenza, valutazione meteo e arrivo previsto

Requisiti non funzionali

- Il software deve funzionare sui tre principali sistemi operativi: Windows, Linux, macOS.
- La finestra principale deve essere ridimensionabile.
- I risultati vengono forniti in tempo brevi.

1.2 Modello del dominio

Il dominio applicativo si articola in due funzionalità principali percepite dall'utente: l'analisi delle condizioni meteo in una singola città e la pianificazione di un viaggio tra due località tenendo conto delle condizioni atmosferiche previste lungo il tragitto.
Questi due casi d'uso principali vengono analizzati in Figure 1.3 e 1.4.

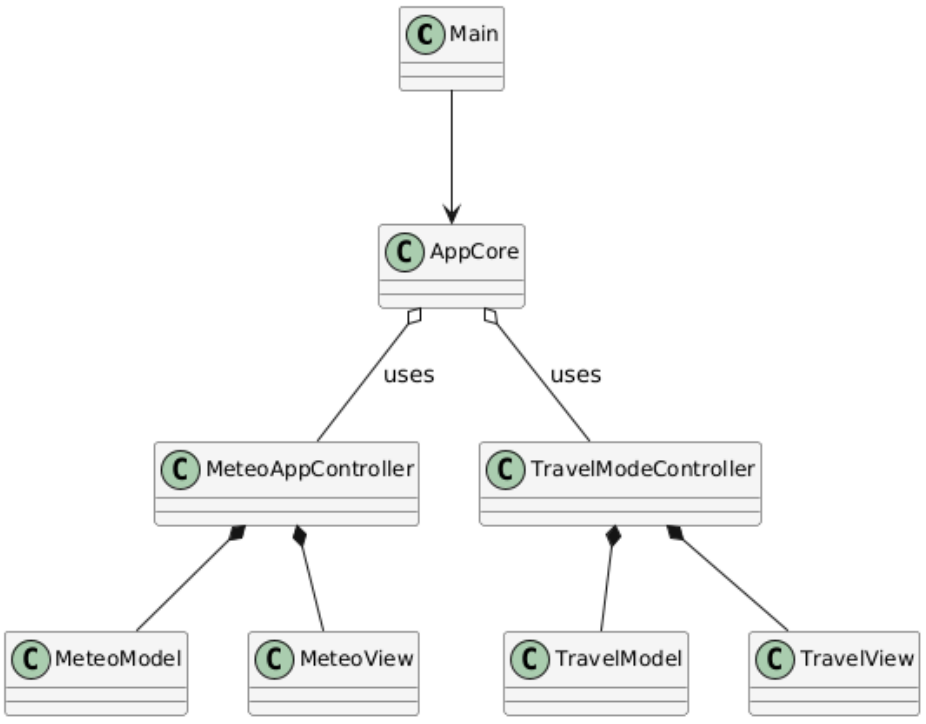

Figura 1.2: UML generale del dominio

Weather Mode

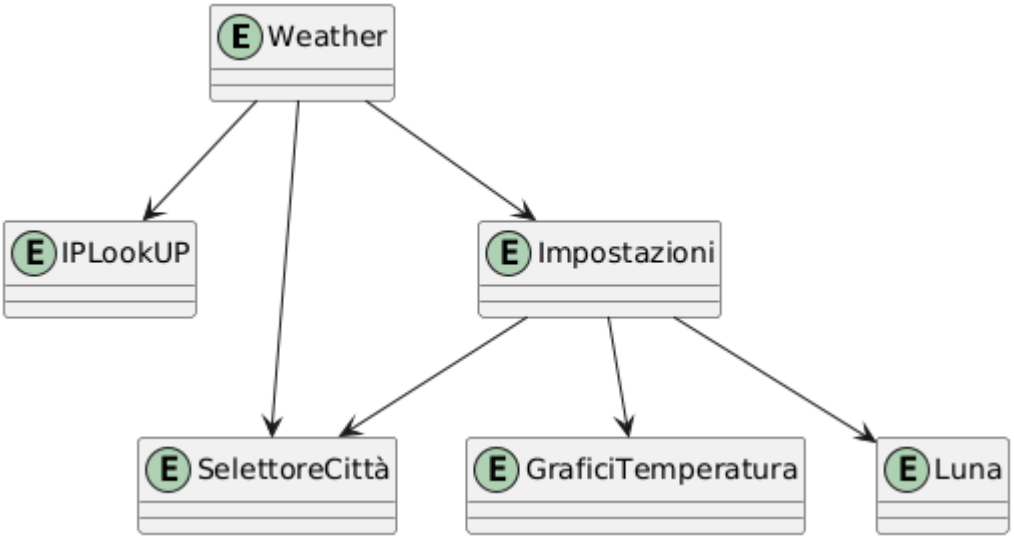

Figura 1.3: UML della parte Meteo

La modalità meteo ha lo scopo di far visualizzare all'utente tutte le informazioni utili sul meteo e sulla città selezionata.
All'inizio viene chiesto all'utente se farsi geolocalizzare tramite l'indirizzo IP o se scegliere direttamente la città desiderata.
In ogni caso, presa la città, **Weather** fa il retrieval delle informazioni meteo e anche delle informazioni della città e le visualizza. Da qui è inoltre possibile accedere alle **Impostazioni** da cui è possibile visualizzare i **grafici della temperatura** settimanali, le informazioni sulla **luna** odierne e anche di cambiare città di cui visualizzare le informazioni.

Travel Mode

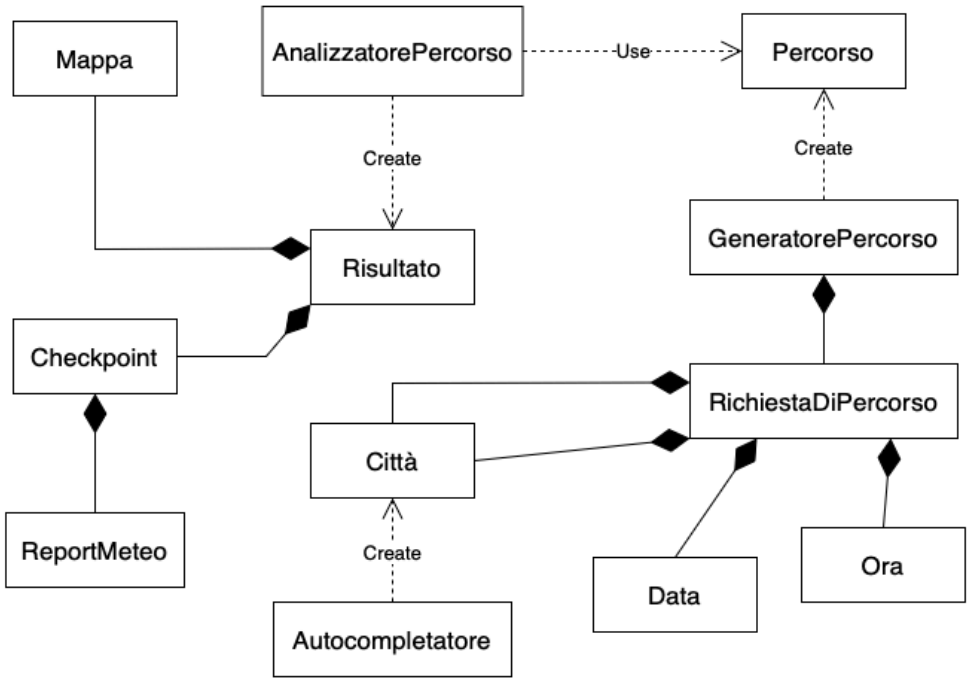

Figura 1.4: Schema UML dell'analisi del problema della modalità viaggio con rappresentate le entità principali e i rapporti fra loro

La modalità viaggio ha lo scopo di assistere l'utente nella scelta del percorso migliore tra due città, tenendo conto non solo della distanza o del tempo di percorrenza, ma anche delle **condizioni meteorologiche previste** lungo il tragitto.

Il flusso base prevede che l'utente selezioni una **città di partenza** e una **città di arrivo**, assistito da un sistema di **completamento automatico** per facilitare l'inserimento dei nomi. A partire da queste due città, il sistema calcola **uno o più percorsi alternativi**, ciascuno dei quali costituisce una possibile soluzione al problema.

Ogni percorso viene internamente suddiviso in una sequenza ordinata di **checkpoint**. In corrispondenza di ciascun checkpoint, il sistema verifica le **condizioni meteo previste** (es. pioggia, vento, neve, temperature estreme) alla data e ora stimata del passaggio.

A partire dai dati raccolti per ciascun checkpoint, viene generato, per ogni percorso proposto, un **risultato** che aggrega le seguenti informazioni:

- Elenco dei checkpoint e delle condizioni meteo previste
- Indicatori di eventuali criticità meteo
- Un **punteggio sintetico** che valuta la qualità meteorologica complessiva del percorso
- **Mappa** nella quale il percorso è visualizzato graficamente; in presenza di condizioni avverse, vengono mostrati **segnalini visivi**

Le entità identificate (in grassetto) costituiranno la base astratta su cui sarà successivamente costruita l'architettura software.

Capitolo 2

Design

2.1 Architettura

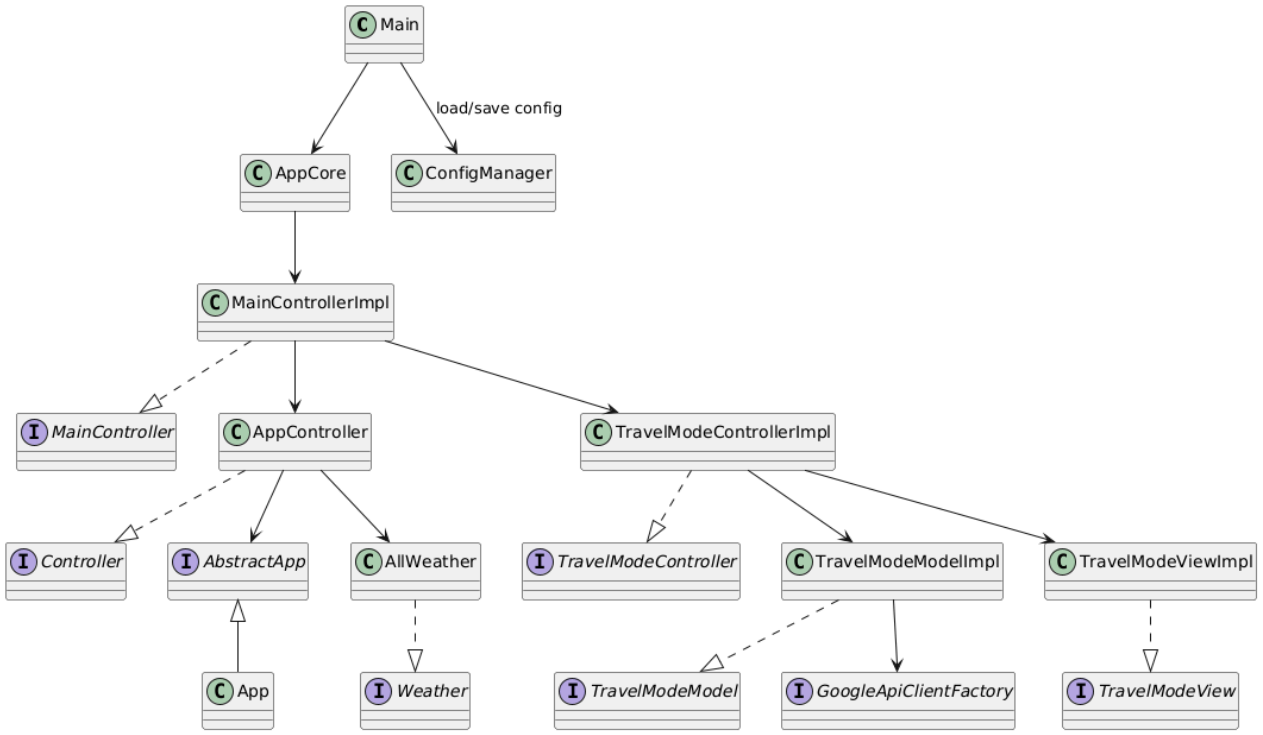

Figura 2.1: Schema UML dell'implementazione del pattern MVC

Per mantenere una chiara separazione tra la logica di presentazione, quella di controllo e quella di business, questo progetto Java adotta pienamente il pattern MVC. Tale pattern è stato declinato in modo gerarchico, poiché l'applicazione offre due funzionalità distinte e parallele: la **Modalità Meteo** e la **Modalità Viaggio**. Questi due sottosistemi sono coordinati da un controller principale, che funge da punto di accesso centralizzato per l'intera applicazione.

La struttura architetturale è pensata per favorire la separazione delle responsabilità, la leggibilità del codice e la scalabilità futura del progetto.

Al vertice, la classe `Main` istanzia `AppCore` che affida a `MainControllerImpl` il compito di orchestrare l'avvio e gestire il passaggio da una modalità all'altra, offrendo un punto d'ingresso unico e astratto all'intera logica applicativa, seguendo una logica facade-like, che nasconde i dettagli di inizializzazione e coordinamento.

Il `MainControllerImpl` da un lato legge e salva la configurazione tramite `ConfigManager`, dall'altro crea due controller specializzati.

Il primo, *AppController*, gestisce la logica generale dell'applicazione, compresa l'interazione con il servizio meteo attraverso l'interfaccia `Weather` e la sua implementazione concreta `AllWeather`.

Il secondo, `TravelModeControllerImpl` che implementa l'interfaccia 'TravelModeController', si occupa esclusivamente della modalità "Travel Mode": quando l'utente fa partire questa funzione, la view associata (`TravelModeViewImpl`) trasmette gli eventi al controller, che invoca sul model (`TravelModeModelImpl`) le operazioni di calcolo del percorso.

Il model, a sua volta, sfrutta un ApiClientFactory, per esempio `GoogleApiClientFactory` (Figura 2.1)che permette di interfacciarsi con le API di Google Maps, e sfrutta altri componenti per l'elaborazione dei dati, la creazione dei risultati e la conservazione dello stato interno di tappe, coordinate e dati meteorologici.

Una volta pronti i risultati, il controller li passa alla view, che si limita a mostrarli sullo schermo senza conoscere i dettagli della logica o delle chiamate esterne. In questo modo la user interface resta completamente scollegata dal business logic e dal coordinamento degli eventi, garantendo estrema modularità, testabilità e la possibilità di sostituire facilmente la parte grafica senza impattare sul model o sul controller.

Le classi principali:

- **Main**: punto di ingresso dell'applicazione, avvia *AppCore* e chiama MainControllerImpl.
- **AppCore**: inizializza i servizi di base.
- **ConfigManager**: si occupa di caricare e salvare la configurazione da file.
- **MainControllerImpl**: orchestratore generale; legge le impostazioni e crea i sotto-controller (AppController, TravelModeControllerImpl).
- **AbstractApp** e AppController: controller per la logica globale, delega a AllWeather la gestione delle previsioni.
- **AllWeather**: implementazione di Weather; recupera e aggiorna i dati meteo.
- **TravelModeControllerImpl**: implementazione di TravelModeController; è il controller dedicato alla funzionalità "Travel Mode": riceve eventi dalla view e coordina modello e vista.
- **TravelModeModelImpl**: implementazione di TravelModeModel; calcola il percorso, lo analizza, mantiene stato e interroga le API Google tramite GoogleApiClientFactory.

- **TravelModeViewImpl**: implementazione di TravelModeView; disegna la UI della modalità di viaggio e invia comandi al controller.
- **GoogleApiClientFactory**: factory per istanziare client per le Google Maps API.

2.2 Design dettagliato

2.2.1 Giacomo Casadei

AdvancedJsonReader

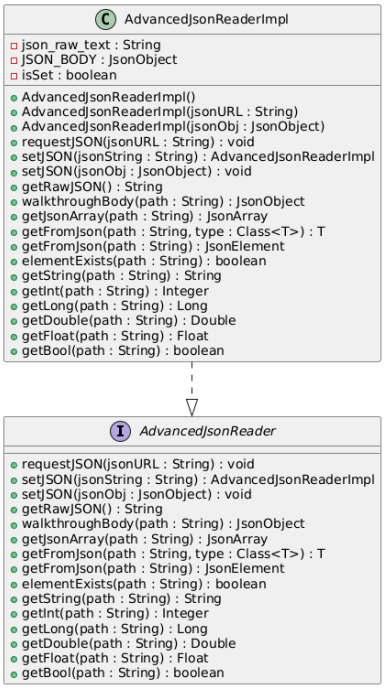

Figura 2.2: Schema UML di AdvancedJsonReader

Problema: Fornire un modo semplice e riutilizzabile per scaricare o ricevere testo JSON, per parsarlo e prendere i dati nei tipi dei dati del linguaggio Java.

Soluzione: AdvancedJsonReaderImpl incapsula sia la logica di richiesta HTTP GET - leggendo tutto il contenuto da un URL tramite HttpURLConnection e BufferedReader - sia quella di parsing del testo JSON in un oggetto JsonObject di GSON. A partire dalla stringa grezza memorizzata in json_raw_text, il metodo parseAndSetJson() costruisce l'albero JSON_BODY e abilita l'istanza all'uso. I metodi getJsonArray, getFromJson e

i convenienti getString, getInt, getDouble ecc. consentono di specificare un percorso in notazione “punto” (ad esempio "daily.temperature_max") e ottenere direttamente il valore con il tipo desiderato, gestendo internamente il ritrovamento del nodo genitore e le conversioni. Se si tenta di settare il JSON più di una volta, o di leggere prima di averlo impostato, le funzioni assertNotAlreadySet() e assertIsSet() sollevano eccezioni chiare, evitando incoerenze di stato. In questo modo, qualunque client che abbia bisogno di consumare un’API REST JSON può affidarsi a un unico oggetto in grado di scaricare, parsare, navigare, estrarre ed eventualmente controllare l’esistenza di elementi nel documento, senza scrivere ogni volta il boilerplate di parsing e validazione.

Weather

Figura 2.3: Schema UML di Weather

Problema: Fornire in modo automatizzato dati meteorologici (correnti e previsioni) per una specifica località, aggregando informazioni da diverse API e arricchendole con dati aggiuntivi (es. popolazione della città).

Soluzione: Il codice *AllWeather* risolve il problema della raccolta e dell'organizzazione automatica di dati meteorologici per una determinata località sfruttando diversi servizi esterni. Dopo aver impostato latitudine, longitudine e nome della città, la classe invia una richiesta all'API Open-Meteo per ottenere le previsioni orarie e giornaliere: queste vengono quindi elaborate e raggruppate in strutture dati interne che associano a ogni data e ora i valori di temperatura, umidità, precipitazioni, vento, pressione e altre grandezze, nonché i riassunti giornalieri con minime, massime, durata della luce e indice UV. Parallelamente, un secondo endpoint fornisce il meteo attuale, la cui risposta viene memorizzata con un meccanismo di caching che evita di rifare la chiamata più di una volta ogni venti minuti. Per esigenze di dettaglio puntuale, il metodo dedicato arrotonda l'ora richiesta al quarto d'ora più vicino e recupera dati di precipitazione, visibilità e vento per quell'istante specifico. Infine, per arricchire le informazioni, la classe estrae l'altitudine dal JSON dell'API e, tramite uno scraping con Jsoup del sito "ilMeteo.it", cerca di ricavare il numero di abitanti della città d'interesse. Il risultato è un'interfaccia unica che unifica e normalizza dati provenienti da più fonti, offrendo metodi per ottenere previsioni a breve e medio termine, condizioni correnti e dettagli puntuali, il tutto con conversioni automatiche di unità di misura e traduzioni delle direzioni del vento in termini comprensibili.

CSVParser

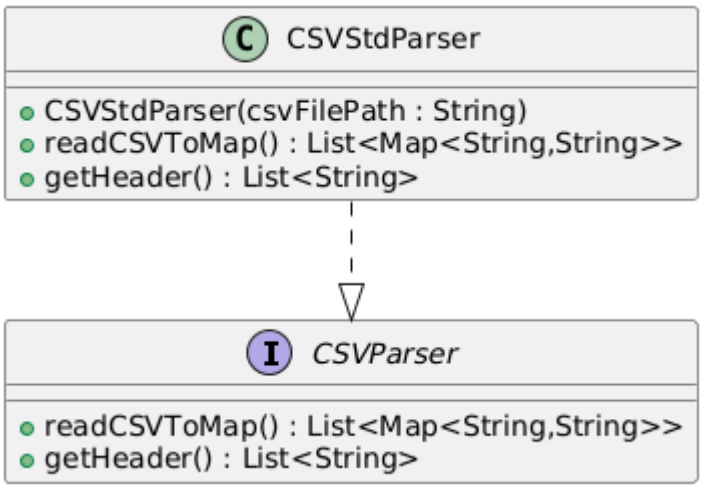

Figura 2.4: Schema UML di CSVParser

Problema: Convertire in modo semplice e riutilizzabile il contenuto di un file CSV in una struttura dati Java, evitando

di gestire manualmente l'analisi di righe, separatori e casi di colonne mancanti.

Soluzione: La classe estende *CSVReader* di *OpenCSV* e implementa l'interfaccia *CSVParser*, sfruttando *readAll* per caricare tutte le righe.
getHeader estrae la prima riga come lista di nomi di colonna. *readCSVToMap* prende l'intestazione e, per ogni riga successiva, crea una mappa abbinando ogni nome di colonna al corrispondente valore di cella (o a stringa vuota se mancante), restituendo infine la lista di tutte le mappe. Questo approccio incapsula errori di I/O e parsing in eccezioni propagate, lasciando al chiamante la gestione degli stessi.

LocationSelector

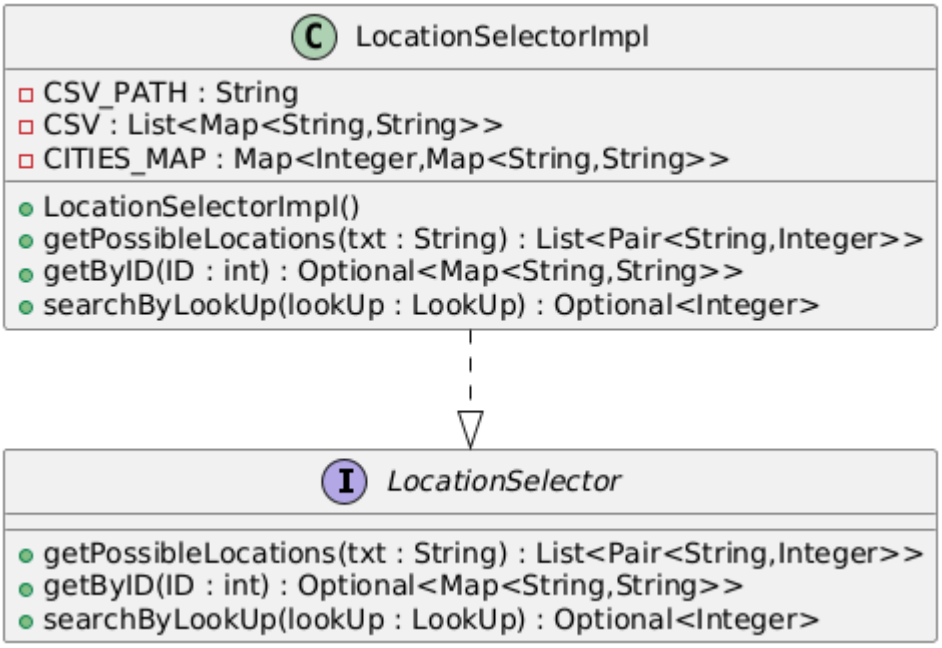

Figura 2.5: Schema UML di LocationSelector

Problema: Fornire un modo efficiente di cercare e recuperare oltre 47'000 record di città (worldcities.csv) in memoria, che permetta di accedere facilmente alle città con diversi metodi, compreso il look up dell'IP.

Soluzione: All'istanziazione viene utilizzato il parser CSV per leggere tutte le righe in una lista di mappe e contemporaneamente costruire una mappa hash che associa a ciascun ID la corrispondente mappa di valori, permettendo accessi in tempo costante; le funzioni di ricerca testuale effettuano uno scan case-insensitive della lista in memoria,

quella per ID sfrutta direttamente la hashmap e il lookup geografico confronta nome città e codice paese per restituire l'ID corrispondente, mentre un eventuale errore di caricamento interrompe immediatamente l'avvio dell'applicazione.

Moon Phases

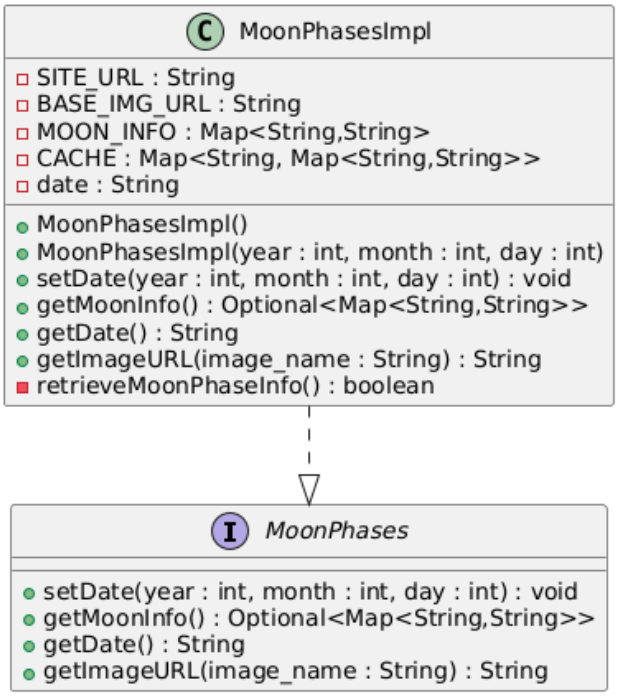

Figura 2.6: Schema UML di MoonPhases

Problema: La classe si preoccupa di ottenere per una data specifica le informazioni sulla fase lunare (descrizione, percentuale di illuminazione e immagine corrispondente) da un sito esterno (moongiant.com), superando la mancanza di un'API ufficiale e garantendo al contempo prestazioni ragionevoli evitando richieste ripetute per la stessa data.

Soluzione: All'impostazione della data, formattata nel formato richiesto dal sito (dd/MM/yyyy), il metodo di recupero controlla anzitutto una cache in memoria; se il dato non è già presente, esegue uno scraping della pagina HTML tramite Jsoup, estrae i singoli elementi (testo della fase, percentuale e nome del file immagine) gestendo eventuali errori di layout con un blocco di try/catch, popola una mappa interna e ne conserva una copia immutabile per future richieste, esponendo infine tutte le informazioni in un Optional per segnalare eventuali fallimenti di parsing.

Weather Mode

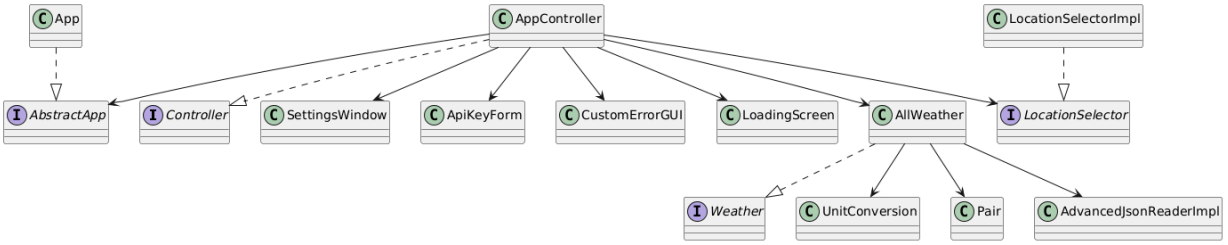

Figura 2.7: Schema UML per la parte WeatherMode

Nella **WeatherMode** ogni componente ha un ruolo preciso: il **controller** fa da regista, intercetta le azioni dell'utente (come la scelta della città), chiede al modello i dati meteorologici e poi aggiorna **l'interfaccia**, gestendo anche eventuali finestre di caricamento o di errore. Il **modello** si occupa di scaricare, interpretare e convertire le previsioni — per esempio trasformando le temperature da Celsius a Fahrenheit — e tiene traccia della località selezionata. Infine, la **view** è ciò che si vede sullo schermo: mostra i dati meteo, offre i moduli per inserire la chiave API e per modificare le impostazioni, e invia al controller i comandi che inserisci. In questo modo l'app separa chiaramente il compito di presentare le informazioni, di gestire la logica e di recuperare i dati, rendendo il flusso più lineare e il codice più semplice da mantenere.

2.2.2 Lorenzo Magni

Creazione sicura e controllata della richiesta di viaggio

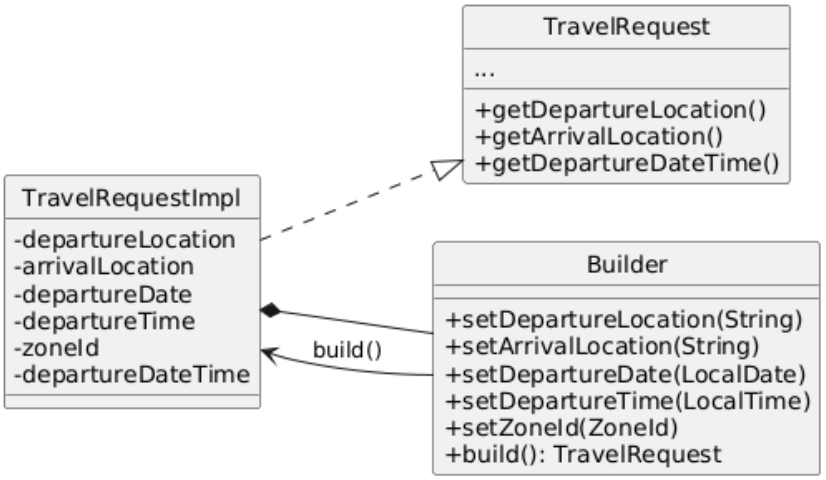

Figura 2.8: schema UML di TravelRequestImpl e del Builder associato

Problema: Uno dei primi problemi affrontati nella modalità viaggio è stato definire come rappresentare in modo robusto, coerente e immutabile una richiesta di viaggio. Questa richiesta deve contenere dati eterogenei (nomi delle città, identificativi PlaceID, orari e date di partenza, fuso orario), che provengono da input utente e da chiamate a servizi esterni (Google Places). Le sfide principali sono:

- Evitare richieste parziali o incoerenti.
- Gestire correttamente la combinazione tra LocalDate, LocalTime e ZoneId.
- Evitare oggetti parziali o inconsistenti che avrebbero reso fragile l'interazione con il model.
- Permettere la creazione progressiva dell'oggetto, man mano che i dati diventano disponibili.
- Mantenere l'immutabilità e garantire la coerenza logica interna.

Soluzione: Per affrontare il problema si è introdotto l'oggetto TravelRequestImpl che implementa l'interfaccia TravelRequest. Esso rappresenta una richiesta completa di viaggio, incapsulando tutte le informazioni essenziali: località di partenza e arrivo, identificativi geografici, data, orario e fuso orario. Questo oggetto è immutabile e può essere istanziato solo attraverso un builder dedicato.
Il builder consente di:

- Accumulare i dati gradualmente, man mano che vengono recuperati o inseriti.
- Derivare automaticamente informazioni composte, come la data e ora completa (ZonedDateTime).

- Validare l'integrità dell'oggetto prima della sua creazione, evitando errori a runtime.

Questa soluzione rispetta i principi di buona progettazione orientata agli oggetti:

- **Separazione delle responsabilità:** la costruzione è separata dalla logica applicativa.
- **Encapsulation & immutability:** l'oggetto risultante è completo, non modificabile e sicuro da utilizzare.
- **Facilità d'uso:** il builder fornisce un'interfaccia chiara per l'utente del componente.

Pattern riconosciuti

- **Builder:** per costruire oggetti complessi in modo controllato e leggibile.
- **Information Hiding:** l'utente del sistema non ha accesso diretto al costruttore o allo stato interno.

Alternative considerate: L'utilizzo di un costruttore tradizionale con molti parametri è stato scartato per motivi di leggibilità e fragilità. Anche l'uso di un oggetto mutabile con metodi setter è stato evitato per ridurre il rischio di errori e mantenere un comportamento prevedibile durante l'uso nel model.

Gestione e isolamento delle API esterne

1) Isolamento e astrazione delle API esterne

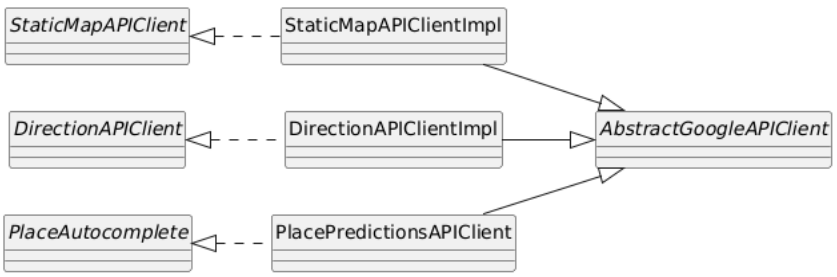

Figura 2.9: struttura generale di un ApiClient

Problema: Le funzionalità principali della modalità viaggio richiedono un'interazione intensiva con API esterne, tra cui:

- Google Directions per il calcolo dei percorsi.
- Google Places per l'autocompletamento dei luoghi.
- Google Static Maps per rappresentare visivamente il tragitto e le condizioni avverse.

Tali API, se utilizzate direttamente all'interno del model, avrebbero introdotto una dipendenza diretta con il provider, rendendone difficile la sostituzione.

Soluzione: Per affrontare questo problema, si è scelto di introdurre un insieme di interfacce dedicate che rappresentano le funzionalità richieste dal sistema:

- DirectionApiClient
- PlaceAutocomplete
- PlaceDetails
- StaticMapApiClient

Queste interfacce si trovano in un package interno al model, ma sono da esso utilizzate senza conoscerne l'implementazione sottostante.
Le classi che implementano queste interfacce estendono tutte una classe astratta comune: AbstractGoogleApiClient, che fornisce funzionalità condivise come gestione della chiave API, costruzione dell'URL e invio delle richieste HTTP.

In particolare, AbstractGoogleApiClient incapsula:

- La gestione della chiave API.
- La gestione dell'URL di base che identifica l'API utilizzata
- Il parsing della risposta HTTP in JSON.

Pattern riconosciuti

- **Adapter:** ogni classe concreta (es. DirectionApiClientImpl, StaticMapApiClientImpl) funge da adattatore tra il formato delle API e gli oggetti interni.
- **Dependency Inversion Principle:** il model dipende da interfacce astratte, non da implementazioni concrete.

2) Modularità e costruzione delle richieste alle API

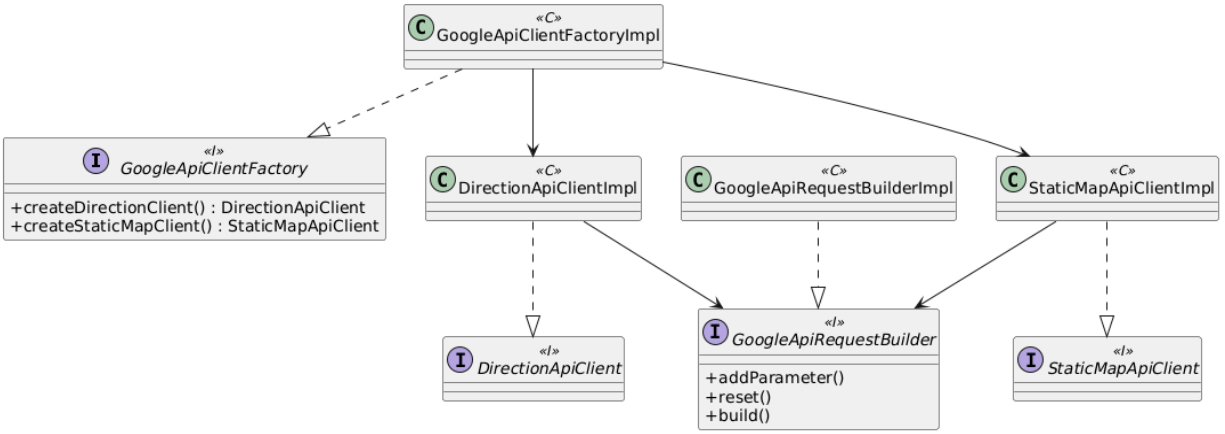

Figura 2.10: struttura di GoogleApiClientFactory

Problema: Tutte le API esterne utilizzate nella modalità viaggio (Google Maps, Static Maps, Place Details, ecc.) richiedono la costruzione di richieste HTTP complesse. Ogni servizio richiede una chiamata personalizzata, con parametri specifici, vincoli di encoding e gestione della chiave API. In assenza di una struttura modulare, le chiamate avrebbero rischiato di essere costruite in modo manuale e ripetitivo all'interno di ciascun client, con conseguenti problemi di:

- Duplicazione di codice
- Scarsa leggibilità e possibilità di errori nell'assemblaggio delle URL.
- Accoppiamento tra costruzione della richiesta e logica del client
- Mancanza di controllo centralizzato sulla configurazione (es. chiave API).

Inoltre, è necessario garantire che l'accesso ai vari ApiClient sia coerente e standardizzato in tutto il sistema, evitando che ogni componente decida arbitrariamente come istanziarli.

Soluzione: Per affrontare questi problemi, si è deciso di introdurre due componenti fondamentali:

a) **GoogleApiRequestBuilderImpl: Costruttore modulare delle richieste**
   Si tratta di una classe dedicata alla costruzione fluente e sicura delle richieste HTTP verso i servizi Google.
   Questa classe incapsula:
   - L'aggiunta di parametri alla query (addParameter(...)).
   - La definizione dell'URL specifico in base al servizio da interrogare.
   - La codifica dell'URL finale.

   Il builder viene utilizzato da tutte le sottoclassi di AbstractGoogleApiClient, garantendo uniformità e riutilizzo nella costruzione delle chiamate.

   Vantaggi:
   - Centralizzazione del formato delle richieste.
   - Separazione tra logica di costruzione e invio della richiesta.
   - Maggiore leggibilità e sicurezza del codice.

b) **GoogleApiClientFactoryImpl – Factory per l'accesso ai client**

   Questa classe rappresenta una fabbrica centralizzata per la creazione delle istanze dei vari ApiClient (Directions, Static Map, Place Autocomplete, ecc.).
   All'interno, la factory configura correttamente ciascun client, inserendo la chiave API e altri parametri

necessari. In questo modo, tutti i componenti che necessitano di accedere a un servizio esterno lo fanno tramite la factory, senza doversi occupare della logica di inizializzazione o configurazione.

Vantaggi:

- Evita la duplicazione della logica di creazione.
- Rende più semplice introdurre modifiche.
- Apre alla futura sostituzione dei client con versioni alternative.

Pattern riconosciuti

- **Builder Pattern:** utilizzato in GoogleApiRequestBuilderImpl per creare richieste complesse con una sintassi fluente, leggibile e sicura.
- **Abstract Factory Pattern:** GoogleApiClientFactoryImpl centralizza l'istanziazione e configurazione dei client.
- **Single Responsibility Principle:** ogni classe ha un compito specifico e ben definito (costruzione, creazione, esecuzione).

Alternative considerate: Inizialmente, ciascun client implementava autonomamente la costruzione del proprio URL e l'invio della richiesta. Questo approccio, nonostante fosse più semplice, generava molto codice duplicato e accoppiato. La rifattorizzazione attraverso builder e factory ha permesso di rendere il sistema più modulare, coerente e facilmente manutenibile.

Rappresentare e gestire le condizioni meteo nella modalità viaggio

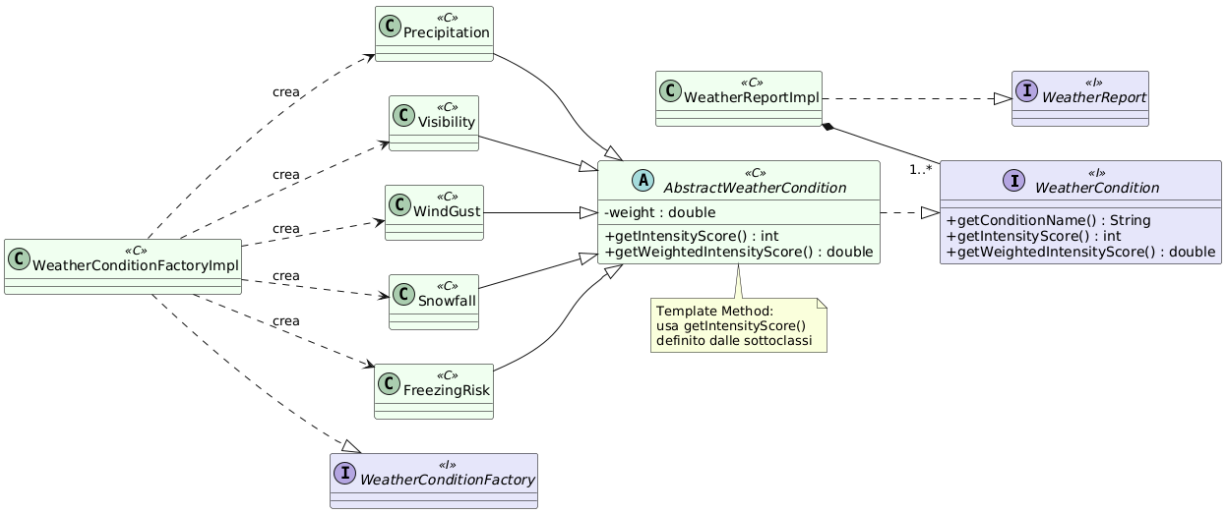

Figura 2.11: Rappresentazione delle condizioni meteorologiche

Problema: Durante l'elaborazione di un percorso, uno degli aspetti cruciali per la modalità viaggio consiste nel valutare l'impatto delle condizioni meteorologiche su ogni tratto del tragitto. Tuttavia, questa esigenza comporta diverse problematiche:

- Le condizioni meteorologiche sono molteplici e diverse (es. pioggia, neve, visibilità, vento, rischio gelo), ciascuna con logiche di valutazione differenti.
- È necessario raccogliere, rappresentare e valutare queste condizioni in un formato uniforme, pur mantenendo la possibilità di gestire ogni condizione in modo indipendente.
- L'intensità deve inoltre essere pesata per riflettere l'importanza relativa di ciascun fenomeno nel contesto di un checkpoint.
- Serve un meccanismo flessibile per creare un report meteo aggregato per ogni checkpoint, tenendo conto della gravità complessiva delle condizioni. Inoltre questo meccanismo deve garantire una semplice futura estendibilità tramite l'aggiunta di nuove condizioni meteorologiche senza comportare modifiche al codice esistente.

Il problema può quindi essere sintetizzato così: come rappresentare più tipi di condizioni meteorologiche diverse, trattarle sfruttando il polimorfismo e aggregarle in un report coerente.

Soluzione: Per affrontare queste sfide, è stato progettato un sottosistema modulare basato su un insieme di interfacce e classi astratte, pensato per garantire flessibilità, estendibilità e disaccoppiamento.

- Interfaccia WeatherCondition
  -Rappresenta una generica condizione meteo.
  -Definisce metodi comuni per ottenere descrizione, punteggio di rischio e altri indicatori utili.
  -È implementata da tutte le condizioni meteo specifiche.

- Classe astratta AbstractWeatherCondition
  -Fornisce un'implementazione base per le funzionalità comuni.
  -Implementa il metodo getWeightedIntensityScore(), che calcola il punteggio pesato dell'intensità.
  -Questo metodo segue il pattern **Template Method**: definisce la struttura dell'algoritmo e delega alle sottoclassi il calcolo dell'intensità tramite il metodo getIntensityScore().

- Condizioni concrete (es. Precipitation, Visibility, Snowfall, WindGust, FreezingRisk)
  -Ogni classe modella una condizione meteorologica reale.
  -Implementano il metodo getIntensityScore() in base a criteri specifici.
  -Ereditano il comportamento di getWeightedIntensityScore() senza doverlo riscrivere.
  -Incapsula la propria logica di valutazione della gravità.

- Interfaccia WeatherConditionFactory e la sua implementazione
  -Costruisce dinamicamente le condizioni meteo concrete a partire da dati grezzi (es. temperatura, velocità vento, visibilità, ecc.).
  -Incapsula la logica di creazione, seguendo il pattern Factory.

- Interfaccia WeatherReport e implementazione WeatherReportImpl
  -Aggrega tutte le condizioni meteo associate a un punto del percorso.
  -Permette di calcolare il punteggio complessivo e individuare rapidamente eventuali rischi o condizioni avverse.
  -Espone un'interfaccia semplice e adatta all'interazione con la logica della modalità viaggio.

Pattern riconosciuti

- **Template Method:** getWeightedIntensityScore() definisce una struttura di calcolo comune, delegando a getIntensityScore() il comportamento specifico.
- **Composite:** WeatherReportImpl aggrega una lista di WeatherCondition, trattandole in modo uniforme.
- **Factory:** WeatherConditionFactoryImpl costruisce dinamicamente le condizioni in modo disaccoppiato.
- **Strategy:** Ogni condizione concreta incapsula la propria logica di valutazione, selezionabile tramite la factory.

Alternative considerate: Una possibile alternativa era quella di utilizzare strutture dati generiche come mappe o JSON interni per rappresentare ogni condizione meteo. Tuttavia, questa scelta avrebbe ridotto la tipizzazione, la leggibilità e la manutenibilità del codice. Si è preferito, quindi, un approccio fortemente tipizzato e orientato agli oggetti, in cui ogni condizione è un'entità autonoma.

Mappare la risposta dell'API Directions in oggetti interni

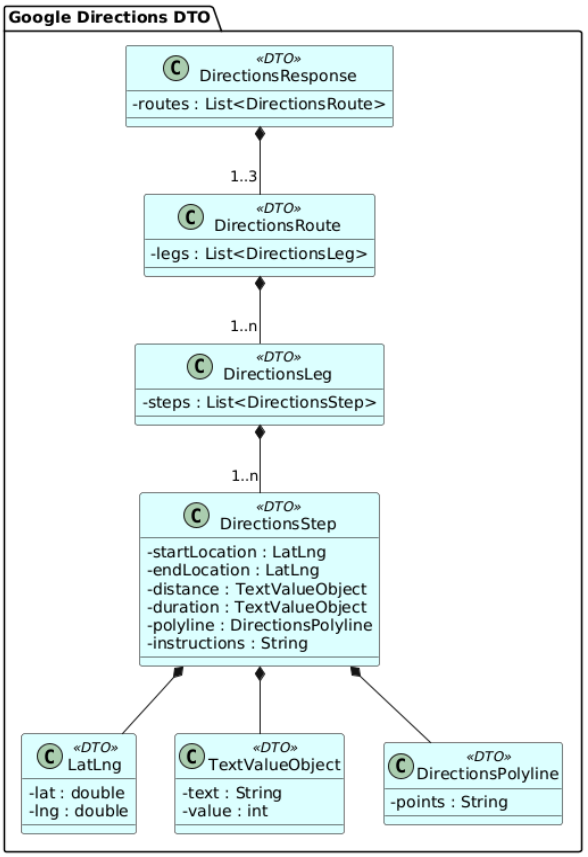

Figura 2.12: schema UML degli oggetti utilizzati per mappare la risposta dell'API Directions

Problema: La Google Directions API restituisce una risposta complessa in formato JSON contenente:

- Un array di possibili percorsi (routes)
- Per ogni percorso, un array di tratti (legs)
- Per ogni tratto, una lista dettagliata di segmenti (steps)
- Per ogni step, coordinate, distanza, durata e istruzioni testuali

L'obiettivo dell'app è quello di analizzare questi percorsi, suddividerli in checkpoint, e valutare le condizioni meteo lungo il tragitto. Per farlo, è necessario convertire la risposta JSON in una struttura tipizzata e navigabile, mantenendo flessibilità e manutenibilità. Il problema principale è quindi: come rappresentare questa struttura esterna in modo robusto, disaccoppiato e riutilizzabile all'interno dell'applicazione?

Soluzione: Sono state definite diverse classi DTO (Data Transfer Object) che riflettono fedelmente la gerarchia e i campi della risposta JSON.

Le principali classi includono:

- DirectionsResponse: rappresenta la risposta completa, contenente un array di DirectionsRoute.
- DirectionsRoute: contiene uno o più DirectionsLeg.
- DirectionsLeg: corrisponde a un segmento tra due località intermedie (o città), e contiene una lista di DirectionsStep.
- DirectionsStep: rappresenta un singolo step del percorso (es. svolta a destra) e include:
  - coordinate di inizio/fine (LatLng)
  - distanza e durata (TextValueObject)
  - polilinea compressa (DirectionsPolyline) per tracciare il tratto.

Classi di supporto come LatLng, TextValueObject e DirectionsPolyline completano il modello.
Questa struttura è stata progettata in modo:

- fedele al formato della risposta API
- tipizzato per evitare errori di runtime
- immutabile o comunque senza logica, secondo il principio dei DTO

Le istanze di queste classi vengono automaticamente create dalla libreria di deserializzazione JSON (es. Gson o Jackson), senza dover scrivere parsing manuale.

Pattern riconosciuti

- **DTO (Data Transfer Object)**: tutte le classi sono classi dati senza logica.
- **Separation of Concerns**: l'intero parsing della risposta JSON è separato dalla logica applicativa dell'app.
- **Information Hiding**: nessun oggetto esterno accede direttamente alla struttura JSON; interagisce solo con oggetti tipizzati.

Alternative

Inizialmente si sarebbe potuto optare per l'accesso diretto al JSON tramite strutture dinamiche (ad es. JsonObject, Map<String, Object>, ecc.), ma questa scelta avrebbe comportato:

- Maggiori rischi di errore (es. key non trovate)
- Minore leggibilità e riusabilità
- Difficoltà nell'autocompletamento e refactoring

La definizione di DTO personalizzati ha comportato un costo iniziale maggiore, ma ha garantito maggiore robustezza e chiarezza a lungo termine, rendendo il sistema facilmente estendibile in caso di evoluzioni dell'API.

Rappresentare i tratti del percorso come unità semplificate

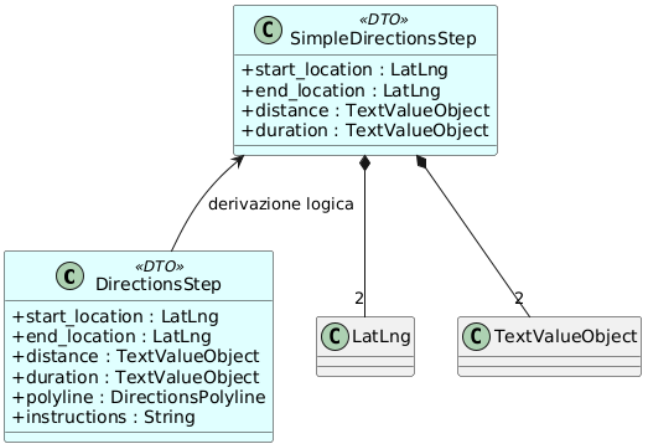

Figura 2.13: schema UML di un SimpleDirectionStep

Problema: La struttura dei dati fornita dall'API Directions (analizzata nel punto precedente) è estremamente dettagliata, ma non immediatamente adatta a operazioni fondamentali per l'applicazione, come:

- campionare punti intermedi a intervalli regolari lungo un percorso,
- effettuare analisi meteo localizzate,
- calcolare durate e distanze su segmenti elementari.

Serviva quindi una rappresentazione più semplice, compatta e navigabile, che mantenesse le informazioni essenziali e potesse essere processata con facilità da componenti successivi.

Soluzione: Per rispondere a questa necessità, è stata introdotta la classe SimpleDirectionsStep, che rappresenta un tratto del percorso in forma semplificata.

Ogni oggetto SimpleDirectionsStep contiene:

- una coppia di coordinate (inizio e fine)
- la durata del tratto
- la distanza del tratto

Tutti i campi sono immutabili e fortemente tipizzati.
Questa semplificazione avviene subito dopo il parsing della risposta dell'API e fornisce una rappresentazione pulita e uniforme del percorso, che può essere usata per estrarre i checkpoint.

Pattern riconosciuti

- **DTO semplificato:** SimpleDirectionsStep è un oggetto dati che astrae la struttura complessa originale.
- **Information Hiding:** nasconde la complessità della struttura Google e fornisce solo i dati utili.

Alternative
Si sarebbe potuto usare direttamente la classe DirectionsStep della gerarchia dei DTO, ma questa includeva riferimenti superflui e una struttura più complessa non necessaria.
La creazione di un oggetto intermedio ha portato a maggiore chiarezza, facilità di test e flessibilità nell'uso in fasi successive (come la generazione dei checkpoint e l'analisi meteo).

Suddividere il percorso in step omogenei

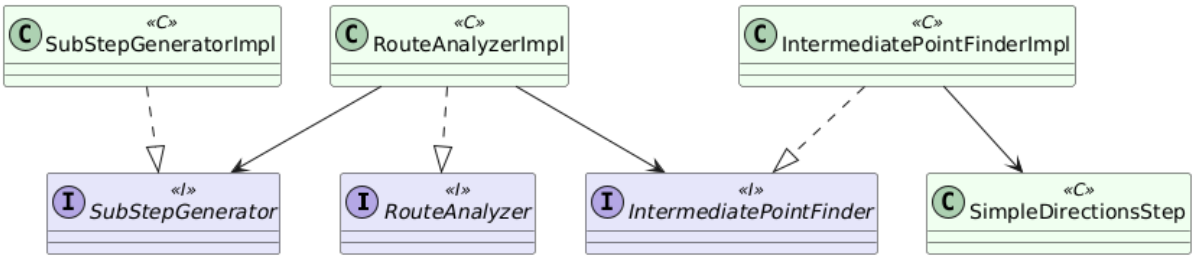

Figura 2.14: schema UML di RouteAnalyzer che evidenzia l'utilizzo di IntermediatePointFinder e SubStepGenerator

Problema: Per valutare le condizioni meteo lungo un tragitto tra due città, è necessario analizzare il percorso in modo distribuito. Il percorso restituito da Google Directions è composto da step di lunghezza irregolare. Per una verifica meteo affidabile servono tratti omogenei (es. ≈ 30 km) su cui campionare i dati.

Soluzione: La suddivisione è coordinata da **RouteAnalyzerImpl** che ha il compito di trasformare l'intero percorso in una serie di SimpleDirectionsStep di lunghezza omogenea. Per far questo sfrutta:

1. **IntermediatePointFinderImpl** che raggruppa DirectionsStep brevi,
2. **SubStepGeneratorImpl** che suddivide DirectionsStep molto lunghi

Successivamente la lista di SimpleDirectionStep così ottenuta verrà passata al componente CheckpointGeneratorImpl per la generazione dei checkpoint, ovvero punti spazio-temporali che rappresentano il passaggio del veicolo, tenendo conto della data/ora di partenza e del tempo di percorrenza stimato per ogni tratto.

Pattern riconosciuti

- **Facade:** RouteAnalyzerImpl nasconde la complessità dei calcoli interni.
- **Strategy:** Finder e Generator sono dietro interfacce e quindi facilmente sostituibili.

Alternative considerate

Accorpare tutto in un'unica classe avrebbe ridotto la testabilità e reso più difficile la possibilità di modificare le strategie di suddivisione del percorso. Delegare la logica al controller UI avrebbe violato la separazione MVC.

Arricchire i checkpoint e valutare il percorso

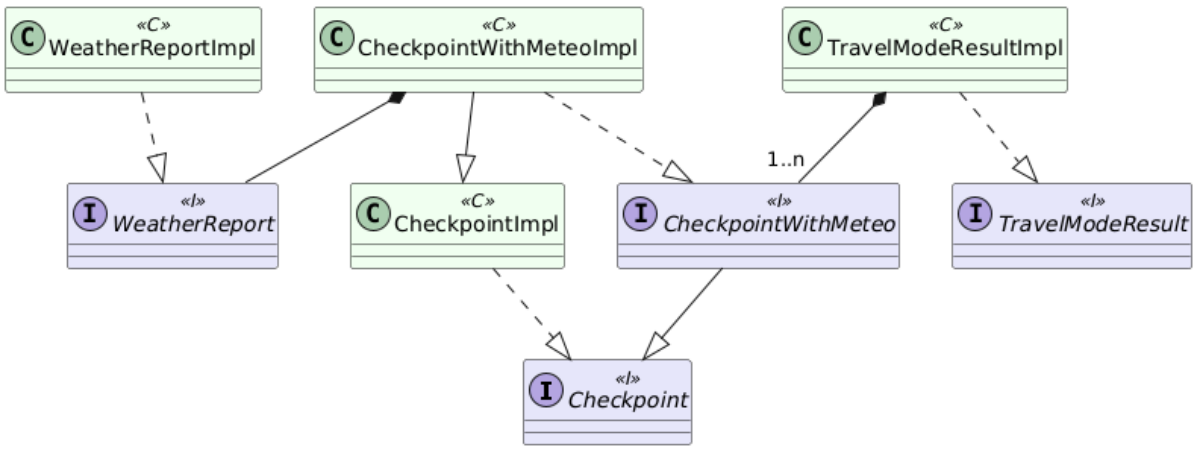

Figura 2.15: schema UML della struttura di CheckpointWithMeteo e TravelModeResultImpl

Problema: Dopo aver generato i checkpoint temporizzati lungo il tragitto, il sistema deve:

1. Ottenere le condizioni meteo previste per ciascun checkpoint, considerando coordinate e orario.
2. Valutare l'impatto delle condizioni meteorologiche sull'intero percorso.
3. Restituire un risultato aggregato, interpretabile dall'utente, che sintetizzi l'affidabilità e la sicurezza del percorso dal punto di vista climatico.

Soluzione: Dopo aver generato i checkpoint, ciascuno di essi viene arricchito con le informazioni meteo previste nel luogo e nell'orario stimato di passaggio. Ogni checkpoint arricchito viene rappresentato tramite un oggetto CheckpointWithMeteo, che include sia le coordinate la data/ora di passaggio, sia un

WeatherReport che raccoglie tutte le condizioni atmosferiche rilevanti. Questa fase è gestita da un componente chiamato WeatherInformationService.
Infine, l'intera lista di checkpoint arricchiti viene utilizzata per costruire un oggetto TravelModeResult, che sintetizza i vari risultati ottenuti dall'analisi del percorso. Questo risultato comprende un punteggio numerico, una breve descrizione del percorso, la durata di percorrenza stimata e una mappa per permettere una visualizzazione grafica dei risultati.

Pattern riconosciuti

- **Decorator**, utilizzato nella classe CheckpointWithMeteo che estende la funzionalità del semplice Checkpoint aggiungendo informazioni meteo, senza modificarne l'interfaccia.
- **Data aggregator:** TravelModeResultImpl raccoglie informazioni da tutti i checkpoint e le sintetizza in un risultato compatto e facilmente interpretabile.

Coordinare la logica del model: TravelModeModelImpl

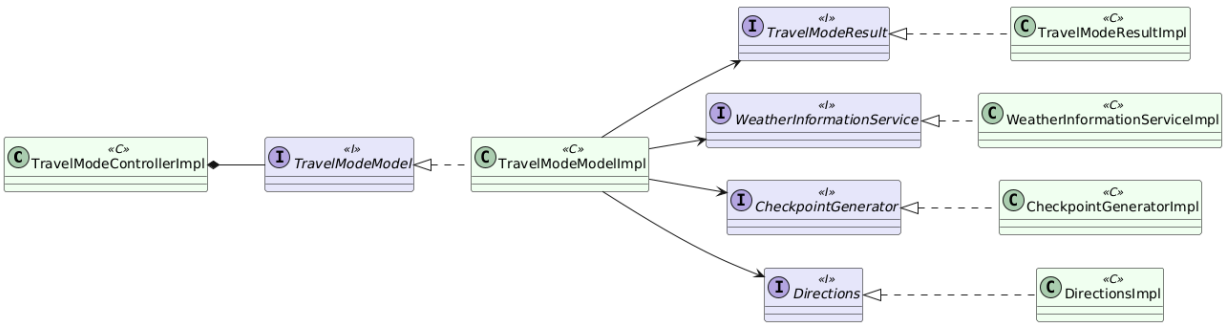

Figura 2.16: diagramma UML della struttura di TravelModeModelImpl

Problema: Come si può notare dalle sezioni precedenti, la modalità viaggio dell'applicazione coinvolge una serie di componenti eterogenei, ciascuno responsabile di una parte specifica del processo: creazione dei checkpoint, recupero dei dati meteo, calcolo del punteggio finale, ecc. Per rendere questa logica utilizzabile dal controller, era necessario introdurre un modulo capace di coordinare tutti questi elementi in modo trasparente, mantenendo al tempo stesso una buona separazione tra la logica applicativa e l'interfaccia utente.

Soluzione: La soluzione è rappresentata dalla classe TravelModeModelImpl, che funge da facciata verso il sottosistema model della modalità viaggio. Implementando l'interfaccia TravelModeModel, questa classe riceve le richieste dal controller, come la creazione di un nuovo percorso (TravelRequest) o l'analisi delle condizioni meteo, e le inoltra alle componenti specializzate del modello.

Internamente, TravelModeModelImpl si occupa di orchestrare la catena completa delle operazioni: dalla chiamata al modulo Directions, alla generazione dei checkpoint, fino alla valutazione del risultato meteorologico tramite TravelModeResultImpl. Inoltre, gestisce le eccezioni e gli eventuali fallimenti nella comunicazione con le API, fornendo al controller un'interfaccia semplice e stabile.
Questa architettura consente di isolare completamente la logica di elaborazione dal livello di presentazione, rendendo l'app più modulare e manutenibile. La presenza dell'interfaccia TravelModeModel favorisce infine la testabilità e la possibilità di sostituzioni future.

Pattern riconosciuto

Il ruolo di TravelModeModelImpl ricalca quello del **Facade**, in quanto offre un'interfaccia unificata e semplificata per interagire con un insieme complesso di sottosistemi. Allo stesso tempo, implementando un'interfaccia, segue anche il principio **Dependency Inversion**, facilitando il disaccoppiamento tra controller e model.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Tutti i test di seguito riportati sono stati creati usando JUnit.

3.1.1 Giacomo Casadei

- **AllWeatherTest**: per questa classe, istanziata in setUp con coordinate di Roma per isolare ogni prova da chiamate esterne. Vengono verificate la conversione dei gradi in direzioni di vento in italiano e il reset delle cache al cambio di località, oltre all'arrotondamento degli orari al quarto d'ora più vicino e al controllo del tempo trascorso rispetto a una soglia, quest'ultimi testati via reflection sui metodi privati.
- **IPLookUpTest**: grazie a *MockedConstruction* di *Mockito*, simulo la creazione di *AdvancedJsonReaderImpl* senza chiamate di rete: verificano che lookup() restituisca Optional.empty() con tutti i tentativi falliti e che, al primo successo, popoli correttamente IP, nazione, città e coordinate invocando il costruttore una sola volta. Inoltre, via reflection viene testato il metodo privato clear() per assicurarsi che resetti i campi ip e coords allo stato iniziale.
- **LocationSelectorTest**: usando *Mockito*, *MockedConstruction* per intercettare la creazione di *AdvancedJsonReaderImpl* senza chiamate di rete: verifica che lookup() torni vuoto e resetti ip e coords dopo 10 tentativi falliti, che popoli correttamente i campi al primo successo costruendo

una sola istanza, e, via reflection, che il metodo privato clear() azzeri effettivamente ip e coords.

3.1.2 Lorenzo Magni

Durante lo sviluppo della modalità viaggio è stata adottata una strategia di testing focalizzata principalmente sulla logica del model, con l'obiettivo di validare i componenti critici responsabili dell'elaborazione del percorso. L'approccio seguito ha privilegiato l'uso di unità di test per le singole classi, sfruttando la struttura modulare dell'architettura per isolare ogni funzionalità da testare. I test, scritti con il framework JUnit, sono stati eseguiti regolarmente durante lo sviluppo per garantire la correttezza delle funzionalità principali.

Le classi IntermediatePointFinderImpl, RouteAnalyzerImpl e CheckpointGeneratorImpl sono state testate per verificare la corretta generazione dei punti intermedi lungo il percorso, la suddivisione in step omogenei (SimpleDirectionsStep) e la creazione dei checkpoint. Particolare attenzione è stata posta sulla gestione di casi limite, come distanze molto brevi o distribuzioni non regolari.

Sono state inoltre testate le componenti di supporto, come il decoder delle polilinee (PolylineDecoderTest) e la classe per il calcolo delle distanze geografiche (GeographicDistanceCalculatorTest), entrambe essenziali per l'analisi e la rappresentazione del tragitto.

Infine, WeatherScoreCategoryTest ha permesso di verificare la corretta classificazione qualitativa dei punteggi meteo, assicurando coerenza tra il valore numerico e la categoria assegnata.

Nel complesso, il sistema di test ha coperto in modo significativo tutte le funzionalità principali della modalità viaggio, fornendo una base solida per identificare eventuali anomalie e migliorare progressivamente la qualità del codice.

3.2 Note di sviluppo

3.2.1 Giacomo Casadei

- **Utilizzo di lambda function**: utilizzate in vari punti del progetto per semplificare e rendere più leggibile il codice. Un esempio significativo è visibile qui: https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/284f72f95a3c6b465f3561c6ef88b34b80b63ed7/src/main/java/org/app/weathermode/view/CustomErrorGUI.java#L43
- **Utilizzo di stream**: utilizzati in vari punti del progetto per semplificare e rendere più leggibile il codice. Un esempio significativo è visibile qui: https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/284f72f95a3c6b465f3561c6ef88b34b80b63ed7/src/main/java/org/app/weathermode/controller/AppController.java#L428
- **Utilizzo di Optional**: largo uso degli Optional, un esempio è visibile qui: https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/284f72f95a3c6b465f3561c6ef88b34b80b63ed7/src/main/java/org/app/weathermode/model/AllWeather.java#L212
- **Utilizzo di librerie esterne**: nel progetto sono state usate diverse librerie esterne fra cui JavaFX, Gson, Jackson e Jsoup.

3.2.2 Lorenzo Magni

Di seguito sono elencate alcune funzionalità avanzate del linguaggio Java e dell'ecosistema utilizzate nello sviluppo della modalità viaggio:

- **Uso delle Stream API**:
  Utilizzate ampiamente per elaborare collezioni e strutture complesse. Ad esempio, nella classe RouteAnalyzerImpl: https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/analysis/impl/RouteAnalyzerImpl.java#L64-L67

  Un altro esempio si trova nella classe DirectionsImpl, nel metodo calculateRouteDuration:

https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/routing/impl/DirectionsImpl.java#L202-L204

- **Lambda expressions e method reference**
  In diverse parti del codice sono state impiegate lambda expressions e method reference per rendere il codice più conciso. Sono presenti in molte classi tra cui TravelModeControllerImpl, RouteAnalyzerImpl e WeatherInformationServiceImpl:

  https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/controller/TravelModeControllerImpl.java#L159-L160

  https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/analysis/impl/WeatherInformationServiceImpl.java#L67-L68

- **Uso di Optional**
  Utilizzati per gestire in modo sicuro la presenza o assenza di valori. Un esempio rilevante è DirectionsImpl, che utilizza Optional per restituire in modo esplicito risultati che potrebbero essere assenti come nella generazione dei percorsi alternativi e per forzare il chiamante a gestire l'assenza del risultato.
  https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/routing/impl/DirectionsImpl.java#L130

  Optional viene usato anche come campo di istanza per rappresentare informazioni che non sono immediatamente disponibili al momento della creazione dell'oggetto, ma che vengono eventualmente inizializzate in una fase successiva del flusso logico:
  https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/routing/impl/DirectionsImpl.java#L53-L55

- **Uso di API esterne (Google APIs)**
  Sono state integrate diverse API di Google, tra cui Directions API, Static Maps API e Place Autocomplete API. Queste sono incapsulate in client specifici come DirectionApiClientImpl, StaticMapApiClientImpl, PlaceDetailsApiClient:

https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/google/impl/StaticMapApiClientImpl.java#L54-L64

https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/google/impl/DirectionApiClientImpl.java#L17-L19

- **Uso di componenti avanzati della JDK (Networking)**
  La modalità viaggio sfrutta le API di basso livello del package java.net per eseguire richieste HTTP dirette. Un esempio significativo si trova nella classe responsabile della generazione delle mappe (StaticMapApiClientImpl), dove viene inviata una richiesta GET all'API Google Maps Static Map:
  https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/google/impl/StaticMapApiClientImpl.java#L55-L64

- **Uso di librerie esterne (JavaFX, Gson)**
  Il progetto integra diverse librerie esterne:
  - **JavaFX** è usato per la gestione dell'interfaccia grafica.
  - **Gson** è impiegato per il parsing automatico delle risposte JSON dalle API, tramite classi DTO come DirectionsResponse e DirectionsRoute.
  - **JUnit:** adottato per il testing automatico delle componenti principali della modalità viaggio.

  https://github.com/MagniLorenzo/OOP24-WeatherApp/blob/deb069c60e87da7bf2048dce9e52b513f62cca63/src/main/java/org/app/travelmode/model/google/impl/DirectionApiClientImpl.java#L62-L63

3.2.3 Altre note

I files contenenti il codice per la gestione dei file di configurazione, salvo *ConfigBuilder*, è stato sviluppato da una persona non più facente parte del gruppo, la cui logica è solo stata leggermente modificata da Giacomo per inserire il supporto dei null nella lettura e scrittura del file JSON di configurazione.

La parte che gestisce il look up dell'indirizzo IP, è stata implementata da un componente che non fa più parte del gruppo, ma il design (l'interfaccia) è stato fatto da Giacomo dopo la richiesta di aiuto da parte dell'ex componente.

All'interno della modalità viaggio è presente la classe PolylineDecoder, utilizzata per decodificare la polilinea restituita dalla Google Directions API.
Il codice di questa classe non è stato scritto manualmente, ma è stato adattato partendo da una delle implementazioni ufficiali fornite da Google, accessibile nella documentazione delle Google Maps Utility Libraries.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Giacomo Casadei

Il mio compito è stato inizialmente quello di sviluppare tutta la parte concernente il retrieving delle informazioni relative al meteo e le informazioni di una città scelta, nonché della parte di scelta della città. Dopo l'esclusione di due membri del gruppo originale, il mio compito è anche stato quello di sviluppare la GUI principale e il relativo controller per gestire la visualizzazione delle informazioni.
Questo progetto ha portato diverse difficoltà, in particolare dopo la riduzione del gruppo, ma questo ha portato anche un aumento di esperienza nella gestione del lavoro in gruppo compresi tutti gli imprevisti che ciò può comportare.
Questo progetto ha anche aumentato le mie abilità e conoscenza della programmazione ad oggetti nonché nel farmi conoscere ed approfondire pattern di programmazione.
Sono cosciente che il mio lavoro non è impeccabile e ci potrebbero essere alcuni aspetti da migliorare, ma sono complessivamente soddisfatto del mio operato.

4.1.2 Lorenzo Magni

Lo sviluppo della modalità viaggio all'interno dell'applicazione meteo ha rappresentato per me un'opportunità stimolante e formativa. Fin dalle prime fasi, il mio compito è stato quello di progettare e realizzare un modulo in grado di permettere all'utente di inserire un punto di partenza, una destinazione e una data di partenza, con l'obiettivo di ottenere un percorso analizzato anche dal punto di vista delle condizioni meteorologiche previste. L'idea di fondo era quella di proporre non solo il tragitto più breve, ma anche quello potenzialmente più sicuro dal punto di vista meteo.

Tra le sfide principali incontrate vi è stata la necessità di integrare questa funzionalità all'interno di un'applicazione

più ampia, costituita anche dalla modalità meteo, sviluppata da un altro componente del team. Questo ha reso evidente sin da subito la necessità di una struttura architetturale che mantenesse separate le due modalità, permettendo una gestione indipendente ma coerente, e riducendo al minimo l'accoppiamento tra le parti. Per questo motivo è stato introdotto un controller generale che funge da punto di contatto tra le due modalità, garantendo una comunicazione pulita e modulare che apre a future evoluzioni.

Nel corso dello sviluppo ho avuto modo di approfondire e applicare numerosi concetti avanzati della programmazione ad oggetti, con particolare riferimento ai design pattern (come Template Method, Factory, Strategy, MVC), alla gestione robusta delle API esterne, e all'uso di strumenti avanzati del linguaggio Java (come Stream, Optional, lambda expressions). Ho inoltre potuto consolidare l'uso di librerie esterne come JavaFX per la parte grafica, Gson per il parsing JSON, e JUnit per il testing. Tutto ciò mi ha permesso di acquisire nuove competenze tecniche ma anche una maggiore consapevolezza progettuale, soprattutto per quanto riguarda l'importanza della separazione delle responsabilità e della manutenibilità del codice.

Nonostante la struttura progettuale sia risultata nel complesso solida e ben organizzata, durante lo sviluppo sono emerse alcune criticità che potrebbero essere oggetto di miglioramento.
Una prima area riguarda la gestione della concorrenza nelle chiamate alle API, che al momento avviene in modo sequenziale. In scenari futuri, soprattutto in presenza di più checkpoint o percorsi, potrebbe essere utile introdurre una gestione asincrona per migliorare la reattività del sistema.
Un altro aspetto perfezionabile è legato all'esecuzione dei test, che potrebbe essere ampliata ad altri elementi del progetto e che in alcuni casi risulta ancora troppo dipendente dall'implementazione interna piuttosto che dal comportamento astratto.

Guardando al futuro, la modalità viaggio presenta numerosi margini di miglioramento ed evoluzione. Tra le funzionalità che si potrebbero introdurre vi sono:

- una visualizzazione più precisa e dettagliata delle condizioni meteo lungo il percorso, con informazioni granulari per ogni tratto;
- un'interfaccia grafica più moderna e interattiva che permetta all'utente di analizzare nel dettaglio i risultati forniti;
- la possibilità di analizzare percorsi con più tappe intermedie, anche ottimizzate in base al meteo;

- un’integrazione con dati sul traffico per fornire percorsi che bilancino tempo di percorrenza e sicurezza;
- un’analisi più sofisticata delle condizioni atmosferiche, con modelli che considerino l’impatto combinato dei diversi fenomeni;
- la personalizzazione dei parametri di rischio, permettendo agli utenti di decidere quanto ogni tipo di condizione (pioggia, vento, nebbia…) influisce sul proprio stile di guida;
- la generazione automatica di report meteo dettagliati, anche con l’ausilio di tecniche di intelligenza artificiale per fornire suggerimenti contestualizzati;
- infine, un possibile motore di raccomandazione di itinerari turistici, che ordini le tappe ottimali in base alle condizioni meteo previste.

In sintesi, questo progetto ha rappresentato un’occasione preziosa per mettere in pratica concetti fondamentali della progettazione software e per affrontare, in modo guidato ma autonomo, un problema realistico e articolato. Le competenze acquisite non riguardano solo gli aspetti tecnici, ma anche la capacità di lavorare in un contesto collaborativo e modulare, con attenzione alla qualità e alla scalabilità del codice prodotto.

Appendice A

Guida utente

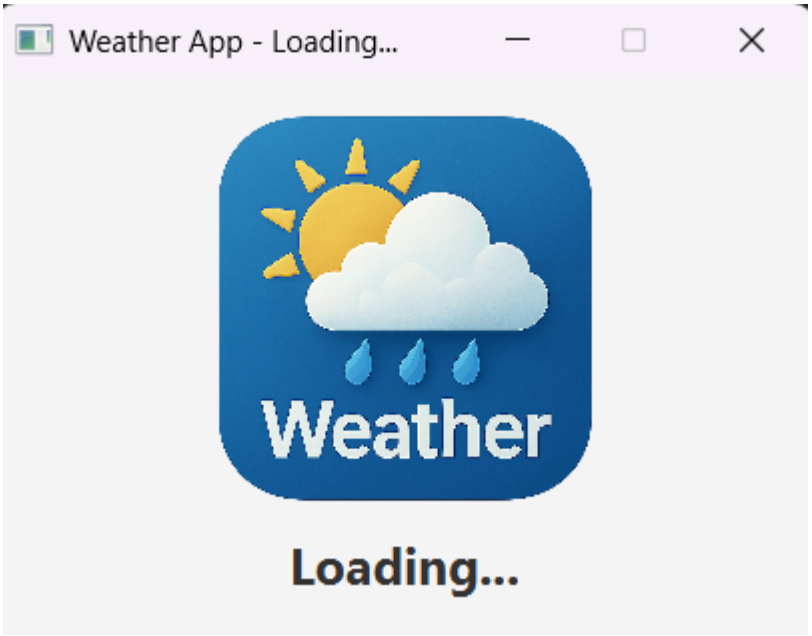

Figura A.1: Schermata di caricamento

Loading All'avvio dell'app viene subito mostrata una schermata di caricamento mentre il file csv che contiene le città viene caricato e parsato e messo in memoria. La schermata di caricamento dura diversi secondi ma il file delle città viene caricato una sola volta durante tutta la durata dell'applicazione.

IP LookUp e Scelta città

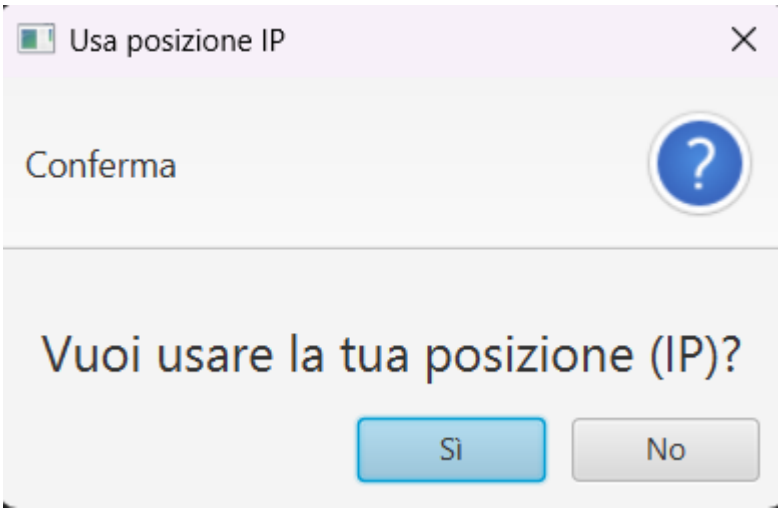

Figura A.2: Alert per scelta di effettuare l'IP Look Up

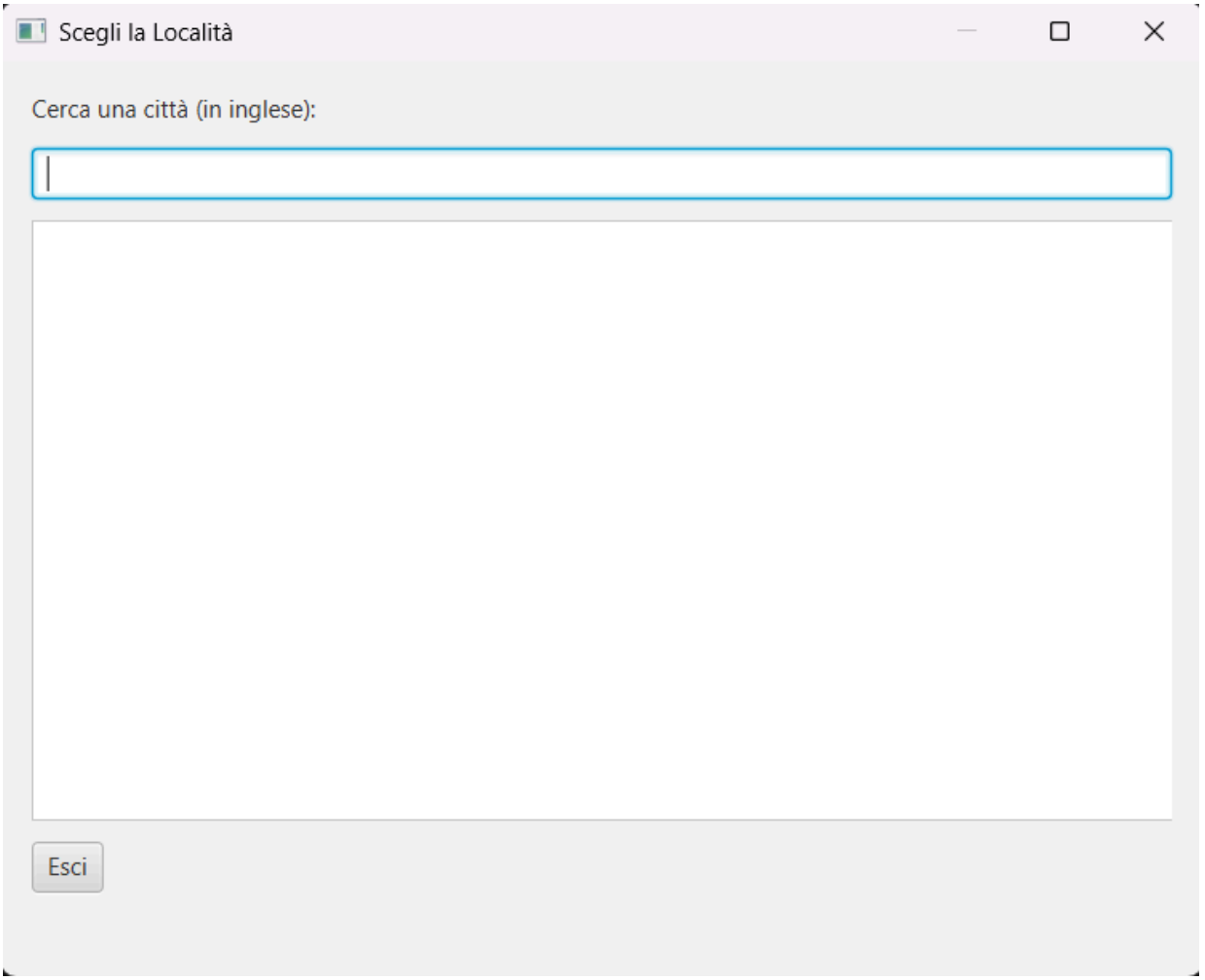

Figura A.3: GUI per scelta città

Finita la parte di loading, se nel file di configurazione non è presente la città, viene chiesto all'utente se vuole farsi localizzare tramite il suo indirizzo IP (Figura A.2), se sceglie "Sì", verrà fatto il look up dell'indirizzo IP e verranno visualizzate nella schermata principale le informazioni della città rilevata.
Se l'utente sceglie "No", allora verrà aperta la GUI per la scelta della città (Figura A.3), nella barra di ricerca bisognerà scrivere la città desiderata. La ricerca è case-insensitive e dopo aver digitato almeno 2 lettere, compariranno subito sotto l'input le possibili città.
Una volta individuata la città nell'elenco basterà cliccare sul pulsante desiderato, la GUI di scelta città viene chiusa e si apre la schermata principale con le varie informazioni.

Schermata principale (Weather Mode)

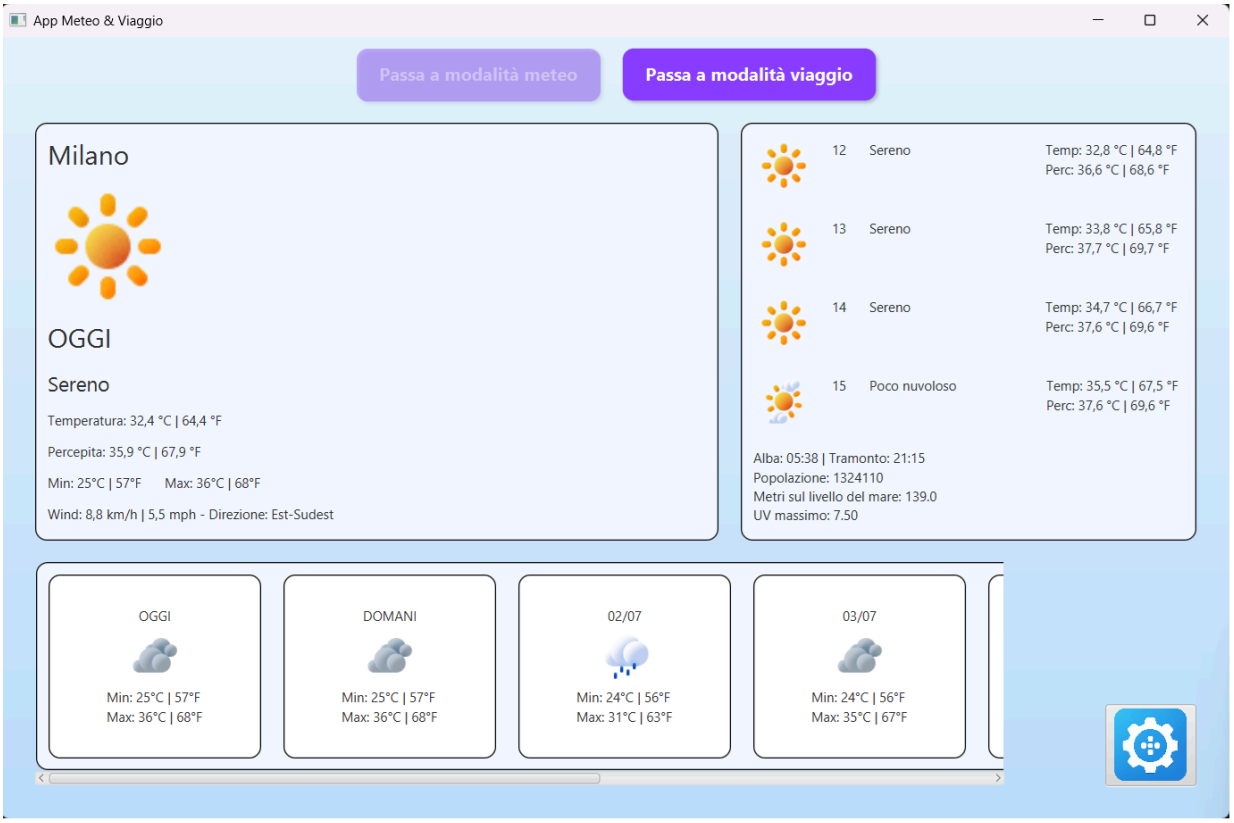

Figura A.4: Immagine della schermata iniziale

La schermata iniziale è impostata di default sulla modalità meteo. Nella barra in alto si trovano due pulsanti con i quali si può accedere alla Travel Mode e tornare alla visione della schermata principale.

Nella schermata principale è possibile visualizzare tutte le principali informazioni sul meteo e sulla città selezionata: in alto a sinistra il nome della città e le informazioni attuali su temperatura, sia reale che percepita e sui venti.
In alto a destra le informazioni meteo sulle ore a venire ed in fondo al box si trovano le informazioni sul giorno (ora di alba e tramonto, UV massimo della giornata) e sulla città (numero di abitati e metri sul livello del mare).

Nel box in fondo si trovano invece le temperature massime e minime ed il meteo previsto nei giorni della settimana.

In fondo a destra si trova il pulsante delle impostazioni, utilizzato per accedere ad informazioni aggiuntive e per modificare la città in esame.

Impostazioni e Informazione aggiuntive

Figura A.5: Schermata impostazioni e info aggiuntive

Nella schermata delle impostazioni si trovano tre pulsanti: i primi due servono per visualizzare informazioni aggiuntive, cioè i grafici delle temperature massime e minime della settimana, e la luna odierna; mentre l'ultimo pulsante serve per cambiare città, una volta premuto viene aperta la GUI per la scelta delle città (Figura A.3).

Travel Mode

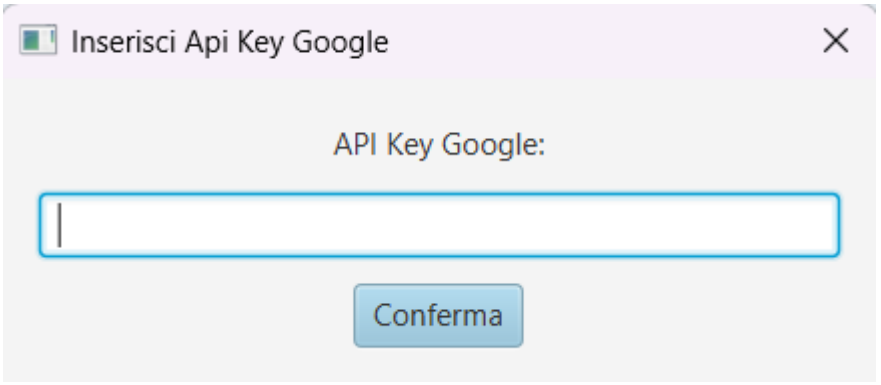

Figura A.6: GUI per inserire chiave API Google

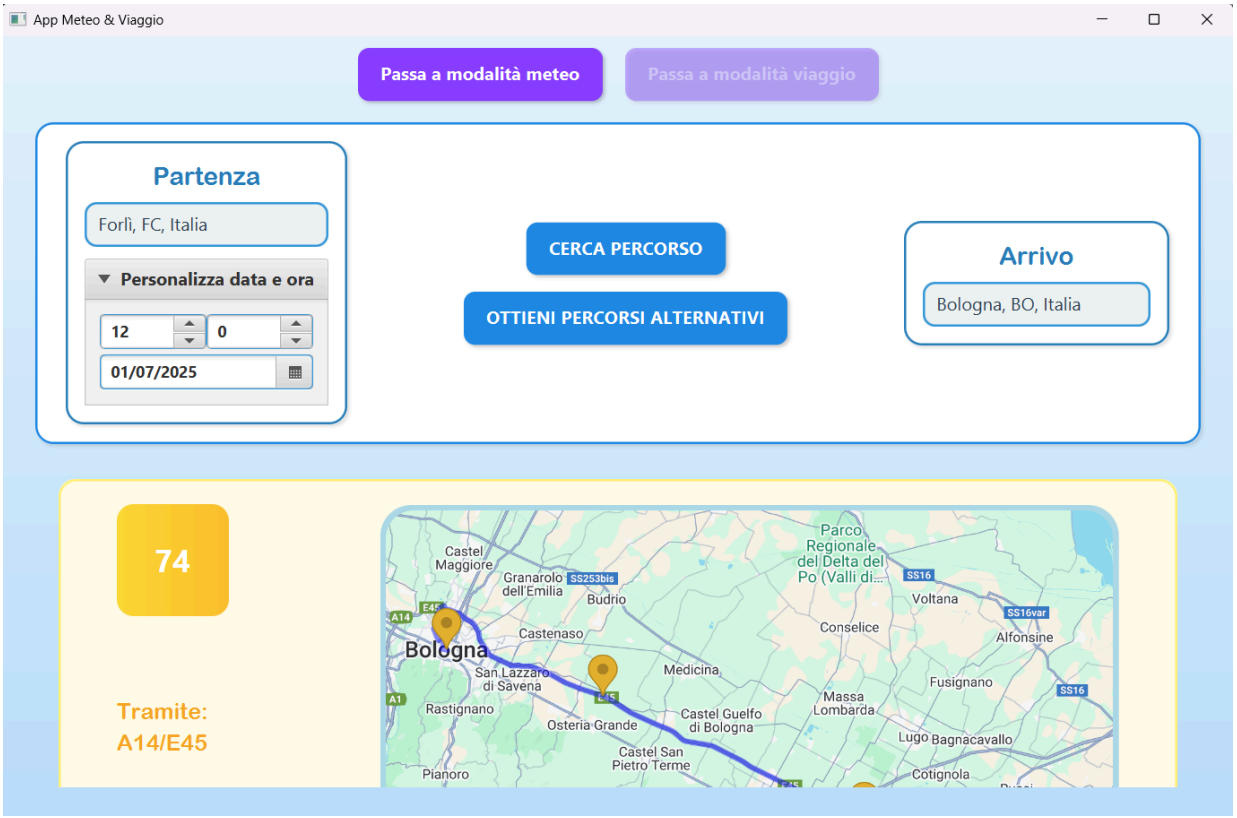

Figura A.7: Schermata della Travel Mode

All'apertura della modalità Viaggio, nel caso non fosse presente la chiave API, necessaria per i servizi Google, all'interno del file di configurazione , viene aperta la GUI per inserire la suddetta chiave (Figura A.6).

Nella modalità Viaggio (Figura A.7) è possibile ottenere uno o più percorsi da percorrere in auto corredati con informazioni meteorologiche. Nella parte dei risultati (zona centrale), viene inserito un box per ogni percorso analizzato. All'interno di ogni box è presente, in alto a sinistra, un punteggio (0-100) che rappresenta una stima generale delle condizioni meteo dell'intero tragitto. Più il voto è alto, più le condizioni stimate sono migliori.

In questa modalità, bisogna inserire una città di partenza ed una di arrivo. È possibile anche specificare una data e ora di partenza; nel caso non venga fatto, vengono usate data e ora attuali. Successivamente premendo il pulsante "Cerca Percorso" viene fornito il percorso più breve con i relativi dati meteo. Premendo il pulsante "Ottieni Percorsi Alternativi" si ottengono eventuali percorsi alternativi che l'utente potrà analizzare.