Relazione per il progetto di "Programmazione ad Oggetti": City Life Simulation

Marco Ravaioli, Rubboli Petroselli Gabriele, Sajmir Buzi, Alessio Bifulco

9 giugno 2024

Indice

NOTA INIZIALE:

In seguito agli avvenimenti comunicati anche via email nella data odierna 9 giugno 2024, le parti di logica del collega Alessio Bifulco, in seguito al suo abbandono del progetto avvenuto 2 giorni oltre la consegna stabilita (ritardata per permettergli di rimediare alla compromissione di tutte le nostre logiche a 1 giorni dalla consegna) sono da considerare parziali e inattendibili. In particolare la logica di licenziamento e assunzione delle persone, che non era nemmeno considerata obbligatoria nella proposta.
La maggior parte delle sue logiche sono comunque state mantenute nell'applicazione perché la nostra volontà non è quella di comprometterlo e speriamo che in data di discussione abbia un ripensamento, e che la sua valutazione sia sufficiente.
Ci dispiace davvero per l'accaduto.

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo propone un modello agent-based per l'osservazione degli spostamenti delle persone per motivi lavorativi in una città.

Requisiti funzionali

- Il simulatore dovrà occuparsi di mostrare lo spostamento delle persone in tempo reale durante un certo intervallo di tempo, con un limite massimo di un anno.
- Saranno presenti diversi grafici rappresentanti l'andamento del flusso dinamico dei vari elementi.
- L'applicazione permetterà all'utente di configurare inizialmente i parametri principali della simulazione, quali ad esempio il numero di abitanti in percentuale.
- Le zone sulla mappa saranno interattive e una volta selezionate, verranno mostrati vari dati relativi a quella zona e al periodo di tempo in corso.

Requisiti non funzionali

- City Life Simulation si pone l'obiettivo di utilizzare al meglio tutte le risorse che adopera, e di risultare efficiente e ben strutturato, in modo da offrire buone performance.
- Il Sistema dovrà essere portabile e adoperabile su differenti sistemi.

- L'Interfaccia dovrà essere ridimensionabile.

1.2 Analisi e modello del dominio

City Life Simulation dovrà essere in grado di simulare il normale svolgimento della vita in una città. La città avrà un determinato numero di abitanti, sarà divisa in zone, collegate da linee di trasporto e ogni zona comprenderà un certo numero di aziende. Dovremo poter visualizzare graficamente lo spostamento delle persone da una zona a un altra per poter recarsi nell'azienda dove lavorano e poi per poter tornare a casa. Le linee di trasporto dovranno mostrare graficamente il loro livello di congestione e ciò influirà sulla durata del viaggio da una zona a un altra. Ogni azienda avrà determinati orari di lavoro e ogni lavoratore avrà il suo turno di lavoro. Gli elementi menzionati sono visibili in Figura 2.1.

Il requisito non funzionale riguardante le performance si tradurrà in un'opportuna fluidità dell'applicazione e nell'evitare blocchi o problemi di rallentamento.

1.2.1 inizio della simulazione

All'inizio della simulazione all'utente verrà fornita la pagina di benvenuto con i pulsanti relativi di START , EXIT e la legenda di spiegazione della simulazione. L'utente alla pressione dello start si presenterà la finestra principale dove sarà possibile settare tutti i parametri necessari al funzionamento della simulazione.

1.2.2 Agenti

Gli agenti coinvolti nella simulazione sono le persone e le aziende della città. Le persone saranno raffigurate da un pallino colorato, rosso se stanno lavorando, mentre blu se sono a casa. Le aziende invece sono invece raffigurate da un quadratino marrone.

1.2.3 Comportamento degli Agenti

I comportamenti delle persone e delle aziende sono i seguenti: Per le persone in base allo scorrere del tempo si sposteranno da zona a zona per motivi lavorativi utilizzando le linee. Mentre per le aziende il loro comportamento è di assumere e licenziare le persone.

1.2.4 Ambiente

L'ambiente sarà rappresentato da una mappa stilizzata di una città presa dal un sito web, che sarà suddivisa in varie zone delimitate in modo predefinito. Durante la simulazione saranno presenti informazioni relative ad ogni zona.

1.2.5 Linee di Trasporto

La città sarà collegata da percorsi che noi chiameremo per semplicità linee, dunque le linee sono dei percorsi che sono sempre disponibili in qualsiasi ora della simulazione. La congestione del percorso dipende dall'afflusso di persone nelle varie linee, e comporterà eventuali ritardi delle persone.

1.2.6 Clock

Tutti i comportamenti degli agenti saranno coordinati dallo scorrere di un clock ad-hoc,con possibilità di mettere in pausa e cambiare tra varie velocità predefinite.

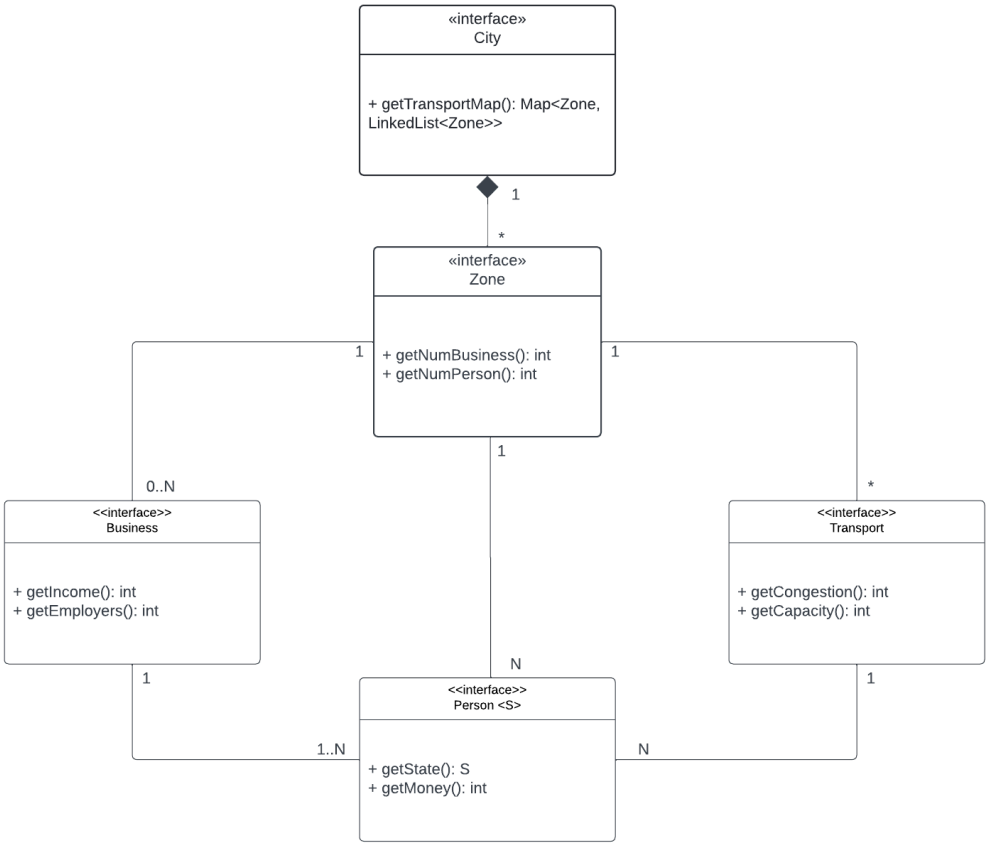

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

In questa sezione approfondiremo la struttura ad alto livello della applicazione.

2.1 Architettura

2.1.1 Model entry point

La struttura di questo progetto usa il pattern MVC , questa divisione ha come entry point la classe *CityModelImpl* che implementa *CityModel* Da questa classe abbiamo accesso agli altri componenti principali del model:

- MapModel : gestisce la logica della mappa.
- clockModel : gestisce la logica dello scorrimento tempo personalizzato.
- Zone : classe che gestice la logica di movimento delle persone
- TransportLine : classe che gestice la logica delle linee.
- DynamicPerson : classe che gestice la logica di movimento delle persone.
- Business : classe che gestice la logica dei comportamenti delle aziende

Si vuole sottolineare che non tutte le classi del model sono state menzionate. Queste sono quelle che si riflettono graficamente sulla view.

View entry point

La view è suddivisa in diverse sottoparti cosi come il model e il controller che vedremo poi successivamente. Esse vengono istanziate nella classe entry point WindowView che si occupa della iniziallizazione grafica della simulazione. Le sottoparti della view sono le seguenti:

- MapPanel : visualizzazione della mappa scelta e degli eventi che la coinvolgono.
- ClockPanel : visualizzazione dello scorrere del tempo e dei pulsanti per modificarne il comportamento.
- InputPanel : pannello contenente i valori di input modificabili.
- InfoPanel : visualizzazione dei dati reletivi alla zona cliccata
- GraphicsPanel : visualizzazione dei grafici relativi ai seguenti oggetti:
  - Business
  - Person
  - TransportLines

Controller entry point

La classe in cui vengono creati tutti i controller subordinati è *WindowController*. Da WindowController è possibile accedere a tutte le sottoparti del controller:

- MapController : collega la view con il model gestendo gli aggiornamenti grafici della mappa.
- ClockController : collega la view con il model gestendo lo scorrimento temporale e le modifiche ad esso apportate.
- InputController : collega la view con il model gestendo i settaggi iniziali scelti dal utente.
- InfoController : collega la view con il model gestendo l'aggiornamento dei dati della zona della mappa cliccata.

- GraphicsController : collega la view con il model gestendo gli aggiornamenti dei grafici.

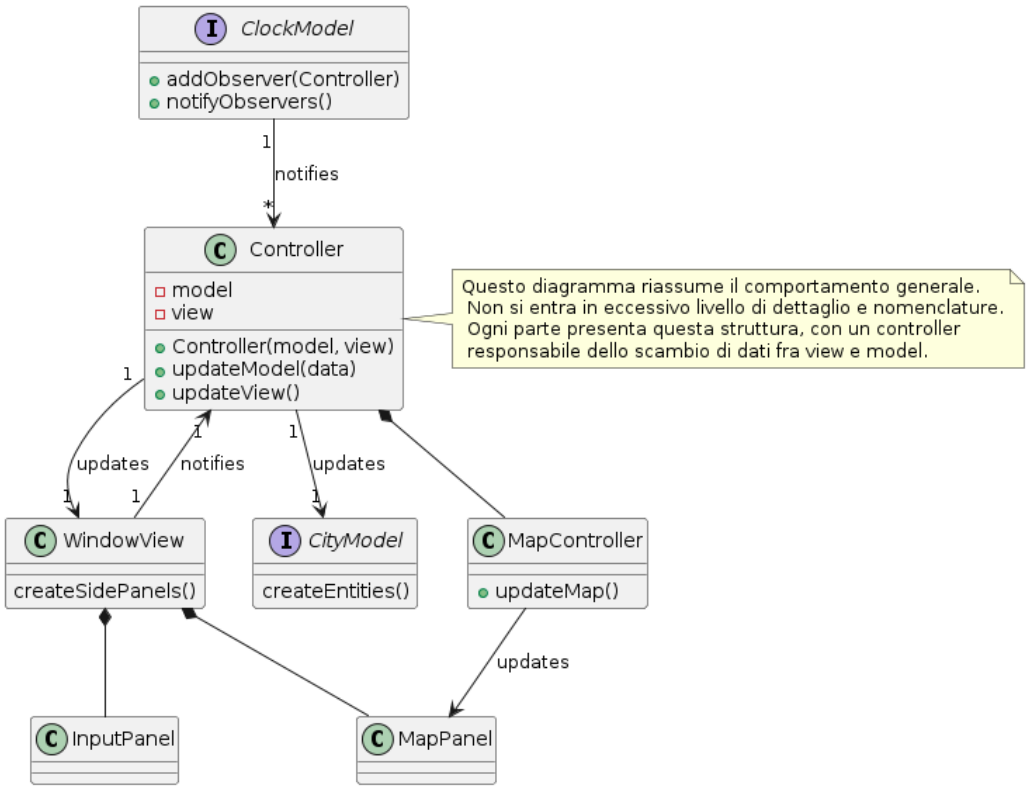

Figura 2.1: Schema UML del design MVC

2.2 Design dettagliato

2.2.1 Sajmir Buzi

La mia parte del progetto consisteva nella gestione dei trasporti ovvero la sua creazione e la sua visualizzazione nella mappa la parte di input della parte della view e la parte di info sempre riguardante la view. Attraverso la parte di input possiamo regolare attraverso specifici slider il numero di persone, la congestione desiderata e il numero di business presenti nella simulazione utilizzando il pattern MVC che magari vedremo in seguito, nella parte di info vediamo delle informazioni riguardanti la mappa come ad esempio il numero di persone nella zona, il numero di linee di dirette la paga media delle persone in una zona ecc.., quindi ogni volta che faremo un click sulla mappa vedremo

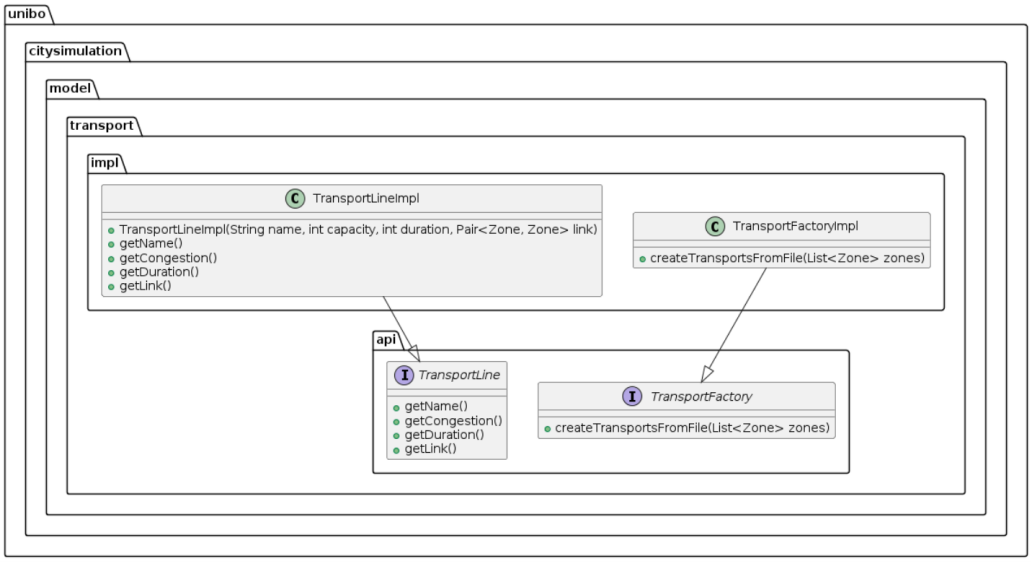

Figura 2.2: Transport UML

i dati aggiornati sempre nel panello di info, tutto ciò deve essere collegato con il tempo(clock) che scorre nella simulazione è ciò è stato abbastanza difficile da gestire.
Andiamo a vedere adesso gli aspetti più difficili scontrati

Gestione Trasporti

Nella parte di gestione Trasporti il problema iniziale che è stato il fulcro del problema era la gestione della congestione per le varie linee, non essendoci una formula specifica globale per la congestione stradale ho pensato che il calcolo della congestione dipendesse dal numero di persone nella linea fratto la capacità:

$$personInLine * 100/capacity;$$

Un altra criticità riguarda le linee di trasporto stesse, insieme anche agli altri compagni di progetto confrontandoci abbiamo pensato di immaginare le linee come dei percorsi stradali e non come linee di trasporto vere e proprie quindi (linee auto, linee bus, linee tram ecc..), ciò ci ha anche aiutato per non dover gestire diverse congestioni per tutte le linee di trasporto che avrebbe inciso molto nel costo computazionale finale , quindi alla fine ci sono vari percorsi che chiamate transportLine che sono sempre attraversabili durante tutto l'arco temporale della simulazione. figura in seguito:

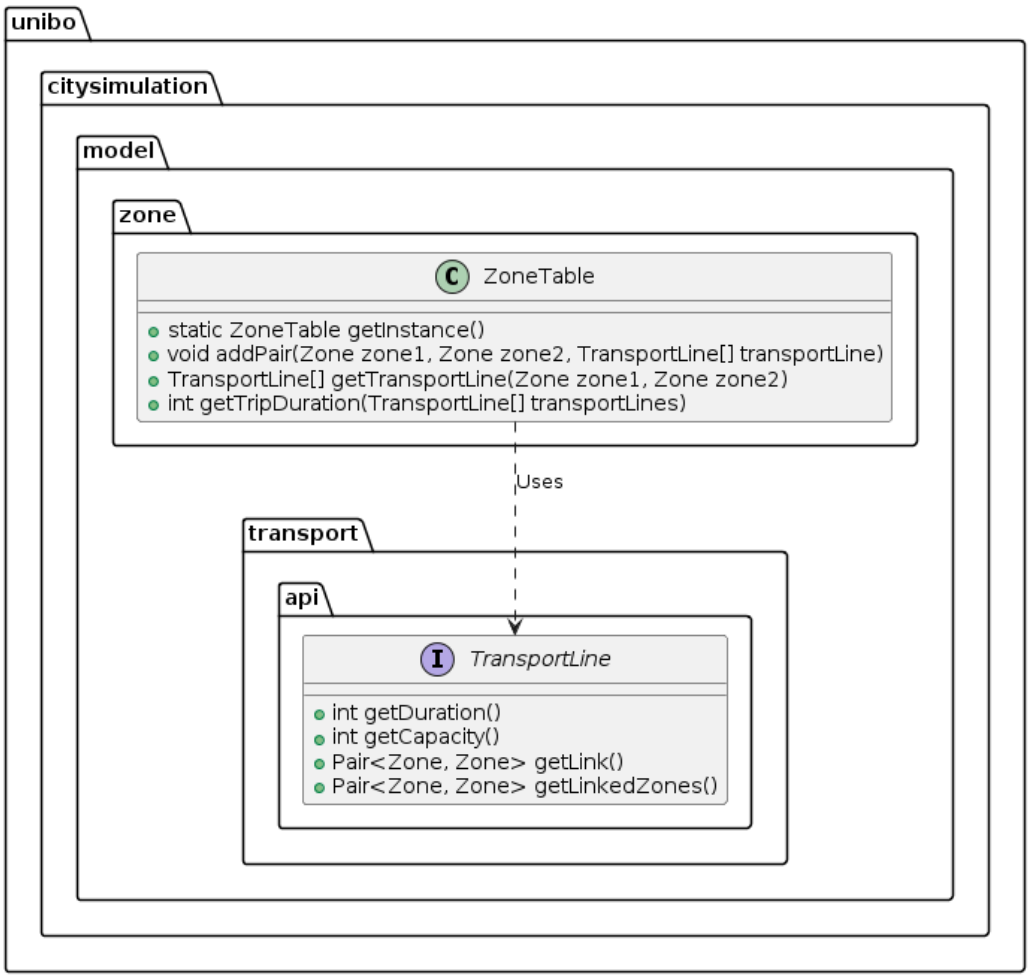

Figura 2.3: ZoneTable UML

Aggiunta trasporti a ZoneTable Map

La classe ZoneTable rappresenta una tabella che mappa coppie di zone (Zone) con le linee di trasporto (TransportLine) che le collegano. Il metodo principale che interagisce con le TransportLine è addPair, che aggiunge una coppia di zone e le linee di trasporto corrispondenti alla tabella. Inoltre, il metodo getTransportLine recupera le linee di trasporto tra due zone, mentre getTripDuration calcola la durata complessiva di un viaggio basato sulle linee di trasporto fornite.

I problemi affrontati durante questa parte si sono rivelati la bidirezionalità e il calcolo della durata del viaggio.

Per quanto riguarda la bidirezionalità, il metodo addPair aggiunge le coppie di zone in entrambe le direzioni (da zone1 a zone2 e viceversa). Questo garantisce che le query per ottenere le linee di trasporto tra due zone siano consistenti indipendentemente dall'ordine delle zone, migliorando l'usabilità e la flessibilità del sistema. Soluzione grezza ma efficace.

Per quanto riguarda il calcolo della durata, ho optato per gli stream. Il metodo getTripDuration() utilizza le funzionalità di stream di Java per calcolare la durata complessiva del viaggio. Questo approccio è conciso e sfrutta le operazioni funzionali per ottenere una soluzione elegante e facilmente manutenibile.

Aggiunta dei trasporti alla ZoneCreation

Una delle principali criticità riguarda la gestione degli errori durante la creazione e l'utilizzo delle TransportLine. È necessario implementare meccanismi robusti per gestire eventuali errori, come la mancanza di dati, errori di input o errori di sistema, al fine di garantire un comportamento affidabile del sistema in situazioni anomale.

È essenziale prevenire e gestire eventuali conflitti o sovrapposizioni nei dati delle TransportLine. Questo potrebbe includere controlli per garantire che non ci siano più linee con lo stesso identificativo o che non siano presenti inconsistenze nei dati delle linee di trasporto.

Dopo aver identificato le principali criticità riguardanti le TransportLine, sono state adottate diverse soluzioni e strategie per affrontarle efficacemente.

Per garantire una gestione robusta degli errori, sono stati implementati controlli di validità e gestione delle eccezioni. Ogni volta che viene creato o utilizzato un oggetto TransportLine, il codice verifica la presenza dei dati necessari e la loro correttezza. Inoltre, vengono catturate e gestite le eccezioni potenziali, assicurando che il sistema possa reagire adeguatamente a situazioni anomale senza interrompere il suo funzionamento.

Inoltre, per prevenire conflitti di dati, è stato adottato un meccanismo di controllo che garantisce l'unicità degli identificativi delle TransportLine. Prima di aggiungere una nuova linea di trasporto, il sistema verifica che non esistano altre linee con lo stesso identificativo, evitando così sovrapposizioni e incoerenze.

Creazione dei Trasporti

La classe TransportFactoryImpl è una fabbrica (factory) responsabile della creazione di oggetti TransportLine basati su una lista di oggetti Zone. Utilizza il formato JSON per configurare le linee di trasporto, leggendo i dati da un file esterno e convertendoli in oggetti TransportLine. La classe utilizza la libreria Gson per il parsing del JSON e gestisce eccezioni come file non trovati e errori di parsing. Problematiche scontrate:

- Dipendenza da File Esterno: La dipendenza da un file esterno (TransportInfo.json) posizionato in una directory specifica potrebbe causare problemi di portabilità e distribuzione dell'applicazione. Eventuali errori di percorso o assenza del file possono causare il fallimento del processo di creazione delle TransportLine.

-

- Accoppiamento tra Zone e TransportLine: Il codice presume che gli indici delle zone nel file JSON corrispondano direttamente agli indici nella lista zones passata come parametro. Questa dipendenza potrebbe portare a inconsistenze se le zone nel file JSON non sono sincronizzate con quelle nella lista.

Le soluzione attuate sono le seguenti:

- Utilizzo della Libreria Gson: La libreria Gson è utilizzata per il parsing del JSON. Questo semplifica la conversione dei dati JSON in oggetti Java, rendendo il codice più manutenibile. Tuttavia, l'assenza di validazioni aggiuntive sul contenuto del JSON può rappresentare un rischio in termini di dati non conformi.

- Inizializzazione di TransportLine con Coppie di Zone: La classe utilizza la struttura Pair<Zone, Zone> per rappresentare le coppie di zone collegate da una TransportLine. Questo approccio è concettualmente chiaro e semplifica l'associazione tra le zone e le linee di trasporto.

parte della view : INPUT

Questa fase effettuata è stata una delle più difficili ma anche stimolanti del progetto, ovvero unire la parte grafica alla parte di modellazione. Utilizzando il Pattern MCV il lavoro della parte di di input è stata divisa ovviamente in Controller, Model e View. Attraverso l'uso del diagramma UML proviamo ad entrare più in merito del design effettuato.

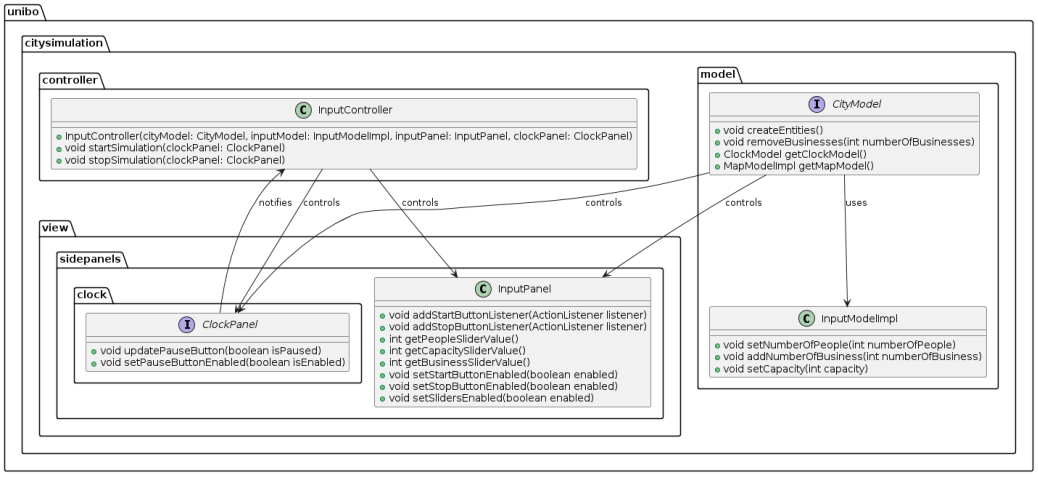

Figura 2.4: MVC-Input

Andiamo adesso a spiegare meglio la struttura del Model Controller e view andando a fare una breve spiegazione ciascuno , elencando gli aspetti positivi o negativi e le principali problematiche affrontate durante la fase di design e le soluzioni apportate.

Parte di model:
Il modello è responsabile della gestione dei dati e della logica dell'applicazione. La classe `InputModel` rappresenta il modello di input per la simulazione della città, contenente informazioni come il numero di persone, il numero di attività commerciali, la capacità e la ricchezza. La classe `CityModel` gestisce la logica della città stessa, inclusa la creazione e la gestione delle entità cittadine.
Come aspetti positivi utilizzando questa soluzione di programmazione abbiamo una centralizzazione della logica, facile gestione e aggiornamento dei dati.
Problematiche Riscontrate: La necessità di scalare i valori degli input, ad esempio il numero di persone, per adattarli ai limiti del sistema.Soluzioni Adottate: Implementazione di controlli e validazioni nei metodi setter del modello (`setNumberOfPeople`) per garantire l'integrità dei dati.

Parte di view:
La vista è responsabile della presentazione dei dati all'utente e della raccolta dell'input dell'utente.
La classe `InputPanel` rappresenta il pannello di input che include slider per il numero di persone, attività commerciali, capacità e pulsanti per avviare e fermare la simulazione.
La classe `ClockPanel` gestisce la visualizzazione dell'orologio della simulazione, quest'ultima è fondamentale per sincronizzare tutte le entità e tutti gli aggiornamenti vari effettuati nella simulazione.
La principale problematica è stata la gestione dello stato dei pulsanti (abilitato/disabilitato) in base allo stato della simulazione.
Come soluzione ho pensato di utilizzare dei metodi dedicati (dei setter) , (`setStartButtonEnabled`, `setStopButtonEnabled`, `setSlidersEnabled`) per gestire l'abilitazione e la disabilitazione dei componenti in modo dinamico.

Parte Controller:
Il controller gestisce l'interazione tra il modello e la vista. La classe `InputController` gestisce gli eventi generati dall'input dell'utente nell'`InputPanel`, aggiornando il `CityModel` e l'`InputModel` di conseguenza. Aspetti Positivi: Centralizzazione della logica di controllo degli eventi, migliorando la separazione delle

responsabilità,questo è proprio l'obiettivo del pattern MVC. Le problematiche riscontrate sono state la necessità di sincronizzare lo stato della simulazione tra diversi componenti dell'applicazione. Come soluzione ho optato per l'implementazione di listener per i pulsanti di avvio e arresto, assicurando che la logica di avvio/arresto della simulazione

Parte della view: INFO

anche in questo caso usiamo il pattern MVC per coordinare al meglio la logica tra model view e controller. In questa fase andremo a visualizzare le principali informazioni che saranno utili all'utente per capire al meglio la simulazione attraverso click dell'utente sulla mappa interattiva. Anche in questo caso andremo a visualizzare il diagramma UML per riuscire a comprendere al meglio la struttura.

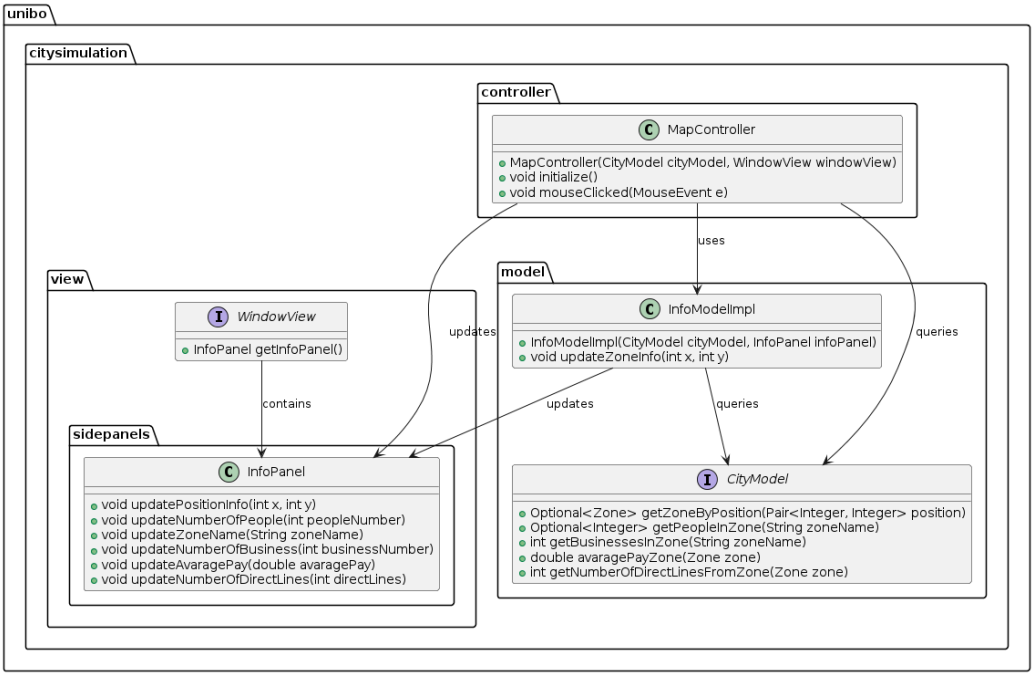

Figura 2.5: relazioni con InfoPanel

Per quanto riguarda la mia parte posso parlare delle relazione che ho effettuato tra la InfoPanel e le altri classi nella quale richiamo i metodi della

infoPanel.
Nella parte di infoPanel quindi abbiamo solamente la visuale del panello , non ci sono state particolari difficoltà da questo punto di vista, ciò che ho fatto e crearmi della JLabel ed ogni volta chiamare la Update di ciascuna JLabel quando faccio un click nella Mappa e tutto ciò che ho appena detto è un pregio perché ci si occupa solo di visualizzare le informazioni sulla UI.
Per quanto riguarda il click come dicevo prima si farà riferimento al controller che è il punto focale per quanto riguarda la cattura delle informazioni sul panello laterale.
Dipende da `CityModel` per ottenere informazioni sulle zone, sulle persone, sulle imprese e sulle linee dirette. Dipende da `MapModelImpl` per ottenere informazioni sulla mappa. Dipende da `InfoPanel` per aggiornare le informazioni visualizzate.
Come aspetto positivo possiamo notare la gestione delle azioni utente: `MapController` gestisce gli eventi di clic dell'utente sulla mappa e anche la separazione delle responsabilità, `MapController` si occupa di aggiornare sia la mappa che il pannello delle informazioni.
Per quanto riguarda un aspetto negativo potrebbe essere la dipendenza tra varie classi e ciò rende il codice più complesso del previsto.

2.2.2 Gabriele Rubboli Petroselli

Il mio compito consisteva nella creazione di un oggetto rappresentativo delle persone nella simulazione, con relativa distribuzione della mappa e relativo aggiornamento di stato durante lo scorrere del tempo.
Essendo la nostra idea di creare uno scorrimento del tempo personalizzato e ovviamente compresso rispetto alla realtà, si è presto palesata la necessità di creare questo modello prima di poter anche solo pensare alla logica dietro alle persone, poiché volevamo esse fossero create attraverso un parametro di input che ne definisse il numero in circolazione sulla mappa, e quindi a posteriori dell'avvio del tempo della simulazione.

Modello del Clock

Problema: Creare un modello temporale adeguato alle nostre esigenze, modificabile a comando dall'utente, il quale può cambiarne la velocità, metterlo in pausa e farlo riprendere da dove si era interrotto, oppure farlo ripartire dal giorno 1 e ora 00:00.
Soluzione: La soluzione proposta, in linea col pattern MVC adottato la livello di applicazione, è stata quella di creare un modello e una view rappresentante lo scorrimento del tempo e i pulsanti per agire su esso, resi in comunicazione da un Controller funzionante in entrambi i versi.
Entrando nel dettaglio del modello, la soluzione è stata creare un oggetto Timer e modificarne il funzionamento in modo da avere una maggiore flessibilità e adattarlo alle nostre esigenze. La velocità impostabile dall'utente, è il parametro che scandisce la start della simulazione.
Essendo la velocità dell'oggetto Timer interpretabile come la frequenza in millisecondi di ripetizione dei Task, si è scelto di impostare un *updateRate* fisso di base, settabile tramite il metodo *setUpdateRate* contenuto nell'interfaccia *ClockModel* e decrementabile per aumentare la frequenza di aggiornamento del tempo.
La view a sua volta è stata divisa in interfaccia e implementazione per migliorare l'incapsulamento e limitare l'esposizione esterna di oggetti mutabili.

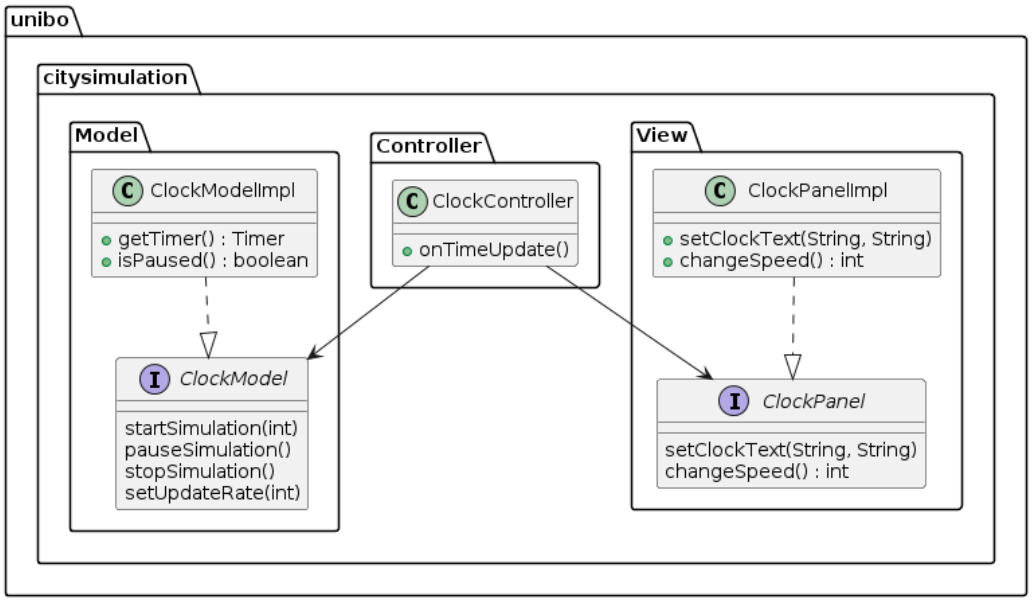

Figura 2.6: Schema UML della logica del Clock

Entità "Persona"

Problema: Progettare correttamente gli aspetti statici dell'entità persona, cercando il più possibile di tenerli separati da quelli dinamici che ne avrebbero permesso il movimento all'interno della mappa.
La persona al momento della sua creazione deve avere un nome, un'età, un eventuale lavoro, una posizione definibile come "casa" e una quantità di soldi iniziale incrementabile col tempo.
Soluzione: Si è scelto di separare gli aspetti immutabili e quelli mutabili o comunque derivabili da altri in due file separati. In *PersonRecord* è stato scelto l'utilizzo di un record per il salvataggio dei dati immutabili, con la conseguente automatizzazione di tutti i "getter" necessari, per evitare un'eccessiva quantità di metodi.
Nell'interfaccia *StaticPerson* e nella sua implementazione *StaticPersonImpl* sono stati incapsulati tutti gli aspetti statici (slegati dalla logica di movimento) derivabili da quelli presenti nel record, ma che comunque necessitavano di un calcolo iniziale e non erano noti al momento della creazione dell'entità, come ad esempio il calcolo del percorso dalla propria zona di residenza alla propria zona di lavoro e la relativa durata del viaggio attraverso le linee di trasporto, o la scelta randomizzata di coordinate rappresentanti il punto esatto di residenza della persona nella sua zona di residenza.
Inoltre, Ho scelto di rappresentare lo stato corrente dell'entità come un enu-

meratore che indichi se la persona è a casa, se sta lavorando o se è in movimento nelle linee di trasporto.

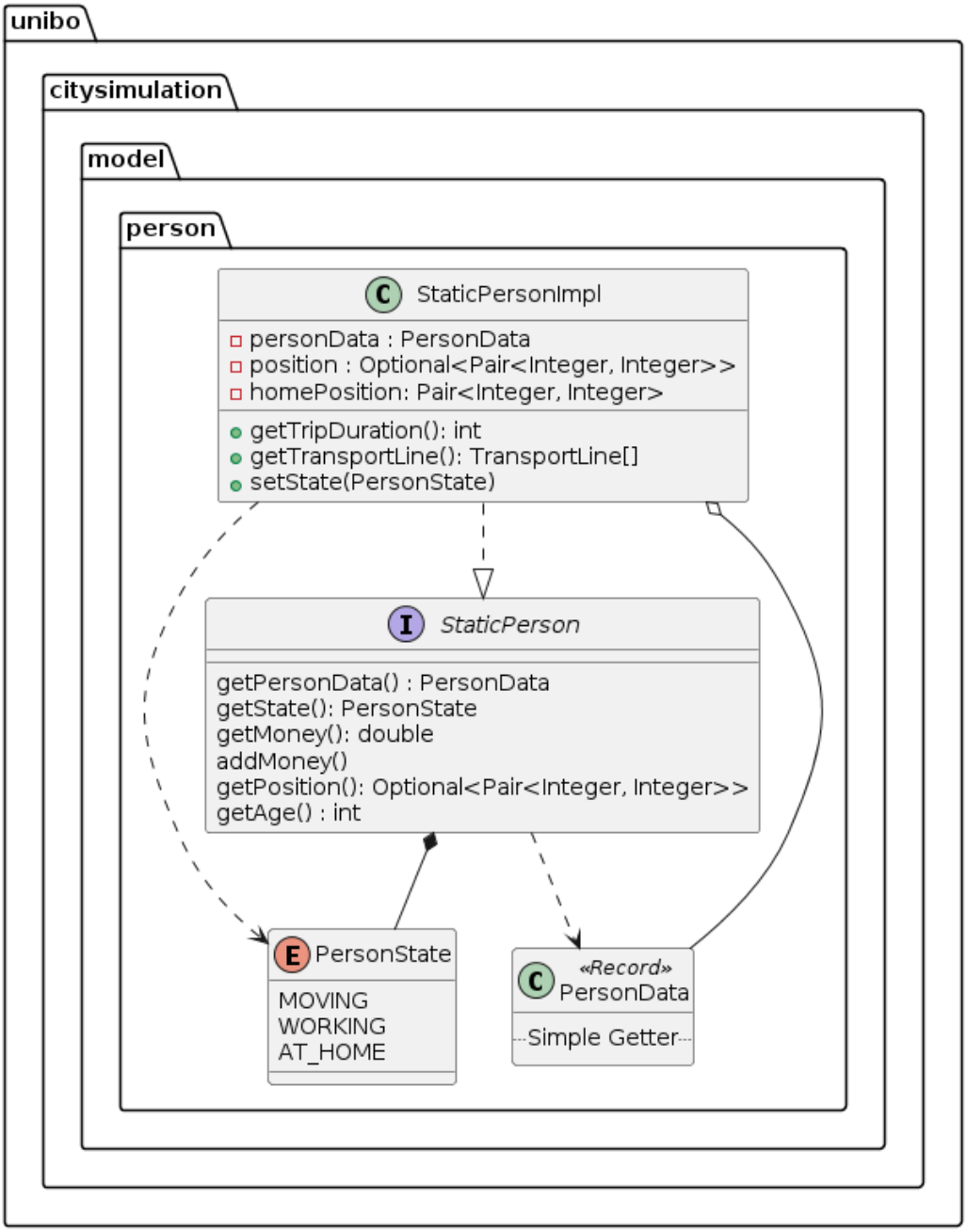

Figura 2.7: Schema UML della gestione degli aspetti statici dell'entità Persona

Aggiornamento dello stato delle persone

Problema: Aggiornare lo stato delle persona in base all'orario corrente; Capire se la persona in quel momento può stare a casa, se è ora che parta per andare a lavorare o se ha finito il turno lavorativo ed è libera di tornare a casa.
Soluzione: Ho scelto di realizzare un'interfaccia *DynamicPerson* estensione dell'interfaccia *StaticPerson* e implementata dalla classe *StaticPersonImpl*, a sua volta estensione di *StaticPersonImpl*, in modo da creare un'entità comprendente tutti gli aspetti statici e immutabili già implementati e avente in aggiunta tutta la logica dinamica di stato e di movimento all'interno della mappa, seguendo la propria routine giornaliera di lavoro.
L'interfaccia presenta un singolo metodo *checkState* che prende in input l'orario corrente e prende una decisione sul comportamento opportuno da adottare per la persona in quel determinato momento, che può essere restare allo stato attuale, muoversi per rispettare gli orari lavorativi o scendere dalle linee di trasporto una volta arrivati a destinazione.
Nell'implementazione, durante la costruzione, vengono salvati gli orari di inizio e fine del turno lavorativo sotto forma di secondi decorati da una lieve componente randomica per ogni persona, e in base alla durata del viaggio da una zona all'altra viene effettuata attraverso vari metodi privati una previsione sugli orari corretti per adottare i diversi comportamenti, in modo da cercare di rispettare gli orari.
In base al comportamento da adottare vengono eventualmente aggiornati lo stato e la posizione della persona.
La posizione è salvata come Opzionale in modo che se si trova sotto stato *MOVING* e quindi sta viaggiando da una zona all'altra, esso viene settato a *empty*.

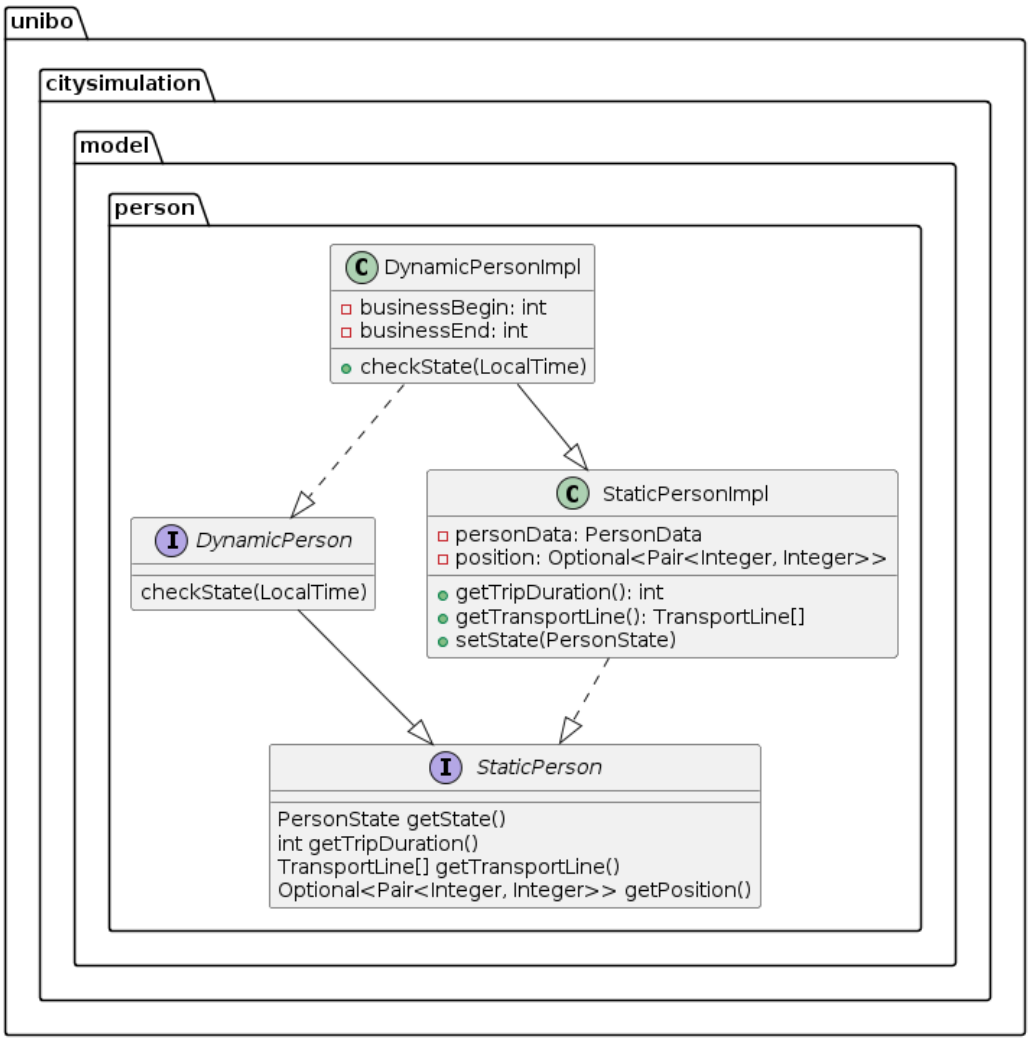

Figura 2.8: Schema UML della gestione degli aspetti dinamici e di aggiornamento di stato e posizione dell'entità Persona

Dipendenza dinamica Clock-Person

Problema: Abbiamo la funzione checkstate che in base all'orario corrente sceglie un comportamento da adottare per la persona e la aggiorna di conseguenza.
La challenge successiva è stata di collegare la chiamata di questa funzione al Clock, in modo che a ogni aggiornamento dell'orario, a frequenza di *updateRate* (già menzionato nella sezione del clock) la funzione venisse chiamata.
Soluzione: La soluzione adottata è stata di applicare il pattern comportamentale *Observer*, in modo da avere una funzione che notificasse le persone dell'aggiornamento del clock con conseguente chiamata alla funzione di aggiornamento di queste ultime, senza dovergli necessariamente passare il modello del Clock.
Ho scelto di percorrere questa strada anche in seguito alla previsione che la notifica di aggiornamento del clock andasse effettuata non solo sulle persona, ma su varie entità della simulazione dipendenti dal tempo corrente, come ad esempio le linee di trasporto che dovevano aggiornare il loro livello di congestione attuale.
La prima mossa è stata quella di rendere il modello del clock il *Subject*, aggiungendogli i relativi metodi come da prassi.
Ho creato poi un'interfaccia *ClockObserver* con un unico metodo chiamato *onTimeUpdate* che prende come input l'orario corrente e il numero del giorno corrente, implementata dal *Concrete Observer ClockObserverPerson*, specializzato per notificare le entità Persona.
Inoltre, ho reso anche *ClockController* un implementazione dell'observer, in modo da aggiornare la view in modo dinamico allo stesso modo delle persone.

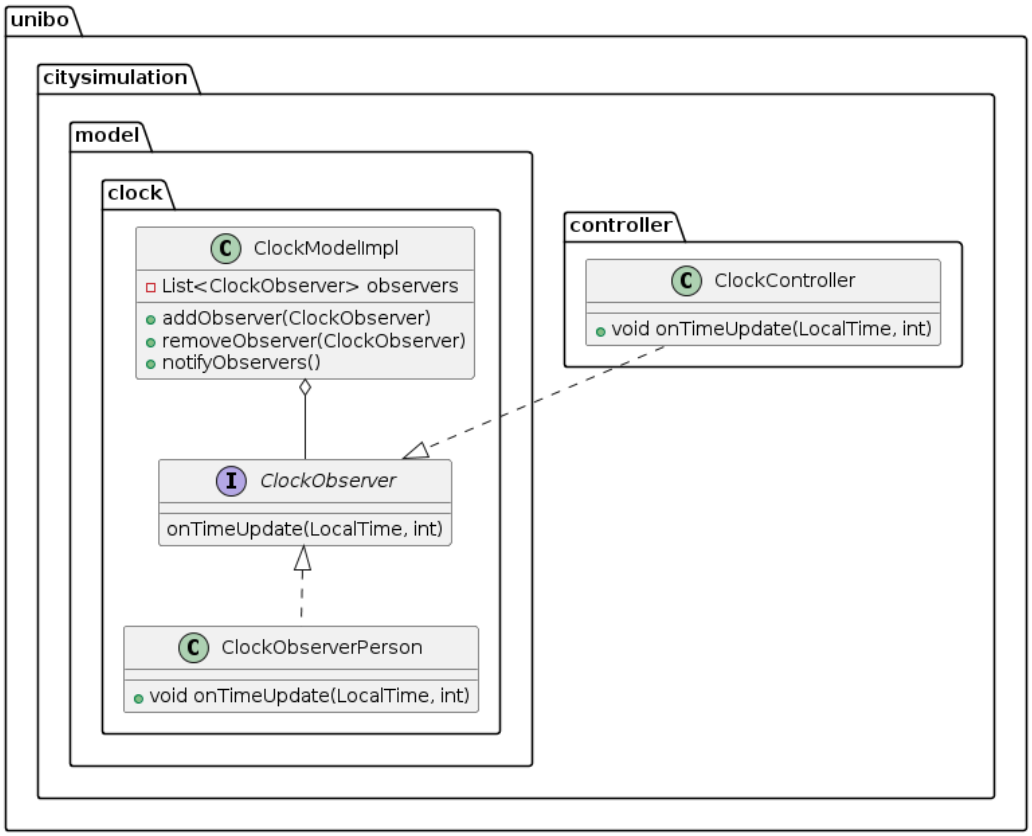

Figura 2.9: Schema UML del pattern Observer utilizzato per la dipendenza dinamica fra clock e persone

Istanziazione delle Persone

Problema: Trovare un modo per una flessibile ed efficace istanziazione di un oggetto persona e di gruppo di persone accomunate dalla zona di residenza al momento dello spawn iniziale. Avere poi un metodo per unire questi ultimi e creare tutte le persone della mappa con il minimo possibile di chiamate a funzioni.
Soluzione: Ho optato per l'utilizzo del pattern creazionale "Factory Method", più concretamente ho creato un interfaccia *PersonFactory*, a sua volta implementata da una classe *PersonFactoryImpl*, con alcuni metodi di creazione differenti.
Il metodo *createPerson* permette la creazione di un singolo oggetto persona e prende come parametri gli attributi immutabili necessari per creare il record e la quantità di soldi iniziale assegnata.
Abbiamo poi un metodo per creare un gruppo di persone accomunate dalla zona di residenza, e fra altri parametri abbiamo un range di possibili quantità di soldi iniziali basato sulla "ricchezza" della zona in questione e il numero di persone da collocare in essa.
Infine, il metodo pubblico *createAllPeople* si occupa dell'istanziazione completa del numero di persone scelte in input per tutte le zone.

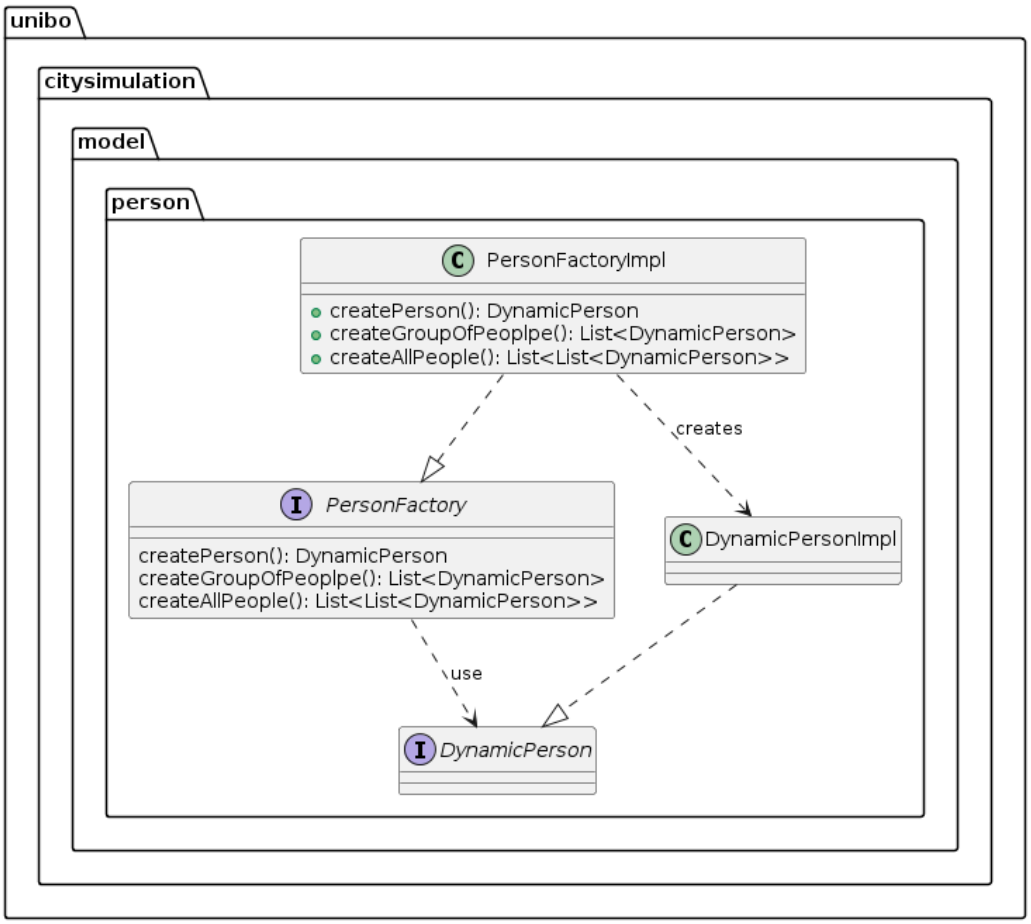

Figura 2.10: Schema UML del pattern Factory Method applicato alle persone

Distribuzione delle persone

Problema: Distribuire le persone in modo sensato a partire da un input dato a livello di view dall'utente.
L'utente prima di far partire la simulazione, selezione attraverso uno slider il numero totale di persone che saranno presenti nella simulazione e a partire da questo numero si devono distribuire le persone sulla mappa secondo una logica.
Soluzione: Tenendo conto della progettazione delle zone della mappa realizzata dai miei colleghi, ho scelto di inserire un campo *personPercent* nel record *Zone*, a sua volta letto da un file JSON dove sono salvate le informazioni su tutte le zone, che rappresenta la percentuale fissa di persone residenti in quella zona sul numero di persone totali scelto in input dall'utente.
Utilizzando questo numero, le persone vengono distribuite in modo premeditato e sensato all'interno delle varie zone della mappa e attraverso il metodo *getRandomPosition* sempre collocato nel record, alla creazione delle persone gli viene assegnata una posizione randomica all'interno della zona di residenza che rappresenterà la loro dimora.

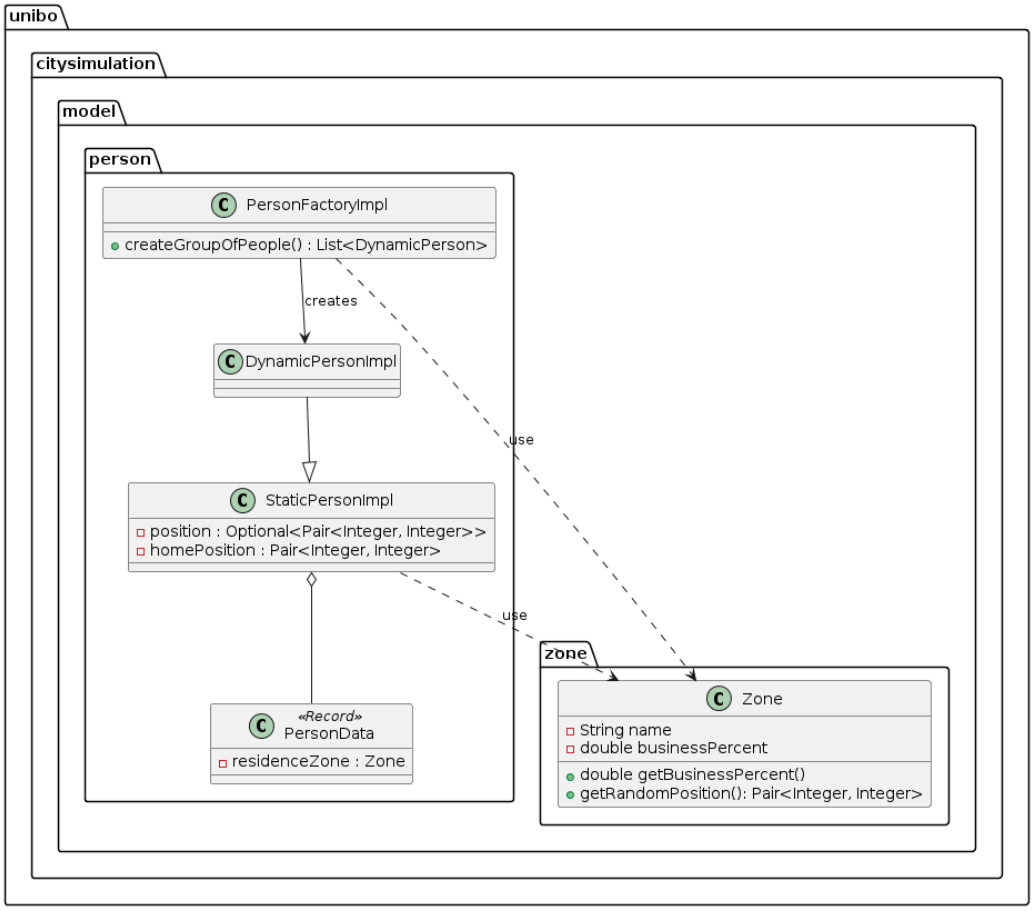

Figura 2.11: Schema UML della logica secondo cui vengono distribuite le persone

2.2.3 Marco Ravaioli

Nel rispetto della suddivisione del lavoro, ho progettato e sviluppato tre sezioni principali dell'applicazione: la logica di gestione della mappa, la sezione dei grafici e l'interazione tra persone e trasporti. Le prime due sezioni seguono una struttura simile: i controller inviano periodicamente informazioni ai model che elaborano i dati grazie a classi di supporto. Le informazioni elaborate vengono poi inserite nella view tramite il controller. Per quanto riguarda l'interazione persone-trasporti, ho lavorato trasversalmente utilizzando le interfacce fornite dai colleghi e i metodi esposti.

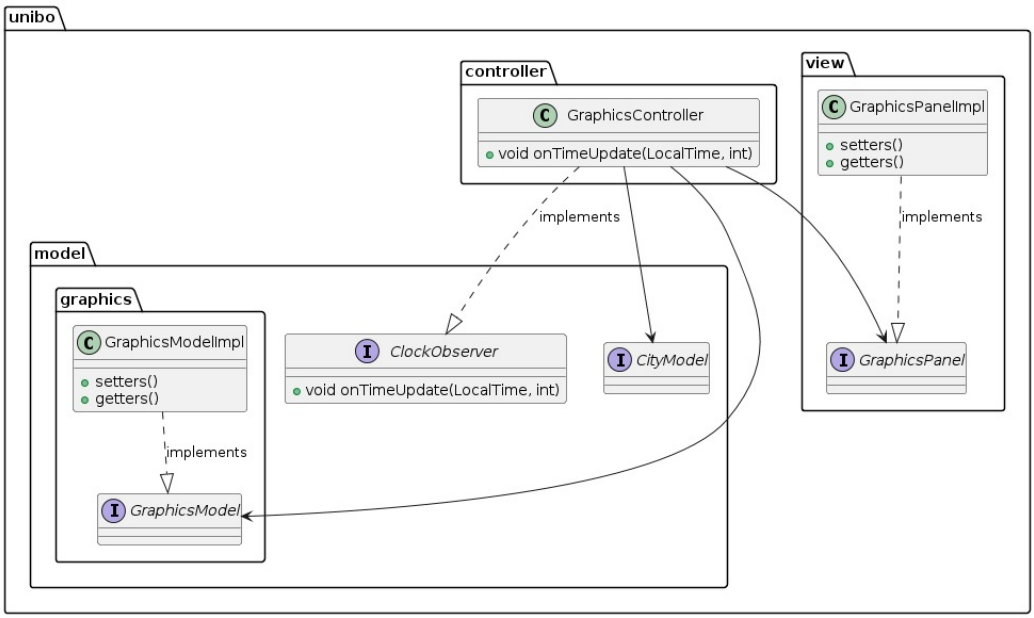

Figura 2.12: UML rappresentante il pattern MVC applicato alla parte dei grafici con il controller che implementa l'Observer

Aggiornamento dei dati

Problema Nell'applicazione, i dati relativi a persone, aziende e trasporti vengono aggiornati costantemente nei rispettivi model. È necessario un pattern che gestisca in modo efficiente l'aggiornamento di questi dati all'interno dei model delle mappe e dei grafici, garantendo che le informazioni visualizzate siano sempre coerenti e aggiornate.

Soluzione Per affrontare questo problema, sono stati creati controller che implementano l'interfaccia ClockObserver. Questa interfaccia include un metodo *onTimeUpdate* che viene chiamato ad ogni variazione del clock. Ogni volta che il clock si aggiorna, i controller recuperano le informazioni dalle entità in *CityModel*. Questi dati vengono elaborati all'interno dei model, dove vengono raffinati e preparati per la visualizzazione. Infine, i dati estratti vengono passati alla view, che aggiorna il suo aspetto di conseguenza.

L'uso del Observer Pattern è stato cruciale in questa implementazione. I controller agiscono come osservatori del clock e aggiornano i model e le view in base ai cambiamenti dello stato del clock. Questo pattern è stato scelto per separare chiaramente la logica di aggiornamento dal clock e per facilitare l'estensibilità del sistema. Inoltre, l'utilizzo del pattern MVC permette una

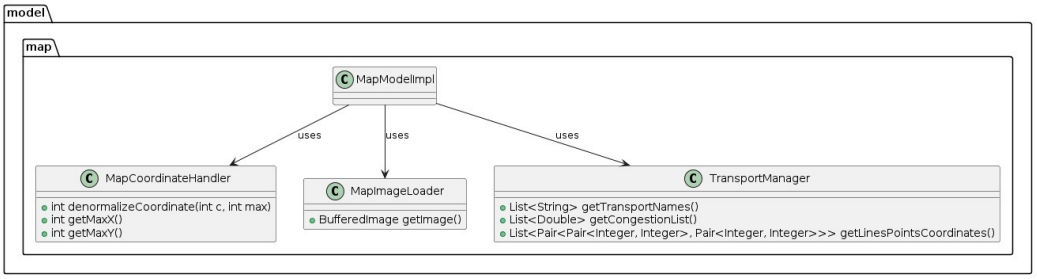

Figura 2.13: UML della parte della mappa in cui le responsabilità vengono divise su più classi

corretta suddivisione delle responsabilità separando aspetti di view da quelli di logica e da quelli di ricezione di segnali.

La logica di gestione delle informazioni

Problema Diversi aspetti distinti devono essere gestiti nella parte della mappa, tra cui la gestione degli eventi di click del mouse con il conseguente aggiornamento della visualizzazione dei dati, il caricamento e la gestione delle immagini che devono essere sempre disponibili e integre, e gli algoritmi per estrarre informazioni rilevanti dai dati per poi passarle alla view. Un problema aggiuntivo è che, durante lo sviluppo, gli oggetti ricevuti in ingresso cambiano frequentemente a causa del refactoring, complicando ulteriormente la gestione del codice.

Soluzione Ho scelto di creare classi di supporto utilizzando un approccio di suddivisione delle responsabilità per mantenere il codice il più invariato possibile durante i cambiamenti. Queste classi di supporto sono utilizzate all'interno del model, riducendo le sue responsabilità dirette e migliorando la leggibilità del codice. Questo approccio ha permesso di modularizzare le diverse funzioni e di rendere il sistema più facile da mantenere e da estendere.

Problema La gestione dei grafici presenta la sfida di dover costruire e mantenere i dataset, oltre a dover raffinare i dati ottenuti tramite il controller per la loro rappresentazione grafica.

Soluzione Ho creato una classe di utilità che fornisce metodi statici per il calcolo dei dati ottenuti dagli elementi da rappresentare, ottenendo così le percentuali da visualizzare a schermo. Inoltre, ho sviluppato una classe specifica per istanziare e gestire i dataset per conto del *GraphicsModel*. Questa

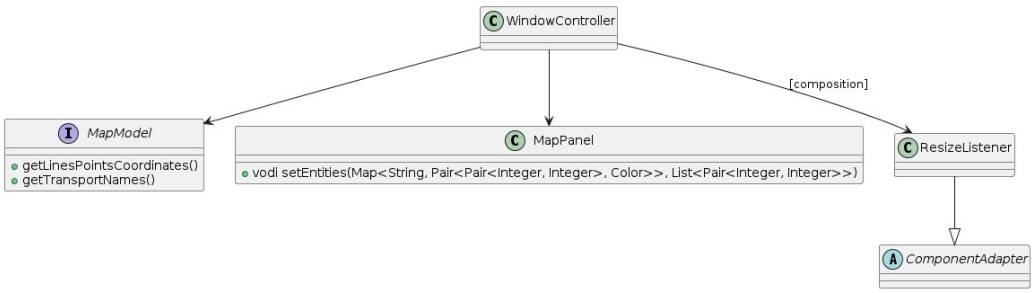

Figura 2.14: Il resize Listener che estende il component adapter aggiorna costantemente i controller

classe funge da intermediario, permettendo di mantenere separata la logica di gestione dei dati dalla logica di visualizzazione.

Visualizzazione della mappa e interazione

Problema La finestra visualizzata è ridimensionabile tramite interazione con il mouse. Il pannello della mappa deve essere ridimensionato di conseguenza e i disegni stampati sopra hanno una posizione data in pixel. Così facendo, certe parti vengono nascoste con il rimpicciolimento della schermata.

Soluzione Adottare un sistema normalizzato. Si ricavano le nuove dimensioni al ridimensionamento della finestra e le figure che vengono stampate ottengono le coordinate denormalizzate per il corretto posizionamento sulla finestra, qualunque sia la sua attuale dimensione.

Problema Alla pressione del mouse all'interno del pannello della mappa serve un aggiornamento di visualizzazione delle informazioni a seconda del punto premuto. Occorre un oggetto che riesca a intercettare le coordinate premute nella view ed elaborarle.

Soluzione Implementare MouseListener al controller in modo da poter intercettare diversi comportamenti del mouse (l'interesse è posto solo sul click però), il controller poi chiama un metodo con cui si ricavano le informazioni dalle zone. Il MouseListener è un'estensione di EventListener che intercetta gli eventi del mouse, indipendentemente dalla libreria grafica usata.

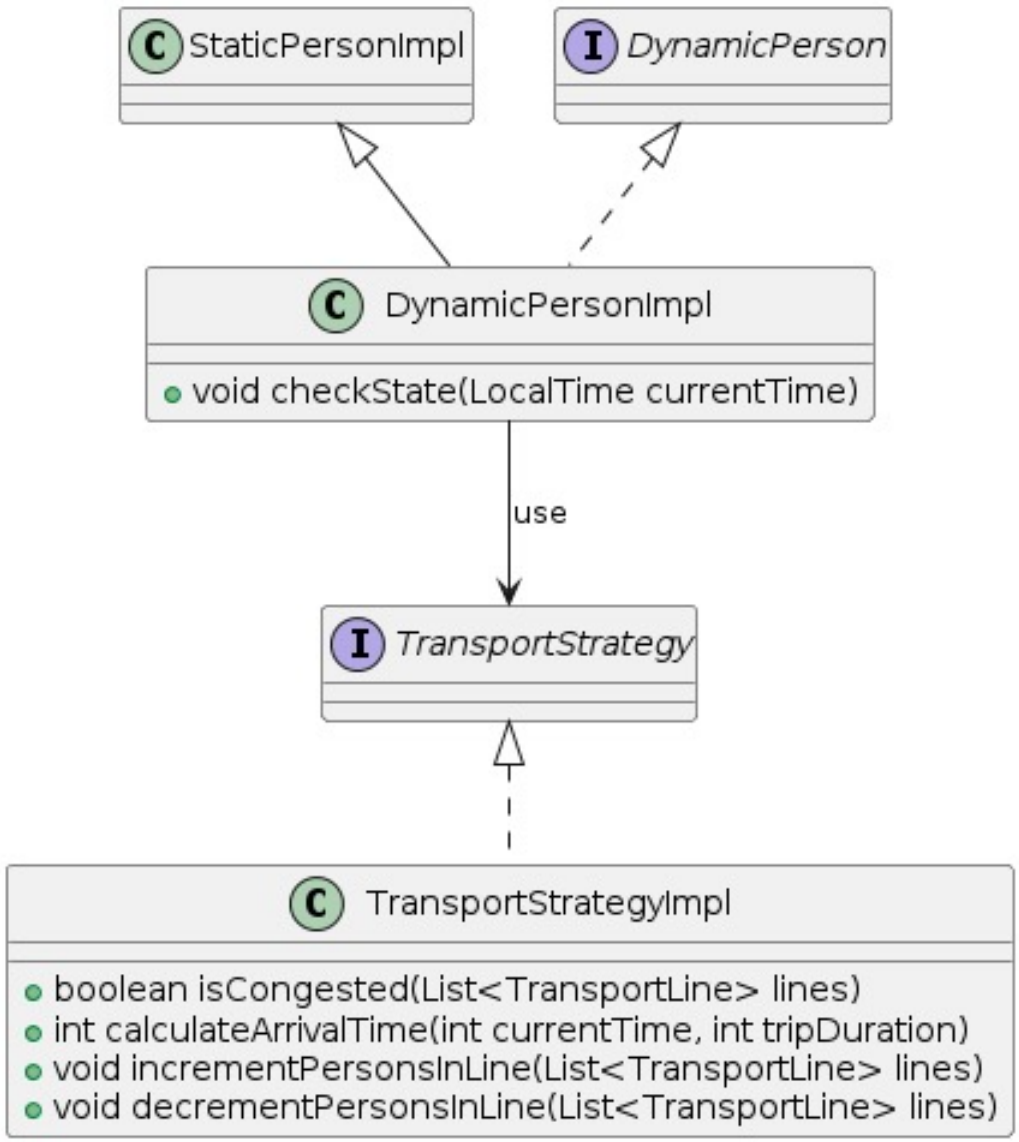

Figura 2.15: Interazione tra DynamicPerson, responsabile del movimento delle persone, e TransportStrategy, che fa i cambiamenti nei trasporti

Gestione dello spostamento delle persone

Problema La gestione dello spostamento delle persone implica l'utilizzo delle linee di trasporto, che hanno una capacità massima e un tempo di percorrenza. Ogni persona deve essere in grado di occupare un posto su ogni linea che percorre per tutta la durata dello spostamento. Inoltre, è necessario gestire la situazione in cui le linee di trasporto sono a capacità massima, mettendo le persone in attesa.

Soluzione Ho creato una classe di supporto dedicata alla gestione delle operazioni con le linee di trasporto. Questa classe implementa la logica di attesa delle persone quando i trasporti sono a capacità massima, assicurando che ogni persona possa occupare un posto disponibile su una linea di trasporto per l'intera durata del viaggio. La classe di supporto si interfaccia direttamente con i modelli delle linee di trasporto e delle persone, aggiornando gli stati e gestendo le code di attesa in modo efficiente.

Extra Un aspetto che avrei voluto implementare è un algoritmo di ricerca dei cammini minimi per ottimizzare i percorsi di spostamento delle persone. Tuttavia, a causa della struttura del programma e del continuo aggiornamento delle informazioni, questo avrebbe richiesto una deviazione significativa dalla metodologia di sviluppo Agile adottata per il progetto. La complessità aggiunta da tale algoritmo avrebbe compromesso la manutenibilità e la flessibilità del sistema, oltre al tempo necessario per implementarlo, pertanto ho deciso di non includerlo.

2.2.4 Alessio Bifulco

La parte di progetto che mi ha riguardato è concentrata nello sviluppo dei **Business**, uno dei due agenti presenti nella simulazione, e di tutto ciò che li riguarda. La creazione e la gestione dei business sono fondamentali per il funzionamento della simulazione della città.

Nel contesto della simulazione, era necessario gestire la creazione di diversi tipi di Business per differenziarli. Ogni tipo di Business ha comportamenti comuni, in un'ottica futura è previsto che i comportamenti attualmente comuni vengano diversificati da ogni tipo di business.

Per affrontare ciò ho utilizzato il pattern *Factory Method* per la creazione delle diverse tipologie di business (BigBusiness, MediumBusiness, SmallBusiness) che vanno tutte a estendere la classe astratta **Business** che contiene

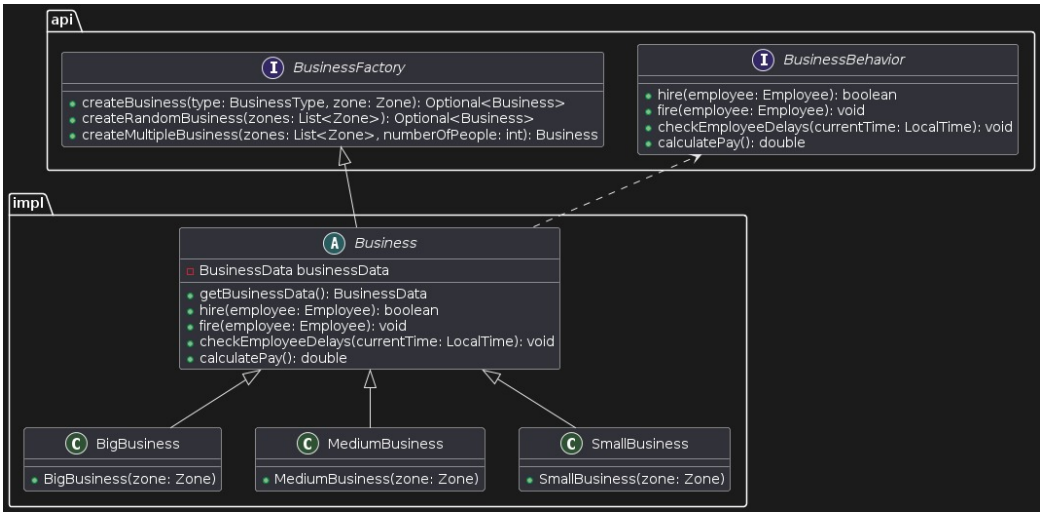

Figura 2.16: Schema UML

i metodi e dati comuni ai tipi di business, mentre l'interfaccia **BusinessFactory** contiene i metodi per la logica di creazione delle istanze di Business sfruttando Optional e uno switch con gli enumerativi dei diversi tipi di business.

Motivazioni: L'uso del pattern Factory Method centralizza la logica di creazione e permette di aggiungere nuove tipologie di business con facilità. L'utilizzo di Optional migliora la gestione delle eccezioni legate a tipi di business non validi. La classe astratta Business permette di condividere metodi comuni tra i vari tipi di business, riducendo la duplicazione del codice e migliorando la manutenibilità, in ottica di estendibilità anche avvantaggiano nel caso si volessero aumentare i tipi di business.
Poiché ogni tipo di business deve poter assumere, licenziare, pagare e gestire i dipendenti, ho creato la classe **Employee** che rappresenterebbe la classe associativa tra un Business e una Persona, ovvero una persona all'interno di un'azienda.

Ogni dipendente deve essere associato a un Business e deve poter cambiare Business nel corso della simulazione. La classe Employee utilizza il pattern *Component* avendo in essa sia un'istanza di Business che di Person.
Per la gestione dei possibili disoccupati ho creato la classe **EmploymentOffice**, un record che mantiene una lista dei disoccupati, mentre la **EmploymentOfficeManager** gestisce la logica di assunzioni, licenziamenti e paghe durante la simulazione.

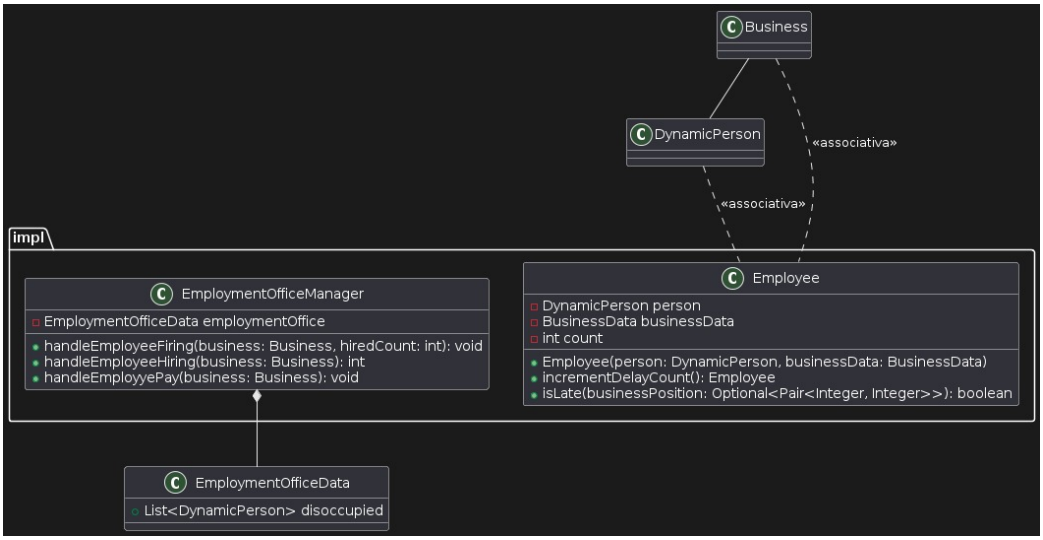

Figura 2.17: Schema UML

Motivazioni: L'uso del pattern Component per la gestione dei dipendenti assicura una chiara separazione delle responsabilità, rendendo il sistema più modulare e manutenibile. La creazione di un oggetto Employee crea una chiara distinzione tra le persone che lavorano e chi no, rendendo più facile le operazioni all'interno dei business.

ClockObserverBusiness è un componente chiave per la gestione del tempo e delle operazioni basate su di esso nella simulazione. Si occupa della tempistica delle assunzioni, che avvengono all'inizio della giornata di lavoro per ogni azienda, dei licenziamenti, che avvengono in modo analogo alla fine della giornata, e delle paghe.

L'uso del pattern *Observer* nella classe ClockObserverBusiness permette di centralizzare la gestione temporale e di sincronizzare le operazioni dei business in modo coerente. Questo approccio evita la duplicazione della logica temporale in ogni singolo business, rendendo il sistema più modulare e manutenibile. Il pattern garantisce che i business vengano notificati delle variazioni temporali.

Graficamente mi sono occupato della creazione e logica sia dello slider che dei grafici relativi ai business presenti nelle classi MapModel.

Infine ho aggiornato le persone con la mia idealizzazione dei business nei loro metodi di creazione e movimento.

Ho riscontrato problemi nello store dei dati e con i passaggi delle classi dovuti al fatto che non fossero oggetti immutabili, per risolvere ciò ho utilizzato i record e usato in modo più conscio i getter e setter, anche se ahimè non sono riuscito a sistemarli tutti.
Ho avuto problemi anche con la congestione, arrivava un indice negativo, non ho chiaro come fosse possibile ma sono riuscito a risolvere la cosa mettendo un controllo quando andavo a decrementare il numero di persone nella linea.
Ho avuto problemi con l'assegnazione dei business alle persone dovuto al fatto che la loro logica era stata implementata senza tener conto che fosse cambiabili, so riuscito a sistemare la cosa rimuovendo il campo dal record delle persone, ricreando il metodo che crea le persone inizializzandole all'inizio con un opzionale vuoto andando in un secondo ciclo ad assegnarglielo, e ho risolto il problema del loro movimento calcolando la strada che devono fare per andare a lavoro all'interno del setter di business che veniva chiamato una volta che gli veniva assegnato un business e venivano assunti, il loro movimento l'ho corretto mettendo un piccolo controllo nello switch nello stato working controllando se il business fosse presente, infine ho settato il costruttore nel quale venivano messi gli orari di apertura e chiusura del proprio business delle persone a 0 e li ho cambiati tramite setter in un secondo momento, sia nel caso di assunzione che licenziamento tramite nella classe **EmploymentOfficeManager**.

Era necessario suddividere la mappa in zone e gestire le informazioni relative a ciascuna zona. Ho utilizzato un record **ZoneData** per gestire le informazioni delle zone e la classe **Boundary** per dividere la mappa in zone. La classe Zone rappresenta una singola zona della città, contenente i dati specifici e le funzionalità necessarie per interagire con gli agenti presenti nella zona. L'uso del record ZoneData per gestire le informazioni delle zone assicura che i dati siano strutturati in modo chiaro, accessibile e sicuro. La divisione viene fatta attraverso le coordinate usando la classe Boundary.

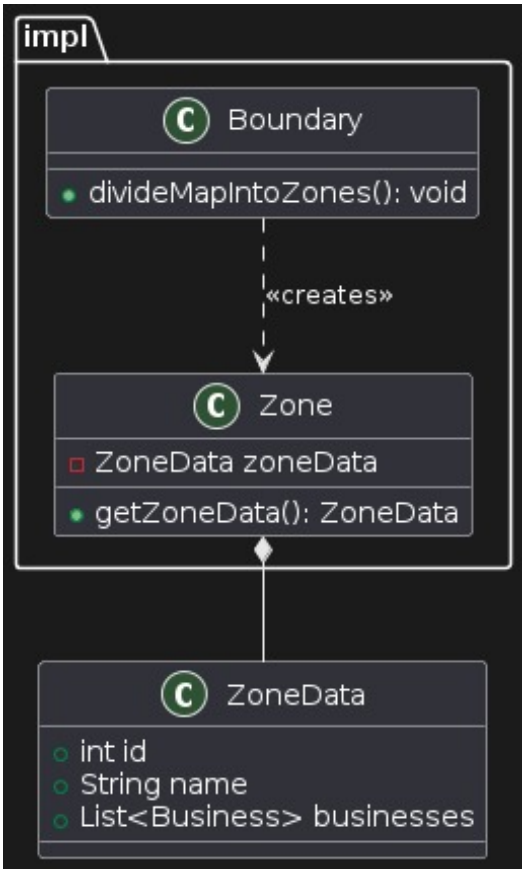

Figura 2.18: Schema UML

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per il testing delle nostre classi abbiamo usato la libreria di JUnit in generale , in più anche l'ausilio della libreria Mockito per esempio per il test del controller. Di seguito forniremo le coperture dei test ed anche una figura rappresentativa utilizzando JaCoCo per la visualizzazione totale delle coperture dei nostri test.

- nel package unibo.citysimulation.model.graphics.impl copertura 100%
- nel package unibo.citysimulation.model.transport.impl copertura 87%
- nel package unibo.citysimulation.model.zone copertura 94%

Di seguito per completezza un immagine con tutte le coperture dei test.

city-life

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
unibo.citysimulation.model.graphics.impl		100%		100%	0	31	0	73	0	25	0	3
unibo.citysimulation.model.business.utilities		100%		n/a	0	6	0	20	0	6	0	6
unibo.citysimulation.model.person.api		100%		n/a	0	2	0	5	0	2	0	2
unibo.citysimulation.model.zone		94%		60%	5	27	7	71	1	22	0	6
unibo.citysimulation.model.map.impl		93%		80%	5	48	10	119	3	43	1	5
unibo.citysimulation.model.transport.impl		87%		66%	3	18	8	45	1	15	0	2
unibo.citysimulation.model.clock.impl		74%		59%	9	34	16	71	3	23	0	4
unibo.citysimulation.model		70%		31%	30	67	43	144	10	45	0	3
unibo.citysimulation.model.business.impl		63%		40%	23	54	41	104	9	33	0	5
unibo.citysimulation.model.person.impl		60%		40%	27	60	54	132	13	38	0	4
unibo.citysimulation.utilities		57%		66%	5	11	4	14	3	8	1	2
unibo.citysimulation.controller		25%		0%	28	35	96	128	25	32	5	8
unibo.citysimulation.view		10%		n/a	11	13	60	70	11	13	1	2
unibo.citysimulation.view.sidepanels.graphics		0%		0%	20	20	95	95	19	19	3	3
unibo.citysimulation.view.sidepanels		0%		n/a	18	18	109	109	18	18	2	2
unibo.citysimulation.view.map		0%		0%	19	19	73	73	16	16	1	1
unibo.citysimulation.view.sidepanels.clock		0%		0%	9	9	38	38	8	8	1	1
unibo.citysimulation		0%		n/a	10	10	37	37	10	10	5	5
Total	3.148 of 6.441	51%	113 of 206	45%	222	482	691	1.348	150	376	20	64

Figura 3.1: Test su JaCoCo

3.2 Note di sviluppo

3.2.1 Sajmir Buzi

In questo paragrafo elencherò brevemente parti di codici dove ho usato espressioni più avanzate , ciò non impedisce che siano state usate anche in altre parti del progetto.

- **Utilizzo dei JSON**: per la creazione della *TransportFactory* e anche per la creazione della *ZoneFactory* fatta dal collega. Il link è il seguente: **TransportFactoryImpl.java su GitHub**

- **Utilizzo degli Stream**: utilizzo degli Stream per i getter delle linee di trasporto e delle durate delle linee di trasporto , atraverso gli Stream che possono sembrare di difficile utilizzo semplifica di tanto la struttura del codice **ZoneTable.java su GitHub**

- **Utilizzo della libreria Mockito** : Attraverso la libreria Mockito possiamo usufruire delle operazioni Di Mocking e Stubbing, attraverso il Mocking si possono creare oggetti fittizi per le classi dipendenti. Questo è utile per isolare l'unità di codice sotto test, permettendo di controllare il comportamento degli oggetti dipendenti. Invece Lo Stubbing consente di definire il comportamento dei metodi degli oggetti mock. Ad esempio, è possibile specificare cosa dovrebbe restituire un metodo quando viene chiamato. In poche parole l'utilizzo di questa libreria è utile ad esempio nei Controller come ad esempio *InputController* dove bisogna testare le caratteristiche singole dei bottoni presenti nel panel di Input: **InputControllerTest.java su GitHub**

3.2.2 Gabriele Rubboli Petroselli

Utilizzo di lambda expressions

Utilizzate in vari punti, un esempio è qui:https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/clock/impl/ClockModelImpl.java#L127

Utilizzo di Stream

Utilizzati in vari punti, come qui:https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/person/impl/PersonFactoryImpl.java#L34

Utilizzo di Optional

Utilizzati in vari punti, ad esempio per rendere la posizione empty quando la persona si trova sulle linee di trasporto e si sta muovendo:https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/person/impl/StaticPersonImpl.java#L115

Utilizzo di record

Utilizzato per mantenere in modo efficace le informazioni immutabili delle persone senza dover fare tutti i getter e appesantire ulteriormente il codice:https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6d
src/main/java/unibo/citysimulation/model/person/api/PersonData.java#L15

3.2.3 Marco Ravaioli

Nelle classi dei model, impiegati per l'elaborazione dei dati, sono presenti diverse parti di codice con stream e lambda per una migliore leggibilità del codice. Inoltre, ho fatto uso di librerie per la gestione dei Chart che leggono i dati all'interno di dataset per ottimizzare la gestione della memoria.

- **Utilizzo degli Stream**: All'interno della classe *DatasetManager* viene fatto un uso costante degli stream che agevola le modifiche alla struttura dati che contiene i dataset. La struttura è una lista di dataset;

ciascuno contiene delle serie. Ogni serie è un oggetto che associa a un valore un altro, nel senso geometrico di ascissa e ordinata. Vorrei porre in evidenza il metodo *RemoveOldColumn* che rimuove la prima linea di ogni serie di ciascun dataset per permettere l'aggiunta successiva di altri termini. Permalink:https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/graphics/impl/DatasetManager.java#L36

- **Utilizzo dello switch expression**: La classe *DynamicPerson* utilizza la switch expression per aumentare la leggibilità del codice, riducendo il numero di righe. Il metodo *checkState*, richiamato dal controller, compie uno switch che determina il comportamento della persona in base al suo stato attuale. Ogni opzione dello switch porta a un metodo differente. Permalink:https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/person/impl/DynamicPersonImpl.java#L90

- **Utilizzo dei lambda**: I lambda expression sono fondamentali per la riduzione del codice, prendendo in input i parametri che utilizzano e compiendo un'espressione va ad accorciare ottenendo immediatamente il risultato. In *MapModelImpl*, viene usata insieme allo stream, ad esempio in *getPersonInfos* per estrarre dati per una struttura complicata (con Map¡¿, Pair¡¿). Nello stream il filtro controlla se una persona ha una posizione opzionale tramite lambda e successivamente viene creata la chiave e il pair come valore, entrambi con l'ausilio di lambda. Permalink:https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12
src/main/java/unibo/citysimulation/model/map/impl/MapModelImpl.java#L85

- **Utilizzo librerie jfree**: Viene adoperata la libreria jfree per la gestione e rappresentazione dei grafici. All'interno di *DatasetManager (MapModelImpl)* si implementano i metodi per la modifica delle XYSeriesCollection, mentre in *GraphicsPanel* vengono creati e aggiornati i XYChart che sono la rappresentazione grafica caratterizzata con la personalizzazione del renderer per facilitare la lettura. Permalink:https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb
src/main/java/unibo/citysimulation/view/sidepanels/graphics/GraphicsPanelImpl.java#L72

Per la rappresentazione dei grafici, studiata insieme ad alcuni colleghi, si è pensato di utilizzare la libreria JFreeChart, che inizialmente non conoscevo. Approfondendo, ho notato che JFreeChart è una libreria molto complessa e altamente personalizzabile, fornendo molti strumenti che la rendono estremamente potente per la creazione di grafici avanzati.

Abbiamo scelto JFreeChart perché è molto consigliata sul web. Tuttavia, durante il progetto, abbiamo utilizzato solo una piccola parte delle funzionalità offerte da JFreeChart, focalizzandoci principalmente sulla gestione delle serie di dati con XYSeriesCollection per rappresentare graficamente l'evoluzione di variabili nel tempo.

1 JFreeChart GitHub Repository

2 Stack Overflow: Add a JFreeChart to JPanel

3 JFreeChart API Documentation: XYSeries

Nella classe *ImageHandler* della mappa viene implementata la gestione della serializzazione e deserializzazione dei campi transienti. Questo è un esempio di utilizzo delle funzionalità di Java per gestire correttamente oggetti che contengono campi non serializzabili. Non si era posto il problema fino alla segnalazione ricevuta da spotbugsMain che mi ha portato a cercare un metodo di serializzazione manuale di *BufferedImage* che non è serializzabile direttamente.

- CodeGuru: Serializing BufferedImage Objects

3.2.4 Alessio Bifulco

Uso di Lambda:`https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/business/impl/EmploymentOfficeManager.java#L107`

Uso di Stream:`https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/business/impl/EmploymentOfficeManager.java#L147`

Uso di Optional:`https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/business/impl/EmploymentOfficeManager.java#L112`

Uso dei record:`https://github.com/RubboGabri/OOP23-city-life/blob/1240221c93a18c4f56d12289b6deedcb84b5ecfe/src/main/java/unibo/citysimulation/model/business/utilities/EmploymentOfficeData.java#L13`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Sajmir Buzi

Il percorso eseguito per la realizzazione del progetto è stato molto difficile ma alla stesso tempo non vedevo l'ora di provare a mettermi in gioco.penso di aver dato un buon contributo al progetto anche se ovviamente si poteva fare meglio dal punto di vista della simulazione, a mio vedere abbastanza in stato embrionale quindi sicuramente migliorabile.Per quando riguarda criticità maggiori devo dire che con un membro del gruppo è mancato a mio vedere la consultazione dei lavori da fare quindi io mi sono trovato in difficoltà ,e penso anche gli altri componenti,a dovere unire i vari pezzi di codice ciò penso abbia compromesso al gruppo di eseguire un lavoro in maniera più efficiente.Dato che è stato il primo progetto di questa portata penso che anche l'inesperiernza abbia giocato un ruolo chiave nella realizzazione del progetto. Alla fine anche se con mille difficoltà sono soddisfatto del lavoro eseguito e spero questo progetto possa essere una base magari per un progetto futuro.

4.1.2 Gabriele Rubboli Petroselli

Onestamente la mia preferenza è sempre stata lavorare in singolo o massimo in gruppi da due persone, conoscendo bene l'altro membro, per evitare qualsiasi tipo di conflitto, ma nonostante questo mi ritengo più soddisfatto che non.
Credo di aver dato un buon contributo alla realizzazione del progetto, occupandomi spesso di risolvere anche situazioni non create da me.
Il progetto è venuto abbastanza bene negli aspetti fondamentali ma credo che si potesse arrivare anche a un livello leggermente superiore magari entrando in qualche funzionalità aggiuntiva.
La criticità che ho riscontrato e che mi sento di dover mettere per iscritto, è l'eccessiva perseveranza di alcuni membri del gruppo a non conosiderare il lavoro svolto dagli altri e i tempi stabiliti in precedenza, con evidenti ritardi dovuti a una programmazione volta solo a se stessa, della quale poi io e altri abbiamo dovuto risolvere i conflitti.
Nonostante questo, credo il progetto sia stato strutturato bene e mi ritengo complessivamente soddisfatto.

4.1.3 Marco Ravaioli

Il percorso che mi ha portato allo sviluppo dell'applicazione è stato molto altalenante e ricco di sfide. Il risultato finale si discosta notevolmente da ciò che avevo inizialmente immaginato e la direzione presa è diversa dal mio solito approccio ai progetti. Tuttavia, questa esperienza si è rivelata estremamente istruttiva, permettendomi di acquisire una maggiore familiarità con il codice e, soprattutto, con la metodologia di sviluppo applicata.

L'inesperienza è stata una delle principali difficoltà affrontate durante lo sviluppo dell'applicazione. Essendo il mio primo progetto in Java, ho dovuto confrontarmi con paradigmi di programmazione che non conoscevo a fondo. Questa incertezza mi ha portato a spendere molto tempo nella ricerca delle soluzioni corrette, rallentando il progresso del progetto. Tuttavia, ogni ostacolo superato ha rappresentato un'opportunità di apprendimento e crescita professionale.

Un'altra sfida significativa è stata la comunicazione all'interno del gruppo di lavoro. Nonostante gli sforzi per coordinare le attività e rispettare le scadenze, ho incontrato difficoltà nel collaborare con un membro del gruppo che ha dimostrato scarsa motivazione. Questo comportamento si è tradotto in una partecipazione intermittente agli appuntamenti e al mancato rispetto delle scadenze stabilite. La situazione è diventata particolarmente critica

nelle fasi finali del progetto, quando modifiche importanti al codice sono state effettuate all'ultimo minuto, costringendo il resto del gruppo a lavorare in fretta per integrare i cambiamenti.

Non nascondo che mi sono trovato più volte nella situazione di dover implementare parti di codice che inizialmente non erano a me designate, creando ancora più confusione nel momento in cui il legittimo proprietario le ha rifatte.

Nonostante queste difficoltà, il risultato finale dell'applicazione è per me motivo di soddisfazione. Il progetto rappresenta una solida base per futuri sviluppi, con ampie possibilità di potenziamento della parte relativa allo spostamento delle persone e di estensione delle funzionalità delle aziende in altri luoghi di ritrovo. La mia esperienza mi ha permesso di trarre molte considerazioni utili per il futuro, migliorando le mie competenze tecniche e la mia capacità di lavorare in team.

Sento di aver dato un buon contributo al gruppo rimanendo un punto di riferimento umano all'interno del gruppo, per quanto le mie conoscenze di programmazione e progettazione sono molto acerbe.

In conclusione, questa esperienza di sviluppo è stata formativa e mi ha fornito gli strumenti per affrontare progetti futuri con maggiore sicurezza e competenza. Le difficoltà incontrate, sebbene impegnative, hanno contribuito a rafforzare il mio approccio alla programmazione e alla gestione delle dinamiche di gruppo. Sono convinto che le lezioni apprese durante questo progetto saranno preziose per il mio percorso accademico e professionale.

4.1.4 Alessio Bifulco

Questo progetto è stato il mio primo progetto di gruppo e il primo di tale grandezza, e ciò si è notato. Per quanto una parte di me si senta soddisfatta del lavoro, non posso che sentirmi frustrato dall'organizzazione, di come è stato svolto l'intero progetto e la ripartizione dei lavori.
Credo, nonostante ciò, che l'esperienza sia costruttiva e dia un minimo di cognizione di come sia lavorare in gruppo con altre persone. Penso di aver appreso molto nell'ambito della programmazione OOP in questi mesi di programmazione e di aver capito l'importanza di programmare con cognizione e non mirando a un mero "funzionamento" ma puntando a un qualcosa di più.
Infine, per quanto trovi interessante l'idea di una simulazione agent-based, sopratutto in un ottica di possibile tesi, e reputi possibilmente riutilizzabile una simulazione del genere, non credo che ci rimetterò le mani se non a tempo perso giusto per vedere se ciò che avevo in mente fosse davvero realizzabile.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Alessio Bifulco

Ritengo che il corso sia ben strutturato e che dia materiale didattico più che sufficiente sia per un aspetto più teorico che uno più pratico. Personalmente, ho preferito le lezioni di laboratorio a quelle in aula per un mero gusto personale. Ritengo che, appreso ciò, sia giusto esprimersi anche su cosa si pensi sia meno giusto. Penso che avere sia un progetto che un esame scritto sia essenziale per capire fino in fondo la materia, ma che il numero di crediti, soprattutto se paragonato con gli altri corsi, sia non proporzionato, sia alla difficoltà, che trovo sia giusto ci sia, che alle tempistiche che richiede il progetto. Inoltre, non mi trovo molto d'accordo su alcune scelte dell'apprendimento. Penso si passi troppo tempo sui primi argomenti e che ciò vada a discapito di argomenti più difficili, che personalmente trovo più interessanti, come l'uso di lambda, stream o l'uso di pattern. Dunque penso che il corso sia interessante e ben strutturato, però reputo che risenta dei limiti temporali imposti dal università, suggerirei l'idea di dividerlo in due in modo da potersi soffermare di più sugli argomenti da me citati e in modo che possa dare un numero di crediti consono.

4.2.2 Marco Ravaioli

Premettendo che ho frequentato il corso nell'anno accademico 2021/2022 e non sono aggiornato sulle pratiche attuali, mi permetto di suggerire di approfondire e trattare maggiormente la gestione pratica di progetti di queste dimensioni. Inoltre, consiglio di rendere obbligatorio il superamento dell'esame di laboratorio per poter procedere allo sviluppo del progetto, poiché la preparazione per l'esame è propedeutica alla realizzazione di codice con costanza e sicurezza in Java.

Appendice A

Guida utente

L'applicazione si apre con tre pulsanti autoesplicativi. Una volta entrati nella schermata principale, si riconoscono i pannelli colorati, ciascuno con una funzione diversa:

- **In alto a sinistra**: Slider di input, in cui si possono regolare aspetti come il numero totale di persone e la capacità massima delle linee di trasporto.
- **In basso a sinistra**: Visualizzazione delle informazioni relative alla zona premuta sulla mappa.
- **In alto a destra**: Orologio, con il pulsante che regola la velocità (x1, x2, x10, x20).
- **In basso a destra**: Grafici:
  - Il primo in alto rappresenta l'andamento dello stato delle persone.
  - Il secondo rappresenta il livello di congestione per ogni linea presente.
  - Il terzo rappresenta il numero di persone assunte per ogni tipo di azienda presente.
- **Al centro**: Mappa, su cui sono disegnate le rappresentazioni delle entità presenti. Le persone sono disegnate come puntini (blu a casa e rosse in azienda), le aziende sono quadrati marroni e le linee di trasporto cambiano colore (da verde chiaro a rosso) in base al livello di congestione.