PROGETTO D'ESAME DI PROGRAMMAZIONE III E LABORATORIO

A.A. 2022/2023

Componenti progetto:

Vetrano Alessio 0124002326

Galluccio Luigi 0124002293

D'Angelo Simone 0124002601

Tipologia progetto:

Gestionale con Intelligenza Artificiale

Sommario

TRACCIA DEL PROGETTO

L'applicazione che si vuole sviluppare gestisce il carico e il rintracciamento di merci nel campo della logistica, utilizzando un algoritmo genetico per ottimizzare il carico dei veicoli. L'algoritmo genetico è un metodo di ottimizzazione basato su principi ispirati alla biologia evolutiva, come la selezione naturale e la riproduzione con mutazione. L'applicazione sarà utilizzata da un'azienda di trasporto che consegna merci in diverse sedi in Italia, e che ha a disposizione un numero di veicoli identificati da un codice, tipo di veicolo e capienza container. Il collo è identificato da un codice, mittente, destinatario e peso. L'applicazione deve gestire il carico dei colli nei container scegliendo il veicolo più adatto. Il corriere, inoltre, aggiorna lo stato del collo ad ogni centro di smistamento, il quale deve essere rintracciato dal destinatario mediante il suo codice.

L'INTELLIGENZA ARTIFICIALE

La traccia del nostro progetto richiede l'implementazione di un particolare algoritmo per il riempimento dei veicoli: viene scelto il veicolo che può contenere un numero maggiore di colli (sempre in relazione alla capienza del veicolo); quindi, si vuole risolvere un problema di ottimizzazione del carico.
L'algoritmo genetico sarà implementato in Java mediante la classe Popolazione, che estende la classe Vector<Cromosoma> e utilizzerà metodi come pickOne(), sumBody() e findBest() per aiutare nel processo di scoperta della soluzione migliore.

RISOLUZIONE DEL PROBLEMA

Il problema sopra citato, trova una sua risoluzione nell'implementazione di un particolare algoritmo, ossia quello genetico.
Quest'ultimo è un metodo di ottimizzazione basato su principi ispirati alla biologia evolutiva, come la selezione naturale e la riproduzione con mutazione. Viene utilizzato in una vasta gamma di campi, tra cui l'intelligenza artificiale, o la robotica. In generale, questo tipo di algoritmo utilizza una popolazione di soluzioni candidate, che evolvono attraverso diverse generazioni attraverso processi di selezione, crossover e mutazione.

IMPLEMENTAZIONE DELL'ALGORITMO

L'implementazione dell'algoritmo di IA è basata sull'algoritmo genetico.

Questo codice rappresenta una classe Java chiamata Popolazione, che estende la classe *Vector<Cromosoma>* e utilizza l'algoritmo genetico per scoprire la soluzione migliore per un singolo veicolo. La classe ha un costruttore che accetta un parametro di dimensione e crea un *Vector* di Cromosoma di quella dimensione.

Il metodo *discoverBestSolutionForSingleVehicle*() accetta tre parametri, un oggetto *Vehicle*, un *Vector* di oggetti Collo e un intero generations. Il metodo inizializza la popolazione con soluzioni casuali, quindi effettua un ciclo per un numero specificato di generazioni, creando una nuova generazione basandosi sulle soluzioni migliori della generazione precedente. Il metodo restituisce infine la soluzione migliore trovata.

I metodi privati *pickOne*(), *sumBody*() e *findBest*() sono utilizzati per aiutare nel processo di scoperta della soluzione migliore.
Il codice rappresentato dalla classe Java Cromosoma estende la classe *Vector<Collo>* e rappresenta un singolo individuo nell'algoritmo genetico. La classe contiene un riferimento a un oggetto *Vehicle* e mantiene traccia del peso totale di tutti gli oggetti Collo contenuti nel vettore, nonché di un valore chiamato "body".

La classe ha un metodo chiamato *random*() che accetta un parametro di tipo *List<Collo>* e inizializza il vettore con elementi casuali scelti dalla lista fornita, finché il peso totale degli elementi non supera la capacità del veicolo. Il metodo privato *selectionRandom*() viene utilizzato per selezionare un elemento casuale dalla lista fornita che non sia già presente nel vettore.

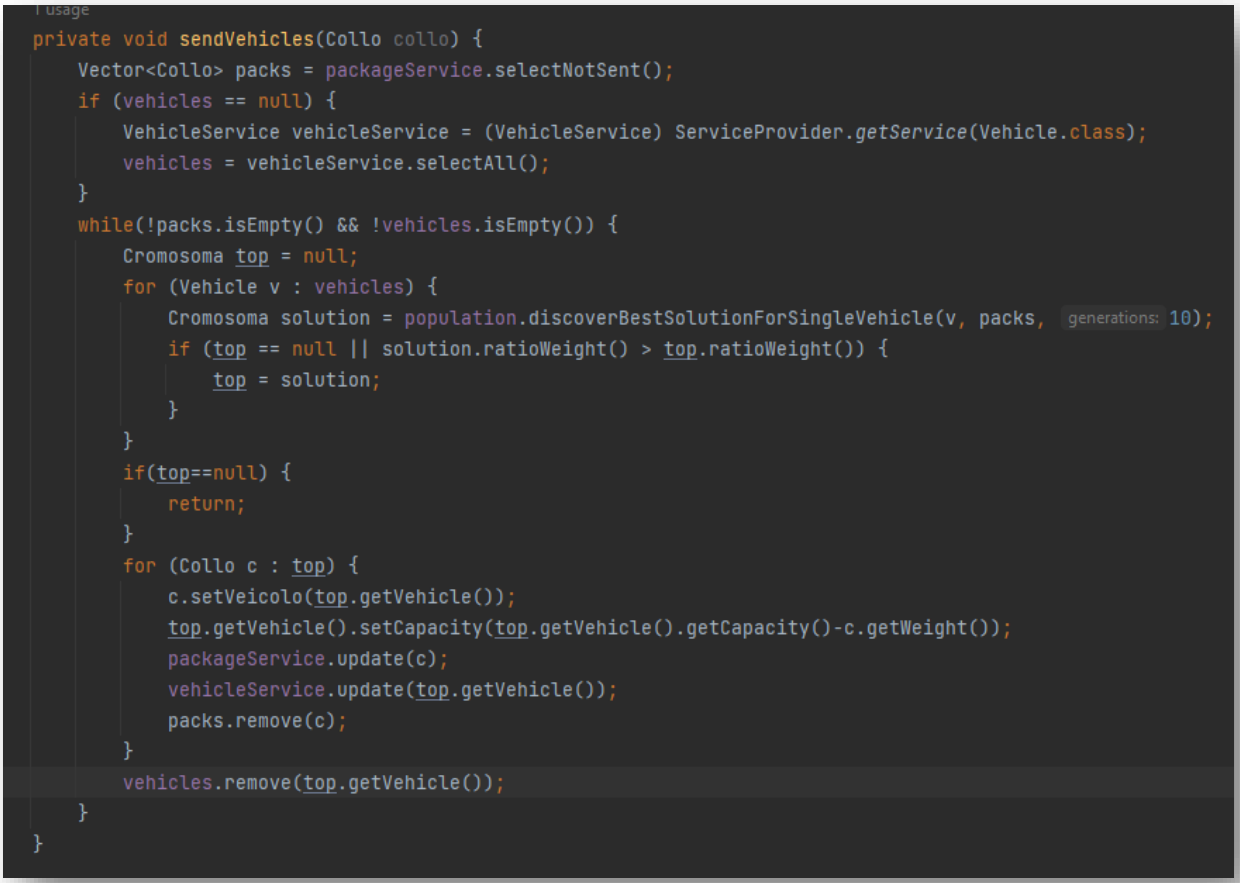
Il metodo *wetsuit*() accetta un parametro di tipo *List<Collo>* e, con una certa probabilità (definita dalla costante di classe *mutationRate*), sostituisce elementi del vettore con elementi casuali scelti dalla lista fornita, finché il peso totale degli elementi non supera la capacità del veicolo.

Il metodo *body*() restituisce il valore "body" del Cromosoma, il metodo *ratioWeight*() restituisce il rapporto tra il peso totale degli elementi del vettore e la capacità del veicolo.

Il metodo *clone*() restituisce una copia del Cromosoma corrente e il metodo *getVehicle*() restituisce il riferimento al veicolo associato al Cromosoma e inoltre questo metodo definisce il primo design pattern utilizzato cioè Prototype trattato in seguito.
Per quanto riguarda il metodo send(), è utilizzato per inserire un nuovo pacco

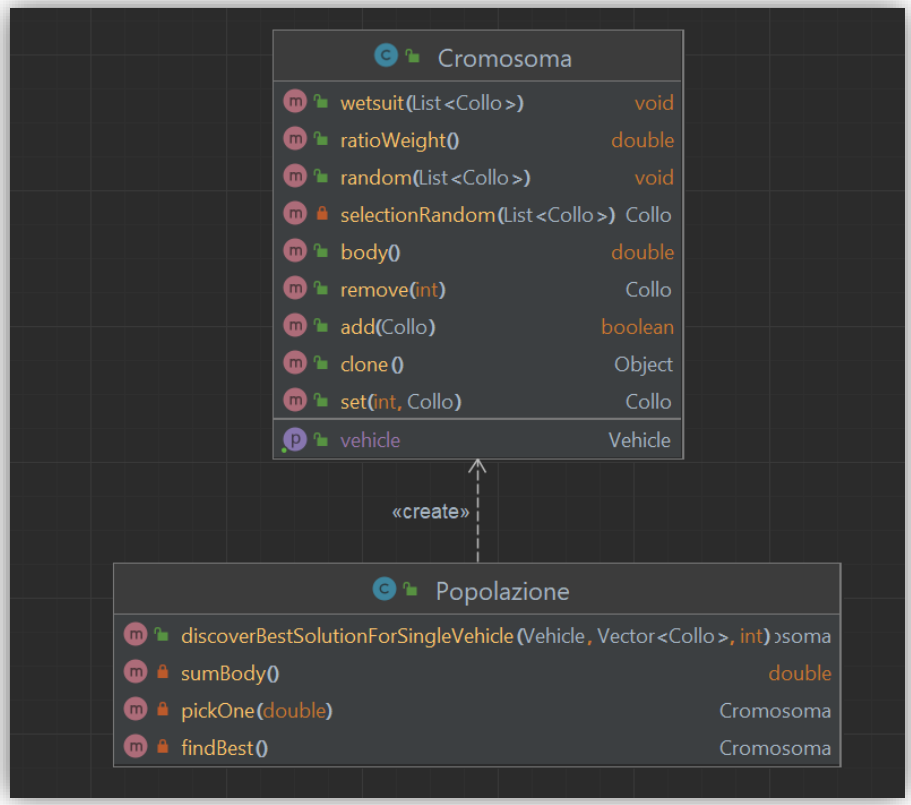
nell'applicazione e avviare il processo di ottimizzazione del carico utilizzando l'algoritmo genetico per trovare la soluzione migliore per l'inserimento del pacco in un veicolo. In questo modo, l'applicazione sarebbe in grado di assegnare il pacco al veicolo più adatto in base alle sue caratteristiche e al peso. In generale, l'applicazione sviluppata utilizzando l'algoritmo genetico sarà in grado di ottimizzare il carico dei veicoli e fornire un sistema di rintracciamento efficiente per le merci in transito, migliorando l'esperienza utente e aumentando l'efficienza dell'azienda di trasporto.

Di seguito l'implementazione del metodo *send*() che consente di spedire un pacco che utilizza l'utilizzo dell'algoritmo genetico in modo da scegliere la soluzione migliore per l'inserimento dei colli nei veicoli che ne possono contenere di più in base alle informazioni sul peso.



```
1 usage
private void sendVehicles(Collo collo) {
    Vector<Collo> packs = packageService.selectNotSent();
    if (vehicles == null) {
        VehicleService vehicleService = (VehicleService) ServiceProvider.getService(Vehicle.class);
        vehicles = vehicleService.selectAll();
    }
    while(!packs.isEmpty() && !vehicles.isEmpty()) {
        Cromosoma top = null;
        for (Vehicle v : vehicles) {
            Cromosoma solution = population.discoverBestSolutionForSingleVehicle(v, packs,  generations: 10);
            if (top == null || solution.ratioWeight() > top.ratioWeight()) {
                top = solution;
            }
        }
        if(top==null) {
            return;
        }
        for (Collo c : top) {
            c.setVeicolo(top.getVehicle());
            top.getVehicle().setCapacity(top.getVehicle().getCapacity()-c.getWeight());
            packageService.update(c);
            vehicleService.update(top.getVehicle());
            packs.remove(c);
        }
        vehicles.remove(top.getVehicle());
    }
}
```

Di seguito il diagramma UML delle Classi per la gestione dell'algoritmo genetico:

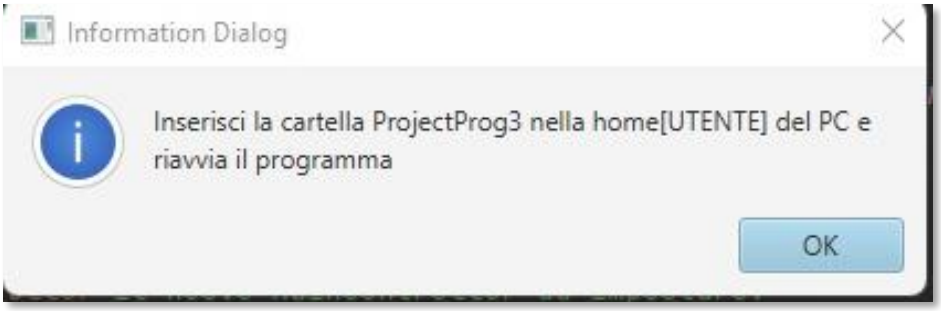


USO DEI FILE CSV

Il progetto è basato sull'utilizzo di file CSV. I file attraverso le classi *Table, Table Adapter e Table Provider* vengono creati all'interno di una cartella chiamata 'ProjectProg3' nella directory home del PC in cui verrà usato l'applicativo. Nel caso ci si trovi in sistemi Linux o MacOS, il programma si occuperà di creare i file sede.csv e il file vehicles.csv popolati attraverso un *BashScript*, lanciato come processo nell'applicativo JAVA alla prima esecuzione del programma, inserendoli nella directory 'ProjectProg3' precedentemente citata. Il controllo della prima esecuzione del programma verrà effettuato mediante la creazione di un file nascosto (".wanted") nella directory home del PC. Nelle seguenti esecuzioni, il programma controllerà l'esistenza del file e si occuperà di non creare nuovamente i file CSV.
Di seguito lo screen dello script utilizzato:

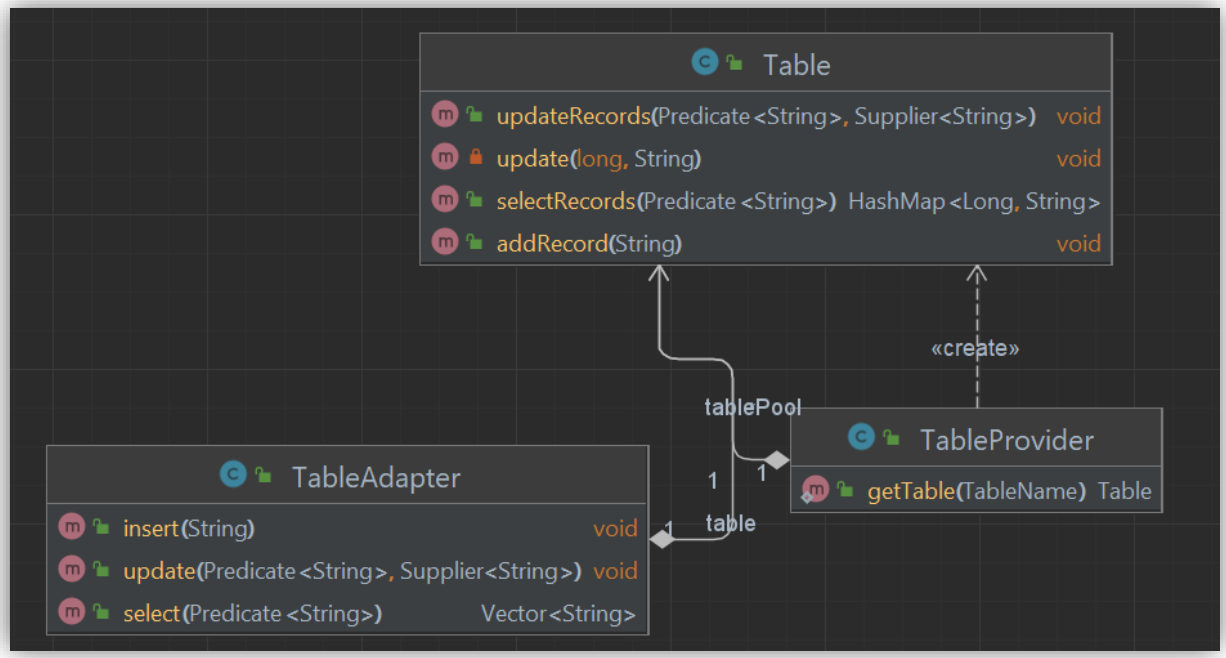
```
#!/bin/bash
mkdir ~/ProjectProg3
echo 'BRT;Via Paolo Borsellino;Casandrino(NA)
SDA;Via Galileo Ferraris, 66;Napoli(NA)
Poste Italiane;Via Abate Minichini, 1;Napoli(NA)
Poste Italiane;Via Polveriera, 4;Napoli(NA)
Poste Italiane;Via Virgilio, 1/B;Roma(RM)
SDA;Via Corcolle, 12/14;Roma(RM)
BRT;Via del Ponte Pisano, 94;Roma(RM)
BRT (Fermopoint);Via Monte Amiata, 4;Milano(MI)
BRT Depot;Via Dione Cassio, 7;Milano(MI)
SDA Express Courier;Via Giudenzio Fantoli, 30;Milano(MI)
SDA;Via Gran Sasso, 23;Milano(MI)
SDA;Via dei Sassetti, 5;Firenze(FI)
Poste Italiane;Via Pellicceria, 3;Firenze(FI)
Poste Italiane;Via dei Mazzetta;Firenze(FI)
Poste Italiane;Via Urbano III, 2;Milano(MI)
Poste Italiane;Via Alfredo Cappellini, 17;Milano(MI)
Poste Italiane;Via Vittorio Alfieri, 10;Torino(TO)
Poste Italiane;Piazza Santa Giulia, 12;Torino(TO)
Poste Italiane;Via Luigi Ferdinando Marsigli, 22;Torino(TO)
BRT Depot;Rivalta di Torino, 7;Torino(TO)
UPS;Via Raspini, 1;Torino(TO)
UPS;Via di Novella, 10;Roma(RM)
UPS;Via di S.Costanza, 30;Roma(RM)
UPS;Via Enea, 83;Roma(RM)
UPS;Via Gaudenzio Fantoli, 15/2;Milano(MI)
UPS(Access Point);Via Foria, 82/82;Napoli(NA)
UPS(Access Point);Via Agostino Depretis, 135;Napoli(NA)
UPS(Access Point);Via Arenaccia, 165;Napoli(NA)
Fedex;Via Locatelli;Casoria(NA)
Fedex;Via di Salome, 124; Roma(RM)
Fedex;Largo XXI Aprile, 2; Roma(RM)
Fedex;Via Alberico Albricci, 10; Milano(MI)' >> ~/ProjectProg3/sede.csv
```

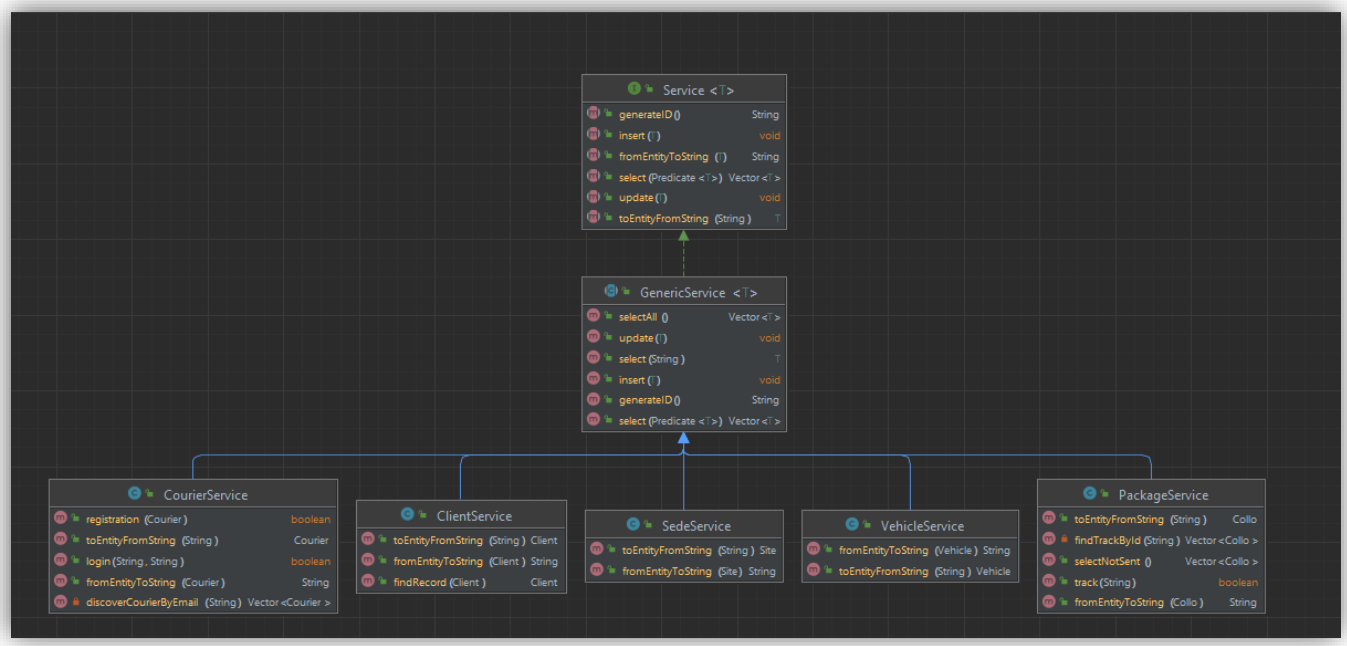
Nel caso ci trovassimo invece in ambiente Windows, l'applicazione si occuperà di far comparire un messaggio d'errore attraverso una *Dialog* che avvertirà l'utente di spostare la cartella (precedentemente inserita all'interno dell'archivio del programma) all'interno della directory 'ProjectProg3' nella home del PC.

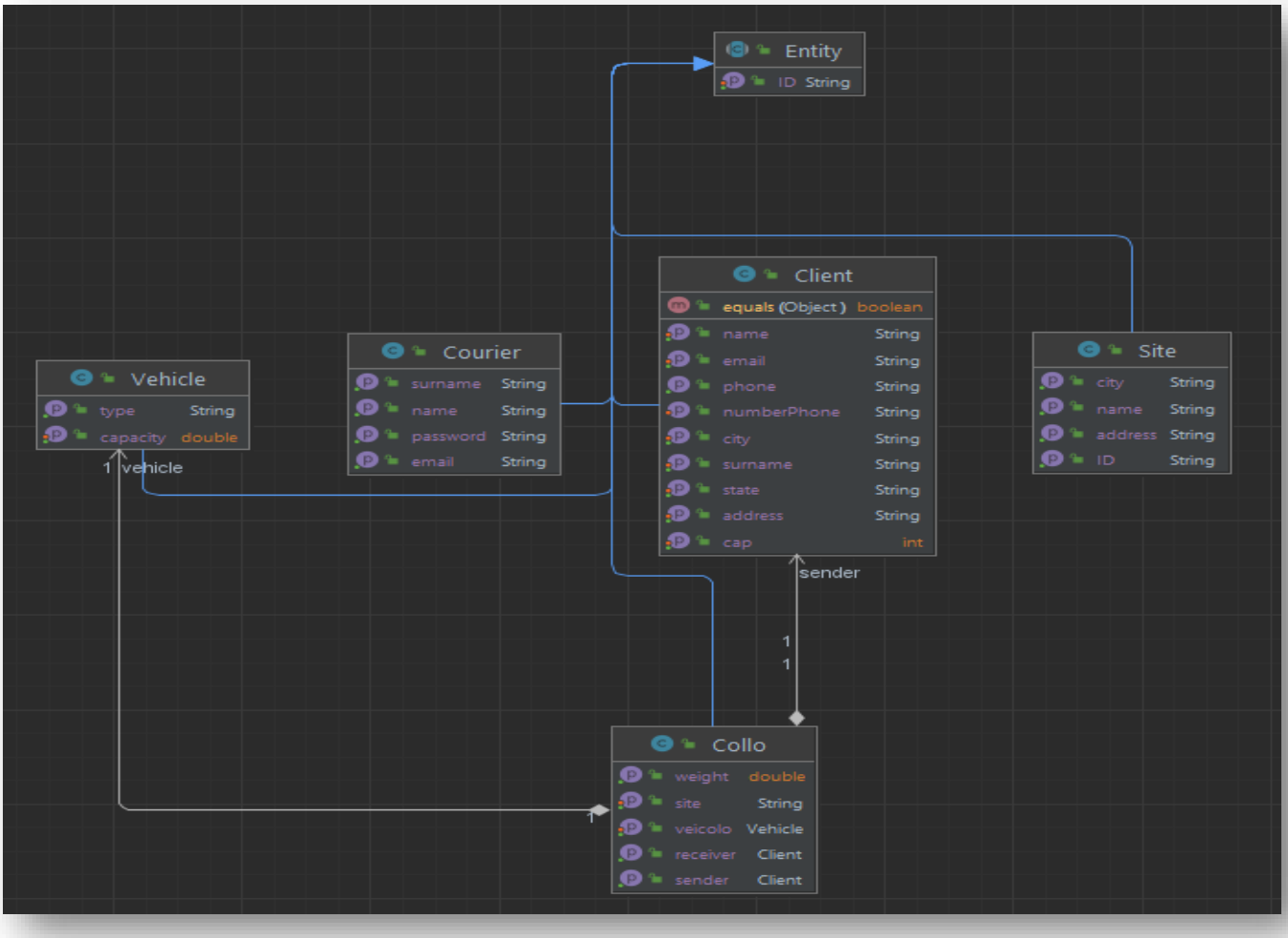


Nello specifico l'utilizzo dei file CSV (*Comma Separeted Value*) sono utilizzati per conservare i dati dell'applicazione in modo permanente nel PC. I file sono associati a delle classi che contengono delle istanze corrispondenti ai valori dei campi del file. La classe TABLE si occupa di gestire la scrittura sul file utilizzando i metodi per inserire, aggiornare o eliminare elementi nel file. Utilizzando questa tipologia di classe facciamo riferimento al secondo Design Pattern ovvero il Template Method che abbiamo utilizzato nel nostro progetto di cui tratteremo seguentemente. Attraverso il metodo

update della classe TABLE riusciamo a scrivere nei file a cui facciamo riferimento.
Oltre l'utilizzo di TABLE, la classe SERVICE rappresenta un'interfaccia di un servizio generico chiamato "Service" in Java. L'interfaccia estende l'entità generica T che estende la classe ENTITY.
L'utilizzo di una interfaccia generica consente di creare una classe di servizio per ogni tipo di entità senza dover ripetere il codice per ogni tipo di classe di entità.
Di seguito i diagrammi UML per la gestione dei *file, delle entità e dei servizi*:







DESIGN PATTERN

I design pattern sono soluzioni di progettazione generiche che possono essere utilizzate per risolvere problemi comuni nello sviluppo di software. Essi forniscono una sorta di "vocabolario comune" per i progettisti, consentendo loro di comunicare efficacemente e di condividere le loro conoscenze.

I design pattern sono divisi in tre categorie: creazionali, comportamentali e strutturali.

I pattern creazionali riguardano la creazione di oggetti e la gestione delle loro dipendenze. Essi includono pattern come Singleton, Factory Method e Builder. Questi pattern consentono di creare oggetti in modo flessibile e controllato, eliminando la dipendenza dalle classi concrete e migliorando la testabilità del codice.

I pattern comportamentali riguardano la comunicazione tra gli oggetti e la gestione dei loro comportamenti. Essi includono pattern come Observer, Mediator e Strategy. Questi pattern consentono di gestire efficacemente la comunicazione tra gli oggetti, rendendo il codice più leggibile.

I pattern strutturali riguardano la disposizione degli oggetti e la loro organizzazione in un sistema. Essi includono pattern come Adapter, Bridge e Composite. Questi pattern

consentono di organizzare gli oggetti in modo flessibile e modulare, migliorando la scalabilità e la manutenibilità del codice.
In generale, l'utilizzo dei design pattern consente di creare software più robusto, scalabile e *con una maggiore possibilità di manutenzione*.

I PATTERN UTILIZZATI NEL PROGETTO

Il progetto utilizza sei Design Pattern:

- Template
- Adapter
- Factory Pattern
- Prototype
- Builder
- Flyweight

Nello specifico, la classe GENERICSERVICE utilizza il Design Pattern **Template Method** e consiste nel definire un algoritmo generale in una classe astratta e lasciare che le classi derivate forniscano le implementazioni dei passi specifici dell'algoritmo.

In questo caso, la classe GENERICSERVICE fornisce un'implementazione generale dei metodi di un servizio per gestire le operazioni CRUD (creazione, lettura, aggiornamento e eliminazione) su una tabella della base dati. I metodi forniscono la logica generale per inserire, selezionare, aggiornare ed eliminare dati dalla tabella, ma lasciano alla classe derivata la responsabilità di convertire gli oggetti *Entity* in stringhe e viceversa.

I metodi "*fromEntityToString*" e "*toEntityFromString*" sono dichiarati come metodi astratti e devono essere implementati dalle classi derivate per convertire gli oggetti Entity in stringhe e viceversa. La classe GENERICSERVICE utilizza questi metodi per inserire, selezionare, aggiornare ed eliminare dati dalla tabella.

La classe TABLEADAPTER utilizza il design pattern **Adapter**. Il design pattern Adapter consente di adattare l'interfaccia di una classe esistente in modo che possa essere utilizzata da un'altra classe.

In questo caso, la classe TABLEADAPTER adatta l'interfaccia della classe TABLE per soddisfare le esigenze della classe che la utilizza. La classe TABLE ha una propria interfaccia per l'aggiunta, la selezione e l'aggiornamento dei record, ma questa interfaccia non soddisfa le esigenze della classe che la utilizza. La classe TABLEADAPTER

fornisce metodi "*insert*", "*select*" e "*update*" che utilizzano l'interfaccia della classe TABLE ma la adattano per soddisfare le esigenze della classe che la utilizza.

In questo modo, la classe TABLEADAPTER consente di utilizzare la classe TABLE senza modificare il suo codice sorgente e senza che la classe che la utilizza debba conoscere i dettagli dell'implementazione della classe TABLE.

La classe SERVICEPROVIDER utilizza il Design Pattern **Factory Method**. Il pattern Factory Method consiste nel fornire un'interfaccia per creare oggetti, ma lasciare alle classi derivate la possibilità di decidere quale classe creare.

In questo caso, la classe SERVICEPROVIDER fornisce un metodo statico "getService" che, dato un tipo di entità, restituisce un oggetto di una classe che implementa GENERICSERVICE per quella tipologia di entità. Il metodo "*getService*" utilizza una mappa per memorizzare i servizi creati in precedenza in modo che non debbano essere ricreati ogni volta che vengono richiesti.

Il metodo "*createService*" è statico ed è utilizzato per creare una nuova istanza di una classe che implementa GENERICSERVICE per una data tipologia di entità. Il metodo "*createService*" utilizza una serie di if-else per determinare quale classe creare in base al tipo di entità fornito come parametro.

In questo modo, la classe SERVICEPROVIDER fornisce un'interfaccia per creare oggetti GENERICSERVICE per diverse tipologie di entità, ma lascia alle classi derivate la decisione su quale classe creare per una data tipologia di entità.

Il Design Pattern **Prototype** è rappresentato dal metodo *clone*() utilizzato nella classe CROMOSOMA. Il design pattern Prototype consiste nel fornire un'interfaccia per creare oggetti copiando un oggetto esistente invece di crearne uno nuovo da zero o attraverso una costruzione complessa. In questo caso, il metodo *clone*() restituisce una copia del Cromosoma corrente. Questo consente di creare nuove istanze di Cromosoma senza dover utilizzare il costruttore e fornire tutti i parametri necessari per creare una nuova istanza. Invece, si può semplicemente chiamare il metodo *clone*() su un'istanza esistente e ottenere una copia identica.

La classe CLIENTBUILDER utilizza il Design Pattern **Builder**. Il pattern Builder consente di creare oggetti complessi attraverso una serie di chiamate a metodi su un oggetto Builder. In questo caso, la classe CLIENTBUILDER fornisce una serie di metodi per impostare i vari campi di un oggetto "*Client*" e un metodo "*getClient*" per recuperare l'oggetto "*Client*" creato. I metodi di CLIENTBUILDER restituiscono sempre l'oggetto "*ClientBuilder*" stesso, in modo che sia possibile chiamare i metodi in modo concatenato. Ciò consente di creare un'interfaccia fluida per la creazione di oggetti

"*Client*". In questo modo, CLIENTBUILDER separa la creazione dell'oggetto "*Client*" dalle classi che lo utilizzano e rende il codice più leggibile e mantenibile.

La classe SERVICEPROVIDER utilizza il Design Pattern **Flyweight**. Il pattern Flyweight consiste nel condividere oggetti tra più utilizzatori in modo da ridurre il numero di oggetti creati e quindi la memoria utilizzata.

In questo caso, la classe SERVICEPROVIDER utilizza una mappa per memorizzare i servizi creati in precedenza in modo che non debbano essere ricreati ogni volta che vengono richiesti. Ogni volta che un chiamante richiede un servizio, la classe SERVICEPROVIDER controlla se il servizio è già stato creato e, in caso contrario, lo crea. In questo modo, vengono creati solo un numero limitato di oggetti GENERICSERVICE per ogni tipo di entità, riducendo la memoria utilizzata.

In questo modo, la classe SERVICEPROVIDER utilizza il pattern Flyweight per ottimizzare l'utilizzo della memoria condividendo gli oggetti GENERICSERVICE tra i chiamanti.

L'INTERFACCIA GRAFICA

Nel nostro progetto ci focalizziamo sulla progettazione dell'interfaccia grafica del nostro sistema. La semplicità d'uso, l'intuitività e la semplicità sono stati i principali obiettivi che ci siamo posti durante la fase di progettazione.

In particolare, ci siamo concentrati sulla creazione di un'interfaccia grafica semplice e intuitiva, che possa essere utilizzata da utenti di tutte le età e di qualsiasi livello di competenza tecnologica. La semplicità è stata ottenuta attraverso la riduzione del numero di elementi visivi e la semplificazione dei flussi di navigazione. L'intuitività è stata invece ottenuta attraverso la creazione di un layout chiaro e logico, che consente all'utente di comprendere rapidamente come utilizzare il sistema.

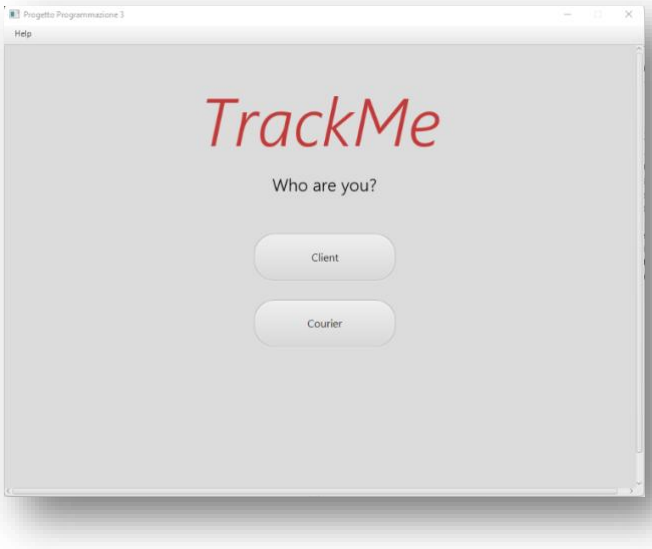
IL NOSTRO SUPPORTO GRAFICO

Nel nostro progetto abbiamo deciso di utilizzare JavaFX come strumento principale per la realizzazione delle interfacce grafiche.

JavaFX è una piattaforma software altamente flessibile e versatile, basata su Java, che consente la creazione di interfacce utente ricche e dinamiche che supportano la multimedialità. La scelta di JavaFX si è rivelata particolarmente adatta per la nostra applicazione, poiché questa piattaforma offre una vasta gamma di funzionalità grafiche avanzate e un vero e proprio linguaggio di programmazione, rendendo la programmazione delle applicazioni grafiche particolarmente agevole.

Per aiutare nella realizzazione della interfaccia grafica, abbiamo deciso di utilizzare un tool compatibile con JavaFX chiamato "Scene Builder". Questo è uno strumento di layout visivo che consente agli utenti di progettare rapidamente interfacce utente senza dover scrivere codice. Ciò ha ulteriormente semplificato la realizzazione dell'applicazione, rendendola più intuitiva e facile da utilizzare.

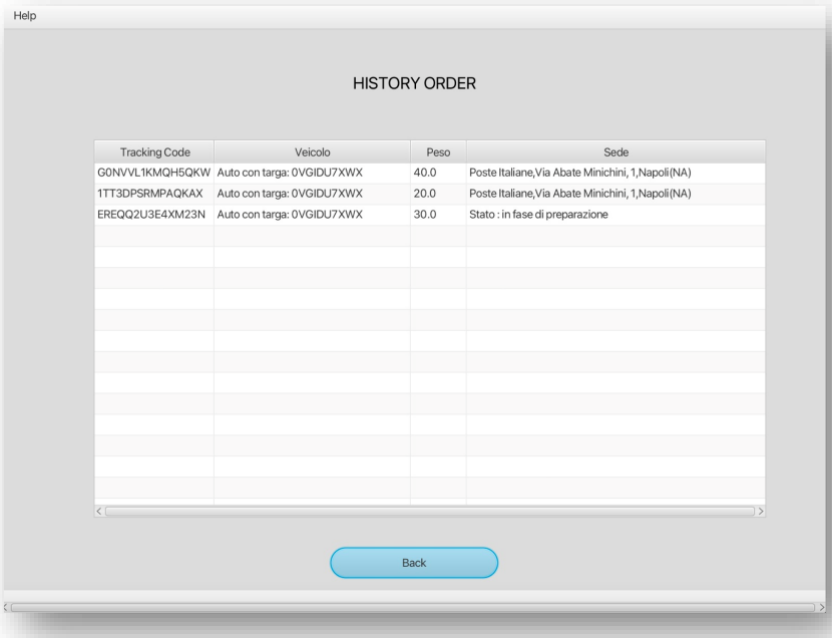
In conclusione, JavaFX si è rivelata una scelta eccellente per la realizzazione delle interfacce grafiche della nostra applicazione. La combinazione delle sue funzionalità avanzate con l'utilizzo di Scene Builder, ha reso la programmazione delle interfacce utente particolarmente agevole e ha permesso di creare un'applicazione user-friendly. JavaFX è uno dei principali attori nel mercato delle applicazioni e si conferma come una scelta valida per la realizzazione di interfacce grafiche avanzate.



Tramite l'utilizzo di Scene Builder è possibile gestire, attraverso l'interfaccia grafica del tool, i file **.fxml* che abbiamo utilizzato per costruire l'intera interfaccia grafica. Ogni file comunicherà attraverso il corrispettivo Controller che gestirà le azioni e le funzioni scaturite dall'interazione con l'utente. Nello specifico, il progetto è strutturato in modo che esista una *MainView* in modo da implementare una *MenuBar* in modo che sarà presente in tutte le schermate dell'applicazione. Nella view dello storico per utilizzare la *TableView* in modo che racchiudesse lo stato di tutti le spedizioni, abbiamo pensato di creare una nuova classe Detail in modo da rappresentare i dettagli di un singolo collo. La classe utilizza la classe *SimpleStringProperty* di JavaFX per creare proprietà di stringa che possono essere utilizzate per leggere e modificare i dettagli del pacchetto mentre Il metodo *initialize*() della classe *HistoryController* viene utilizzato per inizializzare la tabella visualizzando i dettagli dei pacchetti. Utilizza i metodi forniti dalla

classe *PackageService* per recuperare i dati dal database e creare un *Vector* di oggetti *Detail*, che rappresentano i dettagli del pacchetto.

Di seguito la schermata rappresentante lo storico delle spedizioni:



Di seguito il diagramma UML delle classi Controller: