UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Rail Suite
Sistema di supervisione e pianificazione ferroviaria

Autori:
Leonardo Davio
Eros Pinzani

N° Matricola:
7072982
7030989

Corso:
Ingegneria del Software

Docente corso:
Enrico Vicario

Indice

1 Introduzione

1.1 Motivazione

L'applicazione ha lo scopo di facilitare e ottimizzare l'organizzazione dei treni sulle linee, la gestione del personale e tiene conto della pulizia e della manutenzione delle vetture effettuato in deposito o in stazioni autorizzate. L'obiettivo è quello di realizzare un'applicazione utilizzata dal personale attraverso la quale possano essere svolte operazioni fondamentali per avere un funzionamento corretto e agevolato del sistema ferroviario. Inoltre, l'applicazione si interfaccia con un database per poter salvare e gestire i dati garantendo un servizio rapido e sicuro.

1.2 Statement

Il sistema informatico di questo progetto è rivolto sia al personale viaggiante, fornendo una sezione personale con tutte le informazioni necessarie, sia al supervisore, che, se necessario, aggiorna i dati per garantire un servizio efficiente e accurato ai viaggiatori. Inoltre, il sistema gestisce la manutenzione e la pulizia delle vetture.
Diamo adesso una lista di termini tecnici con la relativa definizione che verranno usati nei paragrafi successivi:

- Vettura: entità base del convoglio con il compito di trasportare passeggeri
- Convoglio: insieme di vetture con la motrice.
- Corsa: detta anche servizio, è l'insieme delle stazioni in cui il convoglio ferma ad un determinato orario.
- Linea: insieme di tutte le stazioni comprese tra quella di partenza e quella di arrivo. A differenza della corsa definita per un singolo convoglio nella linea possono essere presenti stazioni in cui il treno non ferma.
- Supervisore: figura facente parte del personale con il compito di gestire linee, corse e convogli ma anche di accettare o meno richieste di manutenzione e pulizia, operazioni spiegate meglio negli statement successivi.

Questo sistema informatico gestisce le seguenti operazioni:

- L'operatore, accedendo alla sua personale sezione dell'applicazione, si occupa di:
  - Visualizzare la lista di convogli a cui è stato assegnato e le relative composizioni di carrozze in aggiunta ai dettagli della corsa a cui è associato il convoglio.
  - Segnalare la necessità di manutenzione di una singola vettura oppure di un insieme di esse attraverso una notifica al supervisore. In tal caso è suo compito direzionarlo verso un'officina solamente nel momento in

cui il convoglio in questione termina il suo servizio su una data linea. La durata in cui staziona in officina per le riparazioni è di sei ore e in quel lasso di tempo il convoglio non può essere utilizzato su nessuna linea.

- Richiedere, tramite un avviso al supervisore, la pulizia di un vagone, o un insieme di essi, che può essere effettuata in una qualunque stazione d'arrivo. La pulizia ha durata un'ora.

- La figura del supervisore, nella relativa pagina dell'applicazione, ha le seguenti possibilità:
  - Visualizzare l'insieme di notifiche inviate dai vari operatori con relativi dettagli e di conseguenza abilitare o meno un treno ad andare in officina per la manutenzione o a effettuare una pulizia. La decisione di accettare o meno tale notifica è legata al fatto che il convoglio in questione potrebbe dover necessariamente eseguire il servizio sulla linea a causa dell'assenza di un altro convoglio che può effettuare la sostituzione.
  - Visualizzare attraverso appositi filtri la lista di corse esistenti corrispondenti alle selezioni effettuate e vederne i dettagli. Da questa nuova sezione il supervisore può svolgere le seguenti operazioni: cambiare il convoglio associato alla corsa, modificare le carrozze associate al convoglio, cambiare l'operatore di turno.
  - Creare una nuova corsa andando ad assegnare operatore e convoglio ad una linea in un determinato orario.
  - Visualizzare la lista di convogli in una certa stazione scelta grazie ad un filtro e quindi creare ed eliminare un nuovo convoglio in tale stazione, ma anche aggiungere o rimuovere vetture da un convoglio scelto dalla lista.

- Le stazioni di testa della linea hanno una officina di manutenzione e le attrezzature di pulizia.

- Per quanto riguarda l'integrazione con il database verrà effettuata attraverso il pattern DAO e sarà utilizzato per memorizzare, rendere sicuri e più facilmente accessibili tutti i dati, ad esempio: nomi, cognomi e identificativi di tutto il personale, gli identificativi e le caratteristiche dei convogli e infine tutti i dati delle stazioni.

- L'applicazione *Rail Suite* contiene quindi tutte le informazioni essenziali per il corretto svolgimento del servizio e permette ai capotreni e supervisori di poter eseguire i rispettivi compiti in modo semplificato grazie ad un'interfaccia grafica semplice ed intuitiva.

1.3 Possibili sviluppi futuri

Alcuni possibili sviluppi futuri sono ad esempio la necessità di migliorare l'efficienza nel caso di aggiunta di un numero di linee molto elevato. Inoltre, i turni del personale sono da considerare non solo vincolati all'orario, ma anche in base alle festività, ferie e malattie. Infine, una possibile modifica è la possibilità di aggiungere più di un supervisore in modo che questi collaborino tra loro nel caso di una rete ferroviaria molto vasta.

2 Analisi dei requisiti

2.1 Metodi utilizzati

Il progetto è stato realizzato in **Java** ed è stato testato utilizzando Per quanto riguarda la fase di progettazione, i class diagram e gli use case sono uniformati allo standard **UML**. Il database utilizzato è **NeonDB**, un database online, scelto per avere i dati sincronizzati in modo automatico. La stesura del presente documento è stata realizzata tramite **Overleaf**.

2.2 Use Case Diagram

Per garantire un funzionamento ideale del software, enunciato nel paragrafo 1.2, è necessario che le varie operazioni siano gestite dal sistema che deve fare da tramite tra le figure coinvolte, dette utenti del software. Per capirne al meglio il funzionamento andiamo ad evidenziare le varie interazioni tra sistema e utenti. Per poter fare ciò utilizziamo gli use case diagrams. Il primo si concentra sulla figura del operatore mentre il secondo su quella del supervisore. Notiamo come attraverso questi diagrammi vengono messe in risalto le operazioni dirette tra utente e sistema ma anche eventuali interazioni che partono dal sistema e che l'utente potrebbe aspettare di ricevere.

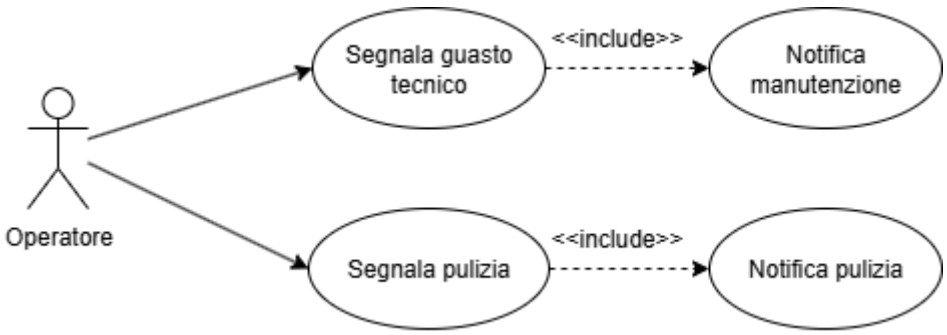

Figura 1: Use case diagram Operatore

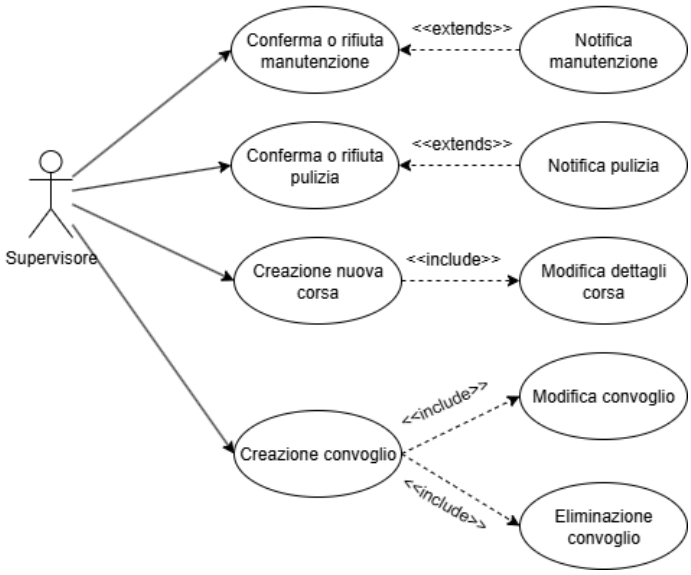

Figura 2: Use case diagram Supervisore

2.3 Templates

Attraverso l'utilizzo degli use case templates possiamo osservare in modo più dettagliato le operazioni che il sistema deve svolgere evidenziate nel paragrafo precedente con gli use case diagrams. In particolare andiamo a definire i seguenti templates:

UC	**Segnalazione pulizia**
Level	User goal
Description	L'operatore richiede la pulizia intensiva della vettura o di un insieme di esse.
Actors	Operatore
Pre-conditions	Il treno deve aver finito il servizio
Post-conditions	Viene generata la notifica di pulizia da mandare al supervisore.
Normal flow	1. Il treno conclude il servizio su una linea 2. L'operatore rileva un atto vandalico o della sporcizia su una vettura 3. L'operatore segnala quale vettura o insieme è stato vandalizzato o è sporco 4. L'operatore richiede la pulizia di tale vettura attraverso un pulsante 5. Il sistema genera la notifica di pulizia 6. Il sistema invia la notifica al supervisore

Tabella 1: Template della richiesta di pulizia

Lo use case template del caso di segnalazione manutenzione non viene mostrato in quanto il suo funzionamento è molto simile al caso appena descritto e si conclude con l'invio di una notifica al supervisore.

Vediamo adesso altri templates dei casi d'uso che riguardano il supervisore:

UC	**Conferma pulizia**
Level	User goal
Description	Il supervisore, dopo aver ricevuto la notifica, conferma o meno la pulizia
Actors	• supervisore (primary) • operatore (secondary)
Pre-conditions	L'operatore deve aver fatto richiesta di pulizia.
Post-conditions	Il treno viene mandato in pulizia o viene rimesso in servizio.
Normal flow	1. Il supervisore riceve la notifica di richiesta di pulizia 2. Il supervisore conferma la pulizia 3. Il sistema manda il treno in pulizia
Alternative flow	2a. Il supervisore osserva l'assenza di un treno sostitutivo per la linea 3a. Il supervisore rifiuta la pulizia 4a. Il sistema rimette il treno in servizio sulla linea

Tabella 2: Template della conferma di pulizia

Il caso di conferma manutenzione, come nel caso precedente della segnalazione, è molto simile al template appena descritto e quindi viene omesso.

UC	**Creazione nuova corsa**
Level	User goal
Description	Il supervisore richiede al sistema di generare una nuova corsa
Actors	• Supervisore (primary) • Sistema (secondary)
Pre-conditions	Necessità di una nuova corsa su una linea
Post-conditions	Il sistema inserisce nel database la nuova corsa appena creata
Normal flow	1. Il supervisore accede all'apposita sezione dell'applicazione 2. Il supervisore seleziona la linea su cui creare la nuova corsa 3. Il supervisore seleziona poi la stazione, la data, l'orario di partenza 4. Il supervisore seleziona l'operatore da una lista di operatori disponibili sulla base dei filtri precedenti 5. Il supervisore seleziona adesso il tipo del convoglio e sceglie quale assegnare alla corsa in base alla disponibilità 6. Il supervisore conferma la creazione della corsa 7. Il sistema aggiunge la nuova corsa al database
Issues	Possibile assenza del operatore nella stazione di partenza o con disponibilità di ore in modo da non sforare l'orario lavorativo

Tabella 3: Template della creazione di una nuova corsa

UC	**Creazione nuovo convoglio**
Level	User goal
Description	Il supervisore richiede al sistema di generare un nuovo convoglio
Actors	• Supervisore (primary) • Sistema (secondary)
Pre-conditions	Necessità di un nuovo convoglio
Post-conditions	Il sistema inserisce n el database il nuovo convoglio appena creato
Normal flow	1. Il supervisore seleziona la stazione in cui vuole che venga creato il nuovo convoglio 2. Il supervisore seleziona l'apposito pulsante per aprire la sezione per la creazione dei convogli 3. Il supervisore seleziona poi il tipo e il modello di vetture che devono comporre il convoglio 4. Il supervisore sceglie dalla lista di vetture disponibili, corrispondenti ai filtri, quale desidera che facciano parte del nuovo convoglio 5. Il supervisore conferma la scelta 6. Il sistema inserisce nel database il nuovo convoglio appena creato e lo associa alla stazione selezionata in precedenza

Tabella 4: Template della creazione di un nuovo convoglio

I casi di modifica ed eliminazione convoglio sono per certi aspetti simili al template appena descritto ma anche molto semplici ed intuitivi, per cui non vengono trattati di seguito.

2.4 Navigation diagram

Il navigation diagram va a rappresentare in modo grafico la struttura dell'applicazione sviluppata per questo progetto. Di seguito, nella figura 3, il diagramma completo che esplica l'organizzazione delle pagine dell'interfaccia grafica.

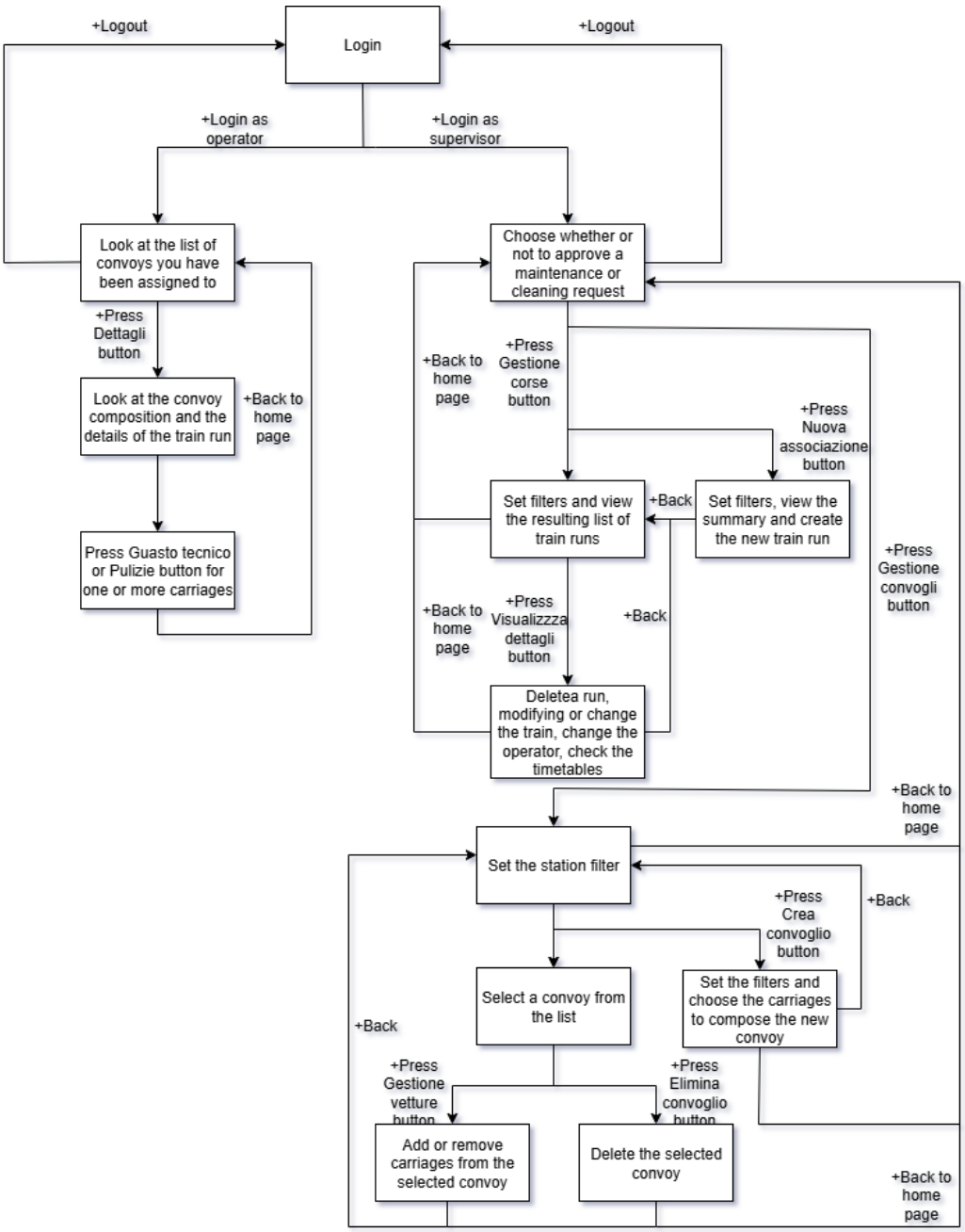

Figura 3: Navigation diagram

3 Progettazione

3.1 Architettura del software

La logica che è stata scelta per la struttura del progetto è Business Logic + Domain Model + DAO, in cui:

- **Business Logic**: si occupa di gestire la manipolazione dei dati e contiene tutti i file necessari per l'interfaccia grafica.
- **Domain Model**: è la rappresentazione sotto forma di classi delle entità e dei concetti utilizzati nel progetto.
- **DAO**: è il design pattern tramite il quale si va a connettere il progetto con il database per avere persistenza dei dati.

3.2 Class diagram

Di seguito sono riportati cinque collegamenti al cui interno sono presenti i diagrammi delle classi divisi per package e uno completo. Attraverso questi è possibile avere una vista grafica della struttura di tutto il progetto e possono essere osservate le relazioni tra le classi e i pacchetti.

- UML completo
- businessLogic
- dao
- mapper
- domain

4 Implementazione

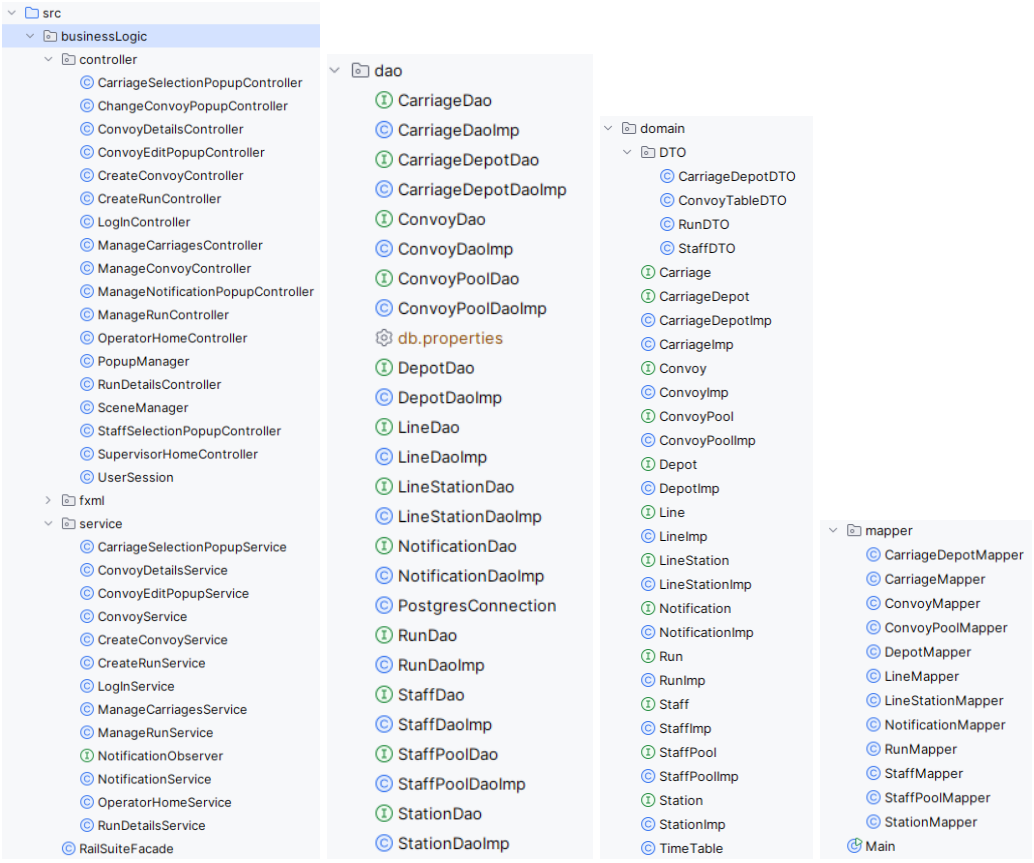

Figura 4: File del progetto

Nei paragrafi a venire verranno analizzati nel dettaglio i packages e le classi del progetto e verranno analizzati i metodi contenuti in esse che necessitano di un approfondimento maggiore.

4.1 Domain

Questo package, la cui funzionalità è stata descritta nel paragrafo 3.1, è composto dalle seguenti classi:

- CarriageImp
- CarriageDepotImp
- ConvoyImp
- ConvoyPoolImp
- DepotImp

- LineImp
- LineStationImp
- NotificationImp
- RunImp
- StaffImp
- StaffPoolImp
- StationImp
- TimeTable

Dato che il package è protetto ogni singola classe implementata, contraddistinta dal suffisso **Imp**, ha una corrispettiva classe **interfaccia** che permette alla relativa implementazione di comunicare con gli altri package, anch'essi protetti. All'interno di questo package ne troviamo un altro: **DTO**, ovvero Data Transfer Object, che contiene le seguenti classi:

- CarriageDepotDTO
- ConvoyTableDTO
- RunDTO
- StaffDTO

Queste classi sono progettate per trasferire i dati tra i livelli del progetto, in particolare il loro compito è quello di fungere da contenitori per dati, andando ad escludere la business logic e permettere il trasporto di informazioni in modo efficiente. Grazie ad esse quindi è possibile evitare l'esposizione diretta degli elementi del package domain.

4.1.1 CarriageImp

Questa classe è l'implementazione concreta dell'interfaccia **Carriage** e definisce un singolo vagone nel domain model dell'applicazione.
Gli attributi che compongono la classe sono tre di tipo intero (*id*, *yearProducted* e *capacity*), due di tipo stringa (*model* e *modelType*) e uno di tipo Integer (*idConvoy*). *id* rappresenta l'identificativo della carrozza, *model*, *modelType*, *yearProducted* e *capacity* identificano rispettivamente il modello, il tipo, l'anno di produzione e la capacità della singola carrozza. *idConvoy* rappresenta invece l'eventuale convoglio al quale il vagone è stato assegnato.
Da notare come tutti gli attributi, fatta eccezione per *idConvoy*, sono marcati final in quanto rappresentano caratteristiche legate al vagone che non variano nel tempo. Al contrario *idConvoy* non è final proprio per il fatto che lo stesso vagone può essere

assegnato a convogli differenti in diversi momenti.
I metodi contenuti in questa classe sono i getter per gli attributi sopra elencati e un singolo setter per l'attributo che indica a quale convoglio la carrozza è associata.

4.1.2 CarriageDepotImp

Questa classe è l'implementazione concreta dell'interfaccia **CarriageDepot** e definisce l'associazione di un vagone con il suo stato all'interno di un deposito.
Gli attributi contenuti nella classe sono due di tipo intero (*idDepot* e *idCarriage*) che rappresentano rispettivamente il vagone e il relativo deposito in cui è situato, due di tipo Timestamp (*timeEntered* e *timeExited*) che identificano gli orari e i giorni di ingresso e uscita del vagone dal deposito e infine un attributo di tipo StatusOfCarrige (*statusOfCarriage*) che indica lo stato semantico del vagone all'interno del deposito. Quest'ultimo tipo è stato definito all'interno dell'interfaccia CarriageDepot e si tratta di un enum i cui valori sono: *CLEANING*, *MAINTENENCE* e *AVAILABLE*, indicano quindi se un vagone è in deposito per pulizia oppure per manutenzione e anche quando è nuovamente avviabile per un nuovo servizio.
In questa classe gli unici attributi final sono *idDepot* e *idCarriage*, gli altri sono invece dinamici e variano nel tempo. I metodi in questa classe sono getter e setter.

4.1.3 ConvoyImp

Questa classe è l'implementazione concreta dell'interfaccia **Convoy** e definisce un convoglio all'interno del domain model.
Gli attributi sono uno di tipo intero (*idConvoy*) che identifica il convoglio e l'altro è una lista di Carriage che rappresenta l'insieme delle carrozze che compongono il convoglio. Entrambi gli attributi sono marcati come final ma questo non impedisce il cambio degli elementi della lista.
Oltre ai getter sono presenti anche i metodi per la modifica della lista riportati nella seguente figura 5.

```java
@Override
public boolean removeCarriage(Carriage carriage) {
    if (carriages.contains(carriage)) {
        carriages.remove(carriage);
        return true;
    }
    return false;
}

@Override
public boolean addCarriage(Carriage carriage) {
    if (!carriages.contains(carriage)) {
        carriages.add(carriage);
        return true;
    }
    return false;
}
```

Figura 5: removeCarriage e addCarriage

Da notare come i metodi *removeCarriage* e *addCarriage* applicano un controllo di presenza tramite *contains(...)* in modo tale da evitare carrozze duplicate nella lista che rappresenta il convoglio.
Inoltre è presente anche il metodo *convoySize* il cui compito è quello di contare il numero di carrozze contenute nella lista.

4.1.4 ConvoyPoolImp

Questa classe è l'implementazione concreta dell'interfaccia **ConvoyPool** e definisce lo stato di un convoglio all'interno di una stazione.
Gli attributi della classe sono due interi (*idConvoy* e *idStation*) che identificano il convoglio e la stazione in cui è situato e uno di tipo ConvoyStatus (*convoyStatus*) che indica lo stato semantico del convoglio all'interno della stazione. Quest'ultimo tipo è definito nella relativa interfaccia ConvoyPool ed è un enum con valori: *DEPOT*, *ON_RUN* e *WAITING*, indicano quindi se un convoglio in una stazione è nel relativo deposito, fermo per la discesa e la salita dei passeggeri per poi ripartire oppure in attesa di essere assegnato a una nuova corsa.
Importante osservare come ad essere marcato final è solamente l'identificativo del convoglio poiché la stazione non è associata in modo univoco al treno che, nella sua corsa, visiterà svariate stazioni.
I metodi della classe sono i vari getter e setter necessari per ottenere e aggiornare i dati.

4.1.5 DepotImp

Questa classe è l'implementazione concreta dell'interfaccia **Depot** e definisce un deposito ferroviario in cui vanno i vagoni in caso di necessità di pulizia o manuten-

zione.
Gli attributi che compongono questa classe sono uno di tipo intero (*idDepot)* che rappresenta l'identificativo del deposito in questione e l'altro è una lista di elementi di tipo *CarriageDepot* (*carriages*) che raffigura l'insieme dei vagoni presenti nel deposito.
In questa classe, oltre ai metodi getter, troviamo anche quelli per aggiungere o rimuovere carrozze dalla lista.

4.1.6 LineImp

Questa classe è l'implementazione concreta dell'interfaccia **Line** e definisce una linea ferroviaria.
Troviamo i seguenti attributi: tre di tipo intero (*idLine*, *idFirstStation* e *idLastStation*) che rappresentano rispettivamente gli identificativi della linea, della stazione di testa e di quella di coda; ci sono poi altri tre attributi di tipo stringa (*lineName*, *firstStationLocation* e *lastStationLocation*) che indicano il nome della linea e i nomi delle stazioni di partenza e di arrivo.
Gli attributi sono tutti marcati final e i metodi sono tutti getter.

4.1.7 LineStationImp

Questa classe è l'implementazione concreta dell'interfaccia **LineStation** e definisce una singola stazione associata a una certa linea.
Qui troviamo due attributi di tipo intero (*stationId* e *order*) che identificano una delle stazioni appartenenti alla linea e la posizione di questa nella lista delle stazioni che compongono la linea, poi troviamo un attributo di tipo Duration (*timeToNextStation*) che rappresenta il tempo di percorrenza necessario per passare dalla stazione presa in considerazione alla successiva appartenente alla stessa linea.
Gli attributi sono tutti marcati final e i metodi sono tutti getter.

4.1.8 NotificationImp

Questa classe è l'implementazione concreta dell'interfaccia **Notification** e definisce una singola notifica generata dal sistema necessaria per il design pattern **observer** che collega l'operatore al supervisore.
Questa è la notifica che viene generata nel momento in cui un operatore decide di premere l'apposito pulsante per segnalare la necessità di manutenzione o di pulizia di una carrozza ed è la stessa notifica che viene quindi mostrata nella tabella della pagina iniziale dell'applicazione se effettuato l'accesso come supervisore.
La classe è composta da tre attributi interi (*idCarriage*, *idConvoy* e *idStaff*) che rappresentano gli identificativi di carrozza, convoglio a cui la carrozza appartiene e operatore che ha fatto la segnalazione. Inoltre è presente un attributo di tipo Timestamp (*dateTimeOfNotification*) che indica l'orario in cui è stata generata la notifica. Ci sono poi cinque attributi di tipo stringa (*typeOfCarriage*, *typeOfNotification*, *staffName*, *staffSurname* e *status*) che definiscono il modello della carrozza,

la tipologia della segnalazione, il nome e il cognome dell'operatore che la ha inviata e lo status di tale notifica: questo può essere inviata, negata o approvata.
Tutti gli attributi sono marcati final e i metodi della classe sono solo getter.

4.1.9 RunImp

Questa classe è l'implementazione concreta dell'interfaccia **Run** e definisce una corsa ferroviaria, ovvero un singolo viaggio di un convoglio su una determinata linea con a bordo uno specifico operatore.
La classe è composta da cinque attributi di tipo intero (*idLine*, *idConvoy*, *idStaff*, *idFirstStation* e *idLastStation*) che indicano gli identificativi necessari per definire la corsa, quattro attributi di tipo stringa (*lineName*, *staffNameSurname*, *firstStationName* e *lastStationName*) che raffigurano i nomi della linea, dell'operatore e delle stazioni di testa e di coda. Troviamo poi due attributi di tipo Timestamp (*timeDeparture* e *timeArrival)* che indicano gli orari di partenza e di arrivo della corsa e infine un attributo di tipo RunStatus (*status*) che indica lo stato della corsa. Quest'ultimo attributo è definito nell'interfaccia Run ed è un enum con valori: *RUN*, *BEFORE_RUN* e *AFTER_RUN*, ovvero indica se la corsa deve partire, è in atto o è già conclusa. Questa condizione viene calcolata in modo dinamico dal costruttore della classe che va a confrontare l'ora attuale con l'orario di partenza e quello di arrivo.
Per quanto riguarda i metodi sono tutti dei getter.

4.1.10 StaffImp

Questa classe è l'implementazione concreta dell'interfaccia **Staff** e definisce un singolo membro del personale all'interno del domain model.
Gli attributi della classe sono uno di tipo intero (*idStaff*) che identifica la singola persona, cinque di tipo stringa (*name*, *surname*, *address*, *email* e *password*) che rappresentano i dati del membro del personale e infine un attributo di tipo TypeOfStaff (*typeOfStaff*) che indica il ruolo del lavoratore. Quest'ultimo tipo è definito nella relativa interfaccia Staff ed è un enum con valori: *OPERATOR* e *SUPERVISOR*, indicano quindi se un membro del personale è un operatore viaggiante, ovvero che lavora a bordo del convoglio, oppure un supervisore. I compiti di queste due figure sono stati ben descritti nel paragrafo 1.2
Tutti gli attributi di questa classe sono final poiché rappresentano la persona e non variano nel tempo.
I metodi della classe sono getter.

4.1.11 StaffPoolImp

Questa classe è l'implementazione concreta dell'interfaccia **StaffPool** e definisce la presenza del personale e il relativo stato all'interno di una stazione, ma anche la sua assegnazione a un convoglio.
La classe è composta da tre attributi di tipo intero (*idStaff*, *idStation* e *idConvoy*)

che rappresentano gli identificativi del personale, della stazione e del convoglio, da due di tipo TimeStamp (*shiftStart* e *shiftEnd*) che indicano gli orari di inizio e fine turno del personale e da un attributo di tipo ShiftStatus (*shiftStatus*) che identifica la reperibilità di un membro del personale in una data stazione. Quest'ultimo tipo è definito nella relativa interfaccia StaffPool ed è un enum con valori: *AVAILABLE*, *ON_RUN* e *RELAX*, indicano quindi se un membro del personale è avviabile per un nuovo turno, in servizio oppure ha finito il proprio turno e sta aspettando il tempo necessario per iniziarne uno nuovo.
I metodi contenuti all'interno della classe sono getter e setter.

4.1.12 StationImp

Questa classe è l'implementazione concreta dell'interfaccia **Station** e definisce una stazione fisica all'interno del domain model dell'applicazione.
Tutti i suoi attributi sono marcati final e sono due di tipo intero (*idStation* e *numBins*) che raffigurano l'identificativo della stazione e il numero di binari che la costituisce, due di tipo stringa (*location* e *serviceDescription*) che rappresentano la posizione geografica della stazione e una breve descrizione dei servizi che offre e infine un attributo di tipo booleano (*isHead*) che indica se la stazione è un capolinea.
I metodi sono tutti getter.

4.1.13 TimeTable

Questa classe definisce la tabella oraria associata a una linea, indica le stazioni appartenenti a tale linea e i relativi orari di arrivo e partenza.
Gli attributi sono uno di tipo intero (*idLine*) che indica la linea in questione e una lista di tipo StationArrAndDep (*stationArrAndDepList*) che rappresenta la lista delle fermate con associate le informazioni di orario.
Questa classe contiene poi, oltre ai metodi getter, una classe annidata statica, StationArrAndDep, che contiene un attributo di tipo intero (*idStation*) e tre attributi di tipo stringa (*stationName*, *arriveTime* e *departureTime*, ovvero attributi necessari per associare a una stazione gli orari di arrivo e di partenza.

4.2 DAO

Il **DAO**, ovvero Data Access Object, è la parte dell'applicazione che si occupa delle interazioni con il database, in particolare va a incapsulare le query SQL e a gestire le connessioni. I suoi scopi principali sono quelli di effettuare le ricerche all'interno del database e di gestire le operazioni CRUD: Create, Read, Update, Delete.
In questo progetto, per rispettare il principio della separazione delle responsabilità, la conversione delle righe del ResultSet in oggetti del dominio è stata delegata al design pattern mapper, analizzato nel paragrafo 4.3. In questo modo il DAO si occupa esclusivamente di costruire ed eseguire le query.
In questo package sono presenti tante classi quante quelle del package domain in modo tale che ogni classe si occupi delle query specifiche che la riguardano. Anche

qui ogni classe che termina con il suffisso *Imp* ha una sua interfaccia per poter comunicare con il package che si occupa della business logic e poter così mantenere il pacchetto protetto. Di seguito la lista completa delle implementazioni delle classi che compongono il DAO:

- CarriageDaoImp
- CarriageDepotDaoImp
- ConvoyDaoImp
- ConvoyPoolDaoImp
- DepotDaoImp
- LineDaoImp
- LineStationDaoImp
- NotificationDaoImp
- PostgresConnection
- RunDaoImp
- StaffDaoImp
- StaffPoolDaoImp
- StationDaoImp

Adesso, nella pagina successiva, viene analizzata nel dettaglio la classe ConvoyDaoImp per osservare la struttura di un DAO, le altre classi, avendo una struttura complessivamente simile, non verranno approfondite.
Come è possibile vedere nella seguente figura 6, la classe presenta in alto tutte le query necessarie, queste possono essere semplici oppure più complesse come nel caso di *convoyForNewRunQuery*.

```
private static final String selectConvoyQuery =
        "SELECT id_convoy FROM convoy WHERE id_convoy = ?";
private static final String selectCarriagesByConvoyIdQuery =
        "SELECT id_carriage, model, model_type, year_produced, capacity, id_convoy " +
                "FROM carriage WHERE id_convoy = ?";
private static final String selectAllConvoyIdsQuery =
        "SELECT id_convoy FROM convoy";
private static final String deleteConvoyQuery =
        "DELETE FROM convoy WHERE id_convoy = ?";
private static final String updateCarriageConvoyQuery =
        "UPDATE carriage SET id_convoy = ? WHERE id_carriage = ?";
private static final String removeCarriageFromConvoyQuery =
        "UPDATE carriage SET id_convoy = NULL WHERE id_carriage = ?";
private static final String insertConvoyQuery =
        "INSERT INTO convoy DEFAULT VALUES RETURNING id_convoy";
private static final String convoyForNewRunQuery = """
        SELECT
          c.id_convoy,
          cp.id_station,
          cp.status,
          ca.id_carriage,
          ca.model,
          ca.model_type,
          ca.year_produced,
          ca.capacity
        FROM convoy_pool cp
        JOIN convoy c ON c.id_convoy = cp.id_convoy
        LEFT JOIN carriage ca ON ca.id_convoy = c.id_convoy
        WHERE cp.id_station = ?
          AND cp.status IN ('DEPOT', 'WAITING')
          AND c.id_convoy NOT IN (
              SELECT r.id_convoy
              FROM run r
              WHERE r.time_departure <= ?::timestamp
                AND r.time_arrival >= ?::timestamp
                AND r.id_line = ?
          )
          AND c.id_convoy NOT IN (
              SELECT ca2.id_convoy
              FROM carriage ca2
              JOIN carriage_depot cd ON ca2.id_carriage = cd.id_carriage
              WHERE cd.status_of_carriage = 'MAINTENANCE'
                AND cd.time_exited IS NULL
          )
        ORDER BY c.id_convoy, ca.id_carriage;""";
```

Figura 6: Query della classe ConvoyDaoImp

Vediamo ora una serie di figure che riportano alcuni metodi della classe ConvoyDaoImp per far vedere che vengono gestite le operazioni CRUD.

```
public Convoy createConvoy(List<Carriage> carriages) throws SQLException {
    int generatedId;
    try (
            java.sql.Connection conn = PostgresConnection.getConnection();
            java.sql.PreparedStatement insertConvoyStmt = conn.prepareStatement(insertConvoyQuery)
    ) {
        try (java.sql.ResultSet rs = insertConvoyStmt.executeQuery()) {
            if (rs.next()) {
                generatedId = mapper.ConvoyMapper.getConvoyId(rs);
            } else {
                throw new SQLException("Failed to retrieve generated convoy id");
            }
        }
        if (carriages != null && !carriages.isEmpty()) {
            try (java.sql.PreparedStatement updateCarriageStmt = conn.prepareStatement(updateCarriageConvoyQuery)) {
                for (Carriage carriage : carriages) {
                    mapper.ConvoyMapper.setConvoyAndCarriageId(updateCarriageStmt, generatedId, carriage);
                    updateCarriageStmt.addBatch();
                }
                int[] results = updateCarriageStmt.executeBatch();
                for (int res : results) {
                    if (res == 0) {
                        throw new SQLException("Failed to update carriage with id_convoy: " + generatedId);
                    }
                }
            }
        }
    } catch (SQLException e) {
        throw new SQLException("Error creating new convoy and updating carriages", e);
    }
    return domain.Convoy.of(generatedId, carriages);
}
```

Figura 7: Metodo createConvoy

La figura 7 riporta il metodo per la creazione di un convoglio all'interno del database. Per poterlo fare esegue una query di inserimento per generare un nuovo record nella tabella dei convogli e recupera l'identificativo generato. Successivamente, se viene fornita una lista di carrozze, aggiorna ciascuna di esse associandole al nuovo convoglio tramite una query di aggiornamento. Infine, viene restituito il nuovo oggetto di tipo Convoy.

```
public ConvoyDetailsService.ConvoyDetailsRaw selectConvoyDetailsById(int id) throws SQLException {
    try (
            java.sql.Connection conn = PostgresConnection.getConnection();
            java.sql.PreparedStatement stmt = conn.prepareStatement(
                    sql: "SELECT ca.id_carriage, ca.model, ca.model_type, ca.year_produced, ca.capacity, " +
                    "l.id_line, l.name AS line_name, s.id_station, s.location AS station_name, " +
                            "ls.station_order, " +
                    "r.time_departure, r.time_arrival, stf.name AS staff_name, stf.surname AS staff_surname, " +
                            "stf.id_staff, " +
                    "r.id_first_station, r.id_last_station, s2.location AS departure_station, " +
                            "s3.location AS arrival_station " +
                    "FROM convoy c " +
                    "LEFT JOIN carriage ca ON ca.id_convoy = c.id_convoy " +
                    "LEFT JOIN run r ON r.id_convoy = c.id_convoy " +
                    "LEFT JOIN staff stf ON stf.id_staff = r.id_staff " +
                    "LEFT JOIN line l ON l.id_line = r.id_line " +
                    "LEFT JOIN line_station ls ON ls.id_line = l.id_line " +
                    "LEFT JOIN station s ON s.id_station = ls.id_station " +
                    "LEFT JOIN station s2 ON s2.id_station = r.id_first_station " +
                    "LEFT JOIN station s3 ON s3.id_station = r.id_last_station " +
                    "WHERE c.id_convoy = ? " +
                    "ORDER BY ls.station_order"
            )
    ) {
        mapper.ConvoyMapper.setConvoyId(stmt, id);
        try (java.sql.ResultSet rs = stmt.executeQuery()) {
            return mapper.ConvoyMapper.toConvoyDetailsRaw(rs, id);
        }
    }
}
```

Figura 8: Metodo selectConvoyDetailsById

La figura 8 riporta il metodo per recuperare dal database i dati relativi a un convoglio specifico. Attraverso una query che coinvolge più tabelle è possibile reperire tutte le informazioni legate al convoglio, quali ad esempio le sue carrozze, le stazioni attraversate e le informazioni sul personale.

```
public boolean removeConvoy(int id) throws SQLException {
    try (
            java.sql.Connection conn = PostgresConnection.getConnection();
            java.sql.PreparedStatement updateCarriageStmt =
                    conn.prepareStatement( sql: "UPDATE carriage SET id_convoy = NULL WHERE id_convoy = ?");
            java.sql.PreparedStatement deleteConvoyStmt =
                    conn.prepareStatement(deleteConvoyQuery)
    ) {
        mapper.ConvoyMapper.setConvoyId(updateCarriageStmt, id);
        updateCarriageStmt.executeUpdate();
        mapper.ConvoyMapper.setConvoyId(deleteConvoyStmt, id);
        int affectedRows = deleteConvoyStmt.executeUpdate();
        return affectedRows > 0;
    } catch (SQLException e) {
        throw new SQLException("Error removing convoy with id: " + id, e);
    }
}
```

Figura 9: Metodo removeConvoy

La figura 9 riporta il metodo per eliminare un convoglio dal database. Prima di effettuare la cancellazione, il metodo aggiorna tutte le carrozze associate, impostando il loro campo *id_convoy* a NULL per rimuovere il collegamento e, alla fine, restituisce un valore booleano per indicare se la rimozione ha avuto successo.

Dalla pagina successiva sono riportate figure e descrizioni di altri metodi del package DAO che meritano un approfondimento.

```
private static final String updateConvoyStatus = """
        UPDATE convoy_pool cp
        SET status = CASE
            WHEN EXISTS (
                SELECT 1
                FROM carriage c
                JOIN carriage_depot cd ON c.id_carriage = cd.id_carriage
                WHERE c.id_convoy = cp.id_convoy
                  AND cd.status_of_carriage IN ('MAINTENANCE', 'CLEANING')
                  AND cd.time_exited IS NULL
            ) THEN 'DEPOT'
            WHEN NOT EXISTS (
                SELECT 1
                FROM run r
                WHERE r.id_convoy = cp.id_convoy
                  AND now() BETWEEN r.time_departure AND r.time_arrival
            ) THEN 'WAITING'
            ELSE 'ON_RUN'
        END
        WHERE cp.id_station = ?
        """;
public List<ConvoyTableDTO> checkConvoyAvailability(int idStation) throws SQLException {
    List<ConvoyTableDTO> availableConvoys;
    try (Connection conn = PostgresConnection.getConnection()) {
        try (PreparedStatement psUpdate = conn.prepareStatement(updateConvoyStatus)) {
            mapper.ConvoyPoolMapper.setIdStation(psUpdate, idStation);
            psUpdate.executeUpdate();
        }
        try {
            availableConvoys = getConvoyTableDataByStation(idStation);
        } catch (SQLException e) {
            throw new SQLException("Error retrieving convoy table data", e);
        }
    } catch (SQLException e) {
        throw new SQLException("Error checking convoy availability", e);
    }
    return availableConvoys;
}
```

Figura 10: Metodo checkConvoyAvailability

Nella figura 10 è riportato il metodo *checkConvoyAvailability*. Questo è interessante perché mostra il caso di operazione Update del paradigma CRUD, infatti, oltre a recuperare i dati, il metodo effettua un aggiornamento dello stato dei convogli presenti in una determinata stazione.
In particolare *checkConvoyAvailability* riceve in input l'identificativo di una stazione e va ad aggiornare lo stato di tutti i convogli associati a tale stazione attraverso la query *updateConvoyStatus*. Quest'ultima va infatti a modificare il campo *status* nella tabella *convoy_pool*.
Una volta effettuato l'aggiornamento, viene chiamato il metodo *getConvoyTableDataByStation* il quale si occupa di recuperare e restituire la lista aggiornata dei convogli presenti nella stazione, insieme a tutte le informazioni necessarie. Ciò

viene fatto attraverso una query di tipo SELECT che unisce le informazioni sui convogli presenti nella stazione specificata; tali dati vengono poi mappati in oggetti DTO per essere utilizzati in seguito.

```
private static final String findTimeTableForRunSQL = """
        SELECT ls.*, s.location
        FROM line_station ls
        JOIN station s ON ls.id_station = s.id_station
        WHERE ls.id_line = ?
        ORDER BY CASE
            WHEN ? = (SELECT id_station FROM line_station WHERE id_line = ?
                      ORDER BY station_order ASC LIMIT 1) THEN ls.station_order
                      ELSE -ls.station_order
        END""";
public List<TimeTable.StationArrAndDep> findTimeTableForRun(int idLine, int idStartStation,
        String departureTime) throws SQLException {
    try (Connection conn = PostgresConnection.getConnection();
         PreparedStatement stmt = conn.prepareStatement(findTimeTableForRunSQL)) {
        mapper.LineStationMapper.setFindTimeTableForRunParams(stmt, idLine, idStartStation);
        ResultSet rs = stmt.executeQuery();
        List<LineStationRow> rows = new ArrayList<>();
        while (rs.next()) {
            int idStation = rs.getInt( columnLabel: "id_station");
            String location = rs.getString( columnLabel: "location");
            int stationOrder = rs.getInt( columnLabel: "station_order");
            String intervalStr = rs.getString( columnLabel: "time_to_next_station");
            Duration d = mapper.LineStationMapper.parseDurationFromPgInterval(intervalStr);
            rows.add(new LineStationRow(idStation, location, stationOrder, d));
        }
        if (rows.isEmpty()) {
            return Collections.emptyList();
        }
        LocalTime depPrev = LocalTime.parse(departureTime);
        List<TimeTable.StationArrAndDep> result = new ArrayList<>();
        boolean forward = rows.getFirst().stationOrder == 1;
        if (!forward) {
            for (int i = 0; i < rows.size(); i++) {
                LineStationRow r = rows.get(i);
                String arr, dep;
                if (i == 0) {
                    arr = "------";
                    dep = depPrev.toString();
                } else {
                    Duration segment = rows.get(i).timeToNext;
                    if (segment == null) {
                        System.out.println("Warning: time_to_next_station is NULL for station "
                                + rows.get(i).idStation + " — assuming Duration.ZERO");
                        segment = Duration.ZERO;
                    }
                    LocalTime arrCurr = depPrev.plus(segment);
                    arr = arrCurr.toString();
                    if (i < rows.size() - 1) {
                        depPrev = arrCurr.plusMinutes( minutesToAdd: 1);
                        dep = depPrev.toString();
                    } else {
                        dep = "------";
                    }
                }
                result.add(new TimeTable.StationArrAndDep(r.idStation, r.location, arr, dep));
            }
        } else {...}
        return result;
    }
}
```

Figura 11: Metodo findTimeTableForRun

Nella figura 11 è raffigurato il metodo *findTimeTableForRun*, interessante poiché mostra come sia possibile ricostruire, partendo da dati su più tabelle, una sequenza temporale di arrivi e partenze, calcolando gli orari dinamicamente.
Nel dettaglio il metodo riceve in input gli identificativi di una linea e di una stazione di partenza e l'orario di tale partenza. Poi, attraverso una query, recupera tutte le stazioni associate alla linea nell'ordine del percorso. Dopo va a calcolare l'orario di arrivo e di partenza per ogni stazione, aggiornando l'orario in base all'intervallo di tempo che intercorre tra una stazione e la successiva, tenendo conto del fatto che tra arrivo e partenza dalla stessa stazione deve essere atteso un minuto. Il risultato è una lista di oggetti DTO contenenti i dati appena calcolati.
Interessante notare come le variabili *arr*, in caso di una stazione di testa, e *dep*, in caso di stazione di coda, abbiano come valore una serie di trattini in quanto quei valori non esistono.
Nella parte finale è presente una condizione *else* che è stata compressa e non riportata nella figura poiché contiene la logica inversa ma molto simile per il calcolo degli orari in caso di percorrenza della linea nella direzione opposta.

4.3 Mapper

Il package Mapper è un insieme di classi che implementano il design pattern **Data Mapper**, il cui compito è quello di convertire una riga del ResultSet ottenuto da una query SQL, ricavata da una delle classi del DAO, in un oggetto del dominio corrispondente. Attraverso questo design pattern è così possibile mantenere separata la business logic da quella di accesso ai dati e avere quindi una migliore manutenibilità del codice.
Il package in questione è composto dalle seguenti classi:

- CarriageDepotMapper
- CarriageMapper
- ConvoyMapper
- ConvoyPoolMapper
- DepotMapper
- LineMapper
- LineStationMapper
- StaffMapper
- StaffPoolMapper
- StationMapper

Come si può notare è presente un mapper per ogni classe del package domain, questo per poter separare chiaramente la logica di conversione tra i ResultSet provenienti dal DAO e gli oggetti del dominio poiché ogni entità ha attributi, tipi di dato e regole di conversione specifiche che necessitano di una gestione separata dalle altre.
Prendiamo ora ad esempio la classe CarriageMapper, nella seguente figura 12, per poter analizzare nel dettaglio la struttura dei Data Mapper.

```
public class CarriageMapper {
    public static Carriage toDomain(ResultSet rs) throws SQLException {
        Integer idConvoy = null;
        try {
            if (rs.getObject( columnLabel: "id_convoy") != null) {
                idConvoy = rs.getInt( columnLabel: "id_convoy");
            }
        } catch (SQLException | IllegalArgumentException e) {
        }
        return Carriage.of(
            rs.getInt( columnLabel: "id_carriage"),
            rs.getString( columnLabel: "model"),
            rs.getString( columnLabel: "model_type"),
            rs.getInt( columnLabel: "year_produced"),
            rs.getInt( columnLabel: "capacity"),
            idConvoy
        );
    }
}
```

Figura 12: CarriageMapper

Osserviamo che i metodi necessari al design pattern sono due: *toDomain* e *Carriage.of*. Il primo metodo trasforma la riga di un ResultSet in un oggetto di dominio, andandone a delegare la creazione al secondo metodo.
La parte più rilevante da osservare è la condizione if nel metodo *toDomain*: questa gestisce il valore dell'attributo *idConvoy* che, come visto nel paragrafo 4.1.1, può avere valore nullo nel database; nel dettaglio il mapper verifica se l'attributo è nullo con *rs.getObject("idConvoy")* e in caso negativo ottiene il valore con *rs.getInt("idConvoy")*, altrimenti assegna *null* come valore dell'attributo.
Gli altri campi vengono mappati, senza necessità di controlli, in modo diretto tramite *rs.getInt(...)* e *rs.getString(...)* e passati insieme a *idConvoy* a *Carriage.of(...)* che crea l'oggetto di dominio.

Le altre classi non verranno analizzate nello specifico poiché il loro comportamento è simile a quello appena osservato.

4.4 Business Logic

La Business Logic è il package centrale del progetto, contiene tutte le regole e le operazioni grazie alle quali possono essere svolte le funzionalità richieste dall'utente.
In questo progetto la business logic è stata organizzata in tre package:

- **fxml**: contiene i file che rappresentano la definizione delle interfacce grafiche dell'applicazione. Sono file scritti in XML, un linguaggio specifico di JavaFX, e contengono la struttura e il layout delle varie schermate.
- **controller**: si occupa di gestire le interazioni tra l'utente e il sistema, andando a ricevere gli input dalle interfacce grafiche fxml, per poi processarli e passarli alla logica applicativa vera e propria gestita dal package service. C'è uno specifico controller per ogni schermata dell'applicazione.
- **service**: implementa la vera logica di business, effettuando le operazioni sui dati, applicando le regole e interfacciandosi con i DAO attraverso il design pattern **Facade** per recuperare e modificare dati nel database.

4.4.1 Fxml

In questo paragrafo verrà brevemente analizzata la struttura di uno dei file contenuti nel package fxml. Non verrà posta molta attenzione su questo tipo di file poiché non sono la parte di principale interesse della relazione che, invece, si concentra sul funzionamento e la struttura di tutta l'applicazione e non sulla relativa parte grafica. Di seguito, nella figura 13, vediamo il file *LogIn.fxml*

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.StackPane?>
<?import javafx.scene.layout.VBox?>

<StackPane prefHeight="400.0" prefWidth="600.0" xmlns="http://javafx.com/javafx/24.0.1"
           xmlns:fx="http://javafx.com/fxml/1"
           fx:controller="businessLogic.controller.LogInController">
    <VBox alignment="CENTER" spacing="20">
        <Label style="-fx-font-size: 24px;" text="Login" />
        <HBox alignment="CENTER_LEFT" maxWidth="330.0" spacing="10">
            <Label prefHeight="18.0" prefWidth="55.0" text="Email:" />
            <TextField fx:id="emailField" minWidth="250.0" />
        </HBox>
        <HBox alignment="CENTER_LEFT" maxWidth="330.0" spacing="9.0">
            <Label prefWidth="55.0" text="Password:" />
            <PasswordField fx:id="passwordField" minWidth="250.0" />
        </HBox>
        <Button fx:id="loginButton" onAction="#handleLogin" text="Accedi" />
        <Label fx:id="errorLabel" textFill="red" />
    </VBox>
</StackPane>
```

Figura 13: File LogIn.fxml

Questo file definisce l'aspetto grafico della schermata di login, in particolare gli elementi su cui porre attenzione sono: il layout principale *StackPane* che definisce le dimensioni della finestra, il *VBox* che organizza verticalmente gli elementi della schermata, l'etichetta *Login* necessaria per il relativo controller, i due *HBox* che contengono i campi per l'inserimento di email e password (ognuno con la propria etichetta), il pulsante *Accedi* che, quando premuto, richiama il metodo *handleLogin* del controller associato e infine un'etichetta che permette la visualizzazione di eventuali messaggi di errore.

4.4.2 Controller

Osserviamo ora la struttura di un controller, come nel paragrafo precedente non si scenderà troppo nei dettagli di questo package poiché composto da file necessari per il render grafico.
Prendiamo come esempio la figura 14 che riporta la classe *LogInController*.

```java
public class LogInController {
    @FXML
    private TextField emailField;
    @FXML
    private PasswordField passwordField;
    @FXML
    private Button loginButton;
    @FXML
    private Label errorLabel;
    private final LogInService logInService = new LogInService();
    private static final Logger logger = Logger.getLogger(LogInController.class.getName());
    @FXML
    public void initialize() {
        emailField.setOnAction( ActionEvent e -> loginButton.fire());
        passwordField.setOnAction( ActionEvent e -> loginButton.fire());
    }
    @FXML
    private void handleLogin(ActionEvent event) {
        String email = emailField.getText();
        String password = passwordField.getText();
        errorLabel.setText("");
        Staff staff = logInService.authenticate(email, password);
        if (staff == null) {
            errorLabel.setText("Credenziali errate");
            return;
        }
        UserSession.getInstance().setStaff(staff);
        try {
            String type = staff.getTypeOfStaff().toString();
            if ("OPERATOR".equalsIgnoreCase(type)) {
                SceneManager.getInstance().switchScene( fxmlPath: "/businessLogic/fxml/OperatorHome.fxml");
            } else {
                if ("SUPERVISOR".equalsIgnoreCase(type)){
                    SceneManager.getInstance().switchScene( fxmlPath: "/businessLogic/fxml/SupervisorHome.fxml");
                } else {
                    errorLabel.setText("Ruolo non supportato");
                }
            }
        } catch (Exception e) {
            logger.log(Level.SEVERE, msg: "Error while switching scene", e);
            errorLabel.setText("Errore interno: impossibile cambiare schermata");
        }
    }
}
```

Figura 14: File LogInController

Questa classe gestisce la logica della schermata di login dell'applicazione. Riceve

le credenziali inserite dall'utente, le verifica tramite *LogInService*, una classe del package service, e poi reindirizza alla schermata corretta sulla base del ruolo dell'utente.
La classe è strettamente collegata al file *LogIn.fxml*, da questo riceve i riferimenti ai componenti grafici (ad esempio i campi di testo per email e password). Inoltre **LogInController** contiene metodi annotati con @FXML, ovvero metodi che vengono chiamati nel rispettivo file fxml tramite gli attributi onAction; questi gestiscono gli eventi generati dall'interfaccia utente.
Attraverso questo esempio è così evidente come i controller fungano da ponte tra la logica applicativa e l'interfaccia grafica.

4.4.3 Service

Come brevemente detto in precedenza, il package service è uno dei più rilevanti del progetto. Le sue classi descrivono la parte centrale della logica di tutta l'applicazione poiché vanno a collegare la facade, quindi il dao, con i controller e, di conseguenza, con l'interfaccia utente. In altre parole, il service si occupa di gestire le operazioni, dette di business, andando ad utilizzare i dati in modo tale che rispettino le regole di dominio e, quindi, il corretto funzionamento dell'applicazione.
Di seguito vengono mostrati alcuni dei metodi più rilevanti di questo package per comprenderne la struttura, i suoi compiti e il modo in cui definisce le regole.

```
public Staff authenticate(String email, String password) {
    try {
        Staff staff = facade.findStaffByEmail(email);
        if (staff != null && staff.getPassword().equals(password)) {
            return staff;
        }
    } catch (Exception e) {
        logger.log(Level.SEVERE, msg: "Error during authentication", e);
    }
    return null;
}
```

Figura 15: Metodo authenticate

Per completezza, nella figura 15 viene riportato il metodo di autenticazione che verifica la correttezza dei dati inseriti nella fase di login.
Questo metodo, contenuto nella classe LogInService, riceve in ingresso i dati inseriti dall'utente e verifica che corrispondano alle credenziali salvate sul database. In particolare, quello che viene fatto è recuperare l'oggetto *Staff* associato all'e-mail tramite facade; se l'utente esiste e la password passata in ingresso coincide con quella salvata allora viene restituito l'oggetto *Staff* permettendo quindi l'acces-

so. In caso contrario viene restituito *null* e appare a schermo un messaggio di errore.

```
public void createConvoy(List<Carriage> carriages) {
    try {
        Convoy newConvoy = facade.createConvoy(carriages);
        int newConvoyId = newConvoy.getId();
        Integer idStation = null;
        if (!carriages.isEmpty()) {
            domain.CarriageDepot depot = facade.findActiveDepotByCarriage(carriages.getFirst().getId());
            if (depot != null) {
                idStation = depot.getIdDepot();
            }
        }
        if (idStation != null) {
            ConvoyPool pool = ConvoyPool.of(newConvoyId, idStation, ConvoyPool.ConvoyStatus.WAITING);
            facade.insertConvoyPool(pool);
        }
        for (Carriage carriage : carriages) {
            carriage.setIdConvoy(newConvoyId);
            facade.updateCarriageConvoy(carriage.getId(), newConvoyId);
            facade.deleteCarriageDepotByCarriageIfAvailable(carriage.getId());
        }
    } catch (Exception e) {
        logger.severe( msg: "Error creating convoy: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

Figura 16: Metodo createConvoy

Il metodo riportato in figura 16 si occupa della creazione di un nuovo convoglio a partire da una lista di carrozze. Inizialmente viene creato il nuovo convoglio tramite facade, poi viene aggiornato lo stato delle carrozze: vengono associate al nuovo convoglio e rimosse dal deposito in cui si trovavano. Se le carrozze provengono da un deposito attivo allora viene creato un gruppo (*ConvoyPool pool*) con stato di attesa, pronto per essere assegnato a una nuova corsa.
Tutte queste operazioni sono fatte da questo singolo metodo in modo tale che venga garantita la coerenza tra i dati delle carrozze, dei convogli e dei depositi. In caso di errore viene generata un'eccezione e salvato un messaggio nei log.

```
public void updateCarriageDepotStatuses() {
    try {
        List<CarriageDepot> depots = facade.getCarriagesInCleaningOrMaintenance();
        long now = System.currentTimeMillis();
        for (CarriageDepot depot : depots) {
            long entered = depot.getTimeEntered().getTime();
            long millisToWait = depot.getStatusOfCarriage() ==
                    CarriageDepot.StatusOfCarriage.CLEANING ? 3_600_000L : 21_600_000L;
            if (now - entered >= millisToWait) {
                facade.updateCarriageDepotStatusAndExitTime(
                    depot.getIdDepot(),
                    depot.getIdCarriage(),
                    CarriageDepot.StatusOfCarriage.AVAILABLE.name(),
                    new java.sql.Timestamp(now)
                );
            }
        }
    } catch (Exception e) {
        logger.severe( msg: "Errore in updateCarriageDepotStatuses: " + e.getMessage());
    }
}
```

Figura 17: Metodo updateCarriageDepotStatuses

In figura 17 è riportato il metodo il cui compito è aggiornare lo stato delle carrozze presenti in deposito per pulizia o manutenzione. Tramite facade il metodo recupera tutte le carrozze presenti in deposito con lo stato CLEANING o MAINTENANCE e per ognuna verifica che sia trascorso il tempo necessario (un'ora per la pulizia e sei ore per la manutenzione) rispetto al tempo di ingresso. Se tale tempo è trascorso aggiorna lo stato delle carrozze in AVAILABLE e genera il tempo di uscita dal deposito.
Questo metodo viene richiamato ad ogni avvio dell'applicazione in modo tale da rendere automatica la gestione della disponibilità delle carrozze, rispettando i vincoli di tempo per le operazioni in deposito.

```java
public void completeRun(Run run) {
    try {
        int convoyId = run.getIdConvoy();
        int tailStationId = run.getIdLastStation();
        List<Notification> approvedNotifications = facade.selectApprovedNotificationsByConvoy(convoyId);
        for (Notification notif : approvedNotifications) {
            int carriageId = notif.getIdCarriage();
            // Rimuovi la carrozza dal convoglio
            facade.updateCarriageConvoy(carriageId, idConvoy: null);
            // Propaga la rimozione alle corse future
            List<Run> futureRuns = facade.selectRunsForConvoyAfterTime(convoyId, run.getTimeArrival());
            for (Run _ : futureRuns) {
                facade.updateCarriageConvoy(carriageId, idConvoy: null);
            }
            // Trova il deposito associato alla stazione di coda
            Depot depot = facade.getDepotByStationId(tailStationId);
            if (depot != null) {
                int depotId = depot.getIdDepot();
                CarriageDepot.StatusOfCarriage status =
                    notif.getTypeOfNotification().equalsIgnoreCase( anotherString: "CLEANING") ?
                    CarriageDepot.StatusOfCarriage.CLEANING :
                    CarriageDepot.StatusOfCarriage.MAINTENANCE;
                java.sql.Timestamp now = new java.sql.Timestamp(System.currentTimeMillis());
                CarriageDepot carriageDepot = CarriageDepot.of(depotId, carriageId, now, timeExited: null, status);
                facade.insertCarriageDepot(carriageDepot);
            }
        }
    } catch (Exception e) {
        logger.severe( msg: "Errore in completeRun: " + e.getMessage());
    }
}
```

Figura 18: Metodo completeRun

Il metodo *completeRun* in figura 18 nella classe *ManageRunService* gestisce la logica che avviene a fine corsa. Quando una corsa termina, il metodo va a rimuovere dal convoglio tutte le carrozze per cui un supervisore ha approvato una notifica di pulizia o manutenzione generata dall'operatore di turno. Si occupa quindi di andare a rimuovere tali carrozze anche da eventuali corse future già pianificate per il convoglio a cui appartenevano. Successivamente, assegna ciascuna carrozza al deposito della stazione di arrivo, impostando lo stato corretto e registrando il tempo di ingresso.
Così facendo viene automatizzata la gestione delle carrozze che necessitano di interventi, garantendo che non vengano assegnate ad altre corse fino al termine delle operazioni necessarie.

5 Design Patterns

5.1 Dao

Il **Data Access Object** viene utilizzato per isolare la logica di accesso al database da quella di business. Ogni classe DAO contiene le query SQL e i metodi CRUD specifici per le rispettive entità, inoltre gestisce la connessione. I dettagli implementativi sono riportati nel paragrafo 4.2.

5.2 Mapper

Il **Data Mapper** separa la conversione delle righe di risultato del database dalla business logic andando a trasformare ogni riga in un oggetto del dominio. Anche in questo caso i dettagli sono già stati analizzati nel paragrafo 4.3.

5.3 Facade

Il pattern **Facade** offre un'interfaccia semplice che unisce diverse operazioni sulle varie classi del DAO e dei service, nascondendo la complessità delle chiamate alla business logic. Attraverso di essa, il codice dei service è molto più pulito e più semplice. Di seguito nella figura 19 vediamo una parte della **Facade** per comprenderne la struttura.

```
private final CarriageDao carriageDao = CarriageDao.of();
private final StaffDao staffDao = StaffDao.of();
private final ConvoyDao convoyDao = ConvoyDao.of();
private final RunDao runDao = RunDao.of();
private final NotificationDao notificationDao = NotificationDao.of();

public Carriage selectCarriage(int id) throws SQLException {
    return carriageDao.selectCarriage(id);
}
public Staff findStaffByEmail(String email) throws SQLException {
    return staffDao.findByEmail(email);
}
public Convoy selectConvoy(int id) throws SQLException {
    return convoyDao.selectConvoy(id);
}
public Run selectRun(int idLine, int idConvoy, int idStaff) throws SQLException {
    return runDao.selectRunByLineConvoyAndStaff(idLine, idConvoy, idStaff);
}
public void addNotification(int idCarriage, int idConvoy, Timestamp timestamp,
                            String workType, int idStaff) throws SQLException {
    notificationDao.addNotification(idCarriage, idConvoy, timestamp, workType, idStaff);
}
```

Figura 19: Facade

5.4 Singleton

Il **Singleton** risulta utilizzato in quelle classi dove è necessario avere una sola istanza condivisa. Nel progetto troviamo ciò nelle classi *PostgrsConnection* e *SceneManager*: nel primo caso il pattern viene presentato tramite il metodo *getConnection()* ma la classe in sé non ha una propria istanza, si tratta quindi di un **Singleton** "stateless" (è quindi una utility class). Nel secondo caso invece c'è un'implementazione classica del pattern poiché troviamo un campo statico *instance*, un costruttore privato e il metodo statico *getInstance()* che restituisce sempre la stessa istanza.

5.5 Factory

Nel progetto questo design pattern è utilizzato nella forma più semplice di **static factory method**. Un esempio concreto è il seguente in figura 20.

```
static Line of(int idLine, String lineName, int idFirstStation,
               String firstStationLocation, int idLastStation, String lastStationLocation) {
    return new LineImp(idLine, lineName,idFirstStation, firstStationLocation,
            idLastStation, lastStationLocation);
}
```

Figura 20: Static factory method

Questo metodo incapsula la creazione dell'oggetto e restituisce il tipo astratto Line andando a nascondere l'implementazione. Si tratta quindi di una **Factory** semplice che migliora la leggibilità, da notare quindi come non sia effettivamente una vera e propria factory o abstract factory.

5.6 Observer

Il pattern **Observer** nel progetto viene utilizzato per gestire il flusso delle notifiche generate dall'operatore verso il supervisore, il quale, di conseguenza, decide se mandare o meno la carrozza in deposito per pulizia o manutenzione. Grazie a questo pattern è così possibile definire una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato interno, ciascuno degli oggetti dipendenti da esso viene notificato e aggiornato automaticamente.
Di seguito nella figura 21 sono riportati frammenti di codice da più classi che compongono il design pattern in questione.

```
public class SupervisorHomeController implements NotificationObserver {
     notificationService.addObserver(this);
     public void onNotificationAdded(Notification notification) {
        if (notification.getStatus() != null && notification.getStatus().equals("INVIATA")) {
            data.add(new NotificationRow(
                notification,
                notification.getIdCarriage(),
                notification.getTypeOfCarriage(),
                notification.getDateTimeOfNotification().toString(),
                notification.getStaffSurname(),
                notification.getTypeOfNotification()
            ));
            adjustTableHeight();
        }
     }
}

public class NotificationService {
    private final List<NotificationObserver> observers = new ArrayList<>();
    public void addObserver(NotificationObserver observer) {
            observers.add(observer);
    }
    private void notifyObservers(Notification notification) {
        for (NotificationObserver observer : observers) {
            observer.onNotificationAdded(notification);
        }
    }
    public void addNotification(int idCarriage, String typeOfCarriage, int idConvoy,
            String typeOfNotification, java.sql.Timestamp notifyTime, int idStaff,
            String staffName, String staffSurname, String status) {
        try {
            // Passa solo i parametri necessari alla Facade
            facade.addNotification(idCarriage, idConvoy, notifyTime, typeOfNotification, idStaff);
            Notification notification = Notification.of(idCarriage, typeOfCarriage,
                    idConvoy, typeOfNotification, notifyTime, idStaff, staffName, staffSurname, status);
            notifyObservers(notification);
        } catch (Exception e) {
            LOGGER.log(Level.SEVERE, "Errore durante l'aggiunta della notifica", e);
        }
    }
}
```

Figura 21: Observer

La classe *NotificationService* ha al suo interno una lista di observer e ogni volta che viene aggiunta una nuova notifica tramite il metodo *addNotification*, contenuto nella classe *NotificationService*, viene invocato il metodo di callback *onNotificationAdded* su tutti gli observer registrati nella lista. Il controller *SupervisorHomeController*, registrato come observer, aggiorna quindi la tabella delle notifiche all'arrivo di nuovi eventi.

6 Testing