Transport Efficiency Manager

ODD: Object Design Document

Transport Efficiency Manager

Riferimento	
Versione	1.3
Data	10/03/2021
Destinatario	Prof.ssa F. Ferrucci
Presentato da	Team NC08
Approvato da	

Revision History

Data	Versione	Descrizione	Autori
20/12/2020	0.1	Stesura parte di introduzione	Federica Attianese
22/12/2020	1.0	Aggiornamento introduzione e stesura versione iniziale	Francesca Moschella
23/01/2021	1.1	Aggiornamento e aggiunta Design patterns	Federica Pica Federica Attianese Francesca Moschella
26/02/2021	1.2	Aggiunta class diagram	Federica Pica
13/03/2021	1.2.1	Revisione generale	Federica Pica Federica Attianese Francesca Moschella
29/03/2021	1.2.2	Aggiornamento class diagram	Federica Pica Federica Attianese Francesca Moschella
10/03/2021	1.3	Revisione finale ed ultimazione	Federica Pica Federica Attianese Francesca Moschella

Indice dei contenuti

1. Introduzione

1.1 Object-design trade-offs

La fase dell'Object Design fa sorgere diversi compromessi di progettazione ed è necessario stabilire quali punti far prevalere e quali invece rendere opzionali.
Per quanto concerne la realizzazione del sistema sono stati individuati i seguenti trade-offs:

Buy vs Build:
È vastissima e variegata la quantità di materiale (framework, librerie...) off-the-shelf disponibile attualmente per la realizzazione di software, ed è a questo che attingeremo per la semplificazione della realizzazione di alcune componenti dell'applicazione web. La scelta effettuata ricade, quindi, su un approccio teso al riutilizzo, tenendo sempre conto della adeguatezza e l'idoneità del materiale rispetto alle necessità per lo sviluppo del programma, di componenti e soluzioni offerte da terze parti.

User-friendliness vs Complessità delle interfacce:
L'implementazione di interfacce inutilmente elaborate sarà evitata, al fine di incentivare una maggiore facilità d'uso per l'utente finale ed un'esperienza di navigazione il meno complessa e disorientante possibile.

Efficienza vs. Tempistiche:

La buona riuscita e portata a termine della realizzazione dell'applicazione necessita che gli sviluppatori abbiano, acquisite, le conoscenze necessarie all'utilizzo delle tecnologie scelte per l'implementazione; solo in questo modo si garantirà come risultato finale un sistema efficiente e di qualità. Per questa ragione si terrà conto delle eventuali tempistiche necessarie agli sviluppatori per consentire loro tale acquisizione; si prioritizzerà quindi l'efficienza, a discapito della tempistica.

Comprensibilità vs Costi:

Una documentazione completa e dettagliata garantirà la comprensibilità del codice dell'applicazione per chi non ne ha partecipato alla creazione e ad eventuali nuovi sviluppatori coinvolti in un secondo tempo ed estranei all'originaria implementazione. A rispondere a questa esigenza sarà l'utilizzo diffuso, nel codice, di commenti, con lo scopo di facilitarne la comprensione e di conseguenza la fase di eventuale modifica.
La comprensibilità del codice sarà quindi un elemento prioritario, a cui non verranno applicati gli effetti di eventuali compromessi economici.

1.1.1 Componenti off-the-shelf

Come rilevato in precedenza, per il progetto software che si intende realizzare si ricorre all'uso di componenti off-the-shelf, queste sono componenti software già testate e funzionanti, disponibili sul mercato ed utilizzate, in questo caso, per

facilitare la creazione del progetto software e per venire incontro al tipo di linguaggi utilizzati per l'implementazione. In particolare, per il back-end dell'applicazione la scelta è ricaduta sul framework Spring; oltre a facilitare l'integrazione fra le varie parti del programma questo framework risponde anche all'esigenza di dover tener conto di un budget limitato per l'implementazione; Spring è infatti un framework open source per lo sviluppo di applicazioni web in Java ed inoltre uno dei più diffusi ed utilizzati nelle aziende che hanno Java come linguaggio di sviluppo, rappresentando quindi uno strumento consolidato, conosciuto e "solido" ,avendo alle spalle molti anni di sviluppo ed utilizzo. I moduli che interesseranno il progetto saranno Spring Web, Spring Security e Spring Data JPA. Per il front-end sarà utilizzato il template engine Thymeleaf.

1.1.2 Riuso e flessibilità

1.1.2.1 Ereditarietà

L'utilizzo dell'ereditarietà permette di ridurre le ridondanze, migliorare l'estendibilità, la modificabilità e il riuso del codice. In questo progetto, introduciamo l'ereditarietà di specifica; quest'ultima definisce la possibilità di utilizzare un oggetto al posto di un altro.

In particolare, l'ereditarietà si ha per la classe Risorsa: viene definita come classe astratta, le cui sottoclassi sono Mezzo, Conducente e Linea.

1.1.2.2 Composizione

La composizione permette di ottenere nuove funzionalità per aggregazione. In questo progetto viene utilizzata la delega tramite alcuni dei design pattern elencati nel punto successivo.

1.1.2.3 Design Pattern

Per rendere più veloce la fase di sviluppo del sistema abbiamo fatto ricorso al riutilizzo di soluzioni che risolvono problemi ricorrenti e prevedibili, presenti anche nel nostro progetto. Di seguito sono elencati i design pattern utilizzati in questo progetto, suddivisi secondo la classificazione GoF e schematizzati mettendo in evidenza, per ognuno la descrizione del pattern e del problema che questo risolve e la sua soluzione.

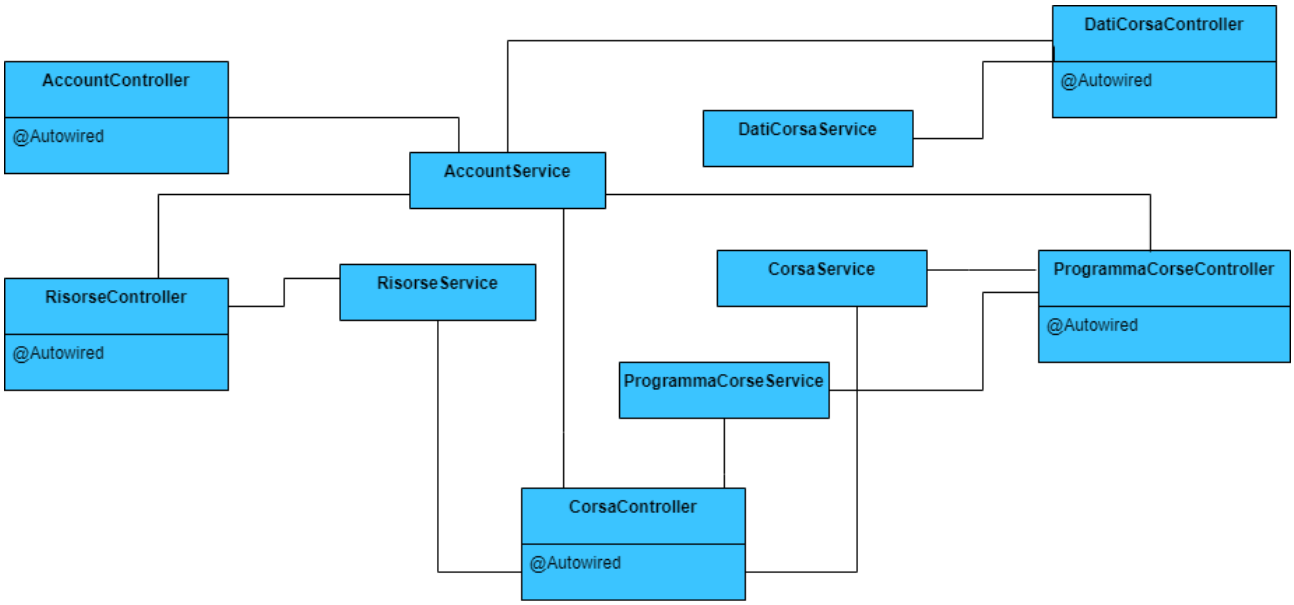
1.1.2.3.1 Design pattern creazionali

Dependency injection pattern

TEM fa uso del pattern Dependency injection per gestire le dipendenze di oggetti. È un procedimento necessario siccome Spring Boot ne fa ampiamente uso, in quanto è una delle metodologie per implementare l'Inversion of Control.

- **Descrizione**: Rende le componenti software il più indipendenti possibile, affinché sia possibile modificarne una parte senza dover modificare le altre.
- **Soluzione**: vengono iniettate le dipendenze marcandole con l'annotazione @Autowired. Questo consente, per gli oggetti che sono stati caricati dal container, di caricare automaticamente le loro dipendenze, fornite sempre dal

container. Questo permette di utilizzare una dipendenza senza dover creare manualmente l'istanza.

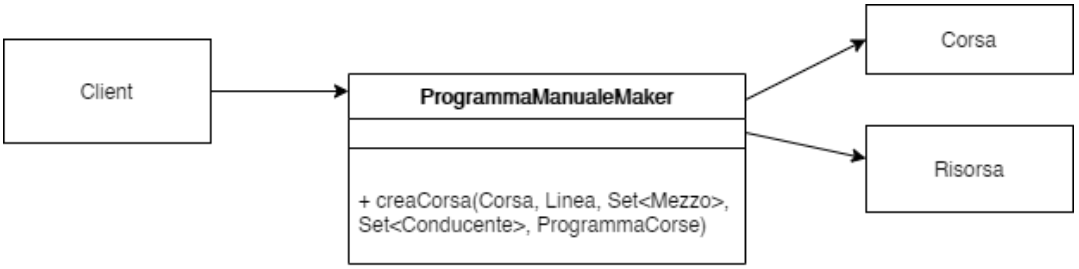


1.1.2.3.2 Design pattern strutturali

Facade pattern

TEM fa uso del Facade pattern per nascondere la complessità dell'operazione di generazione del programma manuale.

- **Descrizione**: Fornisce un'unica interfaccia per accedere ad un insieme di oggetti che prendono parte a blocchi di codice complessi.
- **Soluzione:** viene definita una classe ad alto livello (ProgrammaManualeMaker) che semplifica l'utilizzo della funzionalità, gestendo la creazione delle singole corse che andranno poi a creare il programma completo e l'accesso alle risorse necessarie per definire le singole corse.

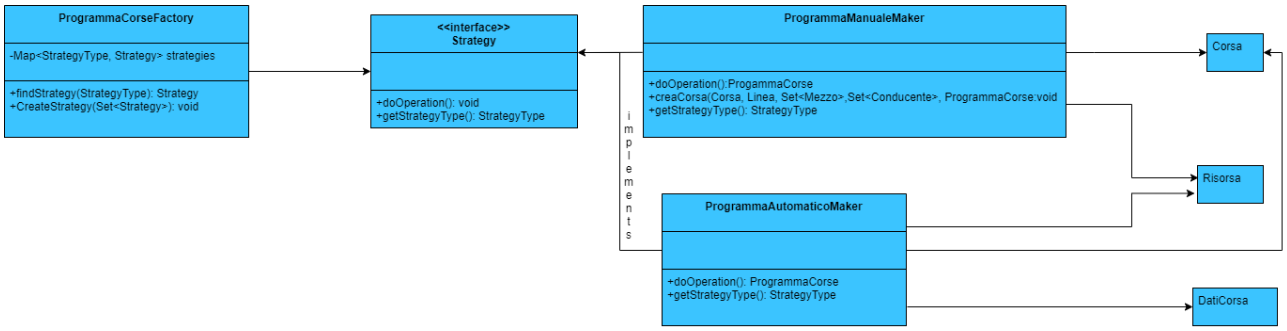


1.1.2.3.3 Design pattern comportamentali

Strategy pattern

TEM fa uso dello Strategy pattern per gestire a runtime la scelta del metodo di generazione del programma delle corse.

- **Descrizione**: Questo pattern prevede che gli algoritmi siano intercambiabili tra loro, in base ad una specificata condizione, in modalità trasparente al client che ne fa uso.
- **Soluzione**: vengono definite due "strade" diverse per i due tipi di generazione del programma delle corse. Nella classe ProgrammaCorse, generationType stabilisce il tipo di generazione che dovrà essere effettuata. All'interno di questa classe è definito il metodo executeStrategy che sceglie una delle due strade e genera un programma corse, chiamando il metodo doOperation di una delle due classi che implementano l'interfaccia Strategy. ProgrammaManualeMaker gestisce, come specificato al punto precedente tramite il facade pattern, la generazione del programma in modalità manuale; ProgrammaAutomaticoMaker, invece, gestisce la generazione del programma in modalità automatica, tramite l'utilizzo dell'intelligenza artificiale. Per far sì che ciò sia possibile, c'è bisogno di prelevare i dati delle singole corse, in quanto permettono di ottenere un programma "intelligente", basato sulle esperienze passate.



1.1.2.4 Altri pattern

Model View Controller

Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- il model fornisce i metodi per accedere ai dati utili all'applicazione;
- il view visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;
- il controller riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti.
  Questo schema, fra l'altro, implica anche la tradizionale separazione fra la logica, a carico del controller e del model, e l'interfaccia utente a carico del view.

  La struttura, nello specifico, di come è utilizzato il pattern in questo progetto, insieme alle motivazioni che ci hanno portato al suo utilizzo sono riportati nell'SDD, sezione 3.2.

Inversion of Control

L'utilizzo di questo pattern è una diretta conseguenza dell'utilizzo del framework Spring, essendo il pattern su cui si basa principalmente. Questo pattern descrive l'inversione del flusso di controllo in un sistema cosicché l'esecuzione del flusso non è controllata da un pezzo centrale di codice. Ciò significa che le componenti dovrebbero dipendere solo dalle astrazioni di altre componenti e non sono responsabili della gestione della creazione di oggetti dipendenti. Invece, gli oggetti istanziati sono forniti a run-time da un IoC container attraverso la Dependency injection.

L'inversione del controllo riduce l'accoppiamento, facilita il test delle componenti software e incrementa la riusabilità del codice.

1.2 Linee guida per la documentazione d'interfaccia

Agli sviluppatori, in fase di implementazione del sistema, è richiesto di attenersi a delle linee guida illustrate di seguito, al fine di rispettare una consistenza e coerenza di forma e facilitare la comprensione di ognuna delle funzionalità messe a punto.

Nomenclatura:

Classi: Ad ogni classe deve essere assegnato un nome che possa associarla in modo univoco e distinguibile all'entità di dominio, funzionalità o servizio a cui fa riferimento, è stato scelto inoltre di seguire la notazione Camel Case; in particolare, i nomi delle classi dovranno iniziare con lettera maiuscola, i nomi di metodi e attributi con lettera minuscola.
Metodi: Ogni metodo segue la notazione in lowerCamelCase.
Eccezioni: Il nome dell'eccezione deve essere correlato al problema che segnala.

Indentazione
Il codice Java dev'essere indentato in maniera appropriata (tramite 2 spazi " ") e l'apertura di un blocco di codice deve avvenire nella stessa riga in cui è specificato il nome della classe o la firma del metodo che quel blocco, in generale ogni istruzione deve essere opportunamente indentata per garantire una chiarezza di contenuto. Questa pratica è principalmente indicata quando si utilizzano istruzioni cicliche o condizionali come FOR ed IF.

Organizzazione delle componenti:

- I nomi dei file, delle operazioni e delle variabili devono essere significativi e permetterne l'immediata identificazione e lo scopo.
- Tutte le classi che realizzano un sottosistema devono essere racchiuse nello stesso pacchetto Java.
- Tutte le risorse statiche (fogli di stile, script e immagini) devono essere collocate nella directory "resources".

Pagine HTML

Il codice HTML, sarà utilizzato per definire la struttura delle pagine dell'applicazione che verrà realizzata, mentre la parte di stile sarà realizzata con linguaggio CSS. La versione di riferimento che verrà utilizzata è la versione 5. Il codice dovrà essere indentato come nell'esempio:

```
<html>
        <head>
        </head>
        <body>
        </body>
</html>
```

Ogni tag di apertura è essere necessariamente seguito dall'apposito tag di chiusura (eccetto i tag self-closing) e l'indentazione dovrà essere effettuata con un TAB.

Fogli di stile CSS

La versione di riferimento scelta per l'utilizzo di questo linguaggio è la 3. Per quanto concerne gli stili in comune che potranno essere utili in più punti devono essere inseriti in un foglio di stile globale, mentre gli stili definiti per una singola pagina vanno inseriti nei fogli di stile CSS presenti nella stessa cartella dove sono contenuti i file HTML a cui si riferiscono.

Organizzazione del repository

Infine, sono definite le convenzioni per condividere il proprio lavoro attraverso un repository Git. Ogni membro del team è tenuto a realizzare il proprio codice coordinandosi con gli altri membri del team, utilizzando, per semplicità, un unico master branch.

1.3 Definizioni, acronimi e abbreviazioni

ODD: Object Design Document
Off-The-Shelf: Servizi esterni di cui viene fatto utilizzo da terzi.
Framework: Software di supporto allo sviluppo web.

HTML: Linguaggio di Mark-up per pagine web.

CSS: Linguaggio usato per definire la formattazione di pagine web.

Spring: framework open source per lo sviluppo di applicazioni su piattaforma Java.

1.4 Riferimenti

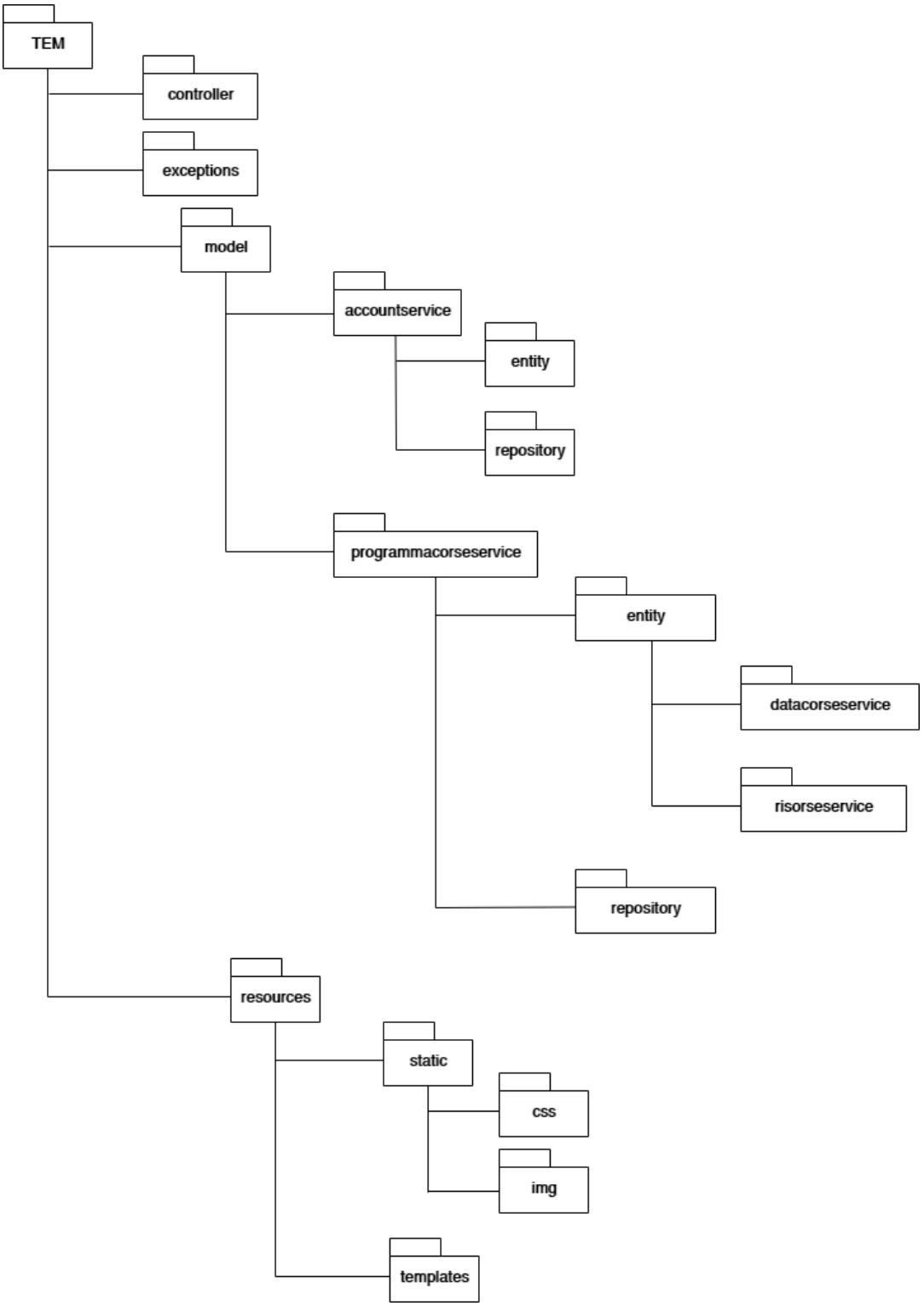
Design goals: sezione 1.2 dell'SDD
Scelta dell'ambiente d'esecuzione: sezioni 3.2 e 3.3 dell'SDD
Libri di testo:

- B. Bruegge et al, Object-Oriented Software Engineering Using UML, Patterns and Java, 2003
- E. Gamma et.al., Design Patterns, 1994

Sito web: https://www.javaguides.net/

2. Packages

Il design della struttura dei packages segue due strategie: esternamente, il progetto è organizzato seguendo la struttura dell'architettura usata, ovvero MVC, con la creazione dei packages Models, Views, Controllers; all'interno di essi, vi è una ulteriore organizzazione in **package-by-feature** che accresce la modularità e la coesione e mantiene minimo l'accoppiamento tra package. In pratica, ogni package rappresenta un dominio o una funzionalità dell'applicazione e contiene classi tra loro fortemente accoppiate. Gli oggetti che lavorano insieme sono raggruppati, non sono distribuiti in tutta l'applicazione. Ciò aumenta anche la manutenibilità del codice.



Il package java contiene principalmente tutti i file java. In particolare, il package **accountservice** contiene le componenti per la gestione dell'account; il package **programmacorseservice** contiene le componenti per la gestione del programma delle corse, oltre ai package **risorseservice** e **daticorsaservice** riservati, a loro volta, alle componenti relative alle risorse e ai dati delle singole corse.

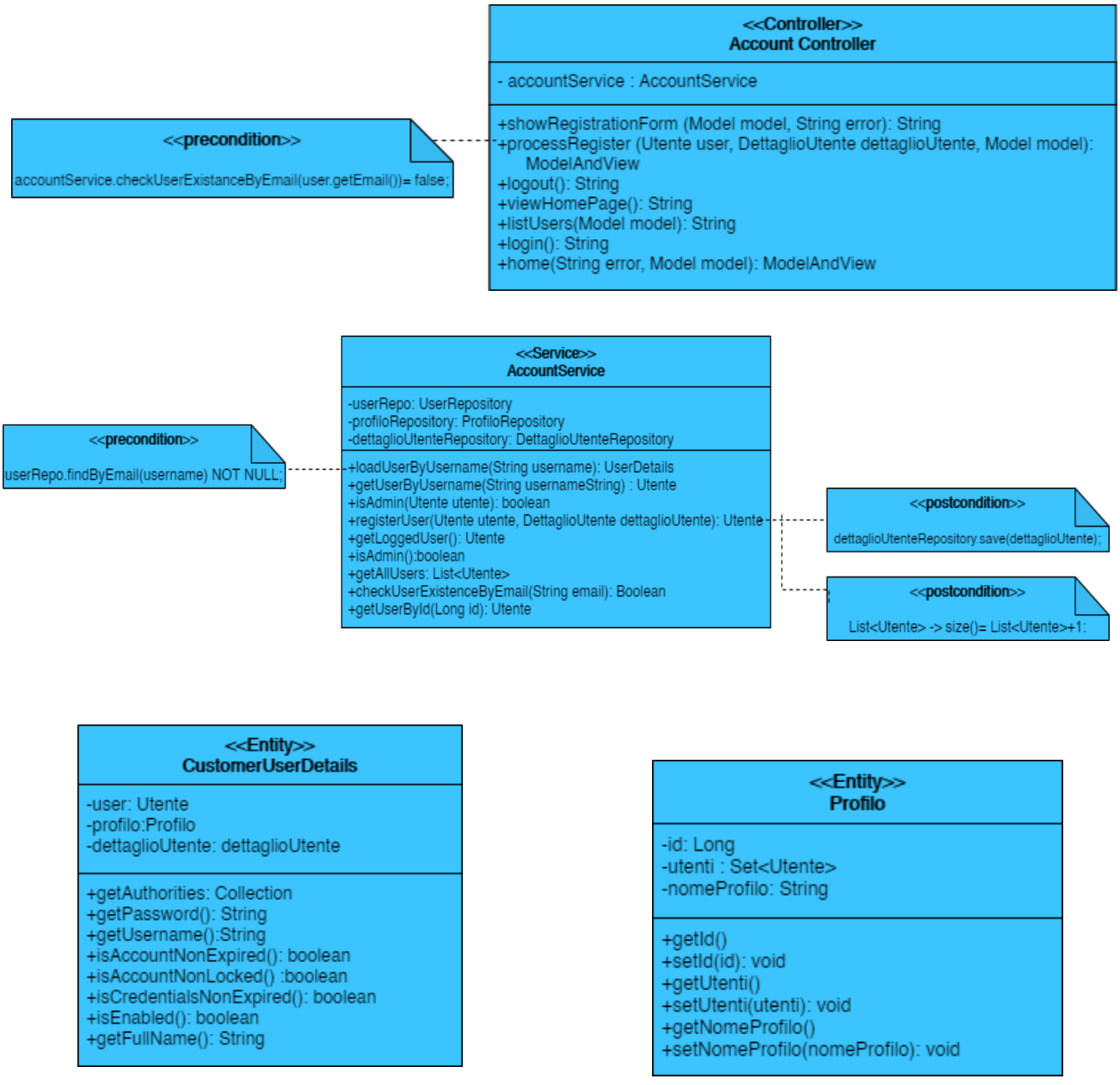
È opportuno specificare che i package **risorseservice** e **daticorsaservice** pur essendo totalmente indipendenti singolarmente, sono stati inseriti all'interno del package **programmacorseservice** in quanto non vale il viceversa, ovvero la funzionalità offerta da quest'ultimo non potrebbe essere attuata senza di esse.

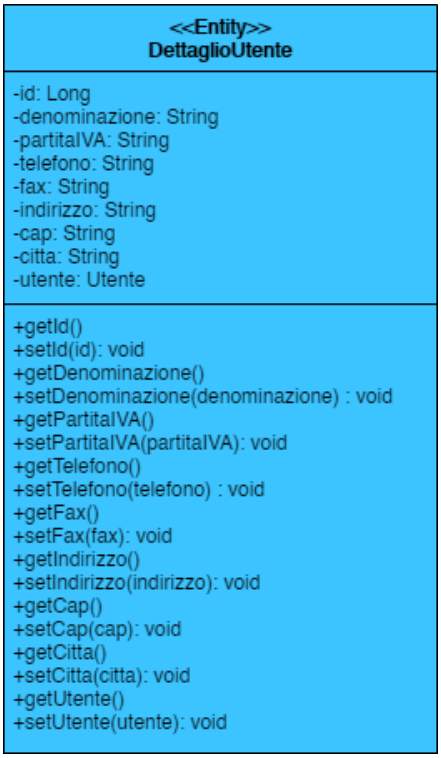
Il package **resources** contiene il resto delle risorse, ovvero quelle non riconducibili a Java. In particolare, il package **static** contiene le immagini, i files di stile del foglio e i fonts utilizzati dalle pagine presenti all'interno del package templates.

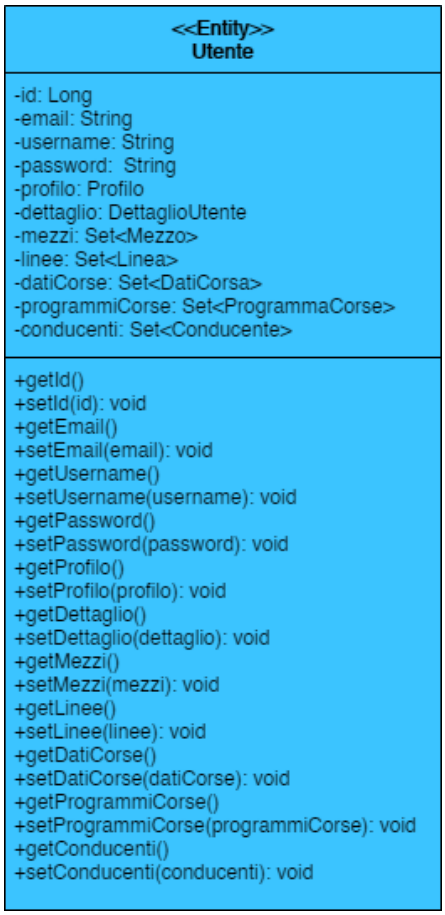
3. Class interfaces

Questa sezione contiene la specifica di ogni classe. Sono inoltre riportati, per ognuna di esse, i contratti rilevati, ovvero precondizioni, postcondizioni e invarianti.

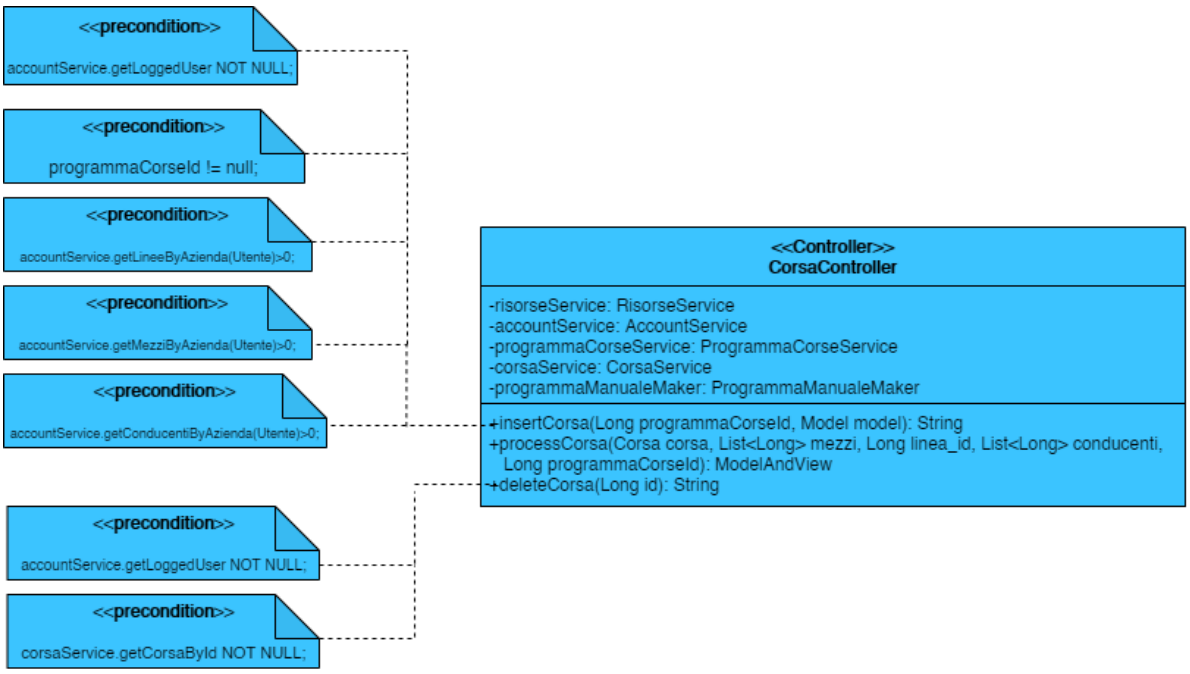
3.1 Gestione account

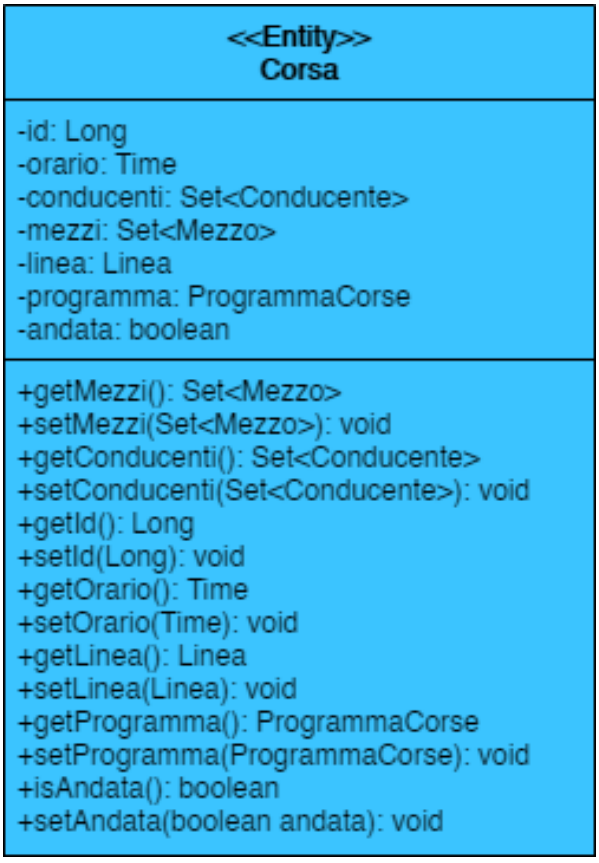


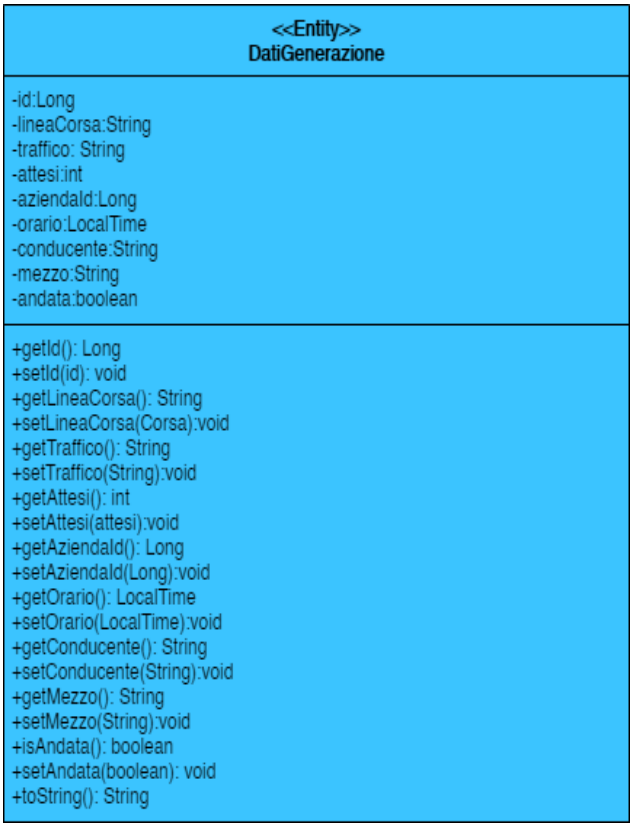


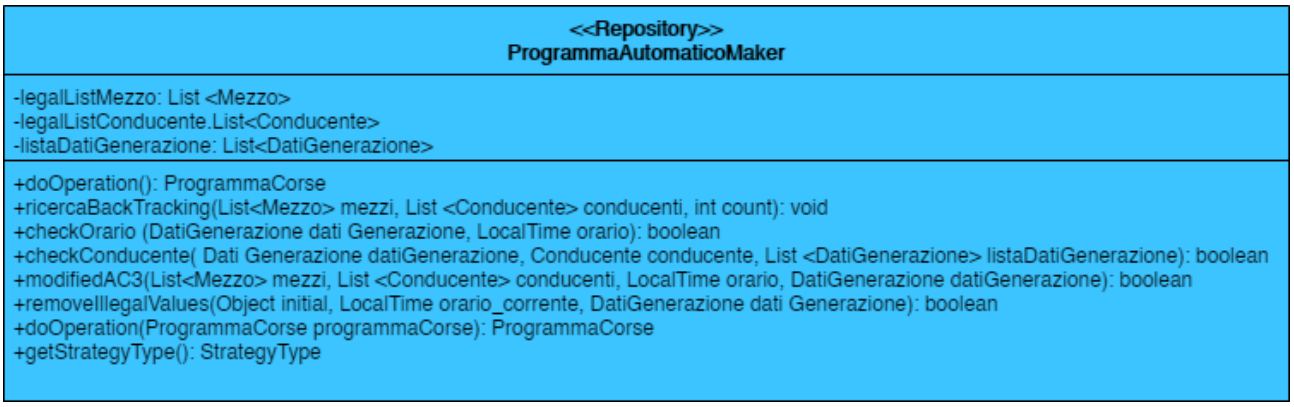


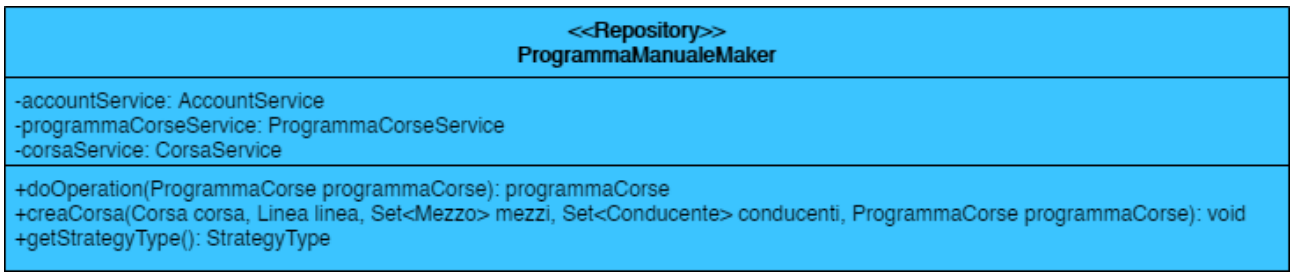
3.2 Gestione Programma delle corse

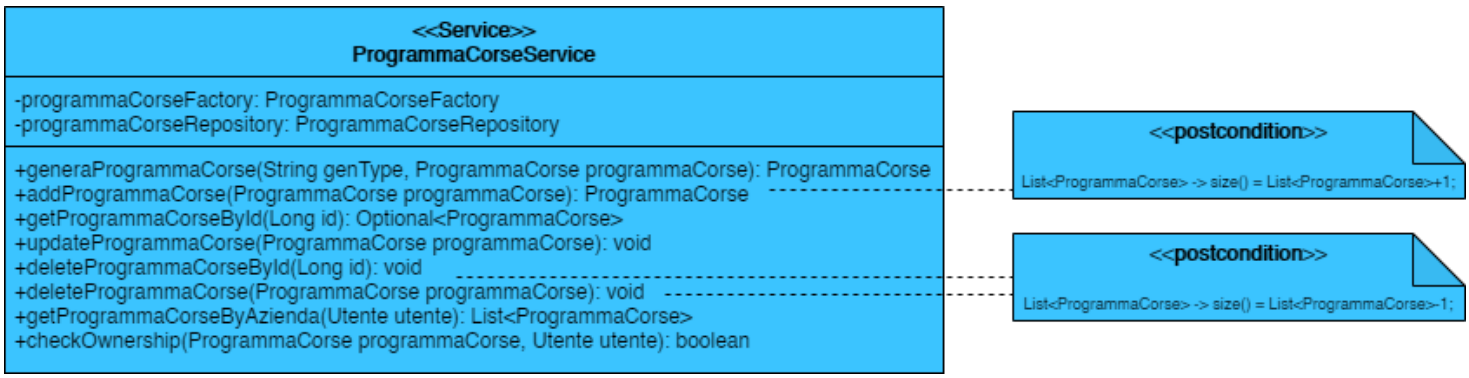


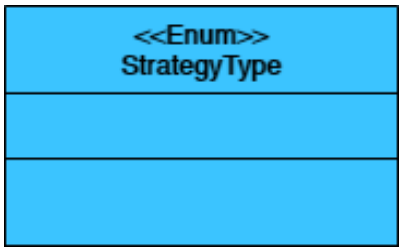

<<Entity>>
Corsa
-id: Long
-orario: Time
-conducenti: Set<Conducente>
-mezzi: Set<Mezzo>
-linea: Linea
-programma: ProgrammaCorse
-andata: boolean
+getMezzi(): Set<Mezzo>
+setMezzi(Set<Mezzo>): void
+getConducenti(): Set<Conducente>
+setConducenti(Set<Conducente>): void
+getId(): Long
+setId(Long): void
+getOrario(): Time
+setOrario(Time): void
+getLinea(): Linea
+setLinea(Linea): void
+getProgramma(): ProgrammaCorse
+setProgramma(ProgrammaCorse): void
+isAndata(): boolean
+setAndata(boolean andata): void

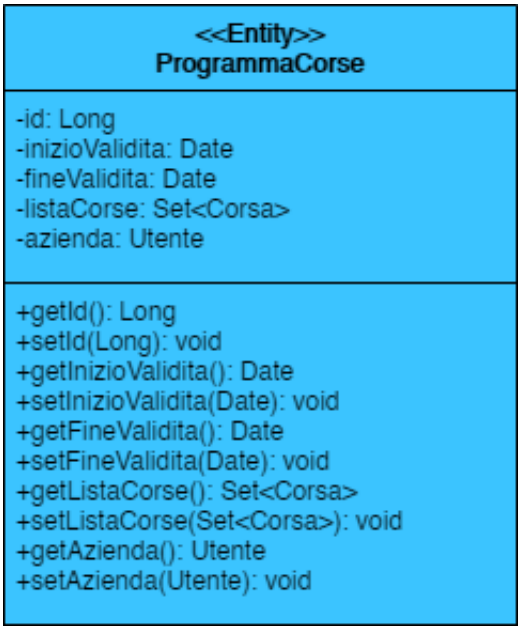

<<Entity>>
DatiGenerazione
-id:Long
-lineaCorsa:String
-traffico: String
-attesi:int
-aziendaId:Long
-orario:LocalTime
-conducente:String
-mezzo:String
-andata:boolean
+getId(): Long
+setId(id): void
+getLineaCorsa(): String
+setLineaCorsa(Corsa):void
+getTraffico(): String
+setTraffico(String):void
+getAttesi(): int
+setAttesi(attesi):void
+getAziendaId(): Long
+setAziendaId(Long):void
+getOrario(): LocalTime
+setOrario(LocalTime):void
+getConducente(): String
+setConducente(String):void
+getMezzo(): String
+setMezzo(String):void
+isAndata(): boolean
+setAndata(boolean): void
+toString(): String

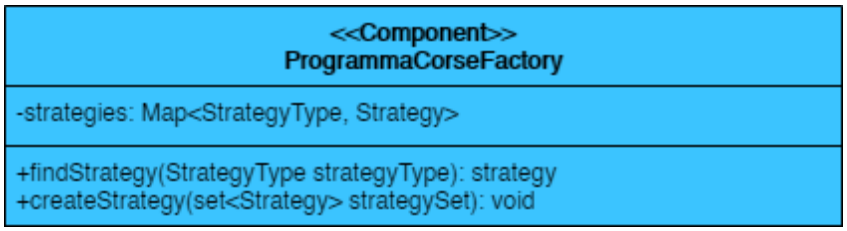

<<Repository>>
ProgrammaAutomaticoMaker
-legalListMezzo: List <Mezzo>
-legalListConducente.List<Conducente>
-listaDatiGenerazione: List<DatiGenerazione>
+doOperation(): ProgrammaCorse
+ricercaBackTracking(List<Mezzo> mezzi, List <Conducente> conducenti, int count): void
+checkOrario (DatiGenerazione dati Generazione, LocalTime orario): boolean
+checkConducente(Dati Generazione datiGenerazione, Conducente conducente, List <DatiGenerazione> listaDatiGenerazione): boolean
+modifiedAC3(List<Mezzo> mezzi, List <Conducente> conducenti, LocalTime orario, DatiGenerazione datiGenerazione): boolean
+removeIllegalValues(Object initial, LocalTime orario_corrente, DatiGenerazione dati Generazione): boolean
+doOperation(ProgrammaCorse programmaCorse): ProgrammaCorse
+getStrategyType(): StrategyType

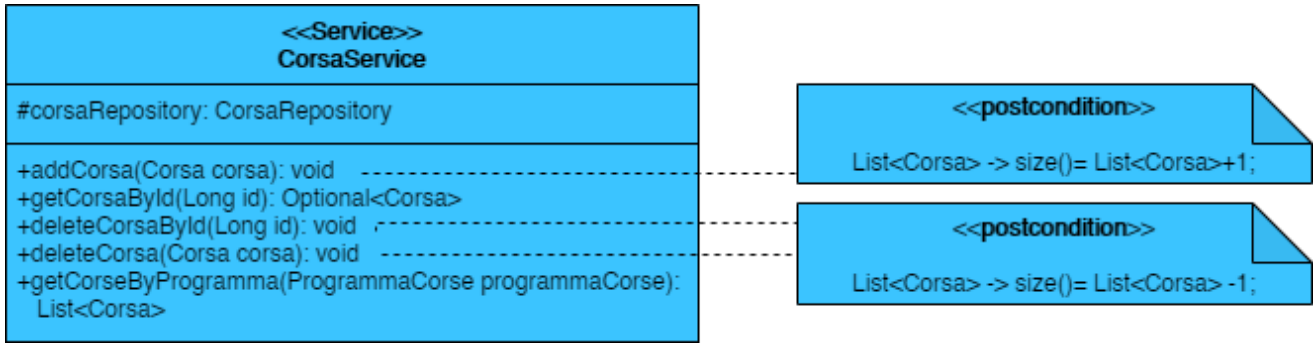

<<Repository>>
ProgrammaManualeMaker
-accountService: AccountService
-programmaCorseService: ProgrammaCorseService
-corsaService: CorsaService
+doOperation(ProgrammaCorse programmaCorse): programmaCorse
+creaCorsa(Corsa corsa, Linea linea, Set<Mezzo> mezzi, Set<Conducente> conducenti, ProgrammaCorse programmaCorse): void
+getStrategyType(): StrategyType

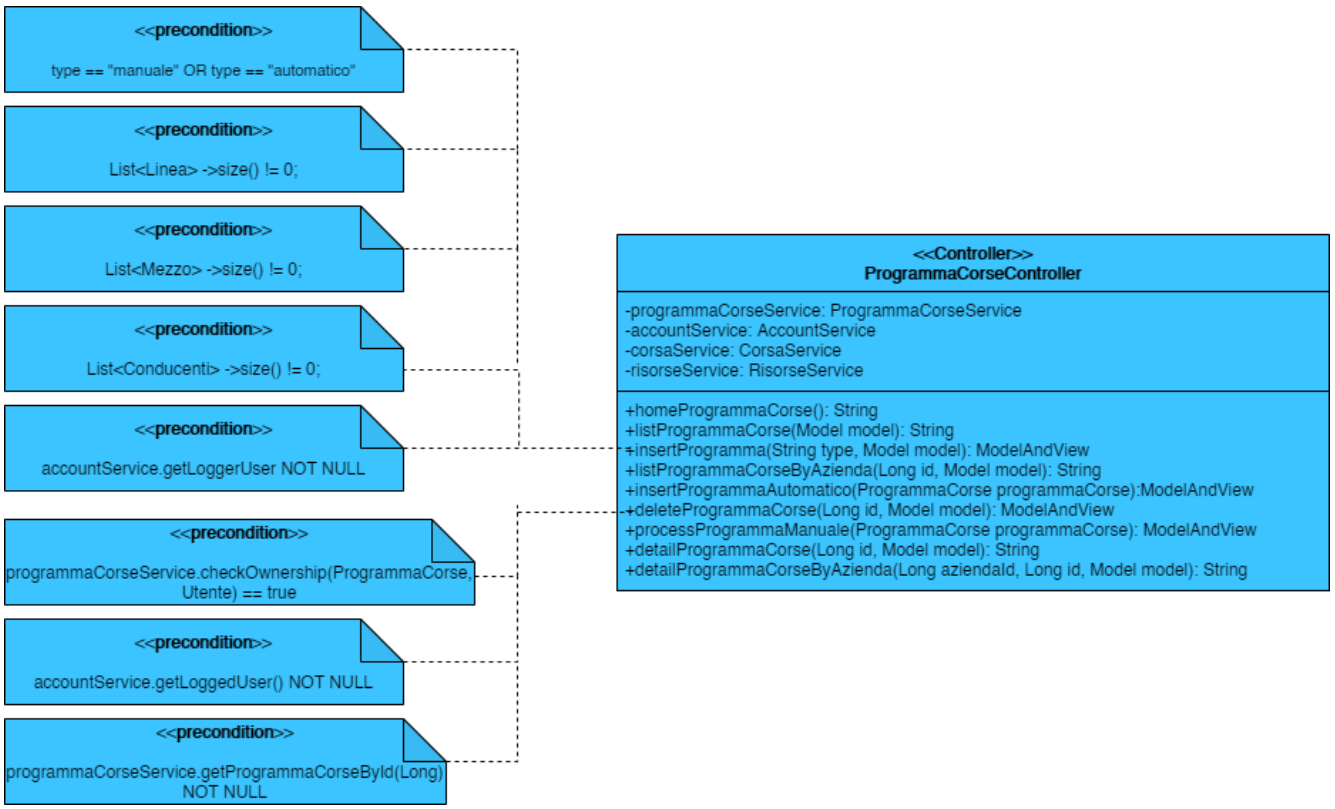

<<Service>>
ProgrammaCorseService
-programmaCorseFactory: ProgrammaCorseFactory
-programmaCorseRepository: ProgrammaCorseRepository
+generaProgrammaCorse(String genType, ProgrammaCorse programmaCorse): ProgrammaCorse
+addProgrammaCorse(ProgrammaCorse programmaCorse): ProgrammaCorse
+getProgrammaCorseById(Long id): Optional<ProgrammaCorse>
+updateProgrammaCorse(ProgrammaCorse programmaCorse): void
+deleteProgrammaCorseById(Long id): void
+deleteProgrammaCorse(ProgrammaCorse programmaCorse): void
+getProgrammaCorseByAzienda(Utente utente): List<ProgrammaCorse>
+checkOwnership(ProgrammaCorse programmaCorse, Utente utente): boolean
<<postcondition>>
List<ProgrammaCorse> -> size() = List<ProgrammaCorse>+1;
<<postcondition>>
List<ProgrammaCorse> -> size() = List<ProgrammaCorse>-1;


<<Enum>>
StrategyType

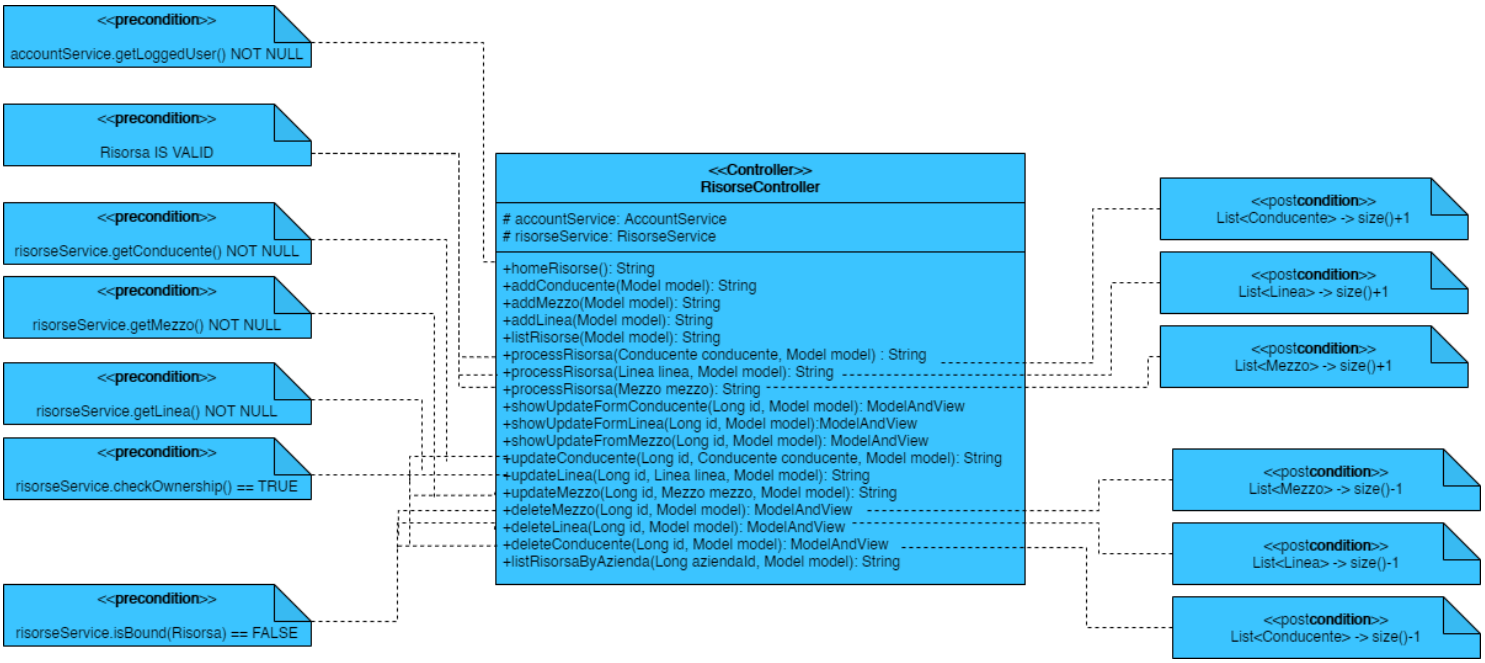

<<Entity>>
ProgrammaCorse
-id: Long
-inizioValidita: Date
-fineValidita: Date
-listaCorse: Set<Corsa>
-azienda: Utente
+getId(): Long
+setId(Long): void
+getInizioValidita(): Date
+setInizioValidita(Date): void
+getFineValidita(): Date
+setFineValidita(Date): void
+getListaCorse(): Set<Corsa>
+setListaCorse(Set<Corsa>): void
+getAzienda(): Utente
+setAzienda(Utente): void


<<Component>>
ProgrammaCorseFactory
-strategies: Map<StrategyType, Strategy>
+findStrategy(StrategyType strategyType): strategy
+createStrategy(set<Strategy> strategySet): void

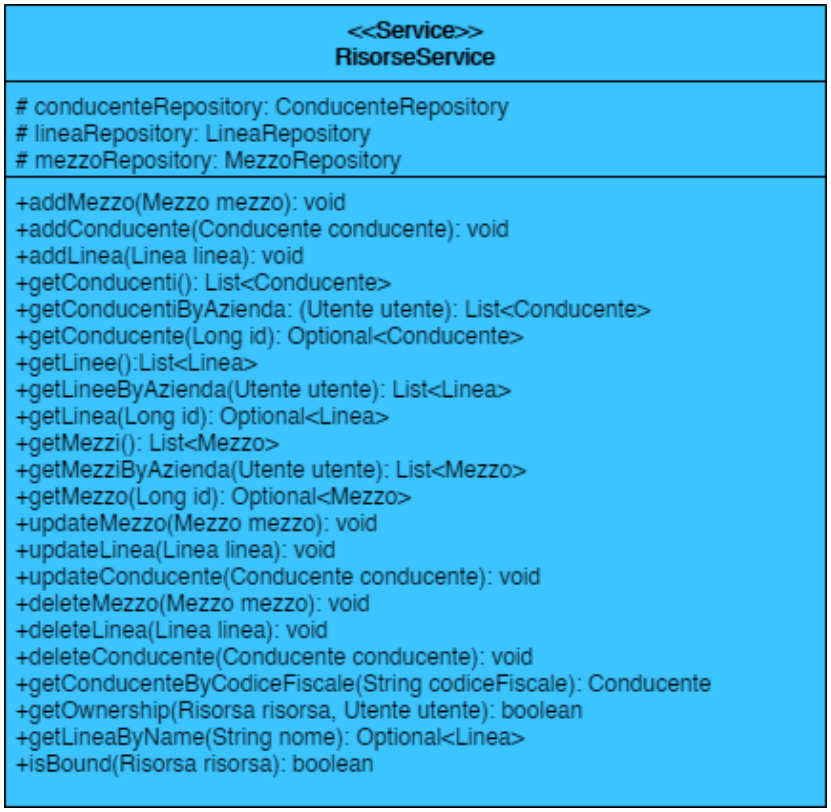

<<Service>>
CorsaService
#corsaRepository: CorsaRepository
+addCorsa(Corsa corsa): void
+getCorsaById(Long id): Optional<Corsa>
+deleteCorsaById(Long id): void
+deleteCorsa(Corsa corsa): void
+getCorseByProgramma(ProgrammaCorse programmaCorse):
List<Corsa>
<<postcondition>>
List<Corsa> -> size()= List<Corsa>+1;
<<postcondition>>
List<Corsa> -> size()= List<Corsa> -1;

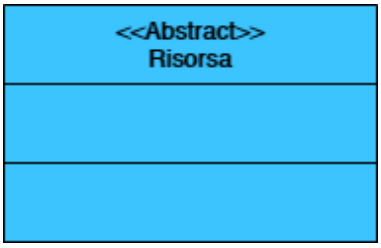


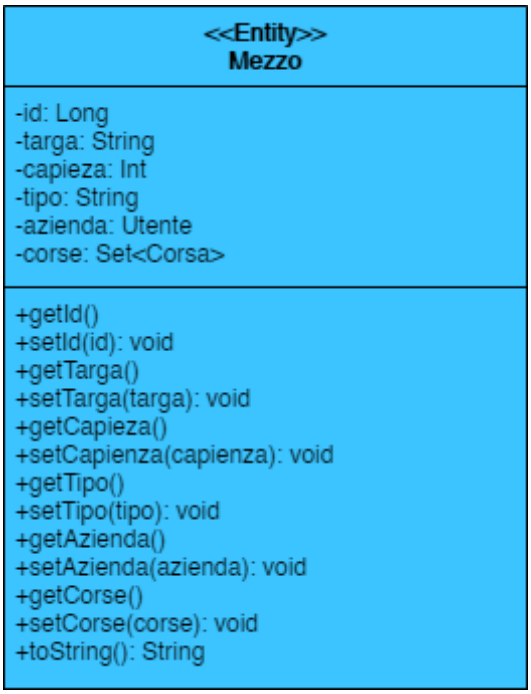
3.3 Gestione Risorse

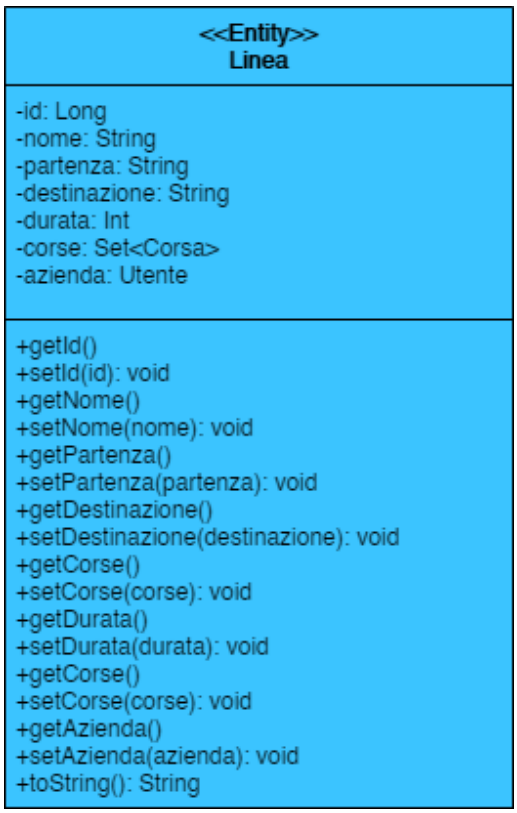


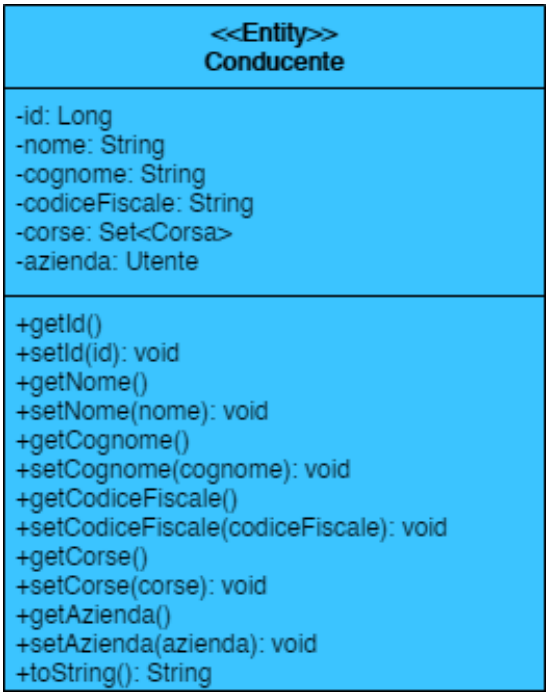

DIPARTIMENTO DI INFORMATICA
UNISA.IT


<<Service>>
RisorseService
conducenteRepository: ConducenteRepository
lineaRepository: LineaRepository
mezzoRepository: MezzoRepository
+addMezzo(Mezzo mezzo): void
+addConducente(Conducente conducente): void
+addLinea(Linea linea): void
+getConducenti(): List<Conducente>
+getConducentiByAzienda: (Utente utente): List<Conducente>
+getConducente(Long id): Optional<Conducente>
+getLinee():List<Linea>
+getLineeByAzienda(Utente utente): List<Linea>
+getLinea(Long id): Optional<Linea>
+getMezzi(): List<Mezzo>
+getMezziByAzienda(Utente utente): List<Mezzo>
+getMezzo(Long id): Optional<Mezzo>
+updateMezzo(Mezzo mezzo): void
+updateLinea(Linea linea): void
+updateConducente(Conducente conducente): void
+deleteMezzo(Mezzo mezzo): void
+deleteLinea(Linea linea): void
+deleteConducente(Conducente conducente): void
+getConducenteByCodiceFiscale(String codiceFiscale): Conducente
+getOwnership(Risorsa risorsa, Utente utente): boolean
+getLineaByName(String nome): Optional<Linea>
+isBound(Risorsa risorsa): boolean

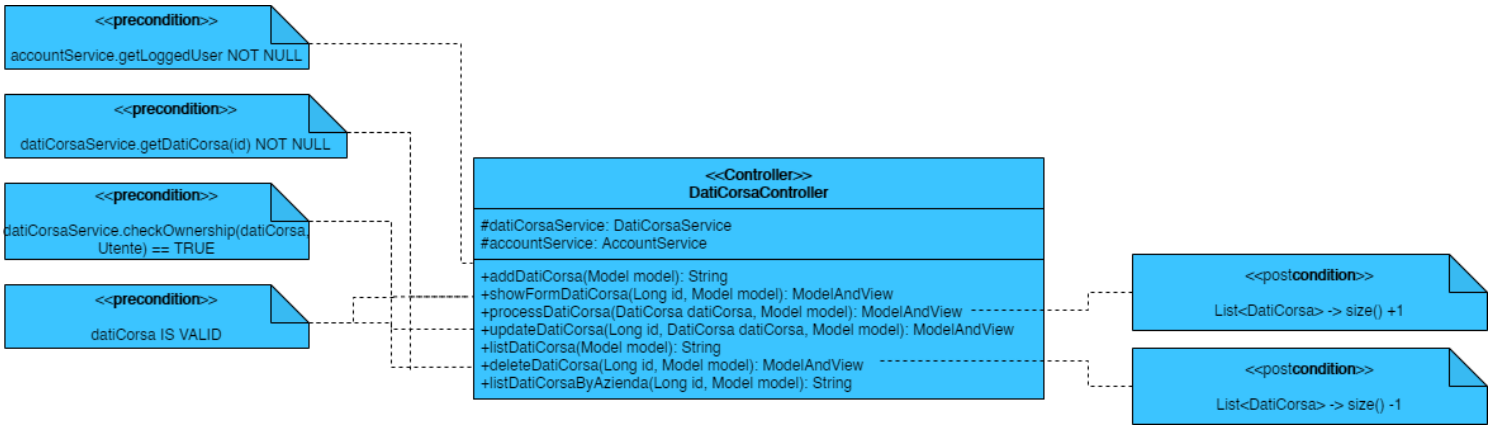

<<Abstract>>
Risorsa

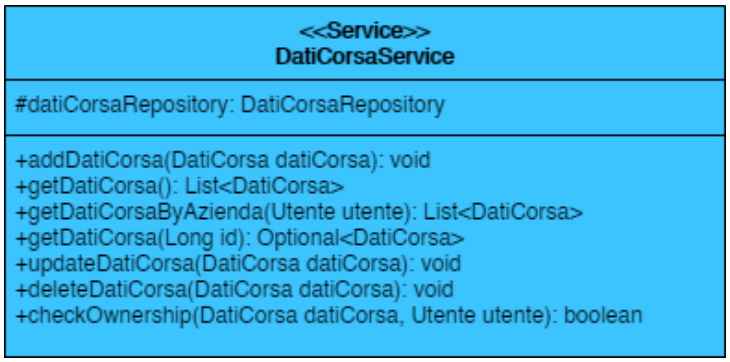

<<Entity>>
Mezzo
-id: Long
-targa: String
-capieza: Int
-tipo: String
-azienda: Utente
-corse: Set<Corsa>
+getId()
+setId(id): void
+getTarga()
+setTarga(targa): void
+getCapieza()
+setCapienza(capienza): void
+getTipo()
+setTipo(tipo): void
+getAzienda()
+setAzienda(azienda): void
+getCorse()
+setCorse(corse): void
+toString(): String

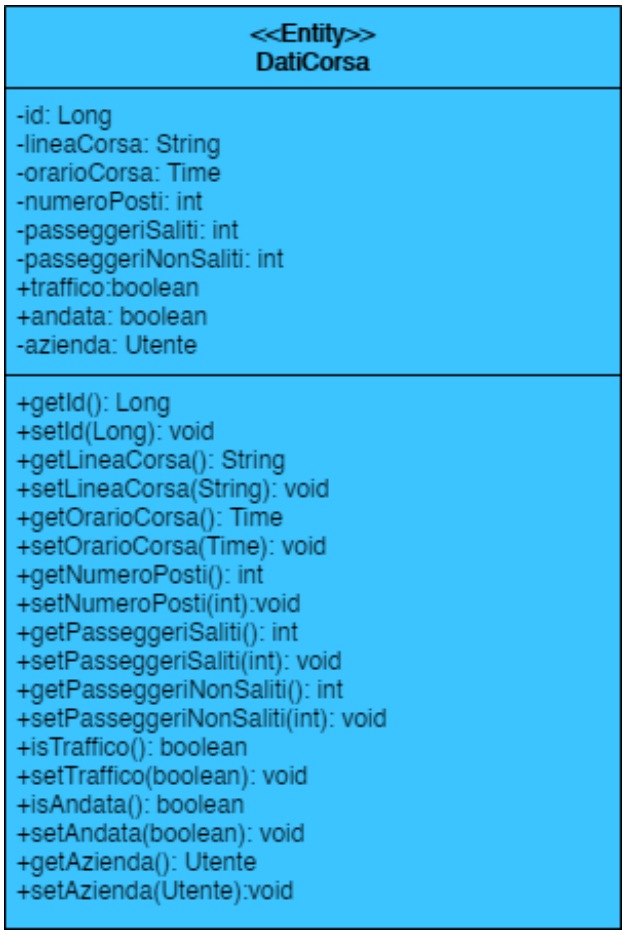

<<Entity>>
Linea
-id: Long
-nome: String
-partenza: String
-destinazione: String
-durata: Int
-corse: Set<Corsa>
-azienda: Utente
+getId()
+setId(id): void
+getNome()
+setNome(nome): void
+getPartenza()
+setPartenza(partenza): void
+getDestinazione()
+setDestinazione(destinazione): void
+getCorse()
+setCorse(corse): void
+getDurata()
+setDurata(durata): void
+getCorse()
+setCorse(corse): void
+getAzienda()
+setAzienda(azienda): void
+toString(): String


<<Entity>>
Conducente
-id: Long
-nome: String
-cognome: String
-codiceFiscale: String
-corse: Set<Corsa>
-azienda: Utente
+getId()
+setId(id): void
+getNome()
+setNome(nome): void
+getCognome()
+setCognome(cognome): void
+getCodiceFiscale()
+setCodiceFiscale(codiceFiscale): void
+getCorse()
+setCorse(corse): void
+getAzienda()
+setAzienda(azienda): void
+toString(): String

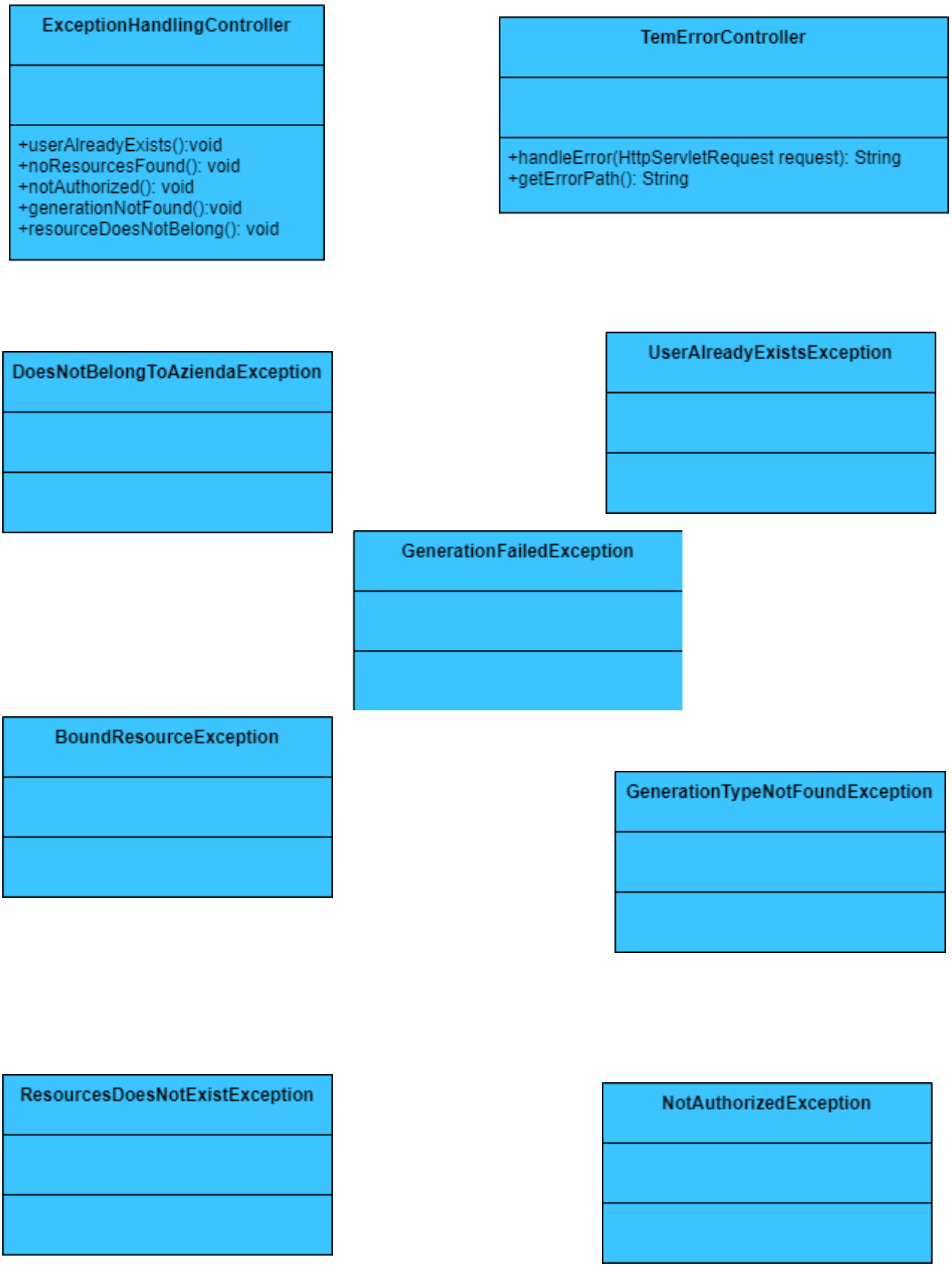
3.4 Gestione Dati delle singole corse







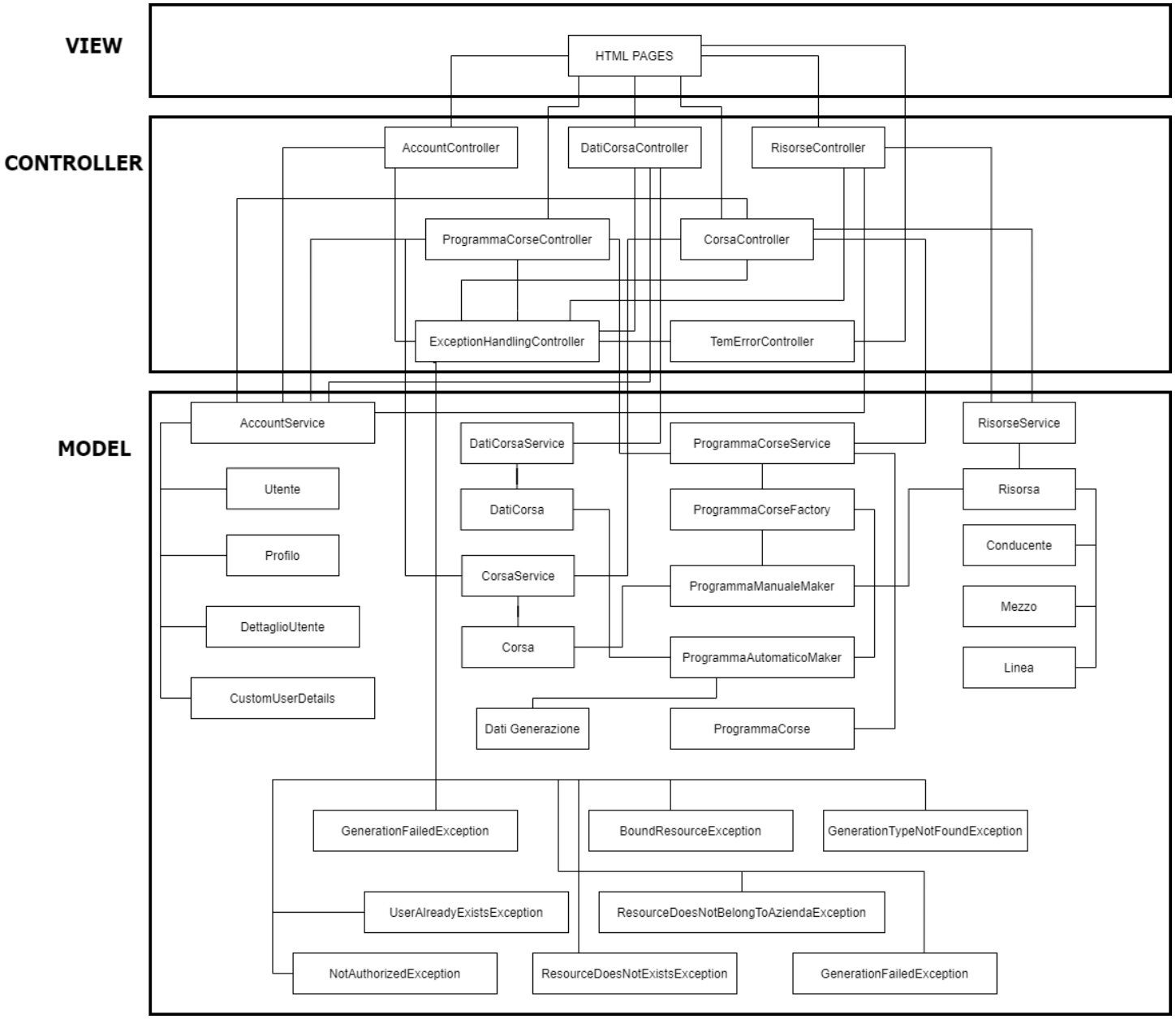
3.5 Gestione delle eccezioni



4. Class diagram

È riportato, di seguito, il Class diagram relativo all'implementazione del progetto.

Per agevolarne la visione, sono stati riportati solo i nomi delle classi, evitando eventuali specifiche già riportate nella sezione precedente.



5. Glossario

Object Design Trade-Offs: Decisioni sulla priorità da assegnare tra design goals che potrebbero andare in conflitto.

Class Interfaces: Descrive le classi e le loro interfacce pubbliche (overview di ogni classe, sue dipendenze con altre classi e package, i suoi attributi e operazioni pubblici, casi eccezionali)

Design pattern: descrivono object design parziali che risolvono questioni specifiche.

Database: Sistema di memorizzazione dati.

HTML: Linguaggio di programmazione utilizzato per lo sviluppo di pagine Web.

CSS: Linguaggio per la definizione degli stili delle pagine web Framework: Software di supporto allo sviluppo.

Framework: Software di supporto allo sviluppo.

Off-The-Shelf: Servizi esterni al sistema di cui viene fatto utilizzo.