



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

---

## **Ita Airways, Software di gestione operativa**

---

Autori:

Loris Vettori

Vincenzo Marturano

Corrado Duccio Piccioni

*Corso principale:*

Ingegneria del software

Matricole:

7073680

7077652

7077207

*Docente Corso:*

Enrico Vicario

## **Indice:**

- 1. Introduzione**
  - 1.1. Statement**
  - 1.2. Architettura**
  - 1.3. Strumenti utilizzati**
- 2. Progettazione**
  - 2.1. Use Case Diagram**
  - 2.2. Use Case Template**
  - 2.3. Class Diagram**
  - 2.4. Page Navigation Diagrams**
  - 2.5. Database**
    - 2.5.1. Dizionario dei dati**
    - 2.5.2. Schema Entity-Relationships**
    - 2.5.3. Schema logico**
- 3. Implementazione**
  - 3.1. Data Access Object (DAO)**
  - 3.2. DB**
    - 3.2.1. DBConfig**
    - 3.2.2. PgDB**
  - 3.3. Domain Model**
    - 3.3.1. Aircraft**
    - 3.3.2. Airport**
    - 3.3.3. DimensionClass**
    - 3.3.4. Credentials**
    - 3.3.5. EmployeeRole**
    - 3.3.6. Employee**
    - 3.3.7. FlightRoute**
    - 3.3.8. Flight**
  - 3.4. System (business logic)**
    - 3.4.1. Scheduling**
    - 3.4.2. SimulatedClock**
    - 3.4.3. FlightSchedule**
    - 3.4.4. ManagementSystem**

### **3.5. Interfaccia (CLI)**

## **4. Testing**

# 1 Introduzione

## 1.1 Statement

La compagnia aerea ITA Airways necessita di un software per la gestione giornaliera dell'impiego dei suoi velivoli nelle rotte offerte. Si ha esigenza di includere l'assegnazione degli aeromobili adeguati e del personale alle tratte, compatibilmente con la distanza dalla destinazione e con il numero di passeggeri, in ottemperanza della capacità di un aeroporto di accogliere aerei di una determinata classe.

I velivoli, di varia tipologia, devono essere adatti alla tratta, con criteri quali: numero di biglietti prenotati (numero di posti maggiore o uguale alle prenotazioni), range dell'aereo (autonomia dichiarata dalla casa produttrice, che deve consentire l'esecuzione senza scali della tratta), classe degli aeroporti di arrivo, scalo e partenza che devono essere in grado di accogliere quella specifica categoria di aeromobili.

A ciascun volo viene assegnato un aeromobile, in base al tipo del quale l'ENAC (Ente Nazionale Aviazione Civile), oltre a comandante e primo ufficiale, prevede un numero minimo di assistenti di volo per poter autorizzare il decollo. Altra norma dell'aviazione è che, per un volo superiore alle nove ore, ci siano due equipaggi presenti in cabina di pilotaggio.

L'abilitazione dei piloti è strettamente legata al modello di aereo. Eccezione risiede nella famiglia Airbus A320: un pilota abilitato per tale modello, può infatti mettersi ai comandi anche di A318, A319, A320 e A321, dovuto al fatto che sono aerei molto simili tra loro. Nel caso in cui un pilota si abiliti per un nuovo velivolo, è previsto il decadimento del brevetto precedente.

Ciascun velivolo è tracciato; in particolare è necessario conoscere, in ogni momento, se stia effettuando una tratta o se si trovi parcheggiato in qualche aeroporto. E' il pilota a comunicare alla compagnia il distacco dal gate e l'atterraggio. Tramite un orologio simulato è possibile interagire con la funzione di monitoraggio degli aerei, conoscendo lo stato dei voli in ogni momento.

## 1.2 Architettura

Si è prevista la suddivisione del progetto in diversi package per garantire la suddivisione delle mansioni delle diverse sezioni del codice.

I package sono i seguenti:

- DAO (Data Access Object): insieme di strutture dati che vengono generate a partire dal database
- DB: contiene le funzioni necessarie a interagire col db e le query che vengono impiegate (formattazione parziale)
- Model: package contenente le strutture dati degli oggetti fondamentali, tramite le quali viene implementato il Design Pattern CRUD (Create Read Update Delete)

- System: raccoglie le strutture dati di gestione del sistema, tra cui l'algoritmo di scheduling dei voli, l'orologio simulato e le varie classi manager
- CLI (Command Line Interface): interfaccia di interazione con l'utente

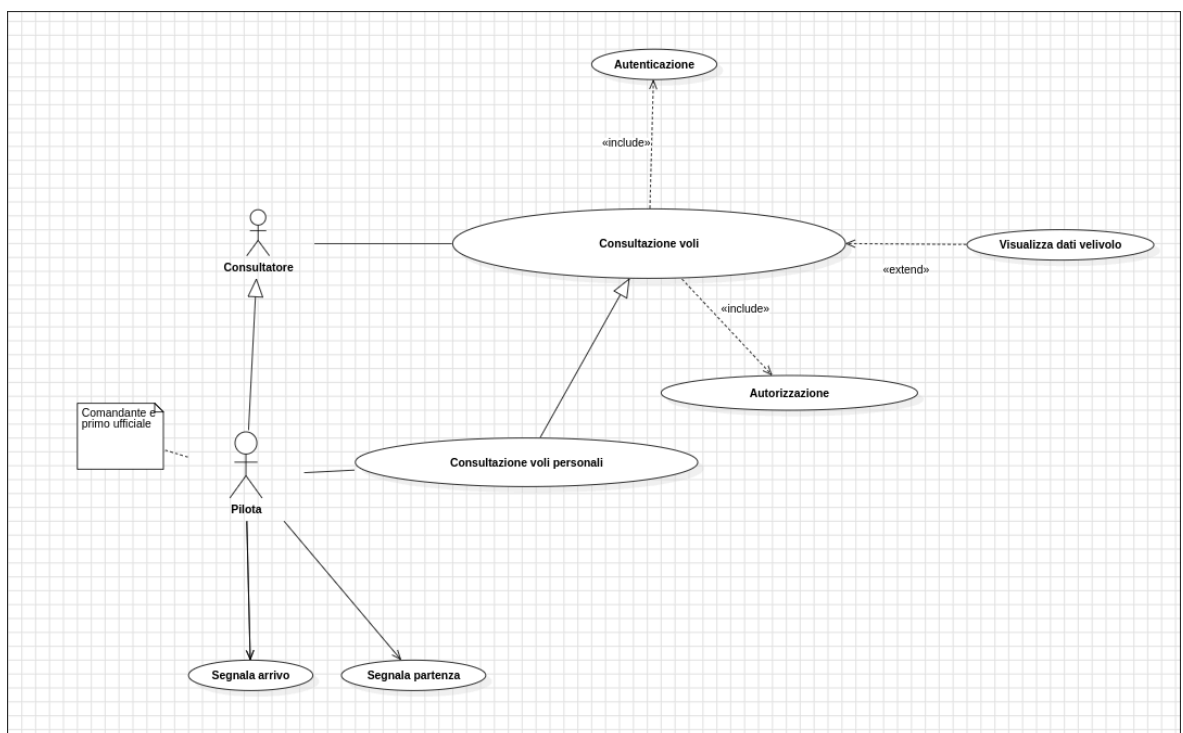
### 1.3 Strumenti utilizzati

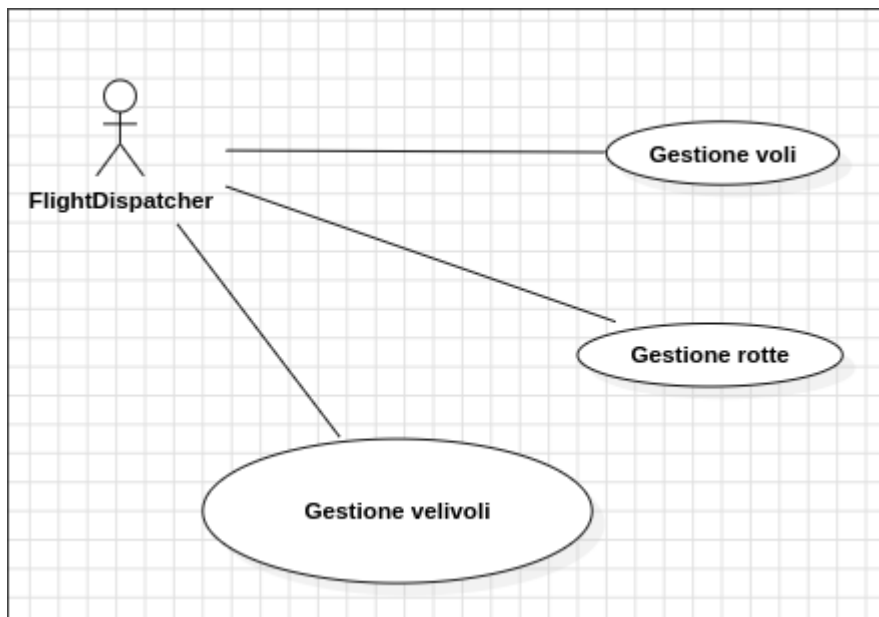
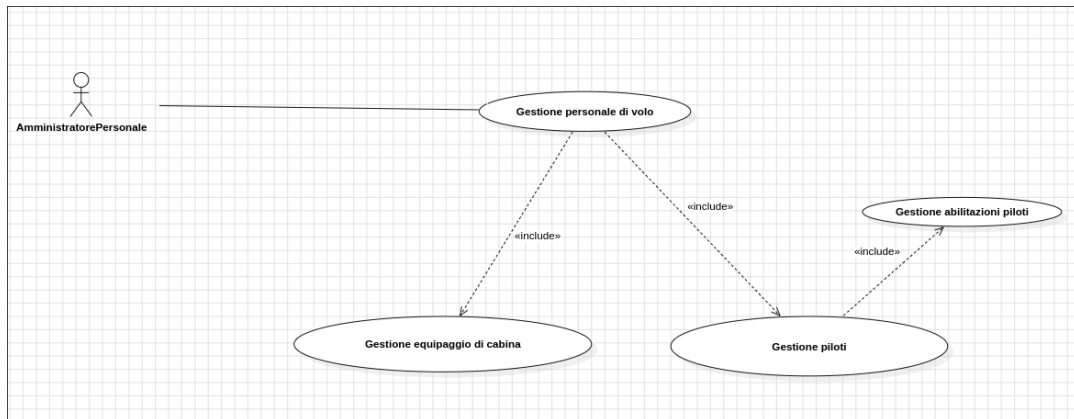
Il software è scritto in linguaggio Java, i dati sono classificati e gestiti tramite un database PostgreSQL; l'interazione tra codice e database è eseguita tramite la libreria JDBC.

Gli Use Case Diagrams e i Class Diagrams rispettano lo standard UML (Unified Modeling Language), e sono stati realizzati mediante il software StarUML.

## 2 Progettazione

### 2.1 Use case diagrams





## 2.2 Use case template

### 2.2.1 Consultazione voli personali

<b>UC</b>	Consultazione voli personali
<b>Scope</b>	
<b>Level</b>	User goal
<b>Actor</b>	Pilota
<b>Basic course</b>	<ol style="list-style-type: none"> <li>1. Il pilota avvia l'applicazione.</li> <li>2. Esegue l'accesso alla piattaforma.</li> </ol>

	3. Seleziona "4 -> Access flight schedule". 4. Consulta i propri voli.
<b>Alternative course</b>	

### 2.2.2 Gestione personale: aggiungere un nuovo dipendente

<b>UC</b>	Gestione personale: aggiungere un nuovo dipendente
<b>Scope</b>	
<b>Level</b>	User goal
<b>Actor</b>	Amministratore personale
<b>Basic course</b>	<ol style="list-style-type: none"> <li>1. L'Amministratore del personale Avvia l'applicazione</li> <li>2. Seleziona "1 -&gt; Access employees data"</li> <li>3. Selezione <ul style="list-style-type: none"> <li>• "6 -&gt; Edit specific employee"</li> </ul> </li> <li>4. Seleziona: <ul style="list-style-type: none"> <li>• "1 -&gt; Insert"</li> </ul> </li> <li>5. Inserisce il nome</li> <li>6. inserisce il cognome.</li> <li>7. Sceglie il ruolo tra <ul style="list-style-type: none"> <li>○ "1 -&gt; Commander"</li> <li>○ "2 -&gt; First Officer"</li> <li>○ "3 -&gt; Flight Assistance"</li> </ul> </li> <li>8. Se ha selezionato "Commander" o "First Officer" ne specifica</li> </ol>

	l'abilitazione.
<b>Alternative course</b>	8.a Viene inserita un'abilitazione errata. 8.a.1 Il sistema chiede di reinserire l'abilitazione fino a quando è corretta.

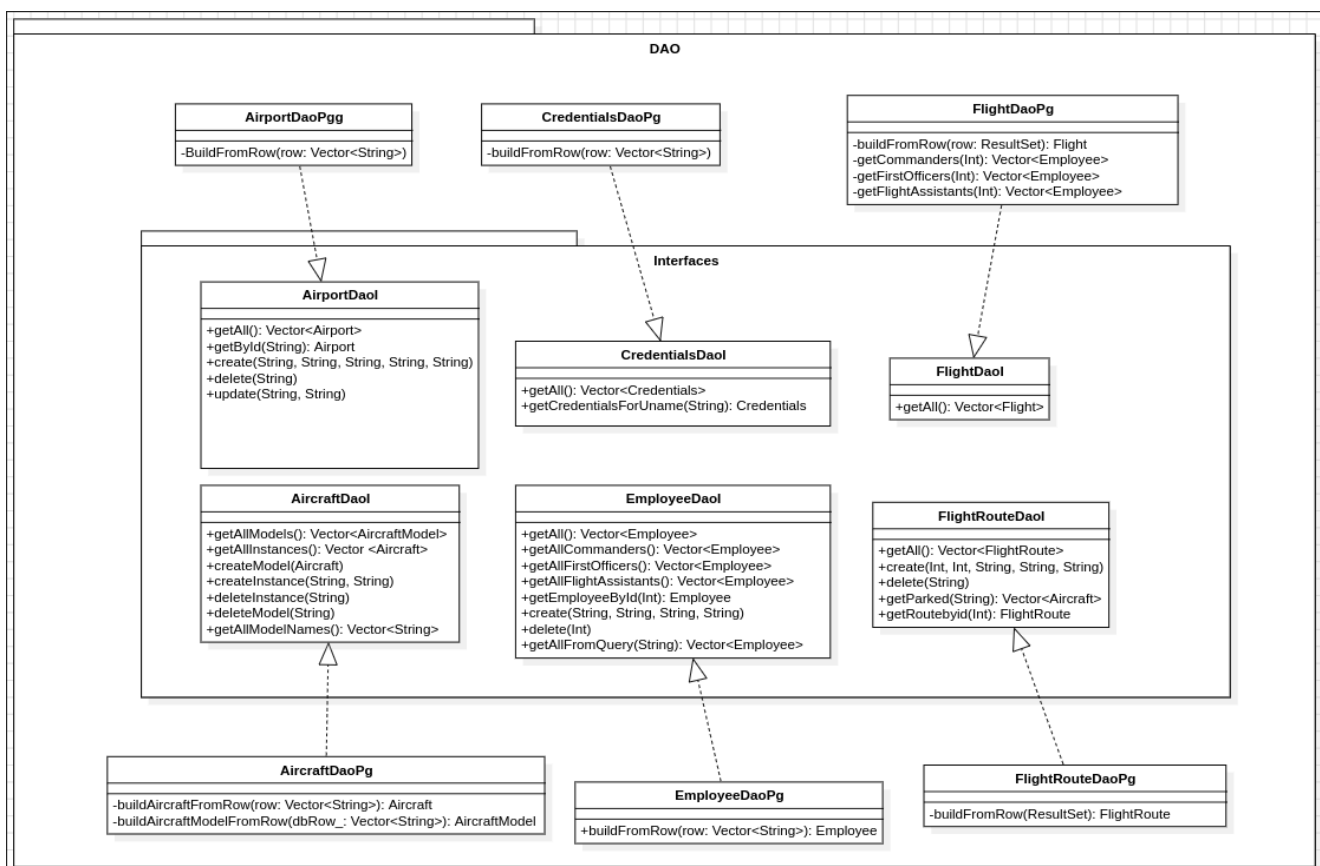
### 2.2.3 Gestione aeromobili: aggiungere un nuovo aeromobile.

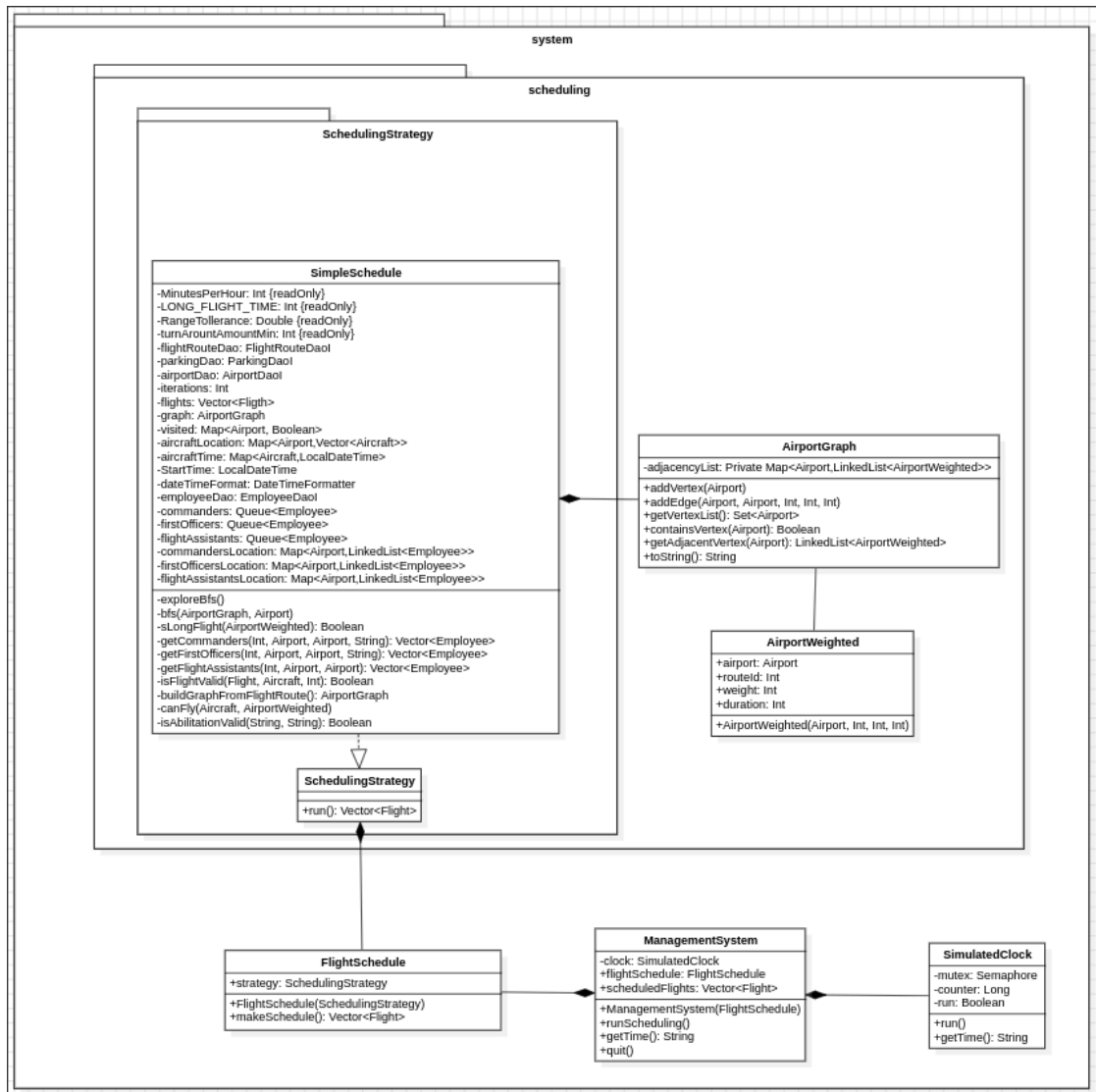
<b>UC</b>	Gestione aeromobili: rimozione di un aeromobile.
<b>Scope</b>	
<b>Level</b>	User goal
<b>Actor</b>	Flight dispatcher
<b>Basic course</b>	<ol style="list-style-type: none"> <li>1. Il flight dispatcher avvia l'applicativo.</li> <li>2. Esegue l'accesso alla piattaforma.</li> <li>3. Seleziona "2 -&gt; Access aircrafts details"</li> <li>4. Seleziona "2 -&gt; Edit aircrafts"</li> <li>5. Seleziona "2 -&gt; Remove"</li> <li>6. Seleziona "2 -&gt; delete Aircraft-instance"</li> <li>7. Inserisce la targa dell'aeromobile da rimuovere</li> </ol>
<b>Alternative course</b>	7.a la targa non esiste 7.a.1 viene stampato un

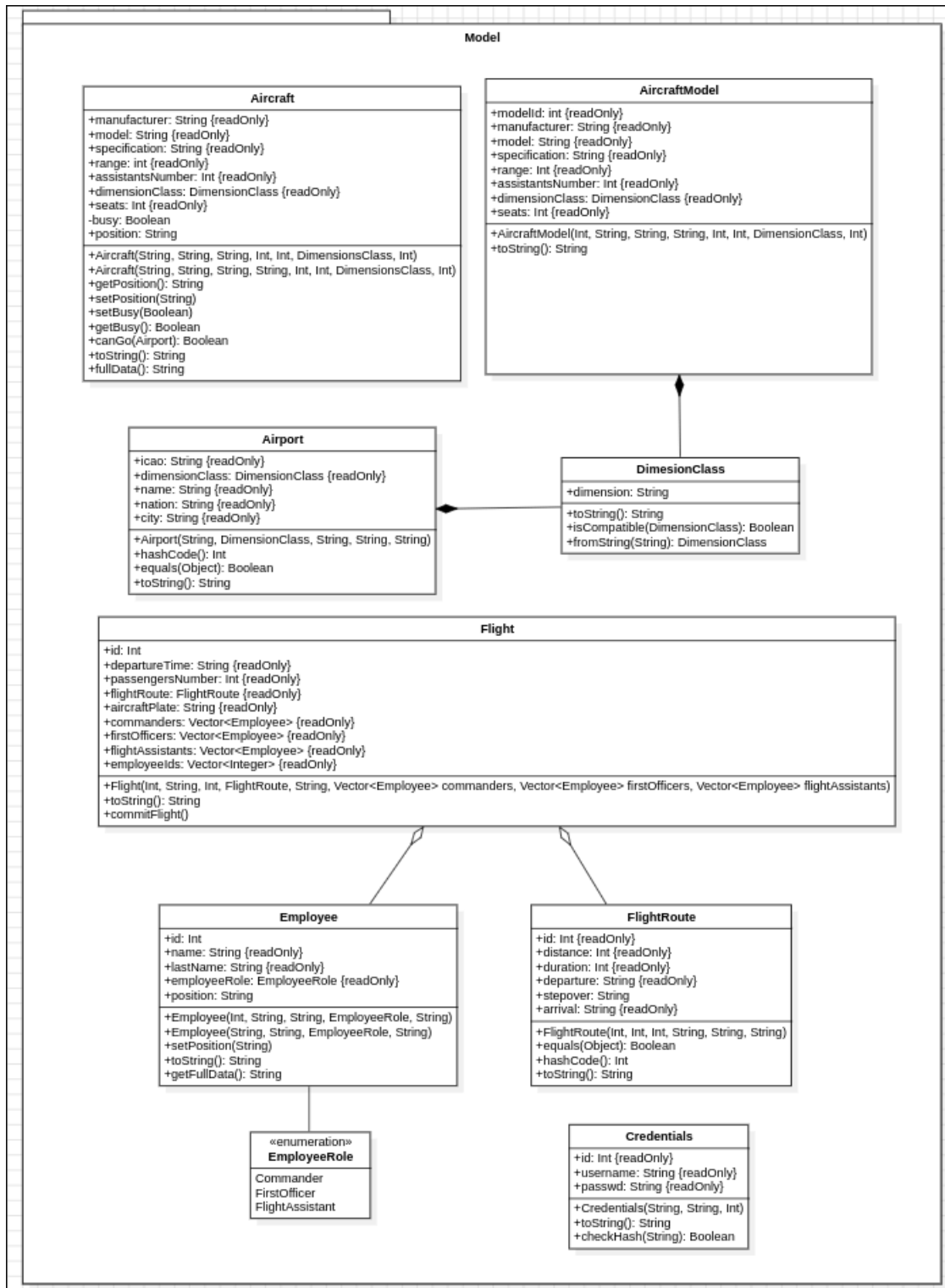


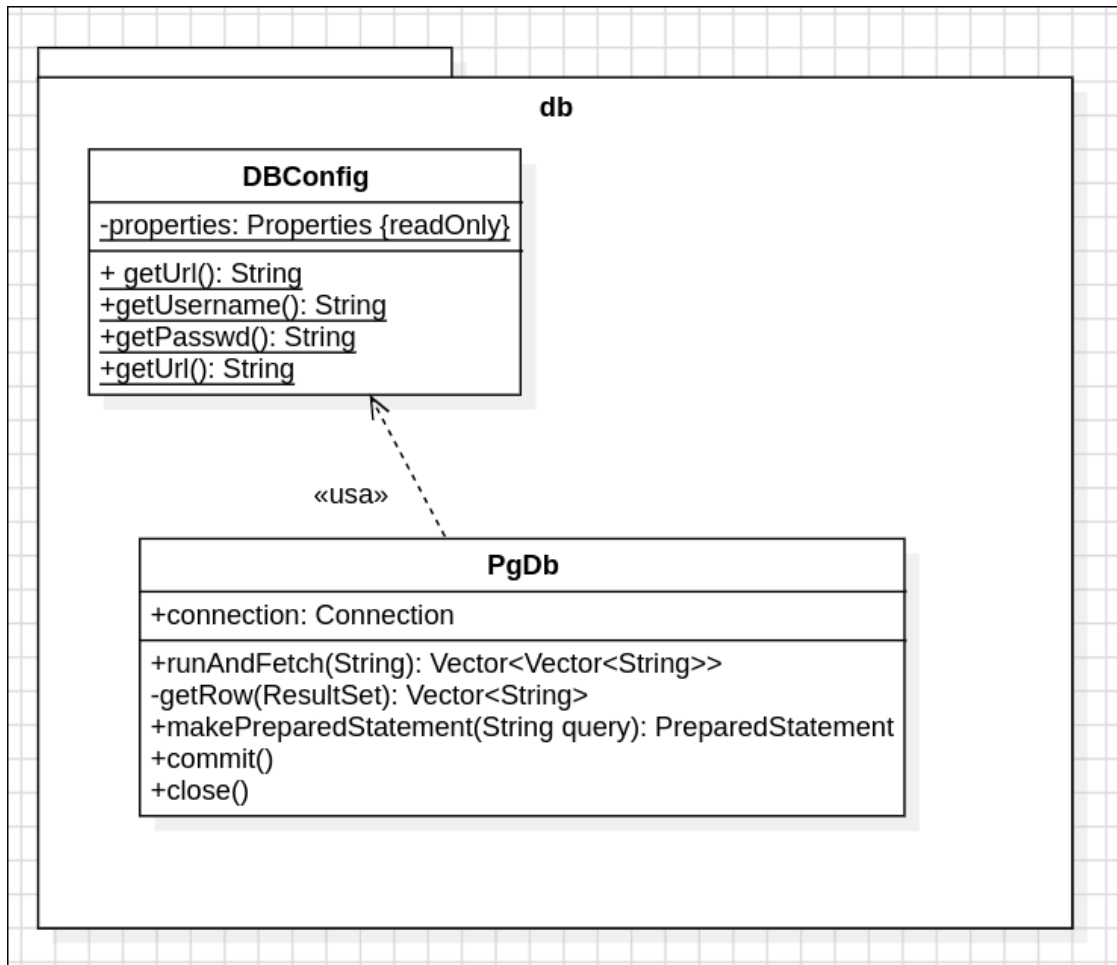
messaggio di errore e l'utente viene portato al menu precedente.

## 2.3 Class Diagram



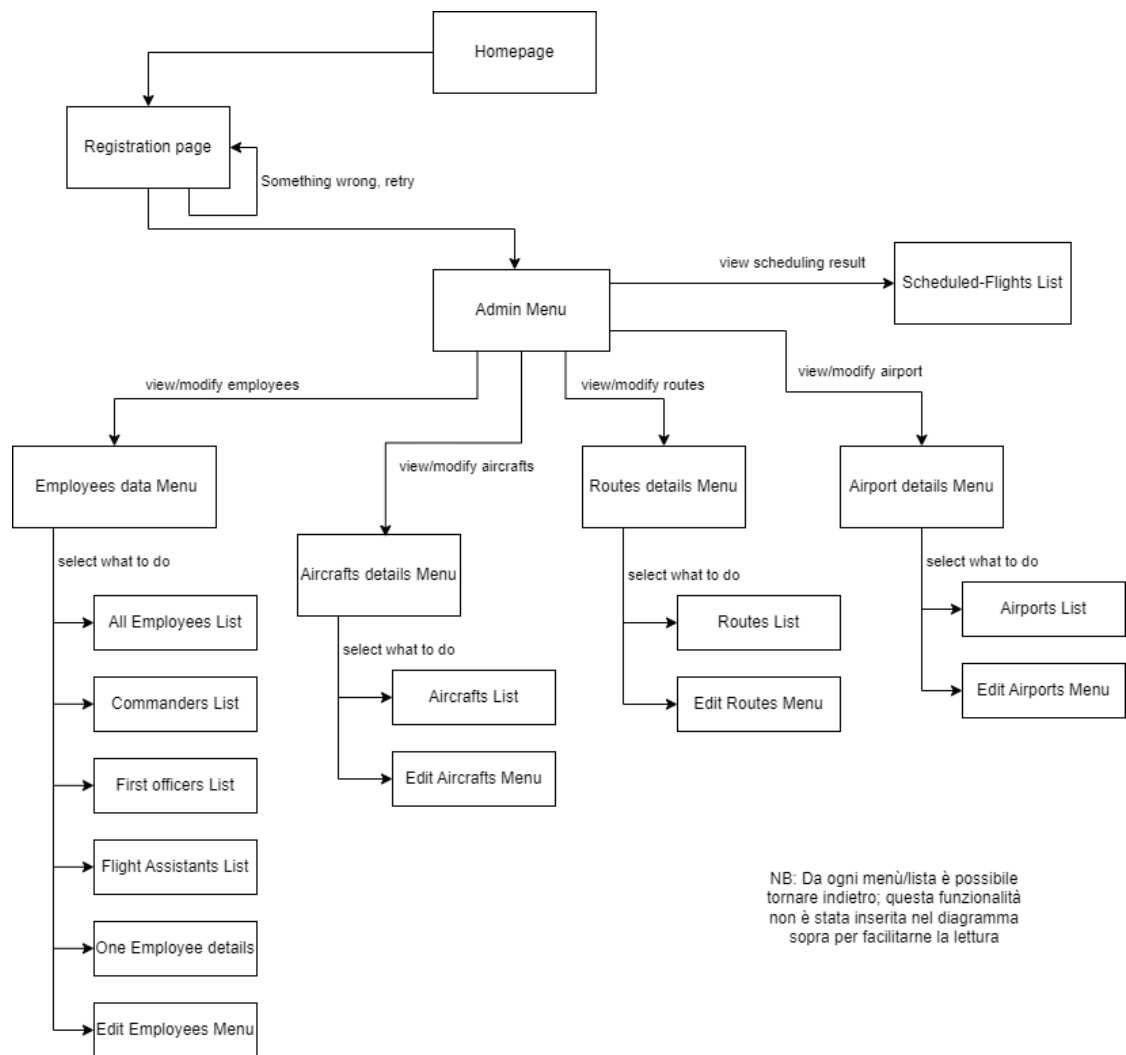




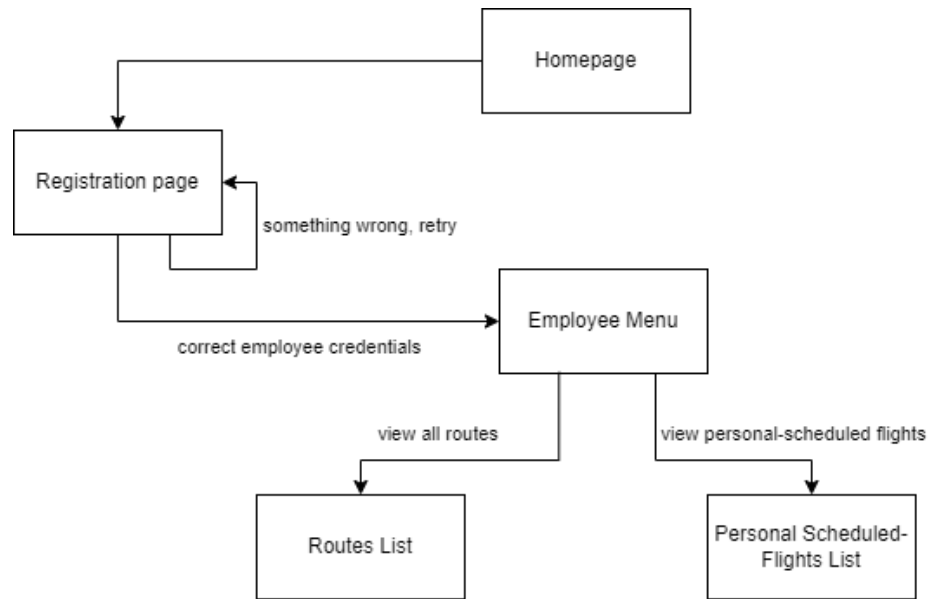


## 2.4 Page Navigation Diagrams

### Page Navigation diagram Admin

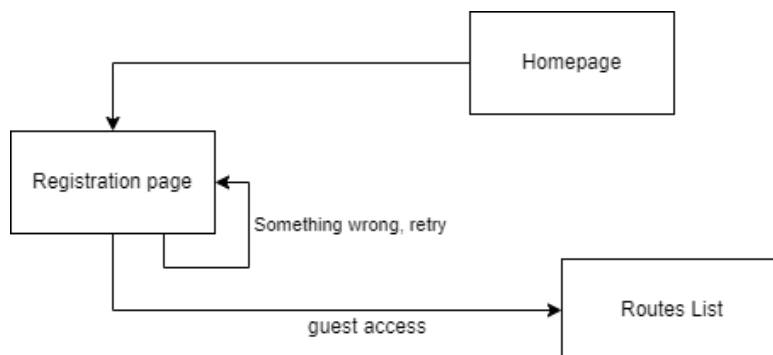


**Page Navigation diagram Employee**



NB: Da ogni menù/lista è possibile tornare indietro; questa funzionalità non è stata inserita nel diagramma sopra per facilitarne la lettura

## Page Navigation diagram Guest



NB: Da ogni menù/lista è possibile tornare indietro; questa funzionalità non è stata inserita nel diagramma sopra per facilitarne la lettura

## 2.5 Database

### 2.5.1 Dizionario dei dati

#### Entità dello schema ER

Entità	Descrizione	Attributi	Identificatore
Aircraft	Tipologia di aeromobile	id, manufacturer, model, specification, range, assistants_number, class, seats	id
Aircraft_instance	Specifico aeromobile	plate	plate
Airport	Aeroporto nel mondo	icao, class, name, nation, city	icao
Route	Tragitto collegato dalla compagnia	id, distance, duration	id
Flight	Specifico volo	id, departure_time, passengers_number	id
Personal	Operatori della compagnia	id, name, lastName, role, abilitation	id

#### Relazioni dello schema

Nome	Descrizione	Entità coinvolte e cardinalità
belong*	Associa a ciascun modello di aeromobile le istanze possedute dalla	Aircraft(0,N) - Aircraft_instance(1,1)

	compagnia	
details_dep*	Associa a ciascun aeroporto di partenza tutti i dettagli	Route(1,1)-Airport(0,N)
details_step*	Associa a ciascun aeroporto di scalo tutti i dettagli	Route(0,1)-Airport(0,N)
details_arr*	Associa a ciascun aeroporto di arrivo tutti i dettagli	Route(1,1)-Airport(0,N)
wait*	Associa a ciascun aeromobile l'aeroporto dove è ubicato	Airport(0,N) - Aircraft_instance(0,N)
concern*	Associa ciascun volo specifico alla corrispondente rotta	Flight(1,1)-Route(0,N)
command*	Assegna comandante (-i nel caso di voli di grande durata) a ciascun volo	Flight(1,2)-Personal(0,N)
monitor*	Assegna primo ufficiale (-i nel caso di voli di grande durata) a ciascun volo	Flight(1,2)-Personal(0,N)
assist*	Assegna a ciascun volo steward/hostess	Flight(N,N)-Personal(0,N)
operate*	Assegna a ciascun volo un velivolo	Flight(1,1)-Aircraft_instance(0,N)

## Regole di vincolo

Aircraft\_instance:

- L'istanza deve far riferimento ad un modello di aereo esistente.

Route:

- L'aeroporto di partenza di una rotta deve esistere tra gli aeroporti del database
- L'aeroporto di scalo (se diverso da null) deve esistere tra gli aeroporti del database
- L'aeroporto di partenza di una rotta deve esistere tra gli aeroporti del database

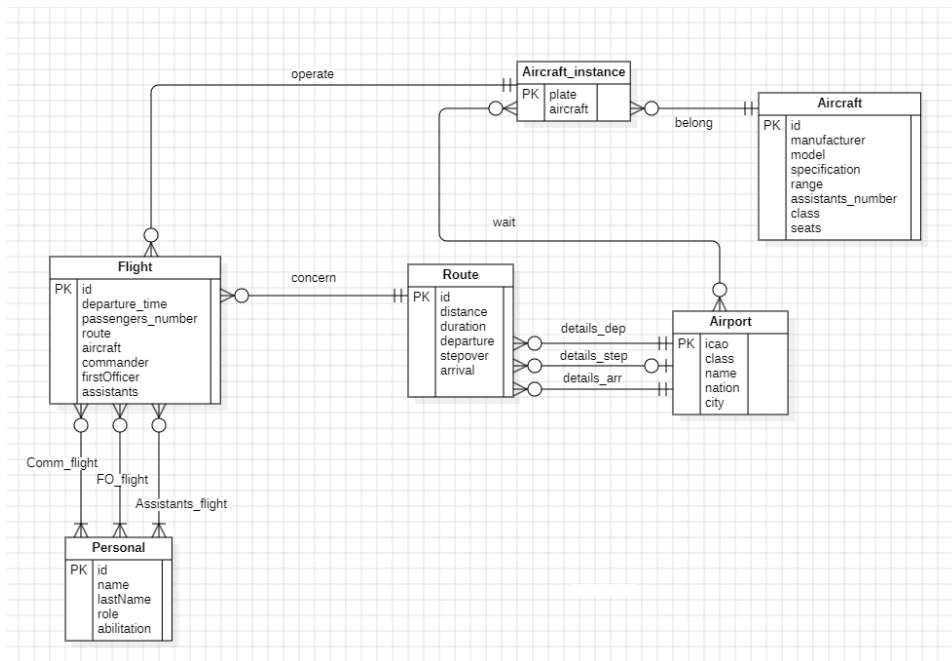
Flight:

- Il comandante/i di un volo deve esistere tra il personale della Compagnia



- Il/i primo ufficiale/i di un volo deve esistere tra il personale della Compagnia
- Gli assistenti di volo devono esistere tra il personale della compagnia
- Un volo deve riguardare una delle rotte offerte dalla compagnia
- L'aeromobile che opera il volo deve comparire tra gli aerei in possesso.

## 2.5.2 Schema Entity-Relationships



## Ristrutturazione dello schema E-R

Dato che non si identificano ridondanze, gerarchie tra le tabelle, attributi multivalore e gli identificatori sono già stati definiti per ciascuna tabella, possiamo considerare la fase di ristrutturazione come già implicitamente completata.

## 2.5.3 Schema logico

Aircraft (id, manufacturer, model, specification, range, assistants\_number, class, seats);

Aircraft\_instance (plate, aircraft IR\*)

Airport(icao, class, name, nation, city)

Route(id, distance, duration, departure IR\* , stepover IR, arrival IR\*)

Parking (aircraft\_instance IR, airport IR)

Flight (id, departure\_time, passenger\_number, route IR\*, aircraft\_instance IR\*)

Command(personal IR, flight IR )

Monitor(personal IR, flight IR )

Assist (personal IR, flight IR)

Legenda:

IR: *sull'attributo indicato vige un vincolo di Integrità Referenziale*

\* *attributo not null*

## 3 Implementazione

### 3.1 Data Access Object (DAO)

Il DAO (Data Access Object) è a tutti gli effetti un pattern architetturale usato nelle applicazioni con lo scopo di separare la logica di accesso al Database dal resto delle responsabilità. Non facendo questa divisione si andrebbe a scrivere del codice poco manutenibile. Volendo citare anche un altro pattern, si noti come ogni classe di questo tipo sia a tutti gli effetti una Factory che produce le istanze necessarie dei relativi modelli.

Il sub-package interfaces contiene le interfacce astratte che verranno poi implementate nelle classi fisiche appartenenti al pacchetto DAO. L'utilizzo di quest'ultime, oltre ad essere una buona pratica di programmazione, agevola anche la fase di unit testing, rendendola più semplice ed efficace.

Di seguito andiamo a riportare il codice per una sola di queste, dato che questo è simile tra le varie classi.

```
public interface AirportDaoI {
    public Vector<Airport> getAll();
    public Airport getById(String icao);
    public void create(String icao, String dimensionClass, String name, String nation, String city);
    public void delete(String icao);
    public void update(String icao, String dimensionClass);
}
```

Le varie classi **\*DaoPg** implementano la relativa interfaccia (specificata nel sub-package **\*interfaces\***)

I metodi dichiarati obbligano l'implementazione (parziale o totale) del design pattern CRUD.

### 3.2 DB

Come già spiegato sopra, questo package nasce con lo scopo di connettere l'applicazione con il database; Contiene anche le query che vengono impiegate.

### 3.2.1 DBConfig

In questa classe viene localizzato e caricato il file di configurazione del database.

#### Metodo costruttore

La prima parte del metodo localizza e restituisce il file “`db.properties`”, contenuto sempre nella cartella dell’applicazione (in questo file sono presenti l’url\*, del database, username e password per l’accesso ad esso).

La localizzazione avviene mediante `DBConfig.class.getClassLoader().getResourceAsStream("Absolute path")`.

Una volta localizzato il file si caricano i dati di configurazione del collegamento, per far questo si usa il metodo `Properties.load()`.

Nel caso in cui il file chiaramente non viene localizzato per qualsiasi motivo, si genera un errore fatale, che impedisce la corretta fruizione dell’applicazione

*\*acronimo di Uniform Resource Locator, è una sequenza di caratteri che identifica univocamente l’indirizzo di una risorsa*

#### Classici metodi getter

### 3.2.2 Classe PgDB

Classe di interazione fisica col DB, che permette di operare effettivamente con la base di dati.

#### Metodo costruttore

Prima si recuperano le tre stringhe fondamentali della classe DBConfig, e una volta fatto questo, il metodo tenta di stabilire la connessione vera e propria. Si carica il primo driver disponibile e in grado di connettersi alla risorsa dalla lista JDBC (Java DataBase Connectivity) e si stabilisce la connessione usando il metodo `getConnection()`.

Anche in questo processo un fallimento è assolutamente terminale, non è possibile infatti compiere operazioni di alcun tipo senza avere a disposizione il database, nel caso in cui questo avvenga si notifica all’utente la natura dell’errore e si chiude il programma.

```
public PgDB() {
    try {
        String url = DBConfig.getUrl();
        String username = DBConfig.getUsername();
        String passwd = DBConfig.getPasswd();

        this.connection = DriverManager.getConnection(url, username, passwd);
    } catch (SQLException e) {
        System.err.println("Fatal error: postgresql db error: " + e.getMessage());
        System.exit(1);
    }
}
```

### **Metodo `Vector<Vector<String>> runAndFetch(String query)`**

E' il metodo principale e la responsabilità primaria che viene affidata a questa classe. Riceve in input la query, sotto forma di stringa, che si vuole eseguire sul Database connesso e restituisce i risultati indicati.

Innanzitutto si istanzia un oggetto di tipo `Statement` per l'esecuzione di istruzioni al runtime. Una volta fatto questo ne viene chiamato il metodo `executeQuery(query)` il quale esegue effettivamente la query; i risultati vengono salvati nella tabella virtuale `result`.

Un errore, esso sia nella procedura di generazione della query che in quella di estrazione di risultati dalla base di dati, provoca in ogni caso la chiusura dell'applicazione, con ovviamente annessa notifica di cosa ha generato il crash.

```

public Vector<Vector<String>> runAndFetch(String query) {
    Vector<Vector<String>> result = new Vector<>();

    try {
        Statement statement = this.connection.createStatement();

        try {
            var dbOutput = statement.executeQuery(query);

            while (dbOutput.next()) {
                Vector<String> row = getRow(dbOutput);
                result.add(row);
            }

            return result;
        } catch (SQLException ex) {
            System.err.println("fatal error: query \"" + query + "\" generated error: " + ex.getMessage());
            System.exit(1);
        }

    } catch (SQLException ex) {
        System.err.println("fatal error: statement creation generated error: " + ex.getMessage());
        System.exit(1);
    }

    return result;
}

```

### Metodo **Vector<String> getRow(ResultSet dbOutput)**

Permette l'estrazione di righe da un **ResultSet**, un tipo di oggetto generato in seguito all'esecuzione di una query su un Database. Quindi si esegue la query, i risultati vengono inseriti in questa struttura dati, e poi se serve grazie a questo metodo se ne possono estrarre alcune righe.

Prima di tutto si esegue il metodo **getMetadata()**, una procedura particolare che permette di estrarre tutte le proprietà delle varie colonne (numero, tipo e caratteristiche varie) e poi esegue un ciclo dove estrae i dati campo per campo.

### Metodo **PreparedStatement makePreparedStatement(String query)**

Metodo usato per la creazione di una istruzione SQL precompilata e memorizzata in modo da poterla usare senza vincoli senza dover ricorrere ogni volta alla sua compilazione. In caso di errore nella generazione viene stampato il corrispondente messaggio di errore.

*Di seguito alcuni dei numerosi PreparedStatement usati nel programma*

```
public class PreparedStatementQueries {
    public static final String deleteAircraftModel= "DELETE FROM aircraft WHERE id=?";

    public static final String getAircraftModels= "SELECT id,manufacturer,model,specification,range,assistants_number, class,seats from aircraft";

    public static final String getCompanyAircrafts = "select ai.plate, a.manufacturer, a.model, a.specification, " +
        "a.range, a.assistants_number, a.class, a.seats " +
        "from aircraft_instance ai join aircraft a on " +
        "ai.aircraft = a.id";

    public static final String getCompanyEmployees = "select * from personal";

    public static final String getFlights = "select * from flight";

    public static String getCommandersForFlightID = "select commander from commanders where flight=?";

    public static String getFirstOfficersForFlightID = "select firstOfficer from first_officer where flight=?";

    public static String getFlightAssistantsForFlightID = "select assistant from flight_assistants where flight=?";

    public static String getEmployeeInfo = "select * from personal where id=?";

    public static final String getPersonalForFlight = "select f.id, c.commander, fo.first_officer, fa.assistant " +
        "from flight f, commanders c, first_officers fo, flight_assistants fa " +
        "where f.id = c.flight and f.id = fo.flight and f.id = fa.flight_assistants " +
        "order by f.id";

    public static final String getAirports = "select * from airport";

    public static final String getRoutes = "select * from route";

    public static String getRouteForID = "select * from route where id=?";

    public static final String getCredentials = "select * from credentials";

    public static final String commitFlight = "insert into flight (departure_time, passengers_number, route, aircraft) values (?, ?, ?, ?)";

    public static String getLastFlightId = "select id from flight where id=(select max(id) from flight)";

    public static final String getParkedAircrafts= "select ai.plate, a.manufacturer, a.model, a.specification, a.range, a.assistants_number, a.class, a.seats " +
        "from parking p " +
        "join aircraft_instance ai on ai.plate = p.aircraft " +
        "join aircraft a on ai.aircraft = a.id " +
        "where p.airport = ?";

    public static final String insertAirCraft = "INSERT INTO aircraft " +
        "(manufacturer, model, specification, range, assistants_number, class, seats) values (?, ?, ?, ?, ?, ?, ?)";

    public static final String insertAircraftInstance = "INSERT INTO aircraft_instance (plate, aircraft) values (?, ?)";

    public static final String insertAirport = "INSERT INTO airport (icao, class, name, nation, city) values (?, ?, ?, ?, ?)";

    public static final String deleteAirport = "DELETE FROM airport where icao=?";

    public static final String updateAirport = "UPDATE airport SET class=? WHERE icao=?";

    public static final String insertPersonal = "INSERT INTO personal (name, lastname, role, abilitation) values (?, ?, ?, ?)";

    public static final String deletePersonal = "DELETE FROM personal where id=?";

    public static final String insertRoute = "INSERT INTO route (distance, duration, departure, arrival) values (?, ?, ?, ?)";

    public static final String deleteRoute = "DELETE FROM route WHERE id=?";

    public static final String getAircraftId = "SELECT id FROM aircraft WHERE manufacturer=? and model=? and specification=?";

    public static final String deleteAircraftInstance = "DELETE FROM aircraft_instance WHERE plate=?";
}
```

## Metodo **void commit()**

Effettua il commit sul database delle modifiche applicate

## Metodo **void close()**

Chiude la connessione tra l'applicazione e il database e quindi, in cascata, chiude il programma notificando all'utente l'errore "fatale" dato dalla mancanza di tale collegamento.

## 3.3 Domain Model

### 3.3.1 Aircraft

Modella gli aeromobili specificando tutti i dettagli necessari per lo scheduling dei voli.

#### Metodo costruttore

Inizializza un oggetto di tipo Aircraft con i valori che vengono forniti in input. Gli attributi usati sono:

- **plate**, di tipo Stringa  
Talvolta chiamato erroneamente codice di registrazione o targa, in realtà ne esegue la medesima funzione; è costituito da una stringa di caratteri alfanumerici, impressi e ben visibili tipicamente vicino alla coda e sulle ali di un aeromobile, che lo identificano univocamente
- **manufacturer**, di tipo Stringa  
Contiene il nome del produttore dell'aereo. In aviazione commerciale i più comuni sono: Airbus, Boeing, Embraer e Bombardier.
- **model**, di tipo Stringa  
Indica il modello dell'aereo (es. A320, 737...)
- **specification**, di tipo Stringa  
Di ogni modello di aereo ne vengono realizzate diverse specifiche: la struttura generale dell'aereo è la medesima ma cambia la lunghezza della fusoliera (blocchi in meno o in più di sedili, motori/sistemi/controlli diversi). Ad esempio del Boeing 737 ne esistono moltissime specifiche: dal 737-100 che risale al 1967, passando per 200,300,400...fino alla versione più recente, il 737-MAX 10 in produzione dal 2017.
- **dimensionClass**  
*Vedi sezione appositamente dedicata*
- **assistantsNumber**, di tipo Integer  
Contiene il numero di assistenti di volo che sono richiesti dalla normativa per poter operare voli commerciali con quel velivolo. Una rule of the thumb, che ne impone un limite minimo, è quella di avere uno steward/hostess ogni 50 passeggeri. Ci sono poi aeromobili che per altri motivi (numero di porte, dimensione della fusoliera,...) ne possono richiedere anche un numero ad essa maggiore.
- **range**, di tipo Integer  
Autonomia del velivolo. Questo parametro, misurato in condizioni ideali (assenza di vento contrario, volo alla IAS (Indicated Air Speed) e altitudine ideali garantendo all'aereo la più grande autonomia possibile), non è da considerarsi in maniera precisa ma indicativa; ci sono infatti moltissimi parametri durante il volo che possono andare a ridurre il valore effettivo della distanza percorribile.
- **seats**, di tipo Integer  
Numero di passeggeri che un aeromobile può ospitare
- **busy**, di tipo Boolean

Attributo di tipo booleano, indica se l'aeromobile è impiegato in un volo (true) oppure è fermo e pronto ad operarne uno (**false**).

- **position**, di tipo String

Contiene il codice icao dell'aeroporto in cui è localizzato l'aeromobile (secondo la simulazione oraria scandita da **SimulatedClock**)

#### **Metodo **boolean canGo(Airport airport)****

Dato in input un aeroporto, restituisce vero se la dimensione dell'aereo è tale da poter atterrare in **airport**, falso se l'aeroporto è invece sottodimensionato rispetto all'aeromobile, che quindi non potrà operarvici.

#### **Metodo **String fullData()****

Restituisce una stringa contenente, sotto forma di elenco "attributo: valore", tutti i dati di un aeromobile.

#### **Classici metodi getter, setter e metodo **toString()****

### **3.3.2 Airport**

Modella tutti gli aeroporti usati dalla compagnia; per ognuno di essi sono specificate chiaramente solo le caratteristiche che si rivelano utili per lo scheduling.

#### **Metodo costruttore**

Inizializza un oggetto di tipo Airport con i valori che gli vengono forniti in input. Gli attributi sono:

- **ICAO**, di tipo Stringa.

Contiene il codice alfabetico , assegnato dall'International Civil Aviation Organization, che identifica univocamente ciascun aeroporto (ad esempio l'aeroporto di Firenze Peretola ha ICAO LIRQ)

- **dimensionClass**

*Vedi sezione appositamente dedicata*

- **name**, di tipo Stringa

Contiene il nome commerciale dell'aeroporto (esempio Aeroporto di Firenze Amerigo Vespucci)

- **nation**, di tipo Stringa

Stato in cui l'aeroporto è ubicato (esempio: Aeroporto di Firenze ->Italia)

- **city**, di tipo Stringa

Città servita dall'aeroporto (da specificare che spesso questo non coincide con l'esatta ubicazione della struttura aeroportuale; se di grande dimensione si trovano infatti fuori città: l'aeroporto di Milano Malpensa, che serve Milano, si trova in realtà nei pressi di Busto Arsizio, a 35km dal centro di Milano)



Classico Override dei metodi `hashCode()`, `equals(Object obj)` e `toString()`

### 3.3.3 DimensionClass

Modella la dimensione di un aeroporto, rappresentata da un numero da 1 a 4 che indica la lunghezza maggiore tra tutte le piste di decollo/atterraggio (da 1, la minore possibile, inferiore a 800m, a 4, la maggiore, superiore a 1800m) e una lettera che si riferisce alle dimensioni degli aeromobili che l'aeroporto può ospitare (da A a F, in ordine crescente).

#### Metodo `boolean IsCompatible()`

Questo metodo esegue il confronto tra due oggetti di `dimensionClass` e stabilisce se quella corrente (tipicamente dell'aeromobile) sia compatibile con `other` (tipicamente quella dell'aeroporto). Se sia il numero che la lettera di `this` sono inferiori a quelli di `other` restituisce `true`, altrimenti `false`.

### 3.3.4 Credentials

Modella le credenziali usate dagli utenti per accedere alle varie sezioni delle applicazioni

#### Metodo Costruttore

Inizializza un oggetto di tipo `Credentials` con i valori che gli vengono forniti in input. Gli attributi sono:

- `id`, di tipo Integer  
Un identificatore univoco
- `username`, di tipo String  
Nome utente, con il quale l'utente viene riconosciuto dall'applicazione
- `passwd`, di tipo String  
Password dell'utente

#### Metodo `boolean checkHash(String passwd)`

Questo metodo viene usato per controllare la correttezza della password inserita. Le password, prima di essere memorizzate, vengono processate mediante una funzione di hash, la quale produce una sequenza di bit (detta "digest"), strettamente correlata con i dati in ingresso. Nel nostro caso si è usato l'algoritmo SHA (Secure Hash Algorithm) 512, in cui l'output è di ben 512 bit.

Per prima cosa si è richiamato, sulla stringa fornita in input, il metodo `getInstance` della classe `MessageDigest` con l'algoritmo SHA-512.

Il passo successivo è la conversione della stringa in UTF-8 (Unicode Transformation Format a 8 bit) mediante il metodo `getBytes()`, il quale restituisce un array di byte, poi processato dal metodo `digest()`.

Il costruttore di `BigInteger` assolve poi il compito di convertire quanto calcolato sopra dalla rappresentazione in segno (in questo caso sempre 1, il numero è positivo) e numero alla rappresentazione numerica canonica.

Infine si effettua la conversione da `BigInteger` a una stringa esadecimale. Se la stringa ricevuta in input, processata come descritto sopra, coincide con la password memorizzata allora viene restituito `true`, altrimenti la password non viene riconosciuta e viene restituito `false`.

*Snippet del codice descritto passo passo sopra*

```
public boolean checkHash(String passwd) {
    try {
        MessageDigest messageDigest = MessageDigest.getInstance("SHA-512");
        var hashed = messageDigest.digest(passwd.getBytes(StandardCharsets.UTF_8));

        BigInteger no = new BigInteger(1, hashed);
        String hexHashed = no.toString(16);

        return this.passwd.equals(hexHashed);

    } catch (NoSuchAlgorithmException e) {
    }
    return false;
}
```

**Metodo `toString()` classico**

### 3.3.5 EmployeeRole

Altro non è che un'enumerazione di tutti i ruoli (tra quelli di interesse nel nostro sistema) dei dipendenti dell'azienda. Questi sono "Commander", "FirstOfficer" e "FlightAssistant".

**Override metodo `toString()`**

Ritorna una stringa corrispondente al ruolo

### 3.3.6 Employee

Modella il personale dell'azienda memorizzando, oltre ai dettagli classici, anche quelli inerenti alle mansioni lavorative. Viene implementato anche un attributo per tenerne traccia della posizione in tempo reale.

**Metodo Costruttore**

Inizializza un oggetto di tipo `Employee` con i valori che gli vengono forniti in input. Gli attributi sono:

- **id**, di tipo Integer  
Un identificatore univoco
- **name**, di tipo String  
Nome del dipendente
- **lastName**, di tipo String  
Cognome del dipendente
- **role**, di tipo **EmployeeRole**  
Contiene il ruolo, uno tra quelli specificati sopra, del soggetto
- **abilitation**, di tipo String  
Contiene, nel caso in cui il ruolo sia **Commander** o **FirstOfficer**, il velivolo sul quale sono abilitati a volare. Le regole dell'aviazione commerciale impongono infatti che si sia abilitati soltanto per una tipologia di velivolo alla volta: l'ottenimento del brevetto per un nuovo modello di aereo (o il cosiddetto passaggio macchina che rinnova una precedente abilitazione) va ad annullare la possibilità di prestare servizio sul velivolo sul quale si è prestato fino a quel momento. Questo non vale per steward e assistenti di volo i quali, a patto di vari corsi, possono essere contemporaneamente abilitati per quanti modelli sia necessario.
- **position**, di tipo String  
Inizializzato a vuoto dal costruttore, conterrà poi il codice ICAO dell'aeroporto nel quale il dipendente della compagnia si trova.

**Classico override del metodo **toString()** e setter per l'attributo **position****

**Metodo **String getFullData()****

Realizza e restituisce una lista puntata con "nome dell'attributo" : "valore"

### 3.3.7 FlightRoute

Modella le rotte che sono effettuate dalla compagnia (si parla di tragitti generici, per il volo specifico vedremo un'altra entità specifica).

**Metodo Costruttore**

Inizializza un oggetto di tipo **FlightRoute** con i valori che gli vengono forniti in input. Gli attributi sono:

- **id**, di tipo Integer  
Un identificatore univoco
- **distance**, di tipo Integer  
Espressa in chilometri, indica la distanza in linea d'aria tra l'aeroporto di partenza e quello di destinazione (nel caso sia previsto uno scalo saranno sommate le distanze tra, rispettivamente, l'aeroporto di partenza e quello di scalo e quella tra l'aeroporto di scalo e quello di destinazione). Importante notare che queste distanze sono puramente teoriche: un volo può in realtà essere più lungo per motivi meteorologici, particolari procedure di partenza e/o avvicinamento, chiusura spazi aerei...

- **duration**, di tipo Integer  
Espressa in minuti, indica la durata schedulata del volo tra l'aeroporto di partenza e quello di arrivo; valgono le stesse considerazioni fatte sopra in merito alla distanza.
- **departure**, di tipo Stringa  
Contiene il codice ICAO dell'aeroporto di partenza
- **stepover**, di tipo Stringa  
Contiene, se previsto dal piano di volo, il codice ICAO dell'aeroporto di scalo
- **arrival**, di tipo Stringa  
Contiene il codice ICAO dell'aeroporto di arrivo

**Classico Override dei metodi `hashCode()`, `equals(Object obj)` e `toString()`**

### 3.3.8 Flight

Modella i singoli voli che sono effettuati dalla compagnia (si intendono proprio voli singoli, ciascuno con del personale, un aereo specifico)

#### **Metodo costruttore**

Inizializza un oggetto di tipo **Flight** con i valori che gli vengono forniti in input. Gli attributi sono:

- **id**, di tipo Integer  
Un identificatore univoco
- **departureTime**, di tipo Stringa  
Orario di partenza previsto per il volo
- **passengerNumber**, di tipo Integer  
Numero di passeggeri che saliranno sul volo. La gestione dei passeggeri, per la maggior parte trascurata in questo applicativo, potrebbe essere effettuata da un altro sistema che si occupi della gestione delle prenotazioni e che vada poi a integrarsi con questo (nonostante non sia stato fatto, abbiamo ugualmente fornito tutti gli attributi perché questo possa eventualmente essere implementato in versioni future)
- **route**, di tipo **FlightRoute**  
Tratta che effettuerà questo volo  
\*Per approfondimenti sulla rotta, consultare l'apposita sezione dedicata\*
- **aircraftPlate**, di tipo Stringa  
Contiene l'identificatore univoco dell'aeromobile che è stato assegnato al volo
- **commanders**, **firstOfficers** e **flightAssistants**; vettori di **Employee**  
Contengono rispettivamente il/i comandante/i, il/i primo/i ufficiale/i e gli assistenti di volo che prestano servizio su questo volo.

#### **Metodo `void commitFlight()`**

Questo metodo permette di aggiungere un volo al database.

Innanzitutto si istanzia un oggetto di tipo **FlightDaoPg** e poi si va a richiamare il metodo **commitFlight(this)** per operarne l'aggiunta.

Override metodo `toString()`

## 3.4 System (business logic)

Questa sezione contiene un ulteriore sotto-package: **scheduling**, nella quale è inserito proprio l'algoritmo usato per la creazione dell'operativo voli

### 3.4.1 Scheduling

#### AirportWeighted

Questa classe istanzia oggetti che saranno di ausilio per la realizzazione del grafo degli aeroporti.

##### Metodo costruttore

Inizializza un oggetto di tipo `AirportWeighted` con i valori che gli vengono forniti in input.

Gli attributi di questa classe sono:

- `airport`, di tipo `Airport`
- `weight`, di tipo `int`
  - Questo valore associa all'aeroporto in questione un "peso", pari alla distanza tra l'aeroporto in questione ed un altro. Ad esempio sia l'aeroporto corrente A e la rotta considerata quella tra A e un secondo aeroporto che indichiamo con B, il peso è la distanza tra A e B.
- `routeId`, di tipo `Int`
  - Identifica univocamente la tratta cui la classe si riferisce
- `route`, di tipo `FlightRoute`
  - Una tratta che ha origine nell'aeroporto in questione
  - Questa viene prelevata dal Database per mezzo del `FlightRouteDaoPg` `routedao` chiamandoci il metodo `getRouteById(String.valueOf(this.routeId))`
- `duration`, di tipo `Int`
  - Contiene la durata del volo

Classico Override dei metodi `hashCode()` e `equals(Object obj)`

#### AirportGraph

Questa classe rappresenta il grafo contenente gli aeroporti e le loro interconnessioni. I metodi principali sono quelli per l'aggiunta dei nodi e degli archi.

### Metodo costruttore

Il costruttore crea una HashMap, implementazione più comune della classe Map. La mappa è struttura dati nella quale i dati sono memorizzati come coppie chiave-valore; nel nostro caso è chiamata `adjacencyList` ed ha come chiave un oggetto di tipo `Airport` e come valore una lista collegata di `AirportWeighted`

### Metodo `void addVertex(Airport airport)`

Dopo aver effettuato grazie al metodo `containsKey(airport)` la verifica che l'aeroporto non sia già contenuto nella mappa, utilizza il metodo `put(...)` per inserirlo.

### Metodo `void addEdge(Airport a1,Airport a2,int weight,int routeId,int routeDuration)`

La prima operazione di questo metodo è il controllo che ambedue gli aeroporti inseriti tra i parametri siano già presenti in `adjacencyList`; in caso contrario viene richiamato il metodo sovrastante `addVertex(airport)` che si occupa di eseguire l'aggiunta.

L'operazione successiva è l'addizione dell'arco tra `a1` e `a2`. L'ordine dei due parametri è molto importante in quanto, trattandosi di un grafo diretto, si crea esclusivamente l'arco `a1 -> a2` e non il viceversa. In sostanza, alla lista collegata di

`AirportWeighted` dell'aeroporto `a1` si va ad aggiungere un nuovo oggetto che rappresenti la connessione con l'aeroporto `a2` e che abbia come altri parametri quelli forniti in input al metodo.

```
public void addEdge(Airport a1,Airport a2,int weight,int routeId,int routeDuration){  
  
    if(!adjacencyList.containsKey(a1)){  
        addVertex(a1);  
    }  
  
    if(!adjacencyList.containsKey(a2)){  
        addVertex(a2);  
    }  
  
    adjacencyList.get(a1).add(new AirportWeighted(a2, weight,routeId,routeDuration));  
}
```

### Metodo `Set<Airport> getVertexList()`

Restituisce semplicemente, usando il metodo `adjacencyList.keySet()`, la lista di tutti gli aeroporti contenuti nel grafo.

### Metodo boolean `containsVertex(Airport r)`

Ritorna vero se l'aeroporto `r` è contenuto in `adjacencyList`, falso altrimenti

**Metodo `LinkedList<AirportWeighted> getAdjacentVertex(Airport airport)`**

Restituisce la lista collegata di `AirportWeighted` associata all'aeroporto.

## SimpleSchedule

Classe che si occupa effettivamente dello scheduling dei voli, assegnando aeromobili e personale alle rotte garantendo il rispetto di tutti i vincoli. Viene implementato il metodo `Vector<Flight> run()` dichiarato nell'interfaccia `SchedulingStrategy`; è questo il metodo che effettua concretamente la mission dell'applicazione.

### Metodo costruttore

In questo costruttore si vanno ad usare i DAO per recuperare i dati dal database, mettendoli in liste/strutture "provvisorie" e andando poi ad usarle nell'esecuzione dello scheduling vero e proprio. Questo non causa comunque problemi di disallineamento dei dati, lo scheduling viene infatti effettuato al momento del caricamento della CLI, prima che l'utente abbia modo di modificare dati nel DB.

### Metodo `AirportGraph buildGraphFromFlightRoute()`

Questo metodo si occupa concretamente della creazione del grafo degli aeroporti. Viene infatti dichiarato un oggetto graph di tipo `AirportGraph`.

Il passo successivo è la creazione di due vettori: il primo, `routes`, contenente tutte le tratte presenti nel database e ricavate mediante l'uso di un `flightRouteDao`, il secondo, `airports`, contenente gli aeroporti, ricavati mediante un `airportDao`

Si inseriscono poi tutti gli aeroporti nella mappa `airportDict`, avente come chiave l'icao dell'aeroporto e come valore un oggetto di tipo `airport`; questo è importante per effettuare quella che in base di dati si direbbe una "join" tra i dati contenuti in `routes` e quelli contenuti in `airports`. Infine, si scorrono tutte le rotte andando ad aggiungere per ognuna di esse un arco del grafo e passandole come parametri i due aeroporti (partenza e arrivo), l'id della rotta, la distanza e la durata. Si restituisce in output il grado appena creato.

### Metodo `boolean canFly(Aircraft aircraft, AirportWeighted airportWeighted)`

Il compito di questo metodo è la verifica che un aereo `aircraft` sia in grado di volare verso un aeroporto `airportWeighted`: si deve controllare sia la compatibilità dimensionale di aeromobile e aeroporto che la distanza tra due, garantendo che il range del velivolo sia ad essa inferiore.

Per evitare il rischio di un'emergenza carburante a bordo dovuta al margine troppo ridotto tra i due valori soprastanti (ad esempio aereo con range di 6500 km si trova a percorrere una rotta di 6495 km; è sufficiente la più piccola deviazione per i più banali

motivi per causare un'emergenza carburante e di conseguenza un atterraggio di emergenza nell'aeroporto più vicino) è stata introdotta una tolleranza, `RangeTolerance`, pari al 15% della distanza della rotta.

Viene quindi effettuata la verifica aggiuntiva:

```
(aircraft.range-airportWeighted.weight) <
airportWeighted.weight*RangeTolerance)
```

```
private boolean canFly(Aircraft aircraft,AirportWeighted airportWeighted){
    if (!aircraft.canGo(airportWeighted.airport)){
        return false;
    }

    if(aircraft.range < airportWeighted.weight || (aircraft.range-airportWeighted.weight) < airportWeighted.weight*RangeTolerance){
        return false;
    }

    return true;
}
```

## Gestione del personale

### Metodo `Vector<Employee> getFlightAssistants(int assistantNumber,Airport source,Airport destination)`

Questo metodo esegue l'assegnazione degli assistenti di volo ad uno specifico volo. Al suo interno, oltre a quanto dichiarato e già descritto precedentemente, viene usata la mappa `flightAssistantsLocation` che ha come chiave un `airport` e come valore una lista di impiegati; in realtà, per quanto sottolineato sopra, la lista è di soli assistenti di volo.

Si dichiara un vettore di impiegati, `currentFlightAssistants`, che sarà la struttura nel quale andremo mano a mano ad aggiungere le hostess/steward associati a quel volo.

Viene innanzitutto fatto partire un ciclo for, con un numero di iterazioni pari ad `assistantNumber`, numero persone da assegnare; ad ogni ciclo verrà infatti compiuto un assegnamento.

I primi assistenti di volo ad essere assegnati sono quelli già disponibili all'aeroporto di partenza (le condizioni imposte sulla mappa perchè questa condizione sia soddisfatta sono che la chiave esista e che la lista collegata ad essa associata non sia vuota). Si procede quindi con la rimozione dalla mappa con il metodo `remove()` e l'aggiunta a `currentFlightAssistants` con il metodo `add(flightAssistant)`.

Se il personale disponibile in aeroporto non dovesse essere sufficiente, verrà prelevato dalla lista generale `flightAssistants`, effettuando rimozione e aggiunta usando i metodi sopra descritti. Volendolo interpretare da un punto di vista pratico, verrà chiesto (con l'opportuno preavviso) agli assistenti di volo di presentarsi in un aeroporto per effettuare il servizio.

Infine verrà aggiunta, se mancante, la destinazione del volo (lista collegata compresa) alla mappa `flightAssistantsLocation` e proprio in questa lista vengono aggiunti tutti gli assistenti di volo in `currentFlightAssistants`. In



quest'ultimo passaggio è come se fossimo andati a simulare l'effettuarsi del volo e il conseguente spostamento del personale di cabina nell'aeroporto di destinazione. Si ritorna infine il vettore `currentFlightAssistants`.

```
private Vector<Employee> getFlightAssistants(int assistantNumber, Airport source, Airport destination){
    Vector<Employee> currentFlightAssistants= new Vector<>();

    for (int i = 0; i < assistantNumber; i++) {
        Employee flightAssistant=null;

        if(flightAssistantsLocation.containsKey(source) && !flightAssistantsLocation.get(source).isEmpty()){
            flightAssistant=flightAssistantsLocation.get(source).remove();
            currentFlightAssistants.add(flightAssistant);
        }else{
            try {
                flightAssistant=flightAssistants.remove();
                currentFlightAssistants.add(flightAssistant);
            } catch (Exception e) {
            }
        }

        if(flightAssistant != null){
            flightAssistantsLocation.putIfAbsent(destination,new LinkedList<>());
            flightAssistantsLocation.get(destination).add(flightAssistant);
        }
    }
    return currentFlightAssistants;
}
```

### Metodo Boolean `isAbilitationValid(String abilitation,String aircraftModel)`

Effettua il controllo che l'abilitazione dei piloti sia coerente con il modello dell'aereo. Prende come parametri due stringhe, `abilitation` e `aircraftModel`. Nel primo costrutto condizionale si effettua con il metodo `equals()` che questi due valori siano uguali.

In caso affermativo restituisce true. Il secondo if è stato inserito per gestire il caso della famiglia Airbus 318/319/320/321, che richiedono tutti la stessa abilitazione; per un pilota abilitato all'A318, ad esempio, questo metodo deve ritornare true anche per tutti gli altri modelli sopra elencati. Se non viene trovata alcuna corrispondenza tra i due valori, viene restituito invece false.

### Metodo `Vector<Employee> getCommanders(int neededCommanders, Airport source, Airport destination,String aircraftModel)`

Il metodo è molto simile a quanto visto per gli assistenti di volo, con l'unica differenza che sta nella verifica dell'aeromobile per cui ha l'abilitazione del comandante prima di assegnarlo al volo.

### Metodo `Vector<Employee> getFirstOfficers(int neededFirstOfficers, Airport source, Airport destination,String aircraftModel)`

*Metodo concettualmente identico a quello sopra, cambiano soltanto i nomi degli oggetti usati; per delucidazioni confrontare la relativa documentazione*

**Metodo `boolean isFlightValid(Flight f, Aircraft a, int neededCommanders)`**

Metodo il cui scopo è la verifica che ci siano in numero di comandanti/primi ufficiali richiesti dal volo e il numero di assistenti di volo richiesti invece dall'aeromobile; restituisce `true` se queste condizioni sono soddisfatte, `false` altrimenti.

**Metodo `boolean isLongFlight(AirportWeighted airportWeighted)`**

Restituisce `true` se il volo è "lungo" ovvero, secondo quanto previsto in fase di progettazione, superiore a nove ore, `false` altrimenti.

## Il cuore dell'applicazione: l'algoritmo di scheduling

**Metodo `Vector<Flight> run()`**

Implementa il metodo omonimo presente nella classe `SchedulingStrategy`.

E' il metodo centrale della nostra applicazione: si occupa di schedulare i voli, assegnargli il personale e i velivoli rispettando tutti i vincoli.

Il tutto inizia costruendo il grafo per mezzo del metodo

`buildGraphFromFlightRoute()`, già spiegato nei minimi dettagli in un paragrafo precedente, e ordinando le liste di adiacenza in modo tale che gli aeroporti contenuti siano ordinati per distanza crescente. Sulla lista di adiacenza di ciascun aeroporto (uno per uno per mezzo di un ciclo for su tutti i nodi del grafo) viene richiamato `sort()`. Il comparatore che gli viene passato calcola semplicemente la distanza tra due aeroporti.

Sempre nel ciclo for, tramite il `parkingDao`, estrae tutti gli aerei parcheggiati nell'aeroporto e tramite un altro ciclo va ad inserirli uno ad uno nella mappa `aircraftTime`, una `HashMap` avente come chiave un aereo e come valore `StartTime`. Quest'ultimo è un oggetto di tipo `LocalTime`, una classe Java che rappresenta il tempo usando il formato ore-minuti-secondi, e contiene il valore del tempo corrente ottenuto, per mezzo del metodo `now()`, dall'orologio di sistema e nel fuso orario di default; è troncato alle sole ore (metodo `truncatedTo(ChronoUnit.HOURS)`).

L'ultima parte di questo primo ciclo riguarda il popolamento della mappa `aircraftLocation`, che ha come chiave un aeroporto e come valore un `Vector<Aircraft>`; rappresenta quindi per ciascun aeroporto l'insieme degli aerei che vi sono in quel momento in sosta.

Per un numero di iterazioni stabilito a priori (queste rappresentano il numero di "spostamenti" di velivoli che vogliamo andare a simulare), viene poi richiamato `exploreBfs()`, che esamineremo nel dettaglio subito dopo.

Questo metodo ritorna il vettore di voli richiesto.

```

public Vector<Flight> run() {
    graph=buildGraphFromFlightRoute();

    for (Airport airport : graph.getVertexList()) {
        graph.getAdjacentVertex(airport).sort(new Comparator<AirportWeighted>() {
            @Override
            public int compare(AirportWeighted airport0, AirportWeighted airport1) {
                return airport1.weight - airport0.weight;
            }
        });

        Vector<Aircraft> parkedAircrafts=parkingDao.getParked(airport.icao);
        for (Aircraft a : parkedAircrafts) {
            aircraftTime.put(a,StartTime);
        }

        aircraftLocation.put(airport, parkedAircrafts);
    }

    for (int iteration = 0; iteration < iterations; iteration++) {
        exploreBfs();
    }

    return this.flights;
}

```

### Metodo **void exploreBfs()**

Questo metodo scorre tutti gli aeroporti presenti come vertici nel grafo e per ciascuno di essi chiama il metodo **bfs(graph, airport)**, passandogli appunto come parametri il grafo che abbiamo costruito e l'aeroporto in esame.

### Metodo **void bfs(AirportGraph G,Airport source)**

Si realizza un attraversamento di tipo BFS sul grafo degli aeroporti. Questo algoritmo, spesso chiamato anche di ricerca in ampiezza agisce per livelli, scoprendo prima tutti i vertici che si trovano a distanza  $k$  dall'origine **airport**, poi tutti quelli che si trovano a distanza  $k+1$  e così via...

Questo procedimento si realizza per mezzo dell'inserimento di tutti i nodi che non sono ancora stati esaminati in una coda con politica First In First Out, collocandoli in un'altra struttura dati una volta che lo saranno.

Questo metodo prende in input due parametri: **G**, il grafo degli aeroporti e **source**, il nodo radice dal quale viene dato inizio all'attraversamento BFS.

Per prima cosa vado a popolare **visited**: una Mappa avente come chiave un aeroporto e come valore un booleano; l'utilità di questa è indicare se un aeroporto è stato visitato o man mano che si procede con l'esecuzione dell'attraversamento. Questa viene inizializzata per mezzo di un ciclo for che inserisce tutti gli aeroporti ponendo a **false** il valore (all'inizio infatti nessuno degli aeroporti è stato visitato). Il passo successivo è la creazione di una coda FIFO di aeroporti, nella quale si va ad

aggiungere `source`; contestualmente si effettua anche la modifica anche, per mezzo del metodo `replace(aiport, newValue)`, del valore in `visited` a `true`.

Inizia poi il ciclo principale, che agisce finché `fifo` non è vuota.

Ad ogni iterazione si inizia rimuovendo un aeroporto, modificando il valore in `visited` a `true`. Si procede poi con la costruzione della lista con priorità `parkedAircraftsPq`, una lista di `aircraft` nella quale gli aerei vengono ordinati sulla base del range, da quello con l'autonomia minore a quello con l'autonomia maggiore; viene riempita estraendo gli aerei da `aircraftLocation` limitandosi però esclusivamente a quelli parcheggiati nell'aeroporto in esame dal ciclo in quel momento.

Viene fatto partire un ciclo interno, il quale procederà fino a quando ci saranno elementi in `parkedAircraftsPq`. Entriamo in un altro ciclo annidato, che scorrerà su tutti gli aeroporti connessi con quello considerato (che corrispondono a quelli adiacenti nel grafo).

In questo ciclo, oltre al controllo iniziale se ci siano ancora oggetti `parkedAircraftsPq` (potrebbe capitare che non ci siano, visto che il ciclo ne prevede una rimozione), viene estratto `nextAircraft`, aereo candidato ad effettuare la rotta; se sono soddisfatti tutti i vincoli (ovvero l'aeroporto di destinazione non è ancora stato visitato e l'aereo scelto è in grado di effettuare il volo, metodo `canFly()`), si può proseguire con la modifica della posizione del velivolo dall'aeroporto corrente a quello di destinazione, tutto questo andando ad agire sulla Hash `aircraftLocation`. Nel caso in cui un aeroporto non sia presente in tale struttura, si rimedia operando l'aggiunta. *Il codice di tutto quest'ultimo paragrafo è visibile nell'immagine sottostante*

```
// Inner loop
while (!parkedAircraftsPq.isEmpty()) {
    for (var airportWeighted : graph.getAdjacentVertex(airport)) {
        if (!parkedAircraftsPq.isEmpty()) {
            var nextAircraft = parkedAircraftsPq.remove();
            if (visited.get(airportWeighted.airport) != true && canFly(nextAircraft, airportWeighted)) {
                aircraftLocation.get(airport).remove(nextAircraft);

                if (aircraftLocation.containsKey(airportWeighted.airport)) {
                    aircraftLocation.get(airportWeighted.airport).add(nextAircraft);
                } else {
                    Vector<Aircraft> aircrafts = new Vector<>();
                    aircrafts.add(nextAircraft);
                    aircraftLocation.put(airportWeighted.airport, aircrafts);
                }
            }
        }
    }
    // ...continues (time schedule, staff, various checks...)
}
```

Una volta scelto il velivolo, si procede a:

- calcolare il tempo di decollo, (dato dal "tempo di simulazione" corrente sommato al tempo di turnaround (Il turnaround, dichiarato come intero, è il tempo di sosta a terra di un aeromobile necessario per le operazioni di sbarco passeggeri, imbarco passeggeri, rifornimento carburante, pulizie, imbarco del catering, merce e posta))

- modificare il suo "tempo di simulazione", rendendolo pari all'orario in cui questo sarà disponibile nel nuovo aeroporto.
- assegnare le assistenti di volo, richiamando la funzione `getFlightAssistants(nextAircraft.assistantsNumber, airport, airportWeighted.airport)`
- Definire il numero di comandanti e primi ufficiali, pari a 2 nel caso in cui il metodo `isLongFlight(airportWeighted)` restituisca `true`, 1 altrimenti
- assegnare comandanti e primi ufficiali, rispettivamente attraverso `getCommanders(neededCommanders, airport, airportWeighted.airport, nextAircraft.model)` e `getFirstOfficers(neededCommanders, airport, airportWeighted.airport, nextAircraft.model)`;

Fatto tutto questo, si costruisce finalmente il volo fornendo tutti i parametri; si istanzia quindi un nuovo oggetto `flight` di tipo `Flight`. Vedi immagine sottostante



```
//bfs execution...

var flight = new Flight(
    0,
    departure.format(dateTimeFormat),
    0,
    airportWeighted.route,
    nextAircraft.plate,
    currentCommanders,
    currentFirstOfficers,
    currentFlightAssistants
);

//calculated flight commit & bfs restart...
```

Prima di registrarlo ufficialmente, si fa infine un ultimo controllo sul numero di componenti dell'equipaggio usando `isFlightValid(flight, nextAircraft, neededCommanders)`.

Se tutto va a buon fine, si aggiunge il volo al vettore `flights` di tutti i voli e lo si registra nel database.

Come da procedura BFS, si aggiunge poi l'aeroporto di destinazione alla coda `fifo`.

### 3.4.2 SimulatedClock

Questa classe implementa un orologio simulato, il quale consente di eseguire la simulazione dei voli eseguiti e degli asset della compagnia in ogni momento.

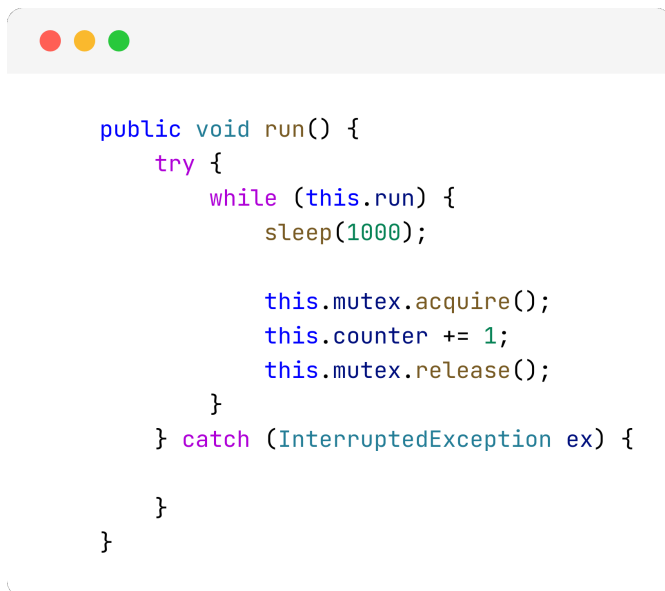
`SimulatedClock` estende la classe `Thread`. Include un counter che scandisce lo scorrere dei secondi: per mantenere sincronizzato il valore ed evitare disallineamenti si ricorre al semaforo `mutex`.

Il metodo principale è il metodo `run()`; secondario è il metodo `getTime()`

### Metodo `void run()`

Avvia "lo scorrere del tempo"; l'attributo `counter` tiene traccia del numero di secondi trascorsi. Ogni 1000 millisecondi (1 secondo) questo valore viene acquisito attraverso il metodo `mutex.acquire()`, incrementato di uno e rilasciato.

L'utilizzo del metodo `sleep()` impone la gestione dell'eccezione `InterruptedException`, che è comunque lasciata di default.



```
public void run() {
    try {
        while (this.run) {
            sleep(1000);

            this.mutex.acquire();
            this.counter += 1;
            this.mutex.release();
        }
    } catch (InterruptedException ex) {
    }
}
```

### Metodo `String getTime()`

Restituisce una stringa composta "ore trascorse : minuti trascorsi".

Usando sempre nelle stesse modalità descritte precedentemente per il semaforo,, questo metodo assegna il valore del contatore all'intero lungo `current`.

A partire da questo valore vengono poi calcolate:

- le ore, String `hour`

Tale valore è dato dal quoziente intero della divisione tra `current` e `timeCoefficient`, una costante intera di valore 60. La conversione a stringa viene poi eseguita con il metodo `valueOf(string)`

- i minuti, String `minutes`

Anche in questo caso si esegue la divisione come sopra, ma il valore di minutes è dato dal resto

*Quella riportata è solo una sezione del metodo descritto sopra*

```
String hour = String.valueOf((current - (current % this.timeCoefficient)) / this.timeCoefficient);
String minutes = String.valueOf(current % this.timeCoefficient);

return hour + ":" + minutes;
```

### 3.4.3 FlightSchedule

Classe che implementa il design pattern Strategy.

Questo design pattern di tipo comportamentale (facente parte dei design pattern della Gang of Four GoF, un gruppo di quattro soggetti che misero assieme 23 design pattern ponendo a tutti gli effetti la base dell'ingegneria del software) consente di definire una famiglia di algoritmi rendendoli intercambiabili; in base alle necessità e al contesto specifico si seleziona a runtime un algoritmo piuttosto che un altro.

In questo caso questa classe è stata creata per far sì che l'applicazione sia in grado di gestire ed eseguire una qualsiasi **SchedulingStrategy** (al momento l'utilità effettiva, data l'esistenza di una sola strategia, è puramente concettuale e dovuta all'Open-Closed Principle, ma nuove strategie potrebbero essere inserite in futuro... )

#### Metodo Costruttore

Prende in input un oggetto di tipo **SchedulingStrategy** che assegnare all'attributo **strategy** e che rappresenta la strategia di scheduling da applicare.

#### Metodo **Vector<Flight> makeSchedule()**

Se l'attributo **strategy** è diverso da **null**, ovvero è stata assegnata una strategia, ne va ad eseguire il metodo **run()** andando così ad effettuare lo scheduling. Nel caso in cui **strategy==null** invece esce subito.

Restituisce in output tutti i voli che sono stati generati dall'algoritmo.

### 3.4.4 ManagementSystem

È la classe che istanzia e gestisce i principali elementi necessari per il funzionamento dell'applicazione.

#### Metodo costruttore

Oltre prendere in input e assegnare un **FlightSchedule**, il costruttore istanzia e avvia anche l'orologio simulato **clock**. Assolve a questo compito il metodo


`start()`, che fa partire il Thread sul quale viene richiamato cambiandone lo stato in *Running*.

#### Metodo `void runScheduling()`

Lancia lo scheduling richiamando il metodo `makeSchedule()` sull'oggetto `flightSchedule`.

#### Metodo `void quit()`

Ferma lo scorrere dell'orologio bloccando il `clock`. L'interruzione vera e propria avviene mediante `interrupt()` ed è però poi seguita dal metodo `join()` per l'attesa che questo abbia finito quanto doveva prima di "morire". Come previsto da quest'ultimo metodo si gestisce l'eccezione `InterruptedException`.



```
public ManagementSystem(FlightSchedule fSchedule) {
    flightSchedule = fSchedule;
    clock.start();
}

public void runScheduling(){
    this.scheduledFlights = flightSchedule.makeSchedule();
}

public void quit() {
    this.clock.run = false;
    this.clock.interrupt();

    try {
        this.clock.join();

        //Interrupt Handling..
    }
```

## 3.5 Interfaccia (CLI)

La classe `CLI` (Command Line Interface) è responsabile dell'interazione con l'utente, fornendo le possibilità di navigazione all'interno del sistema, per poter consultare le informazioni relative a voli e personale.

La navigazione attraverso i moltissimi sottomenù (*Tutti descritti nei Page Navigation Diagrams*) avviene tramite l'inserimento da parte dell'utente in linea di comando del numero corrispondente alla sezione alla quale vuole accedere/al comando che vuole eseguire