

# Sommario

<b>1. Introduzione .....</b>	<b>3</b>
<b>1.1. Descrizione del problema.....</b>	<b>3</b>
<b>2. Progettazione Concettuale .....</b>	<b>4</b>
<b>2.1 - Spiegazione delle classi e rispettivi attributi.....</b>	<b>4</b>
o - Utente.....	4
o - Amministratore .....	4
o - Utente_Generico .....	5
o - Volo .....	5
o – Stato_Volo (Enumeration) .....	5
o - Prenotazione .....	5
o - Stato_Prenotazione (Enumeration).....	5
o - Passeggero .....	6
o - Bagaglio .....	6
o - Stato_Bagaglio (Enumeration) .....	6
<b>3. – Codice Java .....</b>	<b>6</b>
<b>3.1 – Package Model .....</b>	<b>6</b>
o - Classe Utente .....	7
o - Classe Amministratore.....	7
o - Classe UtenteGenerico .....	7
o - Classe Passeggero .....	8
o - Classe Prenotazione.....	8
o - Classe Bagaglio .....	8
o - Classe Volo.....	9
o - Enumerazione StatoBagaglio.....	10
o - Enumerazione StatoPrenotazione .....	10
o - Enumerazione StatoVolo .....	10
<b>3.2 – Package GUI.....</b>	<b>11</b>
o - Classe BasicBackgroundPanel.....	11
o - Classe ConfirmDialog.....	11
o - Classe CustomFlightCellRenderer .....	11
o – Classe CustomLuggageCellRenderer .....	11
o – Classe CustomReservationCellRenderer .....	11

# Elaborato di gruppo per il corso di Object Orientation (Traccia 3)

o – Classe ErrorPanel .....	11
o – Classe FormHelper .....	12
o – Classe ImageLoader .....	12
o – Classe TableSetter .....	12
o – Classe UtilFunctionsForGUI.....	12
o – Classe BagagliAdminGUI .....	13
o – Classe BagagliGUI.....	13
o – Classe CercaVoloAdminGUI .....	13
o – Classe CercaVoloGUI.....	13
o – Classe CheckInGUI .....	13
o – Classe Home.....	13
o – Classe HomePage .....	14
o – Classe HomePageAdmin.....	14
o – Classe InserisciVoloGUI .....	14
o – Classe LoginGUI.....	14
o – Classe PrenotazioniGUI .....	14
o – Classe RegisterGUI .....	14
o – Classe VoliAdminGUI .....	14
o – Classe VoliGUI .....	14
<b>3.3 – Package Custom_Exceptions .....</b>	<b>15</b>
o – Classe EmailAlreadyExistsException.....	15
o - Classe ModifyTableException .....	15
o - Classe UserAlreadyExistsException.....	15
o - Classe UserNotFoundException .....	15
<b>3.4 – Package Controller .....</b>	<b>15</b>
o - Classe Controller .....	15
<b>3.5 – Package DAO .....</b>	<b>18</b>
<b>3.6 – Package Implementazioni_postgres_dao .....</b>	<b>18</b>
o - Classe AmministratoreImplementazionePostgresDAO .....	18
o - Classe UtenteGenericoImplementazionePostgresDAO.....	19
o - Classe UtenteImplementazionePostgresDAO .....	20
o - Classe UserUtilFunctionsForDAO .....	21
<b>3.7 – Package Database .....</b>	<b>21</b>
o - Classe ConnessioneDatabase .....	21
<b>4. Sequence Diagram.....</b>	<b>21</b>

<b>4.1 – Primo Esempio .....</b>	<b>21</b>
<b>4.2 – Secondo Esempio .....</b>	<b>23</b>

# 1. Introduzione

**Il seguente elaborato ha lo scopo di documentare la progettazione e lo sviluppo di un programma, ad opera degli studenti Pocomento Antonio, Ruggiero Giulio e Simone Rampone Eugenio. Il programma nasce come progetto a scopi valutativi per il corso di Object Orientation, ed implementa un sistema di gestione di un aeroporto.**

## 1.1. Descrizione del problema

**Verranno riportate progettazione e sviluppo di un programma che implementa un sistema che permetta l'organizzazione e la gestione dell'aeroporto di Napoli.**

**Il sistema può essere utilizzato da utenti autenticati tramite una login (nome utente o email) e una password. Gli utenti sono suddivisi in due ruoli: utenti generici, che possono prenotare voli, e amministratori del sistema.**

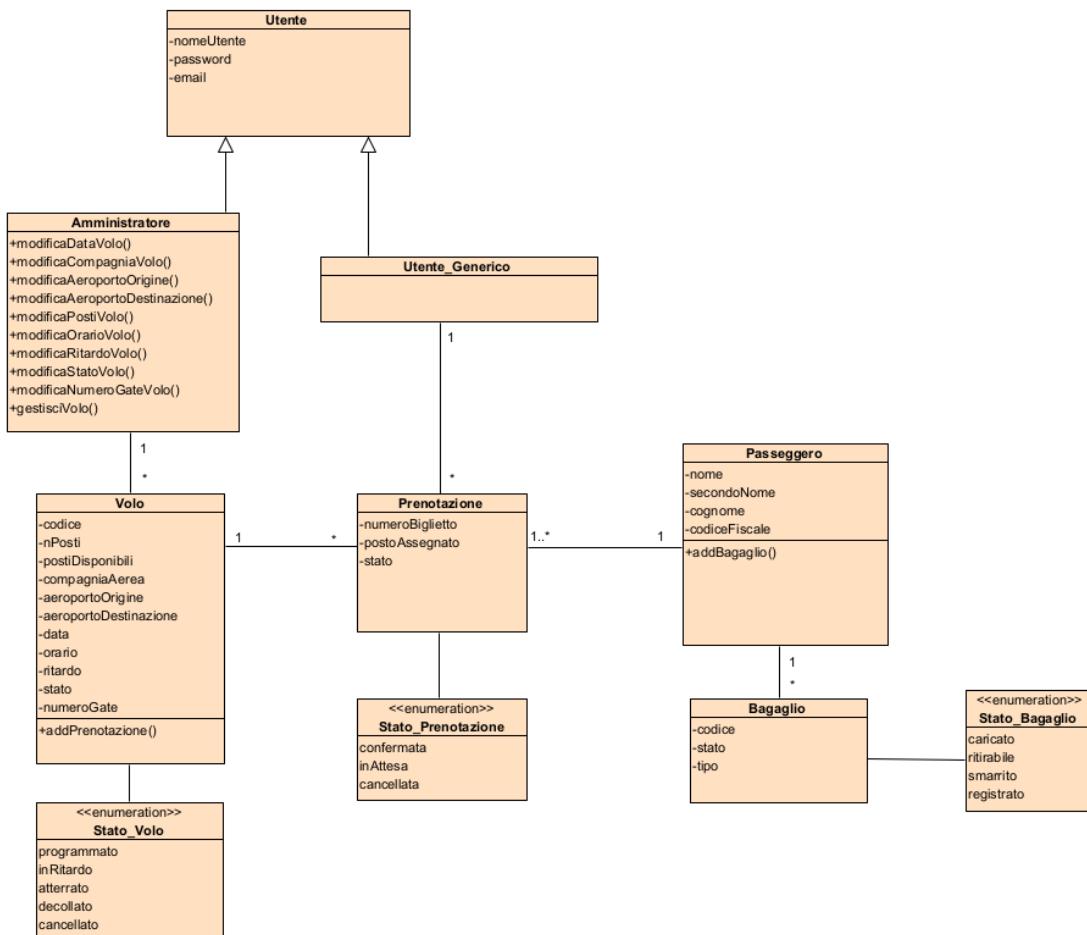
**Il sistema gestisce i voli in arrivo e quelli in partenza. Ogni volo è caratterizzato da un codice univoco, la compagnia aerea, l'aeroporto di origine (per i voli in arrivo a Napoli) e quello di destinazione (per i voli in partenza da Napoli), la data del volo, l'orario previsto, l'eventuale ritardo e lo stato del volo (programmato, decollato, in ritardo, atterrato, cancellato).**

**Gli utenti generici possono effettuare prenotazioni per i voli programmati. Ogni prenotazione è legata a un volo e contiene informazioni come i dati del passeggero (che non deve necessariamente coincidere con il nome dell'utente che lo ha prenotato), il numero del biglietto, il posto assegnato e lo stato della prenotazione (confermata, in attesa, cancellata).**

**Il sistema gestisce anche i gate di imbarco (identificati da un numero), assegnandoli ai voli in partenza. Un'altra funzione importante è il monitoraggio dei bagagli. Ogni bagaglio viene registrato nel sistema durante l'operazione di check-in, generando un codice univoco che consente il**

**tracciamento. Durante il percorso, lo stato del bagaglio viene gestito manualmente dagli amministratori.**

## 2. Progettazione Concettuale



### 2.1 - Spiegazione delle classi e rispettivi attributi

#### o - Utente

È una classe generale di utenti che si interfacciano con il sistema, tutti gli utenti hanno tre attributi: il **nomeUtente** (fittizio o non), **password** ed **email**.

#### o - Amministratore

È la classe che rappresenta gli amministratori del nostro sistema. Ognuno ha dei voli da gestire e modificare.

I metodi della classe si occupano di modificare le informazioni relative ai voli gestiti dall'amministratore (modificaDataVolo, modificaPostiVolo...) e di aggiungere un determinato volo alla lista di quelli gestiti dall'admin (metodo gestisciVolo).

## **o - Utente\_Generico**

È la classe che rappresenta un generico utente del sistema.

## **o - Volo**

È la classe che rappresenta i voli del sistema. Ogni volo possiede un codice (identificativo del volo), un numero di posti (nPosti), il numero di posti disponibili (postiDisponibili), la compagnia Aerea che gestisce il volo (compagniaAerea), l'aeroporto da cui parte il volo (aeroportoOrigine), l'aeroporto di arrivo (aeroportoDestinazione), la data di partenza (data), l'orario di partenza (orario), l'eventuale ritardo nella partenza (ritardo), il numero del gate (numeroGate) e lo stato in cui si trova il volo.

## **o – Stato Volo (Enumeration)**

È una classe enum che rappresenta lo stato in cui può trovarsi il volo:

- *Programmato*

- *inRitardo*

- *Atterrato*

- *Decollato*

- *Cancellato*

## **o - Prenotazione**

È la classe che rappresenta le prenotazioni del sistema. Ogni prenotazione ha un numero di biglietto (numeroBiglietto), il posto assegnato a chi ha prenotato (postoAssegnato) e lo stato della prenotazione.

## **o - Stato\_Prenotazione (Enumeration)**

È una classe enum che rappresenta lo stato della prenotazione:

- **Confermata**

- **inAttesa**

- **Cancellata**

## **o - Passeggero**

È la classe che rappresenta i passeggeri del sistema. Ogni passeggero ha un nome (nome), un secondo nome (secondoNome), un cognome (cognome) e un codice fiscale (codiceFiscale).

La classe passeggero possiede anche un metodo addBagaglio che permette di aggiungere un bagaglio alla lista dei bagagli posseduti dal passeggero.

## **o - Bagaglio**

È la classe che rappresenta i bagagli del sistema. Ogni bagaglio ha un codice identificativo (codice), un tipo di bagaglio (tipo) e lo stato in cui si trova.

## **o - Stato\_Bagaglio (Enumeration)**

È una classe enum che rappresenta lo stato in cui può trovarsi il bagaglio:

- **Caricato**

- **Ritirabile**

- **Smarrito**

- **Registrato**

## **3. – Codice Java**

Di seguito viene riportata una spiegazione nel dettaglio del codice del programma in oggetto. Per rendere la consultazione facile e veloce verranno divisi in paragrafi e sottoparagrafi i package e le classi da descrivere.

### **3.1 – Package Model**

Il Package Model contiene tutte le classi descritte nel diagramma concettuale UML visto in precedenza, opportunamente tradotte in linguaggio Java.

## **o - Classe Utente**

La classe Utente è molto semplice, possiede unicamente dei metodi getter che permettono di ricavare i rispettivi attributi:

--getNomeUtente()

--getEmail()

--getPasswordUtente()

## **o - Classe Amministratore**

La classe Amministratore estende la classe Utente, quindi gli attributi e i metodi (public e protected) della superclasse vengono ereditati.

I metodi univoci dell' admin si occupano, invece, di:

### **1. Ottenere la lista dei voli gestiti dall'amministratore corrispondente**

-- getVoliGestiti()

### **2. Modificare le informazioni dei voli gestiti dall'amministratore**

-- modificaDataVolo(volo, data)

-- modificaCompagniaVolo(volo, compagnia)

-- modificaAeroportoOrigine(volo, aeroporto)

-- modificaAeroportoDestinazione(volo, aeroporto)

--modificaPostiVolo(volo, posti)

--modificaOrarioVolo(volo, orario)

--modificaRitardoVolo(volo, ritardo)

--modificaStatoVolo(volo, stato)

--modificaNumeroGateVolo(volo, numeroGate)

### **3. Aggiungere un volo alla lista dei voli gestiti dall'amministratore**

--gestisciVolo(volo)

## **o - Classe UtenteGenerico**

La classe UtenteGenerico estende la classe, quindi gli attributi e i metodi (public e protected) della superclasse vengono ereditati.

**Questa classe ha solo un metodo getter che ricava la lista di prenotazioni effettuate da un utente:**

-- getPrenotazioniUtente()

## **o - Classe Passeggero**

**La classe Passeggero possiede dei metodi getter che permettono di ricavare i rispettivi attributi:**

--getNome()

--getSecondoNome()

--getCognome()

--getCodiceFiscale()

--getBagagli()

**Vi è, inoltre, un metodo che aggiunge un bagaglio alla lista dei bagagli di un passeggero**

--addBagaglio(bagaglio)

## **o - Classe Prenotazione**

**La classe Prenotazione possiede semplicemente dei metodi getter e setter che ricavano e sovrascrivono i valori degli attributi della classe**

-- getNumeroBiglietto()

--getVolo()

--getPasseggero()

--getStato()

--getUtente()

--setStato(stato)

## **o - Classe Bagaglio**

**Come la classe Prenotazione, anche la classe Bagaglio possiede solo metodi getter e setter che ricavano e modificano i valori degli attributi della classe.**

-- getStato()

--getCodice()

--getCodiceVisualizzato()

--getTipo()

--getPasseggero()

--setStato(stato)

## o - Classe Volo

La classe Volo possiede metodi:

### 1. Getter

--getCodice()

--getNPosti()

--getPostiDisponibili()

--getCompagniaAerea()

--getAeroportoOrigine()

--getAeroportoDestinazione()

--getData()

--getOrario()

--getRitardo()

--getStato()

--getNumeroGate()

--getPrenotazioni()

### 2. Setter

--setCompagniaAerea(compagniaAerea)

--setAeroportoOrigine(aeroportoOrigine)

-- setAeroportoDestinazione(aeroportoDestinazione)

-- setData(data)

-- setOrario(orario)

--setRitardo(ritardo)

--setStato(stato)

--setNumeroGate(numeroGate)

--setPostiDisponibili(posti)

**3. Un metodo che aggiunge una prenotazione alla lista dei voli prenotati e diminuisce di uno il numero di posti disponibili per quel volo**

--addPrenotazione(prenotazione)

## **o - Enumerazione StatoBagaglio**

Describe lo stato in cui può trovarsi un bagaglio.

Quattro valori possibili:

**-REGISTRATO**

**-CARICATO**

**-RITIRABILE**

**-SMARRITO**

## **o - Enumerazione StatoPrenotazione**

Describe lo stato in cui può trovarsi un bagaglio.

Tre valori possibili:

**-CONFERMATA**

**-IN\_ATTESA**

**-CANCELLATA**

## **o - Enumerazione StatoVolo**

Describe lo stato in cui può trovarsi un bagaglio.

Cinque valori possibili:

**-PROGRAMMATO**

**-IN\_RITARDO**

**-ATTERRATO**

**-DECOLLATO**

**-CANCELLATO**

Tutte le classi Enum possiedono un override del metodo `toString()`

Il metodo è sovrascritto in modo da far cancellare “\_” negli stati e mettere al suo posto uno spazio vuoto “ ”.

## 3.2 – Package GUI

### o - Classe BasicBackgroundPanel

La classe **BasicBackgroundPanel** viene utilizzata per aggiungere uno sfondo grafico personalizzato a finestre, pannelli o schermate dell'interfaccia.

### o - Classe ConfirmDialog

**ConfirmDialog** estende **JDialog** e fornisce una finestra di dialogo personalizzata con due pulsanti (Sì e No), utile per ottenere una conferma da parte dell'utente. È utilizzata per mostrare messaggi di conferma.

**--showDialog()**

Rende visibile il dialog. Ritorna true se l'utente preme "Sì" e false se preme "No".

### o – Classe CustomFlightCellRenderer

**CustomFlightCellRenderer** estende **DefaultTableCellRenderer** e si occupa di personalizzare la visualizzazione delle celle in una tabella che rappresenta i voli ed evidenzia i voli in ritardo o cancellati, modificando la formattazione del testo in base ai dati delle celle.

### o – Classe CustomLuggageCellRenderer

**CustomLuggageCellRenderer** estende **DefaultTableCellRenderer** e viene utilizzata per personalizzare l'aspetto delle celle in una tabella che rappresenta i bagagli ed evidenzia i bagagli smarriti cambiando la formattazione del testo in base al valore della cella.

### o – Classe CustomReservationCellRenderer

**CustomReservationCellRenderer** estende **DefaultTableCellRenderer** e viene utilizzata per personalizzare l'aspetto delle celle in una tabella che mostra le prenotazioni ed evidenzia le prenotazioni cancellate modificando il formato del testo delle celle che contengono il valore "Cancellata".

### o – Classe ErrorPanel

Si occupa di creare un pannello personalizzato utilizzato per visualizzare un messaggio di errore in un'interfaccia.

**--showErrorDialog(parent, message, title):**

Permette di mostrare un pannello di errore all'interno di una finestra di dialogo.

## o – Classe FormHelper

La classe FormHelper definisce funzione di utility per la nostra GUI.

I metodi in essa definiti sono:

--bindButtonToTextFields(button, fieldDefaults, optionalFields)

Associa un bottone a dei campi di testo.

Il metodo verrà usato, ad esempio, per rendere il bottone di registrazione utilizzabile una volta che l'utente avrà inserito tutti i dati necessari nei relativi campi di testo.

--bindFormListeners(usernameField, emailField, passwordField, confirmPasswordField, usernameErrMessage, emailErrMessage, passwordRating, confirmPasswordErrMessage, registerButton, controller, defaultUsernameText, defaultPasswordText)

Associa listeners a campi di testo, così che vengano controllate specifiche condizioni (es. password ha almeno 8 caratteri).

## o – Classe ImageLoader

La classe ImageLoader si occupa di caricare icone, immagini e aggiungere uno sfondo ad un Panel (rispettivamente metodi loadIcon, loadImage, addBackgroundPanel), utilizzati per decorare e semplificare l'utilizzo della nostra GUI.

## o – Classe TableSetter

La classe TableSetter si occupa della creazione delle tabelle quali le tabelle dei voli, delle prenotazioni, dei bagagli (rispettivamente create dai metodi setupFlightTable, setupReservationsTable e setupLuggageTable).

## o – Classe UtilFunctionsForGUI

La classe UtilFunctionsForGUI contiene dei metodi utili al funzionamento della GUI:

--addHoverEffect(button)

Aggiunge effetti quando il cursore del mouse si trova su un bottone.

--setupFrame(frame)

Imposta la chiusura del frame e la sua centratura.

--setLayoutAndBackground(frame, contentPanel)

Imposta il layout e uno sfondo.

--disallowSpaces(textComponent)

Rimuove eventuali spazi inseriti da tastiera.

--addPasswordVisibilityToggle(passwordField, passwordEyeIcon, defaultLabelText)

Aggiunge un'icona che, se premuta, rende visibile o no la password al momento della registrazione o del login.

--addTextFieldPlaceholder(textfield, placeholderText)

Aggiunge un testo grigio di base ai campi di testo vuoti e rende i caratteri inseriti dall'utente neri.

--addPasswordFieldPlaceholder(passwordField, placeholderText)

Permette di rappresentare i caratteri inseriti dall'utente nel campo password con '●'.

## o – Classe BagagliAdminGUI

La classe BagagliAdminGUI si occupa di rappresentare l'interfaccia dei bagagli per gli amministratori del sistema.

## o – Classe BagagliGUI

La classe BagagliAdmin si occupa di rappresentare l'interfaccia dei bagagli per gli utenti.

## o – Classe CercaVoloAdminGUI

La classe CercaVoloAdminGUI} rappresenta la schermata di visualizzazione per cercare un volo da parte dell'admin.

## o – Classe CercaVoloGUI

La classe CercaVoloGUI rappresenta la schermata di visualizzazione per cercare un volo da parte dell'utente.

## o – Classe CheckInGUI

La classe CheckInGUI rappresenta la schermata di visualizzazione per il checkin.

## o – Classe Home

La classe Home rappresenta l'interfaccia grafica principale dell'applicazione.

Consente all'utente di scegliere se effettuare il login, registrarsi o uscire dall'applicazione.

Questa schermata contiene tre pulsanti:

-*Login*: apre la finestra di login.

-*Register*: apre la finestra di registrazione.

-*Exit*: chiude l'applicazione.

## **o – Classe HomePage**

**La classe HomePage rappresenta la schermata di visualizzazione dell' homepage dell'utente.**

## **o – Classe HomePageAdmin**

**La classe HomePageAdmin rappresenta la schermata di visualizzazione dell' homepage dell'amministratore del sistema.**

## **o – Classe InserisciVoloGUI**

**La classe InserisciVoloGUI rappresenta la schermata di visualizzazione per inserire un nuovo volo.**

## **o – Classe LoginGUI**

**La classe LoginGUI rappresenta la schermata di visualizzazione per il login di utente/amministratore.**

**Definisce i metodi caricaHomePage() e caricaAdminHomePage() che permettono di arrivare alle rispettive homepage.**

## **o – Classe PrenotazioniGUI**

**La classe PrenotazioniGUI si occupa di costruire l'interfaccia per le prenotazioni dei voli da parte di un utente.**

## **o – Classe RegisterGUI**

**La classe RegisterGUI rappresenta la schermata di visualizzazione per registrarsi.**

## **o – Classe VoliAdminGUI**

**La classe VoliAdminGUI rappresenta la schermata di visualizzazione per gestire i voli da parte dell'admin.**

## **o – Classe VoliGUI**

**La classe VoliGUI rappresenta la schermata di visualizzazione di tutti i voli disponibili da parte dell'utente.**

## 3.3 – Package Custom\_Exceptions

### o – Classe EmailAlreadyExistsException

La classe EmailAlreadyExistsException è un'eccezione personalizzata che estende la classe RuntimeException. Viene utilizzata per segnalare il caso in cui un utente tenta di registrarsi con un'email già presente nel sistema.

### o - Classe ModifyTableException

La classe ModifyTableException è un'eccezione personalizzata che estende la classe RuntimeException. Viene usata per gestire errori che si verificano durante la modifica di tabelle.

### o - Classe UserAlreadyExistsException

La classe UserAlreadyExistsException è un'eccezione personalizzata che estende la classe RuntimeException. Viene utilizzata per segnalare il caso in cui un utente tenta di registrarsi con un nome utente già presente nel sistema.

### o - Classe UserNotFoundException

La classe UserNotFoundException è un'eccezione personalizzata che estende la classe RuntimeException. Viene utilizzata per gestire il caso in cui un utente non venga trovato nel sistema durante il processo di autenticazione o durante operazioni che richiedono la verifica dell'esistenza di un utente.

## 3.4 – Package Controller

### o - Classe Controller

La classe Controller è la componente centrale di controllo dell'applicazione che gestisce la logica dell'interfaccia grafica (GUI) per il sistema di gestione aeroportuale di Napoli. È responsabile dell'interazione tra l'interfaccia utente (GUI), il model e il database (tramite i DAO).

#### Metodi della classe controller:

--registraUtente(nomeUtente, email, password)

--loginUtente (login, password)

Autentica un utente generico o un amministratore e, in base al ruolo, carica i relativi dati (voli gestiti o prenotazioni).

--logout()

Rimuove i dati caricati nella sessione dell'utente e lo disconnette.

--logoutAdmin()

Rimuove i dati caricati nella sessione dell'admin e lo disconnette.

--registraUtente(nomeUtente, email, password)

Registra un nuovo utente se nome ed email sono unici.

**Metodi di supporto per la validazione dei dati inseriti nella GUI:**

--canPressRegister(nomeUtente, email, password, confirmPassword),

--isEmailValid(email)

--isPasswordValid(password, confirmPassword)

--isNameValid(name)

**Metodi relativi alle azioni di un utente generico:**

--getFlightsModel()

--cercaVolo(codice, posti, compagnia, aeroportoOrigine, aeroportoDestinazione, data, orario, ritardo, stato, numeroGate)

Permette all'utente di filtrare la tabella dei voli con criteri specifici.

**Metodi relativi alle azioni di un amministratore:**

--getFlightsAdminModel()

Restituisce i voli gestiti dall'amministratore.

--inserisciVolo(codice, posti, compagnia, aeroportoOrigine, aeroportoDestinazione, data, orario, ritardo, stato, numeroGate)

Consente all'amministratore di aggiungere un nuovo volo.

--salvaModificheDaTabella(table)

Aggiorna le modifiche fatte ai voli direttamente dalla tabella.

--cercaVoloAdmin(codice, posti, compagnia, aeroportoOrigine, aeroportoDestinazione, data, orario, ritardo, stato, numeroGate)

Cerca voli nel contesto amministrativo, con accesso a più informazioni.

--inizializzaPrenotazione(codiceVolo)

Memorizza il codice del volo selezionato per iniziare il processo di prenotazione.

--confermaPrenotazione(nome, secNome, cognome, cf, table)

Completa una prenotazione, genera il posto assegnato, crea i bagagli e aggiorna il DB.

--modificaPrenotazione(numeroBiglietto, stato)

Consente di cambiare lo stato della prenotazione, gestendo anche i bagagli associati.

--getBookingTableModel()

Restituisce la tabella delle prenotazioni dell'utente attuale.

--cercaPrenotazione(codiceVolo, nomePasseggero)

Ricerca prenotazioni per codice volo e nome passeggero.

--getBagsTableModel()

Genera la tabella dei bagagli relativi all'utente.

--segnalaBagaglio(codice)

Cambia lo stato del bagaglio in "smarrito".

--cercaBagaglio( codice, tipo, stato)

Filtrà i bagagli secondo vari criteri.

--getBagsAdminTableModel()

Restituisce la tabella con tutti i bagagli del sistema (per i voli gestiti).

--modificaBagaglio(codice, stato)

Consente all'amministratore di aggiornare lo stato di un bagaglio.

--cercaBagaglioAdmin(( codice, tipo, stato)

Consente una ricerca estesa dei bagagli da parte dell'admin.

--generaPosto()

Genera un posto casuale su un aereo.

**Metodi di parsing sicuro di input:**

--safeParseInteger(value)

--safeParseLocalTime(value)

**Tabelle facenti parte del nostro sistema:**

--flightsTableModel tabella dei voli visibile all'utente generico.

--flightsAdminTableModel: tabella dei voli gestita dagli admin.

--bookingTableModel: tabella delle prenotazioni per ogni utente.

--luggageTableModel: tabella dei bagagli per l'utente

--luggageAdminTableModel: tabella dei bagagli per l'admin.

**Variabili contenenti i nomi delle colonne delle tabelle:**

--luggageColumnNames: definisce le intestazioni di colonna per le tabelle dei bagagli per l'utente

--luggageAdminColumnNames: definisce le intestazioni di colonna per le tabelle dei bagagli per l'admin.

--bookingColumnNames: definisce le intestazioni di colonna per le tabelle delle prenotazioni.

--flightColumnNames: definisce le intestazioni di colonna per le tabelle dei voli sia per l'utente che per l'admin.

## 3.5 – Package DAO

Il package dao contiene interfacce che verranno poi implementate dalle classi del package implementazioni\_postgres\_dao.

Le interfacce in questione sono:

### 1. Amministratore DAO

Contiene i metodi relativi alle azioni che possono essere compiute da un amministratore nel nostro sistema, quali ad esempio l'aggiunta di un volo al db (metodo inserisciVolo())

### 2. UtenteDAO

Contiene metodi che si accertano dell'esistenza di un utente nel database, del login di un utente/admin, o della registrazione di un nuovo utente.

### 3. Utente Generico

Contiene i metodi relativi alle azioni che possono essere compiute da un amministratore nel nostro sistema.

## 3.6 – Package Implementazioni\_postgres\_dao

Contiene le classi che implementano le interfacce del package dao.

### o - Classe AmministratoreImplementazionePostgresDAO

Questa classe implementa l'interfaccia AmministratoreDAO, fornendo un'implementazione a tutti i suoi metodi:

--aggiornaStatoBagaglio(codice, nuovoStato)  
--searchLuggage(model, admin, codice, tipo, stato)  
--showLuggage(luggageModel, admin)  
--addFlights(admin)  
--aggiornaVolo(volo)  
--inserisciVolo(voloDaAggiungere, admin)  
--ricercaVolo(model, admin, codice, posti, compagnia, aeroportoOrigine, aeroportoDestinazione, data, orario, ritardo, stato, numGate)

## **o - Classe UtenteGenerico con Implementazione PostgresDAO**

Questa classe implementa l'interfaccia UtenteGenericoDAO, fornendo un'implementazione a tutti i suoi metodi:

--showFlights(FlightsModel)

Mostra la tabella dei voli.

--segnalaBagaglio(codice)

Modifica lo stato del bagaglio con il codice preso come parametro in 'SMARRITO'

--modificaPrenotazione(numeroBiglietto, stato)

Modifica lo stato della prenotazione con dato numeroBiglietto.

--searchLuggage(model, utente, codice, tipo, stato)

Ricerca dei bagagli di un utente.

--voloDaPrenotare(codiceVolo)

Restituisce il volo da prenotare.

--ricercaVolo(model, codice, posti, compagnia, aeroportoOrigine, aeroportoDestinazione, data, orario, ritardo, stato, numGate)

Ricerca di uno specifico volo.

--ricercaPrenotazione(model, utente, codiceVolo, nomePasseggero)

Ricerca di una specifica prenotazione.

--confirmReservation(nome, secondoNome, cognome, codiceFiscale, tipiBagagli, posto, utente, codiceVolo)

Aggiunge la prenotazione a quelle effettuate da un utente.

--passeggeroExists(conn, codiceFiscale)

Verifica che esista il passeggero nel DB.

--addPassenger(conn, nome, secondoNome, cognome, codiceFiscale)

Inserisce una nuova riga alla tabella Passeggero.

--addReservation(conn, codiceVolo, nomeUtente, codiceFiscale, posto)

Inserisce una nuova riga alla tabella Prenotazione.

--addBagaglio(conn, tipo, codiceFiscale, codiceVolo)

Inserisce una nuova riga alla tabella Bagaglio.

--currentLuggageCode()

Restituisce il codice dell'ultimo bagaglio preso.

--currentReservationCode()

Restituisce il codice dell'ultima prenotazione effettuata.

--caricaPrenotazioniUtente(utente)

Carica le prenotazioni dell'utente.

## o - Classe UtenteImplementazionePostgresDAO

Questa classe implementa l'interfaccia UtenteDAO, fornendo un'implementazione a tutti i suoi metodi:

--esisteUtente(nomeUtente)

Verifica che l'utente con il dato nome sia registrato all'interno del database.

--esisteEmail(email)

Verifica che l'utente con l'email passata per parametro sia registrato all'interno del database.

--loginUtente(login, passwordInput)

Controlla username/email e password e prova ad effettuare il login all'account amministratore o utente generico.

--registraUtente(utente)

Registra un nuovo utente nel database.

## o - Classe UserUtilFunctionsForDAO

Una classe che contiene funzioni di utility per gli utenti. Viene estesa dalla classe AmministratoreImplementazionePostgresDAO e contiene i metodi:

--addFlightRows(ps, model)

Aggiunge al model le righe contenenti i voli selezionati nel prepared statement ps.

--searchLuggageInternal(model, query, initialParams, codice, tipo, stato)

Funzione di ricerca dei bagagli.

--ricercaVololInternal(model, query, initialParams, codice, posti, compagnia, aeroportoOrigine, aeroportoDestinazione, data, orario, ritardo, stato, numGate)

Funzione di ricerca dei voli.

## 3.7 – Package Database

### o - Classe ConnessioneDatabase

Si occupa di gestire la connessione al database attraverso il metodo getInstance().

## 4. Sequence Diagram

Di seguito vediamo qualche esempio di esecuzione di metodi del nostro programma attraverso specifici sequence diagrams.

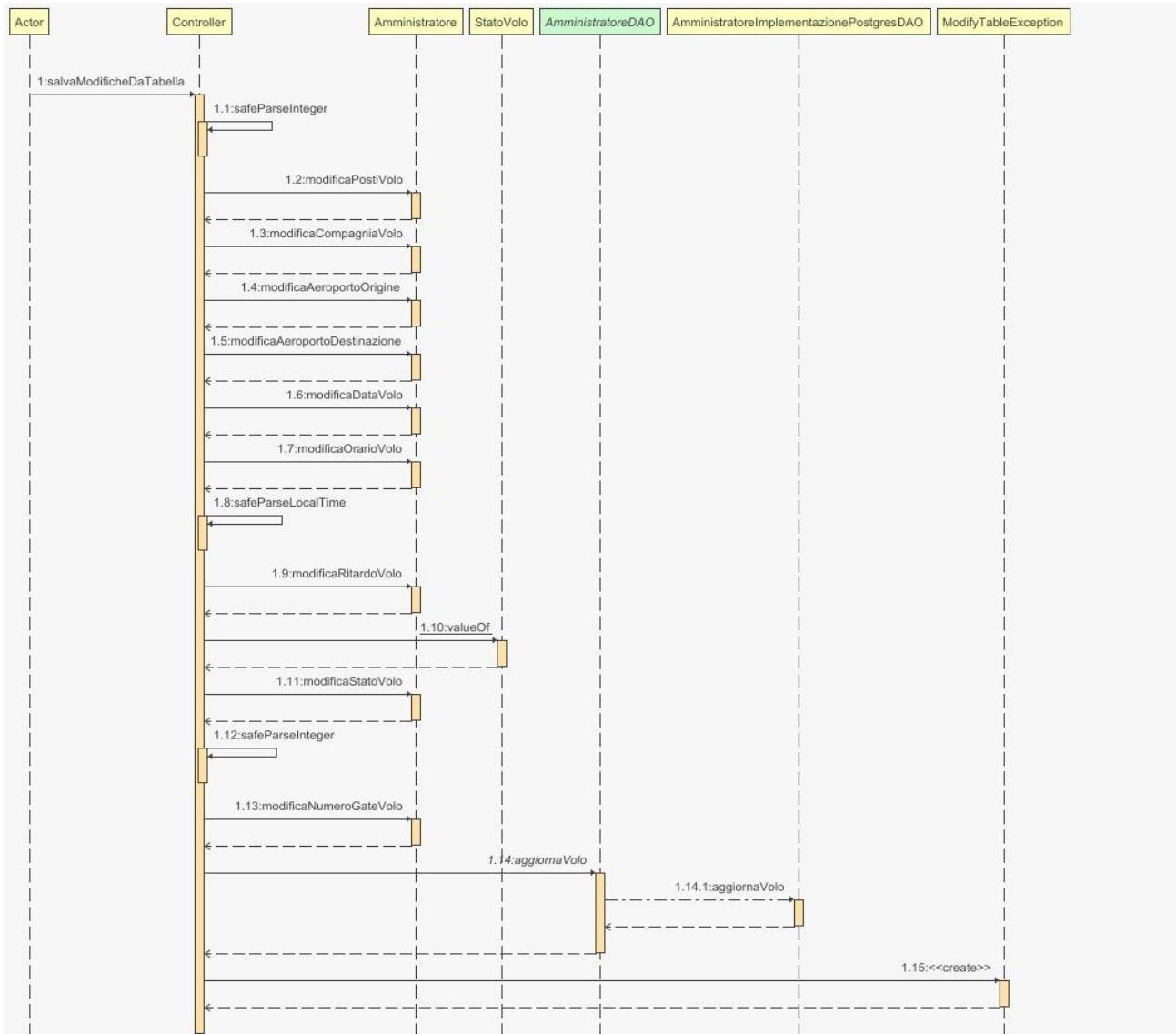
### 4.1 – Primo Esempio

Il primo metodo che andiamo ad analizzare è salvaModificheDaTabella() della classe Controller. Vediamo prima il codice Java attraverso cui viene implementato. 

```
public void salvaModificheDaTabella(JTable table) throws ModifyTableException { 2 usages ± Antonio-Pocemento
    try {
        DefaultTableModel model = (DefaultTableModel) table.getModel();
        Amministratore admin = (Amministratore) utenteAutenticato;
        List<Volo> voliGestiti = admin.getVoliGestiti();

        for (int i = 0; i < model.getRowCount(); i++) {
            String codice = model.getValueAt(i, column: 0).toString();
            Volo v;
            for (Volo volo : voliGestiti) {
                v = volo;
                if (!Objects.equals(v.getCodice(), codice)) {
                    // aggiorna i campi del volo esistente
                    admin.modificaPostivoVolo(v, safeParseInteger(model.getValueAt(i, column: 1)));
                    admin.modificaCompagniaVolo(v, model.getValueAt(i, column: 2).toString());
                    admin.modificaAeroportoOrigine(v, model.getValueAt(i, column: 3).toString());
                    admin.modificaAeroportoDestinazione(v, model.getValueAt(i, column: 4).toString());
                    admin.modificaDataVolo(v, LocalDate.parse(model.getValueAt(i, column: 5).toString(), DateTimeFormatter.ofPattern(DEFAULT_DATE_FORMAT)));
                    admin.modificaOrarioVolo(v, LocalTime.parse(model.getValueAt(i, column: 6).toString()));
                    admin.modificaRitardoVolo(v, safeParseLocalTime(model.getValueAt(i, column: 7)));
                    admin.modificaStatoVolo(v, StatoVolo.valueOf(model.getValueAt(i, column: 8).toString().replace(target: " ", replacement: "_").toUpperCase()));
                    admin.modificaNumeroGateVolo(v, safeParseInteger(model.getValueAt(i, column: 9)));
                    amministratoreDAO.aggiornaVolo(v);
                    break;
                }
            }
        }
    } catch (Exception ex) {
        throw new ModifyTableException(ex.getMessage());
    }
}
```

**Per evitare di complicare eccessivamente il sequence diagram corrispondente, valutiamo solo l'esecuzione dei metodi da noi implementati. Cominciamo quindi dal corpo del costrutto if.**



La nostra analisi parte, quindi, dalla chiamata del metodo `salvaModificheDaTabella()` da parte dell'utente, che utilizza un'istanza della classe **Controller**. Come indicato precedentemente saltiamo le istruzioni che non vogliamo rappresentare, fino alla prima di nostro interesse.

L'esecuzione del metodo `safeParseInteger()` viene rappresentata come un self-message, in quanto questo metodo è definito proprio nella classe **Controller**.

I prossimi metodi (`modificaPostiVolo()`, `modificaCompagniaVolo()`, `modificaAeroportoOrigine`, `modificaDataVolo()`, `modificaOrarioVolo()`) richiedono tutti un'istanza della classe **Amministratore** (in quanto ovviamente metodi di questa classe).

I metodi **safeParseLocalTime()**, **modificaStatoVolo()**, **safeParseInteger()** e **modificaNumeroGateVolo()** agiscono allo stesso modo dei precedenti, quindi sarebbe irrilevante soffermarci.

Il metodo **aggiornaVolo()** è dichiarato nell'interfaccia **AmministratoreDAO**, ma implementato dalla classe **AmministratoreImplementazionePostgresDAO**, verrà quindi utilizzato da una sua istanza.

Per ultima raffiguriamo un'eccezione, che comporta la creazione di una istanza della classe **ModifyTableException**.

## 4.2 – Secondo Esempio

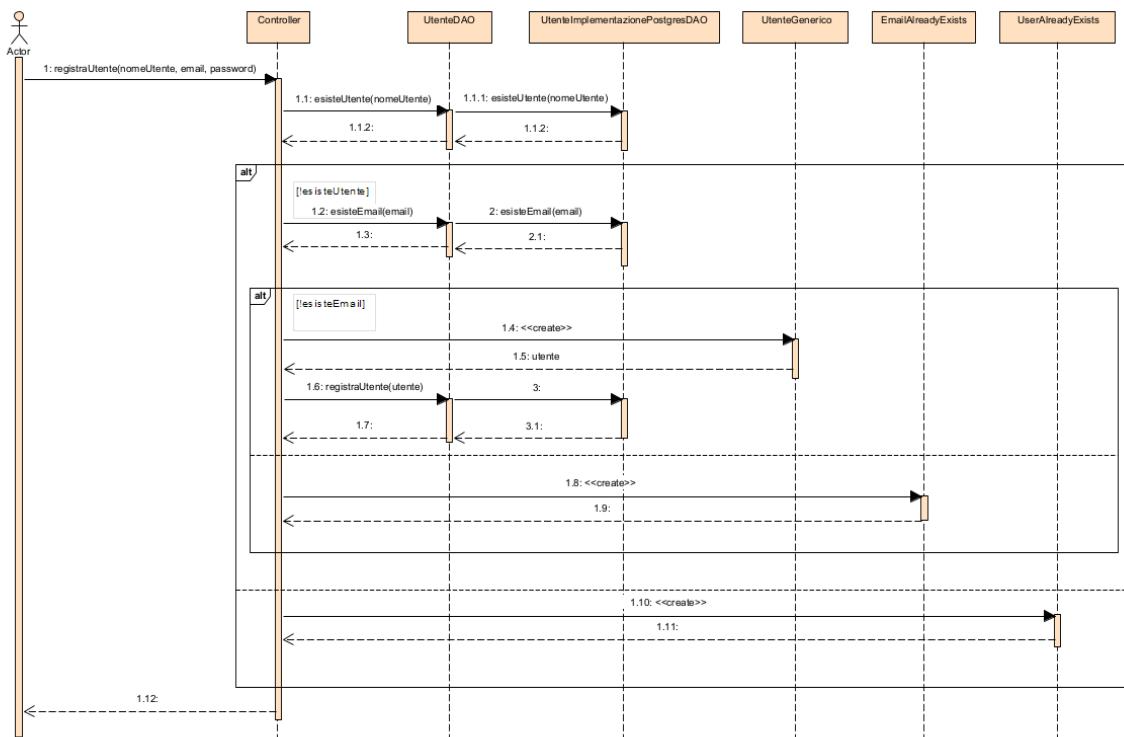
Il secondo metodo che andiamo ad approfondire è **registraUtente(...)**, della classe **Controller**.

Come in precedenza, vediamo prima il codice Java con cui è stato implementato.

```
public void registraUtente(String nomeUtente, String email, char[] password) throws SQLException { 1 usage ↗ Antonio-Pocomento
    if(!utenteDAO.esisteUtente(nomeUtente)) {
        if(!utenteDAO.esisteEmail(email)) {
            UtenteGenerico utente = new UtenteGenerico(nomeUtente, email, new String(password).trim()); //in memoria
            utenteDAO.registraUtente(utente); //su DB
        }
        else throw new EmailAlreadyExistsException("È già presente un account che usa questa email");
    }
    else
        throw new UserAlreadyExistsException("Nome Utente già in uso");
}
```

A differenza dello scorso esempio il metodo in questione è possibile analizzarlo nella sua interezza, in quanto meno articolato.

Il sequence diagram corrispondente è mostrato qui sotto. [⬇️](#)



Essendo questo metodo della classe Controller, la lifeline più importante è proprio quella dell'entità di tipo Controller, da cui, come vediamo nell'immagine, verranno inviati tutti i messaggi di esecuzione degli altri metodi.

Inoltre, abbiamo bisogno di rappresentare due costrutti if-else, di cui uno innestato dentro l'altro. Utilizziamo quindi due blocchi ALT.

I primi metodi eseguiti sono `esisteUtente(...)` e `esisteEmail(...)` ed agiscono allo stesso modo. Sono, infatti, entrambi dichiarati nell'interfaccia `UtenteDAO`, ma ovviamente implementati in una classe apposita `UtenteImplementazionePostgresDAO`. Vediamo, quindi, nel diagramma rappresentate sia le linee di vita dell'interfaccia che della classe, insieme ai relativi messaggi di ritorno.

È importante anche notare che i risultati delle esecuzioni di questi metodi vengono valutati come condizioni dei blocchi ALT, indicando se le istruzioni successive da eseguire sono quelle del corpo dell'`if` oppure dell'`else`.

Il prossimo metodo eseguito è il costruttore di `UtenteGenerico`, che crea un'istanza della classe `UtenteGenerico` e restituisce un riferimento inserito nella variabile `utente`.

Il metodo `registraUtente()` implementato dalla classe `UtenteImplementazionePostgresDAO` utilizza proprio la variabile `utente`.

**Le ultime due istruzioni rappresentate sono le creazioni degli oggetti delle classi di eccezione EmailAlreadyExists e UserAlreadyExists, opportunamente collocate negli spazi else del blocco ALT di riferimento.**