**UNIVERSITÀ DEGLI STUDI DI NAPOLI “FEDERICO II”**

**DIPARTIMENTO DI INGEGNERIA**

**ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE**

**CORSO DI LAUREA IN INFORMATICA**

PROGETTO CORSO

PROGRAMMAZIONE OBJECT-ORIENTED

# Titolo

TRACCIA 3 - AEROPORTO

Candidati
Mario Laurato
N86005474
Mario Menniti
N86005523

Docenti
Prof. Porfirio Tramontana
Dott. Bernardo Breve

Anno Accademico 2024/2025

# TRACCIA 3: Aeroporto

Si sviluppi un sistema informativo per la gestione dell'aeroporto di Napoli, composto da una base di dati relazionale e da un applicativo Java con interfaccia grafica realizzata con Swing. Questo sistema deve consentire di organizzare e monitorare le operazioni aeroportuali in modo efficiente e intuitivo.

Il sistema può essere utilizzato da utenti autenticati tramite una login e una password. Gli utenti sono suddivisi in due ruoli: utenti generici, che possono prenotare voli, e amministratori del sistema, che gestiscono l'inserimento e l'aggiornamento dei voli.

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 amministratori del sistema hanno la possibilità di inserire nuovi voli e aggiornare le informazioni sui voli esistenti.

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). Gli utenti possono cercare e modificare le proprie prenotazioni in base al nome del passeggero o al numero del volo.

Il sistema gestisce anche i gate di imbarco (identificati da un numero), assegnandoli ai voli in partenza. Gli amministratori possono modificare l'assegnazione dei gate.

Il sistema consente agli utenti di visualizzare aggiornamenti sui voli prenotati accedendo alla propria area personale, dove possono controllare eventuali ritardi, cancellazioni o variazioni direttamente dall'interfaccia. Inoltre, all'interno della homepage degli utenti viene mostrata una tabella con gli orari aggiornati dei voli in partenza e in arrivo, fornendo una panoramica immediata delle operazioni aeroportuali.

Infine, il sistema permette di eseguire ricerche rapide per trovare voli, passeggeri e bagagli in base a diversi criteri. Le informazioni più importanti vengono evidenziate, come i voli in ritardo o cancellati, per facilitare la gestione delle operazioni aeroportuali.

**Requisito per gruppi da tre persone.**

Un’altra funzione importante è il monitoraggio dei bagagli. Ogni bagaglio è associato al volo del passeggero e 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, che aggiornano il sistema indicando se è stato caricato sull'aereo o è disponibile per il ritiro. Gli utenti possono visualizzare lo stato aggiornato del proprio bagaglio tramite l'interfaccia del sistema. Se un bagaglio viene smarrito, l’utente può segnalarlo direttamente nel sistema. Gli amministratori possono accedere a un modulo dedicato per visionare le richieste di smarrimento.

# INTRODUZIONE

Il presente documento introduce la concettualizzazione e lo sviluppo di un sistema informativo dedicato all'Aeroporto di Napoli, con l’obiettivo di ottimizzare l'organizzazione e il monitoraggio delle sue molteplici attività.

Il sistema implementa una gestione completa dei voli, sia per i voli in arrivo presso lo scalo napoletano sia quelli in partenza.

Nel presente incarico iniziale, l'obiettivo è quello di produrre i seguenti:

- Un diagramma delle classi di dominio che rappresenti il modello concettuale del problema, corredato da una descrizione testuale che giustifichi le scelte progettuali intraprese.
- Un modello dei package del progetto software, che delinei l'organizzazione delle classi identificate nel diagramma di dominio, tradotte nel linguaggio di programmazione Java.

# CLASSI DI DOMINIO

## USER *(Utente)*

La classe rappresenta un utente e contiene due attributi privati

```
private String username;
private String password;
```

Viene fornito un costruttore

```
public User(String username, String password)  {
   this.username = username;
   this.password = password;
}
```

Include anche i metodi getter e setter per ottenere le credenziali.

## CLIENT *(Cliente)*

Eredita da **USER** i metodi e gli attributi, ed inoltre rappresenta i clienti del sistema:

```
private String name;
private String surname;
private String DOB; // Date of Birth
private String SSN; // Social Security Number
private String email;
private String phoneNumber;
```

Un metodo costruttore

```
public Client(String username, String password, String name, String surname,
String DOB, String SSN, String email, String phoneNumber) {
   super(username, password);
   this.name = name;
   this.surname = surname;
   this.DOB = DOB;
   this.SSN = SSN;
   this.email = email;
   this.phoneNumber = phoneNumber;
}
```

Includendo, oltre i metodi getter e setter,  anche i metodi per:

- creare una nuova prenotazione
- eliminare una prenotazione effettuata
- aggiornare una prenotazione effettuata
- visualizzare le prenotazioni
- avviare un ticket per il bagaglio smarrito

```
public void newBooking()
public void deleteBooking()
public void updateBooking()
public void viewBooked()
public void reportLostLuggage()
```

La relazione di associazione con BOOKING (0..*), indicando che un cliente può effettuare una o più prenotazioni.

## ADMIN *(Amministratore)*

Eredita da USER i metodi e gli attributi, ed inoltre rappresenta gli amministratori del sistema:

```
private String opCode;
```

Un metodo costruttore

```
public Admin(String username, String password, String opCode) {
   super(username, password);
   this.opCode = opCode;
}
```

Includendo, oltre i metodi getter e setter,  anche i metodi per:

- inserire un volo
- aggiornare un volo
- vedere tutti i voli
- eliminare un volo
- vedere tutti i bagagli smarriti
- aggiornare lo status di un bagaglio

```
public void insertFlight()
public void updateFlight()
public void viewAllFlights()
public void deleteFlight()
public void viewLost()
public void updateLostLuggage()
```

Le relazioni:

- MANAGE con FLIGHT (1..*), indicando che un amministratore gestisce uno o più voli.
- UPDATE con BOOKING (0..*), indicando che un amministratore può aggiornare uno o più prenotazoni.
- REPORT con LUGGAGE (0..*), indicando che un amministratore può aggiornare uno o più bagagli.
- CHANGE con GATE (0..*), indicando che un amministratore può modificare zero o più gate.

## BOOKING *(Prenotazione)*

Rappresenta una prenotazione di un volo:

```
private String bookingCode;
private String ticketNumber;
private String seatAssigned;
private BookingStatus status;
```

Un metodo COSTRUTTORE

```
public Booking(String bookingCode, String ticketNumber, String seatAssigned,
BookingStatus status) {
   this.bookingCode = bookingCode;
   this.ticketNumber = ticketNumber;
   this.seatAssigned = seatAssigned;
   this.status = status;
}
```

Includendo, oltre i metodi getter e setter,  anche i metodi per:

- effettuare il check in
- stampare la prenotazione

```
public void checkIn()
public void printTicket()
```

La relazione di associazione con CLIENT (0..*), indicando che una prenotazione è associata a zero o più clienti.

La relazione di associazione con FLIGHT (*..1), indicando che una o più prenotazioni sono per un singolo volo.

La relazione di associazione con LUGGAGE (0..*), indicando che una prenotazione può avere zero o più bagagli associati.

## FLIGHT *(Volo)*

Rappresenta un volo specifico:

```
private String flightNumber;
private String flightCompany;
private String date;
private String departureTime;
private FlightStatus status;
private int delayed;
private String departureAirport;
private String destinationAirport;
```

Un metodo COSTRUTTORE

```
public Flight(String flightNumber, String flightCompany, String date, String
departureTime, FlightStatus status, int delayed, String departureAirport, String
destinationAirport) {
   this.flightNumber = flightNumber;
   this.flightCompany = flightCompany;
   this.date = date;
   this.departureTime = departureTime;
   this.status = status;
   this.delayed = delayed;
   this.departureAirport = departureAirport;
   this.destinationAirport = destinationAirport;
}
```

Includendo i metodi getter e setter.

La relazione di associazione con BOOKING (1..*), indicando che un volo può avere una o più prenotazioni.
La relazione di associazione con GATE (1..*), indicando che un volo può essere assegnato a uno o più gate.
La relazione di associazione con AIRPORT (1..1) con ruolo FROM, indicando l'aeroporto di partenza.
La relazione di associazione con AIRPORT (1..1) con ruolo TO, indicando l'aeroporto di destinazione.

## GATE *(Gate di Imbarco)*

Rappresenta un gate di imbarco:

```
private String gateNumber;
private String terminal;
```

Un metodo COSTRUTTORE

```
public Gate(String gateNumber, String terminal) {
   this.gateNumber = gateNumber;
   this.terminal = terminal;
}
```

Includendo i metodi getter e setter.

La relazione di associazione con FLIGHT (*..1), indicando che un gate può essere assegnato a uno o più voli.
La relazione di associazione con BOOKING (1..*), indicando che un gate può essere assegnato a uno o più prenotazioni.

## LUGGAGE *(Bagaglio)*

Rappresenta un bagaglio:

```
private String luggageNumber;
private LuggageStatus luggageStatus;
```

Un metodo COSTRUTTORE

```
public Luggage(String luggageNumber, LuggageStatus luggageStatus) {
   this.luggageNumber = luggageNumber;
   this.luggageStatus = luggageStatus;
}
```

Includendo i metodi getter e setter.

La relazione di associazione con BOOKING (*..1), indicando che più bagagli possono essere associati ad una singola prenotazione.
La relazione di associazione con ADMIN (*..0), indicando che più bagagli possono essere gestiti da un operatore.

## AIRPORT *(Aeroporto)*

Rappresenta un aeroporto:

```
private String IATA_code;
private String IACA_code;
private String city;
private String nation;
private String name;
```

Un metodo COSTRUTTORE

```
public Airport(String IATA_code, String IACA_code, String city, String nation, String
name) {
   this.IATA_code = IATA_code;
   this.IACA_code = IACA_code;
   this.city = city;
   this.nation = nation;
   this.name = name;
}
```

Includendo i metodi getter e setter.

La relazione di associazione con FLIGHT (1..1), indicando l'aeroporto di partenza.
La relazione di associazione con FLIGHT (1..1), indicando l'aeroporto di destinazione.

## Enumerazioni:

**Status *(Stato Prenotazione)*:** PENDING, CONFIRMED, DELETED.

**FlightStatus *(Stato Volo)*:** SCHEDULED, BOARDING, DEPARTED, ARRIVING, LATE, CANCELED, RESCHEDULED.

**LuggageStatus *(Stato Bagaglio)*:** BOARDED, LOST, AVAILABLE AT PICKUP.

## DIAGRAMMA UML