Flight Planner

Ingegneria del software

MARIA LI

October 2024

Ingegneria Informatica

Indice

1 Introduzione

1.1 Motivazione e Funzionamento

Il presente elaborato prende spunto da un'esperienza personale vissuta l'anno scorso, quando è stato necessario effettuare per la prima volta una ricerca di voli e una prenotazione aerea. L'idea è stata inoltre ispirata da un seminario tenuto durante il corso di Basi di Dati, in cui è stato presentato un caso studio riguardante la gestione di un database aeroportuale. L'applicativo sviluppato consente la gestione dei voli e limita l'accesso alle funzionalità a disposizione esclusivamente agli utenti registrati.
In fase iniziale, l'applicativo prevede l'autenticazione dell'utente o, in alternativa, la sua registrazione mediante inserimento di username, password ed email, dove l'username deve essere univoco all'interno del sistema. È inoltre presente un profilo amministratore con credenziali predefinite, che permette l'accesso alle funzionalità riservate. Il sistema offre due interfacce distinte:

Interfaccia Admin

Attraverso l'interfaccia dedicata all'amministratore è possibile:

- visualizzare un elenco con le informazioni principali dei passeggeri registrati, inclusi i passeggeri aggiuntivi. Questi ultimi, non essendo registrati autonomamente nell'applicazione, possono tuttavia usufruire di alcune funzionalità tramite un utente registrato, quali la prenotazione e la cancellazione di un'intera prenotazione di volo o di singoli biglietti, il cambio di posto e l'aggiunta di bagagli;
- consultare la lista completa dei voli disponibili, riportando informazioni essenziali come il codice del volo, i codici univoci degli aeroporti di partenza e arrivo e le rispettive date e orari;
- aggiungere e rimuovere elementi dai file CSV gestiti dal sistema, tra cui voli, aeroporti, rotte e posti sui voli. In caso di cancellazione di un volo, i passeggeri che hanno selezionato la preferenza di ricevere notifiche relative alle cancellazioni riceveranno un avviso adeguato;
- modificare gli orari di partenza o arrivo dei voli e notificare i passeggeri registrati ai voli interessati, inviando messaggi specifici per situazioni come ritardi, offerte speciali o cambiamenti di gate. Le notifiche verranno inviate esclusivamente ai passeggeri che hanno selezionato la preferenza per tale tipologia di avvisi e saranno trasmesse attraverso il canale di comunicazione da loro scelto;
- gestire il caricamento e l'aggiornamento dei prezzi dei posti a seconda della classe di volo, con tre classi previste: Economy, Business e First;
- effettuare il logout dal sistema.

Interfaccia Passeggero

Attraverso l'interfaccia dedicata ai passeggeri è possibile:

- visualizzare un elenco dettagliato di tutte le prenotazioni effettuate, inclusi i dati relativi ai singoli biglietti e ai bagagli associati;
- effettuare nuove prenotazioni di voli fornendo il codice del volo e i dati personali dei passeggeri associati alla prenotazione. È possibile selezionare i posti a sedere, specificare quantità e dimensioni dei bagagli da includere, e procedere al pagamento finale.
  Se un passeggero aggiuntivo si registra successivamente nell'applicazione, il sistema eseguirà un confronto e una fusione automatica dei dati esistenti con i nuovi, mantenendo però le prenotazioni riservate: il nuovo passeggero non potrà visualizzare prenotazioni effettuate per lui da altri utenti. Tuttavia, il sistema riconosce che il passeggero è già registrato per determinati voli, impedendogli di effettuare doppie prenotazioni sugli stessi voli;

- annullare l'intera prenotazione effettuata, purché la richiesta avvenga entro 24 ore dall'acquisto o almeno 7 giorni prima della data di partenza del volo. In questo caso, è previsto un rimborso pari al 40% del costo di ciascun biglietto, esclusi i costi dei bagagli;
- cancellare singoli biglietti all'interno di una prenotazione, con le stesse condizioni valide per la cancellazione dell'intera prenotazione. Se l'unico biglietto in una prenotazione viene annullato, anche la prenotazione sarà rimossa;
- aggiungere ulteriori bagagli a un biglietto già prenotato, nei limiti della capacità consentita. È possibile aggiungere due tipologie di bagagli, a mano o da stiva, ciascuno con specifiche restrizioni sulle dimensioni e sui pesi in base alla classe del volo;
- ricercare voli specificando il luogo di partenza e arrivo (sia tramite codici aeroportuali sia tramite nomi di città) e la data del viaggio. I risultati della ricerca forniscono dettagli sul volo e sulla rotta, ma se i dati inseriti non corrispondono a voli disponibili, non verranno mostrati risultati. Se non si utilizza il codice univoco dell'aeroporto, la ricerca potrebbe includere anche voli con partenze o destinazioni simili;
- impostare preferenze di notifica personalizzate, scegliendo tra cancellazione voli, cambio gate, offerte speciali e ritardi. È possibile specificare il canale di notifica preferito, tra email e SMS; in caso di scelta SMS, è richiesto l'inserimento di un numero di cellulare comprensivo di prefisso internazionale e che rispetti la lunghezza standard;
- definire un metodo di pagamento personale. In caso di mancata impostazione, il metodo di pagamento potrà essere selezionato durante la procedura di prenotazione, poiché è obbligatorio per concludere la transazione;
- modificare le credenziali personali, incluso email e password;
- disiscriversi dall'applicazione, operazione che comporta la cancellazione di tutti i dati registrati nel sistema, inclusi quelli relativi a eventuali passeggeri aggiuntivi;
- effettuare il logout dal sistema.

1.2 Strumenti utilizzati e metodo di realizzazione

L'applicazione è stata realizzata con il linguaggio Java attraverso l'uso di IDE Intellij IDEA.
Esso proprone all'utente una GUI (Graphic User Interface) realizzato attraverso **JavaFx**[1], un framework per Java, sviluppato da Oracle per la creazione di interfacce utente moderne e ricche di funzionalità in applicazioni Java. Nel progetto, la scelta di JavaFX per la realizzazione dell'interfaccia è stata motivata dalla necessità di avere una soluzione moderna e flessibile, capace di supportare componenti grafiche avanzate e interazioni dinamiche, come pulsanti, finestra, griglie e dialoghi. A differenza di Swing, JavaFX offre una gestione più intuitiva per layout complessi e una migliore integrazione con il modello MVC, facilitando l'organizzazione del codice e migliorando l'esperienza utente. Inoltre, dalle ricerche effettuate su internet, JavaFX risulta generalmente più semplice da utilizzare rispetto a Swing, grazie alla sua struttura più moderna e alla facilità con cui gestisce elementi grafici complessi.
L'approccio di sviluppo ha previsto la codifica diretta, partendo dalla creazione delle singole classi base, passando alla definizione dei manager e culminando nella configurazione dell'applicazione completa. Questo processo ha permesso di costruire gradualmente la logica funzionale, senza passare attraverso Mockup. Tuttavia, è stato preso spunto da applicazioni già note, come quella di EasyJet, per la loro intuitività e praticità, caratteristiche che si desideravano implementare anche nel progetto. Tale approccio ha assicurato coerenza con un modello già testato e apprezzato dagli utenti, consentendo di concentrarsi fin dall'inizio su un'interfaccia semplice orientata all'usabilità.
Nel progetto è stato implementato il pattern architetturale **Model-View-Controller** (MVC), il quale consente una chiara separazione delle responsabilità tra le diverse componenti del sistema. In questo contesto,

il Model è rappresentato dalle classi che gestiscono i dati e la logica di business. La View è costituita dall'interfaccia utente sviluppata con JavaFX, che si occupa di presentare le informazioni all'utente e di raccogliere i dati in input. Infine, il Controller è responsabile della gestione delle interazioni tra il Model e la View, orchestrando il flusso di dati e le azioni dell'utente.
Nel contesto del progetto, la classe `FlightPlannerApp` svolge contemporaneamente il ruolo di Controller e di View. Essa implementa la logica di gestione degli eventi, come le azioni associate ai pulsanti, attraverso le istanze di `FlightPlanner` e `AuthManager`, e funge anche da View, in quanto contiene gli elementi visivi, quali finestre, dialoghi e pulsanti. Inoltre, la classe richiede l'inserimento di dati da parte dell'utente e raccoglie queste informazioni per una successiva elaborazione da parte del Controller.
In aggiunta al pattern MVC, è stato implementato anche il pattern **Observer**, che risulta particolarmente utile per gestire la comunicazione tra oggetti in modo efficiente e flessibile. In questo contesto, il pattern Observer consente al sistema di notificare automaticamente i passeggeri riguardo ai cambiamenti di stato dei voli, come ritardi, cancellazioni, cambi di gate o offerte speciali. La classe `Flight` funge da soggetto (**Subject**), mentre i passeggeri, registrati per ricevere aggiornamenti su determinati voli, sono gli osservatori (**Observers**) che ricevono queste notifiche. Questo approccio migliora l'efficienza dell'applicazione, inviando aggiornamenti solo ai passeggeri interessati e secondo le loro preferenze di notifica, evitando comunicazioni non necessarie e migliorando la reattività del sistema.
Il progetto è suddiviso in tre pacchetti distinti:

- **businessLogic**: Questo pacchetto ospita la logica di business e include il Controller e la View;
- **domainModel**: Questo pacchetto ospita tutte le classi che rappresentano le entità del dominio, come Passenger, Booking, Flight, Ticket, ecc;
- **manager_CSV**: Questo pacchetto contiene i manager responsabili principalmente per la gestione delle operazioni CRUD sui dati, che vengono letti e scritti nei file CSV.

Questa organizzazione modularizzata delle componenti favorisce la manutenibilità e la testabilità del sistema, rendendo il progetto più scalabile e facilmente gestibile.
Per rappresentare in modo chiaro l'interazione tra i vari componenti dell'applicazione, sono stati creati un diagramma delle classi e un diagramma dei casi d'uso in UML utilizzando il software StarUML.
È stato inoltre implementato un livello di persistenza mediante file CSV, gestito per operazioni di lettura e scrittura tramite la libreria **OpenCSV** [2][3]. I file CSV di base sono collocati nella directory *resources* della directory *main*, per agevolare la lettura, mentre una cartella *csv* separata ospita i file aggiornati a seguito delle interazioni degli utenti.
Infine per garantire l'affidabilità del software, sono stati realizzati test coprendo tutte le principali funzionalità dell'applicazione. Questi test si trovano nella directory *test* offerta da IntelliJ e utilizzano JUnit 5 [4] come framework principale. Inoltre, è stato necessario l'utilizzo di TestFX [5] per testare l'interfaccia grafica JavaFX, simulando l'interazione dell'utente con l'applicazione. TestFX permette di validare i comportamenti e le risposte delle componenti grafiche in un ambiente controllato, ideale per evitare malfunzionamenti legati all'interfaccia.
Per mantenere una gestione chiara e coerente dei test, anche la struttura di questi segue la stessa suddivisione in tre pacchetti che caratterizza l'organizzazione del progetto. Ogni pacchetto di test è specificamente dedicato alla verifica delle funzionalità del corrispondente pacchetto del sistema:

- **businessLogicTest**: I test in questo pacchetto si concentrano sulla logica di business, verificando che il Controller e la View interagiscano correttamente e che le funzionalità siano implementate secondo le specifiche. Questi test includono sia test unitari che test funzionali per validare le operazioni legate agli eventi dell'interfaccia utente e la gestione dei flussi di dati;
- **domainModelTest**: Questo pacchetto contiene i test delle classi di dominio. I test verificano la corretta gestione dei dati e la coerenza tra le entità del sistema, oltre alla validità delle operazioni di manipolazione dei dati, come la creazione e la modifica degli oggetti;

- **manager_CSVTest**: I test in questo pacchetto si occupano della gestione delle operazioni CRUD sui file CSV. Verificano che i manager siano in grado di leggere e scrivere correttamente i dati nei file CSV, garantendo la persistenza delle informazioni in modo affidabile.

Questa organizzazione dei test in pacchetti separati, allineata alla struttura del progetto, offre numerosi vantaggi. In primo luogo, consente di mantenere il codice di test ben strutturato e facilmente navigabile, facilitando la gestione e l'aggiornamento dei test man mano che il progetto cresce. Inoltre, separando i test in base alle funzionalità dei pacchetti, è possibile eseguire test più mirati, migliorando la copertura del codice e riducendo il rischio di errori. Infine, questa divisione riflette la modularità del progetto, semplificando l'integrazione di nuove funzionalità e la manutenzione del sistema.

2 Progettazione

2.1 Use Case Diagram

Nell'analisi dei requisiti, lo Use Case Diagram svolge un ruolo essenziale nel rappresentare le relazioni tra gli attori e i casi d'uso principali dell'applicazione. Questo tipo di diagramma illustra le funzionalità offerte dal sistema, evidenziando i principali flussi di interazione e delineando in modo chiaro i ruoli degli attori e le azioni disponibili per ciascuno.
Un attore in un diagramma dei casi d'uso è un'entità esterna, spesso un utente, che interagisce con il sistema e sfrutta le varie funzionalità. Nel contesto in esame, sono presenti due attori principali: l'admin e il passeggero. Entrambi svolgono un ruolo centrale nell'applicazione, rappresentando i principali utilizzatori del sistema e accedendo a specifici set di funzionalità a seconda dei privilegi associati al loro ruolo.

Dall'analisi del diagramma dei casi d'uso (Figura 1), emerge chiaramente la centralità del login, un'operazione preliminare e imprescindibile per accedere alle funzionalità del programma destinate all'utente registrato. Inoltre, il diagramma illustra le funzionalità principali dell'applicazione, organizzate in casi d'uso che offrono una visione dettagliata delle possibili interazioni tra il sistema e gli utenti. Grazie alle relazioni di dipendenza e inclusione rappresentate graficamente, è possibile comprendere come le varie funzionalità siano interconnesse. In particolare, la dipendenza tra le azioni dell'admin e del passeggero permette di evidenziare come le modifiche, le aggiunte o le rimozioni apportate dall'admin influenzino direttamente l'esperienza utente del passeggero. Viceversa, le azioni di registrazione o disiscrizione da parte di un utente impattano direttamente sulla lista dei passeggeri visibile nell'interfaccia dell'admin, consentendo un aggiornamento in tempo reale dei dati relativi agli utenti attivi.

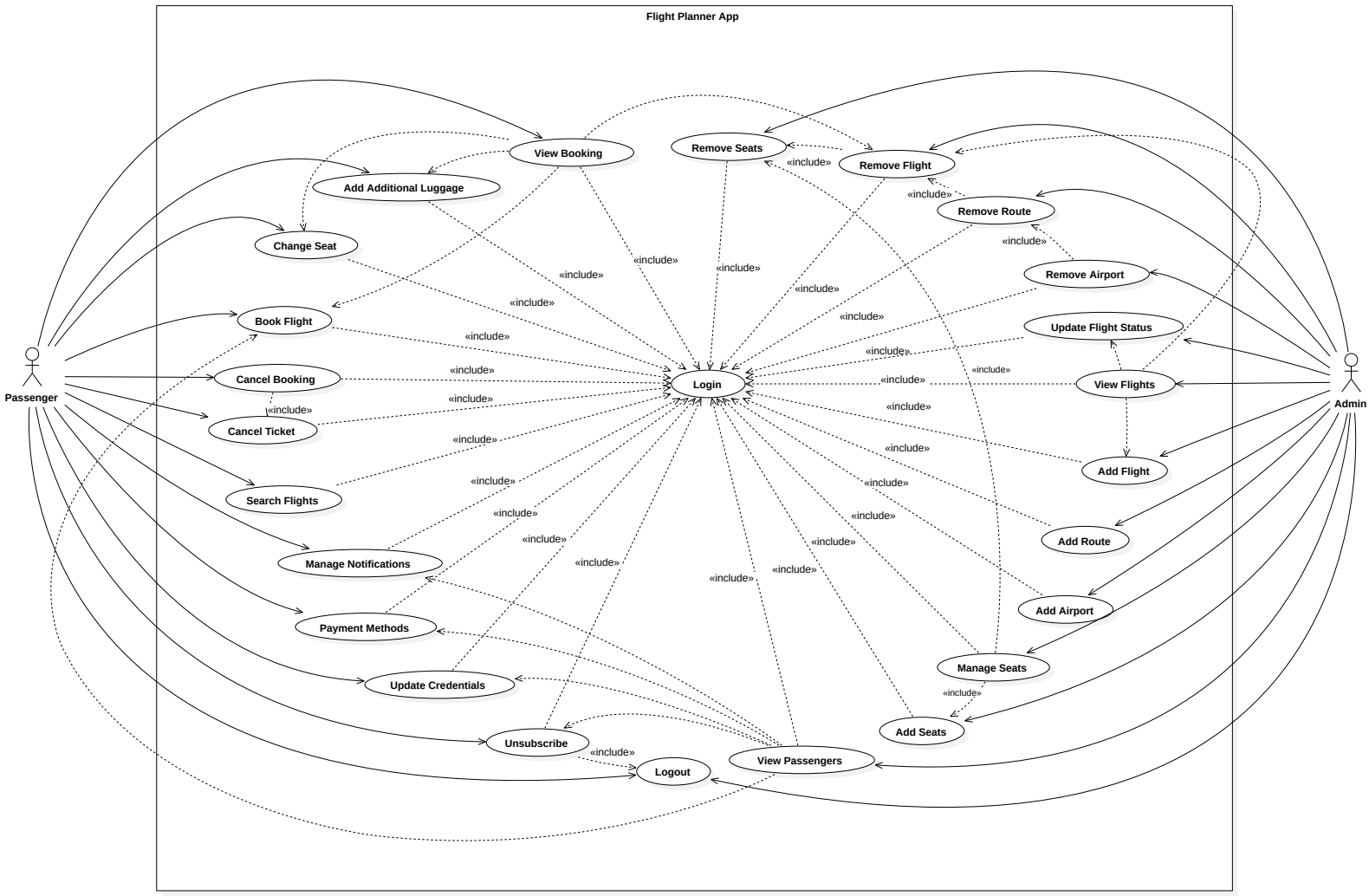

Figura 1: Use Case Diagram

2.2 Class Diagram

Un diagramma delle classi è uno strumento fondamentale nell'ambito della modellazione software, utilizzato per rappresentare la struttura statica di un sistema. Mostra le classi del sistema, le loro attribuzioni, metodi e le relazioni tra di esse, fornendo così una visione d'insieme delle componenti che compongono l'applicazione. Questo tipo di diagramma è particolarmente utile nella fase di progettazione, in quanto facilita la comprensione del sistema e delle interazioni tra le sue componenti.

2.2.1 Analisi del Class Diagram

Nel progetto, tutte le classi sono organizzate in tre pacchetti distinti: uno per la logica di business, che include anche il Controller e la View, uno per i modelli di dominio e uno per i manager che gestiscono le operazioni sui file CSV. Tuttavia, il diagramma delle classi (Figura 2) non fornisce un dettaglio specifico sui singoli pacchetti e le classi al loro interno. Invece, il diagramma semplifica la visualizzazione attraverso una struttura di pacchetti colorata: il pacchetto *businessLogic* è rappresentato in rosa, quello *domainModel* in giallo e quello *manager_CSV* in celeste. Il design adotta il pattern MVC, con il Controller rappresentato dalla classe che gestisce l'interazione tra la logica di business e la View. Inoltre, è stato implementato anche il pattern Observer per gestire i cambiamenti, garantendo che i passeggeri ricevano notifiche sugli aggiornamenti dei voli.

La classe *FlightPlanner* funge da coordinatore centrale tra i vari manager, gestendo la comunicazione tra essi, fatta eccezione per AuthManager, che rimane autonomo. Nonostante FlightPlanner sembri seguire il pattern Facade, il suo ruolo differisce da questo modello. Un pattern Facade crea un'interfaccia semplificata per un insieme complesso di interazioni, celando i dettagli di implementazione. In questo progetto, invece, FlightPlanner non nasconde la complessità interna dei manager, ma espone alcuni metodi che accedono direttamente alle loro funzionalità, inoltre alcuni manager possono anche collaborare tra di loro senza passare attraverso FlightPlanner. Per questi motivi, FlightPlanner non può essere considerato un'implementazione del pattern Facade.

Dal diagramma delle classi si può notare che l'interfaccia *NotificationChannel* possiede il metodo *sendNotification*, implementato dalle classi concrete *SmsNotification* ed *EmailNotification*. Queste classi vengono utilizzate all'interno della classe *NotificationPreferences*, che gestisce un set di tipi di notifica (definiti come enumerazione) e una lista di canali di notifica. Le preferenze di notifica sono gestite come attributi del passeggero, il cui costruttore accetta il set dei tipi di notifica e la lista dei canali.
Questa configurazione mostra una somiglianza con il pattern Strategy, nel quale il comportamento di un oggetto può essere modulato dinamicamente tramite l'uso di diverse "strategie". Tuttavia, non si tratta di una vera implementazione di Strategy in senso tradizionale. In questo progetto, le preferenze di notifica possono essere modificate durante l'esecuzione, e in assenza di specifiche configurazioni da parte dell'utente, vengono impostate delle preferenze predefinite, come le notifiche per *Gate Change* e *Cancellation* con il canale email. Sebbene queste modifiche dinamiche delle preferenze possano sembrare un uso del pattern Strategy, il concetto di "strategia" qui non è inteso come una scelta fissa, ma piuttosto come un'opzione configurabile che può essere personalizzata dall'utente, rendendo il comportamento più vicino a un'architettura personalizzata che a una rigorosa applicazione di Strategy.

Dunque il diagramma delle classi offre una visione chiara della struttura del sistema e delle relazioni tra le sue componenti. L'adozione di pattern come MVC e Observer nel progetto garantisce una struttura modulare e ben organizzata. Sebbene alcuni aspetti del progetto richiamino concetti di pattern come Facade e Strategy, la gestione delle funzionalità e delle preferenze di notifica è stata progettata in modo specifico per rispondere alle esigenze dell'applicazione, dando vita a un'architettura unica e personalizzata.

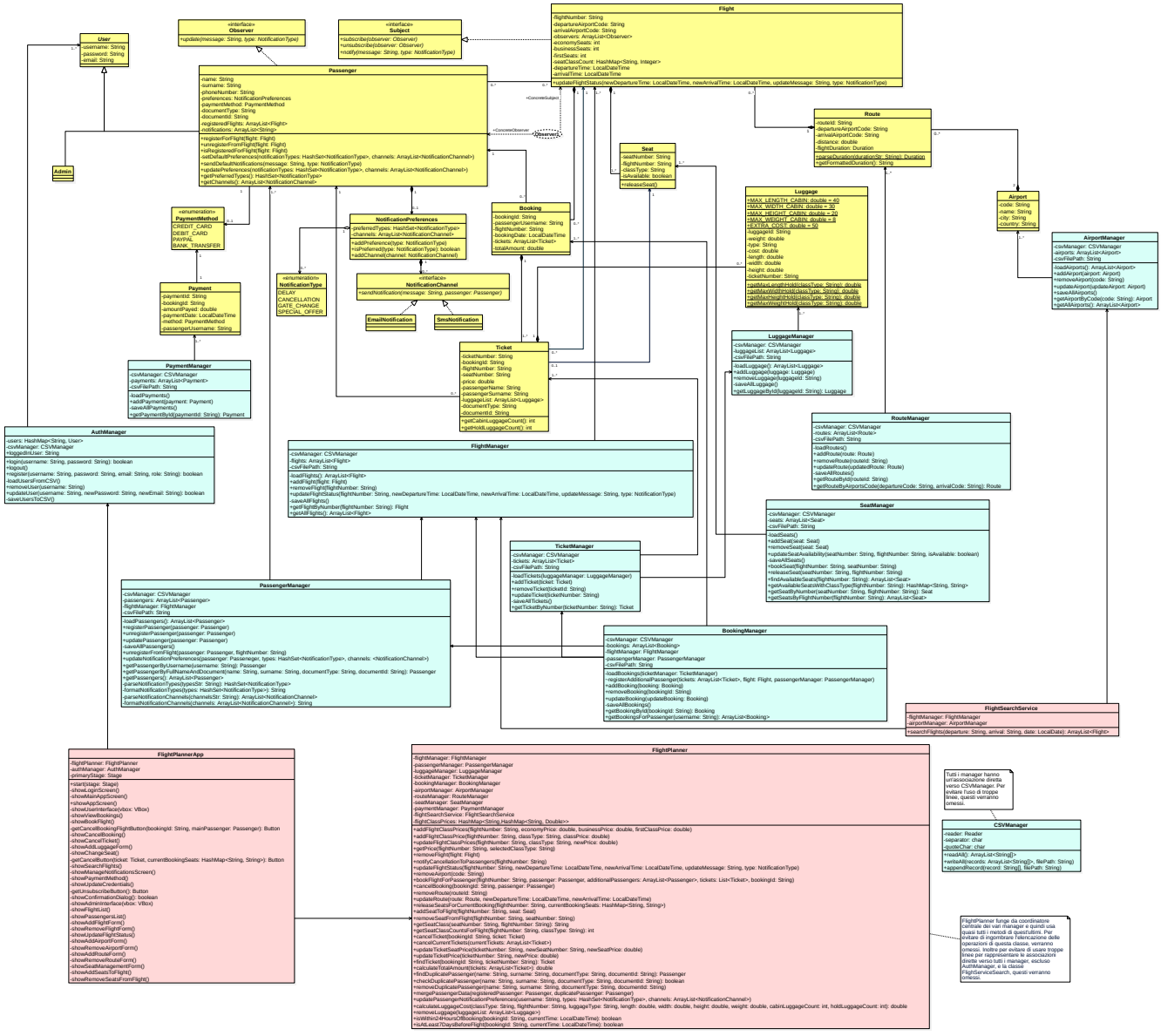

Figura 2: Class Diagram

2.3 Template

Di seguito sono presentate alcune rappresentazioni grafiche dell'interfaccia utente, insieme ai corrispettivi template che definiscono le funzionalità fondamentali del software.

UC	**Login**
Level	User Goal
Actor	Passeggero/Admin
Basic Course	1. L'utente accede al sistema. 2. IL sistema mostra la finestra iniziale dell'applicazione (Figura 3). 3. L'utente inserisce i propri credenziali per effettuare il login: username e password. 4. L'utente clicca su *Login*. 5. Il sistema mostra l'interfaccia Admin o Passenger a seconda dei credenziali di accesso inseriti (Figura 4 e 10).
Alternative Course	1. Il sistema non trova nessuna corrispondenza con i credenziali inseriti perché l'utente non si è ancora autenticato presso il sistema. (a) Il sistema mostra un messaggio di errore. (b) L'utente inserisce un username univoco al sistema. (c) L'utente inserisce una password che rispetta i vincoli imposti. (d) L'utente inserisce un'email che rispetta il formato standard. (e) L'utente clicca su *Register*. 2. Il sistema non trova nessuna corrispondenza con i credenziali inseriti perché l'utente ha inserito scorrettamente l'username o la password. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'utente la possibilità di inserire nuovamente i credenziali.

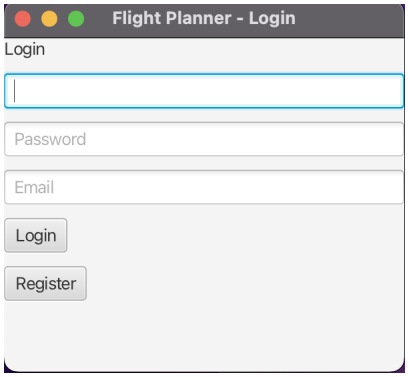

Figura 3: Finestra iniziale

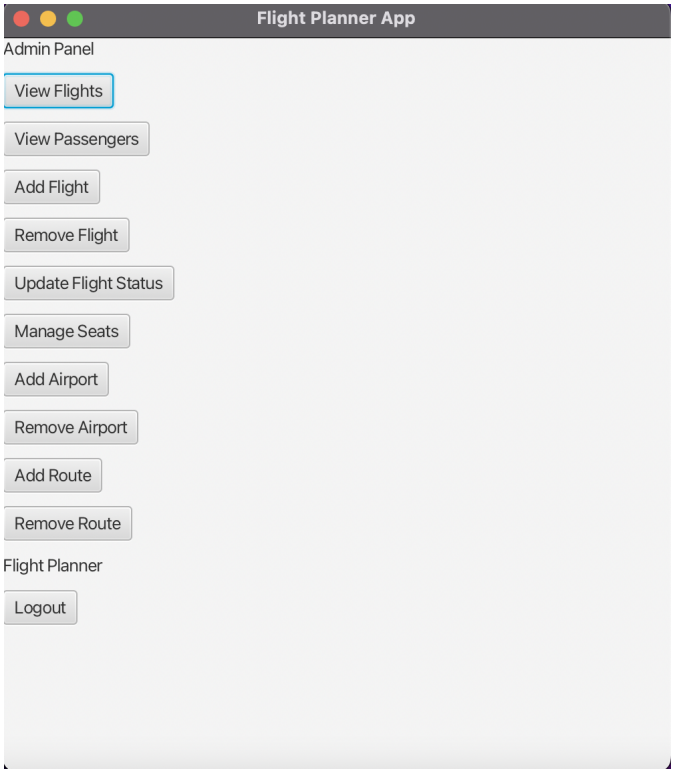

Figura 4: Finestra che mostra l'interfaccia di Admin con le sue funzionalità riservate

UC	**View Flights**
Level	Admin Goal
Actor	Admin
Basic Course	1. Dopo aver fatto accesso al sistema, l'amministratore clicca su *View Flights*. 2. Il sistema mostra l'elenco con i dettagli dei voli disponibili (Figura 5). 3. Dopo aver preso visione dell'elenco, l'amministratore clicca su *Back*. 4. Il sistema mostra il menù principale dell'applicazione (Figura 4).

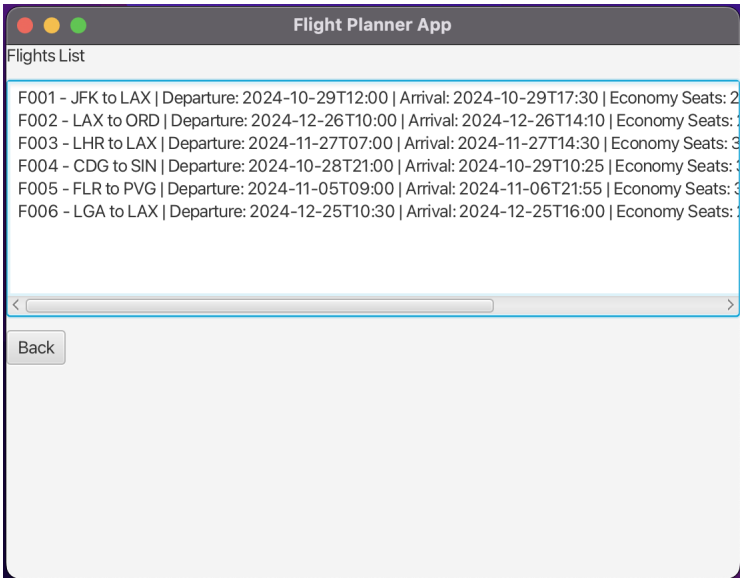

Figura 5: Finestra che mostra la lista di voli disponibili nell'applicazione

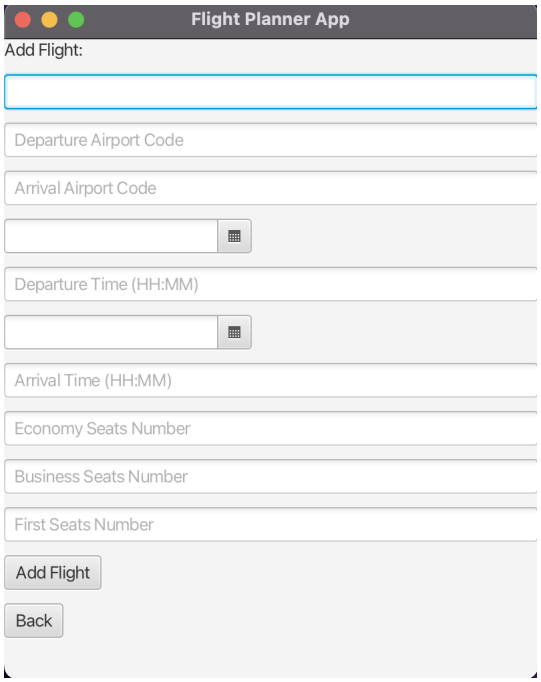

Figura 6: Finestra che mostra *Add Flight* e i campi necessari da compeletare per aggiungere un nuovo volo

UC	**Add Flight**
Level	Admin Goal
Actor	Admin
Basic Course	1. Dopo aver fatto accesso al sistema, l'amministratore clicca su *Add Flight*. 2. Il sistema mostra una finestra per l'inserimento dei dati del nuovo volo da aggiungere (Figura 6). 3. L'amministratore inserisce il codice univoco del volo. 4. L'amministratore inserisce codici univoci validi per gli aeroporti di partenza e di arrivo, che corrispondono a una rotta esistente. 5. L'amministratore seleziona la data di partenza. 6. L'amministratore inserisce l'ora di partenza secondo il formato richiesto. 7. L'amministratore ripete gli stessi passaggi per la selezione della data e l'inserimento dell'ora di arrivo. 8. L'amministratore specifica la quantità di posti disponibili per ciascuna delle tre classi (Economy, Business, First) su un nuovo volo. 9. L'amministratore clicca su *Add Flight*. 10. Dopo aver effettuato l'aggiunta del nuovo volo, l'amministratore clicca su *Back*. 11. Il sistema mostra il menù principale dell'applicazione (Figura 4).

UC	**Add Flight**
Level	Admin Goal
Actor	Admin
Alternative Course	1. L'amministratore omette di compilare i campi obbligatori e clicca su *Add Flight*. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di completare i campi mancanti. 2. L'amministratore inserisce un codice aeroportuale di partenza o di arrivo inesistente. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di inserire nuovamente i codici aeroportuali. 3. L'amministratore inserisce dei codici aeroportuali validi, ma non esiste una rotta corrispondente. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di inserire nuovamente i codici aeroportuali. 4. L'amministratore non seleziona la data di partenza o di arrivo. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di selezionare le date che sono ancora mancanti. 5. L'amministratore non inserisce l'ora di partenza o l'ora di arrivo nel formato richiesto. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di inserire nuovamente le ore di partenza o di arrivo. 6. L'amministratore non inserisce la quantità disponibile dei posti in formato numerica. (a) Il sistema mostra un messaggio di errore. (b) Viene fornito all'amministratore di inserire nuovamente la quantità dei posti disponibili. 7. L'amministratore inserisce un numero totale di posti inferiore a 1. (a) Il sistema mostra un messaggio di errore. (b) Viene fornito all'amministratore di inserire nuovamente la quantità dei posti disponibili.

UC	Update Flight Status
Level	Admin Goal
Actor	Admin
Basic Course	1. Dopo aver fatto accesso al sistema, l'amministratore clicca su *Update Flight Status*. 2. IL sistema mostra la finestra per inserire le informazioni necessari per l'aggiornamento di un volo (Figura 7). 3. L'amministratore inserisce il codice univoco del volo da aggiornare. 4. L'amministratore seleziona la nuova data di partenza del volo. 5. L'amministratore inserisce il nuovo orario di partenza secondo il formato richiesto. 6. L'amministratore ripete gli stessi passaggi per la selezione della nuova data di arrivo e l'inserimento del nuovo orario di destinazione. 7. L'amministratore inserisce un messaggio di notifica per i passeggeri iscritti a questo volo. 8. L'amministratore seleziona la tipologia di notifica. 9. L'amministratore clicca su *Update Flight Status*. 10. Dopo aver effettuato l'aggiornamento, l'amministratore clicca su *Back*. 11. Il sistema mostra il menù principale dell'applicazione (Figura 4).
Alternative Course	1. L'amministratore omette di compilare i campi obbligatori e clicca su *Update Flight Status*. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di ricompilare i campi che sono ancora mancanti. 2. L'amministratore non ha inserito un codice di volo esistente. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di inserire nuovamente il codice univoco di un volo. 3. L'amministratore non inserisce l'ora di partenza o l'ora di arrivo nel formato richiesto. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di inserire nuovamente le ore di partenza o di arrivo. 4. L'amministratore non inserisce il messaggio di notifica da inviare ai passeggeri iscritti al volo. (a) Il sistema mostra un messaggio di errore. (b) Viene fornita all'amministratore la possibilità di scrivere un messaggio.

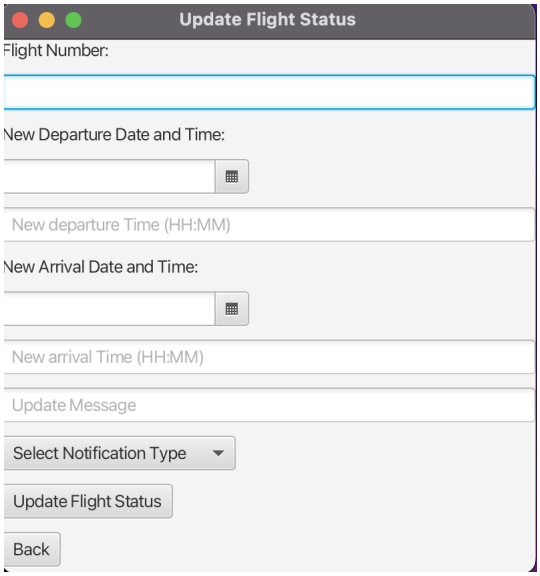

Figura 7: Finestra per *Update Flight Status* che mostra i campi necessari da completare per aggiornare un volo esistente

UC	**Remove Flight**
Level	Admin Goal
Actor	Admin
Basic Course	1. Dopo aver fatto accesso al sistema, l'amministratore clicca su *Remove Flight*. 2. IL sistema mostra la finestra per la rimozione di un volo esistente dall'applicazione (Figura 8). 3. L'amministratore inserisce il codice univoco di un volo esistente. 4. L'amministratore clicca su *Remove Flight*. 5. Dopo aver completato la rimozione del volo, l'amministratore clicca su *Back*. 6. Il sistema mostra il menù principale dell'applicazione (Figura 4).
Alternative Course	1. L'amministratore non inserisce il codice del volo e clicca su *Remove Flight*. (a) Il sistema mostra un messaggio di lavoro. (b) Viene offerto all'amministratore la possibilità di reinserire un codice del volo. 1. L'amministratore inserisce erroneamente il codice del volo. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto all'amministratore la possibilità di reinserire un codice del volo.

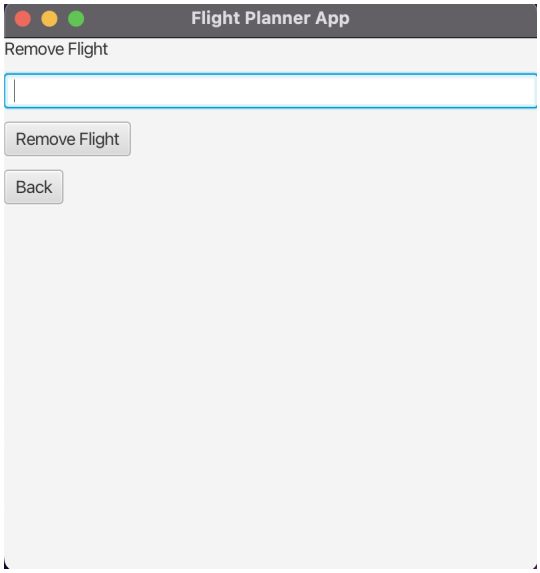

Figura 8: Finestra per *Remove Flight*

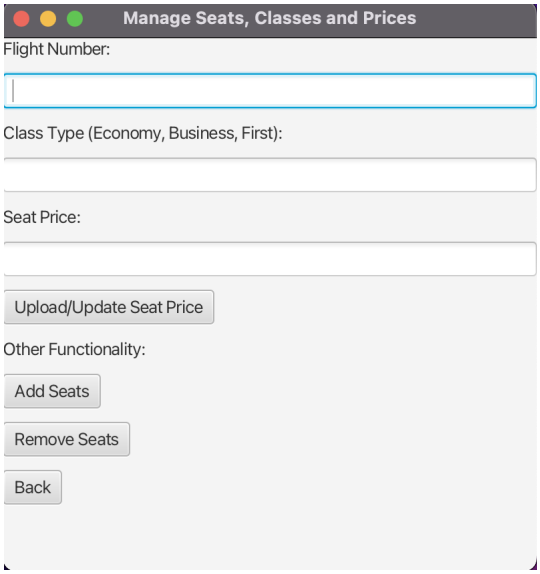

Figura 9: Finestra che mostra le funzionalità contenute in *Manage Seats*

UC	Manage Seats
Level	Admin Goal
Actor	Admin
Basic Course	1. Dopo aver fatto accesso al sistema, l'amministratore clicca su *Manage Seats*. 2. Il sistema mostra la finestra per la gestione dei posti su un volo, consentendo all'amministratore di aggiornare o impostare il prezzo per ciascuna classe di volo, nonché di aggiungere o rimuovere posti tramite i relativi pulsanti (Figura 9). 3. L'amministratore inserisce il codice univoco di un volo esistente. 4. L'amministratore inserisce la classe di volo secondo la scrittura suggerita. 5. L'amministratore inserisce un prezzo valido. 6. L'amministratore clicca su *Upload/Update Seat Price*. 7. Dopo aver aggiornato o caricato il prezzo per una classe specifica di un volo esistente, l'amministratore clicca su *Back*. 8. Il sistema mostra il menù principale dell'applicazione (Figura 4).
Alternative Course	1. L'amministratore non ha inserito un codice di volo o una classe e clicca su *Upload/Update Seat Price*. (a) Il sistema mostra un messaggio di lavoro. (b) Viene offerto all'amministratore la possibilità di reinserire i campi mancanti. 2. L'amministratore inserisce erroneamente il codice del volo. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto all'amministratore la possibilità di reinserire un codice del volo. 3. L'amministratore non inserisce il prezzo in formato numerico. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto la possibilità all'amministratore di reinserire un prezzo. 4. L'amministratore inserisce la tipologia di classe non nella forma suggerita. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto la possibilità all'amministratore di reinserire la classe.

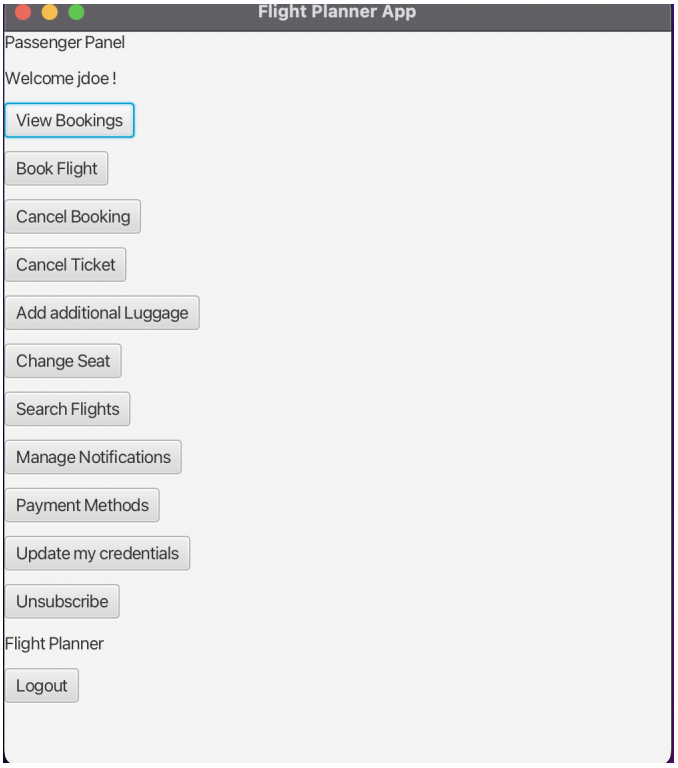

Figura 10: Finestra che mostra l'interfaccia di Passenger con le sue funzionalità riservate

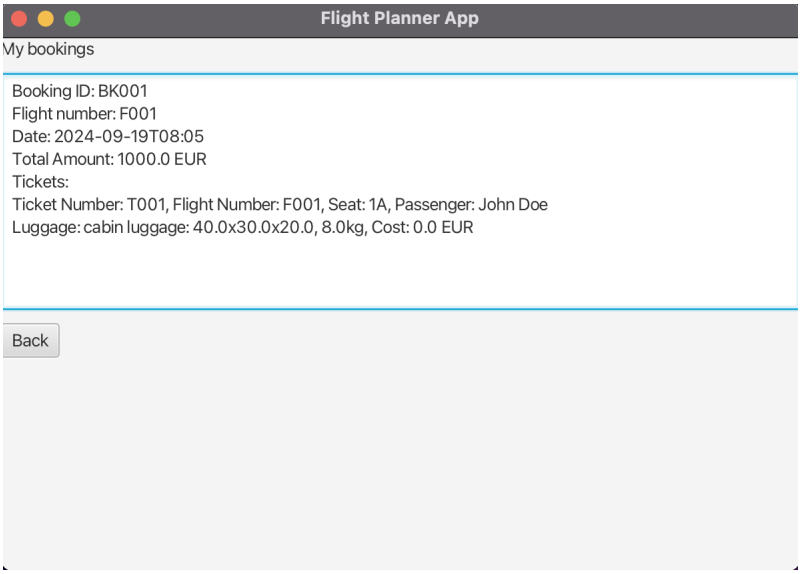

Figura 11: Finestra che mostra la lista di prenotazioni effettuate da un passeggero

UC	View Bookings
Level	Passenger Goal
Actor	Passenger
Basic Course	1. Dopo aver fatto accesso al sistema, il passeggero clicca su *View Bookings*. 2. Il sistema mostra l'elenco con le prenotazioni già effettuate (Figura 11). 3. Dopo aver preso visione dell'elenco, il passeggero clicca su *Back*. 4. Il sistema mostra il menù principale dell'applicazione (Figura 10).

UC	Manage Notifications
Level	Passenger Goal
Actor	Passenger
Basic Course	1. Dopo aver fatto accesso al sistema, il passeggero clicca su *Manage Notifications*. 2. Il sistema mostra la finestra per la gestione delle impostazioni delle preferenze di notifica (Figura 12). 3. Il passeggero seleziona la tipologia di notifica (anche più di una). 4. Il passeggero seleziona il canale di notifica (anche tutte e due). 5. Se viene selezionato *SMS*, il passeggero inserisce anche un valido recapito telefonico. 6. Il passeggero clicca su *Save*. 7. Dopo aver impostato le preferenze di notifica, il passeggero clicca su *Back*. 8. Il sistema mostra il menù principale dell'applicazione (Figura 10).
Alternative Course	1. Il passeggero seleziona *SMS* ma tenta di salvare l'impostazione senza fornire un recapito telefonico. (a) Il sistema mostra un messaggio di avvertimento. (b) Viene offerto la possibilità al passeggero di inserire un recapito telefonico. 2. Il passeggero fornisce un numero di telefono privo di prefisso internazionale e con una lunghezza non conforme agli standard, e clicca su *Save*. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto al passeggero la possibilità di inserire nuovamente il numero di telefono.

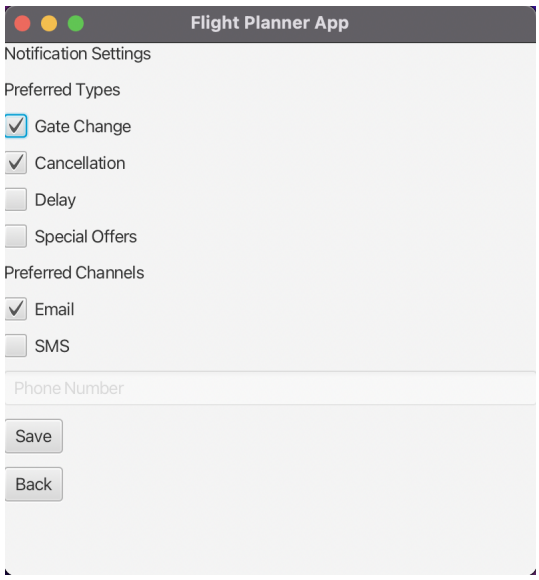

Figura 12: Finestra per la gestione delle impostazioni delle preferenze di notifica

UC	**Payment Methods**
Level	Passenger Goal
Actor	Passenger
Basic Course	1. Dopo aver fatto accesso al sistema, il passeggero clicca su *Payment Methods*. 2. Il sistema mostra la finestra per la gestione dell'impostazione del metodo di pagamento (Figura 13). 3. Il passeggero sceglie il metodo di pagamento da utilizzare per completare il pagamento al momento della prenotazione del volo. 4. Il passeggero clicca su *Confirm*. 5. Dopo aver impostato il metodo di pagamento, il passeggero clicca su *Back*. 6. Il sistema mostra il menù principale dell'applicazione (Figura 10).
Alternative Course	1. Il passeggero non seleziona nessun metodo di pagamento e clicca su *Confirm*. (a) Il sistema mostra un messaggio di avvertimento. (b) Viene offerto la possibilità al passeggero di selezionare nuovamente il metodo di pagamento.

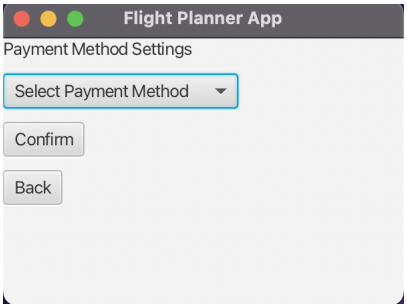

Figura 13: Finestra per la gestione dell'impostazione del metodo di pagamento

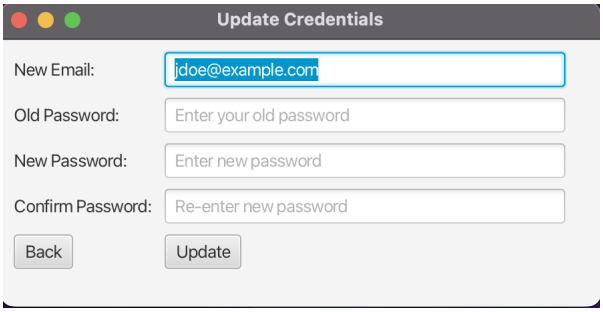

Figura 14: Finestra per l'aggiornamento delle credenziali del passeggero

UC	Update Credentials
Level	Passenger Goal
Actor	Passenger
Basic Course	1. Dopo aver fatto accesso al sistema, il passeggero clicca su *Update my credentials*. 2. Il sistema mostra la finestra per l'aggiornamento delle credenziali del passeggero (Figura 14). 3. Il passeggero inserisce la nuova email. 4. Il passeggero inserisce la vecchia password. 5. Il passggero inserisce la nuova password. 6. Il passeggero inserisce la password di conferma. 7. Il passeggero clicca su *Update*. 8. Dopo aver aggiornato le credenziali, il passeggero clicca su *Back*. 9. Il sistema mostra il menù principale dell'applicazione (Figura 10).
Alternative Course	1. Il passeggero lascia tutti i campi vuoti e clicca su *Update*. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto la possibilità al passeggero di completare nuovamente i campi mancanti. 2. Il passeggero inserisce un'email non in formato standard. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto la possibilità al passeggero di inserire nuovamente l'email. 3. Il passeggero tenta di aggiornare l'email senza fornire la vecchia password. (a) Il sistema mostra un messaggio di errore. (b) Viene fornito al passeggero la possibilità di inserire la vecchia password. 4. Il passeggero non inserisce la nuova password in modo coerente con la password di conferma. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto al passeggero la possibilità di reinserire la nuova password e la password di conferma. 5. Il passeggero tenta di aggiornare la password con una lunghezza inferiore di 6 caratteri. (a) Il sistema mostra un messaggio di errore. (b) Viene offerto al passeggero la possibilità di reinserire la nuova password e la password di conferma.

3 Implementazione

3.1 Design Pattern utilizzati

3.1.1 Observer

Il pattern **Observer** è un design pattern comportamentale che consente a un oggetto (*Subject*) di notificare automaticamente altri oggetti (*Observers*) quando il suo stato cambia. Questa relazione di dipendenza permette agli Observers di rimanere aggiornati senza dover interrogare costantemente il Subject. Il pattern si basa su tre componenti principali: il Subject, gli Observers e la relazione tra di essi, che include metodi per l'iscrizione, la disiscrizione e la notifica.
Nel contesto del presente progetto, il *ConcreteSubject* è rappresentato dalla classe **Flight**, la quale gestisce lo stato del volo e comunica eventuali variazioni. Il *ConcreteObserver* è rappresentato dalla classe **Passenger**, la quale è interessata a ricevere aggiornamenti sui voli a cui è iscritta. Tuttavia, a differenza di una tradizionale implementazione del pattern Observer, in questo caso i passeggeri non ricevono tutte le notifiche relative ai cambiamenti di stato del volo, ma solo quelle che corrispondono alle loro preferenze di notifica. Questo approccio consente di ridurre il rumore informativo e migliorare l'esperienza dell'utente, in quanto i passeggeri ricevono esclusivamente comunicazioni pertinenti. Nel caso in cui un passeggero non specifichi alcuna preferenza, viene automaticamente assegnata una configurazione di default, che include notifiche per i cambi di gate e per le cancellazioni tramite email.
L'implementazione del pattern Observer in questo progetto utilizza la modalità *push*. In questa modalità, il Subject invia attivamente i dettagli della notifica agli Observer, evitando loro di dover richiedere le informazioni. Questo permette al Subject (Flight) di inviare ai passeggeri solo i dettagli specifici della modifica che corrispondono alle loro preferenze. Questo modello è ideale per garantire che ogni Observer riceva informazioni pertinenti e dettagliate in modo efficiente, migliorando l'esperienza utente.
Per implementare il pattern Observer, sono state create due interfacce: *Subject* e *Observer*. Nella classe **Flight**, il metodo *notify()* viene utilizzato per avvisare i passeggeri delle modifiche, mentre nella classe **Passenger** il metodo *update()* è impiegato per ricevere le notifiche. Inoltre, i passeggeri hanno la possibilità di iscriversi o disiscriversi dal volo attraverso i metodi *subscribe()* e *unsubscribe()*.

```
1  public interface Subject {
2      void notify(String message, NotificationType type);
3      void subscribe(Observer observer);
4      void unsubscribe(Observer observer);
5  }
```

Listing 1: Interfaccia Subject

```
1  public interface Observer {
2      void update(String message, NotificationType type);
3  }
```

Listing 2: Interfaccia Observer

Le notifiche vengono inviate attraverso il canale di comunicazione preferito del passeggero, che può scegliere tra SMS ed email. Questo sistema di notifiche offre un ulteriore livello di personalizzazione, consentendo ai passeggeri di ricevere aggiornamenti nel modo più conveniente per loro, facilitando così una comunicazione mirata e pertinente.

```
1  public void updateFlightStatus(LocalDateTime newDepartureTime, LocalDateTime newArrivalTime,
       String updateMessage, NotificationType type) {
2          setDepartureTime(newDepartureTime);
3          setArrivalTime(newArrivalTime);
4          notify(updateMessage, type);
5      }
6
7      @Override
8      public void notify(String message, NotificationType type) {
9          for (Observer observer : observers) {
```

```
            observer.update(message, type);
        }
    }
```

Listing 3: Implementazione di notify() e il suo uso nella classe Flight

```
@Override
    public void update(String message, NotificationType type) {
        if (preferences == null || preferences.getPreferredTypes().isEmpty()) {
            sendDefaultNotifications(message, type);
        } else {
            if (preferences.isPreferred(type)) {
                for (NotificationChannel channel : preferences.getChannels()) {
                    channel.sendNotification(message, this);
                    notifications.add(message);
                }
            }
        }
    }
```

Listing 4: Implementazione di update() nella classe Passenger

3.1.2 Model-View-Controller (MVC)

Il pattern **MVC** è un'architettura utilizzata per separare le responsabilità tra gestione dei dati, presentazione e controllo del flusso applicativo. In questo schema, il Model rappresenta la logica applicativa e i dati, la View è l'interfaccia utente che presenta le informazioni, e il Controller gestisce l'interazione tra Model e View, controllando il flusso delle operazioni. Il pattern MVC consente una migliore modularità e manutenibilità, separando le logiche principali e riducendo la complessità del codice.
Nel progetto, i ruoli di Model, View e Controller sono assegnati come segue:

- **Model**: rappresenta i dati e le logiche principali, incluse le classi gestite dai manager (ad esempio, Passenger, Flight, Booking, Airport, ecc). Ogni classe incapsula i propri attributi e metodi per la gestione delle informazioni, mentre i manager corrispondenti coordinano le operazioni su di essi.

- **View** e **Controller**: in questo progetto, la View e il Controller sono integrati nella classe *FlightPlannerApp*. La scelta di unificare View e Controller in un'unica classe risponde all'esigenza di una gestione più centralizzata, che semplifica l'accesso alle funzioni applicative e riduce il numero complessivo di classi, mantenendo la gestione dell'interfaccia utente (costruita in JavaFX) e del flusso di controllo in un unico componente. Sebbene questa struttura comprometta leggermente la modularità, risulta efficace per il progetto, poiché facilita l'interazione con l'utente e riduce la complessità dell'applicazione.

Da evidenziare il fatto che FlightPlannerApp riesce a svolgere le funzionalità necessarie grazie alle classi *AuthManager* e *FlightPlanner*. La classe FlightPlanner funge da coordinatore centrale tra i vari manager (come PassengerManager, BookingManager, FlightManager, TicketManager, ecc.), gestendo funzioni principali come la ricerca e la gestione dei voli, le prenotazioni e gli aggiornamenti dei passeggeri. Tuttavia, FlightPlanner non si configura come un vero Facade, poiché anziché nascondere le interfacce dei manager, fornisce un accesso diretto e centralizzato alle loro funzioni. Questo approccio evita una complessità superflua e preserva la chiarezza organizzativa.
Quasi tutti i manager gestiscono le operazioni CRUD per la propria classe e permette anche la ricerca di dati in base a caratteristiche specifiche, consentendo una gestione dei dati efficace e semplificando il controllo delle informazioni.
Questa divisione permette di evitare ridondanze, delegando ai manager la responsabilità per la manipolazione e la conservazione dei dati, mentre FlightPlanner si occupa di orchestrare le interazioni tra le varie funzionalità.

3.2 Gestione dei CSV

Nel progetto, i dati sono gestiti tramite file CSV, utilizzati sia per le operazioni di lettura che di scrittura. Questa scelta si rivela vantaggiosa per la semplicità e la leggerezza che i file CSV offrono, permettendo di lavorare con dati strutturati senza ricorrere a un database complesso.
I file di lettura sono collocati nella directory *resources*, mentre i file di scrittura sono organizzati in una directory dedicata chiamata *csv*. Tale separazione permette di distinguere chiaramente i dati di origine (informazioni di base) dai dati modificati e aggiornati durante l'esecuzione del programma, migliorando sia la struttura che la sicurezza dei dati.
A supporto di questo approccio, è stato introdotto un **CSVManager** che centralizza le operazioni di input e output sui file CSV. Questa classe offre metodi come *readAll()* (per leggere tutti i record), *appendRecord()* (per aggiungere nuovi record) e *writeAll()* (per sovrascrivere l'intero contenuto del file). Ogni manager integra un'istanza di CSVManager per gestire le proprie operazioni *CRUD* (Create, Read, Update, Delete), come la creazione di nuovi record, il caricamento dei dati dai file CSV, l'aggiornamento e la cancellazione dei dati, oltre alla ricerca di specifiche informazioni nei file CSV.
Per quanto riguarda **PaymentManager** e **LuggageManager**, questi manager si distinguono in quanto non includono le operazioni di delete o update: un pagamento, una volta effettuato, non può essere né cancellato né modificato, e similmente vale per la gestione dei bagagli, che, una volta registrati per un volo, non sono soggetti a modifica. Questa logica riflette la natura immutabile di tali dati, assicurando che le informazioni una volta registrate rimangano consistenti.

3.2.1 Vantaggio dell'uso di CSVManager e dei file CSV

L'adozione di CSVManager centralizza la logica di accesso ai dati, semplificando il codice e rendendolo più manutenibile. Questo approccio consente ai manager di concentrarsi sulle loro specifiche responsabilità, delegando la gestione dell'interazione con i file a una classe dedicata. Inoltre, l'utilizzo di file CSV facilita le operazioni di testing e debugging, grazie alla possibilità di visualizzare direttamente i dati in formato leggibile.
Di conseguenza, il progetto beneficia di un sistema efficiente per l'accesso e la gestione dei dati, con una struttura semplice e facilmente scalabile, adatta a future estensioni.

```
1  public class CSVManager {
2      private final Reader reader;
3      private final char separator;
4      private final char quoteChar;
5
6      public CSVManager(Reader reader) {
7          this(reader, ',', '\"');
8      }
9
10     public CSVManager(Reader reader, char separator, char quoteChar) {
11         this.reader = reader;
12         this.separator = separator;
13         this.quoteChar = quoteChar;
14     }
15
16     public List<String[]> readAll() throws IOException {
17         List<String[]> records;
18         try (CSVReader csvReader = new CSVReader(reader)) {
19             records = csvReader.readAll();
20         } catch (CsvException e) {
21             throw new RuntimeException(e);
22         }
23         return records;
24     }
25
26     public void writeAll(List<String[]> records, String filePath) throws IOException {
27         try (FileWriter fileWriter = new FileWriter(filePath, false);
```

```
            CSVWriter csvWriter = new CSVWriter(fileWriter, separator, quoteChar, CSVWriter
                .DEFAULT_ESCAPE_CHARACTER, CSVWriter.DEFAULT_LINE_END)) {
            csvWriter.writeAll(records);
        }
    }

    public void appendRecord(String[] record, String filePath) throws IOException {
        try (FileWriter fileWriter = new FileWriter(filePath, true);
            CSVWriter csvWriter = new CSVWriter(fileWriter, separator, quoteChar, CSVWriter
                .DEFAULT_ESCAPE_CHARACTER, CSVWriter.DEFAULT_LINE_END)) {
            csvWriter.writeNext(record);
        }
    }
}
```

Listing 5: Classe CSVManager

```
public class AirportManager {
    private static final Log log = LogFactory.getLog(AirportManager.class);
    private final CSVManager csvManager;
    private final List<Airport> airports;
    private final String csvFilePath = "csv/airports.csv";

    public AirportManager() throws IOException {
        InputStream inputStream = getClass().getClassLoader().getResourceAsStream(
            csvFilePath);
        if (inputStream == null) {
            throw new FileNotFoundException("File not found in resources: " + csvFilePath);
        }
        this.csvManager = new CSVManager(new InputStreamReader(inputStream));
        this.airports = loadAirports();
    }
    private List<Airport> loadAirports() throws IOException {
        List<String[]> records = csvManager.readAll();
        List<Airport> airports = new ArrayList<>();
        for (int i = 1; i < records.size(); i++) {
            String[] record = records.get(i);
            if (record.length >= 4) {
                Airport airport = new Airport(
                        record[0],
                        record[1],
                        record[2],
                        record[3]
                );
                airports.add(airport);
            } else {
                System.err.println("Invalid row format: " + Arrays.toString(record));
            }
        }
        return airports;
    }
    public void addAirport(Airport airport) throws IOException {
        for (Airport existingAirport : airports) {
            if (existingAirport.getCode().equalsIgnoreCase(airport.getCode())) {
                throw new IllegalArgumentException("Airport " + airport.getCode() + "
                    already exists.");
            }
        }
        airports.add(airport);
        System.out.println("Added Airport: " + airport);
        String[] record = {airport.getCode(), airport.getName(), airport.getCity(), airport.
            getCountry()};
        try {
            csvManager.appendRecord(record, csvFilePath);
        } catch (IOException e) {
```

```
            log.error("An error occurred while writing an airport to the CSV file", e);
            throw e;
        }
    }
    public void removeAirport(String code) throws IOException {
        Airport toRemove = null;
        for (Airport airport : airports) {
            if (airport.getCode().equalsIgnoreCase(code)) {
                toRemove = airport;
                break;
            }
        }
        if (toRemove != null) {
            airports.remove(toRemove);
            saveAllAirports();
        }
    }
    public void updateAirport(Airport updatedAirport) throws IOException {
        for (Airport airport : airports) {
            if (airport.getCode().equalsIgnoreCase(updatedAirport.getCode())) {
                if (!airport.getName().equalsIgnoreCase(updatedAirport.getName())) {
                    airport.setName(updatedAirport.getName());
                }
                break;
            }
        }
        saveAllAirports();
    }
    private void saveAllAirports() throws IOException {
        List<String[]> records = new ArrayList<>();
        records.add(new String[]{"code", "name", "city", "country"});
        for (Airport airport : airports) {
            records.add(new String[]{
                    airport.getCode(),
                    airport.getName(),
                    airport.getCity(),
                    airport.getCountry()
            });
        }
        try {
            csvManager.writeAll(records, csvFilePath);
        } catch (IOException e) {
            log.error("An error occurred while saving airports to the CSV file: " + e.
                getMessage());
            throw e;
        }
    }
```

Listing 6: Operazioni CRUD di AirportManager

3.3 Classi e Metodi Principali

Di seguito verranno analizzati le classi e i metodi che meritano un approfondimento riguardo alle scelte implementative.

3.3.1 User e AuthManager

Nel progetto, la classe astratta **User** è progettata per rappresentare tutti gli utenti del sistema ed è caratterizzata da tre attributi principali: *username, email e password*. L'attributo username è dichiarato final, e quindi non può essere modificato una volta creato un utente. Questa caratteristica si rivela particolarmente utile per la gestione delle autenticazioni, in quanto garantisce l'unicità e l'immutabilità dell'identificatore di ciascun utente all'interno del sistema.

L'**AuthManager**, la classe responsabile dell'autenticazione e della registrazione degli utenti, si basa proprio su questa immutabilità. I dati degli utenti vengono memorizzati in un file CSV denominato *users.csv*, mentre in memoria è utilizzata una mappa che associa ogni username al rispettivo oggetto User. La chiave username consente un accesso rapido e sicuro agli utenti registrati e aiuta a prevenire duplicati o ambiguità, poiché l'username rimane fisso per tutta la durata dell'applicazione. Durante la fase di login, l'AuthManager utilizza l'username per verificare se un utente è già registrato, facilitando la gestione delle credenziali nell'interfaccia utente e migliorando l'efficienza dei controlli di autenticazione.
Nel sistema è presente *un solo amministratore*, con credenziali predefinite e non modificabili, garantendo così un ruolo di accesso unico e centralizzato per la gestione del sistema. Non è possibile creare o registrare altri amministratori, poiché tutti gli altri utenti registrati sono automaticamente classificati come Passenger. Questo modello rende più sicura la gestione degli accessi e definisce chiaramente i ruoli, limitando i privilegi di amministrazione a un unico utente.

```
1   public boolean login(String username, String password) {
2       User user = users.get(username);
3       if (user != null && user.getPassword().equals(password)) {
4           loggedInUser = username;
5           return true;
6       }
7       return false;
8   }
9
10  public boolean register(String username, String password, String email, String role)
        throws IOException {
11      if (role.equals("Admin")) {
12          System.out.println("Cannot register another admin.");
13          return false;
14      }
15
16      if (!users.containsKey(username)) {
17          User newUser = new Passenger(username, "", "", email, "", password, null, null,
                null, "", "");
18          users.put(username, newUser);
19
20          String[] record = {
21                  username,
22                  password,
23                  email,
24                  role
25          };
26          try {
27              csvManager.appendRecord(record, "csv/users.csv");
28          } catch (IOException e) {
29              log.error("An error occurred while writing an user to the CSV file", e);
30              throw e;
31          }
32          saveUsersToCSV();
33          return true;
34      } else {
35          System.out.println("User already exists: " + username);
36          return false; // Registrazione fallita
37      }
38  }
```

Listing 7: Metodo login() e register() della classe AuthManager

Per quanto riguarda gli utenti Passenger, essi hanno la possibilità di cancellare il proprio account. Una volta che un passeggero decide di disiscriversi, i suoi dati vengono rimossi dal file CSV users.csv, assicurando che le informazioni personali non siano conservate inutilmente e mantenendo la coerenza e la conformità dei dati.

3.3.2 BookingManager e TicketManager

Come descritto in precedenza, il progetto è strutturato in diverse classi principali, ciascuna delle quali è gestita da un manager specifico. Tra questi, il BookingManager e il TicketManager ricoprono un ruolo particolarmente importante, poiché devono coordinare dati provenienti da più fonti per garantire la coerenza e la correttezza delle informazioni.
Il **BookingManager** è responsabile della gestione delle prenotazioni e dei relativi dati. Quando carica le informazioni dal file CSV *booking.csv*, richiede che i dati della classe Ticket siano già stati caricati da *tickets.csv*, poiché uno dei suoi attributi è una lista di biglietti associati a ciascuna prenotazione. Inoltre, il BookingManager dipende anche dai dati presenti in *passengers.csv* e *flights.csv*, poiché ogni prenotazione è strettamente correlata ai passeggeri e ai voli corrispondenti. In particolare, il BookingManager deve assicurarsi che ogni passeggero sia associato correttamente ai voli su cui ha una prenotazione, garantendo una logica operativa coerente e coordinata. Questo processo di caricamento è complesso perché richiede una precisa sequenza di operazioni: i biglietti, i passeggeri e i voli devono essere disponibili in memoria prima che le prenotazioni possano essere completamente caricate e gestite.

```
1   private List<Booking> loadBookings(TicketManager ticketManager) throws IOException {
2       List<String[]> records = csvManager.readAll();
3       List<Booking> bookings = new ArrayList<>();
4       for (int i = 1; i < records.size(); i++) {
5           String[] record = records.get(i);
6           String bookingId = record[0];
7           String passengerUsername = record[1];
8           String flightNumber = record[2];
9           LocalDateTime bookingDate = LocalDateTime.parse(record[3]);
10          String[] ticketNumbers = record[4].split("\\|");
11          List<Ticket> tickets = new ArrayList<>();
12          double totalAmount = Double.parseDouble(record[5]);
13
14          for (String ticketNumber : ticketNumbers) {
15              Ticket ticket = ticketManager.getTicketByNumber(ticketNumber);
16              if (ticket != null) {
17                  tickets.add(ticket);
18              }
19          }
20
21          Passenger passenger = passengerManager.getPassengerByUsername(passengerUsername)
                ;
22          Flight flight = flightManager.getFlightByNumber(flightNumber);
23          passenger.registerForFlight(flight);
24          registerAdditionalPassenger(tickets, flight, passengerManager);
25
26          Booking booking = new Booking(
27                  bookingId,
28                  passengerUsername,
29                  flightNumber,
30                  bookingDate,
31                  tickets,
32                  totalAmount
33          );
34          bookings.add(booking);
35      }
36      return bookings;
37  }
```

Listing 8: Metodo loadBookings() della classe BookingManager

Similmente, il **TicketManager** gestisce i dati relativi ai biglietti, ciascuno dei quali può includere una lista di bagagli. Per garantire il caricamento completo e accurato dei biglietti dal file *tickets.csv*, è necessario che i dati relativi ai bagagli siano già stati caricati dal file *luggage.csv*. Solo così è possibile associare correttamente ogni bagaglio al biglietto a cui appartiene, mantenendo la coerenza e l'integrità dei dati.

```java
    private void loadTickets(LuggageManager luggageManager) throws IOException {
        List<String[]> records = csvManager.readAll();
        for (int i = 1; i < records.size(); i++) {
            String[] record = records.get(i);

            String[] luggageIds = record[7].split("\\|");
            List<Luggage> luggageList = new ArrayList<>();
            for (String luggageId : luggageIds) {
                Luggage luggage = luggageManager.getLuggageById(luggageId);
                if (luggage != null) {
                    luggageList.add(luggage);
                }
            }

            Ticket ticket = new Ticket(
                    record[0],
                    record[1],
                    record[2],
                    record[3],
                    Double.parseDouble(record[4]),
                    record[5],
                    record[6],
                    luggageList
            );
            tickets.add(ticket);
            ticket.setDocumentType(record[8]);
            ticket.setDocumentId(record[9]);
        }
    }
```

Listing 9: Metodo loadTickets() della classe TicketManager

Questo approccio richiede una complessa coordinazione tra i manager e l'ordine di caricamento dei dati, ma comporta diversi vantaggi. In primo luogo, la struttura dei dati rimane organizzata e accessibile: ogni manager può operare sui dati di sua competenza, mentre la coerenza generale è garantita dall'interazione tra i manager stessi. Inoltre, questo schema facilita le operazioni di aggiornamento, ricerca e verifica, poiché ogni manager ha piena visibilità sulle relazioni tra gli oggetti di propria gestione e i dati ad essi collegati. La complessità derivante dalla gestione coordinata dei dati è compensata da un sistema altamente modulare, scalabile e manutenibile, che permette l'estensione del progetto senza compromettere la stabilità delle funzionalità esistenti.

3.3.3 FlightPlanner

Nel progetto, ciascun manager esegue operazioni CRUD e ricerche specifiche sui dati di cui è responsabile, contribuendo a una gestione modulare e organizzata delle varie entità del sistema. Tuttavia, il **FlightPlanner** funge da coordinatore centrale, orchestrando l'interazione tra i diversi manager, ad eccezione di *AuthManager*, per garantire che le funzionalità dei vari componenti possano convergere per raggiungere obiettivi complessi e integrati. Pur svolgendo un ruolo di coordinamento avanzato, FlightPlanner non rispecchia un classico pattern Facade poiché non si limita a fornire un'interfaccia semplice per l'accesso ai manager. Invece, gestisce direttamente la logica più sofisticata e completa del progetto, permettendo ai vari manager di interagire tra loro secondo flussi operativi specifici e funzionali, che vengono successivamente messi a disposizione della classe **FlightPlannerApp**.
Tra i metodi più importanti offerti da FlightPlanner, **bookFlightForPassenger()** permette a un utente registrato di prenotare un volo sia per sé stesso sia per passeggeri aggiuntivi, non necessariamente registrati nell'app. Questi passeggeri sono identificati tramite dati personali quali nome, cognome, tipo di documento e numero di identificazione. I passeggeri non registrati sono comunque salvati nel file *passengers.csv* con username impostato come "undefined" e con preferenze di notifica predefinite; l'indirizzo email associato è quello dell'utente che effettua la prenotazione. In questo modo, il sistema può mantenere traccia dei passeggeri anche senza che siano utenti effettivi della piattaforma.

```
    public void bookFlightForPassenger(String flightNumber, Passenger passenger, List<
        Passenger> additionalPassengers, List<Ticket> tickets, String bookingId) throws
        IOException {
        Passenger existingPassenger = passengerManager.getPassengerByUsername(passenger.
            getUsername());
        if (existingPassenger == null) {
            passengerManager.registerPassenger(passenger);
            System.out.println("Passenger registered: " + passenger.getUsername() + " " +
                passenger.getName() + " " + passenger.getSurname());
        }

        Flight flight = flightManager.getFlightByNumber(flightNumber);
        if (flight != null) {
            passenger.registerForFlight(flight);
            double totalAmount = calculateTotalAmount(tickets);

            Booking booking = new Booking(bookingId, passenger.getUsername(), flightNumber,
                LocalDateTime.now(), tickets, totalAmount);
            bookingManager.addBooking(booking);
            System.out.println("Flight booked for passenger: " + passenger.getUsername() + "
                 " + passenger.getName() + " "
                    + passenger.getSurname() +
                    ". Total amount to pay is " + totalAmount + " EUR.");

            for (Passenger p : additionalPassengers) {
                p.registerForFlight(flight);
            }

        } else {
            throw new IllegalArgumentException("Flight "+ flightNumber + " not found.");
        }
    }
```

Listing 10: Metodo bookFlightForPassenger() della classe FlightPlanner

Il metodo **cancelBooking()** gestisce la cancellazione di una prenotazione e dei componenti, come biglietti e bagagli associati, e il rilascio dei posti prenotati. In aggiunta, tutti i passeggeri coinvolti sono automaticamente disiscritti dal volo, garantendo un aggiornamento completo e coerente dei dati. Un metodo simile **cancelTicket()** permette la cancellazione di un singolo biglietto, sia per un utente registrato sia per un passeggero aggiunto. Nel caso in cui un passeggero abbia un solo biglietto e questo venga cancellato, anche la prenotazione associata verrà eliminata.

```
public void cancelBooking(String bookingId, Passenger passenger) throws IOException {
        Booking booking = bookingManager.getBookingById(bookingId);
        if (booking != null) {
            passengerManager.unregisterFromFlight(passenger, booking.getFlightNumber());
            List<Ticket> tickets = booking.getTickets();
            Ticket firstTicket = tickets.getFirst();
            seatManager.releaseSeat(firstTicket.getSeatNumber(), firstTicket.getFlightNumber
                ());
            removeLuggage(firstTicket.getLuggageList());
            ticketManager.removeTicket(firstTicket.getTicketNumber());

            tickets.stream().skip(1).forEach(ticket -> {
                String additionalPassengerName = ticket.getPassengerName();
                String additionalPassengerSurname = ticket.getPassengerSurname();
                String documentType = ticket.getDocumentType();
                String documentId = ticket.getDocumentId();
                Passenger additionalPassenger = getPassengerByFullNameAndDocument(
                    additionalPassengerName, additionalPassengerSurname,
                        documentType, documentId);
                passengerManager.unregisterFromFlight(additionalPassenger, booking.
                    getFlightNumber());
                try {
```

```
                    seatManager.releaseSeat(ticket.getSeatNumber(), ticket.getFlightNumber()
                        );
                    removeLuggage(ticket.getLuggageList());
                    ticketManager.removeTicket(ticket.getTicketNumber());
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            });

            bookingManager.removeBooking(bookingId);
            System.out.println("Booking canceled: " + bookingId);

        } else {
            throw new IllegalArgumentException("Booking " + bookingId +" not found.");
        }
    }
```

Listing 11: Metodo cancelBooking() della classe FlightPlanner

```
public void cancelTicket(String bookingId, Ticket ticket) throws IOException {
        Booking booking = findBooking(bookingId);
        if (booking == null) {
            throw new IllegalArgumentException("Booking ID " + bookingId + " not found.");
        }

        List<Ticket> tickets = booking.getTickets();
        String passengerName = ticket.getPassengerName();
        String passengerSurname = ticket.getPassengerSurname();
        String documentType = ticket.getDocumentType();
        String documentId = ticket.getDocumentId();
        double ticketPrice = ticket.getPrice();
        double luggagePrice = 0;

        for (Luggage luggage : ticket.getLuggageList()) {
            luggagePrice += luggage.getCost();
        }

        if (tickets.size() > 1) {
            boolean removed = tickets.remove(ticket);

            if (!removed) {
                throw new IllegalArgumentException("Ticket " + ticket.getTicketNumber() + "
                    not found in booking " + bookingId);
            } else {
                booking.setTotalAmount(booking.getTotalAmount() - ticketPrice);
                System.out.println("Refund of 40% (" + (ticketPrice - luggagePrice) * 0.40 +
                     " EUR, excluding luggage cost) for ticket " + ticket.getTicketNumber()
                    +
                        " from booking " + bookingId + " has been processed.");
                seatManager.releaseSeat(ticket.getSeatNumber(), ticket.getFlightNumber());
                removeLuggage(ticket.getLuggageList());
                ticketManager.removeTicket(ticket.getTicketNumber());
                Flight flight = flightManager.getFlightByNumber(booking.getFlightNumber());
                if (flight != null) {
                    Passenger passenger = passengerManager.getPassengerByFullNameAndDocument
                        (passengerName, passengerSurname, documentType, documentId);
                    passengerManager.unregisterFromFlight(passenger, flight.getFlightNumber
                        ());
                } else {
                    throw new IllegalArgumentException("Flight " + ticket.getFlightNumber()
                        + " not found .");
                }

                System.out.println("Ticket number " + ticket.getTicketNumber() + " canceled
                    from " + bookingId + ". ");
```

```
                bookingManager.updateBooking(booking);
            }
        } else {
            Passenger passenger = passengerManager.getPassengerByUsername(booking.
                getPassengerUsername());
            cancelBooking(bookingId, passenger);
        }
    }
```

Listing 12: Metodo cancelTicket() della classe FlightPlanner

Il metodo **removeAirport()** assicura la consistenza dei dati nel momento in cui un aeroporto viene rimosso dal sistema: la rotta associata viene cancellata, così come i voli ad essa collegati. La cancellazione di un volo comporta automaticamente la rimozione dei posti prenotati e la cancellazione delle relative prenotazioni, e i passeggeri coinvolti ricevono una notifica di cancellazione con rimborso completo, se hanno selezionato la notifica di tipo *cancellation* tra le loro preferenze.

```
public void removeAirport(String code) throws IOException {
        Airport airport = airportManager.getAirportByCode(code);
        if (airport != null) {
            List<Route> routes = routeManager.getAllRoutes();
            for (int i = routes.size() - 1; i >= 0; i--) { // Si itera in modo inverso per
                evitare ConcurrentModificationException
                Route route = routes.get(i);
                if (route.getDepartureAirportCode().equalsIgnoreCase(code) || route.
                    getArrivalAirportCode().equalsIgnoreCase(code)) {
                    removeRoute(route.getRouteId());
                }
            }
            airportManager.removeAirport(code);
            System.out.println("Airport " + code + " removed.");
        } else {
            throw new IllegalArgumentException("Airport " + code + " not found.");
        }
    }

    public void removeRoute(String routeId) throws IOException {
        Route route = routeManager.getRouteById(routeId);
        List<Flight> flights = getAllFlights();
        if (route != null) {
            for (int i = flights.size() - 1; i >= 0; i--) {
                Flight flight = flights.get(i);
                if (flight.getDepartureAirportCode().equalsIgnoreCase(route.
                    getDepartureAirportCode()) &&
                        flight.getArrivalAirportCode().equalsIgnoreCase(route.
                            getArrivalAirportCode())) {
                    removeFlight(flight.getFlightNumber());
                }
            }
            routeManager.removeRoute(routeId);
            System.out.println("Route " + routeId + " removed.");
        } else {
            throw new IllegalArgumentException("Route " + routeId + " not found.");
        }
    }

    public void removeFlight(String flightNumber) throws IOException {
        Flight flight = findFlight(flightNumber);
        List<Seat> seats = seatManager.getAllSeats();
        List<Booking> bookings = bookingManager.getAllBookings();
        if (flight != null) {
            for (int i = seats.size() - 1; i >= 0; i--) {
                Seat seat = seats.get(i);
                if (seat.getFlightNumber().equalsIgnoreCase(flightNumber)) {
                    removeSeatFromFlight(flightNumber, seat.getSeatNumber());
```

```
                }
            }

            notifyCancellationToPassengers(flightNumber);

            for (int i = bookings.size() - 1; i >= 0; i--) {
                Booking booking = bookings.get(i);
                if (booking.getFlightNumber().equalsIgnoreCase(flightNumber)) {
                    cancelBooking(booking.getBookingId(), getPassenger(booking.
                        getPassengerUsername()));
                }
            }
            flightManager.removeFlight(flightNumber);
            System.out.println("Flight " + flightNumber + " has been removed.");
        } else {
            throw new IllegalArgumentException("Flight " + flightNumber + " not found.");
        }
    }

    public void notifyCancellationToPassengers(String flightNumber) {
        Flight flight = findFlight(flightNumber);
        List<Observer> observers = flight.getObservers();
        for (Observer observer : observers) {
            if (observer instanceof Passenger passenger) {
                for (Booking booking : getBookingsForPassenger(passenger.getUsername())) {
                    double refundAmount = booking.getTotalAmount();
                    passenger.update("Sorry for the inconvenience, the flight " +
                        flightNumber + " is cancelled. Your refund of " +
                            refundAmount + " EUR including luggage cost for the booking " +
                                booking.getBookingId() + " is processed automatically.",
                                NotificationType.CANCELLATION);
                }
            }
        }
    }

    public void removeSeatFromFlight(String flightNumber, String seatNumber) throws
        IOException {
        Flight flight = flightManager.getFlightByNumber(flightNumber);
        Seat seat = seatManager.getSeatByNumber(seatNumber, flightNumber);
        if (flight != null) {
            if (seat != null) {
                seatManager.removeSeats(seat);
            } else {
                throw new IllegalArgumentException("Seat " + seatNumber + " not found. ");
            }
        } else {
            throw new IllegalArgumentException("Flight " + flightNumber + " not found.");
        }
    }
```

Listing 13: Metodi remove() associati della classe FlightPlanner

Per garantire un'accurata gestione dei dati dei passeggeri non registrati che potrebbero in futuro decidere di iscriversi all'applicazione, in FlightPlanner vengono implementati i metodi **checkDuplicatePassenger(), findDuplicatePassenger(), removeDuplicatePassenger() e mergePassengerData()**. Questi metodi sono progettati per riconoscere e gestire le informazioni dei passeggeri duplicati. Al momento della prima prenotazione di volo, se un passeggero aggiuntivo decide di iscriversi al sistema, i dati del nuovo utente vengono uniti a quelli esistenti nel file CSV, e il sistema aggiorna automaticamente i voli a cui il passeggero risultava iscritto tramite prenotazioni effettuate da altri. Pur non potendo visualizzare tali prenotazioni nella propria interfaccia utente, il passeggero è comunque riconosciuto come registrato su quei voli, e il sistema impedisce di duplicare le prenotazioni già esistenti.

```
    public Passenger findDuplicatePassenger(String name, String surname, String documentType
        , String documentId) {
        List<Passenger> passengers = getPassengers();
        for (Passenger passenger : passengers) {
            if (passenger.getName().equals(name) &&
                    passenger.getSurname().equals(surname) &&
                    passenger.getUsername().equals("not defined") && passenger.
                        getDocumentType().equals(documentType)
                    && passenger.getDocumentId().equals(documentId)) {
                return passenger;  // Trovato duplicato con informazioni incomplete
            }
        }
        return null;  // Nessun duplicato trovato
    }

    public boolean checkDuplicatePassenger(String name, String surname, String documentType,
         String documentId) {
        Passenger duplicatePassenger = findDuplicatePassenger(name, surname, documentType,
            documentId);
        int count = 0;
        for (Passenger passenger : getPassengers()) {
            if (duplicatePassenger != null && passenger.getName().equals(duplicatePassenger.
                getName()) && passenger.getSurname().equals(duplicatePassenger.getSurname())
                    && passenger.getDocumentType().equals(documentType)
                    && passenger.getDocumentId().equals(documentId)) {
                count++;
            }
        }
        return count > 1;
    }

    public void removeDuplicatePassenger(String username, String name, String surname,
        String documentType, String documentId) throws IOException {
        Passenger currentPassenger = getPassenger(username);
        Passenger duplicatePassenger = findDuplicatePassenger(name, surname, documentType,
            documentId);
        if (duplicatePassenger != null && duplicatePassenger.getUsername().equals("not
            defined") && duplicatePassenger.getName().equals(name)
                && duplicatePassenger.getSurname().equals(surname) && duplicatePassenger.
                    getDocumentType().equals(documentType)
                && duplicatePassenger.getDocumentId().equals(documentId)) {
            mergePassengerData(currentPassenger, duplicatePassenger);
        }
    }

    public void mergePassengerData(Passenger registeredPassenger, Passenger
        duplicatePassenger) throws IOException {
        List<Flight> registeredFlights = duplicatePassenger.getRegisteredFlights();
        for (Flight flight : registeredFlights) {
            registeredPassenger.getRegisteredFlights().add(flight);
        }
        removePassenger(duplicatePassenger);
        updatePassenger(registeredPassenger);
        System.out.println("Data merged of duplicate passenger: " + registeredPassenger.
            getUsername());
    }
```

Listing 14: Metodi per gestire passeggeri duplicati della classe FlightPlanner

3.3.4 FlightPlannerApp

Nel progetto, la classe FlightPlannerApp rappresenta il punto di ingresso dell'applicazione e ha il compito di avviare e gestire l'interfaccia utente. FlightPlannerApp estende la classe **Application**, struttura tipica nelle

applicazioni JavaFX, e fornisce una configurazione completa per inizializzare e rendere disponibile l'intero sistema all'utente.
Il metodo principale **start()** costituisce il cuore della classe, poiché attiva l'interfaccia grafica e mostra la finestra principale dell'applicazione. Questo metodo, richiamato automaticamente dal runtime JavaFX all'avvio, si occupa di caricare il layout della GUI e di inizializzare i componenti necessari per l'interazione con l'utente.
All'interno di start(), l'istanza di *flightPlanner* viene creata per coordinare centralmente i vari manager, gestendo operazioni come prenotazioni e cancellazioni di voli, la gestione di dati relativi a passeggeri, prenotazioni, bagagli,ecc. Viene inoltre istanziato *authManager*, che si occupa dell'autenticazione e registrazione degli utenti. Il parametro *Stage stage*, rappresentante la finestra principale dell'applicazione, viene assegnato a *primaryStage*, mantenendo un riferimento alla finestra su cui si caricano le varie schermate dell'app.
All'avvio, il metodo **showLoginScreen()** imposta la schermata di login, garantendo che gli utenti possano autenticarsi o registrarsi prima di accedere alle funzionalità principali.

```
1   @Override
2   public void start(Stage stage) throws IOException {
3       flightPlanner = new FlightPlanner();
4       authManager = new AuthManager();
5       primaryStage = stage;
6
7       showLoginScreen();
8   }
9
10  private void showLoginScreen() {
11      VBox vbox = new VBox(10);
12      Label titleLabel = new Label("Login");
13
14      TextField usernameField = new TextField();
15      usernameField.setPromptText("Username");
16      usernameField.setId("usernameField");
17
18      PasswordField passwordField = new PasswordField();
19      passwordField.setPromptText("Password");
20      passwordField.setId("passwordField");
21
22      TextField emailField = new TextField();
23      emailField.setPromptText("Email");
24      emailField.setId("emailField");
25
26      Button loginButton = new Button("Login");
27      loginButton.setId("loginButton");
28      Button registerButton = new Button("Register");
29      registerButton.setId("registerButton");
30
31      loginButton.setOnAction(_ -> {
32          String username = usernameField.getText();
33          String password = passwordField.getText();
34          if (authManager.login(username, password)) {
35              showMainAppScreen();
36          } else {
37              showAlert(Alert.AlertType.ERROR, "Login Failed", "Invalid username or
                    password.");
38          }
39      });
40
41      registerButton.setOnAction(_ -> {
42          String username = usernameField.getText();
43          String password = passwordField.getText();
44          String email = emailField.getText();
45
46          if (username.isEmpty() || email.isEmpty() || password.isEmpty()) {
```

```
                showAlert(Alert.AlertType.ERROR, "Input Error", "Please fill in all the
                    fields!");
                return;
            }

            if (username.equals("not defined")) {
                showAlert(Alert.AlertType.ERROR, "Error", "You cannot use this username!
                    Please change to another one.");
                return;
            }

            if (!email.matches("^[a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,}$")) {
                showAlert(Alert.AlertType.ERROR, "Invalid Email", "Please provide a valid
                    email address.");
                return;
            }

            if (password.length() < 6) {
                showAlert(Alert.AlertType.ERROR, "Invalid Password", "Password must be at
                    least 6 characters long.");
                return;
            }

            try {
                if (authManager.register(username, password, email, "Passenger")) {
                    Passenger passenger = (Passenger) authManager.getUsers().get(username);
                    try {
                        flightPlanner.registerPassenger(passenger);
                    } catch (IOException ex) {
                        throw new RuntimeException(ex);
                    }
                    showAlert(Alert.AlertType.INFORMATION, "Registration Successful", "You
                        can now log in.");
                } else {
                    showAlert(Alert.AlertType.ERROR, "Registration Failed", "Username
                        already exists.");
                }
            } catch (IOException ex) {
                showAlert(Alert.AlertType.ERROR, "Error", "An error occurred during
                    registration.");
            }
        });

        vbox.getChildren().addAll(titleLabel, usernameField, passwordField, emailField,
            loginButton, registerButton);
        Scene scene = new Scene(vbox, 300, 250);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Flight Planner - Login");
        primaryStage.show();
    }
```

Listing 15: Metodo start() e showLoginScreen() della classe FlightPlannerApp

FlightPlannerApp funge da livello di presentazione e interazione con l'utente finale, collegandosi ai vari manager tramite il coordinatore flightPlanner. In quanto interfaccia principale, FlightPlannerApp assicura che tutte le operazioni eseguite dagli utenti siano interpretate correttamente e trasmesse ai manager responsabili, mantenendo la logica di business separata da quella di presentazione.
Grazie alla distinzione tra logica di interfaccia, gestita da FlightPlannerApp, e logica di gestione, curata dai manager e da flightPlanner, l'architettura del progetto resta modulare e coerente. FlightPlannerApp contribuisce così a migliorare l'esperienza utente, offrendo un'interfaccia intuitiva e completa supportata da un'architettura robusta e scalabile.

4 Unit Test

4.1 Introduzione ai test

In questa sezione sono analizzati i test effettuati per garantire il corretto funzionamento delle componenti dell'applicazione. I test sono stati implementati utilizzando la libreria JUnit 5 per i test unitari e TestFX per i test di interfaccia grafica, simulando l'interazione dell'utente con l'applicazione.

Ogni classe del progetto è stata sottoposta a test accurati, con particolare attenzione ai metodi più rilevanti, per verificare il corretto comportamento delle funzionalità principali e la robustezza delle logiche implementate. I test sono organizzati in tre pacchetti che riflettono la struttura a pacchetti del progetto:

- **businessLogicTets**;
- **domainModelTest**;
- **manger_CSVTest**.

Questa organizzazione facilita la gestione dei test, garantendo che ogni area del progetto sia testata in modo coerente con la sua struttura.
Tutte le classi manager condividono un metodo **setUp()** che prepara i dati necessari caricando i file CSV pertinenti per la classe di test in esame. Questa configurazione standardizzata consente di creare un ambiente di test realistico, coerente e ripetibile, riducendo al minimo il rischio di errori derivanti da configurazioni di dati non corrette. Inoltre, sono stati inclusi test per le funzionalità CRUD dei manager, che verificano la corretta gestione dei dati e delle associazioni tra oggetti, garantendo che i manager possano gestire e ricercare oggetti in base a caratteristiche specifiche.
Per i test d'integrazione, che verificano il funzionamento dell'intera applicazione e l'interazione tra i vari componenti, è stata utilizzata la libreria TestFX. Questo strumento permette di simulare le interazioni dell'utente con la GUI, testando il corretto comportamento di FlightPlannerApp e verificando che l'interfaccia grafica si integri adeguatamente con la logica di business del progetto. In questo modo, è possibile analizzare non solo l'output dei singoli metodi, ma anche l'efficacia complessiva dell'interfaccia utente e la coerenza con la logica applicativa.
In alcuni casi la copertura dei test include sia scenari con input validi sia situazioni di errore, al fine di assicurare un funzionamento stabile anche in presenza di dati inconsistenti o operazioni non previste. Vengono inoltre testati vari casi limite e comportamenti di fallimento per garantire che il sistema sia robusto e in grado di gestire condizioni impreviste senza compromettere la stabilità complessiva.
Questa strategia di testing consente di individuare prontamente eventuali regressioni o malfunzionamenti, aumentando l'affidabilità del software e la sicurezza nell'adozione del codice sviluppato. La separazione dei test in pacchetti corrispondenti alla struttura del progetto facilita il mantenimento e l'aggiornamento dei test man mano che il progetto evolve, rendendo più semplice l'aggiunta di nuovi test o la modifica di quelli esistenti in base all'evoluzione delle funzionalità.

Premesse Data la vasta quantità dei test effettuati, di seguito verranno riportati solamente quelli più rilevanti, con particolare attenzione ai pacchetti **businessLogicTest** e **manager_CSVTest**. Non è stato incluso un esempio di **domainModelTest**, in quanto tali test riguardano operazioni più semplici e dirette, come la creazione degli oggetti, l'accesso ai metodi getter e la verifica del corretto funzionamento del metodo *toString()*. Sebbene questi test siano fondamentali per garantire il corretto funzionamento dei modelli di dominio, sono considerati di base e non richiedono esempi dettagliati nella documentazione. L'assenza di tali esempi non compromette la comprensione del progetto, poiché si tratta di test standardizzati che non introducono complessità aggiuntiva rispetto a quelli che verranno descritti nelle sottosezioni successive.

4.2 AuthManagerTest

I test in questa sezione, che fanno parte di *manager_CSVTest*, hanno lo scopo di verificare le operazioni principali della classe *AuthManager*.

4.2.1 Test Login

Questi due test hanno l'obiettivo di verificare la procedura di login per un utente registrato che tenta di accedere all'applicazione e per un utente non registrato.

```
    @Test
    @DisplayName("Test that checks when a user is registered, s/he can login successfully")
    public void testLoginSuccessful() {
        boolean loginResult = authManager.login("admin", "admin123");
        assertTrue(loginResult);
        assertEquals("admin", authManager.getLoggedInUser());
    }

    @Test
    @DisplayName("Test that checks when an unknown user tries to login, s/he will fail")
    public void testLoginFailure() {
        boolean loginResult = authManager.login("nonexistentUser", "password");
        assertFalse(loginResult);
        assertNull(authManager.getLoggedInUser());
    }
```

Listing 16: Implementazione di testLoginSuccessful() e testLoginFailure()

4.2.2 Test riguardo alla registrazione

Questi due test hanno l'obiettivo di verificare la procedura di registrazione di un nuovo utente nell'applicazione e di assicurarsi che venga impedita la registrazione duplicata di un utente già esistente.

```
    @Test
    @DisplayName("Test that checks the registration of a new user")
    public void testRegisterNewUser() throws IOException {
        boolean registerResult = authManager.register("newUser", "newPass123", "
            newuser@example.com", "Passenger");
        assertTrue(registerResult);
        Map<String, User> users = authManager.getUsers();
        User user = users.get("newUser");
        assertEquals("newPass123", user.getPassword());
        assertEquals("newuser@example.com", user.getEmail());
    }

    @Test
    @DisplayName("Test that checks when a user is already registered, s/he can't register
        again")
    public void testRegisterDuplicateUser() throws IOException {
        authManager.register("duplicateUser", "password", "duplicate@example.com", "
            Passenger");
        boolean registerResult = authManager.register("duplicateUser", "password", "
            duplicate@example.com", "Passenger");
        assertFalse(registerResult);
    }
```

Listing 17: Implementazione di testRegisterNewUser() e testRegisterDuplicateUser()

4.2.3 Test delle operazioni RUD

Questi tre test sono progettati per verificare il corretto funzionamento delle operazioni di lettura (read), aggiornamento (update) e cancellazione (delete), assicurando al contempo l'integrità e la coerenza dei dati all'interno del file CSV *users.csv*.

```java
    @Test
    @DisplayName("Test that checks loading users from the CSV file works correctly and
        verifies that it's possible to find a user listed in the file")
    public void testLoadUsersFromCSV() {

        assertNotNull(authManager.getUsers());
        assertFalse(authManager.getUsers().isEmpty());

        User admin = authManager.getUsers().get("admin");
        assertInstanceOf(Admin.class, admin);
    }
```

Listing 18: Implementazione del testLoadUsersFromCSV

```java
    @Test
    @DisplayName("Test that checks the validity of a user's unsubscription")
    public void testRemoveUser() throws IOException {
        authManager.register("deleteUser", "password", "delete@example.com", "Passenger");
        authManager.removeUser("deleteUser");

        assertNull(authManager.getUsers().get("deleteUser"));
    }
```

Listing 19: Implementazione del testRemoveUser()

```java
    @Test
    @DisplayName("Test that checks the validity of updating a user's password and email")
    public void testUpdateUser() throws IOException {
        authManager.register("updateUser", "oldPassword", "oldEmail@example.com", "Passenger
            ");
        boolean updateResult = authManager.updateUser("updateUser", "newPassword", "
            newEmail@example.com");
        assertTrue(updateResult);

        Map<String, User> users = authManager.getUsers();
        User user = users.get("updateUser");
        assertEquals("newPassword", user.getPassword());
        assertEquals("newEmail@example.com", user.getEmail());
    }
```

Listing 20: Implementazione del testUpdateUser()

I test delle operazioni CRUD di tutti i manager hanno implementazioni simili.

4.3 FlightSearchServiceTest

Per testare la classe **FlightSearchService** nel contesto di *businessLogicTest*, è stato scelto l'uso dei Mock per isolare il test dai dati nei file CSV, garantendo una valutazione mirata e precisa delle sue funzionalità. I **Mock** nei test sono oggetti simulati che replicano il comportamento di componenti reali, ma senza interagire con dati o risorse esterne. Si utilizzano per isolare la funzionalità da testare, concentrandosi unicamente sul comportamento specifico della classe o del metodo sotto esame. Questo approccio permette di verificare accuratamente le operazioni senza dipendere da risorse come file o database.

```java
    public class FlightSearchServiceTest {

    @Mock
    private FlightManager mockFlightManager;

    @Mock
    private AirportManager mockAirportManager;

    @InjectMocks// I mock vengono iniettati al suo interno
    private FlightSearchService flightSearchService;
```

```
11
12      @BeforeEach
13      public void setUp() throws Exception {
14          try (var _ = MockitoAnnotations.openMocks(this)) {
15              // Usato per inizializzare le annotazioni di Mockito durante il test, es.@Mock
                    per creare oggetti mock
16              // openMocks restituisce un oggetto che implementa l'interfaccia AutoCloseable
                    che deve'essere chiuso per rilasciare correttamente le risorse
17
18              List<Airport> airports = new ArrayList<>();
19              airports.add(new Airport("JFK", "John F. Kennedy International Airport", "New
                    York", "USA"));
20              airports.add(new Airport("LAX", "Los Angeles International Airport", "Los
                    Angeles", "USA"));
21
22              List<Flight> flights = new ArrayList<>();
23              LocalDateTime departureTime = LocalDate.now().atTime(10, 0);
24              flights.add(new Flight("F001", "JFK", "LAX", departureTime,
25                      departureTime.plusHours(5).plusMinutes(30), 240, 54, 10));
26              flights.add(new Flight("F011", "LAX", "JFK", departureTime.plusHours(5),
27                      departureTime.plusHours(10).plusMinutes(30), 250, 50, 12));
28
29              // Simula il comportamento delle dipendenze, testa solo la logica all'interno di
                     questa classe senza interagire con i reali manager
30              when(mockFlightManager.getAllFlights()).thenReturn(flights);
31              when(mockAirportManager.getAllAirports()).thenReturn(airports);
32          }
33      }
34
35      @Test
36      @DisplayName("Test that checks the validity of a search of flight based on departure and
                 arrival(code or city)")
37      public void testSearchFlightsValidDepartureAndArrival() {
38          LocalDate searchDate = LocalDate.now();
39          List<Flight> result = flightSearchService.searchFlights("New York", "LAX",
                searchDate);
40
41          assertEquals(1, result.size());
42          assertEquals("F001", result.getFirst().getFlightNumber());
43      }
44
45      @Test
46      @DisplayName("Test that checks searching with no results works correctly")
47      public void testSearchFlightsNoResults() {
48          LocalDate searchDate = LocalDate.now();
49          List<Flight> result = flightSearchService.searchFlights("JFK", "SFO", searchDate);
50
51          assertTrue(result.isEmpty());
52      }
53
54      @Test
55      @DisplayName("Test that checks if a user enters a wrong date, the result should be empty
                ")
56      public void testSearchFlightsWithDifferentDate() {
57          LocalDate searchDate = LocalDate.now().plusDays(1); // Data diversa
58          List<Flight> result = flightSearchService.searchFlights("JFK", "LAX", searchDate);
59
60          assertTrue(result.isEmpty());
61      }
62  }
```

Listing 21: Implementazione di FlightSearchServiceTest tramite l'uso di Mock

4.4 FlightPlannerTest

Di seguito è riportata una selezione dei test effettuati sulla classe **FlightPlanner** nel contesto di *businesLogicTest* per verificarne le operazioni principali.

4.4.1 Test di remove

I seguenti test illustrano la logica della stretta interdipendenza tra un aeroporto, una rotta, un volo, le prenotazioni e i posti. Quando uno di questi elementi viene rimosso, anche tutti i componenti associati vengono eliminati automaticamente.

```
1     @Test
2     @DisplayName("Test that checks if removing an airport works correctly, ensuring that the
          related routes are also removed.")
3     public void testRemoveAirport() throws IOException {
4         Airport airport = flightPlanner.getAllAirports().stream()
5                 .filter(a -> a.getCode().equalsIgnoreCase("CDG"))
6                 .findFirst()
7                 .orElse(null);
8         assertNotNull(airport);
9
10        List<Route> routes = flightPlanner.getAllRoutes().stream()
11                .filter(r -> r.getDepartureAirportCode().equalsIgnoreCase(airport.getCode())
                      || r.getArrivalAirportCode().equalsIgnoreCase(airport.getCode()))
12                .toList();
13        assertFalse(routes.isEmpty());
14
15        flightPlanner.removeAirport(airport.getCode());
16
17        Airport removedAirport = flightPlanner.getAllAirports().stream()
18                .filter(a -> a.getCode().equalsIgnoreCase("CDG"))
19                .findFirst()
20                .orElse(null);
21        assertNull(removedAirport);
22
23        List<Route> removedRoutes = flightPlanner.getAllRoutes().stream()
24                .filter(r -> r.getDepartureAirportCode().equalsIgnoreCase(airport.getCode())
                      || r.getArrivalAirportCode().equalsIgnoreCase(airport.getCode()))
25                .toList();
26        assertTrue(removedRoutes.isEmpty());
27    }
```

Listing 22: Implementazione di testRemoveAirport()

```
1     @Test
2     @DisplayName("Test that checks if removing a route works correctly, ensuring that the
          related flights are also removed.")
3     public void testRemoveRoute() throws IOException {
4         Route route = flightPlanner.getAllRoutes().stream()
5                 .filter(r -> r.getRouteId().equalsIgnoreCase("R006"))
6                 .findFirst()
7                 .orElse(null);
8         assertNotNull(route);
9         Flight flight = flightPlanner.getAllFlights().stream()
10                .filter(f -> f.getDepartureAirportCode().equalsIgnoreCase(route.
                      getDepartureAirportCode()) &&
11                        f.getArrivalAirportCode().equalsIgnoreCase(route.
                              getArrivalAirportCode()))
12                .findFirst()
13                .orElse(null);
14        assertNotNull(flight);
15        String flightNumber = flight.getFlightNumber();
16
17        Seat seat = new Seat("20A", "Economy", flightNumber, true);
```

```
        flightPlanner.addSeatToFlight(flightNumber, seat);

        int economySeatsCounts = flightPlanner.getSeatClassCountsForFlight(flightNumber, "
            Economy");
        assertEquals(1, economySeatsCounts);

        flightPlanner.removeRoute(route.getRouteId());

        Route removedRoute = flightPlanner.getAllRoutes().stream()
                .filter(r -> r.getRouteId().equalsIgnoreCase("R006"))
                .findFirst()
                .orElse(null);

        assertNull(removedRoute);
        assertNull(flightPlanner.findFlight(flightNumber));

        int economySeatsCountsAfterRemoval = flightPlanner.getSeatClassCountsForFlight(
            flightNumber, "Economy");
        assertEquals(0, economySeatsCountsAfterRemoval);
    }
```

Listing 23: Implementazione di testRemoveRoute()

```
    @Test
    @DisplayName("Test that checks if removing a flight works correctly, ensuring that the
        related seats and bookings are also removed.")
    public void testRemoveFlight() throws IOException {

        Flight flight = flightPlanner.findFlight("F001");
        int seatEconomyCounts = flightPlanner.getSeatClassCountsForFlight(flight.
            getFlightNumber(), "Economy");
        int seatBusinessCounts = flightPlanner.getSeatClassCountsForFlight(flight.
            getFlightNumber(), "Business");
        int seatFirstCounts = flightPlanner.getSeatClassCountsForFlight(flight.
            getFlightNumber(), "First");
        int totalSeatsOnCSV = seatEconomyCounts + seatBusinessCounts + seatFirstCounts;
        List<Booking> bookingsForFlight = flightPlanner.getAllBookings().stream()
                .filter(b -> b.getFlightNumber().equalsIgnoreCase(flight.getFlightNumber()))
                .toList();
        List<Passenger> passengersForFlight = flightPlanner.getPassengers().stream()
                .filter(p -> p.isRegisteredForFlight(flight))
                .toList();
        assertNotNull(flight);
        assertEquals(4, totalSeatsOnCSV);
        assertEquals(2, bookingsForFlight.size());
        assertEquals(2, passengersForFlight.size());

        flightPlanner.removeFlight(flight.getFlightNumber());

        int seatEconomyCountsAfterRemoval = flightPlanner.getSeatClassCountsForFlight(flight
            .getFlightNumber(), "Economy");
        int seatBusinessCountsAfterRemoval = flightPlanner.getSeatClassCountsForFlight(
            flight.getFlightNumber(), "Business");
        int seatFirstCountsAfterRemoval = flightPlanner.getSeatClassCountsForFlight(flight.
            getFlightNumber(), "First");
        int totalSeatsOnCsvAfterRemoval = seatEconomyCountsAfterRemoval +
            seatBusinessCountsAfterRemoval + seatFirstCountsAfterRemoval;
        List<Booking> bookingsForFlightAfterRemoval = flightPlanner.getAllBookings().stream
            ()
                .filter(b -> b.getFlightNumber().equalsIgnoreCase(flight.getFlightNumber()))
                .toList();
        List<Passenger> passengersForFlightAfterRemoval = flightPlanner.getPassengers().
            stream()
                .filter(p -> p.isRegisteredForFlight(flight))
                .toList();
```

```
33
34          assertFalse(flightPlanner.getAllFlights().contains(flight));
35          assertEquals(0, totalSeatsOnCsvAfterRemoval);
36          assertTrue(bookingsForFlightAfterRemoval.isEmpty());
37          assertTrue(passengersForFlightAfterRemoval.isEmpty());
38      }
```

Listing 24: Implementazione di testRemoveFlight()

4.4.2 Test di aggiornamento sullo stato di un volo

Un altro test evidenzia la relazione tra una rotta e un volo: se un volo modifica l'orario di partenza o di arrivo, la durata della rotta viene automaticamente aggiornata per tutti i voli associati a quella rotta.

```
1       @Test
2       @DisplayName("Test that verifies the flight status update works correctly, ensuring that
            the related route is also updated.")
3       public void testUpdateFlightStatus() throws IOException {
4           Flight flight = flightPlanner.findFlight("F002");
5           LocalDateTime originalDepartureTime = flight.getDepartureTime();
6           LocalDateTime originalArrivalTime = flight.getArrivalTime();
7
8           Route route = flightPlanner.getRouteByAirportsCode(flight.getDepartureAirportCode(),
                flight.getArrivalAirportCode());
9           Duration originalDuration = route.getFlightDuration();
10
11          Passenger passenger = flightPlanner.getPassengers().stream()
12                  .filter(p -> p.isRegisteredForFlight(flight))
13                  .findFirst()
14                  .orElse(null);
15
16          LocalDateTime newDepartureTime = LocalDateTime.of(2024, 12, 26, 11, 0);
17          LocalDateTime newArrivalTime = LocalDateTime.of(2024, 12, 26, 15, 30);
18          String updateMsg = "Your flight has delayed!";
19          flightPlanner.updateFlightStatus("F002", newDepartureTime, newArrivalTime, updateMsg
                , NotificationType.DELAY);
20
21          assertNotEquals(originalDepartureTime, flight.getDepartureTime());
22          assertNotEquals(originalArrivalTime, flight.getArrivalTime());
23          assertNotEquals(originalDuration, route.getFlightDuration());
24          assertNotNull(passenger);// Non ha scelto come notifica di preferenza il ritardo
25          assertFalse(passenger.getNotifications().contains(updateMsg));
26      }
```

Listing 25: Implementazione di testUpdateFlightStatus()

4.5 FlightPlannerAppTest

Nella classe **FlightPlannerAppTest** di *businessLogicTest* sono stati utilizzati comandi come *clickOn* e *write* per simulare le interazioni grafiche, ad esempio il clic su pulsanti e la scrittura nei campi di testo. Per semplificare i test, sono stati assegnati *ID* ai componenti grafici principali, permettendo di identificarli facilmente durante le simulazioni. Nei casi in cui vi sia un cambio di finestra, è stato necessario inserire dei comandi *sleep* per evitare problemi di sincronizzazione.
Tutti i metodi di FlightPlannerApp inizialmente erano privati, scelta che rispondeva all'esigenza di mantenere la classe il più incapsulata possibile. Tuttavia, durante i test è emersa l'utilità di un metodo pubblico *showAppScreen()*, che fa un semplice getter di *showMainApp*. Questo approccio consente di evitare di ripetere i passaggi già eseguiti nel test di login e facilita l'accesso diretto alla schermata principale dell'app per i test delle funzionalità successive all'accesso.
Per avviare i test è stato usato il metodo *start(Stage stage)*, con annotazione @Override, nel quale viene

creata un'istanza di FlightPlannerApp e chiamato il suo metodo start. Questa impostazione inizializza *authManager* e *flightPlanner* tramite i rispettivi getter, permettendo di utilizzare le funzionalità di autenticazione e pianificazione nei test.

```
    @Test
    @DisplayName("Test that simulates the login process of a user with valid credentials")
    public void testShowLoginScreenWithValidCredentials() throws IOException {
        authManager.register("testuser", "password123", "testuser@example.com", "Passenger")
            ;

        clickOn("#usernameField").write("testuser");
        clickOn("#passwordField").write("password123");
        clickOn("#loginButton");
        assertThat(flightPlannerApp.getPrimaryStage().getTitle()).isEqualTo("Flight Planner
            App");
    }

    @Test
    @DisplayName("Test that simulates the registration of a user with an invalid password")
    public void testShowLoginScreenRegisterWithInvalidData() {
        clickOn("#usernameField").write("newuser1");
        clickOn("#passwordField").write("pass");
        clickOn("#emailField").write("newuser1@example.com");
        clickOn("#registerButton");

        verifyAlertDialog("Password must be at least 6 characters long.");
    }

    private void verifyAlertDialog(String expectedContent) {
        // Trova il DialogPane di alert
        DialogPane dialogPane = lookup(".dialog-pane").queryAs(DialogPane.class);

        assertThat(dialogPane.getHeaderText()).isNull();
        assertThat(dialogPane.getContentText()).isEqualTo(expectedContent);
    }

    @Test
    @DisplayName("Test simulating a successful flight search with specified departure and
        arrival locations")
    public void testShowSearchFlights() {
        authManager.login("jdoe", "pass123");
        flightPlannerApp.showAppScreen();
        sleep(100);

        clickOn("#searchFlightsButton");

        write("new york");
        clickOn("OK");
        write("LAX");
        clickOn("OK");
        write("2024-12-25");
        clickOn("OK");

        TextArea resultArea = lookup("#resultTextArea").query();

        String expecteString = """
                Flight: F001 Departure Airport Code: JFK Departure Time: 2024-12-25T08:00
                    Arrival Airport Code: LAX Arrival Time: 2024-12-25T13:30
                Route ID: R001, Distance: 2811.0 km, Flight Duration: 5h30m
                Flight: F006 Departure Airport Code: LGA Departure Time: 2024-12-25T10:30
                    Arrival Airport Code: LAX Arrival Time: 2024-12-25T16:00
                Route ID: R006, Distance: 2800.0 km, Flight Duration: 5h30m""";

    assertTrue(resultArea.getText().contains(expecteString));
    }
```

```
57
58      @Test
59      @DisplayName("Test simulating a failed management notification process")
60      public void testShowManageNotificationScreen_Error() {
61          authManager.login("jdoe", "pass123");
62          flightPlannerApp.showAppScreen();
63          sleep(100);
64
65          // Salvare senza fornire un numero di cellulare
66          clickOn("#manageNotificationButton");
67          clickOn("#smsCheckBox");
68
69          clickOn("#saveButton");
70          verifyAlertDialog("Please provide a phone number for SMS notifications.");
71
72          clickOn("OK");
73
74          // Salvare senza fornire un numero nel formato corretto
75          clickOn("#phoneNumberField").write("1234567890");
76          clickOn("#saveButton");
77          verifyAlertDialog("Please include the international prefix (e.g. +39 xxxxxxxxxx for
                Italy) and the phone number must have between 6 and 13 digits (excluding the
                international prefix).");
78      }
79
80      @Test
81      @DisplayName("Test simulating the successful selection of payment method")
82      public void testShowPaymentMethod() {
83          authManager.login("jdoe", "pass123");
84          flightPlannerApp.showAppScreen();
85          sleep(100);
86
87          clickOn("#paymentMethodButton");
88
89          ComboBox<Object> paymentMethodComboBox = lookup("#payMethodComboBox").queryComboBox
                ();
90          interact(() -> paymentMethodComboBox.getSelectionModel().select(PaymentMethod.
                BANK_TRANSFER));
91          clickOn("#confirmButton");
92          verifyAlertDialog("Payment method updated to: BANK_TRANSFER");
93
94          // Per ripristinare all'impostazione originale dell'utente
95          clickOn("OK");
96          interact(() -> paymentMethodComboBox.getSelectionModel().select(PaymentMethod.
                CREDIT_CARD));
97          clickOn("#confirmButton");
98      }
99
100     @Test
101     @DisplayName("Test that simulates the successful update of a user's credentials")
102     public void testShowUpdateCredentials() {
103         String oldEmail = "jdoe@example.com";
104         String oldPassword = "pass123";
105
106         authManager.login("jdoe", "pass123");
107         flightPlannerApp.showAppScreen();
108         sleep(100);
109
110         clickOn("#updateUserCredentialsBtn");
111
112         doubleClickOn("#emailInput").eraseText(((TextField) lookup("#emailInput").query()).
                getText().length());
113         write("john_doe@example.com");
114         clickOn("#oldPasswordInput").write(oldPassword);
115         clickOn("#newPasswordInput").write("password123");
```

```
        clickOn("#confirmPasswordInput").write("password123");

        clickOn("#updateButton");
        verifyAlertDialog("Credentials updated successfully!");

        // Per ripristinare le credenziali originali
        clickOn("OK");
        verifyThat("#emailInput", TextInputControlMatchers.hasText("john_doe@example.com"));
        verifyThat("#oldPasswordInput", TextInputControlMatchers.hasText(""));
        verifyThat("#newPasswordInput", TextInputControlMatchers.hasText(""));
        verifyThat("#confirmPasswordInput", TextInputControlMatchers.hasText(""));

        doubleClickOn("#emailInput").eraseText(((TextField) lookup("#emailInput").query()).
            getText().length());
        write(oldEmail);
        clickOn("#oldPasswordInput").write("password123");
        clickOn("#newPasswordInput").write(oldPassword);
        clickOn("#confirmPasswordInput").write(oldPassword);
        clickOn("#updateButton");
        verifyAlertDialog("Credentials updated successfully!");
    }



    @Test
    @DisplayName("Test that simulates the successful updating of flight status, ensuring
        that the related route is also updated")
    public void testShowUpdateFlightStatus() throws IOException {
        flightPlanner.addFlight(new Flight("F500", "DO", "YOU", LocalDateTime.of(2024, 10,
            30, 8, 40),
                LocalDateTime.of(2024, 10, 30, 14, 30), 260, 40, 10));
        Duration flightDuration = Duration.between(LocalDateTime.of(2024, 10, 30, 8, 40),
                LocalDateTime.of(2024, 10, 30, 14, 30));
        flightPlanner.addRoute(new Route("R500", "DO", "YOU", 10340, flightDuration));

        authManager.login("admin", "admin123");
        flightPlannerApp.showAppScreen();
        sleep(100);

        clickOn("#updateFlightStatusButton");

        clickOn("#flightNumberField").write("F500");
        DatePicker departureDatePicker = lookup("#departureDatePicker").query();
        interact(() -> departureDatePicker.setValue(LocalDate.of(2024, 10, 30)));
        DatePicker arrivalDatePicker = lookup("#arrivalDatePicker").query();
        interact(() -> arrivalDatePicker.setValue(LocalDate.of(2024, 10, 30)));
        clickOn("#departureTimeField").write("08:30");
        clickOn("#arrivalTimeField").write("15:30");
        clickOn("#updateMsgField").write("Flight delayed");
        ComboBox<Object> notificationTypeComboBox = lookup("#notificationTypeComboBox").
            queryComboBox();
        interact(() -> notificationTypeComboBox.getSelectionModel().select(NotificationType.
            DELAY));

        clickOn("#updateFlightStatusBtn");

        verifyAlertDialog("Status for flight F500 has been updated and notified to
            passengers who opted for DELAY. ");

        verifyThat("#backButton", isVisible());
    }
```

Listing 26: Implementazione di acuni test in FlightPlannerAppTest

Riferimenti bibliografici

[1] **JavaFx Download**
https://gluonhq.com/products/javafx/

[2] **OpenCSV Users Guide**
http://opencsv.sourceforge.net/

[3] **OpenCSV Download**
https://mvnrepository.com/artifact/com.opencsv/opencsv/5.9

[4] **JUnit5**
https://junit.org/junit5/docs/current/user-guide/

[5] **TestFx Use Example**
https://github.com/TestFX/TestFX