UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

RELAZIONE CONCLUSIVA DI
OBJECT ORIENTATION

Sistema informativo per la gestione di un Aeroporto

Author:
Giandomenico IAMEO
N86002856

Supervisors:
Porfirio TRAMONTANA

Un elaborato redatto nel rispetto dei requisiti previsti per l'esame di Object Orientation

Corso di Laurea Triennale in Informatica
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

7 luglio 2025

Indice

Capitolo I

Descrizione e analisi del problema

1.1 Enunciato del problema

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

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

Il sistema gestisce i voli in *arrivo* e in *partenza*. Ogni volo è caratterizzato da un codice univoco, la compagnia aerea, l'aeroporto di origine (per i voli in arrivo a Napoli) e quello di destinazione (per i voli in partenza da Napoli), la data del volo, l'orario previsto e lo stato del volo (programmato, decollato, in ritardo, atterrato, cancellato). Gli amministratori di sistema hanno la possibilità di inserire nuovi voli e aggiornare le informazioni sui voli esistenti.

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

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

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

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

1.2 Dominio del problema

In questo paragrafo verranno descritti i concetti che rappresentano gli aspetti importanti del problema affrontato, e che sono stati individuati ed estrapolati dal relativo enunciato. Il lavoro svolto in questa fase sarà necessario per la costruzione di quelle che vengono chiamate le *schede CRC* e di cui si parlerà in seguito. Il primo passo sarà scoprire le classi che devono far parte del sistema, successivamente verranno identificate le associazioni e gli eventuali attributi di ciascuna classe e infine verranno determinati i principali servizi (o responsabilità) che una classe deve fornire.

1.2.1 Individuazione delle entità e delle associazioni

Le entità che sono state individuate nell'enunciato del problema sono:

Utente

Generalizzazione delle classi `Generico` e `Amministratore`. Presenta un insieme di operazioni minime, ma necessarie, per la gestione di un account, come la modifica o inserimento di una password, di un nome utente, ecc. Di ogni *Utente* si conserva il *nome utente* e la *password*.

Generico:

E' una classe specializzata di `Utente`. Estende le funzionalità della classe

`Utente` con l'aggiunta delle operazioni relative alla gestione delle prenotazioni. Di ogni utente `Generico` si conserva il *nome*, il *cognome*, gli *anni* e il numero di *telefono*. La classe Generico è collegata alla classe Prenotazione dal tipo di associazione *gestire*, che associa ciascun utente generico a numerose prenotazioni, ma una stessa prenotazione può essere correlata a un solo utente Generico.

Amministratore:
Specializzazione di `Utente`, utile per per gestire, modificare o cancellare i voli. Nell'enunciato è stato individuato il tipo di associazione *amministra* tra i due tipi di entità *Amministratore* e *Volo*. Il numero di istanze di associazione a cui partecipa un'entità di *Volo* è pari a `1`, mentre il numero di entità di associazione a cui partecipa un'entità di *Amministratore* è `1..*`.

Prenotazione:
Classe che permette di tenere traccia dei dati di prenotazione degli utenti generici. Di ogni `Prenotazione` si riportano il *nome* del passeggero, il *cognome* del passeggero, *numero biglietto*, *posto assegnato*, *numero bagagli* e lo *stato della prenotazione*. La classe *Prenotazione*, oltre ad essere collegata alla classe *Generico*, come si è già detto, è anche collegata alla classe *Volo*. Il tipo di associazione individuata è *esserVincolata* che associa a ciascuna prenotazione uno ed un solo volo, mentre un volo è correlato a `0` o più prenotazioni.

Volo:
Generalizzazione delle classi `Volo in arrivo` e `Volo in partenza`. La Classe è responsabile della memorizzazione delle proprietà che definiscono un *Volo*. Per ogni volo si conserva il *codice del volo*, la *compagnia aerea*, la *data del volo*, l'*orario di arrivo*, l'*orario di partenza*, il *gate*, e lo *stato del volo*. La classe *Volo*, come ribadito, è correlata alla classe *Prenotazione* e *Amministratore*.

Volo in arrivo:
Specializzazione della classe *Volo*. La sottoclasse eredita il concetto di cosa significa essere un *Volo*. E' stata progettata per la rappresentazione e la memorizzazione delle informazioni sui voli in arrivo. Conserva solo l'*aeroporto di origine*, poiché la destinazione è bene nota.

Volo in partenza:
Specializzazione della classe *Volo*. Utilizzata per le analoghe ragioni sopra menzionate. Si conserva solo l'*aeroporto di destinazone*.

Gate:
Classe per permette di tenere traccia della lista dei voli associati e del numero

identificativo del gate. La classe permette di associare i voli ai rispettivi Gate di imbarco tenendo conto della compatibilità degli orari. Ciò significa che gli orari non devono essere sovrapposti.

1.2.2 Individuazione delle responsabilità

Di seguito sono riportati alcuni dei servizi più rilevanti per alcune delle classi individuate.

Generico:
La classe *Generico* offre ai suoi esemplari la possibilità di:

- cercare una prenotazione;
- modificare le proprie prenotazioni in base ai dati del passeggero e al numero del volo;
- visualizzare aggiornamenti dei voli prenotati.

Amministratore:
La classe *Amministratore* offre ai suoi esemplari la possibilità di:

- gestire l'aggiornamento dei voli;
- modifcare il numero dei gate di imbarco.

1.3 Struttura dell'applicazione

Per questa applicazione è stato scelto come pattern architetturale il modello *BCE + DAO*. Il modello *BCE* (*Boundary Control Entity*) è un sistema di raggruppamento delle classi in base alle loro responsabilità.

L'adozione di questo modello ha comportato la seperazione tra interfaccia utente (*Boundary*), logica di controllo dell'applicazione (*Control*) e modello di dominio (*Entity*), al fine di rendere l'applicazione più scalabile, maneggievole e sostenibile nel tempo. Per quanto concerne la sua implementazione, è stato necessario suddividere le classi che afferiscono a questo pattern in diversi package, in base alle rispettive funzioni.

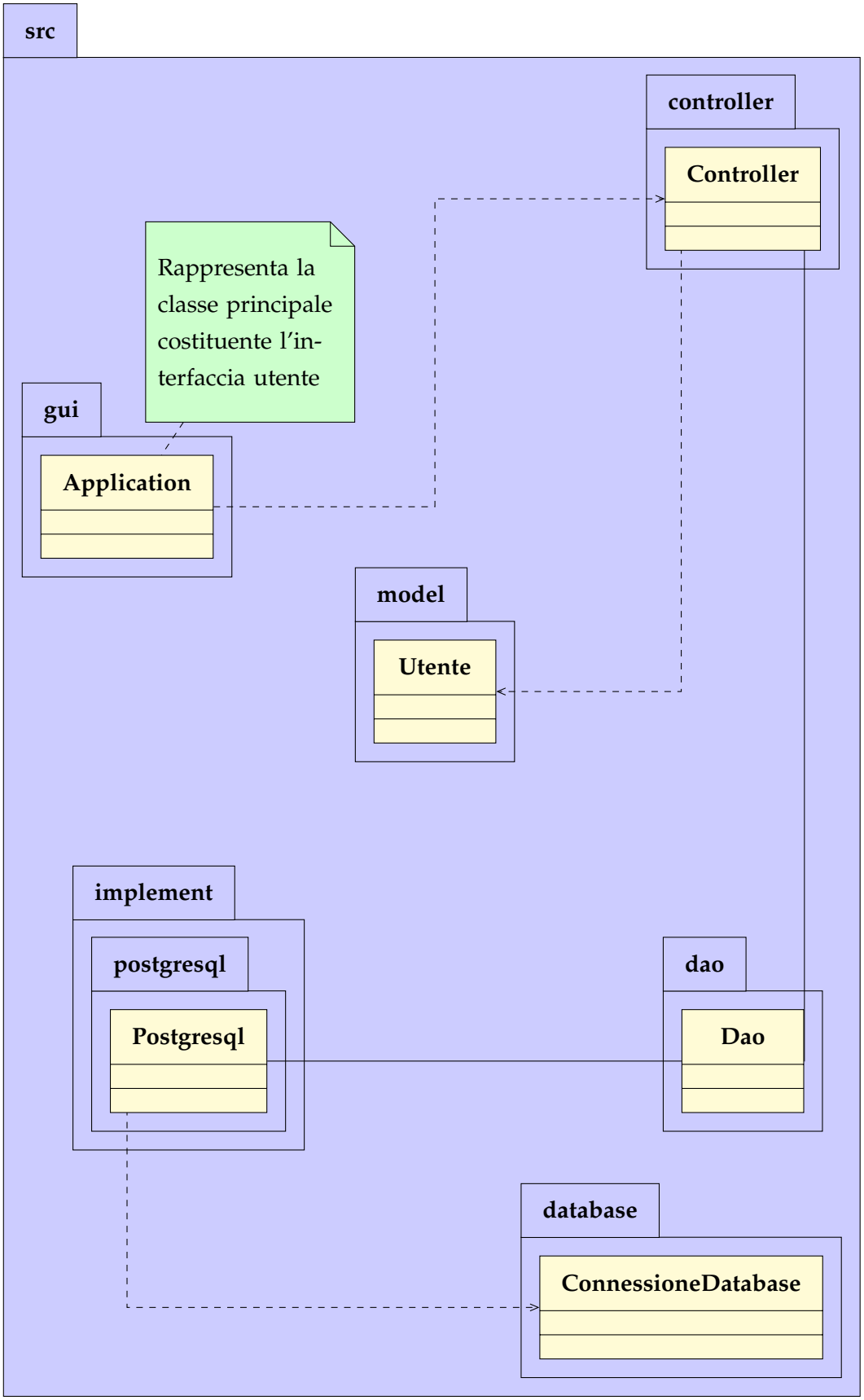
La scelta del modello *BCE* è stata influenzata anche dalla sua facile integrazione con altri modelli, quali il pattern *DAO*. Indipendentemente dalla specifica implementazione e dal particolare framework che ne fa uso, il pattern

DAO (*Data Access Object*) rappresenta la soluzione generica al problema dell'accesso a una o più basi di dati.

L'intento del pattern *DAO* è di disaccoppiare la logica di accesso ai dati persistenti dal resto dell'applicazione. L'implementazione di tale approccio ha richiesto la costruzione di un'unica classe, chiamata *Dao*, che è responsabile della gestione della persistenza dei dati, indipendentemente dalla natura del software di persistenza.

Il *DAO* viene invocato dal sistema di controllo dell'applicazione e si occupa di effettuare l'accesso ai dati restituendoli all'applicazione.

La struttura dell'applicazione è mostrata qui di seguito.


src
controller
Controller
Rappresenta la classe principale costituente l'interfaccia utente
gui
Application
model
Utente
implement
postgresql
Postgresql
dao
Dao
database
ConnessioneDatabase

Componente	Descrizione
`model`	Il package `model` racchiude i concetti, gli attributi, le associazioni e le responsabilità identificati nel dominio del problema e considerati significativi. In altre parole, il pacchetto fornisce una descrizione dettagliata del dominio del problema, elencando le entità di maggiore interesse per la realtà che si è cercato di modellare.
`controller`	Il package `controller` è costituito da un'unica classe, `Controller`, da cui prende il nome. Questa funge da intermediario tra le classi che fanno parte del pacchetto `gui`, e quelle incluse nei package `model` e `dao`. La logica di funzionamento dell'applicazione è dattata da questa unica classe: ogni operazione effettuata dall'utente all'interno dell'interfaccia grafica è supervisionata dal `Controller`, che comunica alle classi dei pacchetti `model` e `dao` i metodi appropriati da invocare. I risultati delle invocazioni vengono poi mostrati all'utente nell'interfaccia grafica.
`gui`	Il pacchetto `gui` comprende le classi che costituiscono l'interfaccia grafica dell'applicazione. Essa offre all'utente elenchi di opzioni e moduli di campi che guidano questo nella formulazione di una richiesta. Questa è stata progettata utilizzando la libreria grafica *Swing* di Java.
`database`	Il pacchetto `database` include un'unica classe che ha la responsabilità di gestire la connessione a un database relazionale gestito tramite il DBMS PostgreSQL.L'implementazione di quest'unica classe si basa sul pattern creazionale denominato *Singleton*. Questo ha lo scopo di garantire che venga istanziata un'unica istanza della classe, al fine di evitare la creazione di più connessioni verso il database durante l'esecuzione dell'applicazione.

`dao`	Il package `dao` fornisce una serie di interfacce che definiscono i metodi per l'accesso alla base di dati, come l'aggiunta, l'eliminazione o il recupero di un record da una tabella generica.
`implement`	Il package `implement` è costituito da classi concrete che implementano le interfacce presenti nel pacchetto `dao`. Dal momento che l'applicazione accede ed elabora i dati contenuti nel database tramite PostgreSQL, è stato creato un unico sottopacchetto denominato `postgresql`, che fornisce le implementazioni specifiche per questo DBMS. Il codice contiene istruzioni di query SQL scritte in linguaggio compatibile con PostgreSQL.

Capitolo II

Un modello statico del dominio del problema

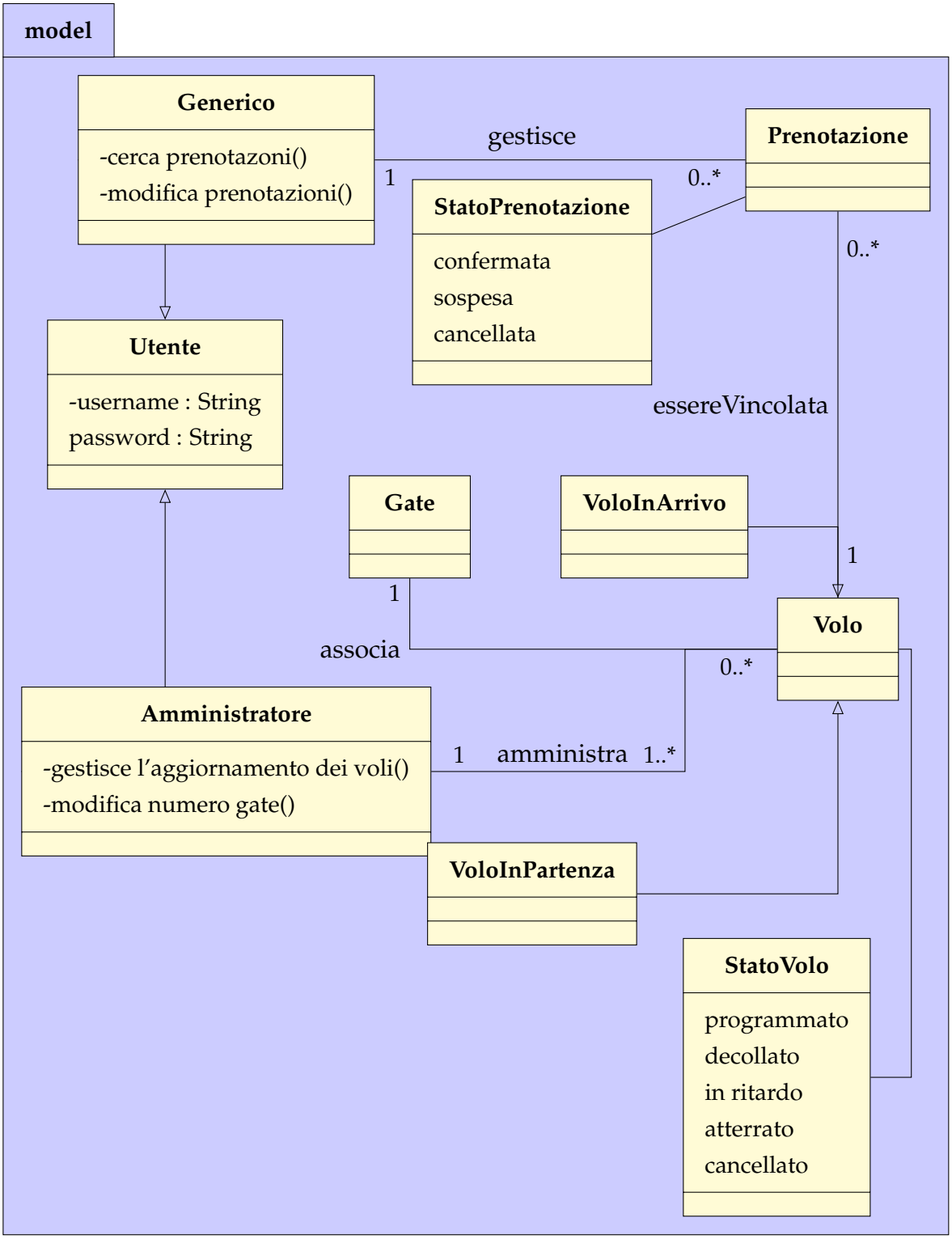
Con l'obiettivo di rispondere alla domanda "*cosa* fa il sistema?", in questo capitolo verrà presentato un modello *statico* del dominio del problema, attravero l'uso di *diagrammi delle classi*, arricchito con dei dizionari per fornire una descrizione più dettagliata delle classi e delle loro relazioni.

Il metodo espositivo che è stato adottato in questo capitolo comincia con l'illustrazionre del diagramma delle classi del dominio del problema, per poi passare alla descrizione delle classi mediante le schede CRC. Pur di privilegiare la chiarezza e la falicità di comprensione del modello, o di una classe in particolare, si è cercato di esprimere i nomi delle varie relazioni, dei metodi, e degli attributi in linguaggio naturale.
Il fine ultimo di tale modello consiste nel far capire anche a chi non possiede competenze in questi campo, la struttura concettuale del sistema software che è stato realizzato. Per faciliare ulteriormente il compito del lettore non specialista o privo di particolari conoscenze in questo ambito, sono state in incluse, come già detto, le schede CRC che forniscono maggiori dettagli sui costrutti collocati nel diagramma.
Chi legge il diagramma, insomma, non dovrebbe imbatttersi in termini di cui non conosce il significato o trovarsi di fronte a un'associazione di cui non può comprendere il concetto.

Prima di presentare il modello, è necessario precisare che, nel delineare gli aspetti più rilevanti del problema affrontato, molte delle responsabilità e attributi delle classi, sebbene deducibili dall'enunciato del problema, sono stati tralasciati, ma riportanti nel dizionario delle classi.

2.1 Schede CRC e diagrammi delle classi

2.1.1 Il pacchetto model



2.1.2 Schede CRC del pacchetto model

Qui di seguito verranno mostrate le schede CRC del pacchetto `model`.

Nome classe	*Amministratore*
Superclasse	*Utente*
Sottoclasse	*Nessuna*
Responsabilità	*gestire l'aggiornamento dei voli*
Responsabilità	*modificare il numero dei gate di imbarco*
Collaboratori	*Volo*

TABELLA 2.1: Scheda CRC della classe *Amministratore*.

Nome classe	*Generico*
Superclasse	*Utente*
Sottoclasse	*Nessuna*
Responsabilità	*cercare una prenotazione*
Responsabilità	*modificare le prenotazioni in base al passeggero e al volo*
Responsabilità	*visualizzare aggiornamenti dei voli*
Collaboratori	*Prenotazione*

TABELLA 2.2: Scheda CRC della classe *Generico*.

Nome classe	*Utente*
Superclasse	*Nessuna*
Sottoclasse	*Amministratore,Generico*
Responsabilità	*gestire le operazioni di inserimento e modifica delle credenziali di accesso*
Collaboratori	*Nessuna*

TABELLA 2.3: Scheda CRC della classe *Utente*.

Nome classe	*Prenotazione*
Superclasse	*Nessuna*
Sottoclasse	*Nessuna*
Responsabilità	*Offre la possiblità di accedere alle proprietà di una specifica prenotazione*
Collaboratori	*Generico,Volo*

TABELLA 2.4: Scheda CRC della classe *Prenotazione*.

Nome classe	*Volo*
Superclasse	*Nessuna*
Sottoclasse	*VoloInArrivo,VoloInPartenza*
Responsabilità	*Offre la possibilità di gestire un volo, accedendo ai suoi attributi*
Collaboratori	*Prenotazione,Amministratore*

TABELLA 2.5: Scheda CRC della classe *Volo*.

Nome classe	*VoloInArrivo*
Superclasse	*Volo*
Sottoclasse	*Nessuna*
Responsabilità	*Permette di gestire un volo in arrivo*
Collaboratori	*Nessuna*

TABELLA 2.6: Scheda CRC della classe *VoloInArrivo*.

Nome classe	*VoloInPartenza*
Superclasse	*Volo*
Sottoclasse	*Nessuna*
Responsabilità	*Permette di gestire un volo in partenza*
Collaboratori	*Nessuna*

TABELLA 2.7: Scheda CRC della classe *VoloInPartenza*.

Nome classe	*Gate*
Superclasse	*Nessuna*
Sottoclasse	*Nessuna*
Responsabilità	*Permette di gestire i gate condivisi tra vari voli*
Collaboratori	*Volo*

TABELLA 2.8: Scheda CRC della classe *Gate*.

Capitolo III

Un modello dinamico del dominio del problema

3.1 Diagramma di sequenza

Nella presente sezione verrà illustrato il comportamento di uno degli scenari più importanti che vengono eseguiti nel sistema. La modellazione del comportamento di questo scenario ha richiesto la trattazione di un particolare diagramma, noto come *diagramma di sequenza*.

I diagrammi delle classi sono *statici*: descrivono la struttura base del sistema software, quindi non sono adatti a modellare gli aspetti dinamici dello stesso. Per questo motivo è stato presentato un diagramma dinamico, usato per documentare gli aspetti dinamici del sistema. La funzionalità che è stata scelta per essere documentata tramite tale diagramma è l'aggiunta di un account Amministratore nella base di dati.

3.2 Creazione di un account Amministratore

Come già detto precedentemente, il presente diagramma mostra il funzionamento del metodo `controller.aggiungiAmministratore()`.
E' bene notare che per questioni di spazio, l'invocazione a tale metodo non è stata rappresentata nel diagramma, ma ciò non impedisce di comprenderne il suo comportamento. Il metodo ha il compito di registrare un account di tipo *Amministratore* all'interno della base di dati. Affinché il lettore comprenda perfettamente quando tale metodo viene invocato, è bene esporre alcune premesse. Un account amministratore all'interno del sistema non può

prescindere dall'esistenza di un volo. Quindi se esiste un amministratore *necessariamente* deve esistere anche un volo che gestirà. Il superutente nel momento in cui decide di creare un account amministratore deve assegnargli un volo da gestire. Da ciò si deduce che il volo deve esistere per poterlo assegnare. Successivamente, il superutente inserisce le credenziali di accesso per l'amministratore e gli assegna un volo. La pressione del pulsante Salva nella sezione Aggiungi Volo, nell'account superutente, scaturisce una sequenza di operazioni tra cui l'invocazione del metodo `controller.aggiungiAmministatore()` a cui vengono passate le credenziali di accesso, ovvero *username* e *password*. Queste credenziali saranno poi necessarie per creare un oggetto di tipo *Amministratore*, infatti verranno passate al suo costruttore. Infine tale oggetto verrà passato al metodo `add` che si preoccuperà di memorizzarlo nella base di dati.