**Documentazione Tecnica Completa: Sistema di Gestione Aeroportuale**

Sviluppatori: Myriam Sorrentino N86005248, Francesco Tibello N86005249

Anno Accademico: 24/25

Progetto: Object Orientation

**1. Introduzione e Obiettivi del Progetto**

Questo documento illustra l'architettura, il design e le funzionalità del sistema informativo per la gestione dell'aeroporto di Napoli. L'obiettivo primario del progetto è sviluppare un'applicazione software robusta e intuitiva, realizzata in **Java**, con un'interfaccia grafica basata su **Swing** e un sistema di persistenza dei dati affidato a un database relazionale **PostgreSQL**.

Il sistema è progettato per essere utilizzato da due categorie di utenti, con livelli di accesso e privilegi distinti, gestiti tramite un meccanismo di autenticazione:

* **Amministratori di Sistema**: Personale autorizzato con pieni poteri sulla gestione dei dati operativi. Le loro responsabilità includono l'inserimento di nuovi voli, l'aggiornamento delle informazioni sui voli esistenti (come ritardi, stati e assegnazione dei gate) e la consultazione di tutte le prenotazioni.
* **Utenti Generici**: Rappresentano i clienti o passeggeri. Possono consultare lo stato aggiornato di tutti i voli in arrivo e in partenza, cercare voli specifici ed effettuare prenotazioni per i voli disponibili.

**2. Architettura del Software e Pattern Utilizzati**

Il progetto è stato sviluppato seguendo un'architettura multi-strato che adotta il design pattern **Model-View-Controller (MVC)**, potenziato dall'uso del pattern **Data Access Object (DAO)**. Questa scelta progettuale garantisce una netta separazione delle responsabilità (Separation of Concerns), migliorando la manutenibilità, la scalabilità e la testabilità del codice.

* **Model (package model)**: Questo strato rappresenta il "cervello" dell'applicazione. Contiene le classi che modellano le entità del dominio del problema: Volo, Passeggero, Prenotazione, Utente, Gate e le loro specializzazioni. Include anche la logica di business e le regole che governano i dati, rendendolo completamente indipendente dall'interfaccia utente.
* **View (package gui)**: È lo strato di presentazione, responsabile di tutto ciò che l'utente vede e con cui interagisce. La classe principale è AppGUI, che utilizza la libreria Swing per costruire l'intera interfaccia grafica. La View è "passiva": visualizza i dati forniti dal Controller e inoltra a quest'ultimo tutte le azioni dell'utente (click, input di testo, ecc.) senza elaborarle.
* **Controller (package controller)**: La classe AppController funge da intermediario e orchestratore. Riceve le richieste dalla View, le interpreta e invoca i metodi appropriati sul Model per manipolare i dati. Successivamente, seleziona la vista corretta da presentare all'utente. Contiene anche logiche di validazione dell'input e di mappatura dei dati tra la View e il Model.
* **Data Access Object (package dao)**: Questo strato cruciale astrae la logica di persistenza.
  + **Interfacce** (VoloDAO, PrenotazioneDAO, PasseggeroDAO): Definiscono un contratto, specificando quali operazioni di persistenza devono essere disponibili (es. creaNuovoVolo, findVoloByCodice).
  + **Implementazioni** (VoloDAOImpl, PrenotazioneDAOImpl, PasseggeroDAOImpl): Contengono l'implementazione concreta di tali operazioni, utilizzando il driver **JDBC** per formulare ed eseguire query **SQL** specifiche per il database PostgreSQL. Grazie a questa astrazione, un eventuale cambio di database richiederebbe la modifica solo di queste classi di implementazione, lasciando inalterato il resto dell'applicazione.
* **Utilities (package util)**: Contiene classi di supporto trasversali. DatabaseConnection è un esempio chiave, centralizzando la logica e i parametri per stabilire una connessione con il database.

**3. Diagrammi di Progettazione**

**3.1 Diagramma delle Classi (UML)**

Il diagramma delle classi UML fornisce una rappresentazione visiva statica della struttura del sistema, mostrando le classi, i loro attributi, i metodi e le relazioni che intercorrono tra di esse (associazioni, generalizzazioni, dipendenze).

Il diagramma evidenzia le relazioni chiave del progetto:

* La gerarchia di generalizzazione tra la classe base Utente e le sue specializzazioni Utente\_generico e Amministratore\_sistema.
* Una struttura simile per Volo, con le sottoclassi Volo\_partenza e Volo\_arrivo.
* L'associazione tra Utente\_generico e Prenotazione, che indica la capacità di un utente di effettuare una o più prenotazioni.
* La composizione tra Prenotazione e Passeggero, indicando che ogni prenotazione è intrinsecamente legata a un passeggero.
* Le associazioni tra Prenotazione e Volo, e tra Volo\_partenza e Gate.

Questo diagramma è uno strumento fondamentale per comprendere l'architettura del modello dei dati e le interconnessioni logiche tra le diverse componenti del software.

***[INSERIRE QUI L'IMMAGINE DEL DIAGRAMMA UML PRESENTE A PAGINA 6 DEL FILE "Documentazione Progetto Object Orientation.pdf"]***

**3.2 Diagramma di Sequenza: Esempio di un Flusso Operativo**

I diagrammi di sequenza sono utilizzati per modellare le interazioni tra gli oggetti in un determinato scenario, mostrando la sequenza temporale dei messaggi scambiati.

Un flusso operativo cruciale da rappresentare sarebbe quello della **"Creazione di una Nuova Prenotazione"**. Un diagramma di sequenza per questo scenario illustrerebbe i seguenti passaggi:

1. L'**utente** interagisce con la AppGUI, cliccando sul pulsante di conferma dopo aver compilato i dati.
2. L'oggetto PrenotazioneDialog (un componente della AppGUI) invoca il metodo creaNuovaPrenotazione() sull'istanza di AppController.
3. L'AppController crea un'istanza di Passeggero e Prenotazione con i dati ricevuti.
4. L'AppController invoca il metodo creaPrenotazione() sull'oggetto PrenotazioneDAOImpl.
5. L'oggetto PrenotazioneDAOImpl invoca a sua volta il metodo save() sull'oggetto PasseggeroDAOImpl per assicurarsi che il passeggero sia salvato nel database.
6. PasseggeroDAOImpl esegue le operazioni SQL sulla tabella Passeggeri.
7. Successivamente, PrenotazioneDAOImpl esegue l'operazione SQL di inserimento sulla tabella Prenotazioni.
8. Il risultato dell'operazione (booleano) viene restituito a catena fino all'AppGUI, che può così mostrare un messaggio di successo o di fallimento all'utente.

Questo diagramma mostrerebbe chiaramente come le responsabilità sono delegate attraverso i vari strati dell'architettura per completare un'operazione complessa.

***[INSERIRE QUI UN DIAGRAMMA DI SEQUENZA CHE MODELLA IL FLUSSO "CREAZIONE NUOVA PRENOTAZIONE"]***

**4. Descrizione Dettagliata delle Classi del Model**

Di seguito è riportata una descrizione approfondita di ciascuna classe appartenente al package model.

1. **Utente**
   1. **Scopo**: Classe base che rappresenta un generico utente del sistema.
   2. **Attributi**:
      1. login (String): Il nome utente per l'autenticazione. È final per garantirne l'immutabilità dopo la creazione.
      2. password (String): La password associata all'utente.
   3. **Descrizione**: Fornisce la struttura fondamentale per l'autenticazione. Non è pensata per essere istanziata direttamente, ma per essere estesa.
2. **Utente\_generico**
   1. **Scopo**: Rappresenta l'utente standard del sistema, tipicamente un passeggero.
   2. **Eredita da**: Utente.
   3. **Metodi Chiave**:
      1. creaPrenotazione(...): Metodo (ora non utilizzato dalla GUI, ma presente nel modello) che assembla un oggetto Prenotazione a partire dai dati del passeggero. Include una logica interattiva da console per aggiungere bagaglio e assicurazione.
      2. modificaPrenotazione(...): Logica da console per modificare i dettagli di una prenotazione esistente.
3. **Amministratore\_sistema**
   1. **Scopo**: Rappresenta un utente con privilegi amministrativi.
   2. **Eredita da**: Utente.
   3. **Metodi Chiave**:
      1. creaVoloPartenza(...) / creaVoloArrivo(...): Metodi (logica da console) per istanziare nuovi oggetti Volo\_partenza o Volo\_arrivo.
      2. modificaVolo(...): Metodi sovraccaricati per modificare i dettagli di un volo (Volo\_partenza o Volo\_arrivo) tramite input da console.
4. **Volo**
   1. **Scopo**: Classe base che modella le informazioni essenziali di un volo.
   2. **Attributi**: codice, compagniaAerea, data, orarioPrevisto, ritardo, origine, destinazione, stato (Stato\_del\_volo).
   3. **Descrizione**: Incapsula tutti i dati fondamentali di un volo, indipendentemente dal fatto che sia in arrivo o in partenza.
5. **Volo\_partenza**
   1. **Scopo**: Specializzazione di Volo per i voli in partenza.
   2. **Eredita da**: Volo.
   3. **Attributi Aggiuntivi**: gate (Gate).
   4. **Descrizione**: Nel costruttore, l'origine viene impostata di default a " Napoli". La presenza dell'oggetto Gate è la sua caratteristica distintiva.
6. **Volo\_arrivo**
   1. **Scopo**: Specializzazione di Volo per i voli in arrivo.
   2. **Eredita da**: Volo.
   3. **Descrizione**: Nel costruttore, la destinazione viene impostata di default a "Napoli".
7. **Prenotazione**
   1. **Scopo**: Modella l'atto della prenotazione di un posto su un volo da parte di un passeggero.
   2. **Attributi**: pnr (String), num\_biglietto (String), stato (Stato\_prenotazione), assicurazione (boolean), bagaglio (boolean), passeggero (Passeggero), codiceVolo (String).
   3. **Descrizione**: È una classe centrale che collega un Passeggero a un Volo (tramite codiceVolo). I metodi updateAssicurazione() e updateBagaglio() invertono il valore booleano delle rispettive opzioni.
8. **Passeggero**
   1. **Scopo**: Contiene i dati anagrafici della persona fisica che viaggia.
   2. **Attributi**: nome, cognome, email, ssn, posto, telefono.
   3. **Descrizione**: Rappresenta l'entità "passeggero", che può non coincidere con l'entità "utente" che effettua la prenotazione.
9. **Gate**
   1. **Scopo**: Modella un gate di imbarco dell'aeroporto.
   2. **Attributi**: numero (int).
   3. **Descrizione**: Una semplice classe contenitore per il numero identificativo di un gate.
10. **Stato\_del\_volo (Enum)**
    1. **Scopo**: Definisce un tipo di dato enumerativo per rappresentare in modo sicuro e controllato i possibili stati di un volo.
    2. **Valori**: cancellato, rinviato, in\_ritardo, in\_orario, atterrato.
11. **Stato\_prenotazione (Enum)**
    1. **Scopo**: Definisce un tipo di dato enumerativo per gli stati di una prenotazione.
    2. **Valori**: confermata, in\_attesa, cancellata.

**5. Interazione con il Database (Strato DAO)**

L'accesso ai dati è gestito in modo sicuro e manutenibile attraverso il pattern DAO.

* **PasseggeroDAOImpl**:
  + Il metodo save(Passeggero passeggero) implementa una logica di tipo "UPSERT" (update/insert) concettuale. Prima di eseguire un INSERT, controlla se un passeggero con lo stesso SSN esiste già, richiamando findBySsn(). In caso affermativo, l'operazione di salvataggio viene saltata per evitare duplicati. Questo è fondamentale per garantire l'unicità dei passeggeri nel database.
* **PrenotazioneDAOImpl**:
  + Il metodo creaPrenotazione(Prenotazione prenotazione) orchestra l'inserimento di una prenotazione. Per prima cosa, invoca passeggeroDAO.save() per assicurarsi che il passeggero sia presente nel database. Successivamente, esegue un INSERT nella tabella Prenotazioni, utilizzando i dati della prenotazione e del passeggero.
  + getPrenotazioniFiltrateAdmin costruisce una query SELECT che effettua una JOIN tra le tabelle Prenotazioni e Passeggeri per recuperare i dati completi, filtrando tramite l'operatore LIKE e LOWER per una ricerca case-insensitive su nome e cognome.
* **VoloDAOImpl**:
  + I metodi getVoliInArrivo() e getVoliInPartenza() eseguono query SELECT sulla tabella Voli, filtrando per il campo tipo\_volo.
* I l metodo creaNuovoVolo(Volo volo) dimostra l'efficacia del polimorfismo. Utilizza l'operatore instanceof per determinare se l'oggetto Volo passato sia un Volo\_partenza o un Volo\_arrivo. In base al tipo, imposta correttamente la stringa tipo\_volo e il valore del gate (che può essere NULL per i voli in arrivo) nella query INSERT.

**6. Analisi dell'Interfaccia Grafica (AppGUI) e Flussi Utente**

La classe AppGUI è il fulcro dello strato di presentazione e gestisce l'intera esperienza utente in modo dinamico.

**6.1 Struttura e Navigazione**

L'interfaccia si basa su un CardLayout di Swing, che permette di gestire più pannelli all'interno di un unico contenitore e di "sfogliarli" come un mazzo di carte.

* **Layout Esterno**: Gestisce la transizione tra lo stato "non autenticato" (pannelli Volo\_Initial e Login) e lo stato "autenticato" (un unico pannello contenitore MainContentArea\_Actual).
* **Layout Interno**: Il pannello MainContentArea\_Actual contiene a sua volta un CardLayout con i pannelli dedicati alle varie funzionalità post-login (es. Volo\_LoggedIn\_View, MiePrenotazioni\_View, AdminCreaVolo\_View), che vengono mostrati in base al ruolo dell'utente e alle sue azioni.

**6.2 Momenti Salienti dell'Interattività**

* **Login e Cambio di Contesto**: Al successo del login, AppGUI rimuove il pannello MainContentArea\_Placeholder e lo sostituisce con MainContentArea\_Actual, che viene costruito dinamicamente in base al ruolo dell'utente (Utente o Amministratore). I pulsanti di navigazione (es. "Gestisci Prenotazioni", "Crea Nuovo Volo") vengono resi visibili o nascosti di conseguenza.
* **Dialogo di Prenotazione (PrenotazioneDialog)**: Questo JDialog modale è un eccellente esempio di UI complessa e interattiva:
  + **Costruzione Dinamica**: All'apertura, interroga il controller per ottenere la lista dei postiOccupati per il volo selezionato.
  + **Mappa dei Posti**: Genera una griglia di JButton. Ogni bottone viene configurato in base al suo stato: disabilitato e colorato di **rosso** se occupato, abilitato e **verde** se libero.
  + **Gestione della Selezione**: Un ActionListener è associato a ogni posto libero. Quando un posto viene cliccato, il metodo selezionaPostoDialog viene invocato. Questo metodo aggiorna lo stato interno (la variabile postoAttualmenteSelezionato), colora il nuovo posto selezionato di **arancione** e ripristina il colore verde del posto precedentemente selezionato, fornendo un feedback visivo immediato.
  + **Validazione Rigorosa**: Prima di procedere, il metodo validaDatiPrenotazione controlla che tutti i campi siano compilati e che l'email sia in un formato valido. In caso di errore, mostra un JOptionPane e, se il problema è la mancata selezione del posto, sposta programmaticamente il focus dell'utente sulla tab "Scelta Posto".
* **Creazione Volo per Admin**: Il pannello di creazione volo impedisce errori di inserimento tramite una logica reattiva. La selezione dal JComboBox del tipo di volo ("In Partenza" o "In Arrivo") scatena un evento che pre-compila e rende non modificabili i campi "Origine" o "Destinazione" con "Napoli NAP" e abilita o disabilita il campo "Gate" di conseguenza.

**7. Flussi Operativi Chiave**

**7.1 Flusso di una Prenotazione**

1. **Autenticazione**: L'utente accede con le proprie credenziali.
2. **Selezione**: Dalla tabella dei voli, l'utente clicca su un volo idoneo.
3. **Controllo di Stato**: L'applicazione verifica internamente se lo stato del volo (cancellato, atterrato, etc.) permette la prenotazione.
4. **Dialogo Interattivo**: Viene mostrato il PrenotazioneDialog. L'utente inserisce i dati e seleziona il posto dalla mappa grafica.
5. **Conferma e Salvataggio**: Dopo la validazione, il Controller riceve i dati, crea gli oggetti Passeggero e Prenotazione, e li passa allo strato DAO che li scrive sul database.
6. **Feedback e Aggiornamento**: L'utente riceve un messaggio di conferma e le viste "Le Mie Prenotazioni" e delle tabelle voli vengono aggiornate automaticamente per riflettere la nuova prenotazione.

**7.2 Flusso di Gestione Amministrativa**

1. **Autenticazione Admin**: L'amministratore accede con credenziali privilegiate.
2. **Accesso a Funzioni Esclusive**: La GUI mostra i pulsanti per la gestione delle prenotazioni e la creazione di nuovi voli.
3. **Ricerca Prenotazioni**: L'admin può cercare le prenotazioni per nome e cognome, ottenendo una vista tabellare completa dei risultati.
4. **Creazione Volo**: L'admin compila il modulo di creazione volo, guidato dalla logica reattiva dell'interfaccia.
5. **Persistenza e Feedback**: Dopo la validazione da parte del Controller, il nuovo volo viene salvato nel database tramite il DAO e la tabella dei voli si aggiorna immediatamente.

Questo approccio strutturato e dettagliato garantisce che il sistema non solo soddisfi i requisiti funzionali, ma sia anche robusto, manutenibile e facilmente estendibile in futuro.