UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Applicativo Java AgroManager

Autore:
Lucian Alexandru Topliceanu

N° Matricola:
7003550

Corso principale:
Ingegneria del software

Docente corso:
Enrico Vicario

Indice

1 Introduzione

1.1 Statement

AgroManager è un sistema informatico progettato per la gestione e il tracciamento completo delle attività agricole. Il sistema si focalizza sul monitoraggio dell'intero ciclo di vita delle colture, dall'acquisto delle piante fino alla registrazione del raccolto, fornendo agli agricoltori strumenti integrati per:

- Gestione terreno: Suddividere e tracciare diverse zone di coltivazione, registrandone le caratteristiche principali, come il tipo di terreno e la dimensione.
- Gestione fornitori: Mantenere un registro centralizzato di tutti i fornitori, con i relativi dati di contatto e la partita IVA.
- Gestione piante: Creare un database di tutte le varietà vegetali, includendo i costi, i fornitori e le note colturali.
- Gestione delle piantagioni: Tenere traccia di ogni singola piantagione, monitorandone la quantità, la data di messa a dimora e l'evoluzione del suo stato nel tempo.
- Gestione raccolti: Documentare con precisione ogni operazione di raccolta, associandola alla rispettiva piantagione e arricchendola con note.
- Analisi dati: Trasformare i dati raccolti in report e statistiche sulla produttività, supportando decisioni data-driven per migliorare l'efficienza.

AgroManager punta a diventare uno strumento indispensabile per modernizzare l'attività agricola, migliorandone la tracciabilità, l'organizzazione operativa e, in ultima analisi, la redditività.

1.2 Tecnologie e strumenti utilizzati

1.2.1 Linguaggio di programmazione

- **Java 17**: Versione LTS (Long Term Support) del linguaggio Java

1.2.2 Framework e librerie

- **JavaFX 21.0.2**: Framework per la creazione dell'interfaccia grafica desktop
- **PostgreSQL 42.7.7** (JDBC Driver): Driver per connessione al database PostgreSQL
- **JUnit 5.8.1**: Framework per unit testing

1.2.3 Database

- **PostgreSQL**: Database relazionale utilizzato per la persistenza dei dati
- **Docker**: Piattaforma per l'esecuzione del container contenente il database (locale)

1.2.4 Build tool

- **Gradle 8.x**: Sistema di automazione della build con gestione centralizzata delle dipendenze

1.2.5 IDE e strumenti di sviluppo

- **IntelliJ IDEA**: IDE principale per lo sviluppo
- **Git**: Version Control System
- **Gradle Wrapper**: Garantisce versione consistente di Gradle

1.2.6 AI

- **Google Gemini**: AI utilizzata per la generazione di codice ripetitivo e per lo stile GUI

1.2.7 Documentazione

- **Overleaf**: Web LaTeX editor
- **PlantUML Editor**: Versione web di PlantUML per la creazione di diagrammi

1.3 Architettura generale

AgroManager è strutturato secondo un'architettura a layer, garantendo la separazione delle responsabilità, la manutenibilità e la testabilità del codice. I principali strati sono:

- **Presentation Layer**
  - **Scopo:** È l'interfaccia utente (GUI). È la parte dell'applicazione con cui l'utente finale interagisce direttamente.
  - **Componenti:**
    - View: Mostra le informazioni all'utente
    - Controller: Riceve l'input dell'utente (es. click del mouse, dati da un form) e decide cosa fare, comunicando con il layer sottostante.
- **Business Logic Layer**
  - **Scopo:** Contiene tutte le regole di business, i calcoli e i processi decisionali che definiscono come funziona l'applicazione.
  - **Componenti:**
    - BusinessLogic / Service / Strategy: Questi componenti implementano le funzionalità principali (es. Calcola produzione media, verifica validità dei dati, genera report). Ricevono richieste dal Presentation Layer e le elaborano.
- **Data Access Layer**
  - **Scopo:** Gestisce tutto ciò che riguarda la memorizzazione e il recupero dei dati. Fa da ponte tra la logica di business e il database.
  - **Componenti:**
    - Database: Il luogo dove i dati sono fisicamente memorizzati.
    - ORM (Object-Relational Mapping): Uno strumento che aiuta a "tradurre" gli oggetti usati nell'applicazione in tabelle del database, e viceversa.
    - JDBC (Java Database Connectivity): Una tecnologia specifica per connettersi ed eseguire comandi sul database.
- **Domain Layer**
  - **Scopo:** Questo layer definisce i concetti e i dati fondamentali.
  - **Componenti:**
    - DomainModel: Rappresenta le entità chiave del sistema con i loro attributi e relazioni.

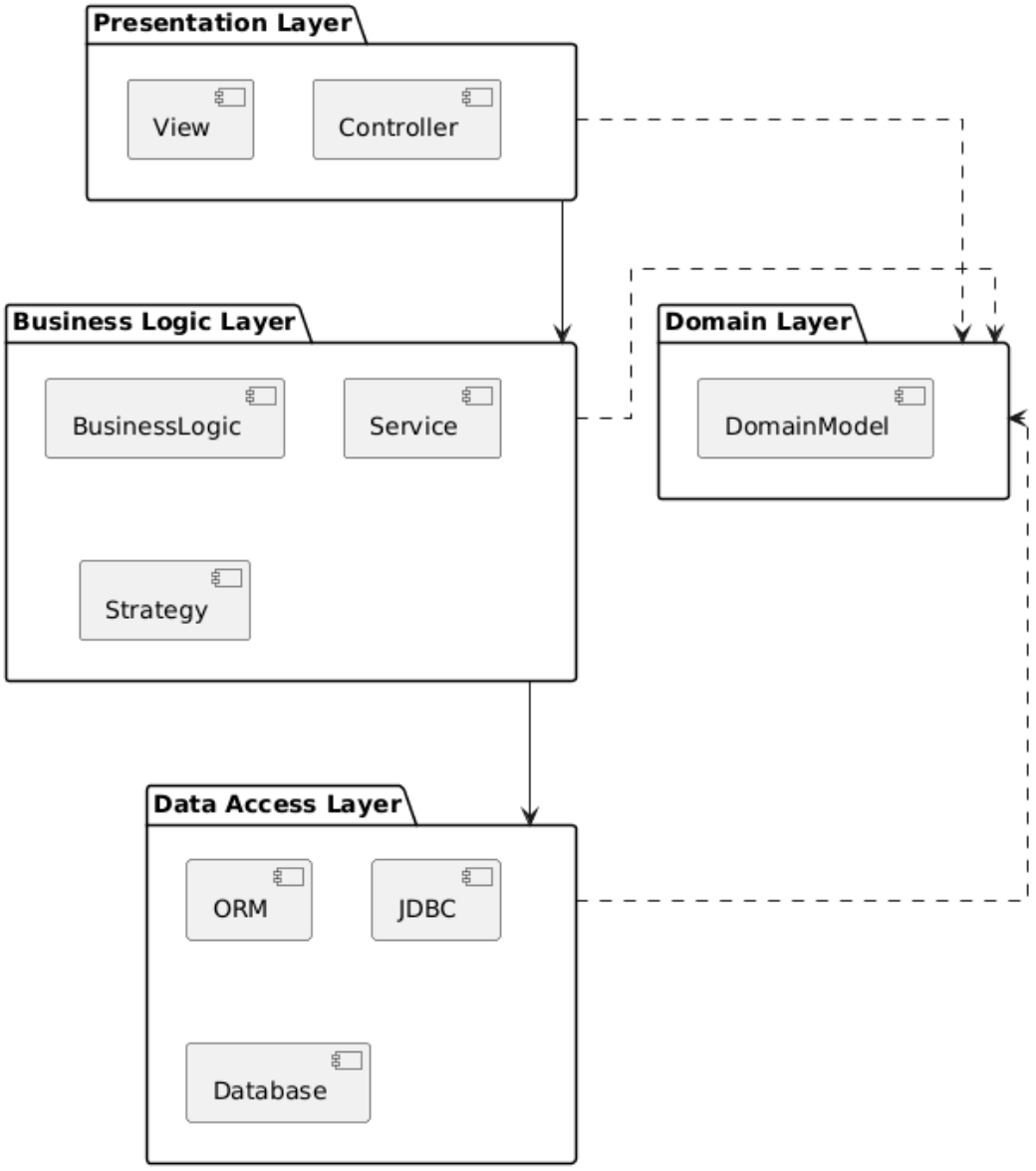

Figura 1: Diagramma rappresentante l'architettura a strati del sistema AgroManager, con i relativi componenti, e le relazioni tra i strati. Le relazioni tratteggiate (cross-cutting) con il Domain Layer indicano che il cambiamento di un elemento può implicare cambiamenti anche negli altri layer.

2 Analisi dei requisiti e design funzionale

2.1 Use case

Il diagramma degli Use Case (Figura 2) illustra le funzionalità principali del sistema AgroManager e le interazioni con l'attore. È stato identificato un singolo attore, il quale ha accesso a tutte le funzionalità.

Le funzionalità principali, identificate, sono:

- **Gestire Fornitori:** Inserimento, modifica, rimozione e filtro dei fornitori.
- **Gestire Piante:** Inserimento, modifica, rimozione e filtro delle varietà di piante.
- **Gestire Zone Agricole:** Inserimento, modifica, rimozione e filtro delle zone di coltivazione.
- **Gestire Piantagioni:** Inserimento, modifica, rimozione delle piantagioni con gestione degli stati del ciclo di vita.
- **Gestire Raccolti:** Inserimento, modifica, rimozione e tracciamento delle attività di raccolta.
- **Elaborare Dati e Report:** Analisi della produttività tramite calcoli e statistiche specializzate e generazione di report con salvataggio in formato TXT.
- **Visualizzare Dashboard:** Monitoraggio in tempo reale dello stato del sistema con statistiche aggregate e azioni rapide.

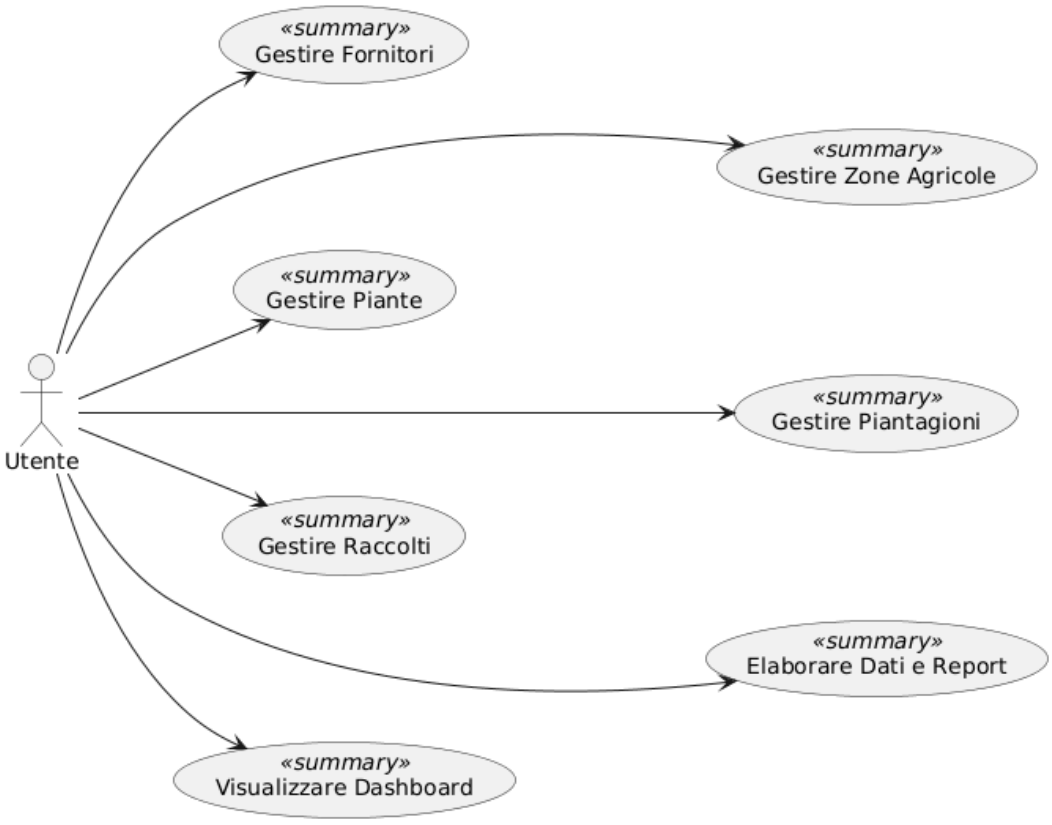

Figura 2: Diagramma degli Use Case del sistema AgroManager.

Nella Figura 3 è riportato il diagramma degli Use Case dettagliato, che include i ”User Goal” principali che rappresentano operazioni specifiche come l’aggiunta, la modifica, l’eliminazione e il filtraggio delle entità gestite dal sistema.

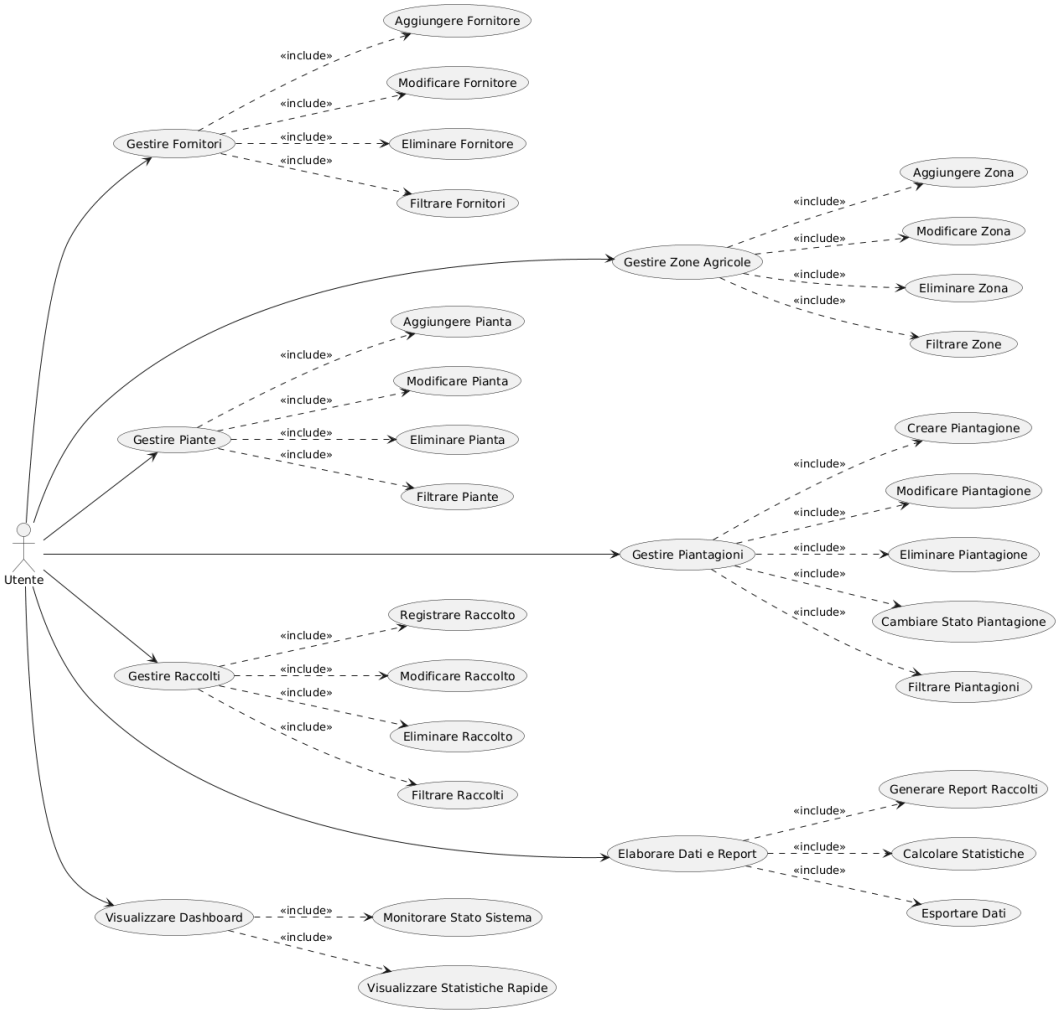

Figura 3: Diagramma degli Use Case del sistema AgroManager dettagliato.

2.2 Use case templates

Di seguito vengono presentati i template per alcuni dei principali casi d'uso individuati nell'applicativo. Le operazioni di base (come aggiungere, modificare, eliminare e filtrare) sono concettualmente identiche per tutte le entità gestite (Fornitori, Piante, Zone Agricole, ecc.). Per evitare ridondanze, verranno analizzati nel dettaglio solo i template relativi alla "Gestione Fornitore" (UC1) e alle sue sotto-funzionalità (UC1.1, UC1.2, UC1.3, UC1.4), considerandoli come modello rappresentativo anche per le altre sezioni gestionali.

2.2.1 UC1 - Gestire Fornitori

UC1	**Gestire Fornitori**
Livello	Summary
Descrizione	L'utente gestisce i fornitori attraverso l'interfaccia dell'applicativo
Attori	Utente
Pre-condizioni	L'utente deve avere accesso al sistema AgroManager e il database deve essere operativo
Post-condizioni	Le informazioni sui fornitori sono aggiornate nel sistema secondo le operazioni richieste
Normale svolgimento	1. L'utente accede alla sezione Fornitori dal menu principale 2. Il sistema visualizza l'elenco dei fornitori esistenti in una tabella 3. L'utente può selezionare un fornitore dall'elenco 4. L'utente può scegliere una delle seguenti operazioni: • Aggiungere un nuovo fornitore (UC1.1) • Modificare un fornitore esistente (UC1.2) • Eliminare un fornitore selezionato (UC1.3) • Applicare filtri di ricerca per nome, città o email (UC1.4) 5. Il sistema esegue l'operazione richiesta 6. Il sistema aggiorna l'elenco dei fornitori
Svolgimenti alternativi	**2a.** Errore di connessione al database: **2a.1** Il sistema mostra un dialog di errore connessione al database **2a.2** L'utente può riprovare l'operazione

2.2.2 UC1.1 - Aggiungere Fornitore

UC1.1	**Aggiungere Fornitore**
Livello	User Goal
Descrizione	L'utente registra un nuovo fornitore nel sistema per ampliare la rete di fornitori disponibili
Attori	Utente
Pre-condizioni	L'utente si trova nella sezione Fornitori e il sistema è operativo
Post-condizioni	Un nuovo fornitore è stato aggiunto al sistema e l'elenco fornitori è aggiornato
Normale svolgimento	1. L'utente clicca sul pulsante "Nuovo Fornitore" 2. Il sistema apre il dialog di inserimento nuovo fornitore 3. L'utente inserisce il nome del fornitore (campo obbligatorio) 4. L'utente inserisce l'indirizzo del fornitore (campo obbligatorio) 5. L'utente inserisce il numero di telefono (campo obbligatorio) 6. L'utente può inserire email e partita IVA (campi opzionali) 7. L'utente clicca "Salva" 8. Il sistema valida i dati inseriti 9. Il sistema salva il nuovo fornitore nel database 10. Il sistema chiude il dialog e aggiorna l'elenco fornitori
Svolgimenti alternativi	**7a.** Dati non validi: **7a.1** Il sistema evidenzia i campi con errori (nome, indirizzo o telefono vuoti, email non valida se inserita, telefono troppo corto) e mostra un dialog di errore di validazione **7a.2** L'utente corregge i dati errati **7a.3** L'utente clicca nuovamente "Salva" **7b.** Fornitore duplicato: **7b.1** Il sistema rileva che esiste già un fornitore con lo stesso nome **7b.2** Il sistema mostra messaggio di errore "Fornitore già esistente" mediante un (dialog di errore duplicato) **7b.3** L'utente modifica il nome del fornitore o annulla l'operazione **6a.** L'utente annulla l'operazione: **6a.1** L'utente clicca "Annulla" nel dialog **6a.2** Il sistema chiude il dialog senza salvare i dati

2.2.3 UC1.2 - Modificare Fornitore

UC1.2	**Modificare Fornitore**
Livello	User Goal
Descrizione	L'utente aggiorna le informazioni di un fornitore esistente per mantenere i dati del sistema aggiornati
Attori	Utente
Pre-condizioni	L'utente si trova nella sezione Fornitoro, esiste almeno un fornitore nel sistema, e l'utente ha selezionato un fornitore dall'elenco
Post-condizioni	Le informazioni del fornitore selezionato sono aggiornate e l'elenco fornitori riflette le modifiche
Normale svolgimento	1. L'utente seleziona un fornitore dall'elenco dei fornitori 2. L'utente clicca sul pulsante "Modifica" 3. Il sistema apre il dialog di modifica precompilato con i dati attuali del fornitore 4. L'utente modifica i campi desiderati (nome, indirizzo, telefono, email, partita IVA) 5. L'utente clicca "Salva" 6. Il sistema valida i nuovi dati inseriti 7. Il sistema aggiorna il fornitore nel database 8. Il sistema chiude il dialog e aggiorna l'elenco fornitori
Svolgimenti alternativi	**1a.** Nessun fornitore selezionato: **1a.1** Il pulsante "Modifica" rimane disabilitato **1a.2** L'utente deve selezionare un fornitore per abilitare la modifica **6a.** Dati non validi: **6a.1** Il sistema evidenzia i campi con errori di validazione **6a.2** L'utente corregge i dati errati **6a.3** L'utente clicca nuovamente "Salva" **5a.** L'utente annulla le modifiche: **5a.1** L'utente clicca "Annulla" nel dialog **5a.2** Il sistema chiude il dialog senza salvare le modifiche

2.2.4 UC1.3 - Eliminare Fornitore

UC1.3	**Eliminare Fornitore**
Livello	User Goal
Descrizione	L'utente rimuove un fornitore non più necessario per mantenere l'anagrafica del sistema pulita e aggiornata
Attori	Utente
Pre-condizioni	L'utente si trova nella sezione Fornitoro, esiste almeno un fornitore nel sistema e l'utente ha selezionato un fornitore dall'elenco
Post-condizioni	Il fornitore selezionato è rimosso permanentemente dal sistema e l'elenco fornitori riflette l'avvenuta rimozione
Normale svolgimento	1. L'utente seleziona un fornitore dall'elenco dei fornitori 2. L'utente clicca sul pulsante "Elimina" 3. Il sistema mostra un dialog di conferma 4. L'utente clicca "Conferma" 5. Il sistema verifica i vincoli di integrità (es. che il fornitore non sia associato a piante esistenti) 6. Il sistema elimina il fornitore dal database 7. Il sistema chiude il dialog e aggiorna l'elenco fornitori
Svolgimenti alternativi	**1a.** Nessun fornitore selezionato: **1a.1** Il pulsante "Elimina" rimane disabilitato **1a.2** L'utente deve selezionare un fornitore per abilitare l'eliminazione **4a.** L'utente annulla l'eliminazione: **4a.1** L'utente clicca "Annulla" nel dialog di conferma **4a.2** Il sistema chiude il dialog senza eliminare il fornitore **5a.** Il fornitore è referenziato (vincolo di integrità): **5a.1** Il sistema rileva che il fornitore è associato a delle piante **5a.2** Il sistema mostra un dialog di errore **5a.3** Il sistema chiude il dialog di conferma e l'eliminazione è annullata

2.2.5 UC1.4 - Filtrare Fornitori

UC1.4	**Filtrare Fornitori**
Livello	User Goal
Descrizione	L'utente utilizza i filtri per cercare e visualizzare un sottoinsieme di fornitori in base a criteri specifici, facilitando la consultazione
Attori	Utente
Pre-condizioni	L'utente si trova nella sezione Fornitori ed esistono fornitori nel sistema da poter filtrare
Post-condizioni	L'elenco dei fornitori mostra solo i record che corrispondono ai criteri di ricerca e i filtri applicati sono resi visibili all'utente
Normale svolgimento	1. L'utente inserisce i criteri di ricerca in uno o più campi appositi (nome, città) 2. L'utente clicca sul pulsante "Applica Filtri" (o la ricerca si aggiorna automaticamente) 3. Il sistema interroga il database con i criteri forniti 4. Il sistema aggiorna l'elenco mostrando solo i fornitori che soddisfano i criteri 5. L'utente può cliccare "Resetta Filtri" per tornare alla visualizzazione completa
Svolgimenti alternativi	**3a.** Nessun risultato trovato: **3a.1** L'elenco dei fornitori viene mostrato vuoto **3a.2** L'utente può modificare i criteri di ricerca e riprovare

2.2.6 UC2 - Gestire Piantagioni

UC2	**Gestire Piantagioni**
Livello	Summary
Descrizione	L'utente gestisce le piantagioni attive per organizzare e monitorare le coltivazioni
Attori	Utente
Pre-condizioni	Il sistema è avviato e operativo, il database è accessibile, esistono zone e piante nel sistema
Post-condizioni	Le informazioni sulle piantagioni sono aggiornate nel sistema secondo le operazioni richieste
Normale svolgimento	1. L'utente accede alla sezione Piantagioni dal menu principale 2. Il sistema visualizza l'elenco delle piantagioni esistenti in una tabella 3. L'utente può selezionare una piantagione dall'elenco 4. L'utente può scegliere una delle seguenti operazioni: • Creare una nuova piantagione (UC2.1) • Modificare una piantagione esistente (UC2.2) • Eliminare una piantagione (UC2.3) • Cambiare lo stato di una piantagione (UC2.4) • Applicare filtri di ricerca per zona, pianta o periodo (UC2.5) 5. Il sistema esegue l'operazione richiesta 6. Il sistema aggiorna l'elenco delle piantagioni
Svolgimenti alternativi	**2a.** Errore di connessione al database: **2a.1** Il sistema mostra un dialog di errore connessione al database **2a.2** L'utente può riprovare l'operazione

2.2.7 UC2.4 - Cambiare Stato Piantagione

UC2.4	**Cambiare Stato Piantagione**
Livello	User Goal
Descrizione	L'utente aggiorna lo stato del ciclo di vita di una piantagione per tracciare il progresso della coltivazione attraverso l'interfaccia dell'applicativo
Attori	Utente
Pre-condizioni	L'utente si trova nella sezione Piantagioni, esiste almeno una piantagione nel sistema, e l'utente ha selezionato una piantagione dall'elenco
Post-condizioni	Lo stato della piantagione è aggiornato con il nuovo stato del ciclo di vita, l'elenco piantagioni riflette il cambiamento, e la data di cambio stato è registrata nel sistema
Normale svolgimento	1. L'utente seleziona una piantagione dall'elenco delle piantagioni 2. L'utente clicca sul pulsante "Cambia Stato" 3. Il sistema apre il dialog di cambio stato 4. Il sistema mostra lo stato attuale della piantagione e gli stati disponibili per la transizione 5. L'utente seleziona il nuovo stato dal menu a tendina 6. L'utente può aggiungere note opzionali nel campo commenti 7. L'utente clicca "Conferma" per applicare il cambio 8. Il sistema aggiorna lo stato della piantagione e registra la data del cambio 9. Il sistema chiude il dialog e aggiorna l'elenco piantagioni
Svolgimenti alternativi	**7a.** L'utente annulla il cambio stato: **7a.1** L'utente clicca "Annulla" nel dialog **7a.2** Il sistema chiude il dialog senza modificare lo stato della piantagione **1a.** Nessuna piantagione selezionata: **1a.1** Il pulsante "Cambia Stato" rimane disabilitato **1a.2** L'utente deve selezionare una piantagione per abilitare il cambio stato

2.2.8 UC3 - Elaborare Dati e Report

UC3	**Elaborare Dati e Report**
Livello	User Goal
Descrizione	L'utente analizza i dati di produzione e genera report per valutare le performance delle attività agricole
Attori	Utente
Pre-condizioni	Il sistema è avviato e operativo, il database è accessibile, esistono dati di raccolti nel sistema
Post-condizioni	I report richiesti sono generati e visualizzati, i dati possono essere esportati se necessario
Normale svolgimento	1. L'utente accede alla sezione Analisi e Report dal menu principale 2. Il sistema visualizza le opzioni di elaborazione disponibili 3. L'utente può scegliere una delle seguenti operazioni: • Generare report raccolti • Calcolare statistiche produzione • Visualizzare analisi dati con strategie 4. Il sistema elabora i dati secondo l'opzione selezionata 5. Il sistema visualizza i risultati dell'elaborazione 6. L'utente può salvare i risultati
Svolgimenti alternativi	**3a.** Nessun dato disponibile per elaborazione: **3a.1** Il sistema mostra messaggio informativo "Nessun dato disponibile per l'elaborazione" **3a.2** L'utente viene indirizzato alla gestione dati base (fornitori, piante, piantagioni)

2.2.9 UC4 - Visualizzare Dashboard

UC4	**Visualizzare Dashboard**
Livello	User Goal
Descrizione	L'utente ottiene una vista d'insieme del sistema e dello stato delle attività agricole tramite una dashboard informativa
Attori	Utente
Pre-condizioni	Il sistema è avviato e l'utente si trova nella schermata principale dell'applicazione
Post-condizioni	La dashboard mostra informazioni aggiornate e l'utente ha una vista d'insieme dello stato del sistema
Normale svolgimento	1. L'utente apre l'applicazione AgroManager o clicca su "Dashboard" dal menu 2. Il sistema carica i dati di riepilogo dal database 3. Il sistema visualizza la dashboard con le seguenti informazioni: • Stato operativo del sistema (connessione database) • Numero totale di fornitori, zone, piante registrate • Statistiche delle piantagioni attive per stato • Statistiche dei raccolti recenti • Azioni rapide per accesso diretto alle funzionalità 4. L'utente può navigare verso sezioni specifiche cliccando sul menu della dashboard o utilizzare le azioni rapide
Svolgimenti alternativi	**2a.** Sistema non operativo (database non accessibile): **2a.1** La dashboard passa in modalità limitata **2a.2** Le azioni rapide vengono disabilitate **2a.3** Viene mostrato messaggio "Sistema in modalità limitata"

2.3 Mockups

In questa sezione vengono illustrate le schermate (screenshot) dell'implementazione finale della GUI (Graphical User Interface). Il codice e lo stile dell'interfaccia sono stati generati mediante Google Gemini e successivamente rifiniti con opportune modifiche.

I prompt utilizzati per generare ogni vista seguono una struttura comune. Per esempio, la view Fornitore è stata generata con un prompt simile al seguente (il prompt completo è riportato a scopo dimostrativo per illustrare la metodologia di lavoro):

- **Contesto del Progetto**: Sto sviluppando un'applicazione JavaFX chiamata **AgroManager** per la gestione agricola seguendo il pattern architetturale **MVC (Model-View-Controller)**. Il progetto utilizza:
  - **JavaFX** per l'interfaccia utente
  - Pattern **MVC** con separazione netta delle responsabilità
  - **DAO Pattern** per l'accesso ai dati
  - **Service Layer** per la business logic
  - **CSS styling** personalizzato per un'interfaccia moderna e professionale
- **Struttura del Progetto**:

```
src/main/java/
- DomainModel/          # Entita del dominio
- Controller/           # Controller MVC
- View/                 # Interfacce utente JavaFX
- BusinessLogic/        # Servizi e logica di business
- ORM/                  # Data Access Objects
```

- **Modello di Dominio - Fornitore**:

```java
public class Fornitore {
    private Integer id;
    private String nome;              // REQUIRED
    private String indirizzo;         // REQUIRED
    private String numeroTelefono;    // REQUIRED
    private String email;             // OPTIONAL
    private String partitaIva;        // OPTIONAL
    private LocalDateTime dataCreazione;
    private LocalDateTime dataAggiornamento;

    // Costruttori, getter e setter standard
}
```

- **Requisiti Funzionali**:

  FornitoreView (Vista Principale)

  - *Layout e Struttura:*
    - Header con titolo "Gestione Fornitori" e sottotitolo descrittivo
    - Sezione ricerca con filtri per nome e città
    - Barra delle azioni con pulsanti: Nuovo, Modifica, Elimina, Applica Filtri, Reset
    - Tabella con colonne: ID, Nome, Indirizzo, Telefono, Email, P.IVA
    - Design a card con ombreggiature e bordi arrotondati
  - *Funzionalità Richieste:*
    - Visualizzazione tabellare dei fornitori con selezione singola
    - Filtri di ricerca in tempo reale
    - Doppio click su riga per modifica rapida
    - Abilitazione/disabilitazione pulsanti basata sulla selezione
    - Conferma eliminazione con dialog personalizzato
    - Messaggi di feedback per operazioni **CRUD**
  - *Integrazione Controller:*

    - Metodi setter per handler degli eventi: `setOnNuovoFornitore()`, `setOnModificaFornitore()`, ecc.
    - Metodo `setFornitori()` per aggiornare i dati della tabella
    - Metodo `getFornitoreSelezionato()` per ottenere l'elemento selezionato
    - Metodo `getCriteriFiltro()` che restituisce un record con i filtri applicati

FornitoreDialog (Dialog Modale)

- *Layout e Struttura:*
    - Dialog modale con titolo dinamico ("Nuovo Fornitore" / "Modifica Fornitore")
    - Form con campi: Nome*, Indirizzo*, Telefono*, Email, Partita IVA
    - Campi obbligatori marcati con asterisco (*)
    - Pulsanti "Salva" e "Annulla" allineati a destra
    - Placeholder text informativi per ogni campo
- *Validazione e Comportamento:*
    - Validazione client-side per campi obbligatori
    - Validazione formato email se compilato
    - Popolamento automatico campi in modalità modifica
    - Flag `confermato` per verificare se l'utente ha salvato
    - Gestione errori con messaggi user-friendly
- *Integrazione Sistema:*
    - Costruttore che accetta `Fornitore` nullable (null = nuovo, oggetto = modifica)
    - Metodo `getFornitore()` per recuperare l'oggetto aggiornato
    - Metodo `isConfermato()` per verificare se salvato
    - Utilizzo di `NotificationHelper` per messaggi di errore

- **Pattern di Integrazione MVC**:

```
public class FornitoreController {
    private final FornitoreService fornitoreService;    // Business
    Logic
    private final FornitoreView fornitoreView;          // View

    // Event handlers che collegano View e Service:
    private void onNuovoFornitore() { /* Dialog + Service.aggiungi */
     }
    private void onModificaFornitore() { /* Dialog + Service.aggiorna
     */ }
    private void onEliminaFornitore() { /* Conferma + Service.elimina
     */ }
    private void onApplicaFiltri() { /* Service.getConFiltri */ }
}
```

- **Stili CSS da Utilizzare**: Il progetto utilizza un sistema di classi CSS predefinite:
    - `.main-container` - Container principale
    - `.styled-card` - Card con ombreggiature
    - `.card-title` - Titoli delle sezioni
    - `.main-title`, `.subtitle` - Header principale
    - `.btn-primary`, `.btn-secondary`, `.btn-danger` - Pulsanti stilizzati
    - `.text-field-standard` - Campi di input
    - `.field-label` - Etichette campi
    - `.input-grid` - Griglia per form
    - `.v-separator` - Separatori verticali

- **Pattern di Gestione Errori**:

  - Utilizzare `NotificationHelper.showError()` per errori
  - Utilizzare `NotificationHelper.showSuccess()` per conferme
  - Utilizzare `NotificationHelper.showWarning()` per avvisi
  - Gestire eccezioni specifiche: `ValidationException`, `DataAccessException`, `BusinessLogicException`

- **Richiesta Specifica**: Genera il codice completo per:

  - `FornitoreView.java` - Vista principale con tabella, filtri e azioni
  - `FornitoreDialog.java` - Dialog modale per **CRUD**
  - Eventuali aggiornamenti `CSS` per stili specifici non presenti

  Rispetta rigorosamente:

  - Pattern MVC con separazione delle responsabilità
  - Naming conventions Java standard
  - Struttura del progetto esistente
  - Stili CSS predefiniti
  - Gestione eventi attraverso handler/callback
  - Validazione robusta dell'input utente
  - User experience fluida e intuitiva

  Il codice deve essere:

  - Production-ready con gestione errori completa
  - Ben commentato e auto-documentante
  - Responsive e accessibile
  - Coerente con lo stile del progetto esistente

2.3.1 Dashboard

Figura 4: Mockup della Dashboard principale, punto di accesso dell'applicazione. Mostra le statistiche di riepilogo (Zone, Fornitori, Piante, ecc.) e i collegamenti alle "Azioni Rapide".

2.3.2 Gestione Zone

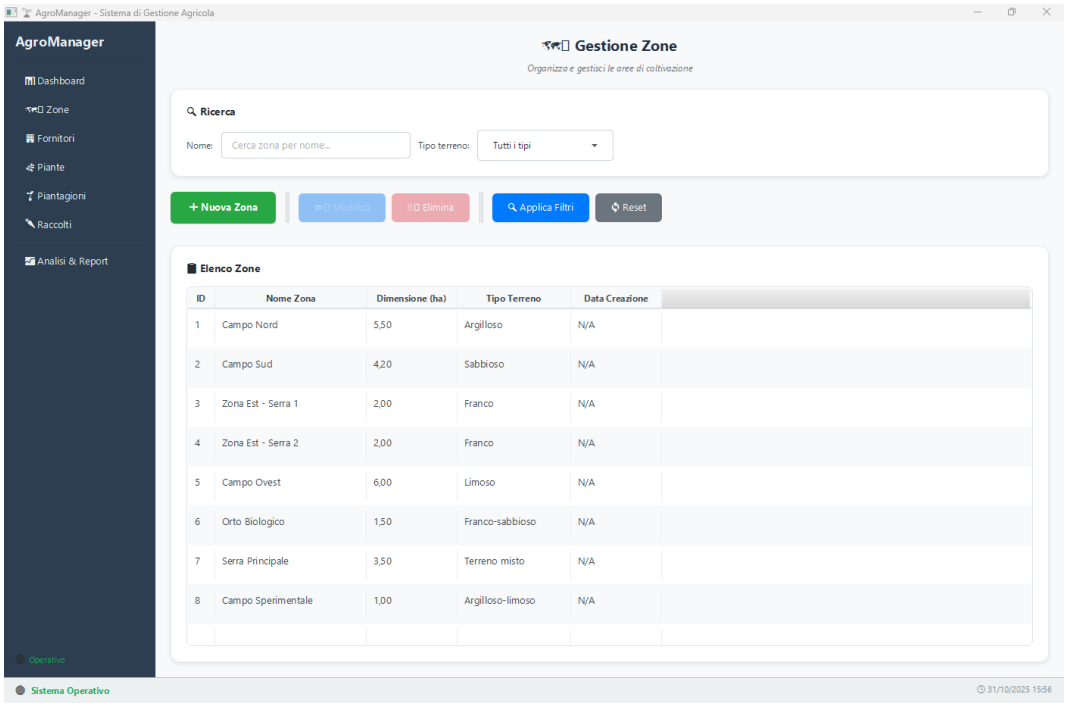

Figura 5: Mockup della vista ”Gestione Zone”. Illustra l’interfaccia **CRUD** (Create, Read, Update, Delete) standard, composta da un pannello di ”Ricerca”, i pulsanti di azione (Nuova Zona, Modifica, ecc.) e la tabella ”Elenco Zone”.

2.3.3 Gestione Fornitori

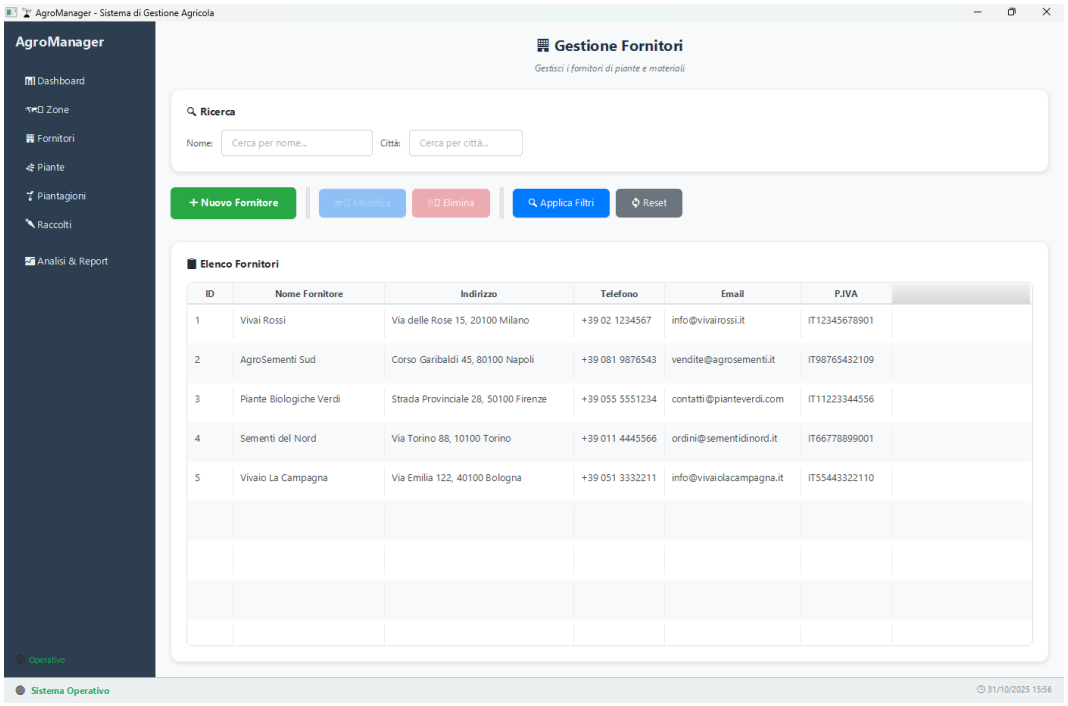

Figura 6: Mockup della vista ”Gestione Fornitori”. Illustra l’interfaccia **CRUD** (Create, Read, Update, Delete) standard, composta da un pannello di ”Ricerca”, i pulsanti di azione (Nuovo Fornitore, Modifica, ecc.) e la tabella ”Elenco Fornitori”.

2.3.4 Gestione Piante

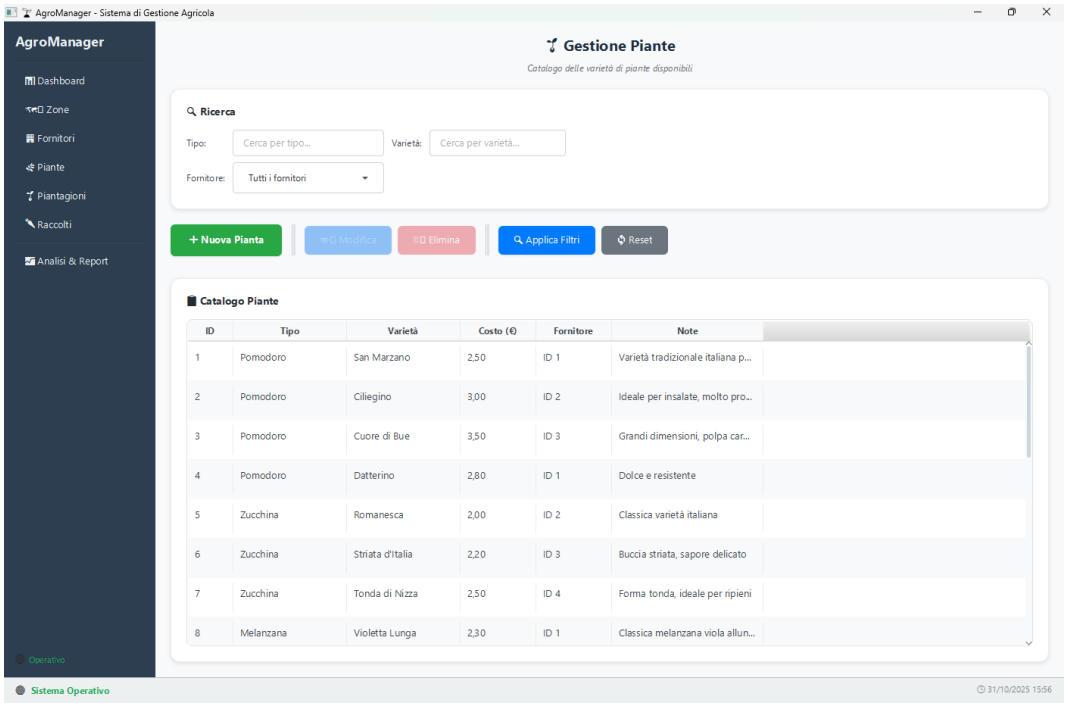

Figura 7: Mockup della vista ”Gestione Piante”. Illustra l’interfaccia **CRUD** (Create, Read, Update, Delete) standard, composta da un pannello di ”Ricerca”, i pulsanti di azione (Nuova Pianta, Modifica, ecc.) e la tabella ”Elenco Piante”.

2.3.5 Gestione Piantagioni

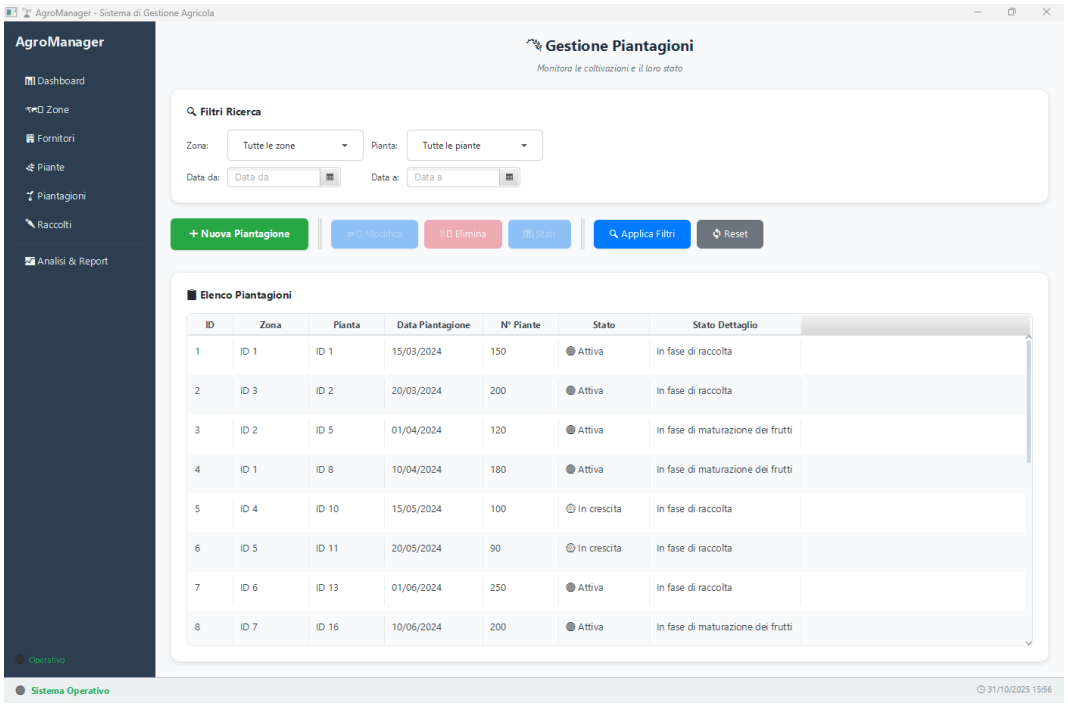

Figura 8: Mockup della vista ”Gestione Piantagioni”. Illustra l’interfaccia **CRUD** (Create, Read, Update, Delete) standard, composta da un pannello di ”Ricerca”, i pulsanti di azione (Nuova Piantagione, Modifica, ecc.) e la tabella ”Elenco Piantagioni”.

2.3.6 Gestione Raccolti

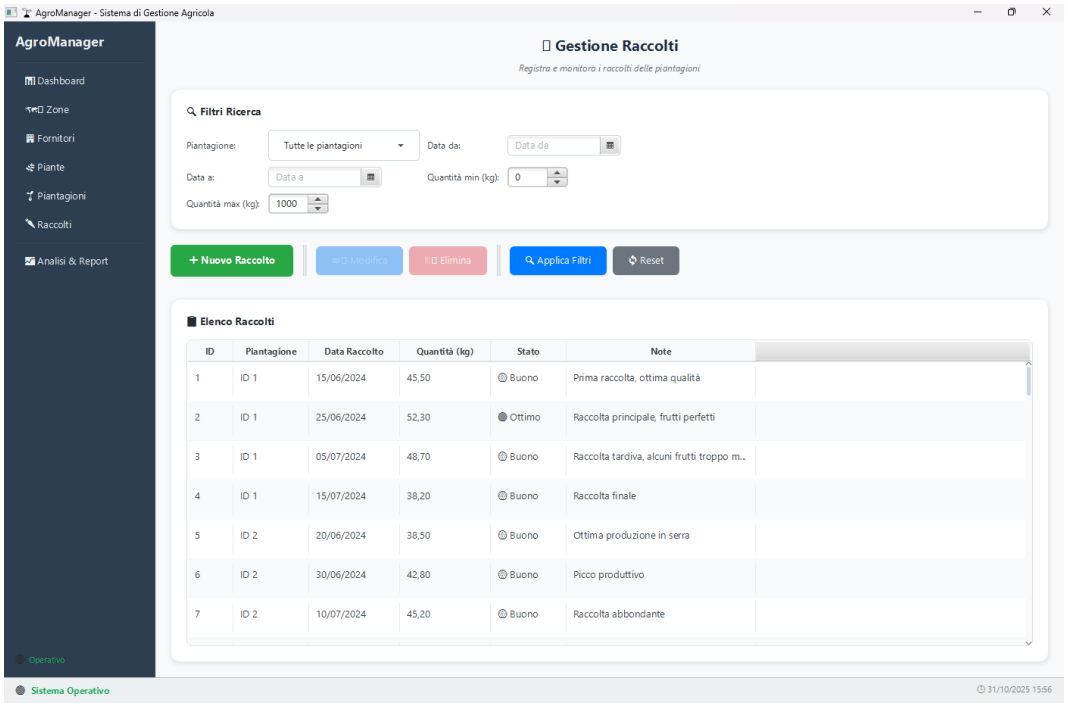

Figura 9: Mockup della vista ”Gestione Raccolti”. Illustra l’interfaccia CRUD (Create, Read, Update, Delete) standard, composta da un pannello di ”Ricerca”, i pulsanti di azione (Nuovo Raccolto, Modifica, ecc.) e la tabella ”Elenco Raccolti”.

2.3.7 Analisi e Report

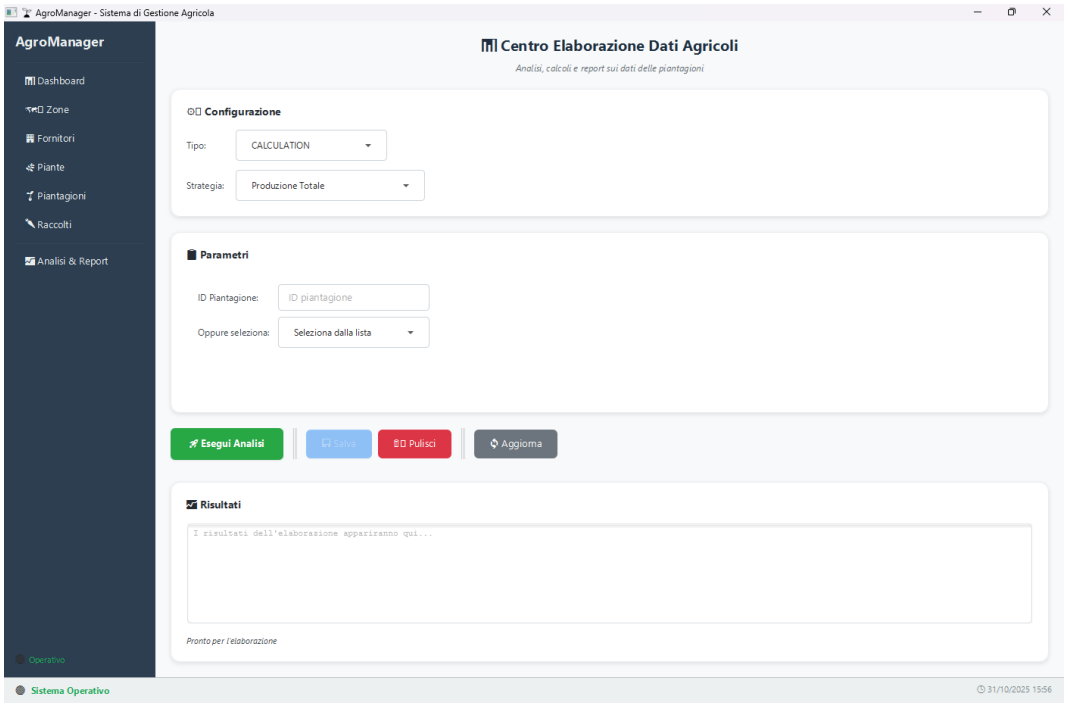

Figura 10: Mockup della vista 'Analisi e Report'. L'interfaccia permette all'utente di configurare ed eseguire analisi sui dati.

2.3.8 Dashbord azioni limitate

Figura 11: Mockup della vista ”Dashboard” con azioni limitate in seguito a una mancata connessione al database.

2.3.9 Dialogs inserimento, modifica e rimozione

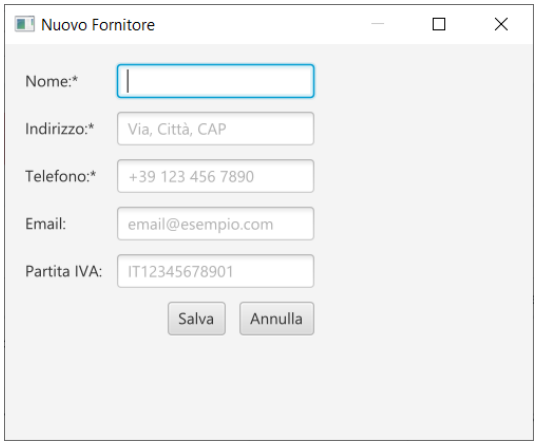

Figura 12: Dialog per l'inserimento di un nuovo fornitore.

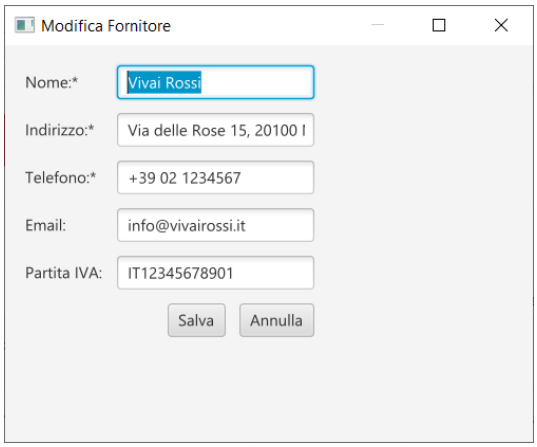

Figura 13: Dialog per la modifica di un fornitore esistente, i campi vengono pre compilati da sistema con i dati del fornitore da modificare.

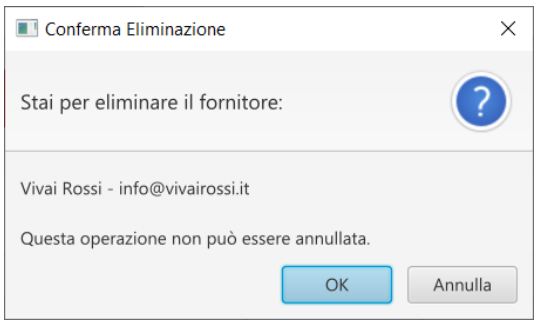

Figura 14: Dialog per la conferma della rimozione di un fornitore.

2.3.10 Dialog cambio stato piantagione

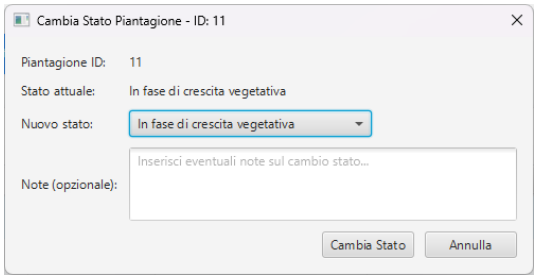

Figura 15: Dialog per il cambio di stato di una piantagione.

2.3.11 Dialogs errori

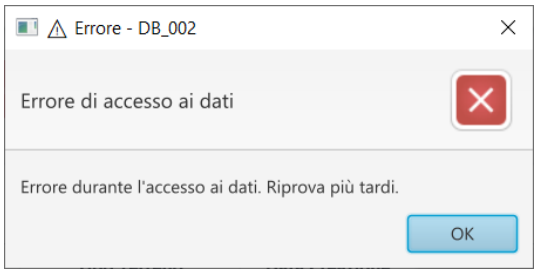

Figura 16: Mockup del dialog di errore connessione al database.

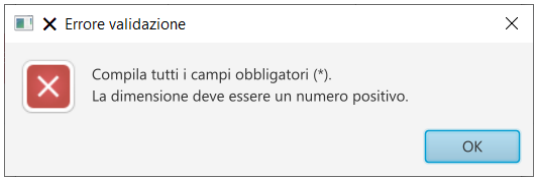

Figura 17: Mockup del dialog di errore di validazione dei dati.

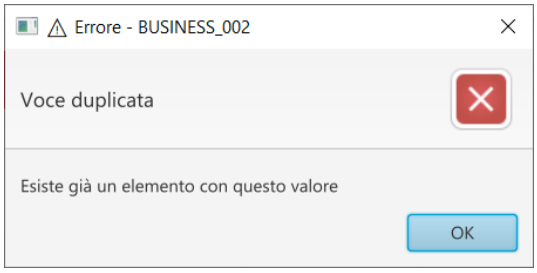

Figura 18: Mockup del dialog di errore in caso di aggiunta di un elemento già esistente.

2.4 Page navigation diagram

Il diagramma di navigazione delle pagine (Page Navigation Diagram) fornisce una visione d'insieme dei flussi di interazione dell'utente con l'interfaccia grafica. Questo diagramma definisce come l'utente si sposta tra le diverse viste dell'applicazione, validando la coerenza dei percorsi definiti negli Use Case e visualizzati nei Mockups.

Come illustrato nella Figura 19, il diagramma evidenzia l'architettura di navigazione "hub-and-spoke" dell'applicativo:

- La **Dashboard** agisce come "hub" centrale, da cui l'utente può accedere a tutte le sezioni principali ("spokes").
- Ciascuna sezione gestionale (es. "Gestione Zone", "Gestione Piantagioni") segue un pattern di interazione coerente, permettendo all'utente di tornare facilmente alla Dashboard.
- Le operazioni CRUD (come "new/edit") vengono gestite tramite finestre di dialogo (es. "Nuovo Fornitore Dialog"), che al termine ("save/cancel") riportano l'utente alla vista gestionale di provenienza.

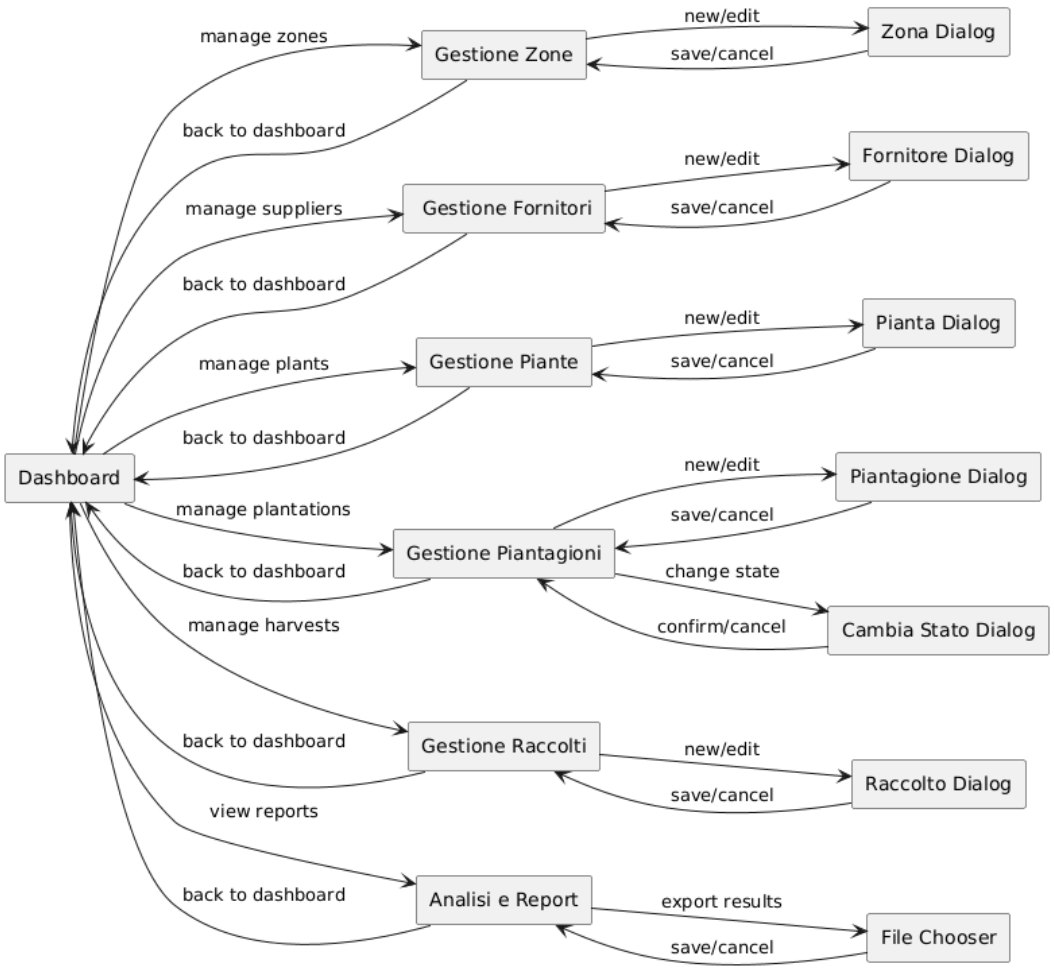

Figura 19: Diagramma di navigazione delle pagine (Page Navigation Diagram), che mostra i flussi utente tra la Dashboard principale e le varie sezioni gestionali dell'applicativo.

3 Progettazione e implementazione

3.1 Architettura e Package Diagram

L'architettura del software è stata progettata seguendo il principio di separazione delle responsabilità e modularità. Questo approccio facilita la manutenzione e l'estensibilità del sistema.

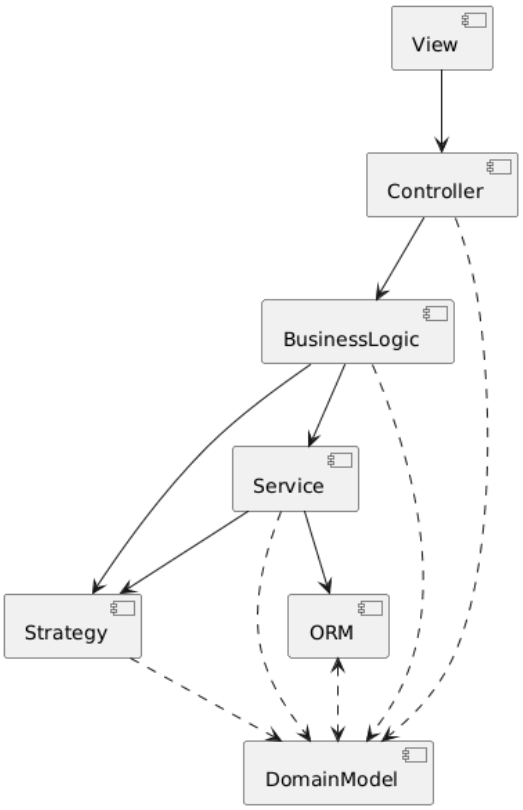

Figura 20: Diagramma raffigurante l'architettura e relazioni tra i package, la relazione tratteggiata (cross-cutting) indica che certe interazioni possono provocare cambiamenti anche in altri package. Il package Exception non è stato aggiunto al diagramma perché viene utilizzato da tutti ed è un'estensione delle eccezioni già esistenti in Java.

Come mostrato in Figura 20, l'architettura è suddivisa nei seguenti package:

- **View:** Rappresenta l'interfaccia utente (GUI) del sistema, gestendo la presentazione dei dati e l'interazione con l'utente.
- **Controller:** Agisce da mediatore tra la `View` e i layer sottostanti. Riceve gli eventi dall'interfaccia utente e invoca le operazioni appropriate (parte dell'architettura MVC).
- **BusinessLogic:** Contiene la logica di business principale e orchestra le operazioni. Coordina i `Service` e le `Strategy` per eseguire le funzionalità richieste.
- **Service:** Contiene classi specializzate (es. `FornitoreService`) che implementano la logica di business specifica per una singola entità o funzionalità.
- **Strategy:** Contiene le implementazioni di algoritmi specifici (es. le strategie di calcolo per i report), seguendo lo Strategy Pattern.
- **ORM (Object-Relational Mapping):** Gestisce la mappatura e la comunicazione tra le classi del `DomainModel` e le tabelle del database (tramite i DAO).
- **DomainModel:** Contiene le classi POJO (Plain Old Java Objects) che rappresentano le entità principali del sistema (es. `Fornitore`, `Piantagione`).

3.2 Class Diagram per package

Di seguito sono riportati i diagrammi delle classi per ciascun package descritto nella sezione precedente. Questi diagrammi illustrano la struttura interna di ogni componente architetturale.

3.2.1 DomainModel

Il package DomainModel definisce le entità fondamentali del sistema (POJO). Come mostrato in Figura 21, esso include le classi Fornitore, Pianta, Zona, Piantagione, Raccolto e StatoPiantagione, che rappresentano i dati gestiti dall'applicazione.

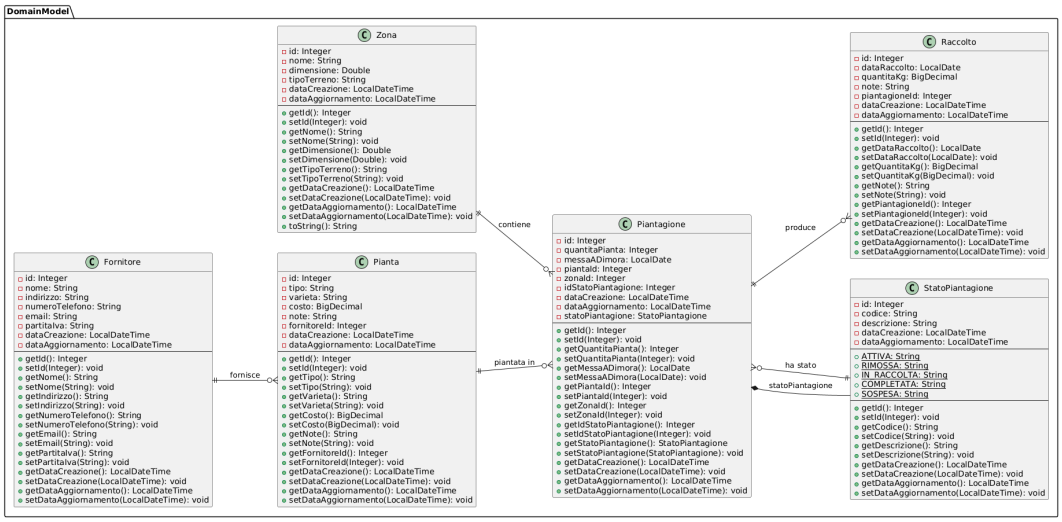

Figura 21: Diagramma delle classi del package DomainModel

3.2.2 ORM

Il package ORM implementa il Data Access Object (DAO) pattern per l'accesso ai dati. Come visibile in Figura 22, la sua struttura è composta da:

- Una classe astratta BaseDAO che implementa il Template Method Pattern per le operazioni CRUD comuni
- DAO concreti (es. FornitoreDAO, PiantagioneDAO) per ogni entità, che ereditano da BaseDAO
- Una DAOFactory (Singleton) per centralizzare la creazione dei DAO
- La classe DatabaseConnection (Singleton) che gestisce la connessione JDBC

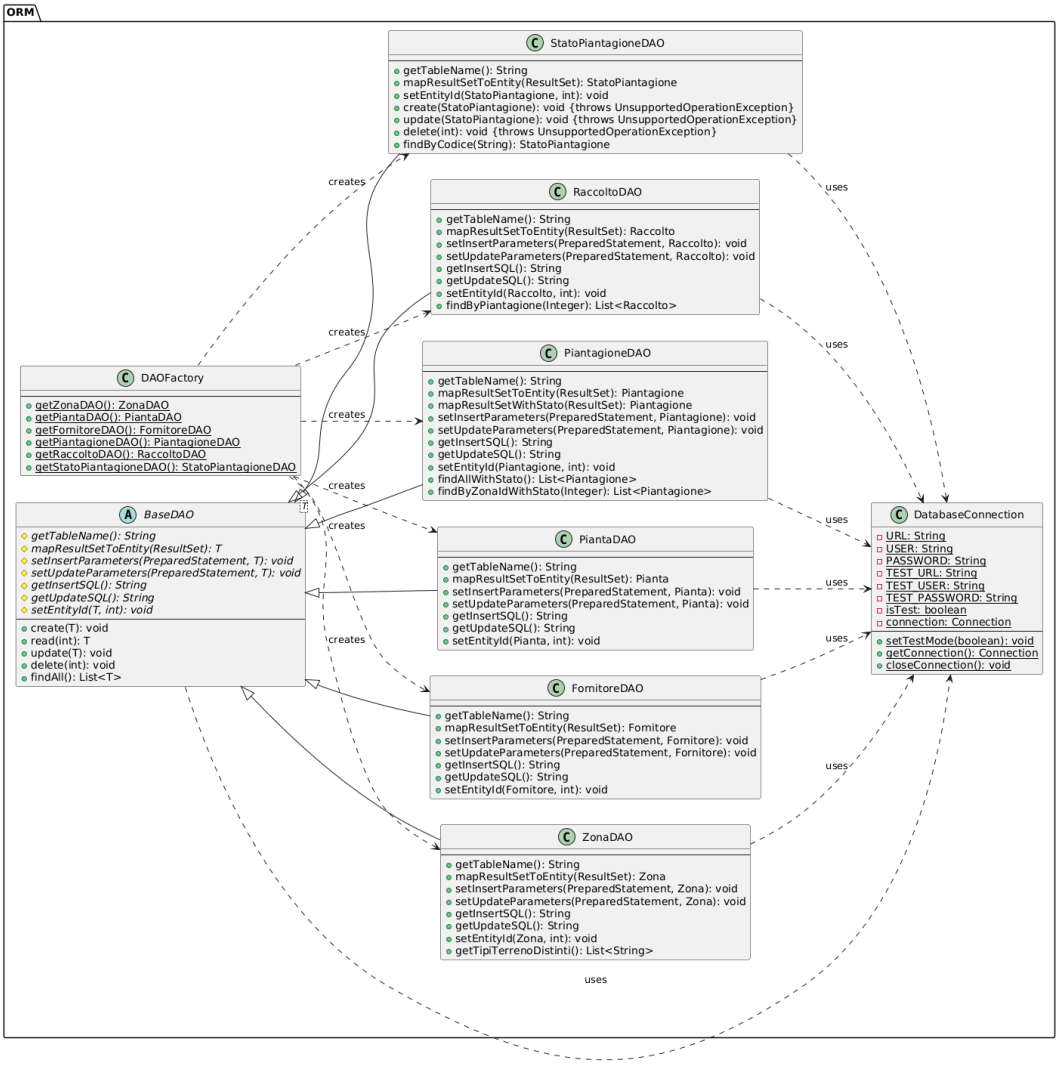

Figura 22: Diagramma delle classi del package ORM

3.2.3 BusinessLogic

Il package BusinessLogic (Figura 23) agisce come Facade per l'intero layer di business. Espone i metodi principali (es. eseguiStrategiaConDati) e coordina le funzionalità dei package Service, Strategy ed Exception sottostanti.

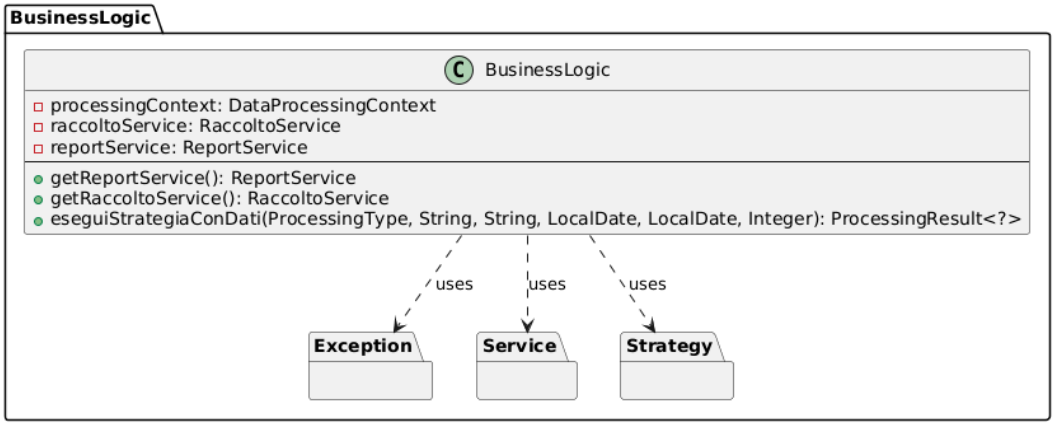

Figura 23: Diagramma delle classi del package BusinessLogic

3.2.4 BusinessLogic - Service

Il package Service (Figura 24) contiene la logica di business specifica. Include un service per ogni entità principale (es. PiantagioneService, RaccoltoService, FornitoreService) che gestisce la validazione e la logica operativa. Include inoltre un ErrorService centralizzato per la gestione delle notifiche e delle eccezioni.

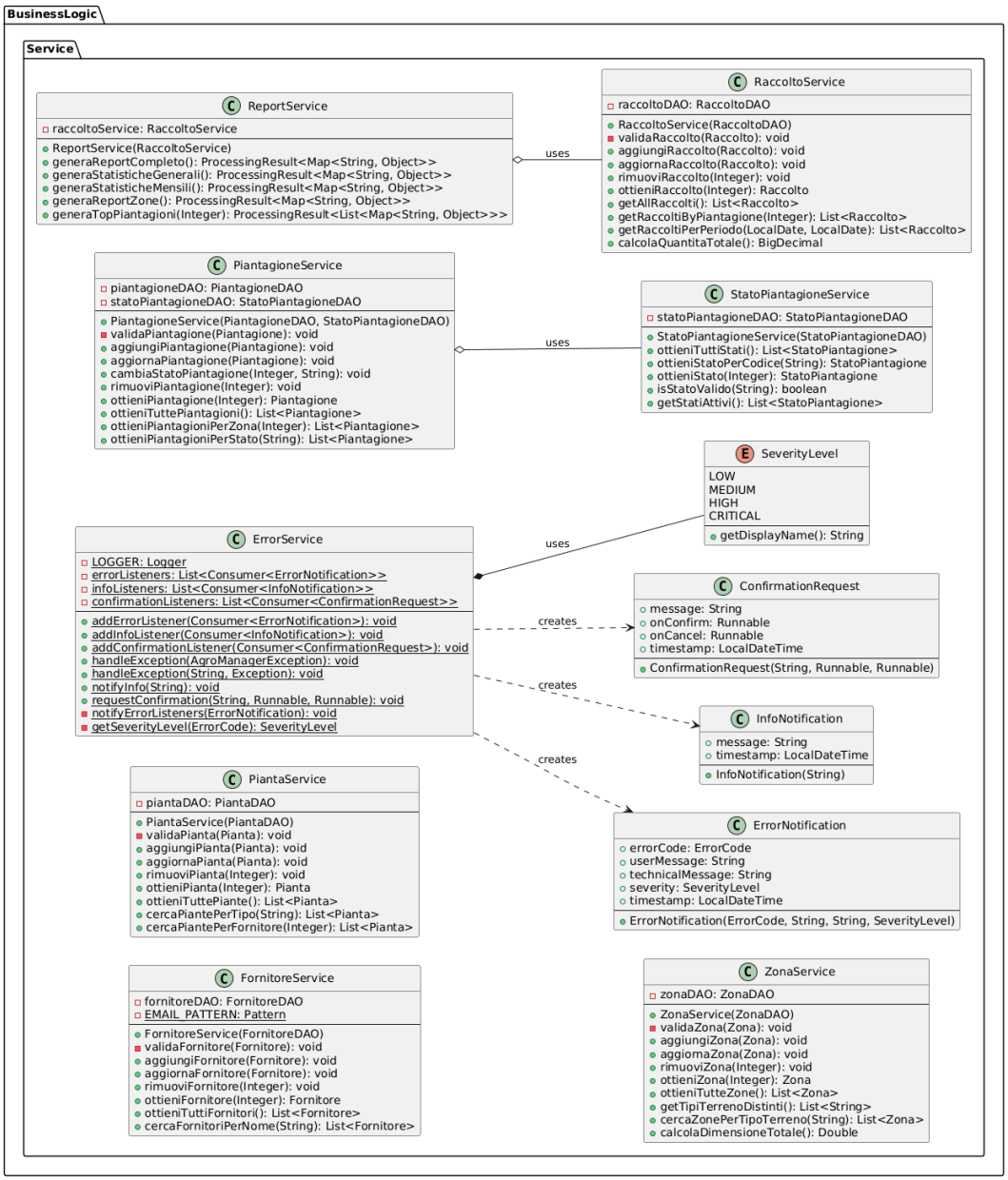

Figura 24: Diagramma delle classi del package BusinessLogic-Service

3.2.5 BusinessLogic - Strategy

Questo package (Figura 25) implementa lo Strategy Pattern per l'elaborazione dei dati. Definisce:

- Un'interfaccia DataProcessingStrategy.
- Implementazioni concrete per Calcolo, Statistiche e Report (es. ProduzioneTotaleStrategy, ReportRaccoltiStrategy).
- Una StrategyFactory che gestisce la creazione e l'esecuzione della strategia appropriata.

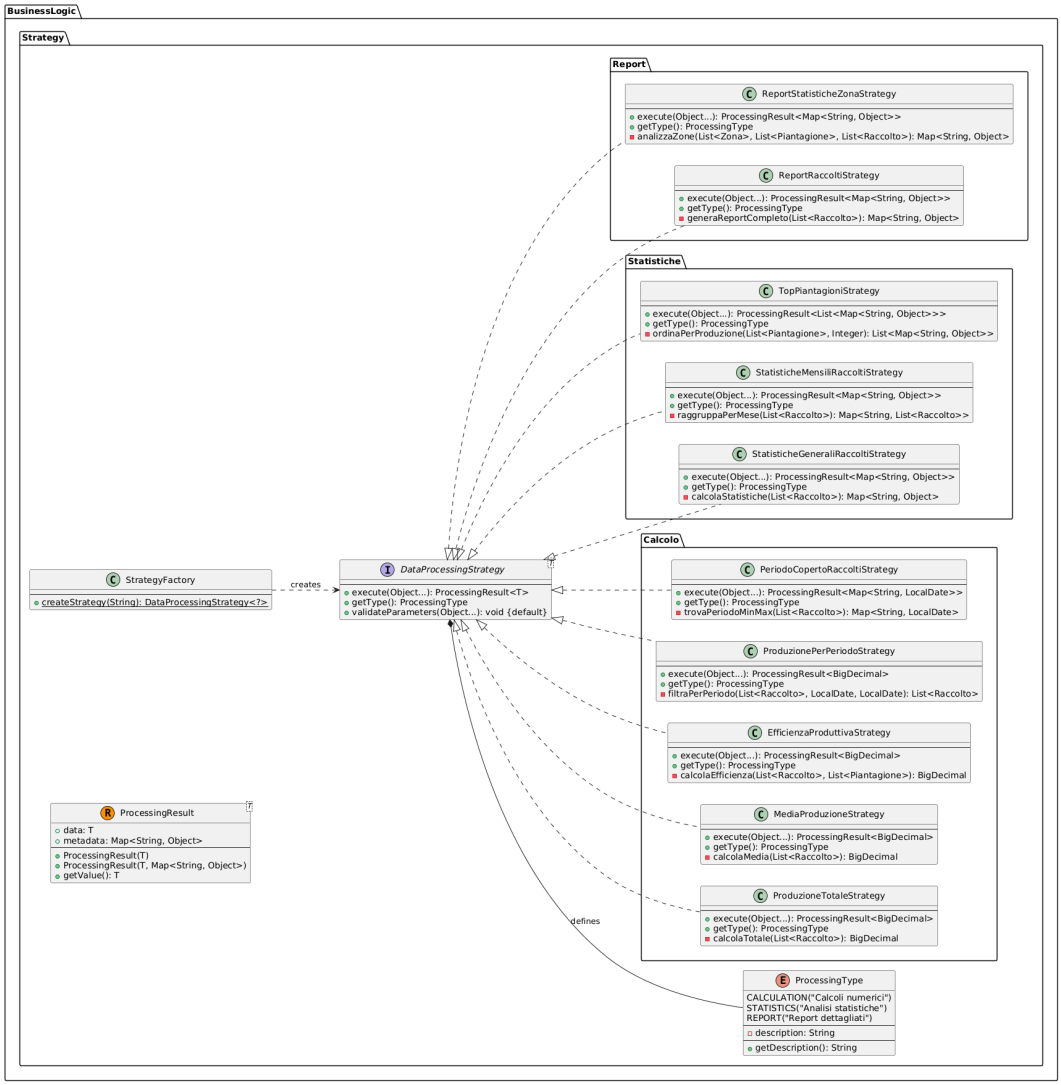

Figura 25: Diagramma delle classi del package BusinessLogic-Strategy

3.2.6 BusinessLogic - Exception

Il package Exception (Figura 26) definisce una gerarchia di eccezioni custom per una gestione degli errori robusta e centralizzata. Tutte le eccezioni ereditano da una classe base AgroManagerException, suddivisa in eccezioni tecniche (es. DataAccessException) e di business (es. ValidationException, BusinessLogicException). L'ErrorCode enum centralizza i codici e i messaggi d'errore.

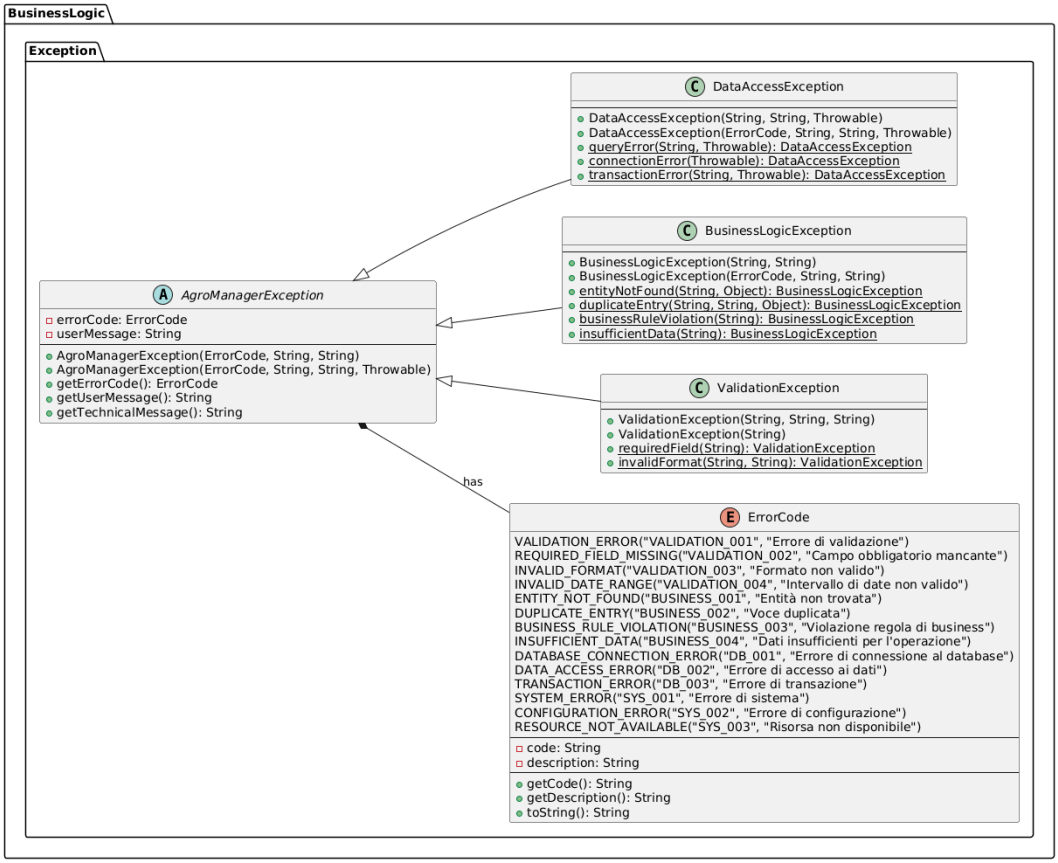

Figura 26: Diagramma delle classi del package BusinessLogic-Exception

3.2.7 Controller

Il package Controller (Figura 27) implementa la parte C del pattern MVC. Ad ogni vista principale è associato un controller (es. FornitoreController, PiantagioneController) che gestisce gli eventi della GUI (es. onNuovoFornitore()) e invoca i metodi del BusinessLogic layer (tramite i suoi Service) per eseguire le azioni richieste (implementazione del pattern Observer).

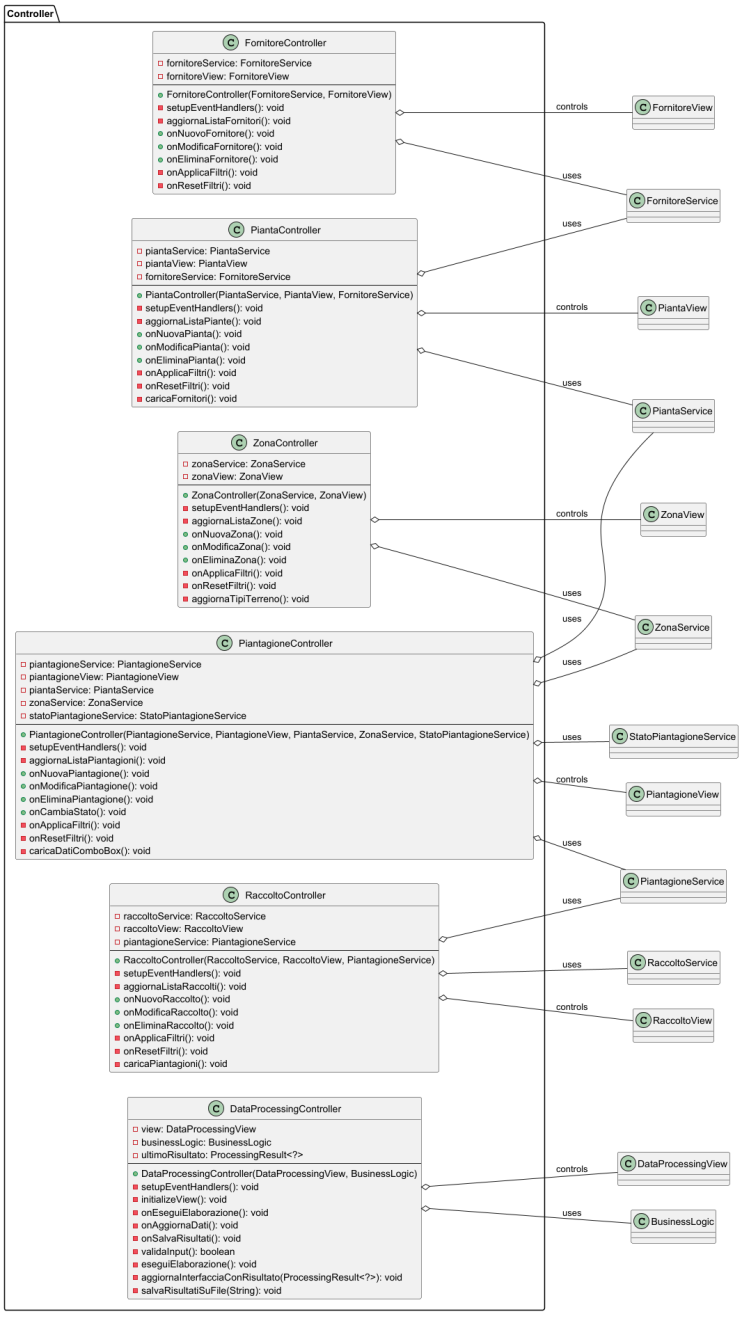

Figura 27: Diagramma delle classi del package Controller

3.2.8 View

Il package View (Figura 28) contiene tutte le classi JavaFX che compongono l'interfaccia utente. È suddiviso logicamente in Main Views (le schermate principali come DashboardView e FornitoreView), Dialogs (le finestre modali per l'inserimento/modifica, es. FornitoreDialog, PiantagioneDialog) e Helpers (classi di utilità come NotificationHelper).

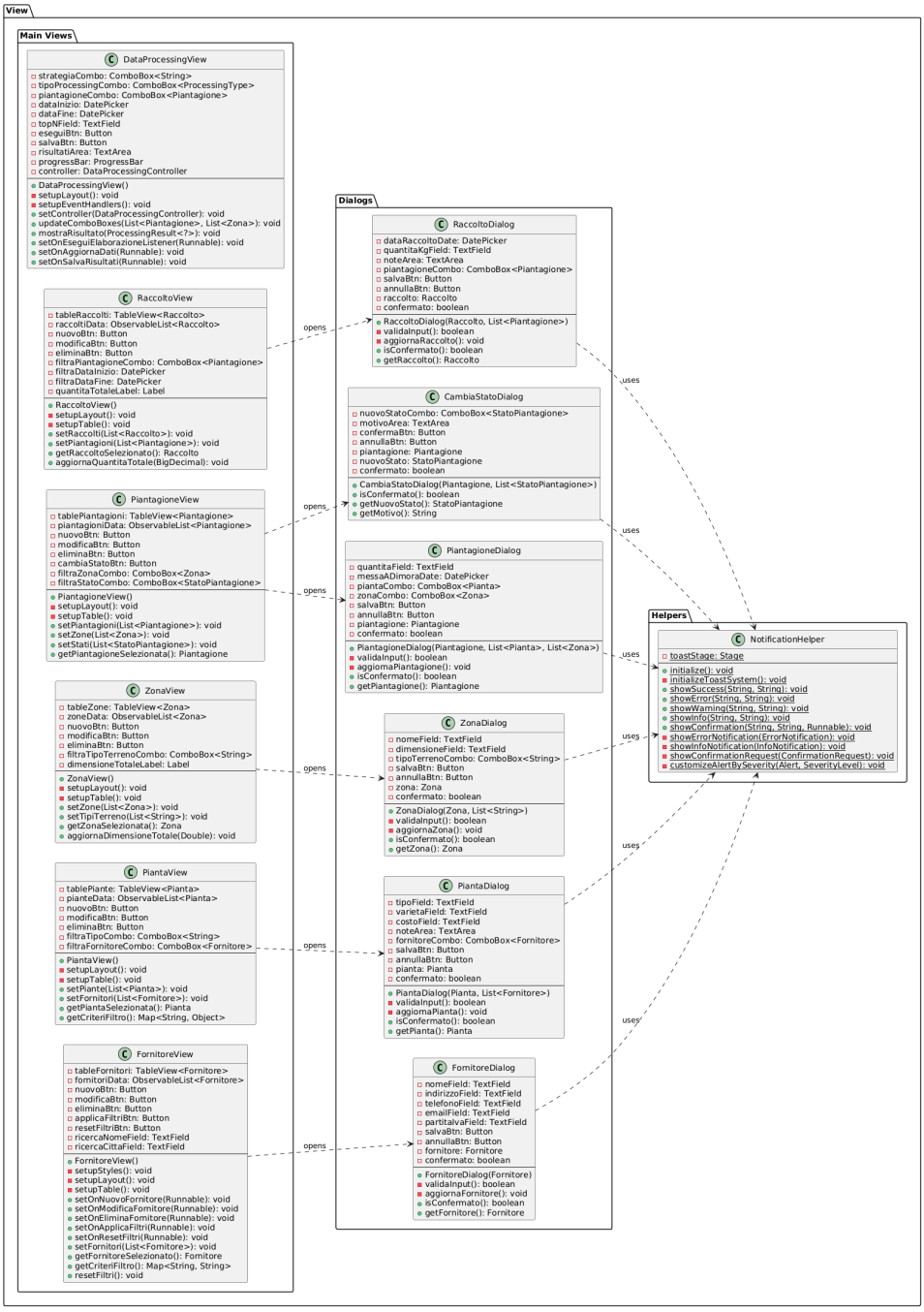

Figura 28: Diagramma delle classi del package View

3.3 Progettazione Database

La progettazione del database deriva direttamente dalle entità identificate nel DomainModel. La persistenza dei dati è gestita da un database relazionale PostgreSQL, implementata tramite il DAO Pattern. La Figura 29 mostra lo schema Entità-Relazione (E-R) del database, che evidenzia le tabelle principali e le loro associazioni.

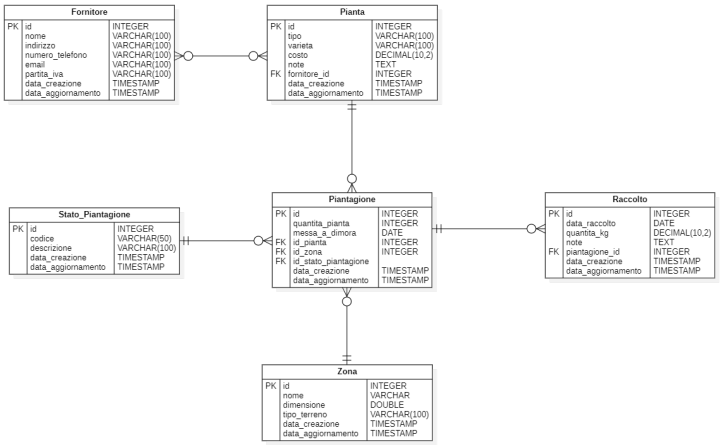

Figura 29: Schema E-R del database

Di seguito è riportato il modello relazionale testuale, che definisce in dettaglio gli attributi, i tipi di dati e i vincoli di chiave per ciascuna tabella:

Fornitore (**id**, nome, indirizzo, numero_telefono, email, partita_iva, data_creazione, data_aggiornamento)

Pianta (**id**, tipo, varieta, costo, note, *fornitore_id*, data_creazione, data_aggiornamento)

Zona (**id**, nome, dimensione, tipo_terreno, data_creazione, data_aggiornamento)

Stato_Piantagione (**id**, codice, descrizione, data_creazione, data_aggiornamento)

Piantagione (**id**, quantita_pianta, messa_a_dimora, *id_pianta*, *id_zona*, *id_stato_piantagione*, data_creazione, data_aggiornamento)

Raccolto (**id**, data_raccolto, quantita_kg, note, *piantagione_id*, data_creazione, data_aggiornamento)

4 Dettagli implementativi

In questa sezione verrà fornita una panoramica dettagliata dell'implementazione del progetto, con particolare attenzione alle responsabilità dei componenti principali, ai design patterns utilizzati e alla gestione degli errori.

4.1 Responsabilità dei componenti principali

4.1.1 MainApp (Entry Point)

La classe MainApp implementa il bootstrap per l'applicativo. Agisce come dependency injector manuale e lifecycle manager per tutti i componenti del sistema, garantendo l'ordine corretto di inizializzazione e la gestione degli stati di errore. MainApp implementa un sistema di dependency injection manuale che:

- Risolve le dipendenze in ordine topologico (DAO → Service → Controller).
- Gestisce il lifecycle completo dell'applicazione.
- Fornisce error handling centralizzato per l'inizializzazione.
- Implementa il pattern fail-fast per errori critici.

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle(" AgroManager - Sistema di Gestione Agricola");
    primaryStage.setMaximized(true);
    NotificationHelper.initialize();
    showLoadingScreen(primaryStage);
    initializeApplicationAsync(primaryStage);
}
```

Listing 1: Snippet del metodo start() in MainApp.

La classe coordina l'inizializzazione di tutti i layer seguendo l'architettura **MVC** in modo asincrono per evitare Thread Blocking durante l'avvio dell'applicazione dovuto ai servizi database che bloccano l'Event Dispatch Thread di JavaFX, causando UI non responsive. Il metodo `start()` (Listing 1) avvia l'inizializzazione in un thread separato, mantenendo l'interfaccia utente reattiva.

```
private void initializeApplicationAsync(StageprimaryStage) {
    Task<Void> initTask = new Task<>() {
        @Override
        protected Void call() throws Exception {
            Platform.runLater(() -> statusLabel.setText("Inizializzazione
     servizi..."));
            try {
                initializeServices();
                systemReady = true;
                Platform.runLater(() -> statusLabel.setText("Sistema
    pronto "));
            } catch (Exception e) {...}
            Platform.runLater(() -> statusLabel.setText("Preparazione
    interfaccia..."));
            initializeViews();
            initializeControllers();
            Thread.sleep(500);
            return null;
        }
        @Override
        protected void succeeded() {
            Platform.runLater(() -> showMainInterface(primaryStage));
        }
        @Override
        protected void failed() {
            Platform.runLater(() -> showErrorScreen(primaryStage,
    getException()));
        }
```

```
25      };
26      new Thread(initTask).start();
27
28      // Altre parti del codice...
29  }
```

Listing 2: Snippet del metodo initializeApplicationAsync() in MainApp.

I layer sono eseguiti nel seguente ordine per rispettare le dipendenze:

- **Data Access Layer:** Vengono create le istanze dei DAO tramite la **DAOFactory**.
- **Service Layer:** Vengono istanziati i Service, iniettando i DAO necessari (es. new ZonaService(DAOFactory.getZonaDAO())).
- **Presentation Layer:** Vengono istanziati i Controller, iniettando i Service e le View corrispondenti (es. new ZonaController(zonaService, zonaView)).

4.1.2 Service Layer- Validazione e Business Logic

Il Service Layer implementa il business logic dell'architettura, separando le regole di dominio dalla persistenza e dalla presentazione. Ogni Service agisce come domain expert per la propria entità, implementando validation rules, business constraints e operazioni multi-entity. In particolare, i Service sono responsabili di:

- Incapsulare la logica di business complessa.
- Eseguire controlli stratificati di validazione.
- Mappare errori tecnici a errori business.

La validazione segue un approccio defense-in-depth con controlli incrementali:

- **Structural validation:** Null checks, required fields, format validation.
- **Semantic validation:** Business rules specifiche del dominio.
- **Integrity validation:** Cross-entity constraints e referential integrity.
- **Business validation:** Complex domain rules e policy enforcement.

```
1   public class FornitoreService {
2       private final FornitoreDAO fornitoreDAO;
3       private static final Pattern EMAIL_PATTERN =
4           Pattern.compile("^[A-Za-z0-9+_.-]+@(.+)$");
5
6       public FornitoreService(FornitoreDAO fornitoreDAO) {
7           this.fornitoreDAO = fornitoreDAO;
8       }
9
10      private void validaFornitore(Fornitore fornitore) throws
        ValidationException {
11          if (fornitore == null) {
12              throw new ValidationException("Fornitore non può essere null"
        );
13          }
14
15          if (fornitore.getNome() == null || fornitore.getNome().trim().
        isEmpty()) {
16              throw ValidationException.requiredField("Nome fornitore");
17          }
18
19          if (fornitore.getIndirizzo() == null || fornitore.getIndirizzo().
        trim().isEmpty()) {
20              throw ValidationException.requiredField("Indirizzo");
21          }
22
```

```
23          if (fornitore.getEmail() != null && !fornitore.getEmail().trim().
    isEmpty()
24                  && !EMAIL_PATTERN.matcher(fornitore.getEmail()).matches()) {
25              throw ValidationException.invalidFormat("Email", "
    formato@esempio.com");
26          }
27
28          if (fornitore.getNumeroTelefono() != null && !fornitore.
    getNumeroTelefono().trim().isEmpty()
29                  && fornitore.getNumeroTelefono().trim().length() < 8) {
30              throw new ValidationException("numeroTelefono", fornitore.
    getNumeroTelefono(),
31                      "deve contenere almeno 8 caratteri");
32          }
33      }
34
35      public void aggiungiFornitore(Fornitore fornitore) throws
    ValidationException, DataAccessException, BusinessLogicException {
36          validaFornitore(fornitore);
37
38          try {
39              List<Fornitore> esistenti = fornitoreDAO.findAll();
40              boolean duplicato = esistenti.stream()
41                      .anyMatch(f -> f.getNome().equalsIgnoreCase(fornitore
    .getNome()) ||
42                                      (f.getEmail() != null && fornitore.
    getEmail() != null &&
43                                       f.getEmail().equalsIgnoreCase(
    fornitore.getEmail())));
44
45              if (duplicato) {
46                  throw BusinessLogicException.duplicateEntry("Fornitore",
    "nome o email", fornitore.getNome());
47              }
48
49              fornitoreDAO.create(fornitore);
50          } catch (SQLException e) {
51              throw DataAccessException.queryError("inserimento fornitore",
     e);
52          }
53      }
54
55      // Altri metodi di servizio (aggiornaFornitore, rimuoviFornitore,
    getFornitoreById, getAllFornitori)
56  }
```

Listing 3: Snippet del metodo di validazione in FornitoreService.

4.2 Design Patterns utilizzati

Nella realizzazione del progetto sono stati adottati diversi design patterns per migliorare la manutenibilità, la scalabilità e la leggibilità del codice. Di seguito sono descritti i principali pattern utilizzati insieme a esempi di codice.

4.2.1 Model-View-Controller (MVC)

L'architettura **MVC** è stata implementata per separare le responsabilità tra la gestione dei dati (Model), la logica di presentazione (View) e il controllo del flusso dell'applicazione (Controller). Questo approccio consente una maggiore modularità e facilita la manutenzione del codice.

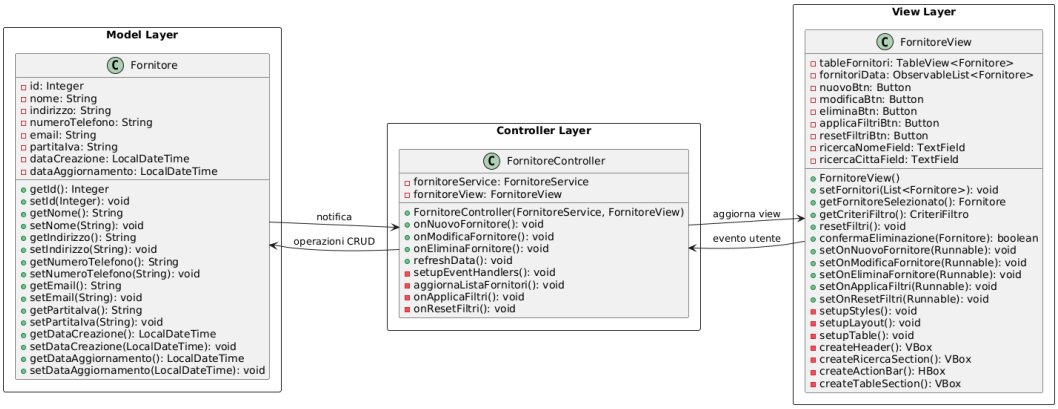

Figura 30: Architettura **Model-View-Controller (MVC)** dell'entità fornitore.

Le dipendenze tra i componenti sono gestite in MainApp, che funge da entry point dell'applicazione. I DAO vengono iniettati nei Service, che a loro volta vengono iniettati nei Controller insieme alle View corrispondenti.

```
1  private void initializeServices() {
2      fornitoreService = new FornitoreService(DAOFactory.getFornitoreDAO())
       ;
3      // Altre inizializzazioni dei services
4  }
5  private void initializeViews() {
6      fornitoreView = new FornitoreView();
7      // Altre inizializzazioni delle views
8  }
9  private void initializeControllers() {
10     fornitoreController = new FornitoreController(fornitoreService,
       fornitoreView);
11     // Altre inizializzazioni dei controllers
12 }
```

Listing 4: Snippet dell'inizializzazione dei componenti **MVC** in MainApp.

4.2.2 Data Access Object (DAO)

Il pattern DAO è stato utilizzato per astrarre l'accesso ai dati e separare la logica di persistenza dal resto dell'applicazione. Ogni entità del dominio ha un DAO dedicato che gestisce le operazioni CRUD e le query specifiche.

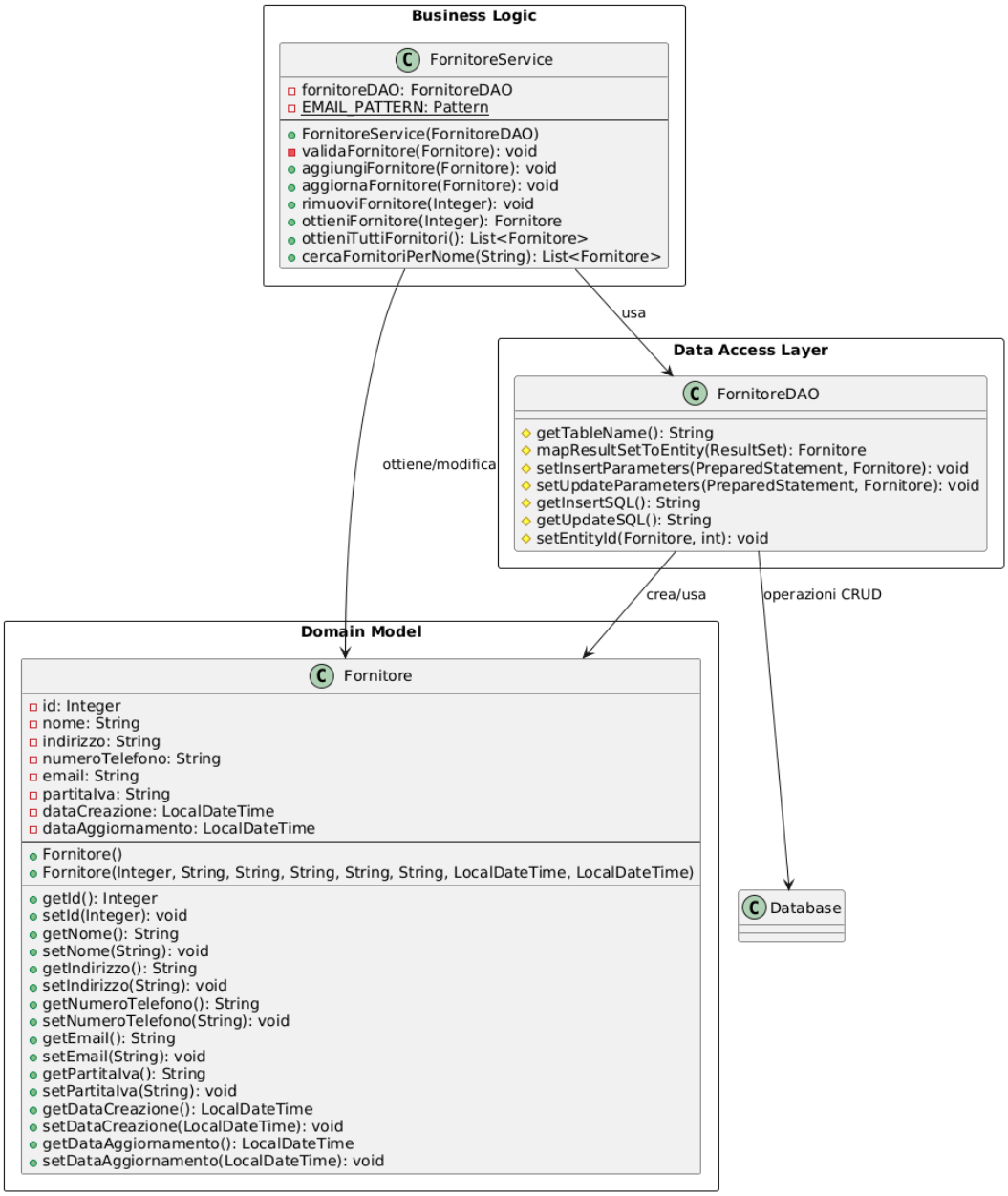

Figura 31: Struttura del pattern Data Access Object (DAO) per l'entità fornitore.

La logica business non ha accesso diretto al database, ma garantisce la persistenza dei dati tramite i DAO che gestiscono le comunicazioni con il database.

```
1 public void aggiungiFornitore(Fornitore fornitore)
      throwsValidationException, DataAccessException, BusinessLogicException
       {
2     validaFornitore(fornitore);
3     try {
4         List<Fornitore> esistenti = fornitoreDAO.findAll();
5         boolean duplicato = esistenti.stream()
6                 .anyMatch(f -> f.getNome().equalsIgnoreCase(fornitore.
      getNome()) ||
7                                 (f.getEmail() != null && fornitore.getEmail
      () != null &&
```

```
                                f.getEmail().equalsIgnoreCase(fornitore.
    getEmail())));
        if (duplicato) {
            throw BusinessLogicException.duplicateEntry("Fornitore", "
    nome o email", fornitore.getNome());
        }
        fornitoreDAO.create(fornitore);
    } catch (SQLException e) {
        throw DataAccessException.queryError("inserimento fornitore", e);
    }
}
```

Listing 5: Snippet della classe FornitoreService che garantisce la persistenza dei dati attraverso il FornitoreDAO.

4.2.3 Template Method

Il BaseDAO implementa il **Template Method Pattern** per standardizzare le operazioni CRUD eliminando code duplication tra DAO specifici. La classe definisce le operazioni database delegando solo i dettagli alle sottoclassi, cio permette una riduzione delle duplicazioni di codice del $N \times M$ dove N è il numero di entità e M il numero di operazioni CRUD. In questo modo, ogni DAO specifico (es. FornitoreDAO) eredita da BaseDAO e implementa solo i metodi astratti per le query specifiche, mentre le operazioni comuni (es. apertura/chiusura connessione, gestione transazioni) sono centralizzate in BaseDAO.

```
public abstract class BaseDAO<T> {
    protected abstract String getTableName();
    protected abstract T mapResultSetToEntity(ResultSet rs) throws
    SQLException;
    protected abstract void setInsertParameters(PreparedStatement stmt, T
     entity) throws SQLException;
    protected abstract void setUpdateParameters(PreparedStatement stmt, T
     entity) throws SQLException;
    protected abstract String getInsertSQL();
    protected abstract String getUpdateSQL();

    public void create(T entity) throws SQLException {...}

    public T read(int id) throws SQLException {...}

    public void update(T entity) throws SQLException {...}

    public void delete(int id) throws SQLException {...}

    public List<T> findAll() throws SQLException {...}

    protected abstract void setEntityId(T entity, int id);
}
```

Listing 6: Snippet della classe BaseDAO che implementa il Template Method Pattern.

Per esempio, il metodo `create(T entity)` (Listing 6) definisce il flusso generale per l'inserimento di un'entità nel database, mentre i dettagli specifici della query e del mapping sono delegati ai metodi astratti `getInsertSQL()`, `setInsertParameters()` e `setEntityId()` che devono essere implementati dalle sottoclassi (FornitoreDAO (Listing 7)).

```
public class FornitoreDAO extends BaseDAO<Fornitore> {
    @Override
    protected String getTableName() {
        return "fornitore";
    }

    @Override
    protected Fornitore mapResultSetToEntity(ResultSet rs) throws
    SQLException {...}

```

```
    @Override
    protected void setInsertParameters(PreparedStatement stmt, Fornitore
    fornitore) throws SQLException {...}

    @Override
    protected void setUpdateParameters(PreparedStatement stmt, Fornitore
    fornitore) throws SQLException {...}

    @Override
    protected String getInsertSQL() {
        return "INSERT INTO fornitore (nome, indirizzo, numero_telefono,
    email, partita_iva, data_creazione, data_aggiornamento) VALUES (?, ?,
    ?, ?, ?, ?, ?)";
    }

    @Override
    protected String getUpdateSQL() {
        return "UPDATE fornitore SET nome = ?, indirizzo = ?, numero_
    telefono = ?, email = ?, partita_iva = ?, data_aggiornamento = ? WHERE
     id = ?";
    }

    @Override
    protected void setEntityId(Fornitore fornitore, int id) {...}
```

Listing 7: Snippet della classe FornitoreDAO che estende BaseDAO.

4.2.4 Strategy

Il Strategy Pattern implementato per l'elaborazione dati agricoli fornisce un framework di computational algorithms intercambiabili. Il sistema consente di applicare diversi algoritmi allo stesso set di dati per ricavare insight specifici, garantendo un disaccoppiamento totale dai dettagli implementativi. Ogni algoritmo implementa l'interfaccia `DataProcessingStrategy` e può essere selezionato dinamicamente in fase di esecuzione in base alle esigenze dell'utente o del contesto.

```
public interface DataProcessingStrategy<T> {

    enum ProcessingType {
        CALCULATION("Calcoli numerici"),
        STATISTICS("Analisi statistiche"),
        REPORT("Report dettagliati");

        private final String description;

        ProcessingType(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }
    }

    ProcessingResult<T> execute(Object... data) throws
    ValidationException, BusinessLogicException;

    ProcessingType getType();

    default void validateParameters(Object... data) throws
    ValidationException {...}
}
```

Listing 8: Snippet dell'interfaccia DataProcessingStrategy.

Per esempio, la classe `ProduzioneTotaleStrategy` (Listing 9) implementa l'algoritmo per calcolare la produzione totale da un elenco di raccolti, incapsulando tutta la logica necessaria per questa specifica elaborazione.

```java
public class ProduzioneTotaleStrategy implements DataProcessingStrategy<
    BigDecimal> {
    @Override
    public ProcessingResult<BigDecimal> execute(Object... data) throws
    ValidationException {...}

    @Override
    public void validateParameters(Object... data) throws
    ValidationException {...}

    private List<Raccolto> castToRaccoltiList(Object obj) {...}

    @Override
    public ProcessingType getType() {...}
}
```

Listing 9: Snippet della classe ProduzioneTotaleStrategy che implementa DataProcessingStrategy.

4.2.5 Factory

Il **Factory Pattern** è stato utilizzato per la creazione di oggetti Strategy e DAO, centralizzando la logica di istanziazione e promuovendo il principio di separazione delle responsabilità. I Factory forniscono metodi statici per ottenere istanze specifiche, nascondendo i dettagli di creazione e configurazione.

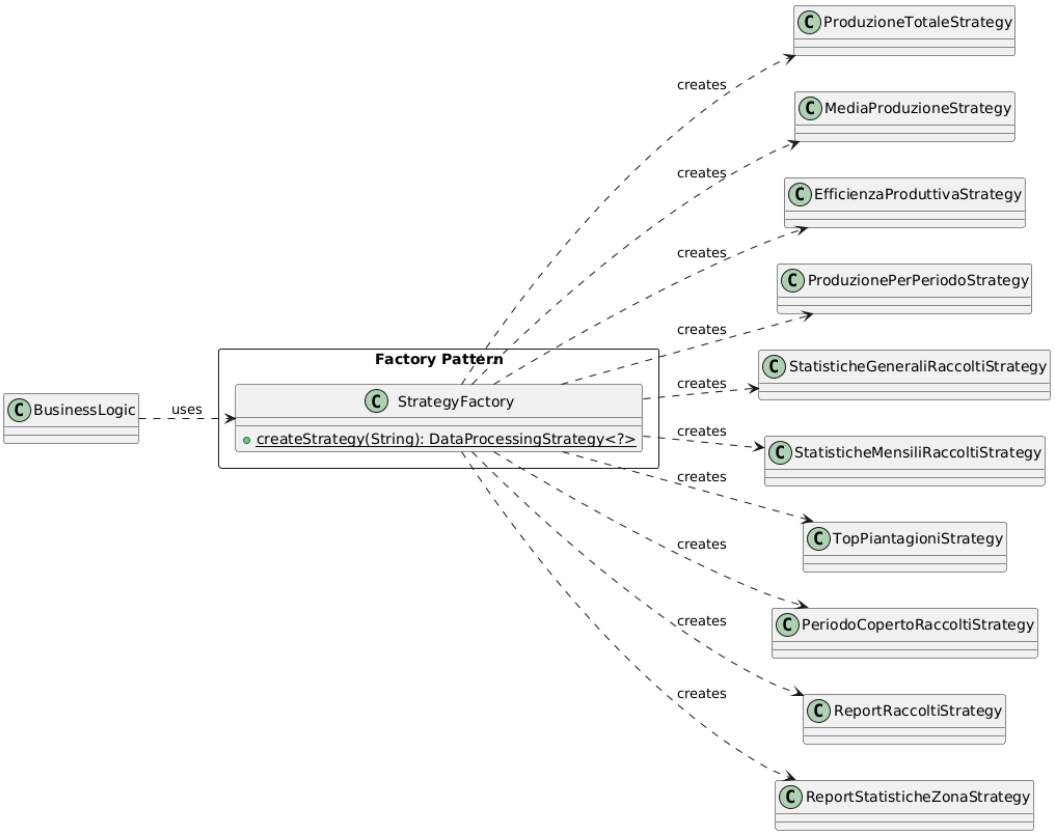

Figura 32: Struttura del Factory Pattern per la creazione di Strategy.

La classe `BusinessLogic` (Listing 10) utilizza il **Factory Pattern** per selezionare e eseguire la strategia di elaborazione dati appropriata in base al tipo di analisi richiesto, garantendo un'architettura flessibile e facilmente estendibile. L'accesso ai DAO avviene tramite la **DAOFactory**, che incapsula la logica di creazione e configurazione dei DAO.

```java
    public ProcessingResult<?> eseguiStrategiaConDati(
    DataProcessingStrategy.ProcessingType tipo, String strategia, String
    piantagioneId,
```

```
                                                        LocalDate
    dataInizio, LocalDate dataFine, Integer topN)
            throws ValidationException, DataAccessException,
    BusinessLogicException {
        try {
            List<Raccolto> raccolti = DAOFactory.getRaccoltoDAO().findAll
    ();
            List<Piantagione> piantagioni = DAOFactory.getPiantagioneDAO
    ().findAll();
            List<Zona> zone = DAOFactory.getZonaDAO().findAll();

            DataProcessingStrategy<?> strategy = StrategyFactory.
    createStrategy(strategia);

            if (strategy.getType() != tipo) {
                throw new ValidationException("tipoStrategia", strategia,
                        "non è del tipo richiesto " + tipo);
            }

            Object[] params = prepareParameters(strategy, raccolti,
    piantagioni, zone,
                piantagioneId, dataInizio, dataFine, topN);

            return strategy.execute(params);

        } catch (SQLException e) {
            throw DataAccessException.queryError("recupero dati per
    elaborazione", e);
        }
    }
```

Listing 10: Snippet della classe BusinessLogic che implementa l'esecuzione delle strategie di elaborazione dati mediante l'utilizzo del Factory Pattern.

4.2.6 Singleton

Il **Singleton Pattern** è stato adottato per garantire che alcune classi abbiano una sola istanza globale accessibile in modo controllato. Questo è particolarmente utile per componenti come la connessione al database, **DAOFactory** e l'ErrorService, dove un'unica istanza è sufficiente e desiderabile per gestire le operazioni di accesso ai dati e la gestione degli errori.

```
public class DatabaseConnection {

    private static Connection connection; // Singleton pattern
    // Altri attributi per la configurazione del database

    public static Connection getConnection() throws SQLException {
        if (connection == null || connection.isClosed()) {
            String url = isTest ? TEST_URL : URL;
            String user = isTest ? TEST_USER : USER;
            String password = isTest ? TEST_PASSWORD : PASSWORD;

            try {
                // Tentativo 1: se il driver è auto-registrato (JDBC 4+),
     questa chiamata basta.
                connection = DriverManager.getConnection(url, user,
    password);
            } catch (SQLException first) {
                // Tentativo 2: prova a caricare esplicitamente il driver
     e ritenta.
                try {
                    Class.forName("org.postgresql.Driver");
                    connection = DriverManager.getConnection(url, user,
    password);
                } catch (ClassNotFoundException cnfe) {
```

```
                    throw new SQLException("Driver PostgreSQL non
    disponibile nel classpath", cnfe);
                }
            }
        }
        return connection;
    }
}
```

Listing 11: Snippet della classe DatabaseConnection che implementa il Singleton Pattern.

4.2.7 Observer

Il pattern Observer è stato implementato per facilitare la comunicazione tra componenti disaccoppiati, in particolare tra le View e i Controller. Questo pattern consente alle View di notificare automaticamente i Controller quando si verificano cambiamenti di stato, migliorando la reattività dell'interfaccia utente e mantenendo una separazione chiara delle responsabilità. Nella classe FornitoreView

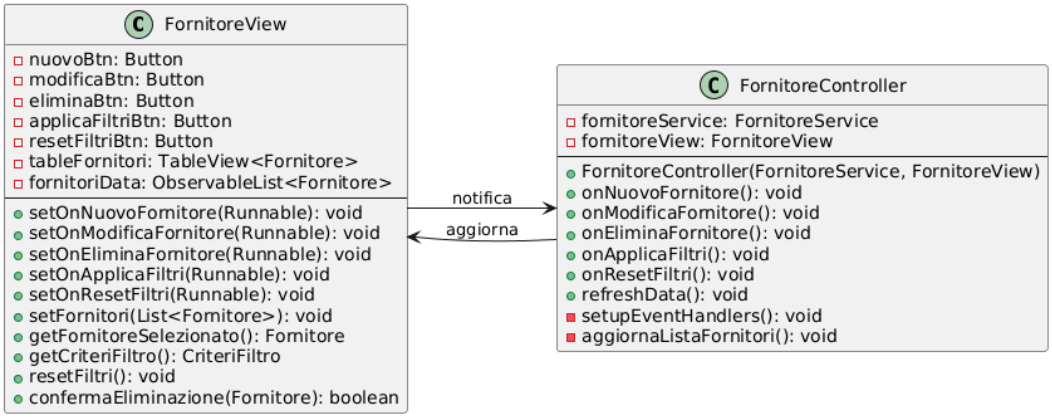

Figura 33: Struttura del pattern Observer dove la classe ViewFornitore è il soggetto e la classe ControllerFornitore è l'observer.

viene mantenuta una lista di observer (implementata internamente a JavaFX) che vengono notificati ogni volta che si verifica un cambiamento di stato rilevante, come l'aggiunta o la modifica di un fornitore. In questo caso il ControllerFornitore si registra come observer della ViewFornitore per ricevere aggiornamenti e reagire di conseguenza.

```
public class FornitoreView extends VBox {
    private final Button nuovoBtn = new Button("Nuovo Fornitore");
    // Altri componenti UI...

    public void setOnNuovoFornitore(Runnable handler) {
        nuovoBtn.setOnAction(e -> handler.run());
    }
    // Altri metodi per gestire l'interfaccia...
}
```

Listing 12: Snippet della classe ViewFornitore che implementa il pattern Observer(JavaFX), e il soggetto del pattern observer.

```
public class FornitoreController {
    private final FornitoreService fornitoreService;
    private final FornitoreView fornitoreView;
    // Altri attributi...

    private void setupEventHandlers() {
        fornitoreView.setOnNuovoFornitore(this::onNuovoFornitore);
        // Altri handler...
    }
    public void onNuovoFornitore() {...}

```

```
    // Altri metodi...
}
```

Listing 13: Snippet della classe ControllerFornitore che implementa il pattern Observer, la classe si registra come observer della ViewFornitore.

4.3 Gestione Errori e Exception Hierarchy

AgroManager implementa una gerarchia di eccezioni personalizzate (già introdotta visivamente nella Figura 26 a pagina 37). Ogni eccezione personalizzata AgroManagerException trasporta tre informazioni fondamentali:

- **ErrorCode:** Un enum ErrorCode che classifica l'errore (es. `VALIDATION_002`, `DB_001`), centralizzando tutti i codici e le descrizioni.
- **User Message:** Un messaggio pulito e comprensibile destinato all'utente finale (es. "Il campo 'nome' è obbligatorio").
- **Technical Message:** Un messaggio di debug dettagliato destinato ai log (es. "Validazione fallita per campo 'nome' con valore 'null'").

```
public class AgroManagerException extends Exception {
    private final ErrorCode errorCode;
    private final String userMessage;

    public AgroManagerException(ErrorCode errorCode, String
    technicalMessage, String userMessage) {
        super(technicalMessage); // Il technicalMessage è per lo
    sviluppatore
        this.errorCode = errorCode;
        this.userMessage = userMessage; // Il userMessage è per l'utente
    }

    public String getUserMessage() {
        return userMessage;
    }
    // ... altri getter ...
}
```

Listing 14: La classe base `AgroManagerException` che definisce la struttura di un errore.

La gerarchia suddivide gli errori in base alla loro origine e a chi è responsabile della loro risoluzione (l'utente o lo sviluppatore):

- **ValidationException** ValidationException: Errori di input dell'utente. Sono errori "sicuri" da mostrare all'utente, che può correggerli (es. campo obbligatorio mancante).
- **BusinessLogicException** BusinessLogicException: Violazioni delle regole di business. L'input è valido, ma l'azione viola una regola (es. "Voce duplicata", "Entità non trovata").
- **DataAccessException** DataAccessException: Errori tecnici, critici e non recuperabili dall'utente (es. "Errore di connessione al database", "Query SQL fallita").

Il vantaggio principale di questa architettura emerge nei Controller (parte dell'architettura MVC). Come mostrato nel Listing 15, il `Controller` orchestra le operazioni all'interno di un blocco `try-catch` che gestisce queste eccezioni in modo differenziato.

```
// Metodo nel FornitoreController
private void onSalvaFornitore() {
    try {
        // 1. Prende i dati dalla View
        Fornitore fornitore = fornitoreView.getDatiDalForm();

        // 2. Chiama il Service per l'operazione
        fornitoreService.aggiungiFornitore(fornitore);

```

```
10          // 3. Operazione riuscita: notifica l'utente
11          NotificationHelper.showSuccess("Successo", "Fornitore salvato
    correttamente.");
12          aggiornaVista(); // Aggiorna la tabella
13
14      } catch (ValidationException | BusinessLogicException e) {
15          // --- GESTIONE ERRORE "SICURO" (per l'utente) ---
16          // L'utente ha sbagliato qualcosa. Mostro il suo messaggio.
17          NotificationHelper.showWarning("Dati non validi", e.
    getUserMessage());
18
19      } catch (DataAccessException e) {
20          // --- GESTIONE ERRORE CRITICO (per lo sviluppatore) ---
21          // Errore tecnico. Mostro un messaggio generico all'utente
22          // e loggo i dettagli tecnici per il debug.
23          NotificationHelper.showError("Errore di Sistema",
24              "Impossibile salvare. Contattare l'amministratore.");
25
26          // Loggo l'eccezione completa per il debug
27          logger.error(e.getTechnicalMessage(), e);
28
29      } catch (Exception e) {
30          // Catch-all per errori imprevisti
31          NotificationHelper.showError("Errore Imprevisto", e.getMessage())
    ;
32          logger.error("Errore non gestito", e);
33      }
34  }
```

Listing 15: Esempio di gestione differenziata delle eccezioni in un Controller.

Come si vede nello snippet, gli errori di validazione (gestibili dall'utente) vengono intercettati e usati per mostrare un avviso NotificationHelper, mentre un errore critico come `DataAccessException` viene usato per mostrare un messaggio di errore generico, registrando però l'errore tecnico per l'analisi.

5 Test

L'infrastruttura di test del progetto è stata sviluppata utilizzando **JUnit 5** per garantire l'affidabilità, la correttezza e la manutenibilità del codice. L'approccio segue una strategia multi-livello, suddividendo i test in tre categorie principali che validano l'architettura implementata:

- **Test Strutturali (Unit Test):** Focalizzati sulla verifica in **isolamento** dei singoli componenti (classi del Domain Model e logica di validazione dei Service), utilizzando *mock object* per simulare le dipendenze esterne (come i DAO).
- **Test di Integrazione (ORM):** Mirati a validare la corretta comunicazione tra l'applicazione e il database PostgreSQL. Verificano l'intero ciclo di vita **CRUD** (Create, Read, Update, Delete) e la correttezza delle query SQL.
- **Test Funzionali:** Verificano una *feature* completa "verticalmente" attraverso più layer (es. la generazione di un report), assicurando che il risultato finale sia corretto.

Per tutti i test che richiedono l'accesso al database, viene attivata una modalità di test (DatabaseConnection.setTestMode(true)) che reindirizza le operazioni verso un database di test dedicato, garantendo l'isolamento dall'ambiente di produzione.

5.1 Test Strutturali (Unit Test)

Questi test "white-box" verificano la logica interna dei componenti senza dipendere dal database o da altri servizi.

5.1.1 Test del Domain Model

I test sulle entità del `DomainModel`, come `PiantagioneTest`, sono i più semplici. Il loro scopo è validare l'integrità dei dati, il comportamento dei costruttori (Listing 16) e la corretta implementazione di getter, setter e della logica di stato di default (es. lo stato `ATTIVA`).

```
@Test
@DisplayName("Test costruttore vuoto")
void testCostruttoreVuoto() {
    Piantagione piantagione = new Piantagione();
    assertNotNull(piantagione);
    assertNull(piantagione.getId());

    // Verifica che lo stato di default sia 1 (ATTIVA)
    assertEquals(1, piantagione.getIdStatoPiantagione());
}

@Test
@DisplayName("Test gestione stato piantagione - Oggetto")
void testGestioneStatoOggetto() {
    StatoPiantagione statoTest = new StatoPiantagione(
        1, StatoPiantagione.ATTIVA, "Piantagione attiva");

    piantagione.setStatoPiantagione(statoTest);

    assertEquals(statoTest, piantagione.getStatoPiantagione());
    // Verifica che l'ID sia stato sincronizzato
    assertEquals(1, piantagione.getIdStatoPiantagione());
}
```

Listing 16: Test unitario per il costruttore e la logica di stato di default dell'entità Piantagione.

5.1.2 Test dei Service (Logica di Validazione)

Questa è la categoria di unit test più importante. Per testare la logica di business (principalmente la validazione) in **totale isolamento**, i `Service` vengono istanziati iniettando **Mock DAO**. Questi mock (Listing 17) simulano il comportamento del database (es. restituendo liste vuote o assegnando ID fittizi) senza mai stabilire una connessione reale.

Questo approccio permette di testare in modo granulare ogni singola regola di validazione definita nei service, come mostrato nel Listing 18.

```
// Mock DAO che simula successo nelle operazioni
static class MockPiantaDAO extends PiantaDAO {
    @Override
    public void create(Pianta pianta) {
        // Simula l'assegnazione di un ID dopo il salvataggio
        pianta.setId(1);
    }

    @Override
    public List<Pianta> findAll() {
        // Restituisce una lista vuota per evitare conflitti
        return new java.util.ArrayList<>();
    }

    // ... altri metodi mockati ...
}

@BeforeEach
void setUp() {
    // Il Service viene istanziato con il MOCK, non con il DAO reale
    piantaService = new PiantaService(new MockPiantaDAO());
    // ...
}
```

Listing 17: Definizione di un **Mock DAO** interno alla classe di test per isolare il service.

```
@Test
@DisplayName("Test aggiunta pianta con tipo vuoto")
void testAggiungiPiantaTipoVuoto() {
    piantaValida.setTipo("");
    ValidationException exception = assertThrows(ValidationException.
    class, () -> {
        piantaService.aggiungiPianta(piantaValida);
    });
    assertTrue(exception.getMessage().contains("Tipo"));
}

@Test
@DisplayName("Test aggiunta pianta con costo negativo")
void testAggiungiPiantaCostoNegativo() {
    piantaValida.setCosto(new BigDecimal("-1.00"));
    ValidationException exception = assertThrows(ValidationException.
    class, () -> {
        piantaService.aggiungiPianta(piantaValida);
    });
    assertTrue(exception.getMessage().contains("costo"));
}

@Test
@DisplayName("Test aggiunta pianta con fornitore ID nullo")
void testAggiungiPiantaFornitoreIdNull() {
    piantaValida.setFornitoreId(null);
    ValidationException exception = assertThrows(ValidationException.
    class, () -> {
        piantaService.aggiungiPianta(piantaValida);
    });
    assertTrue(exception.getMessage().contains("Fornitore"));
}
```

Listing 18: Esempi di test di validazione ”white-box” su PiantaService.

5.2 Test di Integrazione (ORM e Database)

Questi test validano l'intero strato ORM, assicurando che le classi DAO interagiscano correttamente con il database PostgreSQL. Testano la correttezza delle query SQL, la gestione delle `SQLException`

e il mapping tra `ResultSet` e oggetti del dominio, verificando che l'implementazione sia coerente con lo schema E-R del database. Per garantire che i test siano eseguibili più volte senza alterare lo stato del sistema, viene utilizzata una strategia di *setup* e *teardown*:

- `@BeforeAll`: Attiva la modalità test del database.
- `@BeforeEach`: Crea le entità necessarie (incluse le dipendenze da chiavi esterne, come `Fornitore` per `Pianta`) e le salva nel DB.
- `@AfterEach`: Elimina i dati creati durante il test, riportando il DB allo stato iniziale.
- `@AfterAll`: Disattiva la modalità test.

```
@BeforeEach
void createTestObjects() throws SQLException {
    // 1. Crea la dipendenza (Fornitore)
    testFornitore = new Fornitore();
    testFornitore.setNome("Fornitore Piante Test");
    // ... (set altri campi)
    fornitoreDAO.create(testFornitore);

    // 2. Crea l'entità principale (Pianta)
    testPianta = new Pianta();
    testPianta.setTipo("Albero");
    testPianta.setFornitoreId(testFornitore.getId());
    piantaDAO.create(testPianta);
}

@AfterEach
void cleanUp() throws SQLException {
    // Pulisce in ordine inverso
    if (testPianta != null) { piantaDAO.delete(testPianta.getId()); }
    if (testFornitore != null) { fornitoreDAO.delete(testFornitore.getId
    ()); }
}

@Test
@DisplayName("Test aggiornamento pianta")
void testUpdatePianta() throws SQLException {
    String nuovaVarieta = "Pero";
    testPianta.setVarieta(nuovaVarieta);

    // Azione: aggiorna
    piantaDAO.update(testPianta);

    // Assert: rilegge e verifica
    Pianta updated = piantaDAO.read(testPianta.getId());
    assertEquals(nuovaVarieta, updated.getVarieta());
}
```

Listing 19: Test di integrazione per il ciclo **CRUD** completo su PiantaDAO.

5.2.1 Test di Componenti Read-Only

Una strategia di test specifica è stata usata per i DAO che implementano componenti "read-only" del sistema, come `StatoPiantagioneDAO`. In questo caso, i test non solo verificano le operazioni di lettura (es. `findByCodice`, `findAllOrdered`), ma validano esplicitamente che le operazioni di scrittura siano **bloccate**, assicurando che il test fallisca se un'operazione non supportata viene accidentalmente abilitata (Listing 20).

```
@Test
@DisplayName("Test blocco operazione create")
void testCreateBlocked() {
    StatoPiantagione nuovoStato = new StatoPiantagione();
    nuovoStato.setCodice("TEST");

```

```
    assertThrows(UnsupportedOperationException.class, () -> {
        statoPiantagioneDAO.create(nuovoStato);
    }, "L'operazione create deve essere bloccata");
}

@Test
@DisplayName("Test blocco operazione update")
void testUpdateBlocked() {
    StatoPiantagione statoEsistente = new StatoPiantagione();
    statoEsistente.setId(1);

    assertThrows(UnsupportedOperationException.class, () -> {
        statoPiantagioneDAO.update(statoEsistente);
    }, "L'operazione update deve essere bloccata");
}
```

Listing 20: Test di sicurezza che verifica il blocco delle operazioni di scrittura su un DAO read-only.

5.3 Test Funzionali

I test funzionali verificano una *feature* completa (come un caso d'uso) eseguendo il codice attraverso più layer. A differenza dei test unitari, questi test non usano mock ma interagiscono con i servizi reali e il database di test per validare il risultato finale.

Il test `ReportServiceTest` è un esempio di questa strategia. Verifica che la generazione dei report (un'operazione complessa che richiede l'esecuzione di `Strategy` e l'accesso ai dati tramite il `RaccoltoService`) produca un risultato corretto.

Il test include anche logica condizionale: se il database di test non contiene dati di raccolto (`hasRaccoltiDisponibili()`), il test viene contrassegnato come *skipped*, evitando falsi negativi e rendendo la suite di test più robusta.

```
@Test
@Order(2)
@DisplayName("Report completo con raccolti disponibili")
void testGeneraReportCompleto_ConDati() {
    testLogger.startTest("generaReportCompleto - con dati");

    try {
        // Verifica pre-condizione: ci sono dati?
        if (!reportService.hasRaccoltiDisponibili()) {
            testLogger.expectedError("Report completo",
                "Nessun raccolto disponibile - test skipped");
            return; // Skip se non ci sono dati
        }

        // Azione: chiama la logica di business
        ProcessingResult<Map<String, Object>> result =
            reportService.generaReportCompleto();

        // Assert: verifica il risultato
        assertNotNull(result, "Il risultato non dovrebbe essere null");
        assertNotNull(result.data(), "I dati del report non dovrebbero
    essere null");

        Map<String, Object> data = result.data();
        assertTrue(data.containsKey("statisticheGenerali") || !data.
    isEmpty(),
            "Il report dovrebbe contenere dati");

        testLogger.testPassed("generaReportCompleto - OK");

    } catch (BusinessLogicException | DataAccessException |
    ValidationException e) {
        testLogger.testFailed("generaReportCompleto", e.getMessage());
        fail("Errore durante generazione report: " + e.getMessage());
    }
```

```
33 }
```

Listing 21: Test funzionale per la generazione di un report.

6 Conclusioni e Sviluppi Futuri

Questo lavoro ha presentato la progettazione e l'implementazione di **AgroManager**, un sistema software per la gestione delle attività agricole. È stata seguita una metodologia strutturata, partendo dall'analisi dei requisiti (Capitolo 2) fino alla progettazione dettagliata (Capitolo 3) e all'implementazione dei componenti chiave (Capitolo 4).

L'architettura a layer, unita all'adozione rigorosa di design pattern come il Template Method nel DAO layer e lo Strategy Pattern nel service layer, ha permesso di costruire un'applicazione robusta, manutenibile e disaccoppiata. La gestione centralizzata degli errori tramite una gerarchia di eccezioni custom ha ulteriormente contribuito alla solidità del sistema.

6.1 Sviluppi Futuri

L'architettura attuale, basata su principi di modularità e separazione delle responsabilità, pone solide fondamenta per numerose estensioni future. Alcuni dei possibili sviluppi includono:

- **Autenticazione Utenti e Gestione Ruoli:** Introduzione di un sistema di login per differenziare gli utenti (es. Amministratore, Operatore). Questo permetterebbe di implementare controlli di accesso granulari, limitando l'accesso a determinate funzionalità (es. solo l'amministratore può gestire le zone, mentre l'operatore può solo registrare i raccolti).
- **Estensione del Modulo di Analisi:** Grazie all'uso dello Strategy Pattern, è possibile aggiungere facilmente nuove strategie di calcolo e report. Si potrebbero implementare analisi più complesse, come la previsione della data di raccolta basata su dati storici, o un'analisi dei costi per zona agricola.
- **Integrazione IoT e Monitoraggio Live:** L'architettura Service-based è predisposta per ricevere dati da fonti esterne. Un'evoluzione significativa sarebbe l'integrazione con sensori IoT (umidità, temperatura, pH del terreno) che, tramite un servizio di background, aggiornano lo stato delle piantagioni in tempo reale, sfruttando l'infrastruttura del database già esistente.
- **Applicazione Mobile per Operatori:** Sviluppo di un'interfaccia mobile (es. Android) che usa gli stessi Service già sviluppati. Questo permetterebbe agli operatori sul campo di registrare i raccolti o segnalare problemi direttamente da smartphone, migliorando notevolmente l'efficienza operativa e integrando perfettamente con l'attuale architettura MVC.

In conclusione, il sistema AgroManager, pur essendo un prototipo funzionale, è stato progettato con un'attenzione particolare all'estensibilità, dimostrando come il sistema possa supportare il ciclo di vita e l'evoluzione di un prodotto complesso.