



UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

GreenHouse
Software per la gestione di serre e piante

Autori:

Elion Karaboj
Lorenzo Cappellini

Corso principale:

Ingegneria del Software

N° Matricola:

7030984
7049027

Docente corso:

Enrico Vicario

Indice

1 Introduzione	2
1.1 Statement	2
1.2 Struttura e pratiche utilizzate	2
2 Progettazione	4
2.1 Use Case Diagram	4
2.2 Use Case Template	4
2.3 Mock-ups	9
2.4 Class Diagram	11
2.5 ER Diagram e modello relazionale	13
2.6 Navigation Diagram	15
3 Implementazione	16
3.1 Domain Model	16
3.1.1 Utente	16
3.1.2 Pianta	17
3.1.3 Ordine	17
3.1.4 Impianto	18
3.1.5 Sensori	18
3.1.6 Attuatori	18
3.2 Business Logic	19
3.2.1 LoginClienteController	19
3.2.2 LoginPersonaleController	19
3.2.3 ClienteController	19
3.2.4 OperatoreController	19
3.2.5 AdminController	20
3.2.6 AdminExtraController	21
3.3 ORM (Object-Relational Mapping)	22
3.3.1 ConnectionManager	22
3.3.2 ClienteDAO	23
3.3.3 AdminDAO	23
3.3.4 OperatoreDAO	24
3.3.5 SensoreDAO, AttuatoreDAO e OperazioneDAO	24
3.3.6 SettoreDAO, PosizioneDAO	24
3.3.7 OrdineDAO e PiantaDAO	24
3.3.8 PosizionamentoDAO	26
3.4 Database	26
3.5 Interfaccia	27
4 Testing	28
4.1 Business Logic Test	28
4.1.1 LoginClienteControllerTest	28
4.1.2 LoginPersonaleControllerTest	30
4.1.3 ClienteControllerTest	30
4.1.4 OperatoreControllerTest	31
4.2 Domain Model Test	33

1 Introduzione

Elaborato per il superamento dell'esame di Ingegneria del Software, appartenente al modulo Basi di Dati / Ingegneria del Software del corso di Laurea Triennale in Ingegneria Informatica dell'Università degli Studi di Firenze.

Il progetto è stato sviluppato da Elion Karaboj a e Lorenzo Cappellini (matricole 7030984 e 7049027) durante il periodo di Maggio - Settembre 2024 (a.a. 2023/2024).

Il codice sorgente è disponibile su Github al seguente indirizzo:
https://github.com/lcappellini/SWE_Greenhouse.

1.1 Statement

Il progetto modella un sistema di controllo e gestione di serre, e in particolare si occupa della cura e crescita delle piante in esse, al fine ultimo di venderle a clienti. I 3 spazi delle serre sono divisi in 4 settori in cui è presente un sistema di areazione, di illuminazione e controllo della temperatura, luce e umidità, e contengono 5 posizioni ciascuna nelle quali mettere le piante che saranno irrigate e monitorate.

La gestione delle piante è in gran parte automatizzata e possono essere monitorati sia i settori che le posizioni per ottenere un controllo *real-time* da parte dell'Admin.

Ci sono operazioni che dovranno essere effettuate manualmente come la cura e il posizionamento delle piante nelle posizioni designate, che saranno eseguite da operatori. Tale sistema è adottato da un'azienda che permette ai clienti di acquistare e commissionare un certo numero di piante con consegna prevista entro un certo periodo.

1.2 Struttura e pratiche utilizzate

Il software è stato sviluppato in Java, mentre per la gestione e il salvataggio dei dati è stato connesso un database PostgreSQL ed è stata utilizzata la libreria JDBC (Java DataBase Connectivity).

Per mantenere una separazione delle responsabilità, la struttura del progetto è stata divisa in tre parti principali: Business Logic, Domain Model e ORM. Questi tre packages si occupano in modo distinto della logica di business, della rappresentazione dei dati e dell'accesso ai dati (Figura 1):

- **Business Logic:** contiene le classi che implementano la logica di business del sistema.
- **Domain Model:** contiene le classi che rappresentano le entità del sistema.
- **ORM:** contiene le classi che implementano l'Object-Relational Mapping. In questo modo è possibile rendere i dati persistenti e recuperarli dal database.

Per utilizzare il software è stata creata un'interfaccia da riga di comando (CLI) che permette di interagire con il sistema in modo semplice e intuitivo.

Gli Use Case Diagram e i Class Diagram seguono lo standard UML (Unified Modeling Language) e sono stati realizzati con il software StarUML. Infine, per la parte di testing è stato utilizzato JUnit.

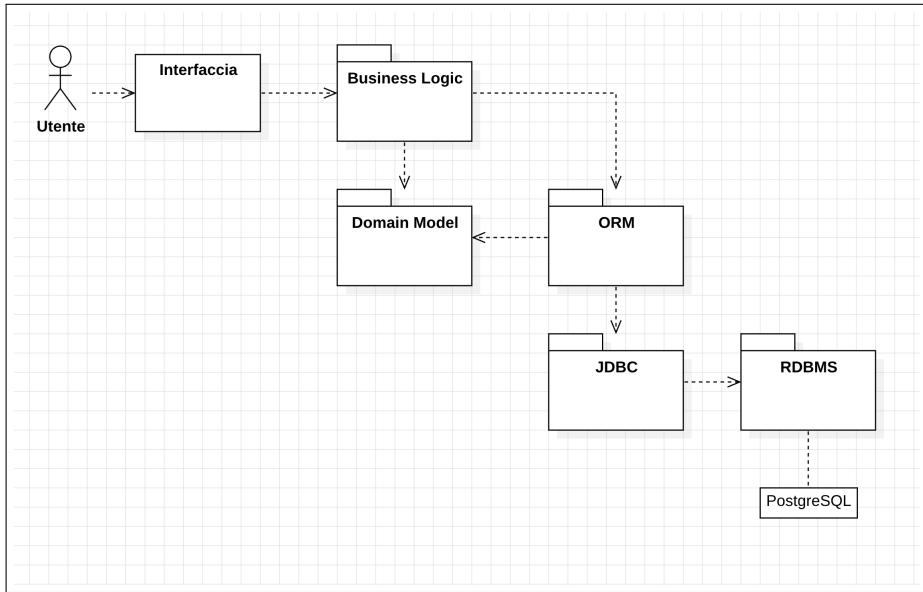


Figura 1: Package Dependency Diagram

2 Progettazione

2.1 Use Case Diagram

Come descritto in precedenza, il sistema è stato progettato per permettere a clienti di ordinare e ritirare le piante desiderate, le quali sono gestite da operatori e monitorate da un Admin. La Figura 2 illustra il diagramma dei casi d'uso del sistema, includendo quelli relativi ai Clienti, agli Operatori e agli Admin.

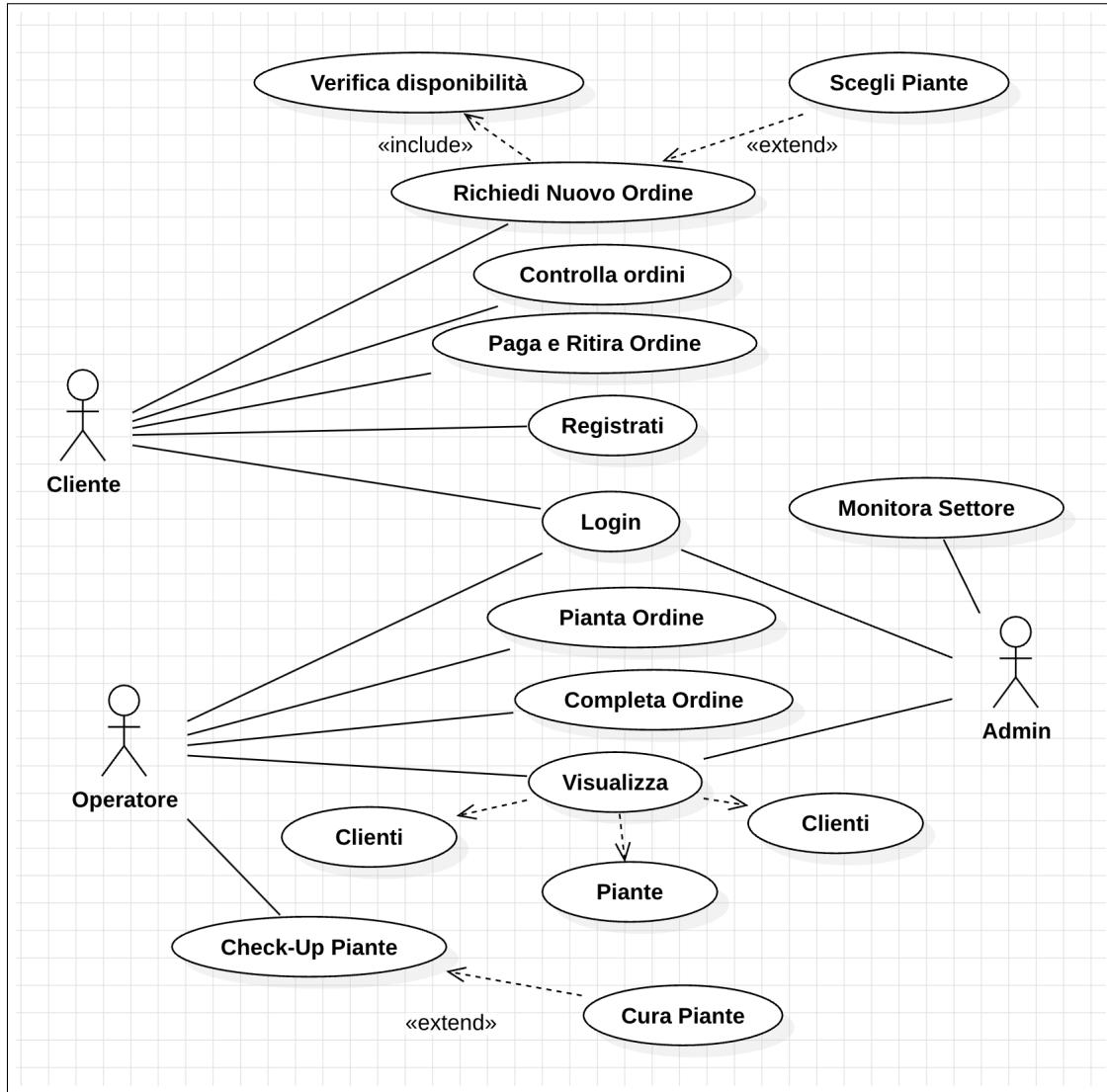


Figura 2: Diagramma degli Use Cases

2.2 Use Case Template

Qui di seguito sono elencati i template di alcuni casi d'uso implementati. Per ciascuno sono specificati dei dettagli: una breve descrizione, il livello del caso d'uso, gli attori coinvolti, le precondizioni, le post-condizioni, il flusso principale e i flussi alternativi.

Use Case #1 Accedi al Sistema (Sign-in)	
Brief Description	L'utente accede al sistema tramite le proprie credenziali (Mockup #1)
Level	User Goal
Actors	Admin, Cliente, Operatore
Pre-Conditions	L'utente deve essere nella pagina dedicata al proprio accesso
Basic Flow	<ol style="list-style-type: none"> 1) L'utente inserisce le proprie credenziali 2) L'utente invia le proprie credenziali 3) Il sistema verifica le credenziali 4) Il sistema autentica l'utente
Alternative Flow	3a) Se le credenziali sono errate, il sistema restituisce un messaggio di errore e permette di ritentare
Post-Conditions	L'utente è autenticato nel sistema e ha accesso alle sue funzioni

Tabella 1: Use Case #1 (Accedi al Sistema)

Use Case #2 Registrati nel Sistema (Sign-up)	
Brief Description	Il cliente si registra nel sistema creando un nuovo account (Mockup #2)
Level	User Goal
Actors	Cliente
Pre-Conditions	Il cliente deve essere nella pagina di accesso del cliente
Basic Flow	<ol style="list-style-type: none"> 1) Il cliente fornisce le informazioni richieste 2) Il cliente conferma la registrazione 3) Il sistema verifica i dati 4) Il sistema crea un nuovo account per il cliente
Alternative Flow	3a) Se il cliente fornisce dati non validi o già presenti (di un altro account), il sistema mostra un messaggio di errore
Post-Conditions	Il cliente deve comunque effettuare il login per accedere al sistema

Tabella 2: Use Case #2 (Registrati al Sistema)

Use Case #3 Richiedi nuovo Ordine	
Brief Description	Il cliente crea un nuovo Ordine (Mockup #3)
Level	User Goal
Actors	Cliente
Pre-Conditions	Il cliente deve aver fatto l'accesso e essere nella pagina "Ordini"
Basic Flow	<ol style="list-style-type: none"> 1) Il cliente seleziona l'opzione per creare un nuovo ordine 2) Il cliente sceglie il tipo di pianta e la quantità desiderata 3) Il cliente sceglie se aggiungere altre piante o se concludere l'ordine 4) Il sistema registra il nuovo ordine
Alternative Flow	<ol style="list-style-type: none"> 2a) Se il cliente inserisce valori non validi, il sistema mostra un messaggio di errore e richiede l'input 4a) Se l'ordine non può essere preso in carico, il sistema restituisce un messaggio
Post-Conditions	Il nuovo ordine è stato registrato nel sistema

Tabella 3: Use Case #3 (Richiedi nuovo Ordine)

Use Case #4 Paga e Ritira Ordine	
Brief Description	Il cliente paga e ritira il proprio Ordine
Level	User Goal
Actors	Cliente
Pre-Conditions	Il cliente è autenticato ed è nella sua pagina "Ordini"
Basic Flow	<ol style="list-style-type: none"> 1) Il cliente seleziona l'opzione per pagare e ritirare l'ordine 2) Il sistema mostra una lista di ordini pronti per essere ritirati 3) Il cliente inserisce l'id dell'ordine scelto 4) Il sistema conferma il pagamento e il ritiro di tale ordine
Alternative Flow	<ol style="list-style-type: none"> 3a) Se l'utente fornisce un id sconosciuto o l'id di un ordine non pronto, il sistema mostra un messaggio di errore
Post-Conditions	Il sistema libera le posizioni associate e rimuove le piante ritirate

Tabella 4: Use Case #4 (Paga e Ritira Ordine)

Use Case #5 Pianta Ordine	
Brief Description	L'Operatore semina le piante associate a un Ordine (Mockup #4)
Level	User Goal
Actors	Operatore
Pre-Conditions	L'Operatore deve essere sulla sua dashboard
Basic Flow	<ol style="list-style-type: none"> 1) L'Operatore sceglie l'opzione "Pianta Ordine" 2) Il sistema mostra una lista di Ordini da piantare 3) L'Operatore inserisce l'id dell'Ordine scelto 4) Il sistema segnala all'operatore di posizionare le piante dell'ordine in posizioni disponibili
Alternative Flow	3a) Se il sistema non trova l'ordine con tale id o l'ordine è già stato posizionato, mostra un messaggio di errore
Post-Conditions	Il sistema genera i Posizionamenti dell'ordine e modifica lo stato dell'Ordine

Tabella 5: Use Case #5 (Pianta Ordine)

Use Case #6 Completa Ordine	
Brief Description	Completa un Ordine (le piante sono cresciute e pronte alla consegna)
Level	User Goal
Actors	Operatore
Pre-Conditions	L'Operatore deve essere sulla sua dashboard
Basic Flow	<ol style="list-style-type: none"> 1) L'Operatore sceglie l'opzione "Completa Ordine" 2) Il sistema mostra una lista di Ordini da completare 3) L'Operatore inserisce l'id dell'Ordine da completare 4) Il sistema imposta l'Ordine come completato e quindi pronto al ritiro
Alternative Flow	3a) Se il sistema non trova l'ordine con tale id o l'ordine è già stato completato, mostra un messaggio di errore
Post-Conditions	Il cliente adesso può pagare e ritirare l'ordine

Tabella 6: Use Case #6 (Completa Ordine)

Use Case #7	Check-Up Piante
Brief Description	L'Operatore verifica lo stato delle piante
Level	User Goal
Actors	Operatore
Pre-Conditions	L'Operatore deve essere sulla dashboard
Basic Flow	<ul style="list-style-type: none"> 1) L'Operatore sceglie l'opzione "Check-Up Piante" 2) Per ogni Pianta del sistema viene eseguito un controllo che valuta se la pianta ha bisogno di cure 3) Se sono presenti piante che hanno bisogno, queste vengono curate 4) Il Check-Up è terminato 5) Il sistema salva l'operazione effettuata dall'operatore nel database
Alternative Flow	2a) Se non sono presenti piante su cui è possibile fare il check-up, il sistema mostra un errore
Post-Conditions	Le Piante sono in salute e possono continuare a crescere

Tabella 7: Use Case #7 (Check-Up Piante)

Use Case #8	Monitora Settore
Brief Description	L'Admin visualizza i parametri dei Sensori e degli Attuatori in tempo reale
Level	User Goal
Actors	Admin
Pre-Conditions	L'Admin deve essere autenticato e sulla dashboard dell'admin
Basic Flow	<ul style="list-style-type: none"> 1) L'Admin sceglie l'opzione "Monitora Settore" 2) Il sistema mostra una lista di settori monitorabili 3) L'Admin inserisce l'id del Settore desiderato 4) Il sistema restituisce i valori dei sensori in tempo reale e lo stato ON/OFF dei rispettivi attuatori 5) L'Admin interrompe il monitoraggio
Alternative Flow	4a) Il sistema non trova il Settore con l'id richiesto e mostra un messaggio di errore
Post-Conditions	L'Admin ritorna alla dashboard

Tabella 8: Use Case #8 (Monitora Settore)

2.3 Mock-ups

Ecco di seguito alcuni possibili Mock-ups relativi alle interfacce grafiche del sistema.

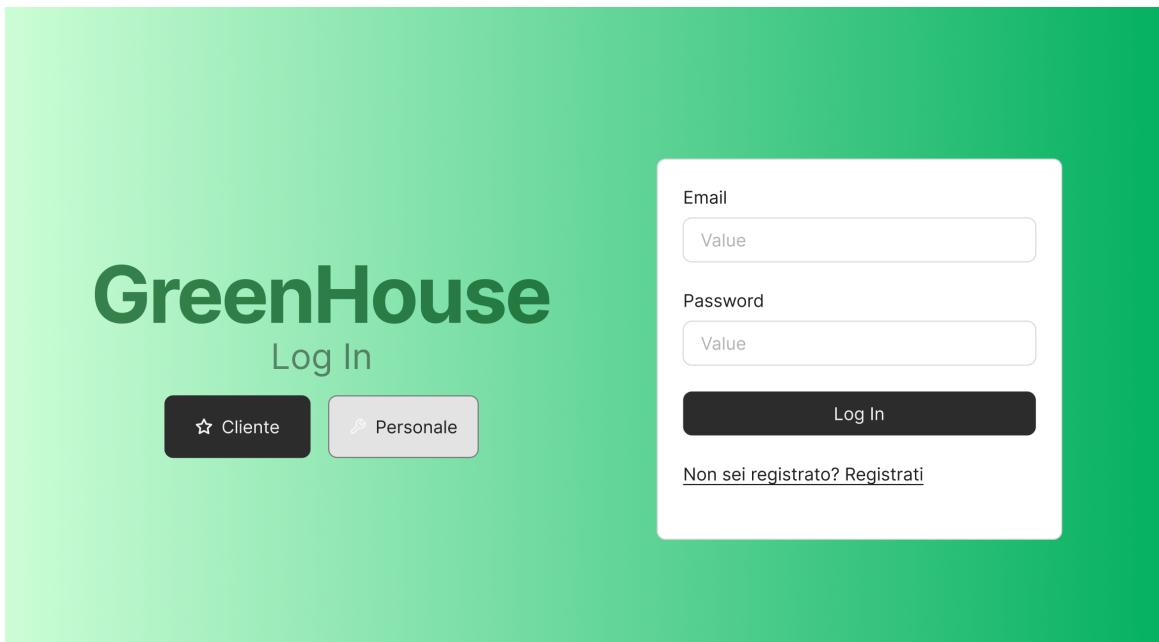


Figura 3: Prototipo della pagina di Sign in - Mockup #1

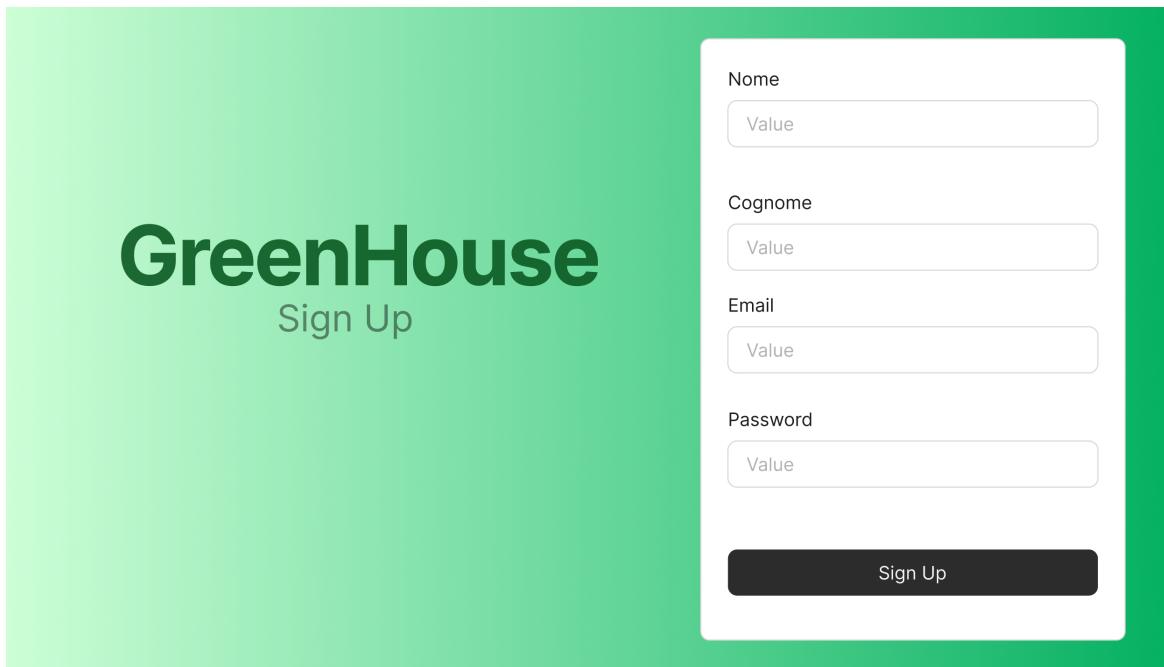


Figura 4: Prototipo della pagina di Sign up - Mockup #2

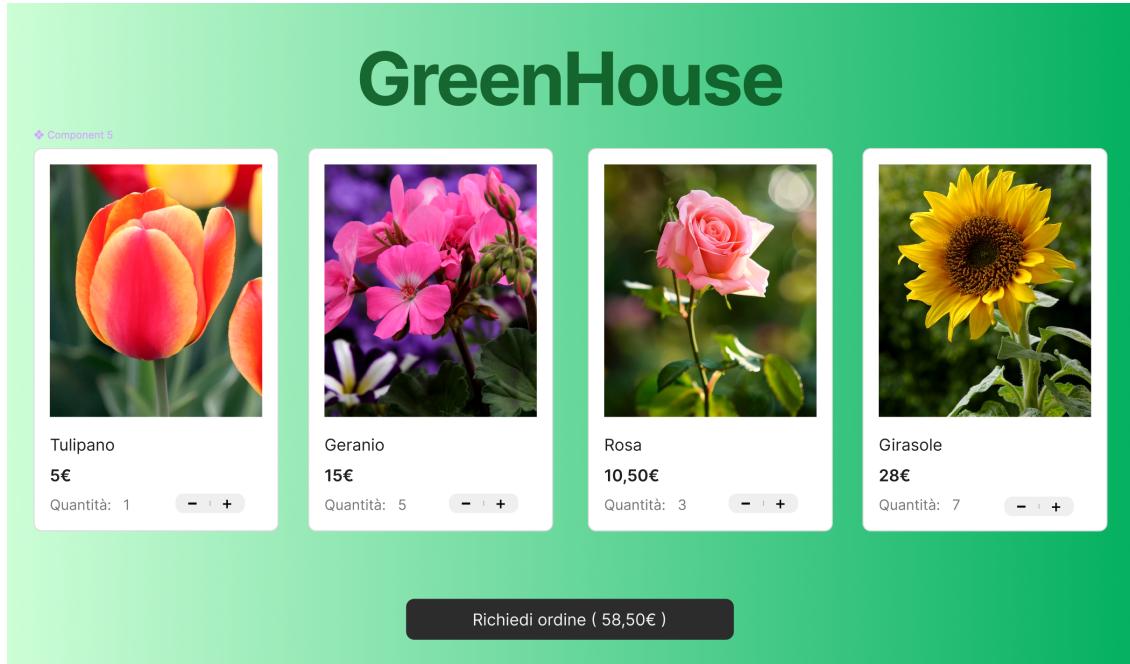


Figura 5: Prototipo della pagina per richiedere nuovi ordini - Mockup #3

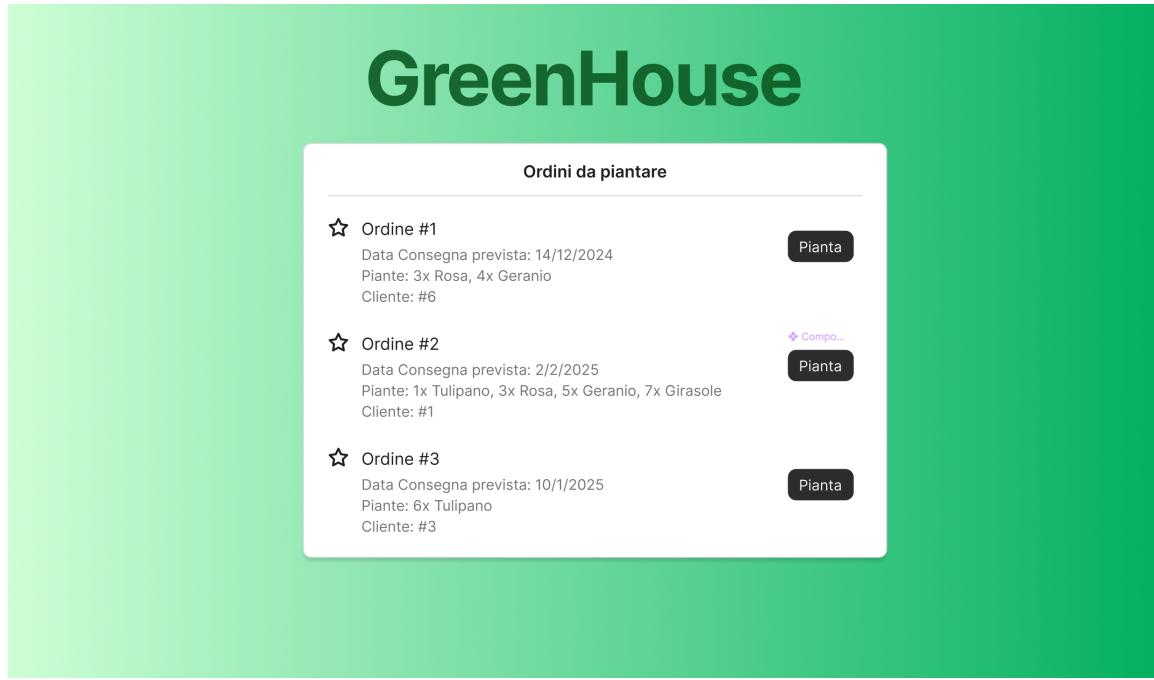


Figura 6: Prototipo della pagina per piantare gli ordini - Mockup #4

2.4 Class Diagram

Vista la divisione strutturale in 3 packages, sono stati realizzati 3 diagrammi delle classi distinte, uno per ogni package:

- **Business Logic** (Figura 7): contiene le classi che implementano la logica di business del sistema, ovvero i seguenti controller: quello che gestisce l'accesso e la registrazione dei nuovi Clienti (LoginClienteController), degli Admin e degli Operatori (LoginPersonaleController), quello che permette ai Clienti di richiedere nuovi ordini, visualizzarli e di ritirarli (ClienteController), quello che gestisce il monitoraggio e visualizzazione dell'impianto da parte dell'Admin (AdminController), infine il controller che segnala le operazioni dell'operatore come piantare un ordine, completarlo o fare il checkup delle piante (OperatoreController).
- **Domain Model** (Figura 8): contiene le classi che rappresentano le entità del sistema ovvero: Ordine, Utente, Cliente, Operatore, Admin, Pianta e il Posizionamento (quest'ultimo funge da Mapper tra Pianta, Ordine e Posizione). In più contiene anche un altro package chiamato Impianto nel quale sono presenti le classi delle entità che compongono la struttura dell'azienda quali: Spazio, Settore e Settore che ne rappresentano l'entità immobile, Attuatore e Sensori che ne rappresentano le entità operative (nello specifico Climatizzatore, Lampada, Irrigatore sono attuatori; IgrometroAria, IgrometroTerra, Termometro e Fotosensore sono sensori).
- **ORM** (Figura 9): contiene le classi che implementano l'Object-Relational Mapping, quindi contiene una classe per ogni entità del sistema: ClienteDAO, OperatoreDAO, AdminDAO, OrdineDAO, PiantaDAO, PosizionamentoDAO, SettoreDAO, PosizioneDAO, AttuatoreDAO e SensoreDAO. In più contiene anche la classe ConnectionManager che si occupa di gestire la connessione al database.

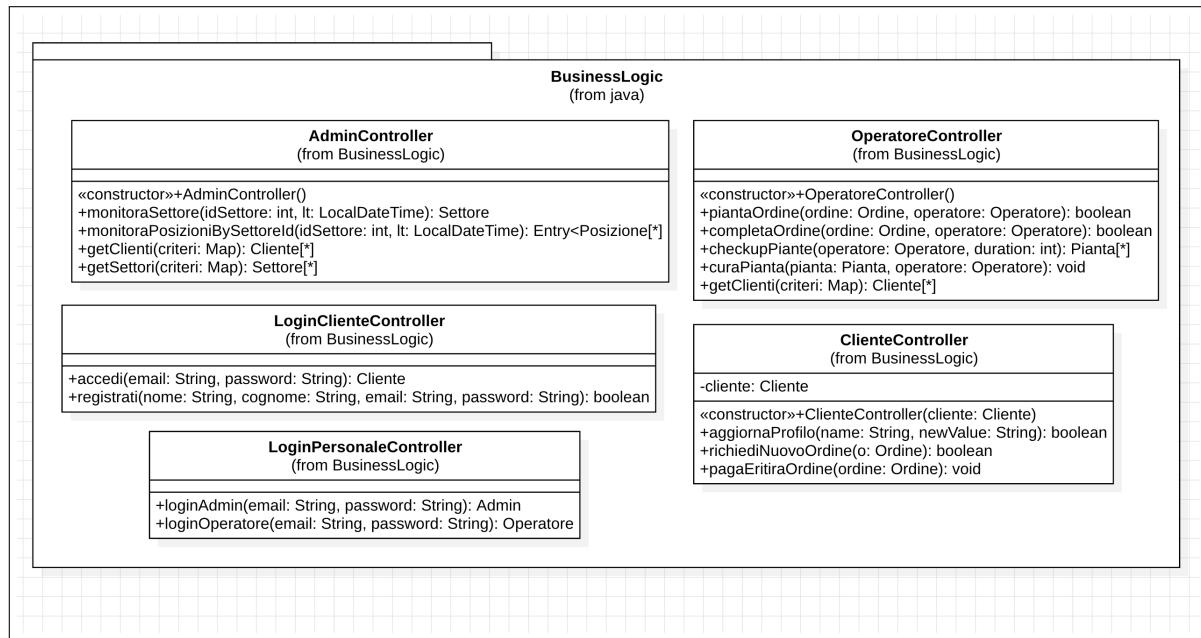


Figura 7: Class Diagram - BusinessLogic

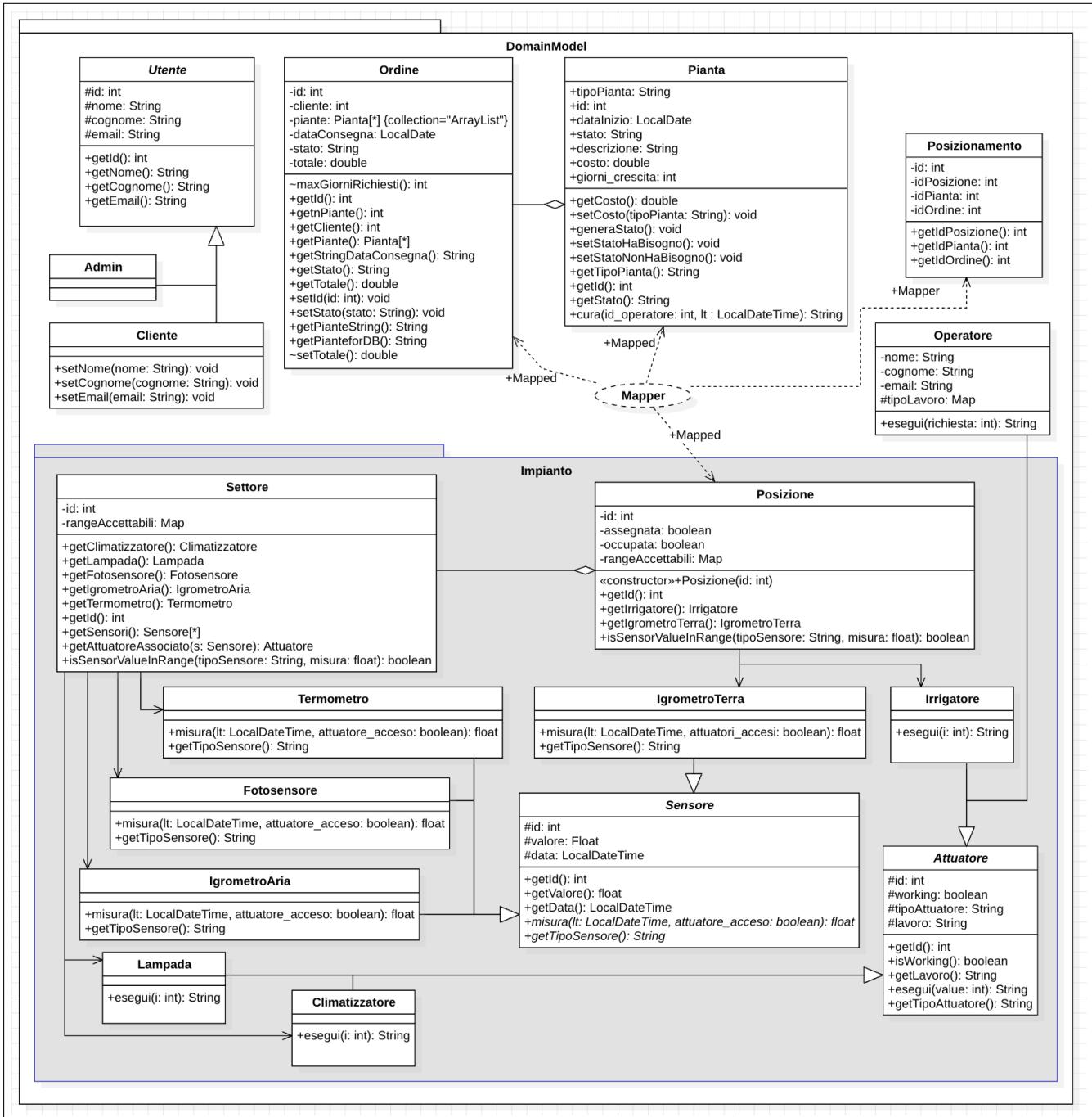


Figura 8: Class Diagram - DomainModel

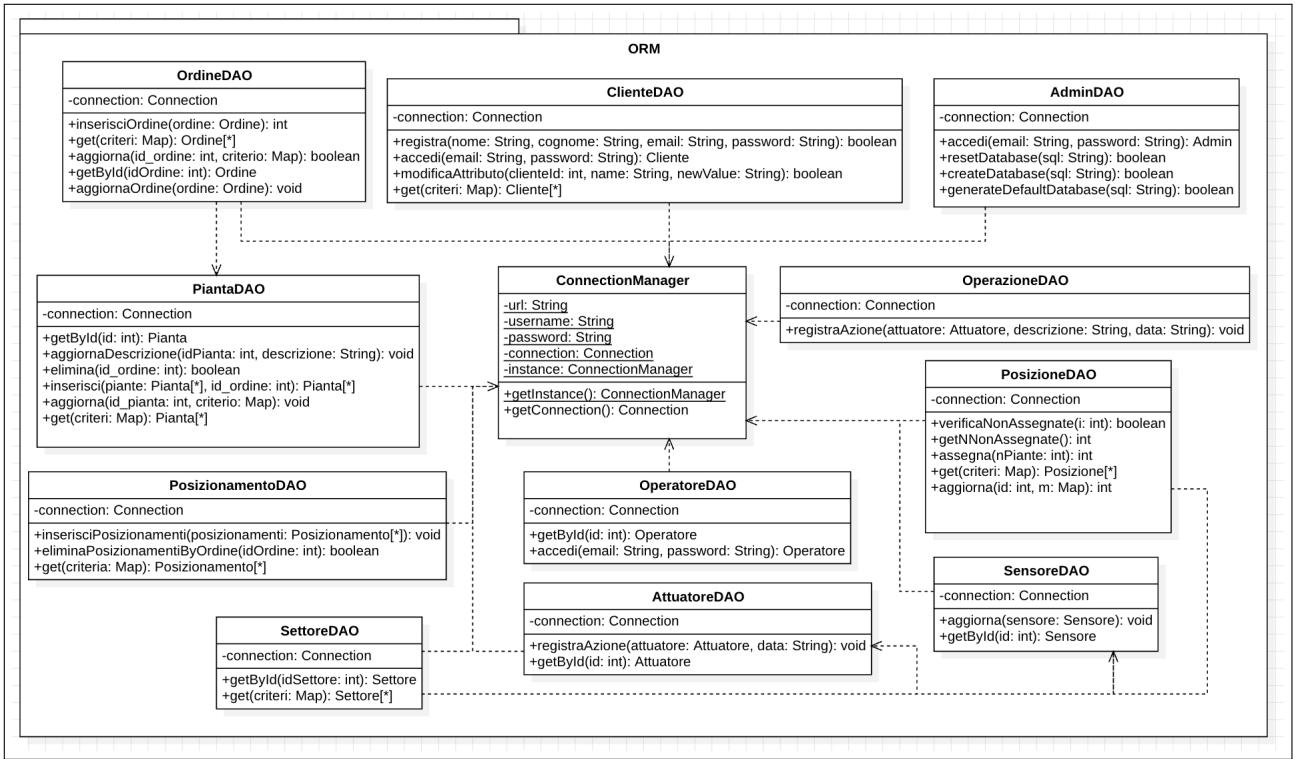


Figura 9: Class Diagram - ORM

2.5 ER Diagram e modello relazionale

Il database è stato progettato seguendo il modello relazionale e facendo attenzione alle relazioni che legano le entità (Figura 10).

Sono state quindi definite le seguenti tabelle:

- **Cliente**: rappresenta l'entità Cliente;
- **Ordine**: rappresenta l'entità Ordine e risolve la relazione "richiedi" con Cliente (un Cliente può fare più Ordini, quindi gli Ordini contengono l'id del Cliente che li ha effettuati);
- **Pianta**: rappresenta l'entità Pianta e risolve la relazione "ha" con Ordine, in quanto un Ordine possiede un insieme di Piante, quindi ogni pianta possiede l'id dell'Ordine a cui appartiene;
- **Admin**: rappresenta l'entità Admin;
- **Operatore**: rappresenta l'entità Operatore;
- **Spazio**: rappresenta l'entità Spazio;
- **Settore**: rappresenta l'entità Settore e contiene l'id dello Spazio a cui appartiene, e gli id dei Sensori e degli Attuatori situati in esso (relazioni di appartenenza);
- **Posizione** rappresenta l'entità Posizione, prende parte alla relazione "Posizionamento" e contiene gli id dei sensori e attuatori che gli appartengono;
- **Posizionamento**: rappresenta l'entità Posizionamento che risolve la relazione di mapping tra Ordine, Pianta e Posizione;

- **Attuatore:** rappresenta l'entità Attuatore.
- **Sensore:** rappresenta l'entità Sensore.
- **Operazione:** rappresenta l'entità Operazione;

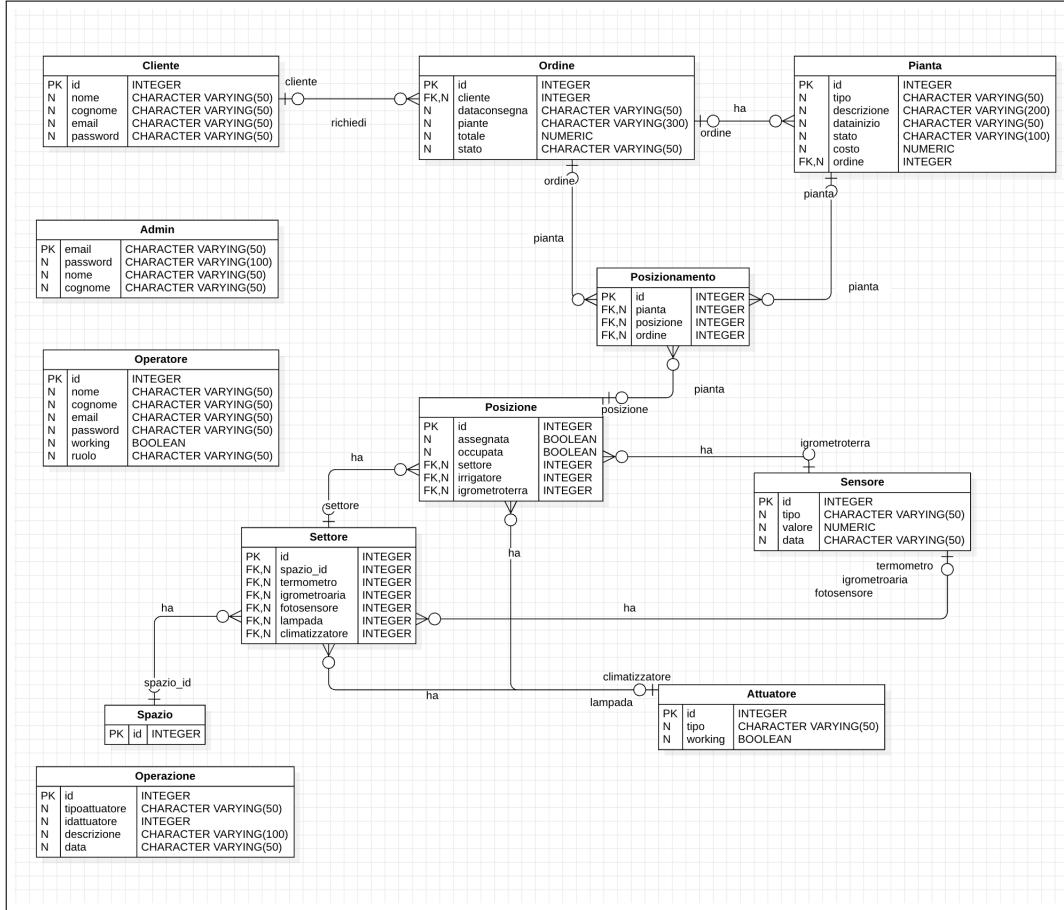


Figura 10: ER Diagram

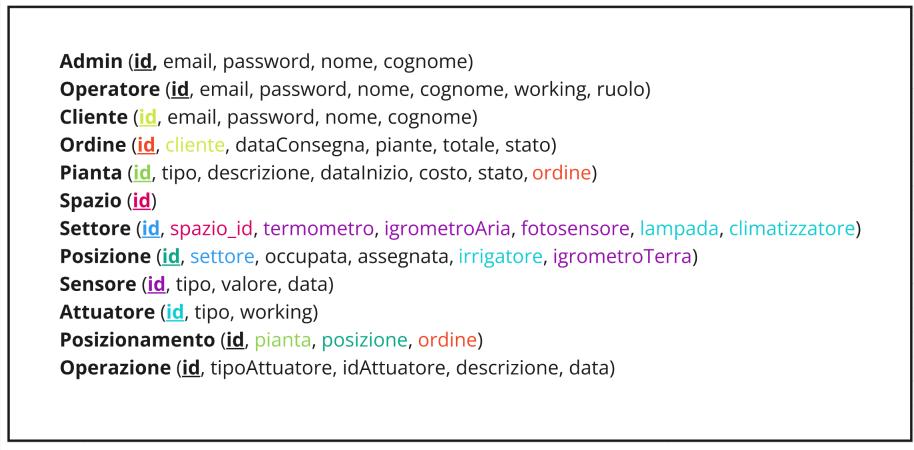


Figura 11: Modello Relazionale

2.6 Navigation Diagram

Il seguente diagramma (Figura 12) rappresenta quelle che sono le pagine principali del sistema e le possibili azioni che l'utente può compiere. Sono anche rappresentati i modi con cui si può navigare tra le varie pagine. Alcune delle pagine sono:

- **Greenhouse:** l'utente sceglie la propria area di appartenenza.
- **Cliente Dashboard:** il cliente può scegliere se entrare nella pagina degli Ordini e quindi crearne uno nuovo, pagarne uno pronto o controllarli, oppure nella pagina Profilo dove può visualizzare o modificare i dati del proprio profilo.
- **Operatore Dashboard:** L'operatore può eseguire i suoi compiti tra quelli elencati quindi pianificare un ordine, completarlo per poter essere ritirato dal cliente oppure può eseguire un controllo sullo stato delle piante.
- **Admin Dashboard:** L'admin ha come funzione principale quella di monitorare i settori e le relative posizioni occupate grazie ai sensori e attuatori. Inoltre può visualizzare, come l'operatore, le tabelle di ordini, piante e clienti.

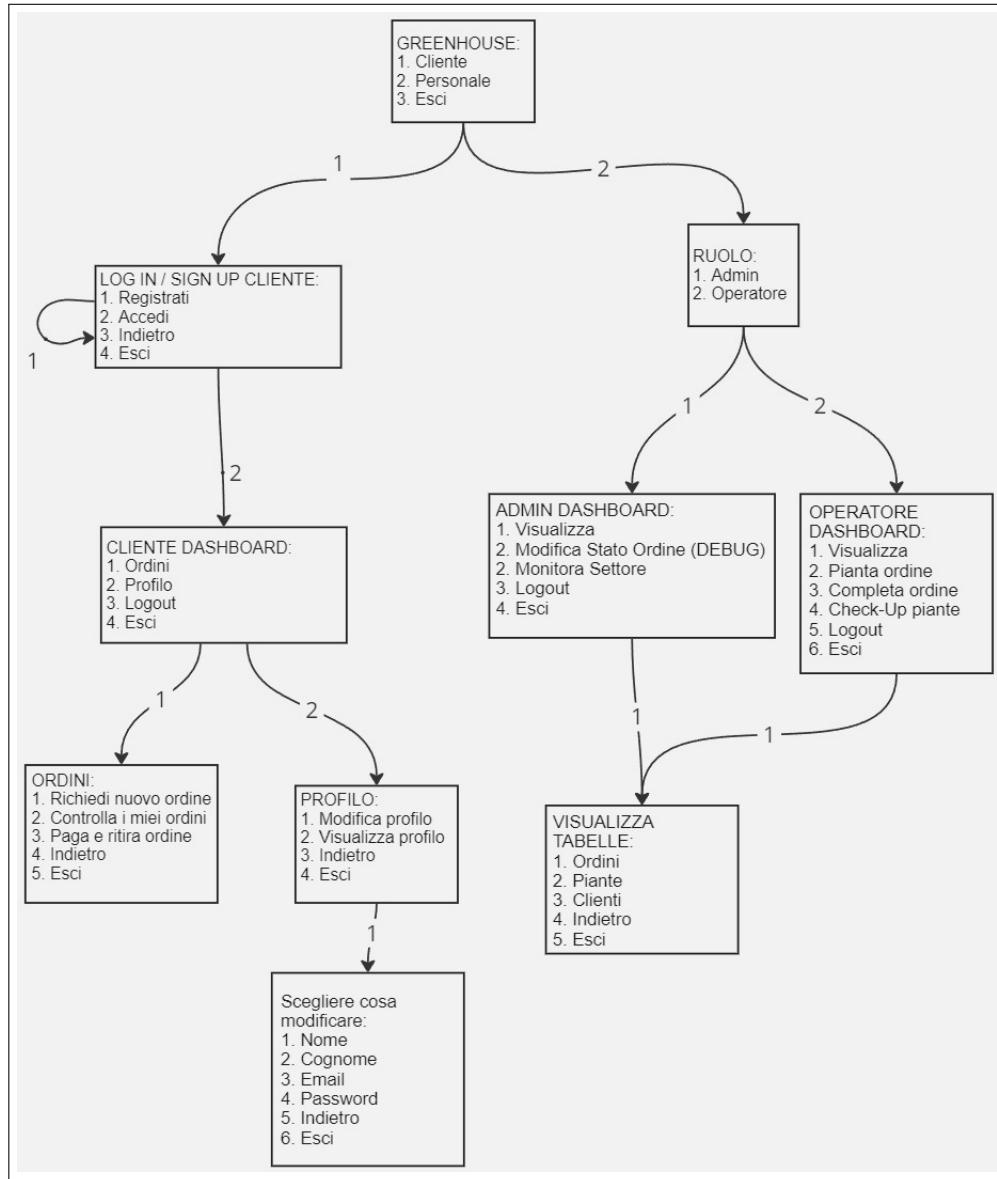


Figura 12: Navigation Diagram

3 Implementazione

Il codice è suddiviso in 3 "packages", che suddividono le classi in base alle loro funzionalità:

3.1 Domain Model

Contiene tutte le classi che rappresentano le entità del sistema e le relative funzioni associate.

3.1.1 Utente

La classe astratta Utente rappresenta un utente del sistema. Qui di seguito sono elencate le 3 classi che ereditano Utente e rappresentano gli agenti reali che accedono e interagiscono con il sistema stesso.

- **Cliente:** Rappresenta un Cliente dell'azienda, interessato a effettuare un acquisto di piante su ordinazione. I campi della classe sono `id`, `nome`, `cognome`, `email` e `password`.
- **Admin:** Identifica l'Admin dell'azienda e, come `Cliente`, contiene `id`, `nome`, `cognome`, `email` e `password`. La sua interfaccia gli permette di visualizzare ordini, piante e clienti e monitorare i settori dell'impianto.
- **Operatore:** Rappresenta un Operatore che lavora nell'azienda. È un utente, che quindi può accedere con le proprie credenziali, come un Utente qualsiasi, però può essere visto anche come un Attuatore con più funzionalità. Infatti può piantare un ordine, prepararlo quando è pronto oppure può controllare lo stato delle piante. I suoi attributi sono `id`, `nome`, `cognome`, `email`, `password` e `working`.

3.1.2 Pianta

Rappresenta una pianta presente nella serra, viene definita con `tipo` di Pianta, una `descrizione` di essa, la `dataInizio` in cui è stata piantata, lo `stato` attuale, il suo `costo` e il numero di `giorni` necessari alla sua crescita.

Il metodo `cura` effettua una cura (simulata) da parte di un Operatore nel caso in cui la Pianta ne necessiti.

3.1.3 Ordine

Identifica un `ordine` di certe piante effettuato da un `cliente` e i relativi dettagli come la `data di consegna`, il `prezzo totale`, un `id` univoco e lo `stato` attuale. I suoi metodi sono tutti getters e setters.

Lo `stato` dell'Ordine può essere:

- "da piantare": l'ordine è stato accettato dal sistema ed è stato preso in carico;
- "posizionato": le piante richieste nell'ordine sono state seminate nelle posizioni;
- "da completare": le piante sono cresciute e sono pronte per essere tolte dalle posizioni;
- "da ritirare": le piante possono essere ritirate dal cliente;
- "ritirato": l'ordine è stato pagato e ritirato dal cliente.

I passaggi di stato sono attuati attraverso precise funzioni del sistema (vedi Figura 13). In particolare `modificaStatoOrdine` permette di impostare uno stato arbitrario ed è utilizzata esclusivamente per "debug", però è indispensabile dato che la parte di simulazione della crescita della pianta e quindi del passaggio dell'ordine a stato "da completare" non è stata implementata.

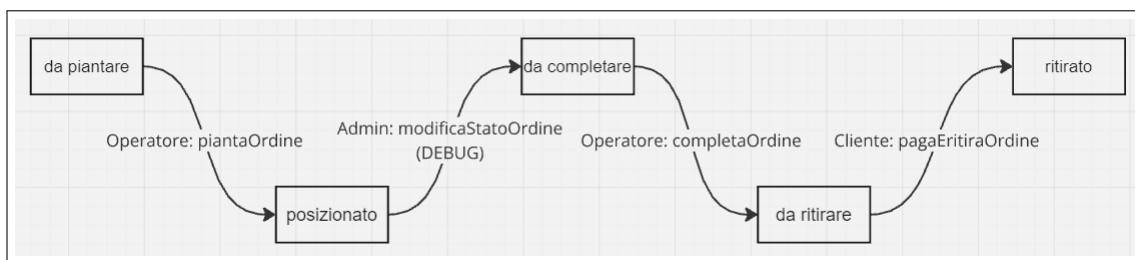


Figura 13: Diagramma degli Stati di un Ordine

3.1.4 Impianto

- **Spazio:** Rappresenta uno Spazio presente nella GreenHouse. È identificato da un **id** e contiene un insieme di Settori.
- **Settore:** Rappresenta una zona chiusa e isolata (fisicamente) avente al suo interno un **Termometro**, **IgrometroAria** e un **Fotosensore** che rilevano il "clima" all'interno. Sono presenti quindi i sensori e attuatori relativi a temperatura, umidità e luce. Contiene anche una **lista di Posizioni** e un **id**.
- **Posizione:** Identifica una Posizione in cui è stata piantata una singola Pianta. Contiene un proprio **Irrigatore** e un **IgrometroTerra**, un **id** univoco e due "flag" boolean **assegnata** e **occupata** che indicano lo stato della Posizione.
Una Posizione è "assegnata" quando un Ordine è preso in carico e ci verrà piantata una Pianta da un Operatore.
La flag "occupata" indica appunto che la semina è avvenuta e tale Posizione è occupata fisicamente.
- **Posizionamento:** Mapper con **id** che associa una Pianta con una Posizione e un Ordine.

3.1.5 Sensori

Classe astratta che definisce le proprietà e funzioni che deve avere un Sensore, ovvero un dispositivo capace di misurare certe grandezze fisiche di un Settore o di una Posizione (Pianta). Ha un **id** e misura un certo **valore**.

Le classi che ereditano questa classe astratta dovranno implementare il metodo **misura**.

- **Termometro:** Sensore che misura la temperatura (dell'aria) del Settore.
- **Fotosensore:** Sensore che misura l'intensità luminosa della luce presente nel Settore in cui è posto.
- **IgrometroAria:** Sensore che misura l'umidità presente nell'aria. È necessario e sufficiente soltanto uno di questi sensori per ogni Settore, in quanto le Posizioni (Piante) in esso condividono la stessa aria.
- **IgrometroTerra:** Sensore che misura l'umidità del terreno in cui è piantata una certa pianta. Ogni Posizione ha il suo.

3.1.6 Attuatori

Classe astratta che identifica un dispositivo in grado di effettuare azioni al fine di modificare parametri fisici del Settore o Posizione in cui esso è situato. Può essere attivato e disattivato con appositi metodi. Come per Sensore, le sottoclassi dovranno implementare il metodo astratto **esegui**.

- **Climatizzatore:** Attuatore che regola la temperatura e l'umidità dell'aria. È sufficiente un Climatizzatore per ogni settore.
- **Irrigatore:** Attuatore che irriga la Pianta presente nella Posizione in cui è situato tale dispositivo.
- **Lampada:** Attuatore che regola un livello di luminosità della luce di un Settore, per soddisfare il fabbisogno delle Piante.

3.2 Business Logic

Contiene tutte le classi e le relative funzionalità che hanno lo scopo di gestione delle varie entità del sistema.

3.2.1 LoginClienteController

Questa classe è utilizzata per effettuare il login del Cliente nel sistema. Ha come metodi `Accedi`, che verifica se le credenziali fornite sono valide, e `Registrati` che registra le credenziali di un nuovo cliente nel sistema.

3.2.2 LoginPersonaleController

Come `LoginClienteController`, si occupa del login del personale lavorativo della Greenhouse. Fa distinzione internamente tra i metodi `loginAdmin` e `loginOperatore`, anche se sono implementati in maniera molto simile.

3.2.3 ClienteController

La classe `ClienteController` definisce un’interfaccia per il cliente, che quindi, secondo gli Use Cases previsti, può:

- aggiornare i dati del proprio profilo, utilizzando `aggiornaProfilo`;
- richiedere un nuovo ordine, con `richiediNuovoOrdine`;
- pagare e ritirare un ordine, con `pagaEritiraOrdine`

È presente anche un altro metodo, `getOrdini`, che ottiene una lista di ordini associati a tale cliente ed eventualmente secondo altri criteri: questo serve quando è necessario mostrare gli ordini nell’interfaccia del programma.

3.2.4 OperatoreController

Questa classe presenta metodi che sono chiamati in maniera ”diretta” dall’operatore per svolgere alcuni compiti che gli competono, come:

- `piantaOrdine`: pianta un ordine selezionato se nello stato ”da piantare”;
- `completaOrdine`: completa un ordine e imposta il suo stato come ”da ritirare”;
- `checkupPiante`: simula il lavoro dell’operatore nel controllare manualmente le piante e ritorna una lista delle piante che necessitano di cure;
- `curaPianta`: simula la cura di una pianta da parte dell’operatore;
- `generaStatoPiante`: stabilisce randomicamente se una pianta ha bisogno di cure (simulando, come se dipendesse da agenti esterni dell’ambiente imprevedibili)
- getters di Clienti, Ordini e Piante utili per mostrare i relativi dati sull’interfaccia utente.

```

public ArrayList<Pianta> checkupPiante(Operatore operatore, int duration) {
    OperazioneDAO operazioneDAO = new OperazioneDAO();
    PiantaDAO piantaDAO = new PiantaDAO();
    PosizionamentoDAO posizionamentoDAO = new PosizionamentoDAO();
    ArrayList<Posizionamento> posizionamenti = posizionamentoDAO.get(new HashMap<>());
    ArrayList<Pianta> pianteDaCurare = new ArrayList<>();

    for (Posizionamento p : posizionamenti){
        Pianta pianta = piantaDAO.get(Map.of("id", p.getIdPianta())).get(0);
        String stato = pianta.getStato();
        System.out.printf("Stato %s [%d] -> %s\n", pianta.getTipoPianta(), pianta.getId(), stato);
        if(pianta.getStato().equals("ha bisogno di cure")){
            pianteDaCurare.add(pianta);
        }
    }
    try {
        Thread.sleep(duration);
    } catch (InterruptedException ignored) {
        return new ArrayList<>();
    }
}
operazioneDAO.registra(operatore, operatore.getLavoro(), LocalDateTime.now().toString());
return pianteDaCurare;
}

public void curaPianta(Pianta pianta, Operatore operatore) {
    PiantaDAO piantaDAO = new PiantaDAO();
    OperazioneDAO operazioneDAO = new OperazioneDAO();
    pianta.cura(operatore.getId(), LocalDateTime.now());
    piantaDAO.aggiorna(pianta.getId(), Map.of("stato", pianta.getStato()));
    piantaDAO.aggiornaDescrizione(pianta.getId(), pianta.getDescrizione());
    operazioneDAO.registra(operatore, operatore.getLavoro(), LocalDateTime.now().toString());
}

```

Snippet 1: metodi `checkupPiante()` e `curaPianta()` nella classe `OperatoreController`

3.2.5 AdminController

In questa classe è presente il metodo `monitoraSettore` chiamato quando viene avviato un monitoraggio da parte dell'Admin. Se richiesto è lanciato anche `monitoraPosizioniBySettoreId` che fornisce dati su sensori e attuatori relativi alle Posizioni del Settore desiderato.

Anche qui sono presenti getters utili alla visualizzazione dei dati.

```

public Settore monitoraSettore(int idSettore, LocalDateTime lt){
    SettoreDAO settoreDAO = new SettoreDAO();
    SensoreDAO sensoreDAO = new SensoreDAO();
    AttuatoreDAO attuatoreDAO = new AttuatoreDAO();

    Settore settore = settoreDAO.getById(idSettore);

    ArrayList<Sensore> sensori = settore.getSensori();
    for (Sensore sensore : sensori) {
        Attuatore attuatoreAssociato = settore.getAttuatoreAssociato(sensore);

        if (attuatoreAssociato != null) {
            // Ottieni il valore della misura
            float misuraVal = sensore.misura(lt, attuatoreAssociato.isWorking());
            sensoreDAO.aggiorna(sensore);

            // Verifica se il valore è nel range valido, altrimenti attiva l'attuatore
            if (settore.isSensorValueInRange(sensore.getTipoSensore(), misuraVal)) {
                attuatoreAssociato.esegui(0);
            } else {
                attuatoreAssociato.esegui(1);
            }
            attuatoreDAO.aggiorna(attuatoreAssociato);
        }
    }

    return settore;
}

```

Snippet 2: metodo `monitoraSettore()` nella classe `AdminController`

3.2.6 AdminExtraController

Infine con i metodi di AdminExtraController l'admin può fare il reset del database con `resetDatabase()` e inserire dei dati "default" con `defaultDatabase()`. Per la comunicazione con il database è utilizzata la classe AdminDAO.

```

public void resetDatabase() {
    StringBuilder sql_tmp = new StringBuilder();
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader("src/main/sql/reset.sql"))) {
        String line;
        while ((line = bufferedReader.readLine()) != null) { sql_tmp.append(line).append("\n"); }
    } catch (IOException e) {
        System.err.println("Errore nella lettura del file \"reset.sql\": " + e.getMessage());
        return;
    }
    String sql = sql_tmp.toString();
    AdminDAO adminDAO = new AdminDAO();
    adminDAO.resetDatabase(sql);
}

```

Snippet 3: metodo `resetDatabase()` nella classe `AdminExtraController`

3.3 ORM (Object-Relational Mapping)

Nel package `main.java.ORM` (percorso `src/main/java/ORM`) sono implementate le interfacce per la comunicazione del sistema con un database. Per una gestione più coerente dei dati. Le classi "DAO" che appartengono a questo package si occupano di formulare e lanciare QUERY, eseguite tramite JDBC.

3.3.1 ConnectionManager

La classe si occupa di gestire la connessione al database per tutte le classi DAO tramite il metodo `getConnection()`. Essendo implementata seguendo il design pattern del *Singleton* non è possibile che due DAO si collegino contemporaneamente al database e quindi che si verifichino perdite di dati. Inoltre questa classe contiene le informazioni esatte su l'URL, username e password per stabilire la connessione.

```
public class ConnectionManager {

    private static final String url = "jdbc:postgresql://localhost:5432/Greenhouse";
    private static final String username = "Greenhouse_admin";
    private static final String password = "admin";
    private static Connection connection = null;

    // singleton instance
    private static ConnectionManager instance = null;

    private ConnectionManager(){}

    public static ConnectionManager getInstance() {

        if (instance == null) { instance = new ConnectionManager(); }

        return instance;
    }

    public Connection getConnection() throws SQLException, ClassNotFoundException {

        Class.forName("org.postgresql.Driver");

        if (connection == null)
            try {
                connection = DriverManager.getConnection(url, username, password);
            } catch (SQLException e) {
                System.err.println("Error: " + e.getMessage());
            }
    }

    return connection;
}
}
```

Snippet 4: metodi `getInstance` e `getConnection()` nella classe `ConnectionManager`

3.3.2 ClienteDAO

La classe è preposta all'accesso al sistema tramite la funzione `accedi(email, password)` e alla registrazione (`registra(...)`) di un nuovo cliente. In più sono presenti i metodi per la modifica delle informazioni (`modificaAttributo(id_cliente, ...)`) e per restituire un cliente presente in database in base a criteri specificati (`get(criteri)`).

```
public class ClienteDAO {

    private Connection connection;

    public ClienteDAO(){
        try {
            this.connection = ConnectionManager.getInstance().getConnection();
        } catch (ClassNotFoundException | SQLException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }

    public boolean registra(String nome, String cognome, String email, String password) throws SQLException {
        String insertQuery = "INSERT INTO \"Cliente\" (nome, cognome, email, password) VALUES (?, ?, ?, ?) RETURNING id";
        boolean registrato = false;
        try (PreparedStatement statement = connection.prepareStatement(insertQuery)) {
            statement.setString(1, nome);
            statement.setString(2, cognome);
            statement.setString(3, email);
            statement.setString(4, password);
            ResultSet resultSet = statement.executeQuery();
            if (resultSet.next()) {
                registrato = true;
            }
        } catch (SQLException e) {
            System.err.println("Errore durante la registrazione del cliente: " + e.getMessage());
        }
        return registrato;
    }

    public Cliente accedi(String email, String password){
        String query = "SELECT * FROM \"Cliente\" WHERE email = ? AND password = ?";
        try (PreparedStatement statement = connection.prepareStatement(query)) {
            statement.setString(1, email);
            statement.setString(2, password);

            try (ResultSet rs = statement.executeQuery()) {
                if (rs.next()) {
                    return new Cliente(rs.getInt("id"), rs.getString("nome"),
                        rs.getString("cognome"), rs.getString("email"));
                }
            }catch (SQLException e) {
                System.err.println("Errore durante l'accesso del cliente: " + e.getMessage());
            }
        } catch (SQLException e) {
            System.err.println("Errore durante l'accesso del cliente: " + e.getMessage());
        }

        return null; // Se non viene trovato alcun cliente con le credenziali fornite, restituisci null
    }
}
```

Snippet 5: metodi `registra()` e `accedi()` nella classe `ClienteDAO`

3.3.3 AdminDAO

La classe svolge la funzione di accesso per l'admin come per il cliente ma ha anche i metodi `resetDatabase(sql)`, `createDatabase(ql)` e `generateDefaultDatabase(sql)` che eseguono il `reset.sql`, `schema.sql` e `default.sql` tali da pulire e resettare il database.

3.3.4 OperatoreDAO

OperatoreDAO permette anch'esso l'accesso al sistema con funzione `accedi(email,password)` con valore di ritorno un booleano. Ha anche un metodo `getById(id)` per restituire l'operatore a partire dall'Id richiesto.

3.3.5 SensoreDAO, AttuatoreDAO e OperazioneDAO

Le classi SensoreDAO e AttuatoreDAO restituiscono i rispettivi oggetti tramite `getById(id)` e hanno anche il metodo `aggiorna(sensore)` (e `aggiorna(attuatore)`) utilizzato per aggiornare i parametri delle tabelle. La classe OperazioneDAO permette di tenere un record delle azioni degli attuatori tramite la funzione `registra(attuatore, descrizione, data)`.

```
public void aggiorna(Sensore sensore) {
    String query = "UPDATE \"Sensore\" SET valore = ?, data = ?, tipo = ? WHERE id = ?";
    try (PreparedStatement pstmt = connection.prepareStatement(query)) {
        pstmt.setDouble(1, sensore.getValore());
        pstmt.setString(2, sensore.getData().toString());
        pstmt.setString(3, sensore.getTipoSensore());
        pstmt.setInt(4, sensore.getId());
        pstmt.executeUpdate();
    } catch (SQLException e) {
        System.err.println("Error: " + e.getMessage());
    }
}
```

Snippet 6: metodo `aggiorna()` nella classe `SensoreDAO`

3.3.6 SettoreDAO, PosizioneDAO

Mentre SettoreDAO ha solo due metodi per la creazione del settore (`getById(id)` e `get(criteri)`), PosizioneDAO non sono ha questi metodi ma anche metodi per assegnare, aggiornare e verificare se ci sono posizioni non assegnate.

3.3.7 OrdineDAO e PiantaDAO

OrdineDAO e PiantaDAO, come le altre DAO, hanno i metodi per inserire, aggiornare e restituire gli oggetti associati, inoltre PiantaDAO ha metodi per eliminare elementi, usato quando l'ordine è espletato.

```

public ArrayList<Pianta> get(Map<String, Object> criteri) {
    StringBuilder query = new StringBuilder("SELECT * FROM \"Pianta\"");

    // Aggiungi condizioni se ci sono criteri
    if (criteri != null && !criteri.isEmpty()) {
        query.append(" WHERE ");
        for (String key : criteri.keySet()) {
            query.append(key).append(" = ? AND ");
        }
        query.setLength(query.length() - 5); // Rimuove l'ultimo AND
    }

    ArrayList<Pianta> piante = new ArrayList<>();

    try (PreparedStatement statement = connection.prepareStatement(query.toString())) {
        // Imposta i parametri se ci sono criteri
        if (criteri != null && !criteri.isEmpty()) {
            int paramIndex = 1;
            for (Object value : criteri.values()) {
                statement.setObject(paramIndex, value);
                paramIndex++;
            }
        }

        // Esegui la query e gestisci il ResultSet
        try (ResultSet resultSet = statement.executeQuery()) {
            while (resultSet.next()) {
                piante.add(new Pianta(resultSet.getInt("id"),
                                      resultSet.getString("tipo"), resultSet.getString("descrizione"),
                                      resultSet.getString("dataInizio"),
                                      resultSet.getString("stato")));
            }
        }
    } catch (SQLException e) {
        System.err.println("Errore durante il recupero delle piante: " + e.getMessage());
    }

    return piante;
}

```

Snippet 7: metodo `get()` nella classe `PiantaDAO`

3.3.8 PosizionamentoDAO

La classe PosizionamentoDAO permette di ottenere i Posizionamenti richiesti dal database, aggiungerne di nuovi o eliminarli in base all'id dell'ordine di cui fanno parte (`snippet_eliminaposizionamentibyordine()`).

```
public boolean eliminaPosizionamentiByOrdine(int idOrdine){  
    String deleteQuery = "DELETE FROM \"Posizionamento\" WHERE ordine = ?";  
  
    try (PreparedStatement deleteStmt = connection.prepareStatement(deleteQuery)) {  
        connection.setAutoCommit(false);  
  
        // Imposta il parametro dell'ordine nella query  
        deleteStmt.setInt(1, idOrdine);  
  
        // Esegui la query di eliminazione  
        int affectedRows = deleteStmt.executeUpdate();  
  
        if (affectedRows > 0) {  
            // Eliminazione eseguita correttamente  
            System.out.println("Righe eliminate con successo: " + affectedRows);  
            connection.commit();  
            return true; // Imposta isDeleted a true se ci sono state righe eliminate  
        } else {  
            System.out.println("Nessun Posizionamento eliminato.");  
            connection.commit();  
            return false;  
        }  
    } catch (SQLException e) {  
        System.err.println("Errore durante l'eliminazione dei posizionamenti: " + e.getMessage());  
    }  
    return false;  
}
```

Snippet 8: metodo `eliminaPosizionamentiByOrdine` della classe `PosizionamentoDAO`

3.4 Database

Per mantenere coerenza tra i dati del sistema è stato utilizzato un database utilizzando PostgreSQL, con il quale è possibile interfacciarsi tramite le classi DAO. Tale database segue lo schema delle operazioni CRUD (CREATE, READ, UPDATE, DELETE).

Per inizializzare il database sono stati creati 3 file `.sql`:

- `reset.sql`: elimina tutte le tabelle del database (`DROP TABLE`);
- `schema.sql`: crea le tabelle del database definendo i vari attributi, chiavi e vincoli (`CREATE TABLE`);
- `default.sql`: inserisce valori nelle tabelle in modo da avere uno stato iniziale già compatibile con il programma (`INSERT INTO`).

Questi 3 file sono stati eseguiti, grazie ai metodi presenti in AdminController, in fase di debug del programma e sono utilizzati anche prima di ogni test (`BeforeEach`) per impostare condizioni ben precise richieste dai test.

```

CREATE TABLE IF NOT EXISTS "Cliente" (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(50),
    cognome VARCHAR(50),
    email VARCHAR(50),
    password varchar(50)
);
CREATE TABLE IF NOT EXISTS "Ordine" (
    id SERIAL PRIMARY KEY,
    cliente INT,
    dataConsegna VARCHAR(50),
    piante VARCHAR(300),
    totale DECIMAL(6,2),
    stato VARCHAR(50),
    FOREIGN KEY (cliente) REFERENCES "Cliente"(id)
);
CREATE TABLE IF NOT EXISTS "Pianta" (
    id SERIAL PRIMARY KEY ,
    tipo VARCHAR(50),
    descrizione VARCHAR(200),
    dataInizio VARCHAR(50),
    stato VARCHAR(100),
    costo DECIMAL(10, 2),
    ordine int references "Ordine"(id)
);

```

Snippet 9: creazione tabelle Cliente, Ordine e Pianta nel file schema.sql

3.5 Interfaccia

Ai fini di debug e test del programma è stata implementata un'interfaccia da terminale che permette di fare il login ed eseguire azioni/compiti sul sistema.

È possibile navigare tra i menu inserendo da tastiera il numero corrispondente all'opzione scelta.

Laddove è richiesto, l'utente deve fornire certe stringhe in input (ad esempio le credenziali di accesso per il login).

Il diagramma dei menu disponibili si trova nella sezione 2.6.

```

public static void handleClienteAction(Cliente cliente) throws SQLException {
    while (true) {
        int index = askForChooseMenuOption("      MAIN - CLIENTE ", new String[]{"ORDINI", "PROFILO", "Logout", "Esci"});
        if (index == 1)
            handleClientOrders(cliente);
        else if (index == 2)
            handleClienteProfile(cliente);
        else if (index == 3)
            return;
        else if (index == 4)
            System.exit(0);
    }
}

```

Snippet 10: metodo handleClienteAction che mostra il menu principale del Cliente

```

private static Ordine handleSceltaOrdineDaPiantare(OperatoreController operatoreController){
    while (true) {
        ArrayList<Ordine> ordiniDaPiantare = operatoreController.getOrdini(Map.of("stato", "da piantare"));
        if (ordiniDaPiantare.isEmpty()) {
            System.out.println("Non ci sono ordini da piantare!");
            return null;
        }
        printOrdini(ordiniDaPiantare);
        int idOrdine = askForInteger("ID Ordine da piantare: ");
        Ordine ord = operatoreController.getOrdineById(idOrdine);
        if (ord == null) {
            System.out.println("Nessun ordine trovato con questo id!");
        } else {
            if (ord.getStato().equals("da piantare"))
                return ord;
            else
                System.out.println("Quest'ordine non è da piantare!");
        }
    }
}

```

Snippet 11: metodo `handleSceltaOrdineDaPiantare`

4 Testing

Per verificare la correttezza e funzionalità del programma sono stati realizzati dei test (vedi cartella `test` in `/src`). Questi riguardano quelle funzioni associate agli use cases obiettivo del progetto, quindi principalmente la `BusinessLogic`, che è la parte più "ad alto livello" del programma e che contiene le funzioni direttamente chiamate dall'interfaccia.

È presente anche un test per il `DomainModel`.

La sezione di `ORM` non presenta test in quanto si occupa "soltanto" di comunicare con il `Database`.

4.1 Business Logic Test

Molti di questi test presentano più variazioni, in quanto verificano sia il caso in cui le condizioni per una certa funzione siano favorevoli, quindi è atteso un esito positivo ("Success"), sia il caso in cui vengano riscontrati problemi e quindi ci si aspetta un esito negativo ("Fail").

I test sono suddivisi in più classi (rispettando la suddivisione che hanno le relative funzioni):

`LoginClienteControllerTest`, `LoginPersonaleControllerTest`, `ClienteControllerTest`, `OperatoreControllerTest`

4.1.1 LoginClienteControllerTest

- `setUp`: reimposta il database a uno stato iniziale default;
- `registrationTest_Success`: esegue il test del metodo `registrati` di `loginClienteController`, utilizzata per la registrazione di un nuovo cliente;
- `registrationTest_Fail`: come `registrationTest_Success`, ma l'`email` con il quale si tenta di registrarsi appartiene a un cliente già presente nel database, quindi il test di aspetta un fallimento nella procedura;
- `loginTest_Success`: verifica la corretta funzionalità del metodo `accedi` che permette al cliente di accedere al sistema;

- `loginTest_Fail1` e `loginTest_Fail2`: verifica l'effettivo fallimento del login nel caso in cui la password o l'email non siano corrette;

```

LoginClienteController loginClienteController;
String nome, cognome, email, password;

@Test
public void registrationTest_Success() { //Nuovo cliente non presente nel db
    nome = "Giuseppe";
    cognome = "Verdi";
    email = "giuseppe@email.it";
    password = "123";
    assertTrue(loginClienteController.registrati(nome, cognome, email, password));
}

@Test
public void registrationTest_Fail(){ //Cliente già presente nel db
    nome = "Mario";
    cognome = "Rossi";
    email = "mario@email.it";
    password = "123";
    assertFalse(loginClienteController.registrati(nome, cognome, email, password));
}

@Test
public void loginTest_Success() { //Cliente presente nel db, credenziali corrette
    email = "mario@email.it";
    password = "123";
    assertNotNull(loginClienteController.accedi(email, password));
}

@Test
public void loginTest_Fail1() { //Cliente presente nel db, password errata
    email = "mario@email.it";
    password = "abc";
    assertNull(loginClienteController.accedi(email, password));
}

@Test
public void loginTest_Fail2() { //Cliente non presente nel db
    email = "luigi.conti@gmail.com";
    password = "1234567890";
    assertNull(loginClienteController.accedi(email, password));
}

```

Snippet 12: test di `LoginClienteControllerTest`

4.1.2 LoginPersonaleControllerTest

Analogo a LoginClienteControllerTest, effettua i test di accesso nel sistema, ma da parte del personale lavorativo (Admin e Operatore) e contiene i seguenti test: registrationTest_Fail, loginAdmin_Fail1, loginAdmin_Fail2, loginOperatore_Success, loginOperatore_Fail1 e loginOperatore_Fail2.

```
private LoginPersonaleController loginPersonaleController;

@BeforeEach
public void setUp(){
    loginPersonaleController = new LoginPersonaleController();
}

@Test
public void loginAdmin_Success() { // Login Admin presente nel db con credenziali corrette
    assertNotNull(loginPersonaleController.loginAdmin("elion", "123"));
}

@Test
public void loginAdmin_Fail1() { // Login Admin presente nel db con credenziali errate
    assertNull(loginPersonaleController.loginAdmin("elion", "abc"));
}

@Test
public void loginAdmin_Fail2(){ // Login Admin non presente nel db
    assertNull(loginPersonaleController.loginAdmin("giacomo", "456"));
}

@Test
public void loginOperatore_Success() { // Login Operatore presente nel db con credenziali corrette
    assertNotNull(loginPersonaleController.loginOperatore("ferrari@email.it", "123"));
}

@Test
public void loginOperatore_Fail1() { // Login Operatore presente nel db con credenziali errate
    assertNull(loginPersonaleController.loginOperatore("ferrari@email.it", "abc"));
}

@Test
public void loginOperatore_Fail2(){ // Login Operatore non presente nel db
    assertNull(loginPersonaleController.loginOperatore("verdi@email.it", "456"));
}
```

Snippet 13: test di LoginPersonaleControllerTest

4.1.3 ClienteControllerTest

Verifica il corretto funzionamento dei metodi di ClienteController, quindi delle funzionalità a disposizione dei Clienti. Sono testate le funzioni di aggiornamento dei dati del profilo, richiesta nuovo ordine (Success e Fail), pagamento e ritiro di un ordine (Success e Fail);

- **testAggiornaProfilo:** chiama il metodo aggiornaProfilo di ClienteController, e modifica il nome del cliente;
- **testRichiediNuovoOrdine_Success:** crea un nuovo ordine e invia la richiesta al sistema. Si aspetta che tale ordine sia accettato senza problemi;
- **testRichiediNuovoOrdine_Fail:** come il precedente, ma il numero di piante richiesto è superiore alla capacità dell'impianto, quindi la richiesta fallisce;
- **testPagaERitiraOrdine:** verifica che il metodo pagaERitiraOrdine di ClienteController funzioni correttamente.

```

private ClienteController clienteController;
Cliente cliente;

@Test
public void testAggiornaProfilo() { // Modifica un dato del profilo di un Utente
    boolean result = clienteController.aggiornaProfilo("nome", "Sergio");
    assertTrue(result);
}

@Test
public void testRichiediNuovoOrdine_Success() { // Crea un nuovo Ordine e lo richiede
    ArrayList<Pianta> piante = new ArrayList<>();
    for (int i = 0; i < 10; i++)
        piante.add(new Pianta("Rosa", "da piantare"));
    Ordine ordine = new Ordine(cliente.getId(), piante);
    boolean result = clienteController.richiediNuovoOrdine(ordine);
    assertTrue(result);
}

@Test
public void testRichiediNuovoOrdine_Fail() { // Crea un nuovo Ordine e lo richiede
    PosizioneDAO posizioneDAO = new PosizioneDAO();
    int posizioniDisponibili = posizioneDAO.getNNonAssegnate();
    ArrayList<Pianta> piante = new ArrayList<>();
    for (int i = 0; i < posizioniDisponibili + 1; i++) // non ci sono 100 posizioni libere
        piante.add(new Pianta("Rosa", "da piantare"));
    Ordine ordine = new Ordine(cliente.getId(), piante);
    boolean result = clienteController.richiediNuovoOrdine(ordine);
    assertFalse(result);
}

@Test
public void testPagaERitiraOrdine() { // Paga e ritira un ordine
    ArrayList<Pianta> piante = new ArrayList<>();
    for (int i = 0; i < 10; i++)
        piante.add(new Pianta("Rosa", "da piantare"));
    Ordine ordine = new Ordine(cliente.getId(), piante);
    clienteController.richiediNuovoOrdine(ordine);
    OrdineDAO ordineDAO = new OrdineDAO();
    ordineDAO.aggiorna(ordine.getId(), Map.of("stato", "da completare"));
    boolean result = clienteController.pagaEritiraOrdine(ordine);
    assertTrue(result);
}

```

Snippet 14: test di ClienteControllerTest

4.1.4 OperatoreControllerTest

- **completaOrdineTest_Success:** verifica che il completamento di un ordine venga eseguito correttamente;
- **completaOrdineTest_Fail:** prova ad eseguire il completamento di un ordine che non è ancora stato piantato, perciò si aspetta un fallimento;
- **checkupPianteTest_Success:** test del check-up delle piante eseguito da parte di un operatore. Alcune piante vengono (forzatamente) impostate come bisognose di cure e si verifica che queste vengano tutte rilevate durante il check-up;
- **checkupPianteTest_Fail:** nessuna pianta ha bisogno di cure (in base a questo setup), quindi il check-up non rileva piante bisognose;

- `curaPiantaTest`: chiama il metodo `curaPianta` di `OperatoreController` e ne verifica il funzionamento;
- `piantaOrdineTest`: testa la funzione utilizzata per piantare un ordine.

```

@Test
public void completaOrdineTest_Success(){ // Completa un ordine
    operatoreController.piantaOrdine(ordine, operatore);
    OrdineDAO ordineDAO = new OrdineDAO();
    ordineDAO.aggiorna(ordine.getId(), Map.of("stato", "da completare"));
    ordine.setStato("da completare");
    assertTrue(operatoreController.completaOrdine(ordine, operatore));
}
@Test
public void completaOrdineTest_Fail(){ // Completa un ordine (non ancora piantato)
    assertFalse(operatoreController.completaOrdine(ordine, operatore));
}
@Test
public void checkupPianteTest_Success(){ // Esegue un Check-Up dello stato delle piante
    operatoreController.piantaOrdine(ordine, operatore);
    ArrayList<Pianta> piante = ordine.getPiante();
    PiantaDAO piantaDAO = new PiantaDAO();
    for (Pianta pianta : piante) {
        pianta.setStatoHaBisogno(); // Alcune piante hanno bisogno di cure
        piantaDAO.aggiorna(pianta.getId(), Map.of("stato", pianta.getStato()));
    }
    ArrayList<Pianta> pianteDaCurare = operatoreController.checkupPiante(operatore, 0);
    assertEquals(pianteDaCurare.size(), piante.size());
}
@Test
public void checkupPianteTest_Fail(){ // Esegue un Check-Up dello stato delle piante
    operatoreController.piantaOrdine(ordine, operatore);
    ArrayList<Pianta> piante = ordine.getPiante();
    PiantaDAO piantaDAO = new PiantaDAO();
    for (Pianta pianta : piante) {
        pianta.setStatoNonHaBisogno(); // Nessuna pianta ha bisogno di cure
        piantaDAO.aggiorna(pianta.getId(), Map.of("stato", pianta.getStato()));
    }
    ArrayList<Pianta> pianteDaCurare = operatoreController.checkupPiante(operatore, 0);
    assertEquals(pianteDaCurare.size(), 0);
}
@Test
public void curaPiantaTest(){ // Cura una pianta
    Pianta pianta = ordine.getPiante().get(0);
    PiantaDAO piantaDAO = new PiantaDAO();
    pianta.setStatoHaBisogno();
    piantaDAO.aggiorna(pianta.getId(), Map.of("stato", pianta.getStato()));
    operatoreController.curaPianta(pianta, operatore);
    assertEquals(pianta.getStato(), "Curata, sta crescendo");
}
@Test
public void piantaOrdineTest(){
    assertTrue(operatoreController.piantaOrdine(ordine, operatore));
}

```

Snippet 15: test di `OperatoreControllerTest`

4.2 Domain Model Test

Per quanto riguarda il Domain Model è stato effettuato un solo test, in quanto sono presenti classi che fungono soltanto da entità, e generalmente sono passive, cioè prive di metodi e funzionalità.

I sensori però contengono un metodo `misura` che prende in ingresso l'orario e lo stato dell'attuatore associato a tale sensore e ritorna una misurazione del parametro fisico che gli compete. Questo simula l'influenza del tempo e degli attuatori sulla misurazione.

Nel caso in cui l'`Irrigatore` sia acceso (`testMisura_Irr_ON`), ci si aspetta un valore di umidità del terreno (misurato da `IgrometroTerra`) crescente nel tempo. Altrimenti (`testMisura_Irr_OFF`) il valore diminuisce (l'acqua viene assorbita dalla pianta). Il test verifica questo.

```
public class IgrometroTerraTest {  
    @Test  
    public void testMisura_Irr_ON(){  
        IgrometroTerra igrometroTerra = new IgrometroTerra(0, null, 10);  
        LocalDateTime lt = LocalDateTime.now();  
        float misura1 = igrometroTerra.misura(lt, false);  
        lt = lt.plusMinutes(1); // rimisura dopo 1 minuto  
        float misura2 = igrometroTerra.misura(lt, true);  
        assertTrue(misura2 > misura1);  
    }  
  
    @Test  
    public void testMisura_Irr_OFF(){  
        IgrometroTerra igrometroTerra = new IgrometroTerra(0, null, 10);  
        LocalDateTime lt = LocalDateTime.now();  
        float misura1 = igrometroTerra.misura(lt, false);  
        lt = lt.plusMinutes(1); // rimisura dopo 1 minuto  
        float misura2 = igrometroTerra.misura(lt, false);  
        assertTrue(misura2 <= misura1);  
    }  
}
```

Snippet 16: test di `IgrometroTerra`