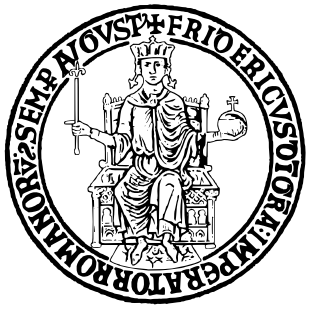
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA TRIENNALE IN INFORMATICA



PROGETTO D'ESAME DI OBJECT ORIENTATION

PROGETTAZIONE E SVILUPPO DI UN APPLICATIVO IN JAVA PER LA GESTIONE DI TO-DO PERSONALI

Docente: Prof. Porfirio Tramontana
Studenti: Simone Michele Mazzarelli (matr. N86004988)
Vito Santolo (matr. N86005309)

Anno Accademico 2024-2025

Questa pagina è stata lasciata intenzionalmente vuota.

Indice

Capitolo 1

Introduzione

1.1 Definizione del problema

Viene richiesto di realizzare un sistema informativo per la gestione di attività personali(ToDo), in maniera ottimizzata, attraverso un'organizzazione strutturata in bacheche. Il software, dovrà inoltre avere funzionalità di condivisione , per permettere agli utenti di condividere specifici ToDo con altri utenti selezionati . Per lo sviluppo dovrà essere utilizzata l'interfaccia grafica Swing, che facilita l'interazione utente attraverso componenti grafici intuitivi. Inoltre il progetto dovrà offrire un collegamento della parte applicativa con i database.

1.2 Obiettivi del sistema

Gli obiettivi principali del sistema ToDo Manager sono:

- Fornire un'interfaccia user-friendly per la gestione di attività personali
- Organizzare le attività in bacheche tematiche personalizzabili
- Permettere la condivisione di attività specifiche con altri utenti
- Gestire scadenze con indicatori visivi di urgenza
- Supportare la personalizzazione visiva delle attività
- Garantire persistenza dei dati attraverso database relazionale

1.3 Funzionalità principali

Il sistema ToDo Manager offre le seguenti funzionalità:
Gestione utenti: Sistema completo di registrazione e autenticazione con credenziali sicure.
Organizzazione in bacheche: Creazione e gestione di bacheche tematiche (Università, Lavoro, Tempo Libero) con possibilità di personalizzazione.
Gestione ToDo completa: Creazione, modifica, eliminazione e completamento di attività con supporto per scadenze, descrizioni, URL, immagini e colori personalizzati.

Condivisione collaborativa: Possibilità di condividere specifiche attività con altri utenti mantenendo la sincronizzazione.
Gestione scadenze: Monitoraggio automatico delle scadenze con indicatori visivi per attività urgenti o scadute.
Azioni multiple: Funzionalità per completare tutte le attività di una bacheca contemporaneamente.

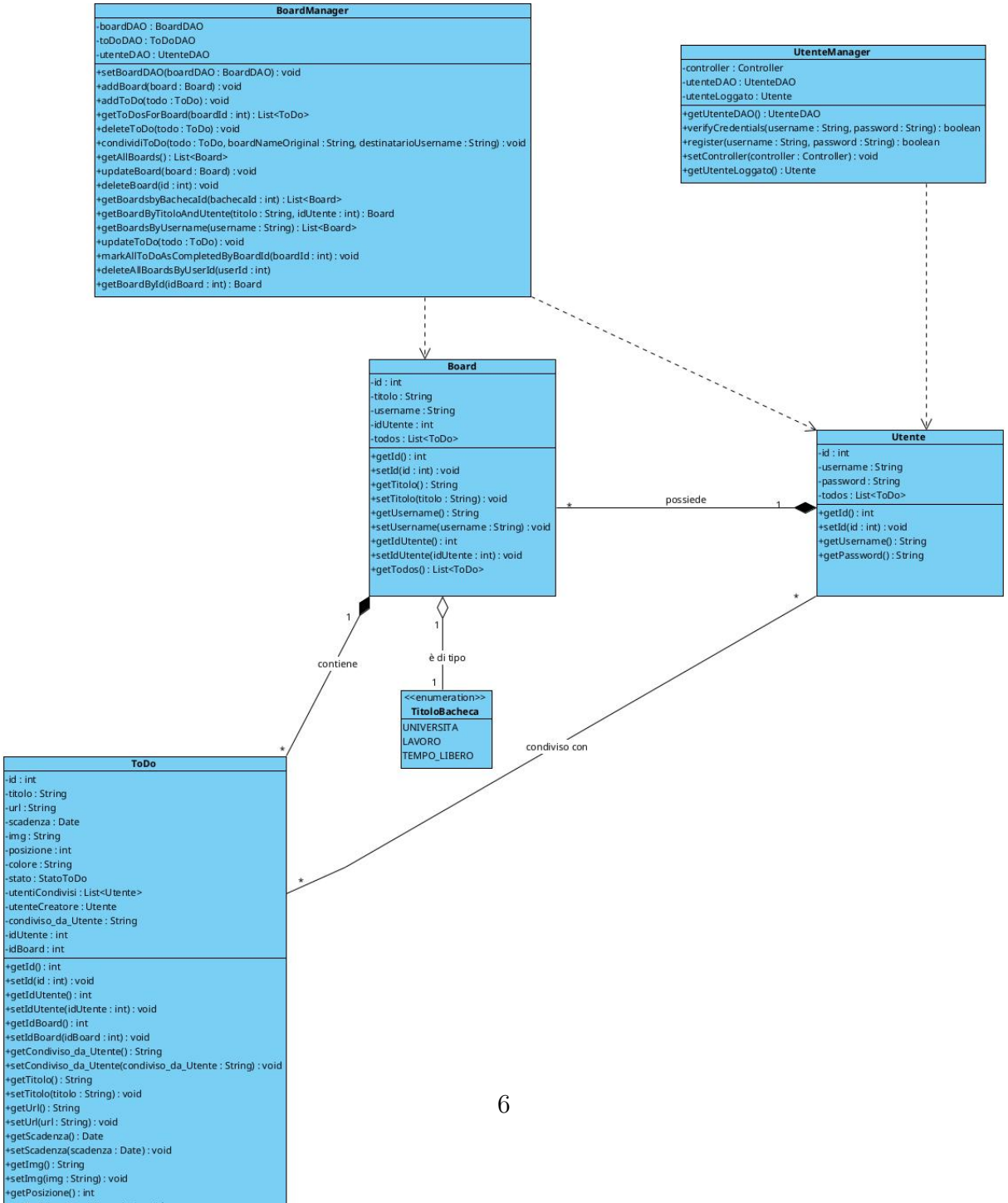
1.4 Link progetto

Il codice sorgente completo del progetto è disponibile al seguente repository GitHub:
`https://github.com/simo-git-crypto/Applicativo.git`

Capitolo 2

Schema del dominio

2.1 Class diagram del dominio



2.2 Descrizione del dominio

Il class diagram del dominio rappresenta le entità concettuali del problema da risolvere, identificate attraverso l'analisi dei requisiti del sistema di gestione ToDo.

2.2.1 Entità principali del dominio

Utente Rappresenta gli utenti del sistema, caratterizzati da:

- **id**: Identificatore univoco numerico
- **username**: Nome utente univoco per l'accesso al sistema
- **password**: Credenziale di autenticazione per la sicurezza

L'entità Utente costituisce il punto di partenza per tutte le operazioni del sistema, poiché ogni attività e bacheca è associata a un utente specifico.
Board (Bacheca) Rappresenta i contenitori organizzativi per le attività:

- **id**: Identificatore univoco numerico della bacheca
- **titolo**: Titolo della bacheca basato su enum TitoloBacheca
- **userId**: Riferimento all'utente proprietario della bacheca

Le bacheche fungono da categorie tematiche per organizzare logicamente i ToDo secondo criteri specifici come area di vita (Università, Lavoro, Tempo Libero).
ToDo L'entità centrale del dominio, rappresenta le attività da svolgere:
Identificatori:

- **id**: Identificatore univoco numerico del ToDo
- **userId**: Riferimento all'utente creatore
- **boardId**: Riferimento alla bacheca di appartenenza

Contenuto informativo:

- **titolo**: Titolo descrittivo dell'attività (campo obbligatorio)
- **descrizione**: Descrizione dettagliata dell'attività
- **url**: URL correlata all'attività per riferimenti esterni
- **immagine**: Percorso all'immagine associata per personalizzazione visiva

Metadata e gestione:

- **scadenza**: Data di scadenza per il completamento
- **colore**: Colore di sfondo personalizzabile per identificazione visiva
- **posizione**: Posizione ordinabile all'interno della bacheca
- **stato**: Stato di completamento basato su enum StatoToDo
- **condiviso_da_utente**: Riferimento per tracciare condivisioni

2.2.2 Enumerazioni del dominio

StatoToDo Enumeration che definisce gli stati possibili per un ToDo:

- `COMPLETATO`: Attività completata con successo
- `NON_COMPLETATO`: Attività ancora da completare

TitoloBacheca Enumeration che definisce i titoli predefiniti per le bacheche:

- `UNIVERSITÀ`: Bacheca dedicata ad attività accademiche
- `LAVORO`: Bacheca per attività professionali
- `TEMPO_LIBERO`: Bacheca per attività personali e ricreative

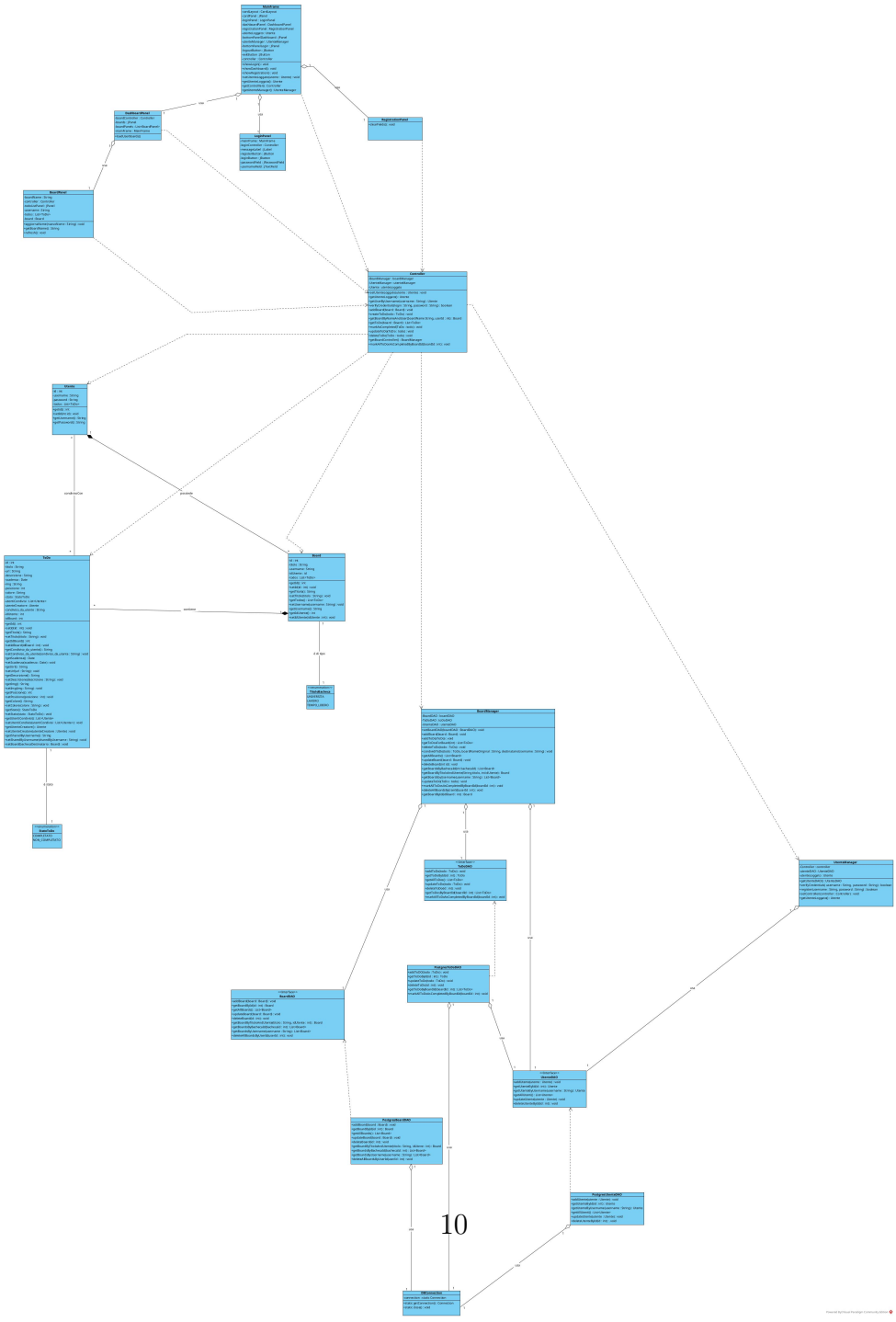
2.2.3 Relazioni del dominio

Il dominio presenta le seguenti relazioni fondamentali:
Utente-Board (1:N): Ogni utente può possedere multiple bacheche per organizzare le proprie attività in categorie diverse, mentre ogni bacheca appartiene esclusivamente a un singolo utente proprietario.
Board-ToDo (1:N): Ogni bacheca può contenere un numero illimitato di ToDo organizzati secondo un ordine personalizzabile, mentre ogni ToDo è associato a una singola bacheca per mantenere l'organizzazione tematica.
Utente-ToDo (1:N): Ogni utente può creare e gestire multiple attività, mentre ogni ToDo mantiene il riferimento al proprio creatore originale.
Condivisione ToDo (N:M): La relazione di condivisione è implementata attraverso il campo `condiviso_da_utente`, che permette di tracciare quando un ToDo viene condiviso da un utente con altri, mantenendo l'informazione sull'origine della condivisione.

Capitolo 3

Schema della soluzione

3.1 Class diagram della soluzione



3.2 Descrizione della soluzione

Il class diagram della soluzione mostra l'architettura implementativa del sistema, che trasforma le entità del dominio in una soluzione software completa e funzionale.

3.2.1 Modello BCE (Boundary-Control-Entity)

Il modello BCE (Boundary-Control-Entity) suddivide le classi del sistema in tre categorie principali:

- **Boundary (Confine):** classi che gestiscono l'interazione tra il sistema e gli attori esterni (ad esempio, l'interfaccia utente).
- **Control (Controllo):** classi che contengono la logica di coordinamento, gestione dei flussi e delle regole di business.
- **Entity (Entità):** classi che rappresentano i dati fondamentali e la logica di dominio persistente.

Nel progetto ToDo Manager, la suddivisione BCE è la seguente:

- **Boundary:**
  - `MainFrame`, `LoginPanel`, `RegistrationPanel`, `DashboardPanel`, `BoardPanel`
    Queste classi gestiscono l'interazione con l'utente tramite l'interfaccia grafica Swing.
- **Control:**
  - `Controller`, `UtenteManager`, `BoardManager`
    Queste classi coordinano le operazioni tra boundary ed entity, implementano la logica applicativa e orchestrano le interazioni tra le varie componenti.
- **Entity:**
  - `Utente`, `Board`, `ToDo`, `StatoToDo`, `TitoloBacheca`
    Queste classi rappresentano i dati persistenti e la logica di dominio del sistema.

Esempio di flusso BCE:
Quando un utente crea un nuovo ToDo tramite l'interfaccia grafica (`Boundary`), l'azione viene gestita dal `Controller` (`Control`), che si occupa di validare i dati e delegare la creazione al `BoardManager` (`Control`), il quale a sua volta crea l'oggetto `ToDo` (`Entity`) e ne gestisce la persistenza tramite i DAO.
Questa suddivisione favorisce la separazione delle responsabilità, la manutenibilità e la scalabilità del sistema, facilitando anche il testing delle singole componenti.

3.2.2 Funzionalità avanzate implementate

Sistema di condivisione:

- Utilizzo del campo `condiviso_da_utente` per tracciabilità delle condivisioni
- Logica di duplicazione intelligente che evita condivisioni duplicate
- Creazione automatica di bacheche di destinazione quando necessario
- Mantenimento della sincronizzazione tra tutti gli utenti coinvolti

Gestione posizionamento:

- Campo `posizione` per ordinamento personalizzabile dei ToDo
- Valore -1 per ToDo senza posizione specifica
- Interfaccia utente per modifica posizione tramite input numerico
- Riordinamento automatico in caso di conflitti di posizione

Gestione scadenze:

- Calcolo automatico dei giorni rimanenti alla scadenza
- Sistema di codifica colori per indicare urgenza delle attività
- Evidenziazione automatica di ToDo scaduti con colore rosso
- Notifiche visive per scadenze imminenti

Operazioni batch:

- Completamento simultaneo di tutti i ToDo di una bacheca
- Eliminazione di massa con richiesta di conferma utente
- Gestione transazionale per garantire coerenza dei dati
- Rollback automatico in caso di errori durante l'operazione

Capitolo 4

CRC Cards e Analisi delle Relazioni

4.1 CRC Cards

Le CRC (Class-Responsibility-Collaboration) cards definiscono le responsabilità di ogni classe e le sue collaborazioni con altre classi del sistema.

4.1.1 CRC Cards del Dominio

Tabella 4.1: CRC Card - Utente

Classe	**Responsabilità**	**Collaboratori**
Utente	- Gestire le credenziali di accesso - Mantenere informazioni identificative - Fornire autenticazione sicura	Board ToDo

Tabella 4.2: CRC Card - Board

Classe	**Responsabilità**	**Collaboratori**
Board	- Organizzare i ToDo per categoria - Mantenere il titolo tematico - Associarsi al proprietario - Fornire contenitore per attività	Utente ToDo TitoloBacheca

Tabella 4.3: CRC Card - ToDo

Classe	Responsabilità	Collaboratori
ToDo	- Gestire le informazioni dell'attività - Mantenere stato di completamento - Gestire scadenze e priorità - Supportare personalizzazione visiva - Tracciare condivisioni - Gestire posizionamento ordinabile	Utente Board StatoToDo

4.1.2 CRC Cards della Soluzione

Tabella 4.4: CRC Card - Controller

Classe	Responsabilità	Collaboratori
Controller	- Coordinare Model e View - Gestire sessione utente - Esporre operazioni di business - Orchestrare le operazioni CRUD - Mantenere stato applicazione	UtenteManager BoardManager Utente Board, ToDo

Tabella 4.5: CRC Card - UtenteManager

Classe	Responsabilità	Collaboratori
UtenteManager	- Gestire autenticazione utenti - Validare credenziali - Gestire registrazione - Coordinare operazioni utente	UtenteDAO Utente Controller

Tabella 4.6: CRC Card - BoardManager

Classe	Responsabilità	Collaboratori
BoardManager	- Gestire operazioni su bacheche - Gestire operazioni su ToDo - Implementare logica condivisione - Gestire operazioni batch - Coordinare persistenza - Gestire integrità referenziale	BoardDAO ToDoDAO Board, ToDo Utente Controller

Tabella 4.7: CRC Card - PostgresUtenteDAO

Classe	Responsabilità	Collaboratori
PostgresUtenteDAO	- Persistenza utenti PostgreSQL - Query database utenti - Ricerca e recupero dati - Ottimizzazione accessi	DBConnection Utente UtenteDAO

Tabella 4.8: CRC Card - MainFrame

Classe	Responsabilità	Collaboratori
MainFrame	- Gestire finestra principale - Coordinare navigazione pannelli - Gestire eventi interfaccia - Mantenere stato GUI	Controller LoginPanel DashboardPanel RegistrationPanel

4.2 Analisi delle Relazioni

Basandoci sui class diagram forniti, abbiamo identificato le seguenti relazioni accompagnate dalle motivazioni che ne giustificano la scelta:

4.2.1 Relazioni del Dominio

Utente → Board (Associazione)

- **Tipo**: Associazione
- **Motivazione**: Ogni bacheca appartiene a uno specifico utente. La classe Board contiene un riferimento all'id dell'utente proprietario, mentre la classe Utente può avere una lista di Board associate.
- **Implementazione**: Campo `userId` nella classe Board

Board → ToDo (Aggregazione)

- **Tipo**: Aggregazione
- **Motivazione**: L'aggregazione rappresenta una relazione logica di contenimento, ma con indipendenza dei cicli di vita. Nel modello corrente, Board e ToDo sono legati da una relazione di aggregazione perchè la Board "ha" dei ToDo, ma i ToDo possono anche vivere senza le Booard(se un ToDo è stato spostato o condiviso, esso sopravvivrà anche se l'utente cancellerà la bacheca o il suo account, e rimarrà nella sua nuova posizione)
- **Implementazione**: Campo `boardId` nella classe ToDo con vincoli di integrità referenziale

Utente → ToDo (Associazione)

- **Tipo**: Associazione
- **Motivazione**: Ogni ToDo deve tracciare il suo creatore originale , ma può esistere indipendentemente dall'utente(tramite condivisione). L'associazione permette di mantenere questa relazione senza dipendenza vitale.
- **Implementazione**: Campo `userId` nella classe ToDo

ToDo → StatoToDo (Associazione)

- **Tipo**: Associazione
- **Motivazione**: Ogni ToDo ha un attributo enum StatoToDo(completato/non completato) che ne definisce la condizione attuale del ciclo di vita. L'associazione ci dà la possibilità di tracciare e gestire il progresso delle attività, consentendo operazioni come il cambio di stato o il filtraggio dei ToDo per stato ed è ideale poichè StatoToDo non contiene e non gestisce oggetti ToDo
- **Implementazione**: Campo `stato` di tipo enum StatoToDo

Board → TitoloBacheca (Associazione)

- **Tipo**: Associazione
- **Motivazione**: Ogni board possiede un titolo predefinito che ne identifica il tipo e il contesto di utilizzo. L'associazione garantisce che le bacheche abbiano denominazioni uniformi e coerenti, facilitando l organizzazione e la categorizzazione delle diverse board nel sistema. Questo tipo di relazione rappresenta il legame strutturale e permanente(poichè rimane costante) tra la bacheca e il suo titolo.
- **Implementazione**: Campo `titolo` di tipo enum TitoloBacheca

4.2.2 Relazioni del modello della Soluzione

Controller → UtenteManager (Associazione)

- **Tipo**: Associazione
- **Motivazione**: Il controller necessità di accedere ai servizi di gestione degli utenti per processare le richieste relative alle operazioni sugli utenti. Quest'associazione consente al Controller di utilizzare l'UtenteManager per eseguire le operazioni specifiche sugli utenti, mantenendo una separazione chiara delle responsabilità secondo l'architettura BCE. L'UtenteManager fornisce i servizi specializzati per la gestione degli utenti che il Controller può invocare durante l'elaborazione delle richieste.
- **Implementazione**: Campo `private final UtenteManager utenteManager`

Controller → BoardManager (Aggregazione)

- **Tipo**: Aggregazione
- **Motivazione**:Si tratta di aggregazione poiché il Controller mantiene un riferimento al BoardManager per delegare operazioni specifiche sui board e ToDo, ma il BoardManager può esistere indipendentemente dal Controller.
- **Implementazione**: Campo `private final BoardManager boardManager`

UtenteManager → UtenteDAO (Aggregazione)

- **Tipo**: Aggregazione
- **Motivazione**: UtenteManager ha una dipendenza verso UtenteDAO per delegare le operazioni di persistenza degli utenti. UtenteDAO può esistere e funzionare indipendentemente da UtenteManager, rendendo questa una relazione di aggregazione
- **Implementazione**:
  - Nel costruttore di UtenteManager (linea dove viene accettato il parametro UtenteDAO utenteDAO)
  - Campo privato utenteDAO nella classe UtenteManager per mantenere il riferimento
  - Metodo getUtenteDAO() che restituisce l'istanza di UtenteDAO
  - Utilizzo di utenteDAO nei metodi verifyCredentials() e register() per delegare le operazioni al DAO

UtenteManager → Utente (Dipendenza)

- **Tipo**: Dipendenza
- **Motivazione**: UtenteManager utilizza oggetti di tipo Utente nei suoi metodi per gestire la logica di autenticazione, registrazione e mantenimento dello stato dell'utente correntemente loggato. La classe dipende da Utente per fornire i suoi servizi ma non mantiene una relazione strutturale permanente.
- **Implementazione**:
  - **Campo utenteLoggato di tipo Utente per mantenere il riferimento all'utente attualmente autenticato**
  - **Metodo verifyCredentials() che recupera e utilizza oggetti utente dal DAO**
  - **Metodo register() che crea nuove istanze di Utente**
  - **Metodo getUtenteLoggato() che restituisce l'istanza di Utente corrente.**

UtenteDAO → BoardManager (Aggregazione)

- **Tipo**: Aggregazione
- **Motivazione**: BoardManager mantiene un riferimento persistente a UtenteDAO per delegare operazioni relative agli utenti (come recuperare utenti per username nella condivisione di ToDo), ma UtenteDAO può esistere indipendentemente da BoardManager.
- **Implementazione**:
  - Campo privato `private UtenteDAO utenteDAO` nella classe BoardManager
  - Iniezione tramite costruttore `BoardManager(BoardDAO boardDAO, ToDoDAO toDoDAO, UtenteDAO utenteDAO)`
  - Utilizzo nel metodo `condividiToDo()` per recuperare l'utente destinatario tramite `utenteDAO.getUtenteByUsername()`
  - Delega delle operazioni specifiche sugli utenti al DAO appropriato

BoardManager → BoardDAO (Dipendenza)

- **Tipo**: Dipendenza
- **Motivazione**: BoardManager necessita di accedere e manipolare i dati delle board presenti nel database. Per farlo fa uso di un oggetto che nasconde i dettagli delle operazioni CRUD sulle board , separando così la logica di bussiness dalla logica di accesso ai dati
- **Implementazione**: Utilizzo dei DAO attraverso interfacce

BoardManager → ToDoDAO (Dipendenza)

- **Tipo**: Dipendenza
- **Motivazione**: Boardmanager dipende da ToDoDAO per eseguire operazioni CRUD sui ToDo, ma non ne gestisce il ciclo di vita e le proprietà concettuali.
- **Implementazione**: BoardManager contiene un attributo di tipo TodoDao che rappresenta il collegamento fra le due classi

PostgresUtenteDAO → DBConnection (Dipendenza)

- **Tipo**: Dipendenza
- **Motivazione**:PostGresUtenteDAO necessita di una connessione attiva al database per eseguire operazioni sugli utenti, ma non possiede ne contiene una DBconnection come attributo proprietario. PostgresUtenteDAO dipende da DBConnection solo per il tempo necessario a svolgere le sue operazioni.
- **Implementazione**: I metodi di PostgresUtenteSAO invocano i metodi statici di DBConnection ogni volta che è necessaria una connessione al database

UtenteDAO → PostgresToDoDAO (Dipendenza)

- **Tipo**: Dipendenza
- **Motivazione**:PostgresToDoDAO necessita di UtenteDAO per popolare il campo 'utenteCreatore' degli oggetti ToDo durante il recupero del database. Quando viene recuperato un ToDo, è necessario risolvere l'ID Utente nel relativo oggetto Utente completo.
- **Implementazione**:PostgresToDoDAO riceve un'istanza di UtenteDAO nel costruttore e la utilizza nei metodi 'getToDoById()', 'getAllToDos()' e 'getToDosByBoardId()' per chiamare 'utenteDAO.getUtenteById(idUtenteCreatore)' e settare correttamente l'oggetto 'utenteCreatore' nel ToDo.

PostgresBoardDAO → DBConnection (Dipendenza)

- **Tipo**: Dipendenza
- **Motivazione**:PostGresBoardDAO utilizza DBconnection solo quando serve una connessione, senza mantenere un riferimento stamile come attributo
- **Implementazione**: I metodi di PostgresBoardDAO invocano i metodi statici di DBConnection ogni volta che è necessaria una connessione al database.

PostgresBoardDAO → BoardDAO (Realizzazione)

- **Tipo**: Realizzazione
- **Motivazione**: PostgresBoardDAO fornisce implementazioni concrete per tutti i metodi dichiarati nell'interfaccia BoardDAO.
- **Implementazione**: La classe PostgresBoardDAO implementa tutti i metodi definiti nell'interfaccia BoardDAO

PostgresToDoDAO → DBConnection (Dipendenza)

- **Tipo**: Dipendenza
- **Motivazione**:PostGresToDoDAO riceve DBconnection come parametro nel costruttore e la mantiene come attributo privato per eseguire operazioni CRUD sui ToDo nel database
- **Implementazione**:La classe ha un attibuto 'private final connection' inizializzato nel costruttore, e tuttii metodi utilizzano questa connessione per le operazioni SQL

PostgresToDoDAO → ToDoDAO (Realizzazione)

- **Tipo**: Realizzazione
- **Motivazione**:La classe PostgresToDoDAO implementa l'interfaccia ToDoDAO per fornire un'implementazione concreta delle operazioni CRUD sui ToDo utilizzando PostgreSQL come database.
- **Implementazione**:La classe PostgresToDoDAO implementa tutti i metodi definiti nell'interfaccia ToDoDAO:
  - **addToDo(ToDo todo)** - Inserisce un nuovo ToDo nel database PostgreSQL utilizzando una query INSERT con RETURNING per ottenere l'ID generato automaticamente
  - **getToDoById(int id)** - Recupera un ToDo specifico tramite il suo ID utilizzando una query SELECT con PreparedStatement
  - **getAllToDos()** - Recupera tutti i ToDo dal database eseguendo una query SELECT senza condizioni
  - **updateToDo(ToDo todo)** - Aggiorna un ToDo esistente utilizzando una query UPDATE con tutti i campi modificabili
  - **deleteToDo(int id)** - Elimina un ToDo dal database tramite una query DELETE
  - **getToDosByBoardId(int boardId)** - Recupera tutti i ToDo appartenenti a una specifica board
  - **markAllToDosAsCompletedByBoardId(int boardId)** - Marca tutti i ToDo di una board come completati

MainFrame → Controller (Associazione)

- **Tipo**: Associazione
- **Motivazione**: MainFrame mantiene un collegamento stabile a Controller per delegare la gestione degli eventi e delle operazioni richieste dall'interfaccia grafica, separando la logica di presentazione dalla logica applicativa
- **Implementazione**: Campo `private Controller controller`

MainFrame → LoginPanel (Composizione)

- **Tipo**: Composizione
- **Motivazione**: MainFrame è responsabile della creazione e del ciclo di vita di LoginPanel. LoginPanel non può esistere senza MainFrame e viene creato direttamente nel costruttore di MainFrame. La relazione inversa(LoginPanel che mantiene un riferimento a MainFrame) è necessaria per la comunicazione, ma non cambia il fatto che MainFrame possiede LoginPanel.
- **Implementazione**:MainFrame crea l'istanza di LoginPanel nel suo costruttore e la mantiene come attributo privato. LoginPanel riceve il riferimento a MainFrame per poter invocare metodi come showDashboard() o showRegistration().

MainFrame → RegistrationPanel (Composizione)

- **Tipo**: Composizione
- **Motivazione**: MainFrame è responsabile della creazione e gestione completa del ciclo di vita di RegistrationPanel. Il pannello di registrazione esiste solo come parte integrante del frame principale e viene creato durante l'inizializzazione di MainFrame. Quando MainFrame viene distrutto , anche RegistrationPanel cessa di esistere, stabilendo una dipendenza del ciclo di vita forte e unidirezionale.
- **Implementazione**: Nel costruttore di MainFrame, viene istanziato RegistrationPanel passando il riferimento a se stesso come parametro. Il pannello viene poi aggiunto al CardLayout con l'identificativo "register" per permettere la navigazione tra le diverse schermate dell'applicazione.

MainFrame → DashboardPanel (Composizione)

- **Tipo**: Composizione
- **Motivazione**: MainFrame possiede completamente l'istanza di DasboardPanel, ne controlla completamente la creazione e la distruzione e DashboardPanel richiede mainframe per esistere.
- **Implementazione**:
  - **Dichiarazione del campo**: `private DashboardPanel dashboardPanel;` - MainFrame mantiene un riferimento privato all'istanza
  - **Creazione controllata**: Nel metodo `showDashboard()`, MainFrame crea una nuova istanza ogni volta che necessario: `dashboardPanel = new DashboardPanel(this);`
  - **Distruzione controllata**: Prima di creare una nuova istanza, MainFrame rimuove esplicitamente quella precedente:
    ```
    if (dashboardPanel != null)
    {
    cardPanel.remove(dashboardPanel);
    }
    ```
  - **Dipendenza nel costruttore**: DashboardPanel richiede obbligatoriamente MainFrame come parametro: `public DashboardPanel(MainFrame mainFrame)`
  - **Reference injection**: DashboardPanel mantiene un riferimento a MainFrame: `this.mainFrame = mainFrame;`
  - **Uso intensivo dei servizi**: DashboardPanel accede costantemente ai metodi di MainFrame (`getUtenteLoggato()`, `getController()`) per funzionare
  - **Ciclo di vita dipendente**: La vita di DashboardPanel è completamente legata a quella di MainFrame - quando MainFrame viene distrutto, anche DashboardPanel cessa di esistere automaticamente

DashboardPanel → BoardPanel (Composizione)

- **Tipo**: Composizione
- **Motivazione**: DashBoardPanel è responsabile della creazione e distruzione degli oggetti contenuti in BoardPanel, che non possono esistere indipendentemente da BoardPanel
- **Implementazione**:
  - DashboardPanel crea istanze di BoardPanel nel metodo loadUserBoards()
  - Queste istanze sono memorizzate nella collezione boardPanels
  - Quando DashboardPanel aggiorna la visualizzazione, rimuove tutti i BoardPanel esistenti e ne crea di nuovi
  - Quando boards.removeAll() viene chiamato, i riferimenti ai BoardPanel vengono rimossi e questi oggetti diventano idonei per la garbage collection
  - DashboardPanel passa al BoardPanel i dati necessari (controller, nome della bacheca, username, ecc.)
  - BoardPanel opera sui ToDo associati alla bacheca specifica ma sempre sotto il controllo del DashboardPanel che lo contiene
  - Ogni BoardPanel viene completamente configurato e inizializzato dal DashboardPanel
  - La visualizzazione del BoardPanel è gestita attraverso il layout impostato dal DashboardPanel
  - Quando avvengono modifiche ai dati, DashboardPanel chiama loadUserBoards() che ricrea tutti i BoardPanel.

Mainframe → UtenteManager (Aggregazione)

- **Tipo**: Aggregazione
- **Motivazione**: L'aggregazione tra MainFrame e UtenteManager esiste perchè MainFrame ha bisogno di gestire l'autenticazione e la registrazione degli utenti attraverso i suoi pannelli, ma non è responsabile del ciclo di vita di UtenteManager.
- **Implementazione**:
  - **Campo di istanza in MainFrame** : private UtenteManager utenteManager;
  - **Inizializzazione nel costruttore** :
    public MainFrame(Controller controller, UtenteManager utenteManager){
    this.utenteManager = utenteManager; // ... resto dell'inizializzazione
    }
  - **Metodo getter per l'accesso**:
    public UtenteManager getUtenteManager() {
    return utenteManager;
    }
  - **Utilizzo nei pannelli GUI**: Nel RegistrationPanel, MainFrame fornisce l'accesso a UtenteManager:
    public RegistrationPanel(MainFrame mainFrame) {
    this.mainFrame = mainFrame;
    this.utenteManager = mainFrame.getUtenteManager();
    // ... }
  - **Delega delle operazioni**: I pannelli utilizzano UtenteManager per le operazioni di autenticazione e registrazione:
    // In RegistrationPanel
    if (utenteManager.register(username, password)) {
    // registrazione riuscita
    }

Capitolo 5

Implementazione dell'interfaccia utente

5.1 Architettura GUI

L'interfaccia utente è stata progettata seguendo principi di usabilità e accessibilità, utilizzando Java Swing con un approccio modulare che facilita manutenzione ed estensioni future.

5.1.1 Gestione della navigazione

Il `MainFrame` utilizza CardLayout per gestire la transizione fluida tra i diversi pannelli dell'applicazione, mantenendo lo stato dell'utente e coordinando le interazioni con il Controller.

5.1.2 Autenticazione e registrazione

I pannelli di login e registrazione implementano validazione in tempo reale delle credenziali, con feedback immediato per guidare l'utente nell'inserimento corretto dei dati.

5.1.3 Dashboard principale

Il `DashboardPanel` costituisce il centro operativo dell'applicazione, fornendo accesso rapido a tutte le funzionalità principali attraverso un'interfaccia intuitiva e organizzata.

5.2 Gestione delle bacheche

Ogni `BoardPanel` implementa una visualizzazione dedicata per i ToDo, con indicatori visivi per stato, scadenze e priorità, facilitando la gestione quotidiana delle attività.

Capitolo 6

Conclusioni

6.1 Obiettivi raggiunti

Il progetto ToDo Manager ha raggiunto con successo tutti gli obiettivi prefissati, implementando una soluzione completa e professionale per la gestione collaborativa di attività personali.

6.2 Punti di forza della soluzione

La soluzione presenta diversi punti di forza significativi:

- **Astrazione mediante interfacce DAO**: Le interfacce DAO(UtenteDAO, BoardDAO, ToDoDAO)rappresentano specifiche funzionali che definiscono le operazioni essenziali per la gestione dei dati, astraendo completamente idettagli implementativi. Questo livello di astrazione crea un'interfaccia uniforme per le operazioni CRUD(Create, Read, Update, Delete) nascondendo la complessità dell'accesso ai dati

- **Polimorfismo e Disaccoppiamento**: In questo progetto viene sfruttato il polimorfismo, ovvero, la capacità di oggetti di classi diverse di rispondere alla stessa interfaccia. al fine di realizzare un'architettura robusta e flessibile. Questo principio consente a classi come UtenteManager di interagire con astrazioni(UtenteDAO) invece di dipendere da implementazioni concrete(UtenteDAOPostgres), creando così un forte disaccopiamento tra i livelli dell'applicazione. Attraverso la gestione delle dipendenze, è possibile fornire a UtenteManager qualsiasi implementazione che rispetti l'interfaccia, permettendo di scambiare l'implementazione reale con alternative senza modificare il codice.

- **Architettura a strati e separazione delle responsabilità**: L'architettura del progetto segue il modello BCE(Boundary - Control - Entity), definendo una netta distinzione delle responsabilità su tre livelli distinti. Le classi Boundary(quelle del package GUI) si occupano esclusivamente dell'interfaccia utente, raccogliendo dati in input e restituendo risultati in output senza contenere logica applicativa. Le classi Control(Controller, UtenteManager, BoardManager), fungono da responsabili funzionali, gestendo le operazioni tra i livelli Boundary ed Entity e implementando le regole di dominio del sistema.

- **Incapsulamento e protezione dei dati**: Le classi del modello come ToDo, Utente e Board implementano un corretto incapsulamento attraverso campi privati e metodi getter/setter pubblici, proteggendo l'integrità dei dati e fornendo un'interfaccia controllata per l'accesso e la modifica degli attributi.

Questi elementi rendono il sistema robusto, mantenibile e pronto per evoluzioni future.

6.3 Sviluppi futuri

Il sistema fornisce una base solida per future evoluzioni verso una piattaforma di produttività più ampia e integrata, con possibilità di estensione per applicazioni desktop e mobili native.