Applicativo per un Gestore di Immagini con Filtri Integrati

Francesco Moschettini, Lorenzo Dinardo

Giugno 2025

Sommario

Questo è il documento di progetto riguardante la realizzazione di un applicativo software per la gestione di un marketplace contenente immagini, con filtri integrati applicabili direttamente sui post.

Indice

1 Analisi dei Requisiti

1.1 Statement

L'applicativo progettato ha come obiettivo la gestione strutturata e interattiva di un database di immagini, integrando funzionalità di autenticazione, manipolazione grafica tramite filtri digitali e pubblicazione di contenuti multimediali. Il sistema è articolato secondo un'architettura a livelli, comprendente componenti per la gestione dell'accesso ai dati, della logica applicativa e dell'interfaccia utente testuale.

Gli utenti possono autenticarsi nel sistema e, a seconda del ruolo (*OSSERVATORE* o *AUTORE*), accedere a funzionalità differenziate. Gli *AUTORI* possono caricare immagini, applicare filtri (singoli o compositi) tramite una struttura modulare basata sull'interfaccia `Filter`, e pubblicare i risultati sotto forma di post. Ogni post può essere arricchito da descrizioni testuali, ricevere apprezzamenti (*like*) e commenti da parte degli altri utenti.

Le immagini caricate vengono salvate nel database, in formato binario, ma possono anche essere elaborate mediante filtri quali sfocatura, contrasto, scala di grigi, seppia, inversione, nitidezza e composizioni multiple. La gestione dei filtri avviene attraverso un registro centralizzato (`FilterRegistry`) che consente l'aggiunta e l'elencazione dei filtri disponibili.

I dati persistenti del sistema (utenti, post, commenti) sono gestiti tramite un sistema relazionale, con accesso mediato da DAO specifici (`UserDao`, `PostDao`, `CommentDao`) interfacciati con il database tramite JDBC.

Il progetto include un sistema di validazione degli input, gestione delle eccezioni, formattazione temporale dei post e serializzazione delle immagini tramite utility interne.

1.2 Architettura e pratiche utilizzate

Il software `Image-Manager` 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 l'interazione diretta con esso.

Per mantenere una separazione delle responsabilità, la struttura del progetto è stata divisa in più parti (package) principali che riflettono un'architettura a livelli. Questi package si occupano in modo distinto della logica di presentazione (interfaccia utente testuale), della logica di controllo, della logica di business (o applicativa), della rappresentazione dei dati del dominio (modello) e dell'accesso ai dati. Nello specifico:

- **Presentation Layer (Interfaccia Utente CLI):** Contenuta principalmente nel package `main` (classe `Main.java`), gestisce l'interazione diretta con l'utente tramite la riga di comando.
- **Controller Layer:** Il package `controller` (con classi come `AuthController`, `ImageController`, `PostController`) agisce da intermediario, ricevendo gli input dell'utente dall'interfaccia e invocando le operazioni appropriate nel Service Layer.

- **Service Layer (Business Logic):** Il package `service` (con le interfacce) e `service.impl` (con le implementazioni come `AuthServiceImpl`, `ImageServiceImpl`, `PostServiceImpl`) contengono le classi che implementano la logica di business del sistema, orchestrando le operazioni e le validazioni.
- **Domain Model:** Il package `model` (con classi come `User`, `Post`, `Image`, `Comment`) contiene le classi che rappresentano le entità e i dati fondamentali del sistema.
- **Data Access Layer (DAO con JDBC):** Questo strato, composto dal package `dao` (interfacce come `UserDao`, `PostDao`) e `dao.impl` (implementazioni come `UserDAOImpl`, `PostDAOImpl`), implementa il pattern Data Access Object. Le classi DAO si occupano dell'accesso diretto al database PostgreSQL tramite JDBC, eseguendo query SQL e mappando i risultati agli oggetti del Domain Model. Questo approccio permette di rendere i dati persistenti e recuperarli dal database senza l'utilizzo di un framework ORM.
- **Utility Packages:** Package come `util` (per `DatabaseConnection` e `ImageUtils`), `filter` (per la logica di applicazione dei filtri alle immagini) e `exception` forniscono funzionalità di supporto trasversali.

Gli Use Case Diagram e i Class Diagram (che verranno discussi in seguito) 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. Questa architettura prevede che l'utente esegua un'azione tramite l'interfaccia (CLI); questa richiama direttamente i metodi esposti nei controller della Logica di Controllo; quest'ultima delega l'esecuzione della logica di business ai Service, i quali a loro volta si avvalgono dei DAO per l'interazione con il database tramite JDBC e per la creazione/recupero degli oggetti del Domain Model; una volta ottenuti o manipolati gli oggetti, i Service restituiscono il controllo ai Controller che possono aggiornare l'interfaccia utente.

2 Progettazione

2.1 Introduzione

In questa sezione viene presentata la progettazione del sistema, tramite diagrammi UML che descrivono la struttura delle classi e l'organizzazione in package.

2.2 Use Case Diagram

I principali casi d'uso individuati sono:

- **Visitatore:**
  - Può effettuare la registrazione
  - Può effettuare il login
  - Può accedere all'elenco dei post

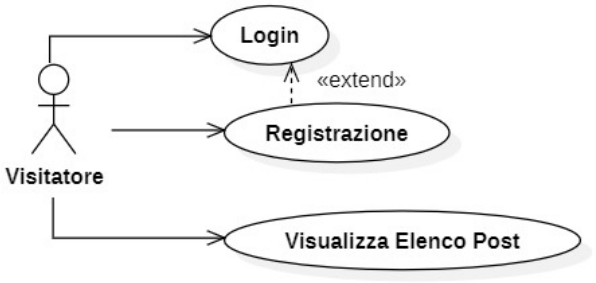

Figura 1: Diagramma Use Case per Visitatore

- **Osservatore:**
  - Può visualizzare l'elenco dei post
  - Può mettere like ad un post
  - Può commentare sotto un post
  - Può effettuare il logout

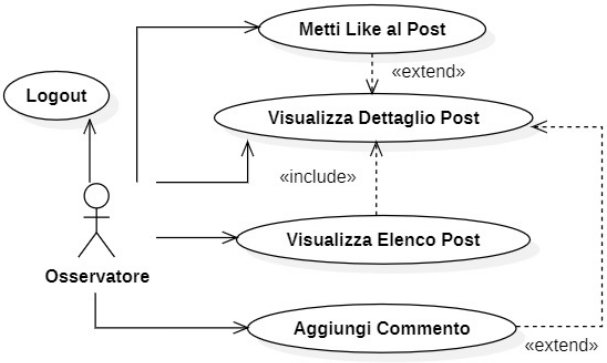

Figura 2: Diagramma Use Case per Osservatore

- **Autore:**
  - Può pubblicare un post
  - Può visualizzare l'elenco dei post
  - Può mettere like ad un post
  - Può commentare sotto un post
  - Può caricare un'immagine
  - Può applicare un filtro all'immagine
  - Può salvare immagini in locale

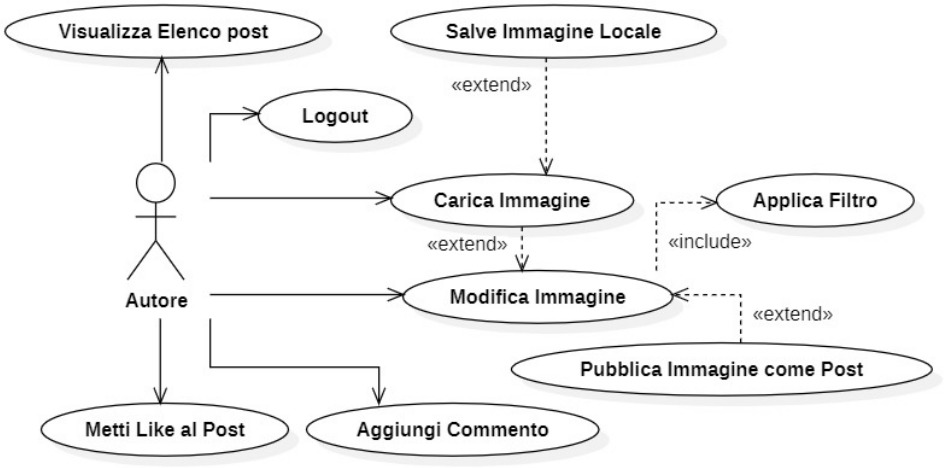

Figura 3: Diagramma Use Case per Autore

- **Gestione Autenticazione:**
  - Utente può registrarsi come nuovo utente. Segue controllo unicità di username ed email, valida del formato mail, e salvataggio dei dati utente
  - Utente può effettuare il login. Il sistema database quindi verifica username e password
  - Utente effettua il logout

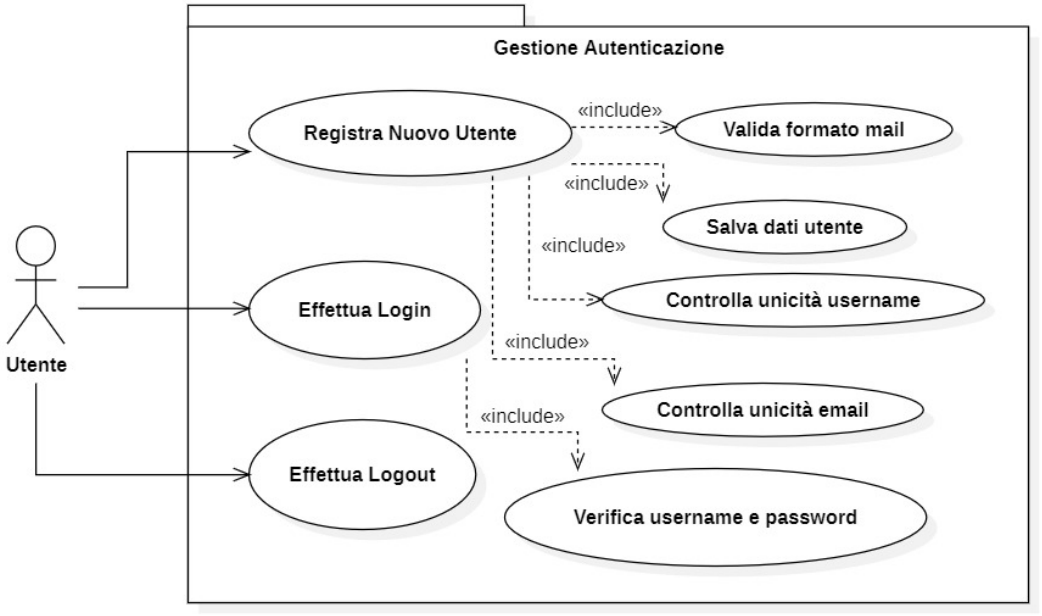

Figura 4: Diagramma Use Case per Gestione Autenticazione

- **Gestione Filtri Immagine:**
  - Autore seleziona il filtro
  - Filter Registry applica il filtro selezionato dall'autore
  - Autore può salvare l'immagine modificata.

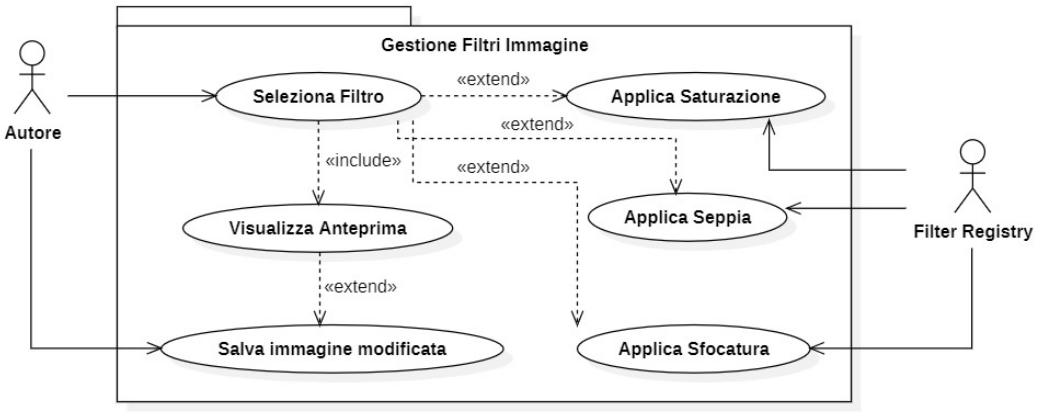

Figura 5: Diagramma Use Case per Gestione Filtri Immagine

2.3 Template dei Casi d'Uso

In questa sezione vengono presentati i template dettagliati dei principali casi d'uso del sistema. Ogni template è corredato da un diagramma UML, che può essere sostituito con il diagramma specifico realizzato per l'applicativo.

2.3.1 Gestione Autenticazione

Questa sezione descrive i flussi relativi a registrazione, autenticazione e logout degli utenti. Vengono illustrate tutte le interazioni e i controlli necessari per la sicurezza e la gestione degli account.

Use Case #1	**Registrazione**
Brief Description	Il Visitatore si registra al sistema per diventare un utente
Level	User Goal
Actors	Visitatore
Pre-Conditions	Il Visitatore non è registrato nel sistema
Basic Flow	1) Il Visitatore seleziona "Registrazione" 2) Il sistema mostra il form di registrazione 3) Il Visitatore inserisce username, nome, cognome, data di nascita, cellulare (opzionale), email, password e ruolo 4) Il sistema valida il formato email (≪include≫ Valida formato mail) 5) Il sistema controlla l'unicità di username ed email (≪include≫ Controlla unicità username, Controlla unicità email) 6) Il sistema salva i dati dell'utente nel database (≪include≫ Salva dati utente) 7) Il sistema conferma la registrazione avvenuta con successo
Alternative Flow	3a) Il Visitatore inserisce dati non validi: il sistema mostra un errore 5a) Username o email già esistenti: il sistema mostra errore "Username o Email già esistente"
Post-Conditions	Il Visitatore è registrato come Utente (Osservatore o Autore) e può effettuare il login

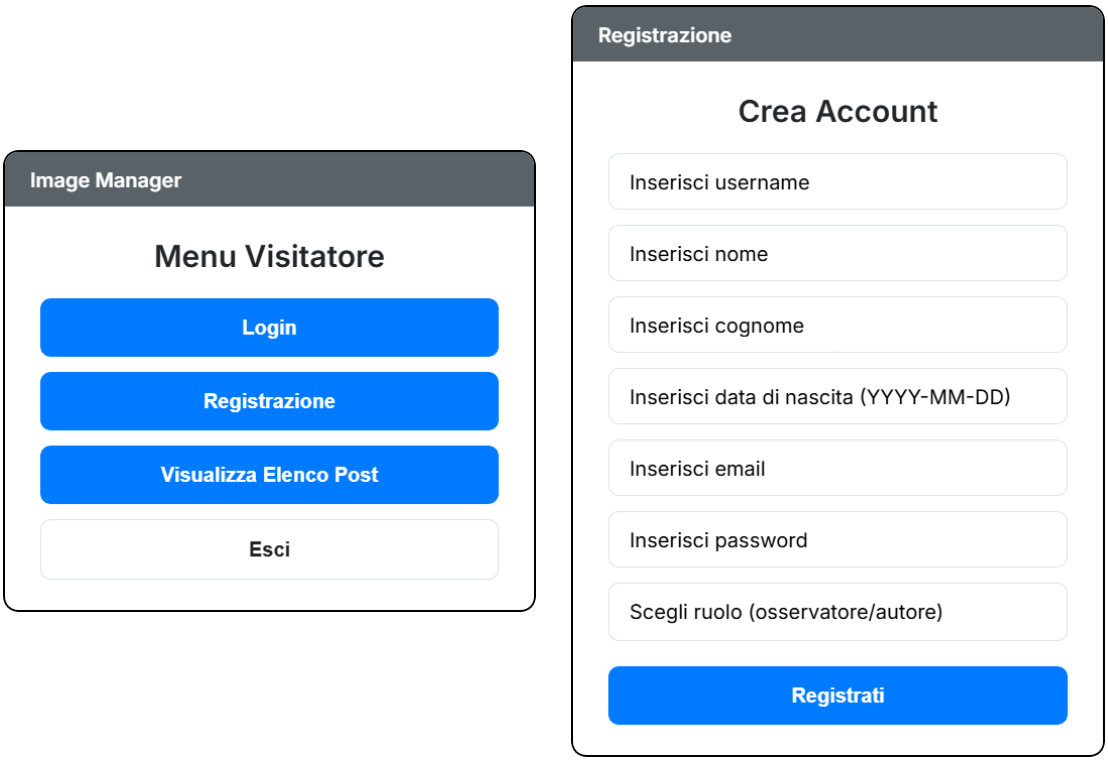

Figura 6: Mockups Use Case 1

Use Case #2	**Login**
Brief Description	Il Visitatore effettua l'accesso al sistema
Level	User Goal
Actors	Visitatore
Pre-Conditions	Il Visitatore è registrato nel sistema
Basic Flow	1) Il Visitatore seleziona "Login" 2) Il sistema mostra il form di login 3) Il Visitatore inserisce username e password 4) Il sistema verifica username e password («include» Verifica username e password) 5) Il sistema autentica l'utente e mostra il menu appropriato al ruolo (Osservatore o Autore) 6) Il sistema registra "Login effettuato con successo! Benvenuto [nome]"
Alternative Flow	4a) Credenziali non valide: il sistema mostra "Login fallito: Credenziali non valide." 4b) Errore database: il sistema mostra "Errore del database durante il login"
Post-Conditions	Il Visitatore è autenticato come Osservatore o Autore

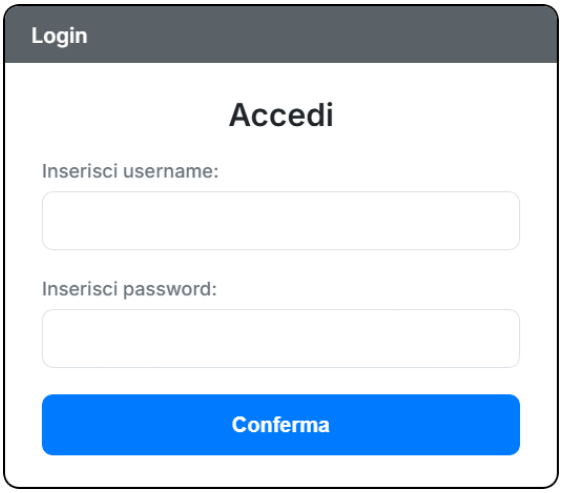

Figura 7: Mockup Use Case 2

Use Case #3	**Logout**
Brief Description	L'Utente termina la sessione corrente
Level	User Goal
Actors	Osservatore o Autore
Pre-Conditions	L'Utente è autenticato nel sistema
Basic Flow	1) L'Utente seleziona "Logout" 2) Il sistema termina la sessione corrente 3) Il sistema resetta l'immagine caricata in memoria (se presente) 4) Il sistema mostra "[username] ha effettuato il logout" 5) Il sistema ritorna al menu Visitatore
Post-Conditions	L'Utente è disconnesso e torna allo stato di Visitatore

Figura 8: Mockups Use Case 3

2.3.2 Gestione Post

Il template per la gestione dei post evidenzia tutte le operazioni legate alla creazione, visualizzazione, modifica e interazione con i post pubblicati dagli utenti.

Use Case #4	**Visualizza Elenco Post**
Brief Description	L'utente visualizza l'elenco di tutti i post pubblicati
Level	User Goal
Actors	Visitatore, Osservatore, Autore
Pre-Conditions	Nessuna
Basic Flow	1) L'utente seleziona "Visualizza Elenco Post" 2) Il sistema recupera tutti i post dal database 3) Il sistema mostra l'elenco con: Post ID, Autore, Descrizione, Likes, Data 4) L'utente può selezionare un post per visualizzarne i dettagli («include» Visualizza Dettaglio Post) 5) L'utente può visualizzare i commenti di un post («include» Visualizza Commenti)
Alternative Flow	2a) Nessun post presente: il sistema mostra "Nessun post da visualizzare al momento" 2b) Errore database: il sistema mostra "Errore del database durante la visualizzazione dei post"
Post-Conditions	L'utente ha visualizzato l'elenco dei post disponibili

Figura 9: Mockup Use Case 4

Use Case #5	**Pubblica Post**
Brief Description	L'Autore pubblica un'immagine caricata come nuovo post
Level	User Goal
Actors	Autore
Pre-Conditions	• - L'utente è autenticato come Autore • - Un'immagine è caricata in memoria
Basic Flow	1) L'Autore seleziona "Pubblica Immagine Caricata come Post" 2) Il sistema richiede una descrizione per il post 3) L'Autore inserisce la descrizione 4) Il sistema converte l'immagine in formato binario (PNG) 5) Il sistema crea un nuovo Post con: autore, immagine, dimensione, formato, descrizione 6) Il sistema salva il post nel database 7) Il sistema mostra "Post pubblicato con successo! ID Post: [id]"
Alternative Flow	1a) Nessuna immagine caricata: il sistema mostra "Nessuna immagine valida caricata in memoria da pubblicare" 1b) Utente non autenticato: il sistema mostra "Devi essere loggato per pubblicare un post" 4a) Errore conversione: il sistema mostra "Errore: impossibile convertire l'immagine" 6a) Errore database: il sistema mostra "Errore del database durante la pubblicazione del post"
Post-Conditions	Il post è pubblicato e disponibile per la visualizzazione

Figura 10: Mockup Use Case 5

2.3.3 Gestione Immagini

Questa sezione si focalizza sulle operazioni di caricamento, salvataggio locale, eliminazione e gestione delle immagini da parte degli autori.

Use Case #6	**Carica Immagine**
Brief Description	L'Autore carica un'immagine dal file system locale
Level	User Goal
Actors	Autore
Pre-Conditions	L'utente è autenticato come Autore
Basic Flow	1) L'Autore seleziona "Carica Immagine" 2) Il sistema richiede il path dell'immagine 3) L'Autore inserisce il percorso del file 4) Il sistema carica l'immagine in memoria 5) Il sistema mostra "Immagine caricata con successo! Dimensioni: [larghezza]x[altezza]" 6) L'immagine è disponibile per modifiche («extend» Salva Immagine Locale)
Alternative Flow	1a) Immagine già caricata: il sistema avvisa e chiede conferma per sovrascrivere 4a) File non trovato: il sistema mostra "File non trovato o non è un file valido" 4b) Formato non supportato: il sistema mostra "Impossibile leggere l'immagine (formato non supportato)" 4c) Path non valido: il sistema mostra "Path non valido"
Post-Conditions	L'immagine è caricata in memoria e pronta per essere modificata o pubblicata

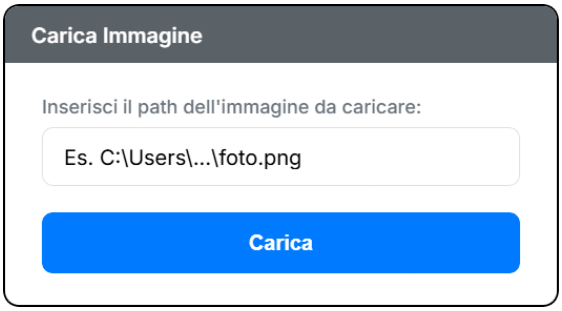

Figura 11: Mockup Use Case 6

Use Case #7	**Modifica Immagine**
Brief Description	L'Autore applica filtri all'immagine caricata
Level	User Goal
Actors	Autore
Pre-Conditions	• - L'utente è autenticato come Autore • - Un'immagine è caricata in memoria
Basic Flow	1) L'Autore seleziona "Modifica Immagine Caricata" 2) Il sistema mostra i filtri disponibili: GrayScale, Invert, Blur, Sharpen, Sepia, Luminosità/Contrasto 3) L'Autore seleziona un filtro (≪include≫ Applica Filtri) 4) Il sistema applica il filtro all'immagine 5) Il sistema mostra "Filtro [nome] applicato" 6) Il sistema chiede se salvare l'immagine modificata 7) Se si, l'immagine viene salvata (≪extend≫ Salva Immagine Locale)
Alternative Flow	1a) Nessuna immagine caricata: il sistema mostra "Nessuna immagine valida caricata" 3a) L'Autore seleziona 0: modifica annullata 3b) L'Autore seleziona Luminosità/Contrasto: il sistema richiede valori specifici 4a) Errore applicazione filtro: il sistema mostra "Errore nell'applicazione del filtro"
Post-Conditions	L'immagine è stata modificata con il filtro selezionato

Figura 12: Mockup Use Case 7

Use Case #8	**Salva Immagine Locale**
Brief Description	L'Autore salva l'immagine caricata o modificata nel file system locale
Level	Sub-function
Actors	Autore
Pre-Conditions	Un'immagine valida è caricata in memoria
Basic Flow	1) L'Autore conferma il salvataggio 2) Il sistema richiede il nome del file (senza estensione) 3) L'Autore inserisce il nome 4) Il sistema salva l'immagine nella cartella "immagini_salvate_localmente" in formato PNG 5) Il sistema mostra "Immagine salvata correttamente al seguente percorso [path]"
Alternative Flow	1a) Nessuna immagine valida: il sistema mostra "Nessuna immagine valida caricata in memoria" 4a) Errore I/O: il sistema mostra "Errore I/O durante il salvataggio dell'immagine" 4b) Parametri non validi: il sistema mostra "Errore nei parametri di salvataggio"
Post-Conditions	L'immagine è salvata nel file system locale

Figura 13: Mockup Use Case 8

2.3.4 Interazioni Social

Questa parte include i casi d'uso relativi alle interazioni tra utenti come like, commenti e visualizzazione delle attività social, che permettono di aumentare l'engagement all'interno della piattaforma.

Use Case #9	**Metti Like**
Brief Description	L'utente mette like a un post
Level	User Goal
Actors	Osservatore, Autore
Pre-Conditions	• - L'utente sta visualizzando i dettagli di un post • - Il post esiste nel sistema
Basic Flow	1) L'utente seleziona "Metti Like" per il post corrente 2) Il sistema incrementa il contatore dei like del post 3) Il sistema aggiorna il database 4) Il sistema mostra "Like aggiunto al post ID: [id]" 5) Il contatore dei like viene aggiornato nella visualizzazione
Alternative Flow	2a) Errore database: il sistema mostra "Errore database durante l'interazione con il post" 2b) Post non trovato: il sistema mostra "Post non trovato con ID: [id]"
Post-Conditions	Il numero di like del post è incrementato di 1

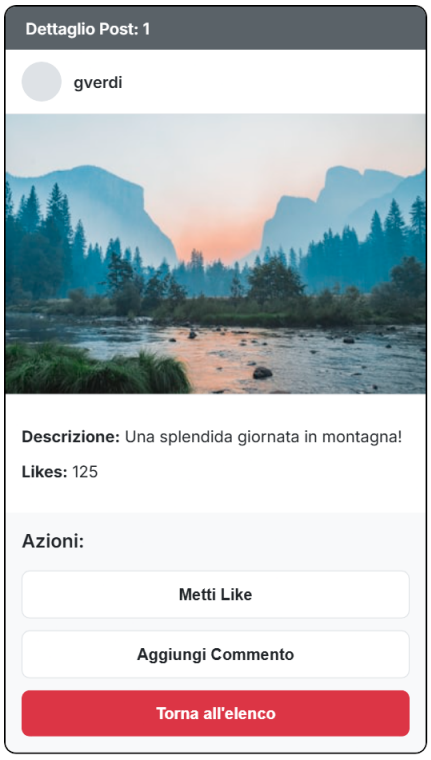

Figura 14: Mockup Use Case 9

Use Case #10	**Aggiungi Commento**
Brief Description	L'utente autenticato aggiunge un commento a un post
Level	User Goal
Actors	Osservatore, Autore
Pre-Conditions	• - L'utente è autenticato • - L'utente sta visualizzando i dettagli di un post
Basic Flow	1) L'utente seleziona "Aggiungi Commento" 2) Il sistema richiede il testo del commento 3) L'utente inserisce il commento 4) Il sistema crea un nuovo commento con: post_id, username, testo, timestamp 5) Il sistema salva il commento nel database 6) Il sistema mostra "Commento aggiunto al post ID: [id]" 7) Il commento appare nella lista dei commenti del post
Alternative Flow	1a) Utente non autenticato: il sistema mostra "Devi essere loggato per commentare" 3a) Testo vuoto: il sistema mostra "Il testo del commento non può essere vuoto" 5a) Errore database: il sistema mostra "Errore del database durante l'interazione con il post"
Post-Conditions	Il commento è aggiunto al post e visibile a tutti gli utenti

Figura 15: Mockup Use Case 10

2.3.5 Gestione Filtri Immagini

La sezione dedicata ai filtri illustra la selezione, l'applicazione e il salvataggio dei filtri digitali sulle immagini, con particolare attenzione al funzionamento del *Filter Registry* e alle possibili composizioni.

Use Case #11	**Seleziona Filtro**
Brief Description	L'Autore seleziona un filtro da applicare all'immagine
Level	User Goal
Actors	Autore, Filter Registry
Pre-Conditions	• - L'Autore ha un'immagine caricata • - L'Autore ha selezionato "Modifica Immagine"
Basic Flow	1) Il sistema mostra la lista dei filtri disponibili dal Filter-Registry 2) L'Autore seleziona un filtro specifico 3) Il sistema può mostrare un'anteprima (≪include≫ Visualizza Anteprima) 4) Il sistema applica il filtro selezionato (≪extend≫ Applica Saturazione, Applica Seppia, Applica Sfocatura) 5) L'Autore può salvare l'immagine modificata (≪extend≫ Salva immagine modificata)
Alternative Flow	2a) L'Autore seleziona "0": operazione annullata 2b) Selezione non valida: il sistema mostra "Filtro non valido"
Post-Conditions	Il filtro è stato applicato all'immagine

Figura 16: Mockup Use Case 11

2.4 Diagrammi delle Attività

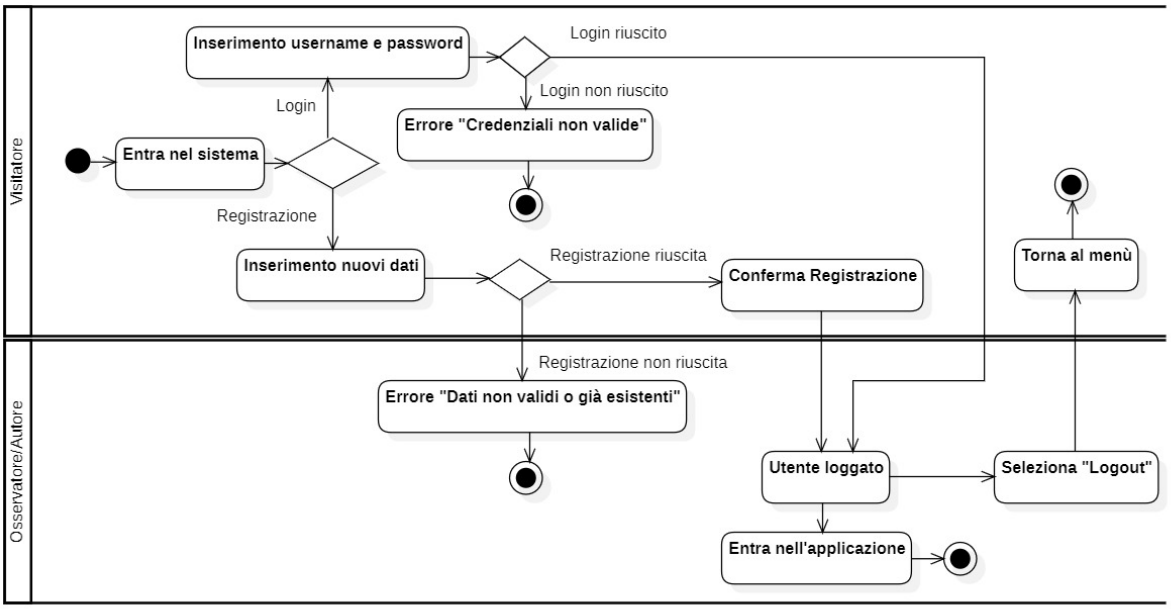

Figura 17: Diagramma delle attività per Gestione Autenticazione

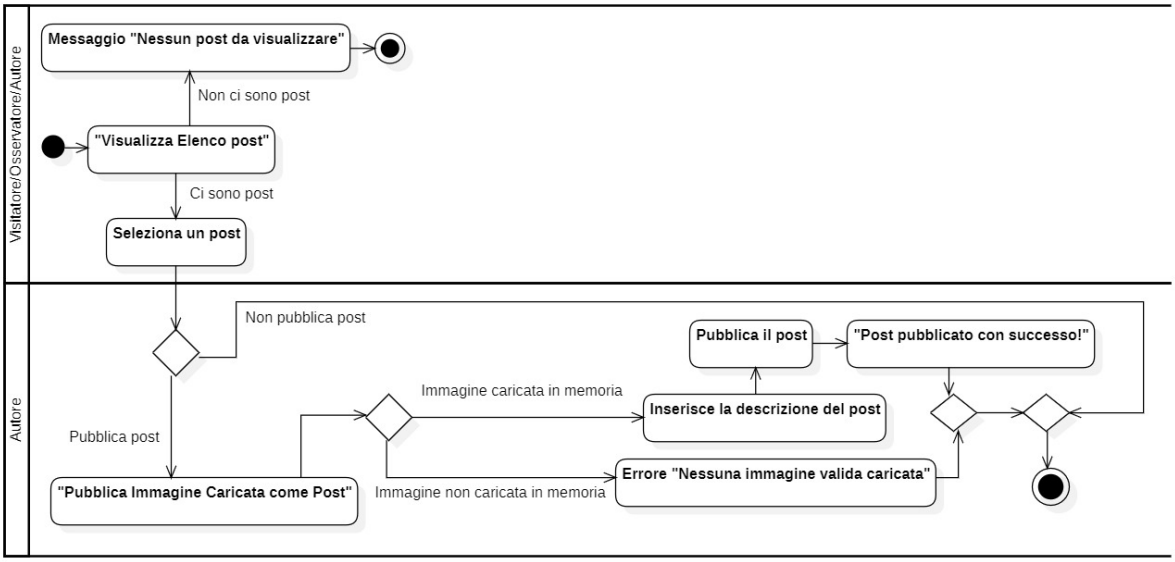

Figura 18: Diagramma delle attività per Gestione Post

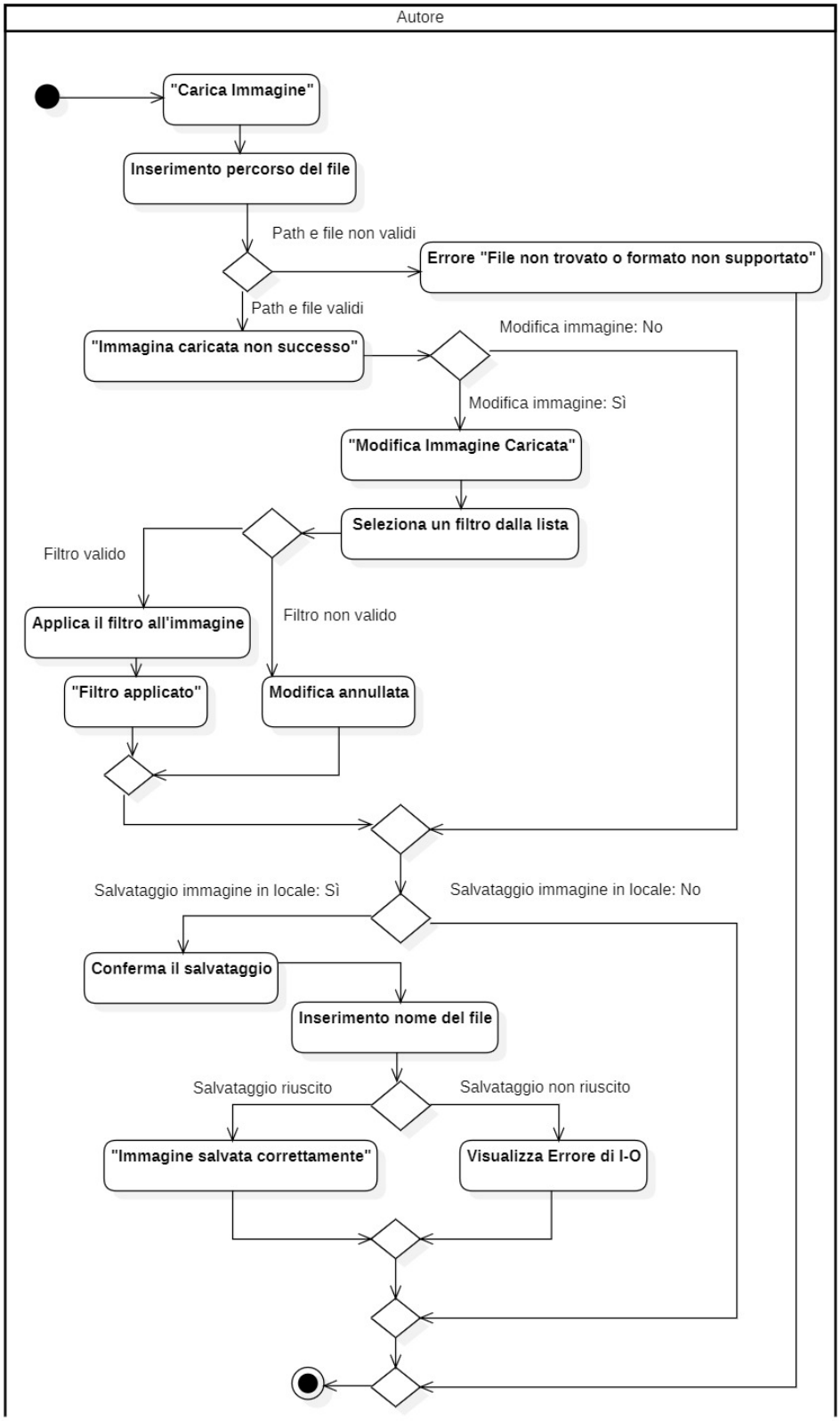

Figura 19: Diagramma delle attività per Gestione Immagini

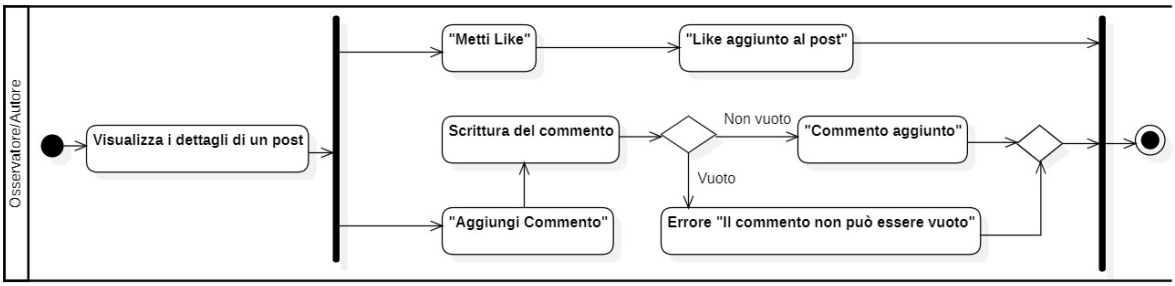

Figura 20: Diagramma delle attività per Interazioni Social

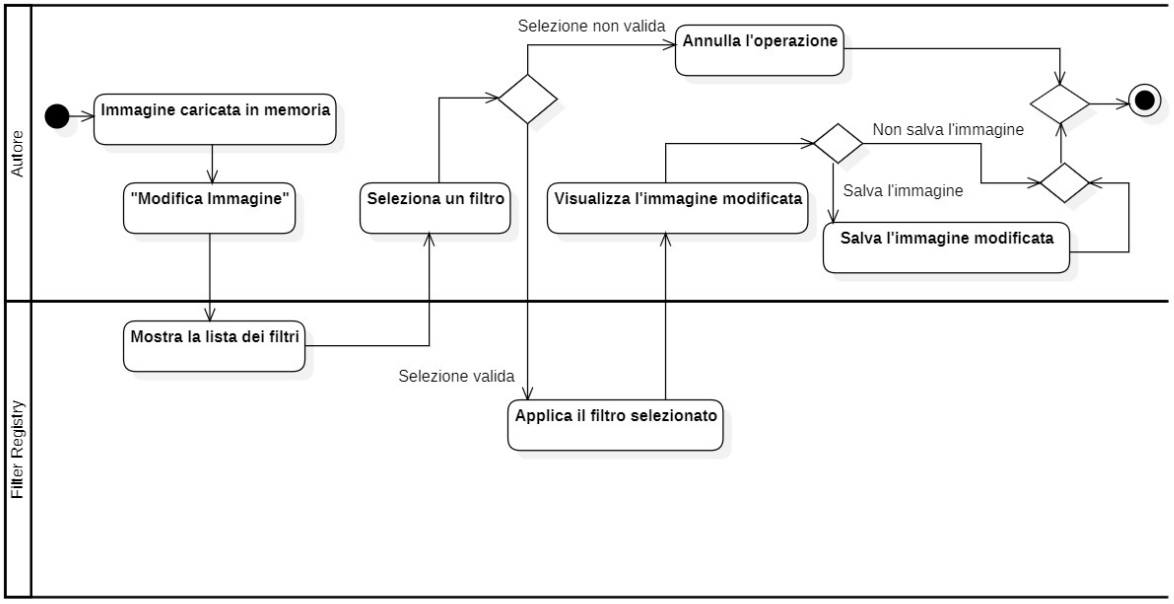

Figura 21: Diagramma delle attività per Gestione Filtri Immagini

3 Implementazione

3.1 Realizzazione Progetto

Questa applicazione implementa tutta la logica di backend prevista, inclusa l'autenticazione degli utenti, il caricamento, la modifica e il salvataggio di immagini, la pubblicazione di post con immagini e l'interazione con un database SQL per la persistenza dei dati. Pertanto, questo documento descrive in dettaglio l'applicazione Java sviluppata come sistema CLI, tralasciando ipotetiche componenti GUI non realizzate.

3.2 Diagramma delle Classi

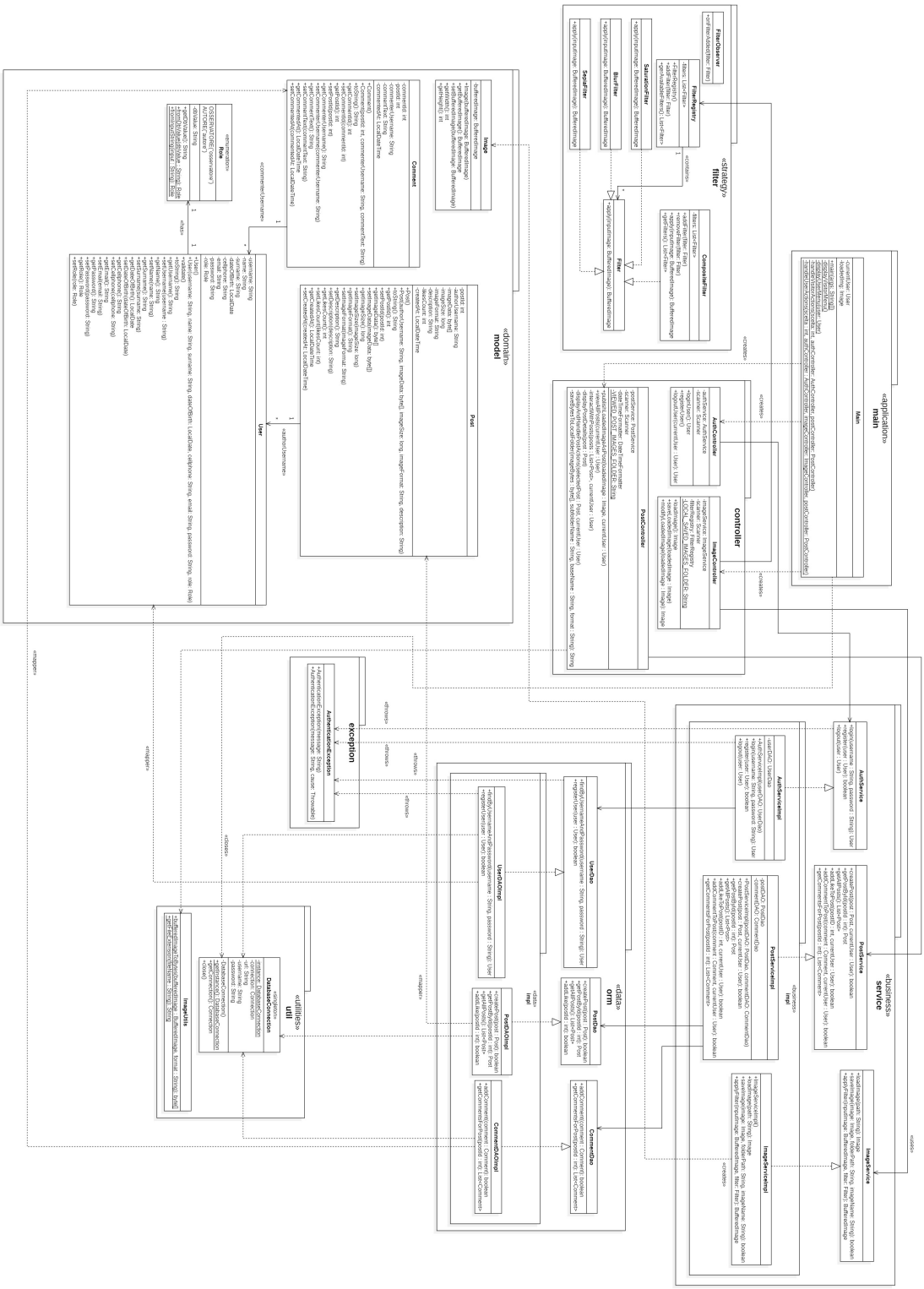

Figura 22: Diagramma delle Classi di Image Manager

3.3 Struttura del Progetto e Analisi dei Package

3.3.1 Package Controller

Il package `controller` rappresenta il cuore dell'architettura MVC implementata nel sistema. Le classi controller fungono da intermediari intelligenti tra l'interfaccia utente a riga di comando e i servizi di business logic, orchestrando il flusso delle operazioni e gestendo la comunicazione bidirezionale tra i vari layer dell'applicazione.

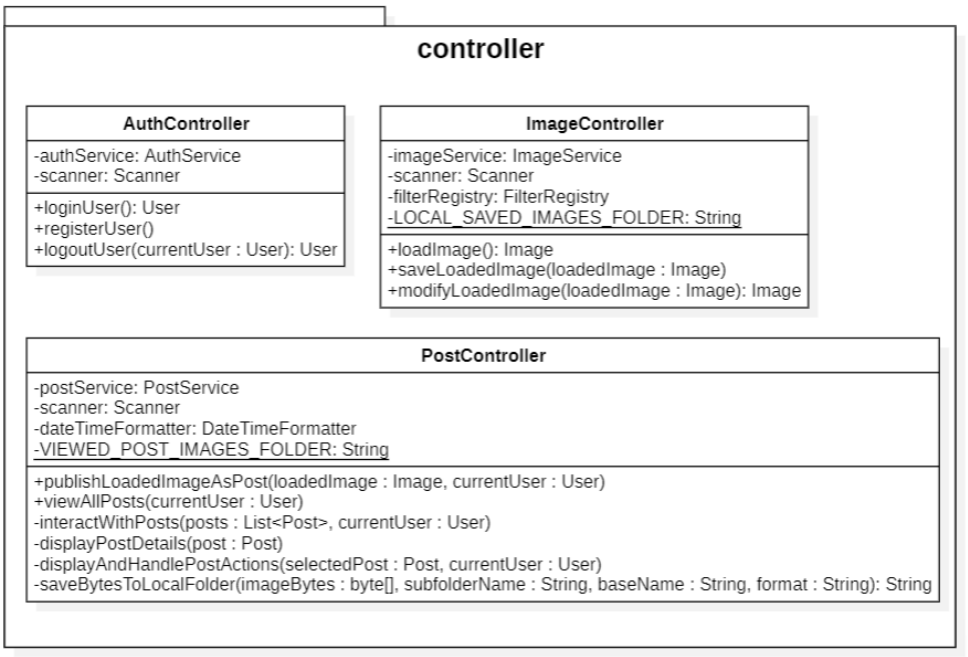

Figura 23: Diagramma UML del package Controller

AuthController La classe `AuthController` gestisce tutte le operazioni relative all'autenticazione e alla gestione degli account utente. Questa classe è responsabile di raccogliere gli input dall'utente tramite la CLI, validarli preliminarmente e delegare le operazioni di business logic al servizio di autenticazione.

```
public class AuthController {
  private final AuthService authService;
  private final Scanner scanner;

  public AuthController(AuthService authService, Scanner scanner) {
    this.authService = authService;
    this.scanner = scanner;
  }

  public User loginUser() {
    System.out.print("Inserisci username: ");
```

```
    String loginUsername = scanner.nextLine();
    System.out.print("Inserisci password: ");
    String loginPassword = scanner.nextLine();

    try {
      User user = authService.login(loginUsername, loginPassword);
      System.out.println("Login effettuato con successo! Benvenuto "
      + user.getName());
      return user;
    } catch (AuthenticationException e) {
      System.out.println("Login fallito: " + e.getMessage());
    } catch (SQLException e) {
      System.out.println("Errore del database durante il login: "
      + e.getMessage());
    } catch (IllegalArgumentException e) {
      System.out.println("Errore nei dati inseriti: " + e.getMessage());
    }
    return null;
  }
}
```

Listing 1: Implementazione del metodo loginUser in AuthController

Il metodo `loginUser` dimostra come il controller gestisca l'intero flusso di autenticazione: raccoglie le credenziali, invoca il servizio appropriato e gestisce tutte le possibili eccezioni in modo user-friendly. La gestione granulare delle eccezioni permette di fornire feedback specifici all'utente in base al tipo di errore verificatosi.

La registrazione degli utenti è gestita con particolare attenzione alla validazione dei dati:

```
public void registerUser() {
  // Raccolta dati con validazione della data di nascita
  LocalDate regDateOfBirth = null;
  while (regDateOfBirth == null) {
    System.out.print("Inserisci data di nascita (YYYY-MM-DD): ");
    String dobStr = scanner.nextLine();
    try {
      regDateOfBirth = LocalDate.parse(dobStr);
    } catch (DateTimeParseException e) {
      System.out.println("Formato data non valido. Usa YYYY-MM-DD.");
    }
  }

  // Gestione del ruolo con validazione
  Role regRole = null;
  while (regRole == null) {
    System.out.print("Inserisci ruolo (osservatore/autore): ");
    String ruoloStr = scanner.nextLine();
    try {
      regRole = Role.fromInputString(ruoloStr);
    } catch (IllegalArgumentException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Listing 2: Gestione della registrazione con validazione completa

ImageController Il `ImageController` gestisce tutte le operazioni relative alla manipolazione delle immagini. Questa classe è particolarmente importante poiché coordina l'interazione tra l'utente, il servizio di gestione immagini e il sistema di filtri.

```
public class ImageController {
  private final ImageService imageService;
  private final Scanner scanner;
  private final FilterRegistry filterRegistry;
  private static final String LOCAL_SAVED_IMAGES_FOLDER =
  "immagini_salvate_localmente";

  public Image loadImage() {
    System.out.print("Inserisci il path dell'immagine da caricare: ");
    String path = scanner.nextLine();
    try {
      Image loadedImage = imageService.loadImage(path);
      System.out.println("Immagine caricata con successo! Dimensioni: "
      + loadedImage.getWidth() + "x" + loadedImage.getHeight());
      return loadedImage;
    } catch (IOException e) {
      System.err.println("Errore durante il caricamento dell'immagine: "
      + e.getMessage());
    } catch (IllegalArgumentException e) {
      System.err.println("Path non valido: " + e.getMessage());
    }
    return null;
  }
}
```

Listing 3: Caricamento e gestione delle immagini

La gestione della modifica delle immagini è particolarmente sofisticata, offrendo all'utente un menu dinamico di filtri disponibili:

```
public Image modifyLoadedImage(Image loadedImg) {
  if (loadedImg == null || loadedImg.getBufferedImage() == null) {
    System.out.println("Nessuna immagine valida caricata.");
    return loadedImg;
  }

  System.out.println("Filtri disponibili:");
  List<Filter> availableFilters = filterRegistry.getAvailableFilters();
  for (int i = 0; i < availableFilters.size(); i++) {
    System.out.println((i + 1) + ". " +
    availableFilters.get(i).getClass().getSimpleName());
  }
  System.out.println((availableFilters.size() + 1) +
  ". Regola Luminosita'/Contrasto");

  // Esempio: le variabili per l'input e l'immagine sono definite altrove
  int filtroSceltoIdxInput = 1;
  BufferedImage originalBI = loadedImg.getBufferedImage();
  BufferedImage filteredBI = null;

  // Gestione della selezione e applicazione del filtro
  if (filtroSceltoIdxInput > 0 &&
  filtroSceltoIdxInput <= availableFilters.size()) {
    Filter selectedFilter = availableFilters.get(filtroSceltoIdxInput - 1);
    filteredBI = imageService.applyFilter(originalBI, selectedFilter);
    System.out.println("Filtro " +
    selectedFilter.getClass().getSimpleName() + " applicato!");
  }
}
```

Listing 4: Sistema di applicazione filtri con menu dinamico

PostController Il `PostController` è responsabile della gestione completa del ciclo di vita dei post, dalla pubblicazione alla visualizzazione e interazione. Questa classe implementa logiche complesse per la gestione dei contenuti social dell'applicazione.

```
public void publishLoadedImageAsPost(Image loadedImg, User currentUser) {
  // Validazioni preliminari
  if (loadedImg == null || loadedImg.getBufferedImage() == null) {
    System.out.println("Nessuna immagine valida caricata in memoria.");
    return;
  }
  if (currentUser == null) {
    System.out.println("Devi essere loggato per pubblicare un post.");
    return;
  }

  System.out.print("Inserisci una descrizione per il post: ");
  String description = scanner.nextLine();
  String imageFormat = "png";

  // Conversione dell'immagine in formato binario
  byte[] imageData = ImageUtils.bufferedImageToBytes(
  loadedImg.getBufferedImage(), imageFormat);

  if (imageData != null) {
    long imageSize = imageData.length;
    Post newPost = new Post(
    currentUser.getUsername(),
    imageData,
    imageSize,
    imageFormat,
    description
    );

    try {
      if (postService.createPost(newPost, currentUser)) {
        System.out.println("Post pubblicato con successo! ID Post: "
        + newPost.getPostId());
      }
    } catch (SQLException e) {
      System.err.println("Errore del database: " + e.getMessage());
    }
  }
}
```

Listing 5: Pubblicazione di un post con validazioni

3.3.2 Package DAO (Data Access Object)

Il package DAO implementa il pattern Data Access Object, fornendo un'astrazione completa tra la logica di business e il layer di persistenza. Questa architettura permette di modificare il sistema di storage sottostante senza impattare il resto dell'applicazione.

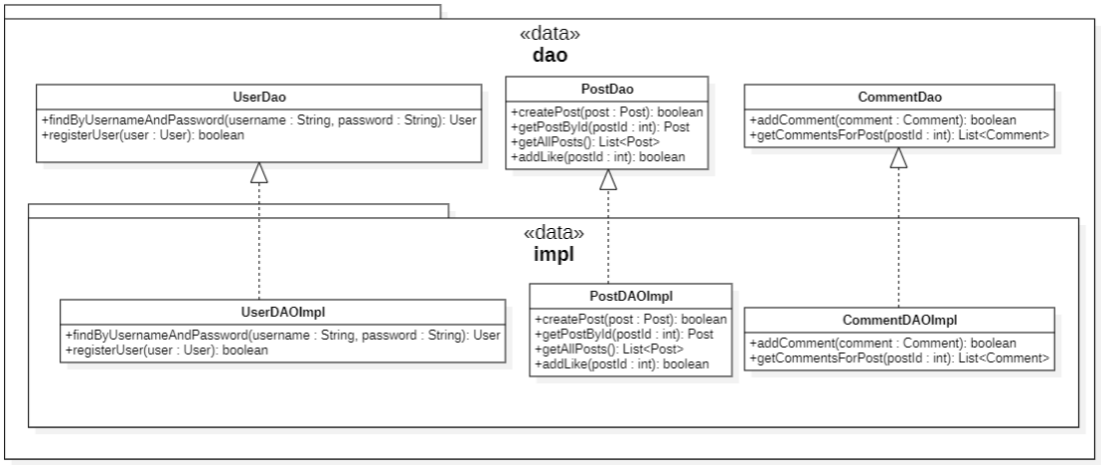

Figura 24: Diagramma UML del package DAO

Interfacce DAO Le interfacce DAO definiscono i contratti per l'accesso ai dati, garantendo un'astrazione pulita:

```
// UserDao.java
public interface UserDao {
  User findByUsernameAndPassword(String username, String password)
  throws SQLException, AuthenticationException;
  boolean registerUser(User user) throws SQLException;
}

// PostDao.java
public interface PostDao {
  boolean createPost(Post post) throws SQLException;
  Post getPostById(int postId) throws SQLException;
  List<Post> getAllPosts() throws SQLException;
  boolean addLike(int postId) throws SQLException;
}

// CommentDao.java
public interface CommentDao {
  boolean addComment(Comment comment) throws SQLException;
  List<Comment> getCommentsForPost(int postId) throws SQLException;
}
```

Listing 6: Interfacce DAO per la definizione dei contratti

3.3.3 Package DAO (Data Access Object)

Il package DAO implementa il pattern Data Access Object, fornendo un'astrazione completa tra la logica di business e il layer di persistenza. Questa architettura permette di modificare il sistema di storage sottostante senza impattare il resto dell'applicazione.

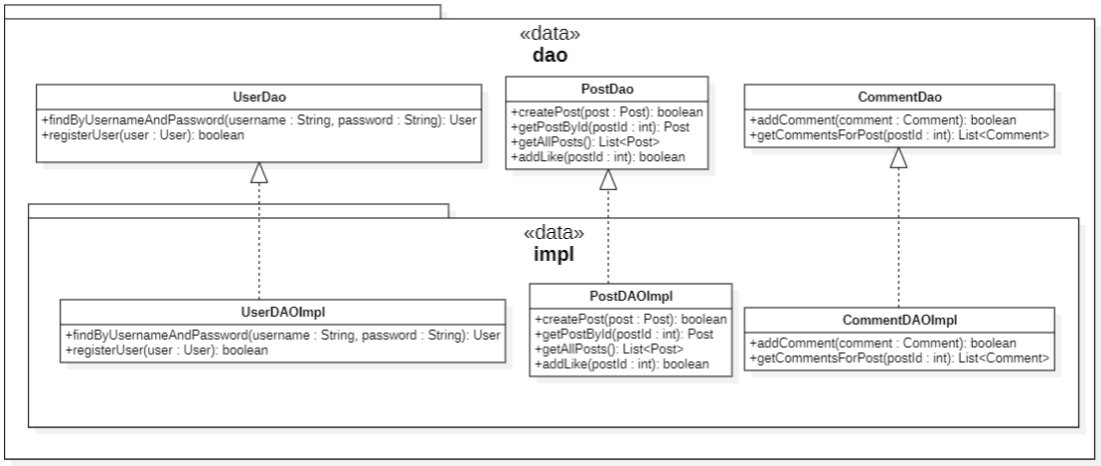

Figura 25: Diagramma UML del package Dao

Interfacce DAO Le interfacce DAO definiscono i contratti per l'accesso ai dati, garantendo un'astrazione pulita:

```
// UserDao.java
public interface UserDao {
  User findByUsernameAndPassword(String username, String password)
  throws SQLException, AuthenticationException;
  boolean registerUser(User user) throws SQLException;
}

// PostDao.java
public interface PostDao {
  boolean createPost(Post post) throws SQLException;
  Post getPostById(int postId) throws SQLException;
  List<Post> getAllPosts() throws SQLException;
  boolean addLike(int postId) throws SQLException;
}

// CommentDao.java
public interface CommentDao {
  boolean addComment(Comment comment) throws SQLException;
  List<Comment> getCommentsForPost(int postId) throws SQLException;
}
```

Listing 7: Interfacce DAO per la definizione dei contratti

Implementazioni DAO Le implementazioni concrete utilizzano JDBC per l'interazione con PostgreSQL. La conversione tra ResultSet e oggetti del dominio avviene attraverso metodi privati dedicati:

```
public class UserDAOImpl implements UserDao {

  @Override
  public User findByUsernameAndPassword(String username, String password)
  throws SQLException, AuthenticationException {
    String sql = "SELECT username, name, surname, dateofbirth, " +
    "cellphone, email, password, role " +
    "FROM users WHERE username = ?";

    try (Connection conn = DatabaseConnection.getInstance().getConnection();
    PreparedStatement ps = conn.prepareStatement(sql)) {

      ps.setString(1, username);

      try (ResultSet rs = ps.executeQuery()) {
        if (rs.next()) {
          String storedPassword = rs.getString("password");
          if (checkPassword(storedPassword, password)) {
            // Conversione diretta da ResultSet a User
            User user = new User();
            user.setUsername(rs.getString("username"));
            user.setName(rs.getString("name"));
            user.setSurname(rs.getString("surname"));
            Date sqlDateOfBirth = rs.getDate("dateofbirth");
            if (sqlDateOfBirth != null) {
              user.setDateOfBirth(sqlDateOfBirth.toLocalDate());
            }
            user.setCellphone(rs.getString("cellphone"));
            user.setEmail(rs.getString("email"));
            user.setPassword(rs.getString("password"));
            String roleValue = rs.getString("role");
            user.setRole(Role.fromDbValue(roleValue));
            return user;
          } else {
            throw new AuthenticationException(
            "Credenziali non valide.");
          }
        } else {
          throw new AuthenticationException(
          "Credenziali non valide.");
        }
      }
    }
  }

  @Override
  public boolean registerUser(User user) throws SQLException {
    String sql = "INSERT INTO users (username, name, surname, " +
    "dateofbirth, cellphone, email, password, role) " +
    "VALUES (?, ?, ?, ?, ?, ?, ?, ?)";

    try (Connection conn = DatabaseConnection.getInstance().getConnection();
    PreparedStatement ps = conn.prepareStatement(sql)) {

      // Setting diretto dei parametri
      ps.setString(1, user.getUsername());
      ps.setString(2, user.getName());
      ps.setString(3, user.getSurname());
      ps.setDate(4, Date.valueOf(user.getDateOfBirth()));
      ps.setString(5, user.getCellphone());
      ps.setString(6, user.getEmail());
```

```
            ps.setString(7, user.getPassword());
            ps.setString(8, user.getRole().getDbValue());

            int affectedRows = ps.executeUpdate();
            return affectedRows > 0;

        } catch (SQLException e) {
            if ("23505".equals(e.getSQLState())) { // Violazione di unique constraint
                throw new SQLException(
                "Errore nella registrazione: Username o Email gia' esistente.",
                e.getSQLState(), e);
            }
            throw e;
        }
    }

    private boolean checkPassword(String storedPassword, String providedPassword)
    {
        // In una implementazione reale, questo metodo dovrebbe confrontare hash
        return storedPassword.equals(providedPassword);
    }
}
```

Listing 8: Implementazione UserDAOImpl con gestione diretta del mapping

3.3.4 Package Exception

Il package delle eccezioni contiene classi personalizzate per gestire situazioni di errore specifiche del dominio applicativo:

```
public class AuthenticationException extends Exception {
    public AuthenticationException(String message) {
        super(message);
    }

    public AuthenticationException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Listing 9: Eccezione personalizzata per l'autenticazione

Questa eccezione viene utilizzata per distinguere gli errori di autenticazione da altri tipi di errori, permettendo una gestione più granulare e messaggi più specifici per l'utente.

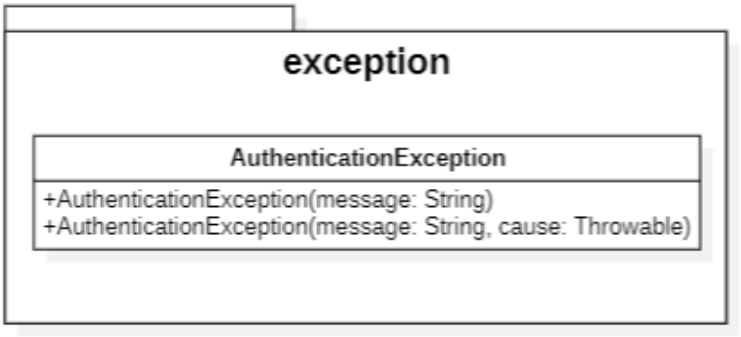

Figura 26: Diagramma UML del package Exception

3.3.5 Package Filter

Il package filter implementa il pattern Strategy per l'applicazione di filtri alle immagini. Questa architettura permette di aggiungere nuovi filtri senza modificare il codice esistente, rispettando il principio Open/Closed.

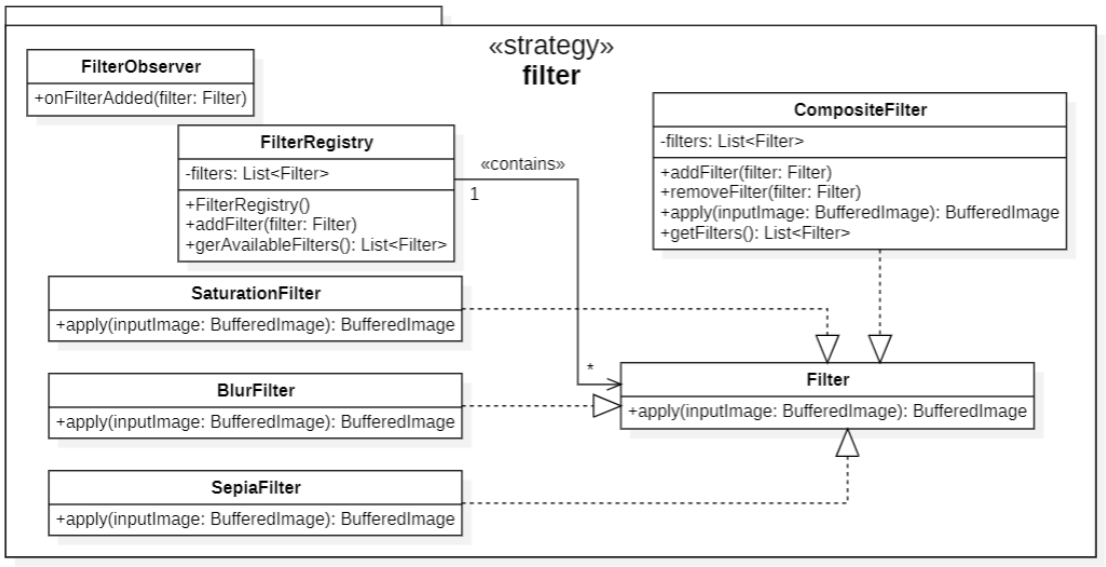

Figura 27: Diagramma UML del package Filter

```
public interface Filter {
  BufferedImage apply(BufferedImage inputImage);
}

public class GrayScaleFilter implements Filter {
  @Override
  public BufferedImage apply(BufferedImage inputImage) {
    int width = inputImage.getWidth();
    int height = inputImage.getHeight();
    BufferedImage outputImage = new BufferedImage(
    width, height, inputImage.getType());

    for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
        Color color = new Color(inputImage.getRGB(x, y));
        // Formula standard per la conversione in scala di grigi
        int gray = (int)(color.getRed() * 0.299 +
        color.getGreen() * 0.587 +
        color.getBlue() * 0.114);
        Color grayColor = new Color(gray, gray, gray);
        outputImage.setRGB(x, y, grayColor.getRGB());
      }
    }
    return outputImage;
  }
}
```

Listing 10: Interfaccia Filter e implementazione GrayScaleFilter

Ogni filtro implementa algoritmi specifici per la manipolazione delle immagini. Ad esempio, il `SepiaFilter` applica una trasformazione matriciale per ottenere l'effetto seppia:

```
public class SepiaFilter implements Filter {
  @Override
  public BufferedImage apply(BufferedImage inputImage) {
    int width = inputImage.getWidth();
    int height = inputImage.getHeight();
    BufferedImage outputImage = new BufferedImage(
    width, height, inputImage.getType());

    for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
        Color c = new Color(inputImage.getRGB(x, y));
        int red = c.getRed();
        int green = c.getGreen();
        int blue = c.getBlue();

        // Applicazione della matrice di trasformazione seppia
        int tr = (int)(0.393 * red + 0.769 * green + 0.189 * blue);
        int tg = (int)(0.349 * red + 0.686 * green + 0.168 * blue);
        int tb = (int)(0.272 * red + 0.534 * green + 0.131 * blue);

        // Clamp dei valori tra 0 e 255
        tr = Math.min(255, tr);
        tg = Math.min(255, tg);
        tb = Math.min(255, tb);

        Color sepia = new Color(tr, tg, tb);
        outputImage.setRGB(x, y, sepia.getRGB());
      }
    }
    return outputImage;
  }
}
```

Listing 11: Implementazione del filtro Seppia con matrice di trasformazione

FilterRegistry Il `FilterRegistry` gestisce la registrazione e l'accesso ai filtri disponibili nel sistema:

```
public class FilterRegistry {
  private final List<Filter> filters = new ArrayList<>();

  public FilterRegistry() {
    // Registrazione automatica dei filtri disponibili
    addFilter(new GrayScaleFilter());
    addFilter(new InvertFilter());
    addFilter(new BlurFilter());
    addFilter(new SharpenFilter());
    addFilter(new SepiaFilter());
  }

  public void addFilter(Filter filter) {
    if (filter != null && !filters.contains(filter)) {
      filters.add(filter);
    }
  }

  public List<Filter> getAvailableFilters() {
    // Restituisce una copia per evitare modifiche esterne
    return new ArrayList<>(filters);
  }
}
```

Listing 12: Registry per la gestione centralizzata dei filtri

3.3.6 Package Model

Il package model contiene le classi che rappresentano le entità del dominio applicativo.

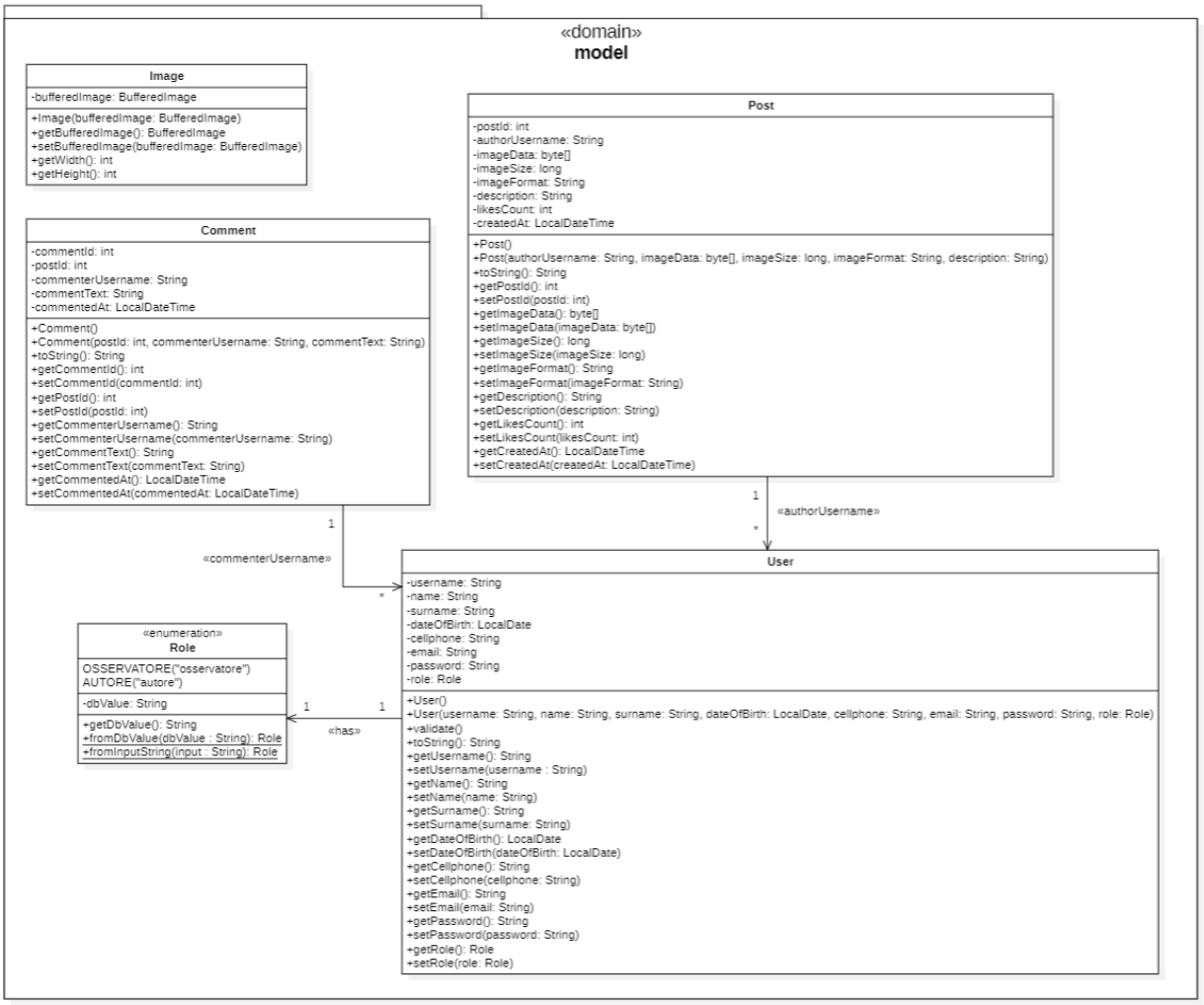

Figura 28: Diagramma UML del package Model

Classe User La classe `User` rappresenta un utente del sistema con tutti i suoi attributi:

```
public class User {
  private String username;
  private String name;
  private String surname;
  private LocalDate dateOfBirth;
  private String cellphone;
  private String email;
  private String password;
  private Role role;


  public User() {}
```

```
  public User(String username, String name, String surname,
  LocalDate dateOfBirth, String cellphone,
  String email, String password, Role role) {
    this.username = username;
    this.name = name;
    this.surname = surname;
    this.dateOfBirth = dateOfBirth;
    this.cellphone = cellphone;
    this.email = email;
    this.password = password;
    this.role = role;
  }

  // Metodi getter e setter...
  public String get\\
  Username() { return username; }
  public void setUsername(String username) { this.username = username; }
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public String getSurname() { return surname; }
  public void setSurname(String surname) { this.surname = surname; }
  public LocalDate getDateOfBirth() { return dateOfBirth; }
  public void setDateOfBirth(LocalDate dateOfBirth) { this.dateOfBirth =
  dateOfBirth; }
  public String getCellphone() { return cellphone; }
  public void setCellphone(String cellphone) { this.cellphone = cellphone; }
  public String getEmail() { return email; }
  public void setEmail(String email) { this.email = email; }
  public String getPassword() { return password; }
  public void setPassword(String password) { this.password = password; }
  public Role getRole() { return role; }
  public void setRole(Role role) { this.role = role; }

  public void validate() {
    Objects.requireNonNull(username, "username non puo' essere null");
    Objects.requireNonNull(name, "name non puo' essere null");
    Objects.requireNonNull(surname, "surname non puo' essere null");
    Objects.requireNonNull(dateOfBirth, "dateOfBirth non puo' essere null");
    Objects.requireNonNull(email, "email non puo' essere null");
    Objects.requireNonNull(password, "password non puo' essere null");
    Objects.requireNonNull(role, "role non puo' essere null");
  }
}
```

Listing 13: Classe User con metodo di validazione

Enum Role L'enum `Role` gestisce i ruoli degli utenti con mappatura tra valori di dominio e valori del database:

```
public enum Role {
  OSSERVATORE("osservatore"),
  AUTORE("autore");

  private final String dbValue;

  Role(String dbValue) {
    this.dbValue = dbValue;
  }
```

```
  public String getDbValue() {
    return dbValue;
  }

  public static Role fromDbValue(String dbValue) {
    for (Role r : values()) {
      if (r.dbValue.equalsIgnoreCase(dbValue)) {
        return r;
      }
    }
    throw new IllegalArgumentException("Unknown role: " + dbValue);
  }

  public static Role fromInputString(String input) {
    for (Role r : values()) {
      if (r.name().equalsIgnoreCase(input) ||
      r.getDbValue().equalsIgnoreCase(input)) {
        return r;
      }
    }
    throw new IllegalArgumentException(
    "Ruolo non valido: " + input +
    ". Usare 'osservatore' o 'autore'.");
  }
}
```

Listing 14: Enum Role con metodi di conversione

3.3.7 Package Service

Il package service contiene la business logic dell'applicazione, implementando le regole di business e orchestrando le operazioni tra i vari componenti.

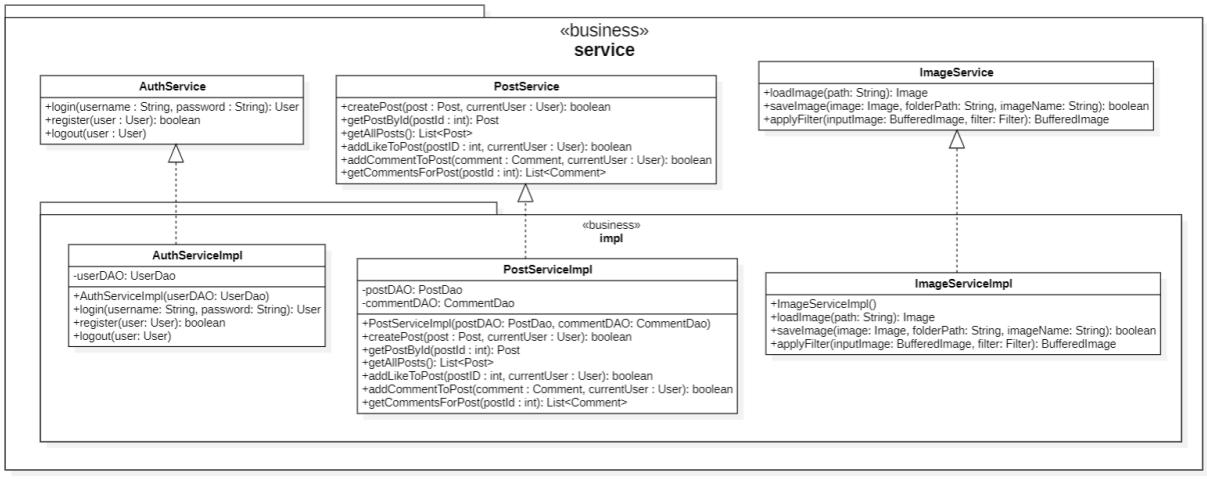

Figura 29: Diagramma UML del package Service

Interfacce Service Le interfacce definiscono i contratti per i servizi di business:

```
public interface AuthService {
  User login(String username, String password)
  throws AuthenticationException, SQLException;
  boolean register(User user)
  throws SQLException, IllegalArgumentException;
  void logout(User user);
}

public interface PostService {
  boolean createPost(Post post, User currentUser)
  throws SQLException, IllegalArgumentException;
  Post getPostById(int postId) throws SQLException;
  List<Post> getAllPosts() throws SQLException;
  boolean addLikeToPost(int postId, User currentUser)
  throws SQLException, IllegalArgumentException;
  boolean addCommentToPost(Comment comment, User currentUser)
  throws SQLException, IllegalArgumentException;
  List<Comment> getCommentsForPost(int postId)
  throws SQLException, IllegalArgumentException;
}
```

Listing 15: Interfacce dei servizi principali

Implementazioni Service Le implementazioni contengono la logica di business e le validazioni:

```
public class AuthServiceImpl implements AuthService {
  private final UserDao userDAO;

  public AuthServiceImpl(UserDao userDAO) {
    this.userDAO = userDAO;
  }

  @Override
  public User login(String username, String password)
  throws AuthenticationException, SQLException {
    // Validazione dei parametri di input
    if (username == null || username.trim().isEmpty() ||
    password == null || password.isEmpty()) {
      throw new IllegalArgumentException(
      "Username e password non possono essere vuoti.");
    }

    // Delega al DAO per l'effettiva autenticazione
    return userDAO.findByUsernameAndPassword(username, password);
  }

  @Override
  public boolean register(User user)
  throws SQLException, IllegalArgumentException {
    try {
      // Validazione completa dell'oggetto User
      user.validate();
    } catch (NullPointerException e) {
      throw new IllegalArgumentException(
      "Errore di validazione: " + e.getMessage(), e);
    }

    try {
      return userDAO.registerUser(user);
```

```
    } catch (SQLException e) {
      // Gestione specifica per violazione di unicita'
      if ("23505".equals(e.getSQLState())) {
        throw new SQLException(
        "Username o Email gia' esistente.",
        e.getSQLState(), e.getErrorCode(), e);
      }
      throw e;
    }
  }
}
```

Listing 16: AuthServiceImpl con validazioni

Il `PostServiceImpl` implementa logiche di business complesse per la gestione dei post:

```
public class PostServiceImpl implements PostService {
  private final PostDao postDAO;
  private final CommentDao commentDAO;

  // Costruttore...
  public PostServiceImpl(PostDao postDAO, CommentDao commentDAO){
    this.postDAO = postDAO;
    this.commentDAO = commentDAO;
  }

  @Override
  public boolean createPost(Post post, User currentUser)
  throws SQLException, IllegalArgumentException {
    // Validazioni multiple
    Objects.requireNonNull(post, "Il post non puo' essere nullo.");
    Objects.requireNonNull(currentUser,
    "L'utente corrente non puo' essere nullo.");

    // Verifica che l'autore del post corrisponda all'utente corrente
    if (!post.getAuthorUsername().equals(currentUser.getUsername())) {
      throw new IllegalArgumentException(
      "L'autore del post deve corrispondere all'utente corrente.");
    }

    // Solo gli AUTORI possono creare post
    if (currentUser.getRole() != Role.AUTORE) {
      throw new IllegalArgumentException(
      "Solo gli utenti con ruolo AUTORE possono creare post.");
    }

    // Validazione dei dati dell'immagine
    if (post.getImageData() == null || post.getImageData().length == 0) {
      throw new IllegalArgumentException(
      "I dati dell'immagine non possono essere vuoti.");
    }

    return postDAO.createPost(post);
  }

  @Override
  public boolean addCommentToPost(Comment comment, User currentUser)
  throws SQLException, IllegalArgumentException {
    // Validazioni del commento
    Objects.requireNonNull(comment, "Il commento non puo' essere nullo.");
    Objects.requireNonNull(currentUser,
    "L'utente corrente non puo' essere nullo.");

    if (comment.getCommentText() == null ||
    comment.getCommentText().trim().isEmpty()) {
      throw new IllegalArgumentException(
      "Il testo del commento non puo' essere vuoto.");
    }

    // Verifica che il post esista
    Post post = postDAO.getPostById(comment.getPostId());
    if (post == null) {
      throw new IllegalArgumentException(
      "Post non trovato con ID: " + comment.getPostId());
    }

    return commentDAO.addComment(comment);
  }
}
```

Listing 17: PostServiceImpl con controlli di autorizzazione

3.3.8 Package Util

Il package util contiene classi di utilità condivise da tutta l'applicazione.

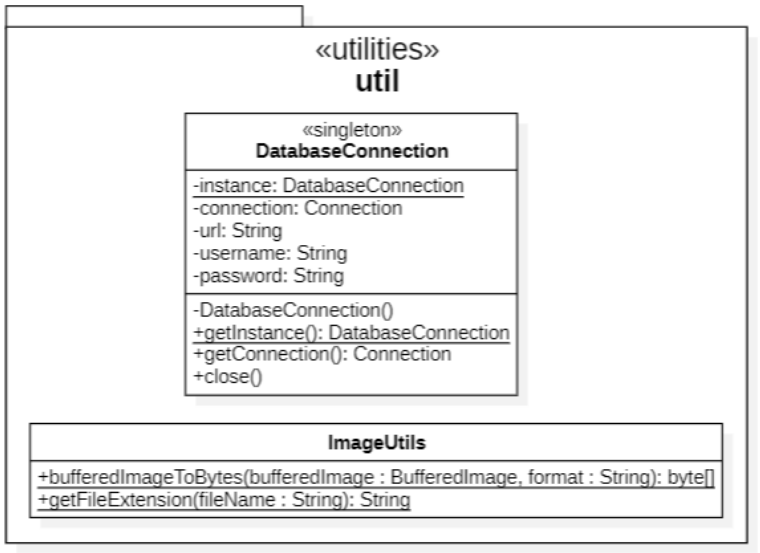

Figura 30: Diagramma UML del package Util

DatabaseConnection Implementa il pattern Singleton per gestire la connessione al database PostgreSQL:

```
public class DatabaseConnection {
  private static DatabaseConnection instance;
  private Connection connection;
  private final String url = "jdbc:postgresql://localhost:5432/ProgettoSWE";
  private final String username = "Caffettino";
  private final String password = "Polloallagriglia";

  private DatabaseConnection() throws SQLException {
    try {
      Class.forName("org.postgresql.Driver");
      this.connection = DriverManager.getConnection(
      url, username, password);
    } catch (ClassNotFoundException ex) {
      System.err.println("Driver PostgreSQL non trovato.");
      ex.printStackTrace();
      throw new SQLException("PostgreSQL driver not found.", ex);
    }
  }

  public Connection getConnection() throws SQLException {
    // Verifica e ri-stabilisce la connessione se necessario
    if (connection == null || connection.isClosed()) {
      try {
        this.connection = DriverManager.getConnection(
        url, username, password);
      } catch (SQLException e) {
        System.err.println(
        "Failed to re-establish database connection.");
```

```
            e.printStackTrace();
            throw e;
        }
      }
      return connection;
    }

    public static synchronized DatabaseConnection getInstance()
    throws SQLException {
      if (instance == null) {
        instance = new DatabaseConnection();
      }
      return instance;
    }

    public void close() {
      try {
        if (connection != null && !connection.isClosed()) {
          connection.close();
          System.out.println("Database connection closed.");
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
}
```

Listing 18: Singleton per la gestione della connessione database

ImageUtils Fornisce metodi di utilità per la manipolazione delle immagini:

```
public class ImageUtils {
  public static byte[] bufferedImageToBytes(
  BufferedImage bufferedImage, String format) {
    if (bufferedImage == null) {
      System.err.println(
      "BufferedImage non valido per la conversione in byte.");
      return null;
    }
    if (format == null || format.trim().isEmpty()) {
      System.err.println(
      "Formato immagine non specificato per la conversione.");
      return null;
    }

    try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
      ImageIO.write(bufferedImage, format, baos);
      return baos.toByteArray();
    } catch (IOException e) {
      System.err.println(
      "Errore durante la conversione dell'immagine: " +
      e.getMessage());
      e.printStackTrace();
      return null;
    }
  }

  public static String getFileExtension(String fileName) {
    if (fileName == null || fileName.lastIndexOf('.') == -1) {
      return "";
    }
    return fileName.substring(
    fileName.lastIndexOf('.') + 1).toLowerCase();
  }
}
```

Listing 19: Utility per la conversione di immagini

Certamente. Il problema nel testo che hai fornito è dovuto alla mancata chiusura degli ambienti LaTeX ‘tabularx‘ e ‘table‘. La sezione si interrompe bruscamente dopo la didascalia della tabella, causando un errore di compilazione.

4 Testing

4.1 Strategia di Testing

Il sistema di testing per la seguente applicazione è stato progettato seguendo una strategia a due livelli, implementata utilizzando il framework JUnit 5. La suite di test è organizzata in due categorie principali che garantiscono sia la correttezza delle singole componenti che il funzionamento end-to-end dell'applicazione.

- **Test Funzionali:** Verificano l'intero flusso dei casi d'uso, simulando le interazioni reali degli utenti con il sistema. Questi test coinvolgono multiple componenti integrate (Controller → Service → DAO → Database) e validano che i requisiti funzionali siano soddisfatti.
- **Test Strutturali:** Esaminano le singole unità di codice in isolamento, utilizzando mock objects dove necessario. Questi test verificano la correttezza della logica interna di ogni classe, i casi limite e la gestione degli errori.

4.2 Test Funzionali

I test funzionali verificano l'intero flusso applicativo per ogni caso d'uso identificato nella fase di analisi.

4.2.1 UC01_RegistrazioneTest

Descrizione: Verifica il flusso completo di registrazione utente, dalla richiesta iniziale al salvataggio nel database.

Componenti testate: AuthController → AuthService → UserDao → Database

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#1 - Registrazione Utente")
public class UC01_RegistrazioneTest extends BaseTest {

  private AuthService authService;
  private UserDao userDao;
  private Connection testConnection;

  @Override
  protected void additionalSetup() throws Exception {
    testConnection = TestDatabaseConfig.getTestConnection();

    userDao = new UserDAOImpl() {
      @Override
      protected Connection getConnection() throws SQLException {
        return testConnection;
      }
    };

    authService = new AuthServiceImpl(userDao);
  }

  @Test
  @Order(1)
```

```
@DisplayName("Registrazione con tutti i campi validi (OSSERVATORE)")
void testRegistrazioneOsservatoreCompleta() throws SQLException {
  User nuovoUtente = TestDataBuilder.createCustomUser(
  "mario_rossi",
  "Mario",
  "Rossi",
  LocalDate.of(1990, 5, 15),
  "3331234567",
  "mario.rossi@email.com",
  "password123",
  Role.OSSERVATORE
  );

  boolean risultato = authService.register(nuovoUtente);

  assertTrue(risultato, "La registrazione dovrebbe avere successo");

  assertUserExistsInDatabase("mario_rossi");

  User utenteLoggato = authService.login("mario_rossi", "password123");
  assertNotNull(utenteLoggato);
  assertEquals("Mario", utenteLoggato.getName());
  assertEquals(Role.OSSERVATORE, utenteLoggato.getRole());
}

@Test
@Order(2)
@DisplayName("Registrazione con tutti i campi validi (AUTORE)")
void testRegistrazioneAutoreCompleta() throws SQLException {
  User nuovoAutore = TestDataBuilder.createCustomUser(
  "giovanni_verdi",
  "Giovanni",
  "Verdi",
  LocalDate.of(1985, 3, 20),
  "3339876543",
  "giovanni.verdi@email.com",
  "securepass456",
  Role.AUTORE
  );

  boolean risultato = authService.register(nuovoAutore);

  assertTrue(risultato);
  assertUserExistsInDatabase("giovanni_verdi");

  User utenteLoggato = authService.login("giovanni_verdi", "securepass456");
  assertEquals(Role.AUTORE, utenteLoggato.getRole());
}

@Test
@Order(5)
@DisplayName("Registrazione con username gia' esistente")
void testRegistrazioneUsernameDuplicato() throws SQLException {
  User primoUtente = TestDataBuilder.createTestUser("utente_esistente", Role.
OSSERVATORE);
  authService.register(primoUtente);

  User secondoUtente = TestDataBuilder.createCustomUser(
  "utente_esistente",
  "Altro",
  "Nome",
  LocalDate.of(1992, 1, 1),
```

```
    "3337777777",
    "altra.email@test.com",
    Role.AUTORE
    );

    SQLException exception = assertThrows(SQLException.class, () -> {
      authService.register(secondoUtente);
    });

    assertTrue(exception.getMessage().contains("Username o Email gia' esistente"
  ));
  }
}
```

Listing 20: Test funzionale per la registrazione utente

4.2.2 UC02_LoginTest

Descrizione: Verifica il processo di autenticazione utente con diversi scenari.
Componenti testate: AuthController → AuthService → UserDao → Database

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#2 - Login Utente")
public class UC02_LoginTest extends BaseTest {

  private AuthService authService;
  private Connection testConnection;

  @Override
  protected void additionalSetup() throws Exception {
    testConnection = TestDatabaseConfig.getTestConnection();

    UserDao userDao = new UserDAOImpl() {
      protected Connection getConnection() throws SQLException {
        return testConnection;
      }
    };

    authService = new AuthServiceImpl(userDao);

    User osservatore = TestDataBuilder.createTestUser("osservatore1", Role.
OSSERVATORE);
    User autore = TestDataBuilder.createTestUser("autore1", Role.AUTORE);
    authService.register(osservatore);
    authService.register(autore);
  }

  @Test
  @Order(1)
  @DisplayName("Login con credenziali corrette (OSSERVATORE)")
  void testLoginOsservatoreValido() throws SQLException, AuthenticationException
   {
    User loggedUser = authService.login("osservatore1", "password123");

    assertNotNull(loggedUser);
    assertEquals("osservatore1", loggedUser.getUsername());
    assertEquals(Role.OSSERVATORE, loggedUser.getRole());
    assertNotNull(loggedUser.getName());
    assertNotNull(loggedUser.getEmail());
  }
```

```
  @Test
  @Order(2)
  @DisplayName("Login con credenziali corrette (AUTORE)")
  void testLoginAutoreValido() throws SQLException, AuthenticationException {
    User loggedUser = authService.login("autore1", "password123");

    assertNotNull(loggedUser);
    assertEquals("autore1", loggedUser.getUsername());
    assertEquals(Role.AUTORE, loggedUser.getRole());
  }

  @Test
  @Order(3)
  @DisplayName("Login con username non esistente")
  void testLoginUsernameNonEsistente() {
    assertThrows(AuthenticationException.class, () -> {
      authService.login("utenteInesistente", "password123");
    });
  }

  @Test
  @Order(4)
  @DisplayName("Login con password errata")
  void testLoginPasswordErrata() {
    AuthenticationException exception = assertThrows(AuthenticationException.
  class, () -> {
      authService.login("osservatore1", "passwordSbagliata");
    });

    assertTrue(exception.getMessage().contains("Credenziali non valide"));
  }
}
```

Listing 21: Test funzionale per il login utente

4.2.3 UC03_LogoutTest

Descrizione: Verifica il processo di logout e reset delle risorse.

Componenti testate: AuthController → AuthService

```
@DisplayName("UC#3 - Logout Utente")
public class UC03_LogoutTest extends BaseTest {

  private AuthService authService;
  private User osservatoreLoggato;
  private User autoreLoggato;

  @Override
  protected void additionalSetup() throws Exception {
    authService = mock(AuthService.class);

    osservatoreLoggato = TestDataBuilder.createTestUser("osservatore1", Role.
  OSSERVATORE);
    autoreLoggato = TestDataBuilder.createTestUser("autore1", Role.AUTORE);
  }

  @Test
  @DisplayName("Logout utente OSSERVATORE")
  void testLogoutOsservatore() {
```

```
    Scanner scanner = new Scanner("");
    AuthController authController = new AuthController(authService, scanner);

    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    PrintStream originalOut = System.out;
    System.setOut(new PrintStream(outputStream));

    User risultato = authController.logoutUser(osservatoreLoggato);

    assertNull(risultato, "Il metodo logout dovrebbe ritornare null");
    verify(authService, times(1)).logout(osservatoreLoggato);

    String output = outputStream.toString();
    assertTrue(output.contains("osservatore1 ha effettuato il logout"));

    System.setOut(originalOut);
  }

  @Test
  @DisplayName("Verifica reset immagine caricata dopo logout")
  void testResetImmagineDopoLogout() {
    Image immaginaCaricata = TestDataBuilder.createTestImage(100, 100);
    assertNotNull(immaginaCaricata);

    User currentUser = autoreLoggato;
    currentUser = null;
    immaginaCaricata = null;

    assertNull(currentUser);
    assertNull(immaginaCaricata);
  }
}
```

Listing 22: Test funzionale per il logout

4.2.4 UC04_VisualizzaPostTest

Descrizione: Testa la visualizzazione dei post per diversi tipi di utente.

Componenti testate: PostController → PostService → PostDao → Database

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#4 - Visualizza Elenco Post")
public class UC04_VisualizzaPostTest extends BaseTest {

  private PostService postService;
  private Connection testConnection;

  @Override
  protected void additionalSetup() throws Exception {
    testConnection = TestDatabaseConfig.getTestConnection();

    PostDao postDao = new PostDAOImpl() {
      protected Connection getConnection() throws SQLException {
        return testConnection;
      }
    };

    CommentDao commentDao = new CommentDAOImpl() {
      protected Connection getConnection() throws SQLException {
        return testConnection;
```

```
        }
      };

      postService = new PostServiceImpl(postDao, commentDao);

      TestDatabaseConfig.getTestConnection().createStatement().execute(
      "INSERT INTO users (username, name, surname, dateofbirth, email, password,
    role) " +
      "VALUES ('autore1', 'Nome', 'Cognome', '1990-01-01', 'autore1@test.com', '
    pass', 'autore')," +
      "('autore2', 'Nome2', 'Cognome2', '1991-01-01', 'autore2@test.com', 'pass',
    'autore')"
      );
    }

    @Test
    @Order(1)
    @DisplayName("Visualizzazione con post presenti nel sistema")
    void testVisualizzazioneConPost() throws SQLException {
      Post post1 = TestDataBuilder.createTestPost("autore1", "Primo post");
      Post post2 = TestDataBuilder.createTestPost("autore2", "Secondo post");

      User autore = TestDataBuilder.createTestUser("autore1", Role.AUTORE);
      postService.createPost(post1, autore);

      autore.setUsername("autore2");
      postService.createPost(post2, autore);

      List<Post> posts = postService.getAllPosts();

      assertNotNull(posts);
      assertEquals(2, posts.size());
      assertTrue(posts.stream().anyMatch(p -> p.getDescription().equals("Primo
    post")));
      assertTrue(posts.stream().anyMatch(p -> p.getDescription().equals("Secondo
    post")));
    }

    @Test
    @Order(5)
    @DisplayName("Ordinamento post per data (piu' recenti prima)")
    void testOrdinamentoPerData() throws SQLException, InterruptedException {
      Post post1 = TestDataBuilder.createTestPost("autore1", "Post vecchio");
      User autore = TestDataBuilder.createTestUser("autore1", Role.AUTORE);
      postService.createPost(post1, autore);

      Thread.sleep(100);

      Post post2 = TestDataBuilder.createTestPost("autore1", "Post nuovo");
      postService.createPost(post2, autore);

      List<Post> posts = postService.getAllPosts();

      assertEquals(2, posts.size());
      assertEquals("Post nuovo", posts.get(0).getDescription());
      assertEquals("Post vecchio", posts.get(1).getDescription());
    }
}
```

Listing 23: Test visualizzazione post

4.2.5 UC05_PubblicaPostTest

Descrizione: Verifica il processo di pubblicazione post con controllo del ruolo AUTORE.

Componenti testate: PostController → ImageUtils → PostService → PostDao

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#5 - Pubblica Post")
public class UC05_PubblicaPostTest extends BaseTest {

  private PostService postService;
  private Connection testConnection;
  private User autoreTest;
  private User osservatoreTest;

  @Override
  protected void additionalSetup() throws Exception {
    testConnection = TestDatabaseConfig.getTestConnection();

    PostDao postDao = new PostDAOImpl() {
      protected Connection getConnection() throws SQLException {
        return testConnection;
      }
    };

    CommentDao commentDao = new CommentDAOImpl() {
      protected Connection getConnection() throws SQLException {
        return testConnection;
      }
    };

    postService = new PostServiceImpl(postDao, commentDao);

    testConnection.createStatement().execute(
    "INSERT INTO users (username, name, surname, dateofbirth, email, password,
role) " +
    "VALUES ('autoretest', 'Autore', 'Test', '1990-01-01', 'autore@test.com', '
pass', 'autore')," +
    "('osservatoretest', 'Osservatore', 'Test', '1991-01-01', 'oss@test.com', '
pass', 'osservatore')"
    );

    autoreTest = TestDataBuilder.createTestUser("autoretest", Role.AUTORE);
    osservatoreTest = TestDataBuilder.createTestUser("osservatoretest", Role.
OSSERVATORE);
  }

  @Test
  @Order(1)
  @DisplayName("Pubblicazione post con immagine caricata e descrizione")
  void testPubblicazionePostCompleto() throws SQLException {
    byte[] imageData = TestDataBuilder.createTestImageData();
    Post nuovoPost = new Post(
    "autoretest",
    imageData,
    imageData.length,
    "png",
    "Questo e' un post di test con descrizione"
    );

    boolean risultato = postService.createPost(nuovoPost, autoreTest);
```

```
    assertTrue(risultato);
    assertNotNull(nuovoPost.getPostId());
    assertTrue(nuovoPost.getPostId() > 0);

    Post postSalvato = postService.getPostById(nuovoPost.getPostId());
    assertNotNull(postSalvato);
    assertEquals("Questo e' un post di test con descrizione", postSalvato.
  getDescription());
    assertEquals("png", postSalvato.getImageFormat());
  }

  @Test
  @Order(7)
  @DisplayName("Pubblicazione come OSSERVATORE (non permesso)")
  void testPubblicazioneComeOsservatore() {
    Post post = TestDataBuilder.createTestPost("osservatoretest", "Test
  osservatore");

    IllegalArgumentException exception = assertThrows(IllegalArgumentException.
  class, () -> {
      postService.createPost(post, osservatoreTest);
    });

    assertTrue(exception.getMessage().contains("AUTORE"));
  }
}
```

Listing 24: Test pubblicazione post

4.2.6 UC06_CaricaImmagineTest

Descrizione: Verifica il caricamento di immagini dal file system.
Componenti testate: ImageController → ImageService

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#6 - Carica Immagine")
public class UC06_CaricaImmagineTest extends BaseTest {

  private ImageService imageService;
  private static final String TEST_IMAGES_DIR = "test/resources/test-images/";
  private static Path testPngPath;
  private static Path testJpgPath;
  private static Path testTxtPath;

  @BeforeAll
  static void createTestFiles() throws IOException {
    Files.createDirectories(Paths.get(TEST_IMAGES_DIR));

    testPngPath = Paths.get(TEST_IMAGES_DIR, "test.png");
    BufferedImage pngImage = new BufferedImage(100, 100, BufferedImage.
  TYPE_INT_RGB);
    Graphics2D g = pngImage.createGraphics();
    g.setColor(Color.RED);
    g.fillRect(0, 0, 100, 100);
    g.dispose();
    ImageIO.write(pngImage, "png", testPngPath.toFile());

    testJpgPath = Paths.get(TEST_IMAGES_DIR, "test.jpg");
```

```
    BufferedImage jpgImage = new BufferedImage(200, 200, BufferedImage.
  TYPE_INT_RGB);
    g = jpgImage.createGraphics();
    g.setColor(Color.BLUE);
    g.fillRect(0, 0, 200, 200);
    g.dispose();
    ImageIO.write(jpgImage, "jpg", testJpgPath.toFile());

    testTxtPath = Paths.get(TEST_IMAGES_DIR, "notanimage.txt");
    Files.write(testTxtPath, "Questo non e' un'immagine".getBytes());
  }

  @Override
  protected void additionalSetup() throws Exception {
    imageService = new ImageServiceImpl();
  }

  @Test
  @Order(1)
  @DisplayName("Caricamento immagine PNG valida")
  void testCaricamentoPngValido() throws IOException {
    Image immagineCaricata = imageService.loadImage(testPngPath.toString());

    assertNotNull(immagineCaricata);
    assertNotNull(immagineCaricata.getBufferedImage());
    assertEquals(100, immagineCaricata.getWidth());
    assertEquals(100, immagineCaricata.getHeight());
  }

  @Test
  @Order(2)
  @DisplayName("Caricamento immagine JPG valida")
  void testCaricamentoJpgValido() throws IOException {
    Image immagineCaricata = imageService.loadImage(testJpgPath.toString());

    assertNotNull(immagineCaricata);
    assertNotNull(immagineCaricata.getBufferedImage());
    assertEquals(200, immagineCaricata.getWidth());
    assertEquals(200, immagineCaricata.getHeight());
  }

  @Test
  @Order(5)
  @DisplayName("Caricamento file non esistente")
  void testCaricamentoFileNonEsistente() {
    IOException exception = assertThrows(IOException.class, () -> {
      imageService.loadImage("percorso/inesistente/immagine.png");
    });

    assertTrue(exception.getMessage().contains("File non trovato"));
  }
}
```

Listing 25: Test caricamento immagini

4.2.7 UC07_ModificaImmagineTest

Descrizione: Verifica l'applicazione dei filtri alle immagini.
Componenti testate: ImageController → ImageService → FilterRegistry

```java
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#7 - Modifica Immagine")
public class UC07_ModificaImmagineTest extends BaseTest {

  private ImageService imageService;
  private Image testImage;
  private FilterRegistry filterRegistry;

  @Override
  protected void additionalSetup() throws Exception {
    imageService = new ImageServiceImpl();
    filterRegistry = new FilterRegistry();

    BufferedImage bufferedImage = new BufferedImage(100, 100, BufferedImage.
  TYPE_INT_RGB);
    for (int x = 0; x < 100; x++) {
      for (int y = 0; y < 100; y++) {
        int r = (x * 255) / 100;
        int g = (y * 255) / 100;
        int b = 128;
        bufferedImage.setRGB(x, y, new Color(r, g, b).getRGB());
      }
    }
    testImage = new Image(bufferedImage);
  }

  @Test
  @Order(1)
  @DisplayName("Applicazione filtro GrayScale")
  void testApplicazioneGrayScale() {
    Filter grayScaleFilter = new GrayScaleFilter();
    BufferedImage original = testImage.getBufferedImage();

    BufferedImage risultato = imageService.applyFilter(original, grayScaleFilter
  );

    assertNotNull(risultato);
    assertEquals(original.getWidth(), risultato.getWidth());
    assertEquals(original.getHeight(), risultato.getHeight());

    Color pixelColor = new Color(risultato.getRGB(50, 50));
    assertEquals(pixelColor.getRed(), pixelColor.getGreen());
    assertEquals(pixelColor.getGreen(), pixelColor.getBlue());
  }

  @Test
  @Order(5)
  @DisplayName("Applicazione filtro Sepia")
  void testApplicazioneSepia() {
    Filter sepiaFilter = new SepiaFilter();
    BufferedImage original = testImage.getBufferedImage();

    BufferedImage risultato = imageService.applyFilter(original, sepiaFilter);

    assertNotNull(risultato);

    Color sepiaColor = new Color(risultato.getRGB(50, 50));
    assertTrue(sepiaColor.getRed() >= sepiaColor.getBlue(),
    "Il filtro seppia dovrebbe aumentare i toni rossi");
  }

  @Test
```

```
    @Order(6)
    @DisplayName("Applicazione Luminosita'/Contrasto")
    void testApplicazioneLuminositaContrasto() {
      Filter brightnessContrastFilter = new BrightnessContrastFilter(50, 1.5);
      BufferedImage original = testImage.getBufferedImage();

      BufferedImage risultato = imageService.applyFilter(original,
    brightnessContrastFilter);

      assertNotNull(risultato);

      Color originalColor = new Color(original.getRGB(50, 50));
      Color modifiedColor = new Color(risultato.getRGB(50, 50));

      assertTrue(modifiedColor.getRed() > originalColor.getRed() ||
      modifiedColor.getGreen() > originalColor.getGreen() ||
      modifiedColor.getBlue() > originalColor.getBlue(),
      "L'immagine dovrebbe essere piu' luminosa");
    }
}
```

Listing 26: Test modifica immagini con filtri

4.2.8 UC08_SalvaImmagineTest

Descrizione: Verifica il salvataggio delle immagini nel file system locale.

Componenti testate: ImageController → ImageService

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#8 - Salva Immagine Locale")
public class UC08_SalvaImmagineTest extends BaseTest {

    private ImageService imageService;
    private Image testImage;
    private static Path testDirectory;

    @Override
    protected void additionalSetup() throws Exception {
      imageService = new ImageServiceImpl();
      testImage = TestDataBuilder.createTestImage(150, 150);

      testDirectory = Paths.get(System.getProperty("java.io.tmpdir"), "
    test_save_images");
      Files.createDirectories(testDirectory);
    }

    @Test
    @Order(1)
    @DisplayName("Salvataggio con nome valido")
    void testSalvataggioNomeValido() throws IOException {
      boolean risultato = imageService.saveImage(
      testImage,
      testDirectory.toString(),
      "immagine_test"
      );

      assertTrue(risultato);

      Path expectedFile = testDirectory.resolve("immagine_test.png");
```

```
    assertTrue(Files.exists(expectedFile), "Il file dovrebbe essere stato creato
  ");
    assertTrue(Files.size(expectedFile) > 0, "Il file non dovrebbe essere vuoto"
  );

    Files.deleteIfExists(expectedFile);
  }

  @Test
  @Order(2)
  @DisplayName("Creazione cartella se non esiste")
  void testCreazioneCartella() throws IOException {
    Path newDirectory = testDirectory.resolve("nuova_cartella");
    assertFalse(Files.exists(newDirectory), "La cartella non dovrebbe esistere
  inizialmente");

    boolean risultato = imageService.saveImage(
    testImage,
    newDirectory.toString(),
    "immagine_in_nuova_cartella"
    );

    assertTrue(risultato);
    assertTrue(Files.exists(newDirectory), "La cartella dovrebbe essere stata
  creata");

    Path expectedFile = newDirectory.resolve("immagine_in_nuova_cartella.png");
    assertTrue(Files.exists(expectedFile));

    Files.deleteIfExists(expectedFile);
    Files.deleteIfExists(newDirectory);
  }

  @Test
  @Order(3)
  @DisplayName("Aggiunta estensione .png automatica")
  void testAggiuntaEstensionePng() throws IOException {
    boolean risultato = imageService.saveImage(
    testImage,
    testDirectory.toString(),
    "immagine_senza_estensione"
    );

    assertTrue(risultato);

    Path expectedFile = testDirectory.resolve("immagine_senza_estensione.png");
    assertTrue(Files.exists(expectedFile), "Il file con estensione .png dovrebbe
   esistere");

    Path fileNoExt = testDirectory.resolve("immagine_senza_estensione");
    assertFalse(Files.exists(fileNoExt));

    Files.deleteIfExists(expectedFile);
  }
}
```

Listing 27: Test salvataggio immagini

4.2.9 UC09_MettiLikeTest

Descrizione: Verifica la funzionalità di aggiunta like ai post.

Componenti testate: PostController → PostService → PostDao

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#9 - Metti Like")
public class UC09_MettiLikeTest extends BaseTest {

  private PostService postService;
  private Connection testConnection;
  private User osservatore;
  private User autore;
  private Post testPost;

  @Override
  protected void additionalSetup() throws Exception {
    testConnection = TestDatabaseConfig.getTestConnection();

    PostDao postDao = new PostDAOImpl() {
      protected Connection getConnection() throws SQLException {
        return testConnection;
      }
    };

    CommentDao commentDao = new CommentDAOImpl() {
      protected Connection getConnection() throws SQLException {
        return testConnection;
      }
    };

    postService = new PostServiceImpl(postDao, commentDao);

    testConnection.createStatement().execute(
    "INSERT INTO users (username, name, surname, dateofbirth, email, password,
role) " +
    "VALUES ('osservatore1', 'Oss', 'Uno', '1990-01-01', 'oss1@test.com', 'pass
', 'osservatore')," +
    "('autore1', 'Aut', 'Uno', '1991-01-01', 'aut1@test.com', 'pass', 'autore')"
    );

    osservatore = TestDataBuilder.createTestUser("osservatore1", Role.
OSSERVATORE);
    autore = TestDataBuilder.createTestUser("autore1", Role.AUTORE);

    testPost = TestDataBuilder.createTestPost("autore1", "Post per test like");
    postService.createPost(testPost, autore);
  }

  @Test
  @Order(1)
  @DisplayName("Like da OSSERVATORE")
  void testLikeDaOsservatore() throws SQLException {
    int likesIniziali = testPost.getLikesCount();

    boolean risultato = postService.addLikeToPost(testPost.getPostId(),
osservatore);

    assertTrue(risultato);

    Post postAggiornato = postService.getPostById(testPost.getPostId());
    assertEquals(likesIniziali + 1, postAggiornato.getLikesCount());
```

```
    }

    @Test
    @Order(3)
    @DisplayName("Incremento contatore like")
    void testIncrementoContatoreLike() throws SQLException {
      Post post = TestDataBuilder.createTestPost("autore1", "Post multipli like");
      postService.createPost(post, autore);
      assertEquals(0, post.getLikesCount());

      postService.addLikeToPost(post.getPostId(), osservatore);
      postService.addLikeToPost(post.getPostId(), autore);
      postService.addLikeToPost(post.getPostId(), null);

      Post postAggiornato = postService.getPostById(post.getPostId());
      assertEquals(3, postAggiornato.getLikesCount());
    }

    @Test
    @Order(5)
    @DisplayName("Like a post non esistente")
    void testLikePostNonEsistente() {
      IllegalArgumentException exception = assertThrows(IllegalArgumentException.
    class, () -> {
postService.addLikeToPost(9999, osservatore);
});

assertTrue(exception.getMessage().contains("Post non trovato"));
}
}
```

Listing 28: Test funzionalita' like

4.2.10 UC10_AggiungiCommentoTest

Descrizione: Verifica la funzionalità di aggiunta commenti ai post.

Componenti testate: PostController → PostService → CommentDao

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#10 - Aggiungi Commento")
public class UC10_AggiungiCommentoTest extends BaseTest {

private PostService postService;
private Connection testConnection;
private User osservatore;
private User autore;
private Post testPost;

@Override
protected void additionalSetup() throws Exception {
testConnection = TestDatabaseConfig.getTestConnection();

PostDao postDao = new PostDAOImpl() {
protected Connection getConnection() throws SQLException {
  return testConnection;
}
};

CommentDao commentDao = new CommentDAOImpl() {
protected Connection getConnection() throws SQLException {
```

```
  return testConnection;
}
};

postService = new PostServiceImpl(postDao, commentDao);

testConnection.createStatement().execute(
"INSERT INTO users (username, name, surname, dateofbirth, email, password, role) "
     +
"VALUES ('osservatore1', 'Oss', 'Uno', '1990-01-01', 'oss1@test.com', 'pass', '
    osservatore'),'" +
"('autore1', 'Aut', 'Uno', '1991-01-01', 'aut1@test.com', 'pass', 'autore')"
);

osservatore = TestDataBuilder.createTestUser("osservatore1", Role.OSSERVATORE);
autore = TestDataBuilder.createTestUser("autore1", Role.AUTORE);

testPost = TestDataBuilder.createTestPost("autore1", "Post per test commenti");
postService.createPost(testPost, autore);
}

@Test
@Order(1)
@DisplayName("Commento da OSSERVATORE")
void testCommentoDaOsservatore() throws SQLException {
Comment commento = new Comment(
testPost.getPostId(),
osservatore.getUsername(),
"Questo e' un commento da osservatore"
);

boolean risultato = postService.addCommentToPost(commento, osservatore);

assertTrue(risultato);

List<Comment> commenti = postService.getCommentsForPost(testPost.getPostId());
assertEquals(1, commenti.size());
assertEquals("Questo e' un commento da osservatore", commenti.get(0).
    getCommentText());
assertEquals("osservatore1", commenti.get(0).getCommenterUsername());
}

@Test
@Order(2)
@DisplayName("Commento da AUTORE")
void testCommentoDaAutore() throws SQLException {
Comment commento = new Comment(
testPost.getPostId(),
autore.getUsername(),
"L'autore commenta il proprio post"
);

boolean risultato = postService.addCommentToPost(commento, autore);

assertTrue(risultato);

List<Comment> commenti = postService.getCommentsForPost(testPost.getPostId());
assertTrue(commenti.stream()
.anyMatch(c -> c.getCommentText().equals("L'autore commenta il proprio post")));
}

@Test
```

```
@Order(3)
@DisplayName("Commento con testo lungo")
void testCommentoTestoLungo() throws SQLException {
String testoLungo = "Questo e' un commento molto lungo che serve per testare " +
"se il sistema gestisce correttamente i commenti con molto testo. " +
"Potrebbe essere una recensione dettagliata o una discussione approfondita " +
"su un argomento specifico. Il sistema dovrebbe gestirlo senza problemi.";

Comment commento = new Comment(
testPost.getPostId(),
osservatore.getUsername(),
testoLungo
);

boolean risultato = postService.addCommentToPost(commento, osservatore);

assertTrue(risultato);

List<Comment> commenti = postService.getCommentsForPost(testPost.getPostId());
Comment commentoSalvato = commenti.stream()
.filter(c -> c.getCommentText().equals(testoLungo))
.findFirst()
.orElse(null);

assertNotNull(commentoSalvato);
assertEquals(testoLungo, commentoSalvato.getCommentText());
}

@Test
@Order(6)
@DisplayName("Commento con testo vuoto")
void testCommentoTestoVuoto() {
Comment commentoVuoto = new Comment(
testPost.getPostId(),
osservatore.getUsername(),
""
);

IllegalArgumentException exception = assertThrows(IllegalArgumentException.class,
    () -> {
postService.addCommentToPost(commentoVuoto, osservatore);
});

assertTrue(exception.getMessage().contains("testo del commento non puo' essere
    vuoto"));
}

@Test
@Order(7)
@DisplayName("Commento a post non esistente")
void testCommentoPostNonEsistente() {
Comment commento = new Comment(
9999,
osservatore.getUsername(),
"Commento a post inesistente"
);

IllegalArgumentException exception = assertThrows(IllegalArgumentException.class,
    () -> {
postService.addCommentToPost(commento, osservatore);
});
```

```
assertTrue(exception.getMessage().contains("Post non trovato"));
}
}
```

Listing 29: Test funzionalita' commenti

4.2.11 UC11_SelezionaFiltroTest

Descrizione: Verifica la selezione e applicazione dei filtri immagine.

Componenti testate: FilterRegistry → ImageService → Filter

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("UC#11 - Seleziona Filtro")
public class UC11_SelezionaFiltroTest extends BaseTest {

private ImageService imageService;
private FilterRegistry filterRegistry;
private Image testImage;

@Override
protected void additionalSetup() throws Exception {
imageService = new ImageServiceImpl();
filterRegistry = new FilterRegistry();
testImage = TestDataBuilder.createTestImage(200, 200);
}

@Test
@Order(1)
@DisplayName("Selezione di ogni filtro disponibile")
void testSelezioneOgniFiltroDisponibile() {
List<Filter> filtriDisponibili = filterRegistry.getAvailableFilters();
BufferedImage original = testImage.getBufferedImage();

assertFalse(filtriDisponibili.isEmpty(), "Dovrebbero esserci filtri disponibili");

for (Filter filtro : filtriDisponibili) {
BufferedImage risultato = imageService.applyFilter(original, filtro);

assertNotNull(risultato, "Il filtro " + filtro.getClass().getSimpleName() +
" dovrebbe produrre un risultato");
assertEquals(original.getWidth(), risultato.getWidth());
assertEquals(original.getHeight(), risultato.getHeight());
}
}

@Test
@Order(2)
@DisplayName("Visualizzazione anteprima")
void testVisualizzazioneAnteprima() {
Filter grayScaleFilter = new GrayScaleFilter();
BufferedImage original = testImage.getBufferedImage();

BufferedImage anteprima = imageService.applyFilter(original, grayScaleFilter);

assertNotNull(anteprima);
boolean isDifferent = false;
for (int x = 0; x < Math.min(10, original.getWidth()); x++) {
for (int y = 0; y < Math.min(10, original.getHeight()); y++) {
  if (original.getRGB(x, y) != anteprima.getRGB(x, y)) {
    isDifferent = true;
```

```
    break;
  }
}
}
assertTrue(isDifferent, "L'anteprima dovrebbe mostrare l'effetto del filtro");
}

@Test
@Order(4)
@DisplayName("Verifica presenza filtri standard")
void testPresenzaFiltriStandard() {
List<Filter> filtri = filterRegistry.getAvailableFilters();

assertTrue(filtri.stream().anyMatch(f -> f instanceof GrayScaleFilter),
"Dovrebbe esserci il filtro GrayScale");
assertTrue(filtri.stream().anyMatch(f -> f instanceof InvertFilter),
"Dovrebbe esserci il filtro Invert");
assertTrue(filtri.stream().anyMatch(f -> f instanceof BlurFilter),
"Dovrebbe esserci il filtro Blur");
assertTrue(filtri.stream().anyMatch(f -> f instanceof SharpenFilter),
"Dovrebbe esserci il filtro Sharpen");
assertTrue(filtri.stream().anyMatch(f -> f instanceof SepiaFilter),
"Dovrebbe esserci il filtro Sepia");
}

@Test
@Order(5)
@DisplayName("Selezione indice filtro non valido")
void testSelezioneIndiceFiltroNonValido() {
List<Filter> filtri = filterRegistry.getAvailableFilters();
int indiceNonValido = filtri.size() + 10;

assertThrows(IndexOutOfBoundsException.class, () -> {
filtri.get(indiceNonValido);
});
}
}
```

Listing 30: Test selezione filtri

4.3 Test Strutturali

I test strutturali verificano il comportamento delle singole classi in isolamento.

4.3.1 TEST04 - UserDAOImplTest

Descrizione: Test unitari per i metodi DAO di gestione utenti.
Classe testata: UserDAOImpl

```
@DisplayName("Test Strutturali - UserDAOImpl")
public class UserDAOImplTest extends BaseTest {

private UserDAOImpl userDAO;

@Test
@DisplayName("TEST04.1 - findByUsernameAndPassword con credenziali
    corrette")
```

```
void testFindByUsernameAndPassword_CredenzialiCorrette()
throws SQLException, AuthenticationException {

User utente = TestDataBuilder.createTestUser("testuser", Role.
    OSSERVATORE);
userDAO.registerUser(utente);


User trovato = userDAO.findByUsernameAndPassword(
"testuser", "password123");


assertNotNull(trovato);
}

@Test
@DisplayName("TEST04.2 - registerUser gestione SQLException 23505"
    )
void testRegisterUser_UsernameDuplicato() throws SQLException {

User utente1 = TestDataBuilder.createTestUser("duplicato", Role.
    OSSERVATORE);
userDAO.registerUser(utente1);

User utente2 = TestDataBuilder.createTestUser("duplicato", Role.
    AUTORE);


SQLException exception = assertThrows(SQLException.class, () -> {
userDAO.registerUser(utente2);
});


assertTrue(exception.getMessage().contains("Username o Email gia
    esistente") ||
exception.getMessage().toLowerCase().contains("unique"));
}
}
```

Listing 31: Test strutturale per UserDAOImpl

4.3.2 TEST05 - PostServiceImplTest

Descrizione: Test della logica di business per la gestione dei post con mock objects.
Classe testata: PostServiceImpl

```
@ExtendWith(MockitoExtension.class)
@DisplayName("Test Strutturali - PostServiceImpl")
public class PostServiceImplTest {
```

```
@Mock
private PostDao postDAO;
private PostServiceImpl postService;

@BeforeEach
void setUp() {
postService = new PostServiceImpl(postDAO, null); // commentDAO
    non usato qui
}

@Test
@DisplayName("TEST05.1 - createPost validazione ruolo AUTORE")
void testCreatePost_ControlloRuolo() {

Post post = TestDataBuilder.createTestPost("user", "Test");
User osservatore = TestDataBuilder.createTestUser("user", Role.
    OSSERVATORE);


assertThrows(IllegalArgumentException.class, () -> {
postService.createPost(post, osservatore);
});

verifyNoInteractions(postDAO);
}
}
```

Listing 32: Test con mock per PostServiceImpl

4.3.3 TEST06 - Filtri Immagine Test

Descrizione: Verifica l'applicazione corretta dei filtri alle immagini.
Classi testate: GrayScaleFilter, SepiaFilter, BlurFilter, etc.

```
@DisplayName("Test Strutturali - Filtri Immagine")
public class ImageFiltersTest {

private BufferedImage testImage;

@BeforeEach
void setUp() {
// Crea immagine di test con gradiente di colori
testImage = new BufferedImage(100, 100, BufferedImage.TYPE_INT_RGB
    );
Graphics2D g = testImage.createGraphics();
for (int x = 0; x < 100; x++) {
for (int y = 0; y < 100; y++) {
  int r = (x * 255) / 100;
```

```
  int g = (y * 255) / 100;
  int b = 128;
  testImage.setRGB(x, y, new Color(r, g, b).getRGB());
}
}
g.dispose();
}

@Test
@DisplayName("TEST06.1 - GrayScaleFilter verifica conversione RGB"
    )
void testGrayScaleFilter() {

Filter grayScaleFilter = new GrayScaleFilter();

BufferedImage result = grayScaleFilter.apply(testImage);

Color pixel = new Color(result.getRGB(50, 50));
assertEquals(pixel.getRed(), pixel.getGreen());
assertEquals(pixel.getGreen(), pixel.getBlue());
}

@Test
@DisplayName("TEST06.2 - SepiaFilter verifica matrice
    trasformazione")
void testSepiaFilter() {

Filter sepiaFilter = new SepiaFilter();

BufferedImage result = sepiaFilter.apply(testImage);
Color sepiaColor = new Color(result.getRGB(50, 50));

assertTrue(sepiaColor.getRed() >= sepiaColor.getBlue());
}
}
```

Listing 33: Test filtri immagine con verifica pixel

4.4 Copertura dei Test

La suite di test implementata garantisce una copertura completa dei seguenti aspetti:

Categoria	Descrizione	Test File
Autenticazione	Registrazione, login, logout con validazioni	UC01, UC02, UC03
Gestione Post	Creazione, visualizzazione, ordinamento	UC04, UC05
Gestione Immagini	Caricamento, modifica, salvataggio	UC06, UC07, UC08
Interazioni Social	Like e commenti con controlli autorizzazione	UC09, UC10
Filtri Immagine	Selezione e applicazione filtri	UC11
Test Strutturali	DAO, Service e filtri in isolamento	UserDAOImplTest

Tabella 1: Riepilogo copertura test per categoria

5 Realizzazione Progetto e Conclusione

Inizialmente, l'ambizione per il progetto *Image Manager* era di creare una soluzione software più estesa, che potesse includere un'interfaccia utente grafica (GUI) dedicata per un'interazione visuale più ricca. La componente principale, tuttavia, è sempre stata concepita come un robusto backend sviluppato in Java, destinato alla gestione e manipolazione delle immagini, all'interazione con gli utenti e alla gestione dei loro contenuti (post, commenti).

Data la complessità associata allo sviluppo di un'applicazione full-stack da parte due sviluppatori e il tempo a disposizione, si è scelto di concentrare gli sforzi sulla realizzazione di un'applicazione command-line (CLI) completa e funzionale.

L'applicativo è stato scritto utilizzando il linguaggio Java, specificamente nella versione Java SE 11. Il progetto conta 41 classi Java principali (escludendo eventuali classi di test ancora non realizzate). Le classi sono organizzate in molteplici package per promuovere la modularità e la manutenibilità del codice. Per l'implementazione è stato utilizzato l'IDE IntelliJ IDEA. L'applicazione interagisce con un database PostgreSQL attraverso il driver JDBC specifico per PostgreSQL per tutte le operazioni di persistenza dei dati.