# Software Engineering Audit Report

**imagemanager.pdf**

CAPRA

February 25, 2026

## Contents

# 1 Document Context

## Project Objective

The application is an image management system for a marketplace with integrated filters. It enables users to authenticate, upload and manipulate images using digital filters, publish posts with images, and interact socially through likes and comments. The system implements a layered architecture with a command-line interface, business logic services, and a PostgreSQL database backend.

## Main Use Cases

- UC01 – Registrazione: Visitor registers as a new user with validation of email format, username and email uniqueness, and user data storage.
- UC02 – Login: Visitor authenticates by verifying username and password against the database.
- UC03 – Logout: Authenticated user terminates the current session and returns to visitor state.
- UC04 – Visualizza Elenco Post: User views all published posts with author, description, likes, and date information.
- UC05 – Pubblica Post: Author publishes a loaded image as a new post with description and binary image conversion.
- UC06 – Carica Immagine: Author loads an image from the local file system into memory.
- UC07 – Modifica Immagine: Author applies digital filters (GrayScale, Invert, Blur, Sharpen, Sepia, Brightness/Contrast) to a loaded image.
- UC08 – Salva Immagine Locale: Author saves a loaded or modified image to the local file system in PNG format.
- UC09 – Metti Like: Authenticated user increments the like counter for a post.
- UC10 – Aggiungi Commento: Authenticated user adds a comment to a post with timestamp and username.
- UC11 – Seleziona Filtro: Author selects and applies a filter from the FilterRegistry to an image with preview capability.

## Functional Requirements

- User registration with validation of email format, username uniqueness, and email uniqueness.
- User authentication with username and password verification against the database.
- User logout with session termination and image memory reset.
- Post creation by authors with image binary conversion and storage in the database.
- Post visualization for all users with filtering and sorting capabilities.
- Image loading from file system with support for PNG and JPG formats.
- Image modification through application of multiple digital filters (GrayScale, Invert, Blur, Sharpen, Sepia, Brightness/Contrast).
- Image saving to local file system in PNG format with automatic folder creation.
- Like functionality with counter increment and database persistence.
- Comment functionality with text, timestamp, and username storage in the database.
- Filter registry management with dynamic filter registration and availability listing.
- Role-based access control distinguishing between OSSERVATORE (Observer) and AUTORE (Author) roles.
- Input validation and error handling with user-friendly error messages.

### Non-Functional Requirements

- Architecture: Layered architecture with Presentation (CLI), Controller, Service, Domain Model, and Data Access layers.
- Database: PostgreSQL with JDBC connectivity for data persistence.
- Technology Stack: Java SE 11, PostgreSQL driver, JUnit 5 for testing.
- Code Organization: Modular package structure (controller, service, dao, model, filter, util, exception) promoting maintainability.
- Design Patterns: Strategy pattern for filters, Singleton pattern for database connection, Data Access Object pattern for persistence.
- Exception Handling: Custom exceptions (AuthenticationException) for domain-specific error management.
- Image Processing: Support for BufferedImage manipulation and binary serialization.
- Testing: Comprehensive test coverage with functional and structural tests using JUnit 5 and Mockito.

### Architecture

The application follows a layered architecture with clear separation of concerns. The **Presentation Layer** consists of a command-line interface in the main package. The **Controller Layer** (package `controller`) contains `AuthController`, `ImageController`, and `PostController` that mediate between the CLI and business logic. The **Service Layer** (packages `service` and `service.impl`) implements business logic with classes like `AuthServiceImpl`, `ImageServiceImpl`, and `PostServiceImpl`. The **Domain Model** (package `model`) defines entities: `User`, `Post`, `Image`, and `Comment`. The **Data Access Layer** (packages `dao` and `dao.impl`) implements the DAO pattern with `UserDAOImpl`, `PostDAOImpl`, and `CommentDAOImpl` using JDBC for PostgreSQL interaction. The **Filter System** (package `filter`) implements the Strategy pattern with a `FilterRegistry` managing filter implementations. **Utility packages** provide database connection management (Singleton pattern), image serialization, and custom exceptions.

### Testing Strategy

The testing strategy employs a two-level approach using JUnit 5. **Functional Tests** verify complete use case flows with integrated components (Controller → Service → DAO → Database), validating end-to-end requirements. Tests cover registration (UC01RegistrazioneTest), login (UC02LoginTest), logout (UC03LogoutTest), post visualization (UC04VisualizzaPostTest), post publication (UC05PubblicaPostTest), image loading (UC06CaricaImmagineTest), image modification (UC07ModificaImmagineTest), image saving (UC08SalvaImmagineTest), likes (UC09MettiLikeTest), comments (UC10AggiungiCommentoTest), and filter selection (UC11SelezionaFiltroTest). **Structural Tests** examine individual units in isolation using mock objects: UserDAOImplTest validates DAO operations, PostServiceImplTest verifies business logic with mocks, and ImageFiltersTest confirms filter pixel-level transformations. Test coverage includes authentication, post management, image operations, social interactions, and filter functionality with comprehensive validation of success paths, error conditions, and edge cases.

## 2 Executive Summary

**Quick Overview**

Total issues: **14** — HIGH: **2** MEDIUM: **8** LOW: 4

Average confidence: **100%**

**Executive Summary: ImageManager Software Engineering Document Audit**

This document presents a comprehensive Software Engineering specification for an ImageManager application, including requirements, use cases, architecture, design, and testing. The document spans approximately 70 pages and covers functional requirements for three user roles (Visitor, Observer, Author), system architecture with layered MVC-style design, detailed use case specifications, and functional test coverage. The purpose is to serve as a complete specification and validation artifact for a university software engineering project.

The audit identified 14 issues across four categories. The predominant pattern is **incomplete and inconsistent specification of requirements and use cases**: use case diagrams contain relationships (include/extend) that are not reflected in textual specifications, several referenced use cases lack formal definitions, and critical business logic (authentication requirements, state management, error handling) is underspecified. A secondary pattern involves **gaps in test coverage and traceability**: while functional tests exist for main flows, alternative and error flows are only partially covered, controller-level behavior is not systematically validated, and the mapping between requirements and tests is implicit rather than explicit.

Two issues are classified as **HIGH severity**. ISS-003 identifies fundamental gaps in the like functionality specification: it is unclear whether users can like the same post multiple times, whether authentication is mandatory, and what happens when currentUser is null. This directly impacts data consistency and security. TST-001 reveals that error and alternative flows across multiple use cases lack corresponding negative tests, creating a systemic validation gap for exception handling and edge cases.

The overall quality assessment is **moderate with significant gaps**. The document demonstrates solid foundational work in architecture, design patterns, and positive-path testing. However, the requirements specification suffers from incompleteness and inconsistency between diagrams and text, and the testing strategy does not adequately validate the full scope of specified behavior, particularly error handling and authorization logic. The document is not yet ready for implementation or acceptance without substantial revision.

**Priority Actions:**

1. **Reconcile and complete use case specifications**: Resolve all discrepancies between use case diagrams and textual templates. Define all referenced use cases (e.g., "Visualizza Dettaglio Post", "Visualizza Commenti", "Visualizza Anteprima") with explicit flows, pre/post-conditions, and error handling. Clarify authentication and authorization requirements for each use case, especially for like, comment, and image operations.
2. **Expand test coverage to include error and alternative flows**: Develop negative test cases for all specified error conditions (invalid input, database errors, missing resources, failed operations). Create end-to-end tests at the Controller level to validate user messages and full use case flows, not just Service/DAO layers. Establish explicit traceability between each use case step and corresponding test methods.
3. **Specify critical business logic and state management**: Document precisely the behavior for edge cases in like functionality (duplicate likes, null user, authentication requirement), image state transitions (cancellation, filter failure, save failure), and comment access control. Ensure all preconditions are explicit and all postconditions are verifiable.

# 3 Strengths

- **Comprehensive Use Case Coverage:** The document presents well-structured use case diagrams and detailed templates covering all major system functionalities, including authentication (registration, login, logout), post management, image handling, social interactions (likes, comments), and filter application. Each use case includes pre-conditions, basic flows, alternative flows, and post-conditions, demonstrating thorough requirements analysis.
- **Well-Defined Layered Architecture:** The project implements a clear separation of concerns through a multi-layer architecture (Presentation, Controller, Service, Domain Model, and Data Access layers) with appropriate design patterns. The use of DAO pattern with JDBC for database interaction and the modular organization into distinct packages (controller, service, dao, model, filter, util, exception) shows solid architectural design principles.

- **Extensive Testing Strategy:** The document includes comprehensive testing coverage with both functional tests (11 use case tests: UC01-UC11) and structural tests (UserDAOImplTest, PostServiceImplTest, Filtri Immagine Test), demonstrating a systematic approach to quality assurance using JUnit framework.
- **Clear Visual Documentation:** The project includes detailed UML diagrams (Use Case Diagrams, Class Diagrams, Activity Diagrams) created with StarUML, along with UI mockups for each use case, providing clear visual representation of system design and user interactions.
- **Modular Filter Implementation:** The filter system is designed with modularity in mind, using a Filter interface and FilterRegistry for centralized management, allowing for flexible composition and extension of image filters (GrayScale, Invert, Blur, Sharpen, Sepia, Brightness/Contrast).

## 4 Expected Feature Coverage

Of 7 expected features: 6 present, 1 partial, 0 absent. Average coverage: **95%**.

| Feature | Status | Coverage | Evidence |
|---|---|---|---|
| Unit testing framework implementation | Present | 100% (5/5) | The Testing chapter shows extensive JUnit 5 usage with assertions (e.g., assertTrue, assertNotNull in UC01_RegistrazioneTest), describes a two-level strategy for Test Funzionali and Test Strutturali, defines many dedicated test classes per UC and component (e.g., UC02_LoginTest, UserDAOImplTest), uses mocks with Mockito in PostServiceImplTest, and explicitly distinguishes functional (integration) and structural (unit) tests. |
| Use of UML Diagrams for system modeling | Present | 100% (8/8) | Section 2.2 contains multiple UML use case diagrams for Visitatore, Osservatore, Autore, Gestione Autenticazione, and Gestione Filtri Immagine, while 3.2 and later subsections show UML class diagrams for the whole system and for packages like controller, dao, filter, model, service, util. Activity diagrams (Diagrammi delle attività) visualize flows, and numerous UI mockups (e.g., Figura 6–16) depict interfaces. The diagrams use standard UML notation with actors, use cases, include/extend, and clarify functional requirements and navigation flows. |

| Feature | Status | Coverage | Evidence |
|---|---|---|---|
| Identification and definition of system actors | Present | 100% (8/8) | Actors Visitatore, Osservatore, and Autore are clearly identified in the Statement and in use case diagrams, with their responsibilities and allowed actions listed (e.g., Autore può caricare immagini, applicare filtri, pubblicare post). Use case templates document interactions between these users and system components (controllers, services, FilterRegistry), with specific user actions and structured functional requirements per role (e.g., Gestione Autenticazione, Gestione Post, Interazioni Social). |
| Definition and Documentation of Use Cases | Present | 100% (5/5) | Section 2.3 provides detailed templates for Use Case #1–#11, each defining specific user interactions, a Brief Description (user goal), Basic Flow and Alternative Flow, and explicit Pre-Conditions and Post-Conditions (e.g., UC01 Registrazione, UC05 Pubblica Post). Relationships between use cases and subcases are indicated via include/extend both in text (e.g., include Valida formato mail) and in the diagrams. |
| User interface and interaction design principles | Partial | 71% (5/7) | The document includes many UI mockups for key interfaces such as Registrazione, Login, Menu Visitatore, Menu Autore, Visualizza Elenco Post, and comment/like screens, with clear navigation buttons (e.g., Visualizza Elenco Post, Logout). Validation mechanisms for user input are described in controller code and use cases (e.g., AuthController.registerUser validates date and role, UC01 validates email format and uniqueness). Structured input fields are shown in mockups (username, email, password, ruolo). Distinct user roles and their functionalities are defined, but there is no reservation creation process and no explicit description of dynamic UI updates of selections. |

| Feature | Status | Coverage | Evidence |
|---|---|---|---|
| Separation of concerns in software architecture | Present | 100% (5/5) | Section 1.2 describes a layered architecture with Presentation (CLI main), Controller, Service, Domain Model, DAO, and Utility packages, and later sections detail each package (controller, dao, model, service, util, filter). Interactions between controllers and domain models are shown in class diagrams and code (e.g., PostController.publishLoadedImageAsPost creates a Post model and calls PostService). Relationships between Controller, Service, and DAO are clearly defined (e.g., AuthController → AuthServiceImpl → UserDao/UserDAOImpl), and the modular package structure is justified for separation of responsibilities and maintainability. |
| Data Access Object (DAO) pattern | Present | 100% (7/7) | The DAO package is documented with interfaces UserDao, PostDao, CommentDao that define CRUD-like operations (e.g., createPost, getAllPosts, addLike, addComment, registerUser), and implementations like UserDAOImpl encapsulate SQL queries using JDBC (SELECT, INSERT). Separate DAOs exist per entity, and services depend on these interfaces to abstract database access (e.g., AuthServiceImpl uses UserDao, PostServiceImpl uses PostDao and CommentDao). Testing strategies for data access are shown in UserDAOImplTest and in functional tests that override getConnection to use a test database. |

## 5 Summary Table

| Category | HIGH | MEDIUM | LOW | Total |
|---|---|---|---|---|
| Architecture | 0 | 0 | 2 | 2 |
| Requirements | 1 | 5 | 1 | 7 |
| Testing | 1 | 3 | 1 | 5 |
| **Total** | **2** | **8** | **4** | **14** |

# 6 Issue Details

## 6.1 Architecture (2 issues)

### UC-1

**ISS-002** — LOW [100%] — Page 5 Nel diagramma dei casi d'uso per Visitatore (Figura 1) il caso d'uso "Login" è esteso da "Registrazione" (relazione «extend»), mentre nel template dei casi d'uso la registrazione e il login sono modellati come UC separati e indipendenti (UC1 e UC2). Analogamente, nei diagrammi per Osservatore e Autore alcune relazioni include/extend (es. Visualizza Elenco Post include Visualizza Dettaglio Post) non sono coerenti con la descrizione testuale, dove Visualizza Dettaglio Post non è definito come UC a sé. Queste discrepanze tra diagrammi e testo possono confondere il lettore e indebolire la qualità della documentazione.

> *2.2 Use Case Diagram I principali casi d'uso individuati sono: • Visitatore: – Pu'o effettuare la registrazione – Pu'o effettuare il login – Pu'o accedere all'elenco dei post*

**Recommendation:** Rivedi i diagrammi UML dei casi d'uso per allinearli rigorosamente ai template testuali: (1) per Visitatore, rimuovi la relazione «extend» tra Login e Registrazione, e rappresentali come due casi d'uso separati entrambi collegati all'attore; (2) per Osservatore e Autore, se decidi di mantenere "Visualizza Dettaglio Post" come UC incluso, definiscilo nella sezione 2.3; altrimenti, elimina l'uso di «include» e descrivi la selezione del post come passo interno di UC4. Dopo l'aggiornamento, controlla che ogni UC presente nei diagrammi abbia un corrispondente template testuale (o sia chiaramente indicato come sotto-funzione non documentata in dettaglio) per evitare incoerenze tra vista grafica e descrizione testuale.

### General Issues

**ISS-001** — LOW [100%] — Page 3 The layered architecture and separation of responsibilities are well described and largely consistent with the class and package structure, but the tests sometimes bypass the Controller layer and interact directly with Services and DAOs. This is acceptable for unit tests, but it means the documented MVC-style separation is not fully validated end-to-end for all flows.

> *Per mantenere una separazione delle responsabilit'a, la struttura del progetto 'e stata divisa in pi'u parti (package) principali che riflettono un'architettura a livelli.*

**Recommendation:** Keep the existing unit and integration tests, but complement them with a small number of true end-to-end tests per major feature that start from the controller methods (or even from Main if feasible) and go through Service and DAO. For example, for authentication and post publication, add tests that use AuthController and PostController together with a test database, verifying that the layered interactions behave as described. This will better demonstrate that the implemented architecture matches the design and that the layers collaborate correctly.

## 6.2 Requirements (7 issues)

### UC-09

**ISS-003** — HIGH [100%] — Page 17 La logica di business per i like non gestisce casi fondamentali: non è specificato se un utente può mettere like più volte allo stesso post, se è richiesto che l'utente sia autenticato (la pre-condizione non lo dice esplicitamente) e cosa succede se currentUser è null. Nei test UC09_MettiLikeTest viene addirittura verificato un like con currentUser null, il che contraddice l'idea di "utente" come attore e rende il requisito incompleto rispetto alla sicurezza e coerenza dei dati.

*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 'e incrementato di 1*

**Recommendation:** Estendi il caso d'uso Metti Like per coprire i vincoli di business: (1) aggiungi tra le pre-condizioni "L'utente è autenticato"; (2) specifica se un utente può mettere like più volte allo stesso post o solo una volta, e in quest'ultimo caso aggiungi un flusso alternativo tipo "Utente ha già messo like: il sistema non incrementa il contatore e mostra 'Hai già messo like a questo post'"; (3) vieta esplicitamente l'uso con utente null e aggiungi un flusso alternativo "Utente non autenticato: il sistema mostra 'Devi essere loggato per mettere like'". Allinea di conseguenza i test UC09_MettiLikeTest rimuovendo o modificando il caso che chiama addLikeToPost con currentUser null, in modo che i test riflettano i vincoli definiti nel requisito.

### UC-04

**ISS-004** — MEDIUM [100%] — Page 3 I requisiti sui ruoli e sui permessi non sono specificati in modo sistematico: il testo introduttivo parla di funzionalità differenziate per OSSERVATORE e AUTORE, ma nei casi d'uso non è sempre esplicitato se certe azioni sono vietate o semplicemente non previste per un ruolo (es. un Osservatore può o non può caricare immagini, può mettere like più volte, può mettere like senza essere autenticato?). Questo rende ambigua la logica di autorizzazione a livello di requisiti.

*Gli utenti possono autenticarsi nel sistema e, a seconda del ruolo (OSSERVATO- RE o AUTORE), accedere a funzionalit'a 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.*

**Recommendation:** Aggiungi una tabella o una sezione di requisiti che elenchi in modo esplicito, per ciascun ruolo (VISITATORE, OSSERVATORE, AUTORE), quali operazioni sono consentite, vietate o non applicabili (registrazione, login, visualizza elenco post, visualizza dettagli, like, commento, carica immagine, modifica immagine, pubblica post, salva immagine locale, seleziona filtro). Poi, per ogni caso d'uso UC04–UC11, aggiungi nelle pre-condizioni un riferimento esplicito al ruolo ammesso (es. "Attori: Osservatore, Autore" + "Pre-condizione: l'utente è autenticato" per Metti Like) e, dove necessario, un flusso alternativo per il ruolo non autorizzato (es. "Utente non autenticato: il sistema mostra 'Devi essere loggato per mettere like'").

**ISS-005** — MEDIUM [100%] — Page 12 UC04 fa riferimento a due casi d'uso inclusi, "Visualizza Dettaglio Post" e "Visualizza Commenti", che non sono definiti nella sezione 2.3 dei template dei casi d'uso. Questo rende incompleto il modello dei requisiti: non sono specificati i flussi, le pre/post-condizioni e gli errori per la visualizzazione del dettaglio di un post e dei relativi commenti, nonostante siano funzionalità centrali (anche nei mockup e nei diagrammi delle attività).

*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, Descri- zione, Likes, Data 4) L'utente pu'o selezionare un post per visualizzarne i det- tagli (≪include≫ Visualizza Dettaglio Post) 5) L'utente pu'o visualizzare i commenti di un post (≪inclu- de≫ Visualizza Commenti)*

**Recommendation:** Aggiungi due casi d'uso dettagliati nella sezione 2.3: (1) "Visualizza Dettaglio Post" con attori (Visitatore, Osservatore, Autore), pre-condizioni (es. "Il post esiste"), flusso base (selezione ID, recupero dal database, visualizzazione immagine/descrizione/likes/data) e flussi alternativi (post non trovato, errore database); (2) "Visualizza Commenti" con attori e flussi analoghi. Aggiorna i diagrammi delle attività (Figura 18 e 20) per fare riferimento esplicito a questi UC, in modo che il modello testuale e grafico sia coerente e completo.

**ISS-007** — MEDIUM [100%] — Page 18 Per Aggiungi Commento la pre-condizione richiede che l'utente sia autenticato, ma non è specificato se l'utente deve essere lo stesso che viene registrato nel commento (commenterUsername) né se un utente non autenticato possa comunque vedere i commenti. Inoltre, in UC04 Visualizza Elenco Post si include "Visualizza Commenti" ma non esiste un caso d'uso dedicato né sono definiti i requisiti di accesso (chi può vedere i commenti, in che ordine, con quali campi). Questo crea un buco di specifica sulla gestione dei commenti lato lettura e sulla coerenza tra utente autenticato e autore del commento.

> *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 'e autenticato • - L'utente sta visualizzando i dettagli di un post*

**Recommendation:** Definisci un caso d'uso separato "Visualizza Commenti" (richiamato da UC04) che specifichi attori, pre-condizioni (es. "Il post esiste"; "L'utente può essere anche Visitatore" se intendi permettere la sola lettura), flusso base (recupero e visualizzazione dei commenti con autore, testo, timestamp) e flussi alternativi (nessun commento, errore database). In UC10, aggiungi una pre-condizione o una nota di business logic che chiarisca che il campo commenterUsername del commento deve corrispondere all'utente autenticato, e aggiungi un flusso alternativo per il caso in cui questa coerenza non sia rispettata (anche solo come vincolo logico, se non gestito a runtime).

### UC-6

**ISS-006** — MEDIUM [100%] — Page 14 Per i casi d'uso di gestione immagini (Carica Immagine, Modifica Immagine, Salva Immagine Locale) le pre-condizioni e i flussi non specificano chiaramente cosa succede allo stato dell'immagine in memoria in tutti i casi di errore o annullamento. Ad esempio, in UC6 se l'immagine è già caricata e l'utente annulla la sovrascrittura, non è detto esplicitamente che l'immagine precedente rimanga invariata; in UC7, se l'applicazione del filtro fallisce, non è chiaro se l'immagine originale viene preservata; in UC8, non è specificato se un salvataggio fallito lascia l'immagine ancora disponibile per ulteriori tentativi. Questo può portare a comportamenti non documentati e a confusione nella logica di business.

> *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 'e autenticato come Autore*

**Recommendation:** Per UC6, UC7 e UC8, estendi le post-condizioni e i flussi alternativi per descrivere esplicitamente lo stato dell'immagine in memoria dopo ogni scenario: (1) in UC6, per il ramo "Immagine già caricata" specifica due sotto-flussi: "utente conferma sovrascrittura" (l'immagine in memoria viene sostituita) e "utente annulla" (l'immagine precedente resta invariata); (2) in UC7, per "Errore applicazione filtro" chiarisci che l'immagine in memoria rimane quella originale e che nessuna modifica viene applicata; (3) in UC8, per gli errori I/O o parametri non validi, specifica che l'immagine resta caricata e l'utente può riprovare il salvataggio. Questo rende il comportamento del sistema prevedibile e allineato con l'implementazione di ImageController.modifyLoadedImage e ImageService.saveImage.

### UC-11

**ISS-008** — MEDIUM [100%] — Page 19 Nel caso d'uso Seleziona Filtro, l'attore "Filter Registry" è modellato come attore nel diagramma di Figura 5, ma nel testo del caso d'uso è elencato tra gli attori insieme all'Autore. In realtà FilterRegistry è un componente interno (classe di servizio) e non un attore esterno. Inoltre, i casi d'uso "Visualizza Anteprima", "Applica Saturazione", "Applica Seppia", "Applica Sfocatura" e "Salva immagine modificata" sono usati come include/extend ma non sono definiti come casi d'uso separati, e la logica di anteprima non è distinta chiaramente dall'applicazione definitiva del filtro.

> *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 pu'o mostrare un'anteprima (≪include≫Visua- lizza Anteprima) 4) Il sistema applica il filtro selezionato (≪extend≫Applica Saturazione, Applica Seppia, Applica Sfocatura) 5) L'Autore pu'o salvare l'immagine modificata (≪ex- tend≫Salva immagine modificata)*

**Recommendation:** Correggi il modello degli attori rimuovendo "Filter Registry" dall'elenco degli attori di UC11 e dal diagramma dei casi d'uso: deve essere rappresentato come componente interno, non come attore. Poi, o (a) definisci esplicitamente i casi d'uso secondari "Visualizza Anteprima" e "Salva immagine modificata" (con flussi, pre/post-condizioni) e sostituisci "Applica Saturazione/Seppia/Sfocatura" con un unico UC generico "Applica Filtro"; oppure (b) semplifica UC11 incorporando anteprima e salvataggio nel flusso base, eliminando gli include/extend non necessari. In entrambi i casi, assicurati che il comportamento implementato (ImageService.applyFilter, FilterRegistry.getAvailableFilters, UC11_SelezionaFiltroTest) sia chiaramente tracciabile ai passi del caso d'uso.

### UC-01

**ISS-009** — LOW [100%] — Page 46 La sezione di testing afferma che i test funzionali coprono ogni caso d'uso identificato, e in effetti esistono classi UC01–UC11. Tuttavia, alcuni UC richiamati come include/extend (es. "Visualizza Dettaglio Post", "Visualizza Commenti", "Visualizza Anteprima", "Salva immagine modificata") non sono definiti come casi d'uso autonomi e non hanno test funzionali dedicati. Questo rende la tracciabilità requisiti–test incompleta: parti importanti del comportamento (dettaglio post, visualizzazione commenti, anteprima filtro) sono testate solo indirettamente o non sono chiaramente mappate a un UC.

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

**Recommendation:** Aggiungi una semplice matrice di tracciabilità che mappi ogni caso d'uso (inclusi quelli inclusi/estesi come "Visualizza Dettaglio Post", "Visualizza Commenti", "Visualizza Anteprima", "Salva immagine modificata") ai test che lo coprono. Se alcune funzionalità sono testate solo indirettamente, esplicitalo (es. "Visualizza Dettaglio Post" coperto da UC04_VisualizzaPostTest.displayPostDetails). In alternativa, se non intendi trattarli come UC separati, rimuovi la notazione include/extend per questi nomi e descrivi il comportamento direttamente nei flussi base dei casi d'uso principali, così la frase "per ogni caso d'uso identificato" rimane vera e non ambigua.

*See also: TST-001, TST-004*

## 6.3 Testing (5 issues)

### UC-01

**TST-001** — HIGH [100%] — Page 9 Error and alternative flows are only partially covered by tests. For many Use Cases, some alternative branches are tested (e.g., duplicate username, invalid credentials, empty comment text, non-existing post), but other specified error conditions (invalid registration data, database errors, image format not supported, path not valid, filter application errors, I/O errors on save, etc.) have no corresponding negative tests. This is a systemic gap across multiple UC.

> *Alternative Flow 3a) Il Visitatore inserisce dati non validi: il sistema mostra un errore 5a) Username o email gi'a esistenti: il sistema mostra errore "Username o Email gi'a esistente"*

**Recommendation:** Systematically review each Use Case's Alternative Flow section and ensure there is at least one test method per distinct error condition. For example: for UC01 add a test that attempts registration with missing mandatory fields and asserts that AuthService.register throws IllegalArgumentException; for UC02 add a test that simulates a SQLException from UserDao.login and verifies the controller prints "Errore del database durante il login"; for UC06 add tests for formato non supportato and path non valido; for UC07 add a test that forces an errore nell'applicazione del filtro; for UC08 add tests that simulate errore I/O and parametri non validi; for UC09/UC10 add tests that simulate database errors. Group these as additional @Test methods in the existing UCxx test classes so they remain aligned with the Use Cases.

*See also: ISS-009*

**TST-004** — MEDIUM [100%] — Page 66 The document claims complete coverage, but there is no explicit, systematic traceability matrix linking each detailed Use Case step (basic and alternative flows) to specific test methods. Traceability is implicit (via UCxx class names) but not documented, which makes it hard for a reviewer to verify that every requirement and flow is actually tested.

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

**Recommendation:** Add a concise traceability table that maps each Use Case step (including alternative flows) to one or more concrete test methods. For example, for UC01 list: step 3a -> UC01_RegistrazioneTest.testRegistrazioneOsservatoreCompleta (invalid data not covered, so mark as TODO), step 5a -> UC01_RegistrazioneTest.testRegistrazioneUsernameDuplicato, etc. Do this for UC01–UC11 so that a reader can immediately see which flows are covered and which are intentionally left out.

*See also: ISS-009*

### UC-03

**TST-002** — MEDIUM [100%] — Page 10 Controller-level behavior (messages and CLI interaction) is only partially tested. Many functional tests exercise Service and DAO layers directly, but do not verify that the controllers actually implement the Use Case flows and user messages as specified (e.g., specific error strings for invalid credentials, database errors, or other failures). Only UC03_LogoutTest explicitly tests AuthController output; other controllers (ImageController, PostController) are not covered at the presentation level.

> *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"*

**Recommendation:** For each main controller (AuthController, ImageController, PostController), add a small set of focused tests that simulate user input via Scanner and capture System.out/System.err to verify the exact messages and branching behavior described in the Use Case templates. For example, for UC02 create a test that calls AuthController.loginUser with a mocked AuthService throwing AuthenticationException and assert that the printed message contains "Login fallito: Credenziali non valide."; similarly, simulate a SQLException and assert the "Errore del database durante il login" message. For ImageController, test loadImage with invalid path and unsupported format to verify the CLI messages match the UC06 alternative flows. This will close the gap between backend logic tests and the specified user-facing behavior.

### UC-05

**TST-003** — MEDIUM [100%] — Page 13 Some important authorization and precondition checks are tested only at the Service layer, not at the full Use Case level. For example, UC05 tests that an OSSERVATORE cannot publish via PostServiceImpl, but there is no end-to-end test that goes through PostController.publishLoadedImageAsPost and verifies the exact messages for missing image, unauthenticated user, conversion error, or database error as described in the Use Case.

> *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: impossi- bile convertire l'immagine" 6a) Errore database: il sistema mostra "Errore del database durante la pubblicazione del post"*

**Recommendation:** Extend UC05_PubblicaPostTest (or add a dedicated controller-level test class) to cover the full UC05 flow via PostController.publishLoadedImageAsPost. Use a mocked PostService and ImageUtils (or inject a test double) to simulate: (1) loadedImg == null, (2) currentUser == null, (3) ImageUtils.bufferedImageToBytes returning null to trigger "Errore: impossibile convertire l'immagine", and (4) postService.createPost throwing SQLException. Capture System.out/System.err and assert that the messages match the Use Case text. This will ensure that both authorization and error handling are correctly wired from controller down to service/DAO.

### UC-07

**TST-005** — LOW [100%] — Page 65 Filter algorithms are well covered for positive behavior (correct grayscale conversion, sepia transformation, brightness/contrast changes), but there are no tests for robustness against invalid inputs at the Filter and ImageService level (e.g., null BufferedImage, extremely small images, or unexpected image types). Given that ImageUtils and ImageService contain explicit null and format checks, this is a minor but visible gap.

> *Classi testate: GrayScaleFilter, SepiaFilter, BlurFilter, etc.*

**Recommendation:** Add a few negative and edge-case tests for filters and ImageService.applyFilter. For example, in ImageFiltersTest add a test that passes null as inputImage to a Filter implementation and asserts that a clear exception is thrown or that the behavior is defined; in UC07_ModificaImmagineTest or a new structural test, pass a 1x1 image and an unusual BufferedImage type to ensure filters still work. Also consider a test where ImageService.applyFilter is called with a null Filter to verify that it either throws an IllegalArgumentException or is handled gracefully. These small additions will strengthen the robustness story of the image-processing part.

## 7 Priority Recommendations

The following actions are considered priority:

1. **ISS-003** (p. 17): Estendi il caso d'uso Metti Like per coprire i vincoli di business: (1) aggiungi tra le pre-condizioni "L'utente è autenticato"; (2) specifica se un u...
2. **TST-001** (p. 9): Systematically review each Use Case's Alternative Flow section and ensure there is at least one test method per distinct error condition.

## 8 Traceability Matrix

Of 19 traced use cases: 18 fully covered, 0 without design, 1 without test.

| ID | Use Case | Design | Test | Gap |
|---|---|---|---|---|
| UC01 | Registrazione | ✓ | ✓ | — |
| UC02 | Login | ✓ | ✓ | — |
| UC03 | Logout | ✓ | ✓ | — |
| UC04 | Visualizza Elenco Post | ✓ | ✓ | — |
| UC05 | Pubblica Post | ✓ | ✓ | — |
| UC06 | Carica Immagine | ✓ | ✓ | — |
| UC07 | Modifica Immagine | ✓ | ✓ | — |
| UC08 | Salva Immagine Locale | ✓ | ✓ | — |
| UC09 | Metti Like | ✓ | ✓ | — |
| UC10 | Aggiungi Commento | ✓ | ✓ | — |
| UC11 | Seleziona Filtro | ✓ | ✓ | — |
| UC-AUTH-VAL-EMAIL | Valida formato mail (include in Registrazione) | ✓ | ✓ | — |
| UC-AUTH-UNIQ-USERNAME-EMAIL | Controlla unicità username / email (include in Registrazione) | ✓ | ✓ | — |
| UC-POST-VISUALIZZA-COMMENTI | Visualizza Commenti (include in Visualizza Elenco Post / Visualizza Dettaglio Post) | ✓ | ✓ | — |
| UC-POST-VISUALIZZA-DETTAGLIO | Visualizza Dettaglio Post | ✓ | ✓ | — |
| UC-FILTER-VISUALIZZA-ANTEPRIMA | Visualizza Anteprima (include in Seleziona Filtro) | ✓ | ✓ | — |
| UC-FILTER-APPLICA-SATURAZIONE | Applica Saturazione | ✓ | × | Design for SaturationFilter exists but there is no dedicated structural or functional test that validates its specific b... |

| ID | Use Case | Design | Test | Gap |
|---|---|---|---|---|
| UC-FILTER-APPLICA-SFOCATURA | Applica Sfocatura | ✓ | ✓ | — |
| UC-FILTER-APPLICA-SEPPIA | Applica Seppia | ✓ | ✓ | — |

## 9 Terminological Consistency

Found **10** terminological inconsistencies (2 major, 8 minor).

| Group | Variants found | Severity | Suggestion |
|---|---|---|---|
| Actor: anonymous vs generic user | «Visitatore», «Visitatori», «utente», «Utente» | MAJOR | Use a single, consistent term for the anonymous/non-logged actor; prefer "Visitatore" everywhere in the requirements and use case templates, and avoid mixing it with the more generic "utente" when referring specifically to that role. |
| Actor: logged-in user | «utente», «Utente», «Osservatore», «Autore», «Osservatore/Autore» | MAJOR | Use one consistent term for the logged-in person; when referring to the generic logged-in user, prefer "Utente" and reserve "Osservatore" and "Autore" strictly as role names, not as generic synonyms. |
| Saving image locally | «Salva Immagine Locale», «Salve Immagine Locale», «Salva immagine modificata», «Salva immagine modificata nella cartella 'immagini_salvate_localmente'», «Salvataggio immagine in locale» | MINOR | Use a single, consistent label for the image-saving use case; prefer "Salva Immagine Locale" (already used as use case title) and align all other mentions to it. |
| Applying filters | «Applica Filtro», «Applica Filtri», «Applicare il filtro selezionato», «Applicazione del filtro» | MINOR | Use a single, consistent name for the filter application use case; prefer "Applica Filtro" and avoid mixing it with the plural "Applica Filtri" unless a different concept is intended. |

| Group | Variants found | Severity | Suggestion |
|---|---|---|---|
| Viewing list of posts | «Visualizza Elenco Post», «Visualizza Elenco post», «elenco dei post», «elenco di tutti i post», «Elenco dei Post», «Visualizza Elenco post» (activity diagram label) | MINOR | Use one consistent term for the post list; prefer "Visualizza Elenco Post" (already used in multiple diagrams and templates) and align all other variants to this exact wording and capitalization. |
| Viewing post details | «Visualizza Dettaglio Post», «visualizzarne i dettagli», «Visualizza i dettagli di un post», «sta visualizzando i dettagli di un post» | MINOR | Use a single, consistent term for the detailed post view; prefer "Visualizza Dettaglio Post" and avoid mixing it with "Visualizza i dettagli di un post" unless a different level of detail is meant. |
| Image loaded in memory | «immagine caricata in memoria», «immagine caricata», «immagine valida caricata in memoria», «immagine valida caricata», «Immagine caricata non successo» | MINOR | Use a single, consistent term for the image in memory; prefer "immagine caricata in memoria" and avoid alternating with shorter or slightly different phrases unless they denote a different state. |
| Filter registry component | «FilterRegistry», «Filter Registry», «Filter Registry applica il filtro», «Filter Registry gestisce la registrazione», «Filter Registry» (actor name in diagram) | MINOR | Use a single, consistent term for the filter registry component; prefer "FilterRegistry" (class name) and avoid mixing it with spaced or partially translated forms. |
| Persistence layer terminology | «DAO», «Data Access Object», «dao», «dao.impl», «ORM», «UseOrm», «PostOrm», «CommentOrm» | MINOR | Standardize the naming of the data access layer; prefer "DAO" and "Data Access Object" as already defined, and avoid mixing with "Orm"/"ORM" in diagrams unless a real ORM is introduced. |
| Command-line interface | «interfaccia utente testuale», «interfaccia utente testuale», «Presentation Layer (Interfaccia Utente CLI)», «sistema CLI», «interfaccia a riga di comando», «applicazione command-line (CLI)» | MINOR | Use a single, consistent term for the command-line interface; prefer "CLI" (already used) and avoid mixing it with longer Italian paraphrases when referring to the same architectural element. |