Software Engineering Audit Report

RelazioneSWE_Hu_Taiti.pdf

CAPRA

February 25, 2026

Contents

1 Document Context

Project Objective

The application is a software system for managing a network of university libraries. It allows users to access a unified catalog containing physical and digital items, and enables borrowing and reservation operations through authentication via university or external credentials. Administrators can manage users, catalog items, loans, and returns across multiple library locations.

Main Use Cases

- UC-1 – Login: User/Admin authentication via university or external credentials system.
- UC-2 – Ricerca catalogo: User searches the library catalog using keywords and advanced filters.
- UC-3 – Prenota articolo: Authenticated user reserves an available item from a specific library location.
- UC-4 – Aggiungi in lista di attesa: User is added to a waiting list for unavailable items.
- UC-5 – Visualizza resoconto prestiti: User views details of their active loans.
- UC-6 – Visualizza resoconto prenotazioni: User views details of their active reservations.
- UC-7 – Cancellazione di una prenotazione: User cancels an existing reservation.
- UC-8 – Login Admin: Administrator authentication via university system.
- UC-9 – Ricerca catalogo Admin: Administrator searches the catalog.
- UC-10 – Registrare utenti esterni: Admin registers external users and sends credentials via email.
- UC-11 – Aggiunta di un articolo: Admin adds new items (Book, Magazine, Thesis) to the catalog.
- UC-12 – Modifica di un articolo: Admin updates item information in the catalog.
- UC-13 – Cancellazione di un articolo: Admin removes items from the catalog.
- UC-14 – Registrare prestito: Admin registers a loan for a user.
- UC-15 – Conferma ritiro: Admin confirms item pickup for a reservation.
- UC-16 – Registrare restituzione articolo: Admin records item return and notifies waiting list users.
- UC-17 – Ricerca utente: Admin searches for user information by keywords.

Functional Requirements

- Support authentication via university system and external library credentials with password hashing.
- Implement catalog search with basic keywords and advanced filters (category, language, publication date, borrowability).
- Enable item reservation with automatic waiting list management and email notifications.
- Support three item types: `Book`, `Magazine`, and `Thesis` with type-specific attributes.
- Manage physical copies across multiple library locations with availability tracking.
- Implement loan registration with maturity dates and automatic email confirmations.
- Support item return registration with automatic waiting list notifications.
- Provide admin CRUD operations for items with constraint validation (no deletion if active loans/reservations exist).
- Implement user registration for external users with email verification and automatic credential generation.
- Support user search and management by administrators.
- Implement automatic email notifications for reservations, loans, returns, and waiting list updates.
- Track banned users with unbanning dates and prevent operations during penalty periods.

Non-Functional Requirements

- Database: `PostgreSQL` relational database with JDBC connectivity for data persistence.
- Architecture: Three-layer design with Domain Model, Business Logic, and ORM (Object-Relational Mapping).
- Transaction management: Support for atomic operations with commit/rollback on multi-table operations.
- Security: Password hashing with salt using SHA-256 algorithm; role-based access control via `Token` objects.
- Email notifications: Automated SMTP-based email system using Jakarta Mail library.
- Database triggers: Automatic updates to available copy counts on reservation/loan operations.
- Singleton pattern: `ConnectionManager` ensures single database connection instance.
- CLI interface: Command-line interface for user interaction with the system.

Architecture

The system follows a three-layer architecture: (1) **Domain Model** contains entity classes (`User`, `Admin`, `Hirer`, `Item`, `Book`, `Magazine`, `Thesis`, `Lending`, `Reservation`, `PhysicalCopies`); (2) **Business Logic** implements controllers (`AdminController`, `HirerController`, `ItemController`, `LendingController`, `ReservationController`) with operation validation and transaction management; (3) **ORM** provides data access objects (`BookDAO`, `MagazineDAO`, `ThesisDAO`, `HirerDAO`, `AdminDAO`, `LendingDAO`, `ReservationDAO`, `PhysicalCopiesDAO`, `WaitingListDAO`) using JDBC. The `ConnectionManager` singleton manages database connections, and `MailSender` handles email notifications. Database triggers maintain consistency of available copy counts.

Testing Strategy

Testing is organized into two main categories: (1) **Business Logic Tests** simulate user interactions by testing controller methods with mocked ORM operations; each test class includes setup methods to populate test data and cleanup methods to clear tables after execution; functional tests cover complete workflows (e.g., admin login, item addition, user registration); (2) **ORM Tests** validate data access operations directly on the database, testing CRUD operations for each DAO class with constraint violation and not-found exception scenarios. Tests verify correct behavior for both success and failure cases, including permission checks, state validations, and database constraint enforcement.

2 Executive Summary

Quick Overview

Total issues: **11** — HIGH: **2** MEDIUM: **6** LOW: **3**

Average confidence: **94%**

Executive Summary: Audit of RelazioneSWE_Hu_Taiti.pdf

This document is a Software Engineering report for a university project describing the design, requirements, and testing of a library management system. The report includes architectural diagrams, use case specifications, domain models, business logic implementation, and test coverage. The purpose is to document the system's functional and non-functional requirements, design decisions, and validation through unit and integration tests.

The audit identified **11 issues across three categories**. The most significant pattern is a **systemic mismatch between documented requirements and implemented behavior**. Use case templates describe high-level flows and preconditions, but the actual Business Logic controllers enforce additional constraints (role checks, working-place restrictions, item borrowability, copy availability) that are not reflected in the requirements documentation. Additionally, several use cases lack complete alternative flows for common error scenarios, and notification behavior is partially documented outside the use case

framework. A secondary pattern involves incomplete test coverage: advanced search functionality, error message validation, and state consistency checks are underspecified or missing.

Critical areas (HIGH severity) require immediate attention. ISS-003 reveals a fundamental inconsistency between the global login rule stated in use case diagrams and the actual permission enforcement via Token in the Business Logic, affecting all protected operations. ISS-004 documents incomplete and inconsistent requirements for reservation and lending workflows, with missing error flows and constraints that are implemented but not specified, creating risk of misunderstanding during maintenance or extension.

The document demonstrates **reasonable architectural design** with proper separation of concerns (DAO, Business Logic, Controllers) and adequate unit test coverage at the method level. However, the overall quality is **MEDIUM**: the gap between requirements and implementation is substantial, test coverage for complex scenarios is incomplete, and the documentation does not serve as a reliable single source of truth for system behavior.

1. **Reconcile use case preconditions and alternative flows with implemented Business Logic constraints.** Audit all use cases (UC-3, UC-4, UC-7, UC-11, UC-13, UC-14, UC-15, UC-16) to document role, working-place, and item-state checks that are enforced in code but missing from templates. Update preconditions and alternative flows to match the actual permission model.
2. **Complete requirements for critical operations (reservation, lending, item management).** Expand UC-3, UC-4, UC-7, UC-11, UC-13, UC-14, UC-15, UC-16 with all error conditions, side effects (e.g., waiting list updates, notifications), and branch-specific behavior. Ensure post-conditions are precise and distinguish between success variants (e.g., new title vs. copy addition).
3. **Extend test coverage for advanced search, error messages, and state consistency.** Add tests for ItemControllerTest.advanceSearchItem with filters, validate error messages against use case templates in all failure paths, and add state-consistency checks to ensure no partial modifications occur on error.

3 Strengths

- **Comprehensive Use Case Coverage:** The document defines 17 distinct use cases covering both Hirer and Admin actors, with detailed templates including brief descriptions, pre-conditions, basic flows, alternative flows, and post-conditions. This systematic approach ensures complete functional specification of the library management system.
- **Well-Structured Layered Architecture:** The project employs a clear three-layer architecture (Domain Model, Business Logic, and ORM) with proper separation of concerns. This design facilitates maintainability and allows independent testing of each layer, as evidenced by the test organization.
- **Extensive Test Coverage:** The document demonstrates thorough testing across multiple controller classes (AdminControllerTest, HirerControllerTest, ItemControllerTest, LendingControllerTest, ReservationControllerTest) and data access objects (BookDAOTest), indicating systematic validation of both business logic and persistence layers.
- **Complete Design Documentation:** The project includes comprehensive design artifacts including Use Case Diagrams, Navigation Diagrams, Class Diagrams (Domain Model, Business Logic, ORM), Entity-Relationship Diagrams, and UI mockups created with professional tools (Balsamiq Wireframes, Lucidchart), providing clear visual specifications.
- **Detailed User Interface Specifications:** Each use case is accompanied by corresponding mockups and templates showing the actual UI implementation, enabling clear communication of functional requirements and user interactions across different user roles.

4 Expected Feature Coverage

Of 7 expected features: 4 present, 3 partial, 0 absent. Average coverage: **85%**.

Feature	Status	Coverage	Evidence
Unit testing framework implementation	Partial	80% (4/5)	Assertion-based verification is used extensively (e.g., assertEquals, assertThrows, assertTrue in AdminControllerTest, HirerControllerTest, ItemControllerTest, LendingControllerTest, ReservationControllerTest, BookDAOTest). A systematic approach is documented: section 4 describes testing of Business Logic and ORM, setup/teardown with @BeforeEach and @AfterAll, and mapping tests back to templates. Dedicated test classes exist for specific functionalities (AdminControllerTest, HirerControllerTest, ItemControllerTest, LendingControllerTest, ReservationControllerTest, BookDAOTest). ORM tests and Business Logic tests together provide both unit-like and integration-like coverage (tests hit controllers plus DB, and pure DAO tests). However, there is no mention of using mock dependencies; tests interact with a real database and DAOs, so isolated testing with mocks is not demonstrated.

Feature	Status	Coverage	Evidence
Use of UML Diagrams for system modeling	Present	100% (8/8)	Use case diagrams are provided for Utente and Admin in section 2.1, showing interactions like login, ricerca catalogo, CRUD articoli, registrare prestito. Class diagrams are documented for Domain Model, Business Logic, and ORM in section 2.4, representing classes and relationships. A structure/architecture diagram is given in the introduction (flowchart with Utente, Interfaccia (CLI), Business Logic, ORM, Domain Model, MailSender, RBMS, JDBC, PostgreSQL). Numerous UI mockups are included in section 2.2 (homepage, login, ricerca catalogo, pagina articolo, resoconto prestiti/prenotazioni, registrazione utente esterno, aggiunta articolo, pagina utente). The diagrams use standard UML-like notation (class boxes with attributes/operations, use case ovals with actors and «extend»). Relationships between actors and use cases are explicitly shown in the use case diagrams. Navigation flows are visualized in the Navigation Diagram (Home Page Hirer, HomePage Admin, Pagina Articolo Hirer/Admin, Resoconto Prenotazioni/Prestiti, Ricerca Utente, etc.). Functional requirements are clarified via these diagrams and templates, tying use cases and UI to system behavior.

Feature	Status	Coverage	Evidence
Identification and definition of system actors	Present	100% (8/8)	User roles are clearly identified as Utente/Hirer and Admin (statement, use case diagrams, templates, Domain Model classes User, Hirer, Admin). Responsibilities are documented: admins handle registrare utenti esterni, CRUD articoli, registrare prestito, conferma ritiro, registrare restituzione, ricerca utente; hirers/utenti perform login, ricerca catalogo, prenota articolo, aggiungi in lista di attesa, visualizza resoconto prestiti/prenotazioni, cancella prenotazione. Interactions between users and system components are defined via use case templates and controller descriptions (e.g., HirerController.loginUniversityHirer, AdminController.loginAdmin, ReservationController.reserveItem). Specific user actions are listed in each use case basic flow. The templates and use case diagrams show a structured approach to user requirements, with separate sections for Templates relativi ad Hirer and Templates relativi ad Admin. Functional requirements per role are documented in the use case descriptions and controller methods (e.g., only Admin can registerExternalHirer, addBook, registerLending). System functionalities are delineated to improve usability (resoconto prestiti/prenotazioni, ricerca utente, pagina utente). User actions are explicitly identified to guide development (click "Prenota", "Aggiungimi in lista", "Resoconto Prestiti", "Registrazione utente esterno", etc.).

Feature	Status	Coverage	Evidence
Definition and Documentation of Use Cases	Present	100% (5/5)	Specific user interactions are defined in detailed templates for use cases #1–#7 and #10–#17 (e.g., Login, Ricerca catalogo, Prenota articolo, Registrare prestito, Registrare restituzione articolo). Each template includes a Brief Description that states the user goal (e.g., "L'utente seleziona l'articolo di interesse e lo prenota", "Un admin registra il prestito di un utente"). Detailed basic flows and, where applicable, alternative flows are provided (e.g., wrong credentials in Login, blocked user in Prenota articolo, article already present in Aggiunta di un articolo, error conditions in Cancellazione di un articolo). Preconditions and post-conditions are explicitly specified for each use case. Relationships between use cases are illustrated in the use case diagrams (e.g., «extend» from ricerca catalogo to prenota articolo, and from visualizza resoconto prenotazioni to annulla prenotazione).

Feature	Status	Coverage	Evidence
User interface and interaction design principles	Partial	71% (5/7)	UI mockups are provided for key interfaces: homepage, login screens, ricerca catalogo with filters, pagina articolo, resoconto prestiti, resoconto prenotazioni, registrazione utente esterno, aggiunta articolo, pagina utente (section 2.2). Navigation elements like Home/Homepage links, breadcrumbs (Home › Registrazione Utente Esterno, Home › Ricerca Articolo › Articolo XXX), and menu options (Home admin with Registrazione Utente Esterno, Ricerca Utente, Ricerca Articolo, Aggiungi Articolo, Registra Prestito) demonstrate clear navigation. The reservation creation process is shown step-by-step via the Ricerca catalogo and Prenota articolo templates and the pagina articolo mockup with "Prenota" buttons per sede. Validation mechanisms are documented, e.g., email verification in Registrare utenti esterni (button "Verifica" and codice verifica) and error alerts for wrong login credentials. Structured input fields for user data submission are visible in forms (registrazione utente esterno fields, login fields, aggiunta articolo fields). Distinct user roles and their functionalities are reflected in separate homepages and options for Hirer vs Admin. However, dynamic updates of user selections in the interface (e.g., live UI changes based on selections) are not explicitly described beyond static mockups and flows.

Feature	Status	Coverage	Evidence
Separation of concerns in software architecture	Partial	60% (3/5)	The project is explicitly divided into three main packages: Domain Model, Business Logic, and ORM, plus CLI and MailSender (section 1.2), which partially aligns with Model/Controller/DAO separation but not a full MVC with a dedicated View or Service layer. Interactions between controllers and domain models are documented in the Business Logic section and class diagrams (e.g., controllers operate on Item, Hirer, Admin, Reservation, Lending objects). Clear relationships between layers such as Controller and DAO are described: controllers call DAOs (e.g., AdminController uses AdminDAO, HirerController uses HirerDAO and WaitingListDAO, ItemController uses BookDAO and PhysicalCopiesDAO, LendingController uses LendingDAO, ReservationController uses ReservationDAO), and ConnectionManager centralizes DB access. The role of each component is described: Domain Model represents entities, Business Logic implements system functionality, ORM handles persistence, CLI is the user interface, MailSender handles notifications. However, there is no distinct Service package separate from controllers, and no explicit View package (only CLI and mockups), so the checklist item about distinct packages for Model, View, Controller, and Service is only partially met.

Feature	Status	Coverage	Evidence
Data Access Object (DAO) pattern	Present	86% (6/7)	DAO classes are clearly defined for different entities (AdminDAO, HirerDAO, BookDAO, MagazineDAO, ThesisDAO, PhysicalCopiesDAO, LendingDAO, ReservationDAO, WaitingListDAO in section 3.3), satisfying separate DAOs per entity. CRUD operations are implemented in DAO classes, e.g., BookDAO has getBook, addBook, removeBook, updateBook, getAllBooks; similar patterns are described for other DAOs. SQL queries are encapsulated within DAO methods (PreparedStatement queries in BookDAO methods). Database access is abstracted through DAOs and a ConnectionManager singleton that provides a shared Connection. Patterns for managing data access are documented, including transaction handling via ConnectionManager (closeAutoCommit, commit, rollback) in Business Logic methods that coordinate multiple DAO calls. Testing strategies for data access methods are provided in section 4.3.1 (BookDAOTest with tests for getBook, addBook, removeBook, updateBook, getAllBooks). There is no explicit DAO interface type (e.g., Java interface) described; DAOs are concrete classes, so a "clear interface" in the sense of a formal interface type is not shown.

5 Summary Table

Category	HIGH	MEDIUM	LOW	Total
Architecture	0	0	2	2
Requirements	2	5	0	7
Testing	0	1	1	2
Total	**2**	**6**	**3**	**11**

6 Issue Details

6.1 Architecture (2 issues)

ISS-001 — LOW [100%] — Page 2 In the architectural flowchart, the Domain Model is shown as connected directly to RBMS (RDBMS), which contradicts the textual description and the ORM design where persistence is handled exclusively by DAO classes via JDBC. This could mislead readers into thinking that domain objects access the database directly, bypassing the ORM layer, which is not the case in the implementation.

> *L'architettura del software prevede che l'utente, tramite la CLI, richiami i metodi della Business Logic ese- guiti sugli oggetti del Domain Model creati dall'ORM partendo dai dati all'interno di un RDBMS utilizzando JDBC. Inoltre 'e presente anche un MailSender che permette di inviare notifiche agli utenti. [IMAGE 1]: Diagram Type: Flowchart Elements: - Utente - Interfaccia (CLI) - Business Logic - ORM - Domain Model - MailSender - RBMS - JDBC - PostgreSQL Relationships: - "Utente" interacts with "Interfaccia (CLI)" - "Interfaccia (CLI)" connects to "Business Logic" - "Business Logic" links to "ORM" and "Domain Model" - "ORM" connects to "JDBC" - "Domain Model" connects to "MailSender" and "RBMS"*

Recommendation: Adjust the architecture diagram to be consistent with the described and implemented layering: 1) Remove the direct connection between Domain Model and RBMS/PostgreSQL; instead, show that only ORM (DAO layer) connects to JDBC/RDBMS. 2) If Domain Model interacts with MailSender (e.g., via controllers), represent that through the Business Logic layer rather than a direct link from Domain Model, unless domain objects really call MailSender directly. 3) Optionally, add a short caption under the diagram clarifying that all persistence operations go through ORM/DAO and that Domain Model objects are in-memory representations only. 4) Ensure that this corrected diagram matches the actual code structure (controllers -> DAOs -> ConnectionManager) to avoid confusion during evaluation.

ISS-002 — LOW [100%] — Page 17 The Domain Model is designed with overridden equals and hashCode to support testing, and the tests rely heavily on equality comparisons between domain objects and DAO/controller results. However, the document does not show any dedicated tests that validate the correctness of these equals/hashCode implementations themselves. If equals/hashCode are incorrect, many existing tests could pass or fail for the wrong reasons, undermining confidence in both ORM and Business Logic tests.

> *In tutte le classi sono stati ridefiniti i metodi equals e hashCode per semplificare la fase di test, mentre nelle classi derivate da Item 'e stato introdotto anche un metodo toStringValues, utile a fornire una rappresentazione leggibile dell'oggetto.*

Recommendation: Add a small set of focused unit tests for equals and hashCode in the Domain Model (e.g., for Book, Magazine, Thesis, Hirer, Admin, Reservation, Lending). For each class, create tests that: (1) compare two instances with identical field values and assert equality and identical hashCode; (2) compare instances differing in one key field and assert inequality; (3) optionally verify behavior in collections (e.g., using HashSet). This will validate the assumptions on which many higher-level tests are built, without requiring large additional effort.

6.2 Requirements (7 issues)

UC-2

ISS-003 — HIGH [100%] — Page 3 Systemic inconsistency between the global login rule in the use case diagrams and several textual templates/flows. The diagram states that login is required for all actions except catalog consultation, but UC-2 (Ricerca catalogo) allows search

without login, while UC-3, UC-4, UC-5, UC-6, UC-7, UC-10, UC-11, UC-12, UC-13, UC-14, UC-15, UC-16, UC-17 each define their own preconditions and sometimes implicitly assume login or specific roles. In addition, the Business Logic methods (e.g., reserveItem, registerLending, removeReservation, confirmReservationWithdraw) enforce permissions via Token, which is not reflected in the use case preconditions. This creates a mismatch between what the requirements say users can do and what the implemented logic actually allows.

> *login necessario per tutte le azioni tranne che per la consultazione del catalogo*

Recommendation: Unify and make explicit the authentication/authorization rules across all use cases and align them with the implemented Token-based checks. Concretely: 1) Add a short global subsection in the requirements (before the templates) that states clearly: which actions require login, which roles (Hirer/Admin) can perform each action, and how the Token concept maps to roles. 2) For each use case that requires authentication (UC-1/8, 3, 4, 5, 6, 7, 10–17), update the Pre-Conditions to explicitly mention that the user is logged in and has the correct role (e.g., "L'utente è autenticato come Hirer" or "L'utente è autenticato come Admin"). 3) For UC-2 (Ricerca catalogo), explicitly state whether login is optional or forbidden, and ensure this matches the diagram note. 4) Where Business Logic enforces role or working-place constraints (e.g., token.getTokenRole(), token.getTokenWorkingPlace()), add these as preconditions or decision points in the corresponding use cases (e.g., UC-14, UC-15, UC-16, UC-17).

ISS-005 — MEDIUM [88%] — Page 6 Several use cases that retrieve or display information (UC-2 Ricerca catalogo, UC-5 Visualizza resoconto prestiti, UC-6 Visualizza resoconto prenotazioni, UC-17 Ricerca utente) lack alternative flows for common negative or boundary scenarios, such as: no results found, invalid search parameters, or errors in accessing data. They also have very weak post-conditions that do not distinguish between "results found" and "no results". This makes the behavior in these frequent cases underspecified and can lead to misunderstandings when implementing or testing the UI and business logic.

> *Use Case #5 Visualizza resoconto prestiti ... Basic Flow 1. L'utente clicca su "Resoconto Prestiti". Post-Conditions L'utente visualizza i dettagli dei suoi prestiti.*

Recommendation: For all read-only/display use cases (UC-2, UC-5, UC-6, UC-17): 1) Add at least one Alternative Flow describing the case where the search or retrieval returns zero results (e.g., "Nessun articolo trovato" or "Nessun utente trovato"), including what the UI shows (empty table, message, etc.). 2) If there are constraints on search parameters (e.g., invalid date range in advanced search, invalid category/language), add an Alternative Flow describing the validation error and the system response. 3) Refine the Post-Conditions to distinguish between: (a) results displayed, (b) no results but operation completed correctly, and (c) operation failed due to an error. 4) Optionally, align these flows with existing tests (e.g., searchItem tests) by ensuring that each tested scenario corresponds to a documented basic or alternative flow.

UC-3

ISS-004 — HIGH [98%] — Page 5 The requirements for reservation and lending flows are incomplete and partially inconsistent with the implemented business rules. UC-3 only mentions the case of a blocked user, but the ReservationController.reserveItem implementation also checks: role (must be Hirer), borrowable flag, and number of available copies > 1. Similarly, UC-14 (Registrare prestito) and UC-15 (Conferma ritiro) only partially describe error conditions, while LendingController.registerLending and ReservationController.confirmReservationWithdraw enforce additional constraints (role must be Admin, hirer not banned, item borrowable, enough copies, correct working place). These missing alternative/error flows mean the documented behavior does not match the real business logic for critical operations (reservation, lending, pickup).

> *Use Case #3 Prenota articolo ... Basic Flow 1. L'utente clicca su "prenota" accanto alla sede di suo gradimento che ha l'articolo disponibile; 2. La prenotazione viene registrata nel database. ... Alternative Flow 2bis. Il software mostra un messaggio di errore in quanto l'utente 'e bloc- cato.*

Recommendation: For all reservation and lending related use cases (UC-3 Prenota articolo, UC-4 Aggiungi in lista di attesa, UC-7 Cancellazione di una prenotazione, UC-14 Registrare prestito, UC-15 Conferma ritiro, UC-16 Registrare restituzione articolo): 1) Derive all business rules from the corresponding controller methods (ReservationController.reserveItem/removeReservation/confirmReservationWithdraw, LendingController.registerLending/registerReturnOfItem, HirerController.addToWaitingList) and list them explicitly as either Pre-Conditions or Alternative Flows. 2) For each rule, add a concrete alternative flow step describing: the condition (e.g., "utente bannato", "articolo non prestabile", "copie insufficienti", "operazione eseguita da ruolo errato", "sede diversa"), the system reaction (error message, no DB change), and any side effects (no email, no waiting list update, etc.). 3) Ensure that the Basic Flow only covers the fully successful path, and that every branch in the code that throws an exception or denies an action is represented by an Alternative Flow in the corresponding use case. 4) Re-check the post-conditions to ensure they only hold when the Basic Flow completes successfully and not in error cases.

UC-7

ISS-006 — MEDIUM [95%] — Page 7 UC-7 describes cancellation of a reservation by the user, but the implemented ReservationController.removeReservation method allows both Admins and Hirers (depending on token.getTokenWorkingPlace) and always notifies and clears the waiting list. The use case does not mention any role/working-place constraints or the possibility that the cancellation is performed by an Admin from their interface. Moreover, the post-condition ignores the side effect on the waiting list. This creates a mismatch between the documented actor and effects and the actual business logic.

> *Use Case #7 Cancellazione di una prenotazione ... Basic Flow 1. L'utente seleziona una delle prenotazioni; 2. L'utente clicca su "Annulla prenotazione". 3. Gli utenti che si trovano nella lista di attesa di tale articolo vengono notificati (e rimossi dalla lista). ... Post-Conditions La prenotazione selezionata dall'utente 'e stata cancellata.*

Recommendation: Clarify and align UC-7 with ReservationController.removeReservation: 1) Decide whether cancellation can be done only by the Hirer, only by the Admin, or by both. If both, split into two use cases (e.g., "UC-7 Cancellazione prenotazione da parte dell'utente" and an Admin-specific UC) or clearly document both actors and their different preconditions. 2) Add to the Pre-Conditions the constraints enforced in code: for Admin, token.getTokenWorkingPlace must match the reservation's storagePlace; for Hirer, tokenWorkingPlace may be "NONE". 3) Extend the Post-Conditions to include the effect on the waiting list: that all users in the waiting list for that item and storagePlace are notified and removed. 4) Add an Alternative Flow for the case where the working place does not match (as in the testRemoveReservation_Fail1 test), describing the error and the fact that the reservation is not removed.

UC-11

ISS-007 — MEDIUM [92%] — Page 8 UC-11 (Aggiunta di un articolo) describes a generic article addition with a single alternative flow when the article already exists, but the implementation in ItemController.addBook is more specific: it checks for an existing Book with same fields, then either adds physical copies in the admin's working place or inserts a new Item+Book+PhysicalCopies. The use case does not mention that copies are added only in the admin's library, nor does it distinguish between adding a completely new title and adding copies

in another branch. The post-condition "Nel catalogo è presente il nuovo articolo aggiunto" is ambiguous in the case where only copies are increased.

> *Use Case #11 Aggiunta di un articolo ... Alternative Flow 3bis. Se l'articolo 'e gi'a presente nel catalogo viene aggiornato solo il numero di copie. Post-Conditions Nel catalogo 'e presente il nuovo articolo aggiunto.*

Recommendation: Refine UC-11 to reflect the actual behavior: 1) In the Basic Flow, explicitly state that the admin also specifies the library (implicitly their workingPlace) and the number of copies and borrowable flag. 2) Split the behavior into two clearly described outcomes: - Case A (new title): Item and specific subtype (Book/Magazine/Thesis) are created, and initial PhysicalCopies are inserted for the admin's workingPlace. - Case B (existing title): only PhysicalCopies for the admin's workingPlace are created or updated. 3) Update the Alternative Flow 3bis to mention that only the number of copies in the admin's library is updated, not globally. 4) Adjust the Post-Conditions to cover both cases, e.g., "Nel catalogo è presente l'articolo con il numero di copie aggiornato nella sede dell'Admin".

UC-13

ISS-008 — MEDIUM [95%] — Page 10 UC-13 mixes two different behaviors (full deletion of the article vs. deletion of copies in one branch) but the post-condition always says that the article is no longer present in the catalog. The implementation in ItemController.removeBook is more nuanced: if there are no copies in any library and no digital link, the Book and Item are deleted; otherwise, only PhysicalCopies in the admin's workingPlace are removed, and deletion is blocked if there are active reservations/lendings. The current post-condition is therefore incorrect in the partial-deletion case and does not mention the branch-specific behavior.

> *Use Case #13 Cancellazione di un articolo ... Alternative Flow 4bis. L'admin riceve un messaggio di errore in quanto ci sono prestiti o prenotazioni attive legate all'articolo. 4ter. Se l'articolo ha una versione digitale o 'e presente in altre sedi, l'articolo non viene cancellato completamente, ma solamente le copie fisiche nella sede in cui l'admin lavora. Post-Conditions Nel catalogo non 'e pi'u presente l'articolo appena cancellato.*

Recommendation: Correct and detail UC-13 to match ItemController.removeBook: 1) Separate the two outcomes in the flow: - Outcome 1: no copies in any library and no digital version -> the article (Item+Book/Magazine/Thesis) is removed from the catalog. - Outcome 2: the article exists in other branches or has a digital version -> only the PhysicalCopies in the admin's workingPlace are removed. 2) Update the Post-Conditions to be conditional, e.g., "Se non esistono altre copie fisiche né versione digitale, l'articolo non è più presente nel catalogo; altrimenti, l'articolo rimane nel catalogo ma non è più disponibile nella sede dell'Admin". 3) Make explicit in the Alternative Flow that the error about active loans/reservations prevents any deletion and leaves both the article and copies unchanged. 4) Optionally, add a small note linking this behavior to the database constraint/ConstraintViolationException logic already implemented and tested.

UC-4

ISS-009 — MEDIUM [100%] — Page 30 The MailSender section lists several notification scenarios (e.g., scadenza prenotazione, rinnovo noleggio, blocco/sblocco account, inserimento in lista d'attesa) that are not covered by any use case or template. Conversely, some use cases that involve notifications (e.g., UC-4 Aggiungi in lista di attesa, UC-7 Cancellazione prenotazione, UC-14, UC-15, UC-16) do not explicitly mention all the emails that are actually sent in the Business Logic. This creates a gap between non-functional/auxiliary behavior (notifications) and the functional requirements, and may confuse readers about when and why emails are sent.

Il sistema MailSender invia notifiche nei seguenti casi: • Conferma di prenotazione o prestito avvenuto con successo; • Invio codice di verifica per mail (Figure 1); • Scadenza di una prenotazione; • Rinnovo di un noleggio; • Blocco o sblocco dell'account utente; • Invio credenziali ad utente esterno; • Restituzione dell'articolo alla sede; • Inserimento dell'utente in lista d'attesa; • Notifica agli utenti in lista d'attesa quando un articolo diventa disponibile.

Recommendation: Align notification behavior with the use cases: 1) For each bullet in the MailSender list, either: - Add or extend a use case that describes the triggering operation and includes the email as a step in the Basic or Alternative Flow (e.g., "Scadenza di una prenotazione" could be part of a scheduled job use case, or removed if not implemented), or - Explicitly mark it as "future work" or "non implementato" if it is only a design idea. 2) In existing use cases that already involve notifications (UC-4, UC-7, UC-10, UC-14, UC-15, UC-16), add explicit steps in the Basic Flow or Post-Conditions describing the email being sent (e.g., "Il sistema invia una mail di conferma prenotazione"), matching the actual MailSender calls in the code. 3) Remove or adjust any notification scenario from this list that is not actually implemented in the Business Logic, or clearly state that it is outside the current scope of the project to avoid over-claiming functionality.

6.3 Testing (2 issues)

TST-001 — MEDIUM [76%] — Page 4 The template for Use Case #2 (Ricerca catalogo) explicitly distinguishes between basic and advanced search in the Alternative Flow, but ItemControllerTest only tests a simple search path and does not exercise the advanced search logic (advanceSearchItem) or the use of filters (category, language, borrowable, date range). This creates a gap between the specified behavior in the template and the tested behavior in the Business Logic.

Use Case #2 Ricerca catalogo ... Test Ricerca Articolo

Recommendation: Add dedicated tests in ItemControllerTest for the advanced search behavior described in Use Case #2. For example, create test methods that: (1) populate the DB with multiple Items differing in category, language, borrowable flag, and publicationDate; (2) call advanceSearchItem with specific filters; (3) assert that only the Items matching all filters are returned. Clearly reference Use Case #2 and the "Alternative Flow" in the test method names or comments to keep traceability with the template.

TST-002 — LOW [95%] — Page 11 For several use cases, the templates describe alternative flows in user-oriented terms (e.g., "viene segnalato un errore"), while the tests assert only that a specific exception type is thrown. There is no check that the error messages are meaningful and consistent with the templates, and no test ensures that the system does not partially modify state when an error occurs (e.g., no partial lending or reservation is recorded). This is a systemic pattern across failure tests in LendingControllerTest, ReservationControllerTest, ItemControllerTest, and HirerControllerTest.

Use Case #14 Registrare prestito ... Alternative Flow 2bis. Se l'utente 'e in un periodo di penalit'a o l'articolo non 'e noleggiabile oppure 'e disponibile una sola copia, viene segnalato un errore e non viene registrato il prestito.

Recommendation: Strengthen the failure-path tests by: (1) in existing tests that expect exceptions (e.g., registerLending_Fail1–4, testReserveItem_Fail1–4, testRemoveBook_Fail1–3, testAddToWaitingList_Fail_1/2), also asserting on the exception message (using getMessage()) to ensure it matches the intended user feedback; (2) after catching the exception, assert that the database state has not changed (e.g., no new lending/reservation rows, no changes to physical_copies) to confirm that the alternative flow "non viene registrato il prestito" or similar is respected. This will better align tests with the narrative in the templates.

7 Priority Recommendations

The following actions are considered priority:

1. **ISS-003** (p. 3): Unify and make explicit the authentication/authorization rules across all use cases and align them with the implemented Token-based checks.

2. **ISS-004** (p. 5): For all reservation and lending related use cases (UC-3 Prenota articolo, UC-4 Aggiungi in lista di attesa, UC-7 Cancellazione di una prenotazione, UC...

8 Traceability Matrix

Of 16 traced use cases: 16 fully covered, 0 without design, 0 without test.

ID	Use Case	Design	Test	Gap
#1, #8	Login	✓	✓	—
#2	Ricerca catalogo	✓	✓	—
#3	Prenota articolo	✓	✓	—
#4	Aggiungi in lista di attesa	✓	✓	—
#5	Visualizza resoconto prestiti	✓	✓	—
#6	Visualizza resoconto prenotazioni	✓	✓	—
#7	Cancellazione di una prenotazione	✓	✓	—
#9	ricerca normale, ricerca avanzata catalogo (Admin)	✓	✓	—
#10	Registrare utenti esterni	✓	✓	—
#11	Aggiunta di un articolo	✓	✓	—
#12	Modifica di un articolo	✓	✓	—
#13	Cancellazione di un articolo	✓	✓	—
#14	Registrare prestito	✓	✓	—
#15	Conferma ritiro	✓	✓	—
#16	Registrare restituzione articolo	✓	✓	—
#17	Ricerca utente	✓	✓	—

9 Terminological Consistency

Found **10** terminological inconsistencies (6 major, 4 minor).

Group	Variants found	Severity	Suggestion
Library user role	Hirer / Utente / utente / Hirers / hirers	MAJOR	Use a single, consistent term for the library user role. Given the Domain Model and controllers, prefer "Hirer" everywhere (actors, use cases, templates, tests, UI labels) when referring to the same role, and reserve "Utente" only for generic, non-role mentions if really needed.
Library administrator role	Admin / admin / amministratori / amministratore / Utente Admin / utente admin	MAJOR	Use one consistent term for the library staff role. Prefer "Admin" everywhere (actors, templates, UI, tests, code comments) instead of mixing with "amministratore" or generic "utente" when the role is administrative.
Library item / article	articolo / articoli / Item / articolo bibliotecario	MAJOR	Use a single, consistent term for the library item. Prefer "articolo" in Italian prose and UI, and "Item" only as the technical class name; avoid mixing "articolo" and "Item" in user-facing descriptions for the same concept.
Waiting list	lista di attesa / lista d'attesa / WaitingList / waiting_list	MAJOR	Use one consistent term for the waiting list concept. Prefer "lista di attesa" (or a single chosen spelling) in all Italian text and UI, and align with the technical name "WaitingList" / "waiting_list" only in code/DB contexts.
Authentication system	sistema di autenticazione universitario / sistema di autenticazione bibliotecario / Sistema Autenticazione Universitario / Sistema Autenticazione bibliotecario / UniversityAuthenticationSystem	MAJOR	Use a single, consistent term for the library system authentication. Prefer either the Italian names ("sistema di autenticazione universitario", "sistema di autenticazione bibliotecario") or the English class name "UniversityAuthenticationSystem" only in technical sections, and avoid mixing them without clarification.
Database system acronym	RDBMS / RBMS	MAJOR	Use a single, correct acronym for the relational database management system. Prefer "RDBMS" consistently in text and diagrams instead of mixing with the misspelled "RBMS".

Group	Variants found	Severity	Suggestion
Command-line interface	CLI / Interfaccia (CLI) / interfaccia a linea di comando	MINOR	Standardize the naming of the command-line interface. Prefer a single form such as "CLI" or "interfaccia a linea di comando (CLI)" and avoid alternating with "Interfaccia (CLI)" or other mixed forms.
Home page naming	Homepage / Home Page / HomePage Admin / Home Page Hirer / Home admin / home page	MINOR	Use a single, consistent term for the home page concepts. Prefer one spelling such as "Homepage" (or "Home Page") and apply it uniformly (including "HomePage Admin" vs "Home Page Hirer").
Library location / branch	sede / sede bibliotecaria / storagePlace / storage place	MINOR	Use a single, consistent term for the library location concept. Prefer one of "sede" or "storagePlace" depending on context, and clarify the mapping between the Italian term and the technical attribute to avoid confusion.
Verification code naming	codice verifica / codice di verifica / codice / codice di verifica da fornire	MINOR	Standardize the naming of the email verification code. Prefer one term such as "codice di verifica" and use it consistently in UI, templates, and MailSender descriptions.