Software Engineering Audit Report

Ingegneria_Software Francesko Blushi.pdf

CAPRA

February 25, 2026

Contents

1 Document Context

Project Objective

The project develops a **Library Loan Management System** that enables users to register, borrow, and return books, while administrators manage the book catalog and registered users. The system implements a client-server architecture with a relational database managed through JPA, automating loan and return operations with differentiation between standard users and administrators.

Main Use Cases

- UC-1 – Accedi al Sistema (Log-in): Users and admins authenticate with credentials to access the system.
- UC-2 – Aggiungi/Elimina Libro: Admins add new books to the catalog or remove existing books when all copies are available.
- UC-3 – Libri in Prestito: Users view the history of borrowed books and check loan expiration dates.
- UC-4 – Prendi/Restituisci Libro: Users borrow books for specified periods (one, two, or three months) or return previously borrowed books.
- UC-5 – Gestione Utenti: Admins search for and delete registered users by card ID.
- UC-6 – Registrazione: New users and admins register by providing personal data; admins require a library identification code.
- UC-7 – Recupera Password: Registered users change their password by verifying personal information.

Functional Requirements

- User registration with unique username and password meeting security criteria (minimum length, uppercase letters, numbers, special characters).
- Admin registration requiring library identification code verification.
- User authentication with credential validation for both standard users and administrators.
- Book catalog management: add books with ISBN, title, author, genre, language, edition, and availability count.
- Book removal only when all copies are available in the library.
- Loan management: users borrow books with three loan period options (one, two, three months).
- Return management: users return books and system updates availability and return date.
- User management: admins view and delete registered users by card ID.
- Loan history: users view all borrowed books with loan date, due date, and return date.
- Password recovery: users change password after verifying personal information (card ID, name, surname, username).
- Book search by title and author.
- Data validation for all user inputs before database operations.

Non-Functional Requirements

- **Security**: Passwords must meet minimum security criteria; authentication differentiates user roles; separate test database prevents data contamination.
- **Performance**: Efficient database queries using JPA/Hibernate; singleton pattern for database connection management.
- **Maintainability**: Separation of concerns through DAO pattern; validation logic isolated in `ValidationUtils` package; DTOs separate data transfer from entity exposure.

- **Scalability**: Generic DAO interface supports multiple entity types; modular controller architecture for independent functionality management.
- **Reliability**: Transaction management with rollback on errors; cascade operations for data consistency; proper resource cleanup in finally blocks.
- **Usability**: Intuitive UI mockups designed in Figma; clear error messages returned to users; role-based access control.

Architecture

The system follows a **three-tier architecture** with clear separation of concerns:

- **Models (Entities)**: JPA-annotated classes (`Utente`, `Admin`, `Users`, `Autore`, `Books`, `Loans`) mapped to database tables with relationships defined via `@OneToOne`, `@OneToMany`, and `@ManyToOne` annotations.
- **DAO Layer**: Generic `Dao<T>` interface with concrete implementations (`UtenteDao`, `AdminDao`, `UserDao`, `AutoreDao`, `BookDao`, `LoansDao`) handling CRUD operations and custom queries using Hibernate EntityManager.
- **Controller Layer**: Five controllers (`AccessController`, `AdminController`, `PasswordController`, `RegisterController`, `UtenteController`) orchestrating business logic and delegating to DAOs.
- **Validation Layer**: Static utility classes (`ValidationUtilsAdmin`, `ValidationUtilsPassword`, `ValidationUtilsUtente`, `ValidatioUtilsAccess`) providing input validation before database operations.
- **DTO Layer**: `ResponseDTO` for structured responses; `LoanBook` and `LibroAutore` for specific data transfer scenarios.
- **Database**: `PostgreSQL` with two databases (`LibraryDB` for production, `TestDB` for testing); singleton `ConnectionDB` manages `EntityManagerFactory`.
- **Design Patterns**: DAO pattern for data access abstraction; Singleton for database connection; Proxy pattern for role-based operations.

Testing Strategy

- **Functional Tests**: Five controller test classes verify end-to-end workflows: `AccessControllerTest` (authentication success/failure), `AdminControllerTest` (book add/remove), `PasswordControllerTest` (password reset), `RegisterControllerTest` (user/admin registration), `UtenteControllerTest` (loan/return operations).
- **Unit Tests**: DAO layer tests verify CRUD operations (`insert`, `delete`, `update`) and custom queries (`findById`, `setDueDate`, `libriloan`) using JUnit assertions.
- **Domain Model Tests**: Constructor and getter/setter methods validated for all entity classes (`Admin`, `Users`, `Books`, `Loans`, `Utente`, `Autore`).
- **Test Database**: Separate `TestDB` database prevents contamination of production data; `isManaged` flag in DAOs switches between production and test connections.
- **Validation Testing**: Input validation tested through controller tests with empty fields, invalid credentials, and missing required data.

2 Executive Summary

Quick Overview

Total issues: **14** — HIGH: **3** MEDIUM: **9** LOW: **2**

Average confidence: **95%**

Executive Summary: Software Engineering Document Audit

This document is a Software Engineering specification for a university library management system, comprising use case descriptions, architectural design, implementation code, and test cases. The purpose is to define functional requirements, design decisions, and validation of a system that manages book inventory, user registration, loans, and account administration. The document spans approximately 33 pages and includes UML diagrams, detailed use case narratives, Java implementation excerpts, and unit test specifications.

The audit identified 14 issues across four categories, revealing a consistent pattern of **misalignment between requirements, design, and implementation**. The primary patterns are: (1) preconditions and alternative flows in use cases that contradict each other, leaving ambiguity about system behavior; (2) use cases that conflate multiple distinct operations (e.g., add and delete in UC2, borrow and return in UC4) without clear separation of flows and error handling; (3) critical business rules and validation requirements stated informally or only in implementation code, rather than formally specified in the requirements section; and (4) incomplete test coverage that does not verify important alternative flows and error conditions described in the use cases, particularly around data validation, stock control, and security constraints.

Critical areas (HIGH severity) require immediate attention. ISS-002 presents a logical contradiction in UC1 preconditions versus alternative flows that must be resolved to clarify whether non-registered users can access login. TST-001 reveals that user registration lacks input validation and uniqueness checks despite requirements assuming these controls exist. TST-002 exposes an untested business rule preventing loans when book stock is zero, a core domain constraint. These gaps undermine the reliability and security of the system.

The overall quality of the document is **below acceptable standards for a professional software specification**. While the document demonstrates understanding of use case notation and basic testing concepts, it suffers from incomplete requirements formalization, insufficient traceability between requirements and tests, and a significant gap between stated design principles (e.g., DTO usage) and actual implementation. The lack of a formal traceability matrix and the absence of explicit business rule specifications make it difficult to verify that the implementation satisfies the requirements.

1. **Resolve requirement contradictions:** Audit and correct all preconditions and alternative flows in use cases (especially UC1, UC2, UC4, UC6) to eliminate logical contradictions. Create a formal requirements specification document that explicitly states all business rules, validation constraints, and security requirements in one place, rather than scattering them across use case narratives and implementation code.
2. **Implement missing validation and error handling:** Add comprehensive input validation to registerUser and all controller methods; implement and test the stock-control rule (TST-002) that prevents loans when disponibili == 0; add test cases for all alternative flows and error conditions currently described in use cases but not tested (UC2 delete constraints, UC4 return errors, UC7 password validation).
3. **Establish and document traceability:** Create a formal traceability matrix mapping each use case (basic and alternative flows) to specific test cases and implementation methods. Ensure that every alternative flow and error condition in the requirements has at least one corresponding test case, and document this mapping explicitly in the test section.

3 Strengths

- **Comprehensive architectural design with clear separation of concerns.** The project demonstrates a well-structured three-tier architecture separating Models, DAO (Data Access Objects), and Controllers. The use of design patterns such as DAO and Singleton, combined with proper package organization (Models, DaoModels, Controller, DTO, ValidationUtils), shows solid software engineering principles and facilitates maintainability.
- **Complete use case documentation with detailed templates.** Seven use cases are thoroughly documented with structured templates including brief descriptions, actors, pre-conditions, basic flows, alternative flows, and post-conditions. This provides clear functional requirements covering all major system operations from login to book management.
- **Extensive visual design documentation including mockups and UML diagrams.** The project includes detailed mockups for eight different views (Home, Utente, Admin, Registration,

Loans, User Management, Add Book, Password Recovery), comprehensive UML diagrams (Use Case, Class, Package, ER), and a relational database model, demonstrating thorough design documentation.

- **Proper implementation of enterprise patterns and technologies.** The use of JPA/Hibernate for persistence, EntityManager for CRUD operations, DTO pattern for secure data transfer, and PostgreSQL as the relational database shows appropriate technology choices and pattern implementation for a library management system.
- **Multi-level testing strategy covering functional, unit, and domain model tests.** The project includes organized test suites across five controllers (AccessController, AdminController, PasswordController, RegisterController, UtenteController) plus unit tests and domain model tests, demonstrating commitment to quality assurance.

4 Expected Feature Coverage

Of 7 expected features: 6 present, 1 partial, 0 absent. Average coverage: **88%**.

Feature	Status	Coverage	Evidence
Unit testing framework implementation	Present	80% (4/5)	The document shows extensive JUnit tests using assertions like assertTrue, assertFalse, assertEquals, assertNull across controller and DAO tests, and describes unit tests for individual components (controllers, DAOs, domain model). Dedicated test classes such as AccessControllerTest, AdminControllerTest, PasswordControllerTest, RegisterControllerTest, UtenteControllerTest, and various DAO tests are presented, and both unit tests and functional/controller-level tests are documented; however, there is no explicit use of mock dependencies, as tests use real DAOs and a separate test database.

Feature	Status	Coverage	Evidence
Use of UML Diagrams for system modeling	Present	88% (7/8)	The report includes a use case diagram with actors Utente and Admin and use cases like Prendi Libro, Storico Prestiti, Registrazione, Log-in, and describes their relationships. It presents a UML class diagram showing Models, DaoModels, Controller, DTO, and ValidationUtils with relationships, and a package diagram (Diagramma dei Package) visualizing architecture. Multiple UI mockups (Home view, Utente View, Admin View, Registrazione View, Libri in Prestito, Gestione Utenti, Aggiungi Libro, Recupera Password) are provided, using standard UML-like notation and clarifying functional requirements; explicit navigation flow diagrams are not shown, though navigation is implied in mockups.
Identification and definition of system actors	Present	100% (8/8)	Actors Admin and Utente are clearly identified in the Use Case Diagram and in section 2.1.1, which explains their roles. Responsibilities and interactions are documented, e.g., Admin can log in, add/eliminate books, gestire utenti, while Utente can authenticate, visualizzare libri disponibili, prendere/restutuire libri, consultare storico prestiti. The Use Case Templates (Accedi al Sistema, Aggiungi/Elimina Libro, Libri in Prestito, Prendi/Restituisci Libro, Gestione Utenti, Registrazione view, Recupera password) define specific user actions, structured flows, and functional requirements per role, delineating system functionalities to support requirements and development.

Feature	Status	Coverage	Evidence
Definition and Documentation of Use Cases	Present	100% (5/5)	Section 2.2 "Use Case Template" defines seven use cases (e.g., Accedi al Sistema, Aggiungi/Elimina Libro, Libri in Prestito, Prendi/Restituisci Libro, Gestione Utenti, Registrazione view, Recupera password) with specific user interactions and goals. Each template includes Basic Flow and Alternative Flow steps, as well as Pre-Conditions and Post-Conditions. While explicit UML relationships between use cases are not diagrammed, combined use cases like Aggiungi/Elimina Libro and Prendi/Restituisci Libro illustrate related sub-behaviors within a single documented use case.
User interface and interaction design principles	Partial	71% (5/7)	The document provides multiple UI mockups (Home view, Utente View, Admin View, Registrazione View, Libri in Prestito, Gestione Utenti, Aggiungi Libro, Recupera Password) with structured input fields such as username, password, Cart ID, book search, and loan period selection, and shows navigation elements like Log in, Registrazione, Password dimenticata, Libri in Prestito, Gestione Libri, Gestione Utenti, Indietro, Esci. Validation mechanisms are described in text via the ValidationUtils package and use case alternative flows (e.g., errors on missing fields or invalid data), and distinct user roles and their functionalities are reflected in different views. However, there is no explicit step-by-step UI process for reservation/loan creation beyond textual use case flows, and dynamic UI updates of user selections are not specifically described.

Feature	Status	Coverage	Evidence
Separation of concerns in software architecture	Present	80% (4/5)	Section 1.2 and the package diagram describe distinct packages: Models (entities), DaoModels (DAOs), Controller, DTO, ValidationUtils, plus test packages; there is no explicit Service layer package. Interactions between controllers and domain models/DAOs are documented in controller descriptions and code listings (e.g., AccessController using UtenteDao and AdminDao, UtenteController using BookDao, LoansDao, UserDao, AutoreDao). The roles of each component (Models, DaoModels, Controllers, DTO, ValidationUtils) are clearly explained, and the architecture is modular with separate responsibilities for persistence, business orchestration, validation, and data transfer.
Data Access Object (DAO) pattern	Present	100% (7/7)	Section 3.4 explicitly introduces the DAO pattern, defines a generic Dao<T> interface with insert and delete methods, and shows implementations in AdminDao and others. CRUD-like operations and encapsulated queries are described for each DAO: AdminDao (findAdminByUsernameAndPassword), UserDao (controllUser, findUser), UtenteDao (updatePassword, userExist, doesUserExist), AutoreDao (libriAutore), LoansDao (setDueDate, libriloan), BookDao (cambiaDisponibilita, findById). The text explains that DAOs abstract database access from business logic, and multiple unit tests (e.g., Insert Admin Test, Insert Book Test, Loans Test, SetDueDate Test, Insert and Delete Utente Test) demonstrate testing strategies for data access methods.

5 Summary Table

Category	HIGH	MEDIUM	LOW	Total
Architecture	0	0	1	1
Requirements	1	6	0	7
Testing	2	3	1	6
Total	**3**	**9**	**2**	**14**

6 Issue Details

6.1 Architecture (1 issues)

ISS-001 — LOW [100%] — Page 28 The architectural description states that DTOs are used to avoid exposing entities directly, but several controller methods still return entity types (e.g., ResponseDTO<Books>, ResponseDTO<Utente>, ResponseDTO<Admin>) instead of DTOs. Tests are written against these entity-based responses, reinforcing this inconsistency between the stated design principle and the implementation.

> • *I DTO sono usati per il trasferimento sicuro dei dati tra il backend e il client, evitando l'esposizione diretta delle entit'a.*

Recommendation: Decide on a consistent boundary for your controllers. If you want to follow the DTO-based design you describe, refactor controller methods and tests to use DTOs instead of entities where data is returned to the UI layer (e.g., introduce a BookDTO or reuse existing DTOs where appropriate, and adjust tests to assert on DTO fields). If you prefer to keep entities in controller responses for this academic project, update the architectural description to clarify that DTOs are used only for specific queries (LibroAutore, LoanBook) and that other operations return entities. In both cases, keep tests aligned with the chosen architecture.

6.2 Requirements (7 issues)

UC-1

ISS-002 — HIGH [90%] — Page 6 The precondition for UC1 states that the user must already be registered, but the alternative flow explicitly handles the case where the user is not registered. This is a logical contradiction: either the system must prevent access to the login use case for non-registered users, or the precondition is wrong. Similar inconsistencies appear in other UCs where preconditions do not match the described flows.

> *Use Case #1 Accedi al Sistema (Log-in) ... Pre-Conditions L'utente deve essere registrato ... Alternative Flow 3a. Se le credenziali sono errate o l'utente non 'e registrato, il sistema restituisce un messaggio di errore e permette di riprovare (Access Test Controller).*

Recommendation: Align preconditions with the actual behavior described in the flows. For UC1, either: - Option A: keep the precondition "L'utente deve essere registrato" and remove the part "o l'utente non 'e registrato" from the alternative flow, clarifying that non-registered users cannot invoke this use case; or - Option B: change the precondition to something like "L'utente ha accesso alla schermata di login" and keep the alternative flow for non-registered users. Then review all other use cases to ensure that preconditions do not exclude situations that are explicitly handled in basic or alternative flows.

UC-6

ISS-003 — MEDIUM [95%] — Page 5 The actor responsibilities described in the use case diagram narrative and the actors listed in UC6 are inconsistent. The description says Admin can manage users (e.g., cancellation) but does not mention that Admin can self-register; UC6 lists both Admin and Utente as actors for registration, but the brief description says "Un user ha la possibilita di iscriversi" without clarifying whether admins can be created only by the system or also via self-registration. This ambiguity affects security and role management requirements.

> *Gli attori identificati sono:Admin, che rappresenta l'amministratore della biblioteca Utente, che rappresenta un utilizzatore finale del sistema Ogni attore ha accesso a un insieme specifico di funzionalit'a: – Admin: pu'o effettuare il login, aggiungere nuovi libri al catalogo, eliminare libri esi- stenti e gestire gli utenti (es. cancellazione). – Utente: pu'o autenticarsi nel sistema, visualizzare i libri disponibili, prendere un libro in prestito, restituirlo e consultare lo storico dei prestiti. ... Use Case #6 Registrazione view ... Actors Admin, Utente*

Recommendation: Clarify the registration policy and align all descriptions: - Explicitly state in the introduction and in UC6 whether admins can self-register or must be created by an existing admin or by the system. - If admins can self-register, explain how the "codice id della bibloteca" is protected and who knows it. - Update the actor list and brief description of UC6 to clearly distinguish the two scenarios: registration as Utente vs registration as Admin, including who is allowed to perform each. - Ensure the use case diagram and textual descriptions are consistent with this policy.

UC-2

ISS-004 — MEDIUM [99%] — Page 6 UC2 mixes two distinct operations (add and delete) in a single use case and basic flow, but the alternative flow only covers missing fields for the add operation. There is no specified alternative flow for important error conditions such as: trying to delete a book that does not exist, trying to delete a book when some copies are on loan, invalid ISBN format, or validation failures for the delete path. The post-condition is also too generic and does not distinguish between add and delete outcomes.

> *Use Case #2 Aggiungi/Elimina Libro ... Basic Flow 1. L'admin compila tutti i campi nel caso in cui si aggiunge un li- bro.(Add/Remove) 2. L'admin inserisce solo l'isbn del libro se vuole cancellarlo. 3. Il sitema elimina il libro solo nei casi in cui tutte le copie disponibili sono in bibloteca. 4. Se vengono compilati tutti i campi il libro viene aggiunto correttamente. (Test #4 #5 (Implementazione)) . Alternative Flow 3a. Se i campi non vengono compilati tutti, il sistema restituisce un messaggio di errore e permette di riprovare (Admin Test Control- ler). Post-Conditions Un messaggio di conferma .*

Recommendation: Split UC2 into two separate use cases (e.g., "Aggiungi Libro" and "Elimina Libro") or clearly separate the two scenarios within the same UC with distinct basic and alternative flows. For each operation: - Add explicit alternative flows for: book not found, book not deletable because some copies are on loan, invalid or missing ISBN, and validation errors. - Refine post-conditions to state the resulting system state, e.g., "Il libro è presente nel catalogo con le copie disponibili aggiornate" for add, and "Il libro è stato rimosso dal catalogo" for delete. Ensure that these flows match the actual behavior implemented in AdminController (e.g., what happens when bookDao.findBookByIsbn returns null).

ISS-008 — MEDIUM [74%] — Page 31 The mapping between use cases and tests is only partially documented and sometimes implicit via comments like "(Test #6)". There is no explicit traceability matrix showing which test cases cover which use case flows (basic and alternative). Some important alternative/error flows described in the UCs (e.g., UC2 delete constraints, UC4

return errors, UC7 invalid identity data) are not clearly linked to specific tests, making it difficult to ensure full coverage.

> *6.1 Test Funzionali ... 6.1.1 Test AccessController ... 6.1.2 Test AdminController ... 6.1.3 Test PasswordController ... 6.1.4 Test RegisterController ... 6.1.5 Test UtenteController ... Use Case #3 Libri in Prestito ... Alternative Flow Se non ci sono libri in prestito l'elenco dei libro sara vuoto. (Test #6).(Utente Test Controller)*

Recommendation: Create a simple traceability table that maps each use case (and each basic/alternative flow) to one or more concrete test methods. For example: - UC1: basic flow → testAccessUser_Success; alt 3a (invalid credentials) → testAccessUser_NotSuccess and testAccessUser_InvalidInput. - UC2 add: success → testAddBook_Success; alt (campi non compilati) → add a dedicated failing test; delete: success and book-not-found tests. - UC3: empty history → listaLibriUtente with no loans; non-empty history → listaLibriUtente after creating loans. - UC4: book not found, not available, successful loan, successful return, invalid user. - UC5: user not found and successful deletion. - UC6: successful user/admin registration and failure cases. - UC7: user not found, invalid password format, successful change. Update comments in the use case templates to reference the exact test method names instead of generic "Test #X" labels, so that reviewers can easily verify coverage.

UC-3

ISS-005 — MEDIUM [94%] — Page 7 UC3 describes only a very high-level basic flow and a single alternative flow (empty list). It does not specify how the user selects which account is used, how pagination or large histories are handled, or whether the user can filter by active vs past loans. More importantly, it does not state any business rules such as maximum number of concurrent loans, overdue handling, or whether overdue loans are highlighted. Given that loans and due dates are central to the domain, the lack of explicit business rules in the requirements is a gap.

> *Use Case #3 Libri in Prestito ... Basic Flow 1. L'utente controllo lo storico dei libri presi in pre- stiti.(Lista Libri in Prestito) (Implementazione 2. L'utente controllo la scadenza del prestito. Alternative Flow Se non ci sono libri in prestito l'elenco dei libro sara vuoto. (Test #6).(Utente Test Controller) Post-Conditions Elenco dei libri in Prestito .*

Recommendation: Extend UC3 and/or add separate business rules to cover the loan management policies of the library. At minimum: - Specify whether there is a maximum number of active loans per user and how this is enforced. - Define how overdue loans are represented in the history (e.g., highlighted, blocked from new loans). - Clarify whether the user can see only their own loans (and how the system identifies the user) and whether filtering (active vs completed) is supported. Update the basic and alternative flows to include these behaviors and ensure they are testable (e.g., add tests for overdue loans and maximum loan limits if applicable).

UC-4

ISS-006 — MEDIUM [99%] — Page 7 UC4 conflates two different business processes (borrowing and returning) into a single basic flow, and the steps are ambiguous about which actions refer to borrowing vs returning. The only alternative flow covers the case where the book is not available for borrowing, but there are no alternative flows for key error conditions: user not found, book not found, user trying to return a book they do not have, invalid dates, or exceeding allowed loan periods. The post-condition only mentions the case where the book is available, and does not describe the system state after a successful return.

> *Use Case #4 Prendi/Retituisci Libro ... Basic Flow 1. L'utente scrive il titolo del libro nel caso di restituzione basta solo scrivere isbn del libro.(Utente view) (Implementazione) 2. L'utente fa search. 3. L'utente ha la possibilita di prendere il libro in prestito con tre possibili periodi. 4. L'utente conferma il prestito del libro. 5. L'utente puo restituire il libro.(Test #7). Alternative Flow 3a. Se il libro non 'e disponibile, il sistema informa l'utente. (Utente Test Controller). Post-Conditions Un messaggio di conferma nel caso in cui il libro sia disponibile .*

Recommendation: Refactor UC4 into two clearer use cases (e.g., "Prendi Libro" and "Restituisci Libro") or at least separate the two scenarios with distinct basic flows. For each scenario: - Define a precise basic flow (input, search, selection, confirmation). - Add alternative flows for: book not found, user not found, book not available, user has already reached a loan limit (if applicable), user attempting to return a book not currently on loan, and invalid or inconsistent dates. - Specify post-conditions that describe the updated system state: for borrowing, a new Loans record created and disponibilità decremented; for returning, Loans.returnDate set and disponibilità incremented. Align these flows with the actual logic in UtenteController.prendiPrestito and restituisciLibro.

UC-7

ISS-007 — MEDIUM [97%] — Page 8 UC7 mentions that the password must satisfy certain requirements, but these requirements are not specified anywhere in the requirements section. The only detailed description of password rules appears later in the ValidationUtils section, which is implementation-level. This makes the security requirement implicit and not testable at the requirements level.

> *Use Case #7 Recupera password ... Pre-Conditions L'user deve essere gia registrato. . Basic Flow 1. L'user scrive il codice della carta id del utente.(Cambia Password Test #9) 2. L'user deve compilare tutti i campio 3. L'utente ha la possibilita di scriver la nuova password.(). Alternative Flow 3a. Se il codice della carta id e i altri campi non coincidano con l'user gia iscritto non e' possibile cambiare la password. La password deve rispettare certi requisiti. (Test Password Controll).*

Recommendation: Promote the password policy to an explicit non-functional/functional requirement in the requirements section. For example, add a subsection that states: minimum length, required character classes (uppercase, digits, special characters), and any forbidden patterns. Then, in UC6 (registration) and UC7 (password recovery), reference this policy explicitly (e.g., "La password deve rispettare i requisiti definiti nella sezione X"). Ensure that ValidationUtilsPassword implements exactly these documented rules so that tests and requirements are aligned.

6.3 Testing (6 issues)

TST-001 — HIGH [99%] — Page 22 The implementation of registerUser does not perform any validation of input data (cartId, nome, cognome, username, password) or uniqueness checks, while the requirements and test descriptions assume validation of empty/invalid fields and password/username constraints. Existing tests only cover the happy path and a failing admin registration, so there is no test that exposes this inconsistency for normal user registration.

> *public ResponseDTO<Utente> registerUser(String cartId, String nome, String cognome, String username, String password) { 4 Utente utente = new Utente(username, password); 5 userDao.addUser(utente); 6 return new ResponseDTO<>(true, "Utente registrato", utente);*

Recommendation: Align tests and implementation for registration. Either: (a) add validation logic to registerUser (e.g., call ValidationUtilsPassword.newPassword or a dedicated validation method, check username uniqueness, enforce password rules) so that it behaves like

described in UC #6 and in RegisterControllerTest comments, and then add tests such as testRegisterUser_Fail_EmptyFields() and testRegisterUser_Fail_WeakPassword() that assert the correct error messages; or (b) if you intentionally keep registerUser simple, update the UC and test documentation to remove claims about validation for user registration. In any case, ensure that for both user and admin registration there are tests that cover invalid/empty fields and duplicate usernames.

TST-002 — HIGH [96%] — Page 23 The UtenteController implementation explicitly handles the case where a book is not available (disponibili == 0), but there is no test that covers this important alternative flow. Existing tests only cover "book not found" and a successful loan and return. This leaves the stock-control rule (preventing loans when no copies are available) unverified by tests.

> *Use Case #4 Prendi/Retituisci Libro ... Alternative Flow 3a. Se il libro non 'e disponibile, il sistema informa l'utente. (Utente Test Controller).*

Recommendation: Add a test method in UtenteControllerTest that sets up a Books entity with disponibilita = 0 and then calls prendiPrestito with that ISBN and a valid user. Assert that the ResponseDTO has success == false and that the message equals the one in the controller ("Libro non disponibile perche g i in prestito." or the corrected text). Also consider adding a test that verifies disponibilita is decremented correctly when a loan is created and incremented when a book is returned, to fully cover the stock management logic.

TST-003 — MEDIUM [99%] — Page 21 For AdminController, tests cover only basic success and not-found scenarios for addBook and removeBook, but do not verify important business rules and alternative flows described in the use case: preventing deletion when not all copies are in the library, and returning validation errors when required fields are missing or invalid. Similar patterns appear in other controllers where alternative flows are described but not tested.

> *Use Case #2 Aggiungi/Elimina Libro ... 3. Il sitema elimina il libro solo nei casi in cui tutte le copie dispo- nibili sono in bibloteca. 4. Se vengono compilati tutti i campi il libro viene aggiunto corre- ttamente. (Test #4 #5 (Implementazione)) . Alternative Flow 3a. Se i campi non vengono compilati tutti, il sistema restituisce un messaggio di errore e permette di riprovare (Admin Test Control- ler).*

Recommendation: Extend AdminController tests to cover the specified business rules and error flows. For addBook, add tests that pass missing/empty fields or invalid numeric values for "disponibili" and assert that ValidationUtilsAdmin errors are propagated (e.g., message contains "Errore nella richiesta"). For removeBook, create a scenario where a book has active Loans and verify that the system does not delete it and returns an appropriate error (you may need to enforce this rule in AdminController or BookDao first). Apply the same approach to other controllers: for each UC alternative flow, add at least one test method that triggers the error condition and checks the ResponseDTO message and success flag.

TST-004 — MEDIUM [97%] — Page 32 PasswordControllerTest only checks a generic "User not found" case and a successful password change, but does not verify the detailed validation rules mentioned in the use case: mismatch of personal data fields and password strength requirements enforced by ValidationUtilsPassword. As a result, important security-related behaviors are not covered by tests.

> *Use Case #7 Recupera password ... Alternative Flow 3a. Se il codice della carta id e i altri campi non coincidano con l'user gia iscritto non e' possibile cambiare la password. La password deve rispettare certi requisiti. (Test Password Controll).*

Recommendation: Extend PasswordControllerTest with additional cases: (1) provide matching cartId/nome/cognome but an invalid new password that violates ValidationUtilsPassword rules (e.g., too short, missing uppercase or special characters) and assert that the ResponseDTO indicates validation errors; (2) provide a correct password format but wrong personal data (e.g., wrong cognome) and assert that the controller returns the specific error message for user not found or data mismatch. Also add at least one test for the admin path (isAdmin = true) to ensure password changes for admins are validated and persisted correctly.

TST-005 — MEDIUM [97%] — Page 33 The use case for "Libri in Prestito" specifies an alternative flow where the list of loans is empty, but the provided UtenteControllerTest only tests the case where the user has at least one loan. There is no test that verifies the behavior when a user has no loans (e.g., empty list vs. error message), even though the UC explicitly references Test #6 for this scenario.

> *Use Case #3 Libri in Prestito ... Alternative Flow Se non ci sono libri in prestito l'elenco dei libro sara vuoto. (Test #6).(Utente Test Controller)*

Recommendation: Add a dedicated test in UtenteControllerTest for the "no loans" scenario. For example, create a user without any Loans and call utenteController.listaPrestito(cartId). Assert that the ResponseDTO is successful and that getData() returns an empty list, or, if you decide to treat it as an error, assert the exact message you document. Make sure the implementation of listaPrestito (or the DAO method) matches the behavior described in UC #3 and update the documentation if you choose a different behavior.

TST-006 — LOW [100%] — Page 31 The document labels controller tests as "Test Funzionali" but they are written as unit tests directly against controllers and DAOs, without explicit traceability back to each use case step or a coverage matrix. While some UCs reference specific test numbers, there is no systematic mapping that shows which tests cover which basic and alternative flows, making it harder to demonstrate complete coverage to the reader or examiner.

> *6.1 Test Funzionali Questa sezione documenta i test unitari delle principali classi controller. Ogni classe viene testata con JUnit per verificare la correttezza delle operazioni implementate.*

Recommendation: Add a simple traceability table that maps each Use Case (and its basic/alternative flows) to the corresponding test methods. For example, for UC #1 (Log-in) list testAccessUser_NotSuccess, testAccessUser_InvalidInput, testAccessUser_Success; for UC #4 (Prendi/Restituisci Libro) list testPrendiPrestito_BookNotFound, testRestituisciLibro_Success, and the new tests you add for "book not available". This can be a small table in the Test section and does not require code changes, but will make your coverage and design–implementation consistency much clearer.

7 Priority Recommendations

The following actions are considered priority:

1. **ISS-002** (p. 6): Align preconditions with the actual behavior described in the flows. For UC1, either: - Option A: keep the precondition "L'utente deve essere registra...
2. **TST-001** (p. 22): Align tests and implementation for registration. Either: (a) add validation logic to registerUser (e.g., call ValidationUtilsPassword.
3. **TST-002** (p. 23): Add a test method in UtenteControllerTest that sets up a Books entity with disponibilita = 0 and then calls prendiPrestito with that ISBN and a valid ...

8 Traceability Matrix

Of 7 traced use cases: 7 fully covered, 0 without design, 0 without test.

ID	Use Case	Design	Test	Gap
UC-1	Accedi al Sistema (Log-in)	✓	✓	—
UC-2	Aggiungi/Elimina Libro	✓	✓	—
UC-3	Libri in Prestito	✓	✓	—
UC-4	Prendi/Retituisci Libro	✓	✓	—
UC-5	Gestione Utenti	✓	✓	Functional test explicitly targeting AdminController.eliminaUtenteController (Gestione Utenti) is referenced as Test #8 ...
UC-6	Registrazione view	✓	✓	—
UC-7	Recupera password	✓	✓	—

9 Terminological Consistency

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

Group	Variants found	Severity	Suggestion
User / Utente / Users / user / userRole / userCart / userDao / UserDao	Utente; Users; User; user; userRole; userCart; userDao; UserDao; registerUser; accessUser; ValidationUtilsUtente; Test UtenteController; "User" (nel testo in inglese); "user" (nel DTO e nelle query); tabella "Users"; classe "Users"; relazione "Utente - Users"	MAJOR	Use a single pair of terms for the two roles and keep them consistent across code, diagrams, and text. For example, use "Utente" for the standard role and "Admin" for the administrator role everywhere (models, DAO, controllers, ER, use cases, UI).
Admin / admin / AdminDao / AdminDAO / addAdmin / doesUtenteExist	Admin; admin; AdminDao; AdminDAO; adminDao; addAdmin; doesUtenteExist (in AdminDao, ma riferito ad Admin); "Admin - Users" (relazione ER); "AdminController"; "registerAdmin"	MAJOR	Choose one naming convention for the administrator entity and its DAO and use it consistently. For example, keep "Admin" (singular) for the entity and "AdminDao" for the DAO, and avoid mixing with "AdminDAO" or other variants.

Group	Variants found	Severity	Suggestion
Utente / User / user (nel testo dei casi d'uso e UI)	Utente; User; user; "Un user ha la possibilita di iscriversi"; "L'user ha la possibilita di cambiare il password"; radio button "User" nella Registrazione View; attore "Utente" nel Use Case Diagram; "Iscrizione User"; "accesso di un USER"	MAJOR	Use a single term for the library user role in the UI and use cases. For example, consistently use "Utente" in all Italian UI labels and use case descriptions instead of alternating with "User".
cartId / Cart ID / cartID / Cart ID (UI)	cartId; Cart ID; cartID; "codice della carta id"; campo "Cart ID" nella UI; attributo "cartId" in Loans; attributo "cartId" in Users; parametro "cartID" nelle query; metodo listaPrestito("12345") che usa cartId	MAJOR	Use a single term for the library card identifier and keep it consistent in all layers. For example, standardize on "cartId" (camelCase) in code and DTOs and "Cart ID" in UI, avoiding mixed forms like "Cart ID", "Cart ID", "cartID".
Books / BookDao / BooksDao / bookDao	Books; BookDao; bookDao; BooksDao (nel testo: "BooksDao"); metodo findBookByIsbn; metodo removeBookByisbn; metodo cambiaDisponibilita; tabella "books"	MINOR	Use a single term for the book entity across DAOs and models. For example, keep "Books" for the entity and "BookDao" for the DAO, and avoid mixing with "BooksDao" in textual descriptions.
Loans / Loan / LoansDao / LoanDao / LoanDaoProxy	Loans; Loan; LoansDao; LoanDao; loansDao; LoanDaoProxy; entità "Loans"; descrizione "entit'a Loan" in LoansDao	MINOR	Use a single term for the loan entity. For example, use "Loans" for the entity and "LoansDao" for the DAO, and avoid mixing with singular "Loan" in the description.
ValidationUtils* / Validation* / ValidatioUtilsAccess / ValidationUtilsAdmin / ValidationUtilsPassword / ValidationUtilsUtente	ValidationUtils; ValidationUtilsAdmin; ValidationUtilsPassword; ValidationUtilsUtente; ValidatioUtilsAccess; ValidationAccess (nel diagramma); ValidationAdmin; ValidationPassword; ValddidationAccess (nel diagramma UML dei package)	MINOR	Use a single naming convention for the validation utility classes and keep it aligned between text and code. For example, standardize on "ValidationUtilsAdmin", "ValidationUtilsPassword", "ValidationUtilsUtente", "ValidationUtilsAccess".

Group	Variants found	Severity	Suggestion
ValidatioUtilsAccess / ValidationUtilsAccess / ValidationAccess	ValidatioUtilsAccess.validateAccess; ValidationAccess (nel diagramma UML); riferimento testuale a "ValidatioUtilsAccess"; assenza di una classe effettiva chiamata esattamente così in ValidationUtils	MINOR	Use a single term for the access validation helper and its usage. For example, use "ValidationUtilsAccess" (or similar) consistently in both code and narrative, and avoid the misspelled "ValidatioUtilsAccess".
DAO / Dao / DAO Interface / DaoModels	DAO; Dao; DAO Interface; DaoModels; "Pattern DAO (Data Access Object)"; "interface DAO"; "implemenetano l'interface DAO"	MINOR	Use a single term for the DAO interface and its implementations. For example, consistently use "Dao" (capital D, rest lowercase) for the interface and "AdminDao", "UserDao", etc. for implementations, and avoid mixing with "DAO Interface" or inconsistent capitalization like "UserDao" vs "UserDao" in lists.
DTO / DTOs / proxy (riferito a DTO)	DTO; DTO Data Transfer Object; "due proxy" riferiti a LoanBook e LibroAutore; "classi pure DTO"; "DTO (LibroAutore, LoanBook)"	MINOR	Use a single term for the DTO pattern and its role. For example, consistently use "DTO" (plural "DTO" or "DTOs") and avoid mixing with "proxy" when referring to the same concept.