Software Engineering Audit Report

ApartamentSWE.pdf

CAPRA

February 25, 2026

Contents

1 Document Context

Project Objective

The application is a Java-based accommodation booking system that manages reservations for B&Bs, apartments, and hotels. Users can search and book accommodations by specifying location, check-in/check-out dates, number of guests, and various filters. Additionally, users can cancel bookings, save accommodations to favorites, and write reviews, while administrators can manage users, accommodations, and reviews.

Main Use Cases

- UC-1 – Login: User or Admin accesses the system by entering email and password credentials with error handling for incorrect credentials.
- UC-2 – Search: User searches for accommodations using location, dates, number of rooms/people, and optional filters to retrieve a filtered list.
- UC-3 – Book: User books an accommodation after search, with system verification of availability and confirmation messaging.
- UC-4 – Registration: User creates a new account by providing name, surname, email, password, username, and favorite location.
- UC-5 – Delete User: Admin removes a user from the database by providing user identification (ID or email).
- UC-6 – Add Accommodation: Admin adds a new accommodation to the database with all required parameters and system validation.
- Browse and Save to Favorites: User saves accommodations to a favorites list for quick access.
- Write Review: User writes a review with comment and rating for a booked accommodation.
- Manage Profile: User updates profile information, views reviews, favorites, and bookings, and manages account settings.
- Recovery Password: User recovers forgotten password through system assistance.
- Delete Review: User or Admin removes a review from the system.
- CRUD Accommodations: Admin performs create, read, update, and delete operations on accommodations.
- CRUD Reviews: Admin views and deletes reviews by user or accommodation.

Functional Requirements

- User registration with email uniqueness validation and password security.
- User login with credential verification and password recovery functionality.
- Advanced accommodation search with dynamic filtering by location, dates, price, rating, and amenities.
- Booking creation with availability verification and state management.
- Booking cancellation with automatic availability and fidelity points updates.
- Review creation and deletion with automatic accommodation rating calculation.
- Favorite accommodation management (save and remove).
- User profile management with editable personal information.
- Fidelity points system that accumulates with purchases and enables discounts.
- Admin user management (search, view, delete users).
- Admin accommodation management (add, update, delete, view accommodations).
- Admin review management (view and delete reviews by user or accommodation).
- Admin password change functionality.
- CLI interface with menu-driven navigation and color-coded output.

Non-Functional Requirements

- Database integrity maintained through SQL triggers for automatic accommodation rating updates.
- Efficient query construction using `StringBuilder` for dynamic SQL generation based on user parameters.
- Connection pooling via Singleton pattern to prevent database connection conflicts.
- Separation of concerns through DAO pattern isolating data access from business logic.
- Prepared statements for SQL injection prevention and query optimization.
- Memory leak prevention through safe resource closure in `DBUtils`.
- User-friendly CLI interface with color highlighting for important information.
- Comprehensive error handling with clear exception messages for database and validation errors.
- Support for multiple accommodation types (B&B, Hotel, Apartment) with flexible attribute management.
- Scalable search parameters handling through Builder pattern with 19 optional attributes.

Architecture

The application follows a layered architecture with four main components: (1) **CLI Interface** – command-line user interaction layer implemented in `Main.java`; (2) **Business Logic** – four controllers (`UserController`, `ProfileUserController`, `ResearchController`, `AdminController`) managing application logic and use case flows; (3) **Domain Model** – entity classes (`RegisteredUser`, `Booking`, `Accommodation`, `Review`, `SearchParameters`, `SearchParametersBuilder`) representing system concepts; (4) **ORM/DAO Layer** – data access objects (`UserDAO`, `BookingDAO`, `AccommodationDAO`, `ReviewDAO`, `PreferenceDAO`) with `DatabaseConnection` singleton managing PostgreSQL connectivity via JDBC. Design patterns employed include Singleton (database connection), DAO (data access separation), Builder Telescoping Constructor (search parameters), and Mapper (entity relationships). Technology stack: Java, PostgreSQL, JDBC, JUnit testing framework.

Testing Strategy

Testing is implemented using JUnit framework across three levels: (1) **Domain Model Tests** – validation of `AccommodationTest` and `SearchParametersBuilderTest` for design pattern correctness; (2) **Business Logic Tests** – functional testing of all controller methods (`UserControllerTest`, `ProfileUserControllerTest`, `ResearchControllerTest`, `AdminControllerTest`) verifying both normal and alternative use case flows with assertions on return values and state changes; (3) **ORM/DAO Tests** – structural testing of database interactions through exception capture and result verification for methods like `getReviewByUser()`, `addBooking()`, and preference operations. Test results show 12 test classes with total execution time of 1 second 424 milliseconds, covering login, registration, search, booking, profile management, and administrative operations.

2 Executive Summary

Quick Overview

Total issues: **15** — HIGH: **2** MEDIUM: **10** LOW: **3**

Average confidence: **97%**

Executive Summary: ApartmentSWE Document Audit

This document is a Software Engineering project report for an apartment booking system developed by a university student. It comprises requirements specification (use cases, ER model), high-level design (architecture, class diagrams), implementation details (Java code snippets, DAO and controller classes), and test documentation. The purpose is to demonstrate a complete software development lifecycle from requirements through testing.

The audit identified **15 issues across three categories**. The most significant pattern is a **systematic gap between documented requirements and implementation**: use cases are incomplete (only 6 of 20+ use cases have textual templates), business rules are described informally rather than as explicit requirements, and pre-conditions and post-conditions are underspecified or missing. A second major pattern is **incomplete test coverage of error and alternative flows**: while happy-path scenarios are tested, exception handling, validation failures, and database errors are only partially covered. A third pattern involves **inconsistencies between specification and code**: the Mapper abstraction is documented but not implemented, search parameters differ between use case and implementation, and booking side effects are not formally specified as atomic operations.

Critical areas (HIGH severity) require immediate attention. First, the booking and cancellation logic lacks formal specification of availability management, fidelity point updates, and transactional consistency (ISS-003). Second, error and exception flows across all use cases are not adequately tested, leaving significant functional gaps uncovered (TST-001). These gaps create risk of undefined behavior in production.

Overall quality assessment: The document demonstrates a reasonable understanding of software engineering concepts and includes most required artifacts (use cases, ER model, class diagrams, code, tests). However, the work is **incomplete and inconsistent**. The requirements specification is partial and informal, the design-to-code mapping is loose, and test coverage does not systematically validate use case flows. The document would not meet professional standards for a production system.

Priority actions for remediation:

1. **Complete and formalize all use case templates**: Write textual specifications for the remaining 14+ use cases (Browse, Save to Favorites, Write Review, Manage Profile, Update Accommodations, etc.) with explicit pre-conditions, post-conditions, and alternative flows. Clarify all business rules (fidelity points, discount policy, review eligibility, booking cancellation constraints) as formal requirements.
2. **Specify and test all error and exception flows**: For each use case, document and implement tests for validation failures, missing mandatory fields, database errors, and recovery scenarios. Ensure that error handling is consistent across all controllers and DAOs, and that tests explicitly map to use case alternative flows.
3. **Reconcile design and implementation**: Remove or implement the Mapper abstraction, align the Search use case with the actual SearchParametersBuilder, and formally specify the booking operation as an atomic transaction with documented side effects and rollback behavior.

3 Strengths

- **Comprehensive Design Documentation** The document provides well-structured design artifacts including Use Case Diagrams with clear actor definitions, detailed Use Case Templates with base flows and alternative flows, and complete Class Diagrams organized into three logical packages (Domain Model, ORM, Business Logic). This demonstrates thorough requirements analysis and architectural planning.
- **Appropriate Design Pattern Implementation** The project correctly applies multiple design patterns for specific purposes: Singleton for database connection management to prevent conflicts, Mapper pattern for managing relationships between entities, Builder Telescoping Constructor for handling complex search parameters with optional attributes, and DAO pattern for data persistence abstraction. Each pattern is justified and properly implemented.
- **Well-Defined Layered Architecture** The system follows a clear four-layer architecture (CLI Interface, Business Logic, Domain Model, ORM) with proper separation of concerns. The architecture diagram clearly shows the interaction flow from client through interface to database, promoting maintainability and testability.
- **Comprehensive Test Coverage** The document includes test sections for Domain Model, Business Logic, and ORM layers with documented test results, indicating systematic validation across multiple architectural components.
- **Detailed Mockup Specifications** Seven detailed UI mockups (MK#1 through MK#7) are provided with clear references to use cases, enabling developers to understand user interface requirements and improving communication between stakeholders.

- **Input Validation and Error Handling** The SearchParametersBuilder implementation demonstrates robust validation logic with specific exception handling for invalid dates, empty fields, and invalid price ranges, showing attention to data integrity and user experience.

4 Expected Feature Coverage

Of 7 expected features: 5 present, 2 partial, 0 absent. Average coverage: **88%**.

Feature	Status	Coverage	Evidence
Unit testing framework implementation	Partial	60% (3/5)	The document explicitly states tests were implemented with JUnit and shows assertion-based tests (assertNotNull, assertEquals, assertNull, assertTrue/assertFalse) and dedicated test classes like UserControllerTest, AdminControllerTest, ReviewDAOTest, etc., covering both business logic and DAO (unit/integration mix). It describes that tests focus on Business Logic and DAO and that for each DAO class structural tests are done, but there is no mention of using mock dependencies for isolation, nor a detailed, systematic testing strategy beyond brief descriptions.
Use of UML Diagrams for system modeling	Present	100% (8/8)	The report includes use case diagrams for User and Admin (Figura 2) and multiple class diagrams for Domain Model, ORM, and Business Logic (Figure 10–12), plus an architecture diagram (Figura 1) and ER/relational diagrams (Figure 13–14). It also provides several UI mockups (Figure 3–9) and uses standard UML notation with actors, use cases, includes/extends, and associations, clearly showing relationships between actors and use cases, navigation flows (e.g., profile, search, booking), and clarifying functional requirements diagrammatically.

Feature	Status	Coverage	Evidence
Identification and definition of system actors	Present	100% (8/8)	The document clearly identifies two roles: User and Admin (section 2.1, use case diagrams). Responsibilities are described in the introduction (User can search, book, cancel, save favourites, write reviews; Admin can delete users, accommodations, reviews, add and modify accommodations) and further detailed in controller descriptions (UserController, ProfileUserController, ResearchController, AdminController). Use case templates define specific user actions and flows, and functional requirements per role are structured via separate use cases and controller responsibilities.
Definition and Documentation of Use Cases	Present	100% (5/5)	Section 2.2 provides detailed templates for use cases Login, Search, Book, Registration, Delete User, and Add Accommodation, each describing user interactions and goals. These templates include base and alternative flows, and several specify pre-conditions and post-conditions. Relationships between use cases (includes/extends) are shown in the use case diagrams in Figura 2.
User interface and interaction design principles	Partial	57% (4/7)	Multiple UI mockups are provided for key interfaces: admin login, user login, account creation, search with filters, search results, accommodation details with booking, and user profile (Figure 3–9), showing structured input fields and navigation elements (buttons like Login, Search, Book Now, Update, Logout, view all). The booking process is visually stepwise (search, view details, Book Now) and input validation is discussed in text for SearchParametersBuilder (date and parameter validation), but the mockups themselves do not document validation behavior or dynamic updates of selections, and distinct user roles are more in text than explicitly differentiated in interaction design beyond separate login screens.

Feature	Status	Coverage	Evidence
Separation of concerns in software architecture	Present	100% (5/5)	Section 1.2 and the architecture diagram (Figura 1) define distinct components: Business Logic (controllers), Domain Model, ORM/DAO, and Interface CLI, and later sections describe each package in detail. Interactions between controllers and domain models/DAOs are documented with code snippets (e.g., ResearchController.doResearch using AccommodationDAO, ProfileUserController.cancelABooking using BookingDAO, UserDAO, AccommodationDAO). The roles of each layer are clearly described, and the modular package structure demonstrates separation of concerns for maintainability.
Data Access Object (DAO) pattern	Present	100% (7/7)	Section 2.5.4 explicitly describes the DAO pattern and its purpose, and the ORM package defines separate DAOs for User, Booking, Preference, Review, and Accommodation (Figure 11). SQL queries are encapsulated inside DAO methods, e.g., BookingDAO.addBooking with an INSERT statement and AccommodationDAO.getAccommodationByParameter building a dynamic SELECT. DatabaseConnection abstracts connection handling, and controllers access data only via DAOs. The ORM Test section shows tests for DAO methods like ReviewDAO.getReviewByUser and PreferenceDAO.save, indicating testing strategies for data access.

5 Summary Table

Category	HIGH	MEDIUM	LOW	Total
Architecture	0	0	2	2
Requirements	1	6	0	7
Testing	1	4	1	6
Total	**2**	**10**	**3**	**15**

6 Issue Details

6.1 Architecture (2 issues)

UC-3

ISS-002 — LOW [98%] — Page 25 Some important business rules related to fidelity points and discounts are only described informally and are not reflected in use cases or tests. The text states that fidelity points "si aggiornano ad ogni acquisto e raggiunta una certa soglia, permette di avere degli sconti", and there is an applyDiscount() method in ResearchController, but there is no requirement specifying the discount policy (threshold, percentage, how many times it can be used) and no use case describing how a user applies a discount during booking.

> *RegisteredUser Contiene le informazioni relative all'utente registrato. Gli attributi della clas- se sono: id (utilizzato come identificativo), username, email (unica all'interno dell'applicazione), password, nome, cognome, punti fedelt'a (che si aggiornano ad ogni acquisto e raggiunta una certa soglia, permette di avere degli sconti), localit'a preferita che indica un genere di esperienza che preferisce, la lista delle prenotazioni effettuate e la lista dei suoi alloggi preferiti.*

Recommendation: Promote the fidelity points and discount mechanism to an explicit requirement. Add a short use case (e.g., "Apply Discount") or extend UC3 Book to describe: how points are earned per booking, what threshold grants a discount, how the discount is applied (e.g., via a "Use Bonus" button as in MK#2), and how points are consumed or reduced. Then ensure there is at least one test in ResearchControllerTest that verifies applyDiscount() against this rule (e.g., enough points vs not enough points). This will make the business logic around discounts clear and verifiable.

See also: TST-003

General Issues

ISS-001 — LOW [98%] — Page 16 The document introduces a Mapper concept in the design section, but there is no corresponding concrete class or interface in the implementation and no tests referring to a Mapper. Relationships between users, reviews, bookings, and accommodations are instead handled directly in DAO and domain classes. This creates a minor inconsistency between the documented architecture and the actual code, and there are no tests that would validate a Mapper abstraction.

> *Mapper Lo scopo del Mapper 'e quello di creare la relazione tra utenti, recensioni e alloggi, e di creare la relazione tra utenti, prenotazioni e alloggi.*

Recommendation: Either (a) remove or adjust the Mapper description from the design section to reflect the actual implementation (relationships handled via DAO and domain associations), or (b) introduce a simple Mapper interface/class as described and refactor the relevant DAO or controller code to use it. If you choose option (b), add at least one unit test that verifies the Mapper behavior (e.g., mapping between Review, RegisteredUser, and Accommodation) and mention this test in the Test section to keep design, implementation, and tests aligned.

6.2 Requirements (7 issues)

UC-3

ISS-003 — HIGH [96%] — Page 7 The booking and cancellation logic is not fully specified from a business perspective, especially regarding availability, consistency, and side effects. UC3 Book does not state how availability is decremented or how fidelity points are updated, and it does not clarify what happens if some steps succeed and others fail. The implementation snippet for

cancelABooking shows multiple side effects (update booking state, update user fidelity points, update accommodation availability) but there is no corresponding requirement or transactional rule describing these as an atomic business operation.

> *Use Case 3 Book Descrizione L'utente prenota un alloggio. ... Flusso Alternativo 2a. Se la disponibilit'a 'e zero, il sistema restituisce un mes- saggio di errore. 2b. Se si verifica un problema durante il salvataggio della prenotazione nel database, il sistema invia un messaggio di errore. 2c. Se l'utente non aveva inserito certi parametri per la ricerca (data check-in check-out e numero di stanze e persone), gli verr'a chiesto di inserirle per effettuare la prenotazione Pre-condizioni L'utente deve aver fatto il login e deve aver effettuato la ricerca. Post-condizioni La prenotazione effettuata verr'a aggiunta a quelle gi'a effet- tuate dell'utente.*

Recommendation: Extend the requirements for booking and cancellation to describe all business effects and their consistency. For booking: specify that, on success, availability is decreased, a booking record is created with state Booking_Confirmed, and fidelity points are increased (if that is the intended behavior). For cancellation: describe that the booking state changes, availability is increased, and fidelity points are decreased, and clarify whether these must succeed together (transactional behavior) or can partially fail. Then align UC3 Book and the (missing) "Cancel Booking" use case with this logic so that tests can verify all side effects, not only the presence of a booking record.

See also: TST-003

UC-1

ISS-007 — MEDIUM [99%] — Page 5 Use case templates systematically lack explicit and complete pre-conditions and post-conditions, and some alternative flows are underspecified. For example, UC1 Login has no pre-condition (e.g., user not already logged in), and the post-condition does not distinguish between success and failure. Similar omissions appear in UC3 Book, UC4 Registration, UC5 Delete User, and UC6 Add Accommodation, where pre-conditions are minimal or missing and post-conditions do not cover error outcomes.

> *Use Case 1 Login (Tst#1) Descrizione L'utente accede al sistema inserendo le sue credenziali. ... Post-condizioni L'utente 'e autenticato dal sistema e ha accesso alle sue fun- zionalit'a.*

Recommendation: For all documented use cases (UC1–UC6), revise the templates to clearly separate success and failure conditions. Add explicit pre-conditions (e.g., "utente non autenticato" for Login, "utente autenticato" for Book, "admin autenticato" for admin operations) and post-conditions for both successful and failed executions (e.g., "nessuna prenotazione viene creata" when Book fails). Where an alternative flow only says "il sistema invia un messaggio di errore", add what remains true about the system state (no changes, no partial updates). This will make the behavior precise and testable.

See also: TST-001

ISS-009 — MEDIUM [99%] — Page 33 The mapping between use cases and tests is only partially documented. While UC1 Login and UC4 Registration are explicitly linked to tests (Tst#1, Tst#2), other critical use cases such as UC2 Search, UC3 Book, UC5 Delete User, and UC6 Add Accommodation are not clearly mapped to specific test cases in the text. The summary of controller tests (figures 20–21) shows that methods are tested, but it is not clear which tests cover which use case flows (especially alternative/error flows).

> *Per i controller sono stati effettuati dei test su tutte le loro funzioni (test funzio- nali), prestando attenzione che seguano le direttive dello use case. Ne vengono riportati alcuni.*

Recommendation: Create a simple traceability table (even half a page) that maps each documented use case (UC1–UC6 and any additional ones you add) to the corresponding JUnit test classes and methods. For each UC, indicate which test covers the main flow and which cover alternative/error flows. Where you find that a use case (e.g., UC2 Search or UC3 Book) has no explicit test for an alternative flow described in the template (e.g., missing mandatory fields, DB error), add at least one test method to cover that behavior and reference it in the table.

See also: TST-001

UC-2

ISS-008 — MEDIUM [92%] — Page 6 There is an inconsistency between the Search use case and the implemented search parameters and builder. UC2 only mentions a few mandatory fields (place, check-in, check-out), while the SearchParametersBuilder and mockups support many optional filters (category, budget, rating, services). The use case does not clarify which fields are mandatory vs optional, nor how invalid combinations are handled (e.g., only one of minRating or specificRating, date validation).

> *Use Case 2 Search Descrizione L'utente cerca l'alloggio di suo interesse. ... Flusso Alternativo 3a. Se l'utente non inserisce alcune informazioni necessa- rie alla ricerca (es: il luogo dove vuole andare, la data di check-in, la data di check-out), il sistema restitui- sce un messaggio di errore. 3b. Se si verifica un problema all'interno del database du- rante la ricerca dell'alloggio, il sistema invia un mes- saggio di errore. Pre-condizioni L'utente deve aver fatto il login. Post-condizioni L'utente ricever'a una lista di alloggi da consultare.*

Recommendation: Update UC2 Search to align with the implemented SearchParameters-Builder and MK#6. Explicitly list which parameters are mandatory (at least place, check-in, check-out, number of people/rooms if required) and which are optional filters. Describe how the system behaves when optional filters are omitted, when invalid combinations are provided (e.g., both min and specific rating), and how validation errors from SearchParametersBuilder (e.g., past dates, inconsistent dates) are reported to the user. This will make the search behavior predictable and consistent with the code.

See also: TST-002, TST-005

General Issues

ISS-004 — MEDIUM [99%] — Page 3 Several important business rules mentioned in the introduction are not captured as explicit requirements or use cases. For example, "cancellare le prenotazioni" and "modificare gli alloggi gi'a presenti" are described informally but there is no dedicated use case template for cancel booking, update accommodation, or for managing favorites and reviews from the user side. This gap makes it unclear what constraints apply (e.g., when a booking can be cancelled, what fields an admin can modify, whether users can edit or only delete reviews).

> *Inoltre, l'utente potr'a anche cancellare le prenotazioni, inserire tra i preferiti gli alloggi e lasciare delle recensioni a essi. 'E inoltre presente un Admin che pu'o cancellare definitivamente gli utenti, gli alloggi o le recensioni, aggiungere nuovi alloggi e modificare gli alloggi gi'a presenti.*

Recommendation: For each business capability stated in the introduction (cancel bookings, manage favorites, write/delete reviews, admin modification of accommodations), create or extend use cases that define: who can perform the action, under which conditions (e.g., cancellation allowed only before check-in), what fields can be changed, and what side effects occur (e.g., rating recalculation, availability changes). Use the existing controllers and DAO methods as a guide to infer the intended behavior and make it explicit in the requirements section.

ISS-005 — MEDIUM [90%] — Page 4 The use case diagram states that "Write Review" includes both "Use Case 2 Search" and "Use Case 3 Book", but there is no textual use case for "Write Review" and no requirement that clearly states the pre-condition for writing a review (e.g., only users who have booked that accommodation can review it). The ER model and triggers show that reviews affect accommodation ratings, but the business rule about who can review and whether multiple reviews per booking are allowed is missing.

> *Write Review Use Case 2 Search Use Case 3 Book*

Recommendation: Define a dedicated "Write Review" use case template that specifies: (1) pre-conditions (user logged in, has at least one completed booking for that accommodation, has not already reviewed it if that is a constraint), (2) main flow (select accommodation, enter rating and comment, save review), (3) alternative flows (invalid rating, DB error), and (4) post-conditions (review stored, rating updated via trigger). Ensure this is consistent with the ER model (one review per user/accommodation pair if that is intended) and with the ReviewDAO methods and tests.

ISS-006 — MEDIUM [100%] — Page 4 The use case model is incomplete: only six use cases (Login, Search, Book, Registration, Delete User, Add Accommodation) have textual templates, while many other use cases shown in the diagrams (e.g., Browse, Save to Favorites, Write Review, Manage Profile, Recovery password, Update Profile, See My Reviews, See My Favourites Accommodation, See My Bookings, Delete Review, Delete Favourite Accommodation, Search User, See Users, See Review By User, Update Accommodations, See Accommodations, Delete Accommodation) have no detailed description. This makes the functional requirements for a large part of the system implicit and undocumented.

> *Sono presenti 2 tipi di utenti: lo User e l'Admin. Nel diagramma sottostante vengono rappresentati i casi d'uso per i due tipi di utenti:*

Recommendation: Identify all use cases that appear in the diagrams but lack a template (e.g., Browse, Save to Favorites, Write Review, Manage Profile and its subfunctions, Recovery password, all CRUD operations for reviews and accommodations). For each of these, add at least a short use case template with: goal, actors, pre-conditions, main flow, alternative/error flows, and post-conditions. Keep them concise but explicit enough that a reader can understand what the system must do and how errors are handled.

6.3 Testing (6 issues)

UC-1

TST-001 — HIGH [99%] — Page 5 Error and exception flows described in the use cases (e.g., wrong credentials, DB errors, recovery password) are only partially covered by tests. The login test covers wrong credentials and wrong password, but there is no explicit test for DB failure handling or for the recovery password flow, and similar error branches in other use cases (Search, Book, Registration, Delete User, Add Accommodation) are not backed by dedicated tests.

> *Flusso Alternativo 3a. Se le credenziali sono errate, il sistema invia un mes- saggio di errore. 3b. Se si verifica un problema all'interno del database du- rante la ricerca dell'utente, il sistema invia un mes- saggio di errore. 3c. Se l'email 'e giusta, ma la password sbagliata, il siste- ma consente un recupero della password o l'immissio- ne di un ulteriore tentativo*

Recommendation: For all use cases that define alternative error flows (Login, Search, Book, Registration, Delete User, Add Accommodation), add explicit tests that simulate these conditions and assert the expected behavior. For example: in UserControllerTest, add a test that simulates

a database failure during login (e.g., by using a test double or forcing DatabaseConnection to throw) and verify that the controller returns the correct error indication; add a separate test for the recovery password path when the email is correct and the password is wrong. Similarly, in tests for ResearchController and BookingDAO, add tests where availability is zero and where the DB operation fails, asserting that the correct error is propagated or handled. Document in each test which use case alternative flow (e.g., UC1-3b, UC2-3b, UC3-2b, UC4-3b, UC5-3b, UC6-3a) it is covering to improve traceability.

See also: ISS-007, ISS-009

UC-2

TST-002 — MEDIUM [98%] — Page 6 The Search use case defines validation behavior when mandatory search parameters are missing, but there is no explicit test that maps to this behavior at controller or DAO level. Existing tests focus on SearchParametersBuilder validation and on ResearchController.doResearch() success, but not on the specific UC-2 alternative flow where the user omits required fields and the system must respond with an error.

> *Flusso Alternativo 3a. Se l'utente non inserisce alcune informazioni necessa- rie alla ricerca (es: il luogo dove vuole andare, la data di check-in, la data di check-out), il sistema restituisce un messaggio di errore.*

Recommendation: Add a dedicated test in ResearchControllerTest (or in the CLI-level tests if you have them) that calls doResearch() with SearchParameters missing mandatory fields (e.g., null or empty place, missing dates) and asserts that the controller either throws a specific exception or returns a clear error result consistent with UC-2 step 3a. Link this test to UC-2 in its name or comments (e.g., testDoResearch_MissingMandatoryFields_UC2_3a) to make the traceability explicit.

See also: ISS-008

TST-005 — MEDIUM [99%] — Page 33 The document states that all controller functions have been tested according to the use cases, but only a subset of concrete tests is shown (login and registration). For other important use cases such as Search (UC-2), Book (UC-3), Delete User (UC-5), and Add Accommodation (UC-6), the report only lists test method names in screenshots without explaining which use case steps and alternative flows they cover. This weakens traceability between requirements and tests.

> *4.2 Business Logic Test Per i controller sono stati effettuati dei test su tutte le loro funzioni (test funzio- nali), prestando attenzione che seguano le direttive dello use case. Ne vengono riportati alcuni.*

Recommendation: Improve traceability by explicitly mapping each controller test method to the corresponding use case and flow. For example, in the text or in comments in the test classes, state that ResearchControllerTest.doResearch() covers UC-2 base flow, booking() covers UC-3 base flow, AdminControllerTest.removeUser() covers UC-5 base and alternative flows, and addAccommodation()/updateAccommodation() cover UC-6. Where tests already exist, add short descriptions in the report explaining which UC steps they verify. If any controller method implements a use case step that currently has no dedicated test, add such tests and reference the UC in the method name (e.g., testDeleteUser_InvalidData_UC5_3a).

See also: ISS-008

UC-3

TST-003 — MEDIUM [98%] — Page 7 The Book use case defines several alternative flows (no availability, DB error on save, missing search parameters). The BookingDAO.addBooking()

method enforces availability and interacts with the DB, but the provided tests for ResearchController.booking() and BookingDAO focus on successful booking and do not explicitly verify each of these alternative flows and their user-visible effects.

> *Flusso Alternativo 2a. Se la disponibilit'a 'e zero, il sistema restituisce un mes- saggio di errore. 2b. Se si verifica un problema durante il salvataggio della prenotazione nel database, il sistema invia un mes- saggio di errore. 2c. Se l'utente non aveva inserito certi parametri per la ricerca (data check-in check-out e numero di stanze e persone), gli verr'a chiesto di inserirle per effettuare la prenotazione*

Recommendation: Extend BookingDAOTest and ResearchControllerTest with separate tests for each UC-3 alternative flow: (1) a test where accommodation.getDisponibility() is 0, asserting that addBooking() throws the expected RuntimeException and that the controller translates it into the correct error behavior; (2) a test that simulates a DB failure (e.g., invalid SQL or mocked Connection) and verifies that the error is caught and reported as specified in UC-3-2b; (3) a test that attempts booking without previously set SearchParameters (or with missing dates/rooms/people) and checks that the controller requests the missing data or fails with a clear error. Name or comment these tests to reference UC-3-2a/2b/2c for traceability.

See also: ISS-002, ISS-003

UC-4

TST-004 — MEDIUM [98%] — Page 8 The Registration use case specifies handling of invalid data (duplicate or empty email) and DB errors. The register() test covers invalid email and duplicate email by expecting a null result, but there is no test that simulates a DB failure during user persistence, and the mapping between the test behavior (returning null) and the UC-4 requirement of sending an error message is not clearly documented.

> *Flusso Alternativo 3a. Se l'utente inserisce dati non validi (es: l'email gi'a usata o vuota), il sistema invia un messaggio di errore all'utente. 3b. Se si verifica un problema durante il salvataggio del- l'utente nel database, il sistema invier'a un messaggio di errore.*

Recommendation: Augment UserControllerTest with a test that forces a DB error during registration (e.g., by using a broken UserDAO or invalid connection) and asserts that UserController behaves consistently with UC-4-3b (e.g., returns null or throws a specific exception). In all registration tests, add comments or naming that explicitly reference UC-4-3a and UC-4-3b, and, if possible, assert not only the null/non-null result but also that an appropriate error message is produced or logged, to better align with the use case description.

General Issues

TST-006 — LOW [97%] — Page 32 The CLI layer (Main.java and menus) is the actual entry point through which users execute the use cases, but there is no mention of tests that exercise the CLI flows end-to-end. All described tests target Business Logic and DAO directly. This means that potential issues in input parsing, menu navigation, and mapping of user choices to controller calls are not covered.

> *3.4 Interfaccia CLI Per l'applicazione 'e stata realizzata un'interfaccia a linea di comando, imple- mentata nel file Main.java. L'utente, inserendo i vari comandi indicati dal sistema, naviga all'interno dell'applicazione, resa pi'u user friendly grazie all'uso di colori per evidenziare le parole importanti o mancanti.*

Recommendation: Add at least a small set of high-level tests for the CLI layer, even if they are simple. For example, create tests that simulate user input via Scanner (or refactor Main to inject an InputStream) and verify that choosing menu options triggers the expected controller

methods and handles invalid input gracefully. Focus on one or two representative flows (e.g., login + search + booking, admin login + delete user) and document in the report that these tests validate the integration between CLI and Business Logic.

7 Priority Recommendations

The following actions are considered priority:

1. **ISS-003** (p. 7): Extend the requirements for booking and cancellation to describe all business effects and their consistency.
2. **TST-001** (p. 5): For all use cases that define alternative error flows (Login, Search, Book, Registration, Delete User, Add Accommodation), add explicit tests that sim...

8 Traceability Matrix

Of 29 traced use cases: 28 fully covered, 0 without design, 1 without test.

ID	Use Case	Design	Test	Gap
Use Case 1	Login	✓	✓	—
Use Case 2	Search	✓	✓	—
Use Case 3	Book	✓	✓	—
Use Case 4	Registration	✓	✓	—
Use Case 5	Delete User	✓	✓	—
Use Case 6	Add Accommodation	✓	✓	—
UC-User-Search-Browse	Brows	✓	✓	—
UC-User-SaveFavorites	Save to Favorites	✓	✓	—
UC-User-WriteReview	Write Review	✓	✓	—
UC-User-RecoveryPassword	Recovery password	✓	×	Use case Recovery password has controller methods and UI link but no explicit JUnit test case is documented.
UC-User-ManageProfile	Manage Profile	✓	✓	—
UC-User-UpdateProfile	Update Profile	✓	✓	—
UC-User-SeeMyReviews	See My Reviews	✓	✓	—
UC-User-SeeMyFavourites	See My Favourites Accommodation	✓	✓	—
UC-User-SeeMyBookings	See My Bookings	✓	✓	—
UC-User-DeleteReview	Delete Review	✓	✓	—
UC-User-DeleteFavouriteAccommodation	Delete Favourite Accommodation	✓	✓	—

ID	Use Case	Design	Test	Gap
UC-User-DeleteBooking	Delete Booking	✓	✓	—
UC-Admin-Login	Login (Admin)	✓	✓	—
UC-Admin-summaryCrudUsers	summary Crud Users	✓	✓	—
UC-Admin-SearchUser	Search User	✓	✓	—
UC-Admin-SeeUsers	See Users	✓	✓	—
UC-Admin-summaryCrudReviews	summary Crud Reviews	✓	✓	—
UC-Admin-DeleteReview	Delete Review (Admin)	✓	✓	—
UC-Admin-SeeReviewByUser	See Review By User	✓	✓	—
UC-Admin-summaryCrudAccommodations	summary Crud Accommodations	✓	✓	—
UC-Admin-UpdateAccommodations	Update Accommodations	✓	✓	—
UC-Admin-SeeAccommodations	See Accommodations	✓	✓	—
UC-Admin-DeleteAccommodation	Delete Accommodation	✓	✓	—

9 Terminological Consistency

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

Group	Variants found	Severity	Suggestion
Main user actor naming	User, Utente, utente, lo User	MAJOR	Use a single, consistent term for the main non-admin actor; prefer "User" (English) or "Utente" (Italian) everywhere, including diagrams, templates, and text.

Group	Variants found	Severity	Suggestion
Administrator actor naming	Admin, admin, L'Admin, l'admin	MAJOR	Use a single, consistent term for the administrator actor; prefer "Admin" everywhere (including mockups and text) or consistently "Amministratore" if choosing Italian.
Accommodation entity naming	alloggio, alloggi, Accommodation, Acccommodation	MAJOR	Use one consistent term for the accommodation entity; prefer "Accommodation" in English or "Alloggio" in Italian, and align table names and text accordingly.
Favorites feature naming	Favourites, Like, My Savings, alloggi preferiti	MAJOR	Use a single consistent term for the favorites feature; choose either "Favourites" (or "Favorites") or "Savings" and apply it across ER model, tables, mockups, and text.
Search parameters naming	SearchParameters, SearchParametersBuilder, ParametriRicerca, parametri di ricerca	MAJOR	Use one consistent term for the search parameters concept; prefer "SearchParameters" / "SearchParametersBuilder" everywhere and avoid mixing with "ParametriRicerca".
Profile controller naming	ProfileUserController, ProfileController, profileMenu	MAJOR	Use a single consistent name for the profile controller; align section titles, class diagram, and code to either "ProfileUserController" or "ProfileController".
Persistence layer naming	DAO, ORM, Dao	MAJOR	Use a single consistent term for the persistence layer; prefer "ORM" or "DAO" as the package/component name and align the architecture diagram and text.
Booking state naming	State, state, State.Booking_Confirmed	MAJOR	Use a single consistent term for the booking state enumeration; align the enum name and its values (e.g., "State" with value "BOOKING_CONFIRMED" or similar) and avoid mixing textual "state" with enum-specific naming.
Search operation naming	Search, Research, doResearch, Accurate Research, ricerca	MINOR	Standardize the naming of the search operation; choose either "Search" or "Research" and use it consistently in use cases, controllers, methods, and UI text.

Group	Variants found	Severity	Suggestion
Save/unsave accommodation naming	Save to Favorites, saveAccommodation, saveAccomodation, unSaveAccommodation, unsaveAccommodation, addPreference	MINOR	Standardize the naming of the favorites-saving operation; choose one spelling and casing (e.g., "saveAccommodation" / "unsaveAccommodation") and apply it consistently in code, tests, and text.