Software Engineering Audit Report

SANSAVINI-FERRITTI-LIN (FINAL).pdf

CAPRA

February 25, 2026

Contents

1 Document Context

Project Objective

BarberShop is a full-stack application for managing barbershop appointments that enables customers to book appointments by selecting date, time, barber, and services, with automatic exclusion of occupied time slots. The application provides appointment management features for both customers (viewing and canceling bookings) and barbers (viewing appointments, sending communications, and managing services), along with a notification system and password recovery mechanism.

Main Use Cases

- UC-01 – Sign up: Users register as barbers or customers, with barbers entering a secret code for verification.
- UC-02 – Sign in: Authenticated users access the application with email and password, with password recovery option.
- UC-03 – View Appointments: Customers and barbers view their respective appointment lists.
- UC-04 – Add Appointment: Customers book appointments by selecting date, time, barber, service, and payment method.
- UC-05 – Delete Appointment: Customers cancel appointments with confirmation, notifying barber and other customers of freed slots.
- UC-06 – View News: Users view received notifications and communications in a dedicated section.
- UC-07 – View Profile: Users access their profile information and perform logout.
- UC-08 – View Services: Barbers view the list of available services they offer.
- UC-09 – Add Service: Barbers add new services with names and prices.
- UC-10 – Delete Service: Barbers remove services with confirmation.
- UC-11 – Send Communication: Barbers send personalized messages to all customers.

Functional Requirements

- Registration and authentication with role-based access (customer or barber).
- Appointment booking with date, time, barber, service, and payment method selection.
- Automatic exclusion of occupied time slots from selection.
- Appointment viewing and cancellation for customers; appointment viewing for barbers.
- Automatic notification generation for appointment bookings, cancellations, and slot availability.
- Service management (add and delete) by barbers with name and price.
- Communication sending from barbers to all customers.
- Profile information viewing and logout functionality.
- Password recovery via email.
- Support for multiple payment methods: PayPal, credit card, and in-shop payment.

Non-Functional Requirements

- Security: Password hashing using BCrypt algorithm via jBCrypt library.
- Performance: Efficient database queries with proper indexing and join operations.
- Scalability: Modular architecture supporting multiple barbers and customers.
- Usability: Modern and interactive GUI using JavaFX and MaterialFX frameworks.
- Reliability: Automatic slot generation for new barbers via database triggers.
- Maintainability: Clear separation of concerns using MVC pattern and DAO design pattern.
- Testing: Comprehensive test coverage of 96% with unit, integration, and functional tests.
- Data Persistence: PostgreSQL for production and H2 in-memory database for testing.

Architecture

The application follows the Model–View–Controller (MVC) pattern with Maven project structure. The architecture comprises: **View** (FXML files for GUI), **PageControllers** (event handling and user interaction), **Helpers** (utility classes for scene switching and alerts), **Services** (business logic), **Authentication** (SessionManager singleton for user session management), **Model** (domain entities), **Persistence** (DAO pattern with DBConnection singleton), **Security** (password hashing), and **Payment** (Strategy pattern for payment methods). External dependencies include JavaFX, MaterialFX, jBCrypt, JDBC, and testing frameworks (JUnit, Mockito, TestFX).

Testing Strategy

Three testing levels are implemented: **Unit Tests** (78 tests using JUnit and Mockito) verify isolated service behavior with mocked dependencies; **Integration Tests** (14 tests) validate service–DAO interactions using PostgreSQL; **Functional Tests** (71 tests using TestFX) simulate user interactions with the GUI using H2 in-memory database. Overall test coverage is 96% calculated via Jacoco, with comprehensive coverage across all packages including Services (99%), Model (100%), Authentication (100%), and PageControllers (97%).

2 Executive Summary

Quick Overview

Total issues: **14** — HIGH: **2** MEDIUM: **7** LOW: **5**

Average confidence: **93%**

Executive Summary: Software Engineering Document Audit

This audit evaluates a comprehensive Software Engineering document for a barber appointment booking system developed by university students. The document encompasses requirements specification, domain and relational modeling, architectural design, implementation details, and a test plan. The purpose is to assess the completeness, consistency, and traceability of the SWE artifacts against industry standards and best practices.

The audit identified **14 issues across four categories**: Architecture (2), Requirements (8), and Testing (4). The predominant patterns include: (1) incomplete specification of error handling and system failure scenarios in use cases, particularly for transactional operations; (2) inconsistencies between the conceptual model (e.g., multi-service appointments, automatic notifications) and the use case descriptions; (3) gaps in traceability between requirements, design decisions, and test coverage; and (4) lack of alignment between the PostgreSQL integration test environment and the H2 functional test environment regarding schema, triggers, and data initialization.

Critical Areas (HIGH Severity): Two issues demand immediate attention. First, use cases UC-03, UC-04, UC-05, and UC-06 lack explicit specification of system error handling, database failures, and network issues, which is particularly severe for transactional operations such as appointment booking and cancellation where atomicity and user notification are not defined. Second, the dual-database testing strategy (PostgreSQL for integration tests, H2 for functional tests) lacks documented synchronization mechanisms for schema, triggers, and initial data, risking environment-specific failures that may not be caught during testing.

Overall Quality Assessment: The document demonstrates solid foundational work with well-structured sections, reasonable architectural patterns (Strategy pattern for payments, DAO layer abstraction), and comprehensive test coverage in terms of quantity. However, the quality is undermined by incomplete requirements specification, insufficient error scenario documentation, and traceability gaps that prevent systematic verification of all functional requirements. The document is suitable for academic purposes but requires refinement before production use.

Priority Actions:

1. **Enhance use case specifications** by adding explicit alternative flows for system errors, database failures, and network issues. Revise UC-04 and UC-05 to clarify transactional semantics (atomicity, rollback behavior) and post-conditions that specify all side effects (slot removal/restoration, notification generation). Add missing use case templates for password recovery and logout.
2. **Align and document the dual-database testing strategy** by creating a synchronization checklist that maps PostgreSQL schema elements (tables, triggers, constraints, initial data) to their H2 equivalents. Document any SQL dialect differences and establish a process to verify that integration tests on PostgreSQL and functional tests on H2 exercise the same business logic.
3. **Complete traceability mapping** by explicitly linking all use case alternative flows (especially empty-state scenarios in UC-03, UC-06, UC-08) to functional or integration tests. Update the test section to include test IDs for all mapped tests and resolve inconsistencies between the Use Case Diagram (Client, Administrator, Recover Password) and use case templates (Customer, Barber).

3 Strengths

- **Comprehensive Use Case Documentation:** The document provides detailed use case templates for all 11 system functions (Sign up, Sign in, View Appointments, Add Appointment, Delete Appointment, View News, View Profile, View Services, Add Service, Delete Service, Send Communication), each including scope, actors, pre-conditions, post-conditions, base flow, alternative flows, and associated tests. This thorough specification ensures clear requirements traceability.
- **Well-Structured MVC Architecture:** The project implements a clean Model-View-Controller pattern with clearly organized packages (PageControllers, Services, Model, Persistence, Authentication, Security, Payment, Helpers). The architecture diagram and detailed package descriptions demonstrate proper separation of concerns and maintainability.
- **Multi-Level Testing Strategy:** The document describes three testing levels (Unit Tests, Integration Tests, Functional Tests) with explicit test coverage metrics and traceability to use cases. Each use case template references specific test identifiers (e.g., UT-01-05, IT-01, FT-01), establishing clear links between requirements and test cases.
- **Comprehensive Visual Documentation:** The project includes extensive visual artifacts including Use Case Diagram, Class Diagram, Page Navigation Diagram, Entity-Relationship Diagram, and 34 GUI mockups showing various application states (success, error, empty states). This visual coverage aids understanding and validates user interface design.
- **Appropriate Technology Stack:** The selection of technologies is well-justified for the project scope: JavaFX for modern desktop UI, PostgreSQL for production data persistence, H2 in-memory database for testing isolation, jBCrypt for password security, and comprehensive testing frameworks (JUnit, Mockito, TestFX).

4 Expected Feature Coverage

Of 7 expected features: 7 present, 0 partial, 0 absent. Average coverage: **86%**.

Feature	Status	Coverage	Evidence
Unit testing framework implementation	Present	100% (5/5)	The Test section shows assertion-based JUnit tests (e.g., assertTrue, assertFalse, assertEquals in SignInServiceTest, NewAppointmentSlotsServiceTest, functional tests). It documents a systematic approach to unit testing of services with JUnit and Mockito, listing specific unit test classes per service (SignUpServiceTest, SignInServiceTest, AppointmentServiceTest, etc.). Dedicated test classes are defined for specific functionalities and use cases. Isolated testing with mock dependencies is demonstrated (e.g., userDAO and SessionManager mocked in SignInServiceTest using Mockito). The document also explicitly distinguishes and implements both unit and integration tests, plus functional tests, and reports overall coverage via Jacoco.
Use of UML Diagrams for system modeling	Present	100% (8/8)	The document includes a Use Case Diagram (Figura 3) depicting user interactions and actor relationships, and a Class Diagram (Figura 4) representing component relationships. It uses a system architecture diagram (Figura 1) and a Persistence diagram (Figura 43) as structure diagrams. Numerous UI mockups are provided for all main views (registration, login, appointments, booking, news, services, profile, etc.). The diagrams use standard UML notation (actors, use cases, «extend»/«include», classes, interfaces, packages). Relationships between actors and use cases are explicitly shown in the Use Case Diagram. Page Navigation Diagrams for Barber and Customer (Figure 5 and 6) visualize navigation flows. Functional requirements are clarified via the Use Case Diagram and detailed use case templates linked to figures.

Feature	Status	Coverage	Evidence
Identification and definition of system actors	Present	100% (8/8)	Section 2.1 explicitly identifies actors Customer and Barber and describes their capabilities (e.g., Customer can manage appointments, consult communications, view profile; Barber can manage customer appointments, send communications, manage services). Interactions between users and system components are defined via the Use Case Diagram and detailed use case templates (UC-01 to UC-11) that specify actions like Sign up, Add Appointment, Send Comunication. The functional requirements section (1.2) documents requirements per role (e.g., customers vs barbieri for appointments, notifications, profile). These structured descriptions delineate system functionalities per role and list concrete user actions, supporting a structured approach to user requirements and streamlining development.
Definition and Documentation of Use Cases	Present	100% (5/5)	Section 2.2 provides detailed use case templates UC-01 to UC-11, each defining specific user interactions (e.g., Sign up, View Appointments, Add Service). For each use case, user goals are stated in the Scope/User Goal fields. The templates include detailed base flows and alternative flows (e.g., UC-04 Add Appointment with multiple alternative steps, UC-05 Delete Appointment with confirmation alternative). Pre-conditions and post-conditions are specified for every use case. Relationships between use cases (include/extend) are illustrated in the Use Case Diagram (e.g., View Appointments includes Sign in, Delete Appointment extends Add Appointment, Sign in extends Recover Password).

Feature	Status	Coverage	Evidence
User interface and interaction design principles	Present	7% (7/7)	Section 2.5 contains UI mockups for all key interfaces: registration, access, appointments (barber and customer), booking flow, news, send communication, services, and profile. Navigation elements are clearly shown via sidebars and Page Navigation Diagrams (Appointments, New Appointment, News, Profile, Services, Send comunication). The booking process is documented step-by-step in UC-04 and illustrated with calendar selection, slot selection, extra service, confirmation, payment selection, and booking confirmation screenshots. Validation mechanisms are shown via error dialogs for empty fields, wrong secret code, invalid credentials, missing service, and empty communication/service fields. Input forms with structured fields (Name, Surname, E-mail, Password, Phone, Title, Message, Name/Price for services) are visible. Distinct user roles and their functionalities are reflected in different navigation menus and views for Barber vs Customer. Dynamic updates of selections are shown, e.g., available slots updating when a time is booked or freed, extra service added messages, and news listing slot availability notifications.

Feature	**Status**	**Coverage**	**Evidence**
Separation of concerns in software architecture	Present	100% (5/5)	Section 1.3 describes an MVC architecture with distinct packages: View, PageControllers, Services, Model, Persistence (DAO, DBConnection), plus Helpers, Authentication, Security, Payment. Interactions between controllers and domain models are documented: controllers call services, which instantiate/use Model entities and DAOs (described in 1.3.1 and illustrated in the architecture diagram Figura 1 and PageControllers/Services diagrams). Clear relationships between Controller, Service, and DAO layers are explained (services hold DAO references, controllers hold service references, DAOs use DBManager). The role of each component is described in detail in 1.3.1 and in sections 3.1–3.8. The design is modular, with separate packages and single-responsibility classes (e.g., SessionManager, HashingPassword, PaymentStrategy, individual services), explicitly motivated as improving maintainability and reducing duplication.

Feature	Status	Coverage	Evidence
Data Access Object (DAO) pattern	Present	100% (7/7)	Section 3.8 describes the DAO package with interfaces like UserDAO, AppointmentDAO, ServiceTypeDAO, AvailableSlotDAO, NewsDAO and their Concrete*DAO implementations. These interfaces define CRUD-style operations (e.g., addUser, removeUserByEmail, findByEmail, addAppointment, deleteAppointment, addServiceType, removeServiceType, addNotification, deleteNotification). SQL queries are encapsulated within DAO classes, as shown in the detailed example of findByEmailOfBarber in ConcreteAppointmentDAO using JDBC. Separate DAOs exist for different entities (Users, Appointments, Service_Types, Available_Slots, News). The text explicitly states that DAOs abstract data access and isolate persistence logic from the rest of the application, defining clear interfaces for data operations. Patterns for managing data access (Singleton DBManager, DAO interfaces + concrete classes) are documented. Testing strategies for data access are evidenced by integration tests using the real PostgreSQL DB (e.g., UserTest, AppointmentAndSlotsTest, ServiceTypesTest, CommunicationTest) and DBManagerTest/DBTestInitializerTest.

5 Summary Table

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

6 Issue Details

6.1 Architecture (2 issues)

ISS-001 — LOW [100%] — Page 36 The architectural description emphasizes an abstract User superclass with Customer and Barber subclasses, which is consistent with the Model diagram.

However, in the relational model and DAO layer, the Users table stores both roles in a single table without a discriminator-based mapping strategy being described, and the example DAO code manually constructs Customer and Barber objects based on query results. The document does not clarify how the role field in Users is mapped to the correct subclass in all DAOs, which can lead to inconsistent instantiation (e.g., creating a Customer where a Barber is expected) if not handled systematically.

> *La classe User è progettata come una superclasse astratta, condividendo attri- buti e metodi comuni con le sue sottoclassi Customer e Barber. Questa scelta garantisce una progettazione modulare e riutilizzabile, evitando l'instanziazione diretta della classe User.*

Recommendation: Make the mapping between the relational model and the OO inheritance explicit. In the Persistence section, briefly describe how DAOs decide whether to instantiate a Customer or a Barber from a Users row (e.g., by checking the role column). Consider centralizing this logic in a small factory method (e.g., in UserDAO or a UserFactory) that takes a ResultSet row and returns the correct subclass. This will keep the implementation consistent with the abstract User design and reduce the risk of errors when adding new queries.

ISS-002 — LOW [100%] — Page 43 The Payment package is described as implementing the Strategy pattern with PaymentStrategy, PaymentContext, and PaymentFactory, and there is a PaymentTest listed among unit tests. However, the document does not show how PaymentContext and PaymentFactory are integrated into the Services or PageControllers, nor how they are exercised in functional tests. For example, FT-04: testBookAppointmentWithOneServiceSuccess verifies that a PayPal payment is selected and that the appointment and News rows are created, but it does not explicitly state that the correct PaymentStrategy implementation is invoked. This weakens the architectural traceability between the Payment design and its usage.

> *Il pacchetto Payment, illustrato in figura 42, è dedicato alla gestione dei metodi di pagamento e implementa il pattern Strategy per incapsulare le diverse modalità di pagamento (PayPal, carta di credito, pagamento in negozio).*

Recommendation: Strengthen the link between the Payment architecture and its implementation by: (1) adding a short paragraph in the Services section explaining which service uses PaymentContext/PaymentFactory when booking an appointment; (2) extending PaymentTest to verify that PaymentFactory returns the correct strategy for each PaymentMethod and that PaymentContext.executePayment delegates to the chosen strategy; and (3) optionally, in one functional test (e.g., FT-04), assert not only the stored PaymentMethod in the database but also some observable effect of the strategy (even if it is just a log or a stubbed method call) to demonstrate that the Strategy pattern is actually used at runtime.

6.2 Requirements (8 issues)

UC-03

ISS-003 — HIGH [100%] — Page 13 I casi d'uso che dipendono fortemente da dati persistenti e da condizioni di errore (UC-03 View Appointments, UC-04 Add Appointment, UC-05 Delete Appointment, UC-06 View News, UC-08 View Services) hanno flussi base e pochi flussi alternativi, ma non descrivono cosa succede in caso di errori di sistema o di persistenza (es. fallimento accesso DB, indisponibilità servizi, problemi di rete). Questo è particolarmente critico per operazioni transazionali come prenotazione e cancellazione appuntamenti, dove non è specificato se l'operazione è atomica, cosa succede se la rimozione dello slot o l'inserimento della notifica falliscono, o come viene informato l'utente.

> *2.2.3 View Appointments UC-03 View Appointments Scope User Goal Attori Customer, Barber Pre-condizioni L'utente è autenticato nel sistema. Post-condizioni L'utente vede l'elenco degli appuntamenti prenotati. Flusso base 1. L'utente apre la schermata degli appuntamenti. [14, 15] 2. Il sistema mostra l'elenco degli appuntamenti prenotati. 3. L'utente può visualizzare i dettagli di ogni appuntamento. Flusso alternativo 2.1 Se l'utente non ha appuntamenti prenotati, il sistema mostra un messaggio di avviso. [17] Test UT-03-02*

Recommendation: Per tutti i casi d'uso che interagiscono con il database o con logica transazionale (almeno UC-03, UC-04, UC-05, UC-06, UC-08, UC-09, UC-10, UC-11): - Aggiungi un flusso alternativo generico di errore di sistema, ad esempio dopo il passo in cui il sistema accede al DB ("Il sistema registra l'appuntamento", "Il sistema rimuove l'appuntamento", "Il sistema mostra l'elenco..."). Descrivi che cosa succede se l'operazione fallisce: nessuna modifica ai dati, messaggio di errore all'utente, eventuale log. - Per UC-04 e UC-05, specifica che la prenotazione/cancellazione è atomica: in caso di fallimento di una delle operazioni (es. aggiornamento Available_Slots, inserimento in News), l'intera transazione viene annullata oppure viene garantita una coerenza minima (es. appuntamento creato ma notifica non inviata, con messaggio all'utente). - Se non implementi una vera transazione DB, almeno documenta il comportamento effettivo (ordine delle operazioni e possibili stati intermedi) così che il docente possa valutare consapevolmente le scelte.

See also: TST-002

ISS-005 — MEDIUM [100%] — Page 10 Il Use Case Diagram non è pienamente coerente con i template dei casi d'uso: l'attore è indicato come "Client" e "Administrator", mentre nei template si usano "Customer" e "Barber"; inoltre il caso d'uso "Recover Password" appare nel diagramma ma non ha un template dedicato, e le relazioni «include»/«extend» (es. View Appointments includes Sign in, Delete Appointment extends Add Appointment) non sono esplicitate nei testi dei casi d'uso, rendendo difficile capire le dipendenze operative.

> *[IMAGE 1]: Diagram Type: Use Case Diagram Elements: - #01 Sign up - #02 Sign in - Recover Password - #03 View Appointments - #04 Add Appointment - #05 Delete Appointment - #06 View News - #07 View Profile - #08 View Services - #09 Add Service - #10 Delete Service - #11 Send Communication Relationships: - Sign in extends to Recover Password - View Appointments includes Sign in - Add Appointment includes Sign in - Delete Appointment extends Add Appointment - View News includes Sign in - View Profile includes Sign in - View Services includes Sign in - Add Service includes Sign in - Delete Service extends Add Service - Send Communication includes Sign in Text: - «function» #02 Sign in - «extend» Recover Password - «extend» #05 Delete Appointment - «extend» #10 Delete Service Actors: - Client - Administrator*

Recommendation: Allinea il diagramma dei casi d'uso con i template: - Rinomina gli attori nel diagramma da "Client" a "Customer" e da "Administrator" a "Barber" (o viceversa nei template), in modo che la terminologia sia uniforme in tutto il documento. - Aggiungi un Use Case Template per "Recover Password" coerente con la relazione «extend» da Sign in, specificando pre-condizioni, post-condizioni, flusso base (inserimento email, invio mail/reset) e flussi alternativi (email non registrata, errori). - Nei template dei casi d'uso che nel diagramma hanno relazioni «include»/«extend» (es. UC-03, UC-04, UC-05, UC-06, UC-07, UC-08, UC-09, UC-10, UC-11), aggiungi una sezione o una frase esplicita che indichi che richiedono l'esecuzione di UC-02 Sign in, e per i casi «extend» chiarisci in quale passo del flusso base si innesta l'estensione.

See also: TST-002

ISS-010 — LOW [100%] — Page 50 La tracciabilità tra casi d'uso e test è in generale buona (ogni UC elenca i test associati e viceversa), ma non è completa: ad esempio UC-03, UC-06, UC-08 hanno solo test unitari o di integrazione elencati (UT-03-02, UT-06-02, IT-08-02) e non test funzionali espliciti, mentre nella sezione 4.4 sono presenti test funzionali per molti controller ma non sempre è chiaro a quale UC corrispondano (es. AppointmentBarberControllerTest, NewsCustomerControllerTest, NewsBarberControllerTest non riportano tra parentesi il riferimento UC). Questo rende più difficile per il lettore verificare rapidamente che ogni requisito sia coperto da almeno un test end-to-end.

> *Ogni metodo è associato a uno use case template, con il riferimento specificato accanto al nome della classe a cui appartiene.*

Recommendation: Completa e rendi più esplicita la matrice di tracciabilità requisiti–test: - Per ogni UC (UC-01..UC-11), verifica che ci sia almeno un test funzionale chiaramente etichettato nella sezione 4.4; se un controller test copre più UC, specifica tra parentesi tutti i codici UC. - Aggiorna il campo "Test" nei template dei casi d'uso per includere anche i test funzionali mancanti (es. aggiungi eventuali FT-* per UC-03, UC-06, UC-08 se esistono; in caso contrario, valuta di aggiungere almeno un test funzionale per ciascuno). - Se il documento diventa lungo, puoi aggiungere una piccola tabella riassuntiva (UC vs UT/IT/FT) per mostrare a colpo d'occhio che non ci sono requisiti critici senza test.
See also: TST-002

UC-06

ISS-004 — MEDIUM [100%] — Page 4 I requisiti funzionali di alto livello non sono tracciati né raffinati nei casi d'uso: mancano casi d'uso espliciti per il recupero password, per la gestione completa delle notifiche/news (inclusa la generazione automatica di notifiche su prenotazione/cancellazione) e per il logout. Alcune funzionalità descritte qui sono solo implicite nei template dei casi d'uso o nella sezione Database, senza un requisito/caso d'uso dedicato, rendendo difficile la verifica sistematica.

> *Le principali funzionalità che il sitema soddisfa sono: • Registrazione e accesso: Gli utenti possono registrarsi come barbieri o clienti, accedere all'applicazione e, se necessario, recuperare la password. • Visualizzazione appuntamenti: I clienti possono visualizzare gli appuntamenti preno- tati e cancellarli, mentre i barbieri possono visualizzare gli appuntamenti prenotati dai clienti. • Prenotazione appuntamenti: I clienti possono prenotare appuntamenti selezionando data, orario, barbiere, servizio e metodo di pagamento. • Notifiche: I clienti ricevono notifiche sugli slot orari tornati disponibili e sulle comu- nicazioni inviate dai barbieri. I barbieri ricevono notifiche quando un cliente prenota o cancella un appuntamento. • Visualizzazione profilo: Gli utenti possono visualizzare le informazioni del proprio profilo e fare logout.*

Recommendation: Aggiungi o esplicita i casi d'uso mancanti per coprire tutti i punti dei requisiti funzionali: - Crea un UC separato per il recupero password (attualmente solo come estensione di Sign in) con pre/post-condizioni, flusso base e flussi alternativi (email non registrata, errori di invio, ecc.). - Introduci un UC per il logout (richiamato da "fare logout" in Visualizzazione profilo) con pre-condizione utente autenticato e post-condizione sessione chiusa e ritorno alla schermata di Sign in. - Per le notifiche, separa chiaramente: (a) UC per la visualizzazione delle news/notifiche (già UC-06) e (b) UC o almeno requisiti funzionali testabili per la generazione automatica di notifiche su prenotazione/cancellazione e slot liberati, allineandoli con quanto descritto nella sezione Database (News e trigger). Collega ogni nuovo UC ai test esistenti o aggiungi test funzionali/integrati specifici.
See also: TST-003

ISS-008 — MEDIUM [100%] — Page 16 UC-06 View News tratta tutte le "news" come un unico tipo di comunicazione, ma nel modello dati e nel testo introduttivo le news includono sia comunicazioni manuali inviate dal barbiere sia notifiche automatiche (prenotazioni, cancellazioni, slot disponibili). Il caso d'uso non specifica se e come questi tipi diversi vengono filtrati o presentati (es. news solo per clienti, news solo per barbieri, priorità, limite massimo di news come gestito da NewsService.deleteOldestNewsIfNecessary). Questo crea una leggera incoerenza tra logica di business implementata e descrizione dei requisiti.

> *2.2.6 View News UC-06 View News Scope User Goal Attori Customer, Barber Pre-condizioni L'utente è autenticato nel sistema. Post-condizioni L'utente ha visualizzato le news. Flusso base 1. L'utente apre la schermata delle news. [26] 2. Il sistema mostra l'elenco delle comunicazioni ricevute a partire dalla più recente. Flusso alternativo 2.1 Se l'utente non ha ricevuto comunicazioni, il sistema mostra un messaggio di avviso. [27] Test UT-06-02*

Recommendation: Chiarisci in UC-06 la semantica delle news: - Specifica che tipo di comunicazioni vengono mostrate (es. tutte le righe della tabella News, sia manuali che automatiche) e se la vista è filtrata per ruolo (Customer vs Barber) come suggerito dai metodi getAllBarberNews/getAllCustomerNews. - Se esiste un limite massimo di news (gestito da deleteOldestNewsIfNecessary), aggiungi una nota nel caso d'uso o nelle post-condizioni per indicare che l'utente vede solo le N news più recenti. - Se sono previste future estensioni (es. filtri per tipo di notifica), puoi aggiungere un breve commento per mostrare che la progettazione le considera.
See also: TST-003

UC-04

ISS-006 — MEDIUM [100%] — Page 14 Il caso d'uso UC-04 Add Appointment non è completamente coerente con il modello dei dati e con il comportamento implementato: nel Domain Model e nel modello relazionale un appuntamento può avere più servizi (tabella Appointment_Services, attributo serviceTypes in Appointment), ma il flusso base e i flussi alternativi non chiariscono come vengono gestiti uno o più servizi (es. cosa succede se l'utente aggiunge più di un servizio, come vengono mostrati nel riepilogo). Inoltre, la post-condizione "Il customer ha prenotato un appuntamento" è troppo generica e non specifica che lo slot è stato rimosso da Available_Slots e che sono state generate le notifiche previste.

> *2.2.4 Add Appointment UC-04 Add Appointment Scope User Goal Attori Customer Pre-condizioni Il customer è autenticato nel sistema. Post-condizioni Il customer ha prenotato un appuntamento. Flusso base 1. Il customer apre la schermata di prenotazione. [18] 2. Il customer seleziona la data. 3. Il customer seleziona il barbiere e il servizio. [19] 4. Il sistema chiede se vuole aggiungere un altro servizio. [21] 5. Il customer lo aggiunge e preme "yes". 6. Il sistema mostra gli slot orari disponibili. [19] 7. Il customer seleziona uno slot orario. 8. Il sistema mostra un riepilogo dell'appuntamento. [23] 9. Il customer conferma e seleziona il metodo di pagamento. [24] 10. Il sistema registra l'appuntamento e invia una notifica al barber. [25] Flusso alternativo 4.1 Se il customer non seleziona il servizio, il sistema mostra un messaggio di errore. [20] 5.1 Il customer non aggiunge un altro servizio e preme "no". 8.1 Se il customer seleziona no alla conferma, ritorna alla scelta degli slot orari. [19] Test UT-04-6, IT-04, FT-04, FT-04-02, FT-04-04.1*

Recommendation: Raffina UC-04 per allinearlo al modello dati e all'implementazione: - Esplicita nel flusso base come viene gestita la selezione di più servizi: ad esempio, ripetizione del passo 4–5 finché l'utente risponde "yes", e nel riepilogo (passo 8) mostra l'elenco completo dei servizi e il prezzo totale. - Aggiorna la post-condizione per riflettere gli effetti sul sistema: ad esempio "È stato creato un nuovo Appointment associato a uno o più ServiceType, lo slot selezionato è stato rimosso da Available_Slots e il barbiere ha ricevuto una notifica". - Se alcune di queste azioni (rimozione slot, creazione News) sono opzionali o implementate tramite trigger DB, specifica comunque nel caso d'uso che fanno parte del risultato osservabile dal punto di vista utente (es. comparsa di una news).

UC-05

ISS-007 — MEDIUM [100%] — Page 15 UC-05 Delete Appointment non specifica chiaramente come viene gestita la riapertura dello slot e la generazione delle notifiche, nonostante il database e i test di integrazione mostrino una logica articolata (reintroduzione in Available_Slots, creazione di News per barber e clienti). Inoltre, la post-condizione è troppo sintetica e non distingue tra il punto di vista del customer (appuntamento non più visibile) e gli effetti sul sistema (slot nuovamente prenotabile, notifiche inviate).

> *2.2.5 Delete Appointment UC-05 Delete Appointment Scope User Goal Attori Customer Pre-condizioni Il customer è autenticato nel sistema. Post-condizioni Il customer ha cancellato un appuntamento. Flusso base 1. Il customer apre la schermata degli appuntamenti. [15] 2. Il customer clicca l'icona del cestino accanto all'appuntamento da cancellare. 3. Il sistema chiede conferma per la cancellazione. [16] 4. Il customer conferma la cancellazione. 5. Il sistema rimuove l'appuntamento e notifica il barber e gli altri customers che lo slot è disponibile. Flusso alternativo 4.1 Se il customer non conferma la cancellazione, l'appuntamento rimane nell'elenco.*

Recommendation: Estendi UC-05 per descrivere meglio la logica di business: - Nel flusso base, dopo il passo 5, aggiungi passi espliciti per: (a) reintroduzione dello slot in Available_Slots, (b) creazione delle notifiche di tipo "Slot available" per i clienti interessati e per il barbiere, in coerenza con quanto mostrato in AppointmentAndSlotsTest. - Raffina la post-condizione in qualcosa come: "L'appuntamento è stato rimosso, lo slot corrispondente è tornato disponibile per la prenotazione e sono state inviate le notifiche di disponibilità". - Se la notifica è visibile nella sezione News (UC-06), aggiungi un riferimento incrociato per chiarire il legame tra i due casi d'uso.

UC-01

ISS-009 — LOW [100%] — Page 11 Nei template dei casi d'uso, il campo "Scope" è usato in modo non uniforme (es. "User Goal" per UC-01, UC-03, UC-04, ecc. e "Function" per UC-02) e non chiarisce il confine del sistema o del sottosistema coinvolto. Inoltre, il campo "Attori" usa talvolta il generico "User" (UC-01, UC-02) e talvolta ruoli specifici (Customer, Barber), mentre nel resto del documento i ruoli sono ben distinti. Questo non compromette la correttezza, ma può penalizzare la valutazione formale della documentazione.

> *2.2.1 Sign up UC-01 Sign up Scope User Goal Attori User Pre-condizioni L'utente non è autenticato nel sistema e non ha un account. Post-condizioni L'utente è registrato nel sistema e può fare l'accesso Flusso base 1. L'utente apre la schermata di registrazione. [7] 2. L'utente inserisce le informazioni richieste e se è un barbiere inserisce il codice segreto. 3. L'utente preme il pulsante di registrazione. 4. Il sistema verifica le informazioni inserite. 5. Il sistema crea l'account dell'utente. 6. L'utente viene reindirizzato alla schermata di accesso. [10] Flusso alternativo 5.1 Se le informazioni inserite non sono valide, il sistema mostra un messaggio di errore. [8, 9] Test UT-01-05, UT-01-05.1, IT-01, FT-01, FT-01-05.1*

Recommendation: Uniforma la struttura dei template dei casi d'uso: - Usa un valore coerente per "Scope" (ad esempio sempre "System" o il nome del sistema) oppure rimuovi il campo se non lo utilizzi in modo significativo. - Sostituisci "User" con i ruoli effettivi dove possibile: per UC-01 e UC-02 puoi indicare "Customer, Barber" se entrambi possono registrarsi/accedere, mantenendo coerenza con la sezione 2.1. - Verifica che tutti i template abbiano la stessa struttura (Scope, Attori, Pre-condizioni, Post-condizioni, Flusso base, Flusso alternativo, Test) per dare un'impressione di maggiore rigore metodologico.

6.3 Testing (4 issues)

UC-03

TST-002 — MEDIUM [66%] — Page 13 For UC-03 View Appointments, only a single unit test (UT-03-02) is listed and it targets the service layer. There is no explicit integration or functional test mapped to this use case, unlike most other UCs, so the end-to-end behavior of viewing appointments (for both Customer and Barber) and the alternative flow with no appointments is only indirectly covered via other tests (e.g., AppointmentCustomerControllerTest, AppointmentBarberControllerTest) and not traceably linked to UC-03.

Test UT-03-02

Recommendation: Extend the traceability and coverage for UC-03 by: (1) explicitly mapping existing controller-level functional tests that verify the appointments tables for Customer and Barber to UC-03 in the "Test" row of the UC-03 template; and (2) if such tests do not yet assert the alternative flow, add at least one functional test per role that starts from sign in, navigates to the appointments view, and verifies both the presence of appointments and the "There are no appointments!" message when the list is empty. Name them consistently (e.g., FT-03, FT-03-02.1) and reference them in the UC-03 table.
See also: ISS-003, ISS-005, ISS-010

UC-06

TST-003 — MEDIUM [67%] — Page 16 For UC-06 View News, only a unit test on the service layer (UT-06-02: getNewsCustomer) is mapped. There is no integration or functional test explicitly associated with this use case, and the alternative flow "Se l'utente non ha ricevuto comunicazioni, il sistema mostra un messaggio di avviso. [27]" is not covered by any listed test ID. This creates a gap between the functional requirement (including the empty state message) and the documented tests.

Test UT-06-02

Recommendation: Add at least one functional test that exercises the NewsCustomerController and NewsBarberController through the UI: start from a clean H2 test database with no News rows for the current user, navigate to the News view, and assert that the "There are no news!" message is displayed. Map these tests to UC-06 in the "Test" row (e.g., FT-06, FT-06-02.1) and, if you already have News*ControllerTest classes, update the UC-06 template to reference the corresponding test IDs so that traceability is explicit.
See also: ISS-004, ISS-008

UC-08

TST-004 — LOW [65%] — Page 17 For UC-08 View Services, only an integration test (IT-08-02: testGetServices) is mapped, and the alternative flow "Se il barber non ha servizi disponibili, il sistema mostra un messaggio di avviso. [33]" has no corresponding test ID. Similarly, for UC-03 and UC-06, the "no data" alternative flows are not explicitly linked to tests. This indicates a systemic pattern where empty-state/"no items" messages are not clearly covered by functional tests or traceability.

Test IT-08-02

Recommendation: For all use cases that define an alternative flow for empty data sets (UC-03, UC-06, UC-08), add or document functional tests that verify the correct warning/empty-state messages in the UI (e.g., "There are no appointments!", "There are no news!", "There are no services!"). Use TestFX to navigate to the relevant views with an empty table and assert the presence of the message. Then, update each UC template "Test" row to include the new test IDs (e.g., FT-03-02.1, FT-06-02.1, FT-08-02.1) so the coverage of these alternative flows is explicit.

General Issues

TST-001 — HIGH [100%] — Page 52 Integration tests are executed directly against the real PostgreSQL database, while functional tests use an in-memory H2 database. This dual-DB strategy is reasonable, but the document does not describe any mechanism to ensure that the PostgreSQL schema, triggers (e.g., create_slots_for_new_barber), and initial data used in integration tests are aligned with the H2 schema and data used in functional tests. This can lead to subtle inconsistencies where a scenario passes on H2 but fails on PostgreSQL (or vice versa), especially around triggers, constraints, and SQL dialect differences.

> *Inoltre, è stato scelto di utilizzare il database reale PostgreSQL per verificare che le operazioni di persistenza funzionino correttamente.*

Recommendation: Document and, if necessary, adjust your test setup so that the PostgreSQL integration-test schema and the H2 functional-test schema are generated from the same SQL scripts. In practice: (1) extract the DDL and trigger definitions (e.g., for Available_Slots and create_slots_for_new_barber_function) into reusable SQL files; (2) ensure DBTestInitializer (and any PostgreSQL test setup) executes the same scripts; and (3) add a brief note in the Test chapter explaining how you keep H2 and PostgreSQL schemas synchronized. Optionally, add a small integration test that verifies the trigger behavior on PostgreSQL (e.g., that inserting a BARBER user creates the expected Available_Slots), mirroring the assumptions used in functional tests.

7 Priority Recommendations

The following actions are considered priority:

1. **ISS-003** (p. 13): Per tutti i casi d'uso che interagiscono con il database o con logica transazionale (almeno UC-03, UC-04, UC-05, UC-06, UC-08, UC-09, UC-10, UC-11): -...
2. **TST-001** (p. 52): Document and, if necessary, adjust your test setup so that the PostgreSQL integration-test schema and the H2 functional-test schema are generated from...

8 Traceability Matrix

Of 11 traced use cases: 11 fully covered, 0 without design, 0 without test.

ID	Use Case	Design	Test	Gap
UC-01	Sign up	✓	✓	—
UC-02	Sign in	✓	✓	—
UC-03	View Appointments	✓	✓	—
UC-04	Add Appointment	✓	✓	—
UC-05	Delete Appointment	✓	✓	—
UC-06	View News	✓	✓	—
UC-07	View Profile	✓	✓	—
UC-08	View Services	✓	✓	—
UC-09	Add Service	✓	✓	—
UC-10	Delete Service	✓	✓	—

ID	Use Case	Design	Test	Gap
UC-11	Send Comunication	✓	✓	—

9 Terminological Consistency

Found **10** terminological inconsistencies (3 major, 7 minor).

Group	Variants found	Severity	Suggestion
End-user role naming	"Customer", "Client", "cliente", "customer"	MAJOR	Use a single term for the end-user role across the whole document (e.g., "Customer" in English sections and avoid mixing with "Client" / "cliente"). Align the Use Case Diagram actor (currently "Client") and all textual descriptions (which use "Customer" / "cliente").
Barber role naming	"Barber", "barbiere", "barber"	MINOR	Choose one primary term for the role (e.g., "Barber") and use it consistently in English contexts; if Italian is preferred in a section, keep it consistently as "barbiere" and avoid switching within the same conceptual description.
Appointment vs prenotazione	"Appointment", "Appuntamento", "prenotazione", "New Appointment"	MINOR	Standardize on one language per section for this concept. For English UI and use cases, consistently use "Appointment" (e.g., "Book Appointment", "Appointments"), and avoid mixing with "Appuntamenti" / "prenotazione" in the same conceptual descriptions.
Communication / News / Notification	"News", "Send Comunication", "Send communication", "SendComunication", "comunicazioni", "comunicazione", "notifiche", "Notification"	MAJOR	Clarify and standardize terminology: if "News" is the user-facing section that lists all messages/notifications, use a single term (e.g., "News" or "Notifications") and align it with the domain class "Notification" and use case "Send Comunication" / "Send communication". Also fix the spelling to a single form (e.g., "Send Communication").

Group	Variants found	Severity	Suggestion
Service / Service Type	"Service", "Services", "Service Type", "Service-Type", "Service_Types", "Service Types", "Service Types Service"	MINOR	Use a consistent term for the business concept of a barber's offering. For example, use "ServiceType" (class), "Service_Types" (table) and always refer to them in text as "service type" (singular) / "service types" (plural), avoiding alternation with plain "Service" when the meaning is the same.
Available Slot naming	"Available Slot", "Available_Slot", "Available_Slots", "AvailableSlot", "slot orari", "slot"	MINOR	Align the naming of this concept: use one canonical English term (e.g., "AvailableSlot" for class, "Available_Slots" for table) and refer to it in prose as "available slot" / "available slots". Avoid mixing with Italian "slot orari" in the same technical explanations.
User entity naming	"User", "Users", "User-DAO", "utente", "utenti"	MINOR	Keep the domain/entity name consistent as "User" (class) and "Users" (table) and refer to the concept in text as "user" / "users" in English sections. Avoid alternating with "utente" / "utenti" in the same technical description to reduce ambiguity.
Payment method naming	"Payment", "Payment-Method", "metodo di pagamento", "metodi di pagamento"	MINOR	Use "PaymentMethod" when referring to the enumeration/type and "payment method" in prose, and keep this consistent instead of alternating with Italian "metodo di pagamento" in the same conceptual context.
Send Communication naming and spelling	"Send Comunication", "Send communication", "SendComunication", "SendCommunicationService", "SendComunicationController"	MAJOR	Unify both spelling and spacing for this feature. Choose a single English form (e.g., "SendCommunication" for class names and "Send Communication" for UI labels) and apply it consistently across controllers, services, diagrams, and menu items.

Group	Variants found	Severity	Suggestion
Database manager / connection naming	"DBConnection", "DBconnection", "DBManager", "DBmanager"	MINOR	Use a single, case-consistent name for the database connection component. For example, keep the class as "DBManager" and the package/section as "DBConnection" and avoid variants like "DBconnection" or "DBmanager" in diagrams and text.