

Software Engineering Audit Report

Elaborato SWE.pdf

CAPRA

February 25, 2026

Contents

1	Document Context	2
2	Executive Summary	3
3	Strengths	4
4	Expected Feature Coverage	4
5	Summary Table	7
6	Issue Details	7
6.1	Architecture (1 issues)	7
6.2	Requirements (8 issues)	8
6.3	Testing (2 issues)	11
7	Priority Recommendations	12
8	Traceability Matrix	13
9	Terminological Consistency	14

1 Document Context

Project Objective

The project is an advanced platform for managing medical visit bookings designed to meet the needs of both patients and doctors (freelancers or healthcare facility staff). The platform provides a dynamic and interactive virtual environment where patients and doctors can connect, collaborate, and organize personalized visits across various medical specializations. It facilitates the meeting of supply and demand for medical visits, simplifying the booking and payment process through a secure online transaction system.

Main Use Cases

- UC-1 – Cerca Visita: Patient searches for medical visits of interest and views search results.
- UC-2 – Prenota Visita: Patient selects an available visit, books it, and pays for the visit.
- UC-3 – Cancella Prenotazione: Patient cancels a booking from the “Le mie prenotazioni” section.
- UC-4 – Visualizza le mie Prenotazioni: Patient views the list of visits they are booked for.
- UC-5 – Crea Visita: Doctor creates a new visit with details such as specialization, time, modality, and fee.
- UC-6 – Cancella Visita: Doctor cancels a visit from the “il mio Calendario” section.
- UC-7 – Modifica Visita: Doctor modifies visit information from the “il mio Calendario” section.
- UC-8 – Visualizza le mie Visite: Doctor views the complete history of their visits regardless of state.

Functional Requirements

- Doctors can create, modify, and delete visit announcements with specialization, time, modality (online or in-person), and fee details.
- Doctors can view published announcements and consult the calendar of visits booked by patients.
- Patients can search for available visit announcements using multiple filter criteria.
- Patients can book visits according to their needs and manage bookings conveniently.
- System prevents doctor from having overlapping visits in the same time slot.
- System prevents patient from booking multiple visits at the same time.
- Visits can be associated with tags for categorization: urgency level, specialty, geographic zone, and modality (online or in-person).
- Advanced search functionality using Decorator pattern with filters for specialty, urgency level, online/in-person mode, geographic zone, date, and price.
- Visit state management with four states: Available, Booked, Cancelled, and Completed.
- Tag management system allowing attachment and detachment of tags from visits.

Non-Functional Requirements

- Architecture based on Java with separation of concerns: `domainModel`, `businessLogic`, and `DAO` packages.
- Data persistence using PostgreSQL database with JDBC connectivity.
- Modular and maintainable code structure following Maven standard directory layout.
- Automated testing using JUnit framework for unit and functional tests.
- Use of design patterns: Decorator pattern for search functionality, State pattern for visit state management, and DAO pattern for data access abstraction.

- Separate test database to avoid conflicts and maintain data integrity during testing.
- Proper validation and error handling with meaningful exception messages.
- Prevention of SQL injection using PreparedStatement in database queries.

Architecture

The system follows a layered architecture with three main components: (1) **domainModel** package containing core classes (**Person**, **Patient**, **Doctor**, **Visit**) and sub-packages for **Search** (using Decorator pattern), **State** (using State pattern), and **Tags** for visit categorization; (2) **businessLogic** package with controller classes (**PeopleController**, **PatientsController**, **DoctorsController**, **VisitsController**, **StateController**, **TagsController**) managing system operations; (3) **DAO** package implementing the Data Access Object pattern with interfaces and PostgreSQL concrete implementations for data persistence. The system uses JDBC for database connectivity and follows the Single Responsibility Principle by separating business logic from data access.

Testing Strategy

Testing is organized in `src/test/java` following Maven conventions with three main test categories: (1) **domainModel tests** verify individual unit behavior including **VisitTest** (visit creation, date validation, tag operations, state management) and **StateTest** (all four visit states); (2) **businessLogic tests** serve as functional tests verifying use case implementations including **VisitsControllerTest** (insert, update, delete, tag attachment/detachment), **DoctorsControllerTest** (add, remove, retrieve doctors), and **VisitsFunctionalTest** (overlapping visit prevention, non-existent doctor validation); (3) **DAO tests** verify correct database persistence and integrity operations. All tests use JUnit framework and have been executed successfully without errors, confirming system robustness and correct functionality implementation.

2 Executive Summary

Quick Overview

Total issues: **11** — **HIGH: 4** **MEDIUM: 5** **LOW: 2**

Average confidence: **98%**

Executive Summary: Audit of Elaborato SWE.pdf

This document is a Software Engineering project report submitted by a university student, detailing the design, implementation, and testing of a medical visit booking system. The report includes use case diagrams, architectural descriptions (with focus on DAO patterns), controller logic, and test coverage claims. The purpose is to demonstrate competency in requirements analysis, system design, and quality assurance practices.

The audit identified **11 issues across three categories**, with a concerning pattern of **misalignment between documented requirements and actual implementation**. The most prevalent issue type is the specification of features (online payments, notifications, refunds, time-window constraints, calendar management) in use case diagrams and templates that are neither implemented nor tested. This creates a fundamental gap between what the document claims the system should do and what it actually does. Additionally, use cases lack sufficient detail in pre-conditions, post-conditions, and alternative flows, making them difficult to validate. A secondary pattern involves incomplete or missing test traceability: while tests are listed by class, there is no explicit mapping between individual use cases and their corresponding test cases, and critical components (notably the **StateController**) lack dedicated test coverage despite implementing complex business logic.

Critical areas (HIGH severity) include: (1) UC-2 and UC-3 describe payment, notifications, and refunds as core behaviors, but these are not implemented or tested, creating contradictions between requirements and reality; (2) UC-3 specifies time-window restrictions for cancellation that are not enforced in the actual code; (3) the use case diagram mixes implemented and non-implemented functionality without clear separation; and (4) the **StateController**, which enforces critical state transitions, has no dedicated test suite despite containing complex validation logic.

Overall Quality Assessment: The document demonstrates partial competency in requirements specification and testing practices, but suffers from significant gaps in requirements-implementation traceability and test coverage. The work shows effort in architectural design and code organization, but the disconnect between documented behavior and actual implementation suggests insufficient review and validation cycles. The document would not meet professional standards for a production system.

Priority Actions:

1. **Reconcile Requirements with Implementation:** Remove or clearly mark as “future work” all features not implemented (payments, notifications, refunds, calendar management, time-window constraints). Rewrite affected use cases (UC-2, UC-3, UC-6, UC-7) to match actual system behavior, or implement the missing features and add corresponding tests.
2. **Create Explicit Traceability Matrix:** Document a mapping from each use case to its test cases, identifying which test classes and methods validate each flow. Identify and document any use case behaviors that lack test coverage, with justification.
3. **Add StateController Test Suite:** Implement comprehensive unit tests for `StateController.deleteBooking()`, `StateController.bookVisit()`, and other state-transition methods, covering both happy paths and error conditions (invalid states, unauthorized users, constraint violations).

3 Strengths

- **Well-structured layered architecture:** The project implements a clear three-tier architecture (businessLogic, domainModel, DAO) with proper separation of concerns. The dependency diagram (Figura 3) demonstrates a clean separation between user interaction, business logic, domain model, and data persistence layers, following the Single Responsibility Principle.
- **Comprehensive use case documentation:** The document provides detailed use case templates for all implemented features (Figura 5-12), including basic courses, alternative courses, pre-conditions, and post-conditions. This thorough specification covers patient operations (search, book, cancel, view) and doctor operations (create, modify, delete, view visits).
- **Appropriate application of design patterns:** The project correctly implements three relevant design patterns: the Decorator pattern for flexible search filtering (Figura 15), the State pattern for managing visit lifecycle states (Figura 17), and the DAO pattern for data persistence abstraction (Figura 18). Each pattern is well-motivated and properly documented.
- **Organized package structure:** The codebase is logically organized into distinct packages (businessLogic, domainModel with sub-packages for search, state, and tags, plus DAO), each with clearly defined responsibilities and purposes, facilitating maintainability and modularity.
- **Testing framework implementation:** The project includes a dedicated testing section (Section 4) utilizing JUnit for automated testing across domainModel, businessLogic, and DAO layers, with documented test results (Sezione 4.4).

4 Expected Feature Coverage

Of 7 expected features: 5 present, 1 partial, 1 absent. Average coverage: 80%.

Feature	Status	Coverage	Evidence
Unit testing framework implementation	Present	80% (4/5)	The document describes JUnit-based tests with assertions (e.g., Assertions.assertTrue, assertThrows, assertEquals), explains a systematic approach to unit and functional testing in section 4, defines dedicated test classes like VisitTest, StateTest, VisitsControllerTest, DoctorsControllerTest, PostgreSQLVisitDAOTest, and DatabaseTest, and mentions both unit tests and functional tests (integration-like tests using a separate test database). However, there is no mention of using mock dependencies; tests rely on a real PostgreSQL test database instead.
Use of UML Diagrams for system modeling	Partial	75% (6/8)	The report includes a use case diagram with actors Paziente and Medico and their use cases (Figura 4), a detailed UML class diagram (Figura 13), and a dependency/architecture diagram showing User, BusinessLogic, DomainModel, DAO, JDBC, RDBMS, PostgreSQL (Figura 3). It explicitly states that UML diagrams were created with StarUML and uses standard UML notation, and the use case diagram shows relationships between actors and use cases. Diagrams and templates are used to clarify functional requirements. However, there are no UI mockups or explicit navigation-flow diagrams of the user interface.
Identification and definition of system actors	Present	100% (8/8)	Section 2.1 and the use case diagram clearly identify user roles Paziente and Medico and describe their responsibilities (e.g., patients search, book, cancel, view prenotazioni; doctors create, modify, cancel, view visits). The templates dei casi d'uso define interactions between users and the system with specific user actions and flows, and the text explains how these interactions support user requirements and delineate functionalities for each role.

Feature	Status	Coverage	Evidence
Definition and Documentation of Use Cases	Present	100% (5/5)	Use case templates (Use case 1–8 such as Cerca visita, Prenota visita, Cancellazione prenotazione, Crea visita, Modifica visita) define specific user interactions and user goals, include basic and alternative courses, and often specify pre-conditions and post-conditions. The use case diagram shows include relationships between use cases (e.g., Prenota visita includes Controllo fascia oraria and Paga online), documenting relationships between use cases and subcases.
User interface and interaction design principles	Absent	29% (2/7)	The document defines user roles and their functionalities and describes step-by-step logical flows for reservation-related use cases (e.g., Prenota visita basic course). However, it does not present UI mockups, screen layouts, or explicit navigation structures, nor does it describe validation mechanisms or dynamic UI updates; interaction is described at the logical/use-case level rather than interface design.
Separation of concerns in software architecture	Present	80% (4/5)	The architecture section and project structure (Figura 3 and Figura 21) show distinct packages: domain-Model (model), businessLogic (controllers/services), and dao (data access). Interactions between controllers and domain models and DAOs are described (e.g., VisitsController uses VisitDAO and TagDAO), and the roles of each layer are clearly explained, emphasizing modularity and maintainability. There is no explicit View package, but the separation among Model, Controller/Service, and DAO is well documented.

Feature	Status	Coverage	Evidence
Data Access Object (DAO) pattern	Present	100% (7/7)	Section 2.3.3 and Figura 18 define a generic DAO interface with CRUD methods (get, getAll, insert, update, delete) and entity-specific DAOs (PatientDAO, VisitDAO, DoctorDAO, TagDAO). SQL queries are encapsulated in concrete DAO classes like PostgreSQLVisitDAO, which use PreparedStatement and Database.getConnection(). The text explains DAO as an abstraction layer separating business logic from database access, and section 4.3 describes tests such as PostgreSQLVisitDAOTest.testDeleteVisit() that verify DAO methods.

5 Summary Table

Category	HIGH	MEDIUM	LOW	Total
Architecture	0	0	1	1
Requirements	3	4	1	8
Testing	1	1	0	2
Total	4	5	2	11

6 Issue Details

6.1 Architecture (1 issues)

ISS-001 — LOW [87%] — Page 18 There is a minor inconsistency between the described architecture and the actual implementation of the DAO layer: the text states that each entity has a specific DAO interface extending the generic DAO interface, but TagDAO is explicitly described as not extending DAO. This is coherent in the detailed explanation but slightly contradicts the earlier general statement and could confuse a reader about the uniformity of the DAO design.

Nel progetto, come illustrato nella figura 18, 'e stata definita un'interfaccia generica DAO contenente le operazioni CRUD che le classi DAO concrete devono implementare per gestire la persistenza. Ogni entita' del sistema 'e associata a un DAO specifico, il quale 'e rappresentato da un'interfaccia (es. PatientDAO, VisitDAO, DoctorDAO e TagDAO). Queste interfacce estendono l'interfaccia generica DAO e specificano i tipi di dati con cui le classi DAO concrete dovranno lavorare. ... Poich'è l'interfaccia DAO fornisce operazioni standard come get, getAll, insert, update e delete, non tutte queste operazioni sono rilevanti per l'entita' Tag. Pertanto, non 'e stata estesa direttamente l'interfaccia base DAO in TagDAO, mentendo cos'i le interfacce specifiche alle entita' focalizzate e snelle.

Recommendation: Clarify the description of the DAO architecture to avoid confusion. For example, adjust the text to say that "most" entity DAOs extend the generic DAO interface, and explicitly call out TagDAO as a deliberate exception because CRUD is not fully applicable. Optionally, add a short note in the class diagram or in the text explaining why TagDAO does not

extend DAO and how this still respects SRP. This will make the architectural rationale clearer to the reader and to examiners.

6.2 Requirements (8 issues)

ISS-002 — **HIGH** [100%] — Page 5 Use case diagram mixes implemented and non-implemented functionality (recensioni, pagamenti, notifiche) without a clear separation. Some of these non-implemented behaviors are nevertheless referenced as post-conditions in implemented use cases (e.g., notifications, refunds, online payment), creating contradictions between requirements and what the system actually does and what is tested.

Nel diagramma dei casi d'uso (Figura 4), sono state rappresentate le interazioni tra gli attori e il sistema. È importante notare che, sebbene i casi d'uso relativi alle recensioni, ai pagamenti e alle notifiche siano stati inseriti nel diagramma, al momento non sono stati implementati nella 5 versione attuale del gestionale, ma sono stati progettati come possibili estensioni future del sistema.

Recommendation: Introduce a clear distinction between implemented and future use cases. Concretely: 1) In the use case diagram, visually separate or mark as «future» all non-implemented use cases (Scrivi recensioni al medico, Visualizza le mie recensioni, Paga online, Notifica Medico, Notifica paziente, Rimborso, and any other notification/payment/review UC). 2) In each textual use case template (UC-2 Prenota visita, UC-3 Cancella prenotazione, UC-6 Cancella visita, UC-7 Modifica visita), remove or explicitly mark as "non implemented in this version" all steps and post-conditions that depend on payments, refunds, or notifications. 3) Add a short subsection in the requirements chapter listing these as "future extensions" with a brief description, so the examiner can clearly see what is in scope for this project and what is not.

ISS-003 — **HIGH** [100%] — Page 6 UC-2 Prenota visita describes payment, notification to the doctor, and calendar updates as part of the basic flow and post-conditions, but the implementation and tests only cover creation of a Visit and state changes, not online payment or notifications. There is no pre-condition about the visit being in state "Available" or about payment success, and no alternative/error flows for payment failure or notification failure. This makes the requirement inconsistent with the actual business logic and untestable as written.

Use case 2 - Prenota visita - Description - Il paziente visualizza la visita e si prenota - Level - User goal - Actors - Paziente - Basic course 1. Il paziente seleziona una visita disponibile 2. Il paziente prenota 3. Il paziente paga la visita - Alternative course 1. Il paziente non può prenotarsi perché ha un'altra visita in quell'orario - Post-conditions 1. Il medico viene notificato dell'avvenuta prenotazione. 2. La visita appena prenotata appare nella sezione "Le mie prenotazioni" del paziente. 3. La visita appena prenotata appare nel calendario del medico 4. La visita non appare più nelle ricerche della piattaforma e il suo stato passa a "Prenotato"

Recommendation: Refine UC-2 so that it matches what you actually implemented and tested: 1) Add explicit pre-conditions, e.g. "Esiste una visita nello stato 'Available'" and "Il paziente è autenticato". 2) If online payment is not implemented, remove step 3 "Il paziente paga la visita" and any payment-related post-conditions, or clearly mark them as future work. 3) Replace "Il medico viene notificato" and calendar updates with what the system really does (e.g., "Lo stato della visita passa a 'Booked'" and "La visita è associata al paziente nel database"). 4) Add alternative flows for key error cases you do handle in code/tests, such as: visit not found, doctor not found, visit not in 'Available' state, overlapping booking rejected. 5) Ensure there is at least one functional test that follows the final UC-2 main flow end-to-end (from selecting an available visit to having it in the patient's booked list).

ISS-004 — **HIGH** [100%] — Page 7 UC-3 Cancella prenotazione assumes features (notification to doctor, refund, calendar management, time-window restriction) that are not clearly implemented or tested. The only described alternative flow is "troppo vicina alla data", but the provided `StateController.deleteBooking()` logic checks only state and patient identity, not time constraints, and there is no mention of refunds or notifications in the business logic or tests. This creates a strong mismatch between the specified business behavior and the actual code/tests.

Use case 3 Cancella prenotazione Description Il paziente cancella la prenotazione ad una visita. Level User goal Actors Paziente Basic course 1. Il paziente va nella sezione "Le mie prenotazioni". 2. Il paziente seleziona la prenotazione desiderata. 3. Il paziente cancella la prenotazione. 4. Il paziente viene reindirizzato alla sezione "Le mie prenotazioni". Alternative course 3. Il paziente non può cancellare la visita perché troppo vicina alla data di essa. Pre-conditions Deve esistere almeno una visita alla quale il paziente è prenotato. Post-conditions 1. Il medico viene notificato dell'avvenuta cancellazione 2. Il paziente viene rimborsato 3. La prenotazione scompare dalla sezione "Le mie prenotazioni" del paziente. 4. La visita scompare dal calendario del medico e il suo stato torna a "Disponibile"

Recommendation: Align UC-3 with the implemented `StateController` and tests: 1) Update pre-conditions to include that the visit exists, is in state 'Booked', and is booked by the requesting patient. 2) If you do not implement time-window restrictions, remove or clearly mark the "troppo vicina alla data" alternative as future work; otherwise, specify the exact rule (e.g., "non cancellabile nelle 24 ore precedenti") and implement and test it. 3) Remove or mark as future the refund and notification post-conditions if they are not implemented; instead, state what actually happens (state change to 'Available', removal of association with patient, etc.). 4) Add at least one functional test that exercises the full successful cancellation flow and one that covers the main error condition you actually check (wrong patient, wrong state).

ISS-005 — **MEDIUM** [100%] — Page 8 Several use cases (UC-1 Cerca visita, UC-4 Visualizza le mie prenotazioni, UC-8 Visualizza le mie visite) lack explicit pre-conditions and detailed post-conditions. They only describe a minimal main flow and a single alternative, without specifying what data is shown, how it is ordered, what happens in case of database errors, or what the system guarantees after completion. This makes them underspecified and hard to validate against tests.

Use case 4 - Visualizza le mie prenotazioni - Description - Il paziente visualizza le visite a cui è prenotato. - Level - User goal - Actors - Paziente - Basic course 1. Il paziente va nella sezione "Le mie prenotazioni". 2. Viene mostrato l'elenco di prenotazioni. - Alternative course 2. Non è presente nessuna prenotazione.

Recommendation: For all "visualization" and simple query use cases (UC-1, UC-4, UC-8): 1) Add clear pre-conditions, e.g., "Il paziente è autenticato" or "Il medico è autenticato". 2) Add post-conditions that describe the result more precisely: what list is returned, which fields are shown, whether only future visits or also past ones are included, and what happens when there are zero results (e.g., "il sistema mostra un messaggio 'Nessuna prenotazione trovata'"). 3) For UC-1, specify how filters are applied (AND/OR), whether only 'Available' visits are shown, and how tags affect the search. 4) Optionally, add a short note on error handling (e.g., "In caso di errore di connessione al database, il sistema mostra un messaggio di errore generico"). Then, check that your existing tests (e.g., search and `getBookedVisits` tests) actually correspond to these clarified behaviors.

ISS-006 — **MEDIUM** [100%] — Page 8 UC-5 Crea visita is underspecified and partially inconsistent: it mentions "la lezione" instead of "la visita" in the post-condition, and the alternative flow "stessi dati" is vague (which fields must match?). The business logic and tests for `addVisit()` actually enforce a different rule (no overlapping visits for the same doctor in the same time range), not "same data". There is no pre-condition about the doctor existing or being authenticated.

Use case 5 / Crea visita - Description / Il medico crea una nuova visita. - Level / User goal - Actors / Medico - Basic course 1. Il medico inserisce informazioni riguardanti la visita da creare. 2. La visita viene creata. - Alternative course 2. La visita non viene creata perché ne esiste un'altra con gli stessi dati. - Post-conditions 1. La lezione dev'essere correttamente mostrata come risultato di ricerca. 2. La visita ha stato "Disponibile".

Recommendation: Clarify and correct UC-5 so it matches `VisitsController.addVisit()` and `VisitsFunctionalTest`: 1) Fix the typo in post-condition 1 ("La visita dev'essere correttamente mostrata come risultato di ricerca"). 2) Replace the vague alternative "stessi dati" with the actual rule you implement: e.g., "La visita non viene creata perché il medico ha già un'altra visita sovrapposta nello stesso intervallo orario". 3) Add pre-conditions: "Il medico esiste nel sistema" and "Il medico è autenticato". 4) Add a post-condition that the visit is stored with state 'Available' and associated to the correct doctor. 5) Explicitly reference the existing functional tests (e.g., `testDoctorCannotHaveOverlappingVisits`, `testCreateVisitWithNonExistentDoctor`) as coverage for this UC, so the mapping between requirement and tests is clear.

ISS-007 — MEDIUM [93%] — Page 9 UC-6 Cancella visita and UC-7 Modifica visita both describe time-window restrictions ("troppo vicina"), notifications, refunds, and calendar updates, but the `StateController` description and tests do not mention any doctor-initiated cancellation/modification logic or time-based constraints. There is also inconsistent terminology ("lezione" vs "visita", "studente" vs "paziente" in UC-7), suggesting copy-paste and making the business rules unclear.

Use case 6 / Cancella visita ... Alternative course 3. Il medico non può cancellarla perché troppo vicina ad essa. Pre-conditions Il medico deve avere almeno una visita nel calendario. Post-conditions 1. Se la visita è nello stato "Prenotata", allora il paziente viene notificato dell'avvenuta cancellazione e rimborsato. 2. Lo stato della lezione permuta in "Cancellata". 3. La visita non compare più nella sezione "le mie Prenotazioni" del paziente.

Recommendation: For UC-6 and UC-7: 1) Decide which behaviors are actually implemented in this project. If you do not implement time-window rules, refunds, or notifications for doctor-initiated actions, remove those parts from the main and alternative flows or clearly mark them as future extensions. 2) Replace vague phrases like "troppo vicina" with a precise rule if you intend to support it (e.g., "non cancellabile nelle 24 ore precedenti all'orario di inizio"). 3) Fix terminology: consistently use "visita" and "paziente" instead of "lezione" and "studente". 4) Add pre-conditions that the visit exists, belongs to the doctor, and is in an allowed state for cancellation/modification. 5) If you keep these UCs as implemented, add corresponding tests in businessLogic (e.g., `StateController/VisitsController` tests) that cover successful and failing doctor cancellations/modifications according to the specified rules.

ISS-008 — MEDIUM [100%] — Page 27 The documented behavior of `StateController.deleteBooking()` is inconsistent and contains clear logical/typing errors (reusing variable name 'visit' for a patient, calling `visitsController.getPerson` on `visitsController`, typo 'patieentCF'). More importantly from a requirements perspective, the described logic (state check, patient identity check) is not reflected in any explicit use case alternative flows, and there is no mapping from UC-3 to tests that verify these specific business rules.

Per esempio, nel metodo `deleteBooking()` vengono effettuati vari controlli, descritti nei commenti nello snippet di codice sottostante: Snippet 4: Metodo "deleteBooking()" di `StateController` / Questo metodo gestisce la cancellazione di una prenotazione verificando prima che la visita e il paziente esistano.*/ public void deleteBooking (String patientCF , int idVisit) throws Exception { // Retrieve the visit associated with the given ID. Visit visit = visitsController .getVisit(idVisit); // If the visit does not exist , throw an exception. if (visit == null) { throw new IllegalArgumentException ("The_given_visit_ ID_does_not_exist."); } //*

```

Retrieve the patient associated with the given Fiscal Code. 28 Visit visit = visitsController
.getPerson(patientCF); // If the visit does not exist , throw an exception. if (visit == null) {
throw new IllegalArgumentException ("The_given_patient_does_not_exist."); } // Check if
the visit is in the 'Booked ' state. if (! Objects.equals(visit.getState (), "Booked")) { // If the
visit is not in the 'Booked ' state , cancelation is not allowed. throw new RuntimeException
("The_booking_cannot_be_ canceled_because_it_is_not_in_a_'Booked '_state." ); } // Check
if the patient associated with the booking attempt is the actual booker. if (Objects.equals(visit.
getStateExtraInfo (), patieentCF )) { // If the patient is the actual booker , change the visit
state to 'Available '. Available av = new Available (); this.visitDAO.changeState(idVisit ,
av); } else { // If the visit is not the actual booker , throw an exception. throw new Run-
timeException ("Patient_" + patientCF + "_is_not_booked_for_visit_" + idVisit + ".");
} }

```

Recommendation: Treat deleteBooking() as the concrete implementation of UC-3 and make the link explicit: 1) Fix the code snippet in the document so it compiles logically (separate variables for Visit and Patient, correct controller used, fix typos). This will also make the described business rule clear. 2) Update UC-3 alternative flows to include the two main error conditions you actually check: a) visit not in 'Booked' state, b) patientCF not matching the booked patient. 3) Add or document tests in businessLogic (e.g., a StateControllerTest or functional test) that cover: successful cancellation, wrong state, wrong patient, and non-existent visit/patient. 4) In the testing section, explicitly state that these tests provide coverage for UC-3, so the examiner can see the mapping from requirement to implementation to tests.

ISS-009 — LOW [100%] — Page 31 The document claims that tests verify the main operations specified in the use case diagrams and templates, but there is no explicit traceability matrix or per-use-case mapping. Some critical behaviors described in the use cases (payments, refunds, notifications, time-window constraints) are not covered by tests because they are not implemented. Without clarifying this, the statement overstates the coverage and may confuse the examiner.

I test inclusi nel package domainModel svolgono una duplice funzione: da un lato, verificano il corretto funzionamento delle operazioni principali specificate nei diagrammi dei casi d'uso e nei template dei casi d'uso (come descritto nella sezione 2 del documento); dall'altro, costituiscono i test funzionali del sistema.

Recommendation: Add a short traceability subsection in the Test chapter: 1) Create a simple table mapping each implemented use case (UC-1..UC-8) to the corresponding test classes and key test methods (e.g., UC-5 → VisitsControllerTest.testInsertVisit(), VisitsFunctionalTest.testDoctorCannotHaveOverlappingVisits()). 2) For each use case, explicitly note which parts of the textual description are not implemented and therefore not tested (e.g., "UC-2: pagamento online, notifiche e rimborsi non implementati in questa versione"). 3) Adjust the sentence quoted above to something more precise, such as: "I test coprono le operazioni principali effettivamente implementate per i casi d'uso...", and refer to the new mapping table. This will make your testing strategy more credible and easier to evaluate.

6.3 Testing (2 issues)

UC-1

TST-001 — HIGH [100%] — Page 35 There is no explicit traceability between the implemented use cases (Cerca visita, Prenota visita, Cancella prenotazione, Visualizza le mie prenotazioni, Crea visita, Cancella visita, Modifica visita, Visualizza le mie visite) and the described tests. Tests are listed by class (VisitTest, StateTest, VisitsControllerTest, DoctorsControllerTest, DAO tests, functional tests) but the document never maps which test cases validate each use case flow. This makes it hard for a reviewer to see whether all functional requirements are actually covered by tests.

I test inclusi nel package domainModel svolgono una duplice funzione: da un lato, verificano il corretto funzionamento delle operazioni principali specificate nei diagrammi dei casi d'uso e nei template dei casi d'uso (come descritto nella sezione 2 del documento); dall'altro, costituiscono i test funzionali del sistema. Questi test controllano attentamente una serie di operazioni significative, mirando a coprire gli scenari previsti nell'ambito delle funzionalità dell'applicazione. Ora verranno descritti solo alcuni dei test effettuati, solo quelli fondamentali:

Recommendation: Add a small traceability table that, for each implemented use case (UC1–UC8), lists the corresponding test classes and methods that validate: basic course, alternative course, pre-conditions and post-conditions. For example, map UC2 Prenota visita to specific methods in VisitsControllerTest and VisitsFunctionalTest, and explicitly state which test covers the alternative course (overlapping visit) and which covers the successful booking. This can be a simple table in the Test section referencing both the UC IDs and the test method names.

General Issues

TST-002 — MEDIUM [100%] — Page 27 The StateController, which encapsulates critical state transitions for visits (booking, cancellation, completion, deletion) and enforces business rules, is not covered by any tests. The document shows a complex deleteBooking() method with multiple validations, but there is no corresponding StateControllerTest in the Test section or in the test result screenshots.

La classe StateController gestisce gli stati delle visite, consentendo di prenotare una visita, cancellare una prenotazione, segnare una visita come completata o eliminarla. Interagisce con la classe VisitsController per ottenere e modificare le informazioni sullo stato delle visite. Questa classe gestisce anche la validità delle operazioni, garantendo che un'azione possa essere eseguita solo se la visita è nello stato appropriato. Per esempio, nel metodo deleteBooking() vengono effettuati vari controlli, descritti nei commenti nello snippet di codice sottostante:

Recommendation: Create a dedicated StateControllerTest class that covers at least: successful booking of an Available visit; failed booking when the visit is not Available; successful cancellation when the visit is Booked and the patientCF matches; failed cancellation when the visit is not Booked; failed cancellation when the patientCF does not match; and transitions to Completed and Deleted where applicable. Use the same test database setup as in VisitsControllerTest so you can assert both the returned behavior (exceptions or return values) and the resulting state stored in the database. This will directly validate the business rules encoded in StateController.

7 Priority Recommendations

The following actions are considered priority:

1. **ISS-002** (p. 5): Introduce a clear distinction between implemented and future use cases. Concretely: 1) In the use case diagram, visually separate or mark as «future» ...
2. **ISS-003** (p. 6): Refine UC-2 so that it matches what you actually implemented and tested: 1) Add explicit pre-conditions, e.g.
3. **ISS-004** (p. 7): Align UC-3 with the implemented StateController and tests: 1) Update pre-conditions to include that the visit exists, is in state 'Booked', and is boo...
4. **TST-001** (p. 35): Add a small traceability table that, for each implemented use case (UC1–UC8), lists the corresponding test classes and methods that validate: basic co...

8 Traceability Matrix

Of 15 traced use cases: 7 fully covered, 6 without design, 8 without test.

ID	Use Case	Design	Test	Gap
UC-1	Cerca visita	✓	×	Functional tests that validate the search use case (query composition via decorators and returned visit list) are not do...
UC-2	Prenota visita	✓	✓	—
UC-3	Cancella prenotazione	✓	×	There is no documented functional test that exercises cancellation of a booking, including state checks, patient identit...
UC-4	Visualizza le mie prenotazioni	✓	✓	—
UC-5	Crea visita	✓	✓	—
UC-6	Cancella visita	✓	✓	Notification and refund side-effects described in the UC post-conditions are not represented in the documented design or...
UC-7	Modifica visita	✓	✓	Behavioral rules specific to modifying already booked visits (notifications, refund, visibility updates) are not explici...
UC-8	Visualizza le mie visite	✓	✓	—
UC-9	Scrivi recensioni al medico	×	×	The reviews functionality is only mentioned in the use case diagram as a future extension and has no corresponding desig...
UC-10	Visualizza le mie recensioni	×	×	The 'Visualizza le mie recensioni' use case is not implemented in the design and has no tests.
UC-11	Controllo fascia oraria	✓	✓	—
UC-12	Paga online	×	×	Online payment is only present in the use case diagram and UC-2 description but has no concrete design components or tes...
UC-13	Notifica Medico	×	×	Notification to the doctor is described in use case post-conditions but no notification subsystem or tests are documente...
UC-14	Rimborso	×	×	Refund handling is mentioned in cancellation-related use cases but no design elements or tests implement financial trans...

ID	Use Case	Design	Test	Gap
UC-15	Notifica paziente	×	×	Notification to the patient on visit cancellation/modification is only described textually and lacks implementation and ...

9 Terminological Consistency

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

Group	Variants found	Severity	Suggestion
Name of the system	"piattaforma", "gestionale", "software", "applicativo"	MAJOR	Use a single, consistent term for the software throughout the document, e.g. prefer "gestionale" or "piattaforma" and avoid mixing them unless clearly distinguished.
Medical visit vs lesson	"visita", "visite", "lezione", "lezioni"	MAJOR	Use one consistent Italian term for the same domain concept; for example, standardize on "visita" / "visite" and avoid mixing with "lezione" / "lezioni".
Visit offering / time slot	"annunci", "slot", "slot per le visite", "fascia oraria"	MAJOR	Use a single consistent term for the time slot concept; for example, standardize on "slot" (or "fascia oraria") and avoid mixing with "annunci".
Visit vs lesson in DAO	"Visit", "visita", "lesson", "lezione" (e.g. "setVisitState(vidVisit, lesson)", "Lo stato della lezione permuta in \"Cancellata\"")	MAJOR	Use a single consistent term for the same entity across layers; for example, standardize on "Visit" / "visita" and avoid using "lesson" in DAO methods.
Actors / users	"Paziente", "paziente", "Medico", "medico", "studenti", "utenti"	MAJOR	Use one consistent term for the same actor type; for example, standardize on "Paziente" / "paziente" and "Medico" / "medico" and avoid mixing with "studente" or generic "utenti" when referring to the same roles.
Business logic layer naming	"businessLogic", "business logic", "Controllers", "API del gestionale"	MAJOR	Use a single consistent term for the same architectural layer; for example, standardize on "businessLogic" and avoid mixing with "business logic" or generic "Controllers" when naming the package.

Group	Variants found	Severity	Suggestion
DAO layer naming	"DAO", "dao", "Data Access Objects"	MAJOR	Use a single consistent term for the data access layer; for example, standardize on "DAO" and avoid mixing with lower-case "dao" when referring to the package or layer.
Database naming	"database", "DB", "data layer"	MINOR	Use a single consistent term for the database; for example, standardize on "database" or "DB" and use the same form in similar contexts.
Unit test naming	"test di unità", "unit test", "test funzionali", "functionalTests"	MINOR	Use a single consistent term for the testing type; for example, standardize on "test di unità" or "unit test" and avoid mixing Italian and English forms in the same context.
Tags package vs Tag class naming	"Tags", "Tag", "tags", "package Tags", "package tags"	MINOR	Use a single consistent term for the tag subsystem; for example, standardize on "Tags" (with that exact capitalization) for the package and abstract class, and avoid mixing with "Tag" / "tags" when referring to the same concept.