Software Engineering Audit Report

Domain Driven Design of a Digital Ticketing Validation System to Prevent Scalping.pdf

CAPRA

February 25, 2026

Contents

1 Document Context

Project Objective

This project develops a domain-driven digital ticketing validation system designed to prevent ticket scalping by reversing the traditional verification flow. Instead of guests presenting QR codes to staff, staff members display verification codes to guests, who scan them to confirm attendance. The system incorporates JWT-based authentication, DTO validation, and database persistence following software engineering best practices.

Main Use Cases

- UC-1 – User Registration: User creates an account with name, surname, email, and password to gain authenticated access to the system.
- UC-2 – User Login: User authenticates with email and password to access personalized system interface.
- UC-3 – Event Creation: Admin creates a new event with title, description, date, ticket quantity, and price.
- UC-4 – Add Staff Member: Admin searches for a user by email and adds them as staff to an event.
- UC-5 – Remove Staff Member: Admin selects and removes a staff member from an event.
- UC-6 – Consultation of all available events: Guest views a list of upcoming events with relevant information.
- UC-7 – Ticket Purchase: Guest selects an event, specifies quantity, chooses payment method, and completes purchase.
- UC-8 – Ticket Verification: Guest and Staff collaborate where staff generates a QR code, guest scans it, system verifies ticket validity and marks it as used.

Functional Requirements

- User registration with email, name, surname, and password validation.
- User login with JWT-based authentication and token generation.
- Event creation, update, and deletion by authenticated admins.
- Staff member management (add/remove) by event admins.
- Guest ticket purchase with payment method selection.
- Ticket verification through QR code scanning with unique verification codes.
- Ticket usage tracking to prevent duplicate validation.
- Event listing filtered by availability and date.
- Role-based access control for Admin, Staff, and Guest actors.
- DTO validation for all user inputs.

Non-Functional Requirements

- Security: JWT-based authentication, bcrypt password hashing, secure verification code generation.
- Database: PostgreSQL relational database with BCNF normalization and indexed email field.
- Performance: Stateless controllers with centralized session management for verification processes.
- Scalability: DAO pattern for modular database access; Strategy pattern for multiple payment methods.
- Testability: 92.5% class coverage and 77.8% method coverage with 106 total tests (76 unit, 30 integration).
- Maintainability: Domain-driven design with clear separation of concerns across business, model, and core packages.

Architecture

The system follows a layered architecture with Domain-Driven Design principles. The **Business** layer contains controllers (`UserController`, `AdminController`, `GuestController`, `StaffController`) serving distinct actors. The **Model** layer defines domain entities (`Event`, `User`, `Ticket`, `VerifySession`) with invariant enforcement. The **Core** layer includes the DAO pattern for database abstraction, DTOs for data transfer, validation utilities, payment strategies (Strategy pattern), and exception handling. `ApplicationManager` implements centralized dependency injection. `ConcreteVerifySessionService` maintains stateful verification session data. The database uses PostgreSQL with tables for `appuser`, `event`, `ticket`, `admin`, and `staff`. Key technologies: Java 17, Maven, JDBC, JWT (`jjwt`), bcrypt, Hibernate Validator.

Testing Strategy

The project employs Test-Driven Development with 106 tests achieving 73.7% line coverage. **Unit Tests** (76 tests) use JUnit and Mockito to isolate components via dependency injection, mocking services and DAOs to avoid database dependencies. Tests cover controllers, services, utilities, validators, and domain models. **Integration Tests** (30 tests) use a real PostgreSQL database instance to verify combined component interactions across three categories: Authentication (user creation/login), Event Management (event lifecycle and staff management), and Ticket Verification (QR code scanning and validation). Database tables are cleared between tests to maintain isolation. Test coverage breakdown: Business layer 92.6%, Model layer 90.7%, DAO layer 58.9%, Validation 86.4%, Utils 90.5%.

2 Executive Summary

Quick Overview

Total issues: **14** — HIGH: **4** MEDIUM: **7** LOW: **3**

Average confidence: **98%**

Executive Summary: Audit of Domain Driven Design of a Digital Ticketing Validation System to Prevent Scalping

This document presents a Domain Driven Design (DDD) specification for a digital ticketing system with anti-scalping mechanisms. The work includes a domain model, use case specifications, architectural design, and comprehensive test documentation. The audit identified 14 issues across architecture, requirements, and testing, with 4 classified as HIGH severity.

The primary patterns of issues are: (1) **underspecified alternative flows and error handling** in use cases, where critical failure scenarios lack precise definitions of system state changes and user-visible behavior; (2) **gaps between requirements and implementation**, particularly regarding the anti-scalping mechanism, which is described informally but implemented with specific state machines not reflected in the requirements section; (3) **incomplete test coverage** for complex flows, especially payment processing and verification code validation, where integration tests focus on happy paths while error scenarios remain unit-tested only; and (4) **inconsistent specification of pre-conditions and post-conditions**, making it unclear what assumptions the system can safely make.

Critical areas (HIGH severity) require immediate attention: (1) UC-7 contains a logically incorrect alternative flow copied from staff management use cases, creating ambiguity about payment failure handling; (2) UC-8 (Ticket Verification) does not map alternative flows to the VerifySessionStatus enumeration or Ticket.used flag, weakening the link between requirements and implementation; (3) alternative flows across all use cases lack precise definitions of resulting system state, making correctness reasoning and test derivation difficult; and (4) integration tests for ticket verification and payment processing do not clearly exercise the documented error scenarios end-to-end.

Overall Assessment: The document demonstrates solid architectural thinking and reasonable implementation coverage, but suffers from a significant disconnect between informal requirements and formal design artifacts. The anti-scalping mechanism is implemented but not rigorously specified at the requirements level. Test coverage is broad but lacks systematic traceability to use case flows, particularly for error conditions. The work is suitable for a prototype but requires substantial refinement in requirements precision and test completeness before production use.

Priority Actions:

1. **Correct and complete use case specifications:** Remove the erroneous alternative flow from UC-7, decompose the anti-scalping requirement into precise functional and non-functional requirements (e.g., define what "fake the verification code" means, specify VerifySessionStatus transitions, clarify ticket state invariants), and ensure all alternative flows explicitly state resulting system state, user feedback, and transaction semantics.
2. **Establish traceability between requirements and tests:** Create an explicit mapping matrix linking each use case flow (basic and alternative) to specific test methods; add integration tests for payment processing (success and failure paths) and verification code validation (replay attacks, invalid codes, used tickets); ensure error scenarios are tested end-to-end with real dependencies.
3. **Align pre-conditions, post-conditions, and authorization:** Clarify what "have access" means in UC-4 and UC-5 by referencing role definitions; replace unrealistic pre-conditions (e.g., "user does not have a profile") with proper alternative flows; explicitly document which operations require authentication vs. authorization and justify any deviations from the stated "all operations require authorization" principle.

3 Strengths

- **Comprehensive Use Case Documentation with Enhanced Traceability:** The project provides detailed use case templates that go beyond standard documentation by including DTO specifications and test references. This alignment between use cases, implementation artifacts (DTOs), and test cases (Integration Tests 1-30) demonstrates strong traceability and facilitates future API development.
- **Well-Structured Domain-Driven Design Architecture:** The implementation follows domain-driven design principles with clearly separated layers including Domain Model, Business Logic, Core Package, and Data Access Layer. The use of design patterns such as Builder pattern (Event Builder), Dependency Injection, and DAO composition shows mature architectural thinking.
- **Comprehensive Testing Strategy with Multiple Test Levels:** The project implements both unit tests (covering UserControllerTest, StaffControllerTest, GuestControllerTest, AdminControllerTest, and utility classes) and integration tests organized by functional areas (Authentication, Event Management, Ticket Verification). Code coverage metrics are documented with specific breakdown tables.
- **Complete User Interface Design with Mockups:** The document includes 11 detailed UI mockups covering all major user workflows (login, sign-up, event creation, staff management, ticket purchase, and verification), providing clear visualization of the system's user-facing functionality across different actor roles.
- **Thoughtful Problem Analysis and Drawback Acknowledgment:** The project explicitly identifies and documents potential drawbacks of the anti-scalping approach (no ticket printing, user inconvenience, technology dependency, scalability concerns), demonstrating critical thinking and realistic assessment of the proposed solution.
- **Secure Implementation with Industry-Standard Libraries:** The project incorporates security best practices including JWT-based authentication (jjwt library), password hashing (jbcrypt), DTO validation (Hibernate Validator), and environment variable management (dotenv-java) for sensitive credentials.

4 Expected Feature Coverage

Of 7 expected features: 6 present, 1 partial, 0 absent. Average coverage: **93%**.

Feature	Status	Coverage	Evidence
Unit testing framework implementation	Present	100% (5/5)	The document describes a systematic unit testing strategy with 76 unit tests and 30 integration tests, using JUnit and Mockito, and provides detailed lists of dedicated test classes such as UserControllerTest, GuestControllerTest, etc. It explicitly shows isolated testing with mocked dependencies in GuestControllerTest (Figure 23), mentions assertion-based verification implicitly via JUnit tests, and reports coverage metrics for both unit and integration tests in Tables 1 and 2.
Use of UML Diagrams for system modeling	Present	100% (8/8)	The report includes a use case diagram (Figure 1) with actors and use cases, multiple UML class and package diagrams for architecture and domain model (Figures 13, 14, 16, 17, 19, 20), and explicitly states the use of StarUML. It also provides many UI mockups (Figures 2–11), uses standardized UML notation (inheritance, associations, cardinalities), shows relationships between actors and use cases in the use case diagram, and clarifies functional requirements and navigation flows (e.g., event browsing and ticket purchase) through these diagrams and mockups.
Identification and definition of system actors	Present	100% (8/8)	Section 2.1.1 explicitly defines the actors Admins, Staff, and Guest and their responsibilities. The use case diagram and detailed use case templates (UC-1 to UC-8) define interactions between these users and system components, list specific user actions (e.g., create event, add staff, buy ticket, validate ticket), and document functional requirements per role, showing a structured approach to user requirements and delineation of system functionalities.

Feature	**Status**	**Coverage**	**Evidence**
Definition and Documentation of Use Cases	Present	100% (5/5)	Section 3.2 provides detailed use case templates UC-1 to UC-8, each defining specific user interactions, user goals (Level and Name fields), preconditions and post-conditions, and basic and alternative flows. Relationships between use cases and subcases are shown in the use case diagram (e.g., Buy Ticket includes Select Payment method, Validate User Ticket includes Perform Verification Session).
User interface and interaction design principles	Present	86% (6/7)	Section 3.3 presents UI mockups for login, sign-up, create event, staff management, events list, event and payment, and ticket validation flows (Figures 2–11), with structured input fields for user data and clear navigation elements like buttons and search bars. Use cases such as Ticket Purchase (UC-7) describe a step-by-step process for purchasing tickets, including payment method selection, and validation mechanisms are documented via field validation steps and error flows in UC-1, UC-2, and UC-3. Distinct user roles and their functionalities are reflected in different screens (e.g., staff vs guest validation mockups). Dynamic updates of user selections in the interface are not explicitly described beyond static mockups, so that item is not fully evidenced.

Feature	Status	Coverage	Evidence
Separation of concerns in software architecture	Partial	80% (4/5)	The project structure and Figures 13, 14, and 20 show distinct packages for model, business (controllers/services), core.DAO, validation, payment, utils, DTO, and exceptions, and the text explains interactions between controllers and domain models (e.g., AdminController using EventDAO and model.Event). Clear relationships between Controller, Service, and DAO layers are documented via ApplicationManager and the UML diagrams, and the roles of each component are described in sections 4.2–4.3, emphasizing modularity. However, there is no explicit View package; UI is only represented as mockups, so a full MVC separation including a View package is not present.
Data Access Object (DAO) pattern	Present	86% (6/7)	Section 4.3.4 explicitly describes the DAO pattern, listing EventDAO, StaffDAO, AdminDAO, UserDAO, and TicketDAO with CRUD operations and a shared DBManager, and Figure 20 shows DAO interfaces and their concrete implementations. SQL queries are encapsulated in DAO classes, exemplified by the getEventById method in Figure 21, and DAOs abstract database access from business logic. The document also discusses testing with a real database in integration tests but does not detail specific testing strategies per DAO method, so that checklist item is only partially evidenced.

5 Summary Table

Category	HIGH	MEDIUM	LOW	Total
Architecture	0	0	2	2
Requirements	2	5	1	8
Testing	2	2	0	4
Total	**4**	**7**	**3**	**14**

6 Issue Details

6.1 Architecture (2 issues)

ISS-001 — LOW [100%] — Page 20 The domain model uses inheritance for User, Admin, Staff, and Guest, while the Core package and DAOs are organized around role-specific responsibilities. The document emphasizes Domain-Driven Design and separation of concerns, but it does not explicitly justify why inheritance was chosen over composition for roles, nor how this choice aligns with the stated DDD principles and the ER model (where roles are represented via separate admin and staff tables). This can create confusion for readers about the intended aggregate boundaries and role modeling strategy.

> *– User: The base class with id, name, surname, passwordHash, and email. – Admin, Staff, and Guest extend User, each adding role-specific associations.*

Recommendation: Add a short architectural rationale explaining the choice of inheritance for Admin, Staff, and Guest versus alternative designs (e.g., a User entity with role associations). Clarify how this maps to the relational schema (admin and staff tables) and how DAOs handle role-specific behavior. This will improve consistency between the conceptual model, the database design, and the implementation, and help reviewers understand that the chosen pattern is intentional and coherent with your DDD approach.

ISS-002 — LOW [100%] — Page 21 The architecture claims stateless controllers with a single shared, stateful ConcreteVerifySessionService instance created in ApplicationManager. While this is acceptable for the prototype, the document does not discuss how this design impacts test isolation and potential cross-test interference, especially for integration tests that use a real database and a shared in-memory session store. Without clear reset logic for the in-memory session state, tests may become order-dependent.

> *Because each controller is stateless, it does not hold any in-memory data between method calls. However, the ConcreteVerifySessionService maintains a stateful data structure to keep track of ongoing verification sessions that link Staff and Guest information.*

Recommendation: Document and, if needed, implement explicit lifecycle management for ConcreteVerifySessionService in tests. For example, add a ‘clearSession()‘ call in the setup/teardown of all integration test classes that use ticket verification, or provide a test-only factory that creates a fresh VerifySessionService instance per test class. Update the Testing Strategy section to state how the in-memory session state is reset between tests to guarantee independence and reproducibility.

6.2 Requirements (8 issues)

ISS-003 — HIGH [100%] — Page 7 Alternative flows across multiple use cases are underspecified: they mention generic exceptions or messages but do not define the resulting system state or user-visible behavior. In UC-4 and UC-5, critical cases like ’staff member already in the event’ or ’user not in the event staff’ are handled with ’do nothing’, which is ambiguous: it is unclear whether the admin receives feedback, whether the operation is considered successful, and what invariants must hold. Similar patterns appear in UC-1, UC-2, UC-3, UC-6, UC-7, and UC-8, where error conditions are described only as ’displays an error message’ or ’verification process fails’ without specifying whether transactions are rolled back, tickets remain unused, or sessions are invalidated. This weakens the ability to reason about correctness and to derive precise tests.

> *Alternative flow • If the staff member is already in the event: do nothing. • If the entered email does not correspond to any user: throw an exception. • If an error occurs while saving the data in the system, the organizer receives an error notification.*

Recommendation: For all use cases UC-1 to UC-8, refine each alternative flow to explicitly state: (1) what the system does internally (e.g., no DB changes, rollback of partial updates, invalidation of sessions), (2) what feedback is shown to the actor (exact outcome: success/failure and reason), and (3) what the final post-condition is. Replace vague phrases like 'do nothing' with concrete behavior, for example: 'The system does not modify the staff list and shows a message "User is already staff for this event"; the use case ends with the same post-conditions as before the request.' For error cases involving persistence or authentication, specify that no partial data is saved and that the operation is considered failed. Then, update integration tests to assert these clarified behaviors (e.g., that staff is not duplicated, that no ticket is marked used on failed verification, etc.).

ISS-004 — HIGH [100%] — Page 8 UC-7 (Ticket Purchase) contains an evidently incorrect and contradictory alternative flow: 'If the user is not in the event staff: do nothing.' This condition is unrelated to ticket purchase by a guest and appears to be a copy-paste from staff management use cases. It conflicts with the actor definition ('Actors: Guest') and with the business logic, where guests buy tickets independently of staff membership. This indicates a logical error in the requirements and makes it unclear how failures in payment or ticket availability should be handled at the requirements level.

> *Alternative flow • If the user is not in the event staff: do nothing. • If an error occurs while saving the data in the system, the organizer receives an error notification.*

Recommendation: Correct UC-7 by removing the irrelevant condition about 'user is not in the event staff' and replacing it with alternative flows that match the actual business logic and tests. For example, add explicit flows for: (1) event does not exist, (2) not enough tickets available, (3) payment method invalid, (4) payment fails, (5) event update (ticket decrement) fails. You already have corresponding unit tests (e.g., 'buyTicketShouldThrowBadRequestExceptionIfNotEnoughTicketsAvailable', 'buyTicketShouldThrowBadRequestExceptionIfPaymentFails', 'buyTicketShouldThrowExceptionIfPaymentMethodIsInvalid'); use these as a guide to write precise alternative flows that describe the user-visible outcome and system state (no ticket created, ticketsAvailable unchanged, etc.). Ensure the post-condition 'a new ticked is issued' is explicitly stated to hold only in the successful basic flow.

ISS-005 — MEDIUM [100%] — Page 3 Non-functional aspects such as security and performance are partially implied (JWT-based authentication, DTO validation) but not captured as explicit non-functional requirements. At the same time, the use case diagram states 'All operations require authorization.' and the system aims to 'make ticket scalping impossible', which are strong security and access-control claims. There is no dedicated section specifying requirements like token expiration, protection against replay of verification codes, or constraints on performance (e.g., acceptable delay for ticket validation at entrance). This weakens the traceability between the architectural choices (JWTUtility, VerifySessionService, Payment subsystem) and the system's quality goals.

> *The project serves as a simple demonstration rather than a complete implementation. However, it includes functional JWT-based authentication, DTO validation, and database persistence, following best practices of software engineering and domain-driven design.*

Recommendation: Add a short but explicit non-functional requirements subsection in the Requirements Analysis that covers at least: (1) Security: authentication via JWT, token expiration policy, required checks before each controller operation, constraints on verification code uniqueness and lifetime; (2) Reliability: what happens on DB or network errors during critical flows like ticket purchase and verification; (3) Performance/Usability: expected maximum time for ticket validation at entrance, acceptable failure rates. Then, ensure that the statement 'All operations require authorization.' from the use case diagram is reflected in each use case's pre-conditions and in the description of controller methods (whoami, buyTicket, startVerificationSession, etc.). Where the implementation intentionally simplifies (e.g., polling instead of WebSockets), mention this as a relaxed non-functional requirement for the prototype, and, if possible, add at least one test or scenario that checks token expiration or invalid tokens in integration tests.

ISS-006 — MEDIUM [100%] — Page 3 The core anti-scalping requirement is described informally and somewhat over-claimed ('this cannot be falsified') without being decomposed into precise functional and non-functional requirements. The later Ticket Verification use case (UC-8) and the domain model (VerifySession, VerifySessionStatus, VerifySessionService) implement a more nuanced process with states like PENDING, WAITING_FOR_GUEST, VALIDATED, INVALID and exceptions when 'the guest attempts to fake the verification code', but these states and constraints are not reflected in the requirements section. This gap makes it hard to verify that the implementation truly enforces 'user can't see or exchange the ticket before arriving at the event' and what 'fake the verification code' concretely means (e.g., replay attacks, using someone else's code, scanning outside the venue).

> *The approach proposed in this work relies on the fact that it is not the client who presents the QR code to the staff, but the staff member who shows a verification QR code to the guest containing a secret, unique code. The user scans it and doing so confirms the presence at the entrance of the event: this cannot be falsified and implies that the user can't see or exchange the ticket before arriving at the event.*

Recommendation: Refine the requirements analysis to break down the anti-scalping goal into explicit, testable requirements. For example, add requirements such as: (1) 'Tickets are not represented as reusable QR codes; only staff-generated verification codes are scanned by guests.' (2) 'A verification session is bound to a specific staff member, event, and time window; codes cannot be reused after validation or timeout.' (3) 'A ticket can be marked used only once; subsequent attempts must be rejected.' Then, align UC-8 with the VerifySessionStatus states by adding steps and alternative flows for PENDING, WAITING_FOR_GUEST, VALIDATED, INVALID, and for attempts to reuse or forge codes. Avoid absolute statements like 'this cannot be falsified'; instead, describe the threat model and what attacks are mitigated. Finally, ensure integration tests 22–30 explicitly cover these refined requirements (e.g., replay of a verification code, validating a session belonging to another staff, validating when user has no tickets).

ISS-007 — MEDIUM [100%] — Page 5 Only a subset of the use cases from the use case diagram is documented with detailed templates. The diagram includes use cases such as 'Perform Verification Session', 'Select Payment method', 'Prove ticket validity', and 'Scan Verification Code', but the detailed templates cover only UC-1 to UC-8. Some of these diagram use cases are implicitly merged into others (e.g., 'Select Payment method' inside UC-7, 'Scan Verification Code' inside UC-8), but this is not clearly stated, and there is no explicit mapping between diagram elements and UC IDs. This can confuse readers and examiners and makes it harder to ensure that every diagrammed behavior is covered by requirements and tests.

> *The following templates outline some of the primary use cases identified in the project.*

Recommendation: Create a small mapping table that links each use case in the diagram to either a detailed UC template or to a specific step within another UC. For example: 'Select Payment method' → step 4 of UC-7; 'Scan Verification Code' → step 3 of UC-8; 'Perform Verification Session' → combined behavior of UC-8 plus StaffController methods startVerificationSession and validateVerificationSession. In the text of UC-7 and UC-8, explicitly mention that they cover these sub-use-cases. If any diagram use case represents a distinct business goal not fully described (e.g., 'Get his events' for admins), add a concise UC template or justify why it is out of scope for the prototype. This will improve completeness and traceability between the diagram, textual requirements, and the implemented controllers and tests.

ISS-008 — MEDIUM [100%] — Page 5 Pre-conditions and post-conditions are inconsistently specified and sometimes unrealistic. In UC-1 (User Registration), the pre-condition requires that the user 'does not have a profile', but the system has no way to know this before the operation; in practice, the system must handle the case where the email already exists as an alternative flow, not as a pre-condition. In UC-4 and UC-5, the pre-conditions say the admin 'should have access to an event along with the e-mail of the user', but they do not define what 'have access' means (admin vs staff vs guest) or how this is enforced. In UC-6 and UC-7, the pre-condition is just 'The guest must be authenticated' / 'The guest is authenticated in the system', but the use case diagram also states 'All operations require authorization.', creating a mismatch between diagram and textual pre-conditions for some use cases (e.g., viewing events might be allowed to unauthenticated users in many systems, but here it is not clearly justified).

> *Pre-conditions The user is not authenticated in the system and does not have a profile. Post-conditions The user has a profile, is authenticated, and has access to the personalized system interface.*

Recommendation: Review all UC-1 to UC-8 pre-conditions and post-conditions to ensure they describe only states that the system can check and enforce. For UC-1, change the pre-condition to something like 'The user is not authenticated' and move 'email already registered' into an explicit alternative flow with a clear error message and unchanged DB state. For UC-4/UC-5, define 'have access to an event' precisely (e.g., 'The admin is authenticated and is an admin of the selected event') and ensure this matches the authorization checks implemented (e.g., EventDAO.isUserAdminOfEvent). For each use case, verify that the post-conditions describe the final persistent state (e.g., 'ticket is created and associated to user and event', 'ticket remains unused on failure') and that they are compatible with the domain model and DAOs. Align these conditions with the 'All operations require authorization.' constraint from the use case diagram, explicitly stating authentication/authorization requirements where needed.

ISS-009 — MEDIUM [100%] — Page 9 UC-8 (Ticket Verification) mentions 'the guest attempts to fake the verification code' and 'guest doesn't have a ticket (or it's used)' but does not define how the system detects these conditions or what exact behavior follows. There is no link to the VerifySessionStatus enumeration (PENDING, WAITING_FOR_GUEST, VALIDATED, INVALID) or to the Ticket.used flag in the domain model. The phrase 'verification process fails' is vague: it does not specify whether the session is marked INVALID, whether the staff is notified with a specific reason, or whether further attempts are allowed. This ambiguity makes it difficult to ensure that the business logic (ConcreteVerifySessionService, TicketDAO.setTicketUsed) and tests (StaffControllerTest, ConcreteVerifySessionServiceTest, Ticket Verification Integration Tests) fully satisfy the intended behavior.

> *Alternative flow • If the guest attempts to fake the verification code, the system will raise an excep- tion. • If the guest doesn't have a ticket (or it's used), the verification process fails (see mockup in Figure 11).*

Recommendation: Extend UC-8 to describe the verification session lifecycle in terms of the domain model. For each alternative case, specify: (1) which VerifySessionStatus the session transitions to (e.g., INVALID on fake code or no tickets), (2) how the Ticket.used flag is affected (must remain false on failure), and (3) what feedback is shown to staff and guest (e.g., 'Invalid ticket – already used', 'No ticket for this event'). Replace 'raise an exception' and 'verification process fails' with explicit outcomes: 'The system marks the session as INVALID, does not modify any ticket, and shows the failure screen with reason X.' Then, cross-check that unit tests (e.g., 'validateVerificationSessionShouldThrowIfNoTicketsFound', 'validateVerificationSessionShouldThrowIfAllTicketsUsed') and integration tests 25–30 cover each of these refined flows. This will tightly couple requirements, domain states, and tests.

ISS-010 — LOW [100%] — Page 32 DTOs are listed and referenced in the use cases (e.g., 'DTO 8' in UC-7), but the use case descriptions do not specify validation rules or constraints that are later enforced in code and tests. For example, BuyTicketDTO has a 'paymentMethod: String', and there are unit tests like 'buyTicketShouldThrowExceptionIfPaymentMethodIsInvalid', but UC-7 does not mention that only certain payment methods are allowed or that invalid values cause a specific error. Similar gaps exist for other DTOs (e.g., email format in CreateUserDTO, date constraints in CreateEventDTO). This reduces the clarity of the contract between the UI/API and the business logic.

> *DTO 8: BuyTicketDTO Arguments: - eventId: Integer - quantity: Integer - paymentMethod: String*

Recommendation: For each use case that references a DTO (UC-1 to UC-8), add a short bullet list of key validation rules derived from the DTO and validation layer. For example, in UC-7, specify: 'The payment method must be one of {"credit_card", "apple_pay", "google_pay"}; otherwise, the system rejects the request with a BadRequest error and does not create a ticket.' In UC-1, mention that email must be in a valid format and unique; in UC-3, that date must be in the future and ticketsAvailable/ticketPrice must be non-negative. Align these rules with MyValidatorTest and the relevant unit tests. This will make the requirements more precise and show a clear link between DTO design, validation, and observed behavior.

6.3 Testing (4 issues)

UC-8

TST-001 — HIGH [100%] — Page 9 Ticket Verification (UC-8) defines explicit alternative flows for faked verification codes and guests without valid/unused tickets, but the documented integration tests for ticket verification do not clearly map to these specific error scenarios. The listed tests focus on session lifecycle and staff authorization, leaving ambiguity about whether the system behavior for fake codes and invalid/used tickets is actually verified end-to-end.

> *Alternative flow • If the guest attempts to fake the verification code, the system will raise an excep- tion. • If the guest doesn't have a ticket (or it's used), the verification process fails (see mockup in Figure 11). DTOs DTOs 9, 10, 11 Test Integration Tests 25, 26, 27, 28, 29, 30*

Recommendation: Make the Ticket Verification integration tests explicitly cover each UC-8 alternative flow. Add or rename tests so that it is clear which scenario they validate, for example: (1) 'guestScanShouldFailWithFakeVerificationCode' that uses an invalid or tampered code in ScanStaffVerificationCodeDTO and asserts that the expected exception or error status is produced; (2) 'staffValidateShouldFailWhenGuestHasNoTicket' and 'staffValidateShouldFailWhenTicketAlreadyUsed' that set up the DB with no ticket or a used ticket and verify that the controller returns the correct failure outcome. Reference these test names directly in the UC-8 Test field to improve traceability.

UC-7

TST-002 — HIGH [100%] — Page 35 Ticket Purchase (UC-7) relies on payment processing and error handling, but there are no integration tests that exercise the payment flow end-to-end (success and failure of payment methods). Existing integration tests for ticket purchase only check ticket availability and quantity, not the payment behavior described in the use case and implemented via the Payment subsystem.

> *Integration test 18: eventPurchaseShouldDecrementAvailableTickets Integration test 19: eventPurchaseShouldFailIsNotEnoughTickets*

Recommendation: Extend the Event Management or Ticket Purchase integration tests to cover the full payment flow for UC-7. For example, add tests such as ‘eventPurchaseShouldSucceedWithValidPaymentMethod‘ and ‘eventPurchaseShouldFailWhenPaymentProviderRejects‘ that: (1) create a real event and guest in the test DB, (2) call the controller method that handles BuyTicketDTO with different ‘paymentMethod‘ values, (3) verify that on success the ticket is created and the event's ticketsAvailable is decremented, and (4) on failure no ticket is created and ticketsAvailable is unchanged. Use the existing PaymentContext/PaymentStrategyFactory to simulate different payment outcomes instead of bypassing them.

UC-1

TST-003 — MEDIUM [68%] — Page 5 Use cases UC-1 to UC-8 list associated integration tests only as numeric identifiers, but the document never provides an explicit traceability matrix or one-to-one mapping between each basic/alternative flow step and specific test methods. This makes it hard to verify that every requirement in the use cases (especially alternative flows and error messages) is actually covered by tests, even though many tests exist.

> *Test Integration Tests 1, 2, 3 … Test Integration Tests 4, 5, 6 … Test Integration Tests 7 … Test Integration Tests 11, 12, 13 … Test Integration Tests 14, 15, 16 … Test Integration Tests 8 … Test Integration Tests 18, 19 … Test Integration Tests 25, 26, 27, 28, 29, 30*

Recommendation: Introduce a simple traceability table that maps each use case step and alternative flow to concrete test method names. For each UC (UC-1 to UC-8), list: (1) basic flow steps and the integration test(s) that cover them, and (2) each alternative flow and the specific unit/integration test that verifies it (e.g., UC-4 alt flow "If the entered email does not correspond to any user: throw an exception." → ‘addStaffShouldThrowNotFoundExceptionIfUserIsNotFound‘ and ‘wrongEmailStaffShouldThrowException‘). Update the Test field in each UC to reference these method names, not just numeric IDs, to strengthen requirement-to-test traceability.

General Issues

TST-004 — MEDIUM [100%] — Page 33 Across multiple controllers, error and alternative flows are tested only at unit level with mocked dependencies, while integration tests mostly cover happy paths and a subset of authorization errors. For example, GuestController's many failure scenarios for ticket purchase are unit-tested but not exercised with the real DAO and DB; similarly, several AdminController and StaffController error conditions appear only in unit tests. This creates a systemic gap where complex flows may behave differently in the integrated system than in isolated unit tests.

> *Unit test 14: buyTicketShouldReturnTicketIfPurchaseIsSuccessful Unit test 15: buyTicketShouldThrowNotFoundExceptionIfEventDoesNotExist Unit test 16: buyTicketShouldThrowBadRequestExceptionIfNotEnoughTicketsAvailable Unit test 17: buyTicketShouldThrowBadRequestExceptionIfPaymentFails Unit test 18: buyTicketShouldThrowBadRequestExceptionIfEventUpdateFails Unit test 19: buyTicketShouldThrowExceptionIfTokenValidationFails*

Unit test 20: buyTicketShouldThrowExceptionIfPaymentMethodIsInvalid Unit test 21: buyTicketShouldThrowExceptionIfPaymentGoesWrong

Recommendation: For all critical controller flows (authentication, event management, ticket purchase, ticket verification), promote at least the most important error/alternative paths from unit-only coverage to integration coverage. Concretely: (1) identify unit tests that simulate important failures (e.g., invalid payment method, DAO failures, unauthorized admin/staff actions); (2) for each category, add one or two integration tests that reproduce the same scenario using the real DB and DAOs (e.g., misconfigured data, missing records, invalid tokens) instead of mocks; and (3) reference these new integration tests in the corresponding UC Test sections so that both success and key failure paths are validated end-to-end.

7 Priority Recommendations

The following actions are considered priority:

1. **ISS-003** (p. 7): For all use cases UC-1 to UC-8, refine each alternative flow to explicitly state: (1) what the system does internally (e.g.
2. **ISS-004** (p. 8): Correct UC-7 by removing the irrelevant condition about 'user is not in the event staff' and replacing it with alternative flows that match the actual...
3. **TST-001** (p. 9): Make the Ticket Verification integration tests explicitly cover each UC-8 alternative flow.
4. **TST-002** (p. 35): Extend the Event Management or Ticket Purchase integration tests to cover the full payment flow for UC-7.

8 Traceability Matrix

Of 15 traced use cases: 15 fully covered, 0 without design, 0 without test.

ID	Use Case	Design	Test	Gap
UC-1	User Registration	✓	✓	—
UC-2	User Login	✓	✓	—
UC-3	Event Creation	✓	✓	—
UC-4	Add Staff Member	✓	✓	—
UC-5	Remove Staff Member	✓	✓	—
UC-6	Consultation of all available events	✓	✓	—
UC-7	Ticket Purchase	✓	✓	—
UC-8	Ticket Verification	✓	✓	—
UC-9	Update Event	✓	✓	—
UC-10	Delete Event	✓	✓	—
UC-11	Get his events (Admin perspective)	✓	✓	Partially covered only indirectly by integration tests; no explicit test dedicated solely to 'Get his events' behavior.
UC-12	Manage Staff (aggregate use case)	✓	✓	—

ID	Use Case	Design	Test	Gap
UC-13	Perform Verification Session (staff side of validation)	✓	✓	—
UC-14	Scan Verification Code (guest side of validation)	✓	✓	—
UC-15	See all events (Guest perspective)	✓	✓	Covered at integration level but without a guest-specific controller test explicitly tied to this use case.