Architectural Blueprint Validation Report

JavaBrew Vending Machine Management Platform

Automated Traceability Analysis

October 19, 2025

Abstract

This report validates the architectural blueprint of the JavaBrew vending machine platform through automated traceability analysis. The assessment examines 62 requirements, 18 use cases, architectural components, and test coverage extracted via LLM-based document analysis. The report identifies critical gaps in requirement coverage, architectural clarity, and test completeness, providing actionable recommendations for improving system design quality.

Contents

1 Executive Summary

1.1 Assessment Overview

This validation analyzes the architectural blueprint using automated traceability extraction from project documentation. The system demonstrates strong coverage in core transaction flows but exhibits critical gaps in resilience and operational edge cases.

Key Metrics

- **Requirements:** 62 total, 59 covered (95.2%), 3 unsupported (4.8%)
- **Use Cases:** 18 defined, 15 fully tested, 3 with coverage gaps
- **Architecture:** Layered pattern with 50+ components across 6 layers
- **Tests:** 63 identified (45 unit, 12 integration, 6 system)

1.2 Critical Findings

Critical Issues Requiring Immediate Attention

1. **Offline Operation Gap:** Three requirements (REQ-18, REQ-19, REQ-20) for disconnected operation are completely unsupported by the architecture, creating a single point of failure on network connectivity (see Section 2.1).
2. **Vague Component Responsibilities:** Multiple core components lack precise responsibility definitions, violating the Single Responsibility Principle and risking architectural erosion (see Section 4.2).
3. **Partial Remote Maintenance:** The remote maintenance use case (UC-18) lacks hardware abstraction components, making the promised remote control functionality unimplementable (see Section 3.1).

Architectural Strengths

- Complete automated traceability from requirements through tests
- Well-defined layered architecture with proper separation of concerns
- Effective design patterns (Builder, DAO, Mapper) applied consistently
- Comprehensive test coverage for happy paths and common error scenarios
- Dual database strategy enabling fast test feedback loops

2 Requirements Coverage Analysis

2.1 Offline Operation: A Critical Gap

The most significant coverage gap involves offline operation capabilities. Three requirements specify behavior when vending machines lose Internet connectivity:

Unsupported Offline Requirements

REQ-18 - Local Transaction Tracking: The system must maintain a local register to track transactions during network outages, ensuring no sales data is lost.
REQ-19 - Offline-Online Synchronization: Once connectivity is restored, locally stored transactions must synchronize with the central database, maintaining data consistency.
REQ-20 - Anonymous Cash Transactions: When QR code scanning fails, anonymous users must be able to perform cash-only transactions as a fallback mechanism.

Architectural Impact:
The current architecture assumes persistent connectivity throughout all transaction flows. No components exist for:

- Local transaction storage on vending machine devices
- Conflict resolution or eventual consistency protocols
- Synchronization state management
- Offline authentication or authorization fallbacks

This represents a fundamental architectural assumption that may not hold in real-world deployments, where network reliability varies by location. Without offline capabilities, vending machines become non-operational during any network disruption, directly impacting revenue and user experience.

2.2 Requirements Quality Issues

Beyond coverage gaps, several requirements suffer from insufficient specificity:

Vague Requirements Impacting Design

REQ-8 - Digital Payment Methods: The requirement to "support digital payment methods" lacks specification of which payment providers (credit cards, mobile wallets, etc.) or compliance standards (PCI-DSS) are needed. This ambiguity prevents proper payment gateway architecture design.
REQ-10, REQ-21, REQ-58 - Performance and Usability Metrics: Requirements mentioning "improved user experience" and "operational efficiency" provide no quantifiable targets, making it impossible to validate whether the architecture achieves these goals or to design appropriate performance optimizations.
REQ-34, REQ-35, REQ-45 - Error Response Formats: Several error-handling requirements specify that errors must be returned but don't define the error response structure, potentially leading to inconsistent error handling across the system.

These vague requirements create architectural ambiguity—designers must make assumptions that may not align with actual business needs, increasing the risk of costly rework.

3 Use Case Analysis

3.1 Test Coverage Gaps

Three use cases exhibit incomplete or missing test coverage:

Untested Use Cases

UC-16 - User Navigation Flow: This use case describes multi-screen navigation for customers, workers, and admins but has no corresponding integration tests. Without navigation tests, UI flow bugs—such as broken transitions, incorrect role-based routing, or missing back navigation—may only be discovered in production.
UC-18 - Remote Maintenance: While backend services and database components exist to track maintenance tasks, no tests verify the actual remote control capabilities (e.g., unlocking jammed products). This gap reflects the deeper architectural issue: the hardware abstraction layer needed to bridge software and physical device control is absent from the architecture entirely.
UC-7, UC-9 - Container Use Cases: Two use cases serve as organizational containers with no defined flows or tests. These add no traceability value and create confusion in the use case model.

3.2 Well-Covered Use Cases

The majority of use cases demonstrate comprehensive test coverage:

Authentication and Authorization: Login and registration flows include nine tests covering valid credentials, invalid passwords, missing fields, null inputs, and system errors—ensuring robust error handling.

Purchase Workflows: The item purchase flow tests eight scenarios including successful purchases, insufficient balance, out-of-stock items, connection failures, and customer not found errors.

Administrative Operations: Analytics viewing, machine creation, and CRUD operations have dedicated tests for both success paths and error conditions (missing fields, save failures, DAO errors).

This coverage pattern reveals a focus on core transactional flows while edge cases (navigation, offline scenarios, hardware integration) remain under-tested.

4 Architectural Quality Assessment

4.1 Layered Architecture: Strength with Clarity Issues

The architecture follows a classic six-layer pattern:

1. **Presentation:** UI components and mockups
2. **Controller:** HTTP request routing and input validation
3. **Service:** Business logic orchestration
4. **DAO:** Data access abstraction with interfaces
5. **Persistence:** ORM (JPA/Hibernate) and database connections
6. **Domain Model:** Business entities and value objects

Layering Benefits Achieved:

- Controllers delegate to services; services call DAOs—no layer skipping observed
- Dependencies flow downward (upper layers depend on lower, not vice versa)
- Technology substitution is feasible (e.g., database swap from PostgreSQL to another RDBMS)

- Independent layer testing enabled through interface-based design

4.2 Component Responsibility Problems

Despite good layering structure, multiple components exhibit vague or overly broad responsibilities:

Single Responsibility Principle Violations

DAO Layer: Described as "manages data access and retrieval"—too generic to guide implementation. Without specific responsibilities per DAO (e.g., UserDao handles user CRUD only, TransactionDao handles financial records only), the risk of monolithic DAO classes increases.
Services Layer: "Contains business logic and processes data" is insufficiently specific. Business logic spans many domains (customer purchases, admin configuration, worker maintenance). Without clear bounded contexts or domain-specific service definitions, this layer risks becoming a "God Object" that centralizes too much logic.
Database Component: "Handles database interactions" overlaps with DAO responsibilities. The distinction between this component, DBManager (connection management), and DAO classes (query execution) is unclear, potentially causing confusion about where database-related code belongs.
User Role Entities: Admin, worker, and customer entities are flagged as potentially encompassing "various functions beyond just user attributes." If these entities contain behavior (methods) rather than just data (attributes), they may violate separation of concerns by mixing domain logic with user roles.

These ambiguities don't indicate implementation problems necessarily exist, but rather that the architectural documentation lacks precision. Without clear boundaries, developers may place responsibilities inconsistently, leading to technical debt accumulation.

4.3 Design Patterns: Effective Application

The architecture demonstrates judicious pattern usage:

Patterns Applied Effectively

Builder Pattern (ConcreteVendingMachine): Creating vending machine instances involves many optional parameters (location, capacity, status, etc.). The Builder pattern provides a fluent, readable construction API while enforcing required fields. This pattern choice is well-suited to the domain.
DAO Pattern: Each entity type has a corresponding DAO interface (UserDao, TransactionDao, ItemDao, etc.), abstracting persistence technology from business logic. This abstraction enables technology changes (e.g., swapping ORM frameworks) without affecting service layer code.
Mapper Pattern: TaskMapper, ConnectionMapper, InventoryMapper, and TransactionMapper classes separate domain models from database entities. This separation reduces coupling—domain logic changes don't force database schema changes and vice versa.

Strategic Pattern Choice: The Builder pattern appears only for ConcreteVendingMachine, suggesting deliberate pattern application rather than overengineering. Simpler entities use standard constructors, avoiding unnecessary complexity.

4.4 Domain Model Analysis

The domain model exhibits characteristics of Domain-Driven Design:

Rich Entities: ConcreteVendingMachine, Inventory, Transaction, and TransactionItem represent business concepts with behavior, not just data containers.

Value Objects: MachineStatus (enumeration) encapsulates machine state as an immutable value, preventing invalid states.

Composition Relationships: ConcreteVendingMachine contains an Inventory (one-to-one), Transaction contains multiple TransactionItems (one-to-many)—modeling real-world relationships.

Inheritance Hierarchy: A base app_user entity extends into admin, worker, and customer roles, enabling role-based polymorphism.

Missing DDD Elements:

- **Aggregate Roots:** No enforcement preventing direct Inventory modification without going through ConcreteVendingMachine. Without aggregate boundaries, invariants (e.g., "inventory cannot exceed machine capacity") may be violated.
- **Domain Events:** No events like ProductPurchased, BalanceRecharged, or MaintenanceTaskCreated for auditing or asynchronous processing, limiting extensibility.

5 Test Strategy Assessment

5.1 Test Infrastructure Quality

The test infrastructure demonstrates maturity:

Modern Testing Stack:

- JUnit 5.11.0 for unit test execution
- Mockito 5.18.0 for dependency isolation
- JaCoCo for code coverage measurement
- H2 in-memory database for fast integration tests
- PostgreSQL for production, ensuring test-prod parity

Test Distribution: The 63 identified tests break down approximately as 71% unit, 19% integration, 10% system—roughly following the ideal test pyramid shape. This distribution supports fast feedback (unit tests) while validating integration points and end-to-end flows.

5.2 Testing Strengths

Comprehensive Error Path Coverage: Most use cases test not just success scenarios but multiple failure modes:

- Login: invalid password, nonexistent email, null/empty inputs, system errors
- Purchase: insufficient balance, out of stock, item not found, connection errors, customer not found
- Machine connection: already connected, out of service
- Task completion: save errors, null status, already completed

This error-first testing approach increases system resilience by ensuring graceful degradation.

DAO-Level Testing: Each DAO implementation has dedicated CRUD tests verifying persistence operations work correctly. Integration tests validate that service-DAO-database interactions function end-to-end.

Service Isolation: Service layer tests use Mockito to simulate DAO responses, enabling fast, deterministic unit tests that don't depend on database state.

5.3 Testing Gaps

Beyond the use case coverage gaps (navigation, remote maintenance), the test suite lacks:

Performance Tests: No load tests verify system behavior under concurrent user load or stress conditions. Without performance baselines, scalability limits remain unknown.

Security Tests: No tests verify input sanitization (SQL injection protection), authentication bypass prevention, or authorization boundary enforcement. Security vulnerabilities may exist undetected.

End-to-End Workflow Tests: Tests validate individual use cases but don't verify complete user journeys spanning multiple use cases (e.g., register → recharge wallet → scan QR → purchase → view history).

6 Critical Risks and Recommendations

6.1 Risk 1: Offline Operation Unavailability

Risk: Vending machines cannot process any transactions during network outages, resulting in complete service unavailability and revenue loss.

Root Cause: Architecture assumes always-on connectivity with no offline fallback design (see Section 2.1).

Related Requirements: REQ-18, REQ-19, REQ-20

Recommendation:

- Design a local transaction storage mechanism on vending machine devices
- Define synchronization protocol with conflict resolution for offline-to-online reconciliation
- Architect offline authentication approach (cached credentials, device tokens, or anonymous transactions)
- Add corresponding components, use cases, and tests to traceability matrix

6.2 Risk 2: Component Responsibility Ambiguity

Risk: Vague component definitions lead to inconsistent code placement, eventual architecture degradation, and maintenance difficulty.

Root Cause: Architectural documentation describes components at too high a level without specific responsibility boundaries.

Recommendation:

- Document precise, single responsibilities for DAO, services, and database components
- Clarify which component handles what: DBManager (connection pooling), DAO interfaces (query contracts), DAO implementations (query execution), services (business rules)
- If services layer is too broad, split into bounded domain services (CustomerService, AdminService, WorkerService) with explicit contexts
- Update architectural diagrams with refined component descriptions

6.3 Risk 3: Remote Maintenance Unimplementable

Risk: Remote maintenance use case promises capabilities (unlocking jammed products) that cannot be implemented without hardware integration components.

Root Cause: Architecture defines backend services for task tracking but lacks the hardware abstraction layer needed to send commands to physical vending machine devices.

Related Use Case: UC-18

Recommendation:

- Define a device gateway or IoT adapter component that bridges software and hardware
- Specify communication protocol with vending machine firmware (MQTT, HTTP, proprietary)
- Design command-response model for remote operations
- Create mock hardware interfaces for testing remote control scenarios
- Update use case to reflect implementation limitations or extend architecture to support full remote control

6.4 Risk 4: Untested Edge Cases

Risk: Navigation flows, complete user journeys, performance limits, and security vulnerabilities remain unvalidated.

Root Cause: Test focus on individual use case validation rather than holistic system behavior (see Section 3.1).

Related Use Cases: UC-16, UC-18

Recommendation:

- Add navigation integration tests covering multi-screen workflows for each user role
- Implement end-to-end tests that span multiple use cases in sequence
- Introduce performance testing to establish scalability baselines
- Add security tests for common vulnerabilities (injection attacks, authentication bypass, authorization boundary violations)

6.5 Risk 5: Requirements Ambiguity

Risk: Vague requirements lead to architectural assumptions that may not match business intent, requiring costly rework when assumptions prove incorrect.

Root Cause: Requirements lack specific, measurable acceptance criteria (see Section 2.2).

Related Requirements: REQ-8, REQ-10/21/58, REQ-34/35/45

Recommendation:

- Specify which payment providers and compliance standards are required
- Define quantifiable performance and usability targets (response times, task completion times, error rates)
- Standardize error response formats across the API
- Detail what "remote maintenance capabilities" concretely entails

7 Positive Practices to Maintain

7.1 Comprehensive Traceability

The automated traceability extraction creates a complete requirements-to-tests mapping, enabling:

- Impact analysis: when requirements change, affected use cases, components, and tests are immediately identifiable
- Coverage verification: ensures every requirement has corresponding design and validation
- Regression prevention: tests linked to requirements prevent unintended behavior changes

This traceability infrastructure should be maintained and evolved as the system grows.

7.2 Layered Architecture Discipline

The architecture maintains clean layer separation without shortcuts or layer-skipping, providing:

- Technology independence: persistence technology can change without affecting business logic
- Testability: each layer can be tested in isolation
- Understandability: clear responsibility distribution across layers

Continue enforcing layering principles as new features are added.

7.3 Pattern-Driven Design

Strategic pattern application (Builder, DAO, Mapper) where appropriate—without overengineering—demonstrates architectural maturity. Patterns solve specific problems rather than being applied universally, keeping the codebase maintainable.

7.4 Error-First Testing

Testing alternative flows and error scenarios alongside happy paths increases system resilience. This practice should extend to edge cases currently under-tested (navigation, hardware integration, offline scenarios).

8 Conclusion

8.1 Overall Assessment

The JavaBrew architectural blueprint demonstrates strong fundamentals: comprehensive traceability, disciplined layering, and effective pattern application. The 95.2% requirements coverage and extensive test suite indicate a mature development process.

However, three critical gaps threaten production viability:

1. **Offline operation** is architecturally unsupported despite explicit requirements (REQ-18, REQ-19, REQ-20), creating a single point of failure on network connectivity (see Risk 1)
2. **Component responsibilities** lack precision, risking architectural erosion as the codebase evolves (see Risk 2)
3. **Remote maintenance** (UC-18) is partially implemented—promised but undeliverable without hardware abstraction (see Risk 3)

8.2 Recommended Actions

Before Production Deployment:

- Design and prototype offline operation architecture
- Refine component responsibility definitions
- Complete or scope-reduce remote maintenance capabilities
- Add navigation and end-to-end tests

For Continuous Improvement:

- Clarify vague requirements with measurable criteria
- Add performance and security testing
- Resolve or remove orphaned use cases
- Consider aggregate root enforcement and domain events for enhanced domain model

8.3 Final Assessment

This architecture provides a solid foundation suitable for initial deployment in controlled environments with reliable connectivity. Addressing the offline operation gap and clarifying component boundaries will elevate the design to production-grade robustness for diverse deployment scenarios.

The automated traceability analysis proves valuable for identifying these gaps early, before implementation costs make corrections expensive. Maintaining this traceability discipline as the system evolves will continue to provide quality assurance benefits.