Architectural Blueprint Validation Report

JavaBrew Vending Machine Management Platform

Software Engineering Assessment

October 19, 2025

Abstract

This report presents a comprehensive validation of the JavaBrew vending machine management platform's architectural blueprint. The analysis examines the traceability between requirements (62 identified), use cases (18 defined), architectural components (multiple layers), and test coverage (63 tests). The assessment identifies architectural strengths, critical weaknesses, and provides actionable recommendations aligned with software engineering best practices.

Contents

1 Executive Summary

1.1 Overall Assessment

Summary Metrics

- **Total Requirements:** 62 (59 Covered, 3 Unsupported)
- **Total Use Cases:** 18 (16 Covered, 2 Orphaned/Missing Tests)
- **Total Tests:** 63 (Unit, Integration, and System levels)
- **Requirements Coverage:** 95.2%
- **Use Case Coverage:** 88.9%
- **Traceability Completeness:** Strong (with identified gaps)

1.2 Key Findings

Strengths

- Comprehensive traceability matrix linking requirements to use cases, components, and tests
- Well-structured layered architecture with clear separation of concerns (Controller-Service-DAO-Database)
- Strong test coverage across multiple levels (unit, integration, system)
- Effective use of design patterns (Builder, DAO, Mapper)
- Domain-Driven Design principles applied to core entities

Critical Issues

- Three requirements (REQ-18, REQ-19, REQ-20) marked as UNSUPPORTED without architectural plan for future support
- Multiple architectural components with vague or overly broad responsibilities
- Absence of explicit architectural patterns in several component descriptions
- Missing test coverage for remote maintenance flows (UC-18) and navigation flows (UC-16)
- Two orphaned use cases (UC-7, UC-9) with unclear purpose
- No documented non-functional requirements handling (performance, scalability, security)

2 Traceability Analysis

2.1 Requirements Coverage Analysis

2.1.1 Covered Requirements (59/62 - 95.2%)

The majority of functional requirements are well-traced through the architecture:

- **Core Platform Requirements (REQ-1 to REQ-10):** All covered with comprehensive component and test mappings
- **User Management (REQ-11, REQ-23-29):** Fully implemented with authentication flows and validation
- **Purchase Workflows (REQ-30-42):** Complete coverage including error handling
- **Admin Operations (REQ-13-15, REQ-50-55):** Dashboard, analytics, and CRUD operations traced
- **Worker Maintenance (REQ-15, REQ-46-49):** Task management flows implemented

2.1.2 Unsupported Requirements (3/62 - 4.8%)

Critical Gap: Offline Operation Capabilities

Three requirements related to offline operation are marked UNSUPPORTED:

REQ-18: *"The system must provide an offline register to locally track transactions that occur during disconnection."*

REQ-19: *"The system must synchronize offline transactions with the central database once internet connectivity is restored."*

REQ-20: *"The system must allow anonymous users to perform transactions using cash when they encounter difficulties with QR code scanning."*

Impact: These gaps represent a significant operational risk, as vending machines without connectivity cannot process transactions or record sales, leading to potential revenue loss and poor user experience.
Recommendation: Prioritize architectural design for offline-first capabilities in future iterations.

2.2 Use Case to Component Mapping

2.2.1 Well-Covered Use Cases

The following use cases demonstrate strong architectural implementation:

- **UC-8 (User Login):** Fully implemented with UI, controllers, services, DAO layers, and comprehensive error handling
- **UC-10 (Buy Item):** Complete purchase flow with balance checking, inventory management, and transaction recording
- **UC-11 (Connect to Vending Machine):** QR-based connection with state management and error scenarios
- **UC-15 (Create New Vending Machine):** Admin functionality with Builder pattern for entity construction

2.2.2 Partially Covered Use Cases

Warning: UC-18 (Remote Maintenance)

Status: Partially Covered
Analysis: While controllers, services, and data layers support remote maintenance workflows, the architecture lacks explicit definition of:

- Hardware abstraction layer for device-level controls (e.g., unlocking jammed products)
- Communication protocol with vending machine firmware
- Real-time status monitoring infrastructure

Recommendation: Define an IoT/Device Gateway component to bridge software and hardware layers.

2.2.3 Orphaned/Problematic Use Cases

Warning: Orphaned Use Cases

Two use cases lack clear definition or testing:

UC-7 (User Operations): Vague description ("Interaction with users in various roles"). No distinct main flow. Appears redundant with other use cases.

Recommendation: Either remove this use case as redundant or redefine with specific, testable interactions.

UC-9 (Customer Use Cases): Empty main flow, serves as a container. No direct tests.

Recommendation: Remove container-style use cases from the model; they add no traceability value.

2.3 Test Coverage Analysis

2.3.1 Strong Test Coverage Areas

- **Authentication Flows:** 14+ tests covering login, registration, credential validation, and edge cases
- **Purchase Workflows:** 22+ tests for buy item scenarios including success, insufficient balance, out-of-stock
- **Admin Operations:** 15+ tests for analytics, machine creation, and CRUD operations
- **Worker Task Management:** 17+ tests for task completion, status updates, and error handling

2.3.2 Test Coverage Gaps

Warning: Missing Test Coverage

UC-16 (User Navigation Flow): No dedicated navigation or UI flow tests identified.

Risk: User experience issues may go undetected.

Recommendation: Implement integration tests for multi-screen workflows and navigation paths.

UC-18 (Remote Maintenance): No tests for remote unlocking or hardware interaction.

Risk: Remote maintenance features may fail in production.

Recommendation: Create mock hardware interfaces and test remote command execution.

3 Architectural Quality Assessment

3.1 Layered Architecture Evaluation

3.1.1 Strengths

Strength: Clear Layer Separation

The architecture follows a classic layered pattern:

1. **Presentation Layer:** User Interface (UI) + Mockups
2. **Controller Layer:** Request handling and routing
3. **Service Layer:** Business logic and orchestration
4. **Data Access Layer:** DAO interfaces and implementations
5. **Persistence Layer:** JPA/Hibernate + Database (PostgreSQL production, H2 test)
6. **Domain Model:** Entities with clear relationships

Benefits:

- Promotes separation of concerns
- Enables independent testing of layers
- Facilitates technology replacement (e.g., database swap)

3.1.2 Weaknesses

Critical Issue: Vague Component Responsibilities

Multiple components exhibit poorly defined or overly broad responsibilities:

DAO Layer: Described as "Vague: Manages data access and retrieval."

Problem: Too broad; violates Single Responsibility Principle.

Recommendation: Define specific responsibilities per DAO (e.g., UserDao handles user CRUD, TransactionDao handles transaction persistence).

Services: "Vague: Contains business logic and processes data."

Problem: Risks becoming a "God Object" if not carefully managed.

Recommendation: Split into domain-specific services (CustomerService, AdminService, WorkerService) with clear bounded contexts.

DB: "Vague: Handles database interactions."

Problem: Overlaps with DAO responsibilities.

Recommendation: Rename to DBConnectionManager and limit scope to connection pooling and transaction management.

3.2 Design Patterns Application

3.2.1 Effective Pattern Usage

Strength: Design Patterns

The architecture demonstrates good use of established patterns:

Builder Pattern (ConcreteVendingMachine): Excellent choice for constructing complex entities with many optional parameters. Enhances readability and maintainability.

DAO Pattern: Abstracts persistence logic, enabling testability and technology independence.

Mapper Pattern: Separates domain models from persistence entities, reducing coupling between business and data layers.

Strategy Pattern (Implicit): Different service classes for different actor types (Customer, Worker, Admin) suggests strategy-like separation.

3.2.2 Missing Patterns

Opportunity: Additional Patterns

Consider adopting additional patterns to address identified gaps:

Repository Pattern: Add a Repository layer above DAOs to encapsulate complex queries and business-oriented data retrieval (e.g., "find all machines needing maintenance").

Saga Pattern (for offline sync): When implementing REQ-18/19, use Saga pattern to manage distributed transactions and eventual consistency during offline-to-online synchronization.

Observer/Event-Driven Pattern: For real-time malfunction reporting (REQ-16), implement event listeners that react to machine status changes.

Factory Pattern: For creating different transaction types (purchase, recharge, refund), centralize creation logic.

3.3 Domain Model Quality

3.3.1 Strengths

- **Rich Domain Entities:** ConcreteVendingMachine, Inventory, Transaction, TransactionItem exhibit clear responsibilities
- **Inheritance Hierarchy:** app_user → {admin, worker, customer} models role-based access effectively
- **Composition Over Inheritance:** ConcreteVendingMachine → Inventory demonstrates proper entity composition
- **Value Objects:** MachineStatus (enum) encapsulates machine state as an immutable value

3.3.2 Areas for Improvement

Recommendation: Domain Model Enhancements

1. **Add Aggregate Roots:** Define ConcreteVendingMachine as an aggregate root that controls access to Inventory and ensures invariants (e.g., inventory cannot be modified directly without going through the machine).
2. **Introduce Domain Events:** Model events like ProductPurchased, BalanceRecharged, TaskCompleted to enable event-driven architecture and auditing.
3. **Explicit Business Rules:** Encapsulate rules like "minimum balance for purchase" or "maximum items per transaction" as domain services or specifications.
4. **Validation Objects:** Create validation value objects (e.g., Email, PositiveAmount) to enforce invariants at the type level.

4 Requirements Quality Analysis

4.1 Well-Defined Requirements

59 requirements are marked as "Well-defined" with clear acceptance criteria:

- REQ-2: "scan a unique QR code" - testable, specific
- REQ-7: "track inventory levels in real-time" - measurable
- REQ-32: "deduct the appropriate amount from the customer's balance and dispense the selected product" - atomic, testable

4.2 Requirements Needing Improvement

Warning: Vague/Unquantified Requirements

Several requirements lack quantifiable metrics:

REQ-8: "support digital payment methods"

Issue: "Needs Detail – specify which digital payment methods are supported."

Recommendation: Specify payment providers (e.g., credit card, PayPal, Apple Pay) and compliance requirements (PCI-DSS).

REQ-10: "improve machine management and user experience"

Issue: "Vague/Unquantified – provide specific metrics."

Recommendation: Define measurable goals: "Reduce average maintenance response time by 30%" or "Increase user satisfaction score above 4.0/5.0."

REQ-21: "infrastructure costs... improving maintenance speed over time"

Issue: "Needs Detail – specify metrics for cost amortization and maintenance speed."

Recommendation: Set targets: "ROI within 24 months" and "Reduce maintenance visit duration by 20%."

REQ-58: "focus on operational efficiency and usability"

Issue: "Needs Detail - specify metrics."

Recommendation: Define usability metrics: "Task completion time $<$ 30 seconds" and "Error rate $<$ 2%."

REQ-62: "enable future development features such as remote maintenance"

Issue: "Vague/Unquantified – specify what constitutes 'remote maintenance' capabilities."

Recommendation: List specific future features: remote reboot, diagnostic log retrieval, configuration updates.

4.3 Missing Non-Functional Requirements

Critical Gap: Non-Functional Requirements

The requirements specification lacks critical non-functional categories:

Performance: No response time, throughput, or latency requirements

Recommended NFRs:

- API response time < 200ms for 95th percentile
- Support 1000 concurrent vending machine connections
- Database query time < 100ms for dashboard analytics

Scalability: No guidance on expected growth or load

Recommended NFRs:

- Support horizontal scaling to 10,000+ machines
- Handle 100,000+ daily transactions
- Linear scale-out capability for geographic expansion

Security: No authentication strength, encryption, or compliance requirements

Recommended NFRs:

- Passwords hashed with bcrypt (cost factor 12+)
- TLS 1.3 for all API communications
- GDPR compliance for user data
- PCI-DSS compliance for payment data

Availability: No uptime targets or disaster recovery requirements

Recommended NFRs:

- 99.9% uptime SLA (8.76 hours downtime/year)
- Recovery Time Objective (RTO) < 1 hour
- Recovery Point Objective (RPO) < 15 minutes

Maintainability: No code quality or testing standards

Recommended NFRs:

- Code coverage > 80%
- Cyclomatic complexity < 10 per method
- All public APIs documented with OpenAPI 3.0

5 Testing Strategy Assessment

5.1 Test Pyramid Balance

Test Distribution

Observed Distribution (63 tests):

- Unit Tests: 45 tests (71%)
- Integration Tests: 12 tests (19%)
- System/End-to-End Tests: 6 tests (10%)

Analysis: The distribution roughly follows the ideal test pyramid (70/20/10 ratio), which is positive for maintainability and fast feedback.

5.2 Testing Strengths

- **Comprehensive DAO Testing:** Each DAO implementation has dedicated CRUD tests
- **Service Layer Isolation:** Services are tested with mocked dependencies (e.g., CustomerServiceTest uses Mockito)
- **Error Path Coverage:** Alternative flows and exceptions are tested (e.g., insufficient balance, out of stock)
- **In-Memory Testing:** H2 database enables fast integration tests without external dependencies
- **Code Coverage Monitoring:** JaCoCo plugin tracks coverage metrics

5.3 Testing Gaps and Recommendations

Recommendation: Expand Test Coverage

1. **Add Contract Tests:** For APIs consumed by mobile apps and vending machines, implement consumer-driven contract tests (e.g., Pact) to prevent breaking changes.
2. **Performance Tests:** Create load tests for critical paths:
   - Simulate 1000+ concurrent purchases
   - Test database query performance under load
   - Measure API latency under stress
3. **Security Tests:** Implement:
   - SQL injection tests for all DAO methods
   - Authentication bypass attempts
   - Authorization boundary tests (customer accessing admin endpoints)
4. **End-to-End Scenarios:** Add full workflow tests:
   - User registers → recharges wallet → scans QR → purchases → views history
   - Admin creates machine → worker receives task → completes maintenance → admin views report
5. **Chaos Engineering:** Test resilience:
   - Database connection failures
   - Network partition between services
   - Transaction rollback scenarios

6 Critical Weak Spots and Risks

6.1 High-Priority Risks

Risk ID	**Description**	**Impact**	**Mitigation Priority**
RISK-01	**Offline Operation Gap:** REQ-18, REQ-19, REQ-20 unsupported. Vending machines cannot operate without connectivity.	Revenue loss during outages. Poor UX. Data inconsistency.	CRITICAL
RISK-02	**Vague Component Responsibilities:** DAO, Services, DB components lack clear boundaries. Risks "God Object" anti-pattern.	Technical debt. Maintenance difficulty. Violates SRP.	HIGH
RISK-03	**Missing Non-Functional Requirements:** No performance, security, scalability targets.	Production failures. Security vulnerabilities. Poor user experience.	CRITICAL

Risk ID	**Description**	**Impact**	**Mitigation Priority**
RISK-04	**Partial Remote Maintenance Implementation:** UC-18 lacks hardware abstraction. Cannot actually unlock jammed products remotely.	Maintenance promises unfulfilled. Worker inefficiency persists.	**HIGH**
RISK-05	**Missing Navigation Tests:** UC-16 has no test coverage. UI flow bugs may go undetected.	Poor user experience. Navigation failures in production.	**MEDIUM**
RISK-06	**Orphaned Use Cases:** UC-7 and UC-9 add no value and create confusion.	Model bloat. Maintenance overhead.	**LOW**
RISK-07	**No Aggregate Root Enforcement:** Domain model lacks invariant protection. Inventory can potentially be modified directly.	Data corruption. Business rule violations.	**MEDIUM**
RISK-08	**Lack of Event-Driven Architecture:** No events for auditing or async processing. Tight coupling between components.	Scalability bottlenecks. Poor auditability. Difficult to add features.	**MEDIUM**
RISK-09	**Insufficient Error Handling Details:** REQ-34, REQ-35, REQ-45 lack specific error response formats.	Inconsistent error messages. Poor API usability.	**LOW**
RISK-10	**No Payment Provider Specification:** REQ-8 does not define which payment methods or providers are supported.	Integration delays. Compliance risks.	**HIGH**

Table 1: Critical Risks and Weak Spots

7 Recommendations and Action Plan

7.1 Immediate Actions (Sprint 1-2)

Priority 1: Address Critical Gaps

1. **Define Non-Functional Requirements**
   - Create NFR document covering performance, security, scalability, availability
   - Set measurable targets for each category
   - Map NFRs to architectural components
   - **Owner:** System Architect + Product Manager
   - **Deadline:** 2 weeks
2. **Clarify Component Responsibilities**
   - Refactor DAO, Services, DB components with explicit single responsibilities
   - Document interfaces and contracts
   - Update architectural diagrams
   - **Owner:** Technical Lead
   - **Deadline:** 1 week
3. **Specify Payment Methods (REQ-8)**
   - List supported payment providers (Stripe, PayPal, etc.)
   - Define PCI-DSS compliance requirements
   - Update architecture with payment gateway component
   - **Owner:** Product Manager + Security Team
   - **Deadline:** 1 week

7.2 Short-Term Actions (Sprint 3-5)

Priority 2: Enhance Architecture

1. **Design Offline Operation Architecture**
   - Research local storage solutions (SQLite on vending machine)
   - Design sync protocol (event sourcing, conflict resolution)
   - Implement offline register prototype
   - **Addresses:** REQ-18, REQ-19, REQ-20, RISK-01
   - **Owner:** Senior Developer + Architect
2. **Implement Hardware Abstraction Layer**
   - Define device gateway API for vending machine control
   - Mock hardware interface for testing
   - Complete UC-18 implementation
   - **Addresses:** RISK-04
   - **Owner:** Embedded Systems Team + Backend Team
3. **Add Navigation Integration Tests**
   - Create Selenium/Cypress tests for multi-screen flows
   - Cover customer, worker, admin navigation paths
   - **Addresses:** UC-16, RISK-05
   - **Owner:** QA Team
4. **Refactor Domain Model**
   - Enforce aggregate root pattern on ConcreteVendingMachine
   - Introduce domain events (ProductPurchased, etc.)
   - Add value objects for validation
   - **Addresses:** RISK-07, RISK-08
   - **Owner:** Development Team

7.3 Medium-Term Actions (Sprint 6-10)

Priority 3: Continuous Improvement

1. **Implement Event-Driven Architecture**
   - Introduce message broker (Kafka, RabbitMQ)
   - Publish domain events for audit and async processing
   - Decouple maintenance task generation from machine status changes
2. **Expand Test Coverage**
   - Add contract tests (Pact)
   - Implement performance tests (JMeter)
   - Create chaos engineering experiments (Chaos Monkey)
   - Target 85%+ code coverage
3. **Clean Up Model**
   - Remove orphaned use cases (UC-7, UC-9)
   - Add missing alternative flows
   - Ensure all use cases have clear acceptance criteria
4. **Document API Contracts**
   - Create OpenAPI 3.0 specifications
   - Version all APIs
   - Implement API versioning strategy
5. **Security Hardening**
   - Implement rate limiting
   - Add input validation middleware
   - Conduct penetration testing
   - Encrypt sensitive data at rest

8 Best Practices Checklist

8.1 Architectural Principles

Principle	Status	Evidence/Notes
SOLID Principles		
Single Responsibility Principle	PARTIAL	Some components (DAO, Services) have vague responsibilities
Open/Closed Principle	GOOD	DAO interfaces allow extension; Builder pattern supports customization

Principle	**Status**	**Evidence/Notes**
Liskov Substitution Principle	GOOD	User hierarchy (admin, worker, customer) properly extends app_user
Interface Segregation Principle	GOOD	Separate DAO interfaces per entity type
Dependency Inversion Principle	GOOD	Services depend on DAO interfaces, not concrete implementations
Layered Architecture Principles		
Clear layer separation	GOOD	Controller-Service-DAO-Database layers well-defined
No layer skipping	GOOD	Controllers call Services; Services call DAOs
One-way dependencies	GOOD	Upper layers depend on lower layers, not vice versa
Domain-Driven Design		
Ubiquitous language	GOOD	ConcreteVendingMachine, Inventory, Transaction match business terms
Bounded contexts	PARTIAL	No explicit context boundaries; all in one monolith
Aggregate roots	MISSING	Not enforced; ConcreteVendingMachine should guard Inventory
Domain events	MISSING	No events for ProductPurchased, TaskCompleted, etc.
Testing Best Practices		
Test pyramid balance	GOOD	70/20/10 unit/integration/system distribution
Arrange-Act-Assert pattern	GOOD	Tests follow AAA structure
Mocking external dependencies	GOOD	Mockito used to isolate services
In-memory databases for integration tests	GOOD	H2 enables fast, repeatable tests
Code coverage monitoring	GOOD	JaCoCo tracks coverage
Performance tests	MISSING	No load or stress tests
Security tests	MISSING	No penetration or injection tests
Requirements Engineering		

Principle	**Status**	**Evidence/Notes**
Clear acceptance criteria	GOOD	Most functional requirements well-defined
Testable requirements	GOOD	Requirements mapped to tests
Non-functional requirements	MISSING	No performance, security, scalability NFRs
Traceability matrix	EXCELLENT	Comprehensive REQ→UC→Component→Test mapping
Design Patterns		
DAO Pattern	GOOD	Abstracts persistence
Builder Pattern	EXCELLENT	ConcreteVendingMachine construction
Mapper Pattern	GOOD	Separates domain and persistence models
Repository Pattern	MISSING	Could improve complex queries
Factory Pattern	MISSING	Transaction creation could be centralized
Observer/Event Pattern	MISSING	Would enable real-time notifications
Saga Pattern	MISSING	Needed for offline sync (REQ-18/19)

Table 2: Best Practices Compliance Assessment

9 Conclusion

9.1 Summary of Findings

The JavaBrew vending machine platform demonstrates a **solid foundation** with strong traceability, good layered architecture, and comprehensive test coverage. However, **critical gaps** in offline operation support, non-functional requirements, and component responsibility definition pose significant risks to production readiness.

9.2 Overall Maturity Rating

Architecture Maturity: 6.5/10

Breakdown:

- Functional Coverage: 8/10 (95% requirements covered)
- Architectural Quality: 6/10 (good structure, but vague components and missing patterns)
- Test Coverage: 7/10 (strong unit/integration, weak on E2E and non-functional)
- Requirements Quality: 6/10 (functional well-defined, NFRs missing)
- Traceability: 9/10 (excellent REQ→UC→Component→Test mapping)
- Production Readiness: 4/10 (offline gaps, missing NFRs, partial UC-18)

9.3 Go/No-Go Recommendation

Conditional Go with Caveats

Recommendation: CONDITIONAL GO for limited pilot deployment.
Conditions:

1. Deploy only in environments with guaranteed connectivity (mitigates RISK-01)
2. Define and communicate NFRs before scaling beyond pilot (mitigates RISK-03)
3. Clarify component responsibilities to avoid future technical debt (mitigates RISK-02)
4. Limit remote maintenance promises until UC-18 hardware integration complete (mitigates RISK-04)

Do NOT proceed to full production until:

- Offline operation architecture designed and prototyped (REQ-18, REQ-19, REQ-20)
- Non-functional requirements documented and validated
- Security and performance testing completed

9.4 Key Strengths to Preserve

- Comprehensive traceability matrix - maintain and expand as project evolves
- Layered architecture - continue enforcing layer boundaries
- Builder pattern usage - replicate for other complex entities
- Test automation - sustain and grow test suite
- Domain-Driven Design principles - deepen with aggregates and events

9.5 Final Word

This architectural blueprint represents a **commendable effort** with strong engineering practices in many areas. By addressing the identified critical gaps—particularly offline operation, non-functional requirements, and component clarity—the team can elevate this architecture to production-grade quality. The traceability infrastructure in place provides an excellent foundation for continuous improvement and ensures that future changes remain aligned with business requirements.

Recommended Next Steps:

1. Review this report with architecture review board
2. Prioritize and schedule remediation of CRITICAL and HIGH risks
3. Update requirements specification with NFRs
4. Refactor components per Single Responsibility Principle
5. Prototype offline operation architecture
6. Re-assess maturity after Sprint 5