Software Engineering Audit Report

SWE2025_Java_TrainingHub.pdf

CAPRA

February 25, 2026

Contents

1 Document Context

Project Objective

Java-TrainingHub is a domain-driven design application that facilitates seamless collaboration between personal trainers and trainees through a unified platform for workout planning, assignment, and progress tracking. The system enables trainers to create and manage personalized workout plans while trainees can log completed sessions and receive real-time notifications of plan modifications. It supports dual-role functionality, allowing personal trainers to also operate as trainees within the same account.

Main Use Cases

- UC-1 – User Registration: User creates an account and specifies their role (Trainee or Personal Trainer).
- UC-2 – Sign-In: User authenticates with email and password to access their personalized dashboard.
- UC-3 – Add Personal Trainer: Trainee associates with a Personal Trainer to receive workout plans.
- UC-4 – Remove Personal Trainer: Trainee removes a Personal Trainer from their associated trainers list.
- UC-5 – View WorkoutRecord: User views their workout history; PT can view assigned trainees' records.
- UC-6 – View Workout Plan: Trainee views assigned workout plans with nested structure and exercises.
- UC-7 – Register Workout(4Record): Trainee logs completed workout sessions with exercise details.
- UC-8 – View Workout(4Record): User views individual workout sessions and their associated exercises.
- UC-9 – Edit Workout(4Record): User modifies recorded workout details (sets, reps, weights, notes).
- UC-10 – Delete Workout(4Record): User removes a recorded workout session from their history.
- UC-11 – Add Trainee (Follow User): Personal Trainer adds a trainee to their client roster.
- UC-12 – Remove Trainee (Unfollow): Personal Trainer removes a trainee from their followed list.
- UC-13 – Create Workout Plan (PT): Personal Trainer designs a new workout plan with sessions and exercises.
- UC-20 – Edit WorkoutPlan (Attach/Detach Workout4Plans): Personal Trainer modifies plan composition.
- UC-21 – Delete WorkoutPlan: Personal Trainer removes a workout plan and its associations.
- UC-22 – Create Workout(4Plan): Personal Trainer creates individual planned workout sessions.
- UC-23 – View Workout(4Plan): Personal Trainer views their created planned workout sessions.
- UC-24 – Edit Workout(4Plan): Personal Trainer modifies planned workout session details.
- UC-25 – Delete Workout(4Plan): Personal Trainer removes a planned workout session.
- UC-26 – View User Workout Record (PT): Personal Trainer monitors trainee progress and activity.

Functional Requirements

- User authentication and role-based access control distinguishing Trainees and Personal Trainers.
- Trainees can view assigned workout plans, log completed workouts, and track progress over time.
- Personal Trainers can create, update, and delete workout plans; assign plans to trainees; and monitor client progress.
- Support for hierarchical workout structure: WorkoutPlans contain Workout4Plans (daily sessions) containing Exercises.

- Distinction between planned workouts (WorkoutPlans/Workout4Plans) and recorded workouts (WorkoutRecords/Workout4Records).
- Exercise management with attributes including name, description, equipment, sets, reps, weight, and training strategy.
- Real-time notification mechanism to alert trainees when their workout plans are modified by trainers.
- Support for multiple training strategies (Strength, Hypertrophy, Endurance) applied dynamically to exercises.
- Trainer-trainee relationship management with follow/unfollow functionality.
- Comprehensive CRUD operations for all domain entities (users, plans, workouts, exercises, records).

Non-Functional Requirements

- Architecture: Layered three-tier architecture (Model, Business Logic, ORM) with clear separation of concerns.
- Design Patterns: Implementation of Observer, Strategy, Factory, and DAO patterns for flexibility and maintainability.
- Database: PostgreSQL relational database with normalized schema and proper foreign key constraints.
- Technology Stack: Java 11, Apache Maven, JUnit 5, Mockito, JDBC for database connectivity.
- Code Coverage: Minimum 77% line coverage (916/1183 lines) with 96% class coverage across all packages.
- Scalability: Interface-based DAO design enables database technology independence and future enhancements.
- Maintainability: Comprehensive documentation, clear naming conventions, and adherence to SOLID principles.
- Error Handling: Robust exception handling with informative error messages and proper resource cleanup.
- Data Integrity: Transaction management with rollback capabilities and constraint violation handling.
- Security: Use of parameterized queries to prevent SQL injection attacks.

Architecture

The system follows a **three-layer layered architecture** pattern:

Model Layer: Contains domain objects implementing core business logic with design patterns. User hierarchy (User base class with Trainee and PersonalTrainer subclasses) implements the Observer pattern for notifications. Workout management uses Strategy pattern for exercise intensity (Strength, Hypertrophy, Endurance) and Factory pattern for object creation (Workout4PlanFactory, Workout4RecordFactory).

Business Logic Layer: Seven specialized controllers (TraineeController, PersonalTrainerController, WorkoutPlanController, Workout4PlanController, WorkoutRecordController, Workout4RecordController, ExerciseController) orchestrate operations between Model and ORM layers. Controllers implement input validation, business rule enforcement, and dependency injection for testability.

ORM Layer: Data Access Object (DAO) pattern with interface-based design. Seven DAO interfaces with concrete implementations (ConcreteTraineeDAO, ConcretePersonalTrainerDAO, ConcreteWorkoutPlanDAO, ConcreteWorkout4PlanDAO, ConcreteWorkoutRecordDAO, ConcreteWorkout4RecordDAO, ConcreteExerciseDAO) provide database abstraction. DBManager handles centralized connection management using JDBC with PostgreSQL.

Database: PostgreSQL relational schema with core entity tables (AppUser, Personal_Trainer, WorkoutPlans, Workout4Plan, Exercises, WorkoutRecords, Workout4Record) and relationship tables managing many-to-many associations.

Testing Strategy

The project employs comprehensive testing using **JUnit 5** for unit and integration tests and **Mockito** for mock object creation.

Unit Tests: Individual classes tested in isolation with mocked dependencies. Model classes verified for constructor logic and business rules. Service/Controller classes use Mockito to mock DAO dependencies. DAO classes tested with mocked JDBC objects (Connection, PreparedStatement, ResultSet). Test suite includes 10 DAO test classes covering CRUD operations and specialized queries.

Integration Tests: Six comprehensive test cases validate end-to-end workflows including trainer-trainee relationship management, access control verification, business rule enforcement (strategy consistency), dynamic plan updates, CRUD operations on workout components, and complex multi-session workout logging. Tests use clean database state with proper foreign key constraint handling and sequence counter resets for test isolation.

Coverage Metrics: Actual code coverage of 77% line coverage (916/1183 lines), 96% class coverage (30/31 classes), and 72% method coverage (161/222 methods). ORM layer achieves 100% class and method coverage with 88% line coverage. Business Logic layer has 100% class coverage but 45% method coverage. Model layer achieves 94% class coverage with 77% line coverage.

2 Executive Summary

Quick Overview

Total issues: **12** — HIGH: **4** MEDIUM: **6** LOW: **2**

Average confidence: **94%**

Executive Summary: SWE2025_Java_TrainingHub Audit Report

This document is a Software Engineering design and implementation report for a Java-based fitness training management system (TrainingHub). The system enables trainees to manage workouts and connect with personal trainers, while trainers create and assign workout plans and monitor trainee progress. The report includes requirements, architecture, database design, and test documentation spanning approximately 50 pages.

The audit identified 12 issues across four categories. The most significant patterns are: (1) **contradictions between high-level role capabilities and concrete use cases**, leaving critical behaviors undefined; (2) **incomplete specification of pre-conditions, post-conditions, and business rules** in use case templates, particularly around ownership, authorization, and data integrity; (3) **misalignment between the domain model (inheritance-based) and persistence model (single-table-with-flag)**, with no documented mapping or verification tests; and (4) **weak test coverage for error conditions and end-to-end workflows**, with no explicit traceability from requirements to tests.

Four issues are classified as **HIGH severity**. ISS-003 reveals that the use case diagram assigns CRUD operations to trainees, but detailed use cases restrict these to trainers only, creating ambiguity about actual permissions. ISS-004 documents systemic gaps in pre/post-conditions across nearly all use cases, omitting critical rules for ownership, authorization, and constraint handling. TST-001 shows that while alternative flows and error conditions are specified in use cases, no controller-level or end-to-end tests validate these scenarios (e.g., invalid registration, unauthorized access, constraint violations). TST-002 indicates the absence of a traceability matrix linking the 26 use cases to integration tests, making it unclear which requirements are verified end-to-end.

Overall Quality Assessment: The document demonstrates solid foundational work in requirements capture and architectural design, with detailed use case descriptions and a well-structured database schema. However, the quality is undermined by internal inconsistencies, incomplete specifications, and insufficient test coverage. The domain-to-persistence mapping gap and the contradiction between role capabilities and concrete use cases are particularly concerning for correctness and maintainability. The BusinessLogic layer has only 45% method coverage, and critical workflows (registration, sign-in, user management) lack explicit end-to-end tests. These gaps suggest the implementation may not fully satisfy the stated requirements and could harbor subtle bugs in authorization and data consistency.

1. **Reconcile role capabilities and use cases:** Resolve the contradiction between the high-level capability matrix (which assigns CRUD Workout/WorkoutPlan to trainees) and the detailed use cases (which restrict these to trainers). Clarify in writing which roles can perform which operations,

update the use case diagram and templates accordingly, and add integration tests for each role-based workflow.

2. **Complete pre/post-condition specifications:** Systematically review all 26 use cases and add precise pre-conditions and post-conditions that cover ownership rules, role-based authorization checks, and data integrity constraints (e.g., what happens to assigned trainees when a plan is deleted). Document the notification and visibility semantics for UC-7 and dependent use cases.

3. **Establish requirements-to-test traceability and expand test coverage:** Create an explicit traceability matrix mapping each use case to corresponding unit, integration, or end-to-end tests. Add controller-level and integration tests for all error conditions and alternative flows (invalid input, unauthorized access, constraint violations), and increase BusinessLogic layer coverage to at least 70% method coverage. Verify the Observer pattern implementation with explicit tests that subscribe observers, trigger updates, and assert notifications.

3 Strengths

- **Comprehensive Use Case Coverage and Requirements Documentation:** The document provides 20 detailed use case templates (Section 3.1) covering all major system functionalities, from user registration and authentication to workout plan management and progress tracking. Each use case is formally documented with preconditions, postconditions, and alternative flows, demonstrating thorough requirements analysis aligned with the identified actors (Trainee and Personal Trainer).
- **Well-Defined Layered Architecture with Clear Separation of Concerns:** The implementation follows a robust three-layer architecture (Model, Business, and ORM layers) as detailed in Section 4.2. The document provides explicit architectural diagrams (Figures 10-13) and rationale explaining how domain-driven design principles are applied to isolate business logic from persistence and presentation concerns, enhancing maintainability and testability.
- **Appropriate Application of Design Patterns:** The system demonstrates correct implementation of established design patterns including Observer Pattern for real-time notifications, Strategy Pattern for exercise variations based on training goals, and Factory Pattern for modular workout component creation. These patterns are explicitly documented and justified in the requirements and implementation sections.
- **Comprehensive Testing Strategy with Multiple Test Levels:** The document outlines both unit tests (covering UserManagementTest, WorkoutManagementTest, and nine DAO tests) and integration tests for core business workflows. Table 21 provides actual code coverage metrics, demonstrating a measured and systematic approach to quality assurance across multiple packages.
- **Complete Database Design with Entity-Relationship Modeling:** Section 3.3 presents a detailed conceptual data model (Figure 9) with explicit entity descriptions and relationship analysis. The database implementation section (4.3) includes schema design, connection management, and configuration requirements, providing a solid foundation for data persistence.
- **Transparent Documentation of Limitations and Trade-offs:** Section 2.4 honestly identifies drawbacks including lack of real-time synchronization, manual database configuration, and absence of mobile support. This critical self-assessment demonstrates professional awareness of design constraints and provides a clear roadmap for future enhancements.

4 Expected Feature Coverage

Of 7 expected features: 6 present, 1 partial, 0 absent. Average coverage: **95%**.

Feature	Status	Coverage	Evidence
Unit testing framework implementation	Present	100% (5/5)	Section 5 describes JUnit 5 and Mockito usage, including unit and integration tests. Listing 7 shows assertion-based verification and isolated DAO testing with mocked JDBC and DatabaseManager. Section 5.1 and 7.1/7.2 list many dedicated test classes and methods, and Tables 21–22 report coverage for both unit and integration testing.
Use of UML Diagrams for system modeling	Present	100% (8/8)	Figure 1 is a use case diagram with actors and use cases; Figure 10 is the complete class diagram; Figures 11–13 are structure diagrams for Model, Business, and ORM layers. Section 3.2 and Figures 2–8 provide UI mockups and navigation flows, and the text explicitly mentions StarUML and standardized UML notation to clarify functional requirements and actor–use case relationships.
Identification and definition of system actors	Present	100% (8/8)	Section 2.2 'Involved Actors' clearly defines Trainee, Personal Trainer, and System Administrator (implicit) with their responsibilities. Section 2.3 and the use case tables (UC-1..UC-26) document interactions and specific user actions, and Figure 1 plus the surrounding text show how different roles map to system functionalities and requirements.
Definition and Documentation of Use Cases	Present	100% (5/5)	Section 3.1 provides detailed use case templates UC-1 to UC-26, each with Id, Name, Level, Actors, Pre-conditions, Post-conditions, Basic flow, and Alternative flow. Figure 1 and the 'Key Observations' text describe relationships between use cases (e.g., includes) and how they support user goals and interactions.

Feature	Status	Coverage	Evidence
User interface and interaction design principles	Partial	71% (5/7)	Section 3.2 and Figures 2–8 show UI mockups for login, trainer selection, My Plan, Record Workout, Add Trainee, Create Plan, and Workout Records, with clear navigation elements and structured input fields. Roles and their functionalities are reflected in separate trainee and PT interfaces, and some dynamic updates (e.g., TODAY highlighting, workout lists updating) are described. However, there is no reservation-creation process in this domain, and explicit UI-level validation mechanisms are not detailed in the mockup descriptions.
Separation of concerns in software architecture	Present	100% (5/5)	Section 4.2 describes a three-layer architecture (Model, Business, ORM) and Figure 10 shows distinct packages BusinessLogic, Model, and ORM. Sections 4.2.1–4.2.3 document interactions between controllers and domain models and between controllers and DAOs, clearly describing each component's role and emphasizing modular, maintainable design.
Data Access Object (DAO) pattern	Present	100% (7/7)	Figure 13 and Section 4.2.3 describe DAO interfaces (e.g., WorkoutPlanDAO, TraineeDAO, ExerciseDAO) and their concrete implementations with CRUD methods. Section 4.2.9 and Listing 3 show SQL encapsulated in DAO classes and abstraction of database access. Section 5 and 7.1.4–7.1.10 list dedicated DAO test classes (e.g., ExerciseDAOTest, WorkoutPlanDAOTest), documenting testing strategies for data access methods.

5 Summary Table

Category	HIGH	MEDIUM	LOW	Total
Architecture	0	1	1	2
Requirements	2	4	0	6
Testing	2	1	1	4
Total	**4**	**6**	**2**	**12**

6 Issue Details

6.1 Architecture (2 issues)

UC-7

ISS-002 — LOW [100%] — Page 31 The business layer class diagram shows controllers that expose low-level operations (e.g., linkWorkout4RecordToWorkoutRecord, addExerciseToWorkout4Record) that are closer to persistence concerns than to user-level use cases. However, the requirements and use cases are written at a higher level (e.g., "Register Workout(4Record)", "Edit Workout(4Record)") and do not describe these linking operations explicitly. This mismatch between the granularity of the business API and the use cases can make it harder to trace how each controller method supports a specific requirement and may lead to overexposing internal operations.

> *WorkoutRecordController +workoutRecordDAO +createWorkoutRecord() +getWorkoutRecordById() +getAllWorkoutRecords() +deleteWorkoutRecord() +linkWorkout4RecordToWorkoutRecord() +getWorkout4RecordsForWorkoutRecord() ... Workout4RecordController +workout4RecordDAO +createWorkout4Record() +getAllWorkout4Records() +getWorkout4RecordById() +updateWorkout4RecordDate() +deleteWorkout4Record() +addExerciseToWorkout4Record() +getExercisesForWorkout4Record()*

Recommendation: Improve traceability between use cases and business-layer operations: 1) For each use case (especially UC-7–UC-10 and UC-26), identify the sequence of controller methods that implement it (e.g., UC-7 might call createWorkout4Record, linkWorkout4RecordToWorkoutRecord, and addExerciseToWorkout4Record). 2) Document this mapping briefly in the System Design section or as a small table, so that reviewers can see how high-level requirements are realized. 3) Consider introducing higher-level service methods (e.g., logCompletedWorkoutSession) that encapsulate the necessary low-level calls, and reference these methods from the use cases; this will make the architecture closer to the business language and simplify future changes. 4) Optionally, hide purely technical linking methods from the public API if they are not meant to be invoked directly by the presentation layer, to keep the boundary aligned with the use cases.

General Issues

ISS-001 — MEDIUM [90%] — Page 29 The architecture section describes a User base class with Trainee and PersonalTrainer subclasses, but the database schema and ORM layer model users differently: AppUser has an is_pt flag and Personal_Trainer is a 1:1 extension table. This means the persistence model is closer to a single-table-with-role-flag design, while the domain model is inheritance-based. The report does not explain how these two representations are mapped or reconciled, and there are no tests that verify the correctness of this mapping (e.g., that a PersonalTrainer domain object is correctly persisted and re-hydrated from AppUser + Personal_Trainer rows). This is an architectural consistency gap that could confuse readers and hide potential bugs.

> *The user management subsystem implements a hierarchical structure with polymorphic behavior: • User (Base Class): Contains core attributes including Id, Name, Age, WorkoutRecord, and PersonalTrainers. This serves as the foundation for all user types in the system. • Trainee (Concrete Observer A): Extends the User class and implements the Observer pattern in- terface. • Personal Trainer (Concrete Observer B): Also extends User and implements Observer pattern.*

Recommendation: Add a short subsection in the Architecture or ORM chapter explicitly explaining how the User/Trainee/PersonalTrainer class hierarchy is mapped to the AppUser and

Personal_Trainer tables (e.g., which DAO constructs which subclass based on is_pt, how 1:1 relationships are handled). Then, add one or two focused tests (unit or integration) that verify this mapping: for example, create a PersonalTrainer via the controller, persist it, reload it via the DAO, and assert that the correct subclass and role information are reconstructed. This will make the design more coherent and demonstrate that the inheritance-based model and the relational schema are aligned.

6.2 Requirements (6 issues)

UC-9

ISS-004 — HIGH [86%] — Page 6 Across almost all use case templates, pre-conditions and post-conditions are only partially specified and do not cover important business rules such as ownership, role-based authorization, and data integrity constraints. For example, UC-9 and UC-10 only state that the user "owns or has permission" but do not define what ownership means or how permissions are determined; UC-13–UC-15 do not state what happens to trainees already assigned to a plan when it is edited or deleted; UC-7 mentions that a PT "may is notified" without specifying under which conditions or via which mechanism. This systemic lack of precise pre/post conditions makes it hard to verify correctness and to derive complete tests.

> *3.1.1 User Registration Field Description Id UC-1 Name User Registration Level System Goal Actors User (Trainee or PT) Pre-conditions The user is not authenticated and does not have an account. Post-conditions The user has a profile (Trainee or PT specified), is authenticated, and has access to the personalized system interface. ... 3.1.2 Sign-In ... 3.1.3 Add Personal Trainer ... 3.1.4 Remove Personal Trainer ... 3.1.5 View WorkoutRecord ... 3.1.6 View Workout Plan ... 3.1.7 Register Workout(4Record) ... 3.1.8 View Workout(4Record) ... 3.1.9 Edit Workout(4Record) ... 3.1.10 Delete Workout(4Record) ... 3.1.11 Add Trainee (Follow User) ... 3.1.12 Remove Trainee (Unfollow) ... 3.1.13 Create Workout Plan (PT) ... 3.1.14 Edit WorkoutPlan (Attach/Detach Workout4Plans) ... 3.1.15 Delete WorkoutPlan ... 3.1.16 Create Workout(4Plan) ... 3.1.17 View Workout(4Plan) ... 3.1.18 Edit Workout(4Plan) ... 3.1.19 Delete Workout(4Plan) ... 3.1.20 View User Workout Record (PT)*

Recommendation: Systematically strengthen pre-conditions and post-conditions for all use cases: 1) For all CRUD operations on shared resources (WorkoutRecord, Workout4Record, WorkoutPlan, Workout4Plan), explicitly define: - Ownership rules (e.g., "The Workout4Record belongs to the authenticated Trainee" or "The PT is the creator of the WorkoutPlan"). - Authorization rules (e.g., "PT must follow the trainee to view their records"). - Data integrity effects (e.g., cascading deletions, updates to N_workouts, last_edit_date). 2) For UC-7, UC-13–UC-15, UC-20, clarify notification and propagation behavior, e.g., "On successful save, if the Workout4Record is linked to a plan assigned by a PT, the PT is notified via the Observer mechanism" and "When a WorkoutPlan is deleted, all assignments to trainees are removed and trainees no longer see it in 'My Plans'." 3) Add explicit post-conditions for list-view use cases (UC-5, UC-6, UC-8, UC-17, UC-23, UC-26) describing what is guaranteed about the returned data (e.g., sorted order, completeness, filtering by role). 4) Use consistent wording for authorization (e.g., always refer to "follows" relationship when PT accesses trainee data) and ensure it matches the ER model and DAOs.

UC-3

ISS-005 — MEDIUM [68%] — Page 7 In several use cases the "Actors" field is broader than the actual pre-conditions and basic flow. For UC-3 and UC-4, the actor is listed as "User (Trainee or PT)", but the pre-conditions and steps clearly assume a Trainee ("Trainee navigates", "Trainee opens 'My Trainers'"). Similarly, UC-5, UC-6, UC-7, UC-8, UC-9, and UC-10 list "User (Trainee or Personal Trainer)" but the flows and constraints differ by role and are not fully separated.

This ambiguity about which roles can initiate which use cases and under what conditions can lead to inconsistent implementations and tests.

> *3.1.3 Add Personal Trainer Field Description Id UC-3 Name Add Personal Trainer Level User Goal Actors User (Trainee or PT) ... Pre-conditions The Trainee is authenticated and not already followed by the selected PT. ... 3.1.4 Remove Personal Trainer Field Description Id UC-4 Name Remove Personal Trainer Level User Goal Actors User (Trainee or PT) Pre-conditions Trainee is authenticated and has at least one associated PT.*

Recommendation: Clarify actor roles per use case and, where necessary, split or parameterize them: 1) For UC-3 and UC-4, change the Actors field to "Trainee" (or "User in Trainee role") to match the pre-conditions and flows, unless you explicitly want PTs to add/remove their own trainers; in that case, add separate flows for PT behavior. 2) For UC-5–UC-10, either: - Split into separate use cases per role (e.g., "View Own WorkoutRecord (Trainee)" and "View Trainee WorkoutRecord (PT)") with distinct pre-conditions and flows, or - Keep a single use case but add explicit sub-flows for each role, and refine pre-conditions to state which role is assumed in each step. 3) Ensure the use case diagram (Figure 1) and "Actors and Capabilities" section reflect the final actor assignments so that every use case has a clearly defined primary actor and role-specific behavior.

UC-7

ISS-006 — MEDIUM [95%] — Page 8 UC-7 introduces a key business rule about trainer notifications ("PT may is notified and can view this record"), but the condition under which a PT is notified is not specified (e.g., only if the workout is linked to a plan assigned by that PT? any PT following the trainee?). Other use cases that depend on this behavior, such as UC-5 (View WorkoutRecord) and UC-26 (View Trainee Workout Record), do not mention notifications or how new records become visible. This leaves the notification and visibility semantics under-specified and makes it hard to verify the Observer pattern implementation against requirements.

> *3.1.7 Register Workout(4Record) ... Post-conditions The completed workout session (Workout4Record) is logged and associated with the User's workout records. PT may is notified and can view this record.*

Recommendation: Make the notification and visibility rules explicit and consistent: 1) Refine UC-7 post-conditions to specify exactly when and whom to notify, for example: "If the Workout4Record is associated with a WorkoutPlan assigned by a PT, that PT is notified" or "All PTs who follow the trainee are notified of new Workout4Records". 2) In UC-5 and UC-26, add notes or post-conditions clarifying that newly logged Workout4Records become immediately visible to authorized PTs according to the same rule. 3) Optionally add a dedicated use case (e.g., "Receive Workout Notifications") describing how observers are registered and how notifications are delivered (even if only conceptually in the current CLI version). 4) Ensure the Observer pattern description in Section 4.2.1 and the integration tests in 5.3.2 explicitly reference and validate these clarified rules.

UC-1

ISS-008 — MEDIUM [100%] — Page 47 The integration test list covers several core workflows (viewing assigned plans, PT viewing trainee records, strategy consistency, updating plans, editing Workout4Plans, logging sessions), but some important use cases have no clearly corresponding integration tests. In particular, there is no explicit integration test for UC-1 (User Registration), UC-2 (Sign-In), UC-3/UC-4 (Add/Remove Personal Trainer), UC-11/UC-12 (Add/Remove Trainee), or UC-15/UC-21 (Delete WorkoutPlan with its associations). Given that these are central to user management and access control, the absence of end-to-end tests for them weakens the validation of the most critical requirements.

> *7.2 Integration Tests • testTraineeCanViewWorkoutPlanAssignedByTheirPT • testPersonalTrainerCanViewTraineeWorkoutRecords • testExercisesInsideWorkout4PlansAreOfSameStrategy • testPTUpdatesWorkoutPlanAssignedToTrainee • testDeleteAndEditWorkout4PlanFromWorkoutPlan • testAddWorkoutSessionsWithMultipleExercisesToUserWorkoutRecord*

Recommendation: Extend the integration test suite to cover the missing critical use cases: 1) Add an integration test for UC-1 and UC-2 that: - Registers a new user (Trainee and PT variants if both are supported), - Verifies that the user is persisted correctly and can subsequently sign in using the same credentials. 2) Add tests for UC-3/UC-4 and UC-11/UC-12 that: - Create a PT and a Trainee, - Establish and remove trainer-trainee relationships using the appropriate controllers/DAOs, - Verify that access to plans and records respects these relationships (e.g., PT cannot view records after unfollow). 3) Add a test for UC-15/UC-21 that deletes a WorkoutPlan already assigned to trainees and linked to Workout4Plans, and verifies that: - The plan is no longer visible in "My Plans" for the trainee, - All junction table entries are removed, - No orphaned references remain. 4) In the testing documentation, explicitly map each integration test to the corresponding UC IDs so that reviewers can see which requirements are covered and which are not.

See also: TST-002

General Issues

ISS-003 — HIGH [100%] — Page 4 The use case diagram description and capability list assign CRUD Workout and CRUD WorkoutPlan to Trainees, and allow Personal Trainers to Create/Edit/Delete WorkoutRecord, but the detailed use cases later restrict plan creation and workout plan CRUD to Personal Trainers only, and do not define any use case where a Trainee can CRUD Workout or WorkoutPlan. This creates a contradiction between high-level role capabilities and the concrete use cases, and leaves important behaviors (e.g., whether trainees can manage their own plans or exercises) undefined.

> *Actors and Capabilities 1. Trainee (User) Core capabilities include: • Sign-In: Access the system • Add/Remove Personal Trainer: Manage trainer associations • View WorkoutPlan: See training schedules • CRUD WorkoutRecord: Create, read, update, delete exercise logs • CRUD Workout: Manage exercise definitions • CRUD WorkoutPlan: Manage training plans • View UserWorkoutRecord: Access workout histories 2. Personal Trainer Core capabilities include: • Add/Remove User: Manage followed Trainees • Create/Edit/Delete WorkoutRecord: Manage exercise logs • Create/Edit/Delete WorkoutPlan: Design training programs*

Recommendation: Align the role capabilities with the detailed use cases. Concretely: 1) Decide the intended behavior: - If only Personal Trainers can create/edit/delete WorkoutPlans and define Workouts, then remove "CRUD Workout" and "CRUD WorkoutPlan" from the Trainee capabilities list and from the diagram, and ensure the text explicitly states that trainees can only view plans and log workouts. - If Trainees are supposed to manage their own plans or exercises, add corresponding use cases (e.g., "Create Personal Workout Plan (Trainee)", "Manage Personal Exercises") with full templates (pre-conditions, post-conditions, flows) and adjust access control rules accordingly. 2) Update Figure 1 and the "Actors and Capabilities" section so that every listed capability has a matching detailed use case and consistent role restrictions. 3) Re-scan the document for any other references to CRUD Workout/WorkoutPlan and make them consistent with the final decision.

ISS-007 — MEDIUM [85%] — Page 44 The database schema defines both direct foreign keys (Workout4Record.wr_id, Personal_Trainer.pt_id referencing AppUser) and additional junction tables marked as "Potentially redundant" (Workout4Record_WorkoutRecords, WorkoutRecords_AppUser). However, the requirements and use cases never explain whether a

Workout4Record can belong to multiple WorkoutRecords, or whether a WorkoutRecords can be shared among users, which would justify these many-to-many tables. This inconsistency between the conceptual model (1:1 and 1:n relationships) and the physical schema introduces ambiguity about the intended cardinalities and can cause confusion when implementing or testing business logic.

> *46 – Workout4Record Table (Represents a specific workout session that was 47 CREATE TABLE Workout4Record (48 w4r_id SERIAL PRIMARY KEY, 49 wr_id INT REFERENCES WorkoutRecords(wr_id) ON DELETE CASCADE, 50 date DATE NOT NULL 51); ... 30 – Link Workout4Records to WorkoutRecords (Potentially redundant) 31 CREATE TABLE Workout4Record_WorkoutRecords (32 w4r_id INT REFERENCES Workout4Record(w4r_id) ON DELETE CASCADE, 33 wr_id INT REFERENCES WorkoutRecords(wr_id) ON DELETE CASCADE, 34 PRIMARY KEY (w4r_id, wr_id) 35); ... 37 – Link WorkoutRecords to AppUsers (Potentially redundant) 38 CREATE TABLE WorkoutRecords_AppUser (39 wr_id INT REFERENCES WorkoutRecords(wr_id) ON DELETE CASCADE, 40 user_id INT REFERENCES AppUser(user_id) ON DELETE CASCADE, 41 PRIMARY KEY (wr_id, user_id) 42);*

Recommendation: Clarify and align the intended cardinalities between users, workout histories, and sessions: 1) Decide whether Workout4Record should belong to exactly one WorkoutRecords and whether a WorkoutRecords should belong to exactly one AppUser (as suggested by the ER model). If so, remove or clearly deprecate the "potentially redundant" junction tables from the schema and documentation. 2) If you actually need many-to-many relationships (e.g., shared histories or sessions), update Section 3.3.2 to describe these use cases and adjust the ER diagram accordingly. 3) Reflect the final decision in the requirements/use cases (e.g., state that each workout session is part of exactly one user's history) so that developers and testers know which relationships to rely on. 4) Update DAO methods and integration tests to use only the chosen relationship mechanism (direct FK or junction table) and remove dead paths to avoid inconsistent data access patterns.

6.3 Testing (4 issues)

UC-1

TST-002 — HIGH [100%] — Page 47 There is no explicit traceability matrix or mapping from the detailed use cases (UC-1..UC-26) to the described integration tests, so it is unclear which requirements are actually verified end-to-end. For example, critical flows like UC-1 User Registration and UC-2 Sign-In are defined in detail but no corresponding tests are listed, and the six integration tests are only described at a high level without linking them to specific UCs. This weakens requirements-to-test traceability and makes it hard to argue complete coverage for grading.

> *The comprehensive test suite validates end-to-end functionality through six distinct test cases that cover the core business workflows of the fitness application.*

Recommendation: Add a simple traceability table that maps each primary use case (UC-1..UC-26) to one or more concrete test methods. For each integration test (e.g., testTraineeCanViewWorkoutPlanAssignedByTheirPT, testPersonalTrainerCanViewTrainee-WorkoutRecords, testAddWorkoutSessionsWithMultipleExercisesToUserWorkoutRecord), explicitly state in the Testing chapter which UC(s) it covers. For important UCs that currently have no integration test (e.g., UC-1 User Registration, UC-2 Sign-In, UC-3/UC-4 Add/Remove Personal Trainer, UC-11/UC-12 Add/Remove Trainee), either add a small integration test that exercises the full flow through controllers and DAOs, or clearly justify why they are out of scope. This can be done in a one-page table without changing code but will significantly improve traceability in the report.

See also: ISS-008

General Issues

TST-001 — HIGH [100%] — Page 6 Across the detailed use cases, alternative flows and error conditions (validation errors, unauthorized access, missing entities, constraint violations) are carefully specified, but the documented tests focus almost entirely on happy paths and DAO-level SQL edge cases. There are no explicit controller-level or end-to-end tests that assert the system's behavior for these alternative flows, such as invalid registration data, duplicate emails, incorrect login credentials, unauthorized access to non-followed trainees, or attempts to delete plans that are in use.

> *Alternative flow • If validation errors occur, system displays an error message. • If email already exists, system displays an error message. • If a system error occurs, display an appropriate message.*

Recommendation: Systematically add negative/alternative-flow tests at the controller or integration level for the most important error scenarios. For example: for UC-1 and UC-2, add tests like testSignIn_WithWrongPassword_ShowsError and testRegistration_WithExistingEmail_Fails; for access-controlled UCs (UC-5, UC-8, UC-12, UC-26), add tests that attempt to access or modify data for a trainee not followed by the PT and assert that an error is returned; for deletion UCs (UC-15, UC-19, UC-10), add tests that try to delete non-existent or constrained entities and verify the correct error handling. You can keep these tests lightweight (e.g., checking returned null, boolean flags, or error messages printed to stderr), but explicitly link them to the alternative flows described in the UC tables.

TST-003 — MEDIUM [100%] — Page 53 The unit tests for the Model layer (UserManagementTest, WorkoutManagementTest) validate internal behavior of domain objects and strategies, but they do not explicitly verify the Observer pattern behavior that is central to the requirements (real-time notifications when plans are modified). The report describes Trainee and PersonalTrainer as concrete observers and WorkoutPlan/WorkoutRecord as subjects, yet there is no test that subscribes observers, triggers a plan update, and asserts that observers are notified or their state changes accordingly.

> *7.1.1 UserManagementTest • testPersonalTrainerFollowedUsers_ ValidUsers_ AddedSuccessfully • testPersonalTrainerWorkoutPlans_ ValidPlans_ AddedSuccessfully • testUserWorkoutRecord_ ValidRecords_ HandledCorrectly*

Recommendation: Add at least one focused unit test that exercises the Observer pattern end-to-end in the model. For example, create a WorkoutPlan (as subject), attach a Trainee (as concrete observer), perform an operation that should trigger notification (e.g., adding a Workout4Plan or changing LastEditDate), and assert that the Trainee's internal state or a notification flag is updated. Document this test under UserManagementTest or a dedicated ObserverPatternTest and explicitly mention in the Testing chapter that it validates the notification requirement described in the Requirements Analysis and Model sections.

TST-004 — LOW [100%] — Page 47 The coverage table shows that the BusinessLogic package has only 45% method coverage and 38% line coverage, significantly lower than the Model and ORM layers. Given that controllers implement important validation and orchestration logic (e.g., in Workout4PlanController, TraineeController, PersonalTrainerController), this suggests that many branches and methods in the business layer are not exercised by unit tests, even though some are indirectly touched by integration tests.

> *Package Class % Method % Line % BusinessLogic 100% (7/7) 45% (27/60) 38% (86/226)*

Recommendation: Identify the most important controller methods that are currently untested (you can use the IDE's coverage report) and add a few targeted unit tests with mocked DAOs to raise BusinessLogic coverage. Focus on methods that contain non-trivial logic, such as input validation, strategy compatibility checks, and relationship management (e.g., followTrainee, getWorkoutPlansByTraineeId, addWorkout4PlanToWorkoutPlan). You do not need to reach 100%, but bringing method/line coverage for BusinessLogic closer to 60–70% will better reflect the importance of this layer and strengthen your argument about overall test quality.

7 Priority Recommendations

The following actions are considered priority:

1. **ISS-003** (p. 4): Align the role capabilities with the detailed use cases. Concretely: 1) Decide the intended behavior: - If only Personal Trainers can create/edit/...
2. **ISS-004** (p. 6): Systematically strengthen pre-conditions and post-conditions for all use cases: 1) For all CRUD operations on shared resources (WorkoutRecord, Workout...
3. **TST-001** (p. 6): Systematically add negative/alternative-flow tests at the controller or integration level for the most important error scenarios.
4. **TST-002** (p. 47): Add a simple traceability table that maps each primary use case (UC-1..UC-26) to one or more concrete test methods. For each integration test (e.g.

8 Traceability Matrix

Of 20 traced use cases: 18 fully covered, 0 without design, 2 without test.

ID	Use Case	Design	Test	Gap
UC-1	User Registration	✓	×	No explicit unit or integration tests cover the User Registration workflow.
UC-2	Sign-In	✓	×	Authentication/Sign-In logic is not explicitly implemented in controllers/DAOs and has no tests.
UC-3	Add Personal Trainer	✓	✓	—
UC-4	Remove Personal Trainer	✓	✓	—
UC-5	View WorkoutRecord	✓	✓	—
UC-6	View Workout Plan	✓	✓	—
UC-7	Register Workout(4Record)	✓	✓	—
UC-8	View Workout(4Record)	✓	✓	—
UC-9	Edit Workout(4Record)	✓	✓	—
UC-10	Delete Workout(4Record)	✓	✓	—
UC-11	Add Trainee	✓	✓	—
UC-12	Remove Trainee (Unfollow)	✓	✓	—
UC-13	Create Workout Plan	✓	✓	—

ID	Use Case	Design	Test	Gap
UC-20	Edit WorkoutPlan (Attach/Detach Workout4Plans)	✓	✓	—
UC-21	Delete WorkoutPlan	✓	✓	—
UC-22	Create Workout(4Plan)	✓	✓	—
UC-23	View Workout(4Plan)	✓	✓	—
UC-24	Edit Workout(4Plan)	✓	✓	—
UC-25	Delete Workout(4Plan)	✓	✓	—
UC-26	View Trainee Workout Record	✓	✓	—

9 Terminological Consistency

Found **10** terminological inconsistencies (5 major, 5 minor).

Group	Variants found	Severity	Suggestion
End user role naming	"User", "Trainee", "regular user/trainee", "client", "clients", "consumer", "end-user"	MAJOR	Use a single primary term for the end user role (e.g., "Trainee") and reserve "User" only when explicitly referring to the generic technical concept (e.g., in database entity "AppUser"). Avoid mixing "client"/"clients" and "consumer" for the same role; if needed, clarify once that a "Trainee" is the trainer's client.
Personal trainer role naming	"Personal Trainer", "PT", "trainer", "trainers", "fitness professionals"	MINOR	Pick one canonical term (e.g., "Personal Trainer") and introduce "PT" once as its acronym, then use either "Personal Trainer" or "PT" consistently. Use generic "trainer" only in high-level narrative if necessary and clarify it refers to "Personal Trainer".
Trainer–trainee relationship naming	"Add Trainee", "Add Trainee (Follow User)", "Remove Trainee (Unfollow)", "followed users", "client roster", "clients", "trainee's records", "Manage Trainees", "followed list"	MAJOR	Standardize on one conceptual verb for the relationship (e.g., "follow") and one noun for the related party (e.g., "Trainee"). For example, consistently use "Add Trainee" / "Remove Trainee" and describe the relation as "followed trainees" instead of mixing "clients", "followed users", and "client roster".

Group	Variants found	Severity	Suggestion
Workout record vs session terminology	"WorkoutRecord", "WorkoutRecords", "Workout4Record", "Workout4Records", "workout record", "workout records", "recorded workout session", "workout session", "session", "Workout Records Screen"	MAJOR	Clearly distinguish and consistently name the aggregate vs single-session concepts. For example, use "WorkoutRecord" only for the aggregate history entity and "Workout4Record" (or a renamed, clearer term like "WorkoutSession") for individual sessions. Avoid using generic "workout record" or "session" interchangeably without specifying which entity is meant.
Workout plan vs daily workout terminology	"Workout Plan", "WorkoutPlan", "WorkoutPlans", "Workout4Plan", "Workout4Plans", "plan", "program", "3-Day Split Program", "weekly schedule", "workout program"	MAJOR	Use a consistent naming scheme that differentiates the overall plan from its daily components. For example, always call the top-level entity "WorkoutPlan" and the daily component "Workout4Plan" (or a clearer name like "DailyWorkout"). Avoid mixing "program" and "plan" unless you define their relationship explicitly.
Exercise intensity / strategy terminology	"strategy", "strategyType", "training strategy", "training approach", "intensity strategies", "exercise intensity", "ExerciseIntensitySetter", "ExerciseStrategyFactory", "Strategy Pattern", "EnduranceExercise", "HypertrophyExercise", "StrengthExercise"	MINOR	Choose one primary term (e.g., "exercise strategy") for the domain concept and align code and text around it. For example, describe the field consistently as an "exercise strategy" rather than alternating between "training approach", "intensity strategies", and "strategyType".
Observer pattern / notification terminology	"Observer Pattern", "Observer pattern", "observer patterns", "real-time notification mechanisms", "real-time notifications in logical terms", "automatic notifications", "notification services"	MINOR	Use a single, consistent term such as "Observer pattern" for the design pattern and a single phrase like "notification mechanism" for the runtime behavior. Avoid mixing "real-time notification mechanisms" with caveats like "in logical terms"; instead, clearly state once that notifications are modeled via the Observer pattern but not implemented as real-time push.

Group	Variants found	Severity	Suggestion
Database manager naming	"DatabaseManager", "DBManager", "DatabaseManager.java", "DatabaseManager" (in tests), "DBManager" (class listing)	MAJOR	Align the class name and all references to a single identifier. If the actual class is named "DBManager", use "DBManager" consistently in code listings, configuration instructions, and tests; if it is "DatabaseManager", update the code and all mentions accordingly.
Application name / branding	"Java-TrainingHub", "TRAINING-HUB", "Training-Hub", "Java-TrainingHub system", "Java-TrainingHub application"	MINOR	Define one official product name (e.g., "Java-TrainingHub") and, if needed, one UI brand label (e.g., "TRAINING-HUB"). Use the official name consistently in all technical sections and clarify once that the UI logo text corresponds to the same system.
System administrator / deployment role naming	"System Administrator", "System Administrator (Implicit)", "initial setup of the application environment", "deployment", "environment configuration", "manual setup of the PostgreSQL database"	MINOR	If you intend to model a distinct "System Administrator" actor, consistently refer to this role with that exact term in all sections that describe environment setup and deployment responsibilities, or else treat these as generic deployment tasks without implying a separate actor.