Integration Testing

1. Purpose and Objectives

The Integration Test Plan makes its purpose very clear: to make sure that the various parts of the Banking Management System—such as the GUI, controllers, business logic, and database access layer—work correctly when combined. Although these components may work perfectly on their own, integration testing checks whether they behave as expected when they communicate with one another.

The objectives highlighted in the plan focus on several important aspects:

- Ensuring that data flows properly through the system, from the user interface all the way to the database.
- Checking that the controllers and business logic coordinate correctly.
- Confirming that operations performed through the GUI (like logging in, creating accounts, or making deposits) are accurately recorded in the database.
- Catching any inconsistencies or interface conflicts before moving onto full system testing.

In simpler terms, the plan aims to ensure the system "plays well together" instead of acting like a collection of disconnected parts.

2. Scope

The test plan clearly explains what will and will not be covered. This helps prevent confusion and keeps the testing efforts focused.

In-Scope

The main areas included in the integration tests involve real user workflows, such as:

- Logging in as an admin
- Registering customers
- Creating customer accounts
- Performing transactions
- Ensuring the model classes interact correctly
- Checking that database operations happen accurately
- End-to-end workflows from **GUI → Controller → Service → DAO → Database**

Out-of-Scope

The plan deliberately excludes areas like:

- GUI aesthetics
- Performance or load testing
- Security features
- Broader non-functional requirements

This keeps the testing practical and aligned with the system's current development stage.

3. Integration Strategy

The plan uses a well-balanced hybrid strategy that combines both **bottom-up** and **top-down** integration testing.

- The **bottom-up approach** starts with the core building blocks of the system—such as the DAO layer and model classes—and verifies these before moving on. This ensures that the "foundation" of the system is reliable.
- The **top-down approach** brings in the GUI and controller layers afterward, allowing testers to validate typical user interactions.
- The **incremental method** means the system is assembled piece by piece, and each step is tested before moving forward. This prevents small issues from growing into bigger ones later on.

This strategy allows the team to gradually build confidence in each part of the system, ensuring smoother integration overall.

4. Test Environment and Items

The plan lists all the tools and technologies needed for testing:

- **Software:** Java 17+, JavaFX, JUnit 5, JDBC, and a compatible SQL database.
- **Hardware:** A typical development machine with at least 8GB RAM and a modern CPU.
- **Test Data:** Sample admin credentials, customer details, and account information.

It also identifies the main test items—GUI screens like LoginView, controllers like AuthController, business logic classes like BankService, and database tables such as accounts and transactions. This helps testers understand exactly what components will be involved.

5. Test Cases

The plan provides clearly defined test cases, each with a specific purpose:

- **Admin login** tests whether the system properly authenticates users.
- **Customer registration** checks if the system can create and manage new customer profiles.
- **Account creation** confirms that the system allows customers to open different types of accounts.
- **Transaction processing** ensures that deposits and withdrawals are handled correctly.
- **Full workflow** tests the entire chain from login all the way to recording transactions in the database.

These test cases reflect real activities that a banking system must perform reliably.

6. Test Execution Procedures

The plan outlines a straightforward and logical process for running the integration tests:

1. Set up and initialize the database.
2. Start the necessary modules.
3. Run the JUnit test suite.

4. Record the results of each test.
5. Investigate and fix any issues that arise, then retest.

This step-by-step approach ensures consistency and reduces the likelihood of mistakes.

7. Entry and Exit Criteria

To avoid rushing into integration testing prematurely, the plan defines clear **entry criteria**, such as:

- All unit tests must already be completed.
- The database must be fully functional.
- The GUI and controllers must compile and run.

Similarly, the **exit criteria** help determine when the integration phase is truly complete:

- All test cases must be executed.
- Critical bugs must be resolved.
- System workflows must operate correctly without failures.

This structured approach helps keep the testing process disciplined and accountable.

8. Risks and Mitigation

The test plan does a good job of identifying potential risks and suggesting practical mitigation strategies:

- Database connectivity problems → Use test containers or an in-memory database
- Difficulty automating the GUI → Simulate inputs programmatically
- Interface mismatches between modules → Conduct an API review
- Bugs from module interaction → Use mocks during early stages

By anticipating these issues, the plan reduces the chances of unexpected delays.

9. Test Deliverables

The expected outputs of the testing phase include:

- The integration test plan
- A set of integration test cases
- JUnit test scripts
- Execution logs
- Bug/defect reports

These deliverables provide transparency and documentation of the testing process.

10. Conclusion

Overall, the Integration Test Plan presents a well-structured and practical approach to verifying the combined functionality of the Banking Management System. It ensures that all major components—from the user interface to the database—work together reliably. By following this plan, the development team can

identify and address integration issues early, leading to a system that is far more stable and dependable by the time it reaches full system testing.

The plan's balanced strategy, clear scope, and defined test cases demonstrate a thoughtful and professional approach to integration testing, helping pave the way for a robust final product.