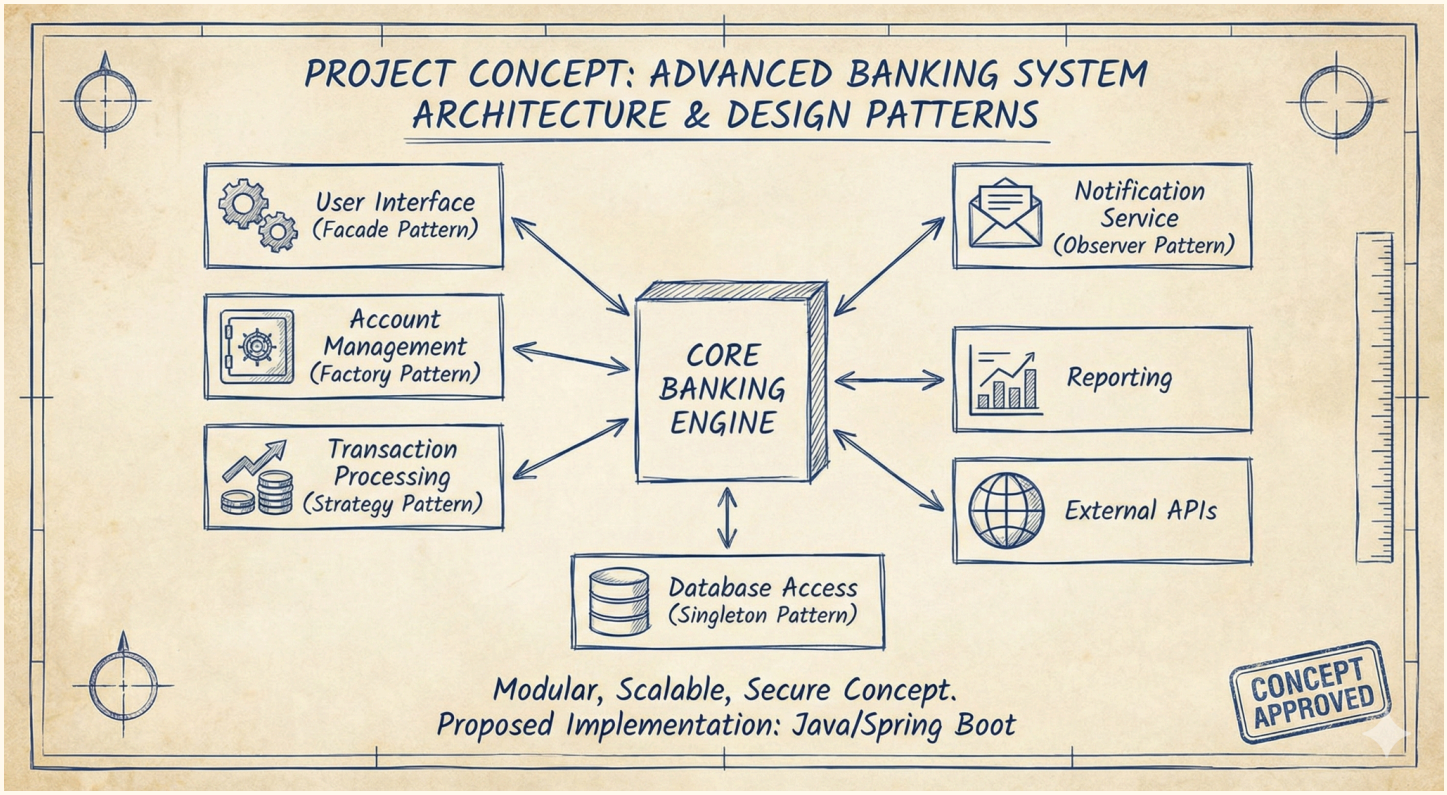
الجمهورية العربية السورية
جامعة دمشق
كلية الهندسة المعلوماتية

Advanced Banking System with Design Patterns

Software Engineering 3

Fifth Year - First Semester

Design and implement a modular, extensible banking system that demonstrates the effective application of both behavioral and structural design patterns. The system should handle core banking operations while maintaining flexibility, scalability, and maintainability through proper design pattern implementation. This project will assess your ability to apply theoretical design pattern knowledge to solve real-world software design challenges in the financial domain.



Functional Requirements ((Apply at least 10))

Account Management Subsystem

- Support multiple account types: savings, checking, loan, and investment accounts
- Implement account creation, modification, and closure operations
- Handle account hierarchy (e.g., parent accounts with child sub-accounts)
- Support account state transitions (active, frozen, suspended, closed)

Transaction Processing Subsystem

- Process deposits, withdrawals, and transfers between accounts
- Implement transaction validation and authorization workflows
- Support scheduled/recurring transactions (e.g., monthly loan payments)
- Provide comprehensive transaction history and audit logging

Customer Service Subsystem

- Implement notification system for account activities (balance changes, large transactions)
- Support customer inquiries and support ticket management
- Provide personalized banking recommendations based on transaction patterns

Administrative Subsystem

- Implement role-based access control (Customer, Teller, Manager, Admin)
- Provide management dashboard with system monitoring capabilities
- Support report generation (daily transactions, account summaries, audit logs)

Non-Functional Requirements ((Apply at least 2))

Extensibility

The system architecture must be designed to accommodate future growth without requiring major rewrites. When the bank wants to introduce new account types (e.g., "Student Accounts," "Business Accounts," "Cryptocurrency Accounts") or new features (e.g., "International Transfers," "Budget Planning Tools," "AI Financial Advisors"), developers should be able to add these with minimal disruption to existing code.

Maintainability

The codebase should be organized logically with distinct responsibilities assigned to different components. Each class, method, and module should have a single, well-defined purpose. This makes the system easier to understand, debug, and modify over time.

Performance

The system must process 100+ transactions simultaneously without significant degradation in response time or failure. This includes deposits, withdrawals, transfers, balance inquiries, and other banking operations happening concurrently from multiple users.

Security

The system must protect sensitive financial data and operations from unauthorized access, manipulation, and attacks. This includes both application-level security (authentication, authorization) and infrastructure-level protection (against common web attacks).

Testability

Every critical component of the system must be designed to be tested in isolation using unit tests, with external dependencies (databases, APIs, file systems) replaced by mock objects. Additionally, integration tests should verify that components work together correctly.

Structural Design Patterns (Minimum 3 required)

Composite Pattern

Application: Implement hierarchical account structure where individual accounts and account groups (e.g., family accounts, business accounts) can be treated uniformly.

Expected Outcome: Single interface for managing both leaf accounts and composite account groups. Any risk management approach is accepted as an outcome.

Adapter Pattern

Application: Integrate with external payment gateways or legacy banking systems.

Expected Outcome: Unified interface for different payment processing systems (e.g., credit card processors, international wire transfers).

Decorator Pattern

Application: Add optional features to accounts (e.g., overdraft protection, premium services, insurance).

Expected Outcome: Dynamic addition/removal of account features without modifying core account classes.

Facade Pattern (Optional but recommended)

Application: Provide simplified interface to complex transaction processing subsystem.

Expected Outcome: Simple API for clients while hiding complex internal operations.

Behavioral Design Patterns (Minimum 3 required)

Observer Pattern

Application: Implement real-time notification system for account activities

Expected Outcome: Multiple notification channels (email, SMS, in-app) updated automatically when account events occur.

Strategy Pattern

Application: Implement different interest calculation algorithms for various account types

Expected Outcome: Runtime selection of interest calculation strategies based on account type and market conditions.

Chain of Responsibility Pattern

Application: Implement transaction approval workflow (e.g., small transactions auto-approved, large transactions require manager approval).

Expected Outcome: Flexible approval chain that can be modified at runtime based on transaction amount and type.

State Pattern (Optional but recommended)

Application: Manage account state transitions and state-specific behaviors.

Expected Outcome: Clean handling of account states (active, frozen, suspended, closed) with appropriate behaviors for each state.

Deliverables (submitted through the form)

https://forms.gle/QYSBtzhQjAcr4Mab7

UML Diagrams:

- Class diagram showing pattern implementations.
- Sequence diagrams for key operations demonstrating pattern usage.

Design Rationale Document (max 1000 words):

- Justification for pattern selection.
- Trade-offs considered and alternative patterns evaluated.
- How patterns address specific requirements.

Source Code Implementation:

- Correct implementation of minimum 6 design patterns (**3** structural + **3** behavioral).
- Clean separation of concerns with appropriate package structure.
- Proper abstraction and interface definitions.

Quality and Testing:

- Comprehensive unit tests with minimum 70% code coverage.
- Clean, maintainable code with proper naming conventions and comments.

- Error handling and validation for edge cases.

Final Report:

- Clear demonstration of design pattern implementations in action.
- Challenges faced and solutions implemented.
- Future improvements and scalability consideration.

Implementation Guidelines

- Use interfaces/abstract classes for pattern abstractions.
- Follow SOLID principles where applicable.
- Implement proper error handling and logging.
- Document all public methods and key design decisions.
- Unit tests must use mocking frameworks where appropriate (Mockito, Moq, unittest.mock).

This project is designed to be challenging but achievable within the timeframe. Focus on clean implementation of the required patterns rather than feature completeness. Quality of design and implementation is more important than quantity of features. Start early, seek help when needed, and iterate on your design based on feedback.

Minimum number of students: **3**

In case the group consists of **4** students, two additional functional requirements **OR** one additional non-functional requirement, are required.

In case the group consists of **5** students, ALL functional requirements + ALL non-functional requirements are expected.

Total Marks: 15 (+2 for bonus).

PARTIAL INTERVIEW: DEC 11 2025

Raghdan Ghabra