

Introduction to Design Patterns (Banking Context)

Scenario:

You are building a core banking system for retail banking. Initially, one large monolithic class handles fund transfer, loan processing, notifications, and interest calculation logic.

Problem:

Adding new business rules (e.g., special interest rates, tax compliance changes) breaks other parts. Over time, the code becomes fragile and hard to test.

Solution:

Apply design patterns to split responsibilities:

- Use **Strategy** for dynamic interest or tax calculations.
- Use **Observer** for sending alerts (SMS/Email) on balance updates.
- Use **Factory** to decide which product (Savings, FD, RD, Loan) is instantiated.

Key Takeaway:

Design patterns help make your banking applications modular, extensible, and maintainable, improving your team's delivery and reducing risk.



1. Creational Patterns – Banking Examples

Singleton Pattern

Use Case: Centralized Logging or Audit Trail

Scenario:

You need to log every customer transaction (debit/credit/failure) across modules.

Solution:

Use a Singleton AuditLogger class that provides a global point for logging. Ensures consistent format and avoids multiple file handles or DB connections.

Factory Method Pattern

Use Case: Dynamic Account Type Creation

Scenario:

When a customer opens a new account via the mobile app, they can choose from Savings, Current, NRI, or Senior Citizen accounts.

Solution:

Use Factory Method to return the correct subclass of Account based on input. This avoids ifelse or switch clutter and supports easy addition of new account types.

Abstract Factory Pattern

Use Case: Regional Product Variants

Scenario:

A bank operates in India, UAE, and the UK. Each region has different compliance rules and tax structures for Loans, Credit Cards, and Savings accounts.

Solution:

Abstract Factory creates regional families of related financial products, e.g., IndiaLoanFactory, UKLoanFactory, etc., ensuring consistent rules per geography.

Builder Pattern

Use Case: Customer Onboarding

Scenario:

Account creation requires collecting optional documents: PAN, Aadhaar, Driving License, Passport, Address Proof, Nominee.

Solution:

Use Builder Pattern to construct the CustomerProfile step-by-step, depending on document availability and KYC flow.

Prototype Pattern

Use Case: Duplicating Loan Templates

Scenario:

Bank managers frequently create pre-approved loan schemes with standard terms.

Solution:

Store a prototype of each loan scheme and clone it for quick customization (e.g., change rate or tenure) without modifying the original object.

2. Structural Patterns – Banking Examples

Adapter Pattern

Use Case: Legacy System Integration

Scenario:

You need to integrate with a legacy COBOL-based core banking system that uses a different data structure for transactions.

Solution:

Use an Adapter to wrap the old system and convert its data structure into the modern DTO used by your Java-based services.

Bridge Pattern

Use Case: Notification System

Scenario:

Different modules (Loans, Cards, Accounts) need to send alerts via SMS, Email, or WhatsApp.

Solution:

Decouple message format (AlertType) from the delivery channel (SMS/Email). Bridge lets you add new channels without touching alert logic.

Composite Pattern

Use Case: Account Portfolio Summary

Scenario:

Customers may have multiple Savings, Fixed Deposits, Mutual Funds, etc.

Solution:

Use Composite pattern to allow unified view — i.e., treat both single products and group of products (like family joint account) uniformly.

Decorator Pattern

Use Case: Transaction Enrichment

Scenario:

You want to enrich a fund transfer with additional features: add GST, add transaction description, or mark it recurring.

Solution:

Use Decorators to layer additional responsibilities dynamically without changing the base Transaction class.

Facade Pattern

Use Case: Loan Disbursal Process

Scenario:

Loan disbursal involves eligibility check, credit score fetch, document verification, and fund release.

Solution:

Use a Facade LoanProcessor to simplify this multi-step workflow for the front-end or API layer.

Proxy Pattern

Use Case: Account Image Loading

Scenario:

KYC images are stored in a secure vault.

Solution:

Use Proxy to lazily load images only when explicitly requested. This improves performance and reduces bandwidth.

• Flyweight Pattern

Use Case: Bank Form Field Templates

Scenario:

Millions of customers fill forms using the same field definitions (Name, DOB, Address, etc.)

Solution:

Use Flyweight to share field metadata and formatting rules instead of duplicating across forms, saving memory.

🔁 3. Behavioral Patterns – Banking Examples

Strategy Pattern

Use Case: Interest Rate Calculation

Scenario:

Savings, FD, and Loans all have different interest calculation strategies (simple, compound, tiered).

Solution:

Encapsulate each strategy and allow dynamic switching. This avoids multiple conditionals and makes rules extensible.

Observer Pattern

Use Case: Balance Notification

Scenario:

When balance drops below a threshold or a large debit occurs, alerts must go to the customer and branch manager.

Solution:

Use Observer — whenever balance changes, all registered listeners are notified (EmailService, FraudService, NotificationService).

Command Pattern

Use Case: Transaction Management

Scenario:

Implement undo/redo for internal banking workflows, e.g., fund reversal, voucher reprocessing.

Solution:

Encapsulate operations (debit, credit, freeze) as Command objects that can be logged, queued, or replayed.

Mediator Pattern

Use Case: Branch Coordination

Scenario:

Account closure involves branch manager, compliance team, and backend DB cleanup.

Solution:

Use Mediator (e.g., ClosureCoordinator) to manage communication without coupling each module directly to others.

Chain of Responsibility Pattern

Use Case: Loan Approval Workflow

Scenario:

A loan application passes from agent \rightarrow branch manager \rightarrow regional head \rightarrow compliance.

Solution:

Each handler processes or forwards the request. Chain is dynamic and promotes clean separation of responsibility.

Memento Pattern

Use Case: Account Snapshot

Scenario:

Support "point-in-time recovery" of account balance/history.

Solution:

Use Memento to store internal state without exposing implementation. Helpful in audits or rollbacks.

Template Method Pattern

Use Case: KYC Validation

Scenario:

Each product (loan, card, account) follows a KYC flow but with slightly different steps.

Solution:

Define skeleton in abstract class and allow subclasses to override specific steps (like PAN verification or credit check).

State Pattern

Use Case: Fund Transfer Lifecycle

Scenario:

A transaction can be in states like INITIATED, PROCESSING, SUCCESS, FAILED, CANCELLED.

Solution:

Encapsulate states and allow transition logic within state classes. Cleaner than large switch statements.

Would you like to proceed to **Anti-Patterns**, **Modern Architecture Patterns**, and **Refactoring Patterns in Banking** now — all in the same converted format?

You said:

yes sure

ChatGPT said:

Perfect. Let's continue with **Anti-Patterns**, **Modern Architecture Patterns**, and **Refactoring Patterns**, **all tailored to the banking domain**. These are designed for advanced engineers and architects to provoke discussion and transformation.

X 1. God Object

Scenario:

A BankService class does everything — transaction posting, interest calculation, fraud checks, customer updates.

Problem:

Class becomes untestable, brittle, and violates **Single Responsibility Principle**.

Solution (Refactor):

Split into specialized services: TransactionService, InterestService, KYCService. Use **Facade**, **Strategy**, or **Mediator** to coordinate.

X 2. Spaghetti Code

Scenario:

Fund transfer code has deep nesting of if-else with business, validation, logging, and DB logic all tangled.

Problem:

Hard to debug, change, or onboard new developers.

Solution (Refactor):

Apply **Template Method** or **Chain of Responsibility** for validations. Use layers (Controller \rightarrow Service \rightarrow Repo) and business rule engines if needed.

X 3. Hardcoded Config

Scenario:

SMTP server, DB credentials, and limits (e.g., ₹2 lakh max/day) are hardcoded in code.

Problem:

Difficult to change per environment or during audit.

Solution (Refactor):

Externalize config via Spring Boot profiles, Spring Cloud Config, or Kubernetes ConfigMaps.

X 4. Lava Flow Code

Scenario:

Old loan eligibility code written in 2013 still exists and is copied for new products.

Problem:

Dead, unmaintained code causes bugs in new modules.

Solution (Refactor):

Identify such flows, write unit tests for behavior, refactor into **Strategy pattern** or migrate to a rules engine.

X 5. Shotgun Surgery

Scenario:

Adding a new type of fee means touching 15 classes: Account, Transaction, Report, Notification, etc.

Problem:

Any small change ripples across many modules.

Solution (Refactor):

Apply **Open-Closed Principle**. Use Decorators, Observer, or Event-Driven (Kafka) design to decouple fee processing.

X 6. Copy-Paste Programming

Scenario:

Loan, Fixed Deposit, and Recurring Deposit share 80% code but are copied into different services.

Problem:

Bug in one means bug in all, but you fix it in only one.

Solution (Refactor):

Apply **Template Method** or **Composite** pattern. Promote reuse through common abstract classes or composition.



🚀 Modern Architecture Patterns in Banking

These are high-level patterns that help build **resilient**, **scalable**, **and cloud-native banking apps**.



🗩 1. Event-Driven Architecture (EDA)

Use Case: Account Opened → Notify → Trigger KYC → Send Welcome Kit

Why:

Avoid tight coupling. Instead of synchronous REST calls, publish events (ACCOUNT CREATED) on Kafka.

Patterns Inside:

- **Observer** → consumers listen to topic.
- **Command/Event Separation** → write via REST, read via topic.



🔆 2. CQRS (Command Query Responsibility Segregation)

Use Case: Loan processing requires transactional DB; analytics requires fast read DB

Why:

Write models (commands) are separate from read models (queries). Allows optimized read performance and scalable writes.

Patterns Combined:

- Command Pattern (write flow)
- **Strategy** for projections

🧩 3. SAGA Pattern (Choreography or Orchestration)

Use Case: Fund Transfer → Debit → Notify → Credit → Generate Receipt

Why:

Avoid distributed transactions (XA). Break flow into smaller local transactions, coordinate via Kafka.

Patterns Used:

- **State** for transaction status
- Command/Compensating Command
- **Observer** for event chaining



🗩 4. Strangler Fig Pattern

Use Case: Migrate Legacy Oracle Forms-based loan module to Spring Boot microservice

Why:

Wrap old system with an API Gateway, gradually replace pieces without breaking entire system.

🗩 5. Bulkhead Pattern

Use Case: Failure in credit score API shouldn't bring down account dashboard

Why:

Isolate features into separate threads/pools so one failure doesn't cascade.

Tool:

Resilience4j, Hystrix (legacy)



🗩 6. API Gateway Pattern

Use Case: Securely expose loans, cards, accounts under a unified /api/v1 URL

Why:

Acts as a reverse proxy. Can apply auth, throttling, and routing centrally.

Spring Tool:

Spring Cloud Gateway or Netflix Zuul



* 7. Circuit Breaker Pattern

Use Case: Third-party CIBIL check is slow or unavailable

Why:

Automatically fallback or degrade service gracefully.

Tool:

Resilience4j, Sentinel



Refactoring Patterns in Banking Projects

These are powerful techniques to **clean and modularize legacy code**. You can use these as *labs in* your workshop.



🔁 1. Replace Conditional with Polymorphism

Scenario:

if(accountType.equals("SAVINGS")) ... else if("CURRENT") ... else if("NRI")

Refactor To:

Create subclasses for each account. Use **Strategy** or Factory Method for dynamic resolution.

2. Replace Magic Numbers/Strings

Scenario:

Hardcoded "INR", "USD", 200000 as max limit.

Refactor To:

Use enums or constants. Move configs to .properties or DB. Support localization.

3. Extract Method

Scenario:

A large method processLoan() does 10 things.

Refactor To:

Split into smaller methods: validateDocuments(), checkCreditScore(),
createDisbursalEntry()

4. Introduce Null Object

Scenario:

if(customer.getNominee() != null) ... else ...

Refactor To:

Use Null Object pattern. Return EmptyNominee instead of null.

5. Move Method / Class

Scenario:

Method calculateInterest() sits in Customer class but uses mostly LoanAccount fields.

Refactor To:

Move method to LoanAccount. Apply SRP.

6. Replace Temp with Query

Scenario:

Temporary variable monthlyFee = calculateFee(); used in multiple lines.

Refactor To:

Inline the method or use lazy evaluation.

7. Encapsulate Collection

Scenario:

Class exposes List<Transaction> getTransactions() and client adds directly to list.

Refactor To:

Encapsulate with methods addTransaction(), getImmutableTransactions()