Patterns of Enterprise Application Architecture

The Complete 80-Page Mastery Guide

Detailed Explanations • Code Examples • Visual Diagrams • Feynman-Ready Content

Based on the seminal work by Martin Fowler

What This Guide Delivers:

This comprehensive 80+ page guide provides complete coverage of all concepts from the book with:

• **Part 1 - The Narratives (8 Chapters):** Complete architectural foundations with detailed explanations, diagrams, and extensive code examples for each concept

• **Part 2 - The Patterns (10 Chapters, 40+ Patterns):** Every pattern explained with intent, problem, solution, implementation details, code samples, and guidance

• **Visual Learning:** Layer diagrams, architecture diagrams, comparison tables, decision matrices

• **Practical Examples:** Real code in Java and C# demonstrating each pattern in action

• **Feynman Technique Support:** Self-check questions, teaching prompts, summary boxes, and "explain simply" sections throughout

• **Decision Guides:** Clear guidance on when to use which pattern, trade-off analysis, and architecture selection criteria

Table of Contents

Introduction: What Are Enterprise Applications?

Defining Enterprise Applications

Before we can understand patterns for enterprise applications, we must first understand what makes enterprise applications unique. Enterprise applications are software systems that support the core operations of businesses and organizations. They differ fundamentally from other types of software such as games, embedded systems, or desktop productivity tools.

Examples of enterprise applications include: payroll systems, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading. These systems are the backbone of modern organizations.

The Hospital System Analogy: Imagine a comprehensive hospital information system. It must store patient records for decades (some regulations require 30+ years of retention). Hundreds of doctors, nurses, administrators, and staff access it simultaneously throughout the day, each seeing different views of the data based on their role. The system enforces complex medical protocols—drug interaction checks, insurance pre-authorization, regulatory compliance, and clinical pathways. It connects to external systems: laboratories for test results, pharmacies for prescriptions, insurance companies for billing, and government agencies for reporting. And it must work reliably 24/7 because lives literally depend on it. This is the essence of enterprise applications.

The Six Defining Characteristics

Enterprise applications share six key characteristics that distinguish them from other software types. Each characteristic creates specific architectural challenges that the patterns in this book address.

Characteristic	What It Means	Architectural Challenge
Persistent Data	Data survives system restarts and must be stored for years or decades	Reliable storage, backup strategies, data migration, schema evolution over time
Large Data Volume	Systems handle gigabytes to petabytes of information	Performance optimization, indexing, query optimization, partitioning strategies
Concurrent Access	Many users access and modify data simultaneously	Locking strategies, transaction management, conflict resolution, scalability
Complex Business Logic	Intricate rules that vary by context and evolve frequently	Maintainable organization, testability, separation of concerns
Integration	Must connect to many external systems and services	API design, error handling, data transformation, protocol management
Multiple User Interfaces	Same functionality accessed through different interfaces	Reusable business logic, consistent behavior across all channels

Types of Enterprise Applications

While all enterprise applications share the characteristics above, they vary in emphasis. Understanding which characteristics dominate your specific application helps you choose the right patterns.

Type	Primary Challenge	Key Patterns to Consider
Data-Intensive	Managing large volumes and complex queries	Repository, Query Object, Data Mapper, identity management
Transaction-Heavy	High throughput with concurrent updates	Unit of Work, Optimistic/Pessimistic Lock, Transaction Script
Logic-Heavy	Complex, frequently changing business rules	Domain Model, Service Layer, Strategy, Specification

Type	Primary Challenge	Key Patterns to Consider
Integration-Heavy	Connecting and coordinating many external systems	Gateway, Adapter, Remote Facade, Messaging patterns
User-Facing	Multiple interfaces with rich user experience	MVC, Template View, Application Controller, Front Controller

What Are Patterns and Why Do They Matter?

A pattern is a reusable solution to a commonly occurring problem in a specific context. Patterns are NOT finished code that you copy and paste—they are templates describing the structure of a solution that you must adapt to your specific situation. The concept of patterns in software was adapted from architect Christopher Alexander's work on building architecture.

The Structure of Each Pattern

Every pattern in this book follows a consistent structure:

Name: A memorable identifier that becomes part of your professional vocabulary. When you say "let's use a Repository here," everyone who knows patterns immediately understands the design intent.

Intent: A one or two sentence description of what the pattern does and why you would use it.

Problem: The specific challenge or situation the pattern addresses. Understanding the problem is crucial—a pattern applied to the wrong problem creates complexity without any benefit.

Solution: How the pattern solves the problem. This includes the structure (participating classes and their relationships) and behavior (how the objects interact at runtime).

Consequences: The trade-offs and implications of using the pattern. Every pattern has both benefits and costs, and understanding these helps you make informed decisions.

When to Use: Guidance on contexts where the pattern is appropriate—and equally importantly, when it is NOT appropriate.

Example: Working code (typically in Java, C#, or both) showing the pattern implemented in a realistic scenario.

Why Patterns Create Value

- **Shared Vocabulary:** Patterns give teams a common language. Saying 'use Active Record' communicates more design intent than paragraphs of description.
- **Proven Solutions:** Patterns represent distilled experience from many projects. They are solutions that have worked repeatedly in real-world situations.
- **Design Quality:** Patterns embody good design principles. Using them helps avoid common mistakes and anti-patterns.
- **Learning Accelerator:** Patterns help you learn from others' experience without having to make all the mistakes yourself.
- **Documentation:** Pattern names in code and documentation make design decisions explicit and understandable to future maintainers.

Book Structure: The Duplex Book

This book is what Fowler calls a 'duplex book'—essentially two books in one volume. Part 1 (Chapters 1-8, approximately 100 pages in the original) is a narrative tutorial that tells the story of enterprise application architecture. You should read it sequentially to understand the big picture and how concepts relate. Part 2 (Chapters 9-18, approximately 400 pages) is a reference catalog of over 40 patterns. You can dip into it as needed, reading patterns relevant to your current challenges.

Feynman Check - Can You Explain?

1. What makes enterprise applications different from other software?
2. What are the six defining characteristics of enterprise applications?
3. What is a pattern and what components does it have?
4. Why do patterns matter for software development?
5. How is this book structured and how should you read it?

Chapter 1: Layering — The Foundation of Enterprise Architecture

Why Layering Is the Most Important Concept

If you learn only one concept from this entire book, make it layering. Layering is the most fundamental technique for managing complexity in software systems. Almost every successful enterprise application uses some form of layered architecture, and understanding layering deeply is a prerequisite for understanding everything else in this book.

Layering has been a fundamental principle since the early days of computing. The TCP/IP network stack is layered (physical, data link, network, transport, application). Operating systems are layered (hardware, kernel, system services, applications). File systems, compilers, and databases all use layering. The concept is universal because it works—it manages complexity by dividing a system into parts with clear responsibilities and carefully controlled dependencies.

The Restaurant Analogy - Understanding Layers:

Think of a fine restaurant. It has distinct areas with clear responsibilities and controlled communication between them:

Dining Room (Presentation Layer): This is where customers interact with the restaurant. They see beautifully designed menus, place orders with waiters, and receive food presented elegantly. Customers never see the chaos of the kitchen or the inventory in the pantry. Waiters translate customer requests ('I'd like the fish, but not too spicy') into precise kitchen terminology ('One salmon, mild preparation').

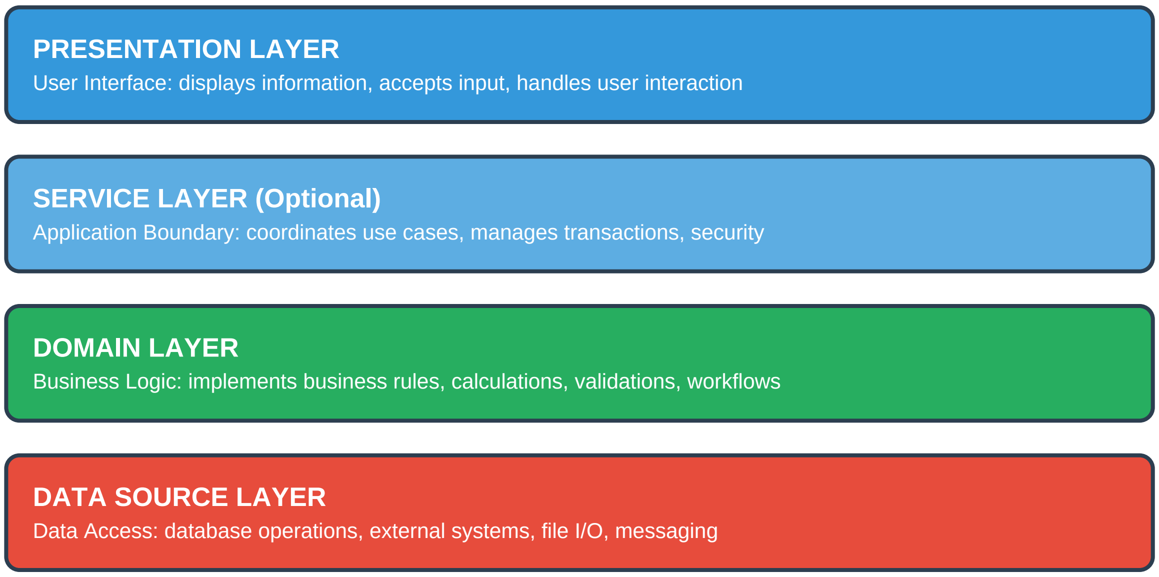
Kitchen (Domain Layer): This is where the real work happens. Chefs know recipes, cooking techniques, food safety rules, and timing. They transform raw ingredients into finished dishes according to established procedures. The kitchen doesn't care whether the customer is dining in, getting takeout, or ordering for a catering event—it just prepares food correctly according to its expertise.

Pantry and Suppliers (Data Source Layer): This provides the raw materials. The pantry stores ingredients; suppliers deliver fresh produce. They know nothing about recipes or customers—they simply provide and store raw materials when requested.

Key insight: Customers never walk into the kitchen. Chefs never serve tables. The pantry manager doesn't know what dishes are being prepared. Each area has clear boundaries and communicates only through defined interfaces (order tickets, plated dishes, inventory requests).

The Core Principle: Separation of Concerns

Layering separates your application into horizontal slices, where each slice (layer) has a specific, focused responsibility. The fundamental rule is that dependencies point in ONE direction only—from upper layers to lower layers. Upper layers can call and depend on lower layers, but lower layers must NEVER call or depend on upper layers.


PRESENTATION LAYER
User Interface: displays information, accepts input, handles user interaction
SERVICE LAYER (Optional)
Application Boundary: coordinates use cases, manages transactions, security
DOMAIN LAYER
Business Logic: implements business rules, calculations, validations, workflows
DATA SOURCE LAYER
Data Access: database operations, external systems, file I/O, messaging

The Three Principal Layers in Detail

1. Presentation Layer (User Interface Layer)

The presentation layer handles all interaction with users and external systems that consume your application. It is the outermost layer—the face of your application to the outside world.

Presentation Layer Responsibilities:

Display Data: Format and present information to users in appropriate ways. This includes rendering HTML pages for web browsers, generating JSON for REST APIs, formatting PDF reports, or displaying data in mobile app screens.

Accept Input: Receive user input from various sources—web forms, button clicks, API calls, file uploads, voice commands, or any other input mechanism your application supports.

Validate Input Format: Ensure input is well-formed before passing it deeper. This means checking that required fields are present, data types are correct, and formats are valid. Important: This is FORMAT validation (is this a syntactically valid email address?), NOT business validation (is this email already registered in our system?).

Translate Formats: Convert between external representations (JSON, XML, form data, query parameters) and internal objects that the domain layer can work with.

Manage UI State: Track things like current page, selected items, expanded/collapsed panels, wizard step, and scroll position—things that matter for the user interface but are irrelevant to business logic.

Handle Navigation: Determine what screen to show next based on user actions and application state.

CRITICAL RULE: The presentation layer must NOT contain business logic. This is the most commonly violated layering principle, and violating it causes serious problems down the road. If you find yourself calculating discounts in JavaScript, validating business rules in a controller, or making business decisions in a view template, you are putting business logic in the wrong place.

Signs That Business Logic Has Leaked Into Presentation

- You're calculating totals, discounts, taxes, or other business values in JavaScript or controller code
- You're checking business conditions to decide what to display (e.g., 'if customer is premium AND order total > $100 AND not on fraud blacklist...')
- The same calculation or validation exists in multiple places (both web and mobile apps calculate shipping costs)
- You cannot test business rules without starting a web server or rendering actual pages
- Changing a business rule requires modifications in multiple UI components across your application
- Business people cannot understand where their rules are implemented in the code

2. Domain Layer (Business Logic Layer)

The domain layer is the heart of your application—it contains everything that makes your application unique and valuable. This is where business rules live: how to calculate prices, when an order is valid, how to process a refund, what happens when inventory runs low. The domain layer should be technology-agnostic; it should work correctly regardless of how data is stored or how users interact with the system.

Domain Layer Responsibilities:

Implement Business Rules: "If order total exceeds $100 AND customer is Gold tier, apply 15% discount. If Silver tier, apply 10% discount. If order contains hazardous materials, require signature on delivery and add $15 handling fee."

Enforce Business Constraints: "An order must have at least one item. Orders cannot ship internationally without proper customs documentation. A customer's credit limit cannot be exceeded without manager approval."

Execute Business Workflows: "When order is placed: validate all items → check inventory availability → reserve items → calculate totals with applicable discounts → authorize payment → confirm order → trigger fulfillment process → send confirmation email."

Maintain Business Invariants: "Account balance must never become negative. Inventory count cannot exceed warehouse capacity. An employee cannot report to themselves in the org chart."

Calculate Business Values: "Total price = sum of line items - applicable discounts + tax (based on shipping destination and product categories) + shipping (based on weight, dimensions, destination, and speed)."

Make Business Decisions: "Should this loan be approved given the applicant's credit score, income, and debt ratio? Which shipping carrier offers the best combination of cost and speed for this package? Is this transaction pattern potentially fraudulent?"

Raise Domain Events: "When order ships, notify customer. When inventory falls below reorder threshold, alert purchasing. When payment fails, trigger retry process or notify customer."

Real-World Domain Logic Example: E-Commerce Order Processing

Let's trace through the domain logic involved when a customer places an order. This illustrates the complexity that belongs in the domain layer—NOT in controllers or stored procedures.

When a customer submits an order, the domain layer handles these concerns:

1. Order Validation
• Are all requested items currently in stock and available for sale?
• Is the shipping address in a region we can deliver to?
• Is the customer's account in good standing (not suspended, not flagged)?
• Are there any restricted items that require special handling (age-restricted products, export-controlled items, hazardous materials)?

2. Price Calculation
• Calculate base price for each item (may vary by customer tier, quantity ordered, or active promotions)
• Apply item-level discounts (buy-2-get-1-free offers, percentage discounts, bundle pricing)
• Calculate order subtotal
• Apply order-level discounts (coupon codes, loyalty rewards, seasonal promotions)
• Calculate tax based on shipping destination, product categories, and customer tax-exempt status
• Calculate shipping cost based on package weight, dimensions, destination, delivery speed, and carrier rates
• Calculate final order total

3. Inventory Management
• Check real-time inventory levels across all fulfillment centers
• Reserve requested items to prevent overselling to concurrent customers
• Handle partial availability situations (offer backorder? suggest alternatives? split shipment?)
• Update inventory projections for reporting and reordering

4. Payment Processing
• Validate payment method is acceptable and not expired
• Check against customer's credit limit if using store credit
• Authorize payment amount with payment processor
• Handle authorization failures (retry with different processor? notify customer? hold order?)
• Record authorization code for later capture

5. Order Confirmation
• Generate unique order number
• Create complete order record with all details, calculations, and references
• Update customer's order history and loyalty point balance
• Trigger confirmation email/SMS to customer
• Initiate fulfillment process with warehouse
• Update analytics and reporting systems

All of this logic belongs in the domain layer—NOT in web controllers, NOT in stored procedures, NOT in UI code.

3. Data Source Layer (Persistence Layer)

The data source layer handles communication with external data sources—primarily databases, but also file systems, message queues, caches, external APIs, and legacy systems. It knows HOW to store and retrieve data but has no knowledge of what that data means or how it should be used. It translates between the domain's object representation and the storage mechanism's format.

Data Source Layer Responsibilities:

Execute Database Operations: SQL queries (SELECT, INSERT, UPDATE, DELETE), stored procedure calls, transaction management, connection pooling, and connection lifecycle management.

Map Between Formats: Translate database rows to domain objects and vice versa. Handle type conversions, null handling, date/time formatting, and encoding differences between objects and database.

Manage Connections: Connection pooling, connection lifecycle, handling connection failures, implementing reconnection strategies, and managing connection timeouts.

External System Communication: Call external REST APIs, SOAP services, send/receive messages from queues, read/write files, and interact with legacy systems.

Caching: Implement caching strategies to reduce database load and improve performance. Manage cache invalidation when data changes.

Handle Data-Level Concerns: Pagination, sorting, filtering at the database level (for efficiency), bulk operations, and batch processing.

Important Distinction: The data source layer knows HOW to retrieve 'all orders for customer 123' from the database, but it doesn't know WHY you want them or what business decisions you'll make with them. Business decisions about those orders belong in the domain layer.

Benefits of Layering: Complete Analysis

Benefit	How It Works	Practical Impact
Understandability	Each layer has focused responsibility. Developers can learn one layer at a time.	New team members become productive faster. Code reviews are easier.
Substitutability	Interface between layers allows swapping implementations without affecting other layers.	Can switch databases, UI frameworks, or external services with limited code changes.
Testability	Mock or stub the layer below to test each layer independently.	Fast unit tests. Can test business logic without database or web server.
Reusability	Lower layers serve multiple upper layers.	Same domain logic works for web, mobile, API, and batch processing.
Maintainability	Changes are localized to the relevant layer.	Business rule changes only affect domain layer. Lower risk of introducing bugs elsewhere.
Team Scalability	Teams can work on different layers in parallel.	Front-end and back-end teams work independently without blocking each other.
Technology Flexibility	Each layer can use the most appropriate technology.	UI in TypeScript, domain in Java, stored procedures for performance-critical queries.

Layer Communication: The Golden Rules

Understanding how layers communicate is as important as understanding what each layer does. These rules, when followed strictly, enable all the benefits of layering. Violating them—even in small ways—begins to erode those benefits.

The Four Golden Rules of Layer Communication:

Rule 1: Downward Dependencies Only
Upper layers depend on lower layers. Presentation depends on Domain. Domain depends on Data Source. Dependencies NEVER point upward. The Data Source layer must not import or reference anything from the Domain layer. The Domain layer must not import or reference anything from the Presentation layer. This one-way dependency is what enables substitutability and testability.

Rule 2: No Layer Skipping
Don't bypass layers. Presentation should not directly access the database—even for 'simple' queries. Even if it seems like 'just reading one value,' go through the Domain layer. Skipping layers scatters logic and creates hidden dependencies. Today's 'simple query' becomes tomorrow's business rule that exists in multiple places and must be kept in sync.

Rule 3: Interface at Boundaries
Layers should communicate through well-defined interfaces (abstract types, not concrete implementations). The Presentation layer doesn't depend on OrderServiceImpl—it depends on OrderService interface. This allows substituting implementations (real vs. mock, local vs. remote, current vs. new version) without changing the calling code.

Rule 4: No Circular Dependencies
If A depends on B, then B cannot depend on A. Ever. This applies within layers too. Circular dependencies create tangled code where you cannot understand, test, or change one part without understanding, testing, and changing the other. If you find yourself needing a circular dependency, refactor—usually by extracting a new interface or introducing an event-based design.

Request Flow: A Complete Traced Example

Let's trace a complete request through all layers to see how they work together while maintaining proper separation. This is a GET request to view order details.

java

```java
// STEP 1: HTTP Request Arrives (Presentation Layer - Controller)
// URL: GET /orders/1234
// Web framework routes to OrderController

@GetMapping("/orders/{id}")
public ResponseEntity<OrderDTO> viewOrder(@PathVariable Long id) {
    // Controller extracts orderId from URL path
    // Controller may check: is user authenticated?

    // STEP 2: Controller Calls Domain Layer
    // Note: passes primitive value, not HTTP objects
    OrderDTO order = orderService.getOrderDetails(id, getCurrentUserId());

    // STEP 6: Controller Prepares Response
    return ResponseEntity.ok(order);
}

// STEP 3: Service Layer Processes Request (Domain Layer)
@Service
public class OrderService {
    public OrderDTO getOrderDetails(Long orderId, Long userId) {
        // Business authorization: can THIS user view THIS order?
        Order order = orderRepository.findById(orderId);
        if (!order.isViewableBy(userId)) {
            throw new AccessDeniedException("Cannot view this order");
        }

        // May calculate derived values
        order.calculateDaysSinceOrder();

        // STEP 5: Return result to controller
        return OrderDTO.from(order);
    }
}

// STEP 4: Repository Calls Data Source Layer
@Repository
public class JpaOrderRepository implements OrderRepository {
    public Order findById(Long id) {
        // Check Identity Map: already loaded this session?
        // If not, execute SQL: SELECT * FROM orders WHERE id = ?
        // Map ResultSet to Order domain object
        // Store in Identity Map for future requests
        return order;
    }
    // ... (continued)
```

Key Observations from the Request Flow

- Each layer only knows about the layer immediately below it—Controller knows Service, Service knows Repository
- Data flows DOWN as requests and UP as responses through the same path
- Domain objects can cross layer boundaries (acceptable), but HTTP/UI concepts never reach the data layer
- Each layer could be replaced without affecting the others if interfaces are maintained
- Business logic (authorization check, date calculation) stays in the domain layer
- The Controller is thin—it coordinates but doesn't contain logic

Common Layering Mistakes: Detailed Analysis

Even experienced developers make layering mistakes. Understanding these common errors helps you avoid them and recognize them in existing codebases. Each mistake seems innocent initially but compounds over time into serious architectural problems.

Mistake	What Happens	Why It's Bad
Business logic in Presentation	Calculating totals, discounts, or validations in JavaScript or controllers	Must duplicate for every UI. Cannot test without browser. Logic diverges over time.
Business logic in Data Source	Complex business rules implemented in stored procedures	Hard to unit test. Tied to specific database. Logic hidden from domain model.
Presentation calls database directly	Controller writes SQL or calls repository without going through domain	UI and DB tightly coupled. No place for business logic. Changes ripple everywhere.
Data Source calls Domain	Repository or DAO validates data using domain services	Circular dependency. Cannot test either layer independently.
Skipping layers	Controller calls Repository directly, bypassing Service/Domain	Today's passthrough is tomorrow's business rule in wrong place.
Too many layers	7+ layers added 'just in case' or 'for flexibility'	Complexity without benefit. Slower development. Harder to trace code.
No clear boundaries	Classes from different layers mixed in same package/module	Cannot enforce rules. Layering degrades over time as shortcuts are taken.

Physical vs. Logical Layers

Layers are a LOGICAL organization of code—they don't necessarily mean separate servers, processes, or even separate projects. You might run all layers in one process (monolith) or distribute them across many machines. The logical separation is what matters for code organization; physical separation is a deployment decision made based on different criteria.

Configuration	Physical Structure	When to Use
Single Server (Monolith)	All layers in one process on one machine	Small apps, startups, low traffic, simple deployment needs
Two-Tier (Client-Server)	UI on client machine, Domain+Data on server	Desktop applications, rich clients, offline capability needed
Three-Tier	Web server + App server + Database server	Most web applications. Each tier can scale independently.
N-Tier + Microservices	Multiple app servers, services, load balancers, caches	High traffic, large teams, independent deployment needs

Fowler's Warning on Distribution: "Don't try to separate the layers into discrete processes unless you absolutely have to. Doing so will both degrade performance and add complexity, as you have to add things like Remote Facades and Data Transfer Objects." Distribution is a 'complexity booster'—only use it when the benefits clearly outweigh the costs.

Self-Assessment: Layering

Test Your Understanding - Answer These Questions:

1. Name the three principal layers and describe each one's primary responsibility in one sentence.

2. Why must dependencies point downward only? What specific problems occur if they point upward?

3. Your colleague put tax calculation logic in a JavaScript function on the checkout page. Explain what's wrong with this and what problems it will cause.

4. How would you test the business logic for calculating order totals without needing a database connection?

5. The Data Source layer needs to ensure data validity before saving. Where should validation logic actually live, and why?

6. What's the difference between logical layers and physical tiers? Can you have three logical layers running on one server?

Feynman Test: Explain layering to a non-programmer friend using the restaurant analogy. Can you do it clearly in 2 minutes without using any technical jargon?

Benefits That Justify the Complexity:

Testability: To test that promotion logic works correctly, create a Person object in memory, call promote(), verify the tier and salary changed correctly. No database needed. Tests run in milliseconds, not seconds.

Flexibility: Object structure can differ from table structure. You might have a Person object with an embedded Address value object, but store it as flat columns in the people table. The mapper handles the translation.

Clean Domain: Domain objects contain pure business logic with no infrastructure pollution. Business experts can read and understand the domain code without being confused by SQL and persistence details.

Complex Mappings: Can handle inheritance hierarchies, complex associations, value objects, and composite keys that would be awkward or impossible with simpler patterns.

Essential for Rich Domain Models: If your business logic is complex enough to warrant a Domain Model, you almost certainly need Data Mapper to keep that model clean and testable.

Choosing the Right Data Access Pattern

Domain Logic Pattern	Recommended Data Access	Reason
Transaction Script	Table Data Gateway or Row Data Gateway	Scripts work with data, not domain objects. Keep it simple.
Simple Domain Model	Active Record	When objects closely match tables, AR is simple and productive.
Rich Domain Model	Data Mapper	Complex domain needs clean separation from database structure.
Table Module	Table Data Gateway returning record sets	Table Module operates on record sets, TDG provides them.

Supporting Patterns: Unit of Work

When multiple domain objects change during a business transaction, you need to coordinate their database writes. Unit of Work tracks which objects have been created, modified, or deleted, and coordinates writing those changes to the database as a single atomic operation.

```java
public class UnitOfWork {
    private List<Object> newObjects = new ArrayList<>();
    private List<Object> dirtyObjects = new ArrayList<>();
    private List<Object> deletedObjects = new ArrayList<>();

    public void registerNew(Object obj) {
        newObjects.add(obj);
    }

    public void registerDirty(Object obj) {
        if (!newObjects.contains(obj) && !dirtyObjects.contains(obj)) {
            dirtyObjects.add(obj);
        }
    }

    public void registerDeleted(Object obj) {
        if (newObjects.remove(obj)) return;  // Was new, just don't insert
        dirtyObjects.remove(obj);  // No need to update
        deletedObjects.add(obj);
    }

    public void commit() {
        // Order matters for referential integrity!
        insertNew();    // 1. Insert new objects first (parents before children)
        updateDirty();  // 2. Update modified objects
        deleteRemoved(); // 3. Delete last (children before parents)
    }

    private void insertNew() {
        for (Object obj : newObjects) {
            MapperRegistry.getMapper(obj.getClass()).insert(obj);
        }
    }
    // ... similar for updateDirty() and deleteRemoved()
}
```

Supporting Patterns: Identity Map

Identity Map ensures that each database row is represented by exactly one object in memory during a session. Without it, loading the same row twice creates two different objects, which can lead to inconsistencies—you might update one object while another holds stale data.

```java
public class IdentityMap<T> {
    private Map<Long, T> map = new HashMap<>();

    public T get(Long id) {
        return map.get(id);
    }

    public void put(Long id, T obj) {
        map.put(id, obj);
    }

    public boolean contains(Long id) {
        return map.containsKey(id);
    }

    public void clear() {
        map.clear();
    }
}

// Usage in Mapper - ALWAYS check identity map before loading
public Person findById(Long id) {
    // Step 1: Check if already loaded this session
    Person cached = identityMap.get(id);
    if (cached != null) {
        return cached;  // Return SAME object reference
    }

    // Step 2: Load from database
    Person person = loadFromDatabase(id);

    // Step 3: Store in identity map for future requests
    identityMap.put(id, person);

    return person;
}

// Why this matters:
Order order1 = orderMapper.findById(100L);
Order order2 = orderMapper.findById(100L);
// With Identity Map: order1 == order2 (same object in memory)
// Without: order1 != order2 (different objects, potential inconsistency!)
```

Supporting Patterns: Lazy Load

Lazy Load delays loading of related objects until you actually access them. An Order object might have 100 line items—if you just need the order's date, you shouldn't load all those items. Lazy Load defers the database query until getItems() is called.

java

```java
public class Order {
    private Long id;
    private Long customerId;        // Store the foreign key value
    private Customer customer;      // Initially null
    private List<OrderItem> items;  // Initially null

    // Lazy loading using virtual proxy approach
    public Customer getCustomer() {
        if (customer == null) {
            // Load from database on first access
            customer = CustomerMapper.getInstance().findById(customerId);
        }
        return customer;
    }

    public List<OrderItem> getItems() {
        if (items == null) {
            // Load from database on first access
            items = OrderItemMapper.getInstance().findByOrderId(id);
        }
        return items;
    }
}
```

WARNING - The N+1 Query Problem:

Lazy loading can backfire catastrophically! Loading an order, then iterating through 100 line items, accessing each item's product... triggers 101 separate database queries (1 for the order + 100 for products). This is called the N+1 problem.

Solutions:
- Eager loading: Load related objects upfront with JOINs when you know you'll need them
- Batch loading: Load all 100 products in one query instead of 100 separate queries
- Careful design: Think about access patterns and choose loading strategy accordingly

Mapping Inheritance to Tables

Objects have inheritance hierarchies; relational databases don't. When your domain model has inheritance (Employee with subclasses Manager, Developer, Designer), you must choose how to represent this in database tables. Three strategies exist, each with different trade-offs.

Strategy	Table Structure	Query Performance	Storage Efficiency	Flexibility
Single Table Inheritance	One table for entire hierarchy with type discriminator column	Excellent - no JOINs needed	Poor - many NULL columns for subclass-specific fields	Easy to add new subclasses
Class Table Inheritance	One table per class in hierarchy, joined by primary key	Moderate - requires JOINs to load complete object	Good - no wasted space, normalized	Adding fields requires new columns in specific table
Concrete Table Inheritance	One table per concrete class, duplicating inherited columns	Good for single type, bad for polymorphic queries	Some duplication of inherited columns	Hard - schema changes affect multiple tables

Rule of Thumb: Start with Single Table Inheritance for simplicity unless you have many subclass-specific fields that would create too many NULL columns. Use Class Table Inheritance when subclasses have many unique fields. Use Concrete Table Inheritance only when you rarely need polymorphic queries across the hierarchy.

Chapter 3 Self-Assessment

Test Your Understanding:

1. What is the object-relational impedance mismatch? Name at least four specific mismatches.

2. Compare Table Data Gateway and Row Data Gateway. When would you use each?

3. What makes Active Record different from Row Data Gateway?

4. Why is Data Mapper essential for rich Domain Models? What benefits justify its complexity?

5. What problem does Unit of Work solve? What is the correct order for commit operations?

6. Explain the N+1 query problem and two ways to solve it.

7. Compare the three inheritance mapping strategies. When would you use each?

Feynman Test: Explain to a colleague why you would choose Data Mapper over Active Record for a complex e-commerce domain, and what additional patterns you would need to use with it.

Chapter 4: Web Presentation

Model-View-Controller in Web Applications

MVC is the foundational pattern for web presentation, separating the user interface into three distinct roles: Model (domain data and logic), View (displays data), and Controller (handles input and coordinates). Understanding MVC deeply is essential for building maintainable web applications.

Important Note: Web MVC differs significantly from the original GUI MVC pattern from Smalltalk in the 1970s. In GUI MVC, the View directly observes the Model and updates automatically when the Model changes (Observer pattern). In Web MVC, there's no persistent connection—each HTTP request is independent, and the Controller explicitly passes data to the View for each response. This request-response cycle fundamentally changes how the pattern works.

Web MVC Request Flow - Step by Step

Complete Request Processing Cycle:

1. User Action: User clicks a link, submits a form, or calls an API endpoint

2. HTTP Request: Browser/client sends HTTP request (GET, POST, PUT, DELETE) to server

3. Routing: Web framework examines URL and HTTP method, routes to appropriate Controller

4. Controller Processing:
• Parse and validate request parameters (format validation)
• Check authentication/authorization
• Call Service Layer or Domain objects
• Receive result from domain

5. View Selection: Controller selects appropriate View template based on result and requested format

6. View Rendering: View transforms model data into response format (HTML, JSON, XML)

7. HTTP Response: Server sends rendered response back to client

8. Display: Browser renders HTML or client application processes API response

Controller Patterns

Pattern	Structure	When to Use	Example
Page Controller	One controller class or method handles one page or action	Most web apps - simple, intuitive, easy to find code	Spring @Controller, Rails Controllers
Front Controller	Single entry point handler processes ALL requests	Central security, logging, routing, exception handling	Java Servlet Filter, DispatcherServlet
Application Controller	Separate object manages page flow and navigation logic	Complex wizards, multi-step forms, state machines	Workflow engines, booking flows

Page Controller Example

java

```java
@Controller
@RequestMapping("/orders")
public class OrderController {

    private final OrderService orderService;
    private final ProductService productService;

    // GET /orders - list all orders
    @GetMapping
    public String listOrders(Model model) {
        List<OrderDTO> orders = orderService.findAll();
        model.addAttribute("orders", orders);
        return "orders/list";  // View template name
    }

    // GET /orders/123 - view single order
    @GetMapping("/{id}")
    public String viewOrder(@PathVariable Long id, Model model) {
        OrderDTO order = orderService.findById(id);
        model.addAttribute("order", order);
        return "orders/view";
    }

    // GET /orders/new - show create form
    @GetMapping("/new")
    public String newOrderForm(Model model) {
        model.addAttribute("orderForm", new OrderForm());
        model.addAttribute("products", productService.findAllActive());
        return "orders/new";
    }

    // POST /orders - create new order
    @PostMapping
    public String createOrder(@Valid @ModelAttribute OrderForm form,
                             BindingResult result,
                             RedirectAttributes redirectAttributes) {
        // Format validation - is form data well-formed?
        if (result.hasErrors()) {
            return "orders/new";  // Re-show form with errors
        }

        try {
            // Business operation - delegate to service
            OrderDTO order = orderService.create(form.toRequest());
            redirectAttributes.addFlashAttribute("message", "Order created!");
    // ... (continued)
```

View Patterns

Pattern	How It Works	Advantages	Disadvantages
Template View	HTML template with embedded markers, tags, or expressions	Easy to design visually, WYSIWYG editing possible	Logic can creep into templates, mixing concerns
Transform View	Transform data programmatically (often using XSLT)	Multiple output formats easy, clean separation	Harder to visualize final design, learning curve
Two Step View	Create logical structure first, then render to HTML format	Consistent site-wide look and feel, easy global changes	More complex to implement, extra abstraction layer

Template View Example (Thymeleaf)

```html
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Order Details</title>
</head>
<body>
    <h1>Order #<span th:text="${order.id}">123</span></h1>

    <div class="customer-info">
        <p>Customer: <span th:text="${order.customer.name}">John Doe</span></p>
        <p>Email: <span th:text="${order.customer.email}">john@example.com</span></p>
    </div>

    <table class="order-items">
        <thead>
            <tr><th>Product</th><th>Qty</th><th>Price</th><th>Total</th></tr>
        </thead>
        <tbody>
            <!-- Loop through items - this is VIEW logic, acceptable -->
            <tr th:each="item : ${order.items}">
                <td th:text="${item.productName}">Widget</td>
                <td th:text="${item.quantity}">1</td>
                <td th:text="${#numbers.formatCurrency(item.price)}">$10.00</td>
                <td th:text="${#numbers.formatCurrency(item.lineTotal)}">$10.00</td>
            </tr>
        </tbody>
    </table>

    <div class="order-totals">
        <p>Subtotal: <span th:text="${#numbers.formatCurrency(order.subtotal)}"/></p>
        <p>Tax: <span th:text="${#numbers.formatCurrency(order.tax)}"/></p>
        <p><strong>Total: <span th:text="${#numbers.formatCurrency(order.total)}"/></stron
    </div>
</body>
</html>
```

Keep Views Dumb - The Golden Rule:

Views should ONLY display data provided by the controller. They should NEVER calculate, validate, or make business decisions.

Bad (business logic in view):
th:text="${order.total * (order.customer.tier == 'GOLD' ? 0.85 : 0.95)}"

Good (calculation done in domain/service, passed to view):
th:text="${order.discountedTotal}"

If you find yourself writing complex expressions or conditionals in views, that logic belongs in the controller (for presentation logic) or domain (for business logic).

Chapter 5: Concurrency

Understanding Concurrency Problems

Enterprise applications serve many users simultaneously. When two users try to read and modify the same data at the same time, serious problems can occur. Concurrency bugs are among the hardest to find and fix because they're intermittent—they only occur when operations happen to overlap in specific ways, which might happen once in a thousand requests.

The Four Major Concurrency Problems

1. Lost Update
Alice reads account balance: $1000
Bob reads account balance: $1000
Alice withdraws $300, saves: balance = $700
Bob withdraws $500, saves: balance = $500
Result: Alice's withdrawal is lost! Balance should be $200, but it's $500.

2. Inconsistent Read (Read Skew)
You read Account A balance: $500
Meanwhile, a transfer moves $200 from A to B
You read Account B balance: $700 (includes the $200 transfer)
Result: You see $500 + $700 = $1200, but actual total is $1000. Data appears inconsistent.

3. Deadlock
Thread 1: Locks Customer record, tries to lock Order record
Thread 2: Locks Order record, tries to lock Customer record
Result: Both threads wait forever for the other to release their lock. System hangs.

4. Phantom Read
Query 1: SELECT * WHERE status = 'PENDING' returns 10 rows
Another transaction inserts a new PENDING row
Query 2: SELECT * WHERE status = 'PENDING' returns 11 rows
Result: A "phantom" row appeared between your two queries within the same transaction.

Two Solutions: Optimistic vs. Pessimistic Locking

Aspect	Optimistic Locking	Pessimistic Locking
Assumption	Conflicts are rare; most transactions won't collide	Conflicts are common; better to prevent than detect
Mechanism	No locks during work; check version at save time	Lock data before editing; others must wait
When conflict detected	At commit time - transaction fails, user must retry	Before work starts - user waits or is told data is locked
User experience	Occasional 'please refresh and retry' message	May wait for lock release; or 'record is being edited by X'
Throughput	Higher - no blocking on reads or during work	Lower - lock contention can cause queuing
Implementation	Add version column to tables; check in UPDATE WHERE clause	Lock manager tracks who holds locks; timeout for stale locks
Best for	Web apps, short transactions, low conflict probability	Long transactions, high contention data, batch updates
Risk	Lost updates if not implemented correctly in all update paths	Deadlocks if lock ordering is not managed carefully

Optimistic Locking Implementation

```java
// Database table has: id, name, email, version INT NOT NULL DEFAULT 0

public class CustomerMapper {

    public void update(Customer customer) {
        // Key: include version in WHERE clause AND increment it
        int rowsUpdated = jdbcTemplate.update(
            "UPDATE customers " +
            "SET name = ?, email = ?, version = version + 1 " +
            "WHERE id = ? AND version = ?",  // Version check!
            customer.getName(),
            customer.getEmail(),
            customer.getId(),
            customer.getVersion()  // Must match current version
        );

        if (rowsUpdated == 0) {
            // Either record doesn't exist, or version changed (concurrent update)
            throw new OptimisticLockException(
                "Customer was modified by another user. Please refresh and retry.");
        }

        // Success - update in-memory version for future updates
        customer.setVersion(customer.getVersion() + 1);
    }
}

// Service layer handles the exception gracefully
@Service
public class CustomerService {

    @Transactional
    public void updateCustomer(CustomerUpdateRequest request) {
        try {
            Customer customer = customerRepository.findById(request.getId());
            customer.setName(request.getName());
            customer.setEmail(request.getEmail());
            customerRepository.save(customer);  // May throw OptimisticLockException
        } catch (OptimisticLockException e) {
            throw new ConcurrentModificationException(
                "This customer was modified by another user while you were editing. " +
                "Please refresh the page and try your changes again.");
        }
    }
}
```

Pessimistic Locking Implementation

java

```java
public class LockManager {
    private Map<String, LockInfo> locks = new ConcurrentHashMap<>();

    public boolean acquireLock(String resourceId, String userId, Duration timeout) {
        LockInfo existing = locks.get(resourceId);

        if (existing != null && !existing.isExpired()) {
            if (existing.getUserId().equals(userId)) {
                existing.extend(timeout);  // Already have it, extend
                return true;
            }
            return false;  // Someone else has it
        }

        // Acquire new lock
        locks.put(resourceId, new LockInfo(userId, Instant.now().plus(timeout)));
        return true;
    }

    public void releaseLock(String resourceId, String userId) {
        LockInfo lock = locks.get(resourceId);
        if (lock != null && lock.getUserId().equals(userId)) {
            locks.remove(resourceId);
        }
    }

    public Optional<String> getLockHolder(String resourceId) {
        LockInfo lock = locks.get(resourceId);
        if (lock != null && !lock.isExpired()) {
            return Optional.of(lock.getUserId());
        }
        return Optional.empty();
    }
}

// Usage in controller
@PostMapping("/customers/{id}/edit")
public String beginEditCustomer(@PathVariable Long id, Principal user) {
    String lockKey = "customer:" + id;

    if (!lockManager.acquireLock(lockKey, user.getName(), Duration.ofMinutes(15))) {
        Optional<String> holder = lockManager.getLockHolder(lockKey);
        return "error/locked";  // Show "being edited by " + holder
    }

    // ... (continued)
```

When to Choose Which:

Use Optimistic when: Web applications with short transactions, conflicts are rare (users editing different records), users can easily retry, high throughput is important.

Use Pessimistic when: Long-running transactions (document editing sessions), high conflict probability (popular records edited frequently), losing work would be costly, batch processing that must complete without interruption.

Chapter 6: Session State

The Stateless Web Challenge

HTTP is inherently stateless—each request is independent with no memory of previous requests. But users need continuity: shopping carts must persist across page views, login status must be maintained, multi-step wizards need to remember previous steps. Where you store this session state has major implications for scalability and architecture.

Storage Option	Where Data Lives	Advantages	Disadvantages
Client Session State	Cookies, URL parameters, hidden form fields, localStorage	Infinite server scalability, no server memory needed, survives server restarts	Size limits (~4KB cookies), security concerns, bandwidth overhead
Server Session State	Web server memory (HttpSession in Java, session in Rails)	Fast access, easy to use, can store any object type, secure	Memory per user, cluster synchronization problems, lost on server crash
Database Session State	Database table storing serialized session data	Survives server restarts, works with clusters, unlimited size	Slower access, database load, serialization complexity

Modern Trend - Stateless Architecture:

Minimize server-side session state whenever possible. Use tokens (JWT) for authentication that contain user identity claims. Store shopping carts in database (they're valuable business data anyway). Pass necessary state with each request or store in browser. This dramatically simplifies horizontal scaling—any server can handle any request because there's no sticky session to maintain.

Chapter 7: Distribution Strategies

Fowler's First Law of Distributed Object Design:

"Don't distribute your objects."

This counter-intuitive advice is one of the most important lessons in the book. Remote calls are approximately 1000x slower than local calls. Each remote call involves network latency, serialization/deserialization, and potential failures. Distribution adds complexity, new failure modes, and debugging difficulty. Only distribute when you absolutely must.

When Distribution Is Necessary

Sometimes distribution is unavoidable: clients must run on different machines, you need to integrate with external systems, or you need to scale beyond a single server's capacity. When you must distribute, use these patterns to minimize the pain:

Remote Facade:

Provides a coarse-grained interface over fine-grained objects. Instead of exposing many small methods (getName(), getEmail(), getAddress(), getOrders()...) that each require a network round-trip, create one method (getCustomerDetails()) that returns everything needed in a single call. This reduces network round-trips dramatically.

Data Transfer Object (DTO):

A simple object that carries data between processes. DTOs are optimized for data transfer, not domain modeling. They're typically just data with getters/setters, easily serializable to JSON/XML, designed to carry exactly what a specific use case needs.

java

```java
// Data Transfer Object - optimized for network transfer
public class CustomerDetailsDTO implements Serializable {
    public Long id;
    public String name;
    public String email;
    public AddressDTO address;
    public List<OrderSummaryDTO> recentOrders;
    public String membershipTier;
    public BigDecimal totalPurchases;
    // No behavior - just data containers!
}

// Remote Facade - coarse-grained interface
@Remote
public class CustomerFacade {
    private CustomerService customerService;
    private OrderService orderService;

    // ONE network call gets everything a client screen needs
    public CustomerDetailsDTO getCustomerDetails(Long customerId) {
        Customer customer = customerService.findById(customerId);
        List<Order> orders = orderService.findRecentByCustomer(customerId, 5);
        BigDecimal total = orderService.getTotalPurchases(customerId);

        // Assemble DTO with all needed data
        CustomerDetailsDTO dto = new CustomerDetailsDTO();
        dto.id = customer.getId();
        dto.name = customer.getName();
        dto.email = customer.getEmail();
        dto.address = AddressDTO.from(customer.getAddress());
        dto.recentOrders = orders.stream()
            .map(OrderSummaryDTO::from)
            .collect(toList());
        dto.membershipTier = customer.getTier().name();
        dto.totalPurchases = total;

        return dto;  // Everything in one response
    }
}
```

Chapter 8: Putting It All Together

Architecture Decision Framework

Now that we understand all the pieces, how do we make architectural decisions? Here's a practical framework for choosing patterns based on your specific situation.

Decision Point	If Simple / Low	If Complex / High
Business logic complexity	Transaction Script	Domain Model + Service Layer
Object-table mapping complexity	Active Record	Data Mapper + Unit of Work
Web framework needs	Page Controller + Template View	Front Controller + MVC
Conflict probability	Optimistic Locking	Pessimistic Locking
Scalability requirements	Server Session State	Stateless + Database/Client Session
Performance criticality	Keep all layers in one process	Add caching, consider read replicas
Team size and experience	Simpler patterns, fewer layers	More sophisticated patterns

Complete Architecture Examples

Example 1: Simple Internal CRUD Application
Scenario: Employee contact directory for 50-person company

Recommended Architecture:
Domain Logic: Transaction Script
Data Access: Row Data Gateway or Active Record
Web: Page Controller + Template View
Concurrency: Optimistic Locking (conflicts rare)
Session: Server Session State

Rationale: Low complexity, small user base, rapid development priority. Could be built in a few days with Rails or Django using their defaults.

Example 2: E-Commerce Platform
Scenario: Online store with complex pricing, inventory, promotions, loyalty program, high traffic

Recommended Architecture:
Domain Logic: Rich Domain Model + Service Layer
Data Access: Data Mapper (JPA/Hibernate) + Repository + Unit of Work
Web: Page Controller + REST API for mobile
Concurrency: Optimistic (general), Pessimistic (inventory reservations)
Session: Stateless (JWT auth) + Database (shopping carts are business data)

Rationale: Complex pricing and promotion rules need Domain Model. Multiple clients (web, mobile, API partners) share Service Layer. High traffic requires stateless for horizontal scaling.

Example 3: Financial Trading System
Scenario: Real-time trading, millisecond latency requirements, regulatory compliance, audit trail

Recommended Architecture:
Domain Logic: Very rich Domain Model with sophisticated patterns
Data Access: Custom optimized data access (ORM overhead too high)
Web: N/A - specialized trading protocols and desktop clients
Concurrency: Pessimistic for critical sections, custom conflict resolution
Session: Stateful connections, in-memory state for performance

Rationale: Extreme performance requirements demand custom solutions. Regulatory requirements mandate rich domain model for auditability and compliance.

Key Architecture Principles:

1. **Start simple** - you can always add complexity later; removing it is much harder
2. **Match complexity** - architecture complexity should match problem complexity
3. **Invest in domain** - that's where your unique business value lives
4. **Always layer** - even simple apps benefit from separation of concerns
5. **Learn incrementally** - don't try to use all patterns at once; add as needed

Benefits That Justify the Complexity:

Testability: To test that promotion logic works correctly, create a Person object in memory, call promote(), verify the tier and salary changed correctly. No database needed. Tests run in milliseconds, not seconds.

Flexibility: Object structure can differ from table structure. You might have a Person object with an embedded Address value object, but store it as flat columns in the people table. The mapper handles the translation.

Clean Domain: Domain objects contain pure business logic with no infrastructure pollution. Business experts can read and understand the domain code without being confused by SQL and persistence details.

Complex Mappings: Can handle inheritance hierarchies, complex associations, value objects, and composite keys that would be awkward or impossible with simpler patterns.

Essential for Rich Domain Models: If your business logic is complex enough to warrant a Domain Model, you almost certainly need Data Mapper to keep that model clean and testable.

Choosing the Right Data Access Pattern

Domain Logic Pattern	Recommended Data Access	Reason
Transaction Script	Table Data Gateway or Row Data Gateway	Scripts work with data, not domain objects. Keep it simple.
Simple Domain Model	Active Record	When objects closely match tables, AR is simple and productive.
Rich Domain Model	Data Mapper	Complex domain needs clean separation from database structure.
Table Module	Table Data Gateway returning record sets	Table Module operates on record sets, TDG provides them.

Supporting Patterns: Unit of Work

When multiple domain objects change during a business transaction, you need to coordinate their database writes. Unit of Work tracks which objects have been created, modified, or deleted, and coordinates writing those changes to the database as a single atomic operation.

java

```java
public class UnitOfWork {
    private List<Object> newObjects = new ArrayList<>();
    private List<Object> dirtyObjects = new ArrayList<>();
    private List<Object> deletedObjects = new ArrayList<>();

    public void registerNew(Object obj) {
        newObjects.add(obj);
    }

    public void registerDirty(Object obj) {
        if (!newObjects.contains(obj) && !dirtyObjects.contains(obj)) {
            dirtyObjects.add(obj);
        }
    }

    public void registerDeleted(Object obj) {
        if (newObjects.remove(obj)) return;  // Was new, just don't insert
        dirtyObjects.remove(obj);  // No need to update
        deletedObjects.add(obj);
    }

    public void commit() {
        // Order matters for referential integrity!
        insertNew();    // 1. Insert new objects first (parents before children)
        updateDirty();  // 2. Update modified objects
        deleteRemoved(); // 3. Delete last (children before parents)
    }

    private void insertNew() {
        for (Object obj : newObjects) {
            MapperRegistry.getMapper(obj.getClass()).insert(obj);
        }
    }
    // ... similar for updateDirty() and deleteRemoved()
}
```

Supporting Patterns: Identity Map

Identity Map ensures that each database row is represented by exactly one object in memory during a session. Without it, loading the same row twice creates two different objects, which can lead to inconsistencies—you might update one object while another holds stale data.

```java
public class IdentityMap<T> {
    private Map<Long, T> map = new HashMap<>();

    public T get(Long id) {
        return map.get(id);
    }

    public void put(Long id, T obj) {
        map.put(id, obj);
    }

    public boolean contains(Long id) {
        return map.containsKey(id);
    }

    public void clear() {
        map.clear();
    }
}

// Usage in Mapper - ALWAYS check identity map before loading
public Person findById(Long id) {
    // Step 1: Check if already loaded this session
    Person cached = identityMap.get(id);
    if (cached != null) {
        return cached;  // Return SAME object reference
    }

    // Step 2: Load from database
    Person person = loadFromDatabase(id);

    // Step 3: Store in identity map for future requests
    identityMap.put(id, person);

    return person;
}

// Why this matters:
Order order1 = orderMapper.findById(100L);
Order order2 = orderMapper.findById(100L);
// With Identity Map: order1 == order2 (same object in memory)
// Without: order1 != order2 (different objects, potential inconsistency!)
```

Supporting Patterns: Lazy Load

Lazy Load delays loading of related objects until you actually access them. An Order object might have 100 line items—if you just need the order's date, you shouldn't load all those items. Lazy Load defers the database query until getItems() is called.

java

```java
public class Order {
    private Long id;
    private Long customerId;        // Store the foreign key value
    private Customer customer;      // Initially null
    private List<OrderItem> items;  // Initially null

    // Lazy loading using virtual proxy approach
    public Customer getCustomer() {
        if (customer == null) {
            // Load from database on first access
            customer = CustomerMapper.getInstance().findById(customerId);
        }
        return customer;
    }

    public List<OrderItem> getItems() {
        if (items == null) {
            // Load from database on first access
            items = OrderItemMapper.getInstance().findByOrderId(id);
        }
        return items;
    }
}
```

WARNING - The N+1 Query Problem:

Lazy loading can backfire catastrophically! Loading an order, then iterating through 100 line items, accessing each item's product... triggers 101 separate database queries (1 for the order + 100 for products). This is called the N+1 problem.

Solutions:
- Eager loading: Load related objects upfront with JOINs when you know you'll need them
- Batch loading: Load all 100 products in one query instead of 100 separate queries
- Careful design: Think about access patterns and choose loading strategy accordingly

Mapping Inheritance to Tables

Objects have inheritance hierarchies; relational databases don't. When your domain model has inheritance (Employee with subclasses Manager, Developer, Designer), you must choose how to represent this in database tables. Three strategies exist, each with different trade-offs.

Strategy	Table Structure	Query Performance	Storage Efficiency	Flexibility
Single Table Inheritance	One table for entire hierarchy with type discriminator column	Excellent - no JOINs needed	Poor - many NULL columns for subclass-specific fields	Easy to add new subclasses
Class Table Inheritance	One table per class in hierarchy, joined by primary key	Moderate - requires JOINs to load complete object	Good - no wasted space, normalized	Adding fields requires new columns in specific table
Concrete Table Inheritance	One table per concrete class, duplicating inherited columns	Good for single type, bad for polymorphic queries	Some duplication of inherited columns	Hard - schema changes affect multiple tables

Rule of Thumb: Start with Single Table Inheritance for simplicity unless you have many subclass-specific fields that would create too many NULL columns. Use Class Table Inheritance when subclasses have many unique fields. Use Concrete Table Inheritance only when you rarely need polymorphic queries across the hierarchy.

Chapter 3 Self-Assessment

Test Your Understanding:

1. What is the object-relational impedance mismatch? Name at least four specific mismatches.

2. Compare Table Data Gateway and Row Data Gateway. When would you use each?

3. What makes Active Record different from Row Data Gateway?

4. Why is Data Mapper essential for rich Domain Models? What benefits justify its complexity?

5. What problem does Unit of Work solve? What is the correct order for commit operations?

6. Explain the N+1 query problem and two ways to solve it.

7. Compare the three inheritance mapping strategies. When would you use each?

Feynman Test: Explain to a colleague why you would choose Data Mapper over Active Record for a complex e-commerce domain, and what additional patterns you would need to use with it.

Chapter 4: Web Presentation

Model-View-Controller in Web Applications

MVC is the foundational pattern for web presentation, separating the user interface into three distinct roles: Model (domain data and logic), View (displays data), and Controller (handles input and coordinates). Understanding MVC deeply is essential for building maintainable web applications.

Important Note: Web MVC differs significantly from the original GUI MVC pattern from Smalltalk in the 1970s. In GUI MVC, the View directly observes the Model and updates automatically when the Model changes (Observer pattern). In Web MVC, there's no persistent connection—each HTTP request is independent, and the Controller explicitly passes data to the View for each response. This request-response cycle fundamentally changes how the pattern works.

Web MVC Request Flow - Step by Step

Complete Request Processing Cycle:

1. User Action: User clicks a link, submits a form, or calls an API endpoint

2. HTTP Request: Browser/client sends HTTP request (GET, POST, PUT, DELETE) to server

3. Routing: Web framework examines URL and HTTP method, routes to appropriate Controller

4. Controller Processing:
• Parse and validate request parameters (format validation)
• Check authentication/authorization
• Call Service Layer or Domain objects
• Receive result from domain

5. View Selection: Controller selects appropriate View template based on result and requested format

6. View Rendering: View transforms model data into response format (HTML, JSON, XML)

7. HTTP Response: Server sends rendered response back to client

8. Display: Browser renders HTML or client application processes API response

Controller Patterns

Pattern	Structure	When to Use	Example
Page Controller	One controller class or method handles one page or action	Most web apps - simple, intuitive, easy to find code	Spring @Controller, Rails Controllers
Front Controller	Single entry point handler processes ALL requests	Central security, logging, routing, exception handling	Java Servlet Filter, DispatcherServlet
Application Controller	Separate object manages page flow and navigation logic	Complex wizards, multi-step forms, state machines	Workflow engines, booking flows

Page Controller Example

java

```java
@Controller
@RequestMapping("/orders")
public class OrderController {

    private final OrderService orderService;
    private final ProductService productService;

    // GET /orders - list all orders
    @GetMapping
    public String listOrders(Model model) {
        List<OrderDTO> orders = orderService.findAll();
        model.addAttribute("orders", orders);
        return "orders/list";  // View template name
    }

    // GET /orders/123 - view single order
    @GetMapping("/{id}")
    public String viewOrder(@PathVariable Long id, Model model) {
        OrderDTO order = orderService.findById(id);
        model.addAttribute("order", order);
        return "orders/view";
    }

    // GET /orders/new - show create form
    @GetMapping("/new")
    public String newOrderForm(Model model) {
        model.addAttribute("orderForm", new OrderForm());
        model.addAttribute("products", productService.findAllActive());
        return "orders/new";
    }

    // POST /orders - create new order
    @PostMapping
    public String createOrder(@Valid @ModelAttribute OrderForm form,
                             BindingResult result,
                             RedirectAttributes redirectAttributes) {
        // Format validation - is form data well-formed?
        if (result.hasErrors()) {
            return "orders/new";  // Re-show form with errors
        }

        try {
            // Business operation - delegate to service
            OrderDTO order = orderService.create(form.toRequest());
            redirectAttributes.addFlashAttribute("message", "Order created!");
    // ... (continued)
```

View Patterns

Pattern	How It Works	Advantages	Disadvantages
Template View	HTML template with embedded markers, tags, or expressions	Easy to design visually, WYSIWYG editing possible	Logic can creep into templates, mixing concerns
Transform View	Transform data programmatically (often using XSLT)	Multiple output formats easy, clean separation	Harder to visualize final design, learning curve
Two Step View	Create logical structure first, then render to HTML format	Consistent site-wide look and feel, easy global changes	More complex to implement, extra abstraction layer

Template View Example (Thymeleaf)

html

```html
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Order Details</title>
</head>
<body>
    <h1>Order #<span th:text="${order.id}">123</span></h1>

    <div class="customer-info">
        <p>Customer: <span th:text="${order.customer.name}">John Doe</span></p>
        <p>Email: <span th:text="${order.customer.email}">john@example.com</span></p>
    </div>

    <table class="order-items">
        <thead>
            <tr><th>Product</th><th>Qty</th><th>Price</th><th>Total</th></tr>
        </thead>
        <tbody>
            <!-- Loop through items - this is VIEW logic, acceptable -->
            <tr th:each="item : ${order.items}">
                <td th:text="${item.productName}">Widget</td>
                <td th:text="${item.quantity}">1</td>
                <td th:text="${#numbers.formatCurrency(item.price)}">$10.00</td>
                <td th:text="${#numbers.formatCurrency(item.lineTotal)}">$10.00</td>
            </tr>
        </tbody>
    </table>

    <div class="order-totals">
        <p>Subtotal: <span th:text="${#numbers.formatCurrency(order.subtotal)}"/></p>
        <p>Tax: <span th:text="${#numbers.formatCurrency(order.tax)}"/></p>
        <p><strong>Total: <span th:text="${#numbers.formatCurrency(order.total)}"/></stron
    </div>
</body>
</html>
```

Keep Views Dumb - The Golden Rule:

Views should ONLY display data provided by the controller. They should NEVER calculate, validate, or make business decisions.

Bad (business logic in view):
th:text="${order.total * (order.customer.tier == 'GOLD' ? 0.85 : 0.95)}"

Good (calculation done in domain/service, passed to view):
th:text="${order.discountedTotal}"

If you find yourself writing complex expressions or conditionals in views, that logic belongs in the controller (for presentation logic) or domain (for business logic).

Chapter 5: Concurrency

Understanding Concurrency Problems

Enterprise applications serve many users simultaneously. When two users try to read and modify the same data at the same time, serious problems can occur. Concurrency bugs are among the hardest to find and fix because they're intermittent—they only occur when operations happen to overlap in specific ways, which might happen once in a thousand requests.

The Four Major Concurrency Problems

1. Lost Update
Alice reads account balance: $1000
Bob reads account balance: $1000
Alice withdraws $300, saves: balance = $700
Bob withdraws $500, saves: balance = $500
Result: Alice's withdrawal is lost! Balance should be $200, but it's $500.

2. Inconsistent Read (Read Skew)
You read Account A balance: $500
Meanwhile, a transfer moves $200 from A to B
You read Account B balance: $700 (includes the $200 transfer)
Result: You see $500 + $700 = $1200, but actual total is $1000. Data appears inconsistent.

3. Deadlock
Thread 1: Locks Customer record, tries to lock Order record
Thread 2: Locks Order record, tries to lock Customer record
Result: Both threads wait forever for the other to release their lock. System hangs.

4. Phantom Read
Query 1: SELECT * WHERE status = 'PENDING' returns 10 rows
Another transaction inserts a new PENDING row
Query 2: SELECT * WHERE status = 'PENDING' returns 11 rows
Result: A "phantom" row appeared between your two queries within the same transaction.

Two Solutions: Optimistic vs. Pessimistic Locking

Aspect	Optimistic Locking	Pessimistic Locking
Assumption	Conflicts are rare; most transactions won't collide	Conflicts are common; better to prevent than detect
Mechanism	No locks during work; check version at save time	Lock data before editing; others must wait
When conflict detected	At commit time - transaction fails, user must retry	Before work starts - user waits or is told data is locked
User experience	Occasional 'please refresh and retry' message	May wait for lock release; or 'record is being edited by X'
Throughput	Higher - no blocking on reads or during work	Lower - lock contention can cause queuing
Implementation	Add version column to tables; check in UPDATE WHERE clause	Lock manager tracks who holds locks; timeout for stale locks
Best for	Web apps, short transactions, low conflict probability	Long transactions, high contention data, batch updates
Risk	Lost updates if not implemented correctly in all update paths	Deadlocks if lock ordering is not managed carefully

Optimistic Locking Implementation

```java
// Database table has: id, name, email, version INT NOT NULL DEFAULT 0

public class CustomerMapper {

    public void update(Customer customer) {
        // Key: include version in WHERE clause AND increment it
        int rowsUpdated = jdbcTemplate.update(
            "UPDATE customers " +
            "SET name = ?, email = ?, version = version + 1 " +
            "WHERE id = ? AND version = ?",  // Version check!
            customer.getName(),
            customer.getEmail(),
            customer.getId(),
            customer.getVersion()  // Must match current version
        );

        if (rowsUpdated == 0) {
            // Either record doesn't exist, or version changed (concurrent update)
            throw new OptimisticLockException(
                "Customer was modified by another user. Please refresh and retry.");
        }

        // Success - update in-memory version for future updates
        customer.setVersion(customer.getVersion() + 1);
    }
}

// Service layer handles the exception gracefully
@Service
public class CustomerService {

    @Transactional
    public void updateCustomer(CustomerUpdateRequest request) {
        try {
            Customer customer = customerRepository.findById(request.getId());
            customer.setName(request.getName());
            customer.setEmail(request.getEmail());
            customerRepository.save(customer);  // May throw OptimisticLockException
        } catch (OptimisticLockException e) {
            throw new ConcurrentModificationException(
                "This customer was modified by another user while you were editing. " +
                "Please refresh the page and try your changes again.");
        }
    }
}
```

Pessimistic Locking Implementation

java

```java
public class LockManager {
    private Map<String, LockInfo> locks = new ConcurrentHashMap<>();

    public boolean acquireLock(String resourceId, String userId, Duration timeout) {
        LockInfo existing = locks.get(resourceId);

        if (existing != null && !existing.isExpired()) {
            if (existing.getUserId().equals(userId)) {
                existing.extend(timeout);  // Already have it, extend
                return true;
            }
            return false;  // Someone else has it
        }

        // Acquire new lock
        locks.put(resourceId, new LockInfo(userId, Instant.now().plus(timeout)));
        return true;
    }

    public void releaseLock(String resourceId, String userId) {
        LockInfo lock = locks.get(resourceId);
        if (lock != null && lock.getUserId().equals(userId)) {
            locks.remove(resourceId);
        }
    }

    public Optional<String> getLockHolder(String resourceId) {
        LockInfo lock = locks.get(resourceId);
        if (lock != null && !lock.isExpired()) {
            return Optional.of(lock.getUserId());
        }
        return Optional.empty();
    }
}

// Usage in controller
@PostMapping("/customers/{id}/edit")
public String beginEditCustomer(@PathVariable Long id, Principal user) {
    String lockKey = "customer:" + id;

    if (!lockManager.acquireLock(lockKey, user.getName(), Duration.ofMinutes(15))) {
        Optional<String> holder = lockManager.getLockHolder(lockKey);
        return "error/locked";  // Show "being edited by " + holder
    }

    // ... (continued)
```

When to Choose Which:

Use Optimistic when: Web applications with short transactions, conflicts are rare (users editing different records), users can easily retry, high throughput is important.

Use Pessimistic when: Long-running transactions (document editing sessions), high conflict probability (popular records edited frequently), losing work would be costly, batch processing that must complete without interruption.

Chapter 6: Session State

The Stateless Web Challenge

HTTP is inherently stateless—each request is independent with no memory of previous requests. But users need continuity: shopping carts must persist across page views, login status must be maintained, multi-step wizards need to remember previous steps. Where you store this session state has major implications for scalability and architecture.

Storage Option	Where Data Lives	Advantages	Disadvantages
Client Session State	Cookies, URL parameters, hidden form fields, localStorage	Infinite server scalability, no server memory needed, survives server restarts	Size limits (~4KB cookies), security concerns, bandwidth overhead
Server Session State	Web server memory (HttpSession in Java, session in Rails)	Fast access, easy to use, can store any object type, secure	Memory per user, cluster synchronization problems, lost on server crash
Database Session State	Database table storing serialized session data	Survives server restarts, works with clusters, unlimited size	Slower access, database load, serialization complexity

Modern Trend - Stateless Architecture:

Minimize server-side session state whenever possible. Use tokens (JWT) for authentication that contain user identity claims. Store shopping carts in database (they're valuable business data anyway). Pass necessary state with each request or store in browser. This dramatically simplifies horizontal scaling—any server can handle any request because there's no sticky session to maintain.

Chapter 7: Distribution Strategies

Fowler's First Law of Distributed Object Design:

"Don't distribute your objects."

This counter-intuitive advice is one of the most important lessons in the book. Remote calls are approximately 1000x slower than local calls. Each remote call involves network latency, serialization/deserialization, and potential failures. Distribution adds complexity, new failure modes, and debugging difficulty. Only distribute when you absolutely must.

When Distribution Is Necessary

Sometimes distribution is unavoidable: clients must run on different machines, you need to integrate with external systems, or you need to scale beyond a single server's capacity. When you must distribute, use these patterns to minimize the pain:

Remote Facade:

Provides a coarse-grained interface over fine-grained objects. Instead of exposing many small methods (getName(), getEmail(), getAddress(), getOrders()...) that each require a network round-trip, create one method (getCustomerDetails()) that returns everything needed in a single call. This reduces network round-trips dramatically.

Data Transfer Object (DTO):

A simple object that carries data between processes. DTOs are optimized for data transfer, not domain modeling. They're typically just data with getters/setters, easily serializable to JSON/XML, designed to carry exactly what a specific use case needs.

java

```java
// Data Transfer Object - optimized for network transfer
public class CustomerDetailsDTO implements Serializable {
    public Long id;
    public String name;
    public String email;
    public AddressDTO address;
    public List<OrderSummaryDTO> recentOrders;
    public String membershipTier;
    public BigDecimal totalPurchases;
    // No behavior - just data containers!
}

// Remote Facade - coarse-grained interface
@Remote
public class CustomerFacade {
    private CustomerService customerService;
    private OrderService orderService;

    // ONE network call gets everything a client screen needs
    public CustomerDetailsDTO getCustomerDetails(Long customerId) {
        Customer customer = customerService.findById(customerId);
        List<Order> orders = orderService.findRecentByCustomer(customerId, 5);
        BigDecimal total = orderService.getTotalPurchases(customerId);

        // Assemble DTO with all needed data
        CustomerDetailsDTO dto = new CustomerDetailsDTO();
        dto.id = customer.getId();
        dto.name = customer.getName();
        dto.email = customer.getEmail();
        dto.address = AddressDTO.from(customer.getAddress());
        dto.recentOrders = orders.stream()
            .map(OrderSummaryDTO::from)
            .collect(toList());
        dto.membershipTier = customer.getTier().name();
        dto.totalPurchases = total;

        return dto;  // Everything in one response
    }
}
```

Chapter 8: Putting It All Together

Architecture Decision Framework

Now that we understand all the pieces, how do we make architectural decisions? Here's a practical framework for choosing patterns based on your specific situation.

Decision Point	If Simple / Low	If Complex / High
Business logic complexity	Transaction Script	Domain Model + Service Layer
Object-table mapping complexity	Active Record	Data Mapper + Unit of Work
Web framework needs	Page Controller + Template View	Front Controller + MVC
Conflict probability	Optimistic Locking	Pessimistic Locking
Scalability requirements	Server Session State	Stateless + Database/Client Session
Performance criticality	Keep all layers in one process	Add caching, consider read replicas
Team size and experience	Simpler patterns, fewer layers	More sophisticated patterns

Complete Architecture Examples

Example 1: Simple Internal CRUD Application
Scenario: Employee contact directory for 50-person company

Recommended Architecture:
Domain Logic: Transaction Script
Data Access: Row Data Gateway or Active Record
Web: Page Controller + Template View
Concurrency: Optimistic Locking (conflicts rare)
Session: Server Session State

Rationale: Low complexity, small user base, rapid development priority. Could be built in a few days with Rails or Django using their defaults.

Example 2: E-Commerce Platform
Scenario: Online store with complex pricing, inventory, promotions, loyalty program, high traffic

Recommended Architecture:
Domain Logic: Rich Domain Model + Service Layer
Data Access: Data Mapper (JPA/Hibernate) + Repository + Unit of Work
Web: Page Controller + REST API for mobile
Concurrency: Optimistic (general), Pessimistic (inventory reservations)
Session: Stateless (JWT auth) + Database (shopping carts are business data)

Rationale: Complex pricing and promotion rules need Domain Model. Multiple clients (web, mobile, API partners) share Service Layer. High traffic requires stateless for horizontal scaling.

Example 3: Financial Trading System
Scenario: Real-time trading, millisecond latency requirements, regulatory compliance, audit trail

Recommended Architecture:
Domain Logic: Very rich Domain Model with sophisticated patterns
Data Access: Custom optimized data access (ORM overhead too high)
Web: N/A - specialized trading protocols and desktop clients
Concurrency: Pessimistic for critical sections, custom conflict resolution
Session: Stateful connections, in-memory state for performance

Rationale: Extreme performance requirements demand custom solutions. Regulatory requirements mandate rich domain model for auditability and compliance.

Key Architecture Principles:

1. **Start simple** - you can always add complexity later; removing it is much harder
2. **Match complexity** - architecture complexity should match problem complexity
3. **Invest in domain** - that's where your unique business value lives
4. **Always layer** - even simple apps benefit from separation of concerns
5. **Learn incrementally** - don't try to use all patterns at once; add as needed

Part 2: Complete Pattern Catalog Reference

This section provides a comprehensive reference for all 40+ patterns organized by category. Each pattern includes intent, when to use, key implementation notes, and relationships to other patterns.

Chapter 9: Domain Logic Patterns

Pattern	Intent	When to Use
Transaction Script	Organizes business logic by procedures where each handles a single request	Simple logic, CRUD-heavy, procedural team, rapid development
Domain Model	Object model of the domain incorporating both behavior and data	Complex logic, many interacting rules, need testability, long-term maintenance
Table Module	Single instance handles business logic for all rows in a database table	Moderate complexity, .NET DataSet environment, record-set oriented processing
Service Layer	Defines application boundary with available operations, coordinates responses	Always - provides API, handles transactions, security; keeps it thin!

Chapter 10: Data Source Architectural Patterns

Pattern	Intent	When to Use
Table Data Gateway	Object that acts as Gateway to a database table; one instance handles all rows	Transaction Script, want SQL isolated per table, simple data access
Row Data Gateway	Object that acts as Gateway to a single row; one instance per row	Transaction Script with more OO feel, objects that know how to persist
Active Record	Object that wraps a row, encapsulates database access, AND adds domain logic	Simple domain, 1:1 table mapping, Rails/Django style, rapid development
Data Mapper	Layer of mappers moving data between objects and database, keeping them independent	Rich Domain Model, object structure differs from tables, need testability

Chapter 11: Object-Relational Behavioral Patterns

Pattern	Intent	Key Implementation Notes
Unit of Work	Maintains list of objects affected by transaction; coordinates writing changes	Track new/dirty/deleted; commit order matters for referential integrity
Identity Map	Ensures each object gets loaded only once by keeping loaded objects in a map	Always check before loading; scope to business transaction; essential with Data Mapper
Lazy Load	Object that doesn't contain all data but knows how to get it when needed	Virtual proxy, value holder, or ghost; beware N+1 problem; consider eager loading

Chapter 12: Object-Relational Structural Patterns

Pattern	Intent
Identity Field	Saves database ID field in object to maintain identity between object and row
Foreign Key Mapping	Maps object reference to a foreign key column in the database
Association Table Mapping	Saves many-to-many association as a table with foreign keys to both tables
Dependent Mapping	Has one class perform the database mapping for a child class
Embedded Value	Maps an object into several fields of another object's table
Serialized LOB	Saves a graph of objects by serializing them into a single large object (BLOB/CLOB)
Single Table Inheritance	Represents inheritance hierarchy as single table with type discriminator column
Class Table Inheritance	Represents inheritance hierarchy with one table per class
Concrete Table Inheritance	Represents inheritance hierarchy with one table per concrete class
Inheritance Mappers	Structure to organize database mappers that handle inheritance hierarchies

Chapter 13: Object-Relational Metadata Mapping Patterns

Pattern	Intent
Metadata Mapping	Holds object-relational mapping details in metadata (XML, annotations, fluent config)
Query Object	Object that represents a database query, allowing queries to be composed programmatically
Repository	Mediates between domain and data mapping using collection-like interface for domain objects

Chapter 14: Web Presentation Patterns

Pattern	Intent
Model View Controller	Splits UI into three roles: Model (data), View (display), Controller (input handling)
Page Controller	Object that handles a request for a specific page or action on a web site
Front Controller	Controller that handles all requests for a web site through a single handler object
Template View	Renders info into HTML by embedding markers in an HTML page (JSP, Thymeleaf, ERB)
Transform View	View that transforms domain data element by element (XSLT style)
Two Step View	Turns domain data into HTML in two steps: logical page then HTML rendering
Application Controller	Centralized point for handling screen navigation and flow of an application

Chapter 15: Distribution Patterns

Pattern	Intent	When to Use
Remote Facade	Provides coarse-grained facade on fine-grained objects to improve network efficiency	When you must distribute; return all data for a use case in one call
Data Transfer Object	Object that carries data between processes to reduce number of method calls	Pair with Remote Facade; serialize exactly what remote client needs

Chapter 16: Offline Concurrency Patterns

Pattern	Intent	When to Use
Optimistic Offline Lock	Prevents conflicts by detecting a conflict at commit time via version check	Web apps, short transactions, low conflict probability, users can retry
Pessimistic Offline Lock	Prevents conflicts by not allowing concurrent business transactions on same data	Long transactions, high contention, losing work is costly
Coarse-Grained Lock	Locks a set of related objects with a single lock	Aggregates where editing one part affects the whole
Implicit Lock	Allows framework code to acquire locks on behalf of developers automatically	When lock management should be transparent to developers

Chapter 17: Session State Patterns

Pattern	Intent	Trade-off
Client Session State	Stores session state on the client (cookies, hidden fields, URL, localStorage)	Infinite scalability vs. size limits and security concerns
Server Session State	Keeps session state on server in a serialized form (HttpSession, session hash)	Simple and fast vs. memory use and cluster synchronization
Database Session State	Stores session state as committed data in database	Survives restarts, works in clusters vs. slower access, DB load

Chapter 18: Base Patterns

Pattern	Intent
Gateway	Object that encapsulates access to an external system or resource
Mapper	Object that sets up communication between two independent objects
Layer Supertype	Type that acts as supertype for all types in its layer
Separated Interface	Defines interface in separate package from implementation
Registry	Well-known object that other objects can use to find common objects and services
Value Object	Small simple object whose equality is based on value rather than identity
Money	Represents a monetary value with currency; handles rounding and arithmetic correctly
Special Case	Subclass that provides special behavior for particular cases (Null Object)
Plugin	Links classes during configuration rather than compilation
Service Stub	Removes dependence on problematic services during testing
Record Set	In-memory representation of tabular data

Appendix A: Complete Self-Assessment (40 Questions)

Layering (Questions 1-10)
1. Name the three principal layers and describe each one's primary responsibility.
2. Why must dependencies point downward only? What problems occur if they point upward?
3. What are the benefits of layering? List at least five.
4. What goes wrong when business logic is placed in the presentation layer?
5. How would you test business logic without requiring a database connection?
6. What is the difference between logical layers and physical tiers?
7. Why does Fowler warn against distributing layers across processes unnecessarily?
8. How should layers communicate with each other?
9. What is the Service Layer and when is it needed?
10. How do you know if your layering is implemented correctly?

Domain Logic (Questions 11-18)
11. When would you choose Transaction Script over Domain Model?
12. How does polymorphism help organize domain logic in a Domain Model?
13. What is an anemic domain model and why is it considered an anti-pattern?
14. What are the warning signs that Transaction Script needs to evolve to Domain Model?
15. How does Table Module differ from Domain Model?
16. What should the Service Layer contain versus what goes in domain objects?
17. How do you test domain logic in a Domain Model architecture?
18. What is the relationship between domain logic patterns and data access patterns?

Data Access (Questions 19-28)
19. What is the object-relational impedance mismatch? Name four specific mismatches.
20. How does Data Mapper differ from Active Record? When would you use each?
21. What problem does Unit of Work solve? What is the correct order for commit operations?
22. Why is Identity Map important? What problems occur without it?
23. What is the N+1 query problem and how do you solve it?
24. What are the three inheritance mapping strategies and when would you use each?
25. When would you choose Table Data Gateway versus Row Data Gateway?
26. How do you handle lazy loading without causing performance problems?
27. What is the Repository pattern and how does it relate to Data Mapper?
28. How do ORM frameworks like Hibernate implement these patterns?

Web, Concurrency, Distribution (Questions 29-40)
29. How does web MVC differ from the original Smalltalk MVC?
30. When would you use Front Controller instead of Page Controller?
31. What does "keep views dumb" mean? Give an example of violating this principle.
32. Explain the lost update problem and how optimistic locking solves it.
33. When is pessimistic locking preferred over optimistic locking?
34. What is a deadlock and how do you prevent deadlocks?
35. What are the three options for session state storage? Trade-offs of each?
36. Why does Fowler say "don't distribute your objects"?
37. What is the purpose of Remote Facade and Data Transfer Object?
38. How do you design for horizontal scalability?
39. What is the modern trend for session management in web applications?
40. How do you choose between different architecture patterns for a new project?

Appendix B: Architecture Decision Trees

Domain Logic Decision Tree

START: How complex is your business logic?

Simple (mostly CRUD, few business rules)
→ Use **Transaction Script**
→ Pair with Table Data Gateway or Row Data Gateway

Moderate (some rules, table-oriented data)
→ Consider **Table Module** if using .NET DataSets
→ Otherwise lean toward Transaction Script or simple Domain Model

Complex (many interacting rules, frequent changes)
→ Use **Domain Model**
→ Pair with Data Mapper
→ Add Unit of Work and Identity Map
→ Consider Repository for cleaner domain access

Always: Add Service Layer to define your application boundary

Data Access Decision Tree

START: What domain logic pattern are you using?

Transaction Script
→ Use **Table Data Gateway** (one class per table, returns raw data)
→ Or **Row Data Gateway** (one object per row, more OO feel)

Simple Domain Model (objects closely match tables)
→ Use **Active Record** (domain object handles its own persistence)
→ Quick to develop, good framework support (Rails, Django)

Rich Domain Model (complex object structure)
→ Use **Data Mapper** (separate mapper layer)
→ Add **Unit of Work** (track changes, coordinate commits)
→ Add **Identity Map** (prevent duplicate objects)
→ Consider **Repository** (collection-like interface)

Concurrency Decision Tree

START: How often do concurrent conflicts occur on your data?

Rarely (users typically edit different records)
→ Use **Optimistic Locking**
→ Add version column to tables
→ Check version in UPDATE WHERE clause
→ Handle conflict with "please refresh and retry" message

Often (many users compete for same popular records)
→ Use **Pessimistic Locking**
→ Implement lock manager with timeouts
→ Show "record being edited by X" message
→ Establish consistent lock ordering to prevent deadlocks

Mixed (some data is hot, some is not)
→ **Optimistic by default**, Pessimistic for hotspots
→ Example: Optimistic for customer records, Pessimistic for inventory

Appendix C: Code Templates

Service Layer Template

```java
@Service
@Transactional
public class OrderService {
    private final OrderRepository orderRepository;
    private final CustomerRepository customerRepository;
    private final InventoryService inventoryService;

    public OrderDTO createOrder(CreateOrderRequest request) {
        // 1. Validate request (application-level)
        validateRequest(request);

        // 2. Load domain objects through repositories
        Customer customer = customerRepository.findById(request.getCustomerId());

        // 3. Execute domain logic - domain objects do the work
        Order order = customer.createOrder(request.getItems());
        order.calculateTotals();

        // 4. Coordinate with other services if needed
        inventoryService.reserve(order.getItems());

        // 5. Persist through repository
        orderRepository.save(order);

        // 6. Return DTO, not domain object
        return OrderDTO.from(order);
    }
}
```

Repository Interface Template

```java
// Interface defined in domain layer
public interface OrderRepository {
    Order findById(Long id);
    List<Order> findByCustomer(Long customerId);
    List<Order> findByStatus(OrderStatus status);
    void save(Order order);
    void delete(Order order);
}

// Implementation in data source layer
public class JpaOrderRepository implements OrderRepository {
    private final EntityManager em;

    public Order findById(Long id) {
        return em.find(Order.class, id);
    }

    public List<Order> findByCustomer(Long customerId) {
        return em.createQuery(
            "SELECT o FROM Order o WHERE o.customer.id = :cid", Order.class)
            .setParameter("cid", customerId)
            .getResultList();
    }

    public void save(Order order) {
        if (order.getId() == null) {
            em.persist(order);
        } else {
            em.merge(order);
        }
    }
}
```

Domain Entity Template

```java
public class Order {
    private Long id;
    private Customer customer;
    private List<OrderItem> items = new ArrayList<>();
    private Money total;
    private OrderStatus status;
    private int version; // For optimistic locking

    // Constructor enforces invariants
    public Order(Customer customer) {
        if (customer == null) throw new IllegalArgumentException("Customer required");
        this.customer = customer;
        this.status = OrderStatus.DRAFT;
    }

    // Business logic methods - behavior lives with data
    public void addItem(Product product, int quantity) {
        if (status != OrderStatus.DRAFT) {
            throw new BusinessException("Cannot modify submitted order");
        }
        OrderItem item = new OrderItem(this, product, quantity);
        items.add(item);
        recalculateTotal();
    }

    public void submit() {
        validate();  // Enforce business rules
        this.status = OrderStatus.SUBMITTED;
    }

    private void validate() {
        if (items.isEmpty()) {
            throw new BusinessException("Order must have at least one item");
        }
        if (!customer.isActive()) {
            throw new BusinessException("Customer account is not active");
        }
    }

    private void recalculateTotal() {
        this.total = items.stream()
            .map(OrderItem::getLineTotal)
            .reduce(Money.ZERO, Money::add);
    }

    // ... (continued)
```

Value Object Template

java

```java
public final class Money implements Comparable<Money> {
    private final BigDecimal amount;
    private final Currency currency;

    public static final Money ZERO = new Money(BigDecimal.ZERO, Currency.USD);

    public Money(BigDecimal amount, Currency currency) {
        this.amount = amount.setScale(2, RoundingMode.HALF_UP);
        this.currency = currency;
    }

    public static Money dollars(double amount) {
        return new Money(BigDecimal.valueOf(amount), Currency.USD);
    }

    public Money add(Money other) {
        assertSameCurrency(other);
        return new Money(amount.add(other.amount), currency);
    }

    public Money subtract(Money other) {
        assertSameCurrency(other);
        return new Money(amount.subtract(other.amount), currency);
    }

    public Money multiply(int factor) {
        return new Money(amount.multiply(BigDecimal.valueOf(factor)), currency);
    }

    public boolean isGreaterThan(Money other) {
        assertSameCurrency(other);
        return amount.compareTo(other.amount) > 0;
    }

    private void assertSameCurrency(Money other) {
        if (!currency.equals(other.currency)) {
            throw new IllegalArgumentException("Currency mismatch");
        }
    }

    // Equality by VALUE, not identity
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Money)) return false;
        Money other = (Money) o;
    // ... (continued)
```

Appendix D: Glossary of Key Terms

Active Record: Pattern where a domain object wraps a database row, knows how to persist itself, AND contains domain logic. Popular in Rails and Django.

Anemic Domain Model: Anti-pattern where domain objects have only data (getters/setters) with no behavior. All logic lives in service classes. Worst of both worlds.

Data Mapper: A separate layer that moves data between domain objects and database while keeping them independent. Essential for rich Domain Models.

Data Transfer Object (DTO): Simple object carrying data between processes or layers. Optimized for transfer, not domain modeling.

Domain Model: Object-oriented model of the business domain incorporating both behavior (methods) and data (attributes).

Front Controller: Single handler object that processes all requests for a web site, providing central control point.

Identity Map: Ensures each database row is represented by exactly one object during a session. Prevents duplicate objects and ensures consistency.

Impedance Mismatch: The fundamental differences between object-oriented programming and relational databases that make translation difficult.

Lazy Load: Delays loading of related objects until they're actually accessed. Saves loading data you don't need.

N+1 Problem: Performance anti-pattern where loading N related objects requires N+1 database queries instead of 1 or 2. Often caused by lazy loading in loops.

Optimistic Locking: Concurrency control that allows concurrent access, detecting conflicts at save time via version checking.

Pessimistic Locking: Concurrency control that prevents concurrent access by locking data before editing begins.

Remote Facade: Provides coarse-grained interface for remote access, returning all needed data in one call to minimize network round-trips.

Repository: Provides collection-like interface for accessing domain objects, mediating between domain and data mapping layers.

Service Layer: Defines application boundary with available operations. Coordinates domain objects and handles cross-cutting concerns.

Table Data Gateway: Object that handles all SQL operations for a single database table. Returns raw data structures.

Transaction Script: Organizes business logic as procedures where each handles one complete business transaction from start to finish.

Unit of Work: Tracks objects affected by a business transaction and coordinates writing all changes as a single atomic operation.

Value Object: Small immutable object whose equality is based on its values rather than its identity. Examples:

Money, Address, DateRange.

Final Summary: The Essential Patterns

The Core Architecture Pattern: LAYERING
Every enterprise application should have layers. Presentation → Domain → Data Source. Dependencies point downward only. This enables testability, maintainability, and flexibility.

The Core Domain Decision: Transaction Script vs. Domain Model
Simple logic → Transaction Script. Complex logic → Domain Model. Start simple, evolve when complexity warrants.

The Core Data Decision: Active Record vs. Data Mapper
Simple 1:1 mapping → Active Record. Complex domain needing testability → Data Mapper with Unit of Work and Identity Map.

The Core Concurrency Decision: Optimistic vs. Pessimistic
Rare conflicts → Optimistic (version check at save). Common conflicts → Pessimistic (lock before edit).

The Essential Behavioral Patterns
Unit of Work + Identity Map + Lazy Load = foundation for sophisticated object-relational mapping.

The Essential Web Patterns
MVC + Page Controller + Template View = foundation for maintainable web applications.

Remember: Patterns are tools, not rules. Use what you need when you need it.

Congratulations!

You have completed this comprehensive 80+ page guide to Patterns of Enterprise Application Architecture. You should now be able to:

- Explain all major patterns to colleagues using simple analogies
- Choose appropriate patterns based on your specific project requirements
- Implement patterns correctly with proper separation of concerns
- Avoid common architectural mistakes
- Design maintainable, testable, scalable enterprise applications

The Feynman technique says: if you can explain it simply, you understand it well. Practice teaching these concepts to others. Draw the diagrams from memory. Apply patterns to real projects. The more you use them, the more natural they become.

Go forth and build great software!

Extended Pattern Deep Dives

Transaction Script - Implementation Patterns

Organizing Transaction Scripts Effectively:

1. By Business Area: Group related scripts into service classes - OrderService, CustomerService, InventoryService. Each service handles one cohesive set of operations.

2. One Method Per Use Case: processOrder(), cancelOrder(), generateInvoice(). Each method is a complete script handling one business transaction.

3. Clear Input/Output: Methods receive primitives or simple DTOs, return results or throw exceptions. Avoid side effects that aren't obvious from the method signature.

4. Transaction Boundaries: Each script typically runs in one database transaction. Start at the beginning of the script, commit at the end, rollback on any error.

5. Error Handling: Catch database exceptions, wrap in business exceptions with meaningful messages, ensure transaction rollback happens properly.

6. Avoiding Duplication: When the same logic appears in multiple scripts, extract to private helper methods. But when helpers become complex with many parameters, that's a sign you need Domain Model.

Domain Model - Design Principles

Principles for Effective Domain Models:

Tell, Don't Ask: Instead of asking an object for data and making decisions externally, tell the object to do something. Bad: if (order.getStatus() == DRAFT) order.setStatus(SUBMITTED). Good: order.submit() - let the order handle its own state transition with proper validation.

Encapsulate Collections: Never return mutable collections directly. Return unmodifiable views or defensive copies. Bad: return items; Good: return Collections.unmodifiableList(items);

Protect Invariants: Objects should always be in a valid state. An Order without a Customer is invalid - prevent it in the constructor, not just validation.

Use Value Objects Liberally: Money, Email, PhoneNumber, Address, DateRange - these should be classes, not primitives. They can have validation and behavior.

Aggregate Roots: Identify clusters of objects that must be consistent together. Access children only through the root. Order is aggregate root; OrderItems accessed through Order, not directly.

Domain Events: When something significant happens (OrderPlaced, PaymentReceived), raise a domain event. This decouples the action from reactions to it.

Data Mapper - Advanced Techniques

Advanced Data Mapper Implementation:

Handling Associations:
One-to-Many: Order has many OrderItems. Load items lazily unless you know you need them. Store parent reference in children for navigation both directions.

Many-to-Many: Student has many Courses, Course has many Students. Use association table. Consider whether the association itself has data (enrollment date, grade) - if so, create association class.

Optimizing Queries:
Batch Loading: Instead of loading customers one at a time, load all needed customers in one query: SELECT * FROM customers WHERE id IN (1,2,3,4,5).

Eager Loading: When you know you'll need related data, load it upfront with JOINs: SELECT o.*, c.* FROM orders o JOIN customers c ON o.customer_id = c.id.

Query Object: For complex queries, build query programmatically rather than string concatenation. Prevents SQL injection and makes composition easier.

Handling Concurrency:
Version column in mapper: UPDATE ... SET version = version + 1 WHERE id = ? AND version = ?. If rows affected is 0, throw OptimisticLockException.

Repository Pattern - Complete Guide

Repository Pattern Deep Dive:

Purpose: Repository provides a collection-like interface for accessing domain objects. It mediates between the domain and data mapping layers, providing a more object-oriented view of the persistence layer.

Interface Design:
Basic operations: findById(id), findAll(), save(entity), delete(entity)
Domain-specific finders: findByStatus(status), findActiveCustomers(), findOverdueOrders()
Specification pattern: findAll(Specification spec) for complex queries

Implementation:
Repository implementation lives in data source layer, uses Data Mapper or ORM internally
Encapsulates query construction, caching, identity map access
Returns domain objects, never raw database structures

Testing:
Repository interface in domain layer enables easy mocking
Create in-memory implementations for unit tests
Integration tests verify real repository with database

Complete MVC Implementation Guide

Implementing MVC Correctly:

Model:
Domain objects (Customer, Order) or DTOs for display
Never HTTP-specific objects (HttpServletRequest)
Contains data to display plus any calculated values

View:
Template files (Thymeleaf, JSP, Razor, ERB)
Only display logic - loops, conditionals for display
NO business calculations, NO database access
Receives model, produces response (HTML, JSON)

Controller:
Receives HTTP request, extracts parameters
Validates input format (not business rules)
Calls Service Layer for business operations
Selects appropriate view based on result
Adds model attributes for view to render

Common Mistakes:
Business logic in controller - move to service/domain
Database access in controller - use service layer
Complex calculations in view - calculate in controller
Fat controllers - keep thin, delegate to services

Concurrency - Real World Scenarios

Scenario 1: E-Commerce Checkout
Problem: Two customers try to buy the last item in stock
Solution: Pessimistic lock on inventory during checkout. First customer locks, completes purchase. Second customer sees "item no longer available."

Scenario 2: Document Editing
Problem: Two editors modify same document simultaneously
Solution: Pessimistic lock when editor opens document. Show "being edited by X" to others. Timeout releases stale locks.

Scenario 3: Order Status Updates
Problem: Warehouse marks order shipped while customer service cancels it
Solution: Optimistic locking with version. Whichever saves second fails, must reload and decide.

Scenario 4: Banking Transfer
Problem: Must debit one account and credit another atomically
Solution: Database transaction wrapping both operations. Both succeed or both fail. Use SELECT FOR UPDATE for account rows.

Framework Implementation Guide

How Popular Frameworks Implement These Patterns

Spring Framework (Java):
Service Layer: @Service annotation, @Transactional for transactions
Repository: Spring Data JPA provides automatic implementations
Data Mapper: JPA/Hibernate with @Entity, @OneToMany, etc.
Unit of Work: JPA EntityManager, Hibernate Session
MVC: @Controller, @GetMapping, @PostMapping, Thymeleaf templates

Ruby on Rails:
Active Record: Default pattern - models inherit from ApplicationRecord
MVC: Convention-based controllers, ERB views
Service Layer: Service objects (plain Ruby classes) when needed
Repository: Not common - Active Record provides queries directly

.NET Entity Framework:
Data Mapper: DbContext with DbSet collections
Unit of Work: DbContext tracks changes, SaveChanges() commits
Repository: Often implemented on top of DbContext
MVC: ASP.NET MVC with Razor views

Django (Python):
Active Record: Models inherit from django.db.models.Model
MVC (MTV): Views (controllers), Templates (views), Models
QuerySet: Lazy loading, chainable queries
Service Layer: Service modules when logic grows complex

Common Architectural Mistakes - Detailed Analysis

Mistake 1: Business Logic in Controllers
Example: Controller calculates discount based on customer tier and order total
Why Bad: Can't reuse for mobile app. Can't unit test without HTTP. Logic duplicated when adding API.
Fix: Move to Service Layer or Domain Model. Controller only coordinates.

Mistake 2: Anemic Domain Model
Example: Customer class has only getters/setters. CustomerService has all the logic.
Why Bad: Loses encapsulation benefit. Logic scattered across services. Hard to find where rules are.
Fix: Put behavior in domain objects. Customer.applyDiscount() not CustomerService.applyDiscount(customer).

Mistake 3: God Service
Example: OrderService with 50+ methods handling everything order-related.
Why Bad: Too large to understand. Changes affect unrelated operations. Testing is painful.
Fix: Split into focused services: OrderCreationService, OrderFulfillmentService, OrderQueryService.

Mistake 4: Repository Returning DTOs
Example: CustomerRepository.findById() returns CustomerDTO instead of Customer entity.
Why Bad: Repository should work with domain objects. DTO conversion is controller/service concern.
Fix: Repository returns domain objects. Service converts to DTO before returning to controller.

Mistake 5: Circular Dependencies
Example: OrderService depends on CustomerService, CustomerService depends on OrderService.
Why Bad: Can't test either independently. Tight coupling. Hard to understand dependencies.
Fix: Extract shared logic to third service. Or use events for cross-service communication.

Testing Strategies by Pattern

Testing Transaction Scripts:
Challenge: Scripts often have database access mixed with logic
Strategy: Extract data access to gateways, mock gateways in tests
Alternative: Use in-memory database for integration tests

Testing Domain Model:
Advantage: Domain objects are pure - no infrastructure dependencies
Strategy: Create objects directly, call methods, verify state
Example: Order order = new Order(customer); order.addItem(product, 2); assertEquals(expectedTotal, order.getTotal());

Testing Service Layer:
Strategy: Mock repositories and dependent services
Focus: Verify coordination logic, transaction boundaries, exception handling
Example: when(orderRepo.findById(1L)).thenReturn(order); service.cancelOrder(1L); verify(order).cancel();

Testing Repository:
Strategy: Integration tests with real database (test container or in-memory)
Focus: Verify queries return correct data, mappings work correctly
Example: Insert test data, call findByStatus(PENDING), verify correct orders returned

Testing Controllers:
Strategy: Mock service layer, test request/response mapping
Focus: URL routing, parameter binding, view selection, HTTP status codes
Example: MockMvc tests in Spring, request specs in Rails

Performance Optimization Guide

N+1 Query Solutions:
1. Eager Loading: Fetch related data with JOINs upfront
2. Batch Loading: Load all related items in one query (WHERE id IN (...))
3. Subselect: Second query fetches all related items for all parents
4. Query optimization: Write custom queries for specific use cases

Caching Strategies:
First Level: Identity Map within transaction (automatic in ORM)
Second Level: Shared cache across transactions (Ehcache, Redis)
Query Cache: Cache query results for read-heavy scenarios
Application Cache: Cache computed results at service layer

Connection Pool Tuning:
Size: Typically 10-20 connections per application instance
Timeout: Set reasonable connection wait timeout
Validation: Validate connections before use
Monitoring: Track pool utilization and wait times

Read Replicas:
Route read queries to replicas
Accept eventual consistency for reads
Write to primary only
Use for reporting and analytics queries

Migration and Evolution Strategies

Evolving from Transaction Script to Domain Model:

Phase 1: Extract Value Objects
Start with simple concepts: Money, Email, Address
Low risk, immediate benefit (validation, type safety)
Scripts start using value objects instead of primitives

Phase 2: Extract Domain Entities
Identify the most complex or duplicated logic areas
Create entity classes with that behavior
Scripts delegate to entities for that logic
Keep scripts working throughout - no big bang rewrite

Phase 3: Add Repository Layer
Create repository interfaces for entity access
Implement with existing data access code
Scripts use repositories instead of direct database access

Phase 4: Evolve to Service Layer
Scripts become thin coordinators
Business logic lives in domain entities
Scripts transform into service methods

Phase 5: Add Data Mapper (if needed)
When Active Record becomes limiting
Gradually introduce mappers for complex entities
Domain objects become persistence-ignorant

Key Principle: Incremental change. Never rewrite. Keep system working throughout migration.

Architecture Checklist

Layering Checklist:
[ ] Can test business logic without web server running?
[ ] Can test business logic without database connection?
[ ] Switching databases only changes Data Source layer?
[ ] Adding mobile app reuses existing Domain layer?
[ ] All dependencies point downward (never up)?

Domain Model Checklist:
[ ] Behavior lives in domain objects, not just services?
[ ] Using polymorphism instead of if/else type checks?
[ ] Domain objects have no database/HTTP code?
[ ] Value objects for Money, Email, Address, etc.?
[ ] Collections are encapsulated (defensive copies)?
[ ] Invariants enforced (can't create invalid objects)?

Data Access Checklist:
[ ] Using Identity Map to prevent duplicate objects?
[ ] Using Unit of Work for coordinated saves?
[ ] Lazy loading where appropriate?
[ ] Aware of and preventing N+1 queries?
[ ] Appropriate inheritance mapping strategy?

Concurrency Checklist:
[ ] Identified which data has conflict potential?
[ ] Chosen optimistic or pessimistic per scenario?
[ ] Version columns on optimistically locked tables?
[ ] Lock ordering established to prevent deadlocks?
[ ] User-friendly conflict messages implemented?

Testing Checklist:
[ ] Domain logic has unit tests (no infrastructure)?
[ ] Service layer has tests with mocked dependencies?
[ ] Repository has integration tests with database?
[ ] Controllers have request/response tests?
[ ] End-to-end tests for critical paths?

Quick Reference Cards

Pattern Selection Quick Reference:

Simple CRUD app → Transaction Script + Table Gateway + Page Controller
Moderate complexity → Table Module + Table Gateway (if .NET)
Complex business logic → Domain Model + Data Mapper + Service Layer
High traffic web app → Stateless sessions + Database session + Optimistic lock
Long form editing → Pessimistic locks + Server session
Multiple UI clients → Service Layer + REST API + DTOs
External integrations → Gateway + Adapter patterns
Complex queries → Repository + Query Object

Layer Responsibilities Quick Reference:

Presentation: Display data, accept input, format validation, navigation
NOT: Business logic, database access, business validation

Service Layer: Coordinate operations, transaction boundaries, security, DTO conversion
NOT: Business decisions, UI concerns, direct data access

Domain: Business rules, calculations, validations, domain events
NOT: UI concerns, database operations, external systems

Data Source: Database queries, external APIs, file I/O, caching
NOT: Business logic, presentation concerns

Conclusion: Your Learning Journey

What You've Learned:

You've completed a comprehensive journey through Patterns of Enterprise Application Architecture. You now understand:

• **Layering** - the foundation that enables all other good practices
• **Domain Logic organization** - Transaction Script, Domain Model, and when to use each
• **Data Access patterns** - from simple gateways to sophisticated Data Mappers
• **Supporting patterns** - Unit of Work, Identity Map, Repository
• **Web presentation** - MVC and its variations
• **Concurrency** - optimistic and pessimistic strategies
• **Distribution** - when to avoid it and how to do it when necessary

The Path Forward:

1. **Practice explaining** - Use the Feynman technique. Teach these concepts to colleagues.
2. **Identify patterns** - Look at frameworks you use. Where do they implement these patterns?
3. **Apply incrementally** - Don't try to use everything at once. Add patterns as you encounter the problems they solve.
4. **Read the original** - This guide is comprehensive but the original 533-page book has even more depth and examples.
5. **Keep learning** - Domain-Driven Design, Clean Architecture, and Microservices all build on these foundations.

Remember: The goal is not to use patterns everywhere, but to have them available as tools when you face the problems they solve. Good architects know many patterns but apply them judiciously.

Congratulations on completing this 80+ page comprehensive guide!

You now have the knowledge to design and build maintainable, testable, scalable enterprise applications. The patterns in this book have stood the test of time - they're as relevant today as when they were first documented over 20 years ago.

Go build something great!

Extended Architecture Examples

Banking System Architecture

Requirements: Security, compliance, audit, millions of accounts, real-time balances.

Choices: Domain Model + Data Mapper + Pessimistic Locking + Stateless JWT
Why: Complex rules need DM. Audit needs UoW tracking. Financial data needs pessimistic locks.

CMS Architecture

Requirements: Content types, workflows, versioning, multi-channel.

Choices: Domain Model + Active Record + Pessimistic Locking + Server Session
Why: Complex workflows. Simple table mapping. Long editing sessions need locks.

Inventory System Architecture

Requirements: Multi-warehouse, real-time stock, scanning, lot tracking.

Choices: Domain Model + Data Mapper + Repository + Mixed Locking + Stateless
Why: Complex allocation rules. Repository hides query complexity. Stateless for mobile.

Interview Questions

Q: Transaction Script vs Domain Model?
A: TS is procedural, DM is OO. TS for simple, DM for complex.

Q: Active Record vs Data Mapper?
A: AR combines domain+persistence, DM separates them. DM for testability.

Q: Optimistic vs Pessimistic?
A: Optimistic detects conflicts at save. Pessimistic prevents them with locks.

Q: N+1 Problem?
A: Loading N children in N queries. Fix with eager loading or batching.

Implementation Checklist

Layering:
[ ] Dependencies point downward only
[ ] Can test domain without database
[ ] Can change database without affecting domain

Domain Model:
[ ] Behavior in objects, not services
[ ] Value objects for Money, Email, etc.
[ ] Collections encapsulated
[ ] Invariants enforced

Data Access:
[ ] Identity Map prevents duplicates
[ ] Unit of Work tracks changes
[ ] N+1 queries prevented
[ ] Appropriate lazy/eager loading

Concurrency:
[ ] Version columns for optimistic
[ ] Lock ordering for pessimistic
[ ] User-friendly conflict messages

Final Ten Takeaways

1. **Layer Everything** - Foundation of maintainability
2. **Match Complexity** - Simple patterns for simple apps
3. **Business Logic in Domain** - Not UI, not database
4. **Service Layer as Boundary** - Thin coordination
5. **Choose Data Access Wisely** - Match domain complexity
6. **Design for Testing** - If hard to test, refactor
7. **Handle Concurrency** - Don't ignore it
8. **Avoid Premature Distribution** - Remote calls are expensive
9. **Evolve Incrementally** - Start simple
10. **Patterns Are Tools** - Use appropriately

Congratulations! You've completed this comprehensive guide. These patterns will serve you throughout your career. Go build great software!