Patterns of Enterprise Application Architecture

The Complete Learning Guide

With Detailed Explanations, Visual Diagrams & Code Examples

Based on the seminal work by Martin Fowler, David Rice,
Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford

Master These Concepts in One Week:

- Layer your applications for maintainability and testability
- Choose the right approach for organizing business logic
- Bridge the gap between objects and relational databases
- Structure web presentations with MVC and related patterns
- Handle concurrent users safely with locking strategies
- Manage session state in stateless web environments
- Know when (and when NOT) to distribute your application
- Apply 40+ proven architectural patterns confidently

7-Day Study Plan for Complete Mastery

Day	Focus	Chapters	Key Patterns to Master
1	Foundation	Intro + Ch.1	Layering, Three-tier architecture
2	Domain Logic	Ch.2	Transaction Script, Domain Model, Table Module
3	Data Access	Ch.3 (Part 1)	Table/Row Data Gateway, Active Record
4	Data Access	Ch.3 (Part 2)	Data Mapper, Unit of Work, Identity Map
5	Web + Concurrency	Ch.4-5	MVC, Page Controller, Optimistic Lock
6	State + Distribution	Ch.6-8	Session patterns, Remote Facade, DTO
7	Review + Catalog	Ch.9-18	Complete pattern reference review

How to Use This Guide Effectively

- **Active Reading:** Don't just read—draw diagrams, write notes, explain concepts aloud
- **Code Along:** Type the code examples yourself; modify them and observe the results
- **Connect to Experience:** For each pattern, think of projects where you could apply it
- **Focus on Trade-offs:** Every pattern has pros and cons; understanding these is key
- **Review Daily:** Spend 10 minutes reviewing previous days' material before new content

Introduction: Enterprise Applications Explained

What Are Enterprise Applications?

Enterprise applications are the software systems that run businesses. They handle critical operations like processing orders, managing inventory, tracking finances, and serving customers. Unlike desktop software or games, enterprise applications have unique challenges that require specific architectural approaches.

Think of it this way: A video game needs to be fast and fun. A word processor needs to be responsive and feature-rich. But an enterprise application needs to be reliable, handle thousands of users, store years of data, enforce complex business rules, and integrate with dozens of other systems—all while being maintainable by teams that change over time.

Key Characteristics of Enterprise Applications

Characteristic	What It Means	Real Example
Persistent Data	Data survives restarts, stored for years	Bank stores 20 years of transactions
Lots of Data	Gigabytes to petabytes of information	Retailer tracks millions of products
Concurrent Users	Many users access simultaneously	500 employees using HR system at 9 AM
Complex Logic	Intricate business rules that change often	Tax calculations vary by state, product, date
Integration	Must work with other systems and services	E-commerce connects to payment, shipping, CRM
Multiple Interfaces	Web, mobile, API, reports, etc.	Same inventory data for web, app, POS

Examples of Enterprise Applications

Banking System: Manages accounts, processes transactions, calculates interest, detects fraud, generates statements, handles loans—serving millions of customers through ATMs, websites, apps, and branches.

E-Commerce Platform: Product catalog with millions of items, shopping carts, order processing, inventory management, payment processing, shipping coordination, customer reviews, recommendations.

Healthcare System: Patient records, appointment scheduling, prescription management, billing, insurance claims, lab results, clinical decision support—with strict privacy and regulatory requirements.

ERP (Enterprise Resource Planning): Integrates finance, HR, manufacturing, supply chain, sales, and procurement into one system that thousands of employees use daily.

What Are Patterns?

A pattern is a proven solution to a recurring problem in a specific context. Patterns aren't finished code you copy—they're templates you adapt. Each pattern describes what the solution looks like, when to use it, and what trade-offs it involves.

The Power of Patterns: When you say 'let's use a Repository here,' everyone who knows patterns immediately understands the design. Patterns create a shared vocabulary that speeds up communication and helps teams make better decisions faster. This book catalogs 40+ patterns specifically for enterprise applications.

Chapter 1: Layering — The Foundation

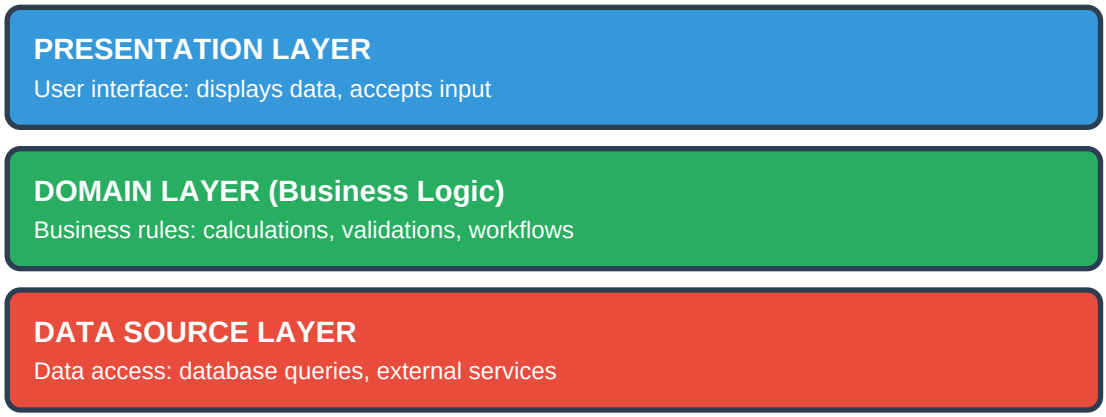
Why Layering Is the Most Important Pattern

If you learn only one concept from this entire book, make it layering. Layering is the most fundamental technique for managing complexity in enterprise applications. Almost every successful enterprise system uses some form of layered architecture.

> **The Restaurant Analogy:** A restaurant has a dining room where customers order and eat (presentation), a kitchen where chefs prepare food according to recipes (domain/business logic), and a pantry with suppliers providing ingredients (data source). Customers don't walk into the kitchen; chefs don't serve tables. Each area has clear responsibilities and communicates through defined interfaces (order tickets, plated dishes). This separation makes the restaurant efficient and allows specialists to focus on what they do best.

The Core Idea: Separation of Concerns

Layering divides your application into horizontal slices. Each layer has a specific responsibility and only knows about the layer directly below it. This one-way dependency is crucial.



Benefits of Layering

Benefit	What It Means	Why It Matters
Understandability	Each layer has one focus	New devs learn one layer at a time
Substitutability	Replace one layer without affecting others	Switch web to mobile UI, MySQL to PostgreSQL
Testability	Test each layer in isolation	Unit test business logic without database
Reusability	Lower layers serve multiple upper layers	Same logic for web, mobile, and API
Maintainability	Changes are localized	Fix a bug without risking other areas

The Three Principal Layers in Detail

1. Presentation Layer

The presentation layer handles all user interaction. It displays information, accepts input, and validates that input is well-formed (not business-valid, just well-formed—like 'is this a valid email format?'). This layer includes web pages,

REST APIs, mobile screens, and CLI interfaces.

- Display data in user-friendly formats (HTML, JSON, etc.)
- Accept and parse user input (form data, API requests)
- Validate input format (required fields, data types, format)
- Translate between external formats and internal objects
- Manage UI state (current page, selected items, form state)

Critical Rule: The presentation layer must NOT contain business logic. If you find yourself calculating discounts in JavaScript or checking inventory in your controller, that logic belongs in the domain layer. Violation leads to duplicated logic and testing nightmares.

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—the business rules. This is where you implement pricing calculations, validate business constraints, execute workflows, and enforce policies.

E-Commerce Domain Logic Examples:
- Calculate order total (sum items, apply discounts, add tax, add shipping)
- Validate order (items in stock? shipping address valid? payment method active?)
- Apply promotional rules (buy 2 get 1 free, 20% off orders over $100)
- Check customer credit limits and payment terms
- Manage inventory reservations during checkout
- Determine shipping options based on location and items

3. Data Source Layer

The data source layer handles communication with external data sources—primarily databases, but also file systems, external APIs, message queues, and legacy systems. It knows HOW to store and retrieve data, but NOT what that data means or what to do with it.

- Execute database operations (SQL queries, stored procedures)
- Map between database records and application objects
- Manage database connections and connection pooling
- Communicate with external systems (APIs, file systems)
- Handle caching to improve performance

Layer Communication: The Golden Rule

Dependencies must point downward only. The presentation layer knows about the domain layer. The domain layer knows about the data source layer. But never the reverse.

Why This Matters: If your data source layer calls methods in your domain layer, or your domain layer creates HTML, you've broken layering. Changes in lower layers will ripple upward, testing becomes impossible, and reusability is lost. The golden rule ensures that upper layers can change without affecting lower layers.

Request Flow Example — View Order Details:

1. **Presentation:** User clicks "View Order #1234" → Controller receives request
2. **Presentation** → **Domain:** Controller calls orderService.getOrderDetails(1234)
3. **Domain** → **Data:** OrderService calls orderRepository.findById(1234)
4. **Data** → **Database:** Repository executes SELECT query, returns raw data
5. **Data** → **Domain:** Repository returns Order object to service
6. **Domain:** Service may enrich order (calculate totals, check status)
7. **Domain** → **Presentation:** Service returns Order to controller
8. **Presentation:** Controller passes Order to view template, renders HTML

Common Layering Mistakes to Avoid

■ **Business logic in the presentation layer:** Calculating totals in JavaScript, validating business rules in controllers. This duplicates logic and makes testing hard.

■ **Business logic in stored procedures:** Complex rules in the database layer make testing difficult, tie you to one database vendor, and scatter business logic across tiers.

■ **Presentation knowledge in the domain:** If your Order class generates HTML or knows about HTTP sessions, you can't reuse it for a mobile app or API.

■ **Skipping layers:** Controllers calling the database directly bypass the domain layer, scattering business logic and creating tight coupling.

■ **Circular dependencies:** If the data layer calls the domain layer, you've created a maintenance nightmare. Information flows down as requests, up as responses.

Self-Check Questions:
Can I test my business logic without starting a web server?
Can I test my business logic without a database?
If I switch databases, does only the data source layer change?
If I add a mobile app, can I reuse my domain layer?

If any answer is 'no,' your layering needs improvement.

Chapter 2: Organizing Domain Logic

The Central Architectural Decision

How you organize your domain layer is one of the most important decisions you'll make. It affects how you write code, how easy it is to test, how well it handles complexity, and how it evolves over time. There are three main approaches.

Approach	Complexity Handling	Learning Curve	Best For
Transaction Script	Low	Easy	Simple CRUD, straightforward logic
Table Module	Medium	Moderate	Record-set based, moderate complexity
Domain Model	High	Steep	Complex rules, rich behavior

Approach 1: Transaction Script

Transaction Script organizes business logic into procedures where each procedure handles one complete business request from start to finish. It's like a recipe—step-by-step instructions that accomplish a task.

```java
// Transaction Script Example
public class OrderService {
    public void processOrder(long orderId) {
        // Step 1: Load data
        ResultSet order = loadOrder(orderId);
        ResultSet items = loadOrderItems(orderId);

        // Step 2: Calculate totals
        double subtotal = 0;
        while (items.next()) {
            subtotal += items.getDouble("price") * items.getInt("qty");
        }

        // Step 3: Apply discount
        double discount = (subtotal > 100) ? subtotal * 0.1 : 0;

        // Step 4: Save and complete
        double total = subtotal - discount;
        updateOrderTotal(orderId, total);
        processPayment(orderId, total);
        sendConfirmation(orderId);
    }
}
```

When Transaction Script Works Well:
- Simple CRUD operations with straightforward logic
- Operations are independent (don't share much logic)
- Rapid development is the priority
- Team is less experienced with OO design

When It Becomes Problematic:
- Scripts grow long and tangled
- You copy-paste code between scripts
- Multiple scripts need the same calculations
- Testing requires database setup

Approach 2: Domain Model

Domain Model creates an object-oriented model of your business. Instead of procedures operating on data, you have objects that represent business concepts and know how to behave. An Order knows how to calculate its total; a Customer knows its discount tier.

```java
// Domain Model Example
public class Order {
    private List<OrderItem> items;
    private Customer customer;

    public Money calculateTotal() {
        Money subtotal = Money.ZERO;
        for (OrderItem item : items) {
            subtotal = subtotal.add(item.getLineTotal());
        }
        return subtotal;
    }

    public Money calculateDiscount() {
        return customer.calculateDiscountFor(calculateTotal());
    }

    public Money getFinalAmount() {
        Money total = calculateTotal();
        return total.subtract(calculateDiscount()).add(calculateTax());
    }
}

public class Customer {
    private String tier;  // GOLD, SILVER, BRONZE

    public Money calculateDiscountFor(Money amount) {
        switch (tier) {
            case "GOLD": return amount.multiply(0.15);
            case "SILVER": return amount.multiply(0.10);
            default: return Money.ZERO;
        }
    }
}
```

Key Insight: Notice how discount calculation lives in Customer, not Order. This is Domain Model's power: behavior lives where the data is. The Order doesn't need to know about tiers; it just asks the Customer for its discount. If discount rules change, you modify one place.

When Domain Model Excels:
• Complex business rules with many conditions
• Logic is shared across many use cases
• Rules change frequently
• You want comprehensive unit tests
• System will evolve over years

Challenges:
• Steeper learning curve
• More upfront design effort
• Object-to-database mapping is complex
• Can be overkill for simple apps

Approach 3: Table Module

Table Module is a middle ground. One class per database table handles all business logic for all rows in that table. It operates on record sets rather than individual objects.

C#

```
// Table Module Example (C#)
public class OrderModule {
    private DataTable orders;

    public decimal CalculateTotal(int orderId) {
        DataRow order = orders.Select($"Id = {orderId}")[0];
        DataTable items = GetItemsFor(orderId);

        decimal total = 0;
        foreach (DataRow item in items.Rows) {
            total += (decimal)item["Price"] * (int)item["Qty"];
        }
        return total;
    }

    public decimal CalculateDiscount(int orderId) {
        decimal total = CalculateTotal(orderId);
        int customerId = GetCustomerId(orderId);
        string tier = new CustomerModule(ds).GetTier(customerId);

        return tier == "GOLD" ? total * 0.15m :
               tier == "SILVER" ? total * 0.10m : 0;
    }
}
```

When to Use Table Module: Your platform has strong record set support (like .NET DataSet), business logic is moderately complex, and database structure closely matches business concepts. It's more organized than Transaction Script but easier than Domain Model.

Service Layer: The Application Boundary

Regardless of which approach you choose, you often want a Service Layer on top. It defines your application's boundary—the operations available to the outside world.

```java
// Service Layer coordinates domain operations
public class OrderService {
    @Transactional
    public OrderDTO processOrder(CreateOrderRequest request) {
        // Coordinate domain operations
        Order order = orderRepo.findById(request.getOrderId());
        order.validate();
        Money amount = order.getFinalAmount();

        // Coordinate external services
        paymentGateway.charge(order.getCustomer(), amount);
        order.markAsProcessed();
        orderRepo.save(order);
        emailService.sendConfirmation(order);

        return new OrderDTO(order);  // Return DTO, not domain object
    }
}
```

Service Layer Responsibilities:
• Define available operations (API surface)
• Coordinate multiple domain objects
• Handle transactions (start, commit, rollback)
• Implement security checks
• Translate between DTOs and domain objects

NOT Service Layer's Job: Contain business logic—that belongs in the domain layer.

Decision Guide: Choosing Your Approach

Question	If Yes →	If No →
Is logic simple and straightforward?	Transaction Script	Continue below
Is logic tied closely to table structure?	Table Module	Continue below
Is logic complex with many interacting rules?	Domain Model	Re-evaluate
Will the system grow and evolve for years?	Domain Model	Simpler may work

Practical Advice: Start simple. Most apps can begin with Transaction Script. Watch for 'script creep'—when scripts grow complex and duplicate code. That's your signal to consider refactoring toward Domain Model. Don't over-engineer from the start.

Chapter 3: Mapping to Relational Databases

The Object-Relational Impedance Mismatch

One of the biggest challenges in enterprise development is bridging objects and databases. They represent data differently, and translating between them is a major complexity source.

Aspect	Objects (Code)	Relational (Database)
Identity	Memory reference	Primary key (ID)
Relationships	Object references	Foreign keys
Inheritance	Class hierarchies	Not supported directly
Collections	In-memory lists	Separate tables + FK
Navigation	obj.related.field	JOIN operations

Four Patterns for Data Access

Pattern 1: Table Data Gateway

One object handles all SQL for one table. It's a dedicated clerk for that table—you ask it to find, insert, update, or delete, and it handles the SQL.

```java
// Table Data Gateway
public class PersonGateway {
    public ResultSet findAll() {
        return query("SELECT * FROM person");
    }
    public ResultSet findById(long id) {
        return query("SELECT * FROM person WHERE id = ?", id);
    }
    public void insert(String name, String email) {
        execute("INSERT INTO person (name, email) VALUES (?,?)",
                name, email);
    }
    public void update(long id, String name, String email) {
        execute("UPDATE person SET name=?, email=? WHERE id=?",
                name, email, id);
    }
}
// Usage with Transaction Script:
ResultSet rs = personGateway.findById(42);
// Process the ResultSet in your script
```

Pattern 2: Row Data Gateway

One object per row. Load a record and get an object with that row's data. The object knows how to save itself back to the database.

java

```java
// Row Data Gateway
public class PersonRow {
    private long id;
    private String name;
    private String email;

    public static PersonRow findById(long id) {
        ResultSet rs = query("SELECT * FROM person WHERE id=?", id);
        PersonRow row = new PersonRow();
        row.id = rs.getLong("id");
        row.name = rs.getString("name");
        row.email = rs.getString("email");
        return row;
    }

    public void update() {
        execute("UPDATE person SET name=?, email=? WHERE id=?",
                name, email, id);
    }

    // Getters and setters...
}
```

Pattern 3: Active Record

Like Row Data Gateway, but includes domain logic too. The same class handles database operations AND business behavior. This is the Ruby on Rails pattern.

```java
// Active Record - Data + Database + Business Logic
public class Person extends ActiveRecord {
    private String firstName;
    private String lastName;
    private String tier;

    // Domain logic
    public String getFullName() {
        return firstName + " " + lastName;
    }

    public double getDiscountRate() {
        return "GOLD".equals(tier) ? 0.15 :
               "SILVER".equals(tier) ? 0.10 : 0.0;
    }

    // Database operations (often inherited)
    public void save() { /* INSERT or UPDATE */ }
    public static Person findById(long id) { /* SELECT */ }
}
```

Active Record Trade-off: Simple and productive for CRUD apps, but creates tight coupling between domain and database. When logic gets complex, this mixing becomes unwieldy.

Pattern 4: Data Mapper

A separate mapper layer moves data between domain objects and the database. Domain objects are pure business logic—they know nothing about the database.

```java
// Domain object - Pure business logic, no database code
public class Person {
    private Long id;
    private String firstName;
    private String lastName;
    private Money income;

    public String getFullName() { return firstName + " " + lastName; }
    public Money calculateTax() { return income.multiply(getTaxRate()); }
}

// Mapper - Handles all database translation
public class PersonMapper {
    public Person findById(Long id) {
        ResultSet rs = query("SELECT * FROM people WHERE id=?", id);
        return mapRow(rs);
    }

    public void insert(Person person) {
        execute("INSERT INTO people (first_name, last_name, income) " +
                "VALUES (?, ?, ?)",
                person.getFirstName(), person.getLastName(),
                person.getIncome().getAmount());
    }

    private Person mapRow(ResultSet rs) {
        Person p = new Person();
        p.setFirstName(rs.getString("first_name"));
        p.setLastName(rs.getString("last_name"));
        p.setIncome(new Money(rs.getBigDecimal("income")));
        return p;
    }
}
```

Why Data Mapper Is Worth It:
- **Testability:** Test domain logic without database
- **Flexibility:** Change schema without touching domain
- **Clean Domain:** Objects model business, not database
- **Complex Mappings:** Handle non-trivial object-table relationships

Data Mapper is essential for rich Domain Models. ORMs like Hibernate implement this pattern.

Choosing Your Data Access Pattern

Domain Logic	Data Pattern	Why
Transaction Script	Table Data Gateway	Simple, direct access
Transaction Script	Row Data Gateway	More OO feel
Simple Domain Model	Active Record	Convenient 1:1 mapping
Rich Domain Model	Data Mapper	Complete separation

Behavioral Patterns: Unit of Work, Identity Map, Lazy Load

Unit of Work

Tracks all objects you've loaded and modified during a business transaction. When you commit, it writes all changes in one coordinated batch. This ensures consistency and batches database operations for efficiency.

```java
// Unit of Work tracks changes
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) { dirtyObjects.add(obj); }
    public void registerDeleted(Object obj) { deletedObjects.add(obj); }

    public void commit() {
        // In a transaction:
        insertNew();    // INSERT new objects
        updateDirty();  // UPDATE modified objects
        deleteRemoved();// DELETE removed objects
    }
}
```

Identity Map

Ensures each database row is represented by exactly one object in memory. Load Customer #42 twice? You get the same object both times. This prevents inconsistencies and confusion.

```java
// Identity Map prevents duplicate objects
public class IdentityMap<T> {
    private Map<Long, T> cache = new HashMap<>();

    public T get(Long id) { return cache.get(id); }
    public void put(Long id, T obj) { cache.put(id, obj); }
}

// In mapper:
public Person findById(Long id) {
    if (identityMap.contains(id)) {
        return identityMap.get(id);  // Return cached object
    }
    Person p = loadFromDatabase(id);
    identityMap.put(id, p);  // Cache for next time
    return p;
}
```

Lazy Load

Don't load related objects until you actually access them. An Order might reference a Customer, who has Addresses... Loading everything would fetch your entire database! Lazy Load delays loading until the data is needed.

```java
// Lazy Load - load on demand
public class Order {
    private Long customerId;
    private Customer customer;  // Not loaded initially

    public Customer getCustomer() {
        if (customer == null) {
            customer = CustomerMapper.findById(customerId);
        }
        return customer;
    }
}
```

N+1 Problem Warning: Lazy loading can backfire. Loading an order then iterating through 100 items, each triggering a query, gives you 101 database calls! Monitor queries and use eager loading when you know you'll need related data.

Mapping Inheritance to Tables

Objects have inheritance; databases don't. If Employee has subclasses Manager and Engineer, how do you store them? Three strategies:

Strategy	How It Works	Pros	Cons
Single Table Inheritance	One table, all columns, type discriminator	Simple queries, no joins	Wasted space, many NULLs
Class Table Inheritance	One table per class, linked by ID	Normalized, no waste	Joins required for every query
Concrete Table Inheritance	One table per concrete class	No joins for single type	Duplicated columns, hard polymorphic queries

Rule of Thumb: Start with Single Table Inheritance for simplicity. Switch to Class Table if you have many subclass-specific fields causing too many NULLs.

Chapter 4: Web Presentation

Model-View-Controller (MVC)

MVC is the foundation of web presentation. It separates the UI into three roles: Model (domain data/logic), View (displays data), and Controller (handles input, coordinates).

Restaurant Analogy: Model = Kitchen (prepares food). View = Dining Room (presents to customers). Controller = Waiter (takes orders, coordinates between kitchen and dining room).

Controller Patterns

Pattern	Structure	When to Use
Page Controller	One controller per page	Most web apps (simple, intuitive)
Front Controller	Single entry point for all	Central security, logging, routing
Application Controller	Manages page flow/navigation	Complex wizards, multi-step flows

```java
// Page Controller (Spring MVC)
@Controller
public class OrderController {
    @GetMapping("/orders/{id}")
    public String viewOrder(@PathVariable Long id, Model model) {
        Order order = orderService.findById(id);
        model.addAttribute("order", order);
        return "order/view";
    }

    @PostMapping("/orders")
    public String createOrder(@ModelAttribute OrderForm form) {
        Order order = orderService.create(form);
        return "redirect:/orders/" + order.getId();
    }
}
```

View Patterns

Template View (most common): HTML templates with markers for dynamic content. JSP, Thymeleaf, Razor, Jinja all use this. The template looks like the output.

```html
<!-- Template View (Thymeleaf) -->
<h1>Order #<span th:text="${order.id}">123</span></h1>
<table>
    <tr th:each="item : ${order.items}">
        <td th:text="${item.productName}">Product</td>
        <td th:text="${item.quantity}">1</td>
        <td th:text="${item.price}">$0.00</td>
    </tr>
</table>
<p>Total: <span th:text="${order.total}">$0.00</span></p>
```

Best Practice: Keep views dumb. No business logic in views—no calculations, no business rule conditionals. If a view needs to show a discount, the controller calculates it.

Chapter 5: Concurrency

Why Concurrency Matters

Enterprise apps serve many users simultaneously. When two users modify the same data, things can go wrong. Concurrency bugs are among the hardest to find and fix.

Lost Update Example: Account has $1000. Alice reads balance, withdraws $300 (thinks result is $700). Bob reads balance, withdraws $500 (thinks result is $500). Both save. Final balance could be $700 OR $500—one withdrawal is lost!

Two Solutions: Optimistic vs Pessimistic

Approach	How It Works	Best When
Optimistic	Let all work proceed; check for conflicts at save	Conflicts are rare; want max throughput
Pessimistic	Lock data before editing; others wait or are blocked	Conflicts likely; conflict resolution costly

java

```
// Optimistic Locking with Version
public void update(Customer c) {
    int rows = execute(
        "UPDATE customers SET name=?, version=version+1 " +
        "WHERE id=? AND version=?",
        c.getName(), c.getId(), c.getVersion());

    if (rows == 0) {
        throw new OptimisticLockException(
            "Data was modified. Please reload.");
    }
    c.setVersion(c.getVersion() + 1);
}
```

Choose Optimistic for most web apps (conflicts rare, users can retry). **Choose Pessimistic** when conflicts are common or users would lose significant work.

Chapter 6: Session State

The Stateless Web Problem

HTTP is stateless—each request is independent. But users need continuity: shopping carts, login status, multi-step forms. Where you store session state affects scalability.

Storage	Where	Pros	Cons
Client	Cookies, URL, hidden fields	Infinite scale, no server memory	Size limits, security concerns
Server	Server memory	Fast, simple, any data type	Memory usage, cluster issues
Database	DB table	Survives restarts, cluster-friendly	Slower, DB load

Modern Trend: Minimize server-side state. Use tokens (JWT) for auth, store carts in DB, pass necessary state per request. This simplifies horizontal scaling.

Chapter 7: Distribution Strategies

Fowler's First Law: "Don't distribute your objects." Remote calls are ~1000x slower than local calls. Distribution adds complexity, failure modes, and debugging difficulty. Only distribute when you absolutely must.

When You Must Distribute

Sometimes distribution is unavoidable. When required, minimize pain with these patterns:

Remote Facade: Instead of many fine-grained remote methods, create coarse-grained methods that do more per call. One getCustomerDetails() instead of 10 getX() calls.

Data Transfer Object (DTO): Pack all needed data into a simple object for transfer. DTOs are optimized for network efficiency, not domain modeling.

java

```
// DTO - optimized for transfer
public class CustomerDetailsDTO {
    public Long id;
    public String name;
    public List<OrderSummaryDTO> recentOrders;
    public AddressDTO address;
    // No behavior, just data
}

// Remote Facade - coarse-grained interface
@Remote
public class CustomerFacade {
    public CustomerDetailsDTO getDetails(Long id) {
        // One call gets everything for a screen
        Customer c = repo.findById(id);
        List<Order> orders = orderRepo.findRecent(id);
        return buildDTO(c, orders);
    }
}
```

Chapter 8: Putting It All Together

Architecture Decision Framework

Decision	If Simple →	If Complex →
Domain Logic	Transaction Script	Domain Model
Data Access	Active Record	Data Mapper
Web Layer	Page Controller	Front Controller
Concurrency	Optimistic Lock	Pessimistic Lock
Session	Client/Stateless	Server/Database

Example Architectures

Simple CRUD App (Employee contact management):
• Transaction Script + Table Data Gateway
• Page Controller + Template View
• Optimistic locking + Server session
Why: Fastest to develop, easy to understand, appropriate complexity

Complex E-Commerce Platform:
• Domain Model + Service Layer + Data Mapper
• Page Controller + REST API for mobile
• Optimistic (most), Pessimistic (inventory)
• Stateless with JWT + Database session for carts
Why: Complex rules need Domain Model; multiple clients share Service Layer

Final Advice:
1. Start simple—you can always add complexity
2. Match architecture complexity to problem complexity
3. Invest in the domain layer—that's where value lives
4. Always layer, even for simple apps
5. Learn patterns incrementally—don't use everything at once

Part 2: Complete Pattern Catalog

This section provides a quick reference to all patterns in the book. Each pattern is summarized with its purpose and when to use it. Refer to the full book for detailed examples.

Domain Logic Patterns (Chapter 9)

Pattern	Purpose	When to Use
Transaction Script	Procedure per business transaction	Simple, straightforward logic
Domain Model	OO model with behavior	Complex, rich business rules
Table Module	One class per table, operates on record sets	Moderate complexity, record set tooling
Service Layer	Application boundary, coordinates domain	Always—defines your API

Data Source Patterns (Chapter 10)

Pattern	Purpose	When to Use
Table Data Gateway	One object handles all SQL for one table	Transaction Script
Row Data Gateway	One object per row, knows how to save itself	More OO Transaction Script
Active Record	Row Gateway + domain logic	Simple Domain Model
Data Mapper	Separate mapper layer, domain knows nothing of DB	Rich Domain Model

Object-Relational Behavioral Patterns (Chapter 11)

Pattern	Purpose	When to Use
Unit of Work	Tracks changes, coordinates saves	Always with Data Mapper
Identity Map	One object per row, prevents duplicates	Always with ORM
Lazy Load	Load related data on demand	Avoid loading everything

Object-Relational Structural Patterns (Chapter 12)

Pattern	Purpose
Identity Field	Store DB primary key in object
Foreign Key Mapping	Map object reference to foreign key
Association Table Mapping	Many-to-many via join table
Dependent Mapping	Child saved by parent's mapper
Embedded Value	Value object as columns in owner's table
Serialized LOB	Complex object as serialized blob
Single Table Inheritance	One table, all fields, type column
Class Table Inheritance	One table per class, joined by ID
Concrete Table Inheritance	One table per concrete class

O-R Metadata Mapping Patterns (Chapter 13)

Pattern	Purpose
Metadata Mapping	Drive mapping from metadata (XML, annotations)
Query Object	Represent query as object, not string
Repository	Collection-like interface for domain objects

Web Presentation Patterns (Chapter 14)

Pattern	Purpose
Model View Controller	Separate data, display, and input handling
Page Controller	One controller per page/action
Front Controller	Single entry point for all requests
Template View	HTML template with embedded markers
Transform View	Transform data to HTML programmatically
Two Step View	Logical screen → HTML in two steps
Application Controller	Manage complex page flows

Distribution Patterns (Chapter 15)

Pattern	Purpose
Remote Facade	Coarse-grained interface for remote access
Data Transfer Object	Simple object carrying data between processes

Offline Concurrency Patterns (Chapter 16)

Pattern	Purpose
Optimistic Offline Lock	Detect conflicts at commit time via version
Pessimistic Offline Lock	Lock data before editing, prevent conflicts
Coarse-Grained Lock	Lock set of related objects together
Implicit Lock	Framework acquires locks automatically

Session State Patterns (Chapter 17)

Pattern	Purpose
Client Session State	Store session on client (cookies, URL)
Server Session State	Store session in server memory
Database Session State	Store session in database

Base Patterns (Chapter 18)

Pattern	Purpose
Gateway	Encapsulate access to external system
Mapper	Set up communication between independent objects
Layer Supertype	Common supertype for all types in a layer
Separated Interface	Interface in separate package from implementation
Registry	Well-known object for finding services
Value Object	Small object, equality by value not identity
Money	Represents monetary value properly
Special Case	Subclass for particular cases (Null Object)
Plugin	Link classes at configuration, not compile time
Service Stub	Replace problematic services for testing
Record Set	In-memory tabular data representation

Conclusion: Key Takeaways

Congratulations! You now have a comprehensive understanding of enterprise application architecture patterns. Here are the most important lessons to remember:

✓ **Layer Your Application:** Separate presentation, domain, and data source. This is the foundation of maintainability.

✓ **Choose Domain Logic Wisely:** Match approach to complexity. Start simple, refactor as needed.

✓ **Bridge Objects and Databases Carefully:** Use appropriate patterns for your domain complexity.

✓ **Handle Concurrency Explicitly:** Don't ignore it—choose optimistic or pessimistic based on your needs.

✓ **Don't Distribute Unless Necessary:** Remote calls are expensive. Keep layers together when possible.

✓ **Patterns Are Tools, Not Rules:** Use them to solve specific problems you actually have.

✓ **Start Simple, Evolve:** Begin with the simplest architecture that works. Add patterns as needed.

Your Next Steps:

1. Read the original book for detailed code examples
2. Examine frameworks you use—identify patterns they implement
3. Review current projects—where could patterns improve them?
4. Practice explaining patterns to colleagues
5. Start small—apply one pattern at a time

Remember: Great architecture isn't about using the most patterns—it's about using the right patterns for your specific situation.