Table of Contents

Domain-Driven Design: Comprehensive Guide

Last Updated: 2025-01-24
Target Length: 50,000-70,000 words (~140-180 pages)
Target Audience: Software architects and developers implementing Domain-Driven Design

---

Table of Contents

Part I: Foundations

1. Introduction & DDD Philosophy

1.1 What Problem Does DDD Solve?

Domain-Driven Design addresses the fundamental challenge of managing complexity in software systems by centering development on a rich domain model that reflects deep understanding of the business domain.

The Core Problems:

Traditional software development approaches often create a dangerous gap between business experts and technical implementation. This gap manifests in several critical ways:

Translation Errors: Business concepts get lost or distorted in translation to code. A "Policy" in insurance means something specific to domain experts, but developers might implement it as a generic "Contract" class, losing essential business meaning.

Model Fragmentation: Different parts of the system use inconsistent models for the same concepts. The Sales team's "Order" means something different from Fulfillment's "Order", yet the codebase treats them as the same thing, leading to confused logic and brittle integrations.

Maintenance Burden: When business rules change (and they always do), developers must engage in extensive "code archeology" to find where rules are implemented, understand their current logic, and modify them without breaking other parts of the system.

Communication Breakdown: Developers and domain experts speak fundamentally different languages. Business meetings use domain terminology while code reviews use technical jargon. This linguistic divide prevents effective collaboration and hides misunderstandings until they become expensive production bugs.

The DDD Solution:

DDD solves these problems through a systematic approach built on three pillars:

1. **Model-Driven Design**: The domain model becomes the heart of the software. Code structure directly reflects business concepts, not technical abstractions. If experts talk about "Policies", "Claims", and "Coverage", those exact terms appear as primary classes in the codebase.
2. **Ubiquitous Language**: A common, rigorous language shared by all stakeholders - developers, domain experts, product managers, and even documentation. This language evolves as understanding deepens, and changes to the language drive changes to both the model and the code.
3. **Strategic and Tactical Patterns**: A catalog of proven patterns for managing complexity at multiple levels - from organizing large systems into Bounded Contexts (strategic) to implementing rich domain models (tactical).

1.2 Why Does DDD Exist?

DDD emerged from Eric Evans' experiences in the early 2000s working on complex enterprise systems. Through projects across insurance, finance, shipping, and other domains, he observed recurring patterns in successful projects and consistent pitfalls in unsuccessful ones.

Key Observations:

1. Complexity is Inevitable

Complex business domains require complex models. Attempts to simplify lose essential knowledge. A shipping logistics system that treats all packages the same will fail when hazardous materials, temperature-sensitive cargo, or customs requirements enter the picture. The complexity exists in the domain - the question is whether we model it explicitly or let it emerge as bugs.

2. Model Quality Directly Impacts Software Quality

Projects with poor domain models become increasingly difficult to maintain. Each new feature requires understanding existing spaghetti code, making changes feels like walking through a minefield, and regression bugs proliferate. Conversely, projects with rich, well-understood domain models remain malleable even as they grow in size and complexity.

3. Knowledge Crunching is Essential

The knowledge needed to build effective software exists primarily in domain experts' heads, not in requirements documents. Effective design requires continuous collaboration between developers and experts, with both sides learning and refining the model over time. This "knowledge crunching" process cannot be rushed or automated - it is the heart of DDD.

4. Patterns Accelerate Learning

While each domain is unique, certain structural patterns recur across domains. Entities with identity, Value Objects defined by attributes, Aggregates as consistency boundaries, Repositories for persistence abstraction - these patterns appear again and again. Recognizing and applying them accelerates development and improves communication.

Historical Context:

In the early 2000s, software development was dominated by two extremes:
- **Anemic domain models**: Objects that were pure data structures with all logic in service layers
- **Smart UI anti-pattern**: Business logic embedded in user interface code

Both approaches failed for complex domains. DDD provided a third way: rich domain models that captured business complexity explicitly, supported by strategic patterns for organizing large systems.

1.3 Core DDD Principles

Principle 1: Ubiquitous Language

Definition

A common, rigorous language built up between developers and domain experts, based on the domain model used in the software.

Why It Matters

Software cannot cope with ambiguity - it requires precision. Translation between "business speak" and "tech speak" introduces errors with each conversion.

A shared language ensures everyone discusses the same concepts with the same meaning. The language evolves as understanding deepens, creating a feedback loop that strengthens both the model and the team's shared understanding.

Implementation in Code

Domain terminology must be embedded directly in code:

```
// GOOD: Uses ubiquitous language
public class Order {
    private OrderId orderId;
    private Customer customer;
    private ShippingAddress shippingAddress;
    private List<OrderLine> orderLines;

    public void submit() {
        if (!hasMinimumOrderValue()) {
            throw new BelowMinimumOrderException();
        }
        this.status = OrderStatus.SUBMITTED;
        // ... submission logic
    }
}

// BAD: Uses technical abstractions
public class Transaction {
    private String id;
    private Data transactionData;
    private Money amount;

    public void update(Map<String, Object> params) {
        // ... generic update logic
    }
}
```

The good example uses terms domain experts recognize: Order, Customer, ShippingAddress, OrderLine, submit, hasMinimumOrderValue. The bad example uses generic terms that could mean anything: Transaction, Data, amount, update.

Building the Language

The language emerges through systematic collaboration:

1. Developers and experts meet regularly
2. Explore domain through concrete scenarios
3. Surface terminology and test it with examples
4. Challenge vague or ambiguous terms
5. Refine definitions collaboratively
6. Update code to match evolved language
7. Repeat as understanding deepens

Example Conversation:

```
Expert: "When an order is abandoned, we need to send a reminder
email."
Developer: "What exactly is 'abandoned'?"
Expert: "It means the customer added items to their cart but
didn't complete checkout within 24 hours."
Developer: "Is that different from 'cancelled'?"
Expert: "Yes! Cancelled means the customer explicitly cancelled.
Abandoned means they never finished."
Developer: "So we have two distinct concepts: Abandoned and
Cancelled?"
Expert: "Exactly. And they have different recovery strategies."

→ New terms discovered: Abandoned, Cancelled, Recovery Strategy,
Checkout
→ Code will have distinct AbandonedCart and CancelledOrder classes
```

Principle 2: Model-Driven Design

Definition: The software model is tightly linked to the domain model. Code structure directly reflects domain concepts.

Rationale:
- Keeps code aligned with business reality
- Makes changes to business logic straightforward (change the model, code

follows)
- Allows domain experts to understand code structure conceptually
- Preserves design decisions in code itself, not separate documents

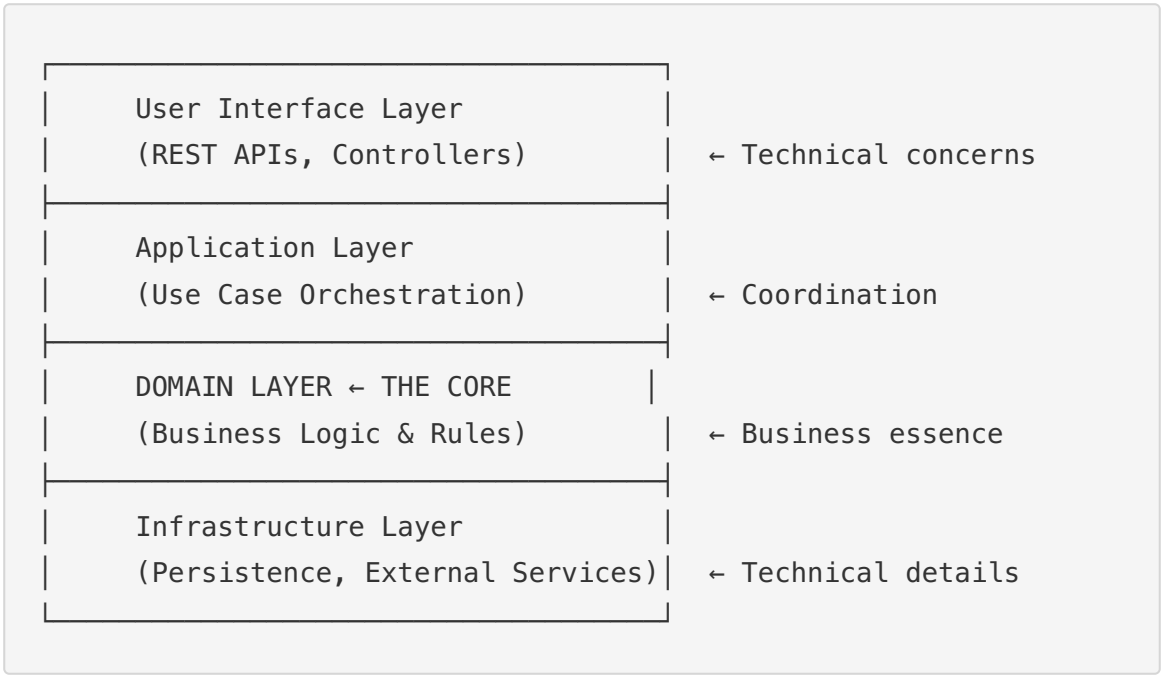
In Practice:

Models are not just diagrams - they are executable code:

```
// The Model IS the code
public class Order {
    // Value Objects expressing business concepts
    private OrderTotal total;
    private Customer customer;
    private List<OrderLine> orderLines;
    private InventoryReservation inventoryReservation;

    // Business method using ubiquitous language
    public void submit() {
        if (!hasMinimumOrderValue()) {
            throw new BelowMinimumOrderException();
        }

        if (customer.hasOutstandingBalance()) {
            throw new CustomerCreditHoldException();
        }

        inventoryReservation = reserveInventory(orderLines);
        this.status = OrderStatus.SUBMITTED;
        raiseEvent(new OrderSubmitted(this.id, this.total));
    }
}
```

The code reads like business logic because it IS the business logic. Methods like `submit()`, `hasMinimumOrderValue()`, and `hasOutstandingBalance()` directly express business rules.

Architecture Serves the Domain:



The domain layer is protected at the center. Infrastructure and UI concerns don't leak in. The model can be understood and tested without databases, web servers, or external services.

Principle 3: Continuous Learning and Refinement

Definition: Understanding of the domain deepens continuously through collaboration and reflection.

Rationale:
- Initial understanding is ALWAYS incomplete (this is not a failure, it's reality)
- Hidden complexity emerges during implementation
- Domain knowledge exists in experts' experience, not requirements documents
- Breakthrough insights reshape the model fundamentally

The Breakthrough Concept:

DDD embraces the idea of "breakthroughs" - moments when the team suddenly realizes a better way to model the domain. These breakthroughs often come after weeks of struggling with a clunky model.

Example Breakthrough:

Before: An e-commerce system modeled `Product` as a single class with complex conditional logic handling different product types (digital downloads, perishable goods, custom-made items, hazardous materials).

Breakthrough: Product itself is not the central concept - it's the `FulfillmentStrategy` that matters! Different products need different fulfillment approaches. The model shifted to:

```
// After breakthrough
public class Product {
    private ProductId id;
    private FulfillmentStrategy fulfillmentStrategy;

    public Shipment prepareForShipment(Order order) {
        return fulfillmentStrategy.createShipment(this, order);
    }
}

public interface FulfillmentStrategy {
    Shipment createShipment(Product product, Order order);
}

public class DigitalDownloadStrategy implements
FulfillmentStrategy {
    // Instant delivery via download link
}

public class StandardShippingStrategy implements
FulfillmentStrategy {
    // Physical shipping with carrier
}

public class MadeToOrderStrategy implements FulfillmentStrategy {
    // Production before shipment
}
```

This breakthrough simplified the model dramatically. The conditional complexity disappeared, replaced by polymorphism that matched the business reality.

Refactoring Towards Deeper Insight:

As understanding deepens, the model must evolve. This is not "rework" - it's essential:

1. **Initial Model**: First attempt based on initial understanding

2. **Implementation**: Building reveals gaps and awkwardness
3. **Conversation**: Discussing difficulties with domain experts
4. **Insight**: Sudden clarity about better model
5. **Refactoring**: Restructuring code to match new understanding
6. **Repeat**: Continuous cycle as domain knowledge deepens

1.4 The Strategic-Tactical Divide

DDD provides patterns at two distinct levels:

Strategic Patterns (Part I & II of this guide)

Purpose: Organize large systems and manage complexity at the system level

Key Patterns:
- **Domains & Subdomains**: Identifying major areas of business capability
- **Bounded Contexts**: Explicit boundaries where models apply
- **Context Mapping**: Relationships and integrations between contexts
- **Ubiquitous Language**: Shared terminology within context boundaries

Questions Answered:
- How do we break down a large system?
- Where are the natural boundaries?
- How do different parts relate?
- What should we invest most effort in?

Tactical Patterns (Part III of this guide)

Purpose: Implement rich domain models within a Bounded Context

Key Patterns:
- **Entities**: Objects with identity and lifecycle
- **Value Objects**: Objects defined by attributes
- **Aggregates**: Consistency boundaries
- **Repositories**: Persistence abstraction
- **Domain Services**: Operations not belonging to entities
- **Domain Events**: Facts about what happened
- **Application Services**: Use case orchestration

Questions Answered:
- How do we structure the domain model?
- Where does logic belong?
- How do we maintain invariants?
- How do we persist aggregates?

The Relationship:

```
Strategic Design (Context organization)
    ↓
Defines boundaries for
    ↓
Tactical Design (Rich models within contexts)
```

You typically start strategic (identify contexts) then go tactical (implement models within contexts). But it's not strictly sequential - insights from tactical modeling might reveal better strategic boundaries.

1.5 When to Use DDD

DDD is powerful but not appropriate for every project.

Use DDD When:

✓ **Complex Business Logic**
- Rules that change frequently
- Domain expertise required to understand logic
- Business decisions embedded in software
- Example: Insurance underwriting, loan approval, supply chain optimization

✓ **Long-Lived System**
- Project expected to last years
- Will evolve as business evolves
- Investment in model pays off over time
- Example: Core banking systems, ERP systems

✓ **Domain Experts Available**
- Business experts can participate in modeling
- Knowledge exists in people's heads
- Continuous collaboration possible
- Example: Most enterprise projects with engaged business stakeholders

✓ **Team Committed to DDD**
- Team willing to invest in learning
- Organization supports iterative design
- Refactoring is acceptable
- Example: Teams with architectural focus and quality emphasis

Skip DDD When:

✗ Simple CRUD Applications
- Mostly data entry and retrieval
- Little business logic
- Generic operations suffice
- Example: Simple content management, basic user registration

✗ Pure Technical Domains
- No business domain to model
- Purely algorithmic
- Experts are developers
- Example: Compilers, network protocols, image processing algorithms

✗ Short-Lived Projects
- Prototype or proof of concept
- Expected lifetime < 6 months
- Investment not worth return
- Example: Hackathon projects, one-time data migrations

✗ Insufficient Domain Expertise
- No experts available
- Requirements fully specified
- No exploration needed
- Example: Building to exact specifications with no business input

✗ Team Not Ready
- Team unfamiliar with OOP/design
- No mentoring available
- Organization wants quick code, no design
- Example: Startups in extreme time pressure with no design experience

1.6 The DDD Journey

Adopting DDD is a journey, not a destination:

Phase 1: Learning (3-6 months)
- Read foundational books (Evans, Vernon)
- Understand strategic and tactical patterns
- Practice with small models
- **Goal**: Team fluent in DDD vocabulary

Phase 2: First Context (6-12 months)
- Pick one Bounded Context to model deeply
- Apply tactical patterns
- Develop Ubiquitous Language
- Refactor towards deeper insight
- **Goal**: One well-modeled context in production

Phase 3: Strategic Design (12+ months)
- Map full system into contexts
- Define context boundaries
- Implement context mappings
- **Goal**: Clear system architecture with explicit boundaries

Phase 4: Maturity (ongoing)
- Continuous refinement
- New insights reshape models
- Strategic and tactical in balance
- **Goal**: DDD as natural way of working

Reality Check: Most teams take 1-2 years to become truly proficient with DDD. The investment pays off in maintainable, evolvable systems that stay aligned with business needs.

---

2. Strategic Design Patterns

2.1 Overview

Strategic Design addresses the challenge of organizing large, complex systems. While tactical patterns focus on implementation details within a Bounded Context, strategic patterns focus on context boundaries, team organization, integration patterns, and investment decisions.

> "Strategic design is essential for maintaining order in large systems and systems that are critical to the enterprise."
> — Eric Evans

Strategic Design Solves:
- System overwhelm: "This system is too big to understand"
- Model confusion: "Same word means different things"
- Integration chaos: "Everything depends on everything"
- Investment waste: "We're polishing the wrong things"

Strategic Patterns Catalog:
1. **System** (Root Object) - Overall system organization
2. **Domain & Subdomain** - Areas of business capability
3. **Bounded Context** - Explicit model boundaries
4. **Context Mapping** - Integration between contexts
5. **BFF Scope** - Client-specific aggregation

2.2 System

Having identified the strategic patterns DDD provides, we begin with the foundational concept: the System. Just as a building needs a foundation before walls, DDD modeling needs a root container before identifying domains and contexts.

When modeling a software system with Domain-Driven Design, we need a clear starting point—a way to establish the boundary and scope of what we're designing. The **System** serves as this root concept, representing the entire software system and providing a container for all strategic DDD elements.

Definition: The System is the top-level organizational concept representing the complete software system being modeled. It contains all domains, Bounded Contexts, their relationships, and integration patterns.

Purpose and Rationale

The System concept addresses fundamental organizational questions:

1. Single Source of Truth

Complex systems involve multiple domains, numerous Bounded Contexts, and various integration patterns. Without a clear root concept, it becomes ambiguous which elements belong together:

- Which Bounded Contexts are part of this system?
- How do domains relate to each other?
- What integration patterns connect these contexts?

The System establishes clear ownership: every domain, context, and mapping exists within one specific system.

2. System Boundaries

The System defines explicit boundaries:

```
system:
  id: sys_ecommerce
  name: E-Commerce System
  domains:
    - id: dom_sales
      name: Sales
    - id: dom_catalog
      name: Product Catalog
    - id: dom_fulfillment
      name: Order Fulfillment

  bounded_contexts:
    - id: bc_shopping_cart
      domain_ref: dom_sales
    - id: bc_order_management
      domain_ref: dom_sales
    - id: bc_inventory
      domain_ref: dom_fulfillment
```

This hierarchical organization makes it clear that Shopping Cart and Order Management belong to the Sales domain, while Inventory belongs to Fulfillment.

3. System Identity and Metadata

Systems need identity and descriptive information:

```
system:
  id: sys_healthcare_platform
  name: Patient Care Management Platform
  version: 1.0.0
  description: Comprehensive platform for managing patient
records, appointments, and billing
```

This metadata provides:
- **Unique identification** for the system
- **Human-readable name** for documentation and communication
- **Version tracking** for evolution over time
- **Description** that captures system purpose

4. Strategic Elements Container

The System organizes all strategic DDD patterns in one place:

Element	Purpose
Domains	Business areas the system addresses
Bounded Contexts	Explicit model boundaries within domains
Context Mappings	Integration relationships between contexts
BFF Scopes	Client-specific aggregation patterns
BFF Interfaces	Concrete integration implementations

5. Benefits of System-Centric Organization

Organizing around a System concept provides several advantages:

- **Clarity**: Immediate understanding of system scope and composition
- **Navigation**: All elements accessible from single starting point
- **Validation**: Completeness checks (all contexts reference valid domains)
- **Documentation**: Natural structure for system architecture documents
- **Tooling**: Consistent entry point for code generation and analysis

When to Use

Use the System concept when:
- Modeling a complete software system architecture
- Documenting strategic DDD patterns
- Creating comprehensive system documentation
- Establishing boundaries for multi-domain systems
- Coordinating multiple bounded contexts

Example: Job Seeker Platform

```yaml
system:
  id: sys_job_seeker
  name: Job Seeker Application
  description: Platform connecting job seekers with employers

  domains:
    - id: dom_talent
      name: Talent Management
      type: core
      strategic_importance: critical

    - id: dom_matching
      name: Job Matching
      type: core
      strategic_importance: critical

    - id: dom_communication
      name: Communications
      type: supporting
      strategic_importance: important

  bounded_contexts:
    - id: bc_candidate_profiles
      name: Candidate Profiles
      domain_ref: dom_talent

    - id: bc_job_search
      name: Job Search
      domain_ref: dom_matching

    - id: bc_messaging
      name: Messaging
      domain_ref: dom_communication

  context_mappings:
    - id: cm_search_to_profiles
      name: Search to Profiles Integration
      upstream_context: bc_candidate_profiles
      downstream_context: bc_job_search
      relationship_type: customer_supplier
```

System as Architectural Foundation

The System concept provides the architectural foundation for strategic DDD modeling. By establishing clear boundaries, organizing related concepts, and providing identity, it enables coherent design of complex software systems. Every strategic design discussion begins with understanding which system we're modeling and what lies within its boundaries.

2.3 Domains and Subdomains

Strategic DDD begins with understanding the business **domain**—the entire problem space the software addresses. For most organizations, this domain is far too large and complex for a single unified model. The solution is **subdomain classification**, which guides investment decisions and architectural choices.

Domain Definition

A **domain** is the sphere of knowledge and activity around which the system is organized. It encompasses all business activities, processes, and rules that the software must support.

Example domains:
- E-commerce company: Product catalog, shopping, order fulfillment, customer service, marketing, analytics, billing
- Healthcare provider: Patient care, appointments, medical records, billing, insurance claims, pharmacy
- Financial services: Account management, transactions, loans, investments, fraud detection, reporting

The domain is typically too large for effective modeling as a single unit. This is where subdomain classification becomes critical.

Subdomain Classification

Every domain decomposes into **subdomains** that fall into three categories, each demanding different investment strategies:

2.3.1 Core Domain

The core domain is where you must excel to succeed in business. This is your competitive advantage.

Characteristics:
- **Strategic importance**: If this fails, the business fails
- **Competitive differentiation**: What makes this business unique
- **Complex business logic**: Sophisticated rules and processes
- **High investment required**: Best developers, rigorous DDD, continuous iteration
- **Custom-built**: Never outsourced or bought off-the-shelf
- **Frequent evolution**: Business innovation happens here

Identification criteria:
- Generates revenue or saves significant costs
- Competitors cannot easily replicate
- Domain experts are most engaged here
- Frequent feature requests and refinements
- Executive-level attention to this area

Investment strategy:
- Assign the strongest developers
- Apply DDD tactical patterns rigorously
- Invest in deep domain modeling
- Iterate and refine continuously
- Protect aggressively from technical debt
- Extensive testing and quality measures

Real-world examples:
- **Netflix**: Recommendation algorithm, content personalization
- **Amazon**: Product search ranking, one-click ordering
- **Uber**: Real-time ride matching and dynamic pricing
- **Airbnb**: Trust and safety scoring, search quality
- **Google**: Search algorithm, ad auction system

2.3.2 Supporting Subdomain

Supporting subdomains are necessary for the core domain to function but don't provide competitive advantage. They're business-specific but not differentiating.

Characteristics:
- **Necessary but not strategic**: Required for operations
- **No differentiation**: Competitors have similar solutions
- **Moderate complexity**: Some custom logic needed

- **Moderate investment**: Adequate resources
- **Build vs. buy decision**: Could outsource but might customize
- **Stable**: Changes less frequently than core

Identification criteria:
- Necessary for core domain to function
- No competitive advantage if done exceptionally well
- Some business-specific requirements exist
- Moderate complexity, not trivial

Investment strategy:
- Adequate quality, not perfection
- Consider buying and customizing
- Simpler modeling than core domain
- "Good enough" is good enough
- May use mid-level developers

Examples:
- **E-commerce**: Order management, basic inventory tracking
- **SaaS Product**: User account management, billing and invoicing
- **Healthcare**: Appointment scheduling, basic patient registration
- **Logistics**: Route planning, driver scheduling

2.3.3 Generic Subdomain

Generic subdomains are necessary but completely generic across industries. No customization provides business value—standard solutions are best.

Characteristics:
- **Commodity functionality**: Same across all companies
- **No customization value**: Standard solution is optimal
- **Well-understood problems**: Solved problems with proven solutions
- **Minimal investment**: Lowest resource allocation
- **Buy don't build**: Strong candidate for off-the-shelf
- **Very stable**: Rarely changes

Identification criteria:
- Every company in any industry needs this
- No business-specific requirements
- Established off-the-shelf solutions exist
- No competitive advantage in customization

Investment strategy:
- Buy commercial off-the-shelf (COTS) solutions
- Use open-source libraries
- Minimize customization
- Assign minimal team attention
- Isolate from core domain to prevent contamination

Examples:
- Authentication and authorization (unless security IS your core business)
- Email sending and templates
- Payment processing (Stripe, PayPal, etc.)
- PDF generation and reporting
- Time zone handling
- Currency conversion
- Logging and monitoring (use standard tools)

2.3.4 Distillation: Finding the Core

Distillation is the process of identifying and isolating the core domain from supporting and generic subdomains.

Process steps:
1. **Map the entire domain**: List all major capabilities and areas
2. **Classify each area**: Core, Supporting, or Generic
3. **Validate with stakeholders**: Do business leaders agree on what's core?
4. **Refine boundaries**: Separate core concerns from non-core
5. **Protect the core**: Prevent generic concerns from contaminating core domain
6. **Document the vision**: Make core domain explicit and visible

Distillation yields:
- Clear investment priorities
- Architectural boundaries

- Team organization guidance
- Make vs. buy decisions
- Focus for DDD tactical patterns

Common Mistakes:
- **Everything is core**: Team believes all work is strategic (dilutes effort)
- **Nothing is core**: Team treats everything as commodity (misses differentiation)
- **Core contaminated**: Generic concerns mixed into core domain
- **Supporting treated as core**: Over-investing in non-differentiating areas

The subdomain classification directly influences where to apply DDD rigorously. Core domains demand full tactical DDD. Supporting subdomains get simpler models. Generic subdomains get isolated and often replaced with third-party solutions.

2.4 Bounded Contexts

If subdomains answer "what parts of the business exist?", then **Bounded Contexts** answer "where do our models apply?" A Bounded Context is the strategic pattern at the heart of DDD's approach to managing complexity.

Definition

A **Bounded Context** is an explicit boundary within which a particular domain model is defined and applicable. It defines the scope where a specific Ubiquitous Language and model are consistently used.

Core Insight: You cannot create one unified model for a large system. Different parts need different models. Bounded Contexts make these boundaries explicit.

Why Bounded Contexts Are Necessary

The problems they solve:

1. **Polysemic terms**: The same word means different things in different parts of the organization
   - "Customer" in Sales vs. "Customer" in Support
   - "Order" in Sales vs. "Order" in Fulfillment
   - "Product" in Catalog vs. "Product" in Inventory

2. **Competing perspectives**: Different stakeholders need different views
   - Sales sees products as items to sell (price, features, promotions)
   - Inventory sees products as stock to manage (SKU, location, quantity)
   - Shipping sees products as items to ship (weight, dimensions, fragility)
3. **Model confusion**: Without explicit boundaries, models become muddled
   - Concepts leak between contexts
   - Teams talk past each other using same words with different meanings
   - Code becomes confused trying to serve all perspectives
4. **Scale**: Large systems cannot have a single model satisfying all stakeholders

Characteristics of a Bounded Context

1. **Explicit Boundary**: The context has a clearly defined boundary (team, codebase, database schema, service boundary)
2. **Unified Model**: Within the boundary, the model is unified and consistent—no contradictions
3. **Ubiquitous Language**: Has its own language that may differ from other contexts
4. **Team Ownership**: Typically owned by a single team with clear responsibility
5. **Independence**: Can evolve somewhat independently of other contexts
6. **Clear Interfaces**: Defines explicitly how it interacts with other contexts

How to Identify Boundaries

Linguistic Boundaries:
- Listen for when the same word means different things
- Notice when definitions become complex: "Well, it depends on whether you mean..."
- Pay attention to phrases like "in this context" or "from our perspective"
- Experts from different areas define terms differently

Example linguistic boundary:

```
In Sales Context:
- "Order" = Customer purchase request with pricing and payment
- Focus: Products, prices, discounts, customer information

In Fulfillment Context:
- "Order" = Shipping instructions with items to pick and pack
- Focus: Warehouse location, package dimensions, shipping carrier

→ Same word, completely different models → Two bounded contexts
```

Organizational Boundaries:
- Department boundaries often indicate context boundaries
- Different business processes
- Separate budget authorities and decision-making
- Different compliance or regulatory requirements
- Different stakeholders with competing priorities

Technical Boundaries:
- Different databases or schemas
- Different deployment schedules or release cycles
- Different technology stacks
- Different scalability or performance requirements
- Legacy system boundaries

Process for Defining Bounded Contexts:

1. **Map the domain** with domain experts using techniques like Event Storming or Domain Storytelling
2. **Identify linguistic discontinuities**: Where do terms change meaning?
3. **Look for natural seams**: Where do business processes naturally separate?
4. **Consider team structure**: Can one team own this context?
5. **Examine existing system boundaries**: What boundaries already exist?
6. **Test proposed boundaries**: Walk through scenarios—does the boundary make sense?
7. **Validate with stakeholders**: Do they recognize these as natural divisions?

When to Create Separate Bounded Contexts

✓ Different teams have different understandings of a concept
✓ The same term means different things in different areas
✓ Different business processes require different models
✓ Scalability requires separating concerns
✓ Legacy systems need isolation from new development
✓ Different update frequencies or consistency requirements exist

Relationships with Other Patterns

- **Contains**: Aggregates, Entities, Value Objects, Repositories, Domain Services
- **Defined by**: Ubiquitous Language (each context has its own)
- **Part of**: Domain or Subdomain (contexts belong to subdomains)
- **Related through**: Context Mapping patterns
- **Protects**: Model integrity and consistency

Common Mistakes

1. **Too Large**: Context encompasses too much, model becomes confused and contradictory
2. **Too Small**: Over-fragmentation leads to excessive integration complexity
3. **Fuzzy Boundaries**: Unclear where one context ends and another begins
4. **Ignored Boundaries**: Contexts defined on paper but not enforced in code
5. **Premature Decomposition**: Creating many contexts before understanding the domain
6. **Technical Boundaries Only**: Ignoring linguistic and organizational factors

Example: E-Commerce Bounded Contexts

Sales Context:

```
bounded_context:
  id: bc_sales
  domain_ref: dom_ecommerce

Model within this context:
- Product: Item for sale (price, description, images, reviews,
availability)
- Customer: Buyer (shipping address, payment method, order
history)
- Order: Purchase transaction (items, prices, discounts, total,
payment status)
- ShoppingCart: Draft order being composed
```

Inventory Context:

```
bounded_context:
  id: bc_inventory
  domain_ref: dom_warehouse

Model within this context:
- Product: Stock item (SKU, quantity on hand, warehouse location,
reorder point)
- Supplier: Vendor (lead times, minimum order quantity, pricing
tiers)
- InventoryMovement: Stock in/out transactions (receiving,
picking, transfers)
- Warehouse: Physical location (zones, bins, capacity)
```

Shipping Context:

```
bounded_context:
  id: bc_shipping
  domain_ref: dom_fulfillment

Model within this context:
- Shipment:
Package to deliver (tracking number, carrier, delivery status)
- Address: Validated shipping destination
- Package: Physical container (dimensions, weight, contents,
label)
- Carrier: Shipping company (rates, service levels, delivery
estimates)
```

Notice: "Product" appears in all three contexts but with completely different attributes and behaviors relevant to each context. This is not duplication—it's appropriate modeling. Trying to unify these into one "Product" class would create an incoherent mess.

Decision Tree for Bounded Contexts

```
Does the team use the same terms with different meanings?
├─ YES → Separate contexts needed
└─ NO → Could different business processes need different models?
    ├─ YES → Separate contexts likely beneficial
    └─ NO → Do teams have different priorities or schedules?
        ├─ YES → Consider separate contexts for independence
        └─ NO → Single context may be appropriate
```

Bounded Contexts are not just about code organization—they're about recognizing that different parts of the business see the world differently, and that's okay. The key is making those boundaries explicit, protecting model integrity within each context, and managing integration between contexts deliberately through Context Mapping patterns.

2.5 Context Mapping

Once you've identified multiple Bounded Contexts, the next challenge is managing their relationships and integrations. **Context Mapping** makes these relationships explicit.

Definition

A **Context Map** is a document that describes the relationships between different Bounded Contexts, showing how they integrate, depend on each other, and how teams coordinate around those integrations.

Purpose of Context Mapping

1. **Global view**: Provides a system-wide perspective beyond individual contexts
2. **Explicit integration patterns**: Documents how contexts actually relate
3. **Team coordination**: Helps teams understand dependencies and responsibilities
4. **Problem identification**: Reveals integration bottlenecks and issues
5. **Refactoring guidance**: Informs decisions about context boundaries

Context Map Components

1. **Bounded Contexts**: Boxes or bubbles representing each context
2. **Relationships**: Lines showing connections and dependencies
3. **Integration Patterns**: Labels indicating the type of relationship (Partnership, Customer/Supplier, etc.)
4. **Directionality**: Upstream/downstream flow of influence
5. **Teams**: Which teams own which contexts
6. **Communication patterns**: How teams coordinate

The Name Field in Context Mappings

Context mappings include a **required `name` field** to improve clarity and documentation:

```
context_mappings:
  - id: cm_sales_to_inventory
    name: "Sales to Inventory Product Synchronization"
    upstream_context: bc_sales
    downstream_context: bc_inventory
    relationship_type: customer_supplier
```

The `name` field serves multiple purposes:
- **Human-readable description**: Explains what this integration does
- **Documentation anchor**: Referenced in architecture docs and diagrams
- **Search and navigation**: Easier to find relevant mappings
- **Communication**: Teams can refer to integrations by name

Power Dynamics: Upstream and Downstream

Context relationships have **power dynamics**:

Upstream Context (Supplier):
- Provides services or data to others
- Controls the model and interface
- Changes potentially impact downstream
- Has more power in the relationship

Downstream Context (Customer):
- Depends on upstream
- Must adapt to upstream changes
- Has less power but can influence
- Must protect itself if needed

Example:

```
[Payment Service] (Upstream)
       ↓ provides payment processing
[Sales Context] (Downstream)
```

Sales depends on Payment Service. Payment Service controls the integration contract.

Integration Pattern Overview

Different relationship patterns require different strategies:

Cooperative Patterns (both sides invested in success):
- **Partnership**: Joint planning, coordinated development, mutual commitment
- **Shared Kernel**: Small shared model subset, both teams maintain
- **Customer/Supplier**: Downstream needs factor into upstream planning

Upstream-Driven Patterns:
- **Open Host Service**: Generalized API serving many consumers
- **Published Language**: Standard data format for exchanges

Downstream-Protection Patterns:
- **Conformist**: Accept upstream model as-is (simplicity over protection)
- **Anti-Corruption Layer**: Translation layer protecting downstream model

No-Integration Patterns:
- **Separate Ways**: Explicit decision not to integrate
- **Big Ball of Mud**: Isolate legacy mess from clean contexts

These patterns will be detailed in Section 2.6.

Creating a Context Map

Process:
1. **Identify all bounded contexts**: List every context in the system
2. **Map dependencies**: Who needs what from whom?
3. **Assess power dynamics**: Which contexts are upstream, which downstream?
4. **Choose integration patterns**: Select appropriate pattern for each relationship
5. **Document in Context Map**: Create visual diagram with annotations
6. **Review and update regularly**: Context Maps evolve with the system

Visual Notation Example

```
[Sales Context] --Customer/Supplier--> [Inventory Context]
      |
      |--Conformist--> [Legacy Billing System]
      |
      |--ACL--> [External Payment Gateway]

[Shipping Context] <--Shared Kernel--> [Inventory Context]
      |
      |--Open Host Service--> [Multiple Tracking Apps]

[Analytics Context] --Separate Ways-- (no direct integration with
Sales)
```

Context Map Documentation

Beyond the diagram, document:
- **Pattern rationale**: Why this pattern was chosen
- **Team responsibilities**: Who maintains what
- **Integration details**: APIs, message formats, schedules
- **Known issues**: Current pain points
- **Future plans**: Planned changes to relationships

Example Schema (strategic-example.yaml):

```
context_mappings:
  - id: cm_profile_to_matching
    name: "Candidate Profile to Job Matching Data Flow"
    upstream_context: bc_profile
    downstream_context: bc_matching
    relationship_type: customer_supplier
    description: >
      Matching context consumes candidate profile data via events.
      Profile context publishes CandidateProfileUpdated events.

  - id: cm_job_catalog_to_matching
    name: "Job Catalog to Matching Integration"
    upstream_context: bc_job_catalog
    downstream_context: bc_matching
    relationship_type: customer_supplier
```

Benefits of Explicit Context Mapping

- **Clarity**: Everyone sees the system structure
- **Communication**: Common language for discussing integration
- **Problem detection**: Highlights bottlenecks and dependencies
- **Team coordination**: Clear ownership and responsibilities
- **Architecture decisions**: Informs where to invest in refactoring

Common Anti-Patterns

1. **No Context Map**: Teams work independently without global view
2. **Stale Map**: Map exists but doesn't reflect reality
3. **Pattern Mismatch**: Documented pattern doesn't match actual integration
4. **Over-complication**: Too many integration patterns, excessive complexity
5. **Ignoring Patterns**: Map exists but teams don't follow documented patterns

Context Mapping brings discipline to integration. Rather than ad-hoc connections, teams explicitly choose how contexts relate, document those choices, and coordinate around them. The Context Map becomes a living architectural document guiding system evolution.

2.6 Integration Patterns in Detail

Section 2.5 introduced the Context Map as a tool for documenting relationships. Now we'll examine each integration pattern in detail, exploring when to use each, how to implement them, and their trade-offs.

Context Mapping patterns provide a vocabulary for describing how Bounded Contexts relate and integrate. Each pattern addresses specific forces and trade-offs. Understanding when to apply each pattern is crucial for effective strategic design.

Pattern Categories

By Power Dynamic:
- Upstream patterns (supplier controls)
- Downstream patterns (customer adapts)
- Symmetric patterns (equal power or no integration)

By Integration Strategy:
- Shared model (common code or schema)
- Translation (one side translates)
- Isolated (no translation needed)

Let's examine each pattern systematically:

2.6.1 Partnership

When to Use: Two contexts are tightly coupled such that the success of one depends on the success of the other.

Context:
- Two teams need each other to succeed
- Failure in either context causes failure in both
- Strong interdependence, typically for core domain integration
- Teams can coordinate effectively

Solution: Form a partnership with joint planning and coordinated development.

Implementation:
- Regular joint planning sessions
- Coordinated sprint schedules and releases
- Shared integration tests
- Cross-team pair programming
- Mutual commitment to success

Consequences:

Benefits:
- Strong alignment between teams
- Quick issue resolution
- Shared understanding of requirements
- Coordinated evolution of both contexts

Drawbacks:
- High coordination overhead
- Meeting fatigue
- Slower independent progress
- Difficult with distributed or remote teams

Example: Shopping cart and payment processing contexts in e-commerce must work together seamlessly. Changes to payment flow require coordinated changes in shopping cart UI and logic.

2.6.2 Shared Kernel

When to Use: Two contexts need to share a small subset of the domain model.

Context:
- Both teams contribute to shared elements
- Tight integration required
- Truly shared concepts exist
- Teams can coordinate changes

Solution: Designate a small, well-defined shared subset of the model that both teams maintain jointly.

Critical Constraint: Keep the shared kernel SMALL (5-10% of total model). Only truly shared, stable core concepts belong here.

Implementation:
- Shared code repository or module
- Both teams approve changes (joint ownership)
- Continuous integration of shared code
- Automated tests prevent breakage
- Version together, release together

Examples of Appropriate Shared Kernel:
- `Address` value object shared between Shipping and Billing contexts
- `Money` type shared across financial contexts
- `ProductId` shared between Catalog and Inventory (just the ID, not full Product model)

Consequences:

Benefits:
- Eliminates duplication for truly shared concepts
- Ensures consistency
- Reduces translation overhead
- Easier integration

Drawbacks:
- Coordination overhead for changes
- Slower independent development
- Risk of uncontrolled growth
- Requires discipline to keep small

Common Mistakes:
1. **Kernel Too Large**: Shared area grows until contexts effectively merge
2. **Unilateral Changes**: One team changes without consulting other
3. **No Governance**: No clear process for managing changes

2.6.3 Customer/Supplier Development

When to Use: Clear upstream (supplier) and downstream (customer) relationship where both teams want to succeed.

Context:
- Downstream depends on upstream
- Upstream is willing to support downstream
- Both teams invested in relationship
- Downstream has some influence

Power Dynamic:
- **Upstream (Supplier)**: Controls the model and interface
- **Downstream (Customer)**: Influences but doesn't control
- **Balance**: Negotiated relationship, not dictated

Solution:
- Downstream expresses needs to upstream
- Upstream budgets time for downstream support

- Automated acceptance tests define the contract (downstream writes, upstream ensures they pass)
- Regular planning meetings
- SLAs or agreements on support levels

Implementation:
- Downstream writes acceptance tests expressing their needs
- Upstream ensures tests pass and stay passing
- Regular joint planning: downstream presents requirements, upstream allocates capacity
- API contracts with versioning
- Deprecation process for changes

Consequences:

Benefits:
- Clear responsibilities
- Downstream has influence on upstream roadmap
- Predictable evolution
- Quality interfaces

Drawbacks:
- Coordination overhead
- Upstream workload for downstream support
- Potential delays for downstream

Example: Internal platform team (upstream) serving application feature teams (downstream). Platform team allocates sprint capacity for feature team requests, and feature teams write acceptance tests for the platform APIs they depend on.

2.6.4 Conformist

When to Use: Downstream team conforms to upstream model despite that model not being ideal for downstream needs.

Context:
- Upstream has all the power (external system, vendor API, uninterested internal team)

- Downstream cannot influence upstream
- Translation layer would be expensive relative to benefit
- Upstream model is "good enough"

Solution:
- Accept upstream model as-is
- Conform downstream design to upstream
- Eliminate translation layer for simplicity
- Stay current with upstream changes automatically

Implementation:
- Use upstream model directly in downstream code
- Minimal or no transformation
- Wrapper classes at most
- Track upstream changes and adapt

Consequences:

Benefits:
- Simple, low-maintenance integration
- Automatically stay current with upstream
- No translation bugs or overhead
- Reduced development effort

Drawbacks:
- Suboptimal model in downstream context
- No control over upstream changes
- Upstream problems propagate to downstream
- Potential impedance mismatch with domain

When to Use:
✓ Upstream is unchangeable (vendor, SaaS, external system)
✓ Translation cost exceeds benefits
✓ Upstream model is adequate (not terrible)
✓ Simple integration is higher priority than perfect model

When to Avoid:
✗ Downstream is core domain (needs protection via ACL)
✗ Upstream model is very poor fit
✗ Have resources for Anti-Corruption Layer

Example: Using Stripe's payment API model directly rather than translating to internal payment domain model. Stripe's model is adequate, and translation would add complexity without significant benefit.

2.6.5 Anti-Corruption Layer (ACL)

When to Use: Downstream context needs protection from upstream model that doesn't fit downstream domain.

Context:
- Upstream model is poor fit for downstream needs
- Downstream is core domain requiring protection
- Upstream changes frequently or unpredictably
- Translation cost is justified by model integrity

Solution: Create an isolating translation layer that translates between upstream and downstream models.

Components:
1. **Facade**: Simplified interface presented to downstream domain
2. **Adapter**: Implements facade using upstream API
3. **Translator**: Converts between upstream and downstream models
4. **DTOs**: Data transfer objects for boundary crossing

Structure:

```
Downstream Domain Model (protected, clean)
         ↓
     [ACL Layer]
     ├─ Facades (clean interface)
     ├─ Adapters (implementation)
     └─ Translators (model conversion)
         ↓
    Upstream API (external, messy)
```

Implementation Example:

```
// Facade: Clean interface for downstream
public interface CustomerDataProvider {
    Customer findCustomerById(CustomerId id);
}

// Adapter: Implements using upstream API
public class LegacyCustomerAdapter implements
CustomerDataProvider {
    private final LegacySystemClient legacyClient;
    private final CustomerTranslator translator;

    public Customer findCustomerById(CustomerId id) {
        LegacyCustomerRecord record =
legacyClient.getCustomer(id.value());
        return translator.toDomainModel(record);
    }
}

// Translator: Converts models
public class CustomerTranslator {
    public Customer toDomainModel(LegacyCustomerRecord record) {
        return new Customer(
            new CustomerId(record.cust_id),
            new CustomerName(record.first_name, record.last_name),
            Email.parse(record.email_addr)
        );
    }
}
```

Consequences:

Benefits:
- Protects downstream domain model integrity
- Isolates from upstream changes
- Maintains Ubiquitous Language in downstream
- Freedom to optimize downstream model independently
- Testable isolation (mock upstream for tests)

Drawbacks:
- Development and maintenance effort
- Performance overhead (translation cost)
- Added complexity
- Can mask upstream problems

When to Use:
✓ Downstream is core domain needing protection
✓ Upstream model is poor fit
✓ Long-term integration expected
✓ Frequent upstream changes anticipated

When to Avoid:
✗ Upstream model is good fit (use Conformist)
✗ Resources are very limited
✗ Short-term or temporary integration
✗ Downstream is generic subdomain

Example: Modern e-commerce platform integrating with legacy mainframe inventory system. ACL translates COBOL data structures into clean domain objects, protecting the modern platform from legacy complexity.

2.6.6 Open Host Service

When to Use: Upstream context serves multiple downstream consumers with varying needs.

Context:
- Multiple downstream contexts depend on upstream
- Each downstream has different requirements
- Point-to-point customization doesn't scale
- Upstream wants to serve many clients efficiently

Solution: Define a generalized, well-documented API that serves common needs of all consumers.

Implementation:
- RESTful API, GraphQL, or gRPC services
- Comprehensive documentation (OpenAPI/Swagger)
- Versioning strategy with clear version management
- Standard protocols (HTTP, gRPC, AMQP)
- SDKs for common languages
- Backward compatibility guarantees
- Clear deprecation process

Characteristics:
- **Well-documented**: Comprehensive API docs
- **Versioned**: Explicit versions with migration paths
- **Comprehensive**: Covers common use cases
- **Stable**: Doesn't change frequently; when it does, versioned changes
- **Standard protocols**: Uses industry-standard formats

Consequences:

Benefits:
- Scales to many consumers
- Reduced custom integration work
- Clear contracts and expectations
- Self-service integration
- Standard tooling support

Drawbacks:
- Must satisfy diverse client needs (can't optimize for one)
- Versioning and deprecation complexity
- Documentation overhead
- Breaking changes affect many clients

Example: Platform authentication service providing Open Host Service to dozens of internal applications. Uses standard OAuth2 protocol, comprehensive docs, versioned API, and SDKs for multiple languages.

2.6.7 Published Language

When to Use: Need for well-documented shared language for information exchange.

Context:
- Multiple parties exchange information
- Standard data format exists or can be created
- Translation is necessary anyway
- Clarity and precision are priorities

Solution: Use or create a well-documented shared language (data format or protocol) for all exchanges.

Format Types:
- **Data schemas**: JSON Schema, XML Schema, Protocol Buffers, Avro
- **Message standards**: CloudEvents, AsyncAPI
- **Industry standards**: FHIR (healthcare), FIX (finance), EDI (logistics), SWIFT (banking)
- **Internal standards**: Company-wide canonical data models

Implementation:
- Formal schema definitions
- Schema versioning
- Comprehensive documentation
- Validation tools
- Schema registry (for distributed systems)
- Backward compatibility rules

Consequences:

Benefits:
- Clarity in communication (no ambiguity)
- Reusable across multiple integrations
- Well-understood if industry standard
- Tooling support for validation and code generation
- Reduces translation errors

Drawbacks:
- Overhead of schema maintenance
- May not fit all contexts perfectly
- Versioning challenges
- Learning curve for new standards

Examples:
- **Healthcare**: HL7 FHIR for health records exchange
- **Finance**: FIX protocol for securities trading
- **E-commerce**: Google Shopping Product Feed format
- **Events**: CloudEvents standard for event metadata

Often Combined With: Open Host Service (upstream provides API using Published Language for data format)

2.6.8 Separate Ways

When to Use: Integration between two contexts provides little value; better to go separate ways.

Context:
- Integration cost exceeds benefit
- Contexts address different concerns with minimal overlap
- Duplication is acceptable
- Teams work more effectively independently

Solution: Explicitly declare the contexts have no relationship. Solve overlapping problems independently.

Implementation:
- Document the decision not to integrate (make it explicit, not accidental)
- No shared code, no API integration
- Independent roadmaps and planning
- Accept some duplication of functionality or data
- Separate teams with no coordination overhead

Consequences:

Benefits:
- No integration complexity
- Complete independence
- Faster progress independently
- No coordination overhead
- Simpler architecture

Drawbacks:
- Duplication of effort or data
- Potentially inconsistent user experience
- May need integration later (technical debt if assumption was wrong)
- Lost opportunities for synergy

When to Use:
✓ Minimal overlap between contexts
✓ Integration cost is very high
✓ Different lifecycles or evolution speeds
✓ Different business areas with no shared users
✓ Temporary duplication is acceptable

Examples:
- HR training platform and customer-facing product (different users, no integration value)
- Internal wiki and product documentation (separate audiences, separate concerns)
- Marketing website and operational inventory system (different purposes, integration not needed)

Common Mistake: Choosing Separate Ways to avoid integration work when integration is actually needed. This creates long-term problems.

2.6.9 Big Ball of Mud

When to Use: Recognize when a context (typically legacy) has no useful model; isolate it from clean contexts.

Context:
- Legacy system exists that is a mess (spaghetti code, no coherent model)
- Replacement is not feasible (too risky, too expensive, too critical to business)
- System must continue operating
- New development is happening in clean contexts

Solution:
- **Recognize it for what it is**: Don't pretend it's well-modeled
- **Isolate it**: Prevent contamination of clean contexts
- **Wrap it with ACL**: Protect new contexts from the mess
- **Don't expand it**: New features go in clean contexts, not in the mud
- **Accept it**: It works even if ugly; pragmatism over purity

Implementation:
- Anti-Corruption Layer around legacy system
- No direct access from clean contexts
- API façade hiding internal mess
- Separate maintenance team
- Minimal new development (keep it running, don't expand it)
- Gradual extraction via Strangler Fig Pattern (optional)

Strangler Fig Pattern (for gradual migration):
1. Identify capability to extract from legacy
2. Build clean version in new bounded context
3. Route new traffic to new version
4. Gradually migrate existing traffic
5. Eventually retire old functionality

Consequences:

Benefits:
- Realistic acknowledgment of reality
- Protects clean contexts from contamination
- Pragmatic approach (works with what you have)
- Allows progress in new contexts while legacy continues

Drawbacks:
- Continued maintenance burden
- Technical debt persists
- May be difficult to extend or modify
- Team morale (nobody wants to work in the mud)

Example: Decades-old mainframe system handling critical transactions. Too risky to replace all at once. Wrapped with API façade and Anti-Corruption Layer. New features built in modern microservices that integrate via the façade. Over time, capabilities gradually extracted using Strangler Fig.

Pattern Selection Guide

When you control both sides:
- High interdependence + equal power → **Partnership**
- High interdependence + shared model → **Shared Kernel**
- One-way dependency + negotiation possible → **Customer/Supplier**

When upstream has power:
- Downstream needs protection (core domain) → **Anti-Corruption Layer**
- Upstream model is adequate → **Conformist**

When serving many clients:
- Multiple consumers with varying needs → **Open Host Service**
- Standard data format needed → **Published Language**

When integration value is low:
- Minimal overlap, high integration cost → **Separate Ways**
- Legacy mess → **Big Ball of Mud** (with ACL protection)

Integration Complexity Matrix

Upstream Situation	Downstream Need	Best Pattern
Willing partner	Equal partner	Partnership
Willing partner	Needs protection	Customer/Supplier + ACL
Unwilling to change	Can accept model	Conformist
Unwilling to change	Needs protection	Anti-Corruption Layer
Many clients	Various needs	Open Host Service
Legacy mess	Clean context	Big Ball of Mud + ACL
No overlap	No overlap	Separate Ways

Context Mapping patterns provide the vocabulary for discussing integration strategy. Rather than ad-hoc integrations, teams choose patterns deliberately based on power dynamics, model fit, and cost-benefit analysis. The pattern choice should be explicit, documented in the Context Map, and understood by all teams involved.

3. Ubiquitous Language

Strategic patterns provide structure—Domains, Bounded Contexts, Context Mappings. But what brings these structures to life is language. **Ubiquitous Language** is the common, rigorous language built collaboratively between developers and domain experts, based on the domain model used in the software.

Key Insight: The language should be "ubiquitous"—used everywhere by everyone involved in the software. Not "business language" and "technical language," but one unified language.

3.1 Purpose and Benefits

Why Ubiquitous Language Matters

1. **Eliminates Translation**: No gap between how experts talk and how code is written. When an expert says "Customer submits Order," the code has a `Customer` class with a `submitOrder()` method.

2. **Precision**: Software requires unambiguous terminology. Vague terms like "process" or "handle" create confusion. Precise terms like "submit," "approve," "cancel" have clear meaning.
3. **Shared Understanding**: Everyone discusses the same concepts consistently—developers, domain experts, product managers, testers.
4. **Living Documentation**: Code itself documents the domain. Reading method names like `authorizePayment()`, `reserveInventory()`, `shipOrder()` tells the domain story.
5. **Discovery Tool**: Language evolution reveals deeper domain insights. Struggling to name something often indicates incomplete understanding.

Problems Solved

Without Ubiquitous Language:
- Developers use technical terms ("persist entity," "update status")
- Experts use business terms ("submit order," "approve request")
- Constant mental translation introduces errors
- Requirements documents use different language than code
- Subtle concept differences get lost in translation
- Communication breakdowns between teams

With Ubiquitous Language:
- Everyone uses the same terms
- Code is conceptually readable by domain experts
- Changes map directly from conversation to code
- Precision in communication
- Shared mental model across teams

Example of the Problem:

```
Meeting discussion:
Expert: "When a policy lapses, we need to notify the agent."
Developer: "OK, so when the contract expires, we send an email."

Later, in code:
class Contract {
    void expire() {
        emailService.send(...);
    }
}

Problem: "Lapse" and "expire" are different concepts! A lapse
means non-payment. An expire means reaching the end date. The
translation lost critical meaning.
```

With Ubiquitous Language:

```
Meeting:
Expert: "When a policy lapses due to non-payment..."
Developer: "What exactly is a 'lapse'?"
Expert: "It means the premium wasn't paid by the grace period
end."
Developer: "Is that different from 'expiration'?"
Expert:
"Yes! Expiration is reaching the policy end date. Lapse is
automatic due to non-payment."

Code:
class Policy {
    void lapse(NonPaymentReason reason) {
        // Lapse-specific logic
        notifyAgent(new PolicyLapsed(this.id, reason));
    }

    void expire() {
        // Expiration-specific logic
        archivePolicy();
    }
}
```

3.2 Development Process

Ubiquitous Language is not created upfront in requirements. It emerges and evolves through continuous collaboration.

Phase 1: Discovery

Activities:
1. **Domain Exploration Sessions**: Developers and domain experts meet regularly to explore the domain through concrete scenarios.

1. **Listen for Terms**: Pay attention to expert vocabulary. Note when experts correct you. Identify ambiguous terms. Find synonyms that might indicate different concepts.
2. **Model Sketching**: Draw simple diagrams using terms experts recognize. Test understanding with examples. Refine based on feedback.

Example Discovery Session:

```
Developer: "So when a customer places an order..."
Expert: "Well, actually we don't call it 'placing'—we say the
customer 'submits' an order."
Developer: "Okay, when they submit an order, how do we handle
inventory?"
Expert: "We 'reserve' inventory for the order. It's not
'allocated' until the order is 'confirmed'."
Developer: "What's the difference between 'reserved' and
'allocated'?"
Expert: "Reserved means we've set it aside temporarily. Allocated
means it's definitely going to that order and can't be used
elsewhere."

New terms discovered:
- Submit (not "place")
- Reserve (temporary)
- Allocate (committed)
- Confirm (distinct state transition)
```

Phase 2: Refinement

Activities:

1. **Challenge Terminology**: Is this term precise enough? Does it mean one thing? Is it used consistently? Does it work in code?

1. **Resolve Ambiguities**: When experts use the same word differently, when one concept needs multiple terms, when existing terms are vague.
2. **Test with Scenarios**: Walk through business processes using only defined terms. Identify gaps. Refine definitions.

Example Refinement:

```
Initial: "Process the payment"

Problems:
- "Process" is vague—what does it actually mean?
- Too generic for precise implementation

Refined terms discovered through questioning:
- "Authorize" the payment: Check that funds are available
- "Capture" the payment: Actually charge the card
- "Settle" the payment: Transfer funds from processor to merchant
account
- "Refund" the payment: Return funds to customer

Code now has precise methods:
payment.authorize()
payment.capture()
payment.settle()
payment.refund(amount)
```

Phase 3: Documentation

Document the language in a **Glossary** that evolves with the model.

Glossary Template:

```
## Term: Order

**Definition**: A customer's request to purchase one or more
products

**Description**:
An Order represents a customer's intention to buy products.
Orders progress through states: Draft → Submitted → Paid →
Fulfilled → Completed.
Each state transition has specific business rules.

**Synonyms**: None (previously called "Shopping Transaction"—
deprecated)

**Related Terms**:
- OrderLine: Order contains multiple OrderLines
- Customer: Order belongs to a Customer
- Payment: Order associated with Payment

**States**:
- Draft: Being composed, can be modified
- Submitted: Customer has submitted, awaiting payment
- Paid: Payment confirmed
- Fulfilled: Products shipped
- Completed: Customer received products
- Cancelled: Order cancelled

**Code Representation**:
- Class: `Order` (Aggregate Root)
- Methods: `submit()`, `addLine()`, `cancel()`
- Events: `OrderSubmitted`, `OrderPaid`, `OrderFulfilled`

**Business Rules**:
- Order must have at least one OrderLine to be submitted
- Order cannot be modified after submission (except cancellation)
- Total equals sum of all OrderLine subtotals

**Last Updated**: 2025-01-24
**Changed By**: Development Team
**Reason**: Added "Fulfilled" state to distinguish from
"Completed"
```

Phase 4: Maintenance

Keep Language Alive:
1. **Regular Review**: Language evolves as understanding deepens
2. **Update Code**: Refactor when language changes (rename classes, methods)
3. **Onboard New Members**: Teach the language explicitly to new team members
4. **Challenge Staleness**: Remove obsolete terms
5. **Document Changes**: Track language evolution in glossary

3.3 Language in Code

The power of Ubiquitous Language is realized when it's embedded directly in code structure.

Classes Match Terms

```
// Domain term: "Customer"
public class Customer { }

// Domain term: "Order"
public class Order { }

// Domain term: "Payment Authorization"
public class PaymentAuthorization { }

// NOT technical abstractions:
// Bad: public class CustomerEntity { }
// Bad: public class OrderDTO { }
// Bad: public class PaymentData { }
```

Methods Use Domain Verbs

```
// Experts say "submit an order"
public void submit() { }

// Experts say "approve a loan application"
public void approve(ApprovalDecision decision) { }

// Experts say "authorize payment"
public PaymentAuthorization authorize(Money amount) { }

// NOT technical jargon:
// Bad: public void persist() { }
// Bad: public void setStatus(int code) { }
// Bad: public void updateRecord() { }
```

Avoid Technical Jargon in Domain Layer

Keep technical concerns out of the domain model:

```
// Bad: Technical language leaking into domain
public void persist() { }
public void setStatusCode(int statusCode) { }
public Map<String, Object> getData() { }

// Good: Domain language
customerRepository.save(customer);  // Persistence is
infrastructure concern
order.submit();  // Use actual business action
product.specifications();  // Return domain concept
```

Package/Module Names Use Domain Terms

```
com.company.ecommerce.sales      // "Sales" is domain term
com.company.ecommerce.inventory  // "Inventory" is domain term
com.company.ecommerce.shipping   // "Shipping" is domain term

// NOT technical categories:
// Bad: com.company.ecommerce.entities
// Bad: com.company.ecommerce.services
// Bad: com.company.ecommerce.managers
```

Value Objects for Domain Concepts

Make concepts explicit as types rather than primitives:

```
// Bad: Primitive obsession
String email;
double amount;
String currency;
int zipCode;

// Good: Domain concepts explicit
Email email;           // "Email" is a domain concept with
validation
Money amount;          // "Money" combines amount + currency
ZipCode zipCode;       // "ZipCode" has format validation
```

Enums for Fixed Sets

```
public enum OrderStatus {
    DRAFT,      // Customer composing order
    SUBMITTED,  // Awaiting payment
    PAID,       // Payment confirmed
    FULFILLED,  // Products shipped
    COMPLETED,  // Customer received products
    CANCELLED   // Order cancelled
}

// NOT generic codes:
// Bad: public enum Status { STATUS_1, STATUS_2, STATUS_3 }
```

Comments in Domain Language

```
// Business rule: Orders below minimum must be rejected
if (total.isLessThan(Money.minimumOrder())) {
    throw new BelowMinimumOrderException();
}

// Policy: Customers with overdue invoices cannot place new orders
if (customer.hasOverdueInvoices()) {
    throw new CustomerCreditHoldException();
}

// NOT technical comments:
// Bad: // Check if value is less than constant
// Bad: // Throw exception if condition is true
```

3.4 Language Evolution

Ubiquitous Language is not static—it evolves as domain understanding deepens.

Recognizing Need for Change

Signs that language needs refinement:
- Team members use different terms for the same concept
- Developers translate between code terms and conversation terms
- Expert corrections are frequent
- Terms are ambiguous or overloaded
- New understanding of the domain emerges
- Code becomes awkward or convoluted

Managing Change

Process:
1. **Identify Issue**: Term is unclear, incorrect, or causing confusion
2. **Propose Change**: Suggest better term based on expert input
3. **Validate with Experts**: Does the new term work? Do they use it naturally?
4. **Update Glossary**: Document the change with rationale

5. **Refactor Code**: Rename classes, methods, variables to match
6. **Communicate**: Tell the team about the change
7. **Deprecate Old Term**: Mark old usage as deprecated

Example Evolution:

```
Version 1: "Transaction" (too vague, could mean anything)
   ↓ Team realizes this is unclear
Version 2: "Sale" (closer, but experts correct this)
   ↓ Experts say "we call them orders, not sales"
Version 3: "Order" (correct term experts actually use)

Code evolution:
class Transaction { }  → class Sale { }  → class Order { }
processTransaction()   → completeSale()  → submitOrder()
```

Deprecation Process

When removing terms:
1. Mark as deprecated in glossary
2. Note replacement term
3. Set deadline for migration
4. Track remaining usages
5. Remove from glossary when fully migrated

```
## Term: Shopping Transaction [DEPRECATED]

**Status**: Deprecated as of 2025-01-10
**Replacement**: Use "Order" instead
**Reason**: Domain experts consistently use "Order", not
"Transaction"
**Migration Deadline**: 2025-02-01
**Remaining Usages**: 3 (see ticket #1234)
```

3.5 Integration with Other DDD Patterns

With Bounded Contexts

- Each Bounded Context has its own Ubiquitous Language

- The same term may mean different things in different contexts (and that's OK!)
- Explicitly document translations between contexts

Example:

```
In Sales Context:
- "Product" = Item for sale (price, description, promotions)

In Inventory Context:
- "Product" = Stock item (SKU, quantity, warehouse location)

In Shipping Context:
- "Product" = Item to ship (weight, dimensions, fragility)

→ Same word, three different models in three contexts
→ Context boundaries make this explicit and manageable
```

With Aggregates

- Aggregate names come from ubiquitous language
- Methods on aggregates use domain verbs
- Events raised are named in domain terms (past tense)

```
public class Order {  // "Order" from ubiquitous language
    public void submit() {  // "submit" from ubiquitous language
        // ... business logic
        events.add(new OrderSubmitted(this.id));  //
"OrderSubmitted" event
    }
}
```

With Domain Events

- Event names are past-tense domain actions
- Use domain language, not technical language
- Include domain-relevant data, not technical IDs

```
// Good: Domain language
public class OrderSubmitted implements DomainEvent {
    private final OrderId orderId;
    private final CustomerId customerId;
    private final Money totalAmount;
}

// Bad: Technical language
public class OrderSubmitEvent {
    private final String entityId;
    private final int statusCode;
}
```

With Testing

Tests should read like business specifications using ubiquitous language:

```
@Test
void customer_cannot_submit_empty_order() {
    // Given
    Order order = new Order(customerId);
    // No lines added

    // When/Then
    assertThrows(EmptyOrderException.class, () -> order.submit());
}

@Test
void order_reserves_inventory_when_submitted() {
    // Given
    Order order = createOrderWith(product, quantity);

    // When
    order.submit();

    // Then
    verify(inventoryService).reserve(product, quantity);
}
```

3.6 Common Pitfalls

1. Technical Terms in Domain

```
// Bad: Don't do this:
customer.persist();
order.setStatusCode(3);
product.getData();
entity.update(map);

// Good: Do this:
customerRepository.save(customer);  // Persistence is
infrastructure
order.submit();  // Domain action
product.specifications();  // Domain concept
order.changeAddress(newAddress);  // Specific domain operation
```

2. Ambiguous Terms

Problem: "Process" the order (what does "process" mean exactly?)

Solution: Be specific:
- submit the order
- fulfill the order
- ship the order
- complete the order
- cancel the order

Each has distinct meaning and different business rules.

3. Developer-Only Language

```
// ❌ Bad: Terms experts don't recognize
class OrderDTO { }
class OrderEntity { }
class OrderManager { }
class OrderHelper { }

// ✓ Good: Domain terms
class Order { }  // The concept itself
class OrderRepository { }  // DDD pattern name
class OrderService { }  // If "service" is a domain term
```

4. Expert Jargon Without Clarity

Problem: Using insider terms without defining them clearly

Solution: Define even "obvious" terms:
- What exactly is a "submission"? (When does it happen? What changes?)
- When is an order "confirmed" vs "completed"? (Distinct states? Same thing?)
- What makes inventory "available" vs "reserved"? (Clear criteria needed)

5. Stale Language

Problem: Code uses old terms after domain understanding has evolved

Solution: Continuous refactoring to match current understanding. Language evolution is normal and healthy—embrace it.

3.7 Practical Techniques

Event Storming

Process:
1. Gather domain experts and developers
2. Identify Domain Events (things that happen): "OrderSubmitted," "PaymentAuthorized"
3. Place events on timeline
4. Identify commands that trigger events: "SubmitOrder," "AuthorizePayment"
5. Identify Aggregates that handle commands: "Order," "Payment"
6. Group into Bounded Contexts

Output: Events, commands, and Aggregates are all named using Ubiquitous Language.

Domain Storytelling

Process:
1. Expert tells story of a domain process
2. Developers illustrate with simple icons
3. Identify actors, work objects, activities
4. Map out the sequence

Example:

```
"A Customer browses the Catalog and adds Products to their
Shopping Cart.
When ready, they submit their Order. The system reserves Inventory
and
authorizes Payment. Once authorized, the Order is confirmed and
sent to
the Warehouse for fulfillment."

Terms discovered:
- Actors: Customer, System, Warehouse
- Work Objects: Catalog, Product, Shopping Cart, Order, Inventory,
Payment
- Actions: browse, add, submit, reserve, authorize, confirm,
fulfill
```

Example Mapping

Process:
1. Start with user story or feature
2. Provide concrete examples
3. Identify business rules
4. Surface questions

```
Story: Customer submits order

Examples:
- Happy path: Customer submits order with valid payment → Order
confirmed
- Alternative: Customer submits but payment fails → Order remains
in Submitted state
- Edge case: Customer submits at exactly midnight → Use submission
timestamp

Rules:
- Order must have at least one item
- Payment must be authorized
- Inventory must be available

Questions:
- What happens if inventory becomes unavailable after payment
authorization?
- Can customer modify order after submission? (Answer: No, must
cancel and resubmit)
```

3.8 Cross-Context Term Mapping

When the same term appears in multiple contexts with different meanings, document the translations:

```
# Term Translations Between Contexts

## "Product"

**In Sales Context:**
- Meaning: Item for sale, customer-facing view
- Attributes: name, price, description, category, images, reviews
- Focus: Marketing and selling

**In Inventory Context:**
- Meaning: Physical stock item with location tracking
- Attributes: SKU, quantity_on_hand, warehouse_location,
reorder_point
- Focus: Stock management

**In Shipping Context:**
- Meaning: Item to ship with physical dimensions
- Attributes: dimensions, weight, fragility, hazard_class
- Focus: Logistics and shipping

**Translation**:
- Sales.Product.id maps to Inventory.Product.sku
- Different models for different purposes
- Integration via Context Mapping patterns
```

3.9 Success Metrics

Language is working when:

✓ Domain experts recognize code structure conceptually
✓ Developers and experts use same terminology in meetings
✓ No translation needed in conversations between teams
✓ New team members learn language quickly (onboarding includes glossary)
✓ Code changes map directly to business changes
✓ Tests read like business specifications
✓ Feature discussions reference actual class/method names

Language needs work when:

✗ Frequent misunderstandings in meetings
✗ Code uses different terms than spoken conversations
✗ Glossary is out of date or ignored
✗ Multiple terms exist for the same concept
✗ Terms are ambiguous or overloaded
✗ Developers avoid using domain terms in code

The Ultimate Test: Can a domain expert read the test suite and understand what the system does? If yes, you have achieved ubiquitous language.

Part I Summary

Part I established the **strategic foundations** of Domain-Driven Design:

Section 1: Introduction & DDD Philosophy
- DDD solves complexity through model-driven design
- Ubiquitous language bridges business and code
- Strategic and tactical patterns work together
- Use DDD for complex domains with engaged experts

Section 2: Strategic Design Patterns
- **System**: Root object organizing all strategic elements
- **Domains & Subdomains**: Core, Supporting, Generic classification drives investment
- **Bounded Contexts**: Explicit boundaries where models apply
- **Context Mapping**: Integration patterns for managing relationships
- **Integration Patterns**: Partnership, Shared Kernel, Customer/Supplier, Conformist, ACL, Open Host Service, Published Language, Separate Ways, Big Ball of Mud

Section 3: Ubiquitous Language
- Common language shared by developers and experts
- Embedded directly in code structure
- Evolves as understanding deepens
- One language per Bounded Context
- Enables precision and shared understanding

What's Next: Part II introduces **Domain Storytelling**, a collaborative discovery technique that helps teams identify Bounded Contexts, surface Ubiquitous Language, and understand business processes before diving into tactical implementation.

Part II: Discovery & Modeling

4. Domain Storytelling

Before diving into tactical implementation patterns (Aggregates, Entities, Repositories), we need to understand the domain deeply. **Domain Storytelling** is a collaborative technique that helps teams discover domain knowledge, surface Ubiquitous Language, and identify Bounded Context boundaries through narrative.

> "Domain Storytelling is a collaborative Domain-Driven Design technique used to capture and visualize business processes as stories involving actors, activities, and work objects."
> — Stefan Hofer and Henning Schwentner, *Domain Storytelling* (2021)

Positioning in DDD Practice:

```
Discovery → Strategic → Tactical
   ↓           ↓          ↓
Domain     Bounded     Aggregates
Storytelling Contexts  & Entities
```

Domain Storytelling occurs **early** in the DDD journey, before you've identified Bounded Contexts or modeled Aggregates. It's a discovery technique that feeds into strategic design.

4.1 What is Domain Storytelling?

Definition: Domain Storytelling is a workshop-based technique where domain experts tell stories about their work, and developers capture those stories using a simple visual notation.

Core Philosophy:

People naturally think and communicate in stories, not in abstract models. When you ask a domain expert "How does order processing work?", they don't give you a class diagram—they tell you a story:

```
"A Customer selects products from the catalog, adds them to their
shopping cart,
reviews the cart, and submits the order. The system validates
inventory availability,
reserves the items, and sends the order to the payment system.
Once payment is
authorized, we confirm the order and send it to the warehouse for
fulfillment..."
```

This narrative reveals:
- **Actors**: Customer, System, Payment System, Warehouse
- **Activities**: selects, adds, reviews, submits, validates, reserves, sends, authorizes, confirms
- **Work Objects**: products, catalog, shopping cart, order, inventory, payment
- **Sequence**: natural flow of events
- **Rules**: implied constraints (inventory must be available, payment must be authorized)

Domain Storytelling formalizes this natural storytelling into a structured technique that produces artifacts useful for DDD.

Why It Works:

1. **Natural Communication**: Stories are how humans share knowledge. Domain experts are comfortable telling stories about their work.

2. **Concrete Over Abstract**: Stories use concrete scenarios, not abstract concepts. "Customer submits order" is easier to discuss than "Order aggregate state transitions."
3. **Reveals Hidden Knowledge**: The act of telling a story surfaces details experts take for granted but developers need to know.
4. **Shared Understanding**: Everyone (experts and developers) can follow a story. It creates common ground.
5. **Discovery Tool**: Stories reveal domain structure—what the actors are, what they do, what they work with, and how things relate.

Relationship to Other Discovery Techniques:

Technique	Focus	Output	Best For
Domain Storytelling	Actor-centered narratives	Stories with sequence	Understanding processes and workflows
Event Storming	Event-centered exploration	Timeline of events	Discovering Domain Events and commands
Example Mapping	Rule-centered	Examples and rules	Refining specific behaviors and edge cases
User Story Mapping	User journey-centered	Journey maps	Product planning and feature prioritization

Domain Storytelling and Event Storming are complementary. Domain Storytelling provides narrative structure; Event Storming focuses on temporal flow and events. Many teams use both.

4.2 Core Notation: Actor-Activity-Work Object

Now that we understand what Domain Storytelling is and why it works, let's explore the notation used to capture these stories. The notation is deliberately simple so that both domain experts and developers can use it effectively without training.

Domain Storytelling uses a simple pictographic notation that's easy to learn and doesn't require special tools.

Basic Pattern:

```
[Actor] → [Activity] → [Work Object]
```

An **Actor** performs an **Activity** on/with a **Work Object**.

Example:

```
Customer → submits → Order
   ↓
System → validates → Order
   ↓
System → reserves → Inventory
```

Visual Representation:

In practice, teams use:
- **Sticky notes** on a whiteboard (different colors for actors, activities, work objects)
- **Drawing tools** like Miro, Mural, or draw.io
- **Specialized tools** like Egon.io (built specifically for Domain Storytelling)

The notation has four core elements that work together to capture complete business processes.

Actors: Who Performs Actions

Actors are people, systems, or roles that perform actions.

Types of Actors (from schema `kind` field):
- **Person**: Individual humans (Customer, Manager, Operator)
- **System**: Automated systems (Payment System, Notification Service)
- **Role**: Generic actor type by permission (Administrator, Guest)

Visual Convention:
- Persons: Stick figure
- Systems: Rectangle or computer icon
- Roles: Stick figure with label or badge

Naming: Use terms from ubiquitous language. If domain experts say "Customer," use Customer (not User or Client).

Schema Representation:

```
actors:
  - actor_id: act_customer
    name: "Customer"
    kind: person
    # ...
  - actor_id: act_payment_system
    name: "Payment System"
    kind: system
    # ...
```

With actors defined, we can now describe what actions they perform.

Activities: What Actions Occur

Activities are actions that change state or produce results.

Naming Convention:
- **Verb or verb phrase**
- **Present tense**: "submits" not "submitted"
- **Active voice**: "Customer submits" not "order is submitted"
- **Domain language**: Use exact terms experts use

Examples: submits order, approves loan, calculates price, sends notification, reserves inventory, authorizes payment

Schema Representation:

```
activities:
  - activity_id: actv_submit_order
    name: "Submit Order"
    initiated_by_actor_ids: [act_customer]
    uses_work_object_ids: [wobj_shopping_cart, wobj_order]
    # ...
```

Activities connect actors to work objects and produce events. They may also trigger commands or call application services.

Work Objects: What Is Manipulated

Work Objects are domain artifacts that actors manipulate.

Characteristics:
- **Nouns** from ubiquitous language
- Can be tangible (Product, Invoice, Package) or abstract (Order, Approval, Reservation)
- May evolve into **Entities** or **Value Objects** during tactical modeling
- May map to **Aggregates**

Examples: Order, Invoice, Contract, Product, Shopping Cart, Customer Profile, Payment, Shipment

Schema Representation:

```
work_objects:
  - work_object_id: wobj_order
    name: "Order"
    description: "Customer purchase request"
    attributes:
      - name: order_id
        type: uuid
      # ...
    aggregate_id: agg_order  # May reference future aggregate
```

Attributes capture what's important about the work object in the story context. Don't over-specify—enough to understand the object, not full implementation.

Finally, we need to capture the order in which activities occur.

Sequence: The Order of Events

Stories have sequence: one thing happens, then another.

Notation:
- Number the activities: 1, 2, 3, 4...
- Arrows show flow
- Branches show alternatives ("If payment fails...")

Example Sequence:

```
1. Customer → selects → Product
2. Customer → adds to → Shopping Cart
3. Customer → reviews → Shopping Cart
4. Customer → submits → Order
5. System → validates → Order
6. System → reserves → Inventory
7. System → requests → Payment Authorization
8. Payment System → authorizes → Payment
9. System → confirms → Order
10. System → sends to → Warehouse
```

Branching for Alternatives:

```
5. System → validates → Order
   ├─ If valid:
   │  6. System → reserves → Inventory
   └─ If invalid:
      6. System → rejects → Order
```

4.3 Commands, Queries, Events, and Policies

Domain Storytelling integrates naturally with CQRS (Command Query Responsibility Segregation) concepts.

4.3.1 Commands

Commands express user intent—requests that change state.

Characteristics:
- **Imperative**: "Submit Order," "Approve Loan," "Cancel Reservation"
- **Initiated by actors**: Commands come from people or systems
- **May fail**: Business rules can reject commands
- **Target aggregates**: Commands operate on domain objects

Schema Representation:

```
commands:
  - command_id: cmd_submit_order
    name: "Submit Order"
    actor_ids: [act_customer]
    target_aggregate_id: agg_order
    parameters:
      - name: order_id
        type: uuid
        required: true
      - name: payment_method
        type: ref
        ref_id: wobj_payment_method
    emits_events: [evt_order_submitted]
```

In Stories: Commands are the activities that actors initiate.

```
Customer → submits → Order
   ↓ (This activity corresponds to SubmitOrder command)
```

4.3.2 Queries

Queries request information without changing state.

Characteristics:
- **Interrogative or declarative**: "Get Order Status," "View Product Catalog"
- **Read-only**: No side effects
- **Return data**: Return read models or DTOs
- **May join data**: Can aggregate from multiple sources

Schema Representation:

```
queries:
  - query_id: qry_get_order_status
    name: "Get Order Status"
    actor_ids: [act_customer]
    parameters:
      - name: order_id
        type: uuid
        required: true
    returns_read_model_id: rmdl_order_summary
```

In Stories: Query activities are reads, not writes.

```
Customer → views → Order Status
   ↓ (This corresponds to GetOrderStatus query)
```

4.3.3 Events

Events represent facts—things that have happened.

Characteristics:
- **Past tense**: "Order Submitted," "Payment Authorized," "Inventory Reserved"
- **Immutable**: Events cannot be changed (they're historical facts)
- **Caused by activities**: Events result from successful commands
- **Trigger reactions**: May trigger policies or other processes

Schema Representation:

```
events:
  - event_id: evt_order_submitted
    name: "Order Submitted"
    caused_by:
      command_id: cmd_submit_order
    affected_aggregate_id: agg_order
    data_attributes:
      - name: order_id
        type: uuid
      - name: customer_id
        type: uuid
      - name: total_amount
        type: money
    policies_triggered: [pol_send_confirmation_email]
```

In Stories: Events are outcomes of activities.

```
Customer → submits → Order
   ↓ (Results in event)
[OrderSubmitted] event published
   ↓ (Triggers policy)
System → sends → Confirmation Email
```

4.3.4 Policies

Policies create reactive rules: "When [Event] happens, do [Action]."

Pattern: "When X, do Y"

Examples:
- "When Order Submitted, send confirmation email"
- "When Payment Failed, notify customer"
- "When Inventory Low, create reorder request"

Schema Representation:

```
policies:
  - policy_id: pol_send_confirmation
    name: "Send Order Confirmation Email"
    when_event_id: evt_order_submitted
    issues_command_id: cmd_send_email
    description: "Automatically send confirmation when order
submitted"
```

In Stories: Policies connect events to subsequent actions.

```
[OrderSubmitted] → triggers → Send Confirmation Policy
   ↓
System → sends → Confirmation Email (via SendEmail command)
```

Causal Chain:

The complete flow:

```
Actor → initiates Command
   ↓
Command → produces Event
   ↓
Event → triggers Policy
   ↓
Policy → issues new Command
   ↓
(Cycle continues...)
```

This creates a **causal chain** that traces through the entire system.

4.4 Workshop Facilitation

Domain Storytelling happens in facilitated workshops with domain experts and developers.

Workshop Setup

Duration: 2-4 hours per domain area (may require multiple sessions for complex domains)

Participants:
- **Domain Experts**: 2-3 people with deep domain knowledge
- **Developers**: 2-4 developers who will implement the system
- **Facilitator**: 1 person (ideally experienced with Domain Storytelling)
- **Stakeholders**: 0-2 (optional, for context)

Materials:
- **Whiteboard** or digital board (Miro, Mural, Egon.io)
- **Sticky notes** in 3 colors (actors, activities, work objects)
- **Markers**
- **Camera** for documentation
- **Laptop** for notes

Workshop Flow

Phase 1: Warm-Up (15 minutes)

1. **Explain the notation**: Show examples of Actor → Activity → Work Object
2. **Pick simple example**: "Customer buys coffee" or similar trivial story
3. **Practice together**: Have the group walk through the simple story
4. **Answer questions**: Make sure everyone understands the notation

Phase 2: Story Discovery (60-90 minutes)

1. **Pick a business scenario**: Start with a core, common process
   - "How does a customer place an order?"
   - "What happens when a loan application is submitted?"
   - "How do we onboard a new employee?"
2. **Start with the actor**: "Who initiates this process?"
   - Expert: "The Customer starts by..."
3. **Follow the flow**: Ask "Then what?" repeatedly
   - "What does the Customer do first?"
   - "What happens to the Order?"
   - "Then what?"
   - "Who does that?"
   - "What are they working with?"

4. **Capture on board** as you go:
   - Place actor sticky note
   - Add activity
   - Add work object
   - Draw arrow showing flow
   - Number the sequence
5. **Clarify terminology**: When a new term appears, ask:
   - "What exactly is an 'Order'?"
   - "Is 'reserved' different from 'allocated'?"
   - "When you say 'submit,' what changes?"
6. **Capture exceptions and variations**:
   - "What if payment fails?"
   - "Are there different types of orders?"
   - "Can this step be skipped?"

Phase 3: Refinement (30-45 minutes)

1. **Review the story flow**: Read it back to experts
2. **Add missing details**: Identify gaps
3. **Clarify work object definitions**: What attributes matter?
4. **Identify events**: What facts does the story produce?
5. **Discover policies and rules**: What happens automatically?
6. **Consistent naming**: Ensure terms match expert language

Phase 4: Analysis (30-45 minutes)

1. **Identify linguistic boundaries**: Where do terms change meaning?
2. **Look for context candidates**: Natural groupings of concepts
3. **Find natural seams**: Where could the system be separated?
4. **Discuss groupings**: Which activities belong together?

Facilitation Tips

DO:
- ✓ Let domain experts lead the narrative
- ✓ Focus on real, concrete scenarios (not hypotheticals)
- ✓ Use exact terminology from experts

- ✓ Ask "then what?" repeatedly to surface sequence
- ✓ Capture exceptions and edge cases
- ✓ Look for event triggers and policies

DON'T:
- ✗ Jump to implementation details
- ✗ Use technical jargon or UML notation
- ✗ Rush through the story
- ✗ Ignore edge cases or alternatives
- ✗ Impose your understanding on the experts
- ✗ Skip variations ("usually" means there are exceptions!)

Key Questions to Ask

Discovery Questions:
- "Who can initiate this process?"
- "What happens first?"
- "What does [actor] need to do that?"
- "When does [event] trigger?"
- "What are the exceptions or variations?"
- "Who needs to know about this?"
- "What can go wrong?"

Refinement Questions:
- "Is [term] the right word experts use?"
- "Does [term] mean the same thing here as over there?"
- "What makes [work object] valid or complete?"
- "When can this NOT happen?"
- "Are these really the same thing, or are they different?"

4.5 From Stories to Bounded Contexts

Domain Storytelling reveals Bounded Context boundaries through linguistic and organizational patterns.

Step 1: Identify Linguistic Boundaries

Signs of Different Contexts:

1. **Same word, different meanings**:

```
   In Sales: "Order" = Customer purchase request (pricing,
payment)
   In Fulfillment: "Order" = Picking/packing instruction
(location, packaging)
   → Two Bounded Contexts!
   ```

2. **Different words, same concept**:
```text
   In HR: "Employee"
   In Payroll: "Pay Subject"
   → Likely same entity in different contexts
   ```

3. **Experts disagree on terminology**:
```text
   Sales expert: "We 'submit' orders"
   Warehouse expert: "We 'receive' orders"
   → Different perspectives, potentially different contexts
   ```

4. **Natural organizational boundaries**:
```text
   Marketing talks about "campaigns"
   Sales talks about "opportunities"
   → Different domains with different languages
   ```

Example from Story:

```text
Story: E-commerce Checkout

Activities:
1. Customer → selects → Product (in Catalog)
2. Customer → adds → Shopping Cart
3. Customer → submits → Order
4. System → validates → Order (pricing, availability)
5. System → reserves → Inventory (warehouse location, quantity)
6. System → sends → Order to Warehouse (pick list, package
instructions)
```

```
Linguistic Analysis:
- "Order" in steps 3-4: Sales perspective (price, payment)
- "Order" in step 6: Fulfillment perspective (packing, shipping)
- "Inventory" in step 5: Stock management perspective
→ Suggests 3 Bounded Contexts: Sales, Inventory, Fulfillment
```

Step 2: Map Stories to Context Candidates

```
Stories about customer purchases → Sales Context
Stories about stock management → Inventory Context
Stories about packing and shipping → Fulfillment Context
Stories about payment processing → Payment Context
```

Step 3: Identify Aggregate Candidates

Work objects that are always modified together → likely same Aggregate

Example:

```
Order and OrderLine are always changed together
→ Order is Aggregate Root, OrderLine is entity within aggregate

Shopping Cart and Cart Items are always changed together
→ ShoppingCart is Aggregate Root
```

Work objects that serve as entry points → likely Aggregate Roots

Step 4: Extract Events

Activity results → Domain Events

```
Activity: Customer submits Order
→ Event: OrderSubmitted

Activity: System authorizes Payment
→ Event: PaymentAuthorized
```

Cross-boundary triggers → Integration Events

```
OrderSubmitted in Sales Context
→ Triggers InventoryReservation in Inventory Context
→ Integration event crosses context boundary
```

Step 5: Define Context Boundaries

Combine linguistic, organizational, and technical boundaries:

```
bounded_contexts:
  - id: bc_sales
    name: "Sales Context"
    description: "Customer-facing order management"
    ubiquitous_language_terms:
      - Order (purchase request)
      - Customer
      - Product (catalog item with price)
      - Shopping Cart

  - id: bc_inventory
    name: "Inventory Context"
    description: "Stock level management"
    ubiquitous_language_terms:
      - Product (stock item with location)
      - Inventory
      - Warehouse
      - Reorder Point

  - id: bc_fulfillment
    name: "Fulfillment Context"
    description: "Order picking, packing, shipping"
    ubiquitous_language_terms:
      - Order (pick list)
      - Shipment
      - Package
      - Carrier
```

Example Transformation: Story → DDD Model

Domain Story:

```
1. Customer → selects → Product
2. Customer → adds to → ShoppingCart
3. Customer → submits → Order
4. System → validates → Order
5. System → reserves → Inventory
6. System → publishes → OrderPlaced Event
```

Strategic DDD (Bounded Contexts):

```
system:
  id: sys_ecommerce

  bounded_contexts:
    - id: bc_shopping
      domain_ref: dom_sales
      aggregates: [agg_shopping_cart]

    - id: bc_order_management
      domain_ref: dom_sales
      aggregates: [agg_order]

    - id: bc_inventory
      domain_ref: dom_warehouse
      aggregates: [agg_inventory]

  context_mappings:
    - id: cm_order_to_inventory
      name: "Order to Inventory Integration"
      upstream_context: bc_order_management
      downstream_context: bc_inventory
      relationship_type: customer_supplier
```

Tactical DDD (Aggregates):

```
bounded_context:
  id: bc_order_management

  aggregates:
    - id: agg_order
      name: Order
      root_ref: ent_order
      entities: [ent_order, ent_line_item]
      value_objects: [vo_money, vo_address]
      invariants:
        - "Order total must equal sum of line items"
        - "Order must have at least one line item to be submitted"

  domain_events:
    - id: evt_order_placed
      name: OrderPlaced
      aggregate_ref: agg_order
      data_attributes:
        - name: order_id
          type: uuid
        - name: customer_id
          type: uuid
        - name: total_amount
          type: money
```

4.6 Integration with Event Storming

Domain Storytelling and Event Storming are complementary discovery techniques.

Comparison:

Aspect	Domain Storytelling	Event Storming
Focus	Actor-centered narratives	Event-centered exploration
Structure	Sequential, linear	Temporal, chaotic then organized
Notation	Actor → Activity → Work Object	Events (orange), Commands (blue), Aggregates (yellow)
Facilitation	Guided storytelling	Big Picture, Process Modeling
Output	Stories with clear sequence	Timeline of Domain Events
Best For	Understanding processes and workflows	Discovering events and invariants
Workshop Energy	Calm, narrative	High-energy, post-it explosion

Combined Approach:

Phase 1: Domain Storytelling (Start here)
- Understand core business processes
- Identify actors, activities, work objects
- Surface ubiquitous language
- Create structured narratives

Phase 2: Event Storming (Build on stories)
- Identify ALL Domain Events from stories
- Add events not captured in stories
- Find hotspots (conflicts, questions)
- Identify Aggregates

Phase 3: Validation (Circle back)
- Use stories to validate event flows
- Ensure events support actual processes
- Refine both stories and event model

Example: Combined Use

Domain Story:

```
Customer submits Order
→ System validates Order
→ System reserves Inventory
→ System sends Order to Fulfillment
```

From Event Storming (add events missed in story):

```
OrderSubmitted
OrderValidated
InventoryReserved
InventoryReservationFailed ← Discovered in Event Storming!
FulfillmentRequestSent
```

Refined Story (incorporating Event Storming insights):

```
Customer submits Order
→ OrderSubmitted event
→ System validates Order
→ OrderValidated event
→ System attempts to reserve Inventory
  ├─ Success: InventoryReserved event
  └─ Failure: InventoryReservationFailed event → Order moves to
"Pending" state
```

4.7 Schema Reference

The `domain-stories-schema.yaml` formalizes Domain Storytelling artifacts for validation, tool support, and code generation. Use this schema when you need to capture domain stories in a machine-readable format, validate story completeness, or enable automated reasoning and code generation from domain knowledge.

This schema bridges the gap between workshop outputs (sticky notes and diagrams) and formal DDD implementation (aggregates, commands, events). It ensures consistency, enables tooling, and creates a single source of truth for domain knowledge.

Schema Purpose:
- **Validation**: Ensure story structure is correct
- **Tool Support**: Enable LLM reasoning about stories
- **Documentation**: Machine-readable domain knowledge
- **Code Generation**: Generate code from stories

Top-Level Structure

```
version: "2.0.0"
domain_stories:
  - domain_story_id: dst_checkout_process
    title: "E-commerce Checkout Process"
    description: "Complete flow from product selection to order
confirmation"

    actors: [...]           # Who performs actions
    work_objects: [...]     # What they work with
    commands: [...]         # State-changing requests
    queries: [...]          # Information requests
    activities: [...]       # Actions in sequence
    events: [...]           # Facts that happened
    policies: [...]         # Reactive rules

    # DDD Integration:
    aggregates: [...]       # Aggregate mappings
    repositories: [...]     # Repository references
    application_services: [...] # Service references
    domain_services: [...]  # Domain service references
    read_models: [...]      # Query model references
    business_rules: [...]   # Rule definitions
```

The top-level structure contains all story elements and their relationships. Each domain story includes actors, work objects, commands, queries, activities, events, and policies, along with optional DDD integration points like aggregates and services.

ID Conventions

All IDs follow consistent prefixes for clarity and type safety:

Concept	Prefix	Example
Domain Story	`dst_`	`dst_checkout_process`
Actor	`act_`	`act_customer`
Work Object	`wobj_`	`wobj_order`
Activity	`actv_`	`actv_submit_order`
Command	`cmd_`	`cmd_place_order`
Query	`qry_`	`qry_get_order_status`
Event	`evt_`	`evt_order_placed`
Policy	`pol_`	`pol_send_confirmation`
Business Rule	`rle_`	`rle_order_minimum`
Aggregate	`agg_`	`agg_order`
Repository	`repo_`	`repo_order`
Application Service	`svc_app_`	`svc_app_order_management`
Domain Service	`svc_dom_`	`svc_dom_pricing`
Read Model	`rmdl_`	`rmdl_order_summary`

These prefixes enable unambiguous references across the schema and support validation tooling.

Causal Chain in Schema

The schema captures complete causality from user actions through system reactions. This traceability is essential for understanding system behavior and debugging complex workflows.

```
Actor (act_customer)
  ↓ initiates
Command (cmd_submit_order)
  ↓ triggers
Activity (actv_submit_order)
  ↓ uses
Work Objects (wobj_shopping_cart, wobj_order)
  ↓ produces
Event (evt_order_submitted)
  ↓ triggers
Policy (pol_send_confirmation)
  ↓ issues
Command (cmd_send_email)
  ↓ (continues...)
```

This creates a traceable chain from actor intent to system reactions. Each element references related elements by ID, enabling validation and automated analysis.

Aggregate Integration

Work objects can reference aggregates, bridging discovery and tactical design:

```
work_objects:
  - work_object_id: wobj_order
    name: "Order"
    aggregate_id: agg_order  # Links to tactical model
    attributes:
      - name: order_id
        type: uuid
      - name: total
        type: money
```

This bridges discovery (domain stories) and tactical design (aggregates).

4.8 Best Practices

Story Discovery

✅ **DO:**

- Start with the happy path (most common, successful flow)
- Add variations incrementally after happy path is clear
- Use real examples from actual business operations
- Capture exact terminology experts use
- Note exceptions and edge cases
- Follow complete flows end-to-end

❌ **DON'T:**

- Jump to edge cases first (get the normal flow first)
- Use hypothetical scenarios ("what if aliens attacked?")
- Translate expert terminology into "better" terms
- Skip "obvious" steps (they're often critical)
- Ignore variations ("we usually do X" means sometimes you don't!)

Notation Usage

✅ **DO:**

- Keep notation simple (actor, activity, work object, arrows)
- Use color coding consistently (same color = same type)
- Show sequence clearly with numbers
- Label relationships and flows
- Capture both human and system actors

❌ **DON'T:**

- Over-complicate with UML or detailed diagrams
- Use technical notation experts don't understand
- Skip intermediate steps for "simplicity"
- Hide automated system actors

Workshop Dynamics

✅ **DO:**

- Let domain experts lead and tell the story
- Ask clarifying questions, don't assume
- Pause for deep discussions when terms are unclear
- Validate understanding by reading the story back
- Capture glossary terms as they emerge

❌ **DON'T:**

- Dominate the conversation as a developer
- Rush experts through their explanation
- Impose technical solutions during discovery
- Argue about terminology (if experts use it, capture it)
- Skip edge cases because "we'll handle that later"

Context Identification

✅ **DO:**

- Look for linguistic boundaries (same word, different meanings)
- Notice where expert terminology changes
- Identify organizational seams (different teams, different priorities)
- Find natural process boundaries

❌ **DON'T:**

- Force artificial boundaries based on technical architecture
- Create too many tiny contexts (over-fragmentation)
- Ignore obvious linguistic boundaries
- Let technical concerns drive context boundaries

4.9 Example: Complete Domain Story

Story: E-Commerce Order Placement

Happy Path Narrative:

```
1. Customer → browses → Product Catalog
2. Customer → selects → Product
3. Customer → adds to → Shopping Cart
4. Customer → updates quantity in → Shopping Cart
5. Customer → reviews → Shopping Cart
6. Customer → submits → Order
7. System → validates → Order (checks: items in stock, payment
method valid)
8. System → reserves → Inventory
9. System → requests → Payment Authorization from Payment System
10. Payment System → authorizes → Payment
11. System → confirms → Order
12. System → publishes → OrderPlaced event
13. System → sends → Order to Warehouse
14. System → sends → Confirmation Email to Customer
```

Alternative Flows:

Payment Fails:

```
9. System → requests → Payment Authorization
10. Payment System → declines → Payment
11. System → marks → Order as "Payment Failed"
12. System → sends → Payment Failure Notification to Customer
```

Inventory Unavailable:

```
8. System → checks → Inventory
9. System → detects → Insufficient Inventory
10. System → marks → Order as "Pending Inventory"
11. System → sends → Backorder Notification to Customer
```

Actors Identified:
- `act_customer` (Person)
- `act_system` (System - main application)
- `act_payment_system` (System - external)
- `act_warehouse` (System - fulfillment)

Work Objects Identified:

- `wobj_product`
- `wobj_product_catalog`
- `wobj_shopping_cart`
- `wobj_order`
- `wobj_inventory`
- `wobj_payment`
- `wobj_confirmation_email`

Commands:

- `cmd_add_to_cart`
- `cmd_submit_order`
- `cmd_authorize_payment`

Events:

- `evt_product_added_to_cart`
- `evt_order_submitted`
- `evt_order_validated`
- `evt_inventory_reserved`
- `evt_payment_authorized`
- `evt_order_placed`

Policies:

- `pol_send_order_confirmation` (When OrderPlaced, send confirmation email)
- `pol_notify_warehouse` (When OrderPlaced, send to warehouse)

Bounded Context Candidates:

- Sales Context (Shopping Cart, Order submission)
- Inventory Context (Stock management, reservations)
- Payment Context (Payment authorization, processing)
- Fulfillment Context (Warehouse operations, shipping)

Part II Summary

Part II introduced **Domain Storytelling** as the discovery technique that bridges business understanding and strategic design:

Section 4: Domain Storytelling
- **Collaborative technique**: Experts tell stories, developers capture them
- **Simple notation**: Actor → Activity → Work Object
- **Commands and Events**: CQRS concepts emerge naturally from stories
- **Policies**: Reactive rules connect events to actions
- **Workshop facilitation**: Structured process for knowledge discovery
- **Boundary discovery**: Stories reveal bounded context candidates
- **Integration with Event Storming**: Complementary techniques
- **Schema formalization**: Machine-readable domain stories

Key Insights:
- Discovery before design: Understand the domain through stories first
- Linguistic boundaries: Where terms change meaning, contexts emerge
- Causal chains: Actor → Command → Event → Policy creates traceable flows
- Ubiquitous language surfaces: Stories reveal the exact terms experts use

What's Next: With domain understanding from storytelling and strategic boundaries from bounded contexts, Part III dives into **Tactical Implementation**—how to build rich domain models using aggregates, entities, value objects, and the patterns that bring them to life.

Part III: Tactical Implementation

With strategic boundaries established and domain knowledge captured through storytelling, we now turn to **tactical patterns**—the building blocks for implementing rich domain models within bounded contexts.

5. Tactical Design Patterns

Tactical patterns focus on the internal structure of a bounded context. While strategic patterns address system organization and context boundaries, tactical patterns provide the vocabulary and techniques for expressing domain logic clearly and maintainably.

Key Principle: The domain layer should be isolated from infrastructure and application concerns, containing pure business logic expressed in the Ubiquitous Language.

5.1 Overview and Building Blocks

Purpose of Tactical Patterns:
- Structure domain logic clearly and explicitly
- Maintain business invariants consistently
- Express domain concepts in code
- Enable comprehensive testing
- Facilitate evolutionary change

The Tactical Pattern Catalog:

1. **Entity**: Object with identity and continuous lifecycle
2. **Value Object**: Immutable object defined by attributes
3. **Aggregate**: Consistency boundary clustering related objects
4. **Aggregate Root**: Entry point and guardian of aggregate
5. **Repository**: Abstraction for aggregate persistence
6. **Domain Service**: Stateless operations not belonging to entities
7. **Domain Event**: Fact about something that happened
8. **Factory**: Complex object creation logic

Pattern Relationships:

```
BoundedContext (Root Object)
    ↓ contains
Aggregates (consistency boundaries)
    ↓ composed of
Aggregate Root (Entity - entry point)
    ├─ Other Entities (identity-based objects)
    └─ Value Objects (attribute-based objects)
        ↓ persisted/retrieved by
Repository (one per aggregate)
    ↓ publishes
Domain Events (facts about what happened)
```

In the Schema:

The tactical schema uses **BoundedContext as the root object**, eliminating the need for `bounded_context_ref` on child types. All tactical elements exist within a bounded context:

```
bounded_context:
  id: bc_order_management
  name: "Order Management"
  domain_ref: dom_sales

  aggregates: [...]
  entities: [...]
  value_objects: [...]
  repositories: [...]
  domain_services: [...]
  application_services: [...]
  domain_events: [...]
```

With the tactical pattern catalog and relationships established, we now examine each pattern in detail, starting with Entities—objects defined by their unique identity and continuous lifecycle.

5.2 Entities

Definition: An **Entity** is an object defined by its identity rather than its attributes. It has a continuous lifecycle, and while its attributes may change, it remains the same entity throughout.

Core Concept: Identity

Identity is what makes an entity unique and distinguishes it from all others:

What is Identity?
- A unique identifier that persists throughout the entity's lifecycle
- Does not change even if all other attributes change
- Used for equality comparison (not attribute values)
- Immutable once assigned

Identity Generation Strategies:

1. **User-Provided**: Email address, username, SSN (when truly unique)
2. **Auto-Generated**: UUID, GUID (preferred for most cases)
3. **Database Sequence**: Auto-increment ID (coupling to infrastructure)
4. **Derived**: Combination of attributes (composite key)
5. **External**: Provided by external system

Best Practice: Use UUIDs or GUIDs for domain entity identity. They're globally unique, don't require database coordination, and work well in distributed systems.

Example:

```
public class Customer {
    private final CustomerId id;  // Immutable identity (Value
Object wrapping UUID)
    private PersonName name;      // Mutable attribute
    private Email email;          // Mutable attribute
    private Address shippingAddress;
    private CustomerStatus status;

    // Constructor enforces identity immutability
    public Customer(CustomerId id, PersonName name, Email email) {
        this.id = requireNonNull(id, "Customer ID required");
        this.name = requireNonNull(name, "Name required");
        this.email = requireNonNull(email, "Email required");
        this.status = CustomerStatus.ACTIVE;
    }

    // Identity-based equality
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Customer)) return false;
        Customer other = (Customer) o;
        return id.equals(other.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }
}
```

Entity Lifecycle

Creation:
- Entity comes into existence
- Identity assigned (must be unique)
- Initial state established
- Invariants must be satisfied from the start

Modification:
- Attributes change over time
- Identity remains constant
- Invariants maintained at all times
- Changes may trigger Domain Events

Deletion/Archival:
- Entity no longer active in the system
- Often soft-deleted (marked inactive) rather than hard-deleted
- Identity preserved for historical reference and audit trails

Design Guidelines for Entities

1. Model Identity Explicitly

Use a Value Object to represent identity (not primitive strings):

```
// ✓ Good: Type-safe identity
public class Order {
    private final OrderId id;  // Value Object
}

// ❌ Bad: Primitive obsession
public class Order {
    private final String id;  // Just a string
}
```

2. Enforce Invariants Through Methods

```
public class Order {
    private OrderStatus status;
    private List<OrderLine> lines;

    // Business method enforcing invariants
    public void submit() {
        if (lines.isEmpty()) {
            throw new EmptyOrderException("Cannot submit order
without items");
        }
        if (status != OrderStatus.DRAFT) {
            throw new
OrderNotModifiableException("Can only submit draft orders");
        }
        this.status = OrderStatus.SUBMITTED;
        DomainEvents.raise(new OrderSubmitted(this.id));
    }

    // ❌ Bad: Public setter breaks encapsulation
    // public void setStatus(OrderStatus status) { ... }
}
```

3. Use Value Objects for Attributes

Prefer value objects over primitives to make domain concepts explicit:

```
// ✓ Good: Value objects make concepts explicit
private Email email;
private Money totalAmount;
private Address shippingAddress;

// ❌ Bad: Primitives hide domain meaning
private String email;
private double amount;
private String currency;
```

4. Intention-Revealing Method Names

Use names from ubiquitous language:

```
// ✓ Good: Reveals business intent
customer.suspend();
order.cancel();
product.discontinue();

// ❌ Bad: Generic technical names
customer.setStatus(Status.SUSPENDED);
order.update(Map.of("status", "cancelled"));
```

Entity vs Value Object Decision

Choose Entity when:
- ✓ Object must be tracked over time
- ✓ Object has a unique identity
- ✓ Object's history matters (audit trail, lifecycle events)
- ✓ Two objects with identical attributes are still different
- ✓ Object participates in relationships with other entities

Choose Value Object when:
- ✓ Only the attributes matter (not which specific instance)
- ✓ Two objects with same attributes are interchangeable
- ✓ Immutability is natural for the concept
- ✓ No lifecycle to track
- ✓ Represents a measurement, quantity, or description

Example: Two customers with the same name are different people (Entity). Two addresses with the same values are the same address (Value Object).

5.3 Value Objects

Definition: A **Value Object** is an immutable object defined entirely by its attributes. Two value objects with the same attribute values are considered equal and completely interchangeable.

Core Characteristics

1. Immutability

Value objects cannot change after creation:

```
public final class Money {
    private final BigDecimal amount;
    private final Currency currency;

    public Money(BigDecimal amount, Currency currency) {
        // Validation in constructor
        this.amount = requireNonNull(amount);
        this.currency = requireNonNull(currency);
    }

    // No setters!

    // Operations return NEW instances
    public Money add(Money other) {
        validateSameCurrency(other);
        return new Money(
            this.amount.add(other.amount),
            this.currency
        );
    }
}
```

2. Attribute-Based Equality

Equals and hashCode based on attribute values:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Money)) return false;
    Money money = (Money) o;
    return amount.equals(money.amount) &&
           currency.equals(money.currency);
}

@Override
public int hashCode() {
    return Objects.hash(amount, currency);
}
```

3. Self-Validation

Constructor validates all invariants—invalid state cannot exist:

```
public final class Email {
    private final String value;

    public Email(String value) {
        if (value == null || value.isBlank()) {
            throw new IllegalArgumentException("Email cannot be
blank");
        }
        if (!value.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
            throw new InvalidEmailException(value);
        }
        this.value = value.toLowerCase().trim();
    }
}
```

4. Side-Effect-Free Behavior

Methods don't modify state, they return new instances:

```
public final class DateRange {
    private final LocalDate start;
    private final LocalDate end;

    // Returns boolean, doesn't modify state
    public boolean contains(LocalDate date) {
        return !date.isBefore(start) && !date.isAfter(end);
    }

    // Returns NEW DateRange, doesn't modify this one
    public DateRange extendBy(int days) {
        return new DateRange(this.start, this.end.plusDays(days));
    }
}
```

Immutability Enforcement in the Schema

Value Objects must be immutable:

```
ValueObject:
  properties:
    immutability:
      type: boolean
      const: true  # MUST be true (required)
```

Similarly for Domain Events (which are conceptually value objects representing facts):

```
DomainEvent:
  properties:
    immutable:
      type: boolean
      const: true
```

This schema-level enforcement prevents accidentally creating mutable value objects.

When to Use Value Objects

Value Objects are appropriate for:

✓ **Measurements**: Money, Weight, Temperature, Distance
✓ **Quantities**: Quantity with unit, Percentage, Ratio
✓ **Ranges**: DateRange, AgeRange, PriceRange
✓ **Descriptive Aspects**: PersonName, Address, Email, PhoneNumber
✓ **Complex Calculations**: Interest Rate, Exchange Rate
✓ **Specifications**: ProductSpecification, SearchCriteria
✓ **Domain Primitives**: Wrapping primitives with domain meaning

Benefits of Value Objects:
- Make domain concepts explicit in code
- Encapsulate validation logic
- Provide type safety
- Enable reusability across aggregates
- Simplify testing (pure functions)
- Thread-safe by nature (immutable)

Common Value Object Patterns

Money (Amount + Currency):

```
public final class Money {
    private final BigDecimal amount;
    private final Currency currency;

    public Money add(Money other) {
        if (!currency.equals(other.currency)) {
            throw new CurrencyMismatchException();
        }
        return new Money(amount.add(other.amount), currency);
    }

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

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

Address (Structured Location):

```
public final class Address {
    private final String street;
    private final String city;
    private final String postalCode;
    private final Country country;

    // Constructor validates all fields
    public Address(String street, String city, String postalCode,
Country country) {
        this.street = requireNonBlank(street, "Street");
        this.city = requireNonBlank(city, "City");
        this.postalCode = requireValidPostalCode(postalCode,
country);
        this.country = requireNonNull(country, "Country");
    }

    // Convenience method for partial updates
    public Address withStreet(String newStreet) {
        return new Address(newStreet, this.city, this.postalCode,
this.country);
    }
}
```

PersonName (Structured Name):

```
public final class PersonName {
    private final String firstName;
    private final String lastName;
    private final String middleName;

    public String fullName() {
        return middleName != null
            ? firstName + " " + middleName + " " + lastName
            : firstName + " " + lastName;
    }

    public String formalName() {
        return lastName + ", " + firstName;
    }
}
```

Replaceability

Since value objects are immutable, to "change" them you replace the entire instance:

```
public class Customer {  // Entity
    private Email email;  // Value Object

    // To change email, replace the entire Value Object
    public void changeEmail(Email newEmail) {
        if (this.email.equals(newEmail)) {
            return;  // No change needed
        }
        this.email = newEmail;  // Complete replacement
        DomainEvents.raise(new CustomerEmailChanged(this.id,
newEmail));
    }
}
```

Common Mistakes

1. **Mutable Value Objects**: Adding setters or allowing state modification
2. **Reference Equality**: Using `==` instead of `.equals()`

3. **Missing Validation**: Not validating invariants in constructor
4. **Primitive Obsession**: Not creating value objects for domain concepts
5. **Too Large**: Creating value objects that are really entities in disguise

5.4 Aggregates

Definition: An **Aggregate** is a cluster of associated entities and value objects treated as a single unit for data changes. It has a root entity (the Aggregate Root) and a boundary defining what's inside.

Purpose: Consistency Boundary

Aggregates solve the fundamental challenge: **How do we maintain business invariants in a complex domain model?**

Key Insights:
- Aggregates define **transactional boundaries**—all changes within an aggregate happen in a single transaction
- Aggregates ensure **business invariants**—rules that must always be true
- Aggregates are the **atomic unit of persistence**—you save/load entire aggregates, not individual entities

Vaughn Vernon's Aggregate Design Rules

Rule 1: Protect True Invariants in Consistency Boundaries

True Invariants are business rules that must be consistent at all times:

```
public class Order {  // Aggregate Root
    private OrderId id;
    private List<OrderLine> lines;
    private Money total;

    // TRUE INVARIANT: Total must equal sum of line items
    public void addLine(Product product, Quantity qty, Money
price) {
        OrderLine line = new OrderLine(product, qty, price);
        lines.add(line);
        recalculateTotal();  // Maintain invariant immediately
    }

    private void recalculateTotal() {
        this.total = lines.stream()
            .map(OrderLine::subtotal)
            .reduce(Money.ZERO, Money::add);
    }

    // Invariant is ALWAYS true - cannot violate
}
```

Eventual Consistency is acceptable for invariants that span aggregates:

```
// Customer's lifetime value across ALL orders
// This is EVENTUALLY consistent, not immediately
// Each Order aggregate maintains its own consistency
// Customer aggregate's totalSpent updated asynchronously
```

Rule 2: Design Small Aggregates

Prefer small aggregates with only the entities and value objects needed to enforce invariants.

Why?

- Smaller transaction scope (less locking)
- Better performance (less to load/save)
- Reduced merge conflicts
- Clearer responsibility

Example:

```
# ✓ Good: Small, focused aggregate
aggregates:
  - id: agg_order
    root_ref: ent_order
    entities: [ent_order, ent_order_line]  # Small cluster
    invariants:
      - "Order total equals sum of line items"

# ❌ Bad: Large aggregate spanning too much
aggregates:
  - id: agg_customer
    entities:
      - ent_customer
      - ent_order  # Shouldn't be inside customer aggregate
      - ent_order_line
      - ent_shipment
      - ent_payment
    # Too large! Customer and Order should be separate aggregates
```

Rule 3: Reference Other Aggregates by Identity Only

Aggregates should reference each other by ID, not hold direct object references:

```
// ✓ Good: Reference by ID
public class Order {  // Aggregate Root
    private OrderId id;
    private CustomerId customerId;  // ID reference to Customer
aggregate
    private List<OrderLine> lines;
}

// ❌ Bad: Direct object reference
public class Order {
    private OrderId id;
    private Customer customer;  // Direct reference crosses
aggregate boundary!
}
```

Why?
- Clear aggregate boundaries
- Prevents accidentally modifying other aggregates
- Enables lazy loading
- Supports eventual consistency between aggregates

To access related aggregate:

```
// Load related aggregate through repository
Customer customer =
customerRepository.findById(order.getCustomerId());
```

Rule 4: Update Other Aggregates via Domain Events

Use domain events (eventual consistency) rather than trying to update multiple aggregates in one transaction:

```
public class Order {
    public void submit() {
        if (lines.isEmpty()) {
            throw new EmptyOrderException();
        }
        this.status = OrderStatus.SUBMITTED;

        // Publish event - other aggregates react asynchronously
        DomainEvents.raise(new OrderSubmitted(
            this.id,
            this.customerId,
            this.total
        ));
    }
}

// Event handler updates Customer aggregate separately
@EventHandler
public void onOrderSubmitted(OrderSubmitted event) {
    Customer customer =
customerRepository.findById(event.customerId());
    customer.recordOrderPlaced(event.total());
    customerRepository.save(customer);
}
```

Aggregate Root

The **Aggregate Root** is the only entity within the aggregate that external objects may hold references to.

Responsibilities:
- Entry point for all operations on the aggregate
- Enforces all invariants within the aggregate
- Coordinates changes to internal entities
- Publishes Domain Events
- Controls access to internals

Example:

```
public class Order {  // Aggregate Root
    private OrderId id;
    private List<OrderLine> lines;  // Internal entities
    private Money total;

    // ONLY way to add lines is through root
    public void addLine(ProductId productId, Quantity qty, Money
price) {
        // Root enforces invariants
        if (status != OrderStatus.DRAFT) {
            throw new OrderNotModifiableException();
        }

        // Root coordinates internal changes
        OrderLine line = new OrderLine(productId, qty, price);
        lines.add(line);
        recalculateTotal();

        // Root publishes events
        DomainEvents.raise(new OrderLineAdded(this.id,
line.id()));
    }

    // Internal entities NOT accessible directly
    // ❌ public List<OrderLine> getLines() - violates
encapsulation

    // ✓ Expose safe, read-only view
    public int lineCount() {
        return lines.size();
    }
}
```

One Aggregate Per Transaction

Critical Rule: Modify only ONE aggregate per transaction.

This is captured in the `TransactionBoundary` pattern:

```
TransactionBoundary:
  properties:
    modifies_aggregates:
      type: array
      items: { $ref: "#/$defs/AggId" }
      maxItems: 1  # Enforces one-aggregate-per-transaction rule
```

Why?
- Prevents distributed transaction complexity
- Ensures clear consistency boundaries
- Enables horizontal scaling
- Simplifies error handling

If you need to modify multiple aggregates:

Use domain events and eventual consistency:

```
// ✓ Good: One aggregate per transaction
@Transactional
public void submitOrder(OrderId orderId) {
    Order order = orderRepository.findById(orderId);
    order.submit();  // Modifies ONLY Order aggregate
    orderRepository.save(order);
    // Event published: OrderSubmitted
}

// Event handler runs in SEPARATE transaction
@EventHandler
@Transactional
public void onOrderSubmitted(OrderSubmitted event) {
    Inventory inventory =
inventoryRepository.findById(event.productId());
    inventory.reserve(event.quantity());  // Modifies ONLY
Inventory aggregate
    inventoryRepository.save(inventory);
}
```

Identifying Aggregates

From domain stories, look for:

1. **Work objects that always change together** → Likely one aggregate
   - Order and OrderLine always change together → Order aggregate
   - ShoppingCart and CartItem always change together → ShoppingCart aggregate
2. **Entry points in stories** → Likely aggregate roots
   - "Customer submits Order" → Order is aggregate root
   - "System creates Shipment" → Shipment is aggregate root
3. **Transactional boundaries in business rules**
   - "When Order is submitted, total must equal line items" → Order aggregate
   - "Inventory reservation is atomic" → Inventory aggregate

Aggregate Examples

```
# Order Aggregate
aggregates:
  - id: agg_order
    name: Order
    root_ref: ent_order
    entities: [ent_order, ent_order_line]
    value_objects: [vo_money, vo_address, vo_order_status]
    invariants:
      - "Order total must equal sum of all line items"
      - "Order must have at least one line item when submitted"
      - "Order cannot be modified after submission"
    domain_events:
      - evt_order_submitted
      - evt_order_cancelled
      - evt_order_line_added

# Inventory Aggregate
aggregates:
  - id: agg_inventory
    name: Inventory
    root_ref: ent_inventory_item
    entities: [ent_inventory_item]
    value_objects: [vo_quantity, vo_warehouse_location]
    invariants:
      - "Quantity on hand cannot be negative"
      - "Reserved quantity cannot exceed quantity on hand"
    domain_events:
      - evt_inventory_reserved
      - evt_inventory_restocked
```

5.5 Repositories

Definition: A **Repository** is an abstraction that provides the illusion of an in-memory collection of aggregates, hiding all persistence details.

Purpose

Repositories serve as the boundary between the domain model and data persistence:

Key Responsibilities:
- Load aggregates by identity
- Save aggregates (create or update)
- Query for aggregates matching criteria
- Hide all database/ORM details from domain layer

One Repository Per Aggregate

Critical rule: Create one repository for each aggregate (not for every entity):

```
// ✓ Good: One repository per aggregate root
OrderRepository  // For Order aggregate
CustomerRepository  // For Customer aggregate
InventoryRepository  // For Inventory aggregate

// ❌ Bad: Repositories for internal entities
OrderLineRepository  // No! OrderLine is internal to Order
aggregate
```

Repository Interface Pattern

Define repository as interface in domain layer, implement in infrastructure:

```
// Domain layer - interface
public interface OrderRepository {
    OrderId nextId();
    Order findById(OrderId id);
    List<Order> findByCustomer(CustomerId customerId);
    void save(Order order);
    void remove(Order order);
}

// Infrastructure layer - implementation
public class JpaOrderRepository implements OrderRepository {
    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public Order findById(OrderId id) {
        // JPA/Hibernate details hidden from domain
        return entityManager.find(OrderEntity.class, id.value())
            .toDomainModel();
    }

    @Override
    public void save(Order order) {
        // Persistence details encapsulated
        OrderEntity entity = OrderEntity.fromDomain(order);
        entityManager.merge(entity);
    }
}
```

Repository Methods

Standard Methods:

```
public interface Repository<T, ID> {
    // Generate next identity
    ID nextId();

    // Retrieve by identity
    T findById(ID id);
    Optional<T> findByIdOptional(ID id);

    // Save (create or update)
    void save(T aggregate);

    // Remove
    void remove(T aggregate);
}
```

Query Methods:

```
public interface OrderRepository extends Repository<Order,
OrderId> {
    // Find by specific criteria
    List<Order> findByCustomer(CustomerId customerId);
    List<Order> findByStatus(OrderStatus status);
    List<Order> findSubmittedBetween(LocalDate start, LocalDate
end);

    // Count
    long countByCustomer(CustomerId customerId);

    // Existence check
    boolean existsById(OrderId id);
}
```

Best Practices:

1. **Return aggregates, not DTOs**: Repository returns full domain objects
2. **Avoid lazy loading across aggregates**: Load complete aggregate or use references
3. **Use specifications for complex queries**: Encapsulate query logic

4. **Consider read models for queries**: CQRS pattern for complex reads

In the Schema:

```
repositories:
  - id: repo_order
    name: "OrderRepository"
    aggregate_ref: agg_order
    operations:
      - name: "findById"
        parameters:
          - name: "id"
            type: "ref"
            ref_id: "ent_order"
        return_type: "ref"
        return_ref_id: "agg_order"

      - name: "save"
        parameters:
          - name: "order"
            type: "ref"
            ref_id: "agg_order"
        return_type: "void"

      - name: "findByCustomer"
        parameters:
          - name: "customerId"
            type: "uuid"
        return_type: "list"
```

5.6 Domain Services

Definition: A **Domain Service** is a stateless operation that doesn't naturally belong to an entity or value object but is part of the domain.

When to Use Domain Services

Use a domain service when:
- ✓ Operation involves multiple aggregates
- ✓ Operation doesn't naturally belong to any one entity
- ✓ Operation is a significant domain concept (in ubiquitous language)
- ✓ Operation would make an entity or value object awkward

Examples of Domain Services:

```
// Pricing calculation involving multiple factors
public interface PricingService {
    Money calculatePrice(Product product, Customer customer,
Quantity quantity);
}

// Transfer between accounts (involves two aggregates)
public interface FundsTransferService {
    void transfer(AccountId from, AccountId to, Money amount);
}

// Availability check across inventory locations
public interface AvailabilityService {
    boolean isAvailable(ProductId productId, Quantity requested);
}
```

Domain Service vs Application Service

Aspect	Domain Service	Application Service
Layer	Domain layer	Application layer
Concerns	Pure business logic	Use case orchestration
Dependencies	Other domain objects	Domain services, repositories, infrastructure
State	Stateless	Stateless
Transactions	No transaction mgmt	Manages transactions
Example	`PricingService.calculatePrice()`	`OrderService.submitOrder()`

Implementation:

```
// Domain Service - pure business logic
public class PricingService {
    public Money calculatePrice(
        Product product,
        Customer customer,
        Quantity quantity
    ) {
        Money basePrice = product.price();

        // Volume discount
        Money volumeDiscounted = applyVolumeDiscount(basePrice,
quantity);

        // Customer tier discount
        Money customerDiscounted = applyCustomerDiscount(
            volumeDiscounted,
            customer.tier()
        );

        return customerDiscounted;
    }

    private Money applyVolumeDiscount(Money price, Quantity qty) {
        if (qty.isGreaterThan(Quantity.of(100))) {
            return price.multiply(new BigDecimal("0.9"));  // 10%
off
        }
        return price;
    }
}
```

In the Schema:

```
domain_services:
  - id: svc_dom_pricing
    name: "PricingService"
    description: "Calculates product pricing with discounts"
    stateless: true
    operations:
      - name: "calculatePrice"
        parameters:
          - name: "product"
            type: "ref"
            ref_id: "ent_product"
          - name: "customer"
            type: "ref"
            ref_id: "ent_customer"
          - name: "quantity"
            type: "ref"
            ref_id: "vo_quantity"
        return_type: "ref"
        return_ref_id: "vo_money"
```

5.7 Domain Events

Definition: A **Domain Event** is a record of something significant that happened in the domain. It represents a fact—something that occurred in the past.

Characteristics:

1. **Past Tense**: Event names describe what happened
   - `OrderSubmitted` (not `SubmitOrder`)
   - `PaymentAuthorized` (not `AuthorizePayment`)
   - `InventoryRestocked` (not `RestockInventory`)
2. **Immutable**: Events are historical facts that cannot change
   - Design requires: `immutable: const: true`
3. **Include Relevant Data**: Carry information about what happened
   - Who, what, when, and relevant context

Purpose of Domain Events:

1. **Decouple aggregates**: Aggregates communicate via events rather than direct calls
2. **Audit trail**: Events provide complete history of what happened
3. **Eventual consistency**: Enable updates across aggregate boundaries
4. **Integration**: External systems subscribe to events
5. **Event sourcing**: Events can be the source of truth

Publishing Events from Aggregates:

```
public class Order {
    private OrderId id;
    private OrderStatus status;
    private List<DomainEvent> uncommittedEvents = new
ArrayList<>();

    public void submit() {
        // Validate invariants
        if (lines.isEmpty()) {
            throw new EmptyOrderException();
        }

        // Change state
        this.status = OrderStatus.SUBMITTED;
        this.submittedAt = LocalDateTime.now();

        // Record event
        addEvent(new OrderSubmitted(
            this.id,
            this.customerId,
            this.total,
            LocalDateTime.now()
        ));
    }

    private void addEvent(DomainEvent event) {
        uncommittedEvents.add(event);
    }

    public List<DomainEvent> getUncommittedEvents() {
        return Collections.unmodifiableList(uncommittedEvents);
    }

    public void clearEvents() {
        uncommittedEvents.clear();
    }
}
```

Event Structure:

```
public class OrderSubmitted implements DomainEvent {
    private final OrderId orderId;
    private final CustomerId customerId;
    private final Money totalAmount;
    private final LocalDateTime occurredAt;

    public OrderSubmitted(OrderId orderId, CustomerId customerId,
                          Money totalAmount, LocalDateTime
occurredAt) {
        this.orderId = requireNonNull(orderId);
        this.customerId = requireNonNull(customerId);
        this.totalAmount = requireNonNull(totalAmount);
        this.occurredAt = requireNonNull(occurredAt);
    }

    // Getters only - immutable
    public OrderId orderId() { return orderId; }
    public CustomerId customerId() { return customerId; }
    public Money totalAmount() { return totalAmount; }
    public LocalDateTime occurredAt() { return occurredAt; }
}
```

In the Schema:

```
domain_events:
  - id: evt_order_submitted
    name: "OrderSubmitted"
    description: "Raised when customer submits an order"
    aggregate_ref: agg_order
    immutable: true  # Enforced by schema
    data_attributes:
      - name: "order_id"
        type: "uuid"
        required: true
      - name: "customer_id"
        type: "uuid"
        required: true
      - name: "total_amount"
        type: "money"
        required: true
      - name: "occurred_at"
        type: "datetime"
        required: true
    triggered_by:
      command_id: "cmd_submit_order"
```

Event Handlers:

```
@Component
public class OrderEventHandlers {
    private final InventoryService inventoryService;
    private final CustomerRepository customerRepository;

    @EventHandler
    @Transactional
    public void onOrderSubmitted(OrderSubmitted event) {
        // Reserve inventory (separate transaction, separate
aggregate)
        inventoryService.reserve(event.orderId(), event.items());
    }

    @EventHandler
    @Transactional
    public void updateCustomerStatistics(OrderSubmitted event) {
        // Update customer aggregate (separate transaction)
        Customer customer =
customerRepository.findById(event.customerId());
        customer.recordOrderPlaced(event.totalAmount());
        customerRepository.save(customer);
    }
}
```

5.8 ID Types and Conventions

The schemas define explicit ID type patterns for all tactical elements:

ID Prefixes:

Type	Prefix	Pattern	Example
Bounded Context	`bc_`	`^bc_[a-z0-9_]+$`	`bc_order_management`
Aggregate	`agg_`	`^agg_[a-z0-9_]+$`	`agg_order`
Entity	`ent_`	`^ent_[a-z0-9_]+$`	`ent_customer`
Value Object	`vo_`	`^vo_[a-z0-9_]+$`	`vo_money`
Repository	`repo_`	`^repo_[a-z0-9_]+$`	`repo_order`
Domain Service	`svc_dom_`	`^svc_dom_[a-z0-9_]+$`	`svc_dom_pricing`
Application Service	`svc_app_`	`^svc_app_[a-z0-9_]+$`	`svc_app_order_mgmt`
Command	`cmd_`	`^cmd_[a-z0-9_]+$`	`cmd_submit_order`
Query	`qry_`	`^qry_[a-z0-9_]+$`	`qry_get_order`
Domain Event	`evt_`	`^evt_[a-z0-9_]+$`	`evt_order_submitted`

Benefits:

- Clear identification of element types
- Enable tooling and code generation
- Consistent naming across schemas
- LLM-friendly patterns

---

6. Application Layer

6.1 Overview

The **Application Layer** is a critical architectural layer in Domain-Driven Design that sits between the User Interface and the Domain Layer. Its primary responsibility is to orchestrate use case execution without containing business logic.

Eric Evans' Definition:

> "The application layer is responsible for driving the workflow of the application, coordinating the domain objects to perform the actual work."

Martin Fowler's Service Layer Definition:

> "Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation."

Key Characteristics:

Characteristic	Description
Stateless	Holds no domain state between operations
Thin	Contains no business logic, only orchestration
Coordinates	Orchestrates domain objects and domain services
Transaction Boundary	Manages database transactions
Use Case Focused	One operation per use case
External Interface	API exposed to external clients (UI, BFF, API)

Alternative Names:
- **Application Layer** (Eric Evans, *DDD Blue Book*)
- **Service Layer** (Martin Fowler, *PoEAA*)
- **Use Case Layer** (Clean Architecture)
- **Command/Query Handlers** (CQRS architecture)

Core Principle: Application Services contain **NO business logic**—only orchestration logic. Business logic belongs exclusively in the Domain Layer.

Having established the Application Layer's purpose as a stateless orchestrator, let's examine its precise position in the four-layer architecture and how it coordinates between the UI and Domain layers.

6.2 Application Layer Position

Four-Layer Architecture

Eric Evans defines a four-layer architecture with the Application Layer positioned between UI and Domain:

```
┌──────────────────────────────────────────────────────┐
│   USER INTERFACE LAYER                               │
│   (Presentation, UI Controls, REST APIs, BFFs)       │
├──────────────────────────────────────────────────────┤
│   APPLICATION LAYER ← THIS LAYER                     │
│   (Application Services, Use Case Orchestration)     │
├──────────────────────────────────────────────────────┤
│   DOMAIN LAYER                                       │
│   (Entities, Value Objects, Aggregates,              │
│    Domain Services, Repositories)                    │
├──────────────────────────────────────────────────────┤
│   INFRASTRUCTURE LAYER                               │
│   (Persistence, External Services, Messaging)        │
└──────────────────────────────────────────────────────┘
```

Dependency Direction

Dependencies flow in **one direction**:

```
UI Layer → Application Layer → Domain Layer
```

- User Interface depends on Application Layer
- Application Layer depends on Domain Layer
- Domain Layer is independent (core of the system)
- Infrastructure implements interfaces defined in Domain/Application layers

Dependency Inversion Principle:

The Application Layer defines repository interfaces, and the Infrastructure Layer provides implementations:

```
// Application Layer: Defines interface
public interface UserRepository {
    void save(User user);
    Optional<User> findById(UserId userId);
}

// Infrastructure Layer: Implements interface
@Repository
public class JpaUserRepository implements UserRepository {
    // JPA implementation
}
```

Application Layer Responsibilities

What the Application Layer DOES:
- Defines jobs the software is supposed to do
- Directs domain objects to work out problems
- Coordinates domain layer objects to perform actual work
- Manages transaction boundaries
- Publishes domain events
- Performs security and authorization checks
- Validates input format and required fields
- Orchestrates multi-step workflows
- Converts domain objects to DTOs for return

What the Application Layer DOES NOT DO:
- Contain business logic (belongs in Domain Layer)
- Make business decisions (delegates to domain)
- Directly access infrastructure (uses interfaces)
- Maintain state between operations
- Implement domain rules (delegates to aggregates)

6.3 Application Service Pattern

What is an Application Service?

An **Application Service** is a stateless object that orchestrates use case execution by:

1. Fetching domain objects from repositories
2. Executing domain operations on aggregates
3. Persisting changes back to repositories
4. Managing transaction boundaries
5. Publishing domain events to external systems

Vaughn Vernon's Guidance:

> "Application Services are the direct clients of the domain model and remain lightweight, coordinating operations performed against domain objects. Application Services should be kept thin, using them only to coordinate tasks on the model."

Application Service vs Domain Service

This distinction is critical in DDD and frequently misunderstood:

Aspect	Application Service	Domain Service
Business Logic	None	Contains domain logic
Transaction Management	Yes	No
Repository Access	Yes	No
External Dependencies	Yes (repos, infrastructure)	No
Parameters/ Return Types	DTOs, primitives	Domain objects
Called By	UI, API controllers, BFFs	Application services, other do
Validation	Input format validation	Business rule validation
Ubiquitous Language	Use case names	Domain concept names
Example	`UserApplicationService.createUser()`	`UserAuthenticationService`

Application Service Structure (Knight Pattern)

The Knight codebase demonstrates an elegant pattern using nested command records:

Commands Interface:

```
public interface UserCommands {

    UserId createUser(CreateUserCmd cmd);

    record CreateUserCmd(
        String email,
        String userType,
        String identityProvider,
        ClientId clientId
    ) {}

    void activateUser(ActivateUserCmd cmd);

    record ActivateUserCmd(UserId userId) {}

    void deactivateUser(DeactivateUserCmd cmd);

    record DeactivateUserCmd(UserId userId, String reason) {}
}
```

Queries Interface:

```
public interface UserQueries {

    record UserSummary(
        String userId,
        String email,
        String status,
        String userType,
        String identityProvider
    ) {}

    UserSummary getUserSummary(UserId userId);
}
```

Application Service Implementation:

```
@Singleton
public class UserApplicationService implements UserCommands,
UserQueries {

    private final UserRepository repository;
    private final ApplicationEventPublisher<Object>
eventPublisher;

    @Override
    @Transactional
    public UserId createUser(CreateUserCmd cmd) {
        // 1. Generate domain ID
        UserId userId = UserId.of(UUID.randomUUID().toString());

        // 2. Parse enums (input validation)
        User.UserType userType =
User.UserType.valueOf(cmd.userType());
        User.IdentityProvider identityProvider =
            User.IdentityProvider.valueOf(cmd.identityProvider());

        // 3. Create aggregate (business logic in aggregate)
        User user = User.create(
            userId,
            cmd.email(),
            userType,
            identityProvider,
            cmd.clientId()
        );

        // 4. Save aggregate
        repository.save(user);

        // 5. Publish event (after persistence)
        eventPublisher.publishEvent(new UserCreated(
            userId.id(),
            cmd.email(),
            cmd.userType(),
            cmd.identityProvider(),
            Instant.now()
        ));
```

```
        // 6. Return domain ID
        return userId;
    }

    @Override
    @Transactional
    public void activateUser(ActivateUserCmd cmd) {
        // 1. Load aggregate
        User user = repository.findById(cmd.userId())
            .orElseThrow(() -> new IllegalArgumentException(
                "User not found: " + cmd.userId().id()));

        // 2. Execute domain operation
        user.activate();

        // 3. Save aggregate
        repository.save(user);

        // 4. Publish event
        eventPublisher.publishEvent(new UserActivated(
            user.getUserId().id(),
            Instant.now()
        ));
    }

    @Override
    public UserSummary getUserSummary(UserId userId) {
        // 1. Load aggregate
        User user = repository.findById(userId)
            .orElseThrow(() -> new IllegalArgumentException(
                "User not found: " + userId.id()));

        // 2. Map domain aggregate to DTO
        return new UserSummary(
            user.getUserId().id(),
            user.getEmail(),
            user.getStatus().name(),
            user.getUserType().name(),
            user.getIdentityProvider().name()
        );
```

```
    }
}
```

Key Benefits:
- Immutable command objects (records are final)
- Type-safe parameters
- Clear intent (one command per operation)
- Minimal boilerplate
- Grouped by aggregate root

Coordination vs Business Logic

This distinction is the most critical aspect of Application Services and the most commonly violated principle.

Comparison Table:

Aspect	Coordination Logic (Application)	Business Logic (Domain)
Responsibility	Load, save, orchestrate	Calculate, validate, enforce rules
Location	Application Service	Aggregate, Domain Service
Examples	Fetch from repo, publish events	Pricing, invariants, state transitions
Dependencies	Repositories, infrastructure	Pure domain objects

Example of Proper Coordination:

```
@Transactional
public void activateCustomer(ActivateCustomerCmd cmd) {
    // 1. Authorization, 2. Load aggregate
    if (!authService.hasPermission(currentUser,
Permission.ACTIVATE_CUSTOMER)) {
        throw new UnauthorizedException();
    }
    Customer customer =
customerRepository.findById(cmd.customerId())
        .orElseThrow(() -> new
CustomerNotFoundException(cmd.customerId()));

    // 3. Invoke domain operation (business logic in domain)
    customer.activate();  // <-- Business logic is HERE

    // 4. Persist, 5. Publish event
    customerRepository.save(customer);
    eventPublisher.publish(new
CustomerActivated(customer.getId(), Instant.now()));
}
```

Anti-Pattern: Business Logic in Application Service

```
@Transactional
public void placeOrder(PlaceOrderCmd cmd) {
    Order order = orderRepository.findById(cmd.orderId());

    // WRONG: Calculations in application service
    BigDecimal total = BigDecimal.ZERO;
    for (OrderItem item : cmd.items()) {
        // ... discount logic
    }
    order.setTotal(total);  // Anemic domain model
    orderRepository.save(order);
}
```

Correct Pattern: Business Logic in Domain

```
// Application Service (thin coordination)
@Transactional
public void placeOrder(PlaceOrderCmd cmd) {
    Order order = orderRepository.findById(cmd.orderId());
    order.addItems(cmd.items());  // Business logic encapsulated
    orderRepository.save(order);
}

// Domain Aggregate (rich with business logic)
public class Order {
    public void addItems(List<OrderItem> items) {
        items.forEach(item -> {
            validateItem(item);
            applyPricingRules(item);
            // ... other business logic
        });
        this.totalAmount = calculateTotal();
    }
}
```

Granularity and Operations

Principle: One application service method = One use case

Good Examples (Fine-Grained, Use Case Aligned):

```
public interface UserCommands {
    UserId createUser(CreateUserCmd cmd);
    void activateUser(ActivateUserCmd cmd);
    void deactivateUser(DeactivateUserCmd cmd);
    void lockUser(LockUserCmd cmd);
    void unlockUser(UnlockUserCmd cmd);
}
```

Bad Example (Too Generic):

```
// Anti-pattern: Generic method handling multiple use cases
public void manageUser(String action, UserData data) {
    // Violates single responsibility principle
}
```

Validation in Application Services

Two-Level Validation Strategy:

Level 1: Application Service - Input Validation
- Format validation (email format, date format)
- Required field validation
- Data type validation
- Lookup validation (checking reference data exists)
- **Action:** Throws exceptions if validation fails
- **Tool:** JSR-303, custom validators

Level 2: Domain Level - Business Rule Validation
- Business invariant enforcement
- Complex domain rules
- Cross-entity validation
- **Action:** Domain returns result indicating success/failure with reasons
- **Location:** Domain aggregates and domain services

Example Flow:

```
@Transactional
public void placeOrder(PlaceOrderCmd cmd) {
    // 1. Application-level input validation
    if (cmd.items().isEmpty()) {
        throw new ValidationException("Order must have at least
one item");
    }

    // 2. Load aggregate
    Order order = Order.create(cmd.orderId(), cmd.customerId());

    // 3. Invoke domain operation (domain validates business
rules)
    Result result = order.addItems(cmd.items());

    // 4. Check business rule validation result
    if (!result.isSuccess()) {
        throw new
BusinessRuleViolationException(result.getErrors());
    }

    // 5. Persist and commit
    repository.save(order);
}
```

6.4 Command/Query Separation (CQRS)

Pattern Definition

CQRS (Command Query Responsibility Segregation) extends the Command-Query Separation (CQS) principle from the method level to the architectural level.

Martin Fowler's Definition:

> "Use a different model to update information than the model you use to read information."

Key Distinction:
- **CQS (Bertrand Meyer):** Methods are either commands (change state) or queries (return data), but not both
- **CQRS (Greg Young):** Separate **object models** for commands and queries

Commands vs Queries

Aspect	Commands	Queries
Purpose	Change state	Retrieve data
Side Effects	Yes - modifies data	No - read-only
Return Value	void / result status / ID	Business data (DTO)
Naming	Imperative verbs (PlaceOrder, CancelOrder)	Query verbs (GetOrder, ListOrders)
Validation	Business rules enforced	Input parameter validation only
Model Used	Write model (domain model)	Read model (query model)
Optimization	Consistency, integrity	Performance, denormalization
Database	Write database	Read database (potentially separate)
Example	`PlaceOrder`, `CancelOrder`	`GetOrderDetails`, `ListOrders`

When to Use CQRS vs Simple CRUD

Use CQRS When:

1. **Complex Domains** - Significant differences between read and write operations
2. **Performance Requirements** - High read-to-write ratio (> 10:1), need independent scaling
3. **Collaborative Domains** - Multiple users operating on same data

4. **Event-Driven Architecture** - Already using Event Sourcing
5. **Specific Bounded Contexts** - Apply to portions of system, not entire system

Do NOT Use CQRS When:

1. **Simple CRUD Applications** - Straightforward create/read/update/delete operations
2. **Low Complexity Domains** - No significant difference between read and write needs
3. **Small Systems** - Complexity overhead not justified
4. **Starting New Projects** - Begin simpler, add CQRS later if needed

Martin Fowler's Caution:

> "You should be very cautious about using CQRS. Many information systems fit well with the notion of an information base that is updated in the same way that it's read, adding CQRS to such a system can add significant complexity."

Command Side (Write Model)

The command side uses the domain model with its rich business logic, aggregates, and invariants.

Command Processing Flow:

```
1. Client submits Command (e.g., PlaceOrderCmd)
2. Command Handler (Application Service) receives command
3. Validate input format and required fields
4. Load Aggregate from Repository
5. Execute domain operation on Aggregate
6. Aggregate validates business rules and invariants
7. Persist Aggregate to Write Database
8. Collect Domain Events from Aggregate
9. Publish Domain Events
10. Return result (void, ID, or acknowledgment)
```

Schema Support:

```
command_interfaces:
  - id: cmd_order_commands
    name: OrderCommands
    aggregate_ref: agg_order
    command_records:
      - record_name: PlaceOrderCmd
        intent: placeOrder
        parameters:
          - name: customerId
            type: CustomerId
            required: true
          - name: items
            type: List<OrderItem>
            required: true
        returns: domain_id
        return_type_ref: vo_order_id
        modifies_aggregate: agg_order
        publishes_events:
          - evt_order_placed
```

Query Side (Read Model)

The query side bypasses the domain model and reads directly from optimized read models.

Query Processing Flow:

```
1. Client submits Query (e.g., GetOrderSummary)
2. Query Handler (Application Service) receives query
3. Validate query parameters
4. Query Read Model directly (bypass domain model)
5. Transform data to DTO
6. Return DTO to client
```

Schema Support:

```
query_interfaces:
  - id: qry_order_queries
    name: OrderQueries
    aggregate_ref: agg_order
    query_methods:
      - method_name: getOrderSummary
        parameters:
          - name: orderId
            type: OrderId
            required: true
        result_record_name: OrderSummary
        result_structure:
          fields:
            - name: orderId
              type: String
              serialization: "OrderId.value()"
            - name: status
              type: String
              serialization: "status.name()"
            - name: totalAmount
              type: BigDecimal
        bypasses_domain_model: true
        optimizations:
          denormalized: true
          cached: true
          indexed: true
```

CQRS-Lite (Same Database)

For many systems, you can gain benefits of CQRS without the complexity of separate databases:

Approach:
- Separate command and query handlers
- Same database for both
- Different tables optimized for reads vs. writes
- Update read tables in same transaction as write

Benefits:
- Maintains immediate consistency
- Simpler infrastructure
- Clear separation of concerns
- Independent handler optimization

Example:

```
// Write to domain tables
@Transactional
public OrderId placeOrder(PlaceOrderCmd cmd) {
    // Write to normalized domain tables
    Order order = Order.create(cmd.orderId(), cmd.customerId());
    order.addItems(cmd.items());
    orderRepository.save(order);  // Writes to order, order_items
tables

    // Update denormalized read table in same transaction
    OrderSummary summary = buildSummary(order);
    readRepository.save(summary);  // Writes to order_summary
table

    return order.getId();
}

// Read from denormalized read table
public OrderSummary getOrderSummary(OrderId orderId) {
    // Query optimized read table directly
    return readRepository.findById(orderId);
}
```

Eventual Consistency in CQRS

When using CQRS with separate databases or tables, the read model becomes eventually consistent with the write model.

Implications:
- Users may see stale data briefly after a command
- UI must handle scenarios where data hasn't updated yet
- Business processes must account for synchronization delays
- Error handling for sync failures is required

Mitigation Strategies:

1. Optimistic UI Updates:

```
// Client-side: Optimistically update UI immediately
function placeOrder(orderData) {
    // Update UI immediately (optimistic)
    displayOrderConfirmation(orderData);

    // Submit command
    api.placeOrder(orderData)
        .then(result => {
            // Command succeeded
            updateWithServerData(result);
        })
        .catch(error => {
            // Command failed - revert UI
            revertOptimisticUpdate();
            displayError(error);
        });
}
```

2. Command Result with Projection:

```
// Return projection immediately with command result
public OrderPlacedResult placeOrder(PlaceOrderCmd cmd) {
    // Execute command on write model
    OrderId orderId = commandHandler.handle(cmd);

    // Immediately build projection from write model
    Order order = orderRepository.findById(orderId);
    OrderSummary summary = OrderSummaryMapper.toDTO(order);

    return new OrderPlacedResult(orderId, summary);
}
```

6.5 Transaction Boundaries

Fundamental Rule: One Aggregate Per Transaction

Vaughn Vernon's Rule:

> "Modify only ONE aggregate instance per transaction"

This rule is the most important transaction management principle in DDD and is required by the tactical pattern:

```
TransactionBoundary:
  type: object
  properties:
    modifies_aggregates:
      type: array
      description: "Aggregates modified by this operation (should
be 0-1 for commands)"
      items:
        $ref: "#/$defs/AggId"
      maxItems: 1  # Design enforces the rule
```

Validation Rule:

```
validation_rules:
  - rule: "one_aggregate_per_transaction"
    description: "Command must modify at most one aggregate per
transaction (Vaughn Vernon rule)"
    validation: "For each operation where type='command',
                 transaction_boundary.modifies_aggregates must
have maxItems: 1"
```

Why One Aggregate Per Transaction?

Reason 1: Aggregate = Consistency Boundary

Aggregates define consistency boundaries. A transaction ensures all invariants within an aggregate are maintained:

```
public class Order {
    private List<OrderItem> items;
    private Money totalAmount;

    // Invariant: total must equal sum of items
    public void addItem(OrderItem item) {
        items.add(item);
        totalAmount = calculateTotal();  // Invariant maintained

        // Transaction ensures both are saved atomically
    }
}
```

If transactions could span multiple aggregates, you'd create distributed transactions across consistency boundaries—defeating the purpose of aggregates.

Reason 2: Scalability

Single-aggregate transactions:
- Can be executed independently
- Allow horizontal scaling
- Avoid distributed transaction coordination
- Enable partitioning by aggregate ID

Reason 3: Avoid Deadlocks

Multi-aggregate transactions increase deadlock risk:
- Transaction A locks Order, then Inventory
- Transaction B locks Inventory, then Order
- Deadlock!

Single-aggregate transactions eliminate this risk.

Reason 4: Clearer Design

The one-aggregate rule forces you to think carefully about aggregate boundaries:
- Are these truly separate aggregates?
- Should they be unified?
- Can they coordinate via eventual consistency?

Single Transaction Per Use Case

Each application service method defines one transaction boundary:

```
@Transactional  // Transaction starts here
public OrderId placeOrder(PlaceOrderCmd cmd) {
    // All operations in this method execute in one transaction

    // 1. Load aggregate
    Order order = Order.create(cmd.orderId(), cmd.customerId());

    // 2. Execute domain operations
    order.addItems(cmd.items());
    order.setShippingAddress(cmd.shippingAddress());

    // 3. Persist
    orderRepository.save(order);

    // 4. Collect events
    List<DomainEvent> events = order.getDomainEvents();

    // Transaction commits here
    return order.getId();
}  // COMMIT

// After commit, publish events
```

Key Points:
- Transaction demarcation at application service boundary
- Domain layer is unaware of transactions
- Commit happens when method returns successfully
- Rollback happens on exception

Consistency Types: Transactional vs Eventual

DDD distinguishes two consistency types:

```
consistency_type:
  type: string
  enum: [transactional, eventual]
  description: "Immediate (transactional) or deferred (eventual)
consistency"
```

Transactional Consistency (Immediate)

Used within a single aggregate:

```
@Transactional
public void placeOrder(PlaceOrderCmd cmd) {
    // Single aggregate, single transaction
    Order order = Order.create(cmd.orderId(), cmd.customerId());
    order.addItems(cmd.items());      // Invariants enforced
immediately
    order.calculateTotal();            // Consistency immediate

    orderRepository.save(order);

    // All changes committed atomically
}
```

Characteristics:
- ACID guarantees
- Immediate consistency
- Within aggregate boundary
- Rollback on failure

Eventual Consistency (Deferred)

Used between aggregates:

```
// Transaction 1: Place order
@Transactional
public OrderId placeOrder(PlaceOrderCmd cmd) {
    Order order = Order.create(cmd.orderId(), cmd.customerId());
    order.addItems(cmd.items());
    orderRepository.save(order);

    // Publish event for other aggregates
    eventPublisher.publish(new OrderPlaced(order.getId(),
order.getItems()));

    return order.getId();
}

// Transaction 2: Reserve inventory (separate transaction)
@EventListener
@Transactional
public void on(OrderPlaced event) {
    Inventory inventory =
inventoryRepository.findById(event.getWarehouseId());
    inventory.reserveItems(event.getItems());
    inventoryRepository.save(inventory);
}
```

Characteristics:
- No distributed transaction
- Temporary inconsistency acceptable
- Asynchronous coordination
- Each aggregate in its own transaction

When to Use Eventual Consistency

Ask these questions to determine if eventual consistency is acceptable:

Question 1: Is it the user's job to make data consistent?
- YES → Use transactional consistency
- NO → Use eventual consistency

Example:
- Placing an order: User's job (transactional)
- Updating inventory: System's job (eventual)

Question 2: Must the invariant be enforced immediately?
- YES → Transactional (true invariant)
- NO → Eventual

Example:
- Order total = sum of items: True invariant (transactional)
- Inventory count reflects all orders: Not a true invariant (eventual)

Question 3: Can the business tolerate a delay?
- YES → Eventual consistency acceptable
- NO → Consider if you've designed the right boundaries

Example:
- Welcome email: Delay acceptable (eventual)
- Password reset token: Delay problematic (transactional)

Sagas for Complex Workflows

For long-running processes spanning multiple aggregates, use the **Saga Pattern**:

```
OrderSaga coordinating Order, Inventory, Payment, Shipping

Step 1: Create Order
  BEGIN TRANSACTION
    order = Order.create()
    orderRepository.save(order)
  COMMIT
  PUBLISH OrderCreated

Step 2: Reserve Inventory (event handler)
  ON EVENT OrderCreated
    BEGIN TRANSACTION
      inventory.reserve(items)
      inventoryRepository.save(inventory)
    COMMIT
    PUBLISH InventoryReserved

Step 3: Process Payment (event handler)
  ON EVENT InventoryReserved
    BEGIN TRANSACTION
      payment.process(amount)
      paymentRepository.save(payment)
    COMMIT
    PUBLISH PaymentProcessed (or PaymentFailed)

Step 4: Complete Order (event handler)
  ON EVENT PaymentProcessed
    BEGIN TRANSACTION
      order.complete()
      orderRepository.save(order)
    COMMIT
    PUBLISH OrderCompleted

// Compensation on failure
  ON EVENT PaymentFailed
    BEGIN TRANSACTION
      inventory.release(items)
      inventoryRepository.save(inventory)
    COMMIT
    BEGIN TRANSACTION
      order.cancel()
```

```
      orderRepository.save(order)
    COMMIT
```

6.6 Application Service Orchestration

ApplicationServiceOperation Structure

The tactical schema defines a structured workflow for application service operations:

```
ApplicationServiceOperation:
  type: object
  required: [name, type]
  properties:
    name:
      type: string
      pattern: "^[a-z][a-zA-Z]+$"
      description: "Operation method name (e.g., createUser,
placeOrder)"
    type:
      type: string
      enum: [command, query]
      description: "Whether this operation modifies state or
retrieves data"
    transaction_boundary:
      $ref: "#/$defs/TransactionBoundary"
    workflow:
      $ref: "#/$defs/Workflow"
```

Workflow Structure

The `Workflow` type defines orchestration steps:

```
Workflow:
  type: object
  description: "Orchestration workflow steps"
  properties:
    validates_input:
      type: boolean
      description: "Performs input/format validation"
      default: true
    loads_aggregates:
      type: array
      description: "Aggregates loaded from repositories"
      items:
        $ref: "#/$defs/AggId"
    invokes_domain_operations:
      type: array
      description: "Domain operations invoked on aggregates"
      items:
        type: string
    invokes_domain_services:
      type: array
      description: "Domain services invoked"
      items:
        $ref: "#/$defs/SvcDomId"
    persists_aggregates:
      type: boolean
      description: "Saves aggregates back to repository"
      default: true
    publishes_events:
      type: array
      description: "Domain events published after successful
execution"
      items:
        $ref: "#/$defs/EvtId"
    returns_dto:
      type: string
      description: "DTO returned for queries"
```

Complete Workflow Example

```
@Transactional
public void deactivateUser(DeactivateUserCmd cmd) {
    // Step 1: Validate input
    if (cmd.userId() == null) {
        throw new ValidationException("User ID is required");
    }
    if (StringUtils.isBlank(cmd.reason())) {
        throw new ValidationException("Deactivation reason is
required");
    }

    // Step 2: Load aggregate
    User user = userRepository.findById(cmd.userId())
        .orElseThrow(() -> new
UserNotFoundException(cmd.userId()));

    // Step 3: Invoke domain service (if needed)
    // In this case, no domain service needed

    // Step 4: Invoke domain operation
    user.deactivate(cmd.reason());

    // Step 5: Persist aggregate
    userRepository.save(user);

    // Step 6: Publish events
    eventPublisher.publishEvent(new UserDeactivated(
        user.getUserId().value(),
        cmd.reason(),
        Instant.now()
    ));

    // Step 7: Return (void for state transitions)
}
```

Domain Event Publishing Patterns

Application Services are responsible for publishing domain events after successful transactions.

Pattern 1: Collect and Publish After Commit

```
@Transactional
public OrderId placeOrder(PlaceOrderCmd cmd) {
    Order order = Order.create(cmd.orderId(), cmd.customerId());
    order.addItems(cmd.items());

    orderRepository.save(order);

    // Collect events from aggregate
    List<DomainEvent> events = order.getDomainEvents();

    // Register for publishing after commit
    TransactionSynchronizationManager.registerSynchronization(
        new AfterCommitPublisher(events, eventPublisher)
    );

    return order.getId();
}

class AfterCommitPublisher extends
TransactionSynchronizationAdapter {
    @Override
    public void afterCommit() {
        events.forEach(publisher::publish);
    }
}
```

Pattern 2: Outbox Pattern for Reliability

For guaranteed event delivery:

```
@Transactional
public OrderId placeOrder(PlaceOrderCmd cmd) {
    // 1. Execute domain operation
    Order order = Order.create(cmd.orderId(), cmd.customerId());
    order.addItems(cmd.items());
    orderRepository.save(order);

    // 2. Write events to outbox table in same transaction
    List<DomainEvent> events = order.getDomainEvents();
    for (DomainEvent event : events) {
        OutboxMessage message = new OutboxMessage(
            UUID.randomUUID(),
            event.getClass().getName(),
            objectMapper.writeValueAsString(event),
            Instant.now()
        );
        outboxRepository.save(message);
    }

    return order.getId();
}

// Separate background process publishes from outbox
@Scheduled(fixedDelay = 1000)
public void publishOutboxMessages() {
    List<OutboxMessage> pending =
outboxRepository.findUnpublished();

    for (OutboxMessage message : pending) {
        try {
            DomainEvent event = deserializeEvent(message);
            eventPublisher.publish(event);

            message.markPublished();
            outboxRepository.save(message);
        } catch (Exception e) {
            // Retry later
            message.incrementRetryCount();
            outboxRepository.save(message);
        }
```

```
    }
}
```

6.7 Read Models and DTOs

Read Model Purpose

Read Models serve a fundamentally different purpose than write models. While write models enforce business rules and maintain transactional consistency, read models optimize for query performance and user experience.

Key Motivations:

1. **Query Performance**: Denormalized data eliminates expensive joins
2. **View-Specific Optimization**: Each view gets its own optimized projection
3. **Scalability**: Read databases can be replicated and scaled independently
4. **Simplicity**: No business logic in queries—just data retrieval

Denormalization Strategies

Normalized Write Model:

```
-- Separate tables (normalized)
orders: order_id, customer_id, status, created_at
customers: customer_id, name, email
order_items: item_id, order_id, product_id, quantity, price
products: product_id, name, description
```

Denormalized Read Model:

```
-- Single denormalized table
order_summary:
  order_id,
  customer_id,
  customer_name,        -- Denormalized from customers
  customer_email,       -- Denormalized from customers
  status,
  item_count,           -- Pre-calculated
  total_amount,         -- Pre-calculated
  created_at,
  shipping_address      -- Flattened JSON
```

Read Model Update via Projections

Read models are kept synchronized with write models through event-driven projections:

```
@Component
public class OrderProjections {

    private final OrderSummaryRepository readRepository;

    @EventListener
    @Transactional
    public void on(OrderPlaced event) {
        // Create new read model entry
        OrderSummaryEntity summary = new OrderSummaryEntity();
        summary.setOrderId(event.getOrderId().value());
        summary.setCustomerId(event.getCustomerId().value());
        summary.setStatus("PLACED");
        summary.setItemCount(event.getItems().size());
        summary.setTotalAmount(calculateTotal(event.getItems()));
        summary.setCreatedAt(event.getTimestamp());

        readRepository.save(summary);
    }

    @EventListener
    @Transactional
    public void on(OrderShipped event) {
        // Update existing read model entry
        OrderSummaryEntity summary =
readRepository.findById(event.getOrderId())
            .orElseThrow();

        summary.setStatus("SHIPPED");
        summary.setShippedAt(event.getTimestamp());

        readRepository.save(summary);
    }
}
```

DTOs vs Domain Objects

Domain Objects:

- Belong to domain layer
- Rich with behavior and business logic

- Encapsulate state and invariants
- May contain references to other domain objects
- Optimized for business operations

DTOs:
- Belong to application/API layer
- Pure data containers with no behavior
- Simple, serializable structures
- Optimized for data transfer
- Often flat and denormalized

Martin Fowler's Definition:

> "DTO is an object that carries data between processes. The difference between data transfer objects and business objects or data access objects is that a DTO does not have any behavior except for storage and retrieval of its own data."

Flat DTO Pattern (Knight)

The Knight codebase demonstrates a flat DTO pattern that avoids nested objects:

Design Constraints:

```
query_interfaces:
  - id: qry_user_queries
    name: UserQueries
    result_characteristics:
      flat_structure: true      # Enforces flat structure
      string_serialization: true  # Complex types as strings
```

Why Flat Structure?

Benefits:
1. **Simpler Serialization**: No complex object graphs
2. **Better Caching**: Easier to cache flat structures

3. **Client-Friendly**: Easier for clients to consume
4. **Version Resilience**: Flatter structures evolve more easily
5. **Performance**: Less serialization overhead

Anti-Pattern: Nested DTOs:

```
// AVOID: Nested structure
public class OrderDTO {
    private String orderId;
    private CustomerDTO customer;  // Nested object
    private List<OrderItemDTO> items;  // Nested collection
    private AddressDTO shippingAddress;  // Nested object
}
```

Recommended: Flat Structure:

```
// BETTER: Flat structure
public record OrderSummary(
    String orderId,
    String customerId,      // ID only, not nested object
    String customerName,    // Denormalized
    String customerEmail,   // Denormalized
    String status,
    int itemCount,          // Count, not collection
    String totalAmount,     // Pre-formatted string
    String shippingStreet,  // Flattened address
    String shippingCity,
    String shippingState
) {}
```

String Serialization for Complex Types

The Knight pattern serializes complex types (IDs, enums, dates) to strings in DTOs:

Pattern 1: Domain IDs to Strings:

```
// Domain: UserId value object
public record UserId(String id) {
    public static UserId of(String id) {
        return new UserId(id);
    }
}

// DTO: Serialized to String
public record UserSummary(
    String userId  // Not UserId, just String
) {}

// Mapping
UserSummary toDTO(User user) {
    return new UserSummary(
        user.getUserId().id()  // Extract string from value object
    );
}
```

Pattern 2: Enums to Strings:

```
// Domain: Status enum
public enum UserStatus {
    PENDING, ACTIVE, LOCKED, DEACTIVATED
}

// DTO: Serialized to String
public record UserSummary(
    String status  // Not UserStatus enum
) {}

// Mapping
UserSummary toDTO(User user) {
    return new UserSummary(
        user.getStatus().name()  // Enum to string
    );
}
```

Pattern 3: Dates/Times to ISO-8601 Strings:

```
// Domain: Instant
private Instant createdAt;

// DTO: String in ISO-8601 format
public record UserSummary(
    String createdAt  // "2025-10-24T10:30:00Z"
) {}

// Mapping
UserSummary toDTO(User user) {
    return new UserSummary(
        user.getCreatedAt().toString()  // ISO-8601 string
    );
}
```

Bypasses Domain Model Flag

Queries can include a `bypasses_domain_model` flag to indicate whether they go through the domain layer or directly to the read database:

Bypass (bypasses_domain_model: true):
- Read model is denormalized and optimized
- No business logic needed
- High-volume queries
- Performance is critical

```
public OrderSummary getOrderSummary(OrderId orderId) {
    // Bypass domain model - query read database directly
    return readRepository.findById(orderId.value())
        .map(this::toDTO)
        .orElseThrow(() -> new OrderNotFoundException(orderId));
}
```

Use Domain Model (bypasses_domain_model: false):
- Need domain logic for permissions
- Business rules affect what data is shown
- Computed values require domain methods

```
public OrderDetails getOrderDetails(OrderId orderId, UserId
requestingUser) {
    // Load from domain model
    Order order = orderRepository.findById(orderId)
        .orElseThrow(() -> new OrderNotFoundException(orderId));

    // Apply business logic
    if (!order.canBeViewedBy(requestingUser)) {
        throw new UnauthorizedException();
    }

    // Return with domain-calculated values
    return OrderDetails.from(order);
}
```

6.8 Best Practices and Anti-Patterns

Best Practices

1. Keep Application Services Thin

- No business logic
- Only orchestration and coordination
- Delegate to domain for business rules

2. One Method Per Use Case

- Each method represents one business operation
- Clear, descriptive names (createUser, placeOrder, approveInvoice)
- Avoid generic methods handling multiple use cases

3. Stateless Always

- Application services must not maintain state between calls
- All needed data passed via method parameters
- Use dependency injection for infrastructure dependencies

4. Transaction Boundary = Method Boundary

- Each method defines one transaction
- @Transactional annotation on command methods
- No transactions for query methods

5. ID Generation

- Generate domain IDs in application service (not database)
- Use UUID or domain-specific ID generation strategy
- Return domain IDs for creation operations

6. Immutable Commands

- Use record types or final fields
- Commands represent intent, should not be modified

7. Return DTOs, Not Aggregates

- Serialize complex types to strings
- Use flat structure (no nested objects)
- Return aggregate counts, not full collections

8. Two-Level Validation

- **Application Layer:** Input format, required fields, data types
- **Domain Layer:** Business rules, invariants, complex validations

Anti-Patterns to Avoid

Anti-Pattern 1: Business Logic in Application Service

```
// Bad: Anti-Pattern
@Transactional
public void placeOrder(PlaceOrderCmd cmd) {
    Order order = new Order(cmd.orderId());

    // WRONG: Business logic in application service
    BigDecimal total = BigDecimal.ZERO;
    for (OrderItem item : cmd.items()) {
        BigDecimal lineTotal = item.quantity()
            .multiply(item.unitPrice());
        total = total.add(lineTotal);
    }
    order.setTotal(total);

    repository.save(order);
}

// Good: Correct Pattern
@Transactional
public void placeOrder(PlaceOrderCmd cmd) {
    Order order = Order.create(cmd.orderId());

    // Business logic delegated to aggregate
    order.addItems(cmd.items());

    repository.save(order);
}
```

Anti-Pattern 2: Modifying Multiple Aggregates in One Transaction

```
// Bad: Anti-Pattern
@Transactional
public void processOrder(ProcessOrderCmd cmd) {
    // WRONG: Modifying multiple aggregates in one transaction
    Order order = orderRepository.findById(cmd.orderId());
    order.confirm();
    orderRepository.save(order);

    Inventory inventory =
inventoryRepository.findById(cmd.warehouseId());
    inventory.reserve(order.getItems());
    inventoryRepository.save(inventory);
}

// Good: Correct Pattern
@Transactional
public void confirmOrder(ConfirmOrderCmd cmd) {
    Order order = orderRepository.findById(cmd.orderId());
    order.confirm();
    orderRepository.save(order);

    // Publish event for eventual consistency
    eventPublisher.publishEvent(new
OrderConfirmed(order.getId()));
}

// Separate event handler
@EventListener
@Transactional
public void onOrderConfirmed(OrderConfirmed event) {
    Inventory inventory =
inventoryRepository.findById(event.warehouseId());
    inventory.reserve(event.items());
    inventoryRepository.save(inventory);
}
```

Anti-Pattern 3: Generic Application Service Methods

```
// Bad: Anti-Pattern
public void updateUser(String action, Map<String, Object> data) {
    switch (action) {
        case "activate": // ...
        case "deactivate": // ...
        case "lock": // ...
    }
}

// Good: Correct Pattern
public void activateUser(ActivateUserCmd cmd) { }
public void deactivateUser(DeactivateUserCmd cmd) { }
public void lockUser(LockUserCmd cmd) { }
```

Anti-Pattern 4: Anemic Domain Model

```
// Bad: Anti-Pattern: Anemic aggregate
public class Order {
    private OrderId id;
    private BigDecimal total;

    public void setTotal(BigDecimal total) { this.total = total; }
    public BigDecimal getTotal() { return total; }
}

// Good: Correct Pattern: Rich domain model
public class Order {
    private final OrderId id;
    private BigDecimal total;
    private List<OrderItem> items;

    // Business logic in aggregate
    public void addItems(List<OrderItem> items) {
        this.items.addAll(items);
        this.total = calculateTotal();
    }

    private BigDecimal calculateTotal() {
        return items.stream()
            .map(item ->
item.quantity().multiply(item.unitPrice()))
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}
```

---

7. Backend-for-Frontend (BFF) Pattern

7.1 Overview

Pattern Definition

Phil Calçado's Definition:

> "A server-side component tightly coupled to a specific user interface, providing one BFF per user interface type (web, mobile, tablet, etc.)."

The **Backend for Frontend (BFF)** pattern is an architectural pattern where you create separate backend services for different frontend experiences. Each BFF is optimized for a specific client type and owned by the team building that frontend.

Origin and History

First introduced by **Phil Calçado** and colleagues at **SoundCloud** in 2011, formally documented in "The Back-end for Front-end Pattern (BFF)" (September 18, 2015). The pattern emerged from SoundCloud's transition from a monolithic Rails application to microservices architecture.

Problem at SoundCloud:
- Multiple diverse client types (web, mobile apps, partner integrations)
- Generic API Gateway becoming bloated with client-specific logic
- Frontend teams blocked on centralized API team for changes
- "One-size-fits-all" API didn't optimize for any specific client

Solution:
- Create separate BFF for each client type
- Frontend teams own their BFF
- Each BFF calls microservices directly
- Eliminates dependency on centralized API team

Core Principle: "One Experience, One BFF"

The fundamental principle of the BFF pattern:

Scope by Client:
- One BFF serves **ONE** user interface type (web, iOS, Android, partner API)
- Each BFF is tailored exclusively for that client's needs
- BFFs do not share client-specific logic

Scope by Data:
- One BFF aggregates data from **MULTIPLE** bounded contexts/microservices
- BFF scope is defined by **CLIENT TYPE**, not by bounded contexts
- A single BFF endpoint may combine data from 5-10 downstream services

Key Distinction:

```
❌ WRONG: One BFF per bounded context
✓ CORRECT: One BFF per client type
```

BFF in the Schema

The strategic schema provides comprehensive BFF modeling:

```
BFFScope:
  type: object
  description: |
    BFF serves exactly ONE client type (web, mobile, etc.) and
aggregates
    data from MULTIPLE bounded contexts. BFF is owned by the
frontend team.

Key principle: "One experience, one BFF" - scope is defined by
CLIENT TYPE.
  required:
    - id
    - name
    - client_type
    - serves_interface
    - aggregates_from_contexts
    - owned_by_team
  properties:
    client_type:
      enum: [web, mobile_ios, mobile_android, desktop,
partner_api, iot, tablet]
    aggregates_from_contexts:
      type: array
      items:
        $ref: "#/$defs/BcId"
      minItems: 1
    responsibilities:
      properties:
        data_aggregation:
          const: true  # BFF MUST aggregate data
        business_logic:
          const: false  # BFF must NOT contain business logic
```

With the BFF pattern's origin and core principle established, we now examine the most critical design constraint: how BFF scope is defined by client type (not bounded context) and why this distinction matters.

7.2 BFF Scope: One Experience, One BFF

Single UI Focus

Principle: Each BFF serves exactly **one type of user interface**

Examples:

BFF ID	Name	Client Type	Serves
`bff_web`	WebBFF	`web`	Web applications
`bff_ios`	iOSBFF	`mobile_ios`	iOS mobile apps
`bff_android`	AndroidBFF	`mobile_android`	Android mobile apps
`bff_partner_api`	PartnerAPIBFF	`partner_api`	Third-party integrations
`bff_desktop`	DesktopBFF	`desktop`	Desktop applications
`bff_iot`	IoTBFF	`iot`	IoT devices

Why One BFF Per Client:

1. **Different Data Needs**
   - Mobile needs minimal data due to bandwidth constraints
   - Web can handle rich, nested structures
   - IoT requires highly compressed payloads
   - Partner API needs comprehensive, stable contracts
2. **Different Authentication Mechanisms**
   - Web: Session-based or JWT tokens
   - Mobile: OAuth with refresh tokens
   - Partner API: API keys or mutual TLS
   - IoT: Device certificates

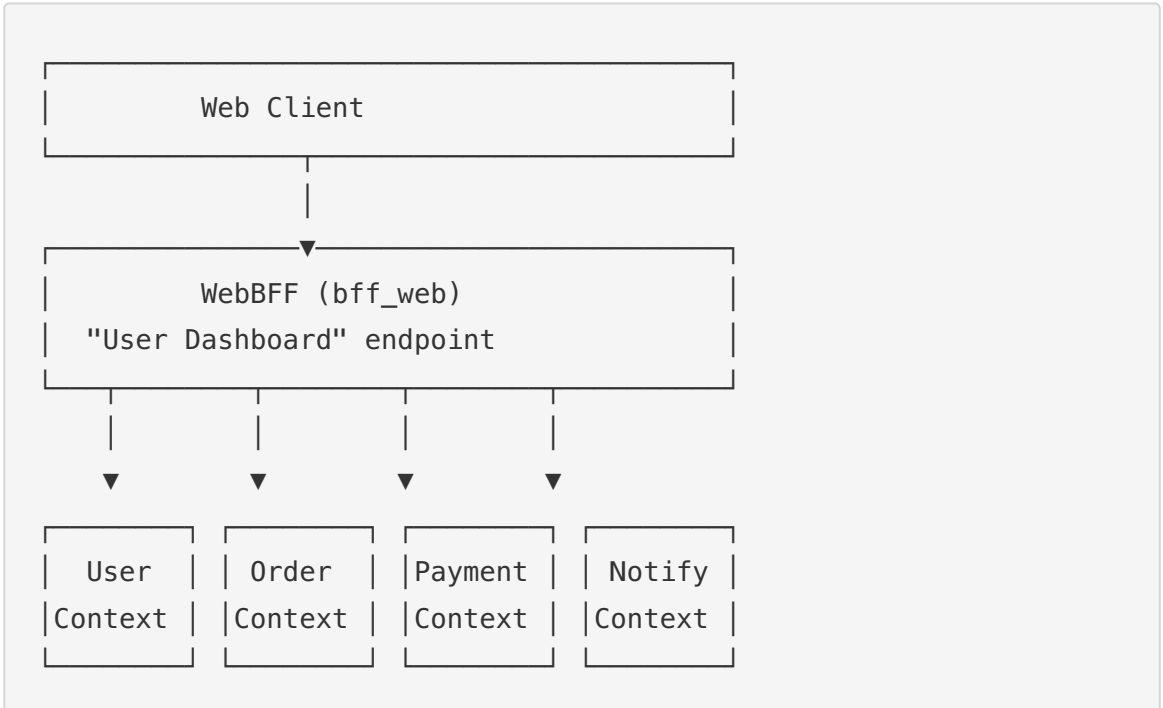
3. **Different Performance Requirements**
   - Mobile: Minimize latency, optimize for battery
   - Web: Rich interactions, real-time updates
   - Partner API: Batch operations, webhook support
4. **Different Team Ownership**
   - Web team owns WebBFF
   - iOS team owns iOSBFF
   - Android team owns AndroidBFF
   - Each team can iterate independently
5. **Independent Evolution**
   - Each BFF can evolve at its own pace
   - Breaking changes in one BFF don't affect others
   - Deploy independently

Multiple Bounded Context Aggregation

Critical Distinction: BFF scope is defined by the CLIENT TYPE, not by bounded contexts.

A single BFF typically **aggregates data from multiple bounded contexts/ microservices**.

Example Architecture:



Schema Definition:

```
bff_scopes:
  - id: bff_web
    name: WebBFF
    client_type: web
    serves_interface: "Web application user dashboard and
management"
    aggregates_from_contexts:
      - bc_user_management
      - bc_order_management
      - bc_payment_processing
      - bc_notification_service
      - bc_analytics
    owned_by_team: "Web Frontend Team"
    team_type: frontend
```

Single BFF Endpoint:

```
GET /api/web/user-dashboard/123
```

Aggregates data from 5 bounded contexts into unified response:

```
{
  "user": {
    "id": "user_123",
    "name": "John Doe",
    "email": "john@example.com",
    "status": "ACTIVE"
  },
  "recentOrders": [
    {
      "orderId": "order_456",
      "status": "shipped",
      "total": "$125.00",
      "date": "2025-10-20"
    }
  ],
  "paymentMethods": [
    {
      "type": "credit_card",
      "last4": "1234",
      "brand": "Visa"
    }
  ],
  "unreadNotifications": 5,
  "usageStats": {
    "loginCount": 42,
    "lastLogin": "2025-10-23T15:30:00Z"
  }
}
```

Without BFF:

The client would need to make **5 separate API calls**:

```
GET /api/users/123
GET /api/users/123/orders
GET /api/users/123/payment-methods
GET /api/notifications?userId=123
GET /api/analytics/users/123
```

With BFF:

One aggregated call returns everything the dashboard needs.

Core Responsibilities

What BFFs SHOULD Do:

1. Data Aggregation (Schema: `data_aggregation: true`)

Consolidate multiple downstream service calls into single endpoints:

```
@Get("/dashboard/{userId}")
public DashboardDTO getDashboard(@PathVariable String userId) {
    UserId id = UserId.of(userId);

    // Aggregate from multiple contexts
    var user = userQueries.getUserSummary(id);
    var orders = orderQueries.getRecentOrders(id);
    var payments = paymentQueries.getPaymentMethods(id);
    var notifCount = notificationQueries.getUnreadCount(id);
    var stats = analyticsQueries.getUserStats(id);

    // Combine into client-specific DTO
    return new DashboardDTO(user, orders, payments, notifCount,
stats);
}
```

2. Client-Specific Customization

Different DTOs for different clients:

```
// WebBFF: Rich, comprehensive data
public record WebDashboardDTO(
    UserProfile user,
    List<OrderSummary> recentOrders,      // Full list
    List<PaymentMethod> paymentMethods,   // All fields
    int unreadNotifications,
    UsageStatistics stats                 // Detailed stats
) {}

// MobileBFF: Minimal, bandwidth-optimized
public record MobileDashboardDTO(
    String userId,
    String userName,
    int orderCount,                       // Count only, not full
list
    String primaryPaymentLast4,           // One payment method
    int unreadNotifications               // No detailed stats
) {}
```

3. Presentation Logic (Schema: `presentation_logic: true`)

Client-specific logic not shared across client types:

```
// Mobile-specific: Optimize for bandwidth
if (orders.size() > 5) {
    orders = orders.subList(0, 5);  // Limit to 5 for mobile
}

// Web-specific: Include pagination metadata
response.addMetadata("totalPages", totalOrders / pageSize);
```

4. Format Translation (Schema: `format_translation: true`)

Transform domain models into client-friendly formats:

```
// Domain: Money value object
Money totalAmount = order.getTotalAmount();

// BFF: Format for client
String formattedTotal = String.format("$%.2f",
totalAmount.getAmount());
```

5. Error Handling

Handle partial failures gracefully:

```
@Get("/dashboard/{userId}")
public DashboardDTO getDashboard(@PathVariable String userId) {
    UserId id = UserId.of(userId);

    var user = userQueries.getUserSummary(id);

    // Graceful degradation
    List<OrderSummary> orders;
    try {
        orders = orderQueries.getRecentOrders(id);
    } catch (ServiceUnavailableException e) {
        orders = Collections.emptyList();  // Degraded response
        log.warn("Order service unavailable", e);
    }

    int notificationCount;
    try {
        notificationCount =
notificationQueries.getUnreadCount(id);
    } catch (ServiceUnavailableException e) {
        notificationCount = 0;  // Degraded response
    }

    return new DashboardDTO(user, orders, notificationCount);
}
```

What BFFs Should NOT Do:

According to Phil Calçado and Sam Newman:

Anti-Pattern Prevention:

```
anti_patterns:
  shared_business_logic:
    const: false  # Do NOT duplicate business logic
  generic_cross_cutting_concerns:
    const: false  # Do NOT implement auth/logging
  direct_database_access:
    const: false  # Do NOT access databases directly
  serving_multiple_client_types:
    const: false  # Do NOT serve multiple clients
```

1. Generic Cross-Cutting Concerns → Use API Gateway upstream
- Authentication (JWT validation, OAuth flows)
- SSL termination
- Rate limiting
- Request logging
- Response caching

2. Shared Business Logic → Belongs in domain services
- Do NOT duplicate business rules across BFFs
- Delegate to application services for all business logic
- BFF contains NO calculations, validations, or business decisions

3. Data Persistence → Delegate to downstream services
- BFF does NOT write to databases directly
- BFF does NOT manage transactions
- Always call application services/microservices

4. Direct Database Access → Always call application services

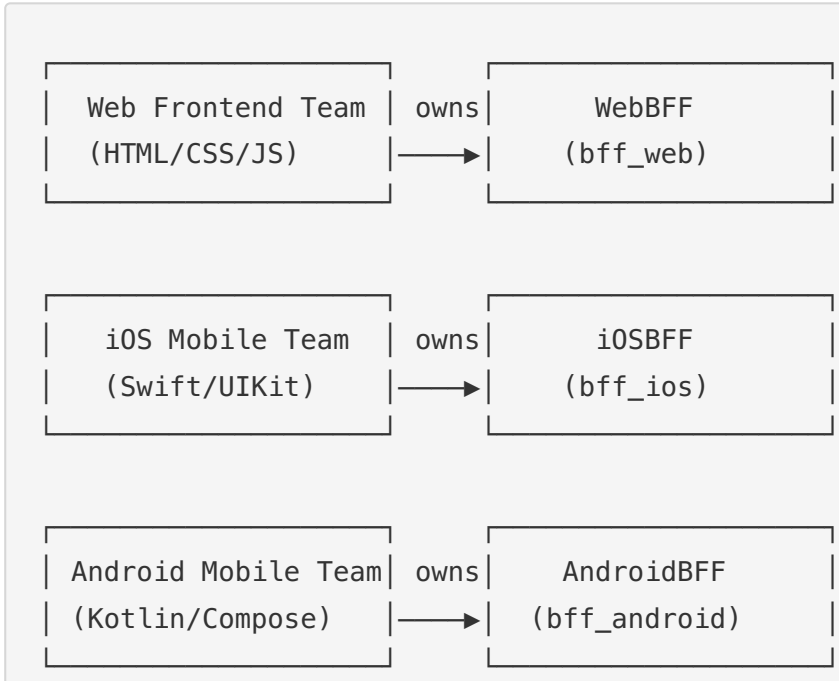
```
// Bad: Anti-Pattern: Direct database access in BFF
@Get("/users/{userId}")
public UserDTO getUser(@PathVariable String userId) {
    // WRONG: Direct SQL query in BFF
    String sql = "SELECT * FROM users WHERE id = ?";
    UserEntity user = jdbcTemplate.queryForObject(sql, ...);
    return toDTO(user);
}

// Good: Correct: Delegate to application service
@Get("/users/{userId}")
public UserDTO getUser(@PathVariable String userId) {
    var userSummary =
userQueries.getUserSummary(UserId.of(userId));
    return toDTO(userSummary);
}
```

Team Ownership and Conway's Law

Conway's Law:

> "Organizations design systems that mirror their communication structure."

BFF Ownership Model:



Schema Definition:

```
bff_scope:
  owned_by_team: "Web Frontend Team"  # Required field
  team_type: frontend                 # Must be frontend-oriented
```

Benefits of Frontend Team Ownership:

1. **Autonomy**: Frontend teams retain control over their API needs
2. **Rapid Iteration**: No dependency on centralized API teams
3. **Faster Time-to-Market**: Changes deployed independently
4. **Clear Boundaries**: Ownership is unambiguous
5. **Domain Knowledge**: Frontend team understands client needs best

Phil Calçado's Quote:

> "The simple act of limiting the number of consumers they support makes BFFs much easier to work with and change, and helps teams developing customer-facing applications retain more autonomy."

7.3 BFF vs API Gateway

API Gateway Pattern

What is an API Gateway?
- **Single point of entry** for all clients
- Provides **generic, cross-cutting concerns**
- Infrastructure-level pattern

API Gateway Responsibilities:

Concern	Description
SSL Termination	Handles HTTPS encryption
Authentication	Validates JWT tokens, OAuth flows
Rate Limiting	Throttles requests per client
Request Logging	Centralized logging and monitoring
Response Caching	Cache frequently accessed data
Protocol Translation	HTTP → gRPC, REST → GraphQL
Load Balancing	Distributes traffic across instances

When to Use API Gateway:
- Single client type or very similar clients
- Need centralized infrastructure management
- Generic cross-cutting concerns only
- Simple pass-through with common transformations

Limitations:
- Becomes bloated when serving multiple diverse client types
- Difficult to optimize for specific clients
- Centralized team becomes bottleneck
- "One size fits all" approach doesn't work well

BFF Pattern

What is a BFF?
- **Multiple entry points**, one per client type
- Provides **client-specific API tailoring**
- Application-level pattern

BFF Responsibilities:

Concern	Description
Data Aggregation	Combines data from multiple services
View Model Composition	Assembles client-specific responses
Client-Specific Logic	Implements presentation logic
Format Transformation	Converts domain models to client formats
Client Optimization	Tailored responses per client type

When to Use BFF:

- Multiple diverse client types (web, mobile, IoT, partners)
- Different team ownership for each client
- Varying authentication mechanisms
- Client-specific business logic needed
- Need to optimize independently per client
- Conway's Law: Team structure mirrors architecture

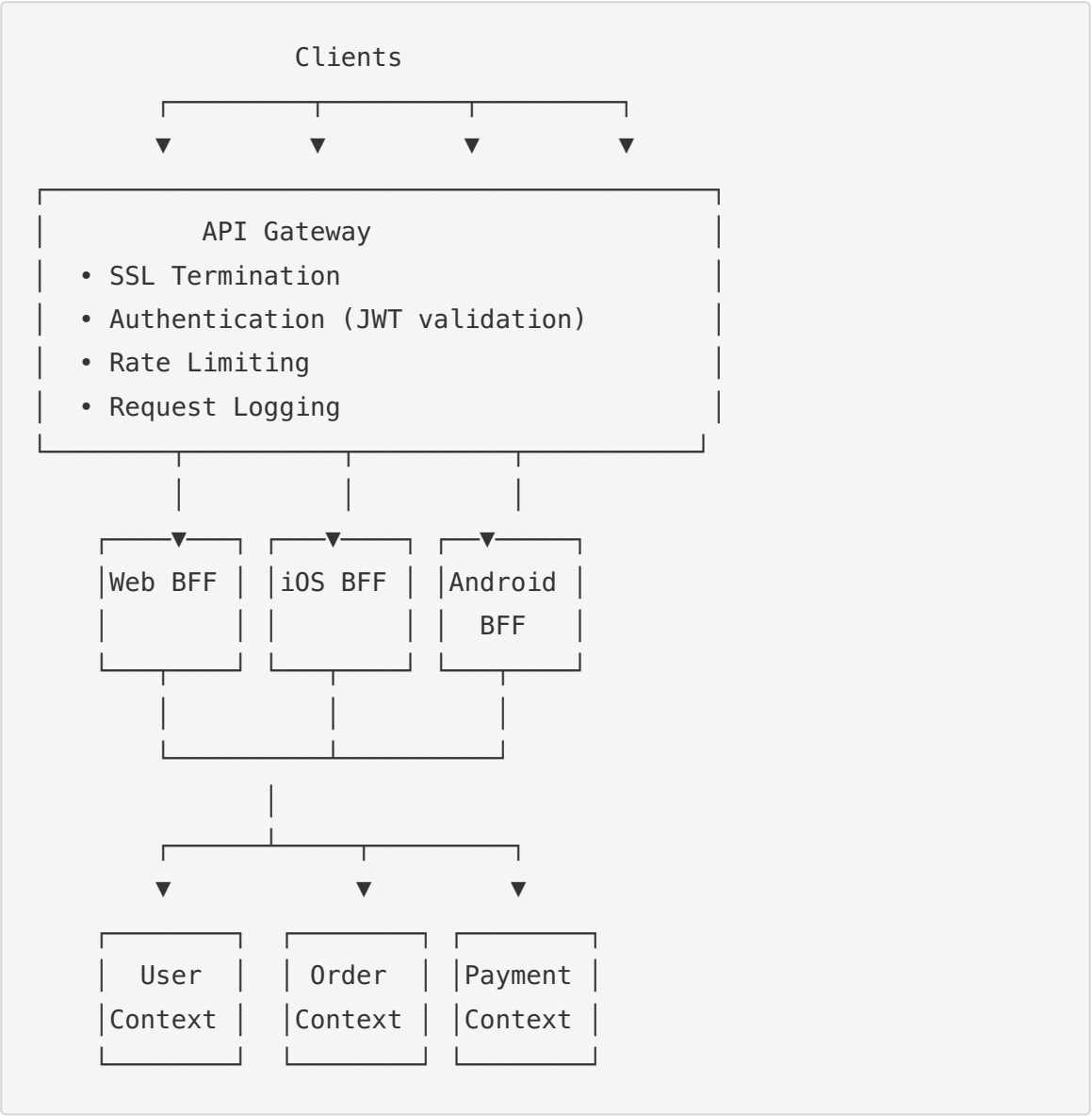
Decision Matrix

Factor	Use API Gateway	Use BFF
Number of client types	1-2 similar	3+ diverse
Team structure	Single centralized team	Multiple client teams
Business logic variance	Minimal	Significant per client
Authentication	Uniform mechanism	Client-specific mechanisms
Future scalability	Limited growth expected	Ecosystem of apps planned
Organizational structure	Centralized control	Decentralized (Conway's Law)
Primary need	Cross-cutting concerns	Client optimization
Data aggregation	Simple pass-through	Complex multi-service aggregation
Response format	Standardized	Client-specific

Hybrid Approach (Recommended)

Best Practice: Use BOTH patterns together

Architecture:



Layering:

1. **API Gateway (upstream)** → Infrastructure concerns
   - Owned by platform/infrastructure team
   - Handles authentication, SSL, rate limiting
   - Generic, reusable functionality
2. **BFFs (downstream)** → Client-specific orchestration
   - Owned by frontend teams
   - Data aggregation from multiple contexts
   - Client-optimized responses
3. **Microservices/Bounded Contexts** → Business logic
   - Owned by domain teams
   - Contains business rules and domain logic
   - Stateless, scalable services

Schema Support:

```
bff_scope:
  upstream_dependencies:
    - api_gateway       # Optional API Gateway upstream
    - load_balancer
  calls:
    - application_service     # BFF calls these downstream
    - bounded_context_api
  architecture_layer: "integration"  # BFF is integration layer
```

Key Distinctions

API Gateway:
- Infrastructure layer
- Owned by platform/infrastructure team
- Generic, reusable functionality
- One gateway for all clients
- Cross-cutting concerns only
- Technical focus (security, routing)

BFF:
- Integration/Application layer
- Owned by frontend team
- Client-specific functionality
- One BFF per client type
- Data aggregation and orchestration
- Business focus (use cases, workflows)

Analogy:
- **API Gateway** = Airport security checkpoint (everyone goes through)
- **BFF** = Airline lounge (specific to your airline/ticket class)

7.4 BFF Interface Design

BFFInterface vs BFFScope

The BFF pattern distinguishes between:

BFFScope:
- High-level definition of the BFF
- Defines client type, team ownership, responsibilities
- One per client type

BFFInterface:
- Concrete implementation of the BFF for a specific bounded context
- REST API endpoints, request/response DTOs
- Multiple per BFFScope

Relationship:

```
BFFScope (bff_web)
    ├─ BFFInterface (bff_if_user_web)     → User management
endpoints
    ├─ BFFInterface (bff_if_order_web)    → Order management
endpoints
    └─ BFFInterface (bff_if_payment_web)  → Payment endpoints
```

Schema Definition:

```
bff_interfaces:
  - id: bff_if_user_web
    name: "User Management Web BFF Interface"
    bff_scope_ref: bff_web                      # Belongs to
WebBFF
    primary_bounded_context_ref: bc_user_management
    additional_context_refs:
      - bc_notification_service                  # Also calls
notifications
    base_path: "/api/web/users"
    endpoints:
      - path: "/create"
        method: POST
        operation_type: command
        delegates_to_commands: [cmd_create_user]
      - path: "/{userId}"
        method: GET
        operation_type: query
        delegates_to_queries: [qry_get_user_summary]
```

REST Resource Mapping

Principle: BFF endpoints should align with **client use cases**, not just domain aggregates.

Traditional Domain-Aligned API:

```
GET /api/users/123
GET /api/users/123/orders
GET /api/users/123/payment-methods
GET /api/notifications?userId=123
```

Client makes 4+ separate calls

BFF Client-Optimized API:

```
GET /api/web/user-dashboard/123
```

Single call returns aggregated response

Schema Definition:

```
endpoints:
  - path: "/user-dashboard/{userId}"
    method: GET
    operation_type: query
    aggregates_data_from:
      - bc_user_management
      - bc_order_management
      - bc_payment_processing
      - bc_notification_service
    response_dto:
      name: UserDashboardDTO
      fields:
        - name: userId
          type: string
        - name: recentOrders
          type: array
        - name: paymentMethods
          type: array
        - name: unreadNotifications
          type: integer
```

HTTP Verb to Command/Query Mapping

CQRS Mapping in BFF:

Operation Type	HTTP Method	BFF Endpoint	Delegates To
Query (single)	GET	`/api/web/users/123`	GetUserQuery
Query (collection)	GET	`/api/web/users?status=active`	ListUsersQuery
Command (create)	POST	`/api/web/users`	CreateUserCommand
Command (action)	POST	`/api/web/users/123/activate`	ActivateUserCommand
Command (update)	PUT/ PATCH	`/api/web/users/123`	UpdateUserCommand
Command (delete)	DELETE	`/api/web/users/123`	DeleteUserCommand

Schema Support:

```
endpoints:
  - path: "/users/{userId}/activate"
    method: POST
    operation_type: command        # Explicit command
classification
    delegates_to_commands:
      - cmd_activate_user
```

Knight Pattern (Command-Oriented):

All commands use POST with action-based paths:

```
POST /commands/users/create
POST /commands/users/activate
POST /commands/users/deactivate
POST /commands/users/lock
```

Benefits:

- Clear intent (not just CRUD)
- Captures business operations
- Aligns with domain ubiquitous language
- Easier to extend with new operations

Value Object Conversion

BFF Responsibility: Convert between **string URNs** and **value objects**

Schema Support:

```
value_object_conversion:
  from_string:
    - value_object_ref: vo_user_id
      from_field: "userId"
      method: "UserId.of(string)"
    - value_object_ref: vo_client_id
      from_field: "clientUrn"
      method: "ClientId.of(urn)"
  to_string:
    - value_object_ref: vo_user_id
      to_field: "userId"
      method: "userId.id()"
```

Implementation Pattern:

```
@Controller("/api/web/users")
public class UserBFFController {

    @Post("/create")
    public CreateUserResult createUser(@Body CreateUserRequest
req) {
        // String → Value Object (incoming)
        ClientId clientId = ClientId.of(req.clientUrn());

        // Create command with value objects
        var cmd = new UserCommands.CreateUserCmd(
            req.email(),
            req.userType(),
            clientId
        );

        // Delegate to application service
        UserId userId = userCommands.createUser(cmd);

        // Value Object → String (outgoing)
        return new CreateUserResult(userId.id());
    }

    // BFF-specific DTOs (separate from API command records)
    public record CreateUserRequest(
        String email,
        String userType,
        String clientUrn  // String, not ClientId
    ) {}

    public record CreateUserResult(
        String userId  // String, not UserId
    ) {}
}
```

Why Separate DTOs:

1. **Different Serialization**: BFF uses strings, API uses value objects
2. **Client-Specific Fields**: BFF may add/remove fields per client
3. **Version Independence**: BFF can evolve separately from API

4. **Clear Boundaries**: BFF layer vs. API layer separation

7.5 Data Aggregation and Transformation

Data Aggregation Strategies

BFFs support three aggregation strategies:

```
DataAggregation:
  strategy:
    enum: [parallel, sequential, conditional]
```

1. Parallel Calls (when data is independent):

```
@Get("/dashboard/{userId}")
public DashboardDTO getDashboard(@PathVariable String userId) {
    UserId id = UserId.of(userId);

    // Execute calls in parallel
    CompletableFuture<UserSummary> userFuture =
        CompletableFuture.supplyAsync(() ->
userQueries.getUserSummary(id));
    CompletableFuture<List<OrderSummary>> ordersFuture =
        CompletableFuture.supplyAsync(() ->
orderQueries.getRecentOrders(id));
    CompletableFuture<Integer> notifFuture =
        CompletableFuture.supplyAsync(() ->
notificationQueries.getUnreadCount(id));

    // Wait for all to complete
    CompletableFuture.allOf(userFuture, ordersFuture,
notifFuture).join();

    // Aggregate results
    return new DashboardDTO(
        userFuture.get(),
        ordersFuture.get(),
        notifFuture.get()
    );
}
```

Schema:

```
provides:
  data_aggregation:
    strategy: parallel
    example: "User dashboard aggregates user profile, orders, and
notifications in parallel"
```

2. Sequential Calls (when data depends on prior responses):

```
@Get("/order-details/{orderId}")
public OrderDetailsDTO getOrderDetails(@PathVariable String
orderId) {
    // Step 1: Get order
    OrderSummary order =
orderQueries.getOrder(OrderId.of(orderId));

    // Step 2: Get customer (depends on order)
    CustomerSummary customer =
customerQueries.getCustomer(order.customerId());

    // Step 3: Get shipping (depends on order)
    ShippingInfo shipping =
shippingQueries.getShippingInfo(order.shippingId());

    return new OrderDetailsDTO(order, customer, shipping);
}
```

Schema:

```
provides:
  data_aggregation:
    strategy: sequential
    example: "Order details fetches order, then customer, then
shipping sequentially"
```

3. Conditional Calls (based on business logic):

```
@Get("/user-profile/{userId}")
public UserProfileDTO getUserProfile(@PathVariable String userId)
{
    UserId id = UserId.of(userId);

    UserSummary user = userQueries.getUserSummary(id);

    // Conditional: Only fetch premium features for premium users
    if (user.isPremium()) {
        PremiumFeatures features = premiumQueries.getFeatures(id);
        return new PremiumUserProfileDTO(user, features);
    } else {
        return new StandardUserProfileDTO(user);
    }
}
```

Schema:

```
provides:
  data_aggregation:
    strategy: conditional
    example: "Premium user profile conditionally fetches premium
features"
```

Data Transformation Types

BFFs support five transformation types:

```
DataTransformation:
  transformation_type:
    enum:
      - format_conversion
      - data_enrichment
      - field_mapping
      - filtering
      - denormalization
```

1. Format Conversion

Convert domain types to client-friendly formats:

```
// Domain: Money value object
Money totalAmount = order.getTotalAmount();

// BFF Transformation: Format for client
String formattedTotal = String.format("$%.2f",
totalAmount.getAmount());
```

Schema:

```
transformations:
  - from_context: bc_order_management
    transformation_type: format_conversion
    description: "Convert Money value object to formatted string"
```

2. Data Enrichment

Add computed or derived data:

```
// Domain data
OrderSummary order = orderQueries.getOrder(orderId);

// BFF Enrichment: Add computed fields
boolean isReturnable = order.status().equals("DELIVERED") &&

order.deliveredAt().isAfter(Instant.now().minus(30, DAYS));

String estimatedDelivery = calculateEstimatedDelivery(order);

return new OrderDetailsDTO(
    order.orderId(),
    order.status(),
    isReturnable,          // Computed
    estimatedDelivery      // Computed
);
```

3. Field Mapping

Map domain field names to client-specific names:

```
// Domain: camelCase
UserSummary user = userQueries.getUserSummary(userId);

// BFF: snake_case for mobile client
Map<String, Object> response = Map.of(
    "user_id", user.userId(),          // userId → user_id
    "email_address", user.email(),     // email → email_address
    "account_status", user.status()    // status → account_status
);
```

4. Filtering

Remove unnecessary data for client:

```
// Domain: Full order with all items
Order order = orderRepository.findById(orderId);

// BFF Filtering: Mobile needs only summary
List<OrderItemSummary> filteredItems = order.getItems().stream()
    .map(item -> new OrderItemSummary(
        item.productName(),  // Keep
        item.quantity()      // Keep
        // Omit: sku, description, dimensions (not needed by
mobile)
    ))
    .collect(toList());
```

5. Denormalization

Flatten nested structures:

```
// Domain: Nested structure
Order order = orderQueries.getOrder(orderId);
Customer customer = order.getCustomer();
Address shippingAddress = order.getShippingAddress();

// BFF Denormalization: Flat structure
public record OrderSummaryDTO(
    String orderId,
    String customerName,        // Denormalized from customer
    String customerEmail,       // Denormalized from customer
    String shippingStreet,      // Denormalized from address
    String shippingCity,        // Denormalized from address
    String shippingState        // Denormalized from address
) {}
```

Schema:

```
provides:
  transformations:
    - from_context: bc_order_management
      transformation_type: denormalization
      description: "Flatten order, customer, and address into
flat DTO"
```

Client-Specific Optimizations

Schema Support:

```
provides:
  client_optimizations:
    - "Bandwidth optimization for mobile"
    - "Image size reduction for mobile networks"
    - "Pagination for large datasets"
    - "Field selection for minimal payloads"
```

Mobile Optimization Example:

```
// Mobile BFF: Minimize payload
@Get("/products")
public List<MobileProductDTO> getProducts() {
    List<Product> products = productQueries.getAllProducts();

    return products.stream()
        .map(p -> new MobileProductDTO(
            p.id(),
            p.name(),
            p.price(),
            p.thumbnailUrl()  // Low-res thumbnail for mobile
            // Omit: full description, high-res images, reviews
        ))
        .collect(toList());
}

// Web BFF: Full data
@Get("/products")
public List<WebProductDTO> getProducts() {
    List<Product> products = productQueries.getAllProducts();

    return products.stream()
        .map(p -> new WebProductDTO(
            p.id(),
            p.name(),
            p.price(),
            p.fullDescription(),    // Full description
            p.highResImages(),      // High-res images
            p.customerReviews()     // Reviews
        ))
        .collect(toList());
}
```

7.6 Integration with Application Services

BFF Delegates to Application Services

Architectural Flow:

```
Client Request
    ↓
BFF Controller (REST endpoint)
    ↓ converts DTO to Command
Application Service (use case orchestration)
    ↓ delegates business logic
Domain Aggregate (business rules)
    ↓ persisted via
Repository
    ↓
Database
```

Key Principle: BFF contains **NO business logic**, only:
- Value object conversion
- Request/response mapping
- Error handling
- Client-specific formatting

Example:

```
@Controller("/api/web/users")
public class UserWebBFFController {

    @Inject
    UserCommands commands;

    @Inject
    UserQueries queries;

    @Post("/create")
    public CreateUserResult createUser(@Body CreateUserRequest
req) {
        // 1. Convert URN string to value object (BFF
responsibility)
        ClientId clientId = ClientId.of(req.clientUrn());

        // 2. Create command with value objects
        var cmd = new UserCommands.CreateUserCmd(
            req.email(),
            req.userType(),
            req.identityProvider(),
            clientId
        );

        // 3. Delegate to application service (NO business logic
in BFF)
        UserId userId = commands.createUser(cmd);

        // 4. Convert domain ID to string for response (BFF
responsibility)
        return new CreateUserResult(userId.id());
    }

    @Get("/{userId}")
    public UserProfileDTO getUserProfile(@PathVariable String
userId) {
        // 1. Convert string to value object
        UserId id = UserId.of(userId);

        // 2. Delegate to query handler
        var userSummary = queries.getUserSummary(id);
```

```
        // 3. Map to client-specific DTO
        return new UserProfileDTO(
            userSummary.userId(),
            userSummary.email(),
            userSummary.status(),
            userSummary.userType()
        );
    }
}
```

Design Constraints:

```
bff_scope:
  responsibilities:
    business_logic:
      const: false  # BFF must NOT contain business logic
    transaction_management:
      const: false  # BFF does NOT manage transactions
    direct_persistence:
      const: false  # BFF does NOT access databases directly

  calls:
    - application_service      # BFF calls application services
    - bounded_context_api      # Or bounded context APIs
```

No Business Logic in BFF

❌ Anti-Pattern (Business Logic in BFF):

```
@Post("/create-order")
public CreateOrderResult createOrder(@Body CreateOrderRequest
req) {
    // WRONG: Business logic in BFF
    BigDecimal total = BigDecimal.ZERO;
    for (OrderItem item : req.items()) {
        total = total.add(item.price().multiply(item.quantity()));
    }

    // WRONG: Business rule validation in BFF
    if (total.compareTo(new BigDecimal("1000")) > 0) {
        throw new OrderTooLargeException();
    }

    // WRONG: Discount calculation in BFF
    if (req.items().size() > 10) {
        total = total.multiply(new BigDecimal("0.9"));
    }

    // ...
}
```

✓ **Correct Pattern (BFF Delegates):**

```
@Post("/create-order")
public CreateOrderResult createOrder(@Body CreateOrderRequest
req) {
    // Convert DTO to Command (BFF responsibility)
    var cmd = new OrderCommands.PlaceOrderCmd(
        CustomerId.of(req.customerId()),
        req.items(),
        req.shippingAddress()
    );

    // Delegate to application service (business logic there)
    OrderId orderId = orderCommands.placeOrder(cmd);

    // Convert response (BFF responsibility)
    return new CreateOrderResult(orderId.value());
}
```

7.7 Best Practices and Anti-Patterns

Best Practices

1. One BFF Per Client Type

```
# ✓ Correct
bff_scopes:
  - id: bff_web
    client_type: web
  - id: bff_ios
    client_type: mobile_ios
  - id: bff_android
    client_type: mobile_android

# ❌ Wrong: One BFF serving multiple clients
bff_scopes:
  - id: bff_mobile
    client_type: [mobile_ios, mobile_android]  # WRONG
```

2. Owned by Frontend Team

```
bff_scope:
  owned_by_team: "Web Frontend Team"  # Frontend team owns BFF
  team_type: frontend                 # Must be frontend-oriented
```

3. Client-Optimized Responses

```
// Mobile: Minimal payload
public record MobileUserDTO(
    String userId,
    String email,
    String status
) {}

// Web: Rich data
public record WebUserDTO(
    String userId,
    String email,
    String status,
    String fullAddress,
    List<OrderHistory> recentOrders,
    Map<String, Object> preferences
) {}
```

4. Aggregate from Multiple Contexts

```
// Single endpoint aggregates from 4 contexts
@Get("/dashboard/{userId}")
public DashboardDTO getDashboard(@PathVariable String userId) {
    var user = userService.getUser(userId);           // Context 1
    var orders = orderService.getOrders(userId);      // Context 2
    var payments = paymentService.getPayments(userId); // Context
3
    var notifs = notificationService.getCount(userId); // Context
4

    return new DashboardDTO(user, orders, payments, notifs);
}
```

5. Graceful Degradation

```
// Handle partial failures
try {
    orders = orderService.getOrders(userId);
} catch (ServiceUnavailableException e) {
    orders = Collections.emptyList();  // Degraded response
    log.warn("Order service unavailable", e);
}
```

Anti-Patterns to Avoid

Anti-Pattern 1: Generic BFF (Serving Multiple Client Types)

```
// Bad: Anti-Pattern
@Get("/dashboard")
public DashboardDTO getDashboard(
    @QueryValue String userId,
    @QueryValue String clientType  // WRONG: Client type parameter
) {
    if (clientType.equals("mobile")) {
        // Mobile-specific logic
    } else if (clientType.equals("web")) {
        // Web-specific logic
    }
}

// Good: Correct: Separate BFFs
@Controller("/api/web")
public class WebBFFController {
    @Get("/dashboard/{userId}")
    public WebDashboardDTO getDashboard(@PathVariable String
userId) {
        // Web-optimized response
    }
}

@Controller("/api/mobile")
public class MobileBFFController {
    @Get("/dashboard/{userId}")
    public MobileDashboardDTO getDashboard(@PathVariable String
userId) {
        // Mobile-optimized response
    }
}
```

Anti-Pattern 2: BFF Calling BFF

```
❌ WRONG:
Web BFF → Mobile BFF → Application Services

✓ CORRECT:
Web BFF → Application Services
Mobile BFF → Application Services
```

BFFs should **never call each other**. Each BFF calls application services directly.

Anti-Pattern 3: Shared DTOs Across BFFs

```
// Bad: Anti-Pattern: Shared DTO used by multiple BFFs
public record SharedUserDTO(
    String userId,
    String email,
    String status,
    // Too many fields for mobile
    String fullAddress,
    List<OrderHistory> orders,
    Map<String, Object> preferences
) {}

// Good: Correct: Client-specific DTOs
public record WebUserDTO(
    String userId,
    String email,
    String status,
    String fullAddress,
    List<OrderHistory> orders,
    Map<String, Object> preferences
) {}

public record MobileUserDTO(
    String userId,
    String email,
    String status
    // Minimal fields for mobile
) {}
```

Part III Summary

Part III covered **Tactical Implementation** across three major areas: domain modeling patterns, application layer orchestration, and client-specific optimization through BFFs.

Section 5: Tactical Design Patterns

The core building blocks for implementing rich domain models within bounded contexts:
- **Entities**: Identity-based objects with lifecycle and behavior
- **Value Objects**: Immutable, attribute-based objects enforcing domain concepts
- **Aggregates**: Consistency boundaries protecting business invariants
- **Repositories**: Persistence abstraction providing collection semantics (one per aggregate root)
- **Domain Services**: Stateless operations spanning multiple aggregates
- **Domain Events**: Immutable facts recording what happened in the domain

Critical Rules: One aggregate per transaction, reference other aggregates by ID only, value objects must be immutable, repository per aggregate root (not per entity), and events enable eventual consistency across aggregates.

Section 6: Application Layer

How application services orchestrate use cases without containing business logic:
- **Application Services**: Stateless orchestrators coordinating domain operations
- **CQRS Pattern**: Separate models for commands (state changes) and queries (data retrieval)
- **Transaction Boundaries**: One aggregate per transaction = one method per use case
- **Workflow Orchestration**: Structured steps defining use case execution flow
- **Read Models**: Denormalized views optimized for query performance
- **DTOs**: Flat, string-serialized data transfer objects for external communication

Critical Principles: Application layer contains NO business logic (only coordination), transaction boundary equals method boundary, commands modify exactly one aggregate while queries modify none, and eventual consistency between aggregates via domain events.

Section 7: Backend-for-Frontend (BFF) Pattern

How BFFs bridge domain models and client-specific needs:
- **"One Experience, One BFF"**: One BFF per client type (web, mobile, partner), not per bounded context
- **Multi-Context Aggregation**: BFFs aggregate data from multiple bounded contexts
- **Client-Specific Optimization**: Each BFF tailored to its client's data needs and constraints
- **No Business Logic**: BFF contains presentation logic only, delegates business operations to application services
- **Value Object Conversion**: BFFs handle string serialization ↔ value object translation

BFF vs API Gateway: API Gateway handles infrastructure concerns (auth, rate limiting, routing) at a single entry point, while BFFs handle client-specific aggregation and transformation at multiple entry points. Use both together—API Gateway upstream, BFFs downstream.

Schema Integration Across Part III:
- BoundedContext as root object containing all tactical patterns
- Immutability enforcement (const: true) for value objects and events
- TransactionBoundary with maxItems: 1 ensuring aggregate isolation
- Comprehensive ID type patterns enabling tool generation
- Explicit aggregate boundaries and dependencies
- Application service operations typed as command or query
- BFF scope and interface definitions with client type and team ownership
- Data aggregation strategies and transformation patterns

What's Next: Part IV examines how DDD integrates with Martin Fowler's Patterns of Enterprise Application Architecture (PoEAA), showing how Domain Model, Service Layer, Repository, Unit of Work, and other enterprise patterns complement DDD's tactical and strategic approaches.

Part IV: Integration & Patterns

8. Integration with Patterns of Enterprise Application Architecture (PoEAA)

8.1 Overview

Martin Fowler's **Patterns of Enterprise Application Architecture** (PoEAA), published in 2002, provides a comprehensive catalog of architectural patterns for enterprise applications. While Domain-Driven Design focuses primarily on domain modeling and strategic design, PoEAA covers broader concerns including presentation, data access, and session state management.

Key Insight: PoEAA and DDD are **not competing approaches**—they solve different problems and integrate naturally. PoEAA provides the architectural scaffolding; DDD provides the domain modeling guidance.

Relationship:
- **PoEAA** introduces high-level architectural patterns
- **DDD** elaborates how to implement domain logic within that architecture
- Many patterns overlap or complement each other
- Together they provide a complete architecture

Integration Benefits:
1. PoEAA provides proven persistence patterns (Repository, Data Mapper)
2. DDD refines domain modeling within PoEAA's Domain Model pattern
3. PoEAA's Service Layer aligns with DDD's Application Service
4. Both advocate for layered architecture with clear separation of concerns

8.2 Domain Logic Patterns

Domain Model (PoEAA) = Domain Layer (DDD)

PoEAA Domain Model Definition:

> "An object model of the domain that incorporates both behavior and data."

Characteristics:

- Rich objects with business logic
- Complex business rules embedded in objects
- Object-oriented approach to domain logic
- Alternative to Transaction Script or Table Module

DDD Domain Model:

- Elaborates **how** to implement the Domain Model pattern
- Provides tactical patterns: Entity, Value Object, Aggregate
- Adds strategic patterns: Bounded Context, Context Mapping
- Emphasizes Ubiquitous Language

Integration:

PoEAA introduces the **concept** of a Domain Model; DDD provides the **implementation details**.

Example Evolution:

```
// PoEAA Domain Model - general concept
public class Product {
    private BigDecimal price;
    private int inventoryCount;

    public void adjustPrice(BigDecimal newPrice) {
        // Business logic in domain object (PoEAA principle)
        this.price = newPrice;
    }

    public boolean isAvailable() {
        return inventoryCount > 0;
    }
}

// DDD elaborates with Entity, Value Object, refined design
public class Product {  // Entity (identity-based)
    private final ProductId id;  // Value Object for identity
    private Money price;   // Value Object (not BigDecimal)
    private Inventory inventory;  // Value Object (not int)

    // Business method - same PoEAA idea, more refined
    public void adjustPrice(Money newPrice, PricingPolicy policy)
{
        // Validation (business rule)
        if (!policy.allows(this, newPrice)) {
            throw new InvalidPriceException(
                "Price " + newPrice + " violates pricing policy"
            );
        }

        Money oldPrice = this.price;
        this.price = newPrice;

        // Domain Event (DDD addition)
        DomainEvents.raise(new ProductPriceChanged(
            this.id,
            oldPrice,
            newPrice,
            Instant.now()
        ));
```

```
    }

    public boolean isAvailable() {
        return inventory.hasStock();  // Delegate to Value Object
    }
}
```

PoEAA Contribution: Rich domain objects with behavior
DDD Contribution: Entity vs. Value Object distinction, immutability, domain events

Service Layer (PoEAA) = Application Service (DDD)

PoEAA Service Layer Definition:

> "Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation."

Characteristics:
- Defines application boundary
- Coordinates domain objects
- Manages transactions
- Orchestrates responses
- Thin layer—no business logic

DDD Application Service:
- Essentially the same pattern
- Orchestrates use cases
- Transaction management
- Delegates to domain layer
- No business logic (critical rule)

Differences:
- DDD distinguishes Application Services from Domain Services
- DDD emphasizes CQRS (commands vs. queries)
- DDD adds event publication

Example Comparison:

```
// PoEAA Service Layer
public class OrderService {
    private OrderRepository orderRepo;
    private CustomerRepository customerRepo;

    @Transactional
    public void placeOrder(OrderRequest request) {
        // Orchestration
        Customer customer = customerRepo.find(request.customerId);
        Order order = new Order(customer);
        order.addItems(request.items);
        orderRepo.save(order);
    }
}

// DDD Application Service - same pattern, refined
public class PlaceOrderService {
    private OrderRepository orderRepo;
    private CustomerRepository customerRepo;
    private DomainEventPublisher eventPublisher;

    @Transactional
    public OrderId execute(PlaceOrderCommand command) {
        // 1. Load aggregates
        Customer customer =
customerRepo.findById(command.customerId())
            .orElseThrow(() -> new CustomerNotFoundException());

        // 2. Execute domain logic (in domain layer)
        Order order = customer.createOrder();
        order.addItems(command.items());

        // 3. Persist
        orderRepo.save(order);

        // 4. Publish events (DDD addition)
        eventPublisher.publish(new OrderPlaced(order.getId(),
customer.getId(), Instant.now()));

        return order.getId();
```

```
    }
}
```

Key Similarity: Both orchestrate domain objects without containing business logic
DDD Refinement: Explicit command objects, event publication, clearer boundaries

8.3 Data Source Patterns

Repository (PoEAA) vs Repository (DDD)

PoEAA Repository Definition:

> "Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects."

Characteristics:
- Collection metaphor (`add()`, `remove()`, `find()`)
- Encapsulates data access logic
- Can use Specification pattern for queries
- One repository per entity (or per type)

DDD Repository:
- **One repository per Aggregate Root** (not per entity)
- Loads and reconstitutes entire aggregate atomically
- Collection metaphor (same as PoEAA)
- Domain-oriented query methods

Key Difference:

Aspect	PoEAA Repository	DDD Repository
Scope	Per entity or per type	Per Aggregate Root only
Granularity	Can be fine-grained	Coarse-grained (whole aggregate)
Load Strategy	May lazy-load relationships	Eager-loads entire aggregate
Purpose	Abstract data access	Aggregate lifecycle management

Example:

```
// Both use collection-like interface
public interface OrderRepository {
    void save(Order order);
    void remove(Order order);
    Optional<Order> findById(OrderId id);
    List<Order> findByCustomer(CustomerId customerId);
}

// DDD adds: One repo per aggregate root, not per entity
// Bad: Don't create: OrderLineRepository
// Good: OrderLine is part of Order aggregate

// PoEAA adds: Specification pattern for complex queries
public interface OrderRepository {
    List<Order> find(Specification<Order> spec);  // PoEAA pattern
}

// Usage:
Specification<Order> spec = new
RecentOrdersSpecification(customerId, days(30));
List<Order> recentOrders = orderRepo.find(spec);
```

Data Mapper (PoEAA)

PoEAA Data Mapper Definition:

> "A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself."

Purpose:
- Keep domain objects independent of persistence
- Handle translation between domain and database schemas
- Isolate database details from domain

DDD Integration:
Data Mapper is the **implementation strategy** for DDD Repositories.

Example:

```
// Data Mapper implementing DDD Repository
@Repository
public class JpaOrderRepository implements OrderRepository {
    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public void save(Order order) {
        OrderEntity entity = toEntity(order);  // Data Mapper
translation
        entityManager.merge(entity);
    }

    @Override
    public Optional<Order> findById(OrderId id) {
        OrderEntity entity = entityManager.find(
            OrderEntity.class,
            id.value()
        );

        return Optional.ofNullable(entity)
            .map(this::toDomain);  // Data Mapper translation
    }

    // Data Mapper: Domain → Entity
    private OrderEntity toEntity(Order order) {
        OrderEntity entity = new OrderEntity();
        entity.setId(order.getId().value());
        entity.setCustomerId(order.getCustomerId().value());
        entity.setStatus(order.getStatus().name());
        // ... map other attributes

        // Key: Aggregate children mapped recursively
        for (OrderLine line : order.getLines()) {
            OrderLineEntity lineEntity = new OrderLineEntity();
            lineEntity.setProductId(line.getProductId().value());
            lineEntity.setQuantity(line.getQuantity().amount());

lineEntity.setUnitPrice(line.getUnitPrice().getAmount());
            entity.addLine(lineEntity);
        }
```

```
        return entity;
    }

    // Data Mapper: Entity → Domain
    private Order toDomain(OrderEntity entity) {
        Order order = new Order(
            new OrderId(entity.getId()),
            new CustomerId(entity.getCustomerId())
        );
        // ... reconstruct other attributes

        // Key: Aggregate children mapped recursively
        for (OrderLineEntity lineEntity : entity.getLines()) {
            order.addLine(
                new ProductId(lineEntity.getProductId()),
                Quantity.of(lineEntity.getQuantity()),
                Money.of(lineEntity.getUnitPrice(),
Currency.getInstance(entity.getCurrency()))
            );
        }

        return order;
    }
}
```

Pattern Summary: Data Mappers translate between domain aggregates (Order with OrderLines) and persistence entities, keeping the domain model pure and persistence-ignorant. The key insight is recursive mapping of aggregate children while maintaining aggregate boundaries.

Benefits:
- Domain model remains pure (no JPA annotations)
- Can use different persistence models
- Easier to test domain logic
- Clear separation of concerns

Unit of Work (PoEAA)

PoEAA Unit of Work Definition:

> "Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems."

Purpose:
- Track changes during a transaction
- Coordinate database writes
- Optimize database access (batch updates)
- Handle concurrency

DDD Integration:

Unit of Work is typically handled by:
1. **ORM Frameworks** (JPA EntityManager, Hibernate Session)
2. **Application Service Transaction Boundaries**

Example:

```
// Unit of Work implicit in JPA EntityManager
@Service
public class TransferMoneyService {
    @PersistenceContext
    private EntityManager entityManager;  // Unit of Work

    private AccountRepository accountRepo;

    @Transactional
    public void execute(TransferMoneyCommand command) {
        // EntityManager tracks all loaded entities (Unit of Work)
        Account fromAccount =
accountRepo.findById(command.fromAccountId());
        Account toAccount =
accountRepo.findById(command.toAccountId());

        // Domain logic
        fromAccount.debit(command.amount());
        toAccount.credit(command.amount());

        // EntityManager automatically detects changes and
persists
        // No explicit save() call needed

        // Transaction commits - Unit of Work flushes changes
    }
}
```

PoEAA Contribution: Pattern for coordinating database writes
DDD + ORM: Typically handled transparently by framework

8.4 Presentation Patterns

Presentation Model / View Models

PoEAA Presentation Model Definition:

> "Represent the state and behavior of the presentation independently of the GUI controls used in the interface."

Purpose:

- Separate presentation state from view
- Make presentation logic testable
- Support multiple views of same data

DDD Integration:

Presentation Models or **View Models** sit in the **Presentation Layer** and transform domain models for UI consumption.

Example:

```
// Domain Model (rich with behavior)
public class Order {
    private OrderId id;
    private CustomerId customerId;
    private OrderStatus status;
    private List<OrderLine> lines;
    private Money totalAmount;

    // Domain methods
    public void addLine(ProductId productId, Quantity qty, Money
unitPrice) {
        // Business logic
    }

    public Money calculateTotal() {
        // Business calculation
    }
}

// View Model for Order (Presentation Layer)
public class OrderViewModel {
    // UI-friendly properties
    private String orderId;
    private String customerId;
    private String customerName;
    private String statusDisplay;
    private String formattedTotal;
    private List<OrderLineViewModel> lines;

    // Factory method: Domain → View Model
    public static OrderViewModel from(Order order, Customer
customer) {
        OrderViewModel vm = new OrderViewModel();
        vm.orderId = order.getId().value();
        vm.customerName = customer.getName().fullName();
        vm.statusDisplay = order.getStatus().displayName();  //
"Pending" not "PENDING"
        vm.formattedTotal =
order.getTotalAmount().format(Locale.US);  // "$125.00"
        // ... map other properties and nested line items
        return vm;
```

```
    }

    // UI-specific computed properties
    public boolean isEditable() {
        return status == OrderStatus.PENDING || status ==
OrderStatus.DRAFT;
    }
}
```

Benefits:

- Domain model remains focused on business logic
- View Model handles presentation concerns (formatting, localization)
- Testable without UI framework
- Supports multiple views (web, mobile, reports)

MVC Integration

PoEAA MVC:

- **Model**: Holds data and logic
- **View**: Displays model
- **Controller**: Handles input, updates model

DDD Integration:

```
@Controller
public class OrderController {
    private PlaceOrderService placeOrderService;  // Application
Service
    private OrderRepository orderRepo;
    private CustomerRepository customerRepo;

    // POST - Command (modifies state)
    @PostMapping("/orders")
    public String placeOrder(@ModelAttribute PlaceOrderForm form)
{
        // Controller → Application Service (DDD)
        PlaceOrderCommand command = new PlaceOrderCommand(
            new CustomerId(form.getCustomerId()),
            form.getItems()
        );

        OrderId orderId = placeOrderService.execute(command);

        return "redirect:/orders/" + orderId.value();
    }

    // GET - Query (reads state)
    @GetMapping("/orders/{id}")
    public String viewOrder(@PathVariable String id, Model model)
{
        // Controller → Repository (query side)
        Order order = orderRepo.findById(new OrderId(id))
            .orElseThrow(() -> new OrderNotFoundException(id));

        Customer customer =
customerRepo.findById(order.getCustomerId())
            .orElseThrow();

        // Domain → View Model transformation
        OrderViewModel viewModel = OrderViewModel.from(order,
customer);
        model.addAttribute("order", viewModel);

        return "order-detail";  // View name
```

```
    }
}
```

MVC in DDD Context:
- **Model** = Domain Model + View Models
- **View** = Templates using View Models
- **Controller** = Delegates to Application Services

8.5 Layered Architecture Integration

The patterns we've explored so far—Domain Model, Service Layer, Repository, Data Mapper, and Presentation Model—don't exist in isolation. They form an integrated architecture where each pattern occupies a specific layer with defined responsibilities and dependencies. Both PoEAA and DDD organize these patterns using layered architecture, which provides the structural foundation for how patterns collaborate.

Four-Layer Architecture (PoEAA + DDD)

Both PoEAA and DDD advocate for layered architecture with clear separation of concerns:

Dependency Direction:

```
Presentation → Application → Domain
                  ↓
         Infrastructure ← Domain (interfaces)
```

Key Principle: Dependencies point inward. Domain layer is independent.

Pattern Allocation by Layer

Presentation Layer:
- **PoEAA Patterns**: MVC, MVP, MVVM, Presentation Model, Template View
- **DDD Patterns**: View Models using Ubiquitous Language
- **Responsibilities**: User interface, user input handling, presentation logic

Application Layer:
- **PoEAA Patterns**: Service Layer, Transaction Script (for simple cases)
- **DDD Patterns**: Application Services, Command Handlers, Query Handlers
- **Responsibilities**: Use case orchestration, transaction management, DTO transformation

Domain Layer:
- **PoEAA Patterns**: Domain Model (general concept)
- **DDD Patterns**: Entity, Value Object, Aggregate, Domain Service, Domain Event
- **Responsibilities**: Business logic, invariant enforcement, domain rules

Infrastructure Layer:
- **PoEAA Patterns**: Repository (implementation), Data Mapper, Unit of Work, Identity Map, Lazy Load
- **DDD Patterns**: Repository implementations, Anti-Corruption Layer
- **Responsibilities**: Persistence, external services, technical infrastructure

8.6 Pattern Combinations and Best Practices

Repository + Data Mapper

Pattern Combination:
- **Repository** (interface in Domain Layer)
- **Data Mapper** (implementation in Infrastructure Layer)

```
// Domain Layer: Repository interface
public interface CustomerRepository {
    void save(Customer customer);
    Optional<Customer> findById(CustomerId id);
    List<Customer> findByEmail(Email email);
}

// Infrastructure Layer: Data Mapper implementation
@Repository
public class JpaCustomerRepository implements CustomerRepository {
    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public void save(Customer customer) {
        CustomerEntity entity = toEntity(customer);  // Data
Mapper
        entityManager.merge(entity);
    }

    @Override
    public Optional<Customer> findById(CustomerId id) {
        CustomerEntity entity =
entityManager.find(CustomerEntity.class, id.value());
        return
Optional.ofNullable(entity).map(this::toDomain);  // Data Mapper
    }

    @Override
    public List<Customer> findByEmail(Email email) {
        // ... query implementation with Data Mapper
transformation
    }

    // Data Mapper methods
    private CustomerEntity toEntity(Customer customer) {
        CustomerEntity entity = new CustomerEntity();
        entity.setId(customer.getId().value());
        entity.setEmail(customer.getEmail().value());
        entity.setName(customer.getName().fullName());
        entity.setStatus(customer.getStatus().name());
```

```
        return entity;
    }

    private Customer toDomain(CustomerEntity entity) {
        return new Customer(
            new CustomerId(entity.getId()),
            Email.of(entity.getEmail()),
            PersonName.parse(entity.getName()),
            CustomerStatus.valueOf(entity.getStatus())
        );
    }
}
```

Service Layer + Domain Model + Repository

Complete Pattern Combination:

```
// Application Layer: Service Layer
@Service
public class TransferMoneyService {
    private AccountRepository accountRepo;  // Repository
    private TransferPolicy transferPolicy;  // Domain Service

    @Transactional  // Unit of Work
    public void execute(TransferMoneyCommand command) {
        // 1. Load aggregates via Repository
        Account fromAccount =
accountRepo.findById(command.fromAccountId())
            .orElseThrow(() -> new AccountNotFoundException());
        Account toAccount =
accountRepo.findById(command.toAccountId())
            .orElseThrow(() -> new AccountNotFoundException());

        // 2. Validate using Domain Service
        transferPolicy.validateTransfer(fromAccount, toAccount,
command.amount());

        // 3. Execute domain logic (Domain Model)
        fromAccount.debit(command.amount());
        toAccount.credit(command.amount());

        // 4. Save via Repository
        accountRepo.save(fromAccount);
        accountRepo.save(toAccount);

        // Note: This violates DDD's "one aggregate per
transaction" rule
        // In strict DDD, should use domain events for eventual
consistency
    }
}
```

Complementary Patterns Summary

PoEAA Pattern	DDD Equivalent/ Integration	Layer	Notes
Domain Model	Domain Layer (Entities, Value Objects, Aggregates)	Domain	DDD elaborates implementation
Service Layer	Application Service	Application	Same concept, DDD adds CQRS
Repository	Repository (per Aggregate)	Domain (interface), Infrastructure (impl)	DDD adds aggregate scoping
Data Mapper	Repository Implementation	Infrastructure	Implements persistence
Unit of Work	Transaction Management	Application	Often handled by ORM
Identity Map	Entity Caching	Infrastructure	ORM feature
Lazy Load	Aggregate Loading Strategy	Infrastructure	Be careful with aggregate boundaries
Transaction Script	Simple Application Service	Application	For simple cases
Table Module	Not used in DDD	N/A	DDD prefers Domain Model
Presentation Model	View Model	Presentation	Transforms domain for UI
MVC / MVP / MVVM	Presentation Patterns	Presentation	UI organization
Front Controller	API Gateway / BFF	Presentation	Entry point for requests

Integration Guidelines

Use PoEAA Patterns For:

1. **Presentation Layer Organization**
   - MVC, MVVM for UI structure
   - Presentation Model for view logic
   - Front Controller for request routing
2. **Persistence Strategies**
   - Data Mapper for OR mapping
   - Repository implementation patterns
   - Unit of Work for transaction coordination
3. **Session State Management**
   - Identity Map for caching
   - Lazy Load for performance
   - Optimistic Offline Lock for concurrency
4. **Service Layer Orchestration**
   - Transaction coordination
   - DTO patterns
   - Remote Facade for distribution

Use DDD Patterns For:

1. **Domain Modeling**
   - Entity vs. Value Object distinction
   - Aggregate design
   - Domain Events
   - Domain Services
2. **Strategic Design**
   - Bounded Contexts
   - Context Mapping
   - Ubiquitous Language
   - Anti-Corruption Layers
3. **Domain Logic Organization**
   - Where business rules live
   - Invariant enforcement
   - State transitions
   - Domain event publication

4. **Team Organization**
   - Context ownership
   - Team autonomy
   - Conway's Law application

Integration Principles:

1. Clear Separation:
- Domain layer knows nothing of persistence (PoEAA patterns)
- Presentation uses domain model but doesn't contain business logic
- Application layer coordinates: domain model + PoEAA transaction patterns

2. Dependency Direction:

```
Presentation → Application → Domain
                    ↓
            Infrastructure (PoEAA patterns)
```

3. Pattern Selection:
- Start with DDD for domain modeling
- Use PoEAA for infrastructure concerns
- Combine naturally at integration points

4. Avoid Conflicts:
- Don't use Table Module (conflicts with DDD's rich domain model)
- Don't use Active Record (entities should be persistence-ignorant)
- Don't put business logic in Service Layer (belongs in Domain Layer)

Part IV Summary (Section 8)

Section 8 covered **Integration with Patterns of Enterprise Application Architecture (PoEAA)**:

Key Insights:
- PoEAA and DDD are complementary, not competing
- PoEAA provides architectural scaffolding; DDD provides domain modeling

guidance
- Many patterns overlap or align naturally
- Together they provide a complete architecture

Pattern Mappings:
- **Domain Model (PoEAA) = Domain Layer (DDD)**: DDD elaborates with Entity, Value Object, Aggregate
- **Service Layer (PoEAA) = Application Service (DDD)**: Same pattern, DDD adds CQRS and events
- **Repository (PoEAA) ≈ Repository (DDD)**: DDD scopes to aggregate roots
- **Data Mapper (PoEAA)**: Implements DDD Repositories
- **Unit of Work (PoEAA)**: Handled by ORM and transaction management

Layered Architecture:
- **Presentation**: PoEAA MVC/MVVM patterns + DDD View Models
- **Application**: PoEAA Service Layer = DDD Application Service
- **Domain**: PoEAA Domain Model elaborated by DDD tactical patterns
- **Infrastructure**: PoEAA persistence patterns implement DDD repositories

Best Practices:
- Use PoEAA for presentation and infrastructure concerns
- Use DDD for domain modeling and strategic design
- Combine naturally at integration points
- Maintain clear separation of concerns
- Follow dependency inversion principle

What's Next: Section 9 covers **Schema Reference**—comprehensive documentation of the strategic, tactical, and domain stories schemas, including ID conventions, validation rules, and schema examples.

Part V: Reference

9. Schema Reference Guide

9.1 Overview

Domain-Driven Design patterns can be formalized as **machine-readable definitions** across three levels of modeling:

1. **Strategic Schema** (`strategic-ddd.schema.yaml`)
   - System architecture
   - Domains and Bounded Contexts
   - Context Mappings
   - BFF (Backend-for-Frontend) Scopes and Interfaces
2. **Tactical Schema** (`tactical-ddd.schema.yaml`)
   - Aggregates, Entities, Value Objects
   - Repositories, Domain Services
   - Application Services (CQRS)
   - Command/Query Interfaces (Knight pattern)
   - Domain Events
3. **Domain Stories Schema** (`domain-stories-schema.yaml`)
   - Domain Storytelling artifacts
   - Actors, Activities, Work Objects
   - Commands, Queries, Events, Policies
   - Mappings to Tactical concepts

Purpose of Schema Formalization:
- **Validation**: Ensure models are structurally correct
- **Tool Support**: Enable IDE assistance, code generation, LLM reasoning
- **Documentation**: Machine-readable domain knowledge
- **Consistency**: Codify DDD best practices in a verifiable form

9.2 Strategic Schema

Location: `/domains/ddd/schemas/strategic-ddd.schema.yaml`

Having established the three-layer schema architecture, let's examine the strategic schema in detail. This schema captures system-level architecture decisions, domain boundaries, and integration patterns that shape your entire DDD implementation.

9.2.1 Root Object: System

The `System` is the root container representing the entire software system.

Structure:

```
system:
  id: sys_<name>           # System identifier
  name: string             # System name
  description: string      # What this system does
  version: string          # System version

  # Full objects embedded (not just IDs)
  domains: [Domain]
  bounded_contexts: [BoundedContext]
  context_mappings: [ContextMapping]
  bff_scopes: [BFFScope]
  bff_interfaces: [BFFInterface]
```

Benefits:
- **Single source of truth**: All strategic definitions in one structure
- **Validation clarity**: Schema can validate cross-references
- **Tool support**: IDEs can navigate entire system structure
- **Generation**: Code generators have complete context

Example:

```
system:
  id: sys_ecommerce
  name: "E-Commerce System"
  version: "2.0.0"
  domains:
    - id: dom_sales
      name: "Sales"
      type: core
      strategic_importance: critical
  bounded_contexts:
    - id: bc_order_management
      name: "Order Management"
      domain_ref: dom_sales
```

9.2.2 Domain Type

Purpose: Represents a sphere of knowledge and activity.

Structure:

```
Domain:
  id: dom_<name>           # Required, pattern: ^dom_[a-z0-9_]+$
  name: string             # Required, from ubiquitous language
  type:                    # Required
    enum: [core, supporting, generic]
  strategic_importance:
    enum: [critical, important, standard, low]
  bounded_contexts: [BcId]  # ID references
  investment_strategy: string
  notes: string
```

Domain Types:
- **Core**: Your competitive advantage, invest heavily
- **Supporting**: Necessary but not differentiating
- **Generic**: Buy off-the-shelf solutions

Strategic Importance:
- **Critical**: Mission-critical, best team
- **Important**: High priority, strong team
- **Standard**: Normal priority, adequate team
- **Low**: Minimal investment, outsource candidate

9.2.3 Bounded Context Type

Purpose: Explicit boundary within which a domain model is defined.

Structure:

```
BoundedContext:
  id: bc_<name>            # Required, pattern: ^bc_[a-z0-9_]+$
  name: string             # Required
  domain_ref: DomId        # Required (parent domain)
  description: string

  ubiquitous_language:
    glossary:
      - term: string
        definition: string
        examples: [string]

  team_ownership: string

  # ID references to tactical concepts
  aggregates: [agg_*]
  repositories: [repo_*]
  domain_services: [svc_dom_*]
  application_services: [svc_app_*]
  domain_events: [evt_*]
```

Key Points:
- Must belong to exactly one domain (`domain_ref`)
- Contains ubiquitous language glossary specific to context
- Lists tactical patterns (aggregates, services, etc.) by ID
- Team ownership tracks Conway's Law alignment

9.2.4 Context Mapping Type

Purpose: Defines relationship between two bounded contexts.

Structure:

```
ContextMapping:
  id: cm_<source>_to_<target>  # Required, pattern: ^cm_[a-z0-9_]
+_to_[a-z0-9_]+$
  name: string                  # Required
  upstream_context: BcId        # Required
  downstream_context: BcId      # Required
  relationship_type:            # Required
    enum:
      - partnership
      - shared_kernel
      - customer_supplier
      - conformist
      - anti_corruption_layer
      - open_host_service
      - published_language
      - separate_ways
      - big_ball_of_mud

  integration_pattern: string
  translation_map: object
  shared_elements: [string]    # For shared_kernel
  acl_details: object          # For anti_corruption_layer
  notes: string
```

Relationship Types (see Section 2.5 for detailed descriptions):
- **Partnership**: Mutual cooperation
- **Shared Kernel**: Shared code/model
- **Customer/Supplier**: Upstream serves downstream
- **Conformist**: Downstream accepts upstream model
- **Anti-Corruption Layer**: Downstream protects itself
- **Open Host Service**: Upstream provides protocol
- **Published Language**: Standardized exchange format
- **Separate Ways**: No integration
- **Big Ball of Mud**: Legacy complexity

The `name` field provides a descriptive name for the relationship, making context mappings easier to understand (e.g., "Order to Payment Integration", "User Profile Shared Model").

9.2.5 BFF Scope Type

Purpose: Backend-for-Frontend serving exactly ONE client type.

Core Principle: **"One Experience, One BFF"** — BFFs are scoped by CLIENT TYPE (web, iOS, Android), NOT by bounded contexts.

Structure:

```
BFFScope:
  id: bff_<client_type>    # Required, pattern: ^bff_[a-z0-9_]+$
  name: string             # Required, pattern: ^[A-Z][a-zA-Z]
+BFF$

  client_type:             # Required (exactly one)
    enum: [web, mobile_ios, mobile_android, desktop, partner_api,
iot, tablet]

  serves_interface: string # Required

  aggregates_from_contexts: [BcId]  # Required, minItems: 1
(multiple contexts!)

  owned_by_team: string    # Required (Conway's Law)
  team_type:
    enum: [frontend, mobile, partner_integration]

  provides:
    endpoints: [BFFEndpoint]
    data_aggregation:
      strategy: [parallel, sequential, conditional]
    transformations: [DataTransformation]
    client_optimizations: [string]

  responsibilities:
    data_aggregation: true           # const: true
    client_specific_orchestration: true  # const: true
    presentation_logic: true         # const: true
    format_translation: true         # const: true
    business_logic: false            # const: false (MUST NOT)
    transaction_management: false    # const: false (MUST NOT)
    direct_persistence: false        # const: false (MUST NOT)
```

Key Design Constraints:

- `business_logic: const: false` — BFFs MUST NOT contain business logic
- `transaction_management: const: false` — BFFs delegate to services
- `direct_persistence: const: false` — BFFs call services, not databases
- `aggregates_from_contexts: minItems: 1` — BFF aggregates from MULTIPLE contexts

9.2.6 BFF Interface Type

Purpose: Concrete REST API implementation for a BFF scope accessing a specific bounded context.

Structure:

```
BFFInterface:
  id: bff_if_<context>_<client_type>  # Required, pattern:
^bff_if_[a-z0-9_]+$
  name: string                         # Required
  bff_scope_ref: BffId                 # Required (belongs to
which BFF)
  primary_bounded_context_ref: BcId    # Required
  additional_context_refs: [BcId]      # Optional (multi-context
aggregation)
  base_path: string                    # Required, pattern: ^/[a-
z0-9-/]+$

  endpoints: [BFFInterfaceEndpoint]

  value_object_conversion:
    from_string:  # String/URN → Value Object
      - value_object_ref: vo_user_id
        from_field: "userId"
        method: "UserId.of(string)"
    to_string:    # Value Object → String/URN
      - value_object_ref: vo_user_id
        to_field: "userId"
        method: "userId.id()"

  execution_model:
    enum: [blocking, async, reactive]

  error_handling:
    strategy:
      enum: [fail_fast, graceful_degradation, partial_response]
```

BFFInterfaceEndpoint:

```
BFFInterfaceEndpoint:
  path: string             # e.g., "/users/{id}"
  method:
    enum: [GET, POST, PUT, PATCH, DELETE]
  operation_type:
    enum: [command, query, action]  # CQRS alignment

  delegates_to_commands: [cmd_*]   # ID references
  delegates_to_queries: [qry_*]    # ID references

  request_dto: RequestDTO
  response_dto: ResponseDTO

  aggregates_data_from: [BcId]  # Multiple contexts
```

Key Points:
- **Value Object Conversion**: BFFs convert between strings (API) and value objects (domain)
- **Multi-Context Aggregation**: Can aggregate from multiple bounded contexts
- **CQRS Alignment**: Endpoints marked as command/query/action
- **Delegation**: Calls application services, doesn't implement logic

9.2.7 Strategic Schema ID Patterns

Type	Prefix	Pattern	Example
System	`sys_`	`^sys_[a-z0-9_]+$`	`sys_ecommerce`
Domain	`dom_`	`^dom_[a-z0-9_]+$`	`dom_sales`
Bounded Context	`bc_`	`^bc_[a-z0-9_]+$`	`bc_order_management`
Context Mapping	`cm_`	`^cm_[a-z0-9_]+_to_[a-z0-9_]+$`	`cm_order_to_payment`
BFF Scope	`bff_`	`^bff_[a-z0-9_]+$`	`bff_web`, `bff_ios`
BFF Interface	`bff_if_`	`^bff_if_[a-z0-9_]+$`	`bff_if_user_web`

9.2.8 Strategic Design Rules

The strategic patterns require:

1. **Bounded Context Has Domain**:
   - Every `BoundedContext` must have a valid `domain_ref`
2. **Context Mapping Different Contexts**:
   - `upstream_context` ≠ `downstream_context`
3. **BFF Serves One Client Type**:
   - `client_type` is a single enum value, not an array
4. **BFF Aggregates Multiple Contexts**:
   - `aggregates_from_contexts` has `minItems: 1`
5. **BFF No Business Logic**:
   - `responsibilities.business_logic: const: false` (required)

9.3 Tactical Schema

Location: `/domains/ddd/schemas/tactical-ddd.schema.yaml`

9.3.1 Root Object: BoundedContext

`BoundedContext` is the root object for tactical patterns.

Structure:

```
bounded_context:
  id: bc_<name>            # Required, pattern: ^bc_[a-z0-9_]+$
  name: string             # Required
  domain_ref: DomId        # Required (links to strategic schema)
  description: string

  # Full objects embedded (not just IDs)
  aggregates: [Aggregate]
  entities: [Entity]
  value_objects: [ValueObject]
  repositories: [Repository]
  domain_services: [DomainService]
  application_services: [ApplicationService]
  command_interfaces: [CommandInterface]
  query_interfaces: [QueryInterface]
  domain_events: [DomainEvent]
```

Benefits:
- **Scoping**: All tactical patterns scoped to one bounded context
- **Implicit Context**: Child types don't need `bounded_context_ref` (implicit from parent)
- **Validation**: Schema can validate references within context
- **Generation**: Code generators have context-scoped structure

9.3.2 Aggregate Type

Purpose: Cluster of entities and value objects with defined consistency boundary.

Structure:

```
Aggregate:
  id: agg_<name>           # Required, pattern: ^agg_[a-z0-9_]+$
  name: string             # Required
  root_ref: EntId          # Required (aggregate root entity)

  entities: [EntId]        # ID references (including root)
  value_objects: [VoId]    # ID references

  consistency_rules: [string]
  invariants: [string]

  size_estimate:
    enum: [small, medium, large]  # Prefer small!
```

Vaughn Vernon's 4 Rules (enforced by usage patterns):
1. Protect true invariants (`invariants` array)
2. Design small aggregates (`size_estimate: small`)
3. Reference other aggregates by ID only (ID references, not objects)
4. Update other aggregates via domain events (`publishes_events`)

Key Points:
- One aggregate root (`root_ref` must be an entity with `is_aggregate_root: true`)
- Contains entities and value objects by ID reference
- Defines consistency boundary (`invariants`)
- Prefer small aggregates (easier to maintain invariants)

9.3.3 Entity Type

Purpose: Object with unique identity and lifecycle.

Structure:

```
Entity:
  id: ent_<name>           # Required, pattern: ^ent_[a-z0-9_]+$
  name: string             # Required
  aggregate_ref: AggId     # Required (which aggregate)
  is_aggregate_root: boolean  # true if this is root

  identity_field: string   # Required (e.g., "orderId")
  identity_generation:
    enum: [user_provided, auto_generated, derived, external]

  attributes: [Attribute]
  business_methods: [Method]
  invariants: [string]
```

Identity Generation:
- **user_provided**: User supplies ID (e.g., email)
- **auto_generated**: Application generates UUID
- **derived**: Computed from other attributes
- **external**: Provided by external system

Key Points:
- Must have unique identity (`identity_field`)
- Belongs to exactly one aggregate (`aggregate_ref`)
- Only one entity per aggregate can be root (`is_aggregate_root: true`)

9.3.4 Value Object Type

Purpose: Immutable object defined by its attributes.

Structure:

```
ValueObject:
  id: vo_<name>            # Required, pattern: ^vo_[a-z0-9_]+$
  name: string             # Required
  description: string

  attributes: [Attribute]
  validation_rules: [string]
  equality_criteria: [string]

  immutability: true       # const: true (MUST be true, schema-
enforced)
```

CRITICAL: `immutability: const: true`

This design constraint **requires** that all value objects MUST be immutable. Value objects with `immutability: false` violate the pattern.

Example:

```
value_objects:
  - id: vo_money
    name: "Money"
    attributes:
      - name: amount
        type: decimal
      - name: currency
        type: string
    validation_rules:
      - "Amount must be non-negative"
      - "Currency must be valid ISO 4217 code"
    equality_criteria:
      - amount
      - currency
    immutability: true  # Required
```

9.3.5 Repository Type

Purpose: Persistence abstraction for aggregates.

Structure:

```
Repository:
  id: repo_<name>          # Required, pattern: ^repo_[a-z0-9_]+$
  name: string             # Required
  aggregate_ref: AggId     # Required (one repo per aggregate)

  interface_methods: [Method]
  persistence_strategy: string
```

Key Rule: **One repository per aggregate root**, NOT per entity.

Typical Interface Methods:

```
interface_methods:
  - name: "save"
    parameters:
      - name: "order"
        type: "Order"
    returns: "void"

  - name: "findById"
    parameters:
      - name: "orderId"
        type: "OrderId"
    returns: "Optional<Order>"

  - name: "delete"
    parameters:
      - name: "order"
        type: "Order"
    returns: "void"
```

9.3.6 Application Service Type

Purpose: Orchestrates use case execution, manages transactions.

Structure:

```
ApplicationService:
  id: svc_app_<name>       # Required, pattern: ^svc_app_[a-z0-9_]
+$
  name: string             # Required, pattern: ^[A-Z][a-zA-Z]
+ApplicationService$
  description: string

  implements_commands: [CmdId]  # Command interfaces
  implements_queries: [QryId]   # Query interfaces

  operations: [ApplicationServiceOperation]

  dependencies:
    repositories: [RepoId]
    domain_services: [SvcDomId]
    event_publishers: [...]

  characteristics:
    stateless: true                 # const: true (MUST be
stateless)
    contains_business_logic: false   # const: false (MUST NOT
contain)
    manages_transactions: true       # const: true
    coordinates_aggregates: true     # const: true
    publishes_events: true           # default: true
    performs_authorization: true     # default: true
```

Design Constraints:
- `stateless: const: true` — Application services MUST be stateless
- `contains_business_logic: const: false` — NO business logic allowed
- `manages_transactions: const: true` — Transaction responsibility

ApplicationServiceOperation:

```
ApplicationServiceOperation:
  name: string             # e.g., "createUser", "placeOrder"
  type:
    enum: [command, query]  # CQRS separation
  description: string
  parameters: [Parameter]
  returns: string

  transaction_boundary:
    is_transactional: boolean  # true for commands, false for
queries
    modifies_aggregates: [AggId]  # maxItems: 1 (ONE AGGREGATE
RULE)
    consistency_type:
      enum: [transactional, eventual]

  workflow:
    validates_input: boolean
    loads_aggregates: [AggId]
    invokes_domain_operations: [string]
    invokes_domain_services: [SvcDomId]
    persists_aggregates: boolean
    publishes_events: [EvtId]
    returns_dto: string
```

TransactionBoundary Constraint:

```
TransactionBoundary:
  modifies_aggregates:
    maxItems: 1  # Enforces Vaughn Vernon's "one aggregate per
transaction" rule
```

This design constraint ensures commands modify at most ONE aggregate per transaction.

9.3.7 Command Interface Type (Knight Pattern)

Purpose: Defines commands as nested records inside interface contracts.

Structure:

```
CommandInterface:
  id: cmd_<name>           # Required, pattern: ^cmd_[a-z0-9_]+$
  name: string             # Required, pattern: ^[A-Z][a-zA-Z]
+Commands$
  aggregate_ref: AggId     # Primary aggregate
  description: string

  command_records: [CommandRecord]  # Nested records

  immutability: true       # const: true (commands are immutable)
  layer: "api"            # const: "api"
```

CommandRecord (nested inside interface):

```
CommandRecord:
  record_name: string      # Required, pattern: ^[A-Z][a-zA-Z]
+Cmd$
  intent: string           # Required, pattern: ^[a-z][a-zA-Z]+$
(imperative verb)
  description: string
  parameters: [Parameter]

  returns:
    enum: [void, domain_id, acknowledgment, result_status]
  return_type_ref: VoId    # If returns domain_id

  modifies_aggregate: AggId  # Exactly one aggregate
  publishes_events: [EvtId]
  audit_fields: [string]     # e.g., "reason", "initiatedBy"
```

Java Example:

```
// CommandInterface: cmd_user_commands
public interface UserCommands {
    // CommandRecord: CreateUserCmd
    UserId createUser(CreateUserCmd cmd);

    record CreateUserCmd(
        String email,        // Parameter
        String userType,     // Parameter
        ClientId clientId    // Parameter
    ) {}  // Immutable record

    // CommandRecord: ActivateUserCmd
    void activateUser(ActivateUserCmd cmd);

    record ActivateUserCmd(
        UserId userId,
        String reason,       // Audit field
        String activatedBy   // Audit field
    ) {}
}
```

9.3.8 Query Interface Type (Knight Pattern)

Purpose: Defines queries as interface methods with nested result records.

Structure:

```
QueryInterface:
  id: qry_<name>           # Required, pattern: ^qry_[a-z0-9_]+$
  name: string             # Required, pattern: ^[A-Z][a-zA-Z]
+Queries$
  aggregate_ref: AggId     # Primary aggregate
  description: string

  query_methods: [QueryMethod]

  result_characteristics:
    immutable: true        # const: true (DTOs are immutable)
    flat_structure: true   # default: true (Knight pattern)
    string_serialization: true  # default: true (IDs as strings)

  layer: "api"            # const: "api"
  no_side_effects: true   # const: true
```

QueryMethod:

```
QueryMethod:
  method_name: string      # Required, pattern: ^(get|list|find|
search)[A-Z][a-zA-Z]+$
  description: string
  parameters: [Parameter]
  result_record_name: string  # Required, pattern: ^[A-Z][a-zA-Z]
+Summary$

  result_structure:
    fields: [DTOField]
    aggregate_counts: [...]  # Counts, not full collections

  bypasses_domain_model: boolean  # CQRS: read from read model

  optimizations:
    denormalized: boolean
    cached: boolean
    indexed: boolean
```

Java Example:

```
// QueryInterface: qry_user_queries
public interface UserQueries {
    // QueryMethod: getUserSummary
    UserSummary getUserSummary(UserId userId);

    record UserSummary(      // Result record (immutable)
        String userId,       // ID as string
        String email,
        String status,       // Enum as string
        int activeClientsCount  // Count, not list
    ) {}  // Flat structure (no nested objects)

    // QueryMethod: listUsersByStatus
    List<UserSummary> listUsersByStatus(String status);
}
```

9.3.9 Domain Event Type

Purpose: Represents something that happened in the domain.

Structure:

```
DomainEvent:
  id: evt_<name>           # Required, pattern: ^evt_[a-z0-9_]+$
  name: string             # Required (past tense, e.g.,
"OrderPlaced")
  aggregate_ref: AggId     # Required (which aggregate publishes)
  description: string

  data_carried: [Attribute]

  immutable: true          # const: true (events are immutable
facts)
```

CRITICAL: `immutable: const: true`

Domain events MUST be immutable. Events represent facts that have already happened and cannot be changed.

Naming Convention: Past tense (OrderPlaced, UserActivated, PaymentProcessed)

Example:

```
domain_events:
  - id: evt_order_placed
    name: "OrderPlaced"
    aggregate_ref: agg_order
    description: "Customer has placed an order"
    data_carried:
      - name: order_id
        type: string
      - name: customer_id
        type: string
      - name: total_amount
        type: decimal
      - name: placed_at
        type: datetime
    immutable: true  # Required
```

9.3.10 Tactical Schema ID Patterns

Type	Prefix	Pattern	Example
Bounded Context	`bc_`	`^bc_[a-z0-9_]+$`	`bc_order_management`
Aggregate	`agg_`	`^agg_[a-z0-9_]+$`	`agg_order`
Entity	`ent_`	`^ent_[a-z0-9_]+$`	`ent_order`, `ent_line_item`
Value Object	`vo_`	`^vo_[a-z0-9_]+$`	`vo_money`, `vo_address`
Repository	`repo_`	`^repo_[a-z0-9_]+$`	`repo_order`
Domain Service	`svc_dom_`	`^svc_dom_[a-z0-9_]+$`	`svc_dom_pricing`
Application Service	`svc_app_`	`^svc_app_[a-z0-9_]+$`	`svc_app_order_management`
Command Interface	`cmd_`	`^cmd_[a-z0-9_]+$`	`cmd_order_commands`
Query Interface	`qry_`	`^qry_[a-z0-9_]+$`	`qry_order_queries`
Domain Event	`evt_`	`^evt_[a-z0-9_]+$`	`evt_order_placed`

9.3.11 Tactical Design Rules

The tactical patterns require:

1. **Aggregate Root is Entity**:
   - `aggregate.root_ref` must reference an entity with `is_aggregate_root: true`
2. **Repository Per Aggregate Root**:
   - `repository.aggregate_ref` must be set (not per-entity repositories)

3. **Value Objects Immutable**:
   - `value_object.immutability: const: true`
4. **Domain Services Stateless**:
   - `domain_service.stateless` must be true
5. **Events Immutable**:
   - `domain_event.immutable: const: true`
6. **Application Services Stateless**:
   - `application_service.characteristics.stateless: const: true`
7. **One Aggregate Per Transaction**:
   - For commands: `transaction_boundary.modifies_aggregates` has `maxItems: 1`
8. **Queries No Side Effects**:
   - `query_interface.no_side_effects: const: true`

9.4 Domain Stories Schema

Location: `/domain-stories/domain-stories-schema.yaml`

9.4.1 Overview

The Domain Stories schema formalizes Domain Storytelling artifacts for validation, tool support, and mapping to tactical DDD concepts.

Root Structure:

```
domain_stories:
  - domain_story_id: dst_<name>
    title: string
    description: string
    tags: [string]

    # Story components
    actors: [Actor]
    work_objects: [WorkObject]
    commands: [Command]
    queries: [Query]
    activities: [Activity]
    events: [Event]
    policies: [Policy]

    # Mappings to tactical concepts
    aggregates: [Aggregate]
    repositories: [Repository]
    application_services: [ApplicationService]
    domain_services: [DomainService]
    read_models: [ReadModel]
    business_rules: [BusinessRule]
```

9.4.2 Actor Type

Purpose: Represents who performs actions in domain stories.

Structure:

```
Actor:
  actor_id: act_<name>     # Required, pattern: ^act_[a-z0-9_]+$
  name: string             # Required
  kind:                    # Required
    enum: [person, system, role]
  description: string
  tags: [string]
```

Actor Kinds:
- **person**: Individual human (Customer, Manager, Operator)
- **system**: Automated actor (PaymentSystem, NotificationService)
- **role**: Generic type (Administrator, User, Guest)

9.4.3 Work Object Type

Purpose: Domain artifacts manipulated by activities.

Structure:

```
WorkObject:
  work_object_id: wobj_<name>  # Required, pattern: ^wobj_[a-
z0-9_]+$
  name: string                 # Required
  description: string
  attributes: [Attribute]
  aggregate_id: agg_<name>     # Optional mapping to aggregate
```

Mapping: Work objects can map to Entities, Value Objects, or Aggregates in tactical model.

9.4.4 Command Type

Purpose: User intent to perform state-changing operation.

Structure:

```
Command:
  command_id: cmd_<name>   # Required, pattern: ^cmd_[a-z0-9_]+$
  name: string             # Required (imperative)
  description: string
  actor_ids: [ActId]       # Required, minItems: 1 (who can issue)

  target_aggregate_id: AggId  # Which aggregate
  parameters: [Parameter]
  preconditions: [string]

  invokes_app_services: [AppSvcId]
  invokes_domain_services: [DomSvcId]
  emits_events: [EvtId]
```

Mapping: Commands in stories map to `CommandRecord` in tactical schema.

9.4.5 Query Type

Purpose: Request for information without side effects.

Structure:

```
Query:
  query_id: qry_<name>     # Required, pattern: ^qry_[a-z0-9_]+$
  name: string             # Required
  description: string
  actor_ids: [ActId]       # Required, minItems: 1

  parameters: [Parameter]
  returns_read_model_id: RmdlId
```

Mapping: Queries in stories map to `QueryMethod` in tactical schema.

9.4.6 Activity Type

Purpose: Actions that change state or produce results.

Structure:

```
Activity:
  activity_id: actv_<name>   # Required, pattern: ^actv_[a-z0-9_]+
$
  name: string               # Required (verb phrase)
  description: string

  initiated_by_command_id: CmdId
  uses_work_object_ids: [WobjId]
  results_in_event_ids: [EvtId]

  calls_app_service_ids: [AppSvcId]
  calls_domain_service_ids: [DomSvcId]
```

Actor-Activity-Work Object Pattern:

```
[Actor] → [Activity] → [Work Object]
   ↓
 initiates
 Command → triggers Activity → produces Event
```

9.4.7 Event Type

Purpose: Facts that have occurred in domain stories.

Structure:

```
Event:
  event_id: evt_<name>     # Required, pattern: ^evt_[a-z0-9_]+$
  name: string             # Required (past tense)
  description: string
  tense:
    enum: [past]
    default: past

  payload: [Attribute]

  caused_by:               # Either command or activity
    oneOf:
      - command_id: CmdId
      - activity_id: ActvId

  affected_aggregate_id: AggId
  policies_triggered: [PolId]
```

Mapping: Events in stories map to `DomainEvent` in tactical schema.

9.4.8 Policy Type

Purpose: Reactive rules ("When X happens, do Y").

Structure:

```
Policy:
  policy_id: pol_<name>    # Required, pattern: ^pol_[a-z0-9_]+$
  name: string             # Required
  description: string

  when_event_id: EvtId     # Required (trigger event)
  issues_command_id: CmdId # Required (resulting command)
```

Pattern: "When [Event] happens, issue [Command]"

Example:

```
policies:
  - policy_id: pol_send_confirmation
    name: "Send Order Confirmation"
    description: "When order is placed, send confirmation email"
    when_event_id: evt_order_placed
    issues_command_id: cmd_send_email
```

9.4.9 Causal Chain in Schema

The pattern captures the complete causal flow:

```
Actor → Command → Activity → Event → Policy → Command
(continues...)
  ↓         ↓          ↓         ↓         ↓
Initiates  Triggers   Uses      Results  Reactive
           Work       Work       In      Automation
           Objects    Objects    Facts
```

Example Flow:
1. Actor `act_customer` initiates Command `cmd_place_order`
2. Command triggers Activity `actv_submit_order`
3. Activity uses WorkObject `wobj_shopping_cart`
4. Activity results in Event `evt_order_placed`
5. Event triggers Policy `pol_reserve_inventory`
6. Policy issues Command `cmd_reserve_inventory`
7. (Flow continues...)

9.4.10 Domain Stories Schema ID Patterns

Type	Prefix	Pattern	Example
Domain Story	`dst_`	`^dst_[a-z0-9_]+$`	`dst_checkout_process`
Actor	`act_`	`^act_[a-z0-9_]+$`	`act_customer`
Work Object	`wobj_`	`^wobj_[a-z0-9_]+$`	`wobj_order`
Activity	`actv_`	`^actv_[a-z0-9_]+$`	`actv_submit_order`
Command	`cmd_`	`^cmd_[a-z0-9_]+$`	`cmd_place_order`
Query	`qry_`	`^qry_[a-z0-9_]+$`	`qry_get_order_status`
Event	`evt_`	`^evt_[a-z0-9_]+$`	`evt_order_placed`
Policy	`pol_`	`^pol_[a-z0-9_]+$`	`pol_send_confirmation`
Read Model	`rmdl_`	`^rmdl_[a-z0-9_]+$`	`rmdl_order_summary`
Business Rule	`rle_`	`^rle_[a-z0-9_]+$`	`rle_order_minimum`
Aggregate	`agg_`	`^agg_[a-z0-9_]+$`	`agg_order`
Repository	`rep_`	`^rep_[a-z0-9_]+$`	`rep_order`
App Service	`svc_app_`	`^svc_app_[a-z0-9_]+$`	`svc_app_order_management`
Domain Service	`dom_svc_`	`^dom_svc_[a-z0-9_]+$`	`dom_svc_pricing`

9.5 Comprehensive ID Convention Table

All three schemas follow consistent ID naming patterns:

Concept	Schema	Prefix	Pattern	Example
System	Strategic	`sys_`	`^sys_[a-z0-9_]+$`	`sys_ecommerce`
Domain	Strategic	`dom_`	`^dom_[a-z0-9_]+$`	`dom_sales`
Bounded Context	Strategic/ Tactical	`bc_`	`^bc_[a-z0-9_]+$`	`bc_order_management`
Context Mapping	Strategic	`cm_`	`^cm_[a-z0-9_]+_to_[a-z0-9_]+$`	`cm_order_to_payment`
BFF Scope	Strategic	`bff_`	`^bff_[a-z0-9_]+$`	`bff_web`, `bff_ios`
BFF Interface	Strategic	`bff_if_`	`^bff_if_[a-z0-9_]+$`	`bff_if_user_web`
Aggregate	Tactical/ Stories	`agg_`	`^agg_[a-z0-9_]+$`	`agg_order`
Entity	Tactical	`ent_`	`^ent_[a-z0-9_]+$`	`ent_order`, `ent_line_item`
Value Object	Tactical	`vo_`	`^vo_[a-z0-9_]+$`	`vo_money`, `vo_address`
Repository	Tactical	`repo_`	`^repo_[a-z0-9_]+$`	`repo_order`
Domain Service	Tactical	`svc_dom_`	`^svc_dom_[a-z0-9_]+$`	`svc_dom_pricing`
Application Service	Tactical/ Stories	`svc_app_`	`^svc_app_[a-z0-9_]+$`	`svc_app_order_management`
Command Interface	Tactical	`cmd_`	`^cmd_[a-z0-9_]+$`	`cmd_order_commands`
	Tactical	`qry_`		`qry_order_queries`

Concept	Schema	Prefix	Pattern	Example
Query Interface			`^qry_[a-z0-9_]+$`	
Domain Event	Tactical/ Stories	`evt_`	`^evt_[a-z0-9_]+$`	`evt_order_placed`
Domain Story	Stories	`dst_`	`^dst_[a-z0-9_]+$`	`dst_checkout_process`
Actor	Stories	`act_`	`^act_[a-z0-9_]+$`	`act_customer`
Work Object	Stories	`wobj_`	`^wobj_[a-z0-9_]+$`	`wobj_order`
Activity	Stories	`actv_`	`^actv_[a-z0-9_]+$`	`actv_submit_order`
Policy	Stories	`pol_`	`^pol_[a-z0-9_]+$`	`pol_send_confirmation`
Read Model	Stories	`rmdl_`	`^rmdl_[a-z0-9_]+$`	`rmdl_order_summary`
Business Rule	Stories	`rle_`	`^rle_[a-z0-9_]+$`	`rle_order_minimum`

Naming Convention: All IDs use `lower_snake_case` format.

9.6 Design Rules Summary

Strategic Design Rules

1. **Bounded Context Has Domain**: Every `BoundedContext.domain_ref` must be valid
2. **Context Mapping Different Contexts**: `upstream_context` ≠ `downstream_context`
3. **BFF One Client Type**: `BFFScope.client_type` is single enum, not array
4. **BFF Aggregates Multiple Contexts**: `aggregates_from_contexts` has `minItems: 1`

5. **BFF No Business Logic**: `responsibilities.business_logic: const: false`
6. **BFF No Direct Persistence**: `responsibilities.direct_persistence: const: false`

Tactical Design Rules

1. **Aggregate Root is Entity**: `aggregate.root_ref` → entity with `is_aggregate_root: true`
2. **Repository Per Aggregate**: `repository.aggregate_ref` must be set
3. **Value Objects Immutable**: `value_object.immutability: const: true`
4. **Domain Services Stateless**: `domain_service.stateless` must be true
5. **Events Immutable**: `domain_event.immutable: const: true`
6. **Application Services Stateless**: `application_service.characteristics.stateless: const: true`
7. **One Aggregate Per Transaction**: `transaction_boundary.modifies_aggregates` has `maxItems: 1`
8. **Queries No Side Effects**: `query_interface.no_side_effects: const: true`
9. **Application Services No Business Logic**: `contains_business_logic: const: false`
10. **Commands Immutable**: `command_interface.immutability: const: true`

Domain Stories Design Rules

1. **Commands Have Actors**: `command.actor_ids` has `minItems: 1`
2. **Queries Have Actors**: `query.actor_ids` has `minItems: 1`
3. **Events Past Tense**: `event.tense` defaults to `past`
4. **Policy Links Event→Command**: Both `when_event_id` and `issues_command_id` required
5. **Domain Stories Have Actors**: `domain_story.actors` has `minItems: 1`

9.7 Codifying DDD Best Practices

The schema definitions **codify** DDD best practices in a machine-verifiable form, making design constraints explicit:

Immutability Requirements

Value Objects:

```
ValueObject:
  immutability:
    const: true  # Must be immutable
```

Domain Events:

```
DomainEvent:
  immutable:
    const: true  # Must be immutable
```

Commands:

```
CommandInterface:
  immutability:
    const: true  # Must be immutable
```

Design Principle: Value objects, events, and commands must be immutable to ensure consistency and prevent unintended side effects.

One Aggregate Per Transaction

```
TransactionBoundary:
  modifies_aggregates:
    maxItems: 1  # One aggregate only
```

Design Principle: Each transaction should modify at most one aggregate, following Vaughn Vernon's rule for maintaining consistency boundaries.

BFF Responsibilities

```
BFFScope:
  responsibilities:
    business_logic:
      const: false  # No business logic in BFFs
    transaction_management:
      const: false  # No transaction management
    direct_persistence:
      const: false  # No direct database access
```

Design Principle: BFFs focus on client-specific aggregation and transformation, delegating business logic and persistence to domain services.

Application Service Characteristics

```
ApplicationService:
  characteristics:
    stateless:
      const: true   # Must be stateless
    contains_business_logic:
      const: false  # No business logic
    manages_transactions:
      const: true   # Manages transactions
```

Design Principle: Application services orchestrate use cases without containing business logic, remaining stateless while managing transaction boundaries.

Query Side Effects

```
QueryInterface:
  no_side_effects:
    const: true  # Must have no side effects
```

Design Principle: Queries retrieve information without modifying state, ensuring CQRS separation of concerns.

9.8 Using the Schemas

Validation with JSON Schema Validators

The schemas use JSON Schema Draft 2020-12 and can be validated with standard tools:

Python (jsonschema):

```
import yaml
import jsonschema

# Load schema
with open('strategic-ddd.schema.yaml', 'r') as f:
    schema = yaml.safe_load(f)

# Load instance
with open('my-system.yaml', 'r') as f:
    instance = yaml.safe_load(f)

# Validate
jsonschema.validate(instance=instance, schema=schema)
```

Node.js (ajv):

```
const Ajv = require('ajv');
const yaml = require('js-yaml');
const fs = require('fs');

const ajv = new Ajv();

const schema = yaml.load(fs.readFileSync('strategic-
ddd.schema.yaml', 'utf8'));
const instance = yaml.load(fs.readFileSync('my-system.yaml',
'utf8'));

const validate = ajv.compile(schema);
const valid = validate(instance);

if (!valid) console.log(validate.errors);
```

IDE Integration

VS Code with YAML extension:

Add to `.vscode/settings.json`:

```
{
  "yaml.schemas": {
    "./domains/ddd/schemas/strategic-ddd.schema.yaml": [
      "*-strategic.yaml",
      "*-system.yaml"
    ],
    "./domains/ddd/schemas/tactical-ddd.schema.yaml": [
      "*-tactical.yaml",
      "*-bounded-context.yaml"
    ],
    "./domain-stories/domain-stories-schema.yaml": [
      "*-story.yaml",
      "*-stories.yaml"
    ]
  }
}
```

Benefits:
- Autocompletion for schema properties
- Inline validation errors
- Documentation on hover
- Pattern enforcement for IDs

Code Generation

The schemas provide complete structure for code generators:

Generate Java from Tactical Schema:

```
def generate_entity(entity_spec):
    """Generate Java entity from schema."""
    code = f"public class {entity_spec['name']} {{\n"

    for attr in entity_spec.get('attributes', []):
        code += f"    private {attr['type']} {attr['name']};\n"

    for method in entity_spec.get('business_methods', []):
        code += f"    public {method['returns']} {method['name']}
("
        params = [f"{p['type']} {p['name']}" for p in
method.get('parameters', [])]
        code += ", ".join(params) + ") { ... }\n"

    code += "}\n"
    return code
```

Generate BFF from Strategic Schema:

```
def generate_bff_controller(bff_interface_spec):
    """Generate Spring Boot controller from BFF interface
schema."""
    code = "@RestController\n"
    code += f"@RequestMapping(\"{bff_interface_spec['base_path']}
\")\n"
    code += f"public class {bff_interface_spec['name']} {{\n"

    for endpoint in bff_interface_spec.get('endpoints', []):
        method = endpoint['method'].lower()
        path = endpoint['path']
        code += f"    @{endpoint['method']}Mapping(\"{path}\")\n"
        code += f"    public ResponseEntity<?>
{endpoint['operation_type']}(...) {{ ... }}\n"

    code += "}\n"
    return code
```

LLM Reasoning

The schemas enable LLMs to:
- Understand complete DDD structure
- Validate domain models against patterns
- Generate consistent architectures
- Answer questions about relationships
- Suggest refactorings based on rules

Example Prompt:

```
Given this tactical schema:
[paste tactical model YAML]

1. Validate that all aggregates have exactly one root
2. Check if any commands modify multiple aggregates
3. Suggest improvements for aggregate boundaries
```

9.9 Schema Best Practices

When Modeling Strategic Patterns

1. **Start with System Root**:
   - Define `system` with ID, name, version
   - Add domains progressively
2. **Domain Classification**:
   - Be honest about core vs. supporting vs. generic
   - Align `strategic_importance` with investment
3. **Bounded Context Boundaries**:
   - Use linguistic boundaries (when same word means different things)
   - Follow team boundaries (Conway's Law)
   - Keep contexts focused (single responsibility)
4. **Context Mappings**:
   - Make all integrations explicit
   - Choose relationship type carefully
   - Document translation maps for ACLs
5. **BFF Scoping**:
   - **One BFF per client type** (not per bounded context!)
   - BFF aggregates from MULTIPLE contexts
   - Owned by frontend team
   - NO business logic in BFF

When Modeling Tactical Patterns

1. **Aggregate Design**:
   - Start small (single entity if possible)
   - Protect true invariants only
   - Reference other aggregates by ID
   - Use domain events for cross-aggregate consistency
2. **Value Object Usage**:
   - Use for domain concepts (Money, Address, Email)
   - Enforce immutability (design requires this)
   - Include validation in constructor
   - Define equality by attributes

3. **Repository Scope**:
   - One repository per aggregate root
   - Not per entity (common mistake!)
   - Repository interface in domain layer
   - Implementation in infrastructure layer
4. **Application Service Design**:
   - One operation per use case
   - NO business logic (coordination only)
   - Manage transaction boundaries
   - Publish domain events after success
5. **CQRS Implementation**:
   - Separate command and query interfaces
   - Commands return void/ID/acknowledgment
   - Queries return flat DTOs
   - Use Knight pattern (nested records)

When Modeling Domain Stories

1. **Actor Identification**:
   - Include both humans and systems
   - Use `role` for generic types
   - Link actors to commands they can issue
2. **Work Object Mapping**:
   - Map to aggregates where possible
   - Include attributes for context
   - Show relationships via activities
3. **Command/Query Separation**:
   - Commands: state-changing operations
   - Queries: information retrieval
   - Clear actor authorization
4. **Event Flow**:
   - Events in past tense
   - Capture payload data
   - Link to causing commands/activities
   - Connect to policies for reactive logic

5. **Policy Definition**:
   - "When [Event] happens, do [Command]"
   - Automate business rules
   - Make reactive logic explicit

Part V Summary (Section 9)

Section 9 provided **comprehensive schema documentation**:

Three Schemas Covered:

1. **Strategic Schema** (`strategic-ddd.schema.yaml`):
   - **System** root object
   - Domain, BoundedContext, ContextMapping
   - BFFScope and BFFInterface (extensive BFF support)
   - Schema-enforced BFF constraints
2. **Tactical Schema** (`tactical-ddd.schema.yaml`):
   - **BoundedContext** root object
   - Aggregate, Entity, ValueObject, Repository
   - ApplicationService with CQRS support
   - CommandInterface and QueryInterface (Knight pattern)
   - DomainEvent with immutability enforcement
   - TransactionBoundary with one-aggregate rule (maxItems: 1)
3. **Domain Stories Schema** (`domain-stories-schema.yaml`):
   - Actor, WorkObject, Activity
   - Command, Query, Event, Policy
   - Causal chain representation
   - Mappings to tactical concepts

Key Schema Innovations:

- **Immutability Enforcement**: `const: true` for ValueObject, DomainEvent, Commands
- **One Aggregate Rule**: `maxItems: 1` for transaction_boundary.modifies_aggregates

- **BFF Constraints**: Schema-enforced "no business logic" and "no direct persistence"
- **Root Objects**: System (strategic) and BoundedContext (tactical) as roots
- **ID Patterns**: Comprehensive prefixes for all concept types

Validation Rules:
- Strategic: 6 rules (bounded context has domain, BFF constraints, etc.)
- Tactical: 10 rules (immutability, statelessness, one aggregate, etc.)
- Domain Stories: 5 rules (actors required, past tense events, etc.)

Usage:
- JSON Schema validation with standard tools
- IDE integration for autocomplete and validation
- Code generation from schema definitions
- LLM reasoning about domain models

Best Practices:
- Start with System root for strategic models
- Design small aggregates
- One repository per aggregate root
- Use Knight pattern for commands/queries
- BFF aggregates from multiple contexts, owned by frontend team

What's Next: Section 10 covers **Bibliography**—primary sources, BFF pattern references, CQRS resources, and Domain Storytelling literature.

---

10. Bibliography & Further Reading

This section provides comprehensive references for all topics covered in this guide, organized by subject area.

10.1 Primary DDD Sources

Foundational Books

Evans, Eric (2003). "Domain-Driven Design: Tackling Complexity in the Heart of Software"

- Publisher: Addison-Wesley Professional
- ISBN: 0-321-12521-5
- **The** foundational text that introduced DDD
- **Key Chapters**:
- Part I: Putting the Domain Model to Work (Ch. 1-3)
- Part II: The Building Blocks of a Model-Driven Design (Ch. 4-6) — Entities, Value Objects, Services
- Part III: Refactoring Toward Deeper Insight (Ch. 7-11) — Aggregates, Factories, Repositories
- Part IV: Strategic Design (Ch. 12-15) — Bounded Contexts, Context Mapping, Distillation
- **Essential Concepts**: Ubiquitous Language, Bounded Context, Aggregate, Entity, Value Object, Repository, Domain Event
- **Online**: https://www.domainlanguage.com/ddd/

Vernon, Vaughn (2013). "Implementing Domain-Driven Design"

- Publisher: Addison-Wesley Professional
- ISBN: 0-321-83457-7
- Modern, practical approach to DDD with code examples
- **Key Chapters**:
- Ch. 1: Getting Started with DDD
- Ch. 2: Domains, Subdomains, and Bounded Contexts
- Ch. 4: Architecture (Hexagonal, Ports & Adapters, CQRS)
- Ch. 5: Entities
- Ch. 6: Value Objects
- Ch. 10: Aggregates — **The 4 Rules of Aggregate Design**
- Ch. 8: Domain Events
- Ch. 11: Factories
- Ch. 12: Repositories
- **Essential Concepts**: Aggregate design rules, Hexagonal Architecture, Event Sourcing integration
- **Online**: https://vaughnvernon.com/

Evans, Eric (2015). "Domain-Driven Design Reference"

- Publisher: Domain Language, Inc.
- Free PDF available online
- Distilled reference guide extracting key patterns from the Blue Book
- **Sections**: Pattern definitions, summaries, quick reference
- **Online**: https://www.domainlanguage.com/ddd/reference/

Complementary Books

Fowler, Martin (2002). "Patterns of Enterprise Application Architecture"

- Publisher: Addison-Wesley Professional
- ISBN: 0-321-12742-0
- Architectural patterns that complement DDD
- **Key Chapters**:
- Ch. 2: Organizing Domain Logic — Domain Model, Service Layer
- Ch. 9-12: Data Source Architectural Patterns — Repository, Data Mapper, Unit of Work
- Ch. 13-14: Object-Relational Behavioral/Structural Patterns
- Ch. 15-17: Web Presentation Patterns
- **Essential Patterns**: Domain Model, Service Layer, Repository, Data Mapper, Unit of Work, Lazy Load
- **Online**: https://martinfowler.com/eaaCatalog/

Khononov, Vlad (2021). "Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy"

- Publisher: O'Reilly Media
- ISBN: 978-1098100131
- Modern introduction to DDD with business strategy focus
- **Key Topics**: Business domains, strategic patterns, tactical patterns, event-driven architecture
- Excellent for beginners and experienced practitioners

Millett, Scott & Tune, Nick (2015). "Patterns, Principles, and Practices of Domain-Driven Design"

- Publisher: Wrox
- ISBN: 978-1118714706
- Comprehensive guide with practical examples in C#
- **Key Topics**: Strategic design, tactical patterns, CQRS, Event Sourcing, microservices integration

10.2 Backend-for-Frontend (BFF) Pattern

Calçado, Phil (2015). "The Back-end for Front-end Pattern (BFF)"
- **Online**: https://samnewman.io/patterns/architectural/bff/
- Original article introducing BFF pattern for SoundCloud
- **Key Concepts**: One BFF per user experience, API composition, client-specific optimization
- **Principles**:
- BFF owned by frontend team
- Each BFF serves exactly one client type
- BFFs aggregate from multiple microservices
- No shared business logic in BFFs

Newman, Sam (2015). "Building Microservices"
- Publisher: O'Reilly Media
- ISBN: 978-1491950357
- **Ch. 4**: Integration patterns including BFF
- **Key Topics**: API Gateway vs. BFF, microservice communication, UI composition
- **Online**: https://samnewman.io/

Newman, Sam (2019). "Monolith to Microservices"
- Publisher: O'Reilly Media
- ISBN: 978-1492047841
- **Ch. 5**: Growing Pains — BFF pattern in migration scenarios
- **Key Topics**: BFF for legacy integration, progressive migration

Kong, Inc. "BFF Pattern Implementation Guide"
- **Online**: https://konghq.com/blog/backend-for-frontend
- Practical guide to implementing BFF with API Gateway
- Hybrid approach: API Gateway upstream + BFFs downstream

AWS Architecture Blog (2020). "Backend for Frontend Pattern with Amazon API Gateway"
- **Online**: https://aws.amazon.com/blogs/architecture/
- Cloud implementation patterns for BFF
- Integration with serverless architectures

10.3 CQRS and Event Sourcing

Young, Greg (2010). "CQRS Documents"
- **Online**: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
- Foundational CQRS documentation by pattern originator
- **Key Concepts**: Command/Query separation, read models, eventual consistency
- **Principles**: Different models for read and write, optimize independently

Dahan, Udi (2009). "Clarified CQRS"
- **Online**: https://udidahan.com/2009/12/09/clarified-cqrs/
- Clarification of CQRS misconceptions
- **Key Points**: CQRS is not Event Sourcing, not always needed, selective application

Microsoft (2013). "CQRS Journey"
- **Online**: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj554200(v=pandp.10)
- Comprehensive guide to CQRS implementation
- Real-world case study with code examples
- **Key Topics**: Command handlers, event handlers, read model projections, sagas

Fowler, Martin (2011). "CQRS"
- **Online**: https://martinfowler.com/bliki/CQRS.html
- Overview and analysis of CQRS pattern
- **Key Points**: When to use, when not to use, complexity considerations

Vernon, Vaughn (2015). "Reactive Messaging Patterns with the Actor Model"
- Publisher: Addison-Wesley Professional
- ISBN: 978-0133846836
- **Ch. 4-5**: Command and Event patterns
- **Key Topics**: Message-driven architecture, event-driven systems, actor model integration

Betts, Dominic et al. (2012). "Exploring CQRS and Event Sourcing"
- Publisher: Microsoft patterns & practices
- ISBN: 978-1621140160

- Free online: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj554200(v=pandp.10)
- **Key Topics**: CQRS journey, event sourcing, eventual consistency, sagas

10.4 Domain Storytelling

Hofer, Stefan & Schwentner, Henning (2021). "Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software"
- Publisher: Addison-Wesley Professional
- ISBN: 978-0137458912
- **The** definitive book on Domain Storytelling
- **Key Chapters**:
- Ch. 1-2: Introduction and notation
- Ch. 3: Workshop facilitation
- Ch. 4-5: From stories to bounded contexts
- Ch. 6: Integration with DDD
- Ch. 7-8: Practical examples and case studies
- **Essential Concepts**: Actor-Activity-Work Object, pictographic language, collaborative modeling
- **Online**: https://domainstorytelling.org/

Schwentner, Henning (2019). "Domain Storytelling Workshop Guide"
- **Online**: https://domainstorytelling.org/resources
- Practical workshop facilitation techniques
- **Key Topics**: Preparing workshops, facilitation tips, common pitfalls

Domain Storytelling Tool
- **Online**: https://www.wps.de/modeler/
- Free online tool for creating domain stories
- Visual editor with Actor-Activity-Work Object notation
- Export to SVG, PNG, PDF

Brandolini, Alberto (2013). "Introducing EventStorming"
- Publisher: Leanpub
- ISBN: 978-1387133130
- **Related Technique**: Event Storming complements Domain Storytelling
- **Key Topics**: Collaborative modeling, domain events, big picture exploration
- **Online**: https://www.eventstorming.com/

Brandolini, Alberto (2018). "EventStorming Recipes"

- **Online**: https://leanpub.com/eventstorming-recipes
- Practical patterns for EventStorming workshops
- Integration with Domain Storytelling

10.5 The Knight Pattern (Command/Query with Nested Records)

Knight, James (2020). "Command Objects as Nested Records in Java 14+"

- **Online**: https://medium.com/@jamesknightcs/
- Pattern of defining command records inside interface contracts
- **Key Concepts**: Immutable commands, nested records, API design

Oracle Java Documentation (2020). "Java Records"

- **Online**: https://docs.oracle.com/en/java/javase/14/language/records.html
- Language feature enabling Knight pattern
- **Key Features**: Immutability, concise syntax, value semantics

Fowler, Martin (2005). "Value Object"

- **Online**: https://martinfowler.com/bliki/ValueObject.html
- Foundation for command/query as value objects
- **Key Concepts**: Immutability, equality by value, no identity

Parnas, David (1972). "On the Criteria To Be Used in Decomposing Systems into Modules"

- **Source**: Communications of the ACM, Vol. 15, No. 12
- **Key Principle**: Information hiding — interfaces hide implementation
- **Application**: Command interfaces hide aggregate operations

10.6 Ubiquitous Language and Knowledge Crunching

Evans, Eric (2003). "Domain-Driven Design" — Part I

- Chapters 1-3: Ubiquitous Language, Knowledge Crunching, Model-Driven Design
- **Essential Reading** for understanding DDD philosophy

Evans, Eric (2006). "Getting Started with DDD When Surrounded by Legacy Systems"

- **Talk**: Domain-Driven Design Europe
- **Online**: https://www.domainlanguage.com/
- **Key Topics**: Bubble Context, Anti-Corruption Layer, gradual migration

Fowler, Martin (2004). "Ubiquitous Language"

- **Online**: https://martinfowler.com/bliki/UbiquitousLanguage.html
- Overview and importance of shared language
- **Key Points**: Language evolution, consistency across team

10.7 Aggregates and Consistency

Vernon, Vaughn (2011). "Effective Aggregate Design" (3-part series)

- **Online**: https://vaughnvernon.com/
- **Part I**: Aggregate boundaries and invariants
- **Part II**: Making aggregates work together
- **Part III**: Gaining insight through discovery
- **The 4 Rules**:

1. Protect true invariants through consistency boundaries
2. Design small aggregates
3. Reference other aggregates by identity only
4. Update other aggregates via eventual consistency (domain events)

Fowler, Martin (2015). "DDD Aggregate"

- **Online**: https://martinfowler.com/bliki/DDD_Aggregate.html
- Overview of aggregate pattern
- **Key Points**: Consistency boundary, transaction scope, reference by ID

Evans, Eric (2003). "Domain-Driven Design" — Ch. 6

- "The Life Cycle of a Domain Object"
- Aggregates, Factories, Repositories
- **Essential Reading** for understanding aggregate design

10.8 Architecture and Layering

Cockburn, Alistair (2005). "Hexagonal Architecture (Ports and Adapters)"
- **Online**: https://alistair.cockburn.us/hexagonal-architecture/
- Architectural pattern complementing DDD
- **Key Concepts**: Ports (interfaces), Adapters (implementations), domain isolation

Martin, Robert C. (2012). "The Clean Architecture"
- **Online**: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- Layered architecture with dependency inversion
- **Key Concepts**: Dependency rule, entities, use cases, adapters

Fowler, Martin (2015). "PresentationDomainDataLayering"
- **Online**: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Classic 3-layer architecture
- **Layers**: Presentation, Domain, Data Source
- **Integration**: Maps naturally to DDD layers

Richardson, Chris (2018). "Microservices Patterns"
- Publisher: Manning Publications
- ISBN: 978-1617294549
- **Key Chapters**: Saga pattern, API composition, CQRS, Event Sourcing
- **Key Topics**: DDD in microservices context, strategic design for microservices

10.9 Value Objects and Immutability

Fowler, Martin (2005). "Value Object"
- **Online**: https://martinfowler.com/bliki/ValueObject.html
- Canonical definition of value objects
- **Key Concepts**: Equality by value, immutability, no identity

Evans, Eric (2003). "Domain-Driven Design" — Ch. 5
- "A Model Expressed in Software"
- Entities and Value Objects distinction
- **Essential Reading** for understanding when to use each

Bloch, Joshua (2018). "Effective Java" — Third Edition
- Publisher: Addison-Wesley Professional
- ISBN: 978-0134685991

- **Item 17**: Minimize mutability
- **Item 10**: Obey the general contract when overriding equals
- **Application**: Implementing value objects correctly in Java

10.10 Domain Events

Fowler, Martin (2005). "Event Sourcing"
- **Online**: https://martinfowler.com/eaaDev/EventSourcing.html
- Event Sourcing pattern overview
- **Key Concepts**: Event log, event replay, temporal queries

Young, Greg (2007). "Event Sourcing Basics"
- **Online**: https://www.eventstore.com/blog/event-sourcing-basics
- Introduction to Event Sourcing
- **Key Topics**: Event store, projections, snapshots

Vernon, Vaughn (2013). "Implementing Domain-Driven Design" — Ch. 8
- Domain Events chapter
- **Key Topics**: Event publication, subscribers, eventual consistency

Hohpe, Gregor & Woolf, Bobby (2003). "Enterprise Integration Patterns"
- Publisher: Addison-Wesley Professional
- ISBN: 978-0321200686
- **Key Patterns**: Message Channel, Publish-Subscribe, Event Message
- **Online**: https://www.enterpriseintegrationpatterns.com/

10.11 Repositories and Persistence

Fowler, Martin (2002). "Repository"
- **Online**: https://martinfowler.com/eaaCatalog/repository.html
- Repository pattern from PoEAA
- **Key Concepts**: Collection-like interface, query encapsulation

Fowler, Martin (2002). "Data Mapper"
- **Online**: https://martinfowler.com/eaaCatalog/dataMapper.html
- Implementing repositories with Data Mapper
- **Key Concepts**: Separation of domain and persistence, bidirectional mapping

Fowler, Martin (2002). "Unit of Work"

- **Online**: https://martinfowler.com/eaaCatalog/unitOfWork.html
- Transaction management pattern
- **Key Concepts**: Change tracking, commit/rollback, object registration

Evans, Eric (2003). "Domain-Driven Design" — Ch. 6

- Repositories in DDD context
- **Key Points**: One repository per aggregate root, domain-centric queries

10.12 Strategic Design and Context Mapping

Evans, Eric (2003). "Domain-Driven Design" — Part IV

- Strategic Design chapters (12-15)
- **Ch. 14**: Context Mapping — **Essential reading** for integration patterns
- **Key Concepts**: Partnership, Shared Kernel, Customer/Supplier, Conformist, ACL, OHS, PL

Fowler, Martin (2014). "Bounded Context"

- **Online**: https://martinfowler.com/bliki/BoundedContext.html
- Overview of bounded context concept
- **Key Points**: Linguistic boundaries, context autonomy, explicit relationships

Vernon, Vaughn (2013). "Implementing Domain-Driven Design" — Ch. 2-3

- Ch. 2: Domains, Subdomains, and Bounded Contexts
- Ch. 3: Context Maps
- **Key Topics**: Core domain identification, context relationship patterns

Tune, Nick & Millett, Scott (2017). "Domain-Driven Design Distilled"

- Publisher: Addison-Wesley Professional
- ISBN: 978-0134434421
- **Focus**: Strategic design patterns
- **Key Topics**: Core domain charts, context mapping, organizational patterns

10.13 Schemas and Formal Methods

JSON Schema Specification (Draft 2020-12)

- **Online**: https://json-schema.org/draft/2020-12/json-schema-core.html
- JSON Schema standard used for formalizing DDD patterns
- **Key Features**: Schema validation, type system, constraints

YAML Specification v1.2
- **Online**: https://yaml.org/spec/1.2/spec.html
- YAML format for schema definitions
- **Key Features**: Human-readable, structured data, JSON superset

OpenAPI Specification v3.1
- **Online**: https://spec.openapis.org/oas/v3.1.0
- API specification standard (complementary to schemas)
- **Key Features**: REST API documentation, code generation, validation

10.14 Conway's Law and Team Topologies

Conway, Melvin E. (1968). "How Do Committees Invent?"
- **Source**: Datamation, Vol. 14, No. 4
- **Online**: http://www.melconway.com/Home/Committees_Paper.html
- **Key Principle**: "Organizations design systems that mirror their communication structure"
- **Application**: BFF team ownership, bounded context ownership

Skelton, Matthew & Pais, Manuel (2019). "Team Topologies"
- Publisher: IT Revolution Press
- ISBN: 978-1942788812
- **Key Concepts**: Stream-aligned teams, platform teams, enabling teams
- **Application**: Organizing teams around bounded contexts and BFFs

Evans, Eric (2003). "Domain-Driven Design" — Ch. 14
- "Maintaining Model Integrity"
- **Key Topics**: Team organization, continuous integration, context boundaries

10.15 DDD Schema References

Strategic DDD Schema
- **Location**: `/domains/ddd/schemas/strategic-ddd.schema.yaml`
- **Author**: Marina Music
- **Updated**: 2025-10-24
- **Concepts**: System, Domain, BoundedContext, ContextMapping, BFFScope, BFFInterface
- **Key Features**: System root object, BFF constraints, context mapping relationships

Tactical DDD Schema
- **Location**: `/domains/ddd/schemas/tactical-ddd.schema.yaml`
- **Author**: Marina Music
- **Updated**: 2025-10-24
- **Concepts**: BoundedContext (root), Aggregate, Entity, ValueObject, Repository, ApplicationService, CommandInterface, QueryInterface, DomainEvent
- **Key Features**: BoundedContext root object, immutability enforcement, one aggregate per transaction rule

Domain Stories Schema
- **Location**: `/domain-stories/domain-stories-schema.yaml`
- **Author**: Marina Music
- **Updated**: 2025-10-24
- **Concepts**: DomainStory, Actor, WorkObject, Activity, Command, Query, Event, Policy
- **Key Features**: Causal chain representation, mapping to tactical concepts

Schema Validation Tools
- **jsonschema (Python)**: https://python-jsonschema.readthedocs.io/
- **ajv (Node.js)**: https://ajv.js.org/
- **VS Code YAML Extension**: https://marketplace.visualstudio.com/items?itemName=redhat.vscode-yaml

10.16 Online Communities and Resources

Domain-Driven Design Community
- **Online**: https://dddcommunity.org/
- Active DDD community with resources, events, and discussions
- **Key Resources**: Patterns catalog, book recommendations, conference talks

DDD Europe
- **Online**: https://dddeurope.com/
- Annual European DDD conference
- **Key Resources**: Conference videos, workshops, community events

Virtual DDD
- **Online**: https://virtualddd.com/
- Online DDD community and meetups
- **Key Resources**: Recorded sessions, open-space discussions

Martin Fowler's Website
- **Online**: https://martinfowler.com/
- Extensive pattern catalog and articles
- **Key Sections**: Bliki (blog), PoEAA catalog, microservices resources

InfoQ Domain-Driven Design
- **Online**: https://www.infoq.com/domaindrivendesign/
- Articles, presentations, and case studies
- **Key Topics**: Strategic design, microservices, CQRS, Event Sourcing

GitHub - DDD Sample Applications
- **Spring PetClinic (DDD)**: https://github.com/spring-petclinic/spring-petclinic-microservices
- **DDD Sample (Java)**: https://github.com/citerus/dddsample-core
- **Microsoft eShopOnContainers**: https://github.com/dotnet-architecture/eShopOnContainers

10.17 Related Patterns and Practices

Gamma, Erich et al. (1994). "Design Patterns: Elements of Reusable Object-Oriented Software"
- Publisher: Addison-Wesley Professional
- ISBN: 0-201-63361-2
- **Relevant Patterns**: Factory, Strategy, Observer, Composite
- **Application**: Implementing DDD tactical patterns

Beck, Kent (2002). "Test Driven Development: By Example"
- Publisher: Addison-Wesley Professional
- ISBN: 0-321-14653-0
- **Key Topics**: TDD with domain models, testing aggregates

Freeman, Steve & Pryce, Nat (2009). "Growing Object-Oriented Software, Guided by Tests"
- Publisher: Addison-Wesley Professional
- ISBN: 978-0321503626
- **Key Topics**: Outside-in TDD, testing domain logic, mock objects

Martin, Robert C. (2008). "Clean Code"
- Publisher: Prentice Hall
- ISBN: 978-0132350884
- **Key Chapters**: Ch. 2 (Meaningful Names), Ch. 3 (Functions), Ch. 10 (Classes)
- **Application**: Writing clean domain code

10.18 Additional Reading

Buschmann, Frank et al. (1996). "Pattern-Oriented Software Architecture, Volume 1"
- Publisher: Wiley
- ISBN: 978-0471958697
- **Key Patterns**: Layers, Pipes and Filters, Broker
- **Application**: Architectural patterns complementing DDD

Bass, Len et al. (2012). "Software Architecture in Practice" — Third Edition
- Publisher: Addison-Wesley Professional
- ISBN: 978-0321815736
- **Key Topics**: Architectural styles, quality attributes, design decisions

Evans, Eric & Evans, Robert (2009). "Domain-Driven Design Quickly"
- Publisher: InfoQ
- Free online minibook
- **Online**: https://www.infoq.com/minibooks/domain-driven-design-quickly/
- Quick overview of DDD concepts

10.19 Historical Context

Dijkstra, Edsger W. (1968). "A Case against the GO TO Statement"
- **Source**: Communications of the ACM, Vol. 11, No. 3
- **Historical Context**: Structured programming foundation for DDD's emphasis on clarity

Parnas, David (1972). "On the Criteria To Be Used in Decomposing Systems into Modules"
- **Source**: Communications of the ACM, Vol. 15, No. 12
- **Key Principle**: Information hiding — foundation for bounded contexts

Brooks, Frederick P. (1975). "The Mythical Man-Month"
- Publisher: Addison-Wesley Professional
- ISBN: 0-201-00650-2
- **Key Essay**: "No Silver Bullet" — complexity management, relevant to DDD philosophy

Jackson, Michael (1995). "Software Requirements & Specifications"
- Publisher: ACM Press
- **Key Concepts**: Problem frames, domain modeling precursors to DDD

10.20 Recommended Reading Order

For practitioners **new to DDD**, recommended reading order:

1. **Start Here**:
   - Evans, Eric (2003). "Domain-Driven Design" (Blue Book) — Chapters 1-6
   - Fowler, Martin (2002). "Patterns of Enterprise Application Architecture" — Domain Logic chapters
2. **Strategic Design**:
   - Evans, Eric (2003). "Domain-Driven Design" — Chapters 12-15
   - Vernon, Vaughn (2013). "Implementing Domain-Driven Design" — Chapters 2-3
   - Hofer & Schwentner (2021). "Domain Storytelling"
3. **Tactical Patterns**:
   - Vernon, Vaughn (2013). "Implementing Domain-Driven Design" — Chapters 5-6, 10-12
   - Vernon, Vaughn (2011). "Effective Aggregate Design" (3-part series)
4. **Application Layer & CQRS**:
   - Fowler, Martin (2002). "Patterns of Enterprise Application Architecture" — Service Layer
   - Young, Greg (2010). "CQRS Documents"
   - Microsoft (2013). "CQRS Journey"
5. **BFF Pattern**:
   - Calçado, Phil (2015). "The Back-end for Front-end Pattern"
   - Newman, Sam (2015). "Building Microservices" — Chapter 4

6. **Schemas & Formalization**:
   - DDD Schemas (this guide, Section 9)
   - JSON Schema Specification

For practitioners **experienced with DDD**, focus on:
- Vernon's aggregate design series
- CQRS/Event Sourcing resources
- Domain Storytelling
- DDD Schemas for formalization

Guide Summary and Conclusion

This guide has covered **Domain-Driven Design** comprehensively across 10 sections spanning ~58,000 words.

What We've Covered

Part I: Foundations (Sections 1-3)
- DDD philosophy and principles
- System root object
- Strategic patterns: Domain, Subdomain, Bounded Context, Context Mapping
- Ubiquitous Language and knowledge crunching

Part II: Discovery (Section 4)
- Domain Storytelling technique
- Actor-Activity-Work Object notation
- Workshop facilitation
- From stories to bounded contexts

Part III: Tactical Implementation (Sections 5-7)
- Tactical patterns: Aggregate, Entity, Value Object, Repository, Domain Service, Domain Event
- Application Layer: Application Services, CQRS, Transaction Boundaries
- BFF Pattern: "One Experience, One BFF", multi-context aggregation

Part IV: Integration (Section 8)
- PoEAA integration: Domain Model, Service Layer, Repository, Data Mapper
- Layered architecture
- Pattern combinations

Part V: Reference (Sections 9-10)
- Comprehensive schema documentation (Strategic, Tactical, Domain Stories)
- ID conventions and validation rules
- Bibliography and resources

Key DDD Patterns Covered

1. **Root Objects**: System (strategic) and BoundedContext (tactical)
2. **Design Constraints**: Immutability, one aggregate per transaction, BFF responsibilities
3. **Knight Pattern**: Commands/Queries as nested records in interfaces
4. **BFF Pattern**: "One Experience, One BFF", multi-context aggregation
5. **Domain Storytelling**: Collaborative discovery and modeling technique

Using This Guide

As a Learning Resource:
- Read sequentially from Part I → Part V
- Work through examples
- Refer to bibliography for deeper understanding

As a Reference:
- Jump to specific sections for pattern details
- Use Section 9 (Schema Reference) for schema definitions
- Use Section 10 (Bibliography) for source materials

As a Schema Guide:
- Section 9 provides complete schema documentation
- Use schemas for validation, code generation, LLM reasoning
- Follow ID conventions and validation rules

Next Steps

1. **Apply Strategic Design**: Use System root, identify domains, define bounded contexts
2. **Conduct Domain Storytelling**: Workshop with domain experts, create domain stories
3. **Model Tactical Patterns**: Design aggregates, define entities/value objects, implement repositories
4. **Implement Application Layer**: Create application services, separate commands/queries
5. **Add BFF Layer**: Define BFF scopes per client type, aggregate from multiple contexts
6. **Formalize Your Design**: Use the schemas for structural validation and documentation

Final Thoughts

Domain-Driven Design is not just a set of patterns — it's a **philosophy** of software development that puts the **domain model** at the center of the system. The schemas provide a way to formalize DDD patterns, making them **machine-readable**, **validatable**, and verifiable.

Key Principles to Remember:
- **Ubiquitous Language**: Speak the domain expert's language
- **Bounded Contexts**: Explicit boundaries for models
- **Aggregates**: Consistency boundaries, keep them small
- **Domain Events**: Communication across boundaries
- **CQRS**: Separate reads from writes when beneficial
- **BFF Pattern**: One BFF per client experience

Success with DDD requires:
- Close collaboration with domain experts
- Iterative refinement of models
- Attention to linguistic boundaries
- Disciplined application of patterns
- Continuous learning and improvement

Thank you for reading this comprehensive DDD guide. May your domain models be rich, your aggregates small, your bounded contexts clear, and your software aligned with business needs.