ADVANCED TRAINING ISAQB-DDD

DAY 6: Strategic Design 2

Context Mapping

> The Bridge Between Business Strategy and Software Architecture — Mapping Bounded Contexts, Defining Relationships, and Implementing Integration Patterns

SESSION FOCUS

Strategic Patterns

OHS • ACL • Partnership

AUDIENCE

Senior Engineers

Architects & Managers

COURSE GOALS

Targeted Learning Goals & Alignment Matrix

Define strategic capabilities and their business value delivery

1

Identify Upstream/Downstream

Map relationships between teams and data flow direction

- Prevents "Big Ball of Mud" architecture

2

Implement Open Host Service

Design stable APIs for external ecosystem integration

- Enables business ecosystems without leaking internal logic

3

Design Anticorruption Layer

Build translation layer to isolate legacy system models

- Protects Core Domain from "polluted" external models

4

Define Customer/Supplier Contracts

Establish development team relationships and ownership

- Aligns DORA metrics with business value streams

5

Apply Domain Events

Decouple contexts using asynchronous messaging patterns

- Increases resilience vs. synchronous REST/RPC

6

Evaluate Integration Patterns

Choose between Partnership, Separate Ways, Conformist

- Balances cost vs. ROI of integration strategies

BIG PICTURE

Visualizing the Context Map

A living document that maps Bounded Contexts and their relationships

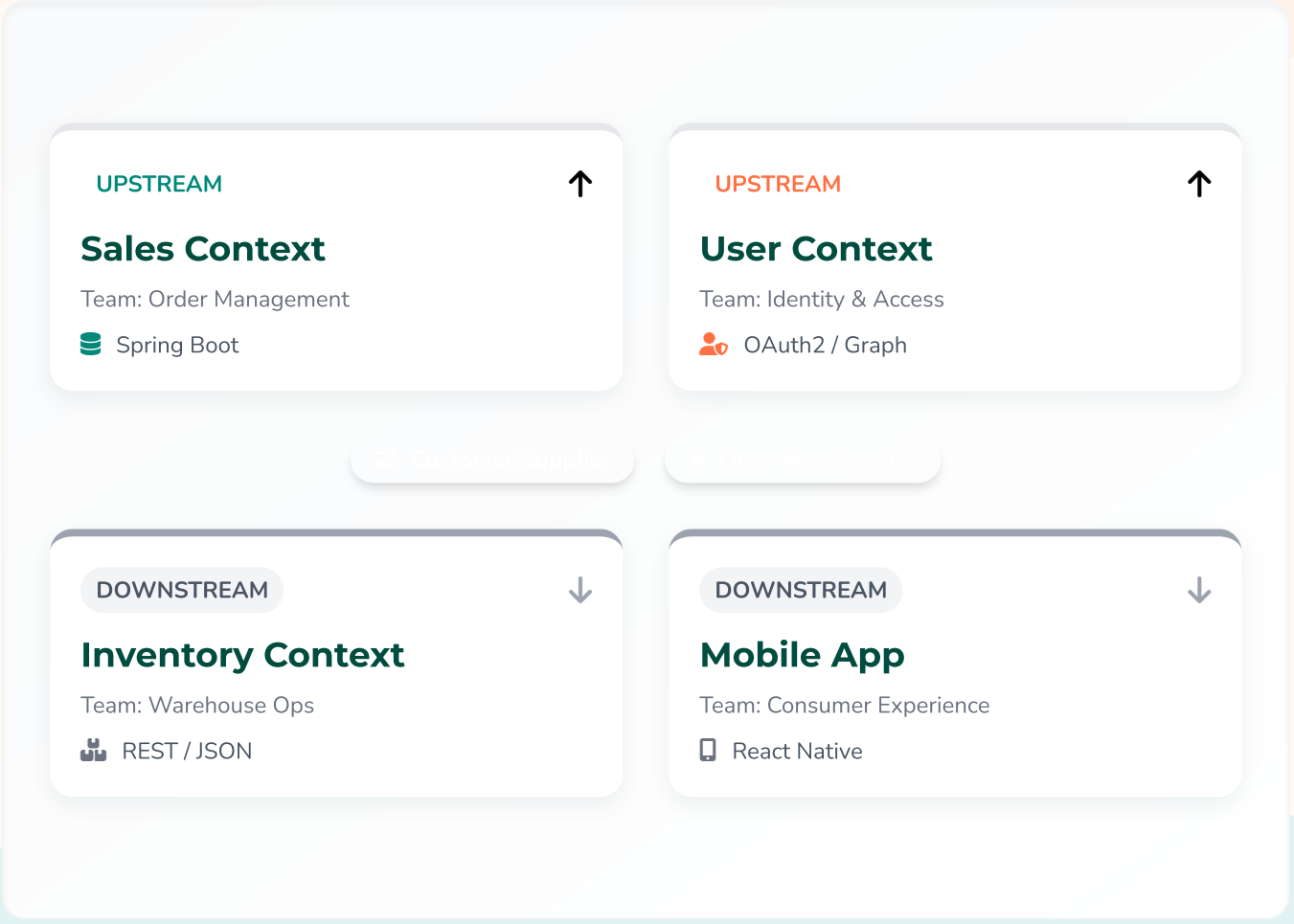
What is a Context Map?

A diagram showing **Bounded Contexts** in the system and the **relationships** between them. It maps the "landscape" of your software organization.

Map Layers

- **Context Name** e.g., Sales, Inventory
- **The Team** Who owns this code?
- **Relationship** OHS, ACL, etc.
- **Technology** Kafka, REST, gRPC

Visual Structure



CORE CONCEPT

Upstream vs. Downstream Dynamics

The fundamental duality defining integration relationships

UPSTREAM

THE PROVIDER

- **Independent** - Less dependent on Downstream
- **High Power** - Can change API
- **Analogy:** "Manufacturer" of a component

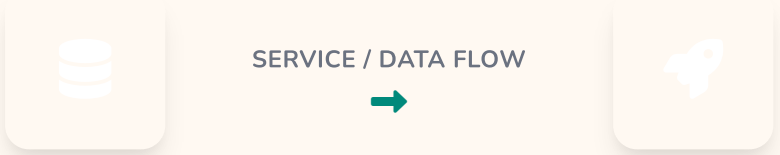
"I have the data"

DOWNSTREAM

THE CONSUMER

- **Dependent** - Needs Upstream to function
- **Low Power** - Suffers if Upstream changes
- **Analogy:** "Assembler" using the component

"I need the data"



"Senior Engineer" Check

When integrating two systems, always ask: **"Who can break whom?"** Mitigation: Use Contract Tests

PATTERN 1

Customer / Supplier Development

Strategic partnership where upstream prioritizes downstream success

Definition

STRATEGIC ALIGNMENT

A relationship where **Upstream (Supplier)** and **Downstream (Customer)** have a distinct working relationship. The Downstream team's success is a **priority** for the Upstream team.

Visual Flow

The Mechanism

1. **Planning**
   Joint roadmap sessions
2. **Negotiation**
   Downstream requests, Upstream schedules
3. **Validation**
   Downstream provides automated tests

Success Criteria

- Same budgetary unit/aligned incentives
- Acceptable communication overhead

Failure Mode

If Upstream has **more important** customers (external vs internal), the Downstream becomes a **second-class citizen**.

→ Decays into *Hostile*

PATTERN 2

Conformist

Abandoning your model to adopt the upstream's language

Definition

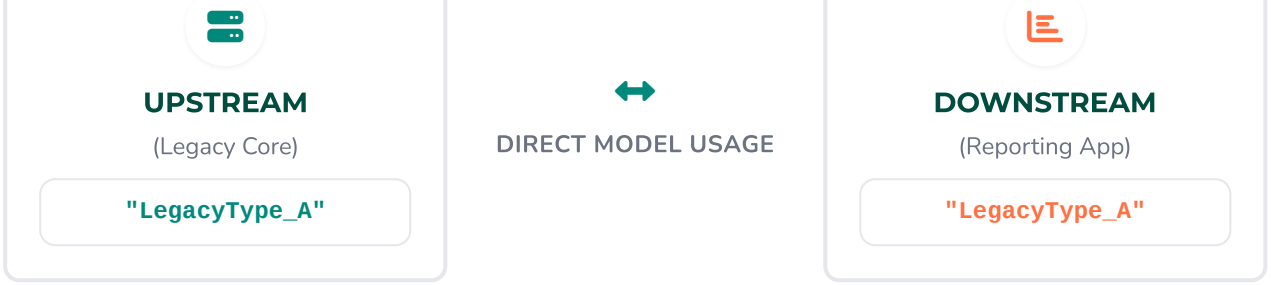
MODEL ADOPTION

Downstream team abandons its own **Ubiquitous Language** and model, adopting the Upstream's model completely to reduce translation costs.

When to Use

- **Industry Standard** - SWIFT (banking), HL7 (healthcare)
- **No Domain Logic** - Simple reporting dashboard
- **ACL Cost > Benefit** - Translation too expensive

Visual Structure



The Risk: "Semantic Saturation"

Your code becomes a **mirror of the external system**. If external uses cryptic codes (e.g., Status = 47), your code must also understand them.

Losing the expressiveness of DDD

PATTERN 3

Partnership

Tight alignment through shared kernel and collaborative development

Definition

COLLABORATIVE UNION

Two contexts are so tightly aligned they effectively **merge into one team** or two sub-teams of one larger initiative. They share a **Shared Kernel** of code and model.

The Shared Kernel

A subset of the **domain model**, **code**, or **database schema** that is shared between two teams.

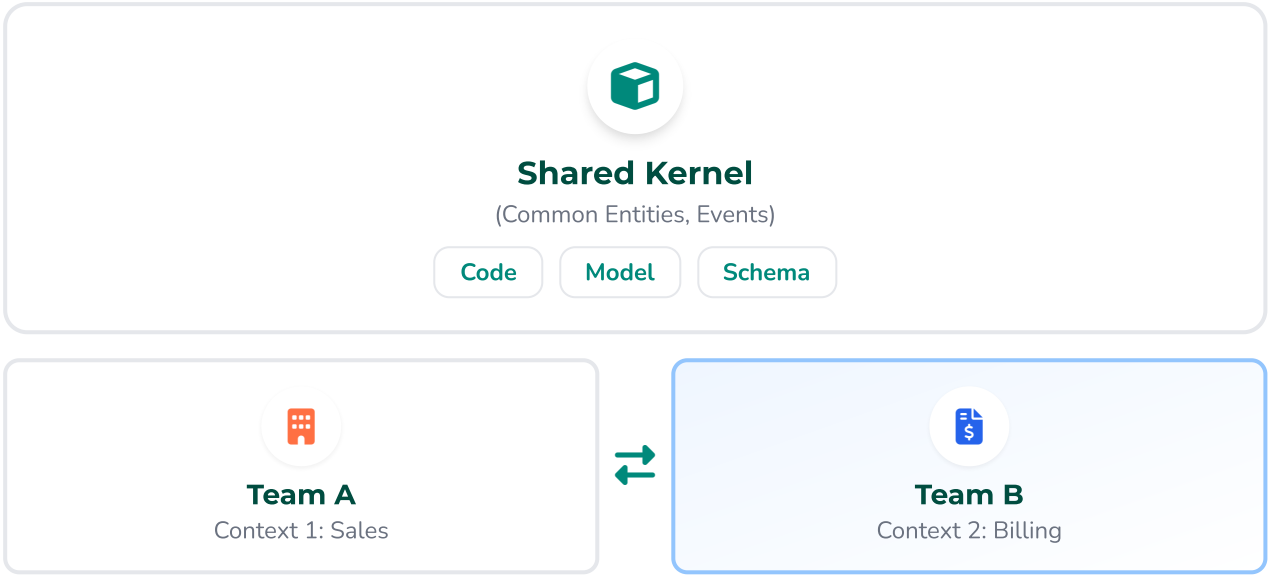
- Common Entities
- Shared Events

Trade-offs

HIGH COST
Changes require coordination, both must deploy

HIGH REWARD
Zero integration friction

Visual Structure



Automotive Example

Toyota & Denso Partnership

Legally separate but often **co-develop engine control units (ECUs)**, sharing a kernel of technical specifications.

PATTERN 4

Separate Ways

The boldest strategic decision — complete autonomy

Definition

COMPLETE INDEPENDENCE

Two contexts have **no relationship**. They do not integrate. They solve their problems **independently**.

Why Do This?

Integration Cost > Value

Effort to map/translate data exceeds sharing benefit

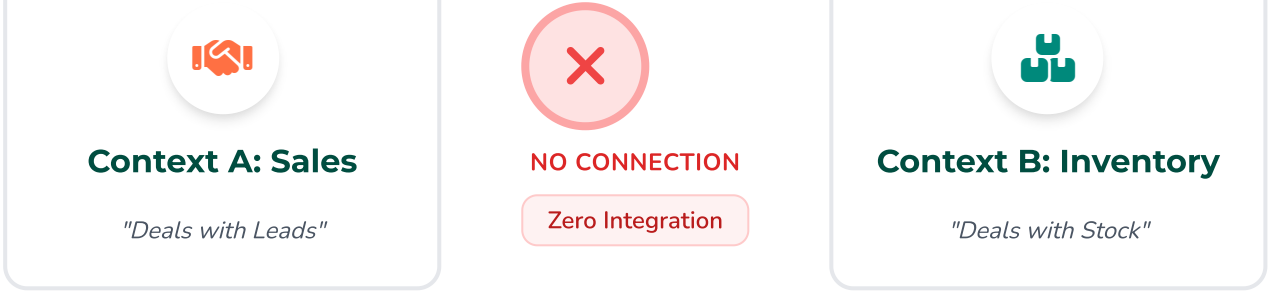
Incompatible Languages

Logistics language cannot speak Finance language

Note on Data Sync

They might eventually sync data via **batch file dump (ETL)** at night, but functionally, they are Separate Ways.

Visual Structure



The Philosophy

"Why force integration when independence works?" — Sometimes the best architecture is no architecture at all. Let systems evolve independently.

PATTERN 5

Open Host Service (OHS)

Publishing stable APIs for universal integration

Definition

PUBLIC API STRATEGY

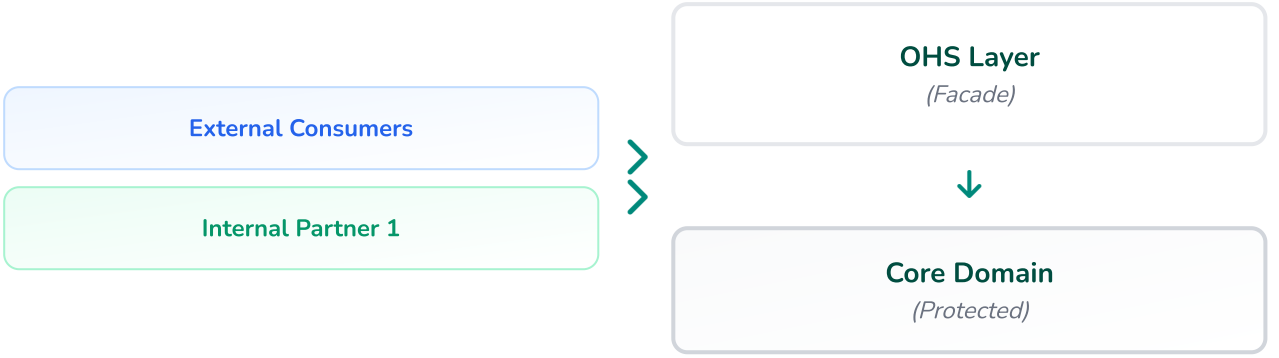
Upstream context defines a **protocol (API)** open to the entire world or enterprise, making integration easy for anyone.

Published Language

A formal language used for communication between contexts, especially with OHS.

- REST / OpenAPI
- GraphQL
- gRPC Protobuf
- SOAP (Legacy)

Architectural View



Key Rules

Versioning Mandatory
Never change v1. Create v2.

Documentation First-Class
If not documented, not Open.

Generic, Not Specific
Serve unknown future consumers.

PATTERN 6

Anticorruption Layer (ACL) — The Deep Dive

The shield pattern protecting your domain from legacy contamination

The "Shield" Pattern

TRANSLATION LAYER

ACL is a layer between contexts that **translates the Upstream model** into the Downstream's Ubiquitous Language. It filters out **"noise" and "bad design"** of the external world.

Critical for Seniors

Most companies have **legacy systems (ERP, Mainframe)**. Wrapping new DDD code without ACL results in **"Anemic Domain Models"** dictated by 1995 database schema.

The Architecture of an ACL

DOWNSTREAM (New Core)
TelemetryData Entity

→ **Messy Input**
DTO: {"type_1": 5}

→ **2. The ACL (Translator)**

Adapters | Facades

Translation logic (Map "ENG_ST_01" to "EngineStatus.OVERHEATING")

→ **Clean Output**
Entity: OrderStatus

→ **UPSTREAM (Legacy ERP)**
CSV/XML, Binary Protocol

Never let legacy DTOs leak into your Domain — Isolate the mess at the boundary.

IMPLEMENTATION

Implementing an ACL — Step-by-Step Guide

Three-phase implementation strategy for clean domain isolation

1 Define the Ideal Interface (Ports & Adapters)

Define what you wish you had inside your new context

```
public interface CarTelemetryPort {
    TelemetryData getCurrentTelemetry(String vin);
}
```

2 Build the Adapter (The ACL)

Implement the interface but talk to the messy world

```
public class LegacyTelemetryAdapter
    implements CarTelemetryPort {

    // Call legacy system (returns CSV/XML)
    LegacyResponse raw = client.get(vin);

    // TRANSLATION LOGIC
    EngineStatus status =
        mapLegacyCode(raw.getCode());
        // "ENG_ST_01" → OVERHEATING

    return new TelemetryData(status, raw.getRpm());
}
```

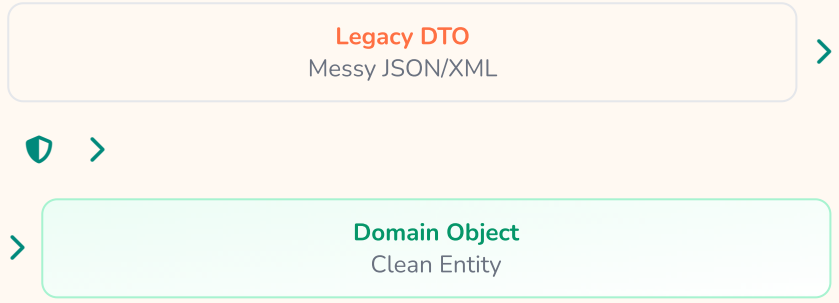
3 Isolation

Legacy DTOs never leak into Domain

Clean Domain Model

Ensure the legacy DTO (LegacyResponse) never leaks past the Adapter package into your pristine Domain logic.

Visual Flow



Translation is key — Don't just copy-paste. Map concepts, not just data.

EXPERT INSIGHTS

ACL — Expert Insights from the Field

Real-world wisdom from StackOverflow & Reddit discussions

SOURCE
r/java, r/architecture

"My ACL is becoming huge"

THE PROBLEM

Upstream model so complex that mapping logic is **50x** the business logic.

THE FIX

- **Refactor Map:** Use MapStruct/ModelMapper to reduce boilerplate
- **Question Strategy:** Push back on Upstream to change their model

If ACL interprets too much, you might be using the wrong pattern.

SOURCE
Community Consensus

"Should ACL contain validation?"

THE ANSWER

YES

ACL is the **border guard**. Throw out or sanitize garbage before it enters your pristine Domain.

EXAMPLES

- Negative prices
- Invalid dates
- Null critical fields

CONCEPT
Pattern Comparison

ACL vs. Facade

FACADE

Takes 10 methods, makes 1

Simplifies complexity

ANTICORRUPTION LAYER

Takes "Party" → "Person"

Translates semantics

> **"ACL translates meaning, Facade reduces complexity."**

COMMUNICATION STRATEGY

Context Communication — Sync vs. Async

Distinguishing Process (Sync) from Events (Async) in Context Mapping

1. Synchronous

REST / GRPC

Tight Coupling in Time

PATTERN

Customer/Supplier, OHS

USE CASE

Real-time inventory check

Flow Example

Sales → *"Stock Available?"* → Inventory → *"Yes"*

2. Asynchronous

KAFKA / RABBITMQ

Loose Coupling

PATTERN

ACL + Event Bus

USE CASE

Shipping notification

Flow Example

Sales → *"Order Placed"* → Event Bus → Shipping, Invoicing, Analytics

Strategic Decision Matrix

- **High Consistency?** → Sync
- **High Availability/Volume?** → Async

CASE STUDY

Tesla — The Agilist Competitor

Software-defined vehicles through strategic context mapping

Context A: Vehicle

THE EDGE

- **C++ Embedded** firmware
- Limited connectivity
- Real-time processing

Context B: Tesla Cloud

THE CORE

- **Java/Python** stack
- AWS infrastructure
- OTA update hub

Context C: Mobile App

THE INTERFACE

- iOS/Android apps
- Third-party API access
- Customer-facing UI

Patterns Implemented

Strategic design in action

Open Host Service

Tesla Cloud exposes robust API for mobile app and third-party developers

- Stable, documented endpoints

Anticorruption Layer

Car firmware translates cloud commands to CAN-bus protocols

- "SetTemp=22" → 0x3F 0x11

Why Tesla Wins vs VW

- **VW:** Tight coupling between Infotainment and Chassis → Integration friction, slow iterations
- **Tesla:** Decoupled using defined bus architecture (OHS) → Cloud team iterates weekly without breaking Car team

CASE STUDY

Stellantis — The Merger Challenge

Integrating legacy systems through Anticorruption Layers

The Challenge

CONVERGENCE NIGHTMARE

Fiat Group
Legacy Italian ERP systems

PSA Group
Newer SAP implementations

Goal
One unified "STLA" software platform

Strategic Application

Cannot rewrite overnight. Must use **Anticorruption Layers (ACL)** to bridge legacy systems while preserving unified API.

Implementation Phases

Build "Unified Ordering Context"
Create new clean domain model as central hub

Build ACL to Fiat Legacy
Mapping Type A → Type B (Italian ERP)

3 **Build ACL to PSA SAP**
Mapping Type X → Type B (SAP)

Result

Internal teams see **clean "Unified API"**. ACLs handle spaghetti translation in background, enabling gradual migration without business disruption.

CASE STUDY

Toyota — The Partnership Model

Supply chain integration through Shared Kernel

The Pattern

PARTNERSHIP MODEL

- **Toyota** with Tier-1 suppliers
- Denso, Aisin, etc.
- Tight feedback loop

Ubiquitous Language

SAME LANGUAGE, SAME METRICS

> **Literally speak the same language regarding "Just-In-Time" delivery metrics.**

Shared Kernel

- Engineering data (CAD)
- Technical specifications
- Shared ECU source code

VW Approach

- Treats suppliers as **"Commodities"**
- Customer/Supplier pattern
- Conformist adaptation

Toyota Approach

- Treats suppliers as **Partners**
- Shared Kernel pattern
- Collaborative development

Software Implication

- **Toyota:** Harder to break, faster & higher quality
- **VW:** Easier to swap, suffers translation friction

ONLINE TOOL

Context Mapper

DSL-based context mapping and diagram generation

Tool Details

CORE INFORMATION

URL
contextmapper.org

TYPE
DSL & Generator

WHY VITAL
Prevents "Draw-only" diagrams • Generates visual diagrams • Suggests refactoring risks

Try It Now

Go to Context Mapper online editor, paste the code, and generate diagrams instantly.

- Visual output in seconds

Example DSL Syntax

```
BoundedContext SalesContext {
  type = TEAM
  responsibilities = "Manage Sales Orders"
  implementationTechnology = "Spring Boot"
}

BoundedContext InventoryContext {
  type = SYSTEM
  responsibilities = "Stock Management"
}

// Define the Relationship
SalesContext -> InventoryContext {
  CustomerSupplier // This is the Pattern!
  implementationTechnology = "REST/HTTP"
  upstreamResponsibilities = "provide stock data"
  downstreamResponsibilities = "consume stock data"
}
```

ONLINE TOOL

Structurizr

C4 Model-based architecture visualization tool

Tool Overview

BASED ON C4 MODEL

LEVEL 1
Context Map (DDD)

LEVEL 2
Containers (Apps, APIs, DBs)

KEY BENEFIT
Clickable, explorable architecture maps

Export Options

- PlantUML
- Mermaid
- Ilograph

How It Fits Day 6

Context Mapper

Good for defining relationships between bounded contexts

Structurizr

Better for management presentation (C-Level)

Business Landscape Integration

Puts DDD Contexts into broader business architecture view for stakeholder communication

MINI-PROJECT

The Unified Fleet System

Integrating mobile apps with legacy systems through DDD patterns

Scenario

YOUR ROLE

You are the **Lead Architect** for a logistics company (competitor to Deutsche Post DHL) integrating modern apps with legacy infrastructure.

Focus on context mapping and integration patterns

The Goal

Connect the **Driver App** to update the **Legacy Tracking system** without the mobile app knowing about `COBOL` or binary protocols.

System Landscape

Legacy Tracking

- Mainframe COBOL
- Binary protocol
- Package locations

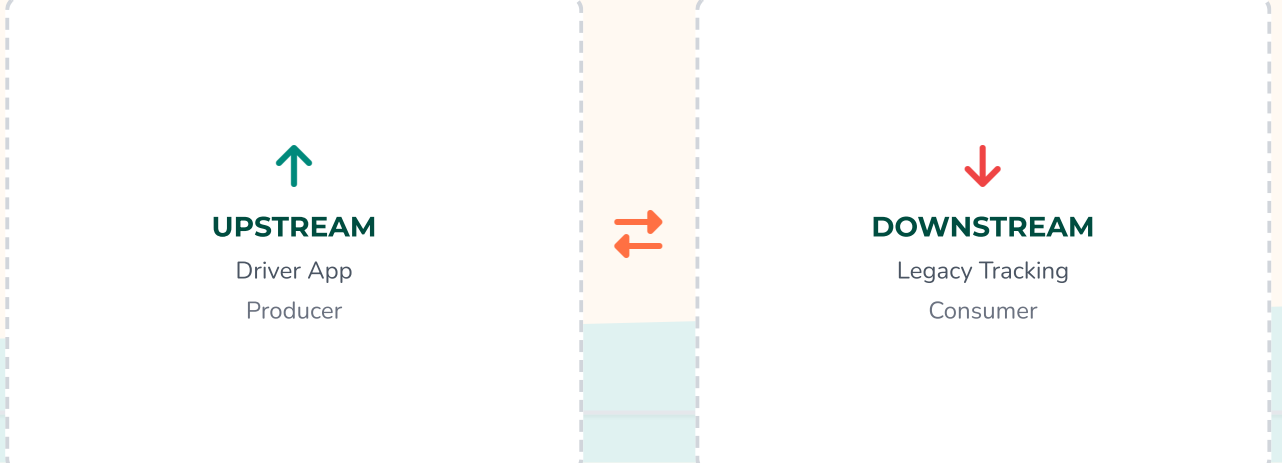
Customer Portal

- React-based
- Frontend UI
- Customer facing

Driver App

- Mobile application
- Status updates
- Real-time sync

Step 1: Draw the Context Map



MINI-PROJECT

Solution Design

Implementing Anticorruption Layer for legacy integration

Pattern Selection

- **Anticorruption Layer (ACL)** — Legacy system too complex for direct access

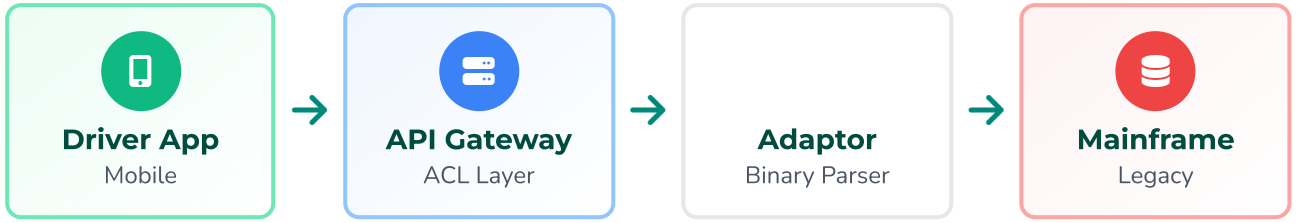
Translation Flow

`JSON` from Mobile

Binary Parser

3 `0x01 0xFF` byte stream

Architecture Diagram



Request Payload

```
{ "status": "DELIVERED", "lat": 52.5, "lon": 13.4 }
```

Response

200 OK

Legacy errors ignored

MINI-PROJECT EXTENSION

The OHS Requirement

Enabling third-party integration through Open Host Service

New Requirement

BUSINESS NEED

Allow **Third-Party E-commerce shops** (Amazon, Shopify) to query package delivery status.

The Challenge

INTERNAL API LIMITATION

Cannot give internal API to external partners — **too internal and changes often**. Need stable, documented interface.

Open Host Service (OHS)

PUBLISHED LANGUAGE & ENDPOINT

PUBLISHED LANGUAGE

Standard JSON API

ENDPOINT

`GET /public/api/v1/tracking/{id}`

Architecture with OHS Layer

Change ACL Logic Freely

Amazon unaffected as long as OHS contract stable

Rate Limiting

Protects Mainframe from overload

COMMUNICATION STRATEGY

Inter-Context Communication Design

Define communication strategies for every context relationship

Driver App → Fleet Mgmt

Mobile to Backend Service

PATTERN Customer/Supplier	DIRECTION Upstream → Downstream
SYNC/ASYNC Async	PROTOCOL MQTT / REST
FAILURE POLICY Queue locally, retry	OWNERSHIP Fleet Team provides SDK

Fleet Mgmt → Legacy ERP

Backend to Mainframe System

PATTERN ACL	DIRECTION Upstream → Downstream
SYNC/ASYNC Sync (Adapter needed)	PROTOCOL gRPC / Proprietary TCP
FAILURE POLICY Fallback to cached data	OWNERSHIP Fleet Team owns Adapter

Key Takeaway

Never leave communication strategy to chance. **Define it early** for every context relationship to prevent integration nightmares.

TERMINOLOGY

Critical Terms & Definitions

Essential vocabulary for strategic design and context mapping

Ubiquitous Language

SHARED VOCABULARY

Shared language used by developers and domain experts within **specific Bounded Context**.

Changes between contexts

Published Language

FORMAL INTERFACE

Formal language used for communication between contexts, especially with OHS.

ISO 20022 EDIFACT OpenAPI

Big Ball of Mud

System with **no clear Bounded Contexts**. Everything connected to everything.

Goal: Identify & break apart

Partnership

Relationship where two contexts **align goals** and development processes.

Maximize integration benefits

Conformist

Downstream adopts Upstream's model **entirely** to reduce friction.

Cost:
Loss of domain expressiveness

Benefit:
Reduced translation overhead

PATTERN COMPARISON

Comparing the Patterns

Continuum of coupling: from tight integration to complete independence

TIGHT COUPLING
HARD TO MOVE • FAST TO EXECUTE

LOOSE COUPLING
EASY TO MOVE • TRANSLATION OVERHEAD

Shared Kernel	Partnership	Customer/Supplier	Conformist	ACL/OHS	Separate Ways
High cost, high reward	Tight alignment	Clear contracts	Adopt upstream model	Translation layer	No coupling

GO UP

Need absolute consistency? Choose Shared Kernel or Partnership

GO DOWN

Want autonomy? Choose ACL or OHS for loose coupling

GO SIDEWAYS

Hate the other team? Choose Separate Ways

PITFALLS & ANTI-PATTERNS

Common Pitfalls & Anti-Patterns

Avoid these traps when implementing strategic design patterns

The "Distributed Monolith"

SYNCHRONOUS CHAIN TRAP

MISTAKE

Split code into microservices but communicate **synchronously in chain**. If Service A is down, whole site is down.

FIX

Introduce **Asynchronous Messaging (Events)** between critical contexts.

The "God" OHS

BLOATED API TRAP

MISTAKE

Trying to make **one API** satisfying every possible use case. Bloated and impossible to version.

FIX

Keep OHS **focused**. Let consumers build their own ACL on top if needed.

"Trust" Fallacy

ASSUMPTION TRAP

MISTAKE

Assuming Upstream team will **never break contract**.

FIX

Use **Consumer-Driven Contracts (Pact)**. Downstream writes tests, Upstream runs them.

Ignoring Organizational Politics

WISHFUL THINKING TRAP

MISTAKE

Drawing map showing "Partnership" when **Team A refuses to talk to Team B**.

FIX

Be honest. If separate ways, draw as separate ways. Reflect **reality**, not wishful thinking.

SUMMARY

Summary & Recap of Day 6

Strategic Design 2 - Context Mapping: Key Takeaways and Learning Outcomes

What We Learned

- **Mapping**
  Visualize enterprise landscape using Upstream/Downstream flows
- **Patterns**
  Toolbox of 7 patterns for organizational problems
- **Implementation**
  ACLs as Adapters, Facades, and Translators
- **Strategy**
  Sync vs Async is a business decision

The "Day 6" Takeaway

Strategic Design is about risk management. By defining boundaries, we limit the blast radius of failures and changes.

Patterns Covered: The Complete Toolkit

Shared Kernel Tight coupling	**Partnership** Aligned goals	**Customer/Supplier** Clear contracts
Conformist Adopt model	**ACL** Translation layer	**OHS** Public API

POST-TRAINING

Connecting to the Next Phase

Self-study roadmap and continued learning journey

Read

ESSENTIAL REFERENCE

Domain-Driven Design by Eric Evans

- Chapter 14: Distillation
- Chapter 15: Context Mapping

Explore

HANDS-ON TOOLS

Download **Context Mapper**

- Model simple e-commerce system
- Sales, Inventory, User contexts

Looking Forward

STRATEGIC REFACTORING

Strangler Fig Pattern
Use OHS to slowly strangle legacy system

Evolutionary Architecture
Incremental pattern application

Target Architecture
Refactor to context-driven design

Practice

Map your current team's interactions. Identify one "pain point" and assign a DDD pattern to it.

Code Lab

Implement ACL for your legacy system. Use Ports & Adapters pattern.

Continue: **Day 7**

EXAM PREP

Q&A Preparation — The iSAQB Exam View

Mock questions and assessment readiness

Isolation Pattern

Isolate core domain from volatile external model?

- ✓ Anticorruption Layer (ACL)

Tight Coupling

Two teams, same code base, tight loop?

- ✓ Shared Kernel

External Integration

Expose capabilities to unknown external developers?

- ✓ OHS + REST/OpenAPI

4 Separate Ways Risk

Primary risk of Separate Ways pattern?

- ✓ Data duplication & inconsistency

Closing Note

ARCHITECTURE PHILOSOPHY

Architecture is the **art of drawing lines that matter**

Thank you for your dedication to Strategic Design and Context Mapping.

Exam Prep Checklist

- ✓ **Pattern Recognition** — Identify 6 patterns in scenarios
- ✓ **Upstream/Downstream** — Direction & power dynamics
- ✓ **Sync vs Async** — Business decision factors
- ✓ **ACL Implementation** — Adapters & isolation