Notes 1

DAY 6: Strategic Design 2 - Context Mapping

iSAQB-DDD Advanced Training for Senior Engineers & Architects Date: 09-Jan-2026

Page 1: Executive Summary & Day Objectives

Session Focus:

Strategic Design is the bridge between business strategy and software architecture. While Tactical Design (Entities, Aggregates) handles code quality, **Context Mapping** handles organizational boundaries and team dynamics.

The "Last Day" Milestone:

We conclude the core training by synthesizing everything learned:

1. **Distillation:** Identifying the Core Domain.
2. **Relationships:** Defining how Bounded Contexts interact.
3. **Integration:** Technical strategies for communication (ACL, OHS).
4. **Transformation:** Moving from current state (Legacy) to future state (DDD).

Critical Outcome:

You will be able to draw a **Context Map** for your organization, identify friction points using DDD patterns (Customer/Supplier, ACL), and propose technical solutions (e.g., implementing an Anticorruption Layer) to decouple your teams and code.

Page 2: Targeted Learning Goals & Alignment Matrix

Learning Goal ID	Goal Description	Strategic Value
LG 6-1	**Identify Relationships:** Map Upstream/Downstream relationships between teams.	Prevents "Big Ball of Mud" architecture by defining clear ownership and data flow direction.
LG 6-2	**Implement OHS:** Design an Open Host Service for external integration.	Enables business ecosystems by exposing stable, documented APIs without leaking internal domain logic.

Learning Goal ID	Goal Description	Strategic Value
LG 6-3	**Design ACL:** Build an Anticorruption Layer to isolate legacy systems.	Protects new Core Domain investments from being "polluted" by chaotic external models.
LG 6-4	**Define Contracts:** Establish Customer/Supplier development teams.	Aligns engineering metrics (DORA) with business value delivery streams.
LG 6-5	**Apply Domain Events:** Decouple contexts using asynchronous messaging.	Increases system resilience and availability compared to synchronous REST/RPC calls.
LG 6-6	**Evaluate Patterns:** Choose between Partnership, Separate Ways, and Conformist.	Balances integration cost against business benefit (ROI of integration).

Page 3: Visualizing the Context Map (The Big Picture)

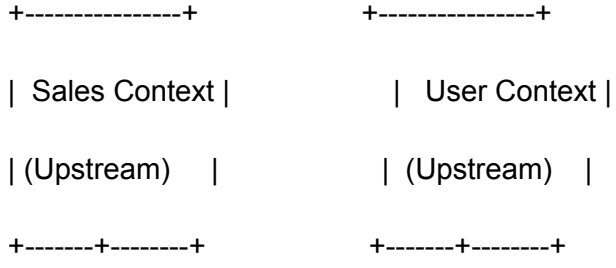
What is a Context Map?

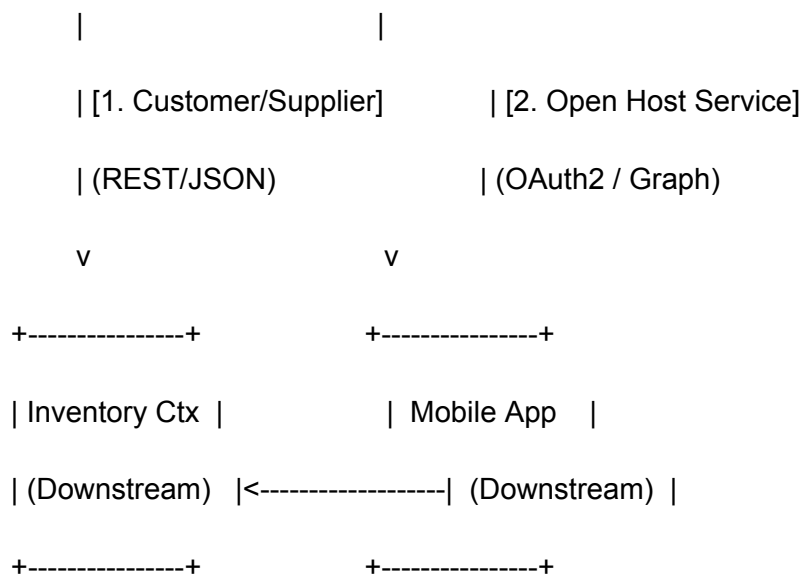
A Context Map is a living document (usually a diagram) that shows the **Bounded Contexts** in the system and the **relationships** between them. It is a map of the "landscape" of your software organization.

The Map Layers:

1. **Name of Context:** (e.g., Sales, Inventory, Shipping).
2. **The Team:** Who owns this code? (Crucial for Conway's Law).
3. **The Relationship:** How do they talk? (OHS, ACL, etc.).
4. **The Technology:** (Kafka, REST, gRPC, Shared DB).

Visual Structure:





*Note: Arrows indicate the direction of the **service/value** or the **flow of language**.*

Page 4: Core Concept - Upstream vs. Downstream Dynamics

The Fundamental Duality:

In any integration, one side defines the model/language (Upstream), and the other consumes it (Downstream).

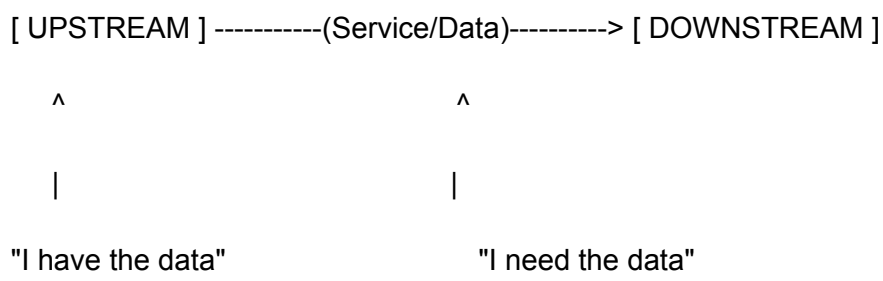
Upstream (The Provider):

- **Characteristics:** Independent, less dependent on the Downstream.
- **Power:** High. They can change the API.
- **Analogy:** The "Manufacturer" of a component.

Downstream (The Consumer):

- **Characteristics:** Dependent, needs the Upstream to function.
- **Power:** Low. They suffer if the Upstream changes or breaks.
- **Analogy:** The "Assembler" using the component.

Visualizing Power & Flow:



The "Senior Engineer" Check:

When integrating two systems, always ask: *"Who can break whom?"*

- If **Inventory** (Upstream) changes its API, **Sales** (Downstream) breaks.
- **Mitigation:** Downstream must implement tests (Contract Tests) to ensure Upstream doesn't break them unknowingly.

Page 5: Pattern 1 - Customer / Supplier Development

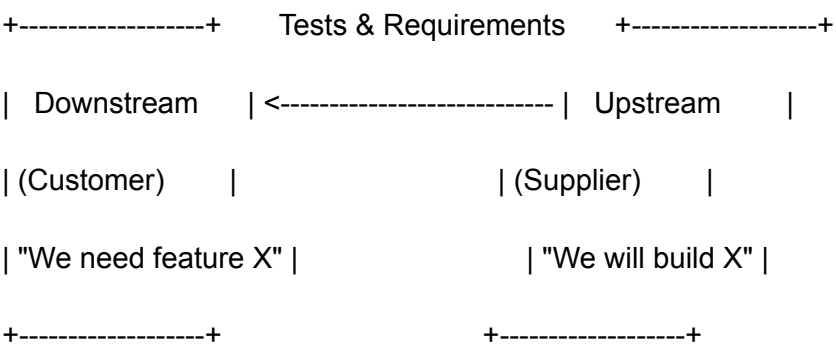
Definition:

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

The Mechanism:

1. **Planning:** Joint roadmap sessions.
2. **Negotiation:** Downstream requests features; Upstream schedules them.
3. **Validation:** Downstream provides automated tests to Upstream.

Visual Flow:



Success Criteria:

- Both teams are part of the same budgetary unit or aligned incentives.
- Communication overhead is acceptable.

Failure Mode:

If the Upstream team has other "更重要" (more important) customers (e.g., external paying clients vs. internal teams), the Downstream internal team becomes a second-class citizen. The relationship decays into **Hostile**.

Page 6: Pattern 2 - Conformist

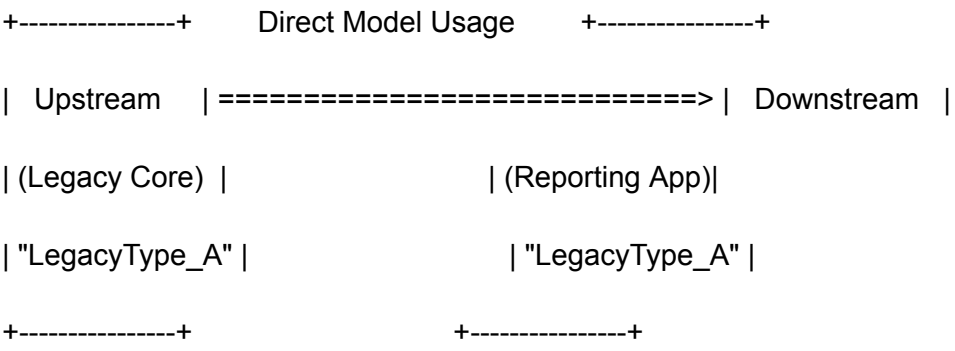
Definition:

The Downstream team chooses to **abandon** its own Ubiquitous Language and model. They adopt the Upstream's model completely to reduce translation costs.

When to Use:

- The Upstream model is an industry standard (e.g., SWIFT for banking, HL7 for healthcare).
- The Downstream context has no distinct domain logic (e.g., a simple reporting dashboard).
- The cost of maintaining an ACL exceeds the benefit.

Visual Structure:



The Risk:

"Semantic Saturation." Your code becomes a mirror of the external system. If the external system uses cryptic codes (e.g., `Status = 47`), your code must also understand `Status = 47`. You lose the expressiveness of DDD.

Page 7: Pattern 3 - Partnership

Definition:

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

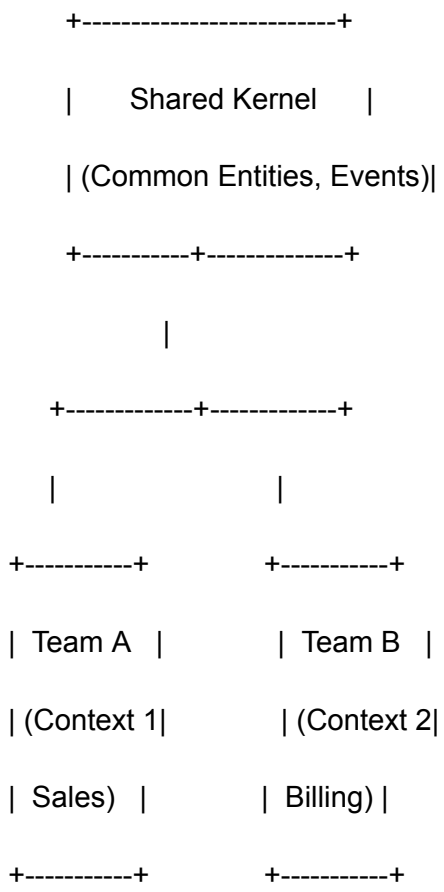
The Shared Kernel:

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

- **High Cost:** Changes to the Shared Kernel require coordination between BOTH teams. If Kernel changes, BOTH must deploy.

- **High Reward:** Zero integration friction for the shared parts.

Visual:



Automotive Example:

Toyota and **Denso** (parts supplier). They are legally separate but often co-develop the engine control units (ECUs). They share a kernel of technical specifications.

Page 8: Pattern 4 - Separate Ways

Definition:

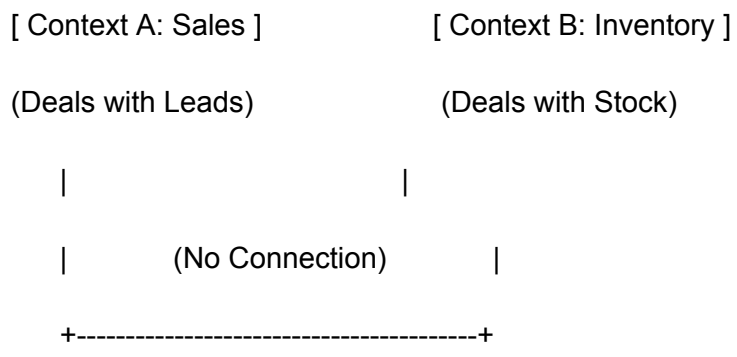
The boldest strategic decision. The two contexts have **no relationship**. They do not integrate. They solve their problems independently.

Why do this?

- **Integration Cost > Value:** The effort to map and translate data is higher than the benefit of sharing it.

- **Ubiquitous Languages are Incompatible:** You cannot force a "Logistics" language to speak "Finance" language without corruption. They are fundamentally different worlds.

Visual:



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

Page 9: Pattern 5 - Open Host Service (OHS)

Definition:

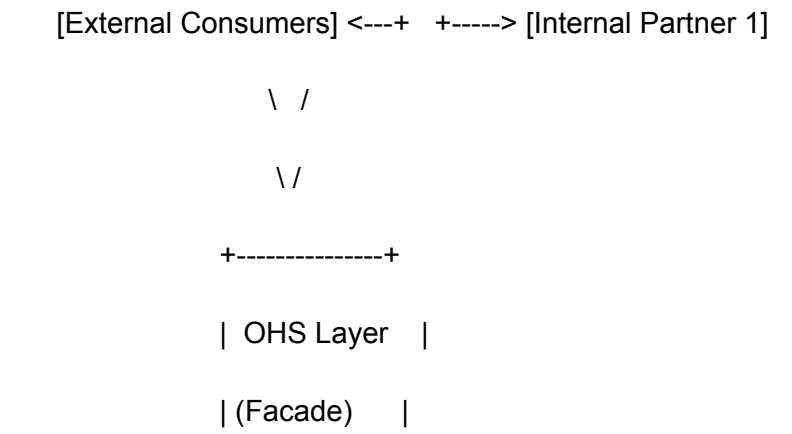
The Upstream context defines a protocol (API) that is open to the entire world or the entire enterprise. It aims to make integration easy for anyone.

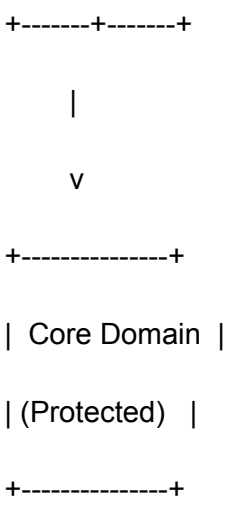
The Published Language:

An OHS requires a "lingua franca"—a standard data format.

- **Examples:** REST (OpenAPI/Swagger), GraphQL, gRPC Protobuf, SOAP (older).

Architectural View:





Key Rules:

1. **Versioning is Mandatory.** Never change v1 of an API. Create v2.
2. **Documentation is First-Class.** If it's not documented, it's not Open.
3. **Generic, not Specific.** The API should be broad enough to serve unknown future consumers.

Page 10: Pattern 6 - Anticorruption Layer (ACL) - The Deep Dive

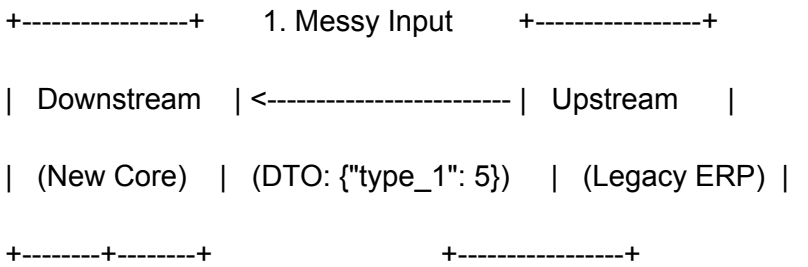
The "Shield" Pattern:

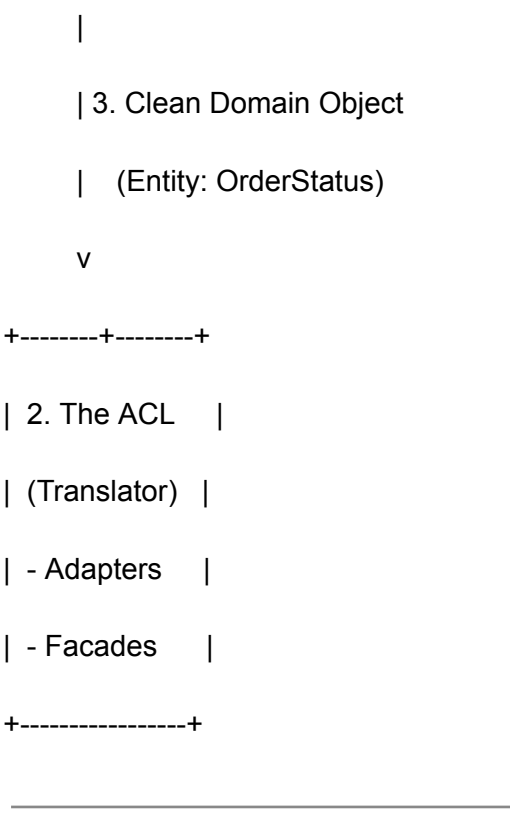
An ACL is a layer between contexts that translates the Upstream model into the Downstream's Ubiquitous Language. It filters out the "noise" and "bad design" of the external world.

Why is it critical for Seniors?

Most companies have legacy systems (ERP, Mainframe). Wrapping new DDD code around these systems without an ACL results in "Anemic Domain Models" where your logic is dictated by a database schema from 1995.

The Architecture of an ACL:





Page 11: Implementing an ACL - Step-by-Step Guide

Step 1: Define the Ideal Interface (Ports & Adapters)

Inside your new context, define what you *wish* you had.

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

Step 2: Build the Adapter (The ACL)

Create a class that implements the interface but talks to the ugly world.

```
public class LegacyTelemetryAdapter implements CarTelemetryPort {
    private ExternalLegacyClient client; // The messy upstream
    public TelemetryData getCurrentTelemetry(String vin) {
        // Call the legacy system (returns CSV or XML)
        LegacyResponse raw = client.get(vin);
```

```
        // TRANSLATION LOGIC (The Anti-Corruption)
        // Map legacy code "ENG_ST_01" to Domain Concept "EngineStatus.OVERHEATING"
        EngineStatus status = mapLegacyCode(raw.getCode());
        return new TelemetryData(status, raw.getRpm());
    }
}
```

Step 3: Isolation

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

Page 12: ACL - Expert Insights from the Field (StackOverflow/Reddit Analysis)

Insight 1: "My ACL is becoming huge."

- **Source:** *r/java, r/architecture*
- **The Problem:** The Upstream model is so complex that the mapping logic is 50x the business logic.
- **The Fix:**
    - **Refactor the Map:** Use libraries like MapStruct or ModelMapper to reduce boilerplate.
    - **Question the Strategy:** If the ACL is doing too much "interpretation," maybe you aren't using the Upstream correctly. Push back on the Upstream to change their model (Customer/Supplier dynamic).

Insight 2: "Should ACL contain validation?"

- **Consensus: Yes.** The ACL is the border guard. If the Upstream sends garbage (e.g., negative prices), the ACL should throw it out or sanitize it *before* it enters your pristine Domain. Never trust external data.

Insight 3: "ACL vs. Facade"

- **Difference:** A Facade simply simplifies a complex interface. An ACL *translates* semantics.
    - *Facade:* Takes 10 methods, makes 1.
    - *ACL:* Takes "Party" (ERP term) and makes it "Person" (CRM term).

Page 13: Context Communication - Synchronous vs. Asynchronous

The Impact of Choice:

Your Context Mapping diagram must distinguish between **Process** (Sync) and **Events** (Async).

1. Synchronous (REST/gRPC):

- **Tight Coupling in Time:** Downstream waits for Upstream.
- **Pattern:** Customer/Supplier or OHS.
- **Use Case:** "Sales" needs real-time "Inventory" check before selling.
- **Visual:**

  Sales ---[Is Stock Available?]--> Inventory ---[Yes]--> Sales

2. Asynchronous (Kafka/RabbitMQ):

- **Loose Coupling:** Upstream fires and forgets. Downstream processes eventually.
- **Pattern:** Often used with ACL to buffer changes.
- **Use Case:** "Shipping" needs to know "Order Placed". It doesn't need to happen in the same millisecond.
- **Visual:**

  Sales ---[Event: OrderPlaced]--> (Bus) --> Shipping

  --> Invoicing

  --> Analytics

Strategic Decision Matrix:

- High Consistency Requirement? -> **Sync**.
- High Availability Requirement / High Volume? -> **Async**.

Page 14: Case Study - Tesla (The Agilist Competitor)

Context:

Tesla is the gold standard for over-the-air (OTA) updates and software-defined vehicles. They treat the **Vehicle** as a downstream node of a massive cloud infrastructure.

The Map:

1. **Context A: Vehicle (The Edge):** C++ embedded, limited connectivity.

2. **Context B: Tesla Cloud (The Core):** Java/Python, AWS/Custom.
3. **Context C: Mobile App (The Interface).**

Patterns Used:

- **Open Host Service (OHS):** The Tesla Cloud exposes a robust API for the Mobile App and Third-Party developers.
- **Anticorruption Layer:** The Car's firmware acts as an ACL for the cloud. The Cloud sends "SetTemp = 22". The Car translates that to specific CAN-bus messages to the HVAC controller (`0x3F 0x11`). The Cloud doesn't know about CAN-bus.

Why they win (vs VW):

VW historically had tight coupling between Infotainment (Harman) and Chassis. Tesla decoupled them using a defined bus architecture (OHS), allowing the Cloud team to iterate weekly without breaking the Car team.

Page 15: Case Study - Stellantis (The Merger Challenge)

Context:

Stellantis (Fiat + Chrysler + Peugeot) is a conglomerate of legacy brands. They face the nightmare of integrating distinct ERPs and supply chains.

The Problem:

- **Fiat Group:** Uses legacy Italian ERP systems.
- **PSA Group:** Uses newer SAP implementations.
- **Goal:** One unified "STLA" software platform.

Strategic Application:

They cannot rewrite everything overnight. They must use **Anticorruption Layers (ACL)**.

1. **Phase 1:** Build a new "Unified Ordering Context".
2. **Phase 2:** Build ACLs to talk to Fiat Legacy (Mapping Type A -> Type B).
3. **Phase 3:** Build ACLs to talk to PSA SAP (Mapping Type X -> Type B).
4. **Result:** The internal teams see a clean "Unified API", while the ACLs handle the spaghetti translation in the background.

Page 16: Case Study - Toyota (The Partnership)

Context:

Toyota invented the "Lean" manufacturing model. Their supply chain is a tight feedback loop.

The Pattern:

Toyota uses a **Partnership** model with Tier-1 suppliers (like Denso, Aisin).

- **Shared Kernel:** They share engineering data (CAD), specifications, and sometimes even source code for ECUs.
- **Ubiquitous Language:** They literally speak the same language regarding "Just-In-Time" delivery metrics.

The Comparison:

- **VW:** Often treats suppliers as "Commodities" (Customer/Supplier or Conformist).
- **Toyota:** Treats them as partners.
- **Software Implication:** Toyota's software integration is harder to break (tightly coupled), but faster and higher quality. VW's is easier to swap suppliers but suffers from translation friction.

Page 17: Online Tools & Platforms - Context Mapper

Tool: Context Mapper

- **URL:** http://contextmapper.org/
- **Type:** Domain Specific Language (DSL) & Generator.
- **Why it's vital:** It prevents "Draw-only" diagrams. You write code-like text, and it generates the diagrams and even suggests refactoring risks.

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"

}
```

Activity:

1. Go to the Context Mapper online editor.
2. Paste the code above.
3. Generate the diagram to see the visual output.

Page 18: Online Tools & Platforms - Structurizr

Tool: Structurizr

- **Concept:** Based on the C4 Model.
- **Usage:**
    - **Level 1 (Context):** The DDD Context Map.
    - **Level 2 (Containers):** Web Apps, APIs, Databases.
- **Benefit:** It creates a clickable, explorable map of your architecture.
- **Export:** Supports PlantUML, Mermaid, and Ilograph.

How it fits Day 6:

While Context Mapper is good for *relationships*, Structurizr is better for *presentation* to management (C-Level). It puts the DDD Contexts into a business landscape.

Page 19: Mini-Project - "The Unified Fleet System"

Scenario:

You are the Lead Architect for a logistics company (competitor to Deutsche Post DHL). You have three main systems:

1. **Legacy Tracking:** An old Mainframe system (COBOL). It knows where packages are but exposes a proprietary binary protocol.
2. **Customer Portal:** A new React-based frontend for customers.
3. **Driver App:** A mobile app for drivers to update status.

The Goal:

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

Step 1: Draw the Map

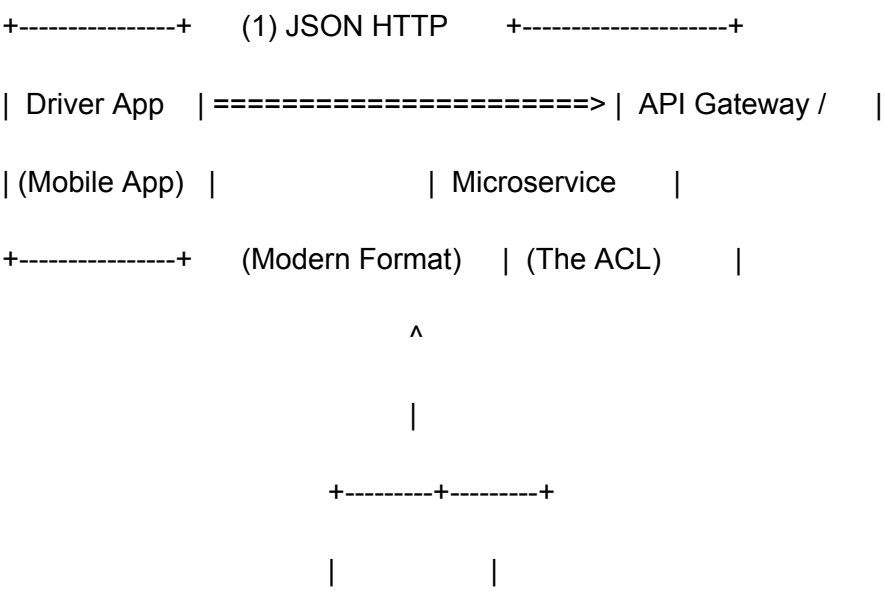
Identify Upstream (Driver App) and Downstream (Legacy). *(Space for your sketch)*

Page 20: Mini-Project - Solution Design

The Pattern Selection:

We need an **Anticorruption Layer (ACL)**. The Legacy system is too complex to be accessed directly by a mobile app.

The Architecture:



(2) Adaptor (3) Database

(Binary Parser) (Read Side)

|

v

+-----------------+

| Legacy Mainframe|

+-----------------+

Translation Detail:

- **App sends:** `{ "status": "DELIVERED", "lat": 52.5, "lon": 13.4 }`
- **ACL receives JSON:** Converts to internal object.
- **ACL Adaptor:** Transforms object into Mainframe byte stream `0x01 0xFF...`.
- **ACL Response:** Returns `200 OK` to App (ignoring Mainframe's obscure error codes).

Page 21: Mini-Project - Extension (The OHS Requirement)

New Requirement:

The company now wants to allow **Third-Party E-commerce shops** (like Amazon or Shopify) to query if a package is delivered.

The Challenge:

We cannot give them the internal API we built for the Driver App. It's too internal and changes often.

The Solution:

Create an **Open Host Service (OHS)** on top of the ACL.

- **Published Language:** A standard JSON API.
- **Endpoint:** `GET /public/api/v1/tracking/{id}`
- **Behavior:** This OHS calls the internal ACL, which calls the Mainframe.

Benefits:

- We can change the Mainframe/ACL logic freely. As long as the OHS contract remains the same, Amazon doesn't break.

- We can rate-limit the OHS (Throttling) to protect the Legacy Mainframe from overload.

Page 22: Designing the Inter-Context Communication Strategy

The Strategy Table:

When designing communication, fill this table out for every link in your map.

Attribute	**Driver App -> Fleet Mgmt**	**Fleet Mgmt -> Legacy ERP**
Pattern	Customer / Supplier	Anticorruption Layer (ACL)
Direction	Upstream (Fleet) -> Downstream (Driver)	Upstream (ERP) -> Downstream (Fleet)
Sync/Async	Async (Mobile background tasks)	Sync (Adapter needed for query)
Protocol	MQTT / REST	gRPC / Proprietary TCP
Failure Policy	Queue locally, retry later	Fallback to cached data
Ownership	Fleet Team provides SDK	Fleet Team owns the Adapter

Key Takeaway:

Never leave communication strategy to chance. Define it early.

Page 23: Critical Terms & Definitions

Ubiquitous Language (Recap)

The shared language used by developers and domain experts within a **specific** Bounded Context. Note: It changes between contexts!

Published Language

A formal language (often document-based) used for communication between contexts, especially with an Open Host Service. e.g., ISO 20022, EDIFACT.

Big Ball of Mud

A system with no clear Bounded Contexts. Everything is connected to everything. The goal of Context Mapping is to identify these and break them apart.

Partnership

A relationship where two contexts align their goals and development processes to maximize integration benefits.

Conformist

A relationship where the Downstream adopts the Upstream's model entirely to reduce friction, at the cost of domain expressiveness.

Page 24: Comparing the Patterns (Infographic)

Continuum of Coupling:

TIGHT COUPLING (Hard to move, fast to execute)

^

|

| [Shared Kernel] [Partnership]

|

|

| [Customer/Supplier]

|

|

| [Conformist]

|

|

| [ACL] [OHS]

|

|

+----------> LOOSE COUPLING

(Easy to move, translation overhead)

SEPARATE WAYS (No Coupling)

Decision Guide:

- Need absolute consistency? Go Up (Shared Kernel).
- Want autonomy? Go Down (ACL / OHS).
- Hate the other team? Go Sideways (Separate Ways).

Page 25: Common Pitfalls & Anti-Patterns

1. The "Distributed Monolith"

- **Mistake:** You split code into microservices (Contexts), but they communicate synchronously in a chain. If Service A is down, the whole site is down.
- **Fix:** Introduce Asynchronous Messaging (Events) between critical contexts.

2. The "God" OHS

- **Mistake:** Trying to make one API that satisfies every possible use case for every consumer. It becomes bloated and impossible to version.
- **Fix:** Keep OHS focused. If a consumer has weird needs, let them build their own ACL on top of the OHS.

3. "Trust"

- **Mistake:** Assuming the Upstream team will never break the contract.
- **Fix:** Use **Consumer-Driven Contracts** (Pact). The Downstream writes the tests, and the Upstream runs them.

4. Ignoring Organizational Politics

- **Mistake:** Drawing a map that shows a "Partnership" when in reality, Team A refuses to talk to Team B.
- **Fix:** Be honest. If they are separate ways, draw them as separate ways. It's better to reflect reality than wishful thinking.

Page 26: Summary & Recap of Day 6

What we learned:

1. **Mapping:** We can visualize the enterprise landscape using Upstream/Downstream flows.
2. **Patterns:** We have a toolbox of 7 patterns (OHS, ACL, Partnership, etc.) to solve specific organizational problems.
3. **Implementation:** ACLs are not just theory; they are Adapters, Facades, and Translators in code.
4. **Strategy:** Sync vs. Async is a business decision disguised as a technical one.

The "Day 6" Takeaway:

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

Page 27: Connecting to the Next Phase (Post-Training)

Recap for Self-Study (Post-Day 6):

- **Read:** *Domain-Driven Design* by Eric Evans, Chapter 14 (Distillation) & 15 (Context Mapping).
- **Practice:** Map your current team's interactions. Identify one "pain point" and assign a DDD pattern to it.
- **Explore:** Download and try **Context Mapper** to model a simple e-commerce system (Sales, Inventory, User).

Looking Forward (Strategic Refactoring):

Now that you have the Map, the next logical step (often Day 7 in extended training or the "Real World" phase) is **Refactoring to the Target Architecture**.

- **The Strangler Fig Pattern:** How to use an OHS to slowly strangle a legacy system.
- **Evolutionary Architecture:** Incrementally applying these patterns without stopping development.

Page 28: Q&A Preparation - The iSAQB Exam View

Mock Questions:

1. **Q:** Which pattern allows a team to isolate their core domain from a volatile external model? **A:** Anticorruption Layer (ACL).

2. **Q:** You have two teams working on the same code base in a tight loop. What is the relationship? **A:** Shared Kernel.

3. **Q:** You want to expose your system capabilities to unknown external developers. Which pattern and protocol do you use? **A:** Open Host Service (OHS) + Published Language (e.g., REST/OpenAPI).

4. **Q:** What is the primary risk of the Separate Ways pattern? **A:** Data duplication and inconsistency.

Closing Note:

Architecture is the art of drawing lines that matter. *Thank you for your dedication to Strategic Design.*

(End of Day 6 Training Material)

Notes 2

iSAQB CPSA-A: Domain-Driven Design

Day 6: Strategic Design 2 - Context Mapping

Context Mapping Patterns & Inter-Context Communication

Volkswagen Group India
For Senior Engineers, Solution Architects & Enterprise Architects

Day 6 Overview: Mastering Context Relationships

By the end of Day 5, you identified bounded contexts and drew your first context maps. Today, we go deeper into the mechanics of how these contexts communicate, integrate, and protect themselves from each other. This is where strategic design becomes operational.

Morning 1: 09:00-10:30	Morning 2: 11:00-12:30	Afternoon 1: 14:00-15:30	Afternoon 2: 16:00-17:30
Customer/Supplier + OHS LG 6-1, LG 6-2	ACL + Shared Kernel LG 6-3, LG 6-4	Separate Ways LG 6-5	Domain Events + Mini-Project LG 6-6

Why Context Mapping Matters in Automotive

In a modern vehicle software platform, 100+ ECUs communicate via various protocols. The BMS talks to Motor Controller. ADAS needs sensor data. Infotainment pulls OTA updates. Without explicit context mapping, implicit dependencies break at the worst times—during safety-critical operations.

VW's MEB platform connects to IONITY, Electrify America, HERE, Google. Each integration requires deliberate choice: conform to their model? translate? expose public API? These are context mapping decisions.

The Nine Context Mapping Patterns

Eric Evans identified patterns describing how contexts relate—from tight coupling to complete independence:

Pattern	Definition	When to Use	Automotive Example
Shared Kernel	Two contexts share common subset of domain model	Teams evolve model together; requires coordination	VIN model: Manufacturing + After-Sales
Customer/Supplier	Upstream serves downstream; downstream influences roadmap	Downstream negotiates requirements	Battery Team → Powertrain Team
Conformist	Downstream adopts upstream model without translation	Upstream won't change; translation cost too high	Internal system adopting SAP model
Anticorruption Layer	Translation layer protects domain from external pollution	Legacy, external, or misaligned models	Translating IONITY OCPP to VW model
Open Host Service	Public protocol for multiple unknown consumers	Multiple downstream contexts need access	Vehicle Data API → App, Fleet, Insurance
Published Language	Industry-standard schema organizations agree on	Cross-organization interoperability needed	OCPP, ISO 15118, AUTOSAR
Separate Ways	No integration; contexts completely independent	Integration cost exceeds benefit	HR system vs Battery Management
Partnership	Teams jointly coordinate interface evolution	Teams succeed or fail together	ADAS + Sensor Fusion teams

LG 6-1: Customer/Supplier Pattern

LEARNING GOAL: Understand Customer/Supplier relationships, designing interfaces serving downstream needs while respecting upstream constraints.

The Dynamic Explained

In Customer/Supplier, one context (Supplier, upstream) provides capabilities another (Customer, downstream) depends on. The defining characteristic: downstream team has INFLUENCE over upstream priorities. The customer can request features, report issues, negotiate timelines.

The metaphor: when you're an important customer, your supplier pays attention because losing you hurts their business. In software: downstream success metrics matter to upstream—shared KPIs, management alignment, or genuine partnership.

✓ Healthy Customer/Supplier	✗ Dysfunctional Relationship
• Regular sync meetings between teams • Downstream files requirements against upstream backlog • Shared acceptance criteria for integration • SLAs for API stability • Clear escalation path	• Upstream ships breaking changes without warning • Downstream requests sit indefinitely • No clear interface ownership • Teams blame each other • Documentation outdated

Case Study: BMS ↔ Powertrain at VW

Battery Management System (BMS) is upstream of Powertrain Control Module (PCM). BMS provides SoC, SoH, temperature, power limits. PCM consumes this to optimize torque, regen braking, thermal management. PCM team CAN request new signals or faster updates—BMS team prioritizes because vehicle performance depends on this integration.

LG 6-2: Open Host Service (OHS)

LEARNING GOAL: Design Open Host Services exposing capabilities through well-defined, stable protocols for multiple consumers.

What Makes OHS Different?

OHS is more than an API—it's commitment to serve multiple, potentially unknown consumers through published, stable interface. You're saying: 'We expose a protocol others can rely on. We maintain backward compatibility and provide migration paths.'

Regular API	Open Host Service	Published Language
Point-to-point Known consumers Can change with	One-to-many Unknown future consumers Versioned,	Industry standard Cross-organization Standards

coordination Internal documentation	backward compatible Public docs, SDKs	body governance Ex: OCPP, ISO 15118

Seven Principles of OHS Design

Principle	Meaning	Automotive Example
1. Stability	Endpoints don't break. Semantic versioning. Deprecate, don't delete.	Vehicle API v2 works after v3 launches. 6-month deprecation.
2. Evolvability	Design for extension. Additive changes over breaking changes.	Adding 'batteryPreconditioning' doesn't break consumers.
3. Documentation	OpenAPI specs, examples, error codes. No tribal knowledge.	Developer portal with sandbox, code samples.
4. Security	OAuth 2.0, scopes, rate limiting. Different access levels.	App: read-only. Fleet: commands. Insurance: trips.
5. Resilience	Graceful degradation. Don't cascade failures.	Battery down? Return cached SoC with 'stale: true'.

LG 6-3: Anticorruption Layer (ACL)

LEARNING GOAL: Design ACLs that translate external models into your domain language, isolating from external changes.

When You Need an ACL

ACL is translation layer between your context and external system with misaligned model. It speaks foreign language externally while translating to your Ubiquitous Language internally. Your domain code NEVER sees external model.

Think embassy translator: understands both cultures, handles protocol differences, ensures your team only deals with concepts they understand.

NEED an ACL When...	Might NOT Need ACL When...
• External uses different concepts • Legacy system with bizarre formats • Third-party API you don't control • System changes frequently • Multiple externals to normalize	• External matches your domain • Customer/Supplier with influence • Integration is temporary • Conformist is acceptable • Published Language both use

ACL Components

Component	Responsibility	Example
Gateway	Technical communication: HTTP, SOAP, retries, timeouts	IonityGateway handles OAuth, connection pooling
Translator	Convert external DTOs to domain objects. Pure mapping.	Maps IONITY evse_id to ChargingPointId
Validator	Check data quality. Reject or repair bad data.	Validates connectors, ensures required fields
Facade	Domain-friendly interface. Hide complexity.	ChargingService.findNearby(loc): ChargingPoint[]

LG 6-4: Shared Kernel

LEARNING GOAL: Identify elements for Shared Kernels, establish governance, understand coupling tradeoffs.

What Belongs in a Shared Kernel?

Shared Kernel is domain model subset contexts share DIRECTLY. Unlike ACL (translates), Shared Kernel creates genuine coupling—same code, definitions, invariants. Intentional when concepts are truly identical.

✓ Good Candidates	✗ Poor Candidates
• Identity: VIN, CustomerId • Value objects: Money, DateRange, GeoLocation • Enumerations: VehicleStatus, OrderState • Domain events: VehicleSold, ServiceCompleted • Validation rules: VIN checksum	• Aggregate roots (too much behavior) • Context-specific business logic • Entities with different lifecycles • UI-specific DTOs • Anything changing at different rates

Governance Essentials

Practice	Implementation
Separate Repository	Kernel in own repo, versioned package (npm/Maven). Consuming contexts declare explicit dependencies.
Semantic Versioning	Follow semver. Breaking = major bump. Consumers pin versions, upgrade on schedule.

Change Process	Non-trivial changes need RFC. All teams review. 'Silence ≠ approval'—explicit sign-off.
Minimal Surface	Only truly shared concepts. If one context uses it, doesn't belong. Resist 'convenience' additions.

LG 6-5: Separate Ways

LEARNING GOAL: Recognize when integration creates more problems than it solves; make deliberate non-integration decisions.

The Case for Non-Integration

Not every context pair should integrate. Sometimes integration cost exceeds value. Separate Ways is DELIBERATE choice to keep contexts independent—accept duplication or manual processes for autonomy.

This isn't failure—it's wisdom about where integration adds value. Eric Evans: 'declaring formal independence between contexts.' Critical in large enterprises where 'connect everything' temptation leads to integration spaghetti.

Decision Framework

Factor	Shared Kernel	ACL	Conformist	Separate Ways
Model Alignment	Identical concepts	Different but translatable	Their model OK	No overlap
Team Relationship	Close collaboration	Separate teams	No influence	Different orgs
Change Frequency	Stable, coordinated	External changes	Upstream stable	Independent
Integration Cost	Worth sharing	Worth translating	Not worth it	Exceeds benefit

Example: HR System vs Battery Management. Both deal with 'people' (HR: employees, BMS: who certified battery). But fundamentally different domains. BMS needs badge ID string for audits—doesn't care about salary, benefits. Integration cost >> value. SEPARATE WAYS: BMS stores badge ID as string, no HR link.

LG 6-6: Domain Events

LEARNING GOAL: Design domain events enabling async, loosely-coupled inter-context communication.

Events: Decoupling Through Facts

Domain Event = something meaningful that happened—a fact about the past. Unlike commands (tell what to do), events let contexts REACT without tight coupling. Publisher doesn't know consumers. Consumer doesn't call publisher. Independent deployment, scaling, evolution.

Event	Publisher	Subscribers
VehicleProduced	Manufacturing	Logistics (transport), Quality (inspection), Sales (available)
VehicleSold	Sales	Finance (receivable), Registration, Connected Car (provision)
ChargingSessionCompleted	Charging	Billing (invoice), Analytics, Mobile App (notification)
ServiceCompleted	After-Sales	Warranty (update), CRM (follow-up), Inventory (parts)
BatteryDegraded	Battery Management	Service (schedule), Warranty (assess), App (notify owner)

Event Naming: Past Tense, Domain Language

VehicleUpdate ✗ (vague) → VehicleLocationUpdated ✓. CreateInvoice ✗ (command) → OrderPlaced ✓ (fact triggering invoice). Data ✗ (technical) → ChargingSessionStarted ✓ (domain language).

Event Structure & Delivery Guarantees

Essential Event Fields

Event Schema	Field Explanations

{"eventId": "evt_abc", "eventType": "ChargingSessionCompleted", "eventVersion": "2.0", "occurredAt": "2026-01-09T14:30:00Z", "source": "charging-service", "correlationId": "sess_xyz", "payload": {"vin": "WVW...", "kwh": 67.5}}	eventId: Unique, idempotency key eventType: What happened (routing) eventVersion: Schema evolution occurredAt: When it happened source: Publishing service correlationId: Trace across services payload: Business data for subscribers

Delivery Guarantees

Guarantee	Meaning	When to Use
At-Most-Once	Delivered 0 or 1 times. May be lost.	Metrics, logging, non-critical notifications.
At-Least-Once	Delivered 1+ times. May duplicate.	Most business events. Handlers MUST be IDEMPOTENT.
Exactly-Once	Delivered exactly 1 time. Requires dedup.	Financial. Often use at-least-once + idempotency.

CRITICAL: With at-least-once (most common), handlers WILL receive duplicates. Design to produce same result whether called once or many times. Store processed event IDs. Check before processing. Use DB transactions to atomically record 'processed' and apply changes.

Competitor Case Studies

Tesla: Vertical Integration

Minimal External Dependencies	Tesla owns entire stack: vehicles, Supercharger, app, energy. Fewer ACLs needed—control both sides.
OHS: Supercharger Network	After opening to non-Tesla EVs, created OHS with NACS standard. Now adopted by Ford, GM, Rivian.
Customer/Supplier: Fleet API	Fleet API serves enterprise (Hertz, rentals). Classic C/S where Tesla responds to fleet needs.

BMW: Multi-Vendor Integration

ACL for Charging Networks	Integrates IONITY, ChargePoint, EVgo. Each needs ACL. Digital Charging Service normalizes all.
Shared Kernel: HERE Maps	BMW + Mercedes + Audi co-own HERE. Navigation data = shared kernel with ownership governance.

Partnership: Mobileye	ADAS dev with Mobileye = true Partnership. Both coordinate on perception interface. Neither alone.

Mercedes-Benz: MB.OS

Separate Ways: CarPlay	Mercedes chose NOT to integrate Apple CarPlay in new EQ (2024). Deliberate—accepted duplication for UX control.
Published Language: AUTOSAR	Founding AUTOSAR member. ECUs use this standard—Bosch, Continental, ZF all deliver compliant components.

Hands-On Exercises

Exercise 1: Identify Patterns (30 min)

VW's connected car integrates with: 1. Google Maps API (don't control, changes frequently) 2. CARIAD battery analytics (sister company) 3. German vehicle registration (government, fixed format) 4. Internal marketing CRM (different dept, minimal overlap) 5. Electrify America (VW subsidiary) Format: [System] → [Pattern] because [2-3 sentences]

Exercise 2: Design ACL (45 min)

Legacy DMS returns:
{"VEH_NO":"ABC123","SVC_DT":"20251215","SVC_TYP":"A","MLG":"45000","COST_EUR":"34500"}
SVC_TYP: A=Annual, B=Brake, O=Oil. COST in cents. Tasks: 1) Design ServiceRecord domain model 2) Write Translator 3) List 3 validation rules 4) Handle missing TECH_ID

Exercise 3: Event Design (30 min)

Design events for 'Vehicle Sold' process: 1) List all reacting contexts 2) Design VehicleSold schema (fields, types, examples) 3) Each subscriber's action 4) Delivery guarantee choice + why 5) How Billing ensures idempotency

Mini-Project: EV Charging Ecosystem (90 min)

Design VW's EV charging platform context map covering: • Public charging (IONITY, Electrify America, ChargePoint) • Home charging via VW wallboxes • Billing/payments (cards, PayPal, VW Pay) • In-vehicle route planning with charging stops • Mobile app (We Connect) • Fleet management

Deliverables

1. Context Map	Diagram: all bounded contexts with relationship pattern labels (OHS, ACL, etc.)
2. Justification	Each relationship: why this pattern? (2-3 sentences)
3. ACL Design	Pick one external integration. Design complete ACL with Gateway, Translator, Validator, Facade.
4. Event Catalog	5 key events flowing between contexts. Complete schemas.
5. Shared Kernel	What to share (ChargingPointId, Money?). Propose governance model.

Tools & Platforms

Context Mapping

Tool	Purpose	Access
Context Mapper	DSL for context maps. Generates diagrams, contracts.	contextmapper.org (Free)
Miro	Collaborative whiteboard. Workshop-style mapping.	miro.com (Free tier)
draw.io	Diagramming with DDD shape libraries.	app.diagrams.net (Free)
Structurizr	Architecture as code. C4 model diagrams.	structurizr.com (Free tier)

Event-Driven Platforms

Platform	What You Learn	Access
Apache Kafka	Event streaming, pub/sub, partitioning.	kafka.apache.org (Docker)
Confluent Cloud	Managed Kafka + schema registry.	confluent.io ($400 credit)
RabbitMQ	Message broker, exchange patterns.	rabbitmq.com (Docker)
Pact	Consumer-driven contract testing.	pact.io (Free, open source)

Key Terms Glossary

Term	Definition
Context Map	Strategic tool visualizing relationships between bounded contexts—technical patterns and organizational dynamics.
Upstream/Downstream	Dependency direction. Upstream provides; downstream consumes. Upstream changes affect downstream.
Customer/Supplier	Downstream (customer) influences upstream (supplier) priorities. Supplier responds to needs.
Open Host Service	Well-defined, stable API for multiple consumers. Commits to backward compatibility.
Published Language	Industry-standard schema organizations agree on (OCPP, ISO 15118, AUTOSAR).
Anticorruption Layer	Translation layer protecting context from external model pollution. Converts to domain language.
Shared Kernel	Model subset shared directly between contexts. Creates coupling, eliminates duplication.
Separate Ways	Deliberate non-integration. Contexts independent. Accept duplication over coordination cost.
Domain Event	Record of what happened. Async, loosely-coupled inter-context communication.
Idempotency	Same result whether executed once or many times. Critical for at-least-once delivery.

Learning Goals Checklist

Before completing Day 6, ensure you can answer:

LG 6-1	Customer/Supplier: □ Difference from Conformist? □ Organizational signals? □ Interface contract design?
LG 6-2	OHS: □ Seven principles? □ Multi-consumer API design? □ Versioning strategies?
LG 6-3	ACL: □ When needed vs other patterns? □ Component design? □ Write translator code?
LG 6-4	Shared Kernel: □ Good vs poor candidates? □ Governance model? □ Coupling tradeoffs?
LG 6-5	Separate Ways: □ When integration hurts? □ Document non-integration? □ Warning signs?
LG 6-6	Domain Events: □ Naming, structure, versioning? □ Delivery guarantees? □ Idempotent handlers?

Training Recap & Self-Study

6-Day Summary

Day	Topic	Key Takeaway
Day 1	DDD Foundations: Domains, Models, UL	Models are purposeful; language alignment crucial
Day 2	Tactical: Entities, VOs, Aggregates	Aggregates = consistency boundaries
Day 3	Tactical: Services, Repos, Factories	Services orchestrate; repos abstract persistence
Day 4	Architecture: Layered, Hexagonal, CQRS	Architecture serves domain; ports/adapters
Day 5	Strategic 1: Bounded Contexts, Subdomains	Contexts = linguistic boundaries; core gets investment
Day 6	Strategic 2: Context Mapping	Integration is strategic; patterns encode relationships

Self-Study Resources

Books	'Domain-Driven Design' (Evans), 'Implementing DDD' (Vernon), 'Learning DDD' (Khononov)
Online	DDD Europe talks (YouTube), Udi Dahan's ADSD, Martin Fowler's bliki
Community	DDD Discord, Virtual DDD meetups (virtualddd.com), DDD Europe conference
Practice	Apply to current project. Start with context map of existing system. Run EventStorming.

Day 6 Complete

Strategic Design 2: Context Mapping Patterns

Key Message

"Context mapping is not just drawing diagrams—it's making conscious decisions about where to invest in integration and where to accept costs of independence."

— Adapted from Eric Evans