

KANDDDINSKY 2026 · HANDS-ON WORKSHOP

FROM MONOLITH TO EVENTS

A practical guide to DDD refactoring

Alessandro Colla · Alberto Acerbis



SCAN FOR THE CODE

github.com/BrewUp/KanDDDinsky-2026



WHAT IS LEGACY?

Legacy is not when it is old, but when no one dares to change it anymore.



OUR LEARNING SYSTEM

Meet BrewUp

An ERP for a brewery. Simple on purpose, real principles.

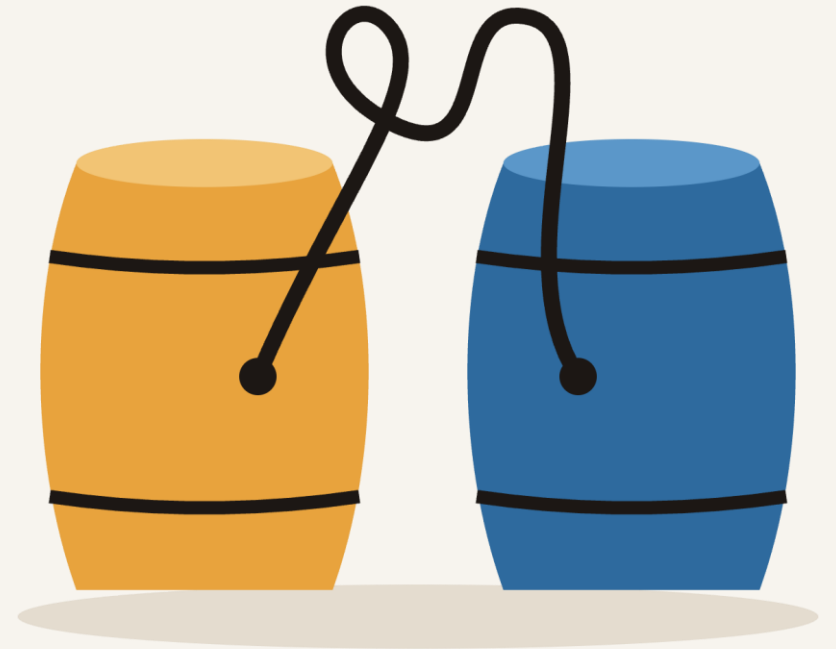
SALES

Creates sales orders

WAREHOUSE

Manages beer availability

Today: one flow. Create a sales order.



A SNEAK PEEK

The monolith, under test

```
$ dotnet test BrewUp.Architecture.Tests
```

```
FAILED Should_SalesArchitecture_BeCompliant
```

```
FAILED Should_WarehousesArchitecture_BeCompliant
```

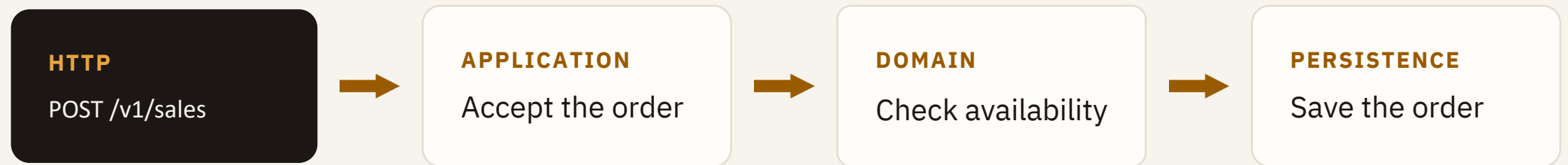
```
FAILED Should_BrewUpArchitecture_BeCompliant
```

```
Total: 3 · Passed: 0 · Failed: 3
```

What are these? You'll find out in an hour.

SAFETY NET

Follow one business flow



One vertical slice. Not the whole system.

EXERCISE • PAIRS • 5 MIN

Before touching the code...

Talk to the person next to you. One idea per post-it, then stick it on the board.

**What is the first step
you would take?**

THE ONE RULE OF REFACTORING

Change the structure. Keep the behaviour.

WE DO

Move code, introduce boundaries, untangle dependencies

WE DO NOT

Fix bugs, add improvements or new business rules

FOUND A BUG?

Write it down. Fix it in a separate step.

THE SAFETY NET

End-to-end first: same payload, same outcome

```
[Fact]
public async Task Can_Create_SalesOrder()
{
    // Arrange: a known order with one row
    SalesOrderJson body = new(...);
    var postResult = await integrationFixture.Client
        .PostAsync("/v1/sales", httpContent);
    Assert.Equal(HttpStatusCode.Created,
        postResult.StatusCode);
}
```

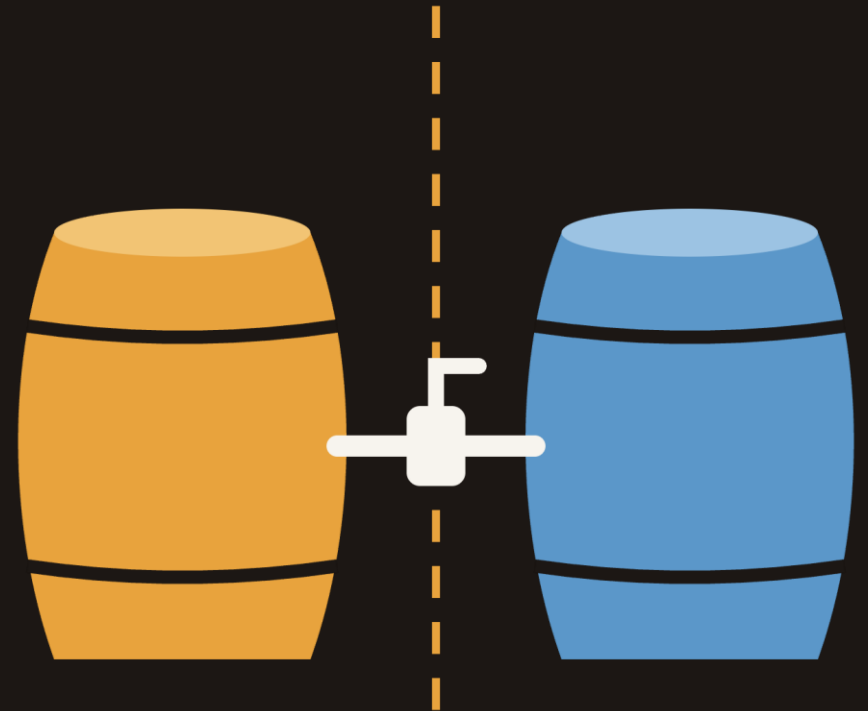
- ✓ Run it before you change anything: a green baseline.
- 🔍 201 Created is not the whole story. Was the order saved?
- 🔒 Test through the public boundary, so it survives the refactor.

03

SALES & WAREHOUSE

Where does Sales end and Warehouse begin?

THREE ROUNDS OF POST-ITS



SHOW ME THE CODE

How an order is created today

```
public async Task CreateSalesOrderAsync(..., rows, ...)
{
    foreach (var row in rows)
    {
        var availability = await warehouseRepository
            .GetByIdAsync<Shared.Entities.Availability>(row.BeerId, ct);
        if (availability != null) beersAvailable.Add(row);
    }
    var aggregate = SalesOrder.CreateSalesOrder(..., beersAvailable);
    await saleRepository.InsertAsync(aggregate.MapToSharedDto(), ct);
}
```

ROUND 1 · 5 MIN

Spot the problem

Look at the code you just saw. One problem per post-it.

**What is the problem
between Sales and
Warehouse?**

ROUND 2 · 10 MIN

We found the coupling. Why do we separate them?

**What does "separate"
actually mean?**

**Where does the
responsibility of
Sales end?**

**What does Sales
really need to know?**

TWO CANDIDATE BOUNDED CONTEXTS

Keep the collaboration. Remove the leak.

SALES

Owns the order

Its meaning, its lifecycle, the decision to create it.

WAREHOUSE

Owns availability

What "available" means, how it is stored, who can change it.



Sales may **ask a question**. It must not **know the internals**: entities, repositories, tables.

ROUND 3 • 10 MIN

Separated. Now, how do they talk?

**Who is upstream?
Who is downstream?**

**Who adapts to whose
model?**

**Which pattern would
you pick?**

STRATEGIC PATTERNS

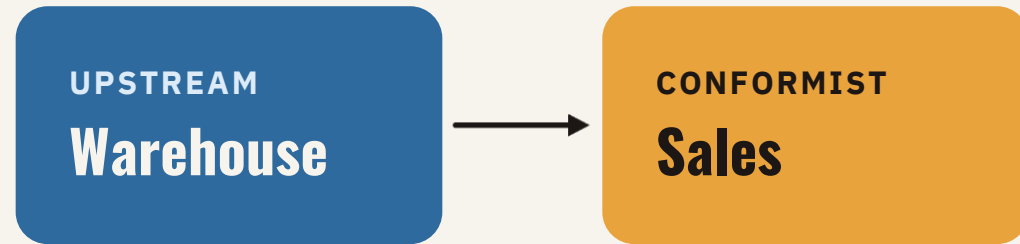
How bounded contexts relate

Pattern	In one line
Shared kernel	Two contexts share a small part of the model, owned together
Customer–supplier	Upstream serves, downstream depends; they negotiate
Conformist	Downstream adopts the upstream model as it is
Anticorruption layer	Downstream translates to protect its own model
Open host service	Upstream exposes a well-defined API for everyone
Published language	A shared, documented schema for the exchange
Separate ways	No integration at all: each context evolves alone

💡 Which one do you choose for Sales and Warehouse? Why?

OUR PICK

Start conformist. Evolve later.



- 🔍 You don't yet know how the two contexts will behave.
- 👤 They may belong to two different teams.
- 📅 Cheapest start. Move to ACL or OHS when the model starts to hurt.

You just did **context mapping**.

- ✓ Found the coupling
- ✓ Drew the boundary
- ✓ Chose the relationship

04

We split them. How do we cover our backs?

5 min: how do we stop someone from bringing the dependency back tomorrow?



FITNESS FUNCTIONS

Two safety nets, two different questions

END-TO-END TESTS

Did the behaviour change?

Same payload, same order, same HTTP response.

FITNESS FUNCTIONS

Did the dependency come back?

Sales must not depend on Warehouse internals.
Ever.

Fitness function: *"an objective integrity assessment of some architecture characteristic"* (Ford, Parsons, Kua).

Tools: NetArchTest · ArchUnit.

SHOW ME THE CODE

Sales must not know Warehouse

```
[Fact]
public void Should_SalesArchitecture_BeCompliant()
{
    var types = Types.InAssembly(typeof(SalesFacade).Assembly);
    var forbidden = new[] {
        "BrewUp.Warehouses.Domain", "BrewUp.Warehouses.Infrastructures",
        "BrewUp.Warehouses.ReadModel", "BrewUp.Warehouses.SharedKernel" };
    var result = types.ShouldNot().HaveDependencyOnAny(forbidden)
        .GetResult().IsSuccessful;
    Assert.True(result);
}
```

REMEMBER THE RED TESTS?

You just wrote fitness functions.

```
$ dotnet test BrewUp.Architecture.Tests
```

```
PASSED Should_SalesArchitecture_BeCompliant
```

```
PASSED Should_WarehousesArchitecture_BeCompliant
```

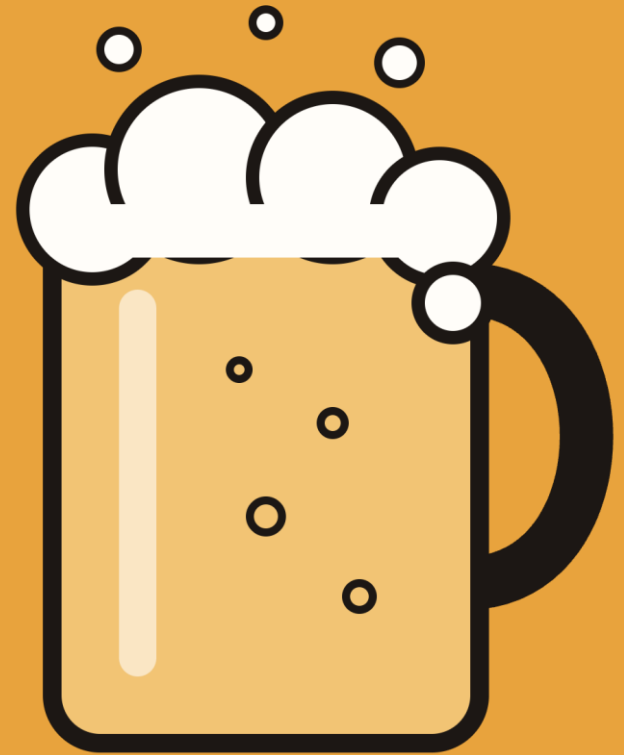
```
PASSED Should_BrewUpArchitecture_BeCompliant
```

```
Total: 3 · Passed: 3 · Failed: 0
```

Same tests as the teaser. Now they guard your boundary in CI.

BREAK

Back in ten for the mediator.



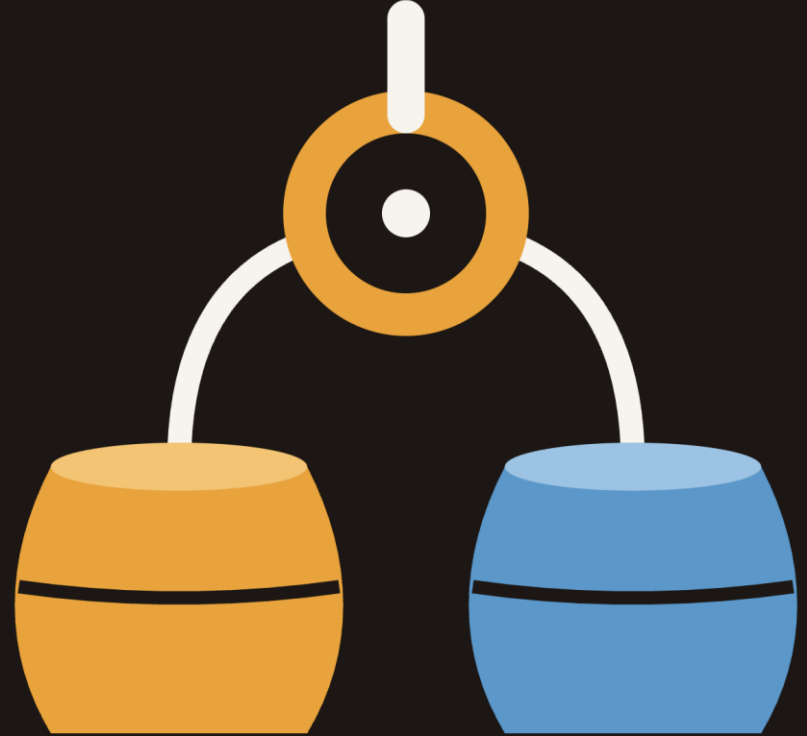
05

MEDIATOR

What is it for?

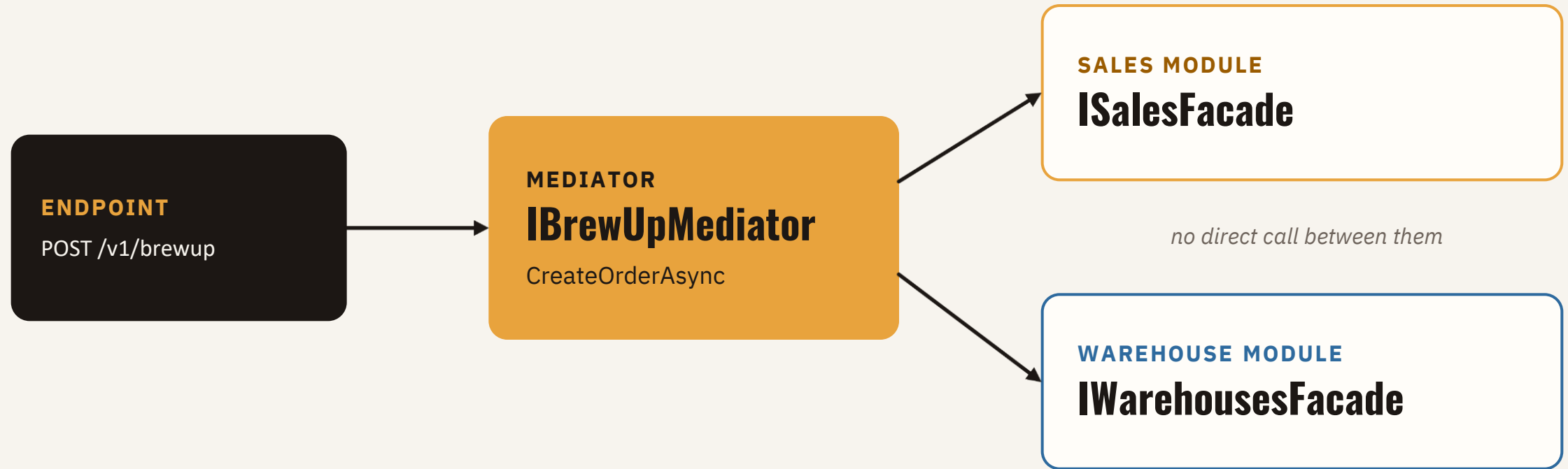
What do I gain?

Where are its limits?



THE SHAPE OF IT

One orchestrator, two isolated modules



What the mediator gives, what it costs

Gains

- Modules no longer know each other
- One place that tells the business flow
- Each module testable on its own

Limits

- Still synchronous: both modules must answer
- More code: facades, contracts, wiring
- It can grow into a god object

SHOW ME THE CODE

The mediator creates the order

```
public class BrewUpMediator(ISalesFacade salesFacade, IWarehousesFacade warehouseFacade) : IBrewUpMediator
{
    public async Task<string> CreateOrderAsync(SalesOrderJson body, ...)
    {
        foreach (var row in body.Rows)
            availability = await warehouseFacade.GetAvailabilityAsync(row.BeerId, ...);
        // keep only the rows that are available
        return await salesFacade.CreateOrderAsync(body, ...);
    }
}
```

06

EVENTS

What is it for?

What do I gain?

What complexity comes with it?



MESSAGES CARRY INTENT

Commands ask. Events tell.

COMMAND · IMPERATIVE

CreateSalesOrder

- A request: it can be rejected
- One handler is responsible
- Producer → consumer

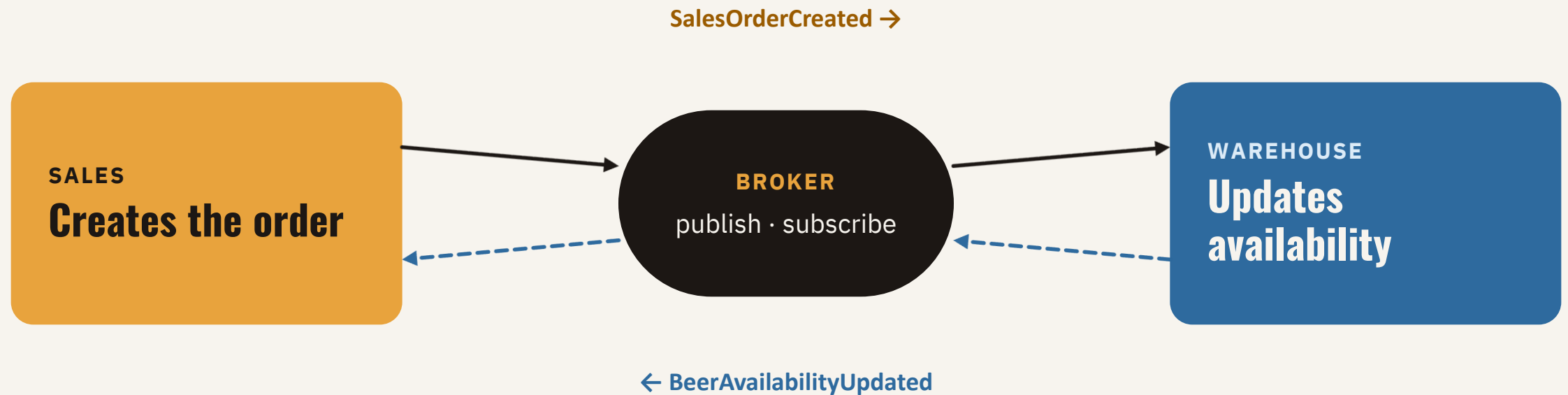
EVENT · PAST TENSE

SalesOrderCreated

- A fact: it already happened
- Anyone interested may react
- Publish → subscribe

THE NEW FLOW

Sales stops asking. It announces.



No one waits for anyone. Consistency arrives a little later.

What events give, what they cost

Gains

- Decoupled in code and in time
- New listeners without touching Sales
- The business story is in the event log

Complexity

- Eventual consistency
- A broker to run, retries, idempotency
- Harder to trace and to test

SHOW ME THE TESTS

Given events, when a command, expect events

```
class CreateSalesOrderSuccessfully
  : CommandSpecification<CreateSalesOrder>
{
  Given() => no previous events;
  When() => new CreateSalesOrder(...);
  OnHandler() => new CreateSalesOrderCommandHandler(...);
  Expect() => new SalesOrderCreated(...);
}
```

Tests the whole lifecycle of the aggregate, not one method.

Reads like the business: domain experts can follow it.

Muflone.SpecificationTests

WHAT WE DID TODAY

From monolith to events, one safe step at a time

01

Safety net

02

Context map

03

Fitness functions

04

Mediator

05

Events

Every step kept the same behaviour. That is what made the next one possible.

DISCLAIMER, AGAIN

Every solution has a price

Step	You gain	You pay
Context map	Clear ownership	Conversations and agreements
Fitness functions	Boundaries that hold	Rules to maintain
Mediator	Decoupled modules	Verbosity, sync coupling
Events	Autonomy in time	Eventual consistency, infra
CQRS	Models fit for purpose	Two models to keep in sync

Don't pair up with the first CQRS that comes along.

“

**Software architecture is abstract by nature.
We must ground it with some implementation
details to make it concrete.**

KEEP READING

Bibliography

Domain-Driven Refactoring · Alessandro Colla, Alberto Acerbis

Implementing Domain-Driven Design · Vaughn Vernon

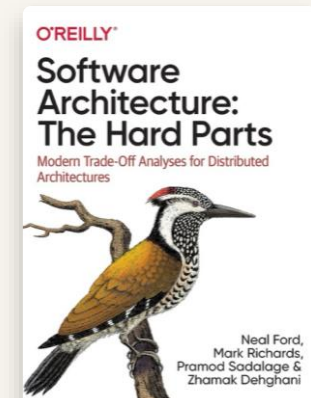
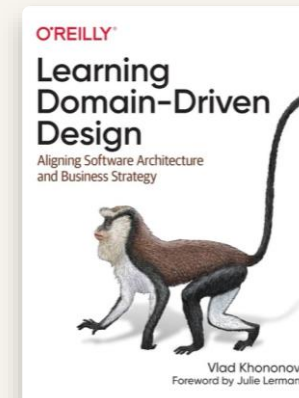
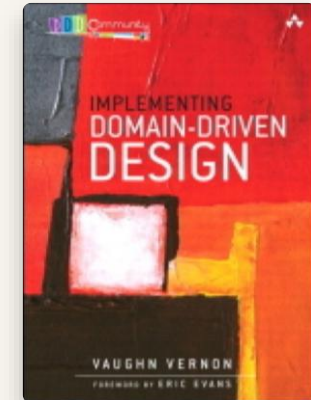
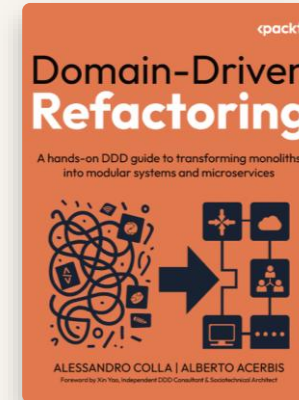
Learning Domain-Driven Design · Vlad Khononov

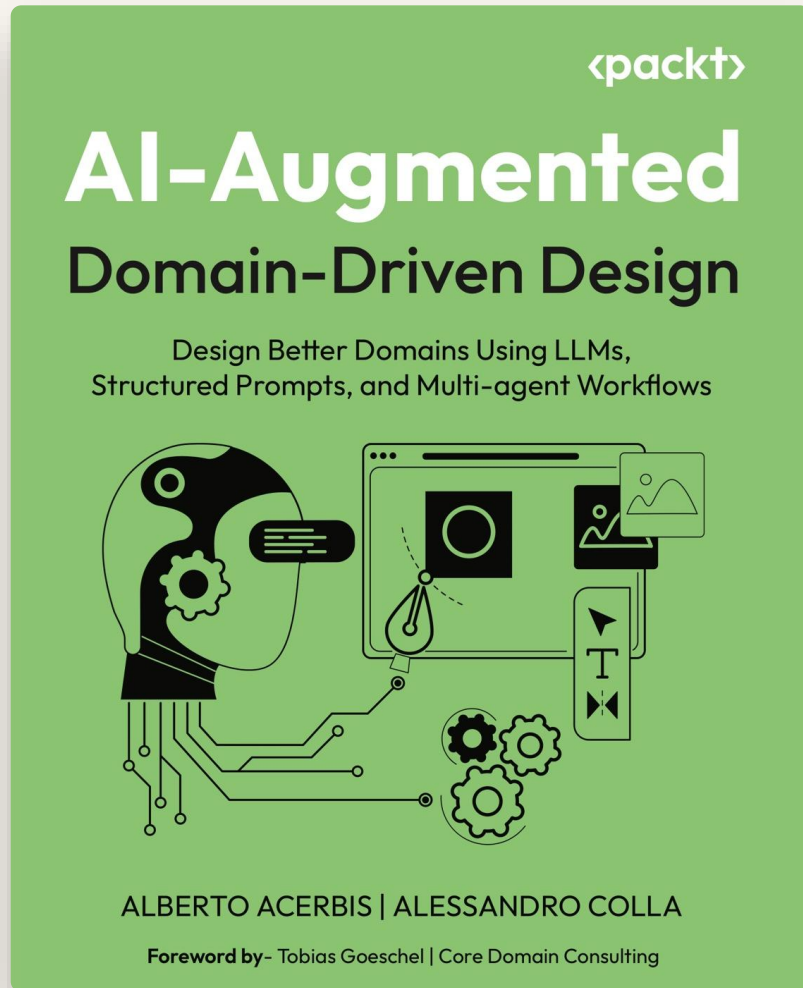
Software Architecture: The Hard Parts · Ford, Richards, Sadalage, Dehghani

Building Evolutionary Architectures · Ford, Parsons, Kua

The Software Architect Elevator · Gregor Hohpe

Hands-On Domain-Driven Design with .NET Core · Alexey Zimarev





OUR NEW BOOK • PACKT

AI-Augmented Domain-Driven Design

Design better domains using LLMs, structured prompts and multi-agent workflows.

Use **KANDDD20** for **20%** off during the conference days

THANK YOU!

We hope you learned something
while having fun.

Questions? Prost!



SCAN FOR THE CODE

github.com/BrewUp/KanDDDinsky-2026

2026

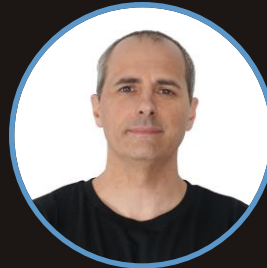
Branches 02+ after the workshop



Alberto Acerbis

 linkedin.com/in/aacerbis

 alberto.acerbis@intre.it



Alessandro Colla

 linkedin.com/in/alessandrocolla

 alessandro.colla@evoluzione.agency



BACKUP · WATCH OUT

Availability rules? That's not refactoring.

Today Sales only checks that an availability *exists*. Checking the **quantity** would change what the system decides.

New business rule = new requirement, new test, separate step.

Events make CQRS almost free

