Modul 05 - Strategic Design

Bounded Contexts & Context Mapping

Lernziele

- Bounded Contexts definieren und abgrenzen können
- Den Unterschied zwischen Subdomain und Bounded Context verstehen
- Alle Context-Map-Patterns kennen und anwenden
- Conway's Law und seine Auswirkungen auf BC-Schnitte verstehen
- Strategic Design auf das Immobilien-CRM anwenden

Was ist ein Bounded Context?

Ein Bounded Context ist ein explizit **abgegrenzter Bereich, in dem ein bestimmtes Modell gilt**. Er beschreibt den **Lösungsraum** einer Sub-Domäne.

Innerhalb eines Contexts haben Begriffe eine eindeutige Bedeutung. Außerhalb kann derselbe Begriff etwas **völlig anderes bedeuten**.

Vom Event Storming zum Bounded Context

Drei Signale für eine Context-Grenze

1. Sprachliche Grenze - gleiche Begriffe, andere Bedeutung

> "Immobilie" in der Objektverwaltung ≠ "Immobilie" in der Vermarktung

2. Grenz-Events ("Pivots") - Events, die eine neue Phase einleiten

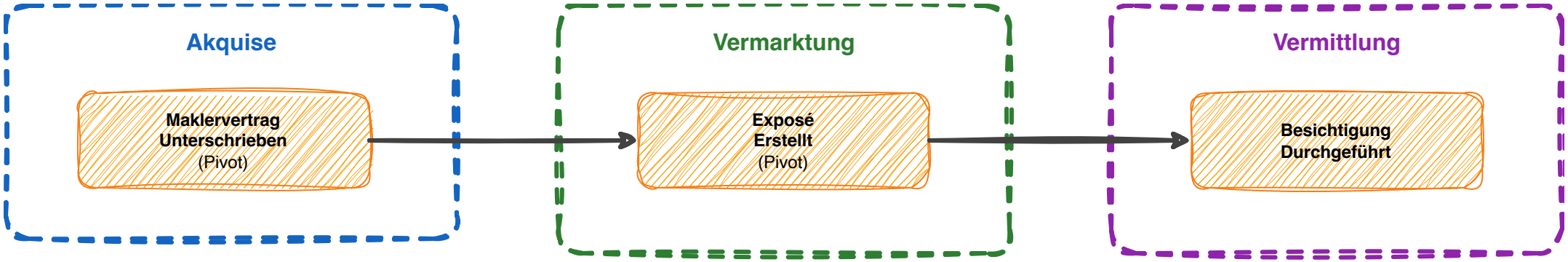
> `MaklervertragUnterschrieben` → Grenze zwischen Akquise und Vermarktung
> `AngebotAngenommen` → Grenze zwischen Vermittlung und Abschluss

3. Akteurwechsel - andere Person übernimmt

> Makler (Akquise) → Marketing-Team (Vermarktung)

Vom Event Storming zu Bounded Contexts

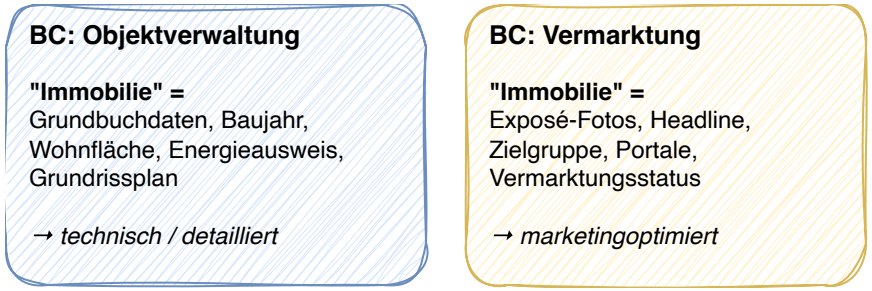
Schnittstellen erkennen



Event verbindet die Kontexte → wird zur Schnittstelle

Beispiel: Der Begriff "Immobilie"

"Immobilie" bedeutet in der Verwaltung etwas anderes als in der Vermarktung!



> Eric Evans: *"A Bounded Context delimits the applicability of a particular model."*

Bounded Context vs. Subdomain

	Subdomain	Bounded Context
Raum	Problemraum	Lösungsraum
Was?	Fachlicher Bereich	Softwaregrenze
Entdeckung	Wird entdeckt / analysiert	Wird bewusst geschnitten
Existenz	Existiert unabhängig von Software	Ist ein Architektur-Artefakt

Idealerweise existiert für jede Subdomäne genau ein Context. Aber in der Praxis zwingen uns Legacy-Systeme manchmal zu Abweichungen.

Ein Context sollte aber nie mehrere Subdomains abdecken (→ Big Ball of Mud).

Conway's Law

Die Organisationsstruktur bestimmt die Softwarearchitektur

> *"Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure."*
> Melvin Conway, 1968

Konsequenz für Context-Schnitte

Ein Context sollte von einem Team verantwortet werden, aber nicht von mehreren. Nur bei kleinen Contexten sollte ein Team mehrere verantworten!

Organisiere Teams entlang der gewünschten Architektur, nicht umgekehrt.

Context Map - Überblick

Eine Context Map zeigt, wie Bounded Contexts zueinander stehen.
Sie dokumentiert Integrations-Beziehungen und Machtverhältnisse

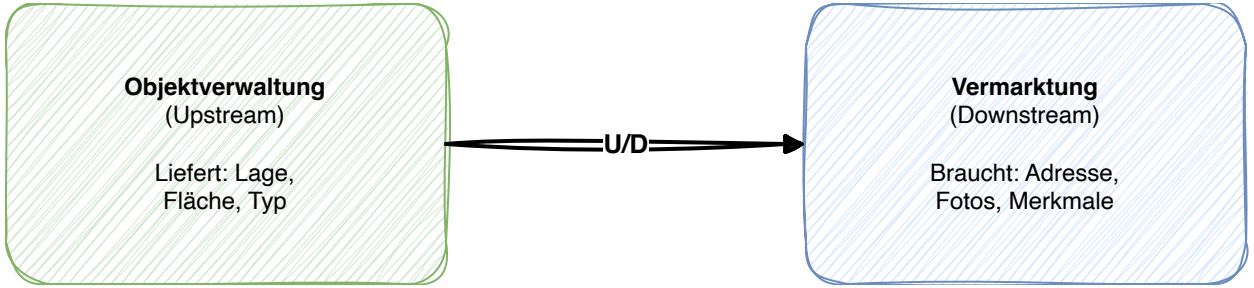
Es geht dabei auch um Team- und Systembeziehungen, nicht nur um Technik.

Die 8 Patterns

Pattern	Kurzbeschreibung
Customer / Supplier	Downstream stellt Anforderungen an Upstream
Conformist	Downstream übernimmt Upstream-Modell 1:1
Anti-Corruption Layer	Downstream schützt sich mit Übersetzungsschicht
Shared Kernel	Geteilter Modellkern, gemeinsame Verantwortung
Published Language	Dokumentiertes, versioniertes Datenformat
Open Host Service	Offene API für viele Consumer
Partnership	Zwei Teams entwickeln gemeinsam, keine U/D-Hierarchie
Separate Ways	Bewusst keine Integration

Pattern: Customer / Supplier (U/D)

Downstream stellt Anforderungen an Upstream



- Upstream liefert Daten oder Services
- Downstream konsumiert und darf Anforderungen stellen
- Beide Teams stimmen sich aktiv ab

Pattern: Conformist

In diesem Pattern übernimmt der Downstream die Konzepte des Upstreams ohne Einfluss.

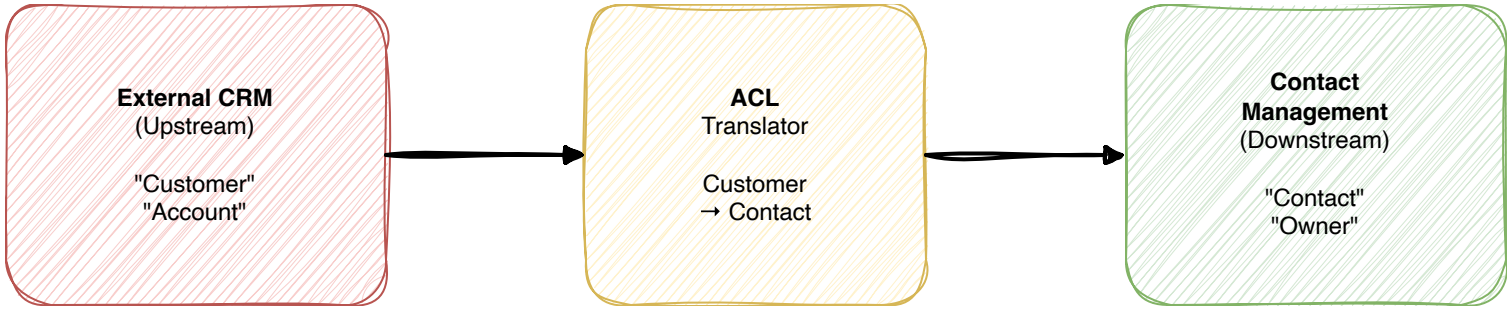
Man setzt dieses Pattern ein, wenn externe Systeme eingebunden werden müssen und wir an ihnen nichts ändern können.

Beispiel: Übernahme des OpenImmo-XML-Standards

> Risiko: Das eigene Modell wird vom Upstream-Modell "infiziert".
> Alternative: ACL, wenn der Aufwand vertretbar ist.

Pattern: Anti-Corruption Layer (ACL)

Schützt das eigene Modell mit einer Übersetzungsschicht



- Übersetzt eingehende Daten in die eigene Ubiquitous Language
- Wird eingesetzt bei der Integration mit Legacy-Systemen oder externen APIs deren Modell nicht zum eigenen passt

Beispiel für einen Anti-Corruption-Layer

```
@Component
public class ExternalCrmTranslator {
    public Contact translate(CrmCustomerDto dto) {
        return new Contact(
            ContactId.generate(),
            dto.getFirstName(), dto.getLastName(),
            ContactType.from(dto.getType()));
    }
}
```

Pattern: Shared Kernel

Geteilter Modellkern zwischen zwei BCs

- Zwei BCs teilen sich einen gemeinsamen Modellteil
- Änderungen müssen abgestimmt werden - enger Kopplungsgrad
- Nur bei engem Team-Alignment sinnvoll

Wann einsetzen?

- Gemeinsame Kernkonzepte, die identisch bleiben müssen
- Beispiel: Gemeinsame Value Objects `Address`, `MonetaryAmount`

> Vorsicht: Shared Kernel ist die engste Kopplung zwischen BCs.
> Je größer der Kernel, desto mehr Abstimmungsaufwand.
> Alternative: Published Language oder ACL.

Pattern: Published Language & Open Host Service

Ein **Published Language** ist ein dokumentiertes, versioniertes Datenformat für Kommunikation. Beispiele: JSON-Schemas, XML-Schemas, Protobuf, Avro.

Ein **Open Host Service (OHS)** stellt eine API für einen Context bereit. Mehrere Contexts greifen dann auf ihn zu. Dies wird oft mit der Published Language kombiniert.

Beispiel:

Ein Stammdaten-Context bietet eine REST-API (OHS) mit versioniertem JSON-Schema (Published Language), die von mehreren anderen BCs genutzt wird

Pattern: Partnership & Separate Ways

Partnership bedeutet, dass zwei Teams gleichberechtigt und in enger Koordination ihre Contexte entwickeln. Dies ergibt Sinn, wenn die Anforderungen eng aneinander gebunden sind.

Umgekehrt bedeutet **Separate Ways**, dass bewusst keine sofortige Integration gewünscht ist.

Wie manifestiert sich ein BC im Code?

Vorgeschmack auf Modul 08 (Paketstruktur)

```
de.realestate/
├── brokerage/              ← BC: Brokerage
│   ├── domain/
│   ├── application/
│   ├── infrastructure/
│   └── adapter/
├── acquisition/            ← BC: Acquisition
│   ├── domain/
│   ├── application/
│   ├── infrastructure/
│   └── adapter/
└── contact/                ← BC: Contact Management
    ├── domain/
    └── ...
```

- Jeder Context ist ein Top-Level-Package (oder Maven-Modul)
- Contexts kommunizieren nur über definierte Schnittstellen (Events, APIs)
- Kein direkter Import von `brokerage.domain` in `acquisition.domain`!

Wie schneidet man Bounded Contexts?

Fünf Heuristiken:

1. Ubiquitous Language - Wo ändert sich die Bedeutung eines Begriffs?
2. Pivot Events - Welche Events markieren Phasenübergänge?
3. Akteurwechsel - Wo übernimmt eine andere Person/Rolle?
4. Daten-Kohäsion - Welche Daten ändern sich gemeinsam?
5. Conway's Law - Welches Team verantwortet welchen Bereich?

Hands-on: Lab 03

Context Map für das Immobilien-CRM erstellen