CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Domain Driven Design

(4) Taktisches Design

Pattern für die Business Logic

PROF. DR.
NANE KRATZKE

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

KAPITEL 14

Domain Driven Design

14.1 Fachlichkeit, Fachlichkeit, Fachlichkeit

14.2 Strategisches Design

- Subdomänen
- Ubiquitous Language
- Bounded Context
- Context Mapping

14.3 Taktisches Design

- Oft genutzte Pattern für Geschäftslogik
  (ETL, Active-Record, Domain Model, Event-Sourcing)
- Oft genutzte Architektur-Pattern
  (Layered Architecture, Ports & Adaptor, CQRS)

14.4 Zusammenfassung

INHALTE

Domain Driven Design

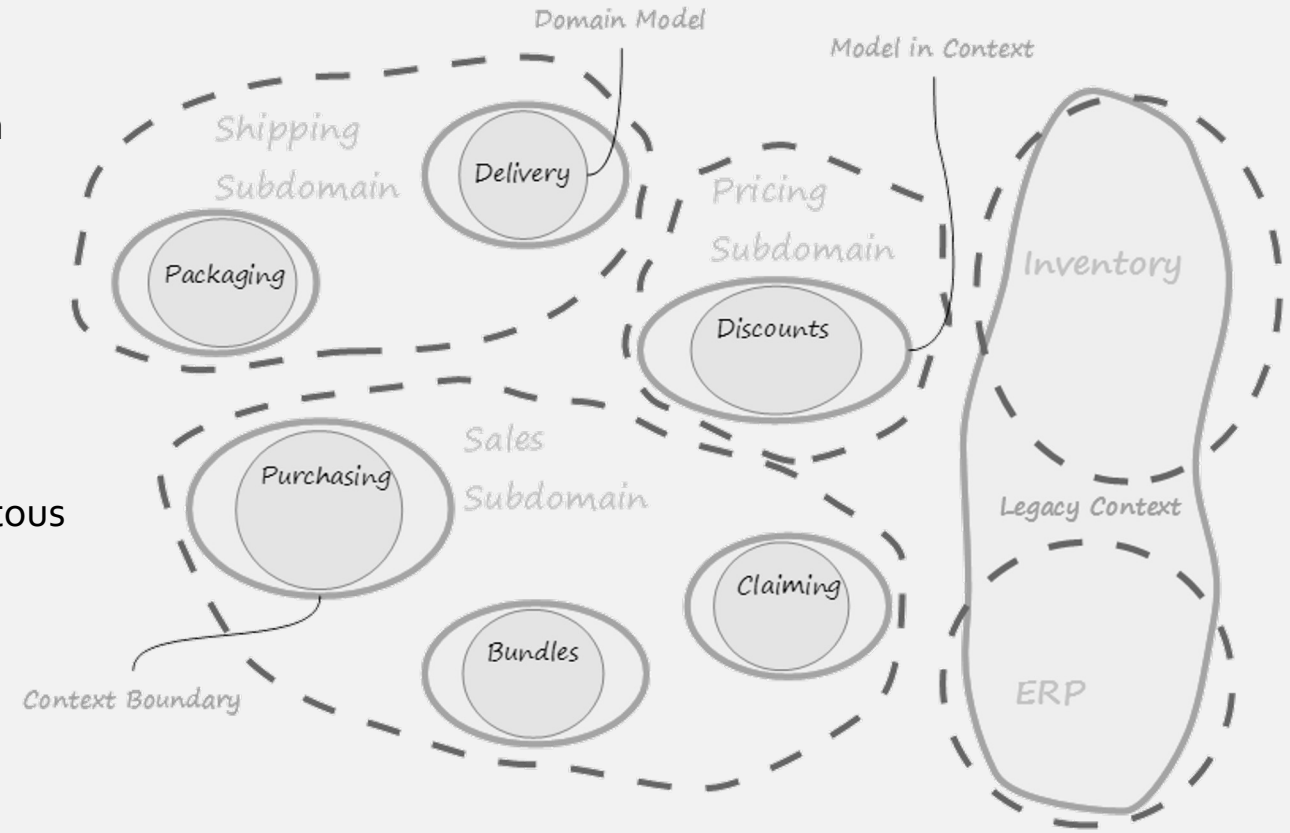
- Was ist das?
- Effektives Software Design
- Strategisches Design
- Taktisches Design

Strategisches Design

- Subdomains
- Bounded Context + Ubiquitous Language
- Context Mapping

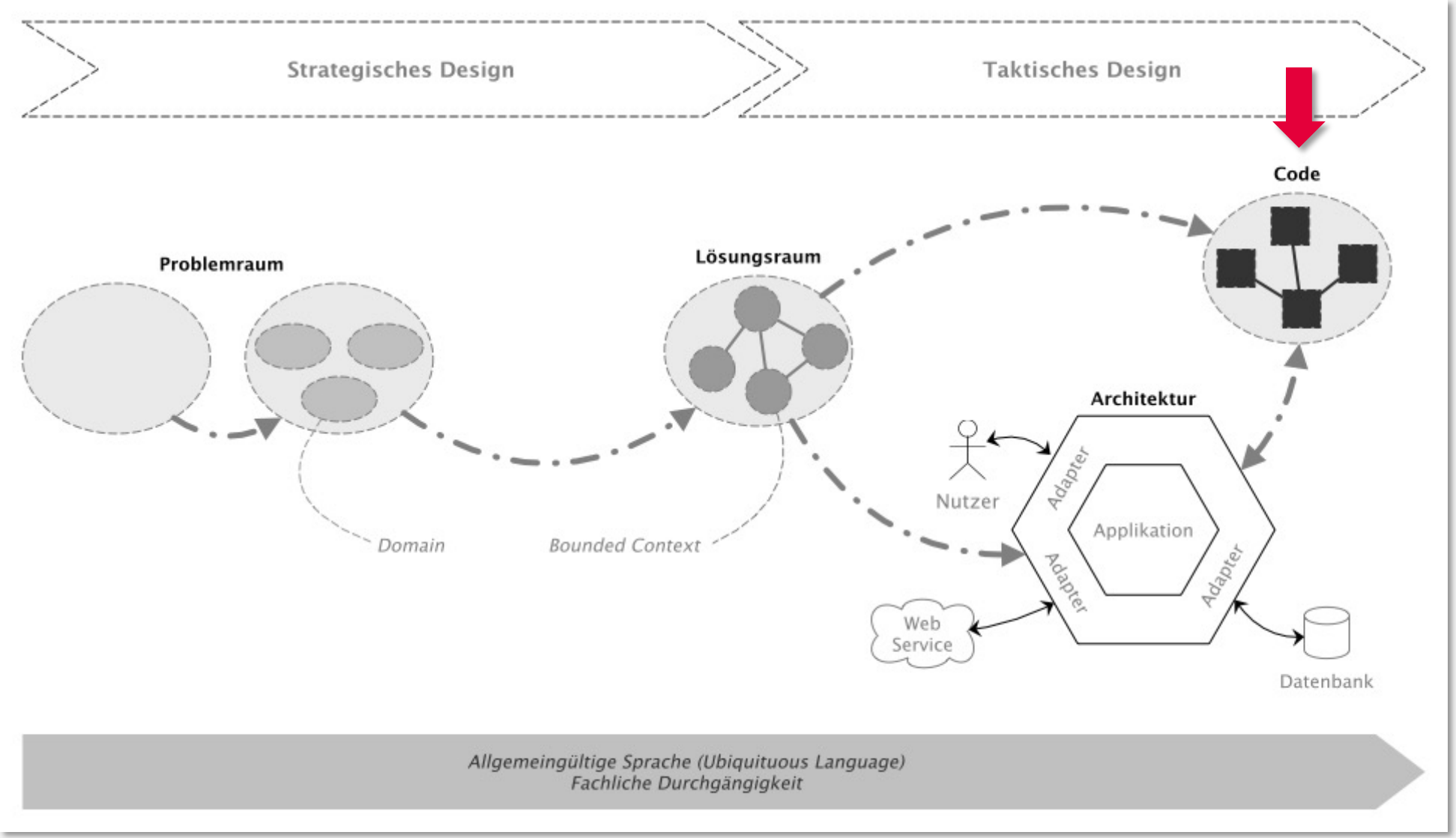
Taktisches Design

- Business Logic Pattern
- Architectural Pattern



DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



TACTICAL DESIGN

CLOUD NATIVE COMPUTING

- Bisher haben wir uns mit strategischen Design-Entscheidungen beschäftigt
- Im Tactical Design geht es um die Implementierung von Komponenten (Bounded Contexts)
- Es werden Entwurfsmuster für das Tactical Design besprochen
- Die Pattern werden auf einem Überblicksniveau präsentiert, um Unterschiede und geeignete Einsatzgebiete zu erläutern

1. Geschäftslogik implementieren
2. Geeignete Architekturmuster
3. Interaktionen zwischen Bounded Contexts

Martin Fowler et al., 2002

Siehe Unit 7

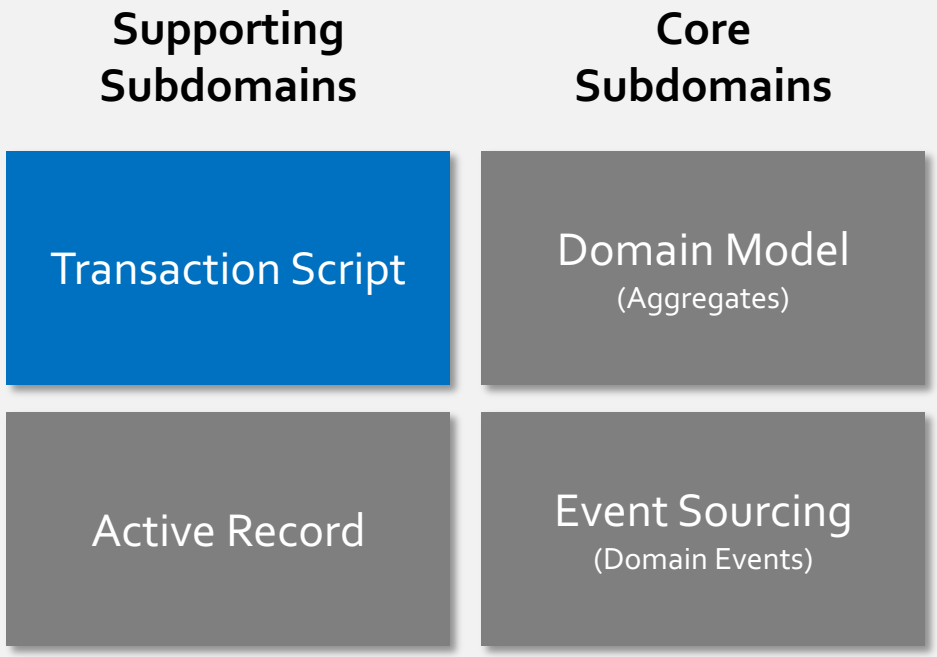
Insbesondere:

- *Req./Resp. (u. a. gRPC, GraphQL, REST)*
- *Messaging*

TACTICAL DESIGN

Implementierungsmuster für die Geschäftslogik

- Nicht alle Subdomänen sind gleich
- Unterschiedliche Subdomänen haben unterschiedliche strategische Bedeutung und Komplexität
- Wir betrachten exemplarisch vier verschiedene Muster für Geschäftslogik
- Jedes Muster ist für einen anderen Grad an Komplexität in der Geschäftsdomäne geeignet



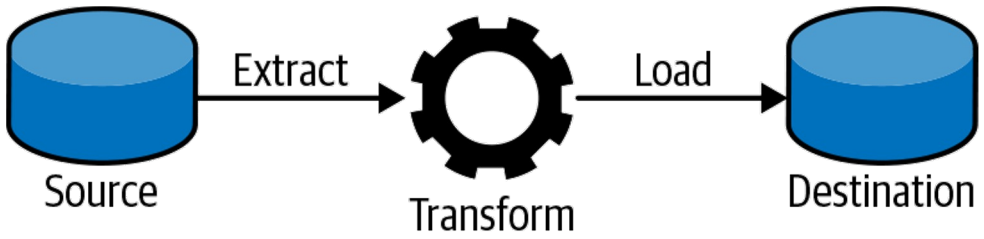
PATTERN FÜR GESCHÄFTSLOGIK

Supporting Subdomains: Transaction Script

CLOUD NATIVE COMPUTING

Extract – Transform - Load

Dieses Muster eignet sich gut für die einfachsten Problemdomänen, in denen die Geschäftslogik ETL-Operationen (Extract-Transform-Load) ähnelt, d. h. jede Operation extrahiert Daten aus einer Quelle, wendet Transformationslogik an, um sie in eine andere Form umzuwandeln, und lädt das Ergebnis in den Zielspeicher.

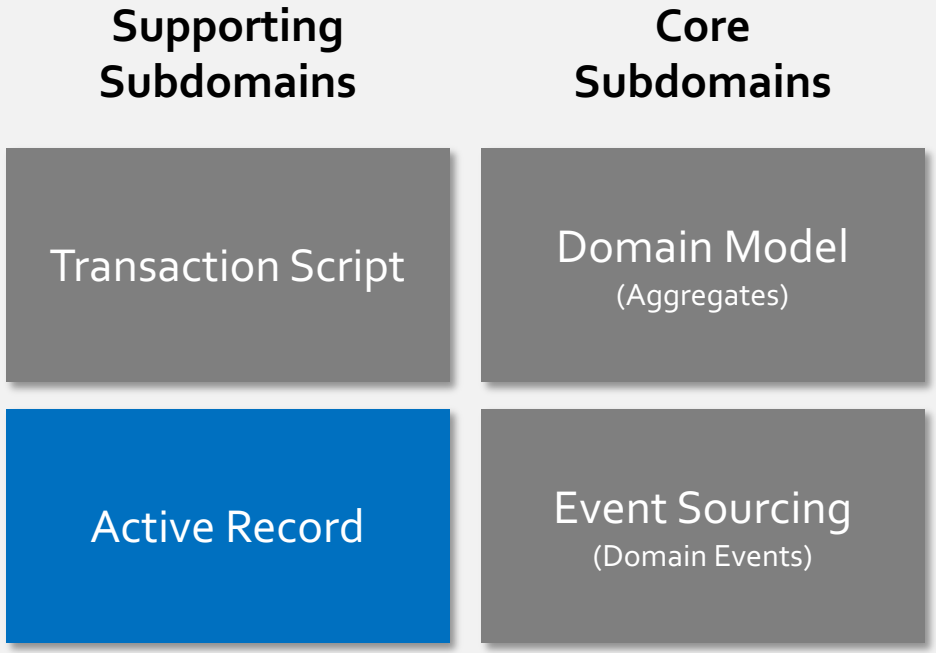


- Geschäftslogik wird durch einfache Prozeduren realisiert
- Jede Prozedur wird als simples, prozedurales Skript implementiert
- Dünne Abstraktionsschicht für Integration mit Speichermechanismen ist optional
- Direkter Zugriff auf Datenbanken ist erlaubt
- Einzige Anforderung ist transaktionales Verhalten
- Operationen müssen erfolgreich sein oder fehlschlagen
- Bei Fehlschlag bleibt das System in einem konsistenten Zustand
- Daher der Name des Musters: Transaktionsskript

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

TACTICAL DESIGN

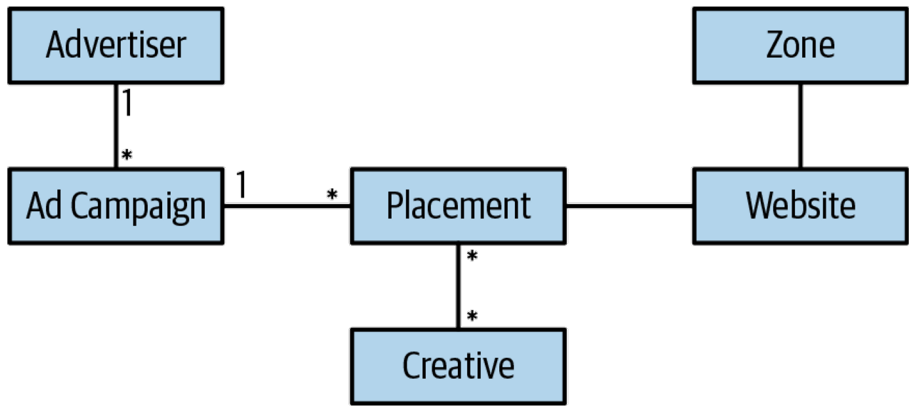
Implementierungsmuster für die Geschäftslogik



PATTERN FÜR GESCHÄFTSLOGIK

Supporting Subdomains: Active Record

Wie das vorherige Muster unterstützt auch dieses Muster Fälle, in denen die Geschäftslogik einfach ist. Hier kann die Geschäftslogik jedoch auf komplexeren Datenstrukturen operieren. Zum Beispiel können wir anstelle von flachen Datensätzen kompliziertere Objektbäume und Hierarchien haben.



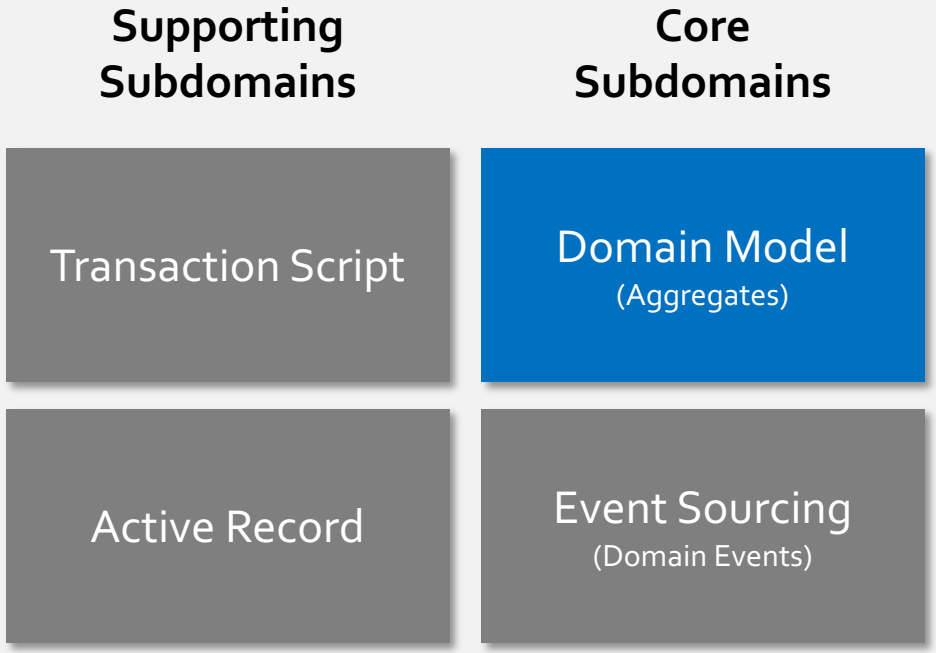
- Bei komplexeren Datenstrukturen führen Transaktionsskripte zu viel sich wiederholendem Code
- Das Active Record Muster verwendet dedizierte Objekte: aktive Datensätze
- Aktive Datensätze repräsentieren Datenstruktur und implementieren erforderliche CRUD-Operationen
- Objekte sind allerdings von Object Relational Mapping Frameworks (ORM) oder anderen Datenzugriffs-Frameworks abhängig
- Zugriff erfolgt nicht direkt auf Datenbank, sondern über Active Records.
- Der Name des Musters leitet sich von der Tatsache ab, dass jedes Record "aktiv" ist und die erforderliche Datenzugriffslogik implementiert.

```
public class CreateUser {
    public void Execute(userDetails) {
        try {
            DB.StartTransaction();

            var user = new User();
            user.Name = userDetails.Name;
            user.Email = userDetails.Email;
            user.Save();

            DB.Commit();
        } catch {
            DB.Rollback();
        }
    }
}
```

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

TACTICAL DESIGN

Implementierungsmuster für die Geschäftslogik



PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Domain Model

Nach der Definition von Martin Fowler ist ein Domänenmodell ein Objektmodell einer Domäne, das sowohl Verhalten als auch Zustand (Daten) enthält. Die taktischen Pattern von DDD liefern u.a. die folgenden Bausteine für solche Objektmodelle:

- Value Object
- Aggregate
- Domain Event

Alle diese Muster haben eines gemeinsam. Die Geschäftslogik steht an erster Stelle und vor allem in direktem Bezug zur Ubiquitous Language.

Dieses Muster ermöglicht es dem Code, die Ubiquitous Language zu "sprechen" und den mentalen Modellen der Domänenexperten konzeptionell sehr eng zu folgen.

Value Objects
Value Objekts sind Objekte, die durch ihre Werte eindeutig identifiziert werden können und sich nicht ändern (Immutables). Ein triviales Beispiel für ein Value Objekt ist das String-Objekt in Java. Dieses ist unveränderlich und alle seine Operationen führen zu einer neuen Instanz eines Strings.

Aggregates
Ein Aggregat ist eine Entität, die eine Hierarchie von Objekten darstellt. Im Gegensatz zu einem Wertobjekt kann eine Entität nicht allein durch ihren Wert identifiziert werden. Das heißt, jedes Aggregat benötigt ein ID-Feld, um identifiziert zu werden. Aggregate sind (anders als Value Objects) Mutables, d.h. sie können ihren Zustand ändern.

Domain Events
Ein Domain Event ist eine Nachricht, die ein bedeutendes Ereignis beschreibt, das in der Geschäftsdomäne aufgetreten ist. Zum Beispiel: Auftrag bezahlt, Lager wieder aufgefüllt, Werbekampagne veröffentlicht, usw. Solche Events sind ein Teil der öffentlichen Schnittstelle eines Aggregats. Ein Aggregat veröffentlicht über sein Public Interface seine Domänenereignisse.

```
class Color {
  int red;
  int green;
  int blue
}
```

Auf folgenden Slides

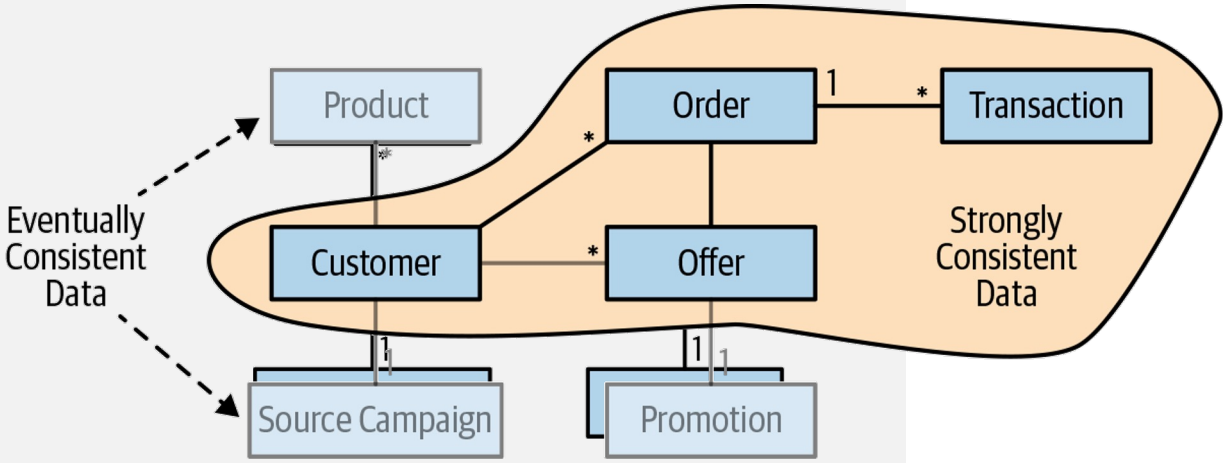
PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Domain Model (Aggregates)

Aggregates
Ein Aggregat ist eine Entität, die eine Hierarchie von Objekten darstellt. Im Gegensatz zu einem Wertobjekt kann eine Entität nicht allein durch ihren Wert identifiziert werden. Das heißt, jedes Aggregat benötigt ein ID-Feld, um identifiziert zu werden. Aggregate sind (anders als Value Objects) Mutables, d.h. sie können ihren Zustand ändern.

Daher ist es entscheidend, die Konsistenz von Aggregat-Zuständen zu schützen. Das Aggregatmuster zieht dazu eine klare Grenze zwischen dem Aggregat und seinem äußeren Bereich. Nur die Geschäftslogik des Aggregats darf seinen Zustand verändern. Alle Prozesse oder Objekte außerhalb des Aggregats dürfen nur seinen Zustand lesen oder seine öffentlichen Methoden ausführen.

Die öffentlichen Methoden des Aggregats sind für die Validierung der Eingabe und die Durchsetzung aller Geschäftsregeln und Invarianten verantwortlich. Diese strenge Abgrenzung stellt auch sicher, dass die gesamte Geschäftslogik in Bezug auf das Aggregat an einer Stelle implementiert wird - im Aggregat selbst.



Aggregate bilden Transaktionsgrenzen

Da der Zustand von Aggregaten nur durch ihre eigene Geschäftslogik geändert werden kann, fungiert die Aggregatgrenze in DDD auch immer als Transaktionsgrenze. Alle Änderungen am Zustand des Aggregats sollten transaktional als eine atomare Operation übertragen werden. Eine Transaktion darf sich also nur auf ein Aggregat beziehen.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

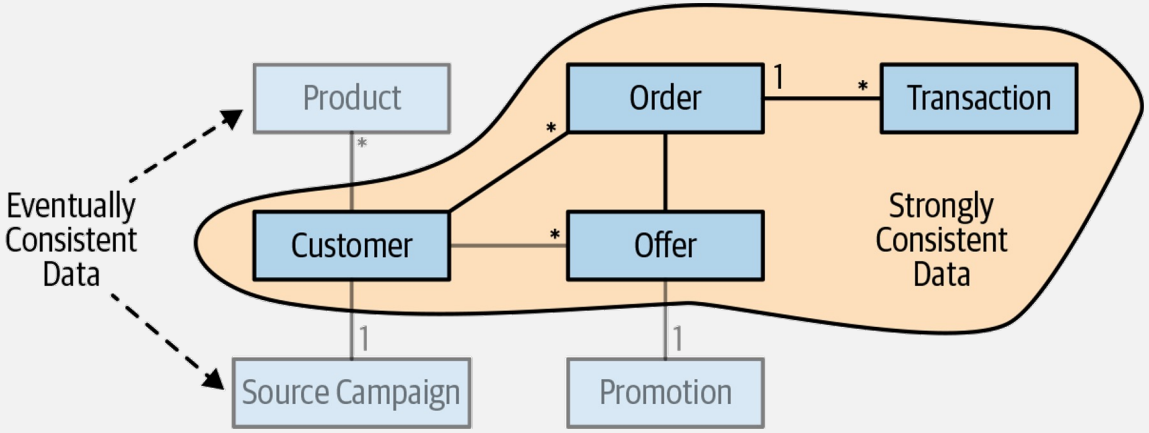
Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

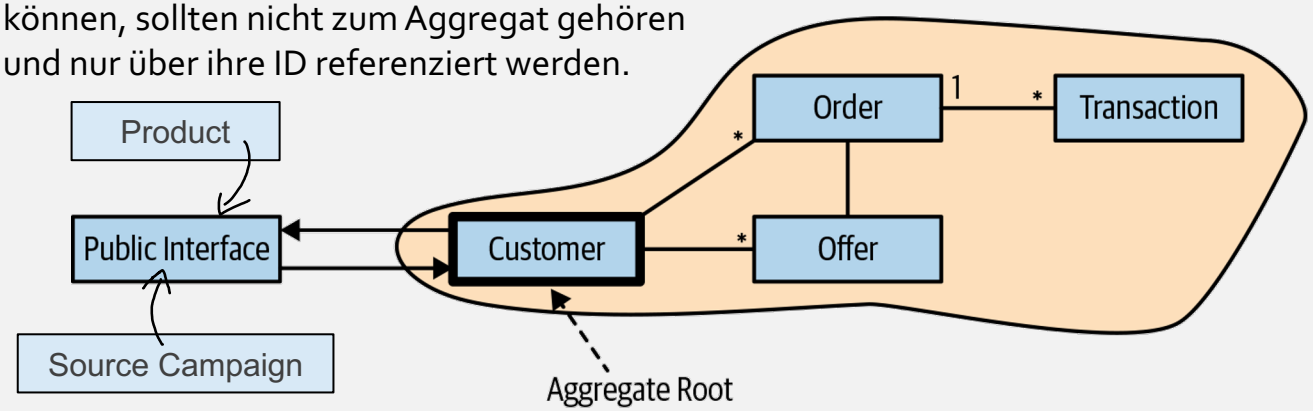
PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Domain Model (Aggregate als Transaktionsgrenzen)

Da alle in einem Aggregat enthaltenen Objekte dieselbe Transaktionsgrenze haben, können Leistungs- und Skalierbarkeitsprobleme auftreten, wenn Aggregate zu groß werden.

Man kann dies nutzen, um Grenzen von Aggregaten anhand eines Strict- und Eventual-Consistency Kriteriums zu schneiden. Die Konsistenz der Daten kann somit eine praktische Heuristik für den Entwurf der Aggregatgrenzen sein. Nur die Informationen, die für die Implementierung der Geschäftslogik des Aggregats streng konsistent sein müssen, sollten sich innerhalb der Grenzen des Aggregats befinden. Objekte, die eventuell konsistent sein können, sollten nicht zum Aggregat gehören und nur über ihre ID referenziert werden.





Aggregatwurzel (Aggregate Root):

Da ein Aggregat meist eine Hierarchie von Objekten darstellt, sollte aus Gründen des Zustandsschutz von Aggregaten nur eines von ihnen als öffentliche Schnittstelle des Aggregats bestimmt werden - die Wurzel des Aggregats

Regel 1:
Schütze fachliche Invarianten innerhalb von Aggregatgrenzen mittels Schnittstellen

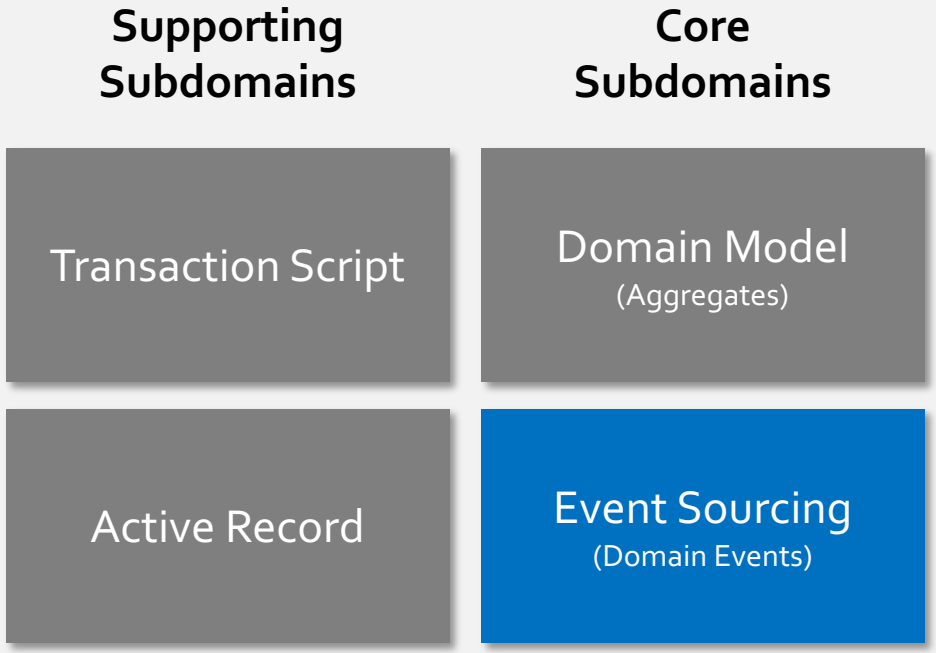
Regel 2:
Entwirf kleine Aggregate

Regel 3:
Referenziere andere Aggregate nur über ihre Identität

Regel 4:
Aktualisiere andere Aggregate unter Verwendung von Eventual Consistency

TACTICAL DESIGN

Implementierungsmuster für die Geschäftslogik



PATTERN FÜR GESCHÄFTSLOGIK

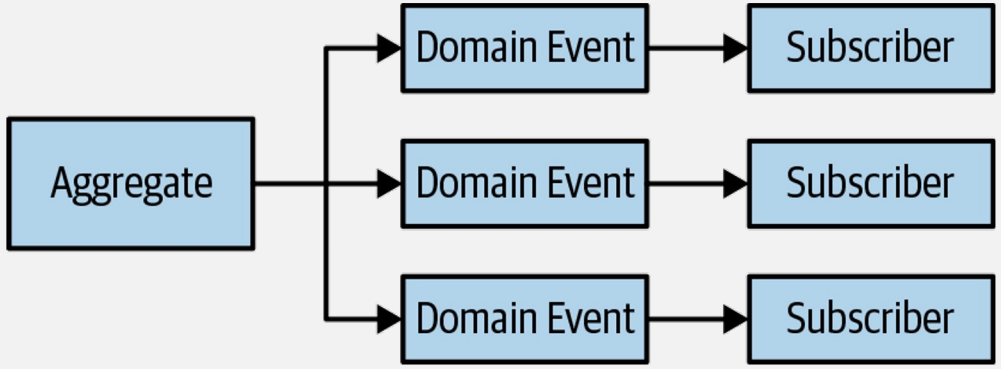
Core Subdomains: Domain Model (Domain Events)

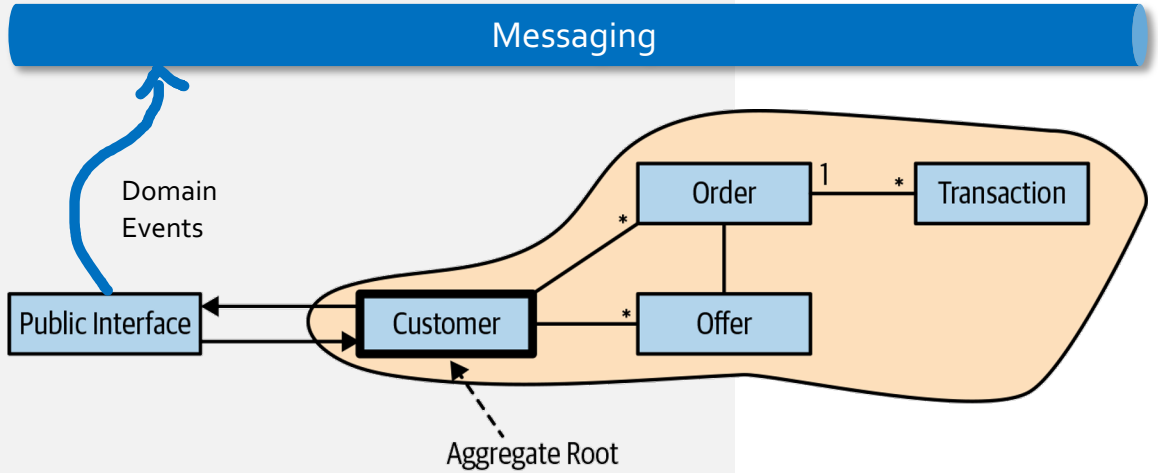
Domain Events
Ein Domain Event ist eine Nachricht, die ein bedeutendes Ereignis beschreibt, das in der Geschäftsdomäne aufgetreten ist. Zum Beispiel: Auftrag bezahlt, Lager wieder aufgefüllt, Werbekampagne veröffentlicht, usw.. Solche Events sind ein Teil der öffentlichen Schnittstelle eines Aggregats. Ein Aggregat veröffentlicht über sein Public Interface seine Domänenereignisse.

Domain Events sind ein Teil der öffentlichen Schnittstelle eines Aggregats. Ein Aggregat veröffentlicht seine Ereignisse. Andere Prozesse, Aggregate oder sogar externe Systeme können die Domänenereignisse abonnieren und als Reaktion darauf ihre eigene Logik ausführen.

Die hierzu passenden Interaktionspattern haben wir in Unit 07 als Pub/Sub und Queueing kennengelernt.

Vgl. auch Smart Endpoints / Dump Pipes





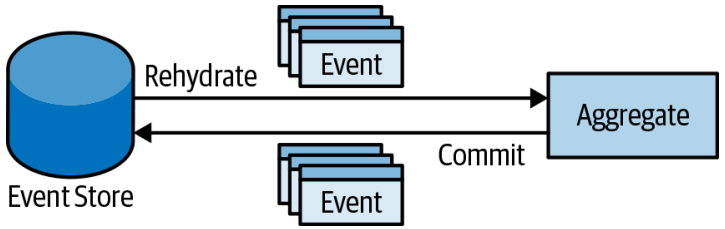
Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Event Sourcing

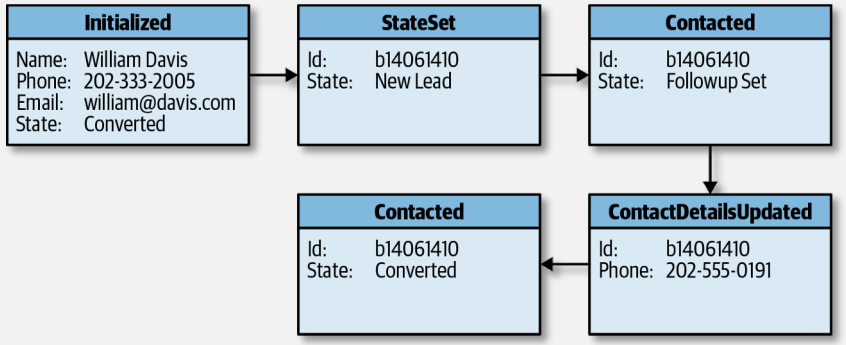
Das Event-Sourcing-Pattern verwendet Domain Events, um die Dimension der Zeit in das Domänenmodell einzuführen. Jede Änderung des Systemzustands sollte als Domänenereignis ausgedrückt und aufgezeichnet werden.



Die Datenbank, die die Domänenereignisse des Systems speichert, ist der einzige stark konsistente Speicher - die „Single Source of Truth" des Systems. Jede Operation an einem Aggregat, das von Ereignissen gespeist wird, folgt diesem Prozess:

1. Laden der Domänenereignisse des Aggregats.
2. Erstellung der Zustandsdarstellung.
3. Ausführen der Geschäftslogik und Erzeugen neuer Domain-Events.
4. Übertragen der neuen Domain-Events in den Event Store.

Event-Sourcing wird bspw. in der Finanzindustrie zur Darstellung von Änderungen in einem Ledger verwendet. Ein Ledger ist ein Append-Only-Log das den Verlauf von Transaktionen protokolliert. Ein aktueller Zustand (z. B. der Kontostand) kann immer durch ein Replaying der Ledger-Datensätze abgeleitet werden.



Replaying Time Machine
Da Domain Events verwendet werden können, um den aktuellen Zustand eines Aggregats wiederherzustellen, können sie auch für die Wiederherstellung aller vergangenen Zustände des Aggregats verwendet werden.

Tiefer Einblick
Event Sourcing bietet tiefe Einblicke in den Zustand und das Verhalten des Systems. Außerdem ermöglicht das flexible Modell die Umwandlung der Ereignisse in verschiedene Zustandsdarstellungen - auch solche, die ursprünglich nicht geplant waren.

Audit log
Die persistierten Domain-Events stellen ein stark konsistentes Audit-Protokoll von allem dar, was mit den Zuständen der Aggregate passiert ist. Gesetze verpflichten einige Geschäftsdomänen, solche Audit-Protokolle zu implementieren (z.B. Finanzindustrie). Event Sourcing bietet dies von Haus aus.

CLOUD NATIVE COMPUTING

Event Sourcing ist besonders praktisch für Systeme, die Geld oder monetäre Transaktionen verwalten. Es erlaubt, die Entscheidungen, die das System getroffen hat, und den Geldfluss zwischen Konten leicht nachzuvollziehen.

Im Vergleich zu einer zustandsbasierten Darstellung erfordert das Event-Sourcing-Modell mehr Aufwand bei der Modellierung. Allerdings ist dieses Muster insbesondere in den blauen Szenarien erwägenswert.

AUSBLICK

Domain Driven Design

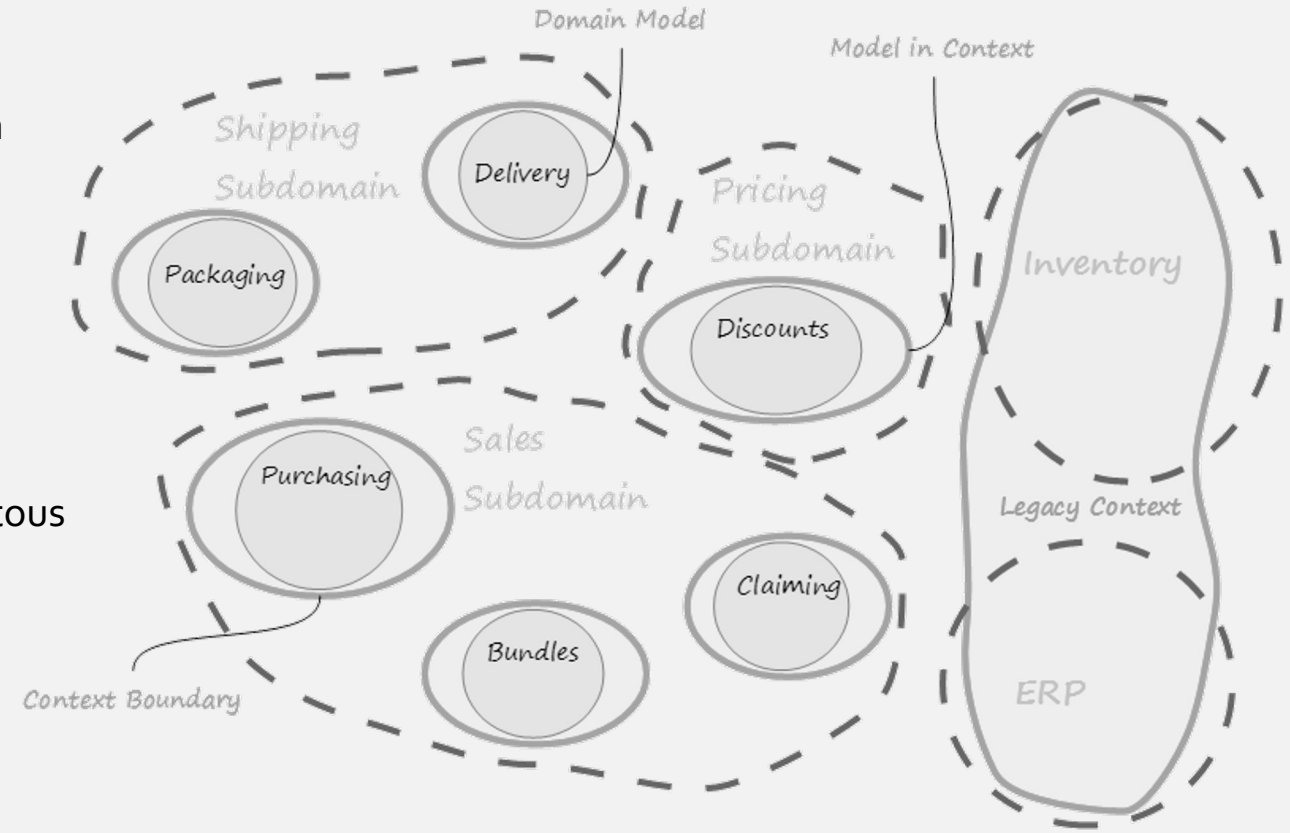
- Was ist das?
- Effektives Software Design
- Strategisches Design
- Taktisches Design

Strategisches Design

- Subdomains
- Bounded Context + Ubiquitous Language
- Context Mapping

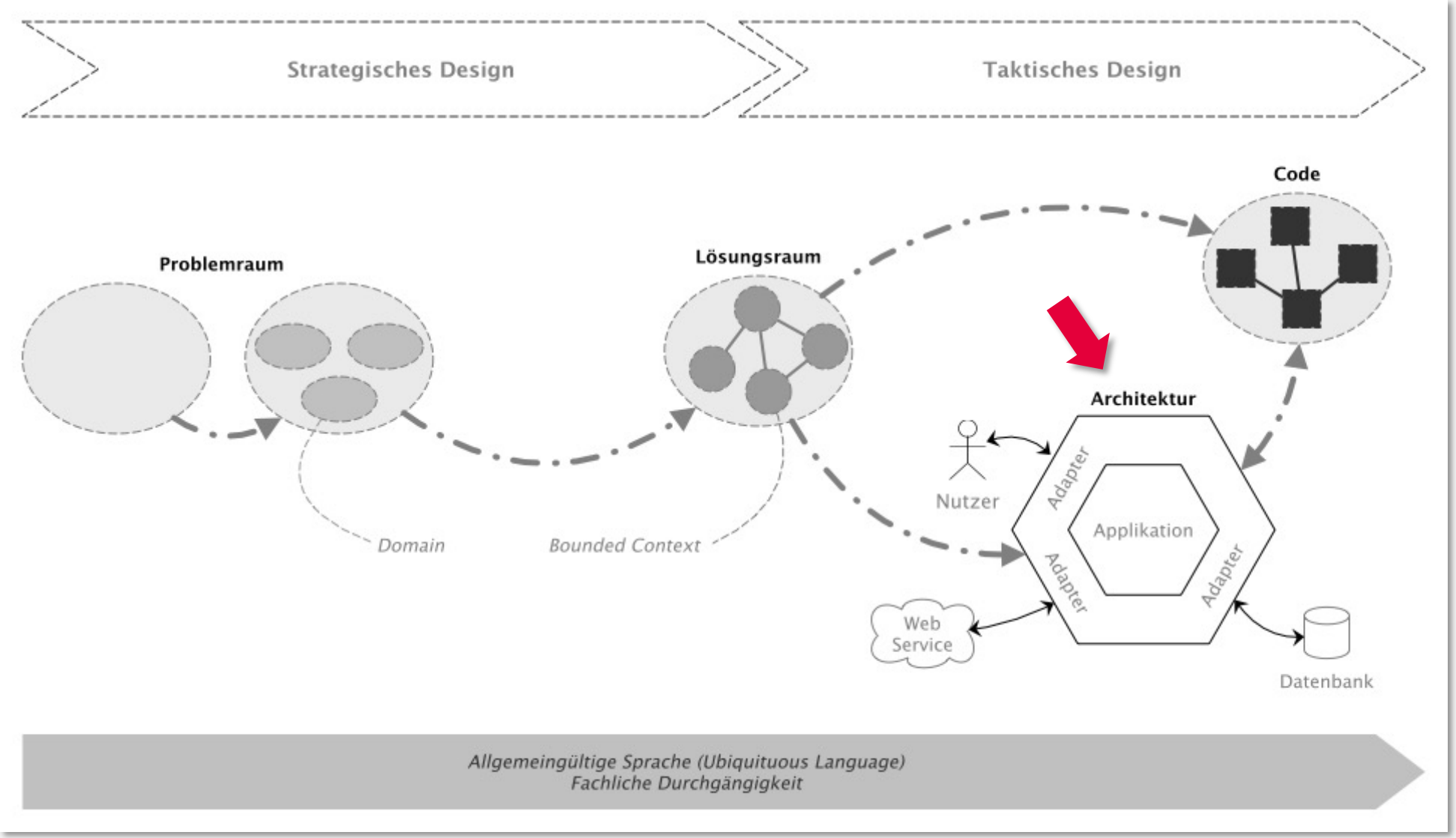
Taktisches Design

- Business Logic Pattern
- Architectural Pattern



DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de