

# DOMAIN DRIVEN DESIGN

Maurice Müller

2023-10

# DOMAIN DRIVEN DESIGN

- Philosophie für die Modellierung (komplexer) Software
- erstmals vorgestellt 2003 im "Big Blue Book" von Eric Evans
- Grundannahme: das Design einer Software wird von der **Fachlichkeit** der zugrunde liegenden **Problemdomäne** getrieben

# INHALTE

- Einführung
- Strategisches Domain-Driven-Design
- Taktisches Domain-Driven-Design

# **TEIL 1: EINFÜHRUNG**

# WOZU SOFTWAREENTWICKLUNG?

- Sinn einer Software ist die Unterstützung bei der Bewältigung von Aufgaben/Problemen innerhalb eines bestimmten Problembereichs
- diesen Problembereich, für den die Software entwickelt wird, nennt man **Domäne** (engl. *domain*)

# DOMÄNE

- auch: *Problemdomäne, Anwendungsdomäne*
- abgrenzbares Problemfeld
- bestimmter Bereich für den Einsatz von Software
- **was** mit der Software gelöst werden soll

A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.

– Eric Evans

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	
Firefox	
Spotify	
Youtube	
Amazon	

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	
Spotify	
Youtube	
Amazon	

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	Internetbrowser
Spotify	
Youtube	
Amazon	

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	Internetbrowser
Spotify	Music-On-Demand
Youtube	
Amazon	

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	Internetbrowser
Spotify	Music-On-Demand
Youtube	Video-Content-Plattform
Amazon	

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	Internetbrowser
Spotify	Music-On-Demand
Youtube	Video-Content-Plattform
Amazon	<ul style="list-style-type: none"><li>• E-Commerce</li></ul>

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	Internetbrowser
Spotify	Music-On-Demand
Youtube	Video-Content-Plattform
Amazon	<ul style="list-style-type: none"><li>• E-Commerce</li><li>• Logistik</li></ul>

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	Internetbrowser
Spotify	Music-On-Demand
Youtube	Video-Content-Plattform
Amazon	<ul style="list-style-type: none"><li>• E-Commerce</li><li>• Logistik</li><li>• Rechenzentrum</li></ul>

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	Internetbrowser
Spotify	Music-On-Demand
Youtube	Video-Content-Plattform
Amazon	<ul style="list-style-type: none"><li>• E-Commerce</li><li>• Logistik</li><li>• Rechenzentrum</li><li>• XY-On-Demand</li></ul>

# BEISPIELE FÜR DOMÄNEN

Software / Firma	Domäne(n)
Signal (App)	sichere Kurznachrichten
Firefox	Internetbrowser
Spotify	Music-On-Demand
Youtube	Video-Content-Plattform
Amazon	<ul style="list-style-type: none"><li>• E-Commerce</li><li>• Logistik</li><li>• Rechenzentrum</li><li>• XY-On-Demand</li></ul>

# DOMÄNENMODELL

- damit Software Probleme aus einer bestimmten Domäne bearbeiten kann, muss sie bestimmte fachliche Aspekte (Daten, Regeln, Verhalten) dieser Domäne im Code abbilden
- diese Abbildung bezeichnet man als **Domänenmodell**

# ABSTRAKTION DER WIRKLICHKEIT

- das Domänenmodell bildet nicht detailgetreu die Wirklichkeit ab, sondern nur die Aspekte, die zur Lösung der geforderten Aufgaben auch relevant sind

# ABSTRAKTION DER WIRKLICHKEIT

- das Domänenmodell bildet nicht detailgetreu die Wirklichkeit ab, sondern nur die Aspekte, die zur Lösung der geforderten Aufgaben auch relevant sind



# ABSTRAKTION DER WIRKLICHKEIT

- das Domänenmodell bildet nicht detailgetreu die Wirklichkeit ab, sondern nur die Aspekte, die zur Lösung der geforderten Aufgaben auch relevant sind



```
class Refrigerator {  
    double length;  
    double width;  
    double height;  
}
```

# **PROZESSE DER MODELLBILDUNG**

# VISUALISIERUNG DES MODELLS

- ein Modell einer Problemdomäne existiert zuerst in der Vorstellung der Entwickler
- damit diese darüber sprechen können, muss das Modell visualisiert werden, z.B. als UML, Architekturdiagramm, etc.
- die Visualisierung ist der **IST**-Zustand und nicht der **SOLL**-Zustand
- **Die wichtigste und konkreteste Form der Visualisierung eines Modells ist der Quellcode**

# DESIGNVORGANG

- für eine gegebene, nicht-triviale Problemdomäne gibt es eine Vielzahl von Möglichkeiten, diese als Modell abzubilden
- es müssen also Entscheidungen getroffen werden, wie welche **Teile der Domäne** im Modell abgebildet werden
- dieser Entscheidungsprozess ist der Designvorgang

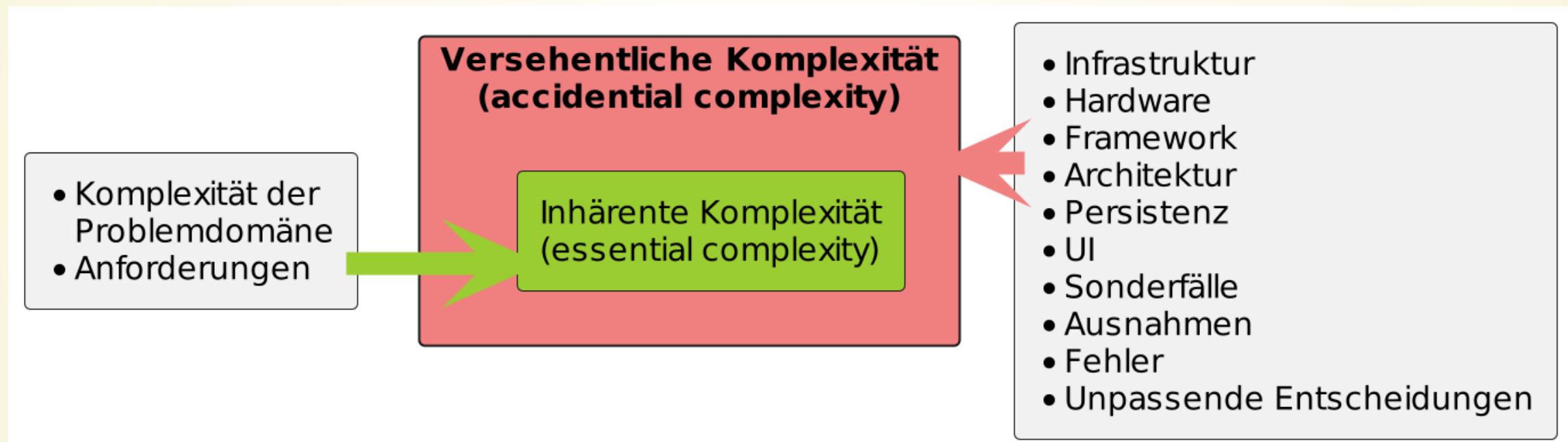
# **DESIGN**

**Design ist die Summe aller bewussten und unbewussten Entscheidungen, die Einfluss darauf haben, wie eine konkrete Problemdomäne als Modell abgebildet wird.**

# **SOFTWARE-KOMPLEXITÄT**

- Software und Code sind sehr **komplex**
- **Komplexität** ist definiert als der Grad in dem ein System oder eine Komponente eine schwierig zu verstehende und zu kontrollierende Implementierung hat
- Es gibt zwei Arten von Komplexität
  - Inhärente Komplexität
  - Versehentliche Komplexität

# SOFTWARE-KOMPLEXITÄT



# SOFTWARE-KOMPLEXITÄT

- die **inhärente Komplexität der Problemdomäne** (*essential complexity*) ist fest gegeben
  - damit sollten sich Entwickler eigentlich beschäftigen
- die zusätzliche **versehentliche Komplexität** (*accidental complexity*) ist ein notwendiges Übel und kann nicht vollständig verhindert werden
  - Ziel: *versehentliche Komplexität* darf nicht die *inherante Komplexität* (negativ) beeinflussen

# BIG BALL OF MUD



*"A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle."*

– Brian Foote & Josphe Yoder

*"Eine Riesenmatschkugel ist ein planlos strukturierter, wuchernder, schludriger, mit Isolierband und Draht umwickelter Spaghetti-Code Dschungel."*

# BIG BALL OF MUD

*Code that does something useful, but without explaining how*

— Eric Evans

- Eigenschaften
  - Code verbirgt Intention (was/warum)
  - technische Belange belasten/verunreinigen die Fachlogik
  - Fachlogik ist über die (gesamte) Anwendung verteilt
- Resultat
  - selbst kleine Anpassungen sind nur mit hohem Aufwand (und hohem Risiko!) umsetzbar

# BIG SMALL BALLS OF MUD

- es lohnt sich nicht immer alles korrekt zu designen
- manchmal ist Geschwindigkeit wichtiger als Qualität
- wichtig ist, dass der *Small Ball of Mud* isoliert vom Rest bleibt und eine saubere Schnittstelle hat
- Microservices

# WIE HILFT DOMAIN-DRIVEN-DESIGN?

- Reduktion des Übersetzungsaufwands
  - gleiche Begriffe in Domäne und Code
  - klare Modellierung der Fachlichkeit
  - Rückschlüsse von Problemdomäne auf Code und umgekehrt möglich
- Methoden und Muster
  - Strategisches DDD
  - Taktisches DDD

# STRATEGISCHES DDD

- beschäftigt sich mit dem Verständnis der Domäne und hilft beim Analysieren, Aufdecken, Abgrenzen, Dokumentieren und Begreifen der Fachlichkeit
- Ziel: präzises und tiefgreifendes, gemeinsames Verständnis der Problemdomäne zwischen Domänenexperten und Entwicklern

# **TAKTISCHES DDD**

# ZUSAMMENFASSUNG

- Software ist komplex
- Komplexität ist definiert als der Grad in dem ein System oder eine Komponente eine schwierig zu verstehende und zu kontrollierende Implementierung hat
  - Inhärente Komplexität
  - Versehentliche Komplexität
- Unbeherrschte Komplexität führt zu einem *Big Ball of Mud*
- DDD hilft dem vorzubeugen durch:
  - Fokus auf ein präzises und tiefgreifendes, gemeinsames
  - Verständnis der Problemdomäne
  - Kapselung von inhärenter Komplexität(Fachlichkeit) und versehentlicher Komplexität

# **STRATEGISCHES DDD**

# UBIQUITOUS LANGUAGE

- gleiche Begriffe in Domäne und Sourcecode
- „allgegenwärtige Sprache“
- jede Domäne besitzt eine eigene Fachsprache
- *nicht* versuchen, diese Fachsprache in eigene Begriffe zu übersetzen
- Ubiquitous Language bezeichnet die von Domänenexperten und Entwicklern gemeinsam im Projekt verwendete Sprache
- hauptsächlich geben die Domänenexperten diese vor

# VERSTÄNDNISPROBLEME IN DER KOMMUNIKATION

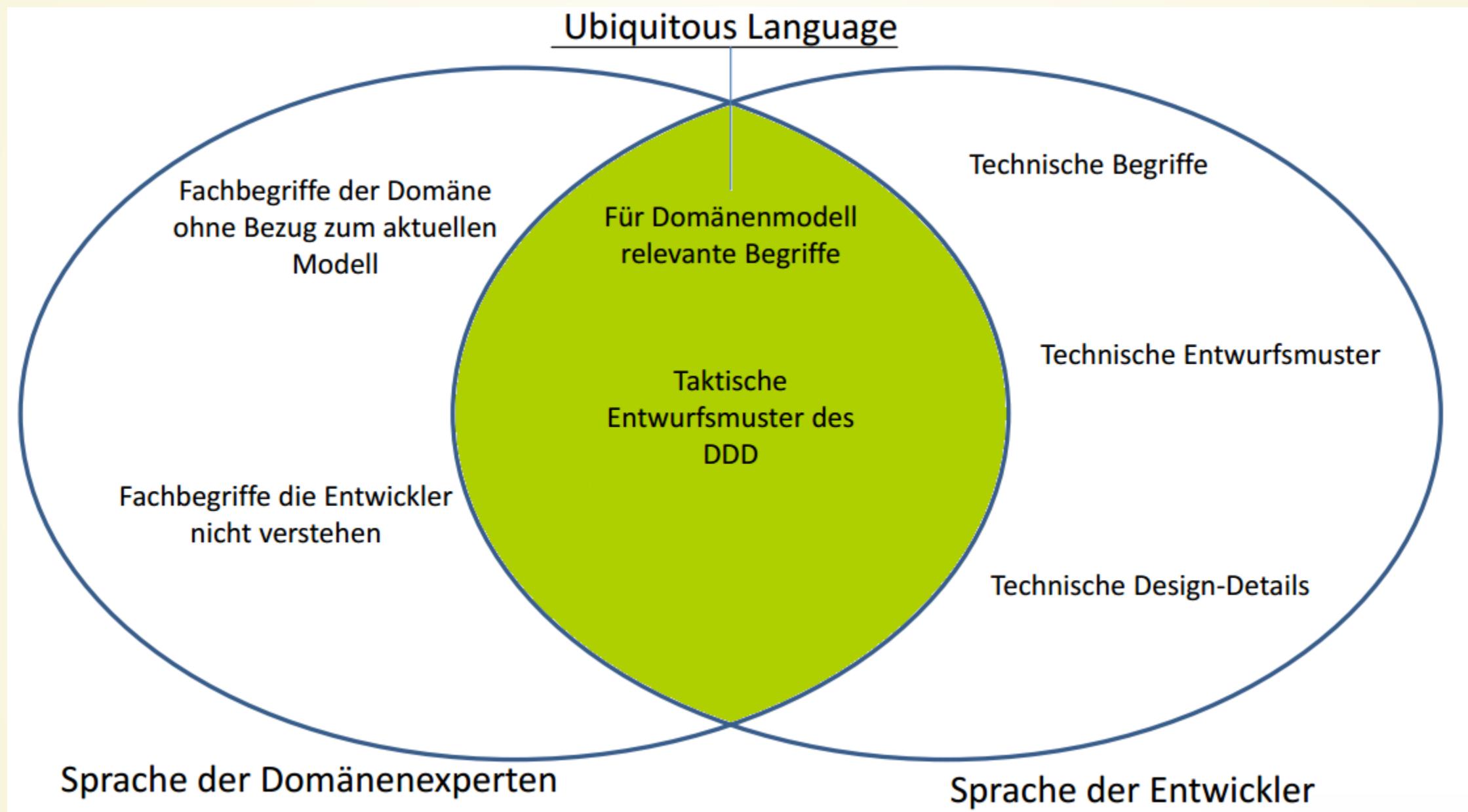
- Domänenexperten verstehen die Sprache der Entwickler meistens nur sehr begrenzt
  - wenn ein Entwickler in seiner Sprache über das Domänenmodell spricht, hängt er den Experten ab
- Entwickler verstehen die Sprache der Domänenexperten meistens nur sehr begrenzt
  - wenn ein Domänenexperte in seiner Sprache über die Domäne spricht, verliert er den Entwickler schnell, weil dieser immer „übersetzen“ will/muss
- keine der beiden Sprachen eignet sich alleine für das Projekt
- jeder Stakeholder hat seine eigene gedankliche Version „der Domäne“ → umso wichtiger ist ein gemeinsames Verständnis

# **VERSTÄNDNISPROBLEME IM CODE**

# BEISPIEL FÜR MEHRDEUTIGKEIT

- ein Online-Marktplatz hat Händler und Käufer
- *User* kann sowohl Händler als auch Käufer sein
- noch schlimmer: Fallunterscheidung zwischen angemeldeten und nicht-angemeldeten Nutzern
- daraus ergeben sich uneindeutige Anforderungen
  - „Jeder User soll seine Rechnungsadresse selbstständig ändern können“

# GEMEINSAME PROJEKTSPRACHE



# GEMEINSAME PROJEKTSPRACHE FINDEN

# **ARTEFAKTE MIT PROJEKTSPRACHE**

# ARTEFAKTE MIT PROJEKTSPRACHE

- Anforderungsdokument bzw. Anforderungen
- Benutzerhandbuch
- Fehlerberichte und Change Requests
- Benutzeroberfläche
- Fachmodell (Domänenmodell)
- Softwaredesign (Klassen- und Methoden)
- Sourcecode (Implementierung)
- entwicklungsinterne Kommunikation (Issues)

# GRENZEN DER PROJEKTSPRACHE

- nicht alle Begriffe und Zusammenhänge sind für das Projekt von Bedeutung
- immer auf den Kern des Projekts konzentrieren
  - Rangbegriffe auch mal unspezifiziert lassen
- es darf innerhalb eines Projekts gut und klar modellierte Kernbereiche und gleichzeitig unscharf oder gar nicht modellierte Randbereiche geben
  - etwas Schlamm (Mud) gibt es in jedem Projekt

# SCHÄRFUNG DER PROBLEMDOMÄNE

- normalerweise ist es nicht sinnvoll, ein einziges, großes Modell für die gesamte Problemdomäne zu entwerfen
  - DDD ist ein aufwendiges Verfahren
  - präzise, eindeutige *Ubiquitous Language* ist umso schwieriger je größer die (betrachtete) Problemdomäne ist
- die (betrachtete) Problemdomäne sollte daher möglichst klein sein
- DDD führt dazu das Konzept *Subdomäne* ein
- Subdomänen unterteilen die Problemdomäne in *Kern-Domäne*, *unterstützende Domäne* und *generische Domäne*

# PROBLEMDOMÄNE: ONLINE-MARKTPLATZ



# KERN-DOMÄNE

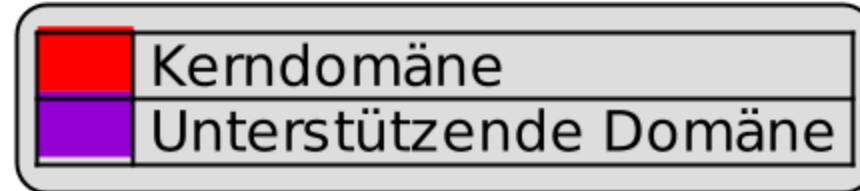
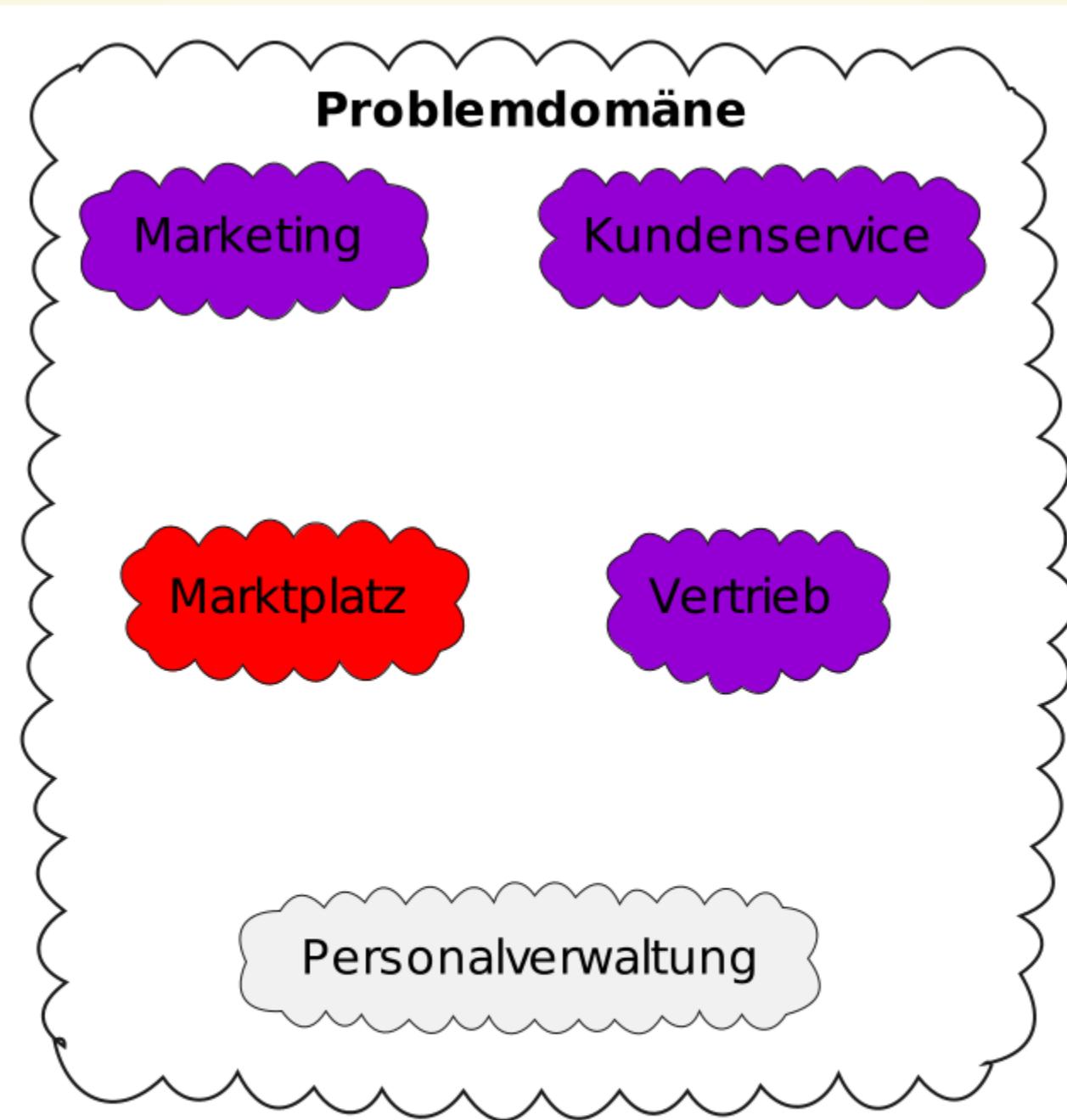
- Kerngeschäft des Unternehmens (womit das Geld erwirtschaftet wird)
  - jedes Unternehmen hat mindestens eine Kerndomäne
  - in diesem Bereich ist Qualität und Flexibilität der Software am wichtigsten
  - für diesen geschäftskritischen Teil der Problemdomäne lohnt sich die Entwicklung eines reichhaltigen Modells mit DDD
- Evans nennt drei Fragen zur Identifikation der Kerndomäne
  - "What makes the system worth writing?"
  - "Why not buy it off the shelf?"
  - "Why not outsource it?"



# UNTERSTÜTZENDE DOMÄNE

*engl. supporting domain*

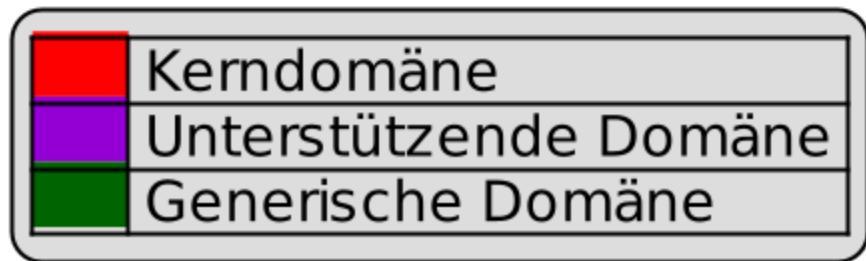
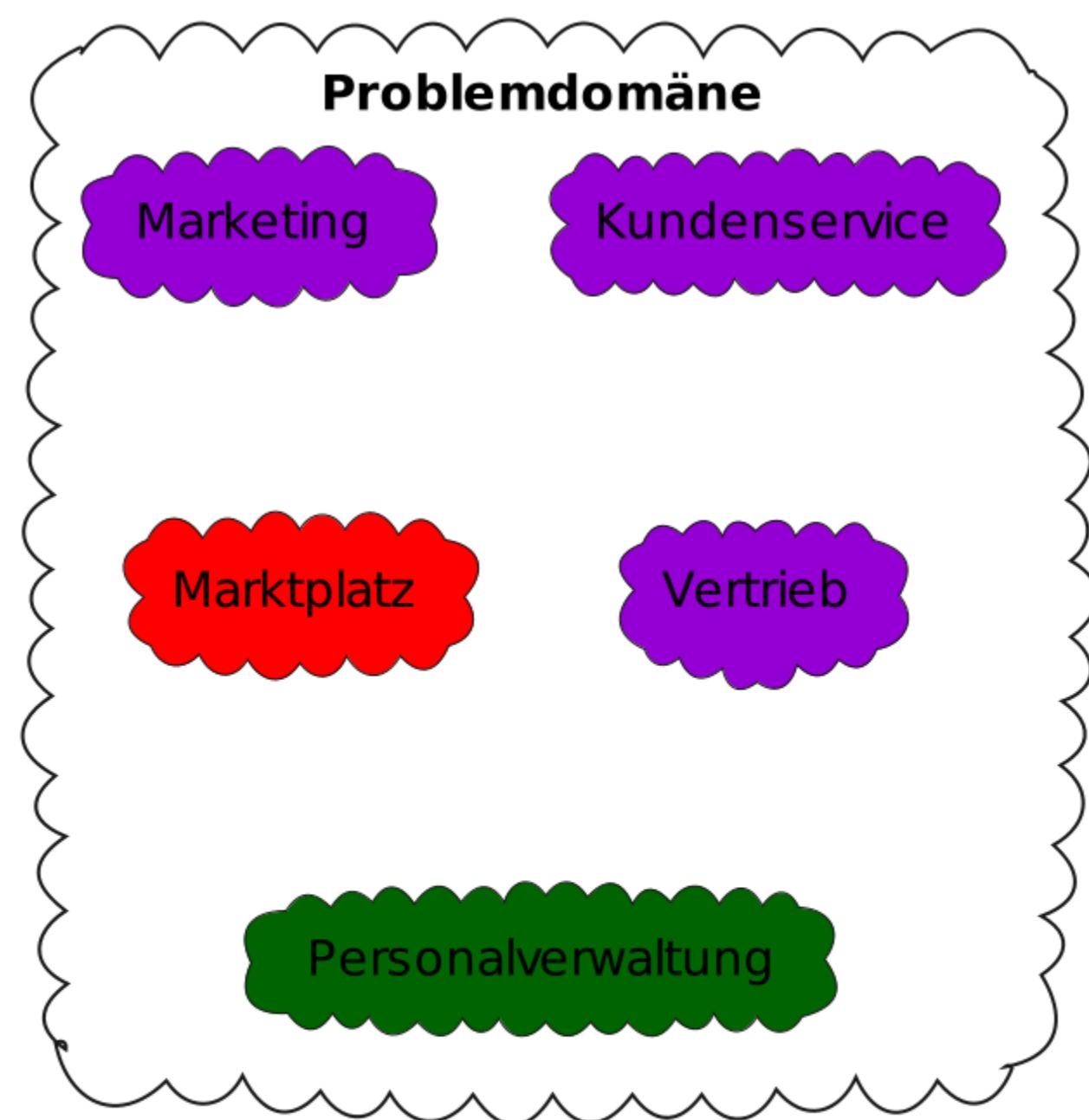
- beschreibt Teile der Problemdomäne, die für die Arbeit der Kerndomäne wichtig sind
- haben *unterstützende* Funktion für die Kerndomäne
- hier reicht ggf. die Entwicklung mit weniger aufwändigen Methoden als DDD oder sogar der Einsatz von Fremdsoftware



# GENERISCHE DOMÄNE

engl. generic domain

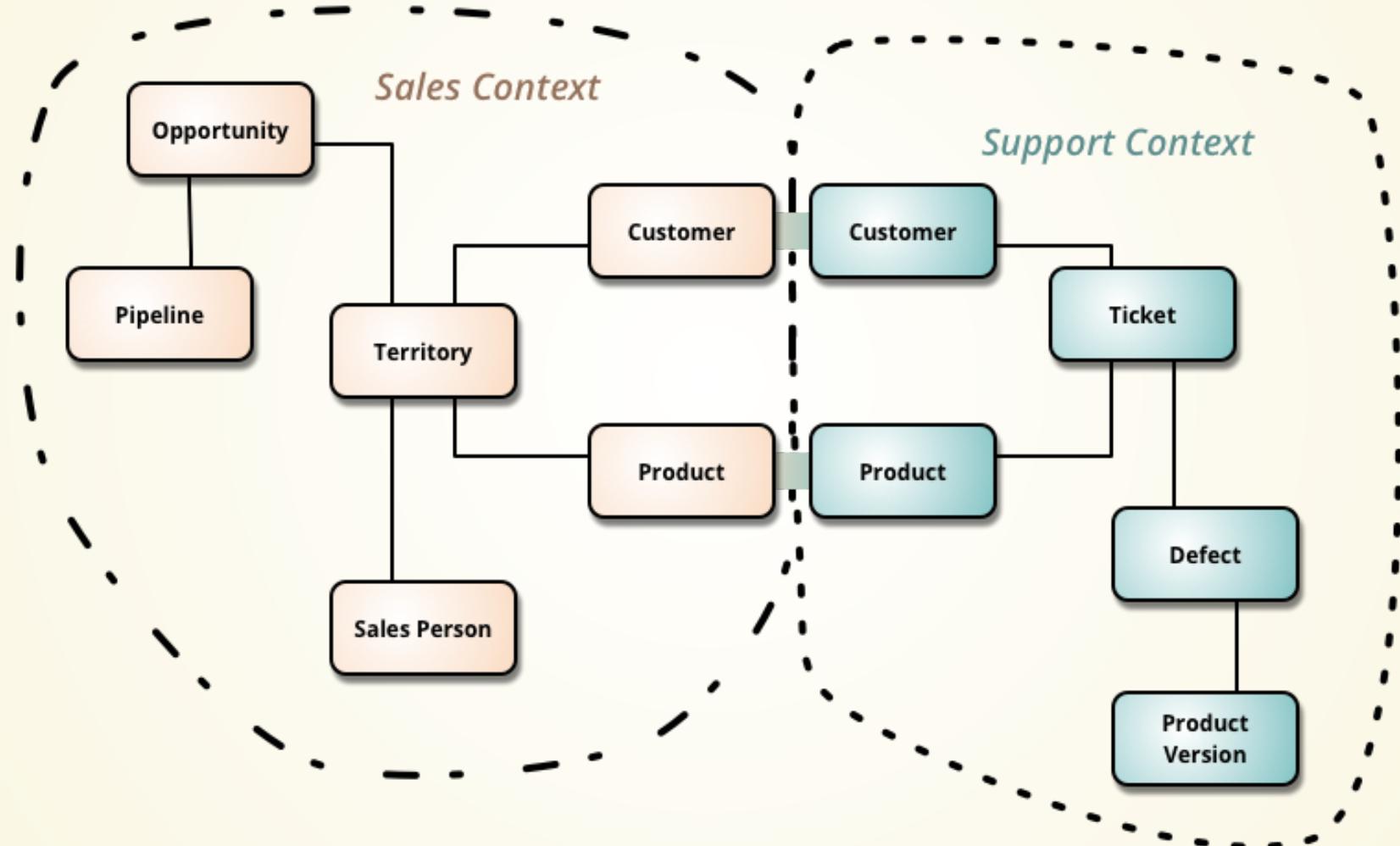
- beschreibt Teile der Problemdomäne, die für das tägliche Geschäft unerlässlich sind, jedoch nicht um Kerngeschäft gehören
- *Beispiel:* die meisten Unternehmen müssen in irgendeiner Art Rechnungen schreiben (das hat aber in der Regel nichts mit dem Kerngeschäft zu tun)
- hier reicht meistens Dritthersteller-Software oder Outsourcing



# BOUNDED CONTEXT

- eine Anwendung - und damit ein Modell - ist selten isoliert
- meistens muss ein Modell mit anderen Modellen kommunizieren bzw. steht zu diesen in Beziehung
  - größere Anwendungssysteme können aus mehreren, wohldefinierten Modellen bestehen
  - Modell benötigt Daten aus anderen Modellen (Schnittstellen für Datenimport/-export)

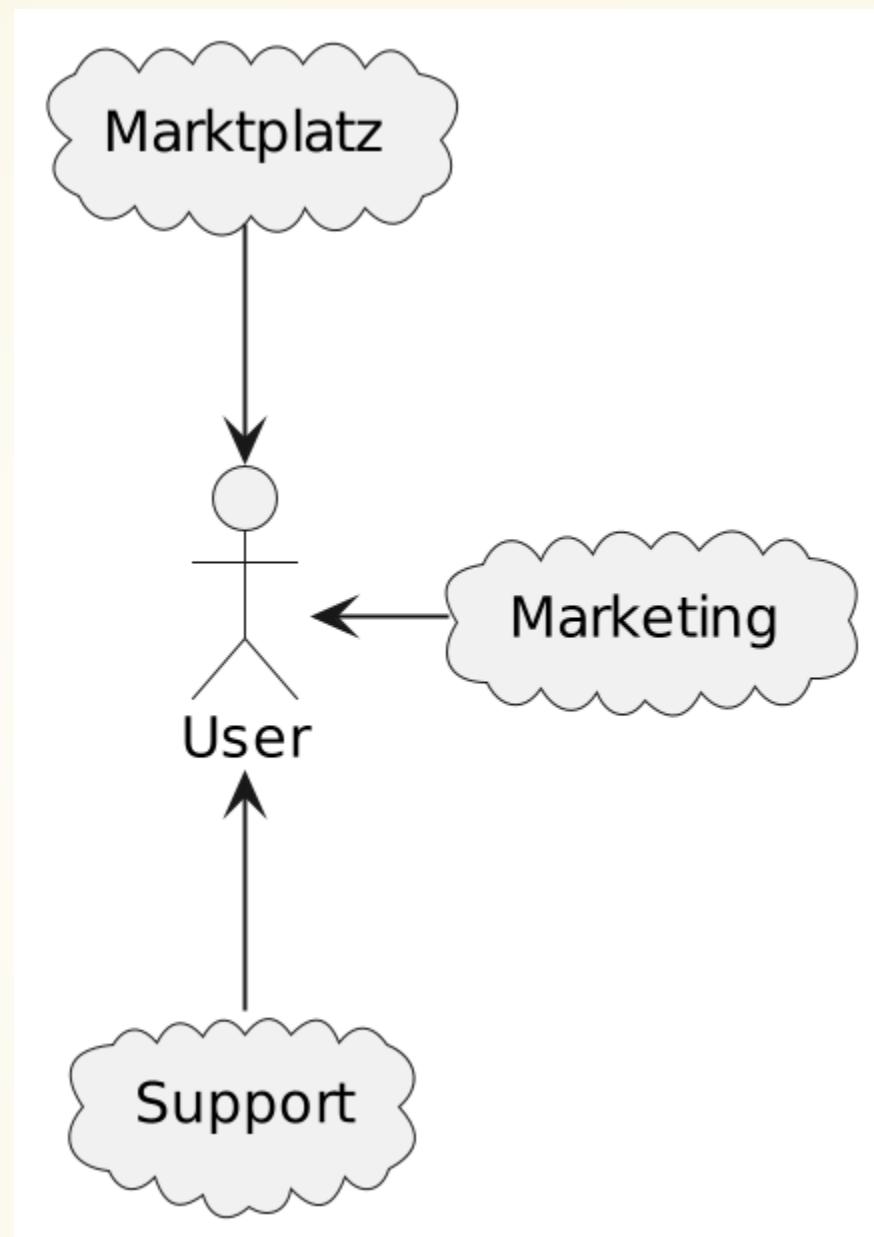
# BOUNDED CONTEXT



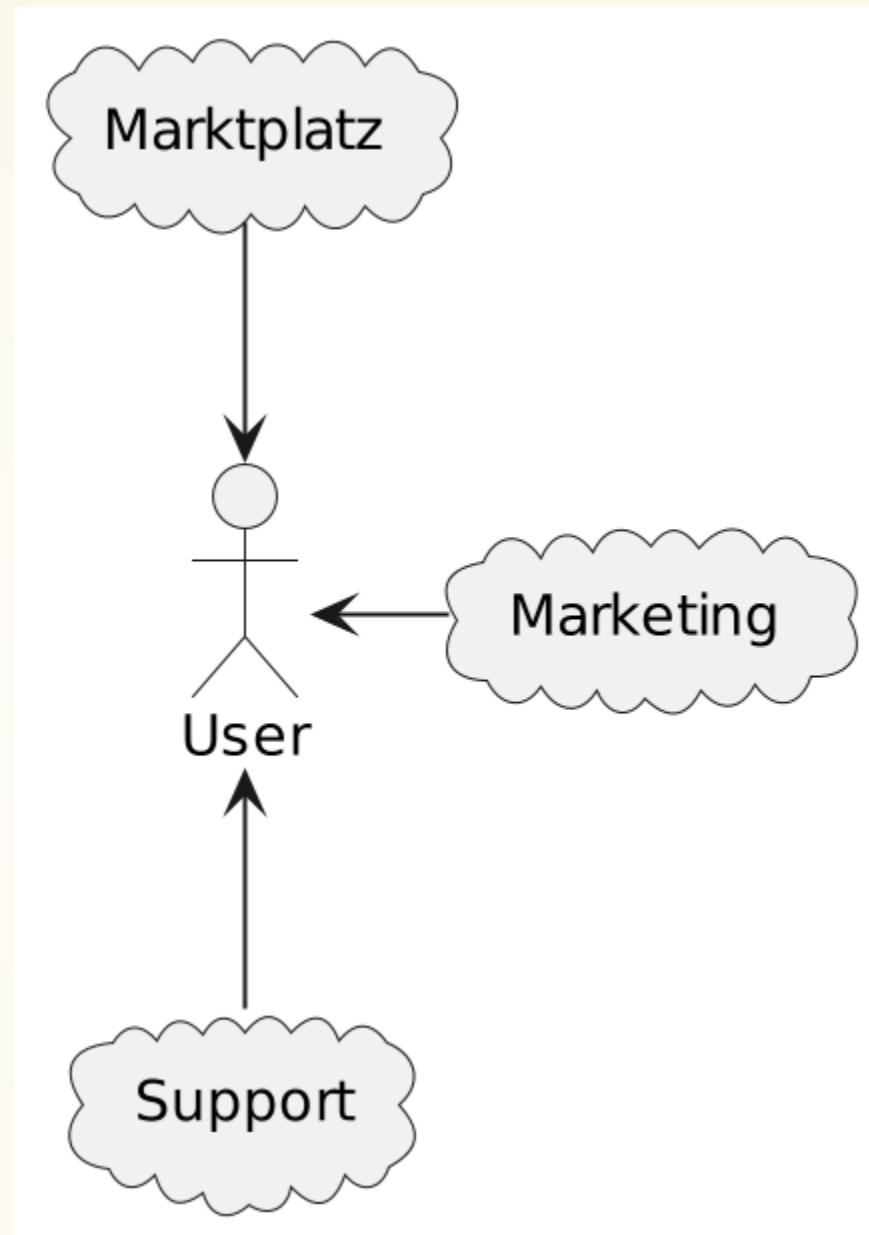
<https://martinfowler.com/bliki/BoundedContext.html#footnote-quote>

# **BOUNDED CONTEXT**

# MEHRDEUTIGER KONTEXT

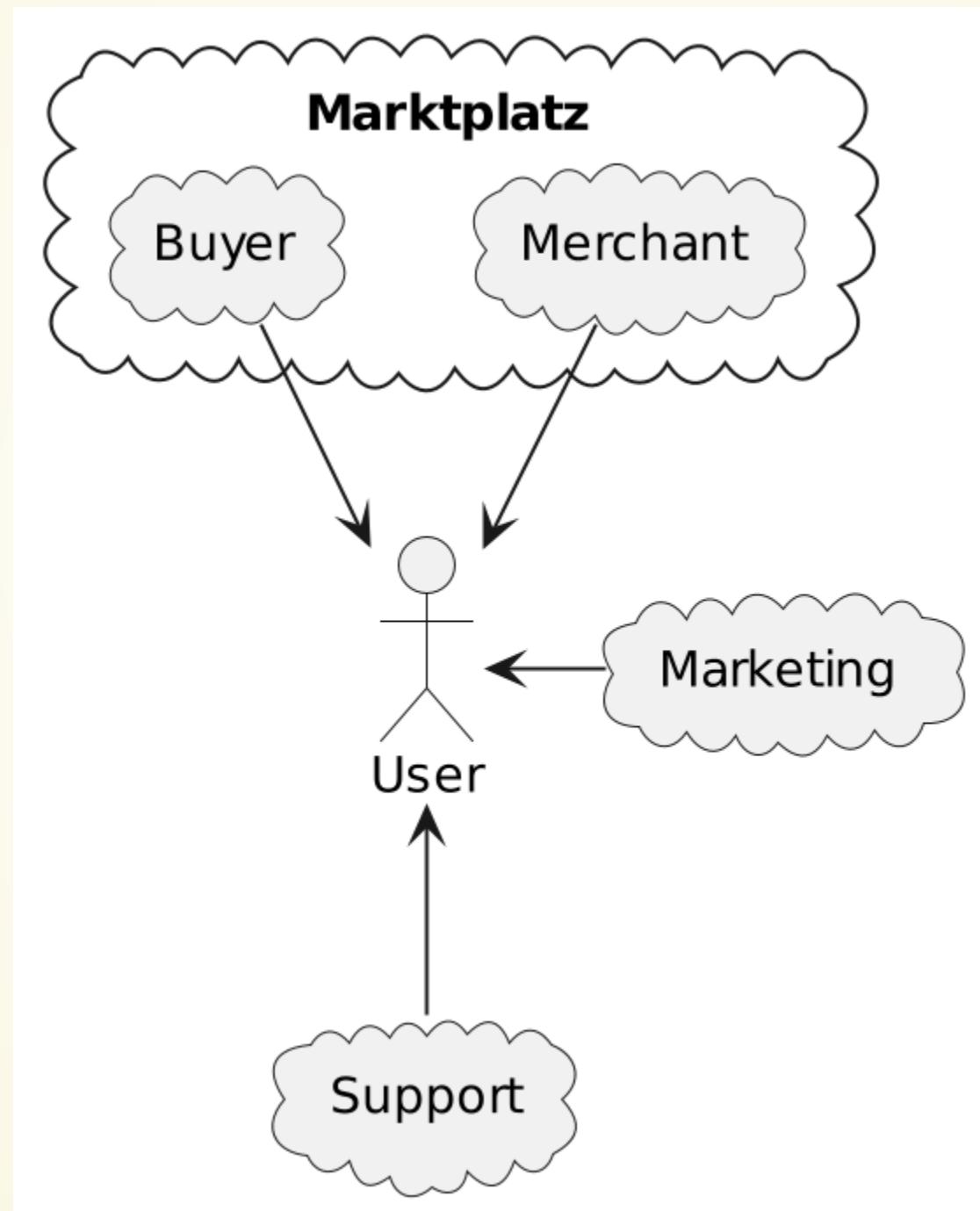


# MEHRDEUTIGER KONTEXT

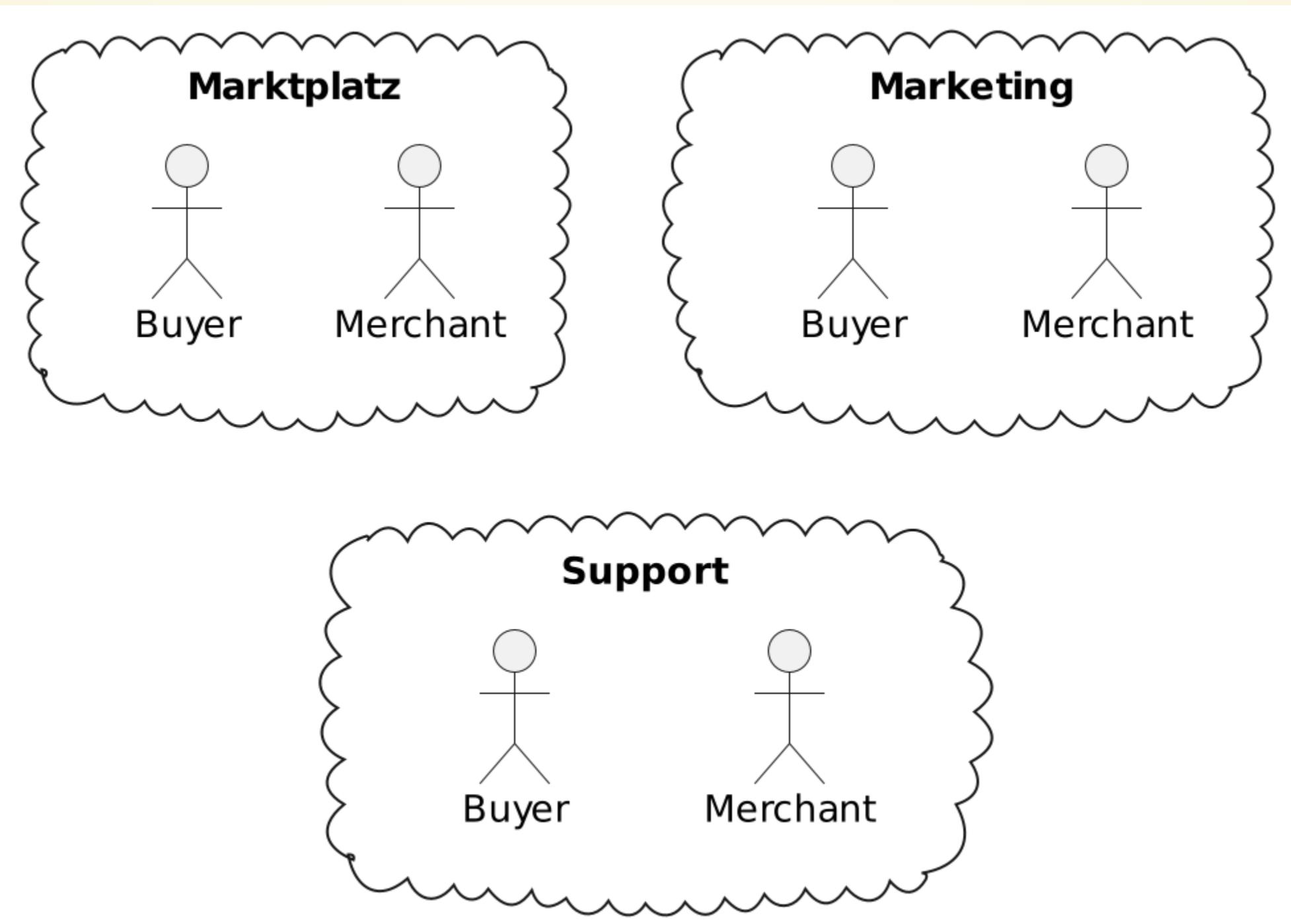


Der Marktplatz muss auch wieder unterscheiden:

# MEHRDEUTIGER KONTEXT



# EINDEUTIGER KONTEXT



# BOUNDED CONTEXT UND PROBLEMDOMÄNE

- idealerweise ist jeder Bounded Context genau einer Subdomäne zugeordnet
- innerhalb einer Subdomäne können aber mehrere Kontexte existieren
- ist das nicht der Fall → möglicherweise ein unscharfes Modell, das Fachlichkeiten vermischt
- um diese Zuordnung zu visualisieren, führt DDD die *Context Map* ein

# CONTEXT MAP

- eine Context Map bildet die existierenden Kontexte innerhalb der Subdomänen einer Problemdomäne ab
- Ziel
- Beziehungen zwischen Kontexten visualisieren und deren Art bestimmen
- die Grenze verdeutlichen
- Bewusstsein über andere Kontexte schaffen

# BEZIEHUNGEN ZWISCHEN DEN KONTEXTEN

- eine Beziehung zwischen zwei Kontexten bedeutet sowohl eine technische als auch eine organisatorische Abhangigkeit
- technisch: Kontext A muss Informationen mit Kontext B austauschen
- organisatorisch: das Team, das Kontext A entwickelt, muss mit dem Team kommunizieren, welches Kontext B entwickelt
- um diese Beziehungen zu beschreiben, definiert DDD mehrere Muster

# BEZIEHUNGSMUSTER ZWISCHEN BOUNDED CONTEXTS

- Anticorruption Layer
- Shared Kernel
- Open Host Service
- Customer-Supplier
- Conformist

# ANTI-CORRUPTION LAYER

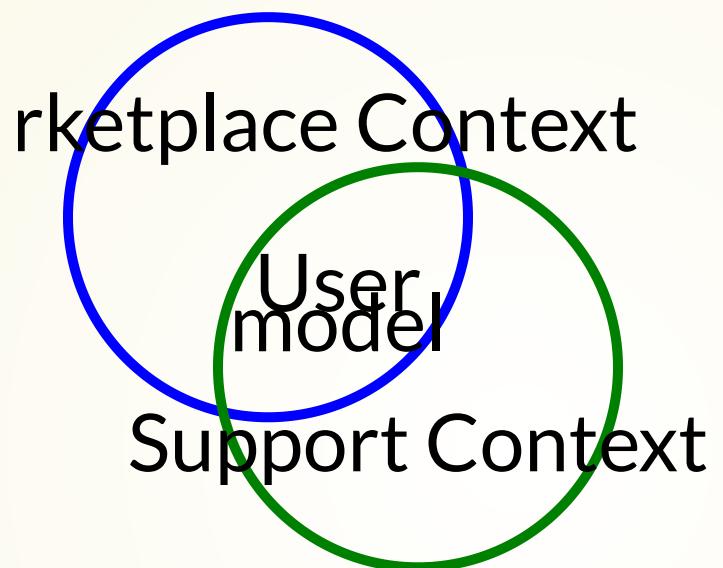
- Modelle müssen häufig Daten mit anderen Modellen tauschen
  - z.B. Import von Kundendaten aus einer CSV-Datei
- ein *anti-corruption layer* isoliert das eigene Modell von Fremdeinflüssen
- Übersetzungsschicht ähnlich dem Adapter-Entwurfsmuster
- enthält keine Geschäftslogik, nur Übersetzungslogik
- schützt vor Änderungen durch das Fremdmodell

# **ANTI-CORRUPTION LAYER**

# SHARED KERNEL

- manchmal modellieren verschiedene Kontexte/Subdomänen der gleichen Problemdomäne sehr ähnliche Konzepte und Regeln
- Teams der Kontexte können sich auf ein gemeinsam verwaltetes Teilmодell einigen ⇒ „shared kernel“
  - sollte gut überlegt sein, auch wenn der initiale Aufwand geringer ist
- bedeutet, dass die unterschiedlichen Teams sehr eng zusammenarbeiten müssen (starke gegenseitige Abhängigkeit)
  - ⇒ geteiltes Modell sollte so klein wie möglich sein

# SHARED KERNEL

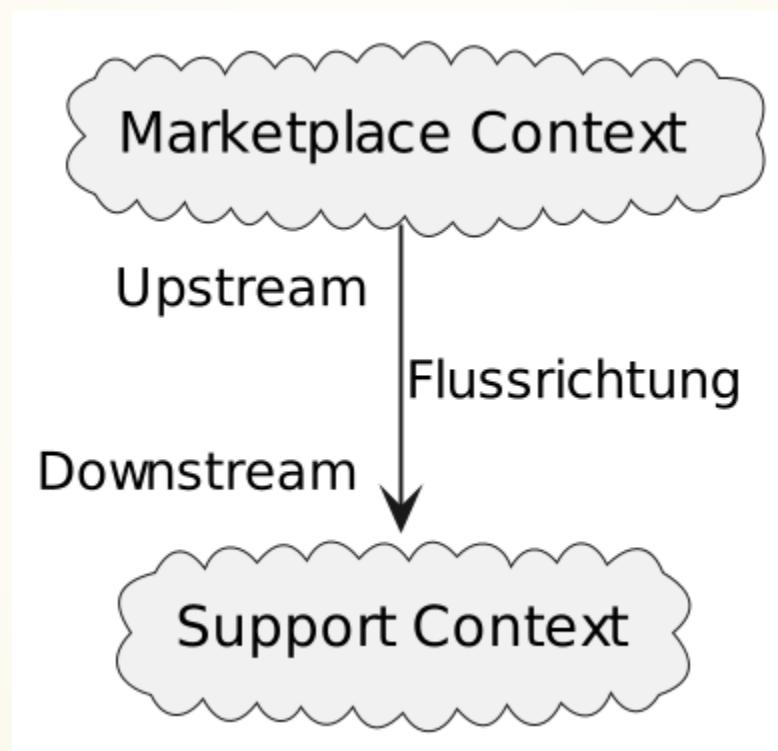


- das *user model* wird von beiden Kontexten genutzt

# CUSTOMER/SUPPLIER

- ein Modell (*Customer*) ist stark von einem anderen Modell (*Supplier*) abhängig
  - ⇒ wenn sich das Modell des Suppliers ändert, muss sich das Modell des Customers ändern - aber nicht umgekehrt
- enge Zusammenarbeit der Teams der Kontexte ist notwendig, aber nur in eine Richtung
  - z.B. sollte das Team des Customer-Modells bei Sitzungen des Supplier-Modells anwesend sein
- Tests sollten sicherstellen/unterstützen, ob Änderungen im Upstream Änderungen im Downstream zur Folge haben

# CUSTOMER/SUPPLIER



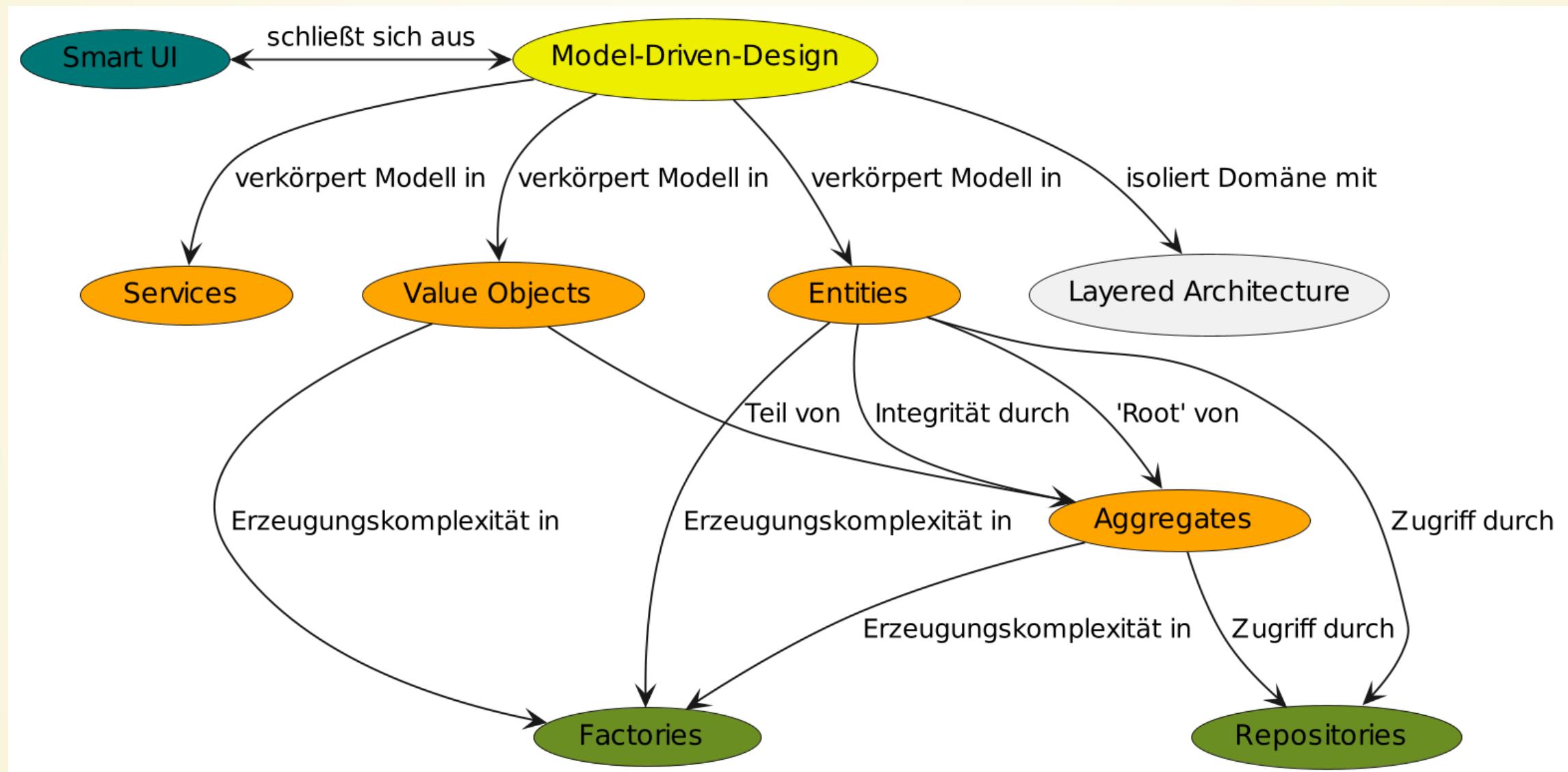
# CONFORMIST

- Modell A passt sich vollständig an die Schnittstelle von Modell B an  
(und akzeptiert Korruption des eigenen Modells)
- nur anwenden wenn
  - keine Team-Kollaboration mit Team B möglich ist *und*
  - die Implementierung eines ACLs zu teuer/aufwendig ist
- "Friss oder stirb"

# TAKTISCHES DDD

- Sinn und Zweck von DDD: Entwurf eines Modells, das komplexe Regeln und Sachverhalte (Invarianten) abbildet
- *taktisches DDD* definiert Entwurfsmuster und Vorgehensweisen für den Entwurf dieses Modells

# ÜBERSICHT GRUNDBAUSTEINE



# ÜBERSICHT GRUNDBAUSTEINE

- Entities, Value Objects, Domain Services, Aggregates
  - Hauptkomponenten des Modells
  - enthalten den Großteil der Geschäftslogik
  - forcieren die in der Domäne geltenden Invarianten und Regeln und machen diese sichtbar
- Repositories, Factories
  - Kapseln die Logik für das Persistieren und Erzeugen von Entities, Value Objects und Aggregates
  - halten das Modell frei von *accidental complexity*
  - eher im Bereich der *Pure Fabrication* (d.h. auf der technischen Seite der Anwendung)

# SMART UI

- viele Anwendungen sind nicht komplex genug, um von einem DDD-Modell zu profitieren
- CRUD – Create, Read, Update, Delete
- Datenbank- und Spreadsheet-Oberflächen
- solche Anwendungen nennt man „Smart UI“
- das ist kein abwertender Begriff
- diese Anwendungen sollten nicht mit einem zusätzlichen Modell künstlich komplexer gemacht werden
- Modelle müssen einen konkreten Nutzen haben

# EXKURS: ANEMIC DOMAIN MODEL

*blutleeres Domänenmodell*

- Antipattern, bei dem Domänobjekte kaum bzw. keine Geschäftslogik enthalten
- d.h. kaum/keine Validierungen, Berechnungen, Invarianten, ...
- Geschäftslogik ist in der Architektur des Programms selbst
- Refactorings und Wartung werden deutlich aufwändiger

# GRUNDBAUSTEIN: VALUE OBJECTS

- Value Objects (VO) sind einfach Objekte **ohne eigene Identität**
- VOimmutable)
- VO
- Zwei VO

# BEISPIEL VALUE OBJECT: GEWICHT

- Jedes Gewicht besteht aus einer Zahl und einer Einheit
- beispielsweise 13 Kilogramm
- Was ist schwerer? 1 Kilogramm Eisen oder 1 Kilogramm Federn?
- Zwei Gewichte sind gleich, wenn ihre Zahlen und Einheiten übereinstimmen (evtl. umrechnen)
- 13 Kilogramm waren schon immer 13 Kilogramm und werden es auch immer sein
- Kein erkennbarer Lebenszyklus

# VORTEILE VON UNVERÄNDERLICHKEIT

- gültig erzeugte Objekte bleiben gültig
- frei von Seiteneffekten
- Einhalten von Regeln der Domäne wird einfach(er)
- sehr leicht zu testen
- Gleichheit ist entweder immer oder nie gegeben

# VALUE OBJECTS ERKENNEN

- VO<sub>s</sub> beschreiben oder messen eine Sache
- Beispiele
  - Geldbetrag (Betrag + Währung)
  - E-Mail-Adresse
  - Gewicht (Betrag + Einheit)
  - Temperatur (Betrag + Einheit)
  - Farbe
  - Telefonnummern
- VO<sub>s</sub> können Aggregate anderer Objekte sein (häufig wieder VO<sub>s</sub>)
  - Geldbetrag: Betrag + Währung
  - Adresse: Straße, Hausnummer, PLZ und Ort

# IMPLEMENTIERUNG VON VOS IN JAVA

- Klasse ist *final* (keine Ableitungen zulassen)
- alle Felder (Properties) sind (*blank*) *final*
- *equals()* und *hashCode()* sind überschrieben
- keine Methoden, die den Zustand mutieren
- Methoden mit Rückgabewert liefern nur
- unveränderliche Typen oder
- defensive Kopien

# VALUE OBJECT: CODEBEISPIEL

```
class Money {  
  
    private final Currency currency;  
    private final int amount;  
  
    public Money(Currency currency, int amount) {  
        this.currency = currency;  
        this.amount = amount;  
    }  
  
    public Money add(int amount) {  
        return new Money(currency, this.amount + amount);  
    }  
  
    @Override  
    public boolean equals(Object object) {/* [...] */}  
  
    @Override  
    public int hashCode() {/* [...] */}  
  
    enum Currency {Dollar, Euro}  
}
```

# GRUNDBAUSTEIN: ENTITIES

- Objekte, die eine Identität haben
- sie existieren *kontinuierlich* (innerhalb ihres Lebenszyklus)
- definieren sich **nicht** durch ihre Werte
- Werte sind über die Zeit veränderlich
- besitzen einen Lebenszyklus
- zwei Entities sind verschieben, wenn sie unterschiedliche Identitäten haben ⇒ selbst wenn die Werte gleich sind
- zwei Entities sind gleich, wenn sie die gleiche Identität haben ⇒ selbst wenn die Werte unterschiedlich sind

# ENTITIES: BEISPIEL

- jede Person hat eine eigene Identität
- jede Person hat einen eignen Namen
- die Identität ist unabhängig vom Namen
- jeder Person hat ein eigenes Gewicht
- welches sich mit der Zeit ändern kann, ohne, dass die Person ihre Identität verliert
- zwei Personen mit gleichem Namen und gleichem Gewicht sind unterschiedliche Personen
- jeder Person hat einen eigenen Lebenszyklus

# GRUNDBAUSTEIN: ENTITIES

- Entities bilden identifizierbare und identitätstragende *Dinge* aus der Domäne ab
- Regeln (Invarianten) der Domäne müssen forciert werden
- der Konstruktor erzeugt nur gültige Instanzen
- spätere Mutationen dürfen keinen ungültigen Zustand erzeugen
- *gültiger Zustand* wird von der Domäne bestimmt

# IDENTITÄT VON ENTITIES

- grundsätzliche Unterscheidung
- natürlicher Schlüssel (aus der Domäne)
- Surrogatschlüssel (künstlich erzeugt)
- Identität einer Entity in der Domäne ist **nicht** die Objektidentität (Speicheradresse) in der Programmiersprache
- insbesondere bei natürlichen Schlüsseln ist häufig die Verwendung von `equals()` und `hashCode()` als Domänenidentität schwierig
- zwei Entities können sogar die gleiche Identität (in der Domäne) haben, obwohl sie Instanzen unterschiedlicher Typen/Klassen sind
- die Identität ist *oft* in der Domäne wichtig - sie ist *immer* für die Entwicklung wichtig

# ENTITIES: NATÜRLICHE SCHLÜSSEL

- Fahrzeug-Identifikationsnummer (FIN)
- Reisepassnummer
- Kursname
- ISBN
- ...
- *natürliche* Schlüssel kommen aus der Domäne

# ENTITIES: NATÜRLICHE SCHLÜSSEL

- (+) sehr aussagekräftig
- (+) keine Gefahr von Duplikaten *falls* global eindeutig
- (-) fremdbestimmt (ändert sich die Reisepassnummer wirklich nicht?)
- (-) ggf. nicht global eindeutig, sondern kontextabhängig (Kurs TINF12B3 existiert an der DHBW KA - evtl. auch an der DHBW Stuttgart?)

# ENTITIES: SURROGATSCHLÜSSEL

- von der Applikation generierte IDs
- eigenes (String-)Format basierend auf Entity-Eigenschaften
- inkrementierender Zähler
- UUID: Universally Unique Identifier (auch GUID)
- 128-Bit-Zahl
- (für Version 4) Wahrscheinlichkeit einer Kollision bei 103 Billionen UUIDS: 1 zu 1 Mrd.
- Vergleich: Wahrscheinlichkeit für Lotto-Jackpot: 1 zu 140 Mio.

# VOS VS. ENTITIES

## Value Object

- keine Identität (in der Domäne)
- unveränderlich (immutable)
- kein Lebenszyklus
- verschieden bei verschiedenen Eigenschaften
- gibt Werten in der Domäne eine Semantik

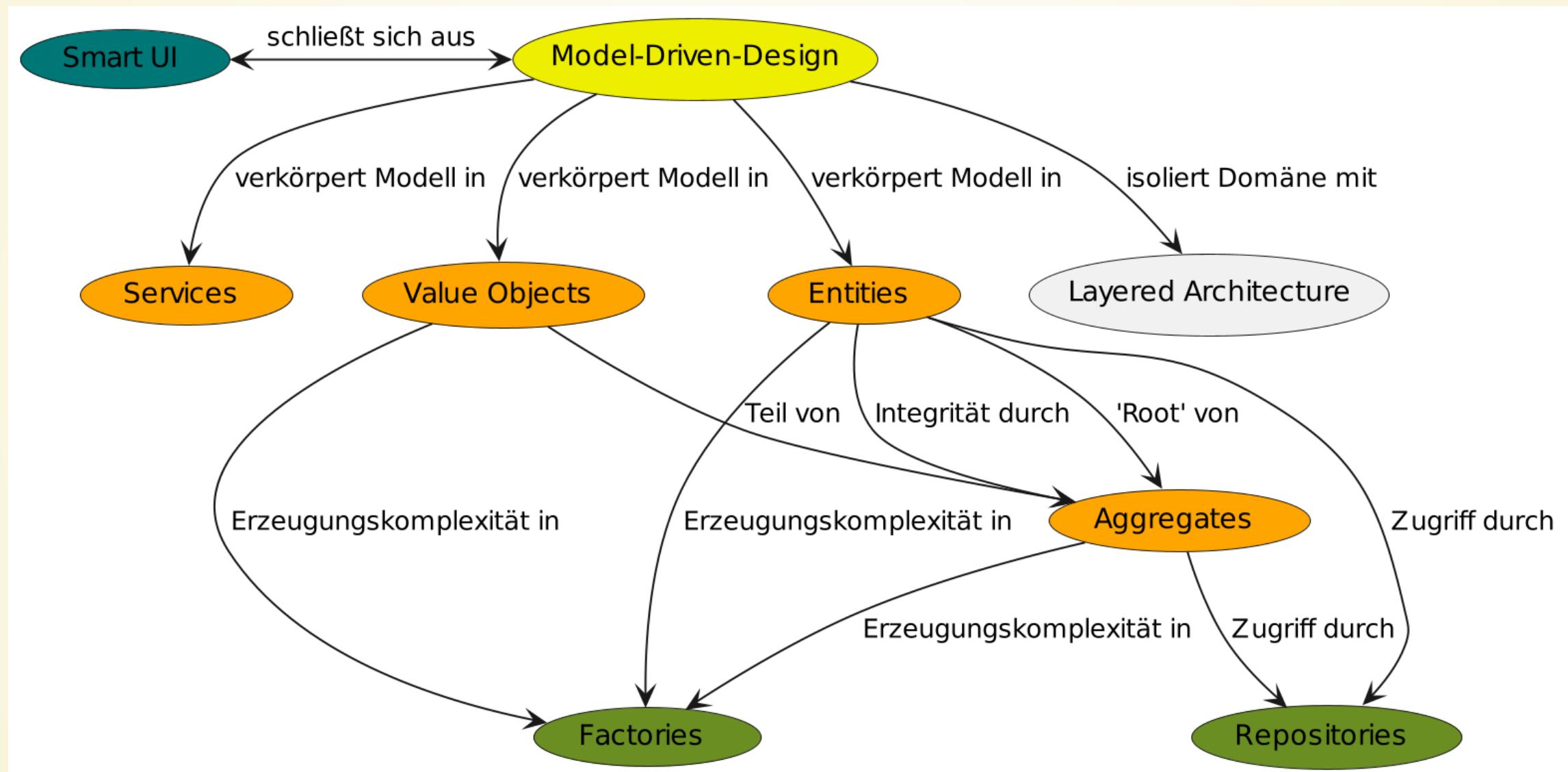
## Entity

- eindeutige Identität in der Domäne
- hat veränderliche Eigenschaften
- eigener Lebenszyklus
- verschieden bei verschiedenen Identitäten
- repräsentiert ein Ding in der Domäne

# ZUSAMMENFASSUNG

- es gibt viele Anwendungen, die nicht komplex genug sind, um DDD zu benötigen oder zu rechtfertigen
- diese Anwendungen nennt man Smart UI- Anwendungen
- ein DDD-Modell besteht grundlegend aus
- Value Objects - uveränderlichen Werten ohne Lebenszyklus oder Identität
- Entities - veränderliche Dinge mit Identität und individuellen Lebenszyklus

# GRUNDBAUSTEINE



# DOMAIN SERVICES

- generelles DDD Problem: Begriffe sind vielfach überladen und/oder umgedeutet
- ein DDD Service ist weder ein Service
- aus der *Serviceorientierten Architektur (SOA)* noch
- im Sinne der Microservices
- daher besser von *Domain Service* sprechen

# DOMAIN SERVICES: ZWECK

## 2 Anwendungsfälle

- beinhaltet komplexes Verhalten, Vorgänge oder Regeln der Domäne, die sich nicht einer bestimmten Entity oder einem VO zuordnen lassen
- in der *Clean Architecture*: Use Case
- definiert einen Funktionsvertrag für einen externen Dienst, damit das Domänenmodell frei von technischer Komplexität bleibt
- in der *Clean Architecture*: Interface für Adapter

# DOMAIN SERVICES: BEISPIEL KOMPLEXER SACHVERHALT

- Berechnung, ob ein Gerät mit Strom versorgt wird
- benötigt das Gerät
- benötigt den angeschlossenen Stromkreis
- benötigt die Sicherung des Stromkreises
- benötigt die Hauptsicherung

# DOMAIN SERVICES: BEISPIEL KOMPLEXER SACHVERHALT

```
class PowerCalculator {  
  
    private CircuitRepository circuitRepository;  
    private FuseRepository fuseRepository;  
    private HouseRepository houseRepository;  
  
    public boolean hasPower(Equipment equipment) {  
        Circuit circuit = circuitRepository.getById(equipment.getCircuitId());  
        Fuse circuitFuse = fuseRepository.getById(circuit.getFuse());  
        House house = houseRepository.getById(equipment.getHouseId());  
        Fuse mainFuse = fuseRepository.getById(house.getMainFuseId());  
        return mainFuse.isClosed && circuitFuse.isClosed();  
    }  
}
```

# DOMAIN SERVICES: BEISPIEL VERTRAG

- die Domäne kann zur Erfüllung der Anforderungen auf externe Unterstützung angewiesen sein, z.B. durch einen Webservice, der von einer Fremdanwendung bereitgestellt wird
- Beispiel: Kreditwürdigkeit
- benötigt den *Customer*
- benötigt eine Überprüfung durch die Schufa

# DOMAIN SERVICES: BEISPIEL VERTRAG

```
interface CreditRatingCheck {  
    CreditRating creditRating(Customer customer);  
}  
  
class SchufaCreditRatingCheck implements CreditRatingCheck {  
  
    @Override  
    public CreditRating creditRating(Customer customer) {  
        // check via HTTP call  
        return creditRating;  
    }  
}
```

# DOMAIN SERVICES: DEFINITION EINES VERTRAGS

- durch die Definition eines Vertrages kann das Domänenmodell vorgeben, was für ein Ergebnis erwartet wird (*CreditRating*) und welche Daten es zur Erfüllung des Vertrages bereitstellt (*Customer*)
- fachlich liegt damit alles relevante innerhalb des Domänenmodells - lediglich die technischen Details werden in einer anderen Schicht implementiert
- in der hexagonalen Architektur in der *Infrastrukturschicht*

# DOMAIN SERVICES: EIGENSCHAFTEN

- ein Domain Service bezieht sich auf ein Domänenkonzept, das nicht natürlicherweise Teil einer Entity oder eines Value Object ist
- die Schnittstelle (die Methodensignatur) verwendet die Begriffe des Domänenmodells
- Ein-/Ausgabeparameter sind Entities und VOs
- der Domain Service ist zustandlos
- jede konkrete Instanz des Domain Service kann verwendet werden
- er darf aber (global sichtbare) Seiteneffekte haben

# **GRUNDBAUSTEINE**

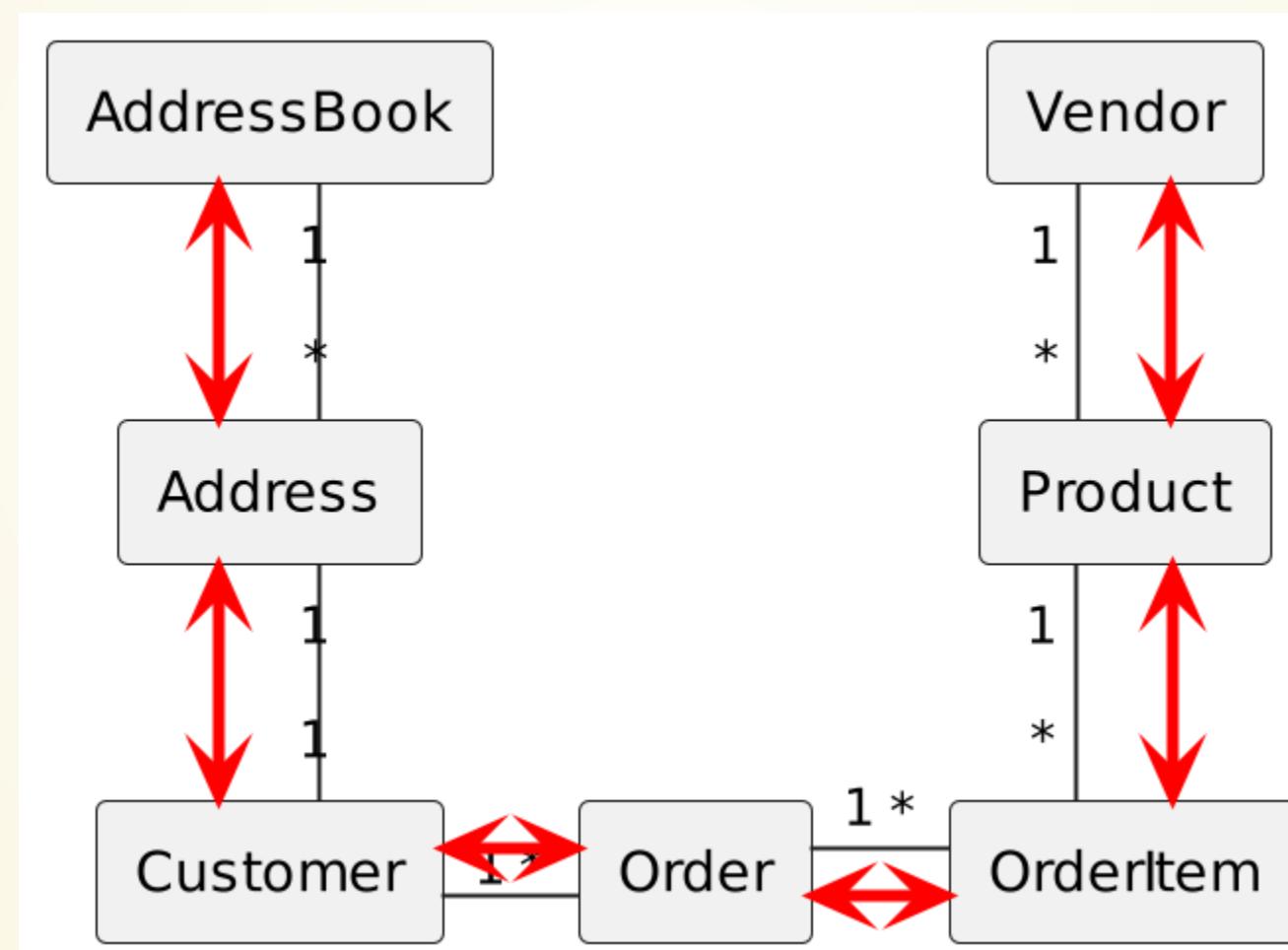
include::diagramm\_übersicht.adoc

# AGGREGATES

- wenn die Domäne maßstabsgetreu modelliert wird, findet man viele Entities und VOs, die große Objektgraphen mit oft bidirektionalen Abhängigkeiten bilden
- das wird schnell ungemütlich
- Wahrscheinlichkeit nicht eingehaltener Regeln steigt
- verstärkt Kollisionen beim gleichzeitigen Bearbeiten
- Performance-Einbußen durch Warten auf Sperrenfreigabe
- lange Wartezeiten beim Laden und Speichern

# AGGREGATE: PROBLEMSTELLUNG

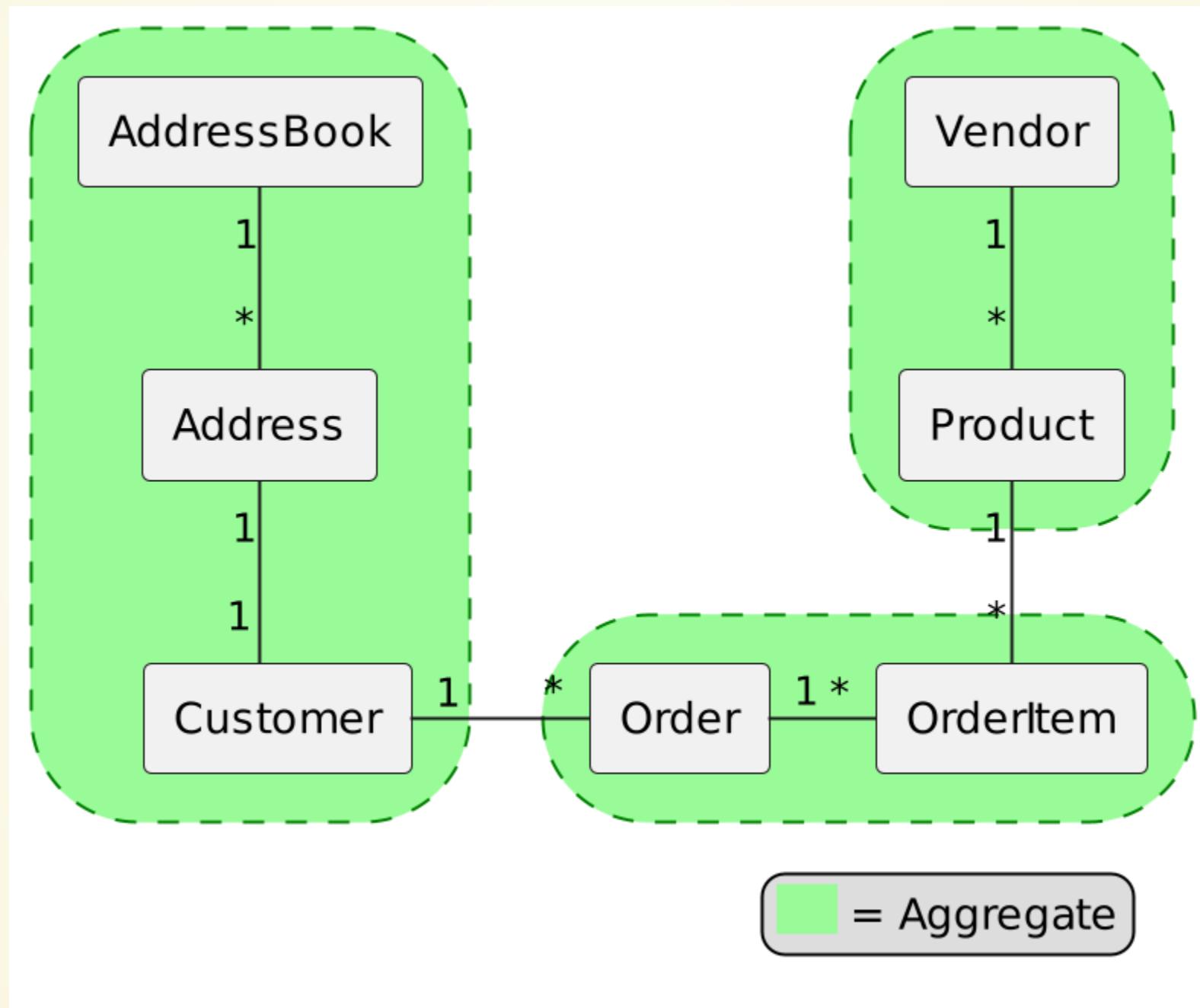
Ausschnitt eines Domänenmodells



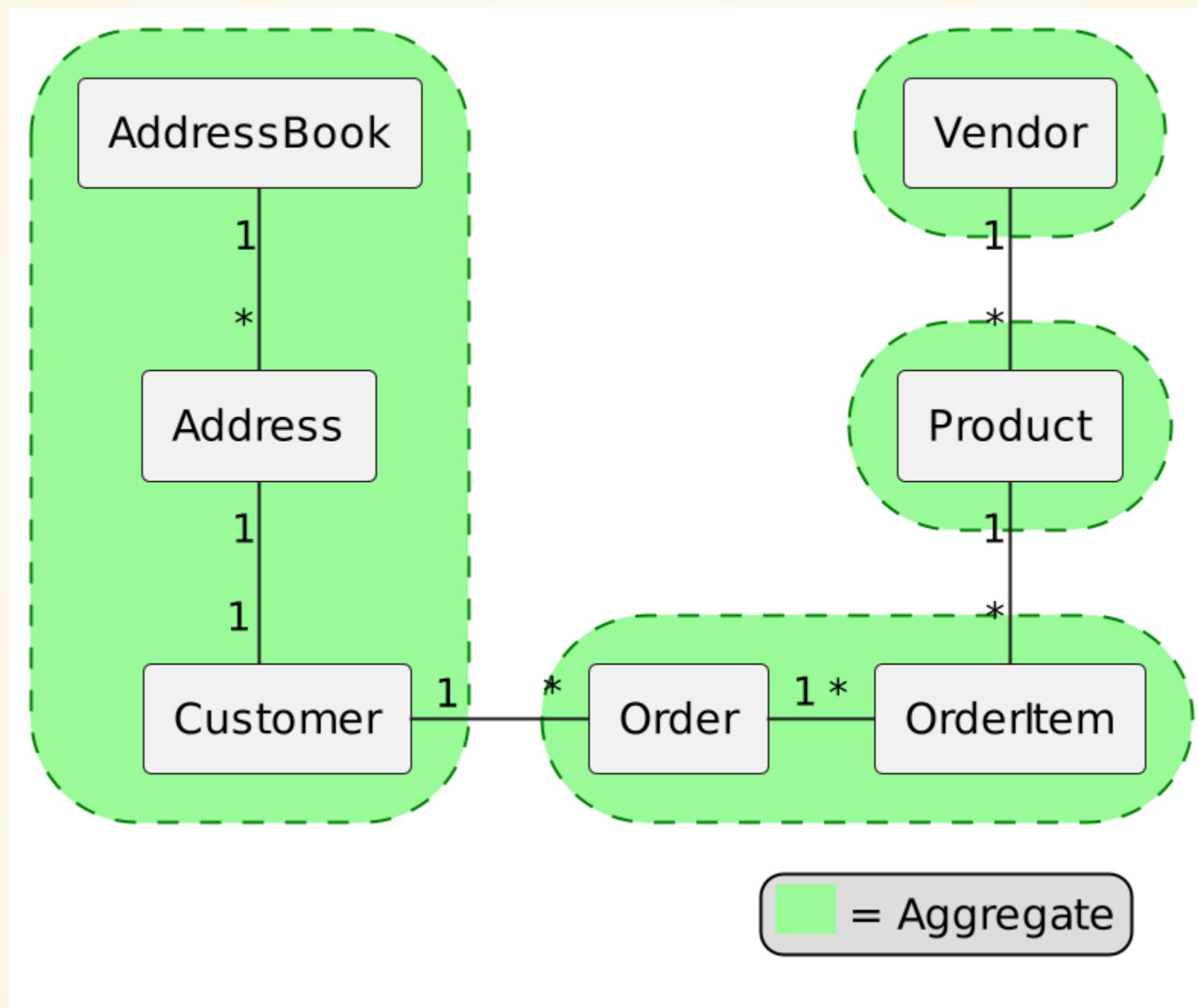
# AGGREGATE ZUR REDUKTION

- Aggregate gruppieren die Entities und VOs zu gemeinsam verwalteten Einheiten
- jede Entity gehört zu einem Aggregat – selbst wenn das Aggregat nur aus dieser Entity besteht
- Aggregate reduzieren die Komplexität der Beziehungen zwischen den Objekten
- das Aggregat wird immer als Einheit betrachtet und verwaltet (geladen und gespeichert)
- es gibt klare Regeln, wie außenstehende Objekte mit dem Aggregat interagieren darf

# AGGREGATE: BEISPIEL V1

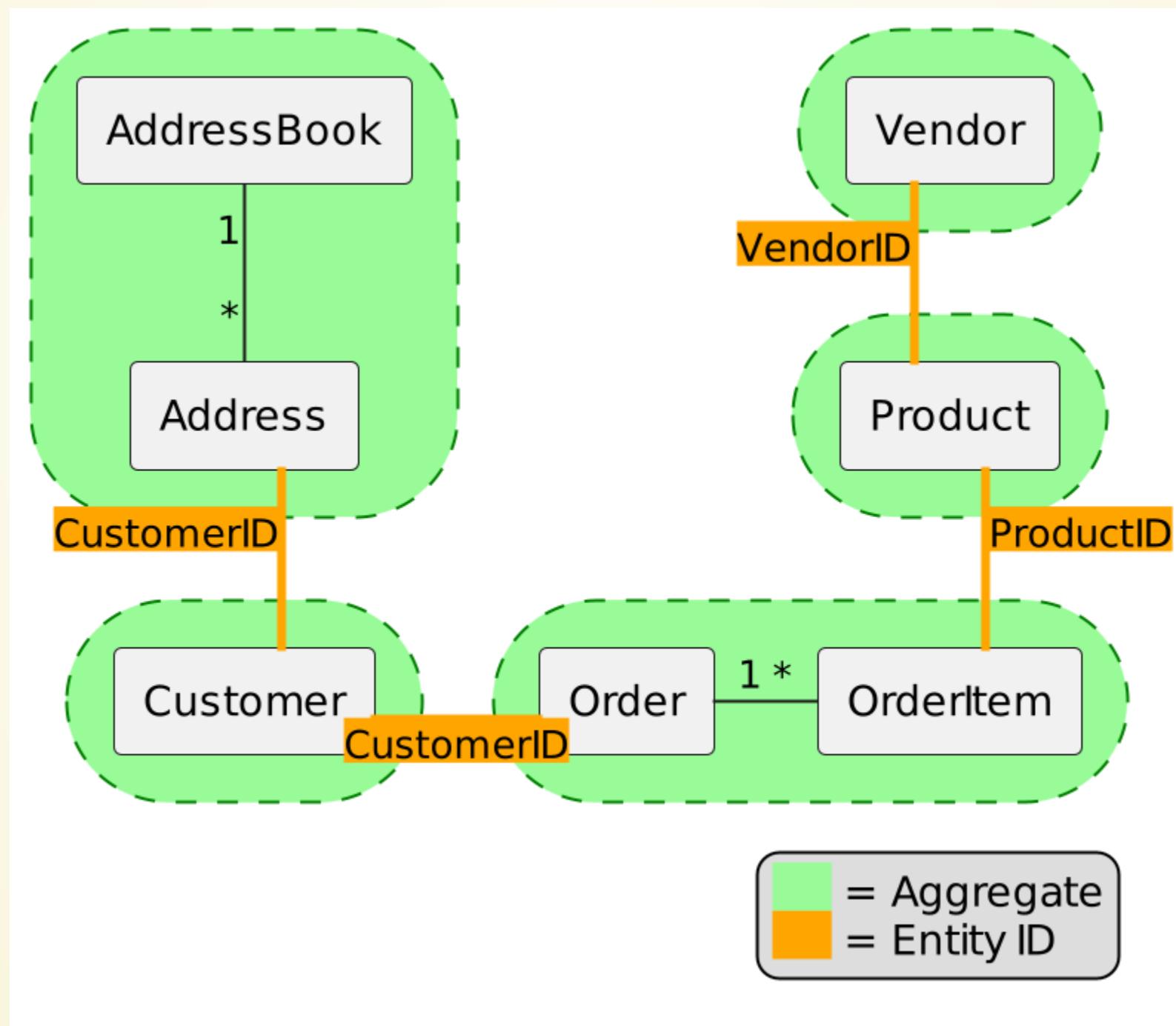


# AGGREGATE: BEISPIEL V2



# AGGREGATE: BEISPIEL V3

Assoziationen lösen: direkte Assoziation durch indirekte Assoziation anhand der Identität der wichtigsten Entity im Aggregat

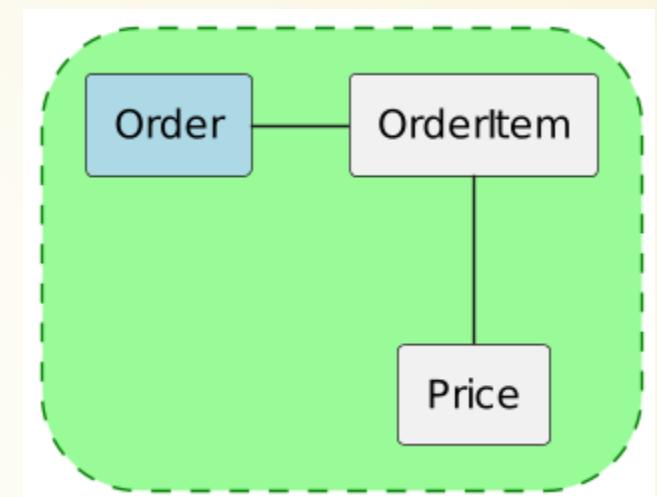


# AGGREGATES: ROOT ENTITY

- in jedem Aggregat übernimmt eine Entity die Rolle der Aggregate Root Entity bzw. des Aggregat Root (AR)
- alle Zugriffe auf das Aggregat müssen über das AR erfolgen
- auch Zugriffe auf die inneren Elemente des Aggregates
- langfristige direkte Referenzen auf innere Elemente sind nicht erlaubt
- nur temporäre Referenzen während einer Berechnung

# AGGREGATE ROOT (AR)

- nur ein AR pro Aggregat
- AR kann als eine Art *Türsteher* alle Zugriffe auf das Aggregat kontrollieren
- zentrale Stelle zur Überwachung der Domäneregeln
- Beispiel: Versandkostenfreiheit ab 100€ Bestellwert
- interne Angelegenheiten bleiben im Aggregat (*Information Hiding*)



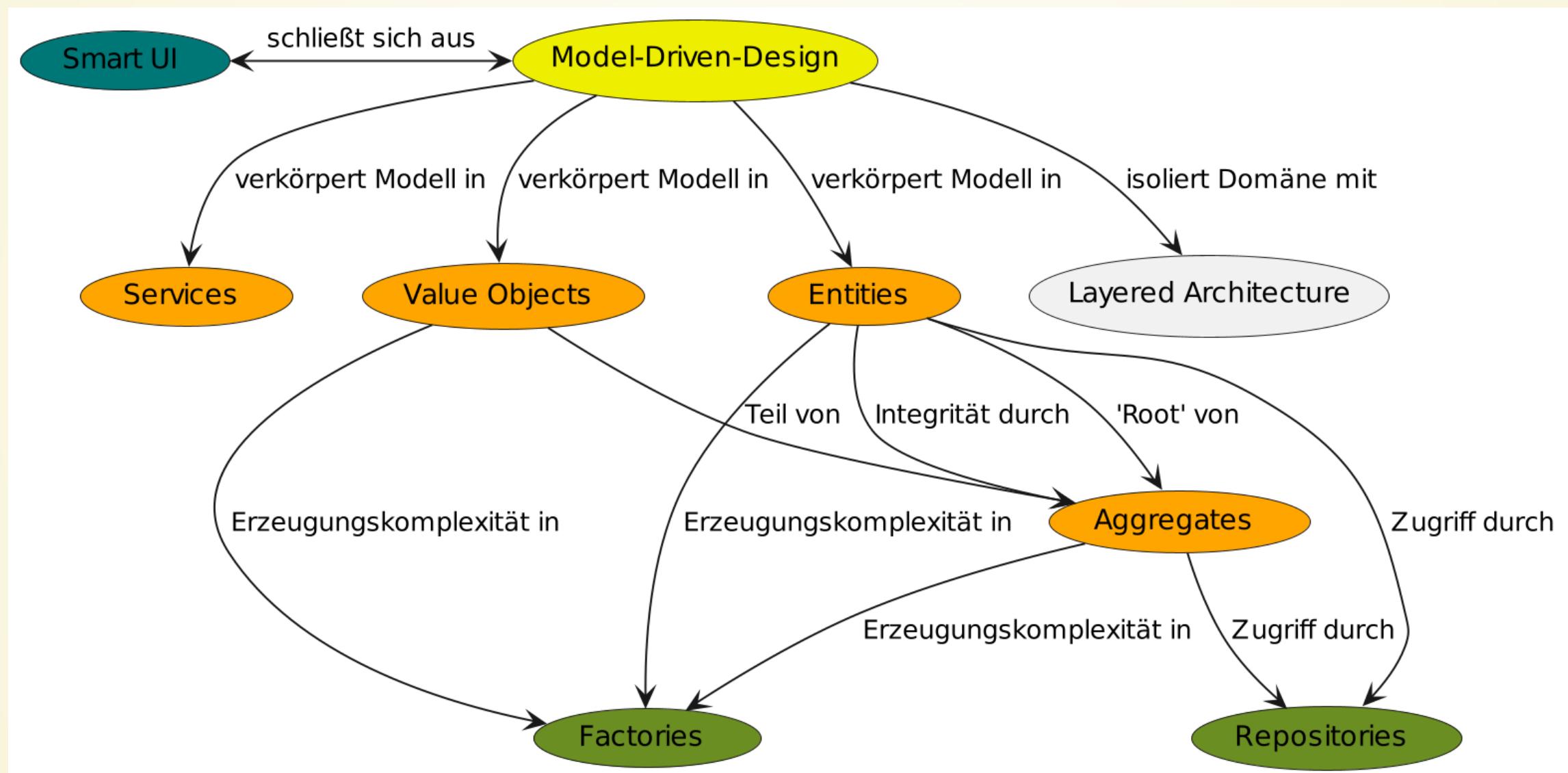
# AGGREGATE UND AUSSENWELT

- Zugriff auf Aggregat nur über Aggregate Root
- wenn das AR Referenzen auf innere Objekte herausgeben muss, sollten das immer defensive Kopien oder Immutable-Dekorierer sein
- das Aggregat sorgt dafür, dass sein Zustand immer den Domänenregeln entspricht
- alle Änderungen gehen über den AR und sind daher bekannt
- wenn die Außenwelt den AR „vergisst“, ist das gesamte Aggregat nicht mehr erreichbar

# AGGREGATES: ZUSAMMENFASSUNG

- Aggregate sind Zusammenfassungen von Entities und Value Objects
- jedes Aggregat bildet eine eigene Einheit (auch für Create, Read, Update, Delete – CRUD)
- wird immer vollständig geladen und gespeichert
- Aggregates
- entkoppeln die Objektbeziehungen
- bilden natürliche Transaktionsgrenzen
- sichern Domänenregeln zu
- **Aggregate sind mächtig, aber auch schwierig**

# (DOMAIN) REPOSITORIES



# (DOMAIN) REPOSITORIES

- Repositories sind die „Vorratschränke“ des Systems
- Repositories bieten dem Anwendungscode einfachen Zugriff auf (persistente) Speicher
  - ohne dass die Anwendung wissen muss/kann, wie genau die Speicherung geschieht
  - „Speichern von Daten“ ist ein technischer Vorgang
- trennt den Code der Domäne von den technischen Details der Speicherung
  - Domain Code vs. Pure Fabrication

# (DOMAIN) REPOSITORIES

- Repositories arbeiten direkt mit Aggregates zusammen
  - meist gibt es für jedes Aggregat ein Repository
  - Repositories liefern immer die Aggregate Roots (und damit den Zugriff auf den Rest) zurück
- Definition der Repositories ist Teil des Domain Code
  - Implementierung findet „außerhalb“ statt
  - vergleichbar Kern und Peripherie der *Hexagonalen Architektur* bzw. *Clean Architecture*

# (DOMAIN) REPOSITORIES

- die Methoden des Repository-Interface werden in der Sprache der Domäne benannt
- der Klassename kann „Pure Fabrication“ sein

```
public interface BuchRepository {  
    void lagere(Buch neuesBuch);  
    Optional<Buch> findeFür(ISBN isbn);  
    Optional<Buch> findeÜber(Buchtitel titel);  
    Iterable<Buch> findeVon(Autor autor);  
    //void entferne(Buch altesBuch); ← Die Bibliothek behält alles!  
}
```

# (DOMAIN) REPOSITORIES

- der Rest der Anwendung muss eine bestimmte Aggregate Root anhand ihrer Eigenschaften finden können
- diese Abfragen sind die wichtigste Aufgabe des Repositories

```
public interface BuchRepository {  
    Optional<Buch> findeFür(ISBN isbn);  
    Optional<Buch> findeÜber(Buchtitel titel);  
    Iterable<Buch> findeVon(Autor autor);  
    Iterable<Buch> findeAlleIn(Erscheinungsjahr jahr);  
    Iterable<Buch> findeAllePassendZu(Kriterium... kriterien);  
}
```

# (DOMAIN) REPOSITORIES

- die Abfragen sollten genau zu den Aufgaben der Domäne passen
  - selbst wenn es eine allgemeine Abfrage (über Kriterien) gibt, lohnen sich die speziellen Methoden
- gerne auch Abfragen, die nur Metadaten zurückgeben
  - sind in der Speicherschicht effizienter umzusetzen

# (DOMAIN) REPOSITORIES

- die Abfragen sollten genau zu den Aufgaben der Domäne passen
  - selbst wenn es eine allgemeine Abfrage (über Kriterien) gibt, lohnen sich die speziellen Methoden

```
Iterable<Buch> findeAlleIn(Erscheinungsjahr jahr);  
Iterable<Buch> findeAllePassendZu(Kriterium... kriterien);
```

- gerne auch Abfragen, die nur Metadaten zurückgeben
  - sind in der Speicherschicht effizienter umzusetzen

# (DOMAIN) REPOSITORIES

- die Abfragen sollten genau zu den Aufgaben der Domäne passen
  - selbst wenn es eine allgemeine Abfrage (über Kriterien) gibt, lohnen sich die speziellen Methoden

```
Iterable<Buch> findeAlleIn(Erscheinungsjahr jahr);  
Iterable<Buch> findeAllePassendZu(Kriterium... kriterien);
```

- gerne auch Abfragen, die nur Metadaten zurückgeben
  - sind in der Speicherschicht effizienter umzusetzen

```
int aktuellerBuchbestand();
```

# REPOSITORIES: ZUSATZNUTZEN

- Repository kann für die Erstellung von Identifikationen (IDs) für neue Root Entities zuständig sein
- kann zusätzliche speicherseitige Prüffelder bei Veränderungen setzen („zuletztGeändertAm“)
- kann für Unit Tests leicht gemockt werden
- kann für Integrationstests ohne Datenbank durch eine In-Memory-Implementierung ersetzt werden

# REPOSITORIES: ZUSATZNUTZEN

- Repository kann für die Erstellung von Identifikationen (IDs) für neue Root Entities zuständig sein

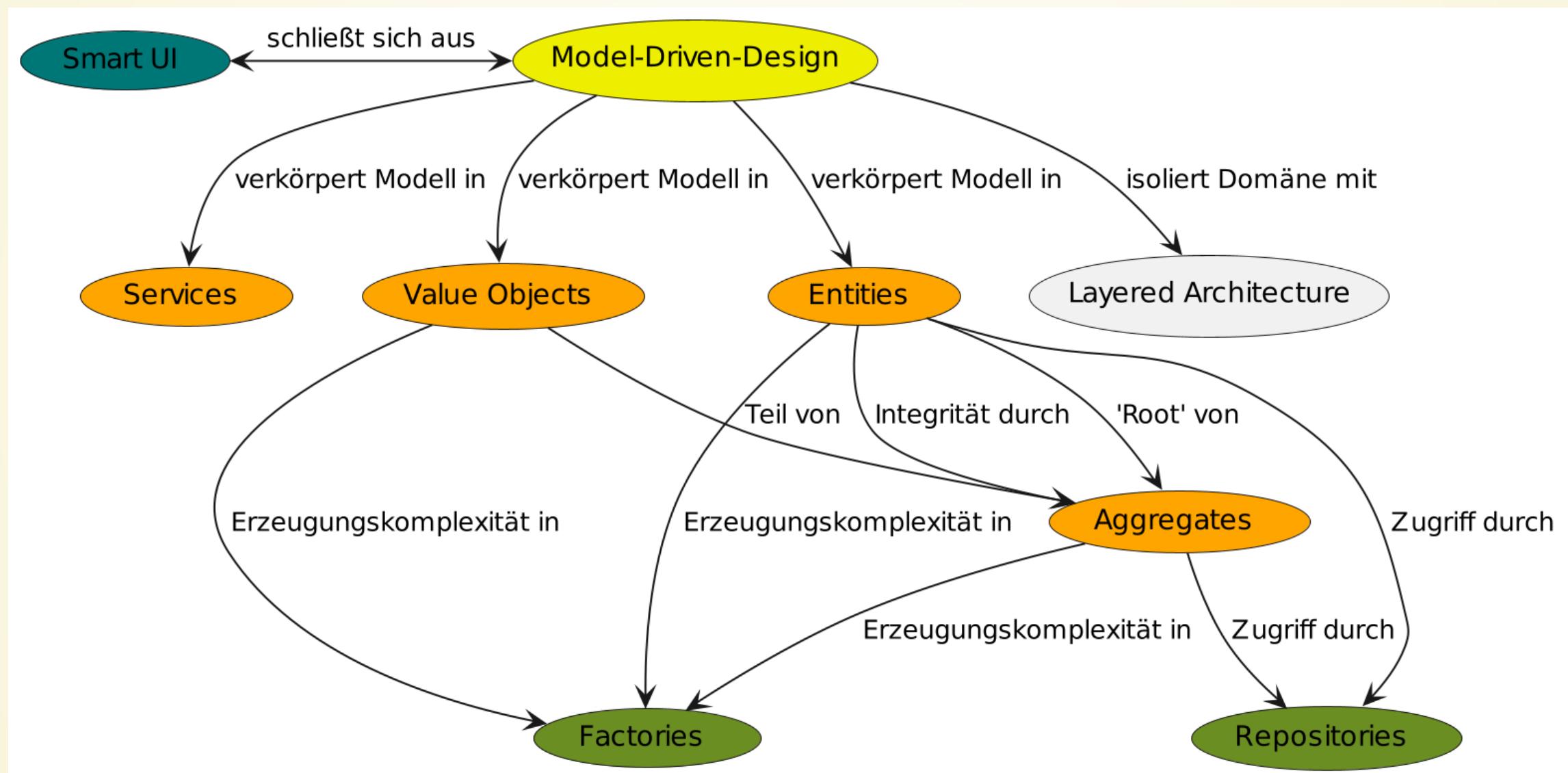
```
public BuchId nächsteId();
```

- kann zusätzliche speicherseitige Prüffelder bei Veränderungen setzen („zuletztGeändertAm“)
- kann für Unit Tests leicht gemockt werden
- kann für Integrationstests ohne Datenbank durch eine In-Memory-Implementierung ersetzt werden

# REPOSITORIES: ZUSAMMENFASSUNG

- Repositories bieten dem Domain Code Zugriff auf persistenten Speicher
  - in der Granularität von Aggregates
  - direkter Zugriff immer nur auf die Aggregate Roots
- Repositories verbergen die konkrete Speicher- technologie vollständig vor dem Domain Code
  - Anti-Corruption-Layer zur Persistenzschicht
- Repositories bieten passend für die Domäne Abfragemöglichkeiten auf den Datenbestand
  - Adapter zwischen Anwendung und Datenbank

# FACTORIES



# FACTORIES

- Factories haben nur einen Zweck: **das Erzeugen von Objekten**
- Factories sind ein *allgemein* nützliches Konzept (unabhängig von DDD)

# FACTORIES

- wenn Logik für die Erzeugung einer Entity, eines Aggregates oder eines VOs (zu) komplex wird, kann dies den eigentlichen Zweck des Objekts verschleiern (Verletzung SRP und SoC)
- Factories helfen, indem sie die Verantwortung für die Konstruktion übernehmen → das Objekt kann sich auf sein Verhalten konzentrieren

# FACTORIES: MEHRDEUTIGKEIT

- der Begriff *Factory* wird in OOP mehrdeutig verwendet
  - das **allgemeine Konzept** einer Factory
    - *irgendein* Objekt oder *irgendeine* Methode zur Erzeugung von Objekten als Konstruktor-Ersatz
  - spezielle **Erzeugungsmuster**
    - Factory Method
    - Abstract Factory
- im DDD meint man üblicherweise das *allgemeine Konzept*

# ALLGEMEINE FACTORY

- allgemein ist eine Factory *irgendein* Objekt bzw. *irgendeine* Methode als Konstruktor-Ersatz

```
public class Product {  
    private Product(Price price) {  
        // init fields  
    }  
    public static Product createWithPrice(Price price) {  
        // validation and checks  
        return new Product(price);  
    }  
}
```

# ZUSAMMENFASSUNG *TAKTISCHES DDD*

- um dEntities und VOs technisch sauber implementieren zu können, benötigen wir unterstützende Strukturen
- **Domain Services** enthalten Use Cases oder entkoppeln von Drittssystemen
- **Aggregates** gruppieren Entities und vereinfachen die Beziehungen zwischen den Gruppen
  - **Aggregate Roots** stellen die primären Objekte dar
- **Repositories** entkoppeln die Persistenz und machen die Aggregate Roots einfach auffindbar
- **Factories** erzeugen Objekte mit komplexen Konstruktionsregeln

# DOMAIN EVENTS

- nicht im Original-Buch enthalten
- sehr wichtiges Konzept
- Domain Events sind Nachrichten über relevante Ereignisse aus der Domäne
- Eigenschaften eines Domain Events:
  - Was ist passiert?
  - Wann ist es passiert?
  - Wer hat es getan?
  - Wer hat das Event erstellt?

# DOMAIN EVENTS: BEISPIEL

- Domäne: Sportsimulation für Fußball
- (Wahrscheinlich) domänenrelevante Ereignisse
  - Ball gepasst
  - Spieler gefoulт
  - Tor geschossen
  - Spieler eingewechselt
- Vermutlich nicht domänenrelevante Ereignisse
  - von A nach B gelaufen
  - auf den Rasen gespuckt
  - Bratwurst und Bier gekauft

# DOMAIN EVENTS ALS STREAM

- das Fußballspiel als „Strom von Ereignissen“
  - *Live-Übertragung im Radio*
- diese Ereignisse an zentraler Stelle sammeln und speichern
  - Ereignisse sind immutable, sie ändern sich nicht
  - Event Bus sorgt für den Transport
    - zwischen Systemen auch Message Queue genannt
- ermöglicht eine erneute Wiedergabe (Replay) der für die Domäne relevanten \* Vorgänge
  - *Event Sourcing* ist eine Extremausprägung

# DOMAIN EVENTS: ZUSAMMENFASSUNG

- Änderungen in der Domäne als „Event“ posten
- jedes Event enthält die relevanten Neuigkeiten
  - mindestens *Was, Wann, Wer, Von wem erstellt*
- durch den Ereignisstrom werden sehr mächtige Architekturen möglich
  - vollständige Entkopplung von Teilsystemen
  - vollständige Rekonstruktion des Anwendungszustands (Event Sourcing)
- Vergleichbare Konzepte
  - Versionskontrolle (Commits), Logbuch des Captain aus Star Trek

