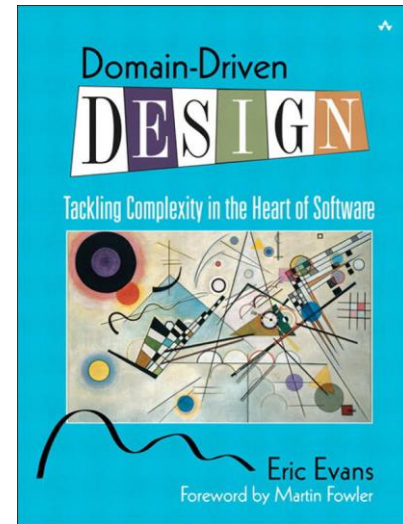

Domain-Driven Design

Domain-Driven Design

- Domain-Driven Design (DDD) ist eine 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

Was ist Design?

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

Domäne

- Auch: *Problemdomäne, Anwendungsdomäne*
- „Abgrenzbares Problemfeld oder bestimmter Einsatzbereich für den Einsatz von Software“
- „**was**“ soll mit Software gelöst werden

“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	Domäne
Wordpress	Blogging
SalesForce	Kundenbeziehungsmanagement
Ebay	Auktionen
DATEV	Finanzbuchhaltung

Domänenmodell

- Damit Software Probleme aus einer bestimmten Domäne bearbeiten kann, muss sie bestimmte fachliche Aspekte (Daten, Regeln, Verhalten) dieser Domäne in 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



Problemdomäne

```
class Beer {  
  
    String name; //Kräusen  
    Brand brand; //Hoepfner  
  
    //...  
}
```

Domänenmodell

Prozesse der Modellbildung

Abgrenzung

- Nichtberücksichtigung irrelevanter Objekte (unserer Anschauung)

Reduktion

- Weglassen von Objektdetails

Dekomposition

- Zerlegung, Auflösung in einzelne Segmente

Aggregation

- Vereinigung, Zusammenfassen von Segmenten zu einem Ganzen

Abstraktion

- Begriffs- bzw. Klassenbildung (i.S.v. Kategorien)

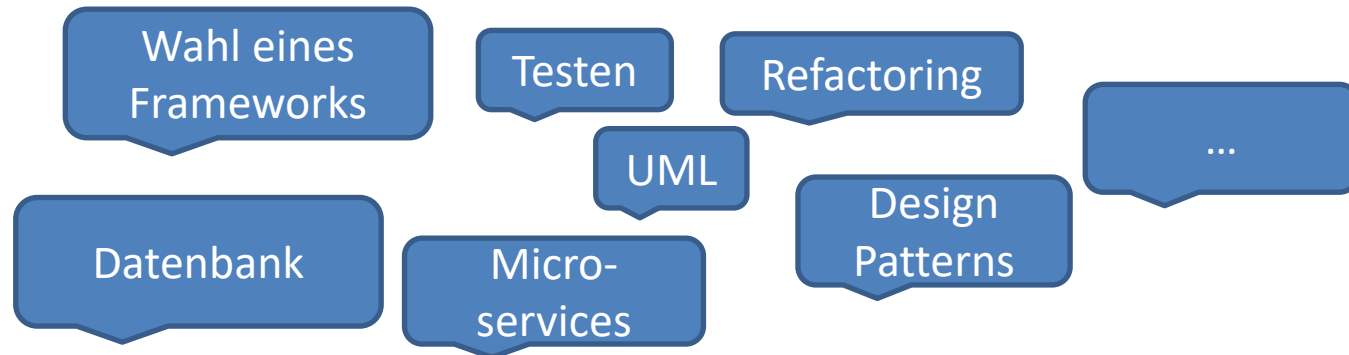
Visualisierung des Modells

- Ein Modell einer Problemdomäne existiert zuerst in der Vorstellung der Entwickler
- Damit diese darüber sprechen können, muss das Modell visualisiert werden, beispielsweise als UML, Architekturdiagramm etc.
- **Die wichtigste und konkreteste Form der Visualisierung eines Modells ist der Quellcode**

Was ist jetzt Design?

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

Design

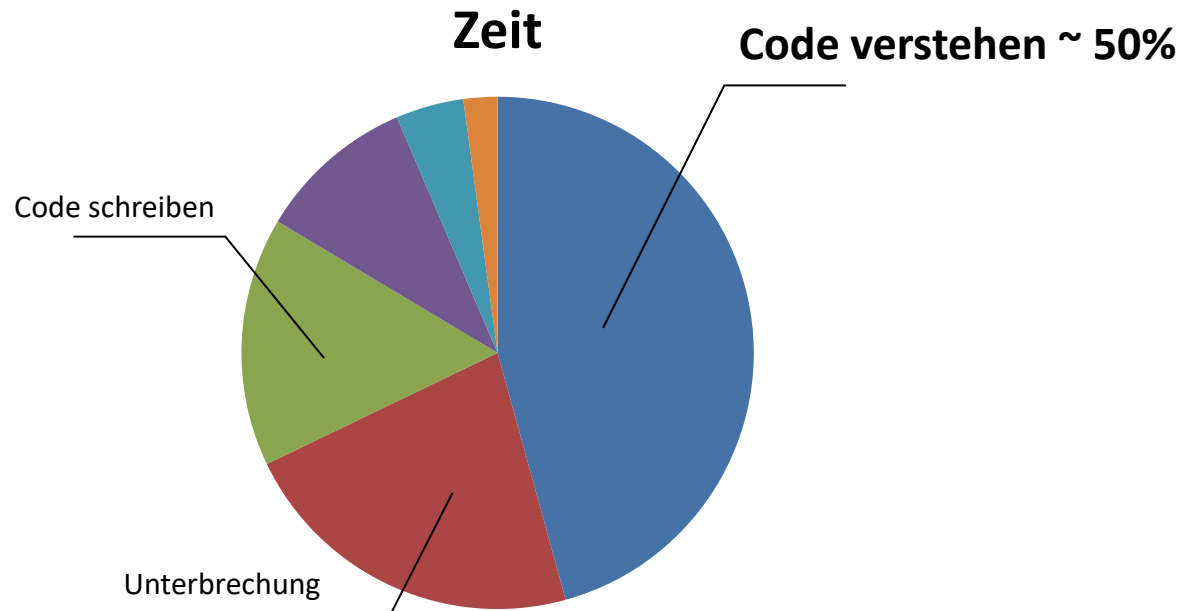


Design ist die Summe aller bewussten und unbewussten Entscheidungen, die Einfluss darauf haben, wie eine konkrete Problemdomäne als Modell abgebildet wird.

“The overwhelming problem with software development is that *everything* is part of the design process. Coding is design, testing and debugging are part of design, and what we typically call software design is still part of design.” [Reeves, 2005]

“Implementation is design continued by other means” [Meyer, 2014]

Womit verbringen Entwickler den Großteil ihrer Zeit?



Ko et al., 2006

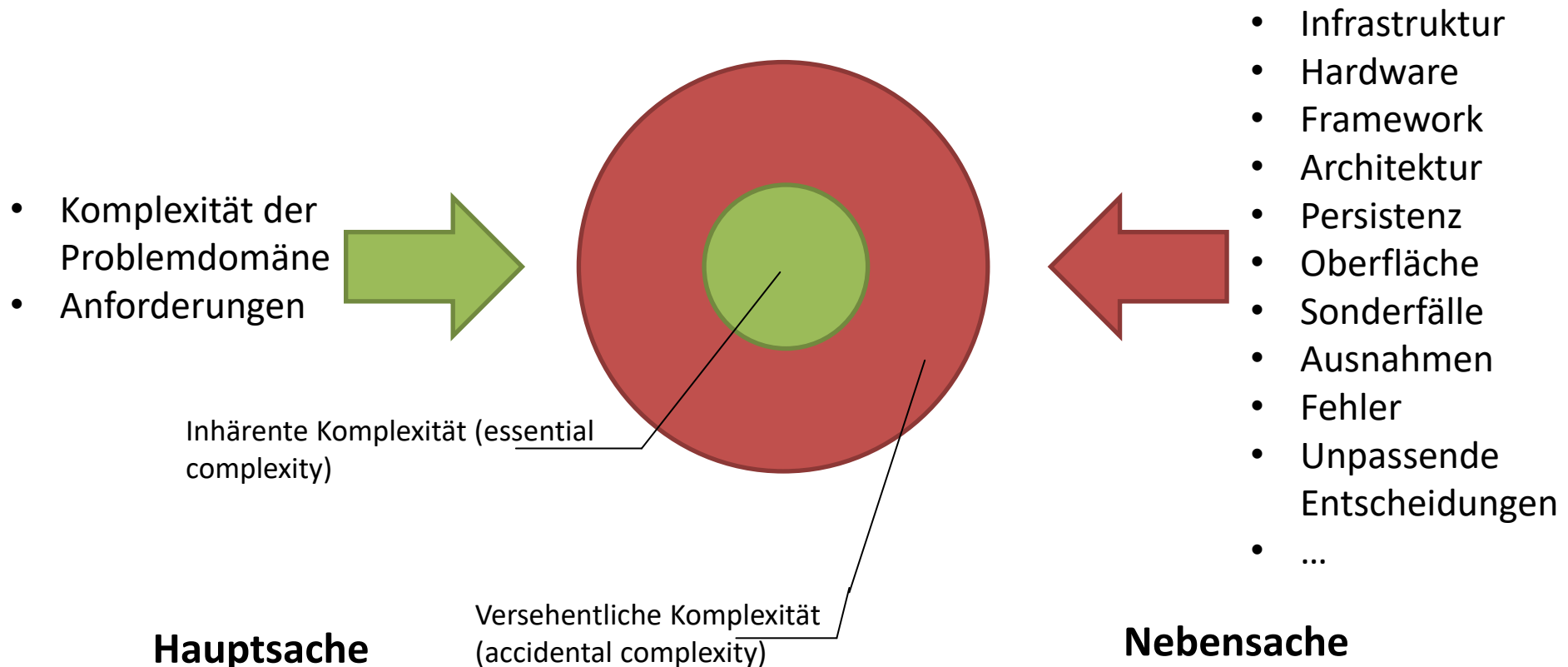
Frage: warum ist das so?

Software-Komplexität

- Softwareentwicklung ist **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

IEEE: „The degree to which a system or component has a design or implementation that is difficult **to understand** and verify.”

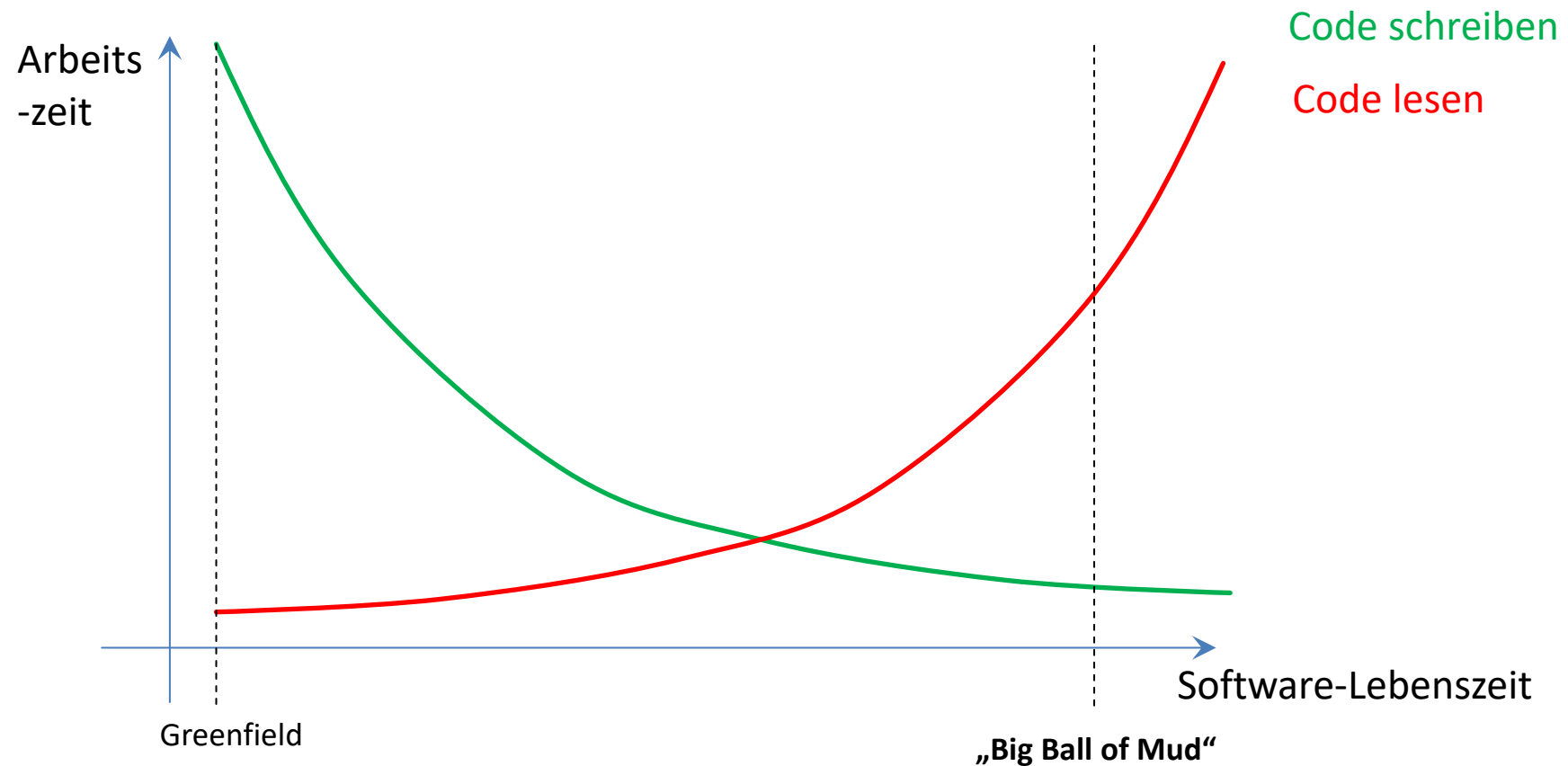
Software-Komplexität



Software-Komplexität

- Die inhärente **Komplexität der Problemdomäne** (**essential complexity**) ist gegeben
 - damit sollte sich ein Entwickler eigentlich beschäftigen
- die zusätzliche **versehentliche Komplexität** (**accidental complexity**) kann nicht komplett eliminiert werden – sie ist ein notwendiges Übel (vgl. Pure Fabrication)
- Man kann aber verhindern, dass die inhärente Komplexität negativ von der versehentlichen Komplexität beeinflusst wird

Auswirkungen von unbeherrschter Komplexität



Big Ball of Mud (BBoM)

- “code that does something useful, but without explaining how” [Evans, 2004]
- Eigenschaften:
 - Code gibt Intention nicht preis
 - Technische Belange belasten/verunreinigen die Fachlogik
 - Fachlogik ist über gesamte Anwendung verteilt
- Resultat:
 - Selbst kleine Anpassungen sind nur mit hohem Aufwand und hohem Risiko umsetzbar



Big Ball of Mud

- Fester Begriff in der Softwarearchitekten-Szene
- Angstgegner (Berufsehre steht auf dem Spiel!)
- Ist nicht immer ein Antipattern:
 - Nicht jeder Teil der Anwendung muss bis ins kleinste Detail bewusst durchdesigned werden
 - Geschwindigkeit ist manchmal wichtiger als Qualität
 - Wichtig ist, dass der Ball in einem engen, fest definierten Kontext bleibt und sich nicht ausbreitet -> kontrolliertes Risiko
- Small Balls of Mud ist eine moderne Architektur (Microservices)

Wie hilft Domain-Driven Design?

- DDD legt den Fokus auf ein präzises und tiefgreifendes, gemeinsames Verständnis der Problemdomäne
 - Rückschlüsse von Problemdomäne auf Code und umgekehrt möglich (Intention)
 - Reduzierter „Übersetzungsaufwand“ erleichtert Nachdenken
- DDD bietet Muster, um ein konsistentes, erweiterbares, funktionierendes Modell der Problemdomäne zu entwickeln
 - Kapselung von Fachlichkeit (inhärenter Komplexität) und versehentlicher Komplexität

Strategisches DDD

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

“[Strategic Design] lays out techniques for recognizing, communicating and choosing the limits of a model and its relationship to others.”

-Eric Evans

Taktisches DDD

- Taktisches DDD beschäftigt sich damit, wie die Fachlichkeit in Code implementiert werden kann
- Im Endeffekt eine Sammlung von Entwurfsmustern und Design-Prinzipien, die besonders gut zu DDD passen

Zusammenfassung

- Software ist ein Modell einer konkreten Problemdomäne, welches das Bearbeiten von Aufgaben dieser Domäne erlaubt
- Das Modell abstrahiert die für die Aufgabenbewältigung relevante Teile der Wirklichkeit
- Der Quellcode ist die wichtigste Visualisierung des Modells
- Design ist die Summe aller bewussten und unbewussten Entscheidungen, **wie** eine konkrete Problemdomäne als Modell abgebildet wird
 - Änderungen am (fachlichen) Quellcode bedeuten eine Änderung am Modell und sind damit Design

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 BBoM
- DDD hilft dem vorzubeugen durch:
 - Fokus auf ein präzises und tiefgreifendes, gemeinsames Verständnis der Problemdomäne
 - Kapselung von inhärenter Komplexität(Fachlichkeit) und versehentlicher Komplexität

Teil 2: Strategisches Domain-Driven Design

- Ubiquitous Language
- Problemdomäne schärfen
- Bounded Context
- Context Mapping

Die Sprache der Domäne

- Damit ein Modell die Regeln, Prozesse und Konzepte (Fachlichkeit) einer Domäne möglichst exakt erfassen kann, ist es wichtig, dass man die Domäne **sehr gut versteht**
- Der Weg zum Verständnis einer Domäne führt über die in dieser Domäne gesprochene **Sprache**
- DDD legt daher größten Wert auf das Vokabular, dass zwischen Domänenexperten und Entwicklern verwendet wird und führt dafür ein eigenes Konzept ein:
die sog. **Ubiquitous Language**
- Die **UL** oder **Domänensprache** ist das **wichtigste Konzept des DDD**

Ubiquitous Language (UL)

- „allgegenwärtige Sprache“
- Gleiche Begriffe in Domäne und Sourcecode
- Jede Domäne besitzt eine eigene Fachsprache
- Verstehen, warum diese Fachsprache gesprochen wird
- Nicht versuchen, diese Fachsprache in „eigene“ Begriffe zu übersetzen
- Ubiquitous Language bezeichnet die von Domänenexperten und Entwicklern gemeinsam im Projekt verwendete Sprache

Verständnisprobleme

- 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 Domänen-experten 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
- Je mehr Stakeholder beteiligt sind, desto mehr gedankliche Versionen „der Domäne“ gibt es, und desto wichtiger ist ein gemeinsames Verständnis

Verständnisprobleme im Code

- Diese Kluft im gegenseitigen Verständnis setzt sich in die Software fort
 - Der Code entfernt sich von der Sprache der Domäne
 - Die Implementierung wird schwerer zu verstehen
- Die Ubiquitous Language soll die Kluft reduzieren, indem Domänenexperten und Entwickler eine gemeinsame Sprache finden, die
 - Alle relevanten Konzepte, Prozesse und Regeln der Domäne beschreibt
 - Zusammenhänge verdeutlicht
 - Mehrdeutigkeiten und Unklarheiten beseitigt

Beispiel für Mehrdeutigkeit

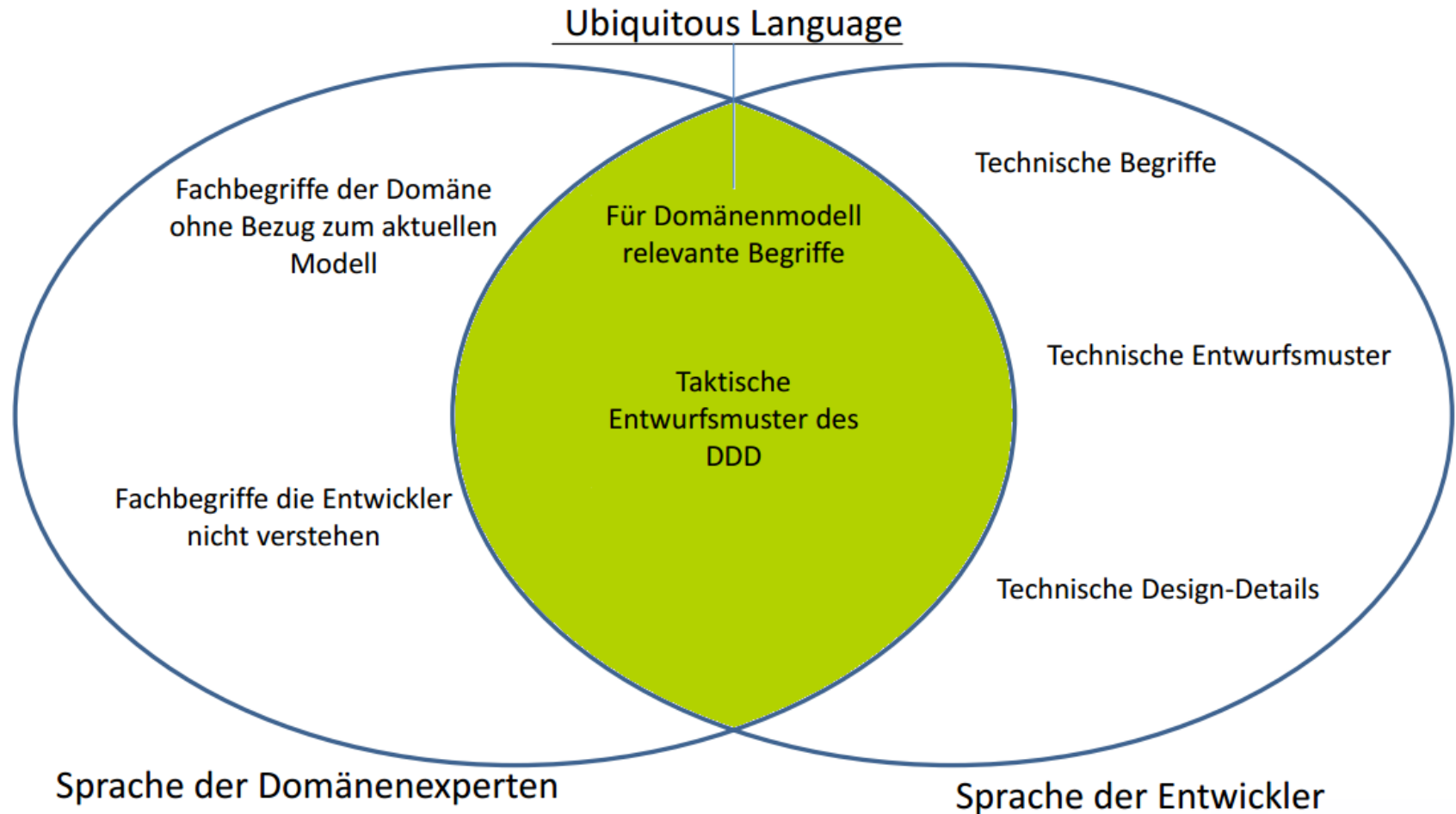
Eine Bibliothek bietet im Internet einen Service mit Verkauf von Nachdrucken alter Zeitschriften an. Dann kann der Begriff „Artikel“ schnell mehrere Bedeutungen haben:

- Ein Ausschnitt aus einer Zeitschrift (Seite x bis y)
- Eine kaufbare Position im Warenkorb (also eine ganze Zeitschrift oder ein Ausschnitt)

Daraus ergeben sich uneindeutige Anforderungen:

- „Alle gekauften Artikel sollen für den Käufer auch als Download zugänglich sein“

Die gemeinsame Projektsprache



Gemeinsame Projektsprache finden

- Mit den Domänenexperten sprechen
 - Begriffe nicht „im stillen Kämmerchen“ festlegen
- Genau zuhören und kritisch nachfragen
 - „Warum darf diese Schaltfläche nicht mehrfach betätigt werden?“
 - „Was bedeutet „Arbeitsgang priorisieren?“
- Ein Glossar für die Begriffe kann helfen
 - Die GP/UL muss aber in allen Projektartefakten zu finden sein
- Nie mit dem Zuhören und Nachfragen aufhören

Artefakte mit Projektsprache

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

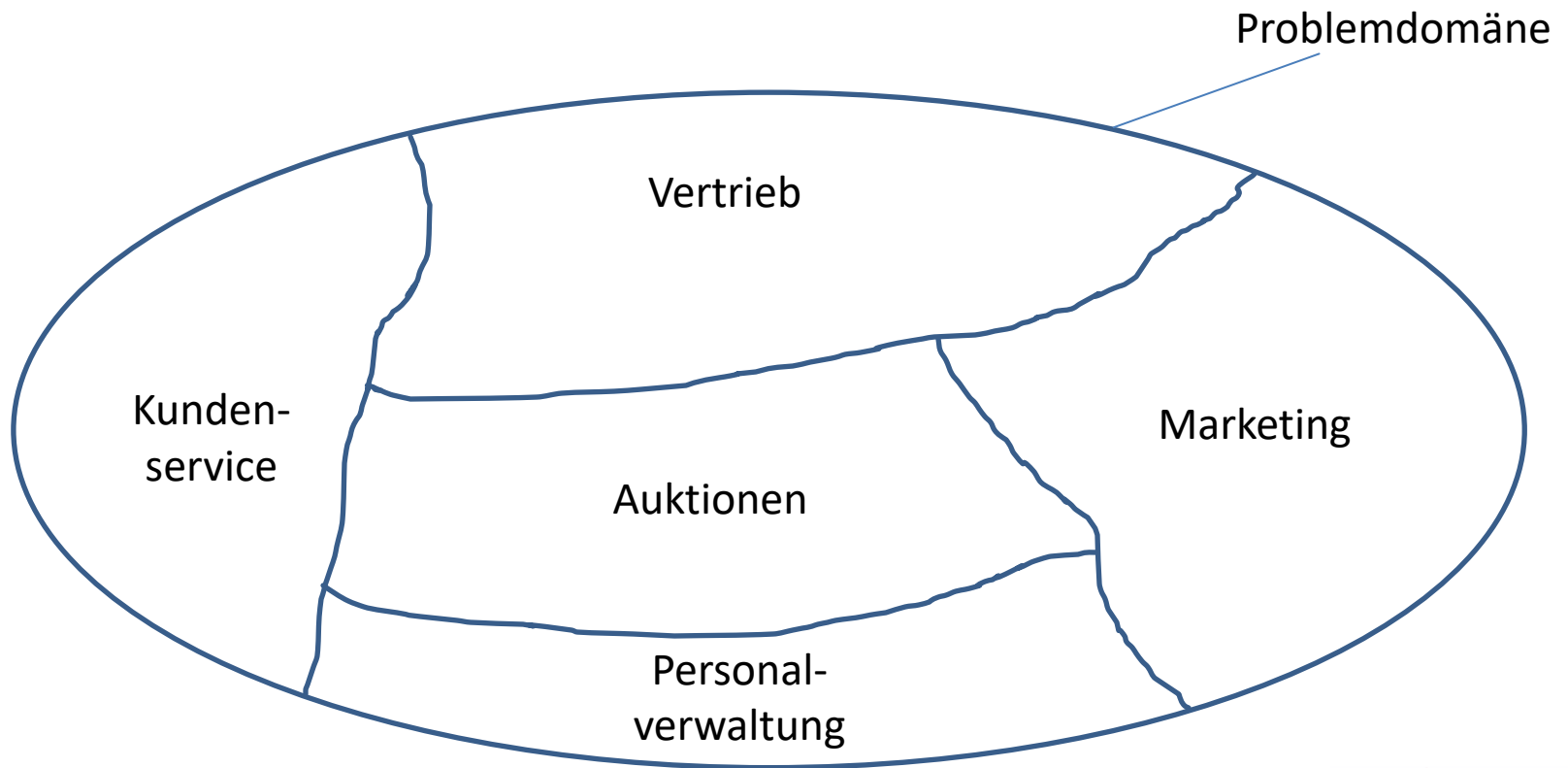
Grenzen der Projektsprache

- Nicht alle Begriffe und Zusammenhänge sind für das Projekt von Bedeutung
- Immer auf den Kern des Projekts konzentrieren
 - Randbegriffe auch mal unspezifiziert lassen
- Es darf innerhalb eines Projekts gut und klar modellierte Kernbereiche und gleichzeitig unscharf oder gar nicht modellierte Randbereiche geben
 - Ein bisschen 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 UL wird 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, unterstützenden Domänen** und **generischen Domäne**

Problemdomäne: Online-Auktionshaus



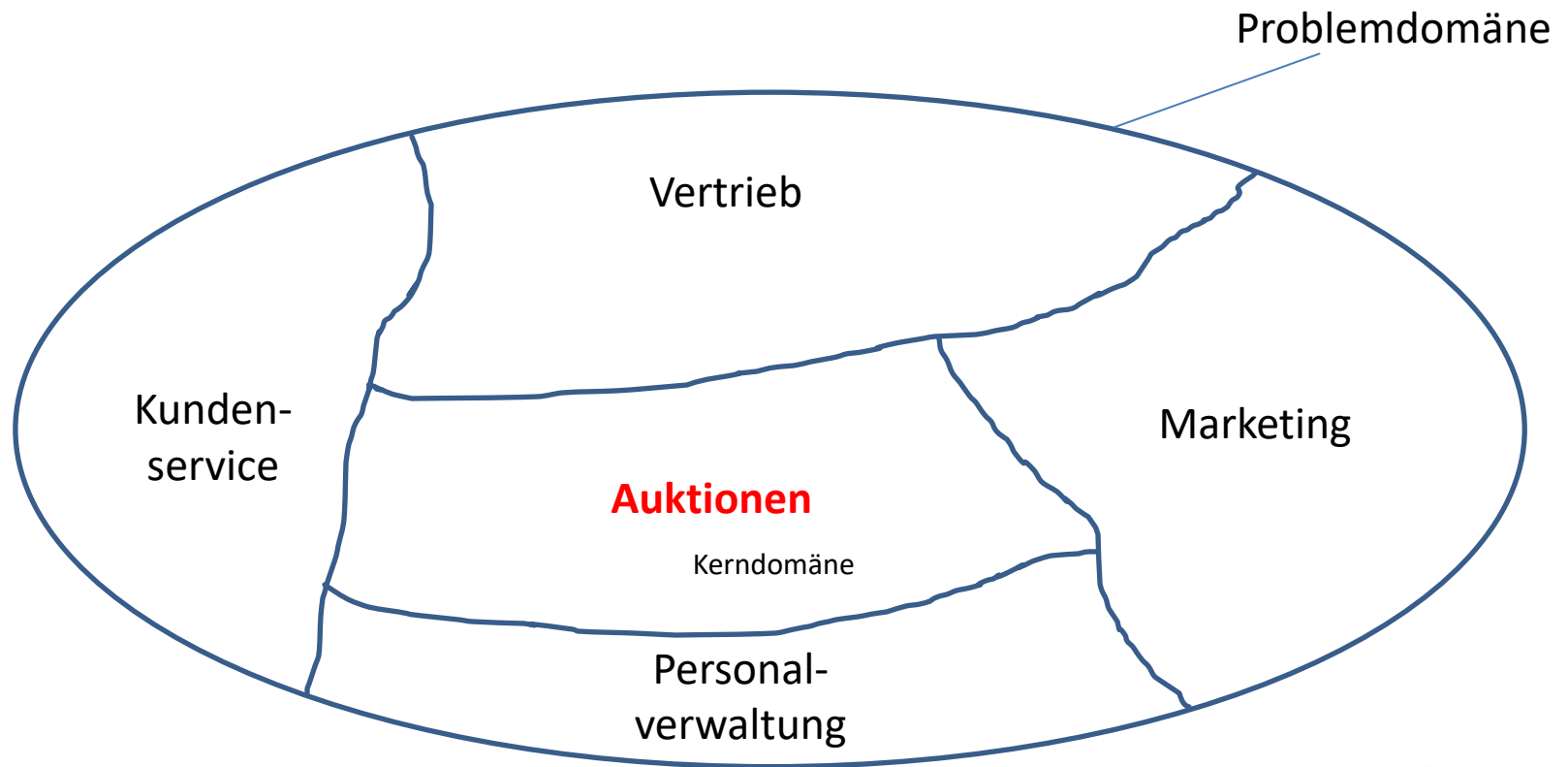
Kern-Domäne

- Kerngeschäft eines Unternehmens („wo wird das Geld erwirtschaftet“)
 - 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 [2004] nennt drei Fragen, die bei der Identifikation der Kerndomäne helfen:
 - **“What makes the system worth writing?”**
 - “Why not buy it off the shelf?”
 - “Why not outsource it?”

“The Core domain should deliver about 20% of the total value of the entire system, be about 5% of the code base, and take about 80% of the effort.”

Problemdomäne: Online-Auktionshaus

Kerndomäne „Auktionen“

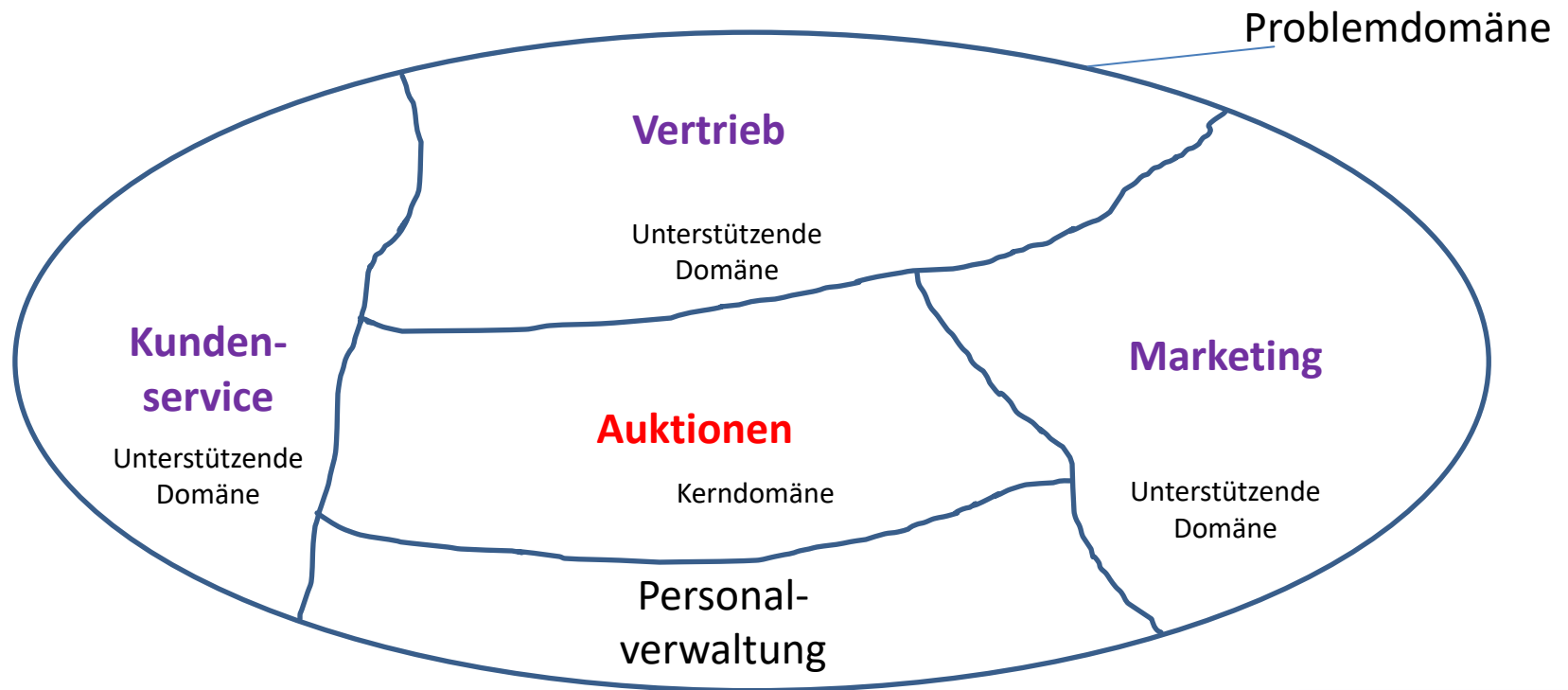


Unterstützende Domäne

- 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 der Einsatz von Fremdsoftware

Problemdomäne: Online-Auktionshaus

Unterstützende Domänen „Marketing“, „Vertrieb“ und „Kundenservice“

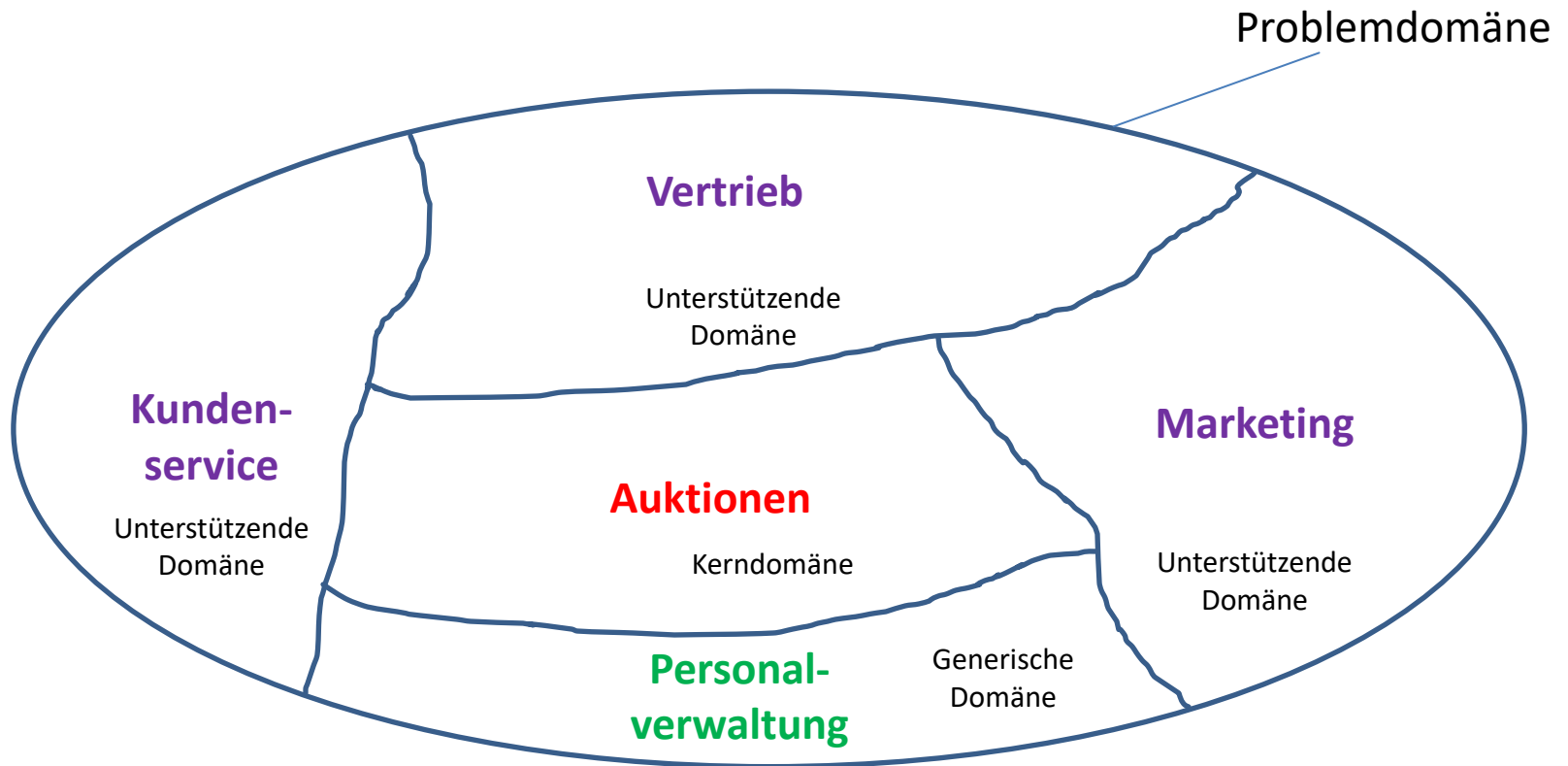


Generische Domäne

- Beschreibt Teile der Problemdomäne, die für das tägliche Geschäft unerlässlich sind, jedoch nicht zum Kerngeschäft gehören
- Beispiel: die meisten Unternehmen müssen in irgendeiner Art Rechnungen schreiben, das hat aber nichts mit dem Kerngeschäft zu tun
- Hier reicht meist Fremdsoftware oder Outsourcing

Problemdomäne: Online-Auktionshaus

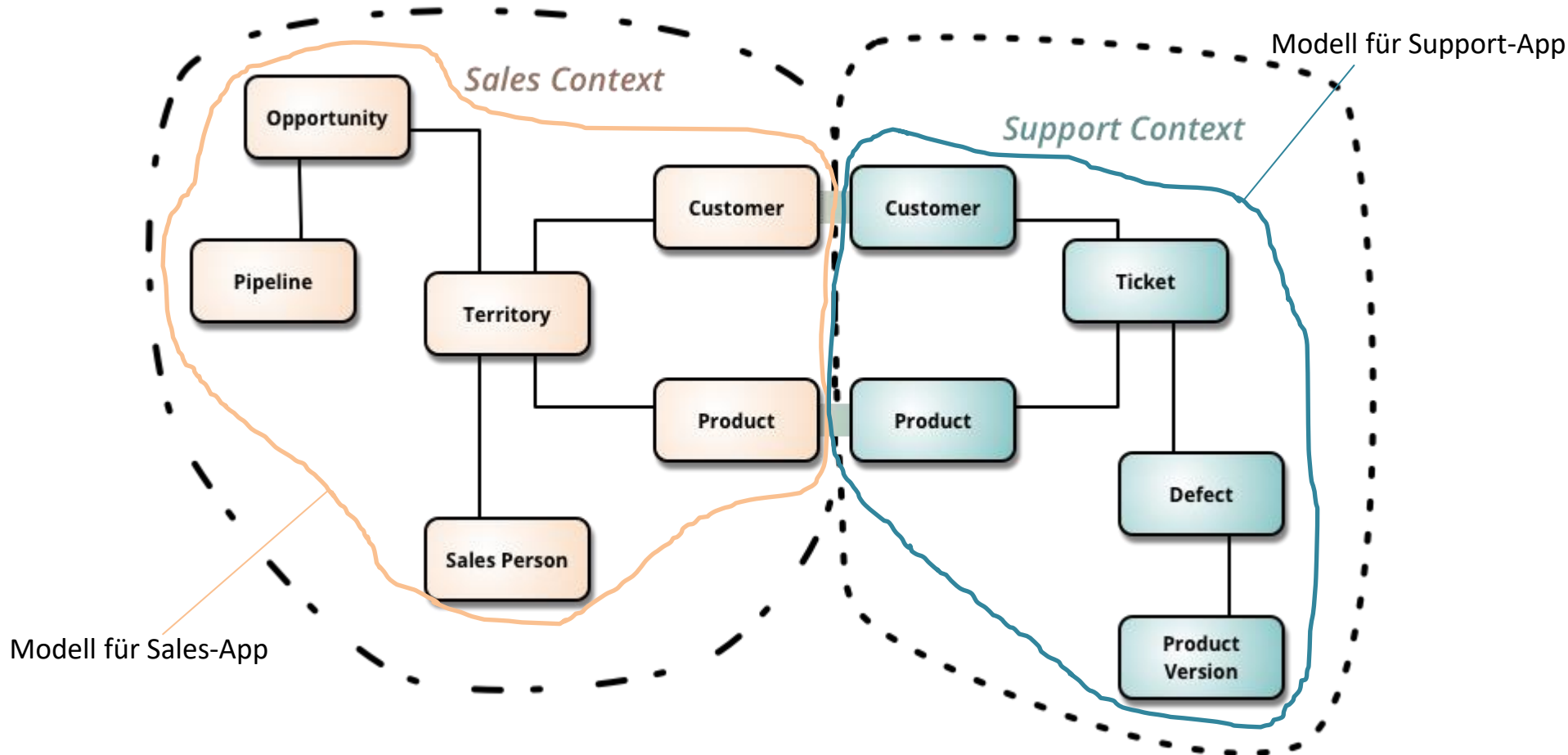
Generische Domäne „Personalverwaltung“



Bounded Context

- Eine Anwendung – und damit ein Modell - steht 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



Modell für Sales-App

Modell für Support-App

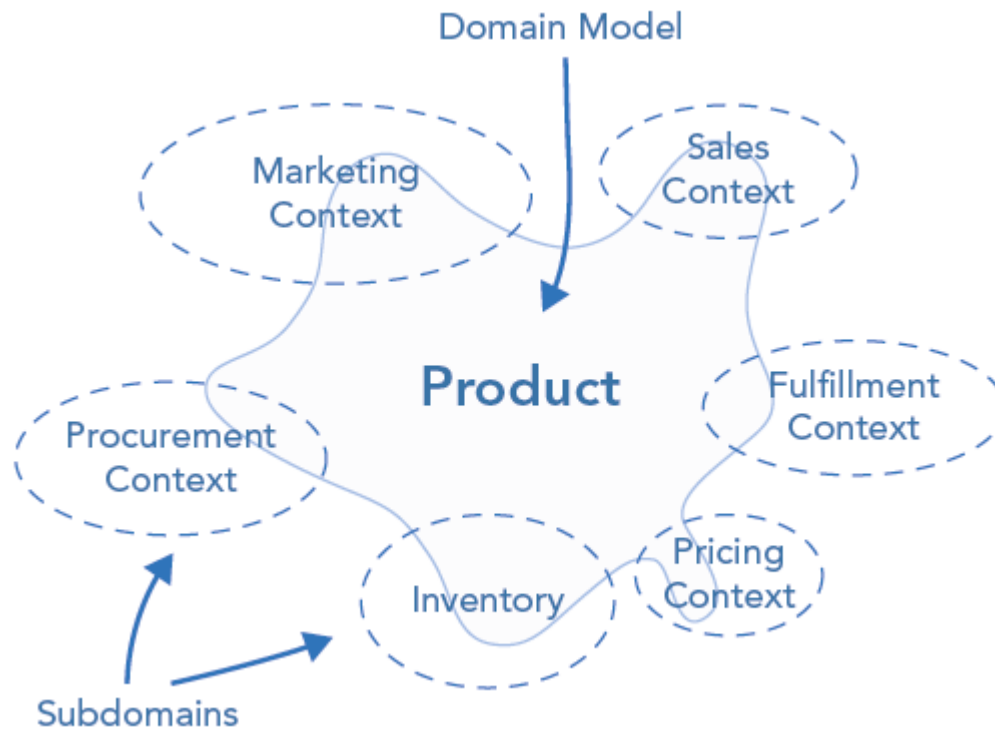
DHBW

45

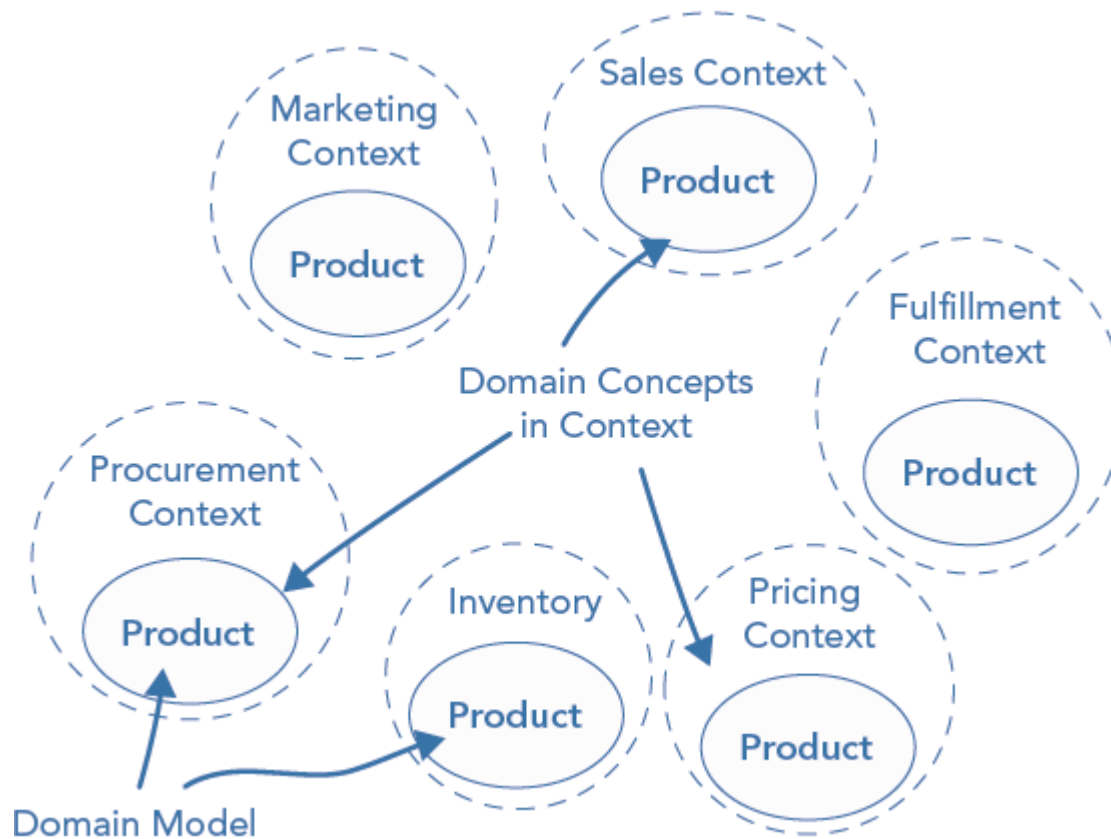
Bounded Context

- Ein **Bounded Context**
 - ordnet ein Modell einem bestimmten fachlichen Kontext zu
 - Ist immer eine logische Grenze („für welchen Zweck ist das Modell gedacht?“)
 - *Kann* eine physikalische Grenze sein (ein bounded context = eine Anwendung)
 - Sollte eine organisatorische Grenze sein (ein Team pro Bounded Context)
- Ein Modell hat also immer nur innerhalb eines bestimmten fachlichen Kontextes Gültigkeit
 - Das Modell ist an einen Kontext „gebunden“ -> bounded context
 - Der Kontext gibt vor, wie die UL des Modells zu interpretieren ist (siehe Beispiel: „Customer“ im Sales- oder Support-Kontext)

Mehrdeutiger Kontext



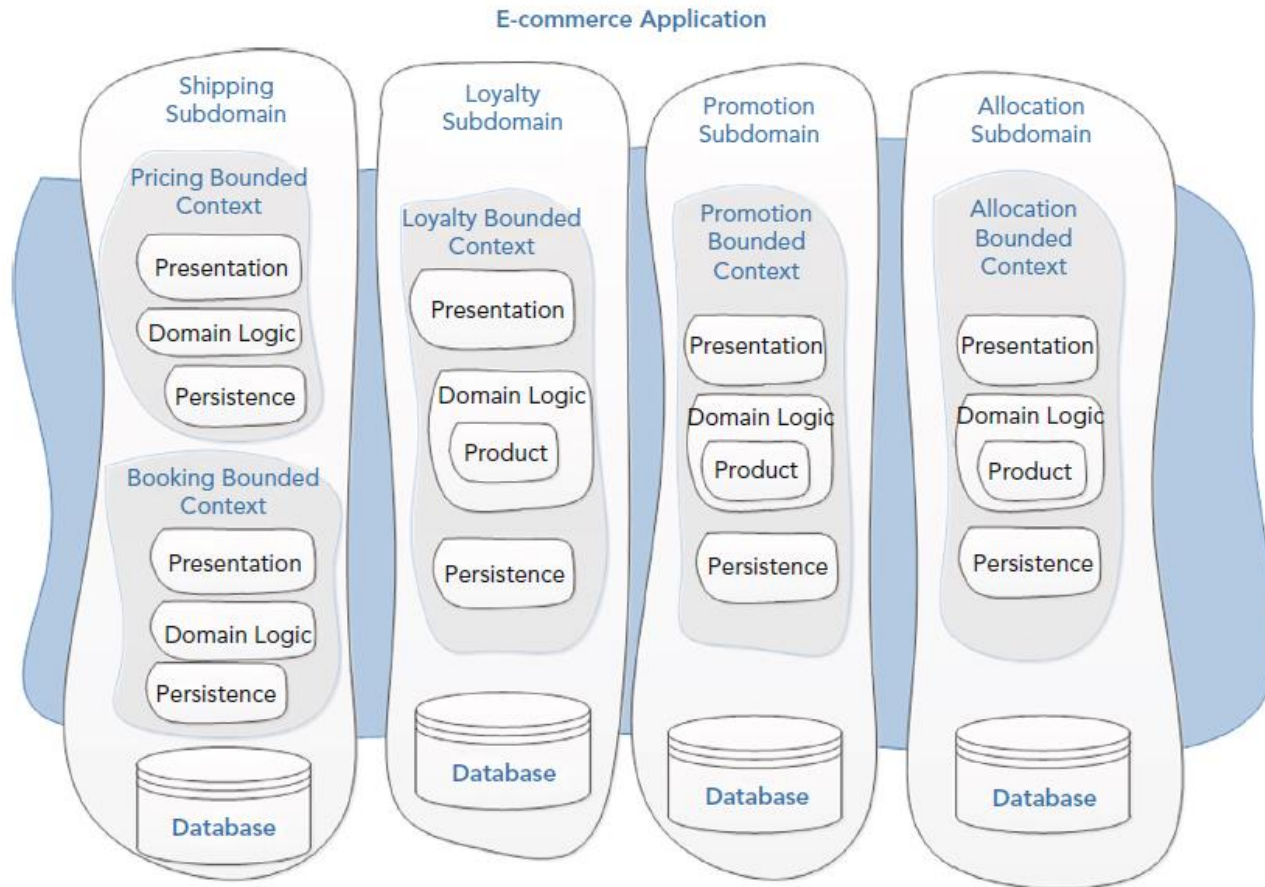
Eindeutiger Kontext



Exkurs: Kontexte und MicroServices

- In einer MicroService-Architektur werden viele kleine Modelle mit je eigenem Kontext zu einem großen Gesamtsystem orchestriert
- Normalerweise ist je ein Team für einen oder wenige dieser MicroServices verantwortlich
- Der eigentliche Vorteil ist also nicht nur die technische Flexibilität, sondern die Skalierbarkeit durch viele kleine Teams

Bounded Context und Zuordnung zur Problemdomäne



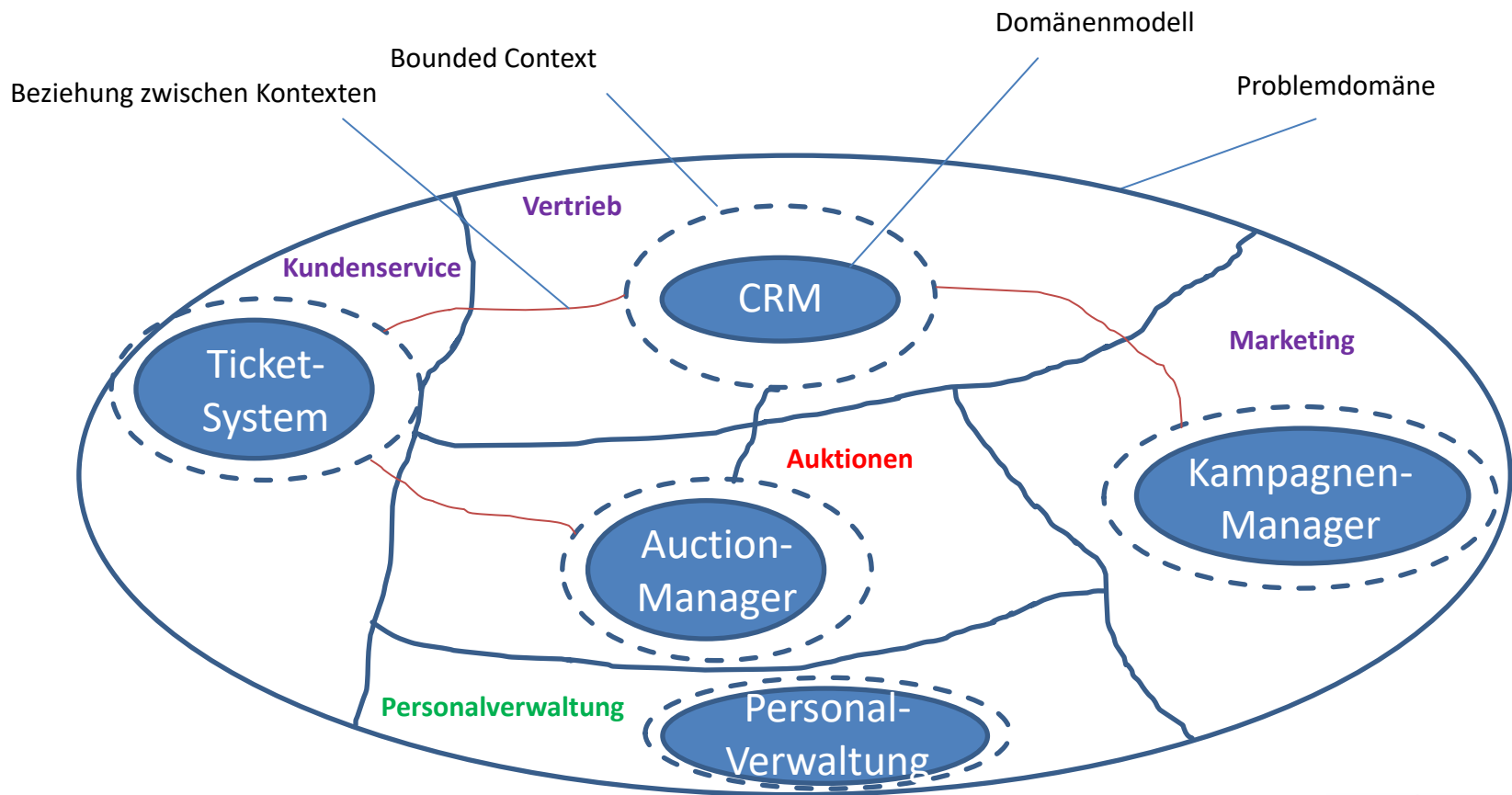
Bounded Context und Zuordnung zur Problemdomäne

- Idealerweise ist jeder Bounded Context genau einer Subdomäne zugeordnet
 - Innerhalb einer Subdomäne können aber mehrere Kontexte existieren!
- Ist das nicht der Fall, dann deutet das darauf hin, dass unscharfe Modelle existieren, die Fachlichkeit aus mehreren Domänen abbilden
- Um diese Zuordnung zu visualisieren, führt DDD die Context Map ein

Context Mapping

- Eine Context Map bildet die existierenden Kontexte innerhalb der Subdomänen einer Problemdomäne ab
- Ziel:
 - Beziehungen zwischen Kontexten visualisieren und deren Art bestimmen

Context Map für Problemdomäne „Online-Auktionshaus“



Beziehungen zwischen Kontexten

- Eine Beziehung zwischen zwei Kontexten bedeutet sowohl eine technische als auch eine organisatorische Abhängigkeit
 - 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

Anticorruption Layer

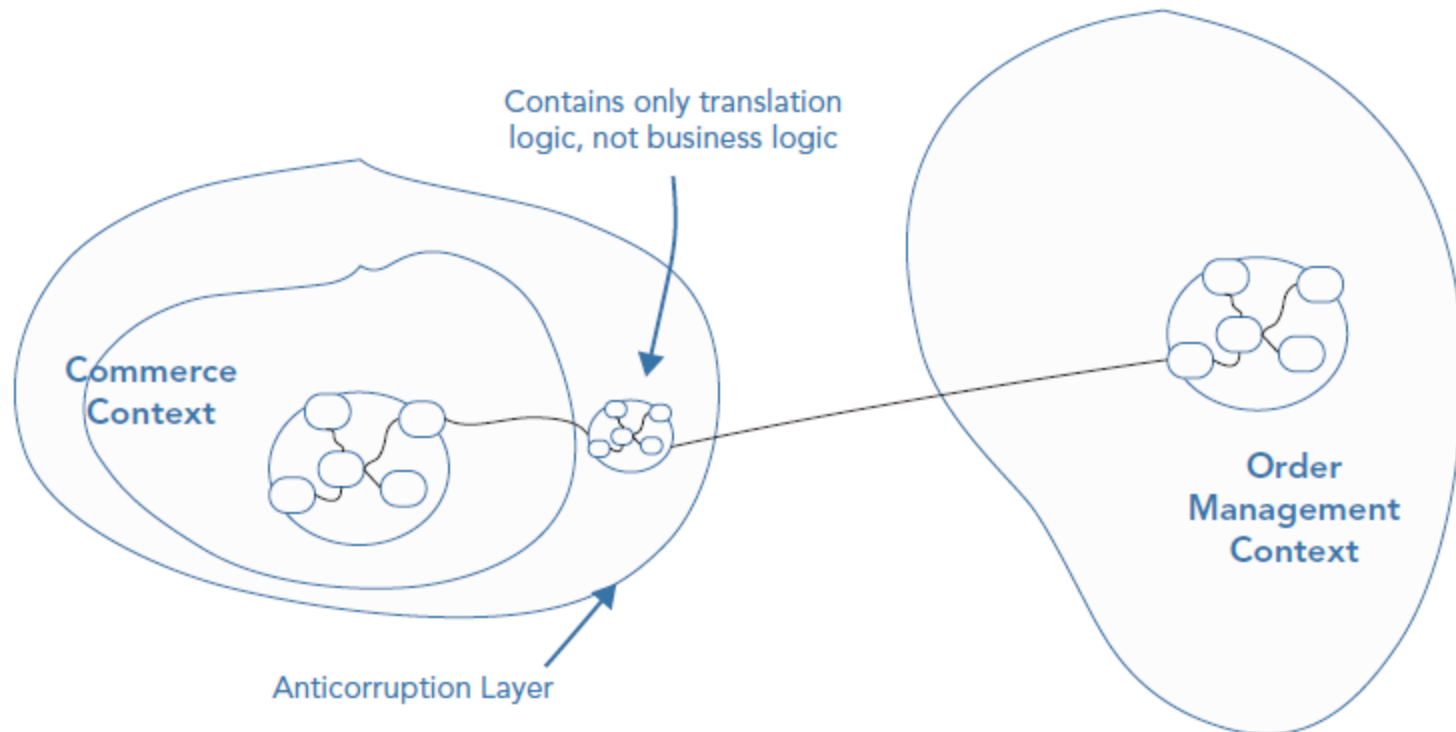
- Modelle müssen häufig mit anderen Modellen Daten austauschen
- Beispiel:
 - Import von Kundendaten aus einer CSV-Datei
 - Abruf des Versandstatus aus einem Logistik-Webservice (DHL o.ä.)
- Die Modelle, mit denen Daten ausgetauscht werden, liegen normalerweise in einem anderen Kontext als das eigene Modell und haben eine eigene Sprache
- Bei der Integration verschiedener Modelle über solche Kontextgrenzen hinweg darf das eigene Modell nicht durch die Eigenheiten des fremden Modells korumpiert werden
- Gefahr: immer wenn sich das fremde Modell ändert, muss man das eigene Modell anpassen

Anticorruption Layer

- Eine Anticorruption Layer isoliert das eigene Modell von Fremdeinflüssen
- Im Grunde eine reine Übersetzungsschicht, die Informationen aus Kontext B so aufbereitet, dass Kontext A diese versteht
- Enthält keine Business-Logik, nur Übersetzungslogik
- Ähnlich dem Adapter-Entwurfsmuster, nur auf Anwendungsebene anstatt Klassenebene

Siehe auch <https://docs.microsoft.com/de-de/azure/architecture/patterns/anti-corruption-layer>

Anticorruption Layer

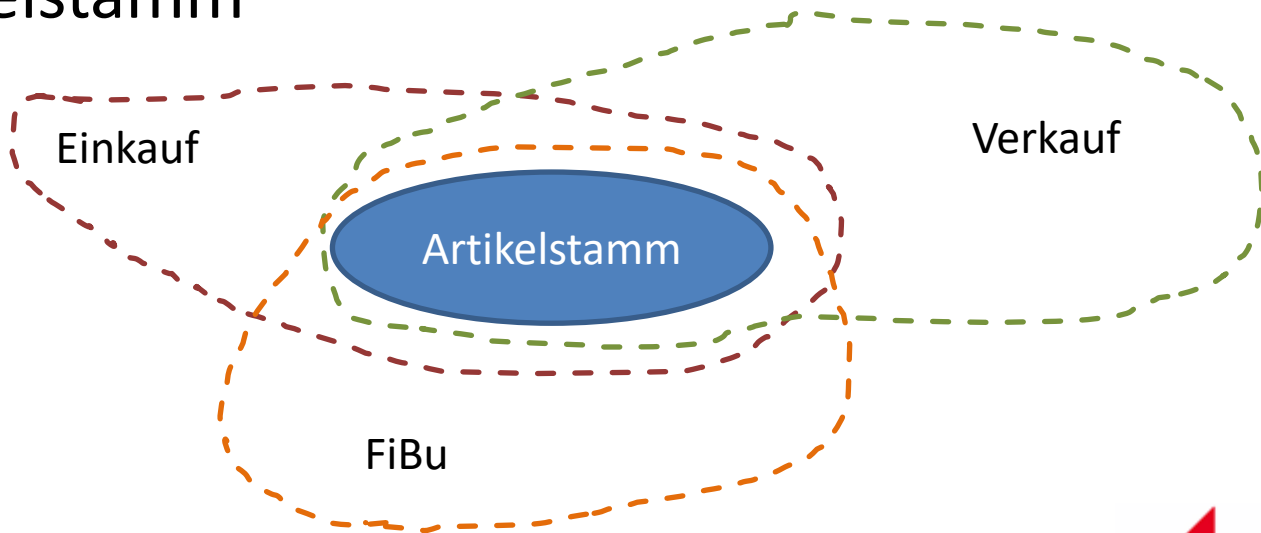


Shared Kernel

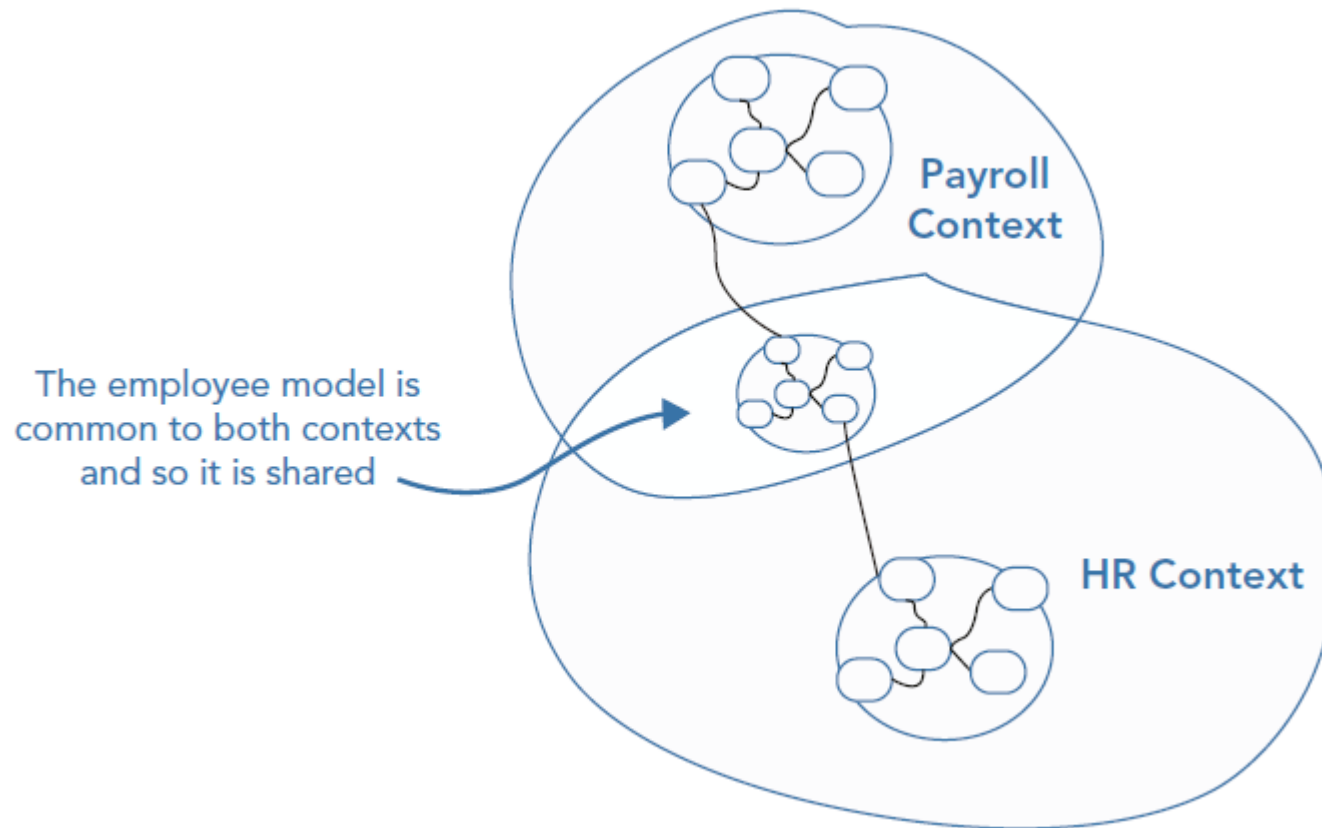
- Manchmal werden in verschiedenen Kontexten der selben Problemdomäne sehr ähnliche Konzepte und Regeln modelliert
- In dem Fall kann es sein, dass sich der Aufwand für eine klare Trennung der Modelle und der Übersetzungsschichten dazwischen nicht lohnt
- Stattdessen können sich die Teams der verschiedenen Kontexte auf ein gemeinsam verwaltetes Teilmodell einigen
- Dieses Teilmodell nennt man „shared kernel“
- Bedeutet, dass die unterschiedlichen Teams sehr eng zusammenarbeiten müssen (starke Abhängigkeit)
 - Geteiltes Modell sollte so klein wie möglich sein!

Shared Kernel

- Beispiel: ERP-Software
 - Es existiert ein Artikelstamm
 - Materialverwaltung, Einkauf, Verkauf, Finanzbuchhaltung usw. verwenden den selben Artikelstamm



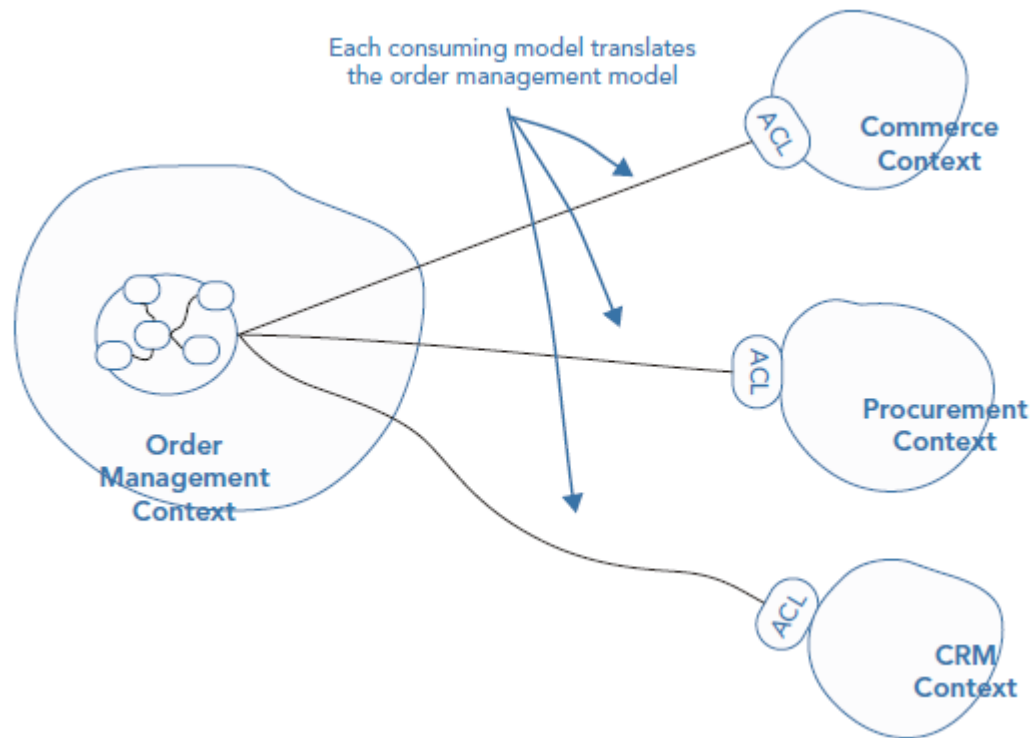
Shared Kernel



Open Host Service

- Es kommt oft vor, dass viele Klienten Daten aus einem zentralen Modell auf jeweils verschiedenen Wegen konsumieren
 - Klient 1: CSV
 - Klient 2: XML
 - Klient 3: ...
- Jeder Klient müsste dann seine eigene Schnittstelle und AC-Layer implementieren

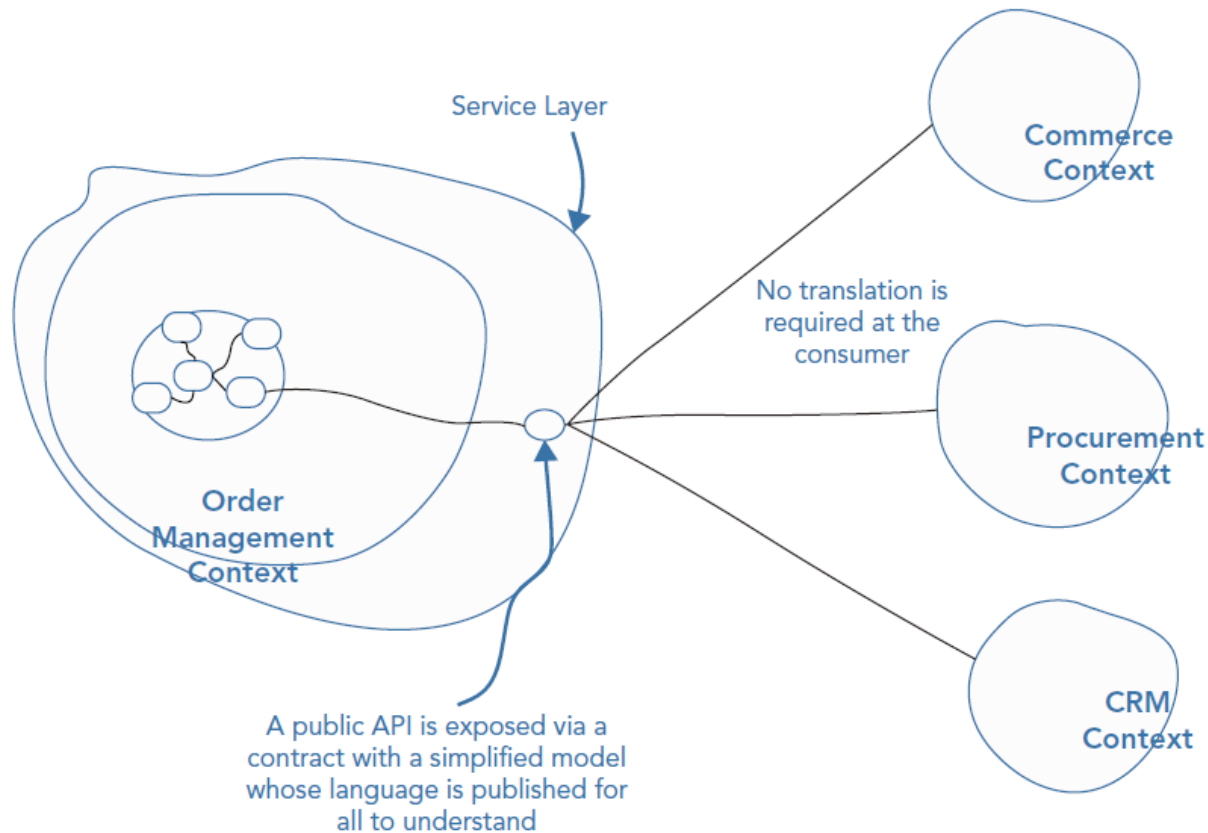
Open Host Service



Open Host Service

- Statt dessen können sich alle Klienten auf eine gemeinsame Schnittstelle in einer gemeinsamen, vereinfachten Sprache einigen
 - Vorausgesetzt, das ist möglich
- Das zentrale Modell implementiert diese Schnittstelle als Services
- Der Übersetzungsaufwand für jeden Klienten entfällt
- Weitere Klienten können angedockt werden (so lange diese mit der gemeinsamen Sprache einverstanden sind)

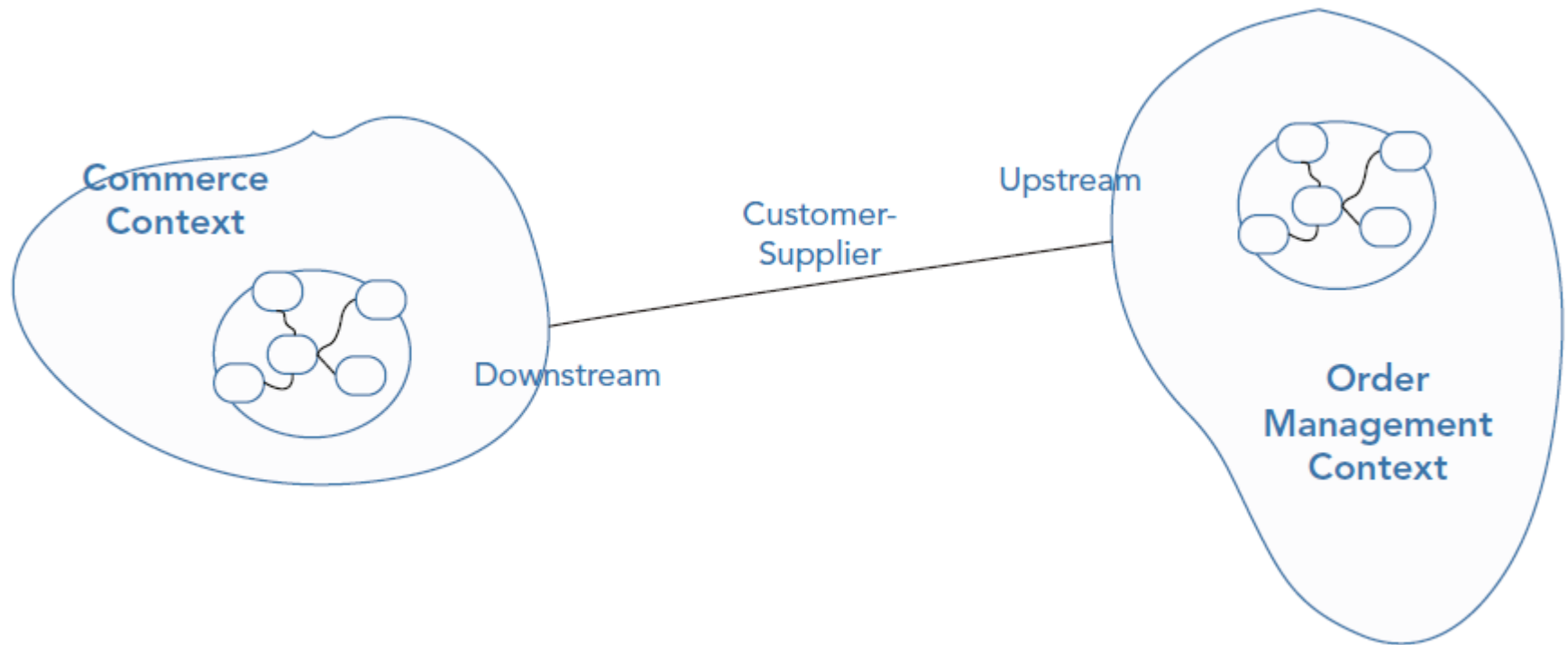
Open Host Service



Customer-Supplier

- Hebt hervor, dass ein Modell (Customer) von einem anderen stark abhängig ist (Supplier)
 - Wenn sich das Modell des Suppliers ändert, muss sich das Modell des Customers ändern – aber nicht umgekehrt
- Dieser Abhängigkeit muss organisatorisch Rechnung getragen werden
 - Bspw. sollte das Team des Customer-Modells bei Sitzungen des Supplier-Modells anwesend sein

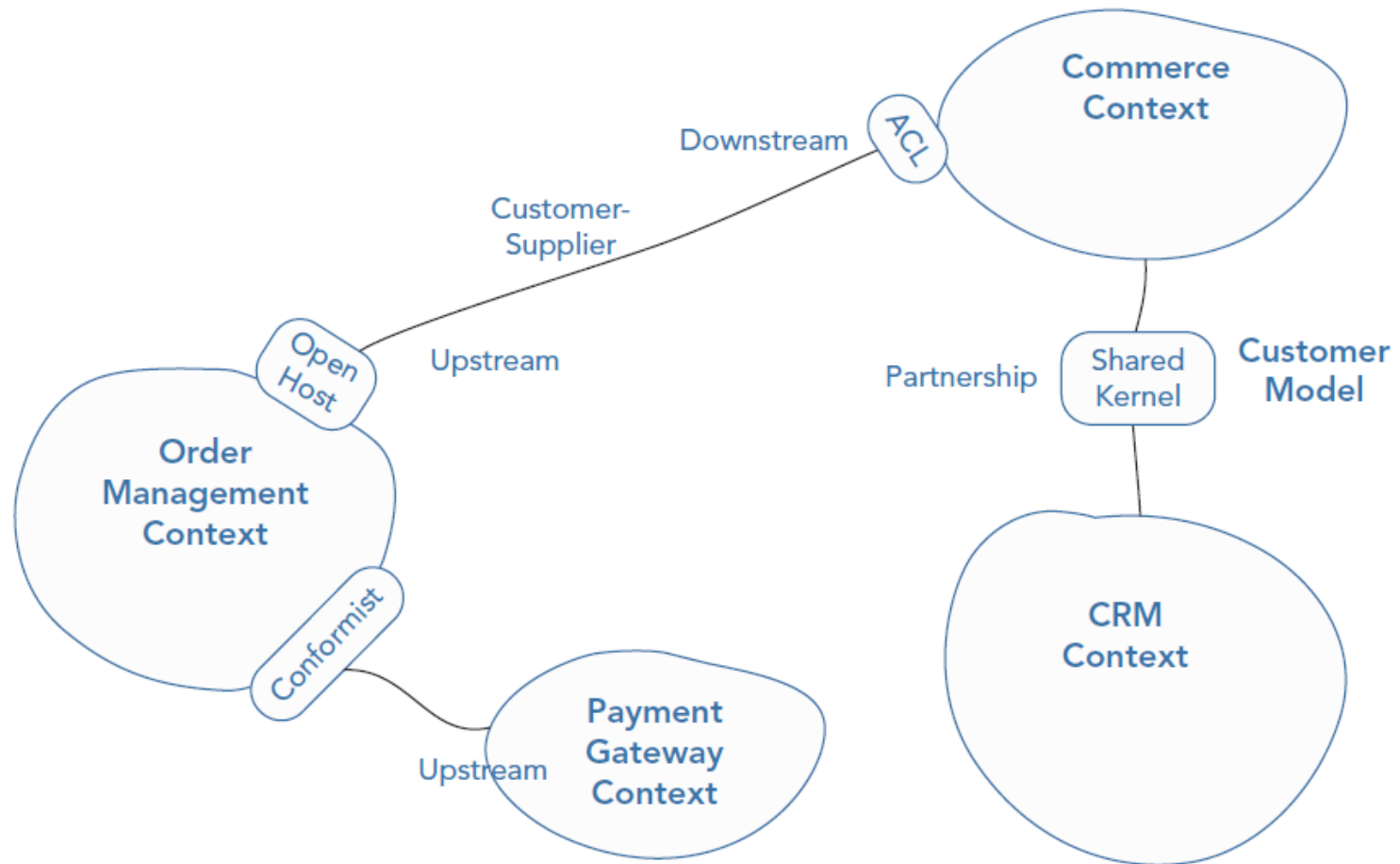
Customer-Supplier



Conformist

- Modell A passt sich vollständig an die Schnittstelle von Modell B an (und akzeptiert Korruption des eigenen Modells)
- Anwendbar wenn
 - Keine Team-Kollaboration mit Team B möglich ist
 - Implementierung einer ACL zu teuer/aufwendig ist

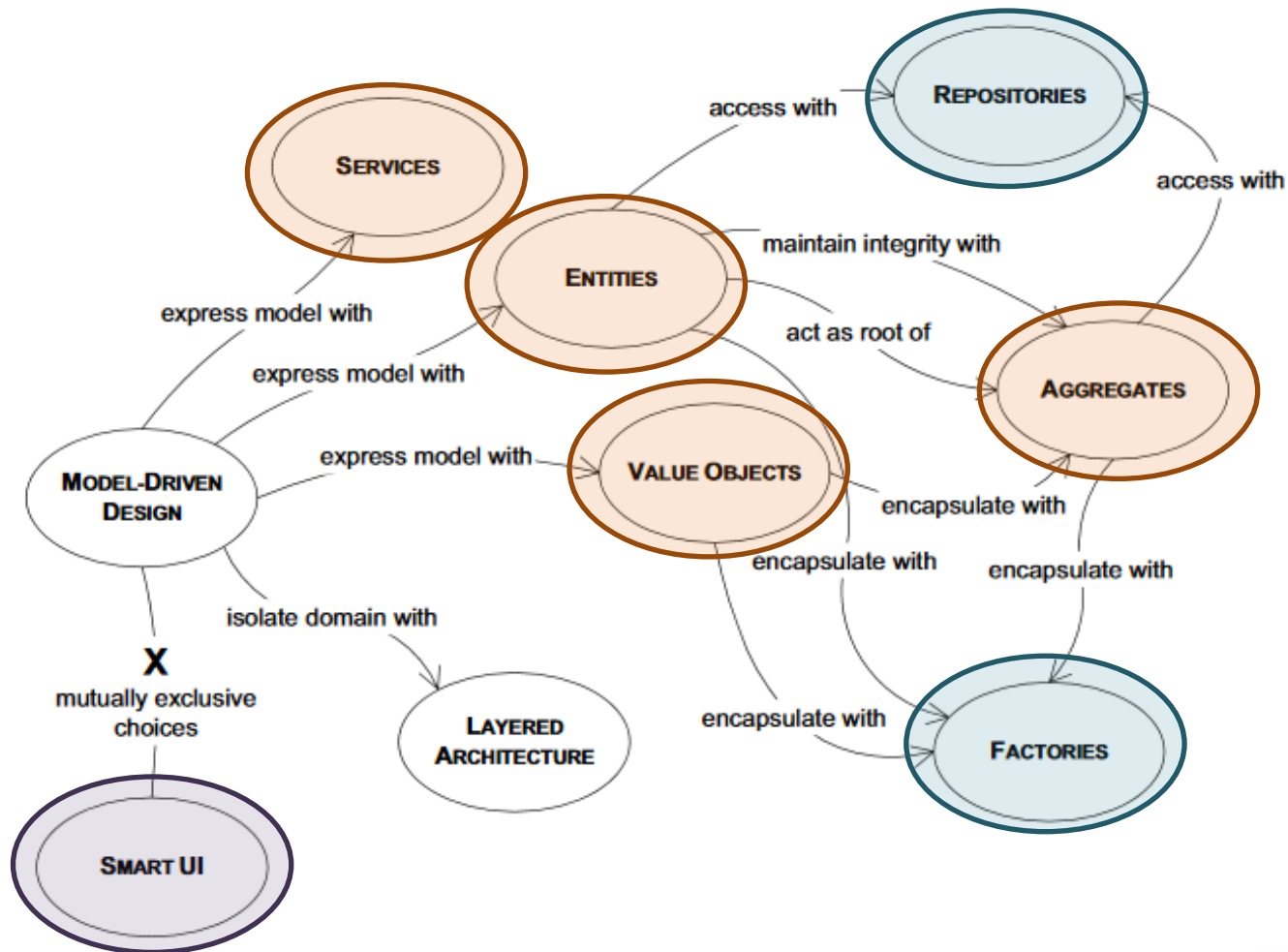
Vollständiges Beispiel



Teil 3: Taktisches DDD

- Der Zweck von DDD ist es, für eine Domäne, in der komplexe Regeln und Sachverhalte (Invarianten) gelten, ein Modell zu entwerfen, dass diese Komplexität beherrschbar macht und frei von „accidental complexity“ hält
- Taktisches DDD unterstützt beim Entwurf eines solchen Modells durch einen Katalog von Entwurfsmustern

Übersicht über taktische Entwurfsmuster
























Übersicht über taktische Entwurfsmuster

- **Entities, Value Objects, Domain Services, Aggregates:**
 - Sind der **Kern** des Domänen-Modells
 - Bilden den **Großteil der Geschäftslogik** ab
 - **Forcieren** die in der Domäne geltenden **Invarianten** und machen diese **sichtbar** (=„intention revealing“)
- **Repositories, Factories:**
 - Kapseln die Logik für das **Persistieren** und **Erzeugen** von Entities, Value Objects und Aggregates
 - Halten das Modell frei von „accidental complexity“

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 nennen wir „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
- Vgl. mit „Anemic Domain Model“

SmartUI - Beispiel

List of Students							
<div> <div>Add New +</div> <div>Export</div> <div>Custom Filter</div> <div>Hide Column</div> <div>List</div> <div>Grid</div> </div> <div>Search in 10 items...</div>							
#	Name ▼	Address	Phone	Email	Photo	Active	Edit / Remove
1	Valarie Martin	322 Columbia Place, Hollymead, Washington, 2672	+91 (836) 514-2729	horton.dotson@ovium.us		A	 
2	Powell Mcleod	178 Douglass Street, Cornucopia, Iowa, 2832	+91 (912) 406-2684	mcclure.langley@exospeed.io		A	 
3	Merle Turner	814 Ocean Avenue, Waikele, American Samoa, 283	+91 (823) 541-2850	cobb.tucker@stralum.tv		A	 
4	Mccarty Gallagher	905 Grattan Street, Delwood, Federated States Of Micronesia, 3524	+91 (886) 406-3225	melinda.kirby@extragen.com		I	 
5	Maxwell Manning	280 Union Street, Somerset, South Carolina, 3846	+91 (993) 417-3232	puckett.mitchell@equitax.ca		A	 
6	Gill Davenport	944 Randolph Street, Jacksonwald, Ohio, 2622	+91 (808) 591-3020	regina.landry@zaggle.biz		A	 
7	Estelle Vance	708 Whitwell Place, Lupton, Montana, 6258	+91 (830) 407-2654	raymond.edwards@liquicom.info		I	 

Value Objects

- Value Objects (VO) sind einfache Objekte **ohne eigene Identität**
- VO sind **unveränderlich (immutable)**
- Ein VO kapselt ein „Wertkonzept“ und wird nur durch seine **Eigenschaften** oder **Werte** beschrieben -> daher: **Value Object**
- Daraus folgt: zwei VO sind **gleich**, wenn sie die **selben Werte** haben

Beispiel Value Objects: Gewicht

- Jedes Gewicht besteht aus einer Zahl und einer Einheit
 - beispielsweise 78 Kilogramm
- Was ist schwerer? 1 Kilogramm Eisen oder 1 Kilogramm Federn?
 - Zwei Gewichte sind gleich, wenn ihre Zahlen und Einheiten übereinstimmen (evtl. umrechnen)
- 78 Kilogramm waren schon immer 78 Kilogramm und werden es auch immer sein
 - Kein erkennbarer Lebenszyklus

Vorteile von Unveränderlichkeit

- Wenn ein Objekt gültig konstruiert wurde, kann es danach nicht ungültig werden
->Einhalten von Invarianten der Domäne im Code wird sehr einfach
- Unveränderliche Objekte sind frei von Seiteneffekten
-> Code ist weniger anfällig für ungewolltes Verhalten

Umsetzung von Unveränderlichkeit im Code

- Keine Änderungen am Objekt selbst möglich (auch nicht durch Vererbung)
- Änderungen nur durch Konstruktion eines neuen Objektes möglich
- Gleichheit muss auf Attributen/Werten der verglichenen Objekte basieren
 - In Java muss daher beispielsweise `equals()` und `hashCode()` überschrieben werden!

Wie erkenne ich ein Value Object?

- VO sind oft ein ganzheitliches Konzept
 - Betrag + Währung: **Money**
 - Straße + PLZ + Stadt: **Adresse**
 - Titel, Anrede, Vorname, Nachname: **Name**
- VO **messen, begrenzen** oder **beschreiben** eine Sache näher
- Beispiele: Geldbeträge, Datum, Zeitperioden, Maße (Länge, Breite, ...), Koordinaten, Farbe, Email-Adresse, Tel.-Nr. (Landesvorwahl, Ortsvorwahl, ...)

Vorteile von Value Objects

- Verbessern die Deutlichkeit und Verständlichkeit durch Modellierung von fachlichen Domänenkonzepten
- Kapseln Verhalten und Regeln
- Unveränderlich (frei von Seiteneffekten , beispielsweise Aliasing)
- Selbst-Validierend
- Leicht testbar

Implementierung von Value Objects in Java

1. Klasse ist „final“ deklariert (Vererbung unterdrücken)
2. Alle Felder sind „blank final“ deklariert
3. equals() und hashCode() sind passend überschrieben
4. Ist nach Konstruktion in gültigem Zustand (andernfalls muss Konstruktion fehlschlagen)
5. Keine Setter oder andere Methoden, durch die Felder geändert werden können
6. Alle Methoden mit Rückgabewert liefern entweder:
 - a. Unveränderliche Rückgabewerte (immutable) oder
 - b. Defensive Kopien

Siehe auch <http://www.javapractices.com/topic/TopicAction.do?Id=29>

Entities

Eine Entity unterscheidet sich in drei wesentlichen Punkten von einem VO:

1. Sie hat eine eindeutige ID innerhalb der Domäne
2. Zwei Entities sind verschieden, wenn sie verschiedene IDs haben; ihre Eigenschaften sind unerheblich
3. Eine Entity hat einen Lebenszyklus und verändert sich während ihrer Lebenszeit

Allgemeine Regeln für Entities

- Wie auch VO sollen Entities die Einhaltung der für sie geltenden Domänenregeln (Invarianten) forcieren:
 - Es darf nicht möglich sein, eine Entity mit ungültigen Werten zu erzeugen
 - Es darf nicht möglich sein, eine Entity nach Konstruktion in einen ungültigen Zustand zu versetzen

Allgemeine Regeln für Entities

- Entities sollten so viel Verhalten wie möglich in VO auslagern
- (mindestens) die öffentlichen Methoden einer Entity sollten Verhalten beschreiben und nicht nur einfache getter/setter darstellen

Strategien für einzigartige Identitäten

- Es gibt **mehrere Strategien**, um eine Entity eindeutig identifizierbar zu machen
- Jede Strategie hat mehr oder weniger ausgeprägte **Vorteile** und **Nachteile**
- Die jeweils passende Strategie hängt (wie immer) **von den Anforderungen** ab
- Es spricht nichts dagegen, **mehrere Strategien** in einer Anwendung zu verwenden
- Grundsätzliche Unterscheidung: **natürliche Schlüssel** und **Surrogatschlüssel**

„Natürliche“ Schlüssel als Identität

Beispiele:



KFZ-Kennzeichen



ISBN



Personalausweis-
Nummer

Vorlesungen TINF13B4

Kurs-Name

	Mi 30.09.	Do 01.10.	Fr 02.10.
5	08:30 -11:45 Kommunikations und		08:30 -09:15 Info Veranstaltung

„Natürliche“ Schlüssel als Identität

Vorteile:

- Sehr **aussagekräftig**
- Keine Gefahr von Duplikaten **wenn global eindeutig**

```
public class Book {  
    private ISBN isbn;  
  
    public Book(ISBN isbn) {  
        this.isbn = isbn;  
    }  
}
```

Nachteile:

- **Fremdbestimmt** (wird sich das Format der ISBN *wirklich* niemals ändern?)
- Ggf. **nicht global eindeutig**, sondern kontextabhängig (Kurs TINF13B4 existiert an DHBW KA – was ist mit DHBW Stuttgart?)

Selbst generierte Surrogatschlüssel

Möglichkeiten:

1. Universally Unique Identifier (UUID)
2. Eigener, inkrementeller Zähler
3. Eigenes String-Format basierend auf Entity-Eigenschaften

UUID

- UUID kann jederzeit generiert werden
- Bietet diverse Implementierungen zur Generierung

```
UUID.randomUUID().toString();  
//4a96b5ba-c5d9-40c3-bc07-e291c4cd256a
```

Möglichkeiten:

- UUID als String verwenden
- UUID in ValueObject kapseln
- UUID über Converter oder CustomUserType o.ä. speichern (siehe Persist. von VO)

UUID als String

```
public class Person {  
  
    @Id  
    private String uuid;  
  
    public Person(UUID uuid) {  
        super();  
        this.uuid = uuid.toString();  
    }  
}
```

UUID als ValueObject

```
@Embeddable
public final class PersonId {

    private final String uuid;

    public PersonId(String uuid) {
        super();
        uuid= uuid;
    }
}
```

```
public class Person {

    @EmbeddedId
    private PersonId personId;

    public Person(PersonId personId) {
        super();
        this.personId = personId;
    }
}
```

```
public class PersonRepository {

    public PersonId nextPersonId() {
        return new PersonId(UUID.randomUUID().toString())
    }
}
```

Vorteile und Nachteile der UUID

Vorteile:

- Jederzeit generierbar
- (sehr) sicher anwendungsübergreifend eindeutig

Nachteile:

- Nicht sprechend
- Ggf. Performanceprobleme bei großen Datenmengen

<http://blog.codinghorror.com/primary-keys-ids-versus-guids/>

Eigener inkrementeller Zähler

Die Anwendung verwaltet einen /mehrere eigenen Zähler.

Vorteil:

- Eigenständige, unabhängige ID-Generierung
- ID steht sofort fest (early id generation)
- Nicht aussagekräftig (einfache Nummer)

Nachteil:

- Nicht sprechend
- Zähler muss irgendwo gespeichert werden
 - Zusätzliche Komplexität
 - Performance-Einbußen für Lesen/Schreiben der Zähler
 - **Daher besser UUID verwenden (gleicher Vorteile, weniger Nachteile)**

Eigenes String-Format

Die Id wird aus den Eigenschaften der Entity zusammengesetzt, Beispiel Fußballspiel:

„BAYERN-WOLFSBURG-VWARENA-27102015“

Vorteile:

- Sprechend
- Jederzeit generierbar

Nachteile:

- Hoher Aufwand, falls sich die Werte ändern, aus denen sich die ID zusammensetzt (Stadionumbenennung, ...)

Persistence-Provider-generierte Surrogatschlüssel

- Die meisten relevanten O/R-Mapper (JPA, Hibernate) bieten die Möglichkeit, automatisch eine ID zu generieren
- Meist stehen verschiedene Strategien zur ID-Generierung zur Verfügung
- In JPA: **Table, Sequence, Identity**

Persistence-Provider-generierte Surrogatschlüssel

Vorteile:

- ID ist eindeutig
- Kein eigener Aufwand

Nachteile:

- ID ist nicht sprechend
- ID steht normalerweise erst bereit, nachdem die Entity das erste Mal durch den O/R-Mapper gelaufen ist (persist oder commit)
- Abhängigkeit von O/R-Mapper
- Abhängigkeit von Datenbank (Identity, Sequence)

Die richtige Strategie für die Generierung von Identitäten wählen

- Selbst verwaltete IDs stehen sofort bereit (early ID generation)
- Dies erleichtert beispielsweise Tests, reduziert die Abhängigkeiten von der Persistenzschicht und erleichtert die Kommunikation in verteilten Systemen
- Allerdings muss sichergestellt sein, dass die ID-Generierung hinreichend eindeutige IDs erzeugt

Die richtige Strategie für die Generierung von Identitäten wählen

- Fremd verwaltete IDs (late ID generation) stehen meistens erst nach einem roundtrip zur DB zur Verfügung
- Dies erschwert Tests und die Kommunikation in verteilten Systemen
- Allerdings hat die Anwendung weniger Eigenverantwortung
- Ist ein funktionierender Standard-Weg

Value Objects 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 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-Modells besteht grundlegend aus
 - Value Objects – unveränderlichen Werten ohne Lebenszyklus oder Identität
 - Entities – veränderlichen Etwassen mit Identität und individuellem Lebenszyklus

Domain Service

- Der Begriff „Service“ ist leider mehrdeutig
 - Service-orientierte Architektur
 - Application Service
 - ...
- **All das sind keine Domain Services**
- Ein Domain Service ist ein kleiner „Helfer“ innerhalb des Domänenmodells



Domain Service

- Ein Domain Service hat zwei Haupt-Einsatzzwecke:
 1. Abbildung von komplexem Verhalten, Prozessen oder Regeln der Problemdomäne, die nicht in eindeutig einer bestimmten Entity oder einem bestimmten VO zugeordnet werden können
 2. Definition eines „Erfüllungs-Vertrages“ für externe Dienste, damit das Domänenmodell nicht mit unnötiger „accidental complexity“ belastet wird

Einsatz von Domain Services:

Abbildung von komplexem Verhalten

- Manchmal kann ein bestimmtes Verhalten oder eine bestimmte Regel nicht eindeutig einer Entity oder einem VO zugeordnet werden
- Beispiel:
 - Berechnung der Zahlungsmoral eines Kunden
 - Benötigt Kunde
 - Benötigt Rechnungen
 - Benötigt Kontenbewegungen

Domain Service

```
public class PaymentMoraleCalculator {  
  
    private final InvoiceRepository invoiceRepository;  
  
    private final AccountRepository accountRepository;  
  
    //...  
  
    public PaymentMorale calculatePaymentMoraleFor(Customer customer) {  
        List<Invoice> invoices = this.invoiceRepository  
                                .findInvoicesBy(customer.getId());  
        //...complex calculation  
        return paymentMorale;  
    }  
}
```


Einsatz von Domain Services: Definition eines Vertrags

- Die Domäne kann zur Erfüllung der Anforderungen auf externe Unterstützung angewiesen sein, beispielsweise durch einen Webservice, der von einer Fremdanwendung bereitgestellt wird

Einsatz von Domain Services:

Definition eines Vertrags

- Innerhalb des Domänenmodells kann dazu ein Domain Service als Vertrag (Interface) definiert werden
- Außerhalb des Domänenmodells kann dann ein „Dienstleister“ (beispielsweise in der Infrastruktur-Schicht – siehe Onion Architecture) diesen Vertrag implementieren und die benötigten Funktionen bereitstellen

Einsatz von Domain Services: Definition eines Vertrags

- **Beispiel: Schufa-Prüfung bei Kreditvergabe durch Webservice der Schufa**
 - Das technische Detail „Webservice“ ist für das Domänenmodell nicht relevant – es soll ja gerade frei sein von technischen Details
 - Wichtig ist nur:
 - „im Domänenmodell soll die **Kreditwürdigkeit** eines **Kunden** geprüft werden“

Einsatz von Domain Services: Definition eines Vertrags

Domänenmodell

```
public interface SchufaCheck {  
  
    CreditWorthiness checkCreditWorthinessOf(Customer customer);  
  
}
```

```
public class SoapSchufaCheck implements SchufaCheck {  
  
    @Override  
    public CreditWorthiness checkCreditWorthinessOf(Customer customer) {  
        //check via SOAP  
        return creditWorthiness;  
    }  
  
}
```

Infrastruktur-Schicht

Einsatz von Domain Services:

Definition eines Vertrags

- Durch die Definition eines Vertrages kann das Domänenmodell vorgeben, was für ein Ergebnis erwartet wird („CreditWorthiness „) und welche Daten es zur Erfüllung des Vertrages bereitstellt („Customer“)
- „fachlich“ ist dann alles relevante im Domänenmodell abgebildet – lediglich die technischen Details werden an eine Schicht außerhalb des Domänenmodells delegiert

Allgemeine Eigenschaften eines Domain Service

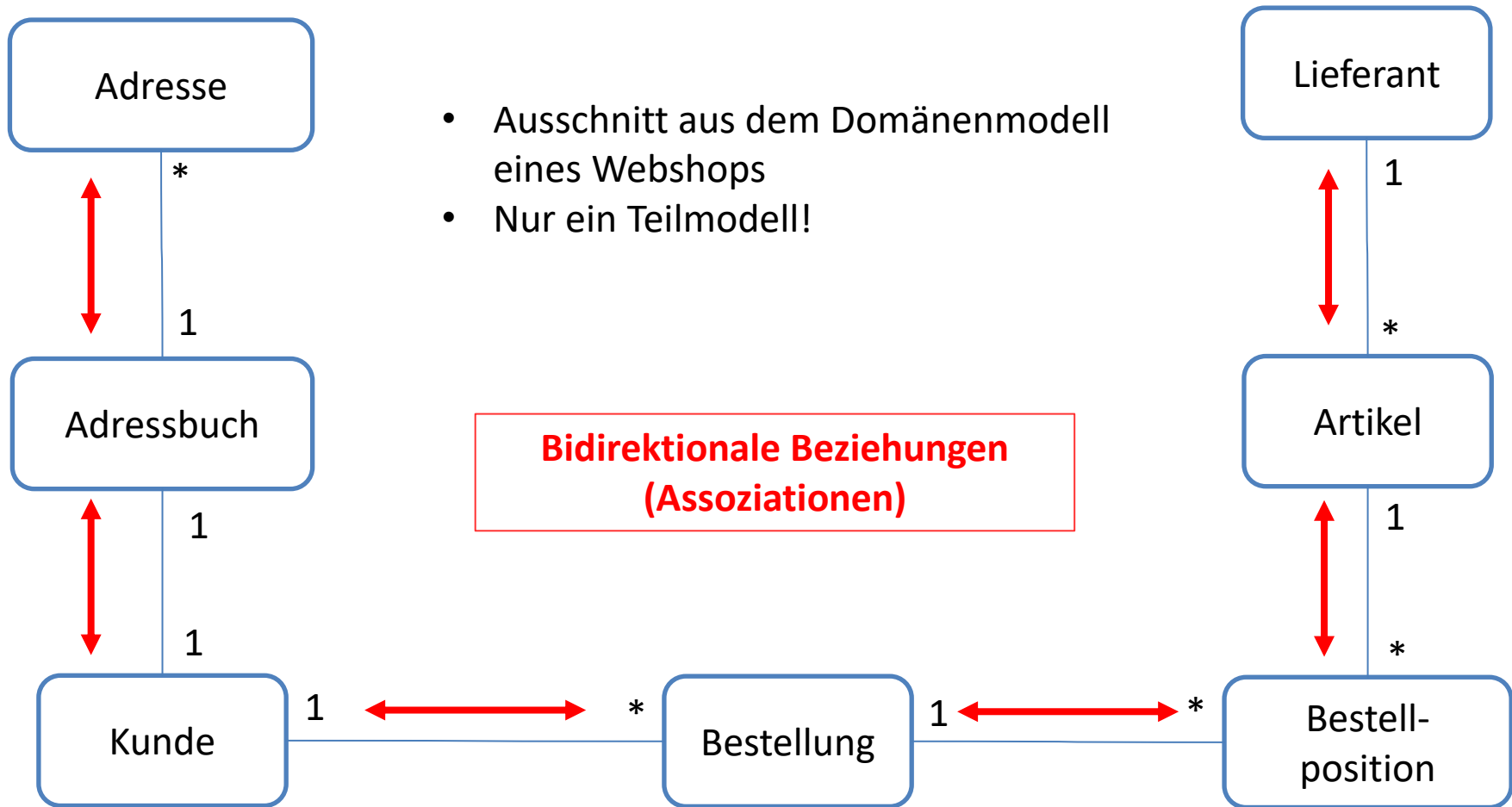
- Der Domain Service bezieht sich auf ein Domänenkonzept, das **nicht** natürlicherweise **Teil einer Entity oder eines Value Object** ist
- Die Schnittstelle (die Methodensignaturen) **verwendet die Begriffe des Domänenmodells**
 - Ein- und Ausgabeparameter sind Entities und VO
- Der Domain Service selbst ist **zustandlos**
 - Jede konkrete Instanz des Domain Service kann verwendet werden (alt oder neu)
 - Er darf aber (global sichtbare) Seiteneffekte haben

Aggregates

- Wenn die Domäne maßstabsgetreu modelliert wird, findet man viele Entities und VO, 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

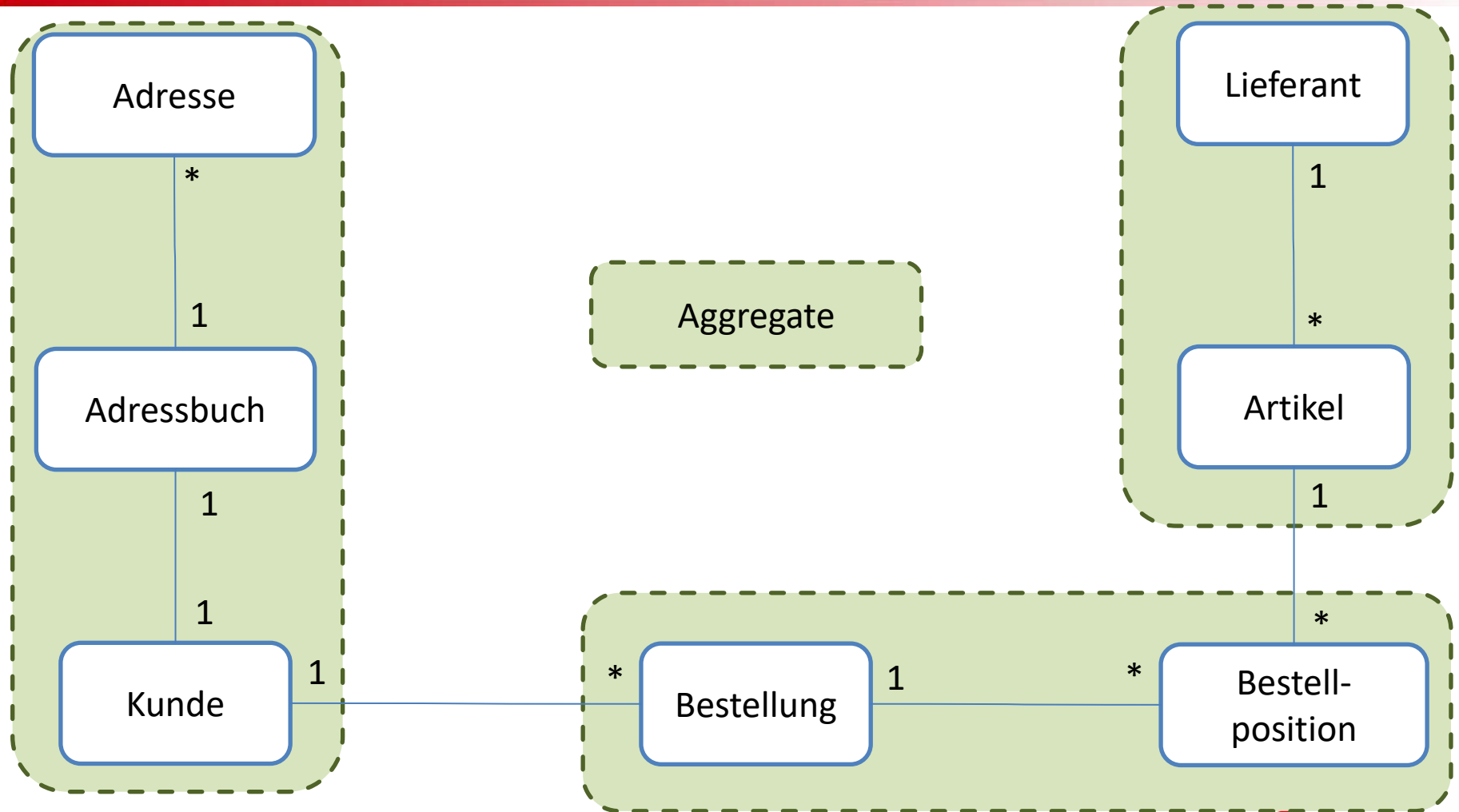
Beispiel eines Domänenmodells



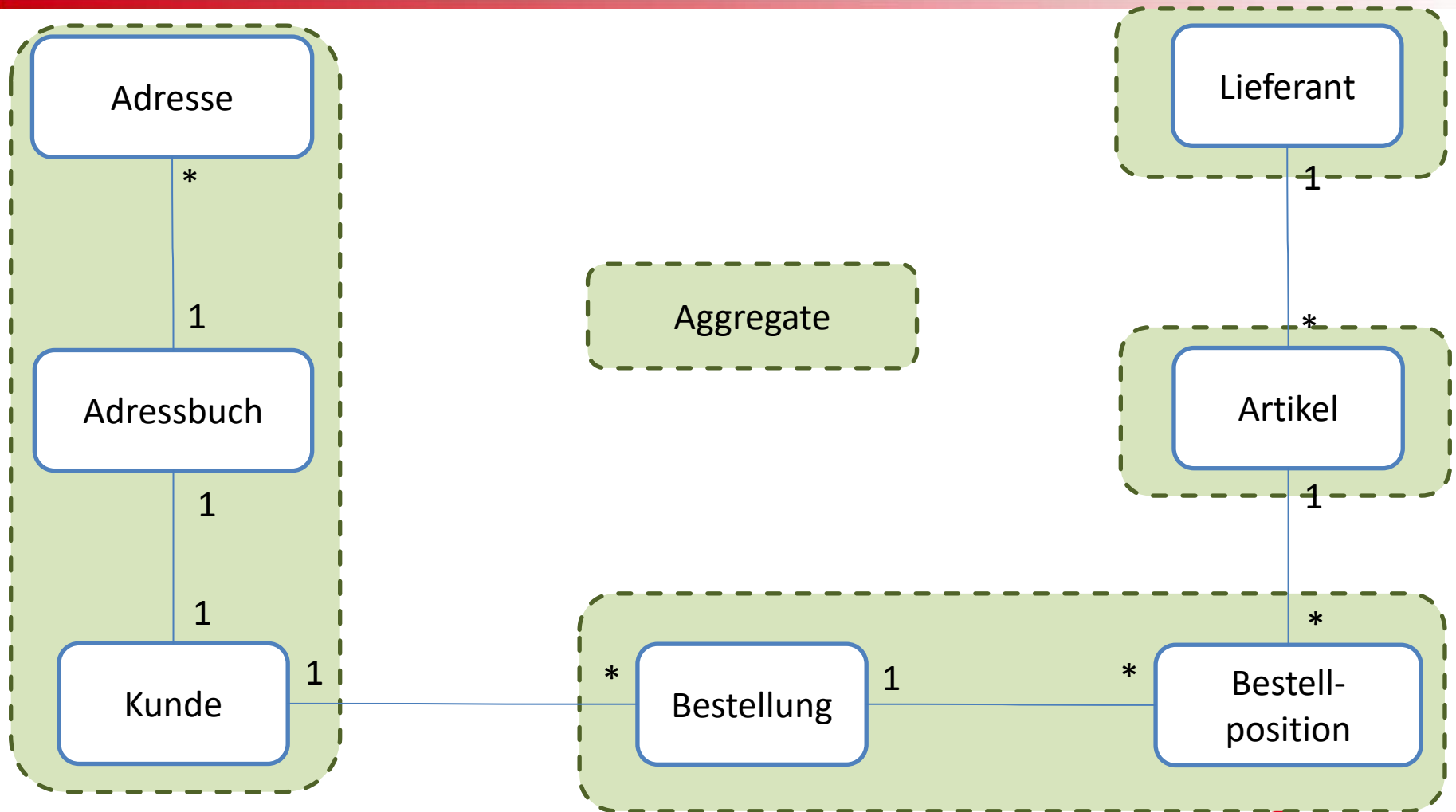
Aggregate zur Reduktion

- Aggregate gruppieren die Entities und VO 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 dürfen

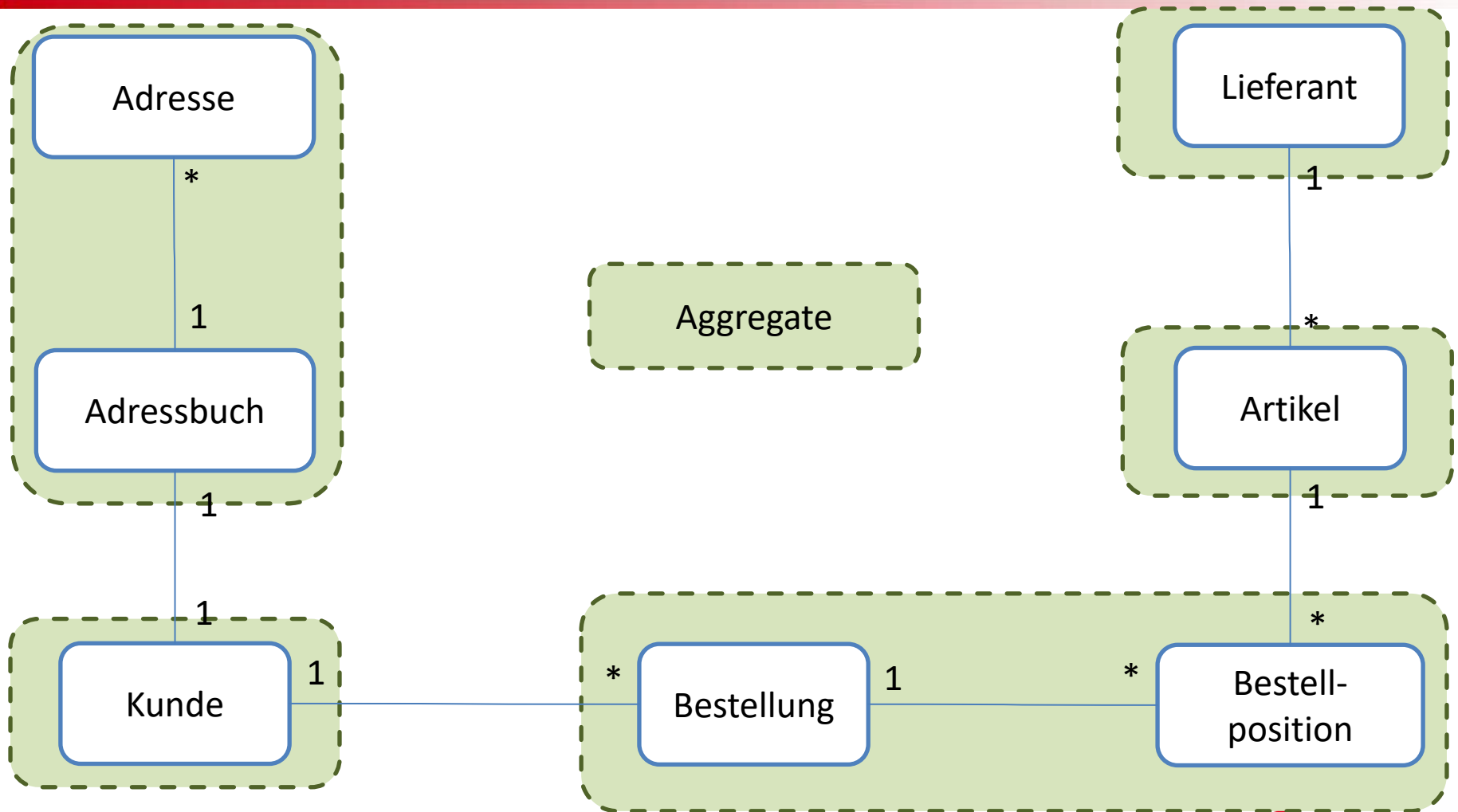
Aggregate im Beispiel



Aggregate im Beispiel – Variante 2

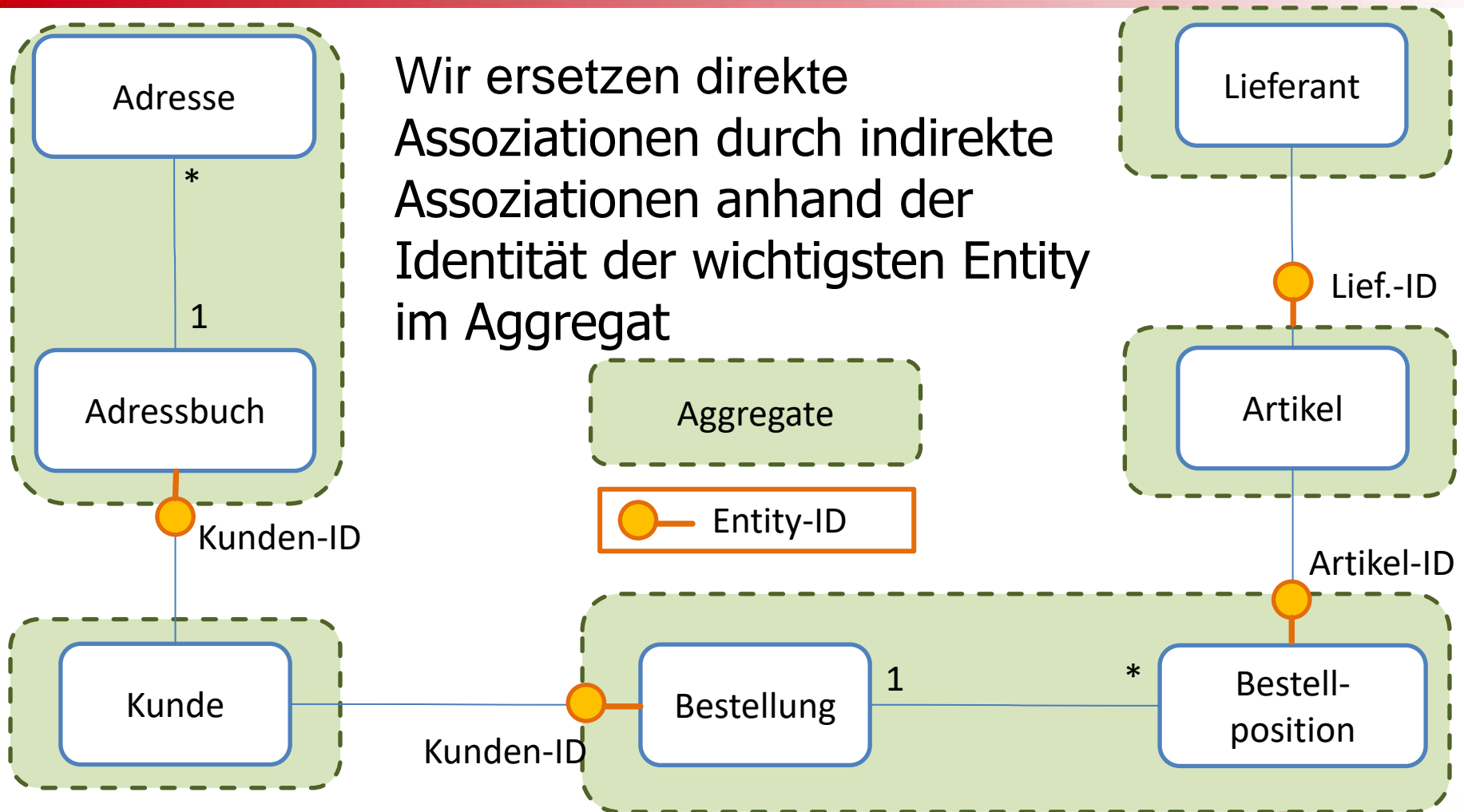


Aggregate im Beispiel – Variante 3



Assoziationen lösen

Wir ersetzen direkte Assoziationen durch indirekte Assoziationen anhand der Identität der wichtigsten Entity im Aggregat

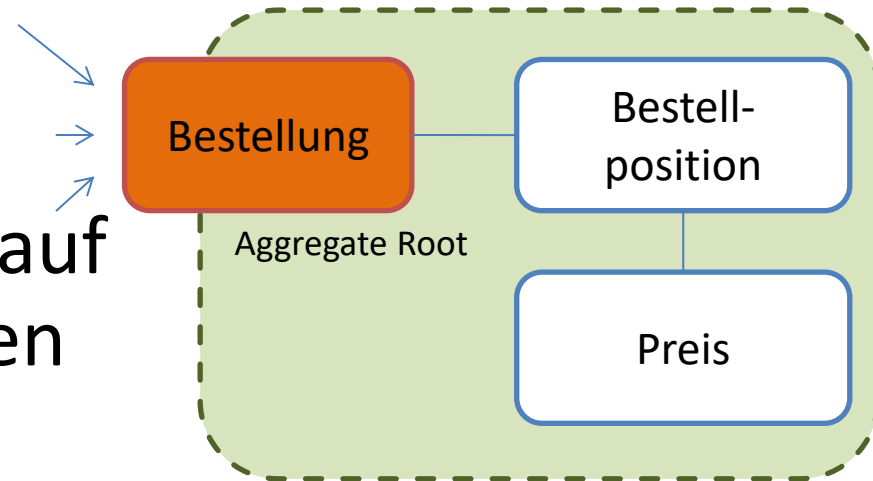


Aggregate Root Entity

- 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 Aggregat
- Langfristige direkte Referenzen auf innere Elemente sind nicht erlaubt
 - Nur temporäre Referenzen während einer Berechnung

Aggregate Root

- Eines pro Aggregat
- Das AR kann als eine Art „Türsteher“ alle Zugriffe auf das Aggregat kontrollieren
- Zentrale Stelle zur Überwachung der Domänenregeln
 - Beispiel: Der Gesamtpreis einer Erstbestellung darf nicht mehr als 100 € betragen
- aggregatsinterne Angelegenheiten bleiben „in der Familie“



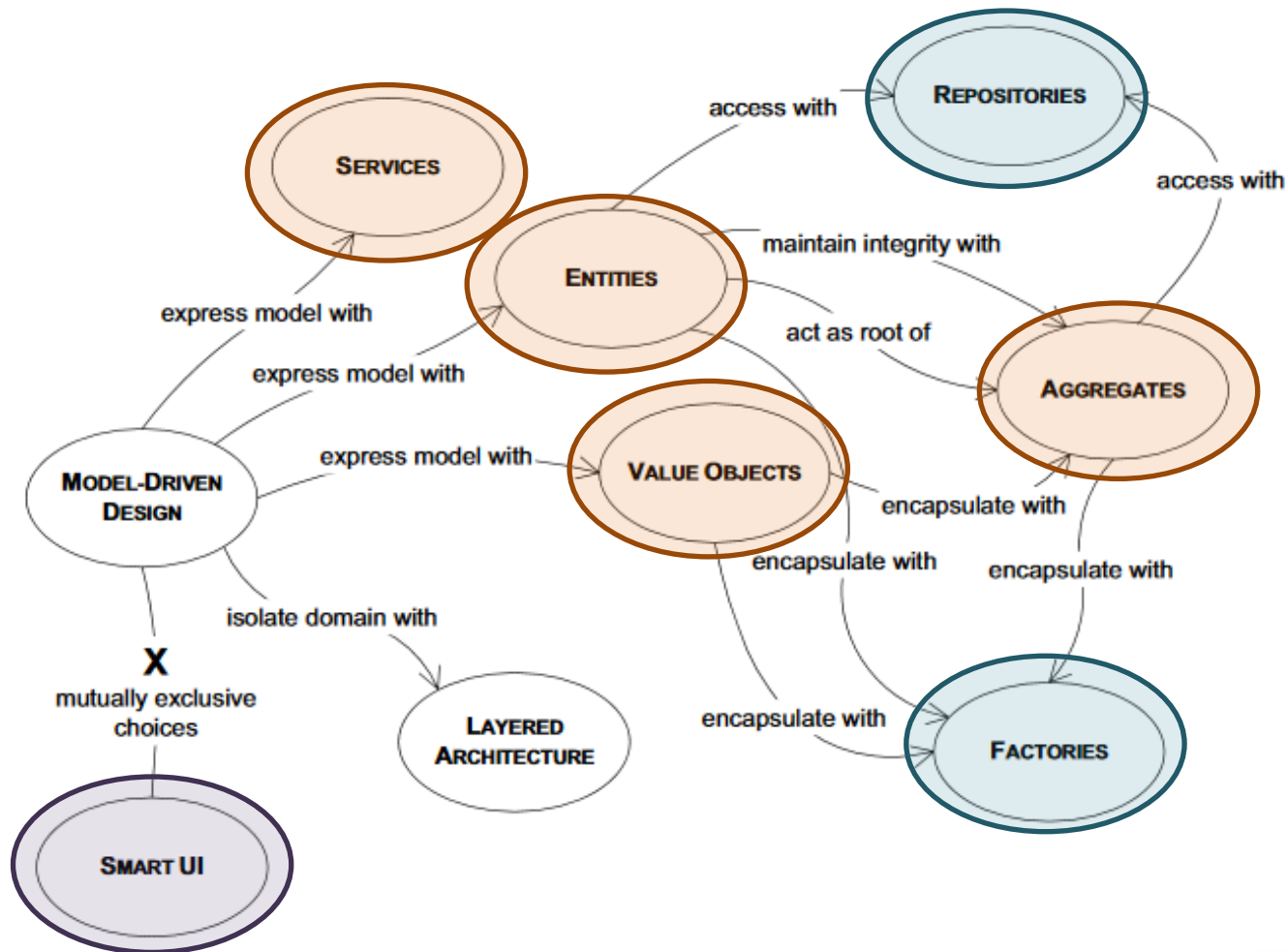
Aggregate und Außenwelt

- 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
 - Außenwelt muss eine Methode auf dem AR aufrufen, wenn der innere Zustand des AR verändert werden soll
- 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

Zusammenfassung Aggregate

- 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
- Aggregate
 - entkoppeln die Objektbeziehungen
 - bilden natürliche Transaktionsgrenzen
 - sichern Entity-/VO-übergreifende Domänenregeln zu
- **Aggregate sind mächtig, aber auch schwierig**

Übersicht über taktische Entwurfsmuster



Repositories

- Repositories vermitteln zwischen der Domäne und dem Datenmodell
- Sie stellen der Domäne Methoden bereit, um Aggregates aus dem Persistenzspeicher zu lesen, zu speichern und zu löschen
- Der konkrete technische Zugriff (accidental complexity) auf den Speicher (relationale DB, NoSQL, XML-Dateien usw.) wird vom Repository verborgen
- Dadurch bleibt die Domäne von technischen Details unbeeinflusst

Repositories

- Repositories arbeiten direkt mit Aggregates zusammen
 - je Aggregate existiert also typischerweise ein Repository
 - Repositories liefern immer die Aggregate Roots (und damit den Zugriff auf den Rest) zurück
- Die Definition der Repositories ist Teil des Domain Code
 - Die Implementierung findet „außerhalb“ statt

Repositories

- Die Methoden des Repository-Interface werden in der Sprache der Domäne benannt
- Der Klassenname 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!  
}
```

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 {
```

```
//...
```

```
Iterable<Buch> findeAlleIn(Erscheinungsjahr jahr);
```

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

```
}
```

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 Persistenzschicht effizienter umzusetzen

```
int aktuellerBuchbestand();
```

Zusatznutzen von Repositories

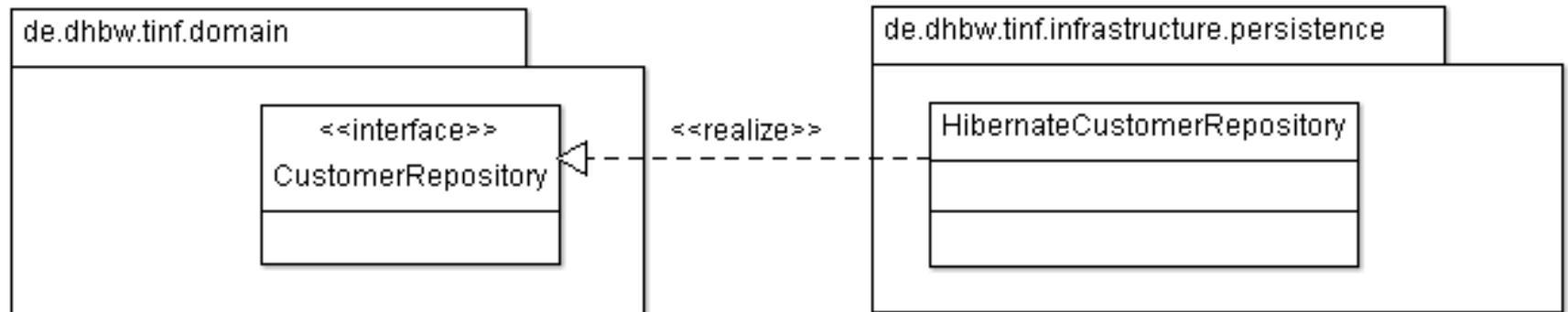
- Das Repository kann für die Erstellung von Identifikationen (IDs) für neue Root Entities zuständig sein

```
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

Implementierung eines Repositories

- Die Implementierung erfolgt normalerweise in einer technischen Schicht der Anwendung (DB, Infrastruktur, ...) und **nicht** innerhalb des Domänenmodells
 - Keine Vermischung von essential und accidental complexity

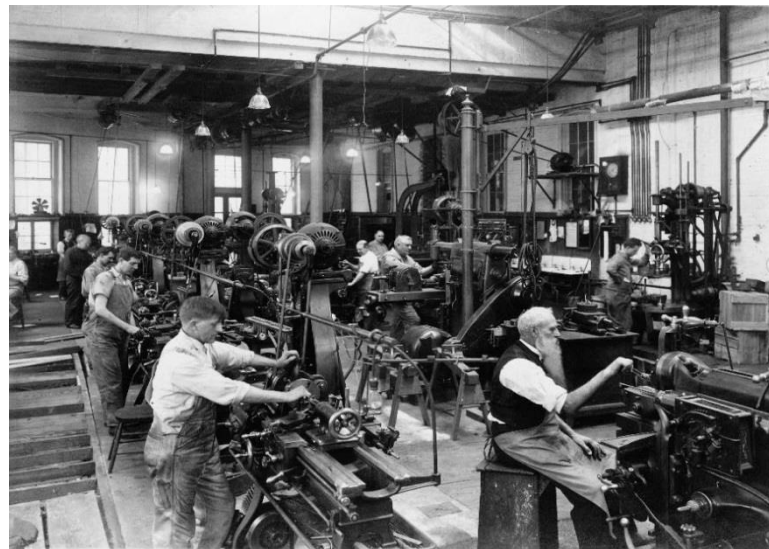


Zusammenfassung Repositories

- 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 Speichertechnologie 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 haben nur einen einzigen Zweck:
das Erzeugen von Objekten
- Factories sind ein **allgemein nützliches Konzept**, unabhängig von DDD



Factories

- Wenn die Logik für das Erzeugen einer Entity, eines Aggregates oder eines VO komplex wird, kann dies den eigentlichen Zweck des Objekts verschleiern (**Verletzung des Single-Responsibility-Prinzips**)
- Factories helfen, indem sie dem Objekt die Verantwortung für seine Konstruktion abnehmen; dadurch kann sich das Objekt auf sein Verhalten konzentrieren

Factory: mehrdeutiges Konzept

Der Begriff „Factory“ wird in OOP **mehrdeutig** verwendet. Es bezeichnet sowohl:

- a. Das **allgemeine Konzept** einer Factory:
 - irgendein Objekt oder irgendeine Methode zur Erzeugung anderer Objekte als Konstruktor-Ersatz
 - b. spezielle **Erzeugungsmuster**
 - Factory Method
 - Abstract Factory
- Im DDD meint man mit Factory normalerweise das „allgemeine Konzept“

Allgemeine Factory

- Allgemein ist eine Factory irgendein Objekt/ irgendeine Methode als **Konstruktor-Ersatz** (siehe unten)

```
public class Product {  
  
    private Product(Price price) {  
        super();  
        //initialise some fields...  
    }  
  
    public static Product createWithPrice(Price price) {  
        //checks, validation  
        return new Product(price);  
    }  
  
}
```

Zusammenfassung

- Um die Entities und Value Objects technisch sauber implementieren zu können, benötigen wir unterstützende Strukturen
- **Domain Services** enthalten Use Cases oder entkoppeln von Drittsystemen
- **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 Objektgraphen mit komplexen Konstruktionsregeln und entlasten den Konstruktor (Einhaltung Single Responsibility Principle)

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?

Beispiel für Domain Events

- Domäne: Sportsimulation für Fußball
- (Wahrscheinlich) domänenrelevante Ereignisse
 - Ball gepasst
 - Spieler gefoult
 - 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“
 - Analogie: 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

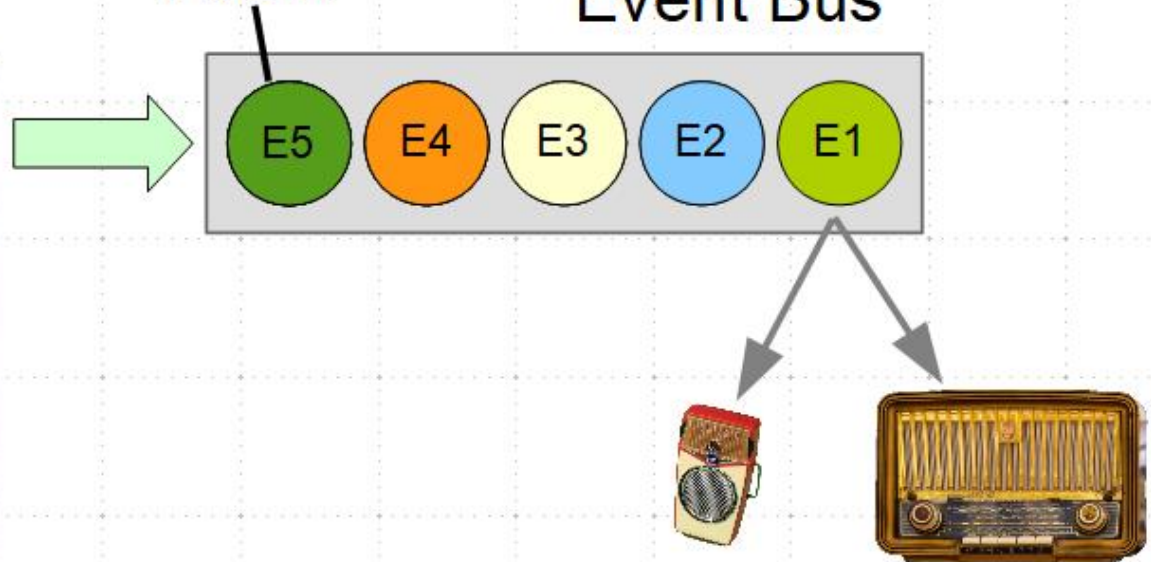
Domain Events und Event Bus

Event Source



(Domain)
Event

Event Bus



Subscribers

Zusammenfassung Domain Events

- 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), Kirks Sternentagebuch

Literatureempfehlungen

