Domain-Driven Design & Onion Architecture

„Oldies but Goldies“

DHBW

Teil 1: Einführung

- Was ist Design
- Warum braucht man Design
- Übersicht Domain-Driven Design

Was ist eigentlich „Design“

- Software ist im Grunde nichts anderes als ein abstraktes Modell, dass einen bestimmten Ausschnitt der Realität (die sog. **Problemdomäne** oder **Anwendungsdomäne**) beschreibt
- „Design“ ist, **wie** ein Modell die realen Gegebenheiten der Problemdomäne abstrahiert
- Design fängt an, sobald man das erste Mal mit einem Kunden spricht und gedanklich versucht, die Problemdomäne zu verstehen

Was ist eigentlich „Design“

Abbildung von Realität auf Modell

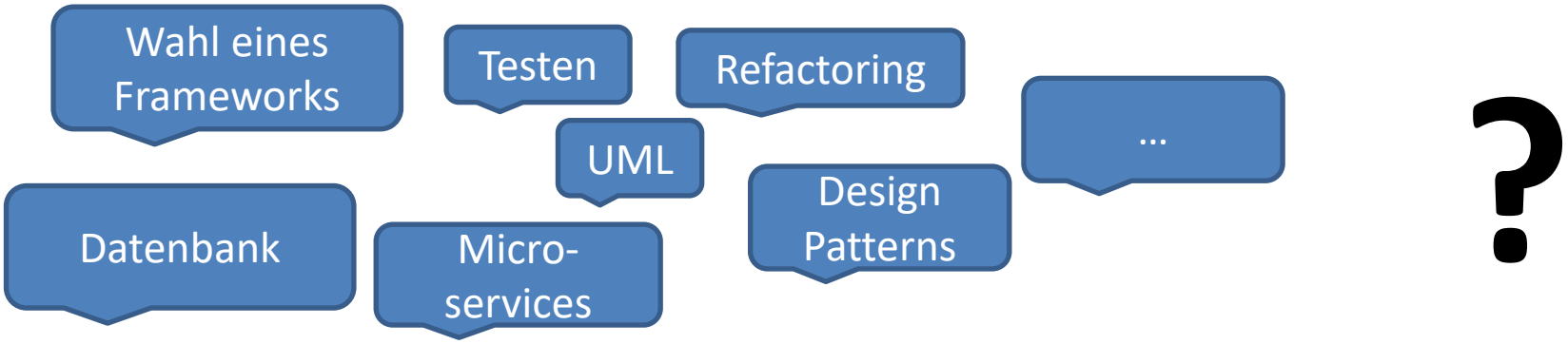
Problemdomäne

```
class Beer {

        String name; //Kräusen
        Brand brand; //Hoepfner

        //...
}
```

Modell

DHBW

Was ist eigentlich „Design“



Design ist die Summe aller bewussten und unbewussten Entscheidungen, die Einfluss darauf haben, wie ein Problem als Softwarelösung modelliert wird.

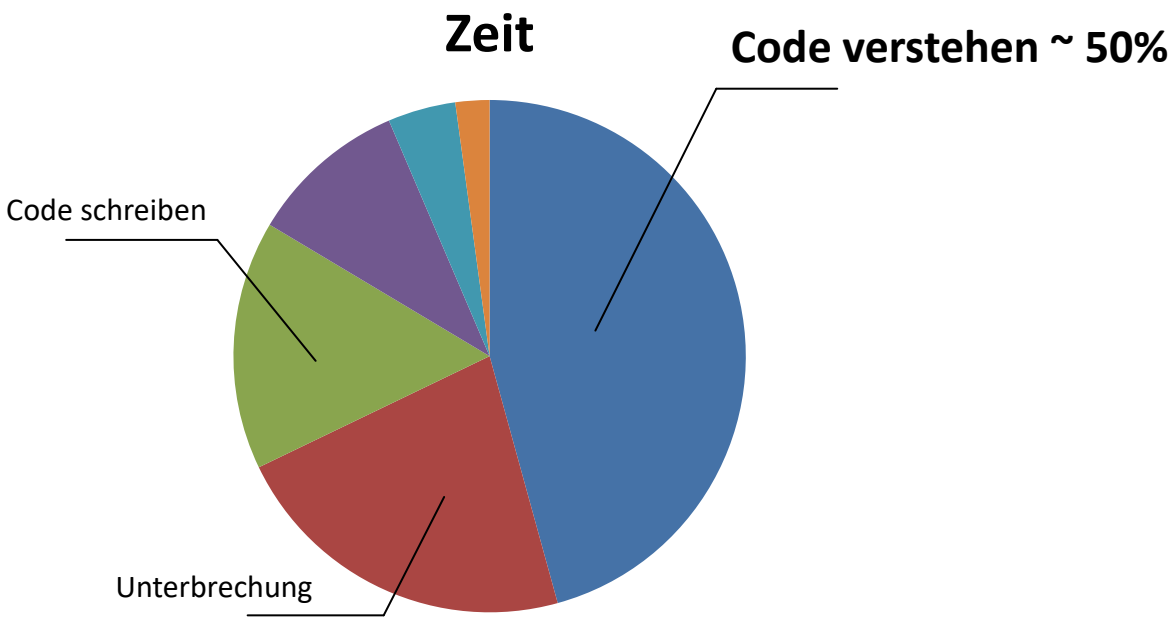
“The overwhelming problem with software development is that *everything* is part of the design process. Coding is design, testing and debugging are part of design, and what we typically call software design is still part of design.” [Reeves, 2005]

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

DHBW

Warum braucht man „Design“

Frage: Womit verbringen Entwickler den Großteil ihrer Zeit?



Ko et al., 2006

Warum braucht man „Design“

- Softwareentwicklung ist komplex
- Neben dem eigentlich zu lösenden Problem aus der Problemdomäne muss ein Entwickler viele verschiedene Nebeneinflüsse beachten, die mit dem Problem nichts zu tun haben, jedoch für dessen Lösung notwendig sind

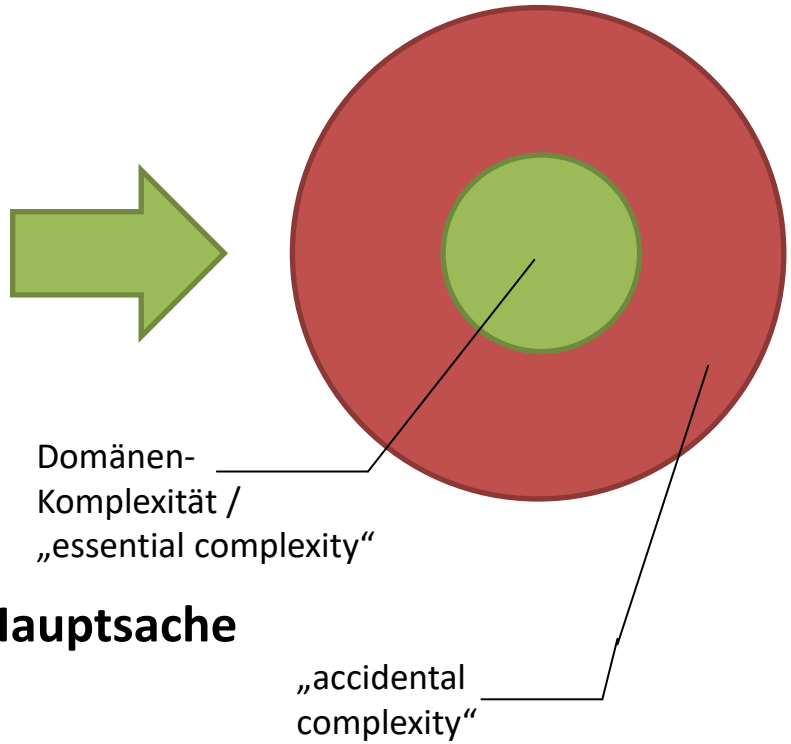
Beispiel für Problem und Nebeneinflüsse

- Siehe Vorlesung

Software-Komplexität

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

- Grund-komplexität der Domäne
- Anforderungen



- Legacy-Systeme
- Persistenz
- UI
- Framework
- Infrastruktur / Hardware
- Deployment
- Designfehler
- Architektur

Hauptsache

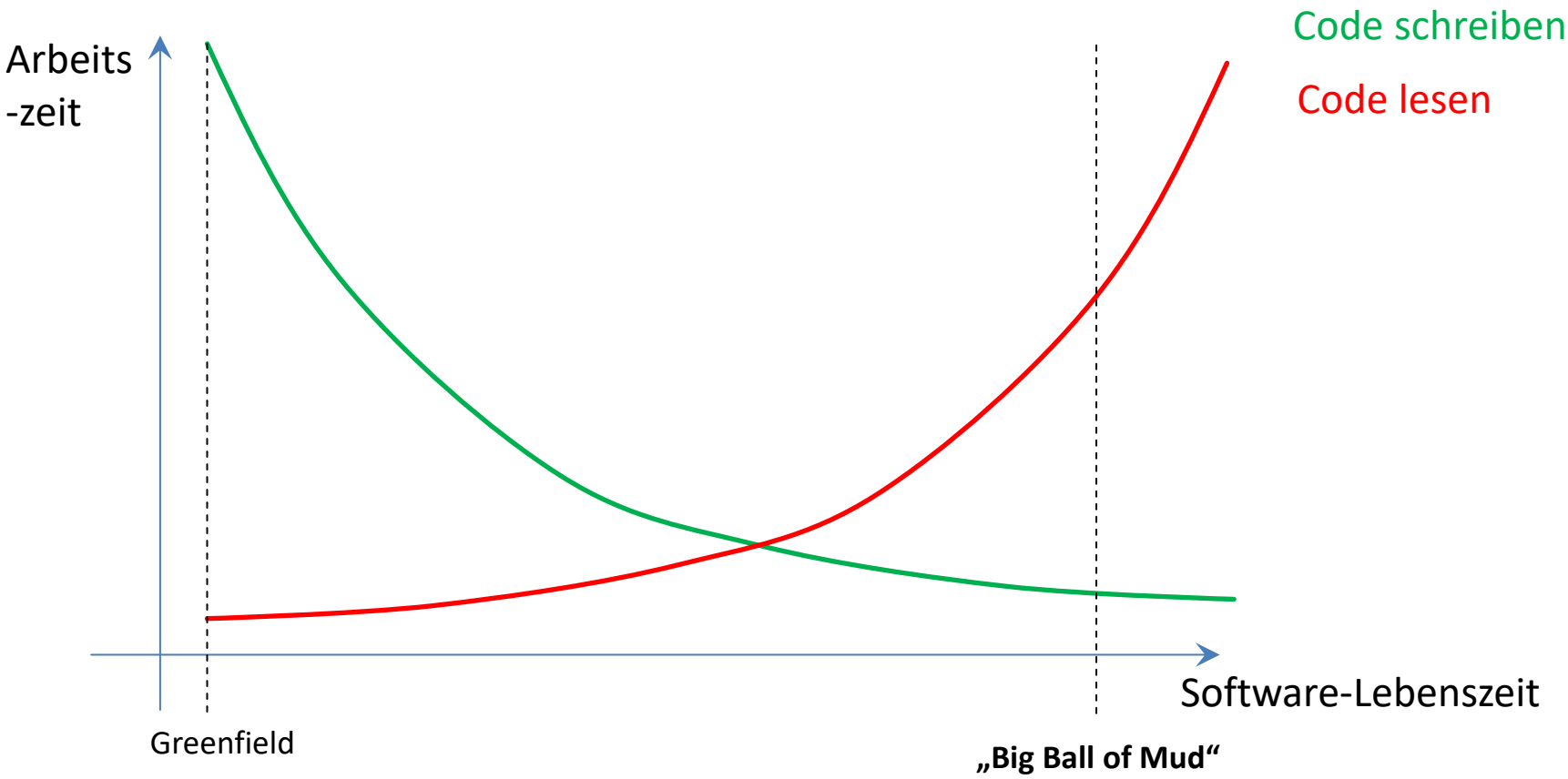
Nebensache

DHBW

Software-Komplexität

- Die **Komplexität der Problemdomäne** (essential complexity) ist gegeben; damit sollte sich ein Entwickler eigentlich beschäftigen
- die **Unfall-Komplexität** (accidential Komplexität) ist ein notwendiges Übel; man kann sie normalerweise nicht komplett eliminieren, aber möglichst verhindern, dass sie die Komplexität der Problemdomäne negativ beeinflusst

Auswirkungen von unbeherrschter Komplexität



Big Ball of Mud

- “code that does something useful, but without explaining how” [Evans, 2004]
- Kein erkennbares Design
  - Code gibt keinen Aufschluss über Intention (“was und warum”)
  - Technische Belange beeinflussen (“verunreinigen”) die Fachlogik
  - Fachlogik über gesamte Anwendung verteilt (“was geht kaputt wenn ich diese Stelle ändere”)

Resultat: Code ist schwer wartbar, schwer erweiterbar, kurz: **ein “Big Ball of Mud”**

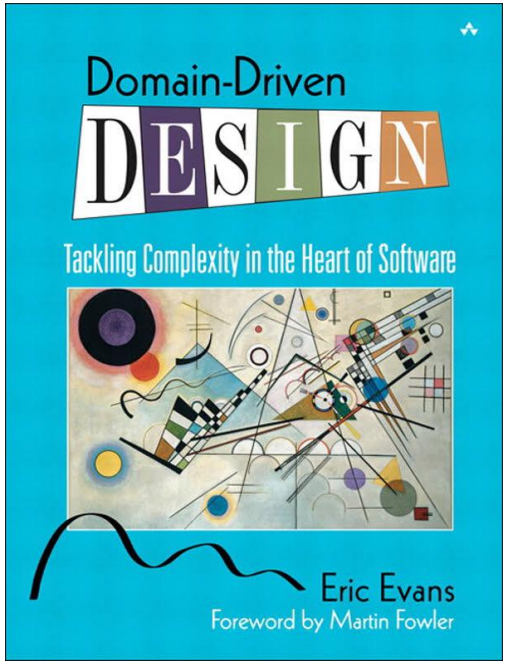
Warum braucht man „Design“

- Man kann keine „Einfachheit“ hinzufügen
- Man kann aber
  - nicht **unnötig** zusätzlich **verkomplizieren**
  - die **Auswirkungen von Komplexität begrenzen** und **kontrollieren** („divide and conquer“)
- **Design macht Komplexität beherrschbar**
  - Design (=Code) verdeutlicht, was die Anwendung tut und wie sie es tut
  - Design hilft, die „essential complexity“ nicht mit der „accidential complexity“ zu verseuchen
  - Design hilft, die Auswirkungen der „accidential complexity“ einzudämmen

DHBW

Domain-Driven Design

- Domain-Driven Design (DDD) ist eine Herangehensweise an die Modellierung von Software, die sich auf die Problemdomäne konzentriert
- Erstmals vorgestellt 2003 im „Big Blue Book“ von Eric Evans
- Grundlagenwerk mit weitreichenden Auswirkungen
  - Microservices
  - Domain-Specific Language (DSL)
  - …
- Wird noch heute interpretiert, diskutiert und erweitert



Übersicht: Domain-Driven Design

- Grundsätze:
  - Designentscheidungen werden von der **Fachlichkeit** und der **Fachlogik** der zugrunde liegenden Problemdomäne getrieben – nicht von technischen Details
  - Domänenexperten und Entwickler arbeiten eng zusammen und entwickeln eine **gemeinsame Domänensprache** – ein striktes, eindeutiges Vokabular, um über die Domäne zu sprechen und sie zu beschreiben
  - Die relevanten fachlichen Zusammenhänge der Problemdomäne werden durch ein **Domänenmodell** erfasst (d.h. sie spiegeln sich in diesem wieder)

Warnung

- DDD verwendet teilweise Begriffe, die esoterisch / dogmatisch bzw. etwas angestaubt oder altbacken klingen
- Dieses Problem haben aber viele Pionierwerke, die ein paar Jahre auf dem Buckel haben
- Viele heute übliche Praktiken, die selbstverständlich im Entwickler-Alltag verwendet werden, gehen auf DDD zurück oder wurden dadurch inspiriert

Strategie und Taktik

- Wie schafft man es nun, die „Fachlichkeit“ der Anwendungsdomäne in die Software zu übertragen?
- DDD liefert dafür eine Reihe von Techniken, Methoden und Mustern; diese werden gerne in „strategisches“ und „taktisches“ DDD getrennt
- Beide ergänzen und beeinflussen sich wechselseitig:
  - Schwierigkeiten bei der „taktischen“ Implementierung können auf mangelndes „strategisches“ Verständnis hinweisen

Strategisches DDD

- strategisches DDD beschäftigt sich mit dem Verständnis der Domäne und hilft beim Analysieren, Aufdecken, Abgrenzen, Dokumentieren und Begreifen der Fachlichkeit

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

Teil 2: Strategisches Domain-Driven Design

- Domäne
- Domänenmodell
- Ubiquitous Language
- Bounded Context

Domäne

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

Beispiele:

Raumfahrt, Logistik, Telekommunikation, Fertigung, eCommerce, …

“A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.” [Evans, 2004]

https://de.wikipedia.org/wiki/Problemdom%C3%A4ne, abgerufen am 03.10.2016

Domäne

Der Begriff „Domäne“ wird in der Software-Entwicklung sehr häufig verwendet – Beispiel:

„More than simply managing a collection of ORM-style objects, stores manage the application state for a particular **domain** within the application.”

https://facebook.github.io/flux/docs/overview.html

Domänenmodell

- Abstraktion, die bestimmte Teile der Problemdomäne beschreibt
- Erlaubt das Lösen von Problemen innerhalb der Problemdomäne
- Beschreibt „wie“ ein Problem mit Software gelöst wird

Domänenmodell

- Das Domänenmodell ist an keine bestimmte „Form“ gebunden – es kann in UML, Prosa oder – am wichtigsten - als Code vorliegen
- Wichtig ist bei DDD, dass die verschiedenen Darstellungsformen konsistent sind:
  - Wenn die Domänenexperten von „Kunde“ sprechen, muss es auch in UML oder Code das Konzept „Kunde“ geben

Domänenmodell

- Da jede Software ein Modell der zugrunde liegenden Problemdomäne ist, hat auch jede Software ein Domänenmodell
- Ein **ausdrucksstarkes** oder **reichhaltiges** Domänenmodell lässt Rückschlüsse auf die Regeln, Prozesse und Konzepte der Problemdomäne zu
- Ein **„anämisches“** Domänenmodell modelliert die Problemdomäne nur sehr oberflächlich und bildet lediglich Konzepte der Problemdomäne im Code ab, während Regeln und Prozesse häufig in prozeduralem Stil programmiert sind

- **Bei DDD geht es um die Entwicklung von ausdrucksstarken Domänenmodellen!**

DHBW

Anämisches Domänenmodell

```java
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String street;
    private String zipCode;
    private String city;

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    //...more of this crap
}
```

```java
public class PersonService {

    public void changeAddress(
            Long personId,
            String street,
            String zipCode,
            String city) {

        Person person =
            PersonDAO.findCustomer(personId);

        if(null == person) {
            //…
        }

        person.setStreet(street);
        person.setCity(city);
        //...
    }
}
```


DHBW

Anämisches Domänenmodell

- Das Domänenmodell dient im Grunde nur als dünne Barriere über der Datenschicht (Konzepte als Datenbank-Entitäten)
- Aufbau der Domänenobjekte orientiert sich an der Datenbank anstatt an der Problemdomäne
- Die Domänenobjekte (hier: Person) enthalten wenig oder keine Fachlogik
- führt zu einem prozeduralen Programmierstil
- meistens existiert in Anwendungen mit einem anämischen Domänenmodell eine „Service Layer“, welche die Fachlogik implementiert

„Businesses regularly put too much effort into developing glorified database table editors“ [Vernon, 2013]

Die Sprache der Domäne

- Damit ein Modell die Regeln, Prozesse und Konzepte (Fachlichkeit und Fachlogik) 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**

DHBW

Ubiquitous Language (UL)

- Domänenexperten verstehen normalerweise nur begrenzt die **technische Fachsprache**, mit der Entwickler über ein Modell sprechen
- Entwickler wiederum können meist nur wenig mit dem spezifischen **Fachjargon** anfangen, der von den Domänenexperten gesprochen wird
- Keine der beiden Sprachen eignet sich allein für das Projekt

Ubiquitous Language (UL)

- Diese **„Kluft“ im Verständnis** pflanzt sich in die Software fort; der Code entfernt sich von der Sprache der Domäne und wird damit schwerer verständlich

Ubiquitous Language (UL)

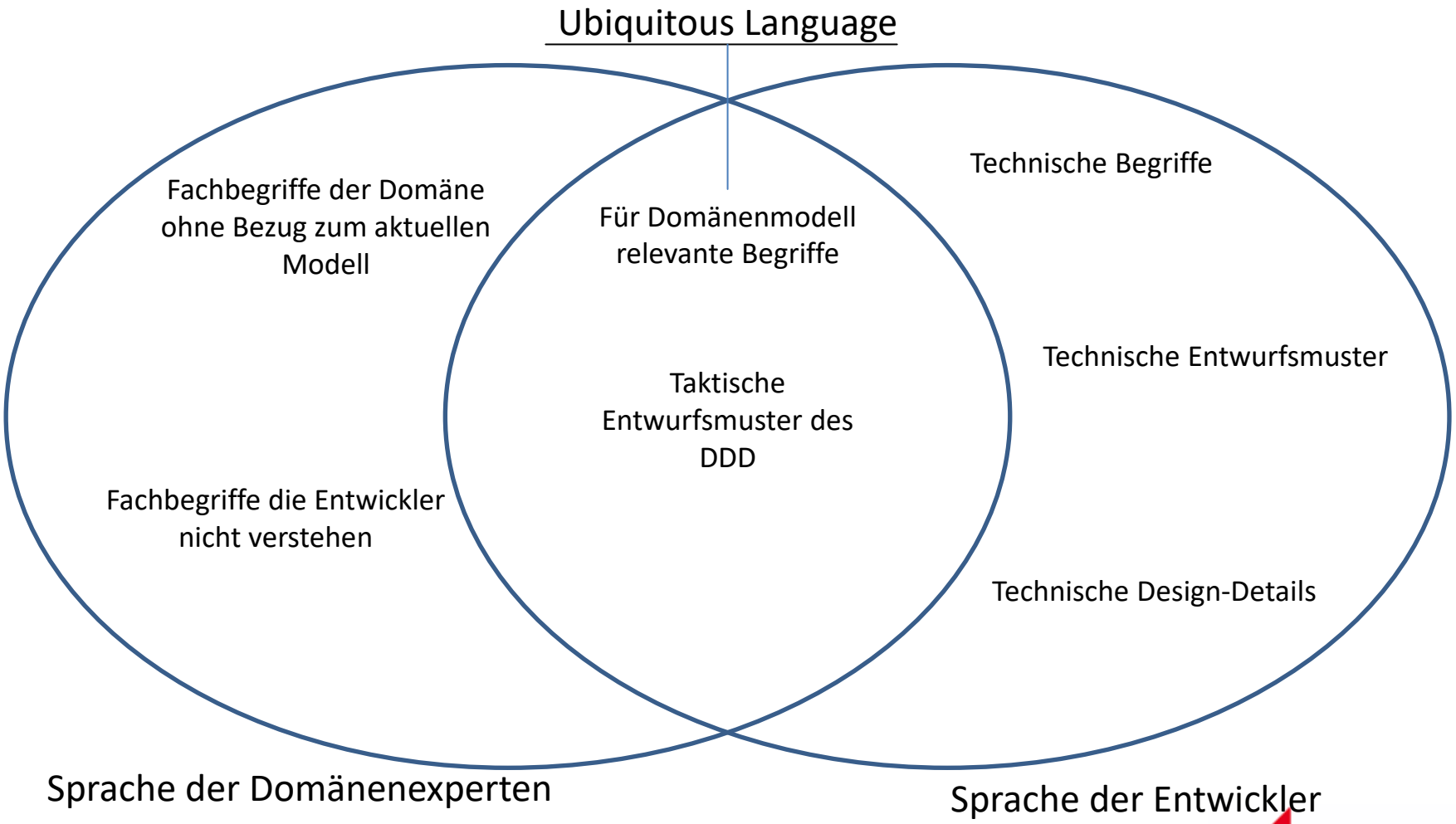
- Der Zweck der UL ist es, diese Kluft zu schließen, indem Domänenexperten und Entwickler eine gemeinsame Sprache entwickeln, welche:
  - Für die Problemlösung relevante Konzepte, Prozesse und Regeln der Domäne erfasst
  - Zusammenhänge verdeutlicht
  - Mehrdeutigkeiten und Unklarheiten beseitigt

Ubiquitous Language (UL)

Beispiel für Mehrdeutigkeit:

In einer eCommerce-Applikation kann der Begriff „Artikel“ sowohl als ein kaufbares Produkt – beispielsweise eine CD – als auch als „Blog-Artikel“ interpretiert werden

Ubiquitous Language (UL)



DHBW

Wie kommt man zu einer gemeinsamen Sprache?

- Kollaboration zwischen Domänenexperten und Entwicklern (agile, anyone?)
- Den Domänenexperten genau zuhören
- Kritisch nachfragen:
  - „Warum darf dieser Button nur ein mal gedrückt werden?“
  - „Was bedeutet ‚der Kunde wird gesperrt‘?“
- Iterativer Prozess – die UL ist lebendig

DHBW

Wie kommt man zu einer gemeinsamen Sprache?

- Bedeutet nicht, dass man für ein Projekt eine ein eigenes Wörterbuch erstellt (obwohl ein Glossar in komplexen Projekten durchaus Sinn machen kann)
- Sondern die Sprache der Domänenexperten bewusst hört und hinterfragt

Aufteilung der Domäne

- Normalerweise ist es nicht sinnvoll, ein einziges, großes Modell für die gesamte Problemdomäne zu entwerfen
- Weil DDD ein aufwendiges Verfahren ist, sollte die betrachtete Problemdomäne, für die ein Modell entworfen wird, so klein wie möglich sein
- DDD führt dazu das Konzept **Subdomäne** ein
- Subdomänen unterteilen die Problemdomäne in K**ern-Domäne**, **unterstützenden Domänen** und **generischen Domäne**

DHBW

Kern-Domäne

- Beschreibt den Teil der Problemdomäne, der für den Auftraggeber am wichtigsten ist
- Normalerweise das Kerngeschäft eines Unternehmens („wo wird das Geld erwirtschaftet“) oder wo ein Wettbewerbsvorteil besteht
- Für diesen geschäftskritischen Teil der Problemdomäne lohnt sich die Entwicklung eines reichhaltigen Modells mit DDD

Kern-Domäne

- 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.”

Beispiel Kern-Domäne

Problemdomäne: Online-Auktionshaus

Kerndomäne „Auktionen“

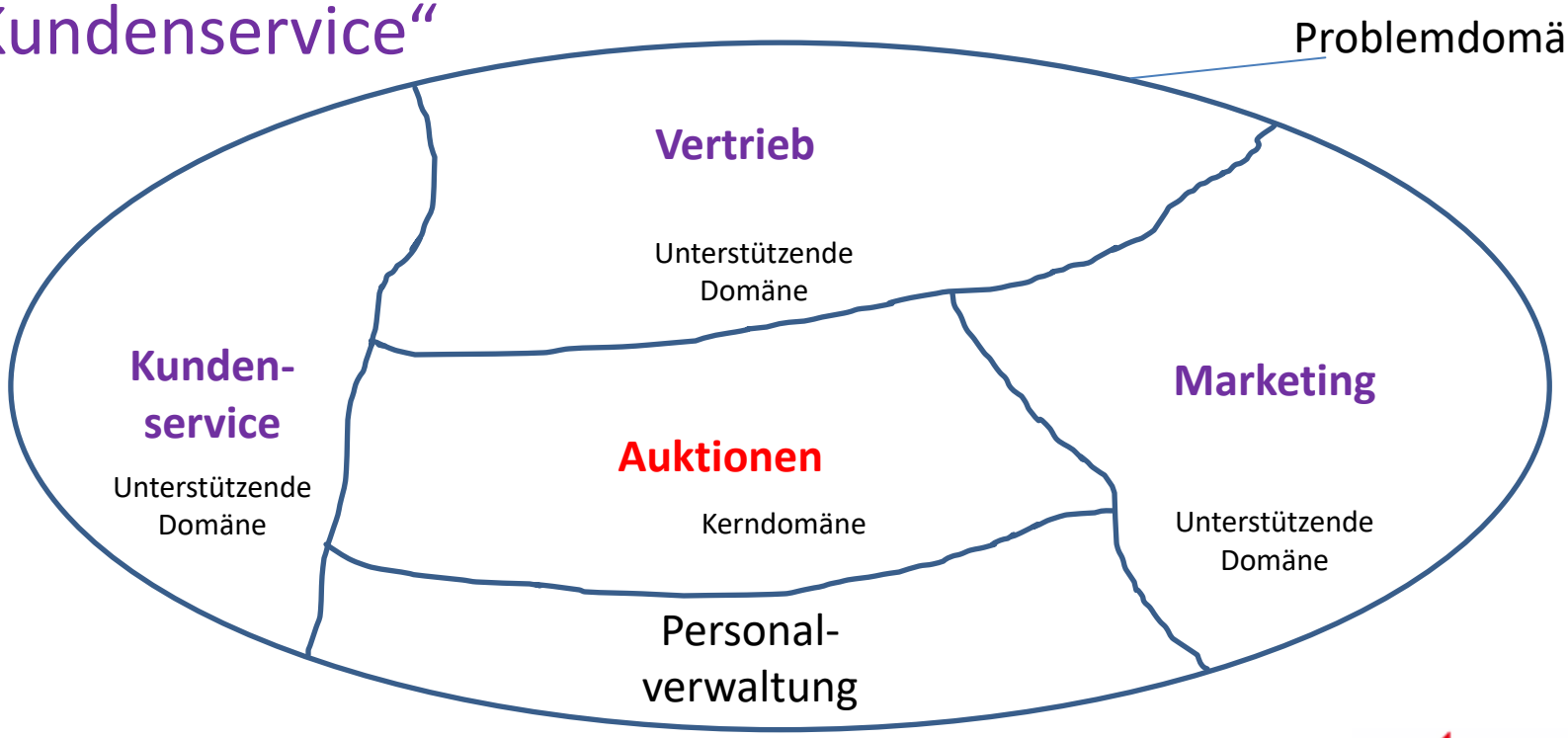
Unterstützende Domäne

- 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

Beispiel unterstützende Domäne

Problemdomäne: Online-Auktionshaus

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



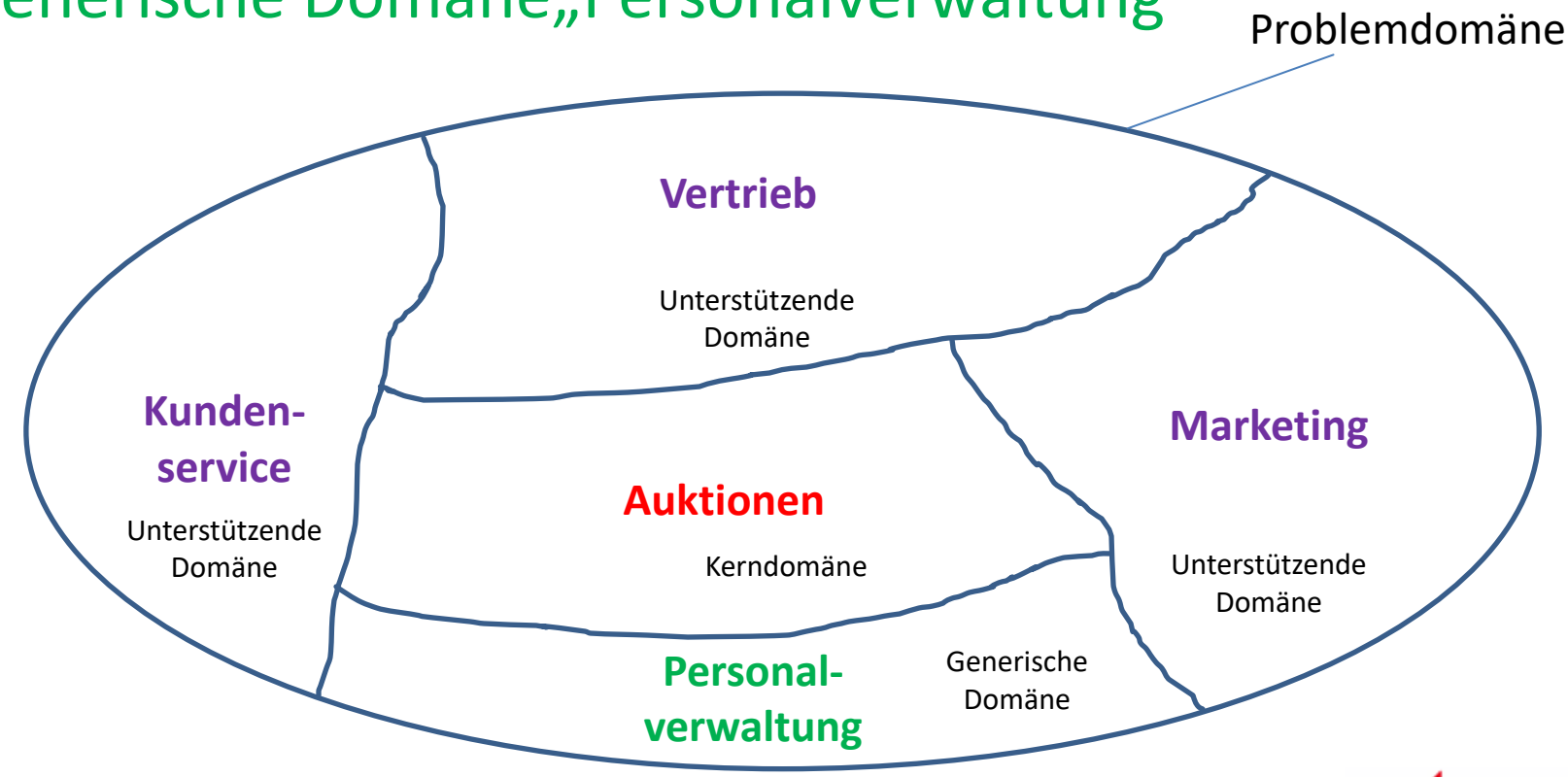
Generische Domäne

- 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

Beispiel generische Domäne

Problemdomäne: Online-Auktionshaus

Generische Domäne„Personalverwaltung“

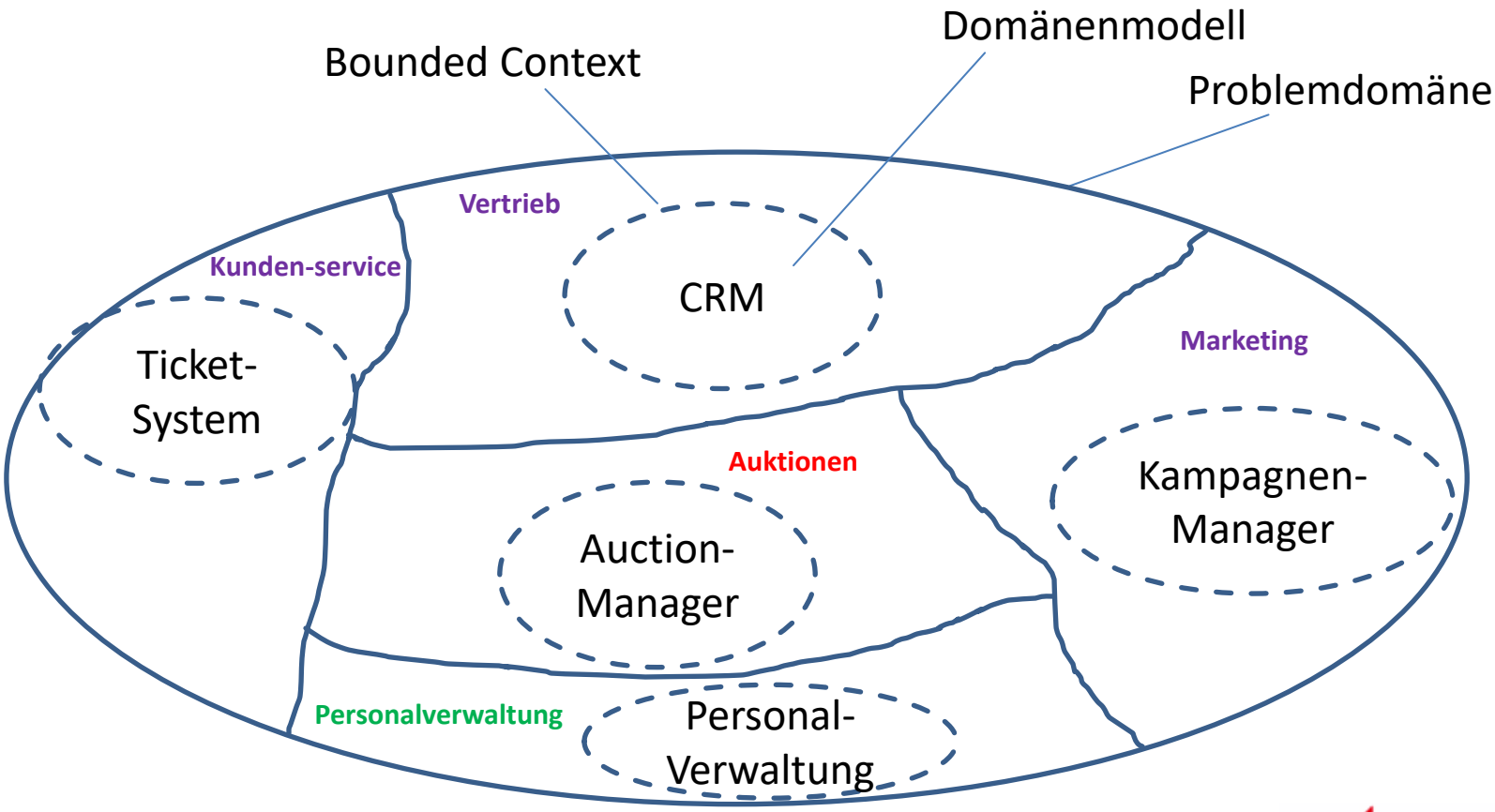


Bounded Context

- Idealzustand:
  - Idealerweise existieren je Subdomäne nur eine oder wenige Anwendungen, egal ob in Form einer eigenen oder einer fremden Anwendung
  - Jede dieser Anwendungen hat ein internes Domänenmodell, das sauber von den Modellen der anderen Anwendungen getrennt ist
  - Die Anwendungen kommunizieren untereinander über sauber definierte Schnittstellen

DHBW

Bounded Context



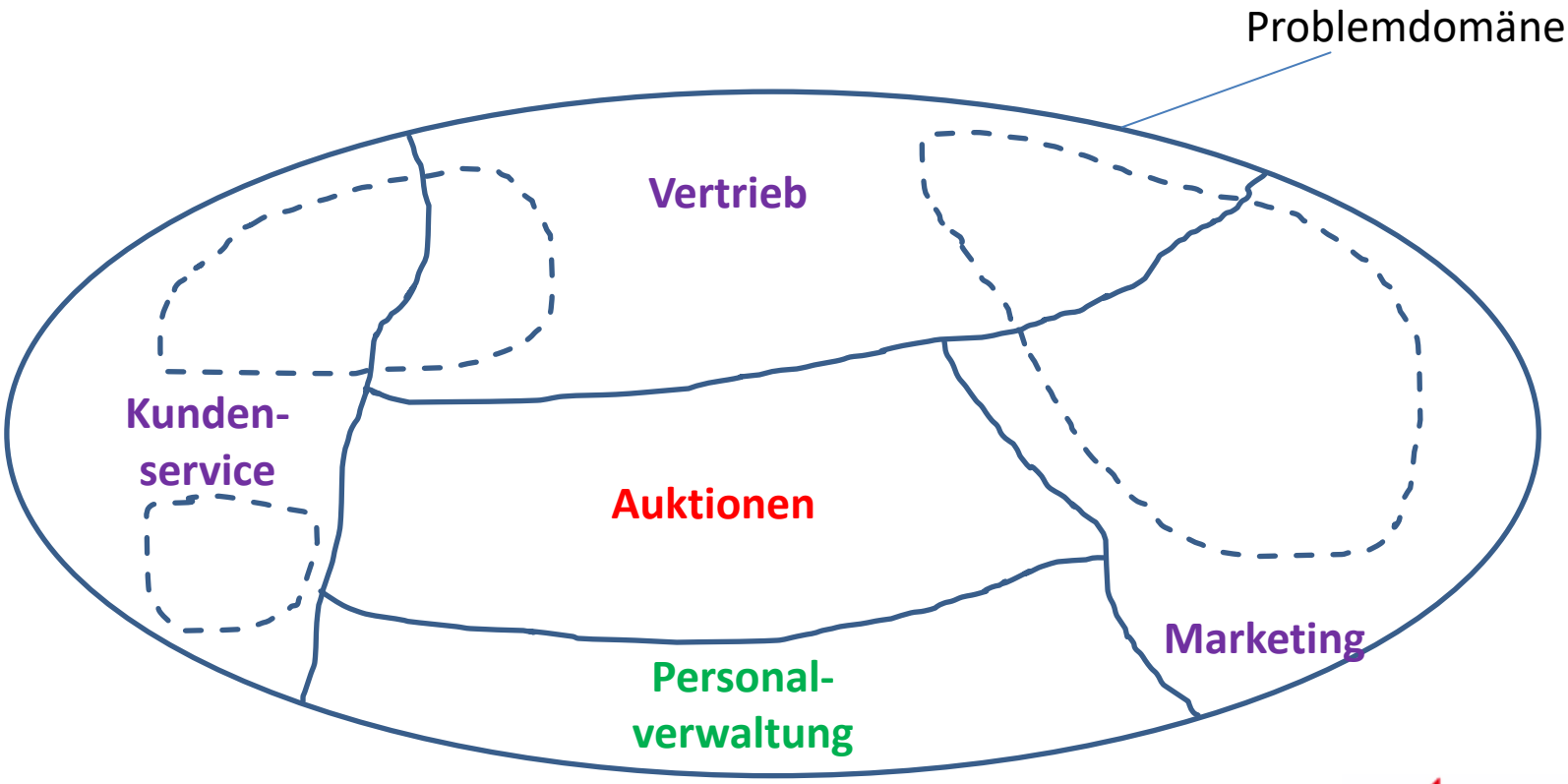
Bounded Context

- In der Praxis trifft man aber normalerweise auf ein sog. „Brownfield“:
  - Komplexe, gewachsene Systemlandschaft
  - Mehrere eigene und fremde Anwendungen mit zu großen/ungenauen Domänenmodellen, die nicht klar nach Subdomänen getrennt sind
  - Komplexe, teils unklare Anhängigkeiten zwischen Anwendungen in Form von Schnittstellen

Bounded Context

- Um diese komplexen Abhängigkeiten und verwachsenen Domänenmodelle greifbar zu machen, führt DDD den sog. **„Bounded Context“ oder „Kontextgrenze“** ein
- Ein **Bounded Context** visualisiert, innerhalb welcher Subdomänen ein Domänenmodell (=eine Anwendung) Gültigkeit besitzt

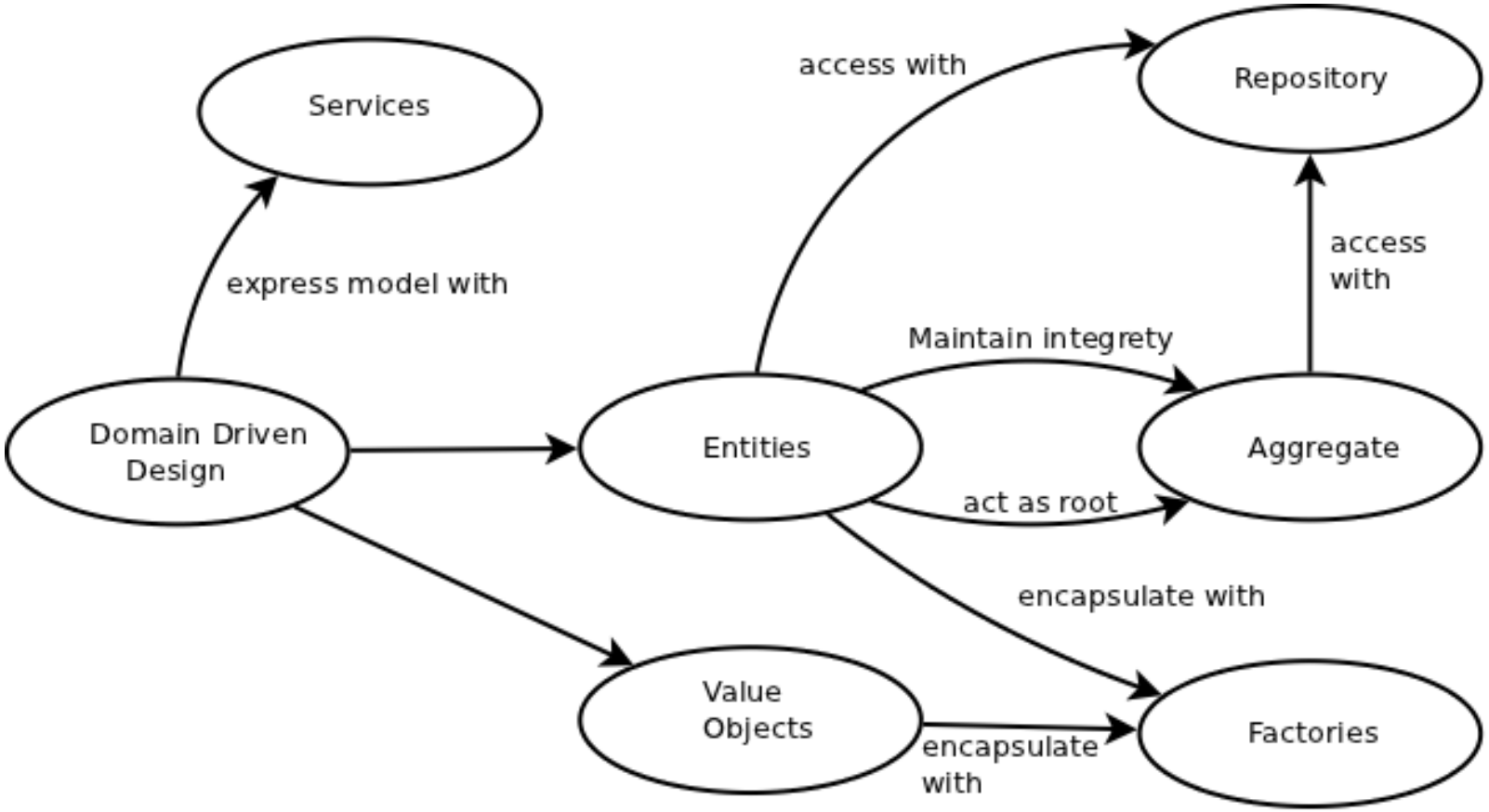
Beispiel für „Brownfield“



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



Von Moritz Graf - Prof in der Vorlesung, Bild-frei, https://de.wikipedia.org/w/index.php?curid=6277659

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“
- Modules:
  - Dienen zur **Strukturierung**, **Unterteilung** und **Kapselung verwandter Domänenobjekte** innerhalb des Domänenmodells und
  - Fördern dadurch **geringe Kopplung** und **hohe Kohäsion**

Value Objects

- Beispiel: Gewicht

DHBW

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

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 sollte „final“ sein (Vererbung unterdrücken)
2. Alle Felder sollten „blank final“ sein
3. Ist nach Konstruktion in gültigem Zustand, andernfalls muss Konstruktion fehlschlagen
4. Es existieren keine „Setter“ oder sonstige Methoden, die den Status des VO ändern
5. 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

DHBW

Persistierung von Value Objects

Es gibt mehrere Möglichkeiten, VO zu speichern:

- Eingebettet in die Tabelle des „Elternobjekts“
- Serialisierung
- In einer eigenen Tabelle (als DB-Entity)

NICHT als DDD-Entity!

Die Wahl kann individuell nach VO getroffen werden.

VO in Elterntabelle speichern am Beispiel einer JPA-Entity

```java
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @Embedded
    private Address address;
}
```

```java
@Embeddable
public class Address {
    private final String strasse;
    private final String plz;
    private final String ort;
    //...
}
```

ID	Name	Straße	PLZ	Ort	…
42	Hugo Müller	Teststr. 1	76131	Karlsruhe	…

DHBW

VO in Elterntabelle speichern

Vorteile:

- Einfache Umsetzung, mit den meisten ORM-Tools problemlos möglich
- Erlaubt Queries über Elemente des VO

Nachteile:

- Ggf. Denormalisierung der DB
- Funktioniert nur bei 1:1-Beziehungen

VO serialisieren

- Das VO wird konvertiert und in einer Spalte der Elterntabelle gespeichert
- Möglichkeiten sind von O/R-Mapper bzw. Persistence Store abhängig
- Beispiele:
  - Converter in JPA
  - CustomUserType in Hibernate

VO serialisieren
am Beispiel einer JPA-Entity

```
@Entity
public class Customer{

    private Long id;

    @Convert(
        converter = NameConverter.class
    )
    private Name name;
}
```

```
public final class Name {

    private final String firstName;
    private final String lastName;

    public Name(String firstName, String lastName) {}
}
```


DHBW

VO serialisieren am Beispiel einer JPA-Entity

```
@Converter
public class NameConverter implements AttributeConverter<Name, String>{

    private static final String DELIMITER = "|";

    @Override
    public String convertToDatabaseColumn(final Name domainName) {
        return domainName.getFirstName() + DELIMITER + domainName.getLastName();
    }

    @Override
    public Name convertToEntityAttribute(final String dbName) {
        String[] nameComponents = dbName.split(DELIMITER);
        return new Name(nameComponents[0], nameComponents[1]);
    }
}
```

ID	Name	…
42	Max|Muster	…

VO serialisieren

Vorteile:

- Komplexe Objekte können gespeichert werden
- 1:n-Beziehungen möglich (Set, List)

Nachteile:

- DB wird ggf. unlesbar oder schwerer verständlich
- Queries über VO schwierig oder nicht möglich
- Aufwändiger

VO als DB-Entity in eigener Tabelle speichern

- Das VO wird aus Sicht der Persistenz-Schicht wie eine eigenständige Entity behandelt
- ->das VO erhält also eine ID
- Die ID sollte innerhalb der Domäne „versteckt“ werden und darf auch bei equals() / hashCode() nicht beachtet werden
- Die ID existiert rein zum Zwecke der Speicherung des VO in der DB

VO als DB-Entity in eigener Tabelle speichern

```
@Entity
public final class Name {

    @Id
    @GeneratedValue
    private Long id;

    private final String firstName;
    private String lastName;
    //…
}
```

Alternative:

JPA 2 erlaubt „derived keys“ für OneToOne und ManyToOne

https://en.wikibooks.org/wiki/Java_Persistence/Identity_and_Sequencing#JPA_2.0

DHBW

VO als DB-Entity in eigener Tabelle speichern

Vorteile:

- Einfach zu implementieren
- Erlaubt Normalisierung
- Queries über Attribute des VO möglich
- Ermöglicht 1:n Beziehungen (Set, List)

Nachteile:

- Verschleiert Natur eines VO durch ID
- Gefahr, dass mehrere Entities dasselbe VO benutzen

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

Allgemeine Regeln für Entities

```
public class Product {
        //...
        private Price price;
        public Product(Price price) {
                super();
                validate(price);
                this.price = price;
        }

        private void validate(Price newPrice) {
                if(newPrice.notCompatibleTo(this.price)) {
                        throw new InconvertiblePriceException();
                }
        }

        public void changePrice(Price newPrice) {
                validate(newPrice);
                this.price = newPrice;
        }
}
```


DHBW

Allgemeine Regeln für Entities

- Entities können equals() (und dann auch hashCode()) überschreiben – wenn angebracht
  - Zwei Personen mit gleicher ID aber unterschiedlichem Status?
  - Gleichheit bei Entities ist abhängig vom Anwendungskontext
- Alternativ: hasSameIdentityAs(otherEntity)

Allgemeine Regeln für Entities

```
public class Product {

        //...

        public boolean hasSameIdentityAs(Product that) {
                return (null != that ) &&
                        (this.getClass() == that.getClass()) &&
                        (this.id == that.id)
        }
}
```

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

Kurs-Name

„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 Identfier (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)

DHBW

UUID als String

```
public class Person {

        @Id
        private String uuid;

        public Person(UUID uuid) {
                super();
                this.uuid = uuid.toString();
        }
}
```


DHBW

UUID als ValueObject

```java
@Embeddable
public final class PersonId {

        private final String uuid;

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

```java
public class Person {

        @EmbeddedId
        private PersonId personId;

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

```java
public class PersonRepository {

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


DHBW

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)**

DHBW

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, …)

DHBW

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

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;
        }
}
```


DHBW

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 Domänemodells 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

DHBW

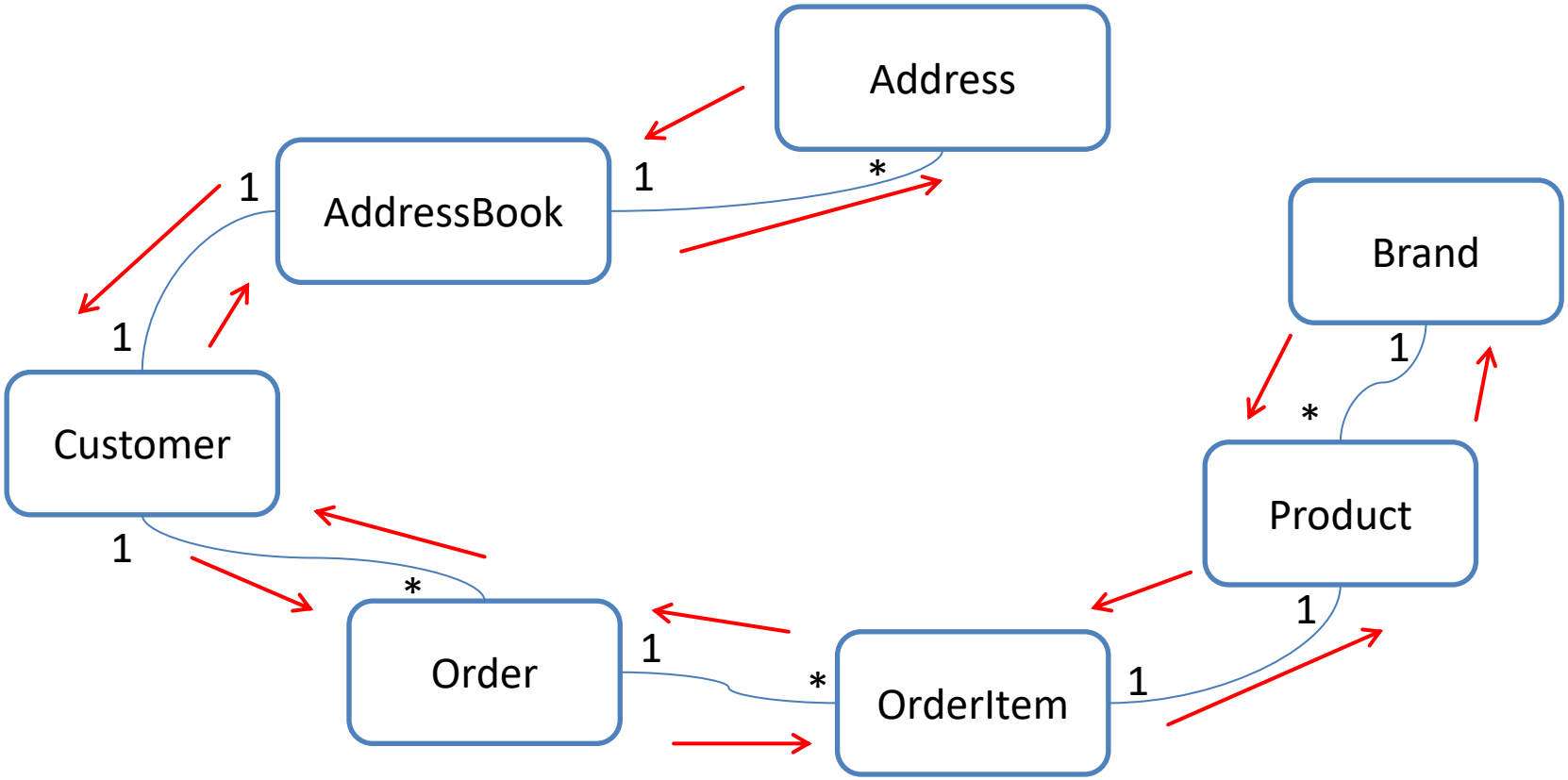
Einsatz von Domain Services: Definition eines Vertrags

- 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

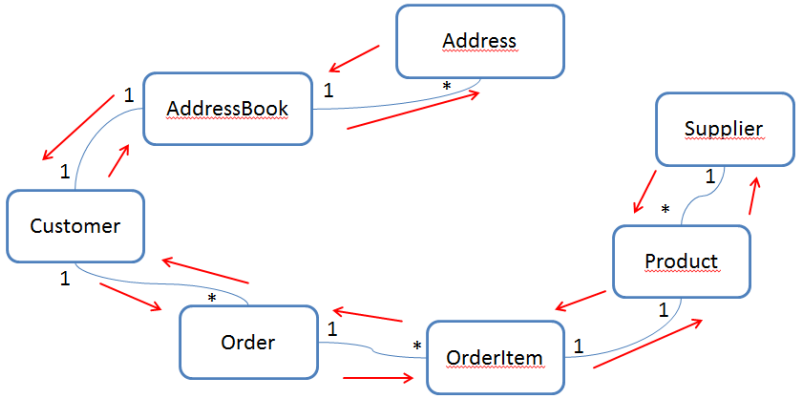
- Erfüllt **Funktion**, die **nicht** in einer **Entity** oder **VO** modelliert werden kann
- Operiert **ausschließlich** mit anderen **Elementen des Domänenmodells** für Eingabe und Ausgabe
- Domain Service und seine öffentlichen Methoden **verkörpern Konzepte der UL**
- Ist **statuslos**

Aggregate
Beispiel eines Domänenmodells



Problem: zu viele Abhängigkeiten

- Beim direkten „Nachmodellieren“ der „realen Welt“ als Entities und VO entstehen große Objektgraphen mit bidirektionalen Abhängigkeiten
- Dies führt zu Problemen:
  - Performance-Einbußen für das Laden und Speichern
  - Wahrscheinlichkeit der Verletzung von Invarianten steigt
  - Wahrscheinlichkeit für Kollisionen beim gleichzeitigen Bearbeiten des Objekt-Graphen durch mehrere Bearbeiter steigt



Die technische Lösung

O/R-Mapper wie JPA bieten Lösungen für einige dieser Probleme:

- Verschiedene Locking-Modi (optimistic, pessimistic, …)
- Lazy/Eager Loading

Aber:

1. Die Lösung dieser Probleme ist dann kein Teil der Domäne
2. Verhalten von O/R-Mappern oft undurchsichtig und schwer kontrollierbar (Beispiel: „select n + 1“)
3. O/R-Mapper bieten keine Lösung für das Einhalten von Invarianten
4. Was tun, wenn beispielsweise mit einer OO-DB oder NoSQL gearbeitet wird?

Die Domänen-Lösung: Aggregate

- Aggregates gruppieren Entities und VO zu gemeinsam verwalteten Einheiten
- Jedes Aggregate definiert genau eine Root Entity (auch Aggregate Root genannt), über welche der Zugriff auf die Teile des Aggregate erfolgen darf
- Aggregates reduzieren Komplexität beim Verwalten von Objekten, erleichtern die Handhabung von Transaktionen und reduzieren die Möglichkeiten, Invarianten zu verletzen

„Aggregates are the most powerful of all tactical patterns, but they are one of the most difficult to get right.“ [Millett, 2015]

Die Domänen-Lösung: Aggregate

- Jede Entity gehört zu einem Aggregat (auch wenn das Aggregat nur aus einer Entity besteht)
- Aggregates werden hinsichtlich Persistenz als Einheit verwaltet (create, read, update, delete)
- Ein Repository arbeiten also immer mit Aggregates und kann dieses komplett innerhalb einer Transaktion lesen/schreiben
- ->Aggregates bilden natürliche Transaktionsgrenzen und machen diese „sichtbar“

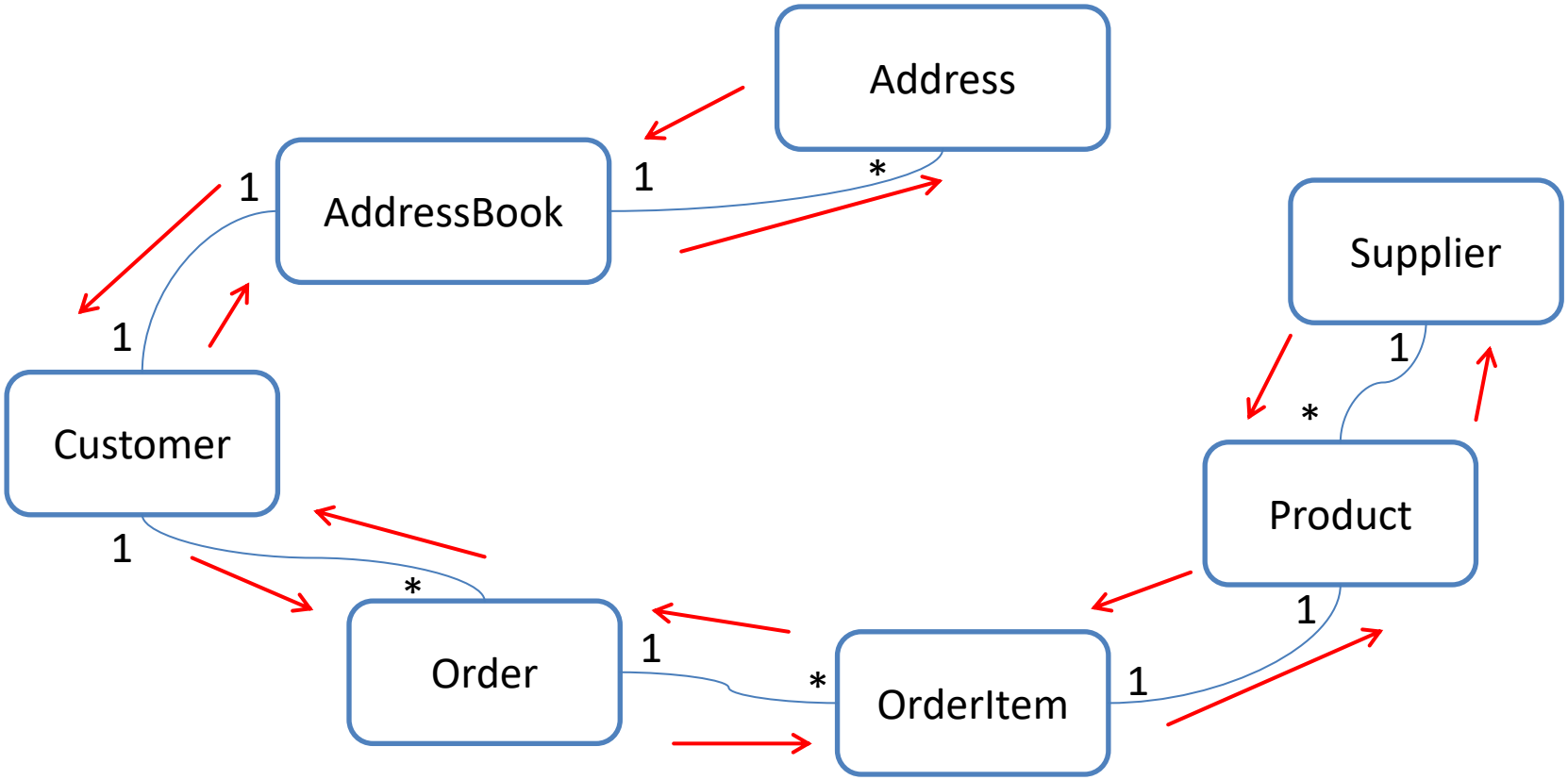
DHBW

Gruppieren von Entities und VO

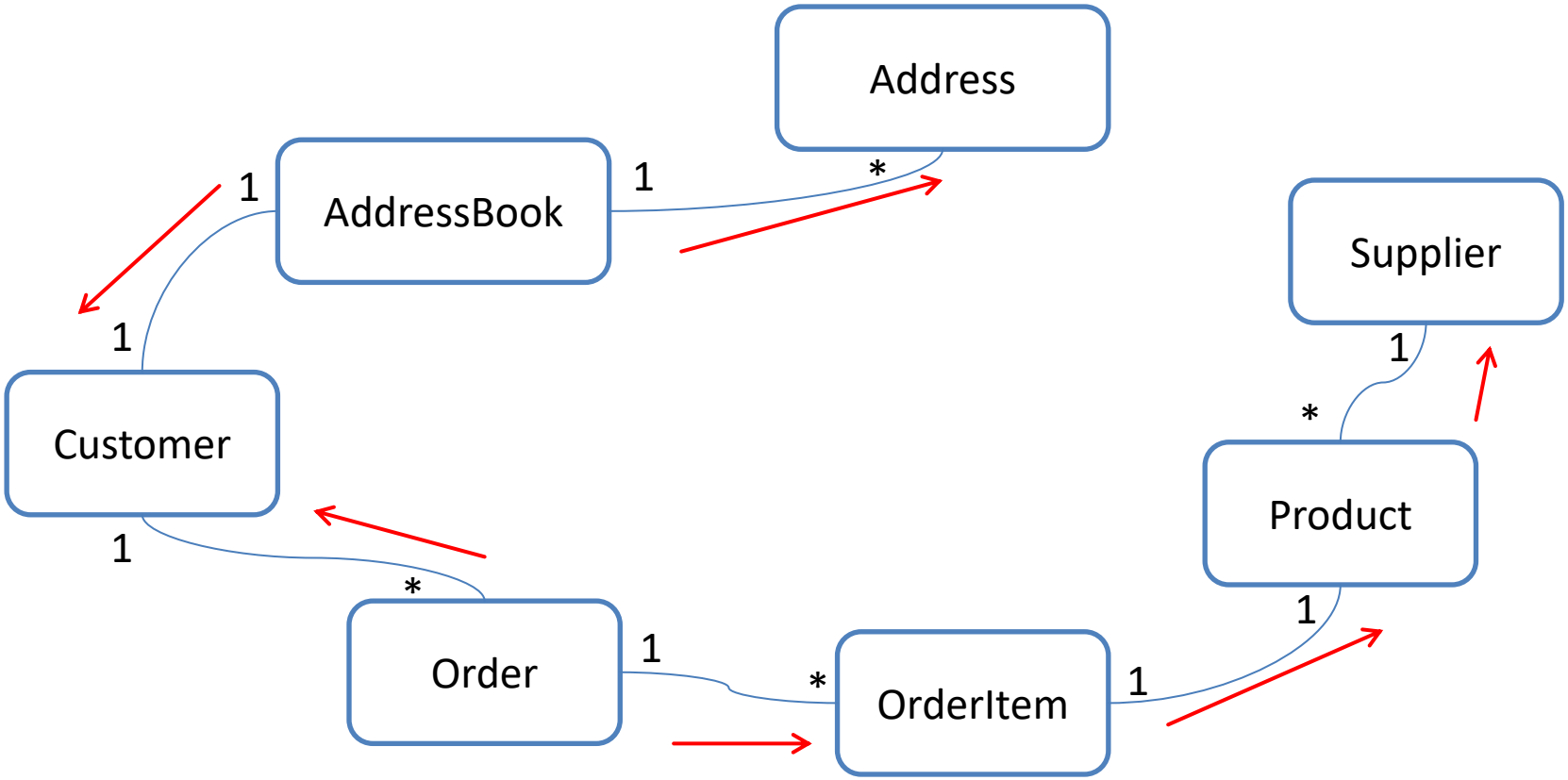
Zwei grundsätzliche Möglichkeiten:

- Einschränkung der Assoziationsrichtung
- Ersetzen von Objektreferenzen durch IDs

Beispiel: Einschränkung der Assoziationsrichtung

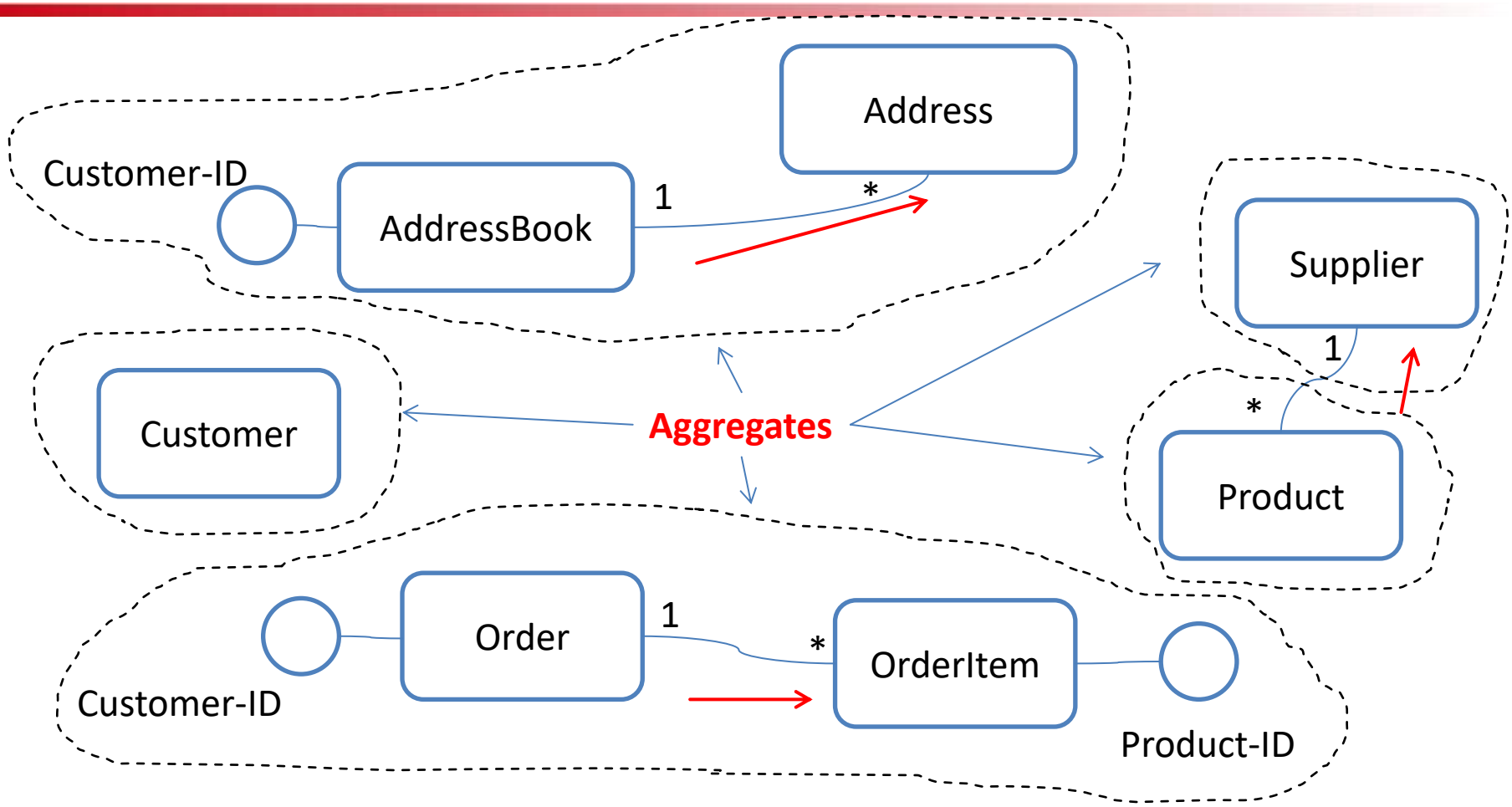


Beispiel: Einschränkung der Assoziationsrichtung



DHBW

Beispiel: Ersetzen von Objektreferenzen durch IDs



Ersetzen von Objektreferenzen durch IDs

- Entspricht der Abbildung in einer relationalen Datenbank (Foreign Key)
- Durch das Entfernen der Referenz geht also erst einmal nichts „verloren“

ID	Name
1	Max Muster

Tabelle „Customer“

ID	CustomerId
2	1

Tabelle „Order“

```
public class Customer {

    private Long id;
    //...
}
```

```
public class Order {

    private Long id;

    private Long customerId;
    //...
}
```


DHBW

Ersetzen von Objektreferenzen durch IDs: Nachteile

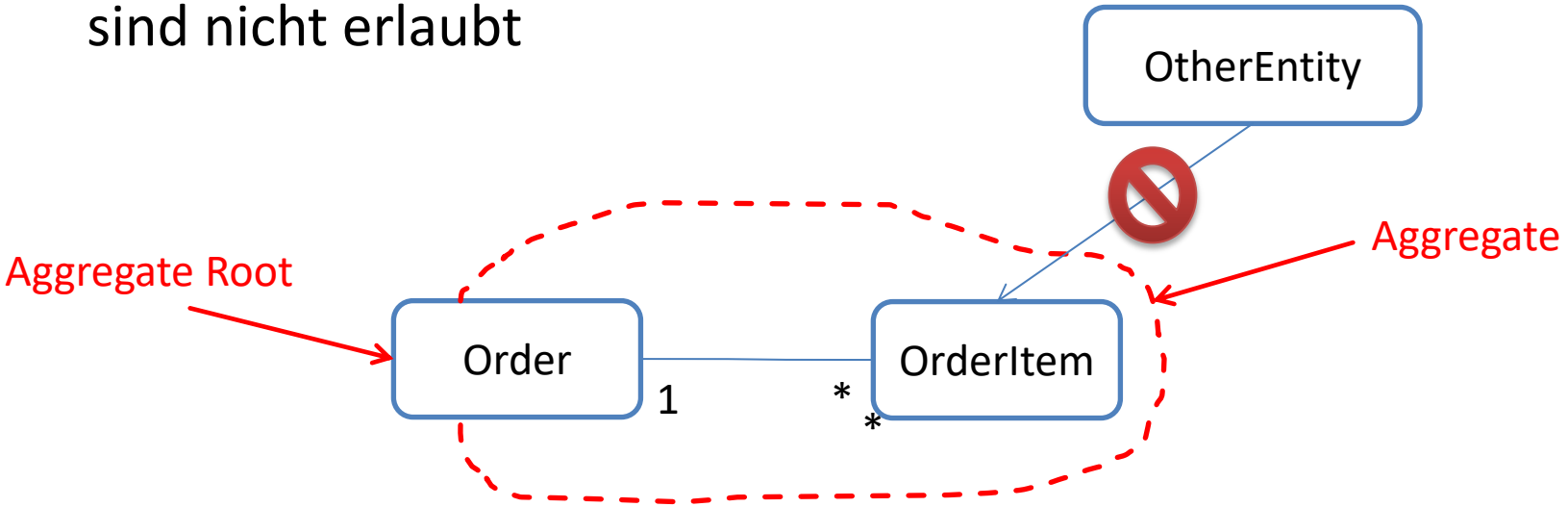
- Wenn das DB-Schema nur über O/R-Mapper erzeugt und aktualisiert wird, sind keine Fremdschlüssel mehr abbildbar (Beispiel „create-drop“ mit JPA)
  - Das ist in der Praxis aber normalerweise nicht relevant, da die DB über eigene Migrationen aufgesetzt und aktualisiert wird
- Es sind weitere Abfragen nötig, um beispielsweise auf das AddressBook eines Customer zuzugreifen
- -> mehr Verantwortung

Ersetzen von Objektreferenzen durch IDs: Vorteile

- Schlankere Objekt-Graphen
- Zuständigkeiten sind klarer getrennt
- Aggregate bildet hinsichtlich Transaktionen implizit eine „Unit of Work“ (UoW)
- ->mehr Kontrolle

Aggregate Root: der Türsteher

- In jedem Aggregate übernimmt eine Entity die Rolle des Aggregate Root (AR)
- Alle Zugriffe auf die „inneren“ Elemente des Aggregates müssen über AR erfolgen
- Referenzen von Außen auf innere Elemente eines Aggregates sind nicht erlaubt



Sinn und Zweck des Aggregate Root: was kommt rein

- Als „Türsteher“ kontrolliert das AR alle Zugriffe auf ein Aggregat und seine Teile
- Dadurch kann es an zentraler Stelle über die Einhaltung von Invarianten wachen
  - Beispiel:
    - Aggregate Root „Order“ wacht über Einhaltung von maximalen Bestellpositionen bei OrderItems

Sinn und Zweck des Aggregate Root: was geht raus

- Externe Klienten (beispielsweise ein Domain Service) müssen normalerweise auf Teile eines Aggregates zugreifen, um sinnvolle Funktionen zu erfüllen
- Beispiel: Zugriff auf OrderItems einer Order, um Versandkosten zu berechnen
- Dabei ist Vorsicht geboten, damit von Außen nicht ungewollte Veränderungen möglich sind

Sinn und Zweck des Aggregate Root: was geht raus

- Ein AR sollte daher wenn möglich keine direkten Referenzen auf sein Inneres ausliefern, sondern **defensive Kopien**
- Möglichkeiten in Java beispielsweise:
  - CopyConstructor
  - Memento-Pattern

Key Facts Aggregates

- Entities und VO werden zu Aggregates zusammengefasst
- Aggregates werden als Einheit verwaltet (create, read, update, delete)
- Aggregates forcieren und visualisieren Transaktionsgrenzen
- Aggregates forcieren Invarianten

Aggregates in der Praxis

- Aggregate zwingen dazu, sich mit wichtig Persistenz-Fragen auseinander zu setzen
- Wenn diese Fragen ohne Nachteile oder Anhäufung von technischer Schuld mit Hilfe von Tools (O/RM) gelöst werden können: **wunderbar**
- Sie sollten diese Entscheidung jedoch **bewusst** treffen und im Zweifel daran denken, dass nicht alles vom O/R-Mapper gemacht werden muss

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 auf den Speicher (relationale DB, NoSQL, XML-Dateien usw.) wird vom Repository verborgen
- Dadurch bleibt die Domäne von technischen Details unbeeinflusst

Repositories

- Repositories arbeiten ausschließlich mit Aggregates –> je Aggregate existiert also typischerweise ein Repository
- Die konkrete Implementierung eines Repositories erfolgt – ähnlich wie bei Domain Services – normalerweise nicht im Domänenmodell

Eigenschaften eines Repositories

- Sollten keine generischen Methoden anbieten, sondern eine aussagekräftige Schnittstelle
- Kann für die Generierung von IDs zuständig sein, wenn diese von der Anwendung erzeugt werden
- Kann Prüffelder wie beispielsweise „LastUpdatedAt“ beim Speichern setzen
- Kann zusätzlich allgemeine Informationen wie beispielsweise die Gesamtzahl an Aggregates im Repository oder eine Zusammenfassung anbieten

Beispiel eines Repositories

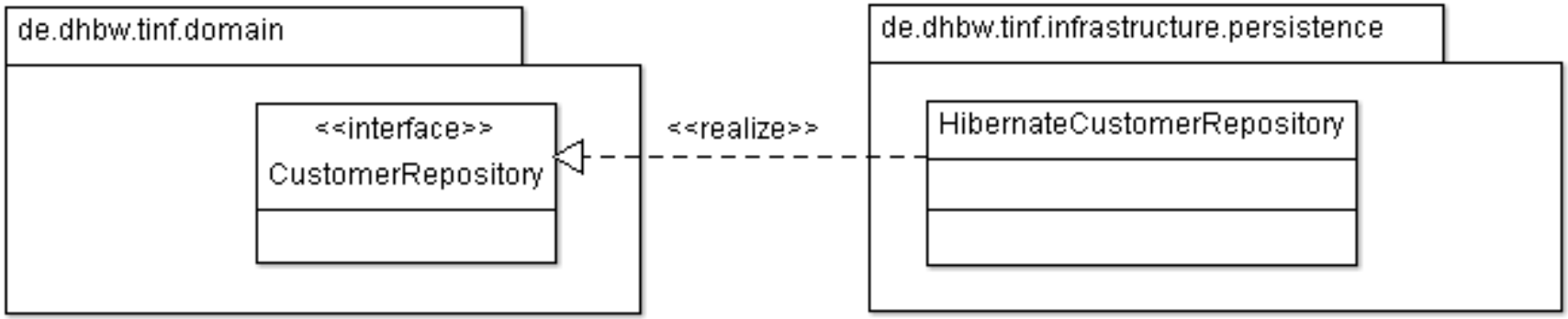
```
public interface CustomerRepository {

        void store(Customer customer);

        void findOneBy(CustomerId customerId);

        void findAllThatAreDeactivated();

        void findAll();

        Summary summary();

        CustomerId nextCustomerId();
}
```

```
public class Summary {

        public int customerCount;
        public int deactivatedCustomers;
        public int activatedCustomers;
}
```


DHBW

Implementierung eines Repositories

- Die Implementierung erfolgt normalerweise in einer technischen Schicht der Anwendung (DB, Infrastruktur, …) und **nicht** innerhalb des Domänenmodells



Factories

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

DHBW

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 andere Objekte als Konstruktor-Ersatz

b. spezielle **Erzeugungs-Muster**
   - Factory Method
   - Abstract Factory

- Im DDD meint man mit Factory normalerweise das „allgemeine Konzept“

DHBW

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);
	}

}
```


DHBW

Modules

„Good authors divide their books into chapters and sections; good programmers divide their programs into modules.”

https://medium.freecodecamp.com/javascript-modules-a-beginner-s-guide-783f7d7a5fcc#.3gv1n6pxa

Vorteile Module

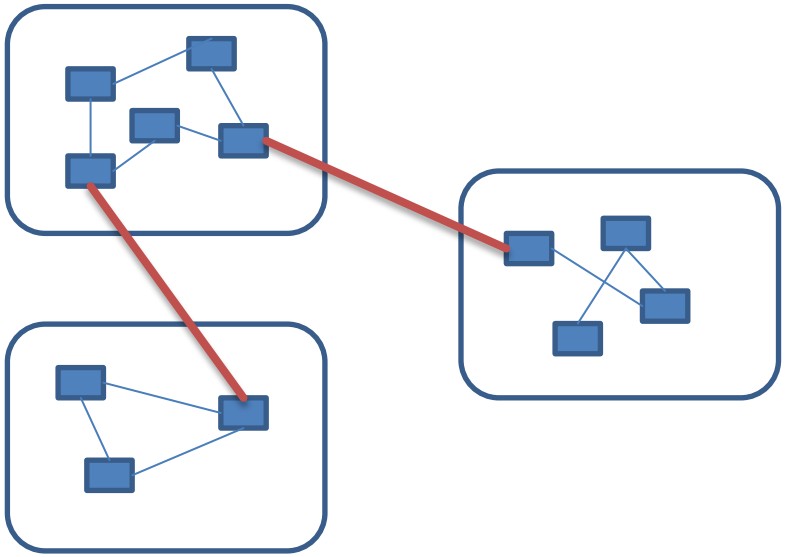
- Sichtbarkeit
- Abhängigkeiten

Modules

- Dienen zur Strukturierung, Unterteilung und Kapselung verwandter Komponenten
- Werden normalerweise über Namespaces, Packages o.ä. abgebildet
- Die Gruppierung sollte nicht nach technischen, sondern nach fachlichen Gesichtspunkten erfolgen
  - >Modulnamen sollen aus der UL stammen

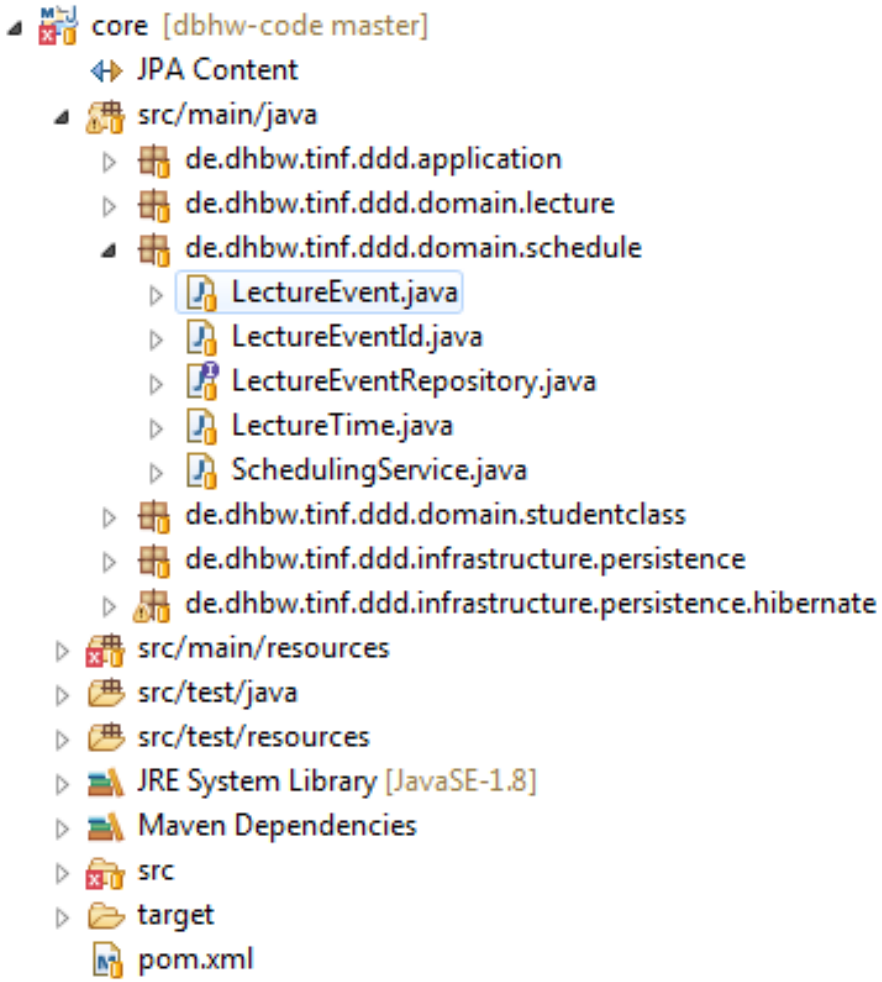
Modules

- Ziel von Modulen ist es, eng zusammengehörende Komponenten zu gruppieren (hohe Kohäsion innerhalb eines Moduls, geringe Kopplung zwischen Modulen)



DHBW

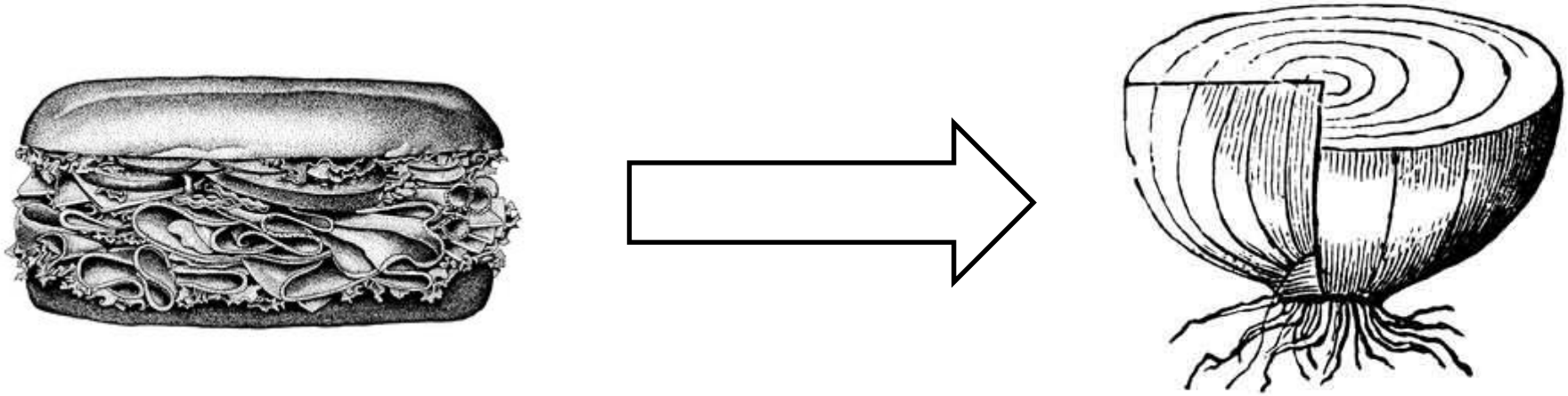
Modules in Java



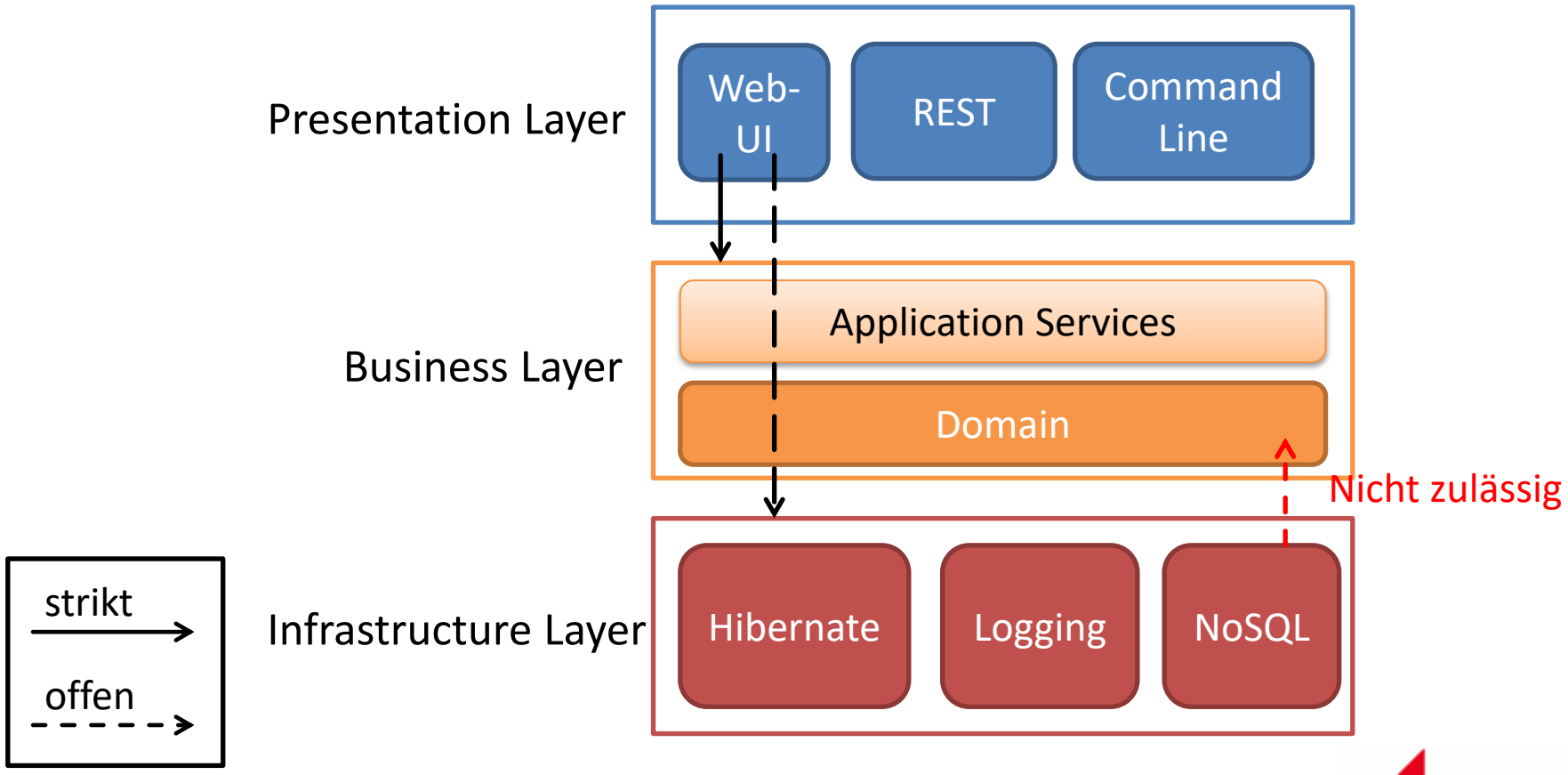
DHBW

Onion Architecture

Alternative zur klassischen Schichtenarchitektur



Klassische Schichten-Architektur



Klassische Schichten-Architektur

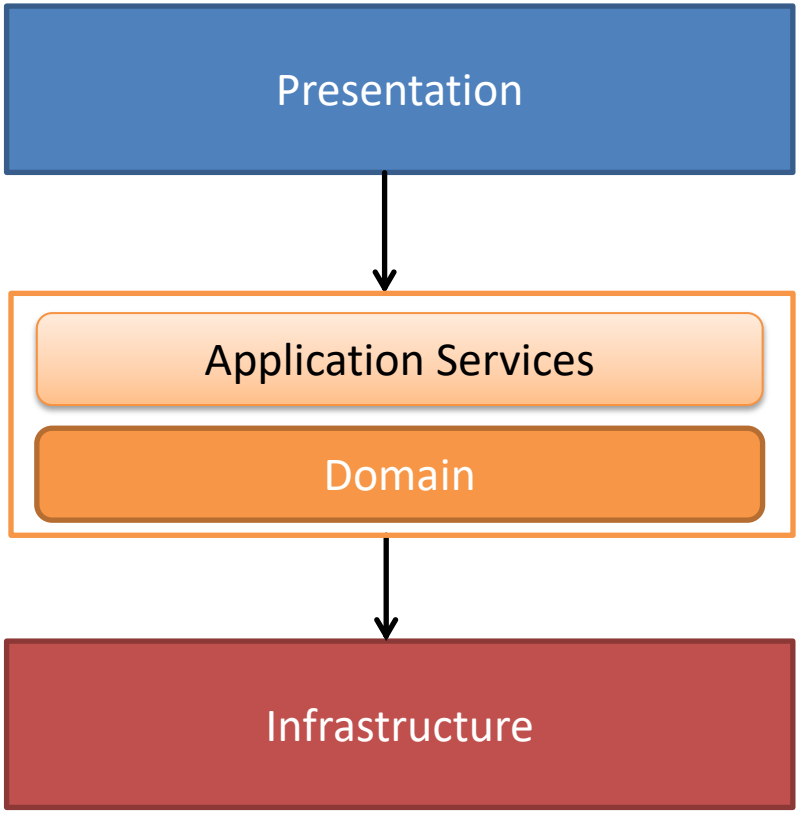
- Die Schichten-Architektur ist **das** „klassische“ Architekturmuster
- Eine Schicht darf nur mit den **unter ihr liegenden** Schichten kommunizieren
- Bei **strikter** Schichten-A. darf nur die **nächstniedrigere** Schicht aufgerufen werden
- Bei einer **offenen** Schichten-Architektur darf **jede beliebige niedrigere** Schicht aufgerufen werden

Gründe für Schichten-Architektur

- Beherrschung der Komplexität durch technische Trennung der Anwendung in mehrere Schichten
- Geringe Kopplung zwischen den Schichten, hohe Kohäsion innerhalb einer Schicht
- Dadurch sollen einzelne Schichten leichter und unabhängig von anderen Schichten geändert werden können

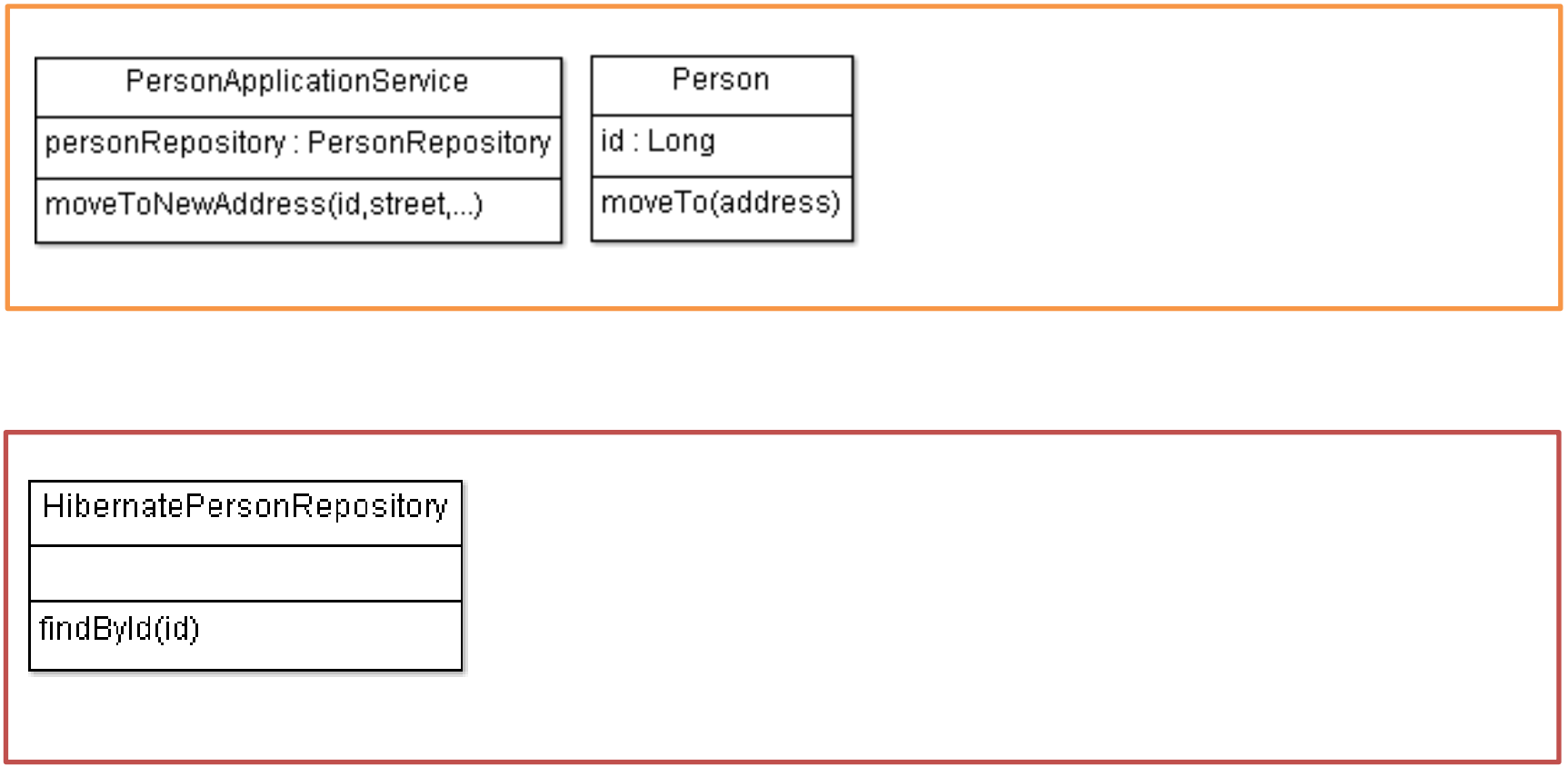
Klassische Schichten-Architektur

Frage: zu welchen Schichten gehören folgende Klassen?



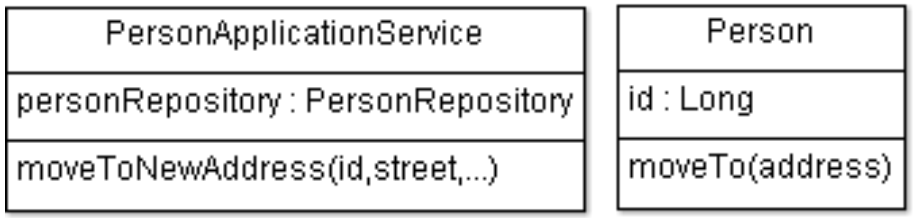
DHBW

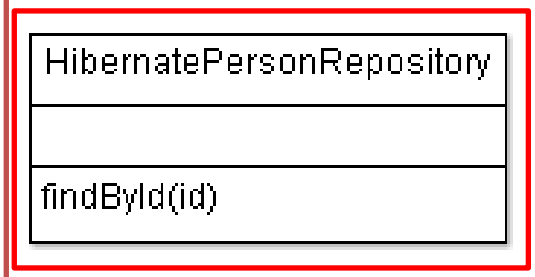
Klassische Schichten-Architektur



DHBW

Problem I





Problem:

- Das Repository ist normalerweise Teil der Domänenschicht bzw. Business-Schicht
- Der konkrete DB-Zugriff (zum Beispiel per Hibernate) gehört aber in die Infrastruktur - Schicht

DHBW

Problem I

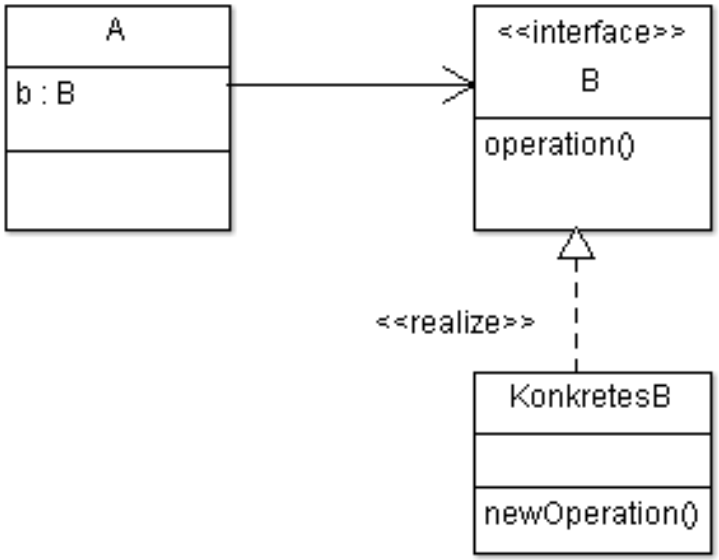
- Die Domänenschicht soll frei von technischen Details bleiben
- Die konkrete Implementierung „HibernatePersonRepository“ darf also nicht in die Domänenschicht
- Gleichzeitig gehört aber das Repository konzeptionell zu Domänenschicht, darf also eigentlich auch nicht in die Infrastruktur-Schicht
- Die Lösung: Dependency Inversion

Dependency Inversion Principle (DIP)

Definition:

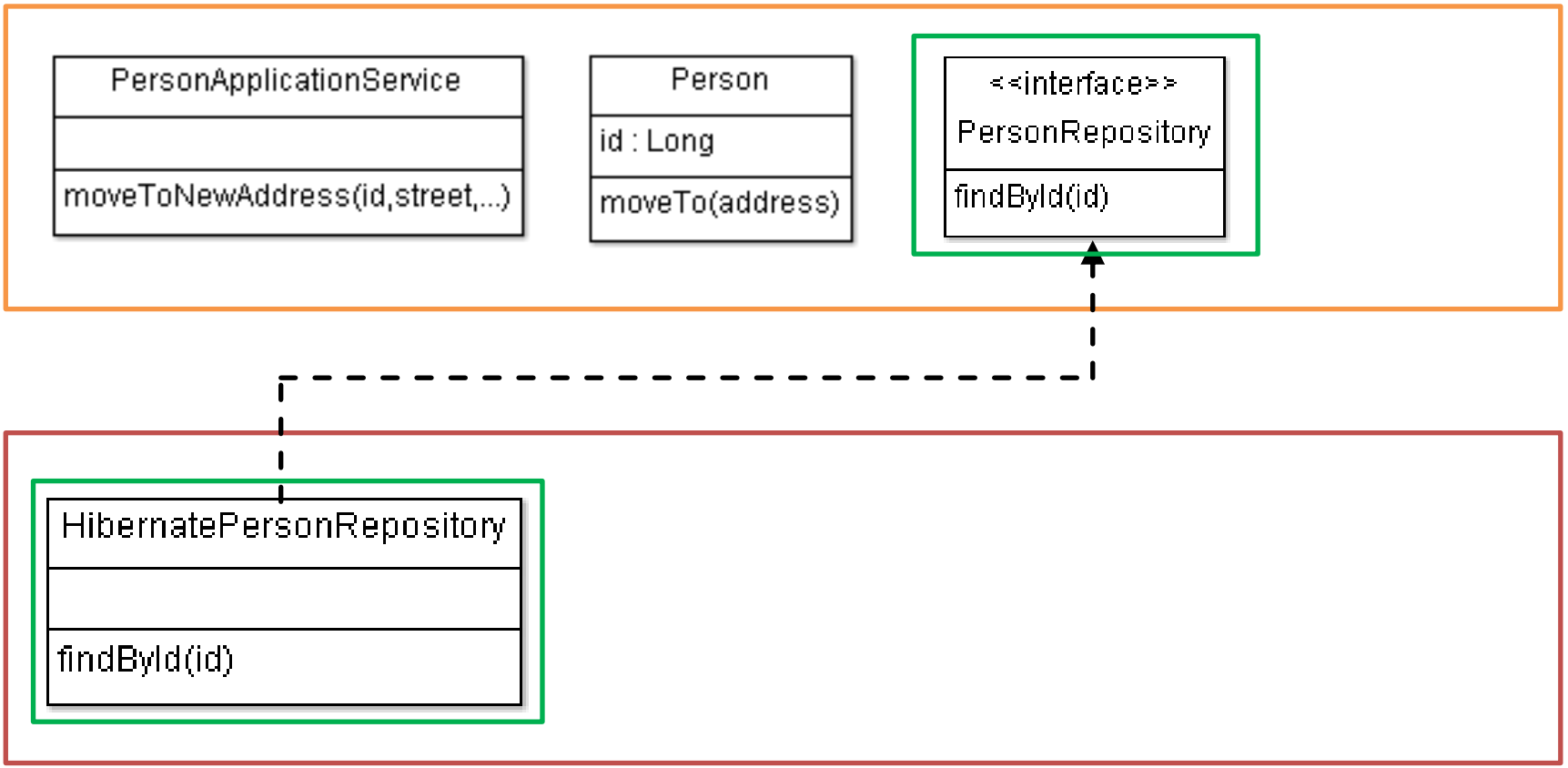
„A. *Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen.*

B. *Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.“* [Martin, 1996]



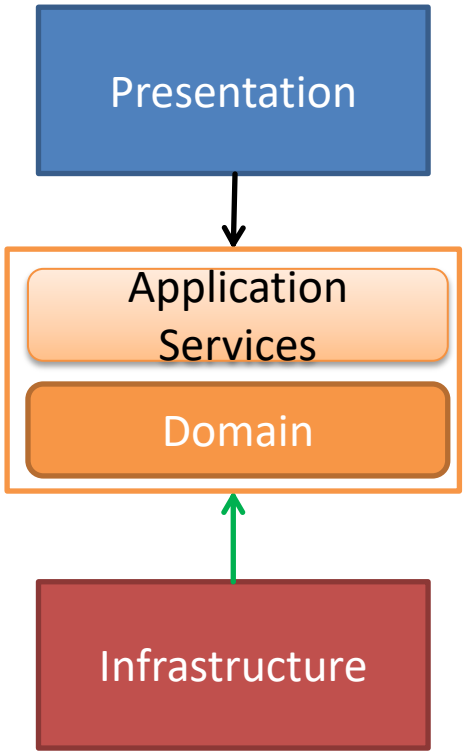
Dependency Inversion Principle (DIP)

Lösung: Definition eines Vertrages durch Dependency Inversion



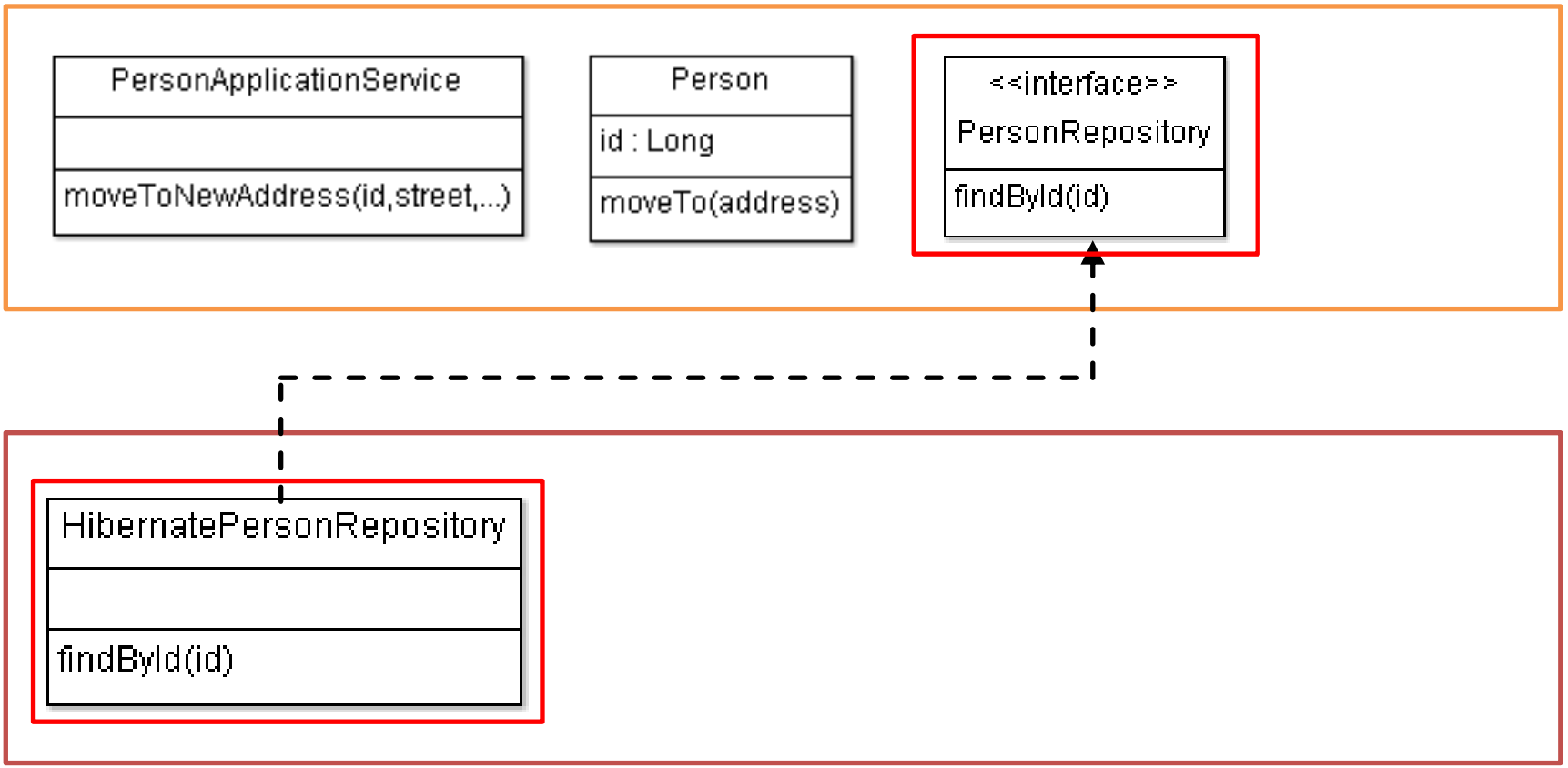
Dependency Inversion Principle

- Die Domänenschicht gibt durch ein Interface einen **Vertrag** vor, der beschreibt, welches Verhalten sie erwartet
- Die **konkrete Implementierung des Vertrages** erfolgt in der Infrastruktur-Schicht
- Die Domäne ist **damit nicht mehr abhängig von Details** (HibernatePersonRepository), sondern von Abstraktionen (PersonRepository)
- die konkrete Implementierung kann dann je nach Anwendungsfall **gewählt** werden, beispielsweise durch **Dependency Injection**



Problem II

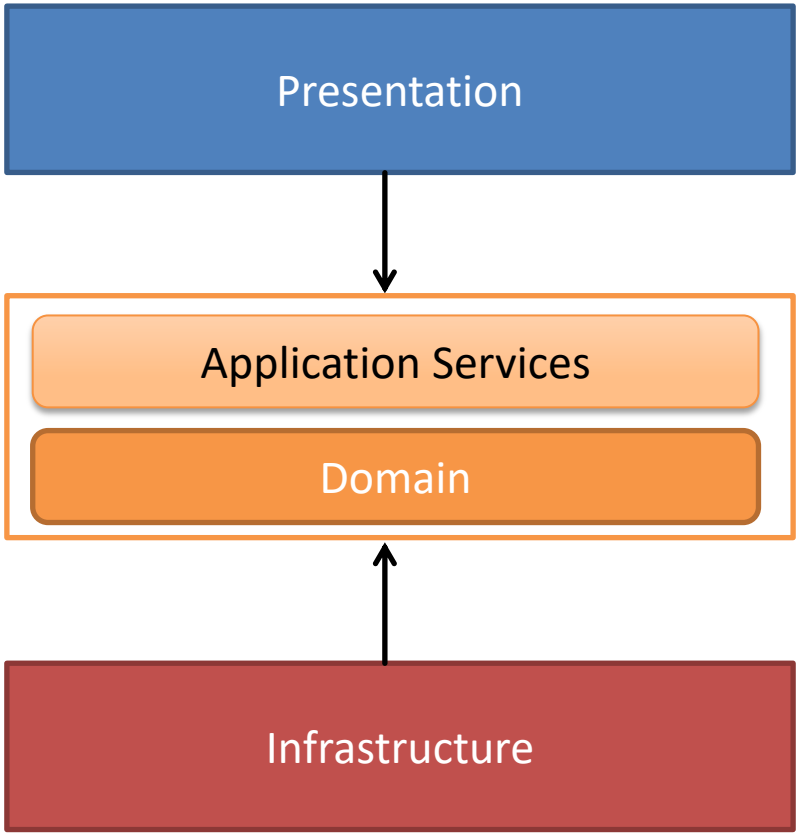
Infrastrukturschicht ist jetzt abhängig von Domänenschicht



Problem II

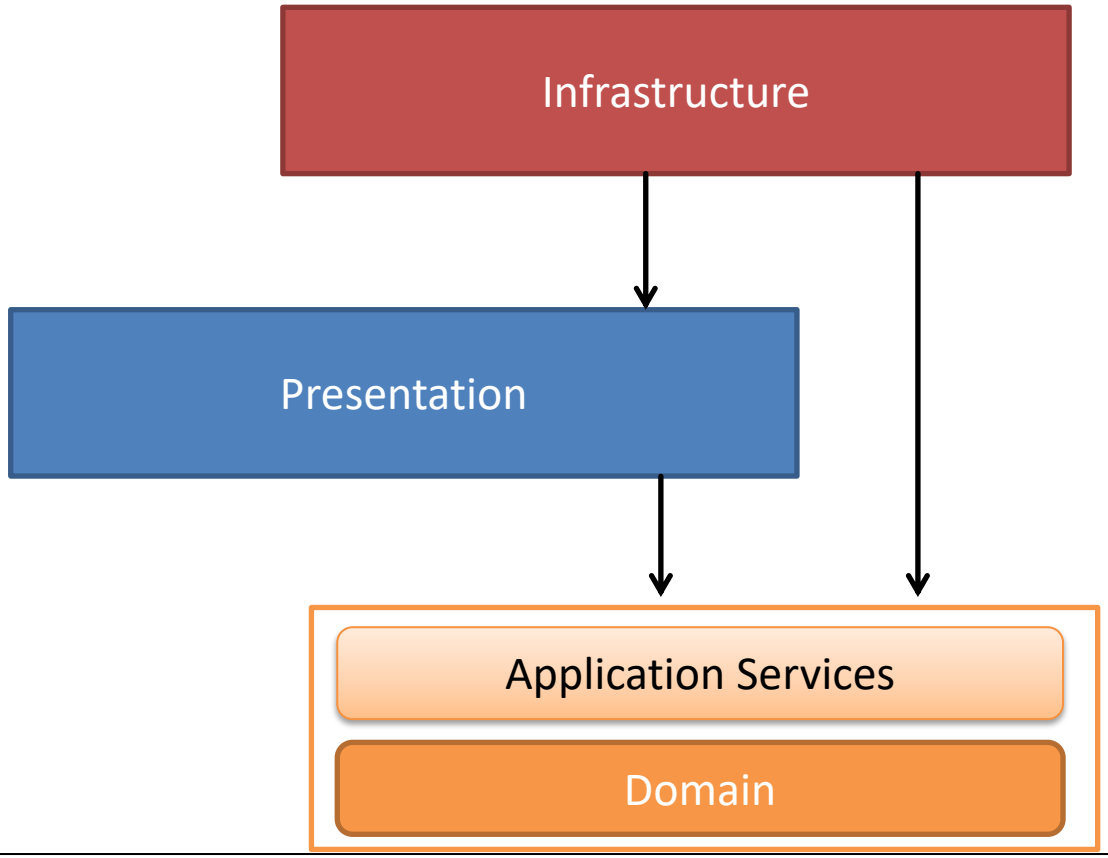
- Die Infrastrukturschicht ist abhängig vom in der Domänenschicht definierten PersonRepository
- Dies verletzt die Regeln der Schichtenarchitektur (Schicht darf nur von darunterliegenden Schichten abhängig sein)
- Die Lösung: das „Verschieben“ der Infrastruktur-Schicht

Umschichtung

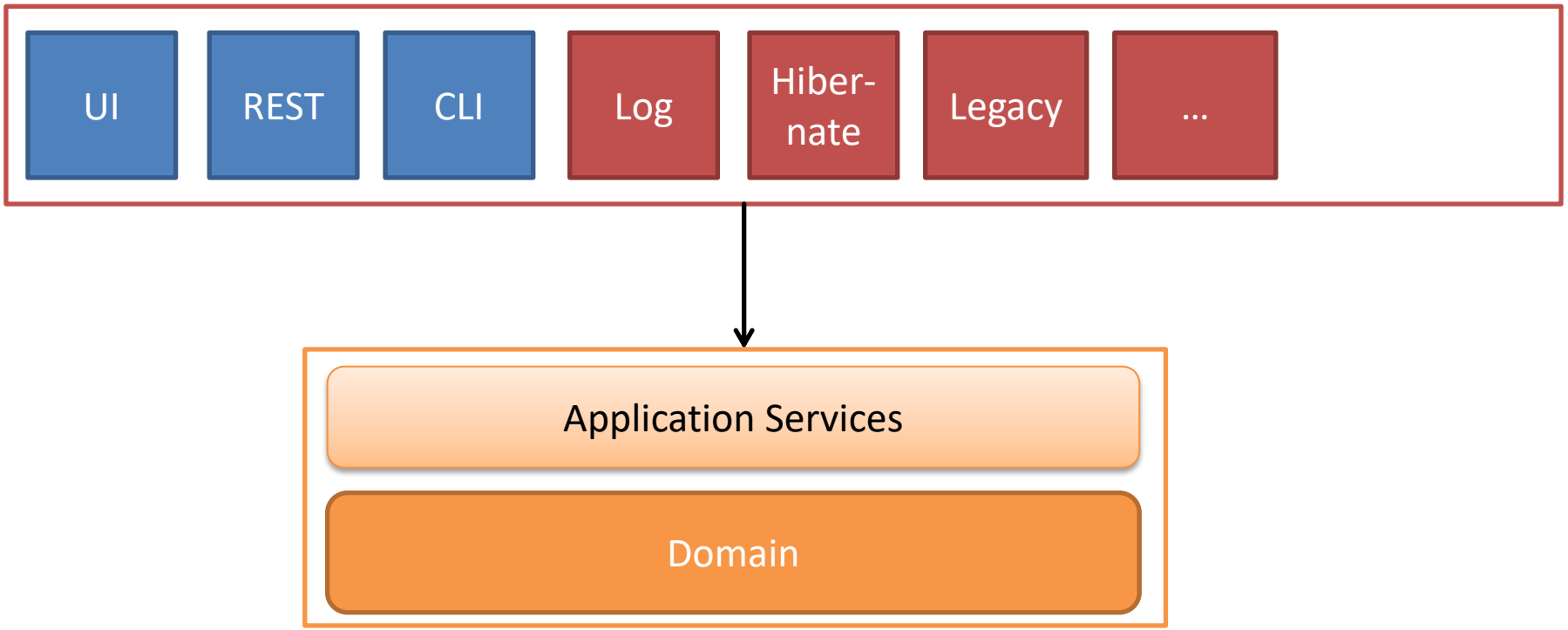


DHBW

Umschichtung



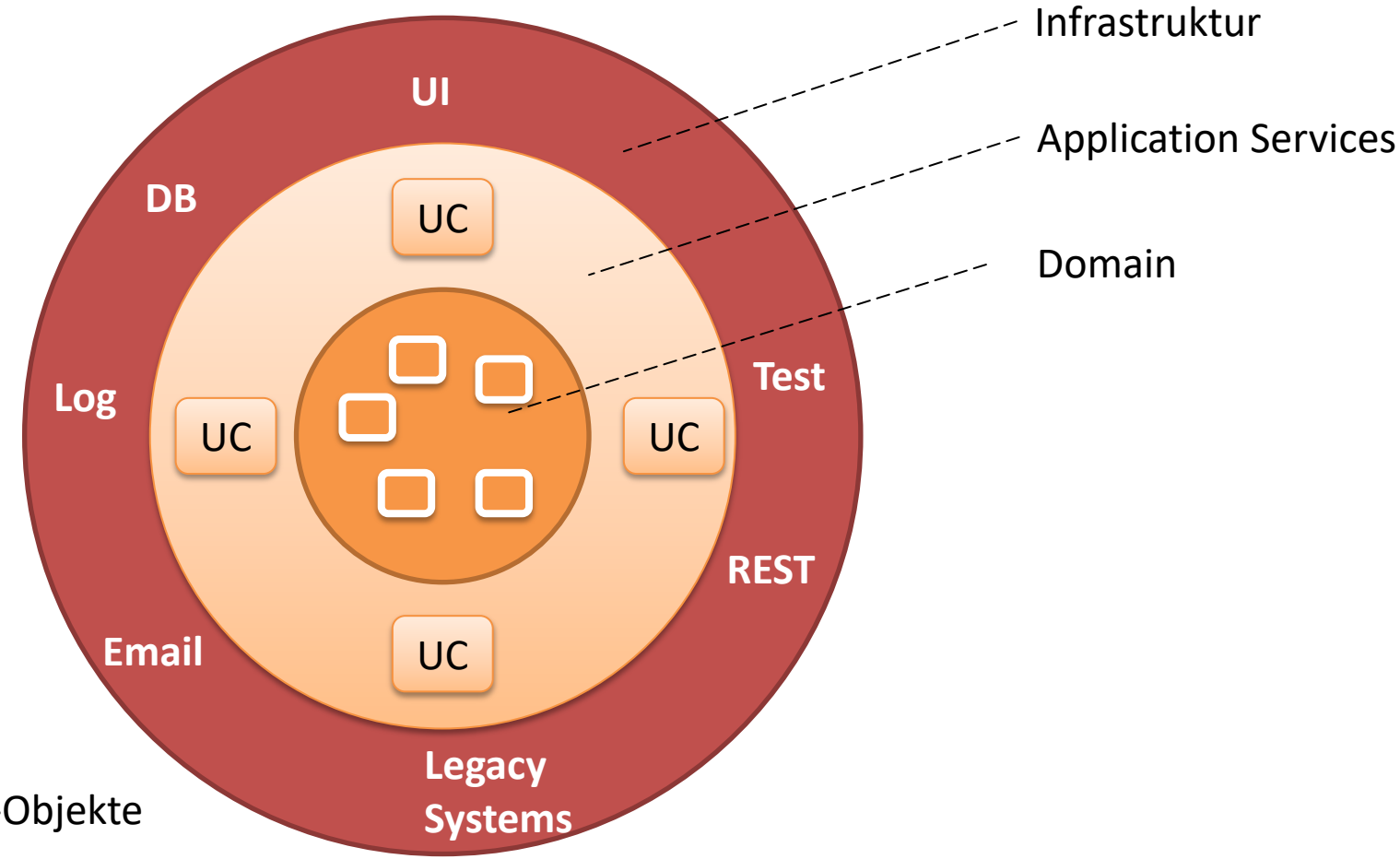
Umschichtung



Umschichtung

- Durch das Verschieben der Infrastruktur-Schicht darf diese konkrete Verträge für alle darunter liegenden Schichten implementieren, ohne die Regeln der Schichten-Architektur zu verletzen
- Die Präsentationsschicht kann dann auch als Teil der Infrastrukturschicht betrachtet werden
- Die Domänenschicht liegt ganz unten und ist von keiner anderen Schicht abhängig -> sie bildet den **Kern der Architektur**

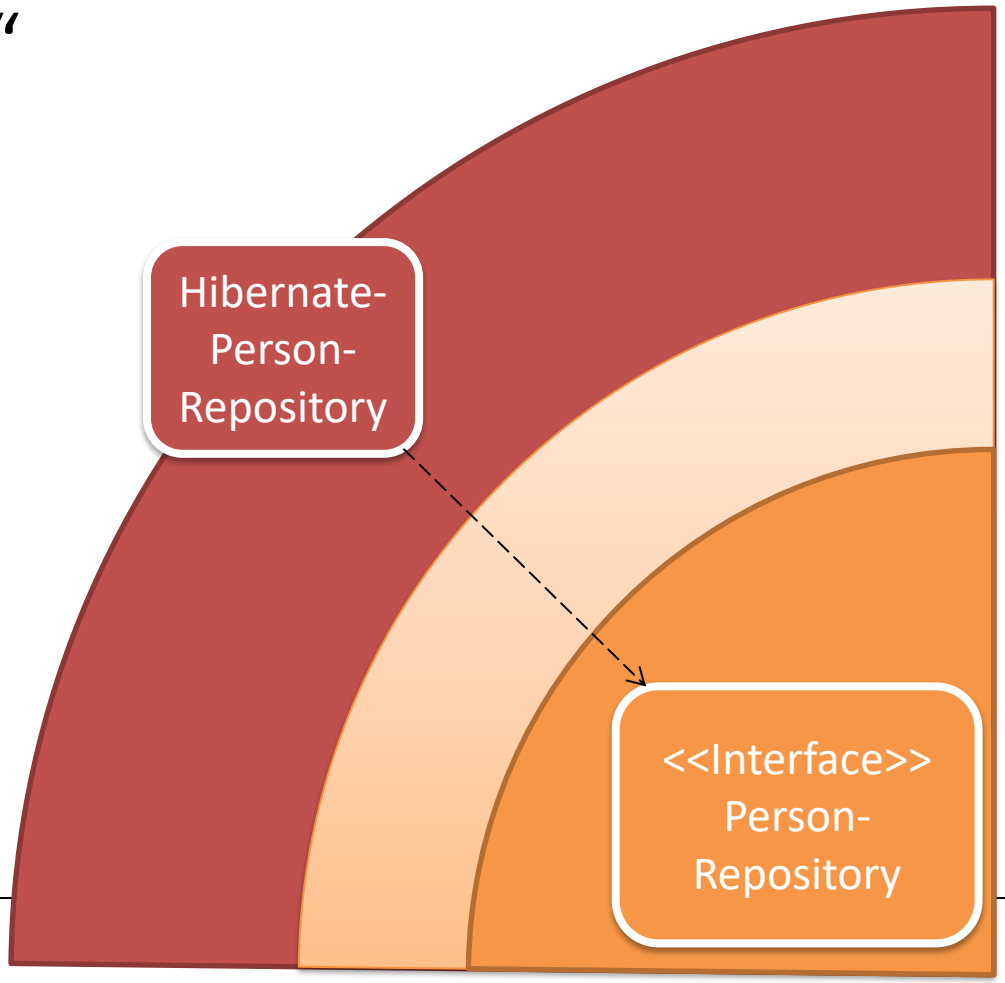
Domäne als isolierter Kern: Zwiebel statt Sandwich



DHBW

Hauptmerkmale der „Onion Architecture“

- Alle Abhängigkeiten zeigen von „Aussen“ nach „Innen“



Hauptmerkmale der „Onion Architecture“

- Alle Abhängigkeiten zeigen von „Aussen“ nach „Innen“ ->
  - Schichten im Inneren sind dann auch nie von weiter außen liegenden Schichten abhängig
- Technische Details werden innerhalb der Infrastruktur-Schicht definiert; UI, DB usw. sind **Klienten** der inneren Schichten
- Die inneren Schichten sind **frei von technischen Details** und daher weniger anfällig für Änderungen der UI usw.

Domain Layer

- Enthält Kernobjekte und Regeln der Fachlogik, im Falle von DDD also das Domänenmodell (Entities, Value Objects, Domain Services, …)
- Definiert Verträge (Abstraktionen) für die Infrastruktur-Schicht (Repositories, Notifications, Logging, …)

Application Service Layer (ASL)

- Normalerweise ist bei der Abbildung eines bestimmten **Anwendungsfalles (Use Case)** mehr als ein Domänenobjekt aus der Domänenschicht involviert
- Beispiel: Anwendungsfall „Ändern der Kundenadresse eines Auftrags“
  - benötigt Zugriff auf **CustomerRepository** und **OrderRepository**

Application Service Layer (ASL)

- Die ASL implementiert diese Anwendungsfälle als sog. **Application Services**
- Die ASL bildet damit eine **API für die Infrastrukturschicht**

Application Service Layer (ASL)

- Gleichzeitig bildet die ASL eine **Isolationsschicht** zwischen Infrastruktur und Domäne: die ASL gibt vor, wie die Außenwelt mit der Domäne kommunizieren darf und versteckt Domänen-Interna vor der Außenwelt
- Die Infrastrukturschicht ist dadurch weniger anfällig für Änderungen des Domänenkerns

Aufgaben eines Application Service

- Implementierung eines Anwendungsfalls
- Validierung, Übersetzung und Aufbereitung von Eingaben und Ausgaben
- Reporting
- Security

Aufgaben eines Application Service:
Implementierung eines Anwendungsfalls

- Ein Application Service bildet einen oder mehrere Anwendungsfälle ab
- Er nutzt dazu die Komponenten des Domänenmodells und orchestriert und koordiniert diese, um den gewünschten Anwendungsfall umzusetzen
- Ein Application Service enthält selbst keine Regeln; er weiß lediglich, welche Domänenobjekte er in welche Reihenfolge aufrufen muss

Aufgaben eines Application Service:
Implementierung eines Anwendungsfalls

- Ein Application Service folgt eher einem prozeduralen Programmierstil (siehe auch Entwurfsmuster „Transaction Script“)
- Bietet normalerweise keine Create/Read/Update/Delete-Methoden, sondern konkrete Methoden für den jeweiligen Anwendungsfall (Erinnerung: Ubiquitous Language)

Aufgaben eines Application Service:
Validierung, Übersetzung, Aufbereitung von Eingaben und Ausgaben

Validierung:

- Application Service stellt sicher, dass alle benötigten **Eingaben** zur Realisierung eines Anwendungsfalls **vorhanden** und **korrekt** sind
- Prüft **keine Regeln der Domäne**, sondern **technische Details** (korrekter Datentyp, korrektes Format, not null usw.)

Aufgaben eines Application Service:
Validierung, Übersetzung, Aufbereitung von Eingaben und Ausgaben

Übersetzung:

- **Übersetzt** die **Eingaben** der Infrastrukturschicht in **Domänenobjekte**
  - Bspw. Mapping von Request-Parametern auf ein Domänenobjekt
- **Übersetzt** bei Bedarf die **Ausgaben** der Domänenschicht für die Außenwelt (beispielsweise Übersetzung eines Kunden in ein allgemeineres **Data Transfer Objects** für eine REST-API)

Aufgaben eines Application Service:
Reporting

- Oft muss eine Anwendung verschiedene Berichte (Reports) liefern, um Auswertungen zu ermöglichen, beispielsweise
  - Umsätze
  - Lagerbestände je Artikel
- Zur Generierung solcher Berichte müssen normalerweise verschiedene Daten abgefragt und zusammengefasst werde – beispielsweise alle Artikel und alle Lagerbewegungen in einem bestimmten Zeitraum
- Ein Application Service kann diese Daten aggregieren und in einem speziellen Report-Objekt (Data Transfer Object) zurückliefern
- Darf ggf. auch nativ (und damit am Domänenkern vorbei) auf die Persistenzschicht zugreifen, wenn Performance dies erfordert

DHBW

Aufgaben eines Application Service:
Security

- Meist existieren in einer Applikation Anwendungsfälle, die nur Benutzern mit bestimmten Rechten zugänglich sind
- Application Services bieten sich daher auch für Authentifizierung und Autorisierung an
- Die konkrete Umsetzung ist von der Art der Zugriffskontrolle abhängig (rollenbasiert, …)

Wie stark soll die Domäne von der Außenwelt abgeschottet sein?

Es gibt zwei Möglichkeiten:

- Entweder dürfen (bestimmte) Objekte des Domänenmodells die ASL passieren und an die Außenwelt weitergegeben werden
- oder die Außenwelt hat keine Kenntnis von den Objekten des Domänenmodells

Diskussion

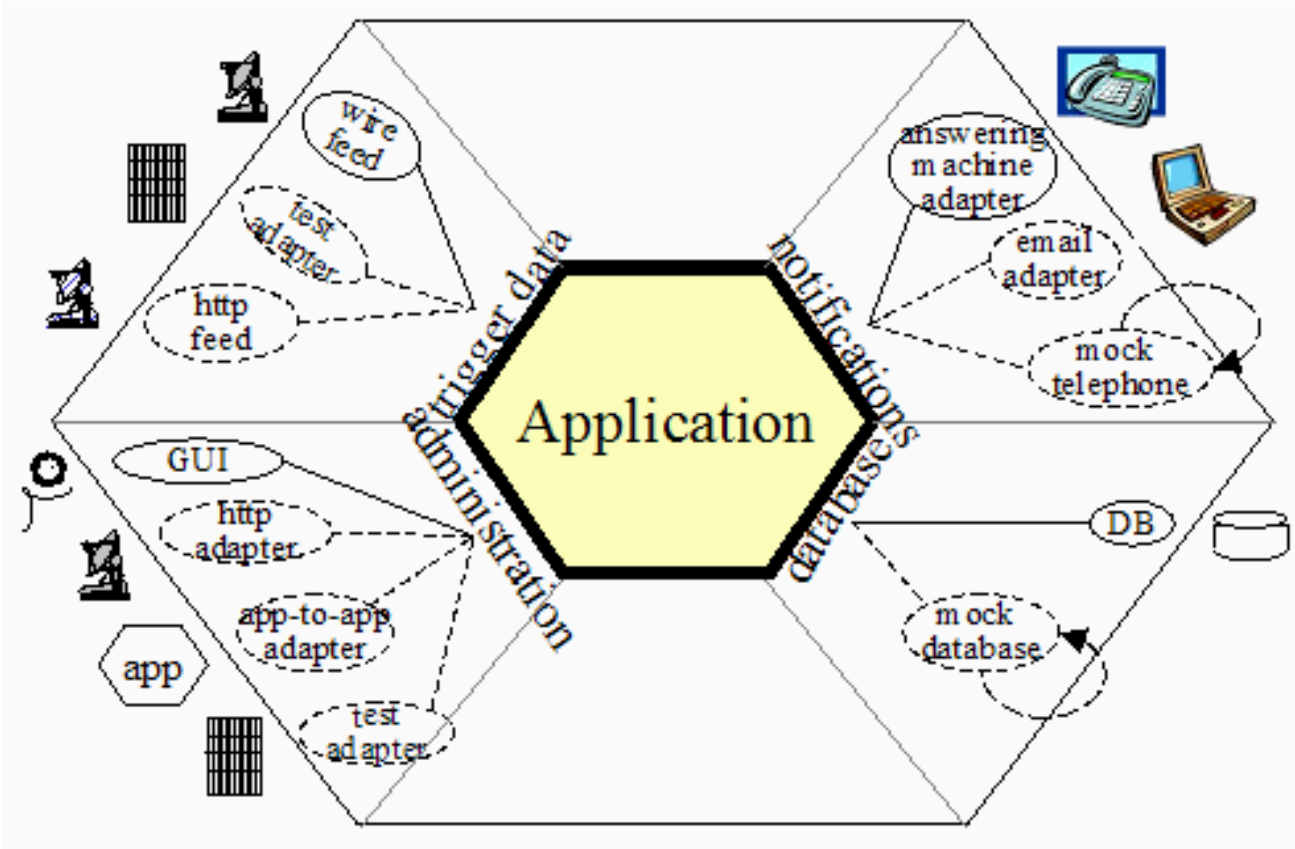
Infrastructure Layer

- Die Infrastruktur-Schicht stellt die **technischen Details** einer Anwendung bereit
- Diese Details sind in der äußersten Schicht angesiedelt – fern vom Kern
- Beispiele:
  - UI
  - Web Services
  - Datenbankzugriff
  - Anbindung von Fremdsystemen

Varianten der Onion Architecture

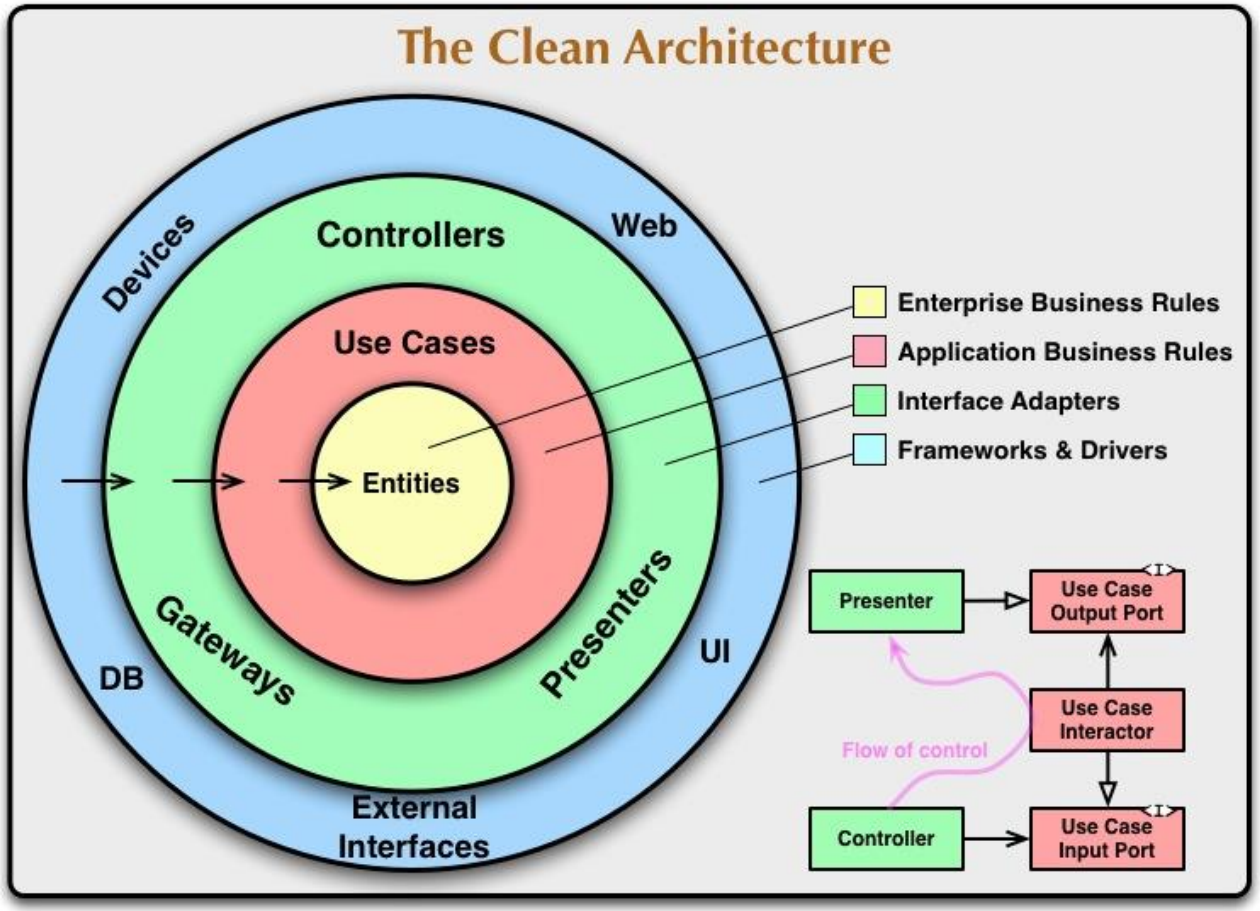
- Aktuell existieren mehrere Varianten der „Onion Architecture“, unter anderem:
  - Hexagonale Architektur / Ports-and-Adapters
  - Clean Architecture
- Zwar unterscheiden sich die genannten Architekturstile im Detail, das Grundprinzip (Abhängigkeiten zeigen von Außen nach Innen) ist aber immer dasselbe.

Hexagonale Architektur
a.k.a. Ports-and-Adapters



By Alistair Cockburn

Clean Architecture



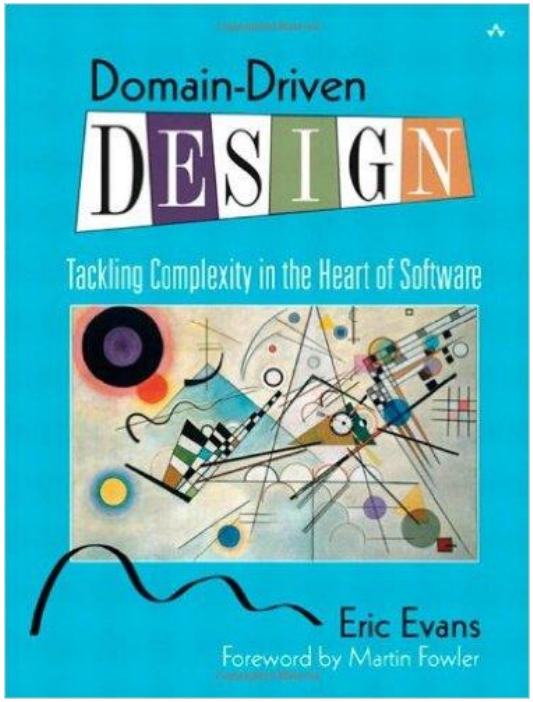
By Robert Martin

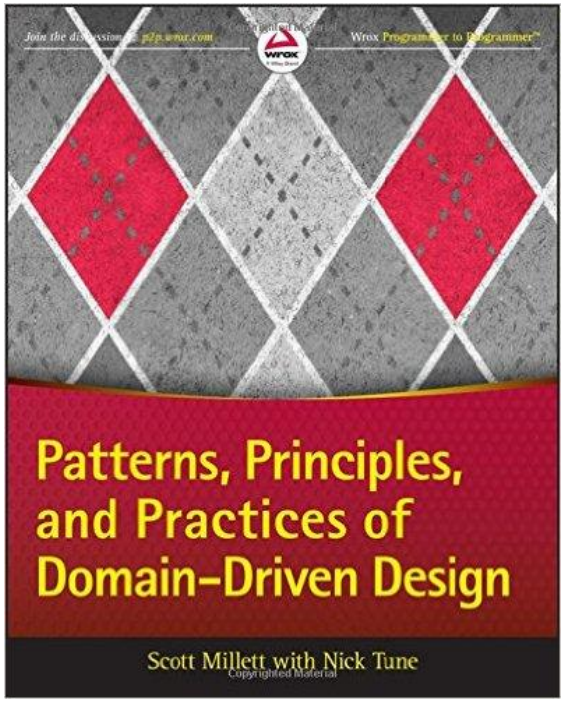
3.3 Fazit und Ausblick

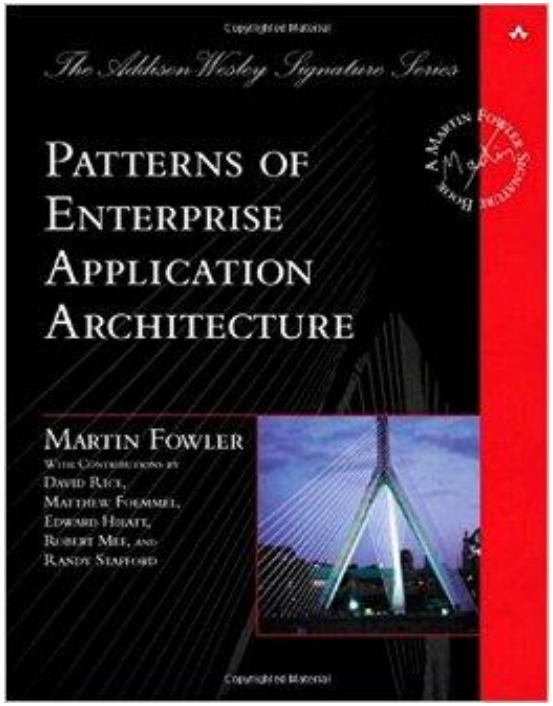
Kritik

- Framework wird isoliert / Arbeit gegen das Framework
- Höhere Komplexität
- Mehr Aufwand durch Zwischencode (Indirektion) und Isolation
- Höhere Komplexität wird nicht überall benötigt
- Architektur ist von Testbarkeit getrieben, nicht von Nutzen

Literaturempfehlungen







Quellen

- Reeves, J. W. (2005), 'Code as Design: Three Essays', *Developer.**
- Meyer, B. (2014), *Agile!: The Good, the Hype and the Ugly*, Springer International Publishing, Switzerland
- Ko, A.J.; Myers, B.A.; Coblenz, M.J.; Aung, H.H., "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," in *Software Engineering, IEEE Transactions on* , vol.32, no.12, pp.971-987, Dec. 2006
- Evans, E. (2004), *Domain-driven design : tackling complexity in the heart of software* , Addison-Wesley , Boston
- IEEE 90: IEEE Standard Glossary of Software Engineering Terminology (1990) by Institute O. Electrical, Electronics E. (ieee)

DHBW