Domain-Driven Design & Onion Architecture

„Oldies but Goldies“

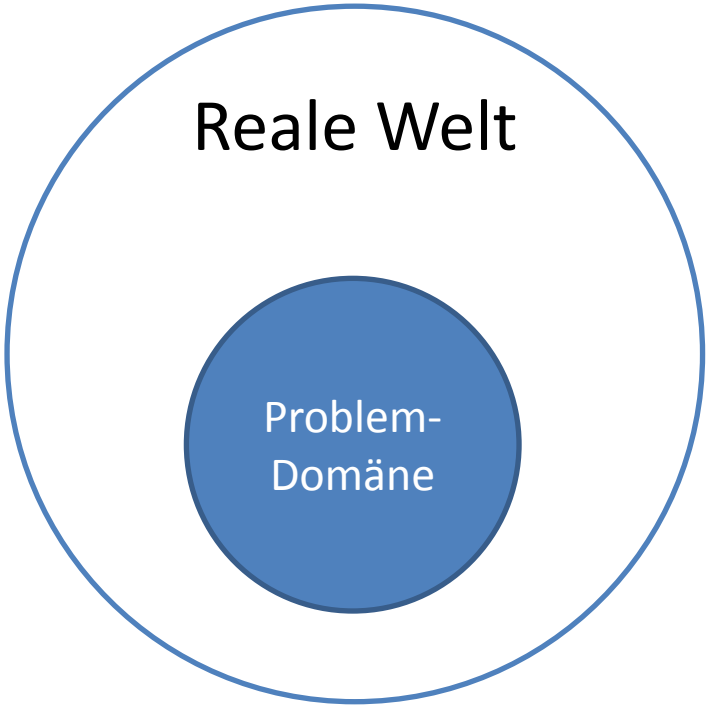
DHBW

Übersicht: Domain-Driven Design

- Domain-Driven Design (DDD) ist eine Herangehensweise an die Modellierung von Software
- Idee:
  - Durch Kollaboration von Domänenexperten und Entwicklern finden sich wichtige Konzepte, Regeln und Prozesse der **Problemdomäne** in der Geschäftslogik der Software wieder
- Ziel: bessere Verständlichkeit, Wartbarkeit und Evolvierbarkeit
- DDD bietet:
  - **Strategische Entwurfsmuster** zur Analyse & Abgrenzung der Problemdomäne
  - **Taktische Entwurfsmuster** zur Abbildung der Problemdomäne in Code

Übersicht: Domain-Driven Design

Der Fokus von DDD:



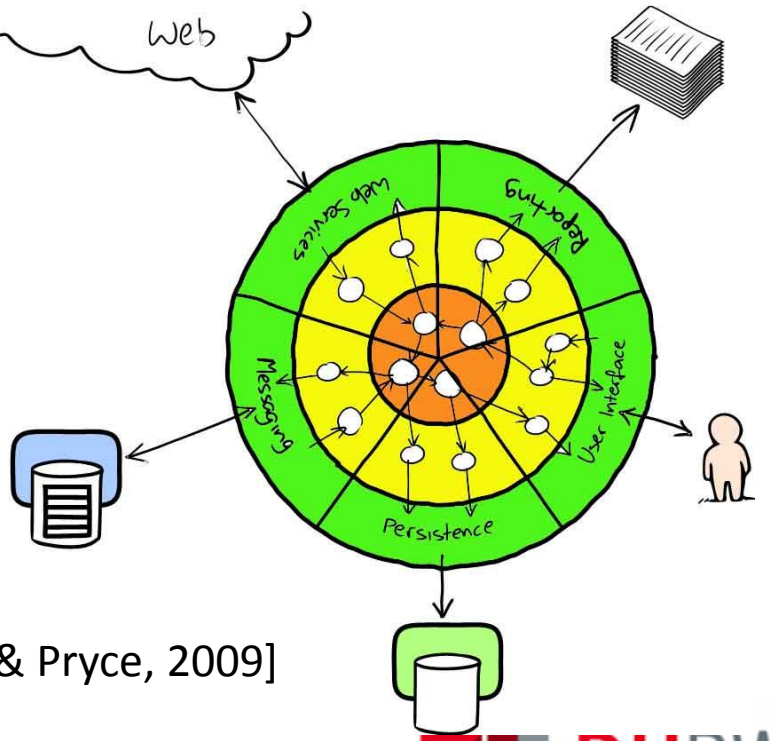
Beispiele für Problemdomänen:

- WebShop
- Lagerverwaltung
- Bankenwesen
- Gesundheitsmanagement
- Schach

DHBW

Übersicht:
Onion Architecture

- Architektur-Stil, der die Geschäftslogik vom Rest der Anwendung isoliert
- Ziel: die Geschäftslogik wird nicht durch Fremdeinflüsse (Persistenz, UI, usw.) „verunreinigt“
- „Natural fit“ für DDD



[Freeman & Pryce, 2009]

Relevanz

- DDD: Grundlagenwerk mit weitreichenden Auswirkungen
- Regelmäßig Vorträge auf Konferenzen
- Überschneidung mit vielen Themen (agile, continuous integration, craftsmanship, MicroServices, …)
- Regelmäßig neue Literatur
- Aktueller Artikel (10/2015) in der iX

Ziele dieser Vorlesung

- Sie verstehen die Probleme bei der und Einflüsse auf die Entwicklung von Software
- Sie erkennen, warum Design für die Entwicklung von Software wichtig ist
- Sie kennen die Kernkonzepte des DDD
- Sie realisieren, dass das Verstehen der Problemdomäne ein wichtiger Teil des Entwicklungsprozesses ist und
- Sie verfügen über grundlegendes Handwerkszeug, um:
  - eine Problemdomäne zu analysieren
  - das für die Anwendung wichtige Fachwissen zu erfassen
  - **das Wissen in der Software „einzufangen“**

Gliederung

1. Einführung
    1. Was ist Design und warum braucht man Design
2. Domain-Driven Design
    1. Der harte Kern: die Grundidee des DDD
    2. Strategie und Taktik: die zwei Seiten des DDD
    3. Das Domänenmodell: Bindeglied zwischen Strategie und Taktik
    4. Die Grundbausteine des DDD
    5. DDD in der Praxis
    6. Fazit & Ausblick
3. Onion Architecture
    1. Vom Sandwich zur Zwiebel: eine zeitgemäße Schichten-Architektur
    2. Port und Adapter: die Umkehrung der Verhältnisse
    3. Fazit & Ausblick

1. Einführung

Was ist Design und warum braucht man Design

Was ist eigentlich „Design“

Frage: Ist das Design?

UML Framework Entwurfsmuster

Architektur Testen

Datenbank Refactoring

“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]

Meine Definition: **die Summe aller Entscheidungen, die Einfluss darauf haben, wie ein Problem mit Software gelöst wird**.

Design und Code

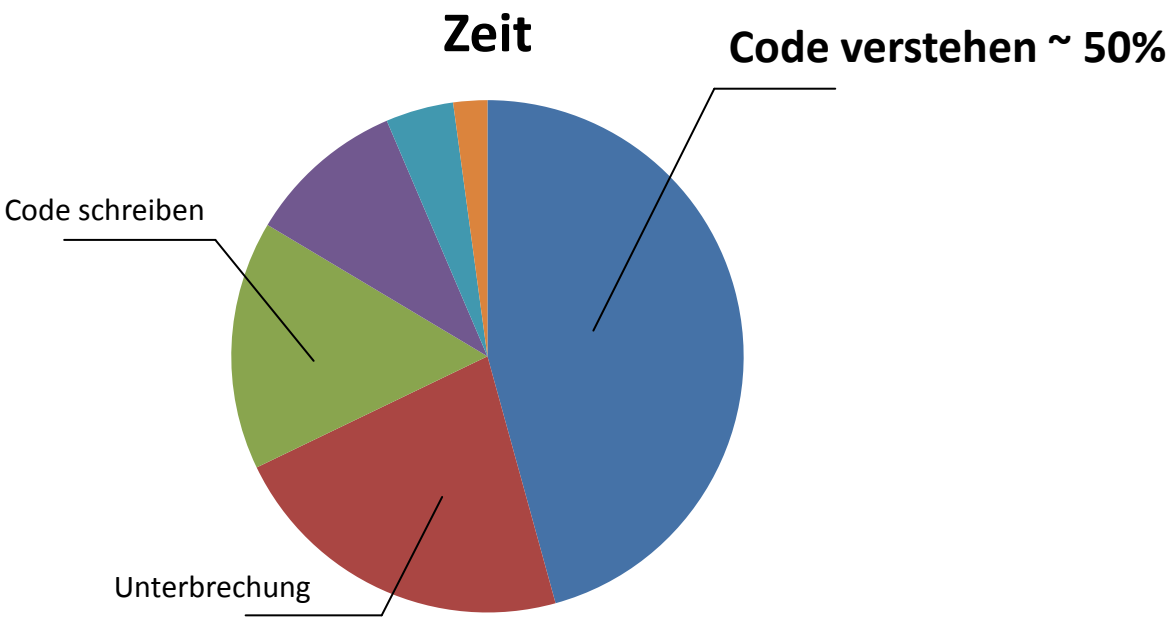
- Frage: Welches Artefakt repräsentiert vollständig was eine Software tut und wie sie es tut?

  -> der Quellcode

- Frage: Welche Relevanz haben Designdokumente, die vom Quellcode abweichen (weil veraltet, in Realität nicht umsetzbar, usw.)

Warum braucht man „Design“

Frage: Womit verbringen Entwickler den Großteil ihrer Zeit?

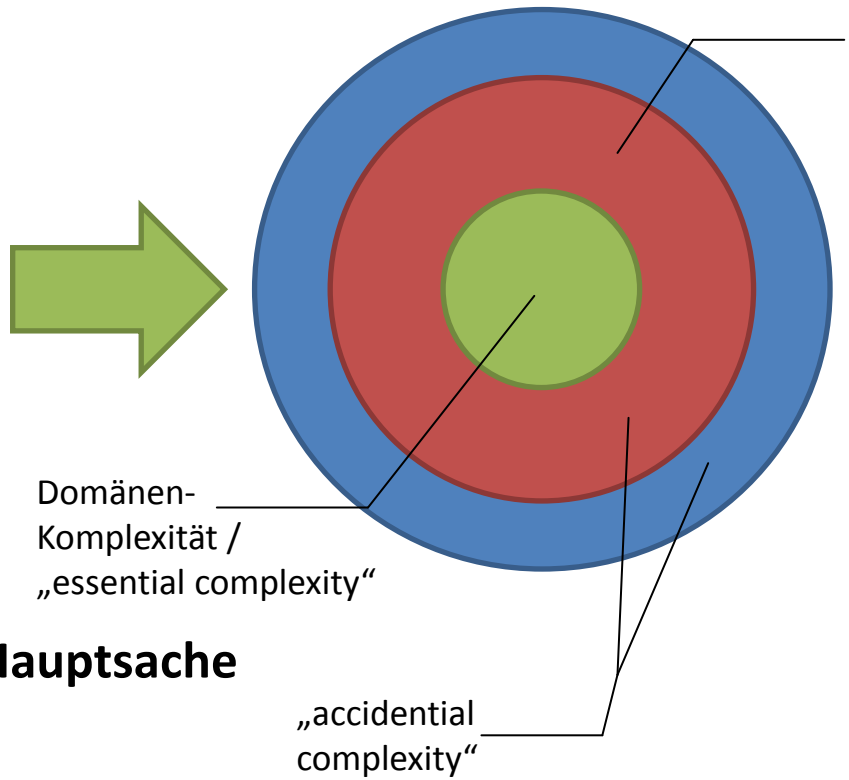


Ko et al., 2006

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



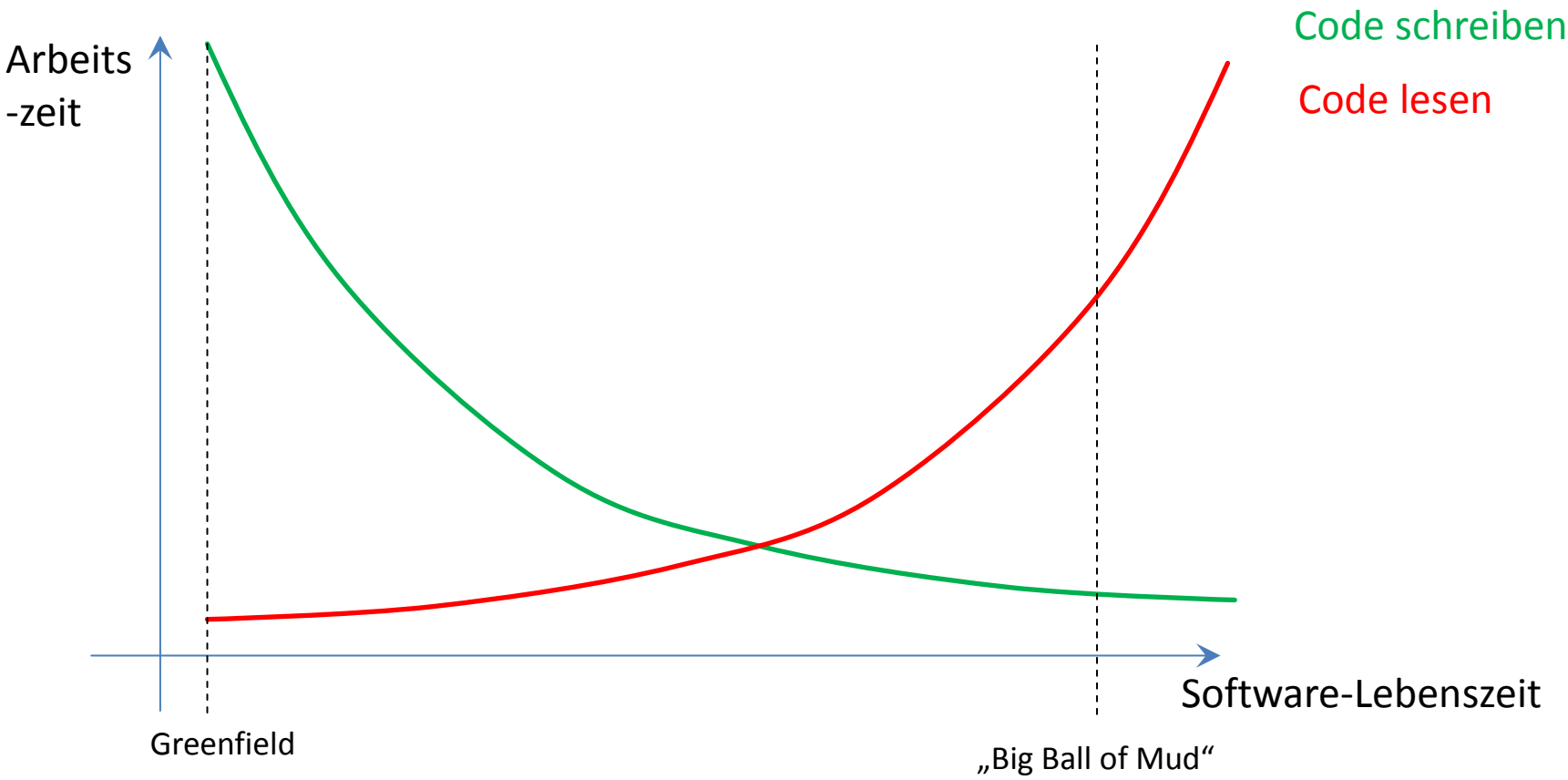
- Legacy-Systeme
- Persistenz
- UI
- Framework
- Infrastruktur / Hardware
- Deployment

Hauptsache

Nebensache

Auswirkungen von Komplexität

Auswirkungen von Komplexität



Big Ball of Mud

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 **großer Klumpen Matsch**

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

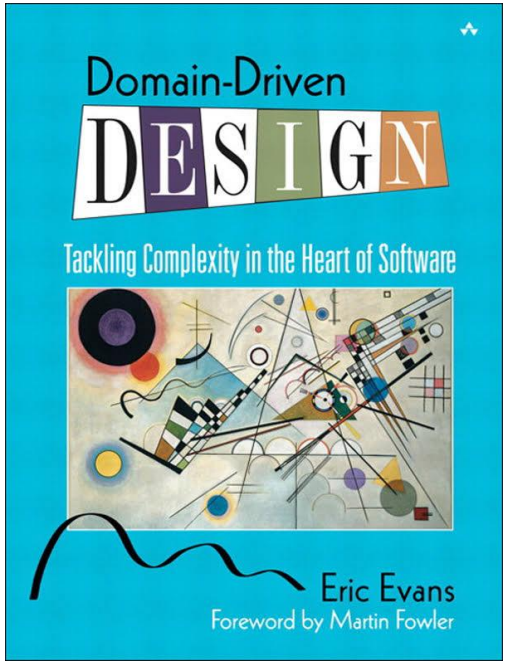
Key Facts

- Analyse, Modellierung, Code: das alles (und mehr) ist Design
- Code ist die „single source of truth“
- Design ist nutzlos, wenn es sich nicht im Code wiederspiegelt
- Jede Software ist komplex
- (gutes) Design macht Komplexität beherrschbar
- Ohne Design breitet sich Komplexität unbeherrscht aus und vervielfacht sich (BBoM)

2. Domain-Driven Design

Übersicht Domain-Driven Design

- Philosophie + Werkzeugkasten (Patterns, Methoden) für das Design **komplexer** Software
- Erstmals vorgestellt 2003 im „Big Blue Book“ von Eric Evans
- Standardwerk der Software-Entwicklung
- Grundannahmen:



“ -Place the project’s primary focus on the **core domain** and **domain logic**.
-Base complex designs on a **model**
-Initiate a creative **collaboration** between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem. ”

http://dddcommunity.org/learning-ddd/what_is_ddd/

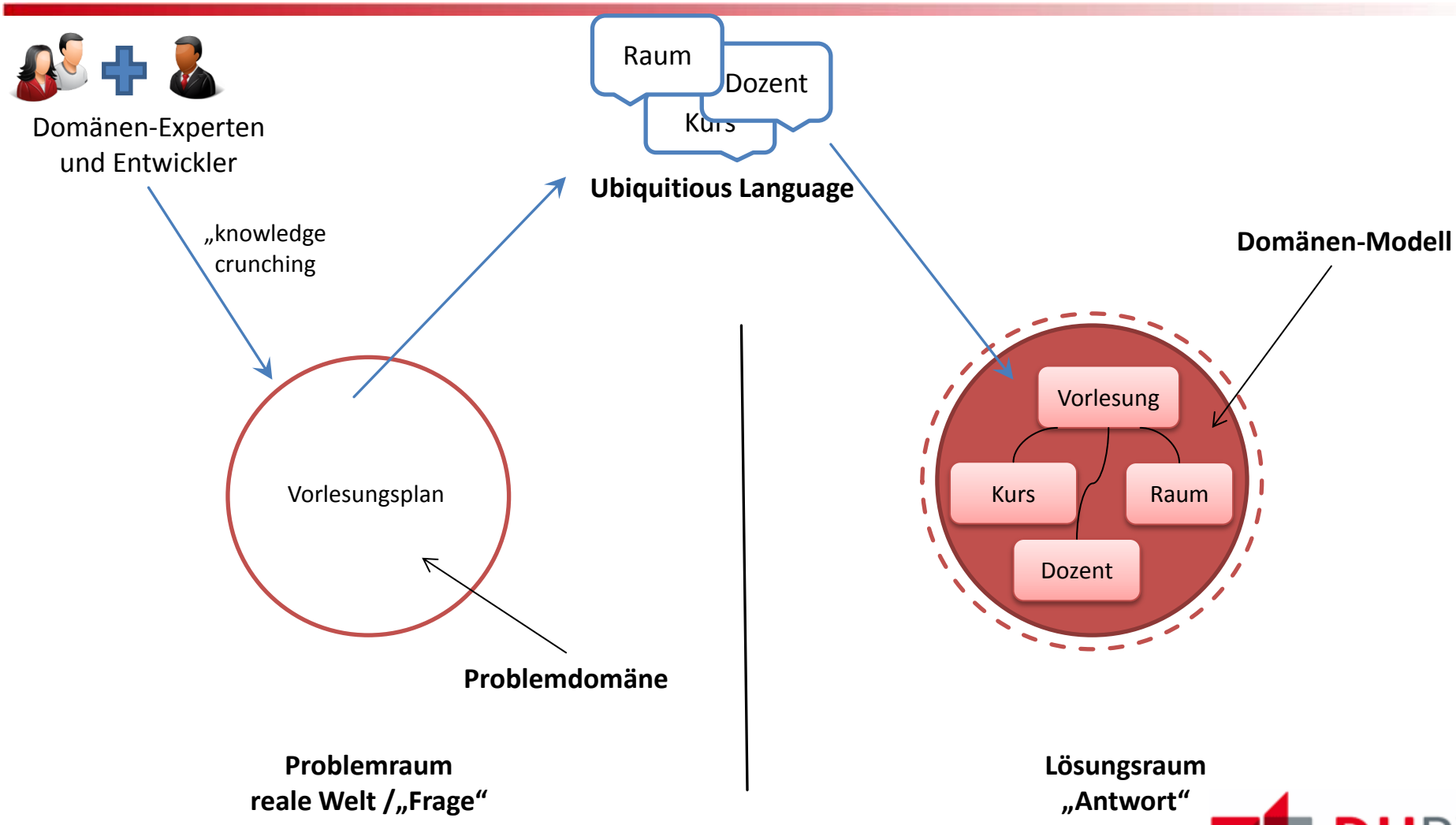
Der harte Kern: die Grundidee des DDD

- Um das **relevante Fachwissen** aus der Problemdomäne zu destillieren, müssen **Domänenexperten** und **Entwickler** eng zusammenarbeiten und eine **gemeinsame Sprache** entwickeln
- Damit verständlich ist, wie Code(=Design) die Anforderungen innerhalb einer Problemdomäne löst, müssen die **relevanten Konzepte, Prozesse und Regeln** (=das Fachwissen) der Problemdomäne sich im Code **wiederspiegeln** (**Fachlogik)**
- Damit das Fachwissen im Code nicht durch Nebeneinflüsse wie Persistenz usw. verschleiert wird, muss die Fachlogik von diesen Nebeneinflüssen **isoliert** werden

Beispiel

Präsidium: „Wir benötigen eine neue Software für die Verwaltung von Vorlesungen“

Was haben wir gerade getan?



Strategie und Taktik: die zwei Seiten des DDD

- Die strategische Seite des DDD beschäftigt sich mit der Analyse und der genauen Abgrenzung der Problemdomäne innerhalb bestehender Systeme
- Die taktische Seite des DDD beschäftigt sich damit, wie das durch die Analyse gehobene Wissen in Code umgesetzt werden kann

Knowledge crunching

- Domänenexperten und Entwickler betreiben iterativ „knowledge crunching“ – sie „beißen“ sich wechselseitig durch die Anforderungen
- Fragen stellen, bis alle Begriffe klar sind und die relevanten Konzepte von den irrelevanten getrennt sind
- Domänenexperten und Entwickler entwickeln eine gemeinsame Sprache:
  die Ubiquitous Language (UL)

Gemeinsame Sprache / Ubiquitious Language

- Begriffe, die zwischen Domänenexperten und Entwicklern geteilt werden
- Wird überall (Diskussion, Schriftverkehr, Code) konsequent verwendet
- Entwickler sorgen dafür, dass Domänenexperten möglichst genaue Begriffe finden und ihre Ideen überdenken
- Domänenexperten sorgen dafür, dass die Entwickler das Problem besser verstehen

Gemeinsame Sprache / Ubiquitious Language

- Beseitigt Mehrdeutigkeit
- Deckt Inkonsistenzen auf
- Erleichtert Kommunikation
- Scheidet relevante Konzepte von irrelevanten
- Erleichtert langfristige, gemeinsame Arbeit am Projekt
- Lern-Werkzeug für Domänenexperten und Entwickler

Gemeinsame Sprache / Ubiquitious Language

User Story Example:

NO

When User logs on with valid credentials, an empty panel is displayed.

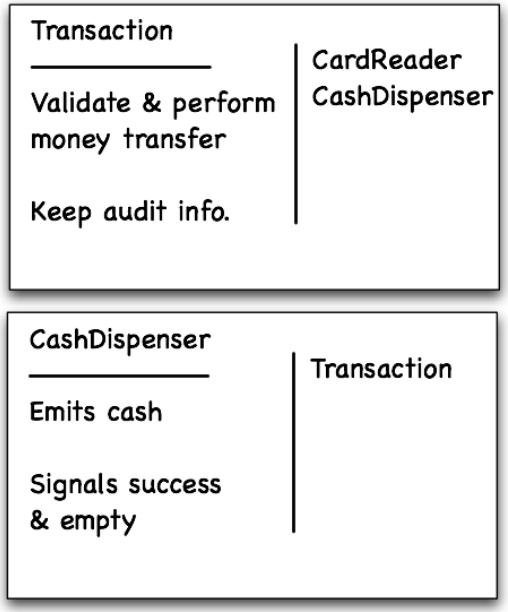
YES

When Player logs on with valid credentials, an empty board game is displayed.

(from a Tic Tac Toe Game software example)

Knowledge crunching: Werkzeuge

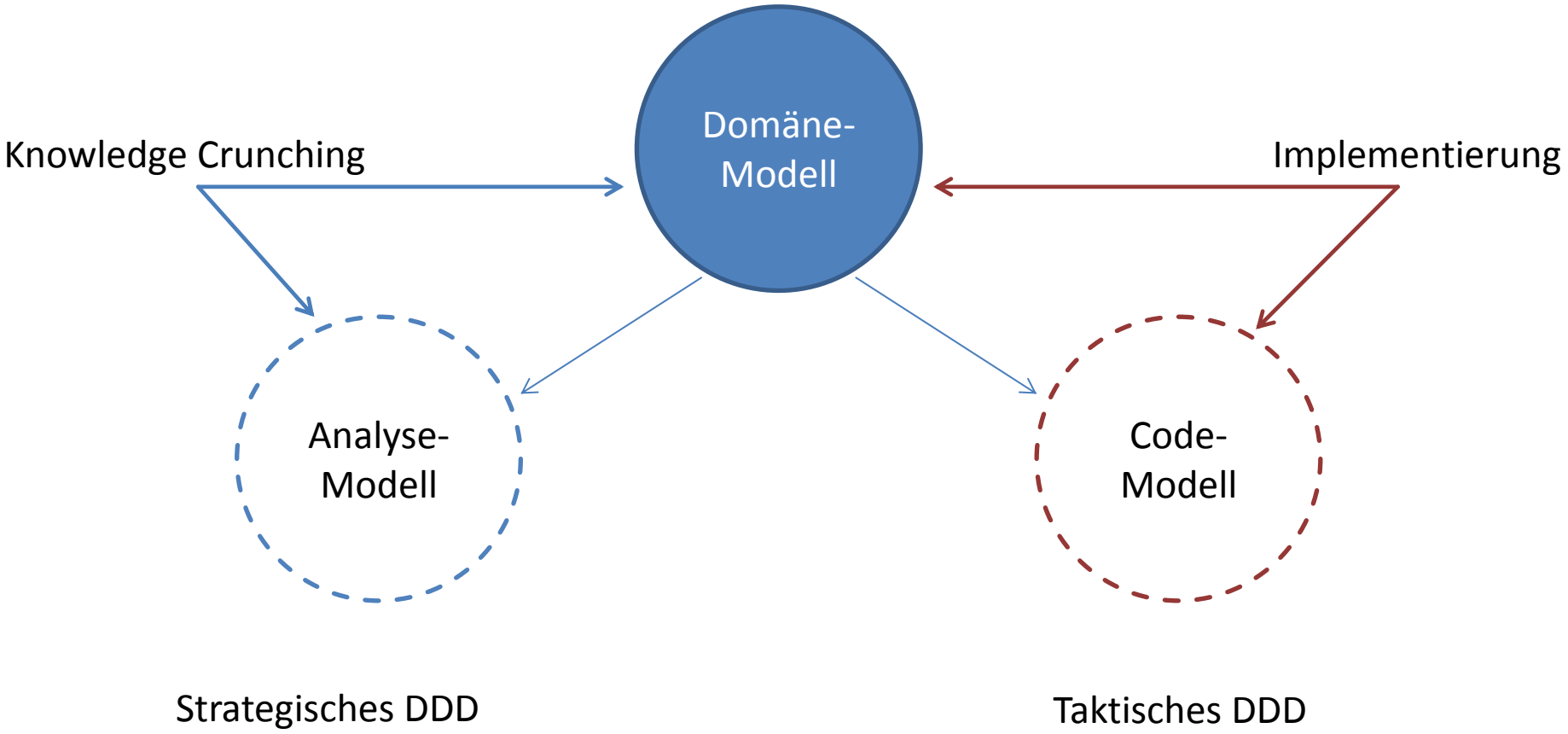
- Use Cases
- Sketches / Mockups
- Class Responsibility Cards
- Rapid Prototyping
- …



CRC

DHBW

Das Domänenmodell: Bindeglied zwischen Strategie und Taktik



2.4 Die Grundbausteine des DDD

Die Grundbausteine des DDD

- Wiederholung DDD:
  - wiedererkennbare Abbildung von Domänenkonzepten in der Software
  - Freiheit des Domänenmodells von technischen Details
- Damit dies gelingt, liefert DDD eine Auswahl von „taktischen“ Entwurfsmustern, welche diese Ziele unterstützen

Die Grundbausteine des DDD

- 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**
- Repositories, Factories:
  - Kapseln die Logik für das **Persistieren** und **Erzeugen** von Entities, Value Objects und Aggregates
- Modules:
  - Dienen zur **Strukturierung**, **Unterteilung** und **Kapselung verwandter Domänenobjekte** innerhalb des Domänenmodells und
  - Fördern dadurch **geringe Kopplung** und **hohe Kohäsion**

Value Objects

Beispiel: Jim und Joe sind **Personen** und haben ein Gewicht

```
public class Person {

    private Float weight;
    private String weightUnit;

    public Person(Float weight, String weightUnit) {
        super();
        this.weight = weight;
        this.weightUnit = weightUnit;
    }

    public void changeWeight(Float newWeight) {
        this.weight = weight;
    }

    public void changeWeightUnit(String newWeightUnit) {
        this.weightUnit = newWeightUnit;
    }
```

```
//...
Person jim = new Person(Float.valueOf(80.0F), "kg");
Person joe = new Person(Float.valueOf(80.0F), "kg");
//...
```

Value Objects

```java
public class Person {

    private Weight weight;

    public Person(Weight weight) {
        super();
        this.weight = weight;
    }

    public void changeWeight(Weight newWeight) {
        this.weight = weight;
    }

    public Weight getWeight() {
        return weight;
    }
}
```

Ergebnis

```java
public final class Weight {

    private final Float value;
    private final String weightUnit;

    public Weight(Float value, String weightUnit) {
        super();
        assertValueIsGreateThanZero(value);
        this.value = value;
        this.weightUnit = weightUnit;
    }

    private void assertValueIsGreateThanZero(Float valueToCheck) {

    public Float getValue() {
        return value;
    }

    public Weight changeValue(Float newValue) {
        return new Weight(newValue, this.weightUnit);
    }

    public String getWeightUnit() {
        return weightUnit;
    }

    @Override
    public String toString() {
        return this.value.toString() + " " + this.weightUnit;
    }
}
```

Value Objects

- Was geschieht beim Vergleich zweier gleicher Gewichte?

```
//...
Weight weight1 = new Weight(80F, "kg");
Weight weight2 = new Weight(80F, "kg");

System.out.println(weight1.equals(weight2));
//...
```

- Gibt „false“ aus. Warum?
  - Weil „equals()“ per Default prüft, ob beide Referenzen auf das selbe Objekt zeigen!

Value Objects

- Da Value Objects keine Identität haben, sind sie allein durch ihre Attribute vergleichbar
- Jedes VO muss deshalb equals() und hashCode() überschreiben

Value Objects: Erkenntnisse

- Value Objects (VO) sind einfache Objekte **ohne eigene Identität**
- VO sind **immutable**
- Ein VO wird durch seine **Eigenschaften** oder **Werte** beschrieben -> daher: Value
- Ein VO **MUSS** in Java equals() und hashCode() überschreiben

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

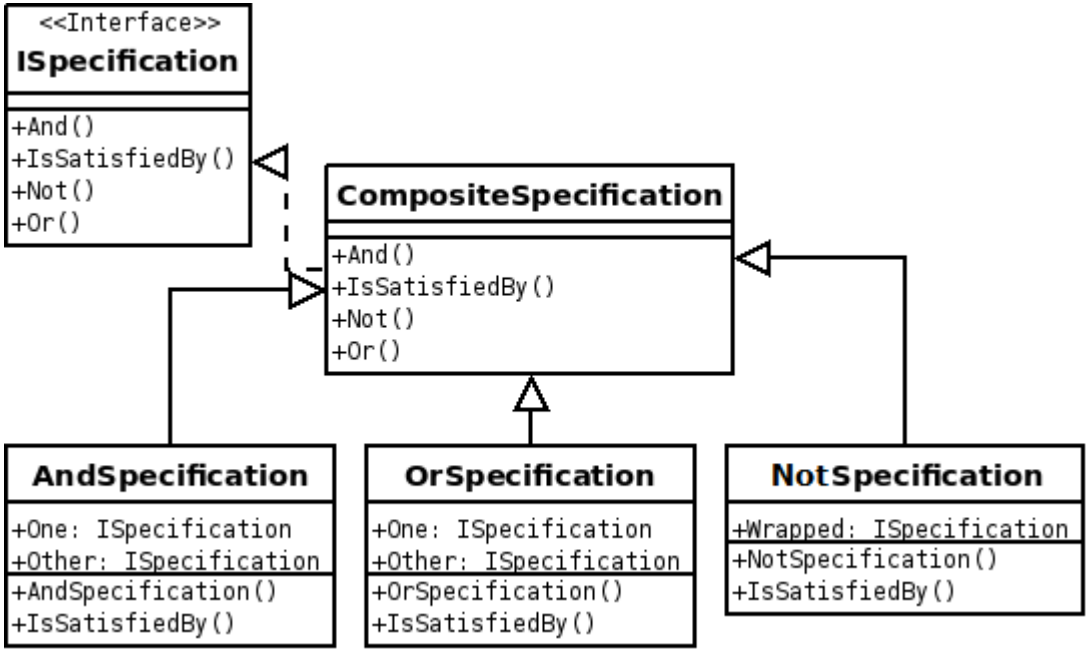
- Verbessern die Deutlichkeit und Verständlichkeit
- Kapseln Verhalten und Regeln
- Frei von Seiteneffekten (beispielsweise Aliasing)
- Selbst-Validierend
- Leicht testbar

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

Exkurs: Specifications



Persistierung von Value Objects: der knifflige Teil

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
- Funktioniert nur bei 1:1-Beziehungen
- Erlaubt einfache Abfragen über Elemente des VO

Nachteile:

- Ggf. Denormalisierung der DB

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) {}
}
```

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

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

Exkurs: 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 PersonManager {

    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);
        //...even more of this crap
    }
}
```

Exkurs: Anämisches Domänenmodell

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

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

Exkurs: Anämisches Domänenmodell

Die Alternative:

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    private Name name;

    private Address address;

    public void moveTo(Address address) {
        //validation if necessary
        this.address = address;
    }
}
```

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;
        }
}
```

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

DHBW

„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)

UUID als String

```
public class Person {

        @Id
        private String uuid;

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

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())
        }
}
```

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

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

Einsatz von Domain Services: Modellieren 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
- Beispiele:
  - USt-IdNr.-Prüfung in einer Rechnung
  - Schufa-Prüfung bei Kreditvergabe
  - Benachrichtigungs-Dienst zum Senden einer Email
  - Logging-Dienst

Einsatz von Domain Services: Definition eines Vertrags

- Innerhalb der Domäne kann dazu ein Domain Service als Vertrag (Interface) definiert werden
- Außerhalb der Domäne kann dann ein „Dienstleister“ (beispielsweise die Infrastruktur-Schicht) diesen Vertrag implementieren und die benötigten Funktionen bereitstellen

Einsatz von Domain Services: Definition eines Vertrags

```
public class SoapSchufaCheck implements SchufaCheck {

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

Infrastruktur-Schicht

Domänen-Schicht

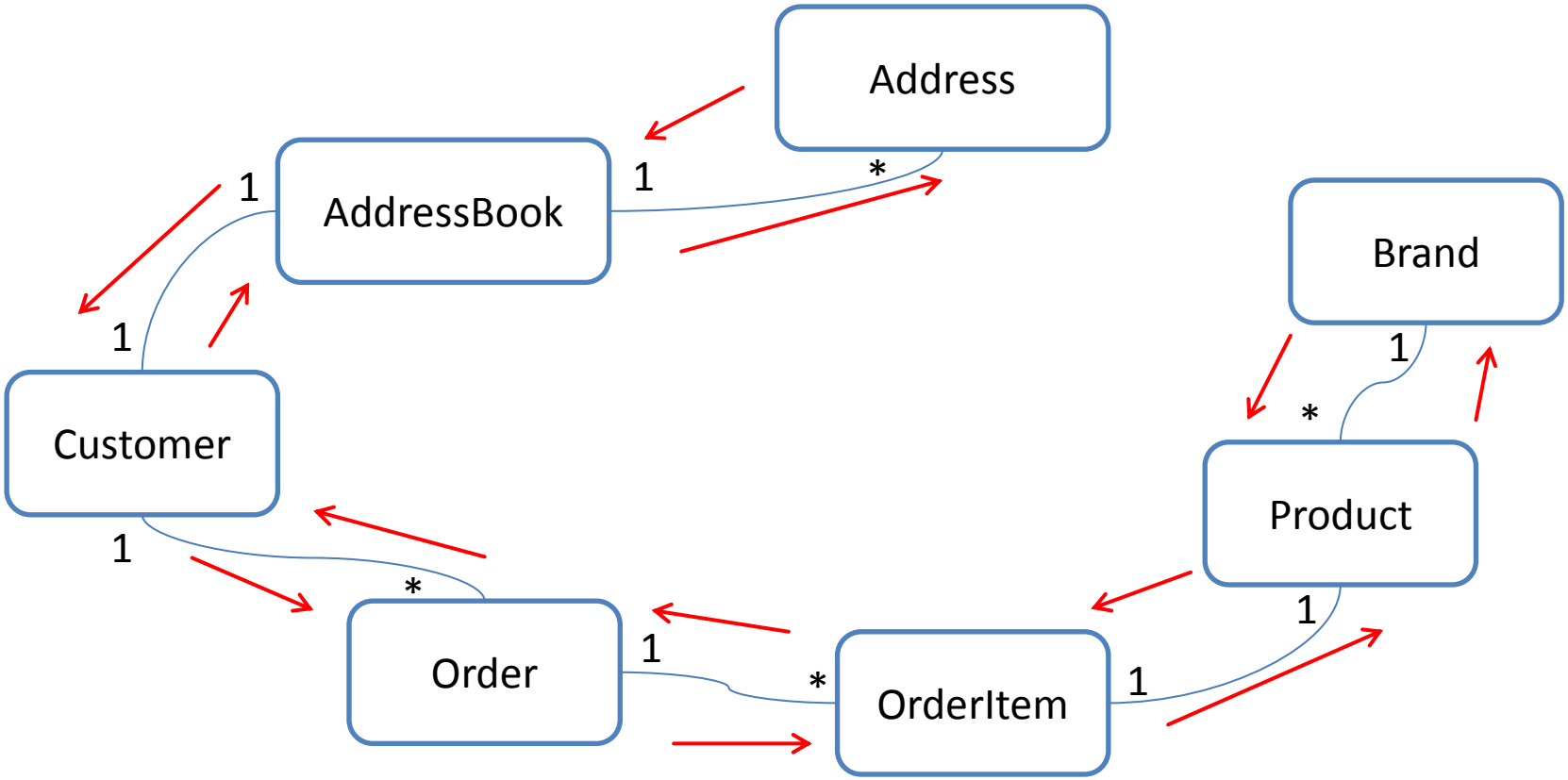
```
public interface SchufaCheck {

        CreditWorthiness checkCreditWorthinessOf(Customer customer);
}
```


DHBW

Eigenschaften eines Domain Service

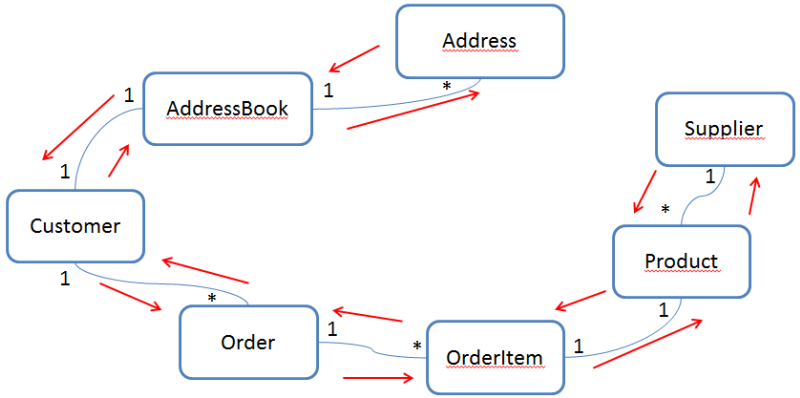
- Erfüllt **Funktion**, die **nicht** in einer **Entity** oder **VO** modelliert werden kann
- Operiert **ausschließlich** mit anderen **Elementen des Domänenmodells** für Eingabe und Ausgabe
- Domain Service und seine öffentlichen Methoden **verkörpern Konzepte der UL**
- Ist **statuslos**

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

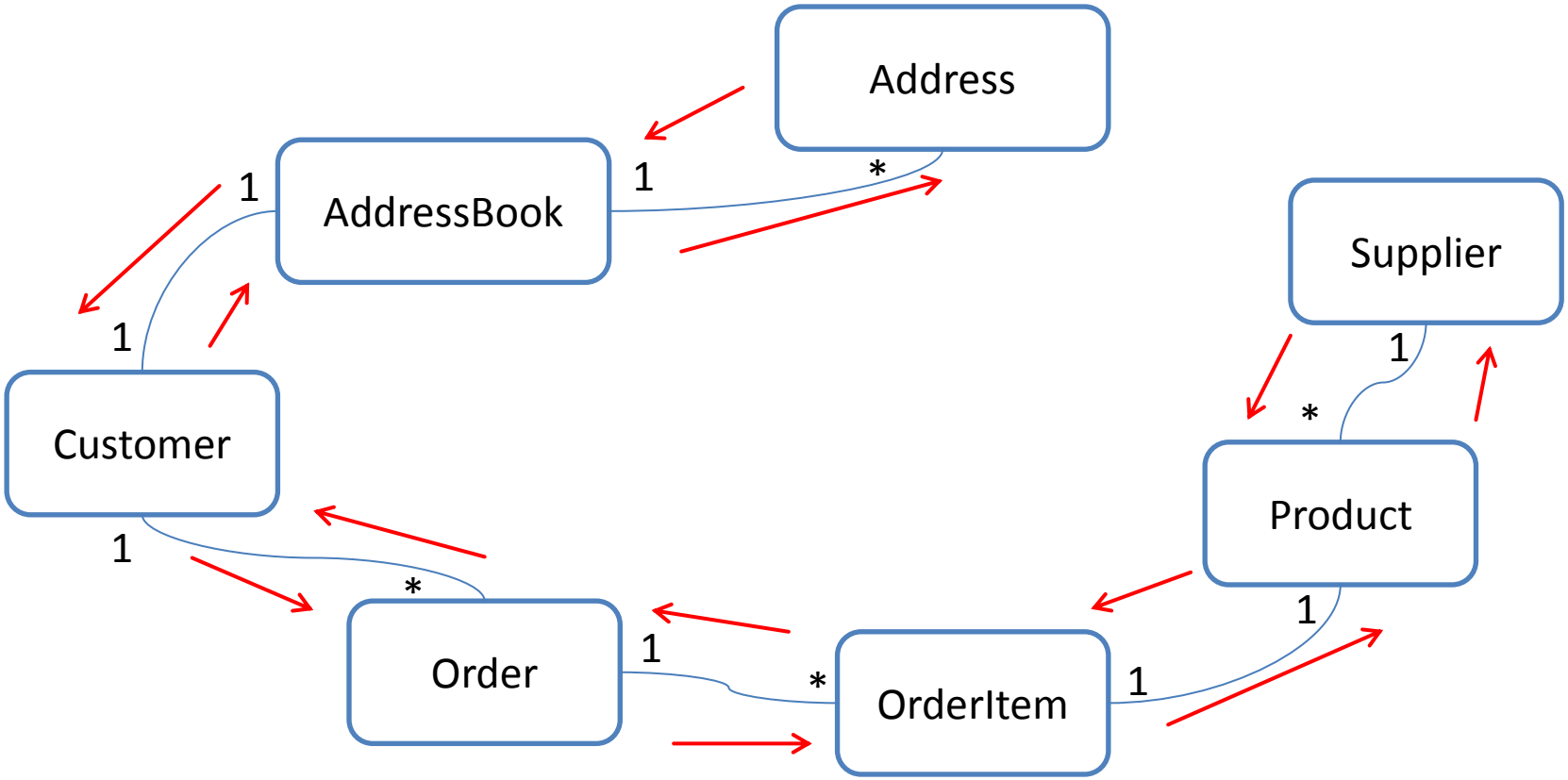
„Aggregates are the most powerful of all tactical patterns, but they are one of the most difficult to get right.“ [Millett, 2015]

Gruppieren von Entities und VO

Zwei grundsätzliche Möglichkeiten:

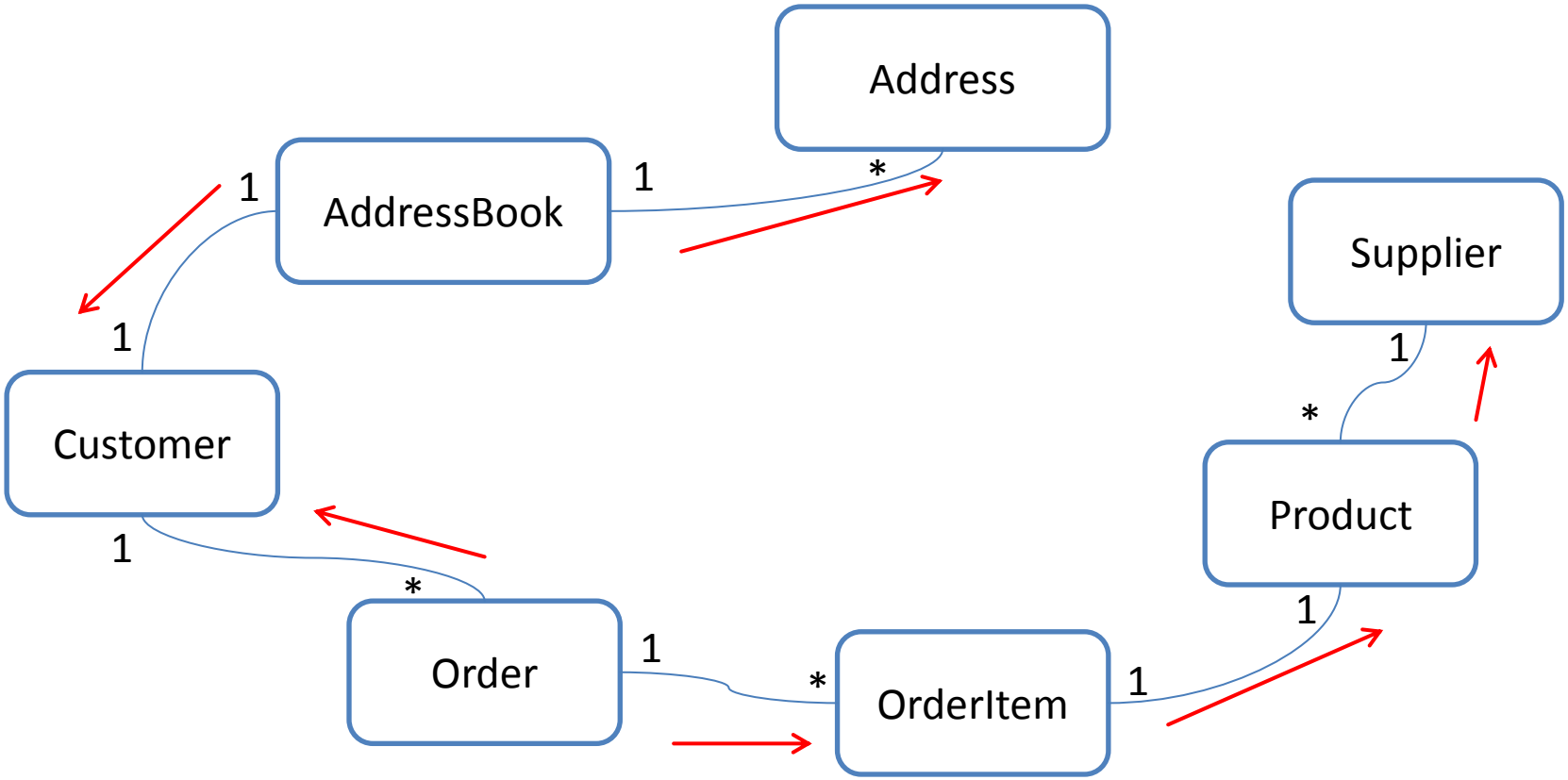
- Einschränkung der Assoziationsrichtung
- Ersetzen von Objektreferenzen durch IDs

Beispiel: Einschränkung der Assoziationsrichtung



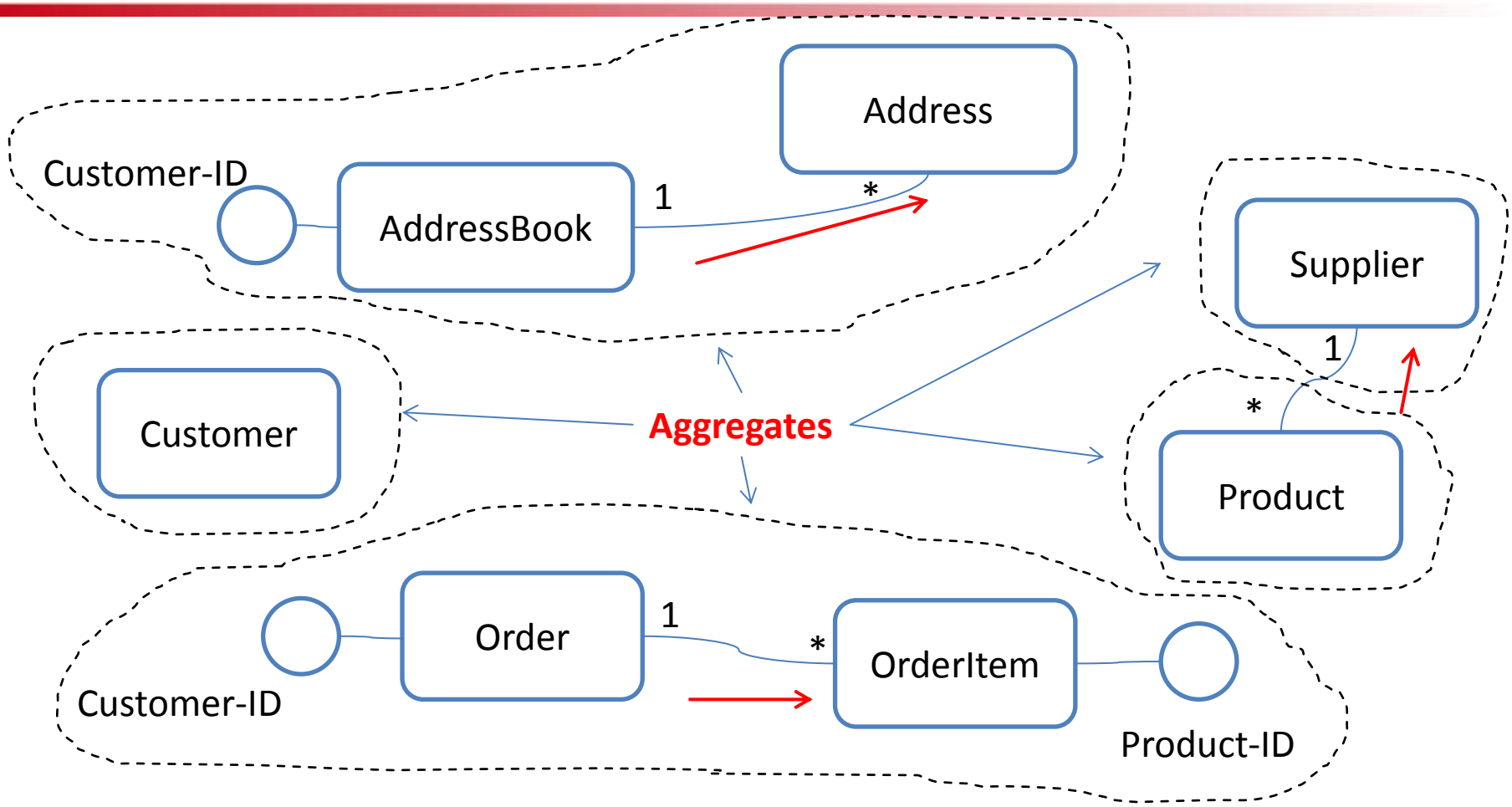
DHBW

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;
    //...
}
```

Ersetzen von Objektreferenzen durch IDs: Nachteile

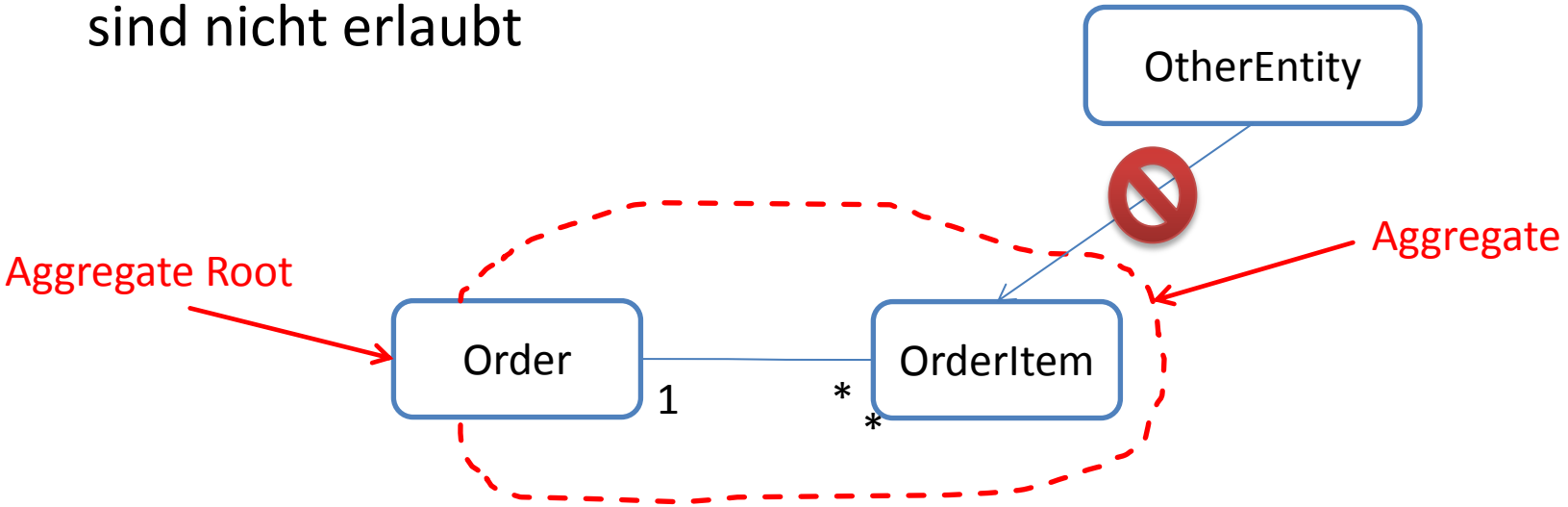
- Wenn DB über O/R-Mapper erzeugt und aktualisiert wird, sind keine Fremdschlüssel mehr abbildbar
- Das ist in der Praxis aber normalerweise nicht relevant, da die DB über eigene Migrationen aufgesetzt und aktualisiert wird
- Es sind weitere Abfragen nötig, um beispielsweise auf das AddressBook eines 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:
    - Einhalten 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

Reduzierter Aufwand für Verwaltung

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

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: was Sie mitnehmen sollten

- Aggregates zwingen dazu, sich mit wichtig Persistenz-Fragen auseinander zu setzen
- Wenn diese Fragen ohne Nachteile oder Anhäufen von technischer Schuld mit Hilfe von Tools (O/RM) gelöst werden können: **wunderbar**
- Sie sollten diese Entscheidung jedoch **bewusst** treffen

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

Eigenschaften eines Repositories

- Repositories arbeiten ausschließlich mit Aggregates –> je Aggregate existiert also typischerweise ein Repository
- Ein Repository ähnelt einem DomainService:
  - Es definiert einen Vertrag, der innerhalb einer technischen Schicht implementiert wird

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

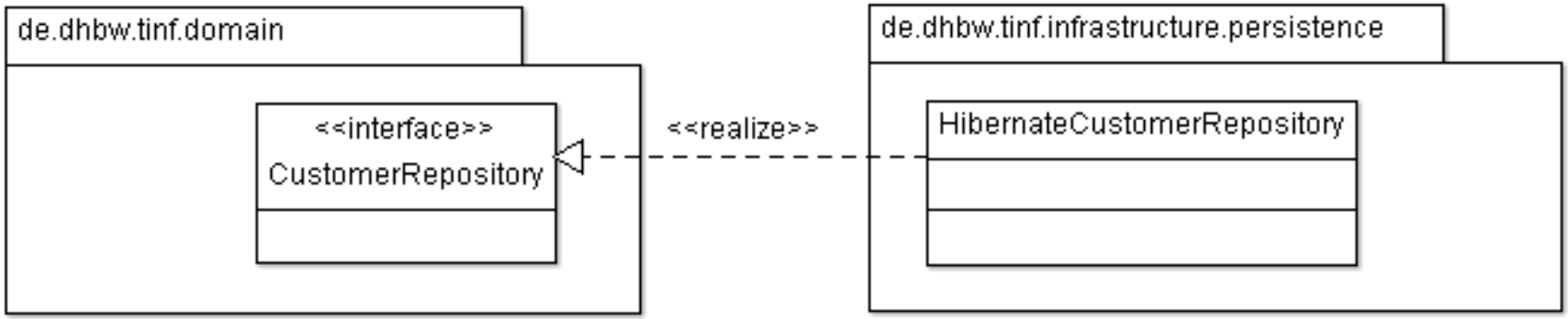
Beispiel eines Repositories

```
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;
}
```

Implementierung eines Repositories

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



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

Factories

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

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

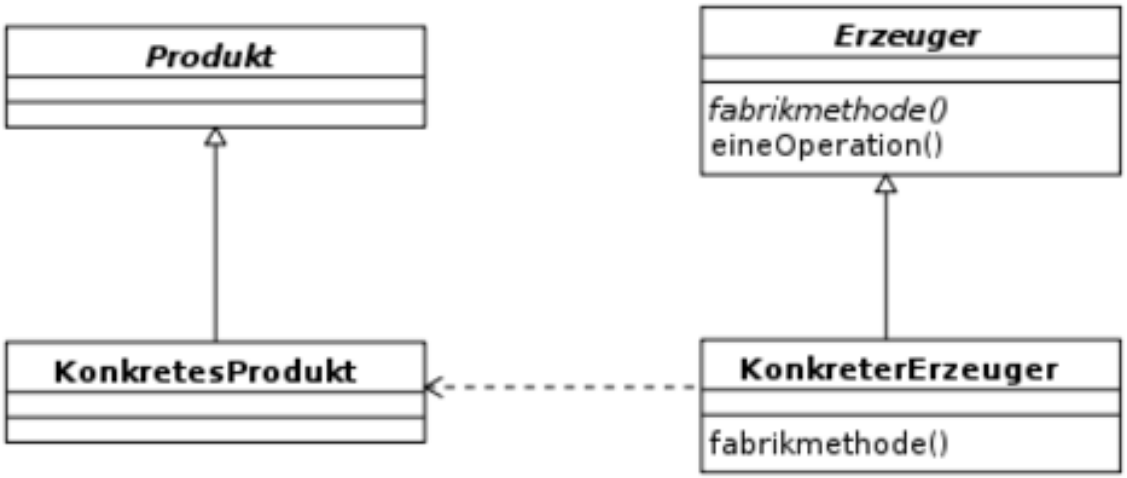
Man sollte also in einer Diskussion im Zweifel klären, wovon gesprochen wird.

Allgemeine Factory

- Allgemein ist eine Factory irgendein Objekt/ irgendeine Methode als **Konstruktor-Ersatz** (siehe unten)
- Weitere Beispiele für allgemeine Factories: Singleton, Builder Pattern

```
public class Product {

	private Product(Price price) {
		super();
		//initialise some fields...
	}

	public static Product createWithPrice(Price price) {
		//checks, validation
		return new Product(price);
	}

}
```

Factory Method

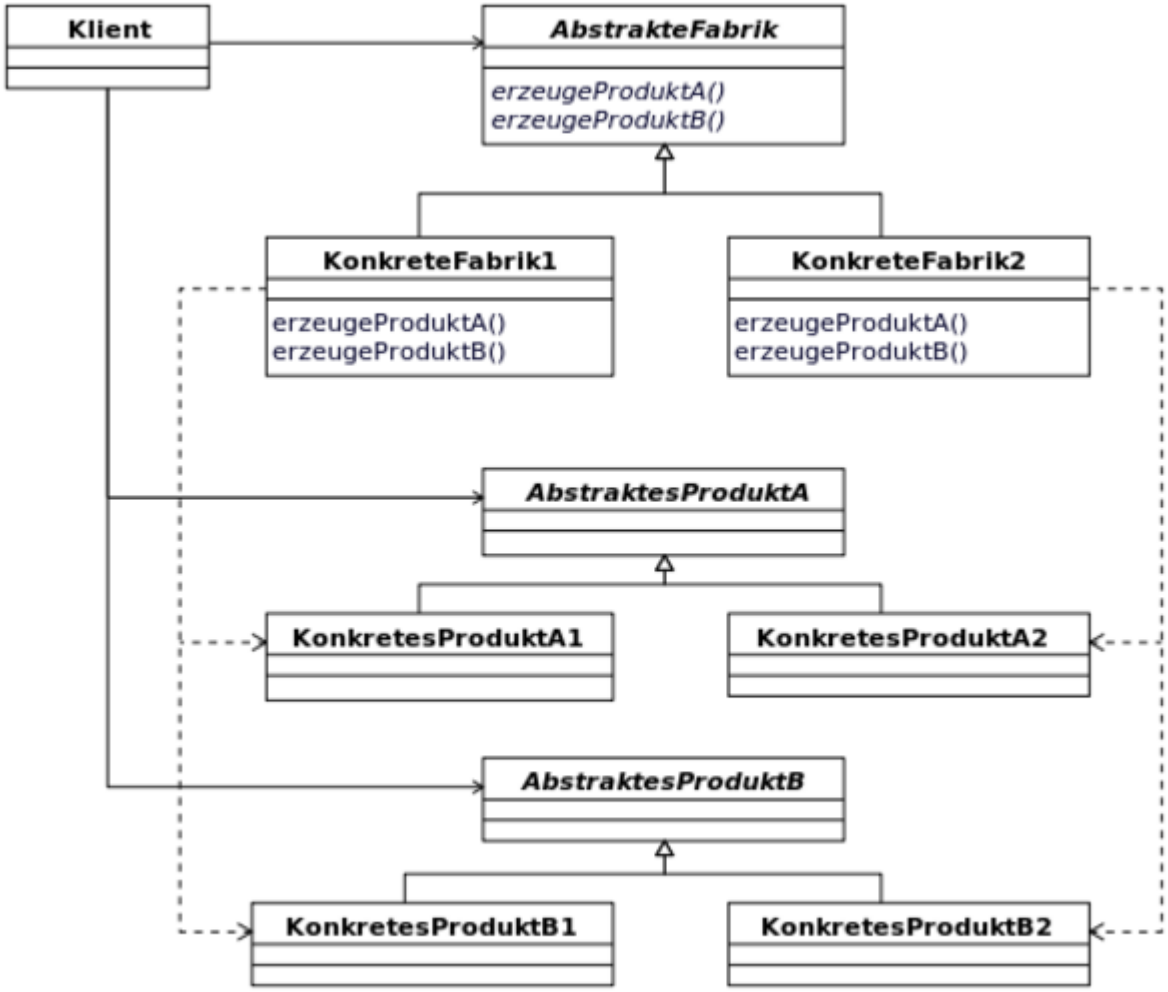


Quelle:
wikipedia

Factory Method

- Pattern zur Erzeugung eines Objektes durch Aufruf einer Methode
- Definiert eine Schnittstelle zur Erzeugung von Objekten, lässt Unterklassen aber entscheiden, welche konkreten Objekte erzeugt werden

Abstract Factory



Quelle:
wikipedia

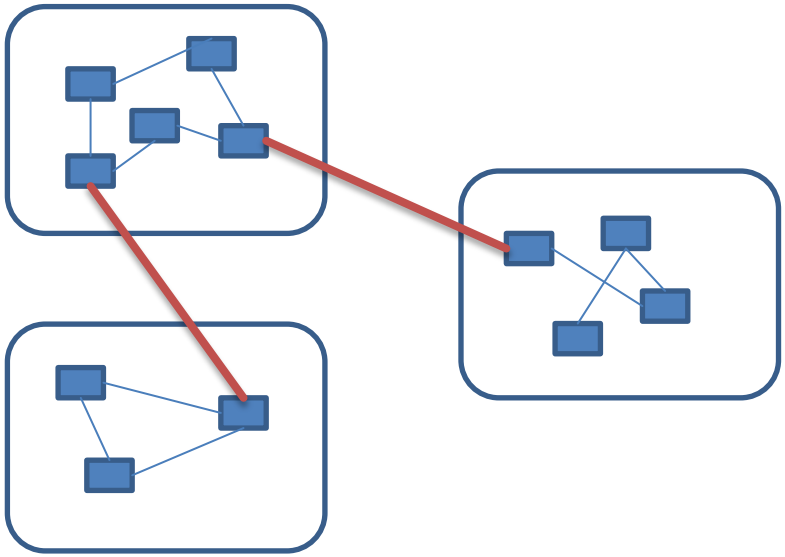
DHBW

Modules

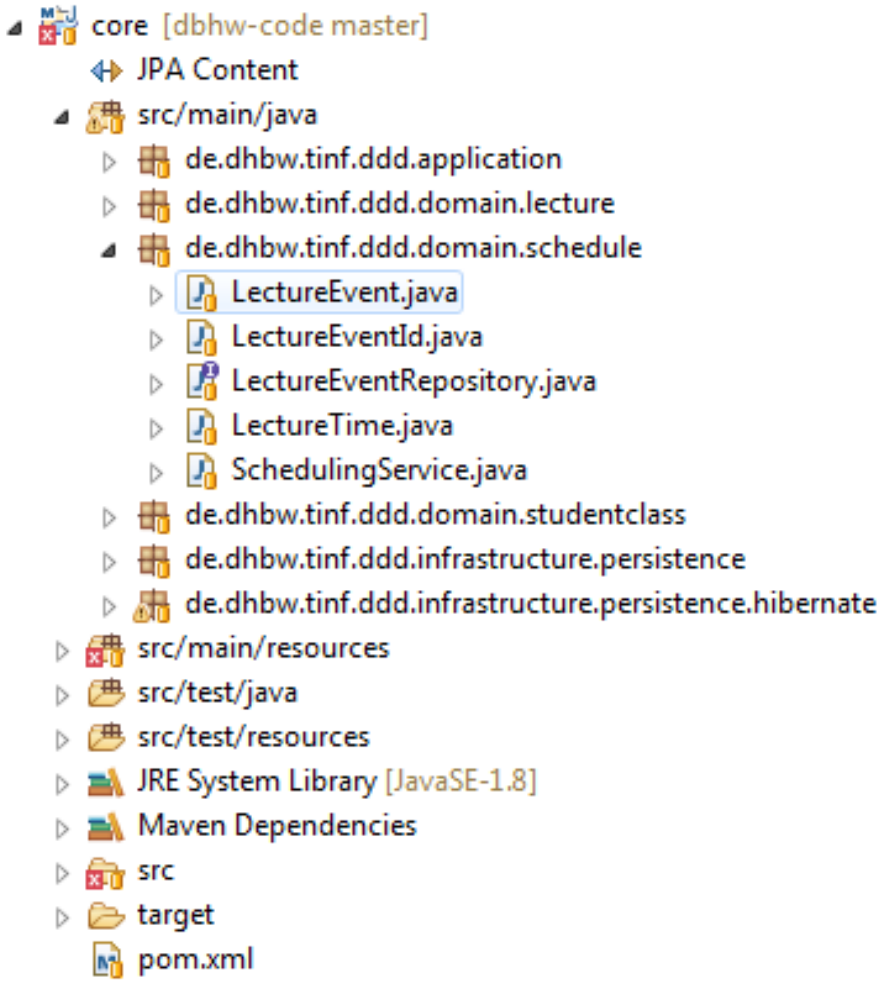
- Dienen zur Strukturierung, Unterteilung und Kapselung verwandter Komponenten
- Werden normalerweise über Namespaces, Packages o.ä. abgebildet
- Die Gruppierung sollte nicht nach technischen, sondern nach fachlichen Gesichtspunkten erfolgen
  - >Modulnamen sollen aus der UL stammen

Modules

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



Modules in Java



DHBW

2.6 DDD in der Praxis

DDD führt zu mehr Aufwand

- Der Einsatz von DDD führt normalerweise nicht sofort zu einem aussagekräftigen, reichhaltigen Modell der Problemdomäne
- Statt dessen muss das Domänenmodell in mehreren Iterationen erarbeitet und immer wieder mit Domänenexperten überprüft werden

DDD führt zu mehr Aufwand

- Insbesondere der strategische Teil von DDD erfordert ausdrücklich die Zustimmung und Mitarbeit des Kunden – er muss die Domänenexperten bereitstellen
- DDD bedeutet grundsätzlich **mehr Aufwand** in einem Projekt
- Dieser zusätzliche Aufwand muss **gerechtfertigt** werden

Für welche Projekte eignet sich DDD

DDD sollten grundsätzlich nur in Projekten von hoher strategischer Bedeutung für den Kunden (Kerngeschäftsfelder, Wettbewerbsvorteil) eingesetzt werden.

Wenn dies gegeben ist, sprechen folgende Punkte zusätzlich für DDD:

- Projekte mit einer komplexen Problemdomäne
- Projekte, die voraussichtlich stark erweitert oder verändert werden
- Projekte, die voraussichtlich eine lange Lebenszeit haben werden

Für welche Projekte eignet sich DDD nicht?

- Projekte mit einfachen oder wohlbekannten Problemdomänen
- Projekte ohne komplexe Geschäftslogik
- Projekte ohne strategische Bedeutung für den Kunden

"creating complex solutions for simple problems is wrong"

Gefahren beim Einsatz von DDD

Zu starker Fokus auf taktisches DDD:

- Die taktischen Entwurfsmuster sind wertvoll, wenn sie auch tatsächlich Fachwissen der Problemdomäne abbilden
- Die taktischen Muster sind nur Mittel zum Zweck, um ein Modell der Problemdomäne zu erstellen
- Regeln können gebrochen werden

Gefahren beim Einsatz von DDD

DDD ohne den Kunden:

- Der Kunde muss sich zum Einsatz von DDD bekennen und diesen mittragen
- Nur so ist gewährleistet, dass Kollaboration und ein gegenseitiges Lernen zwischen Entwicklern und Domänenexperten stattfindet

Wird mein Code durch DDD besser?

- Jede Software bildet eine Modell der Problemdomäne
- Ein Modell ist nicht gut oder schlecht, sondern geeignet für die Problemdomäne oder nicht
- DDD hilft bei der Entwicklung besser geeigneter Modelle

2.6 Fazit und Ausblick

Was nicht behandelt wurde

- Der Großteil des strategischen DDD
- Domain Events


push state change with
DOMAIN EVENTS
SERVICES
access with
REPOSITORIES
access with
express model with
express change with
ENTITIES
encapsulate with
act as root of
express identity with
encapsulate with
AGGREGATES
express state &
computation with
VALUE OBJECTS
MODEL-DRIVEN DESIGN
encapsulate with
encapsulate with
model gives structure to
isolate domain
expressions with
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
define model within
UBIQUITOUS LANGUAGE
cultivate rich
model with
keep model
unified by
CONTINUOUS INTEGRATION
names
enter
BOUNDED CONTEXT
SHARED KERNEL
SHARED KERNEL
CORE DOMAIN
interdependent
contexts form
overlap allied
contexts through
CUSTOMER /SUPPLIER
relate allied contexts as
work in
autonomous, clean
assess/overview
relationships with
CONTEXT MAP
minimize translation
CONFORMIST
avoid overinvesting in
GENERIC SUBDOMAINS
segregate the
conceptual messes
support multiple
clients through
OPEN HOST SERVICE
free teams
to go
translate and insulate
unilaterally with
loosely couple
contexts
through
formalize
BIG BALL OF MUD
ANTI-CORRUPTION LAYER
SEPARATE WAYS
PUBLISHED LANGUAGE

DDD entwickelt sich weiter

- Taktische Entwurfsmuster werden neu interpretiert und erweitert
- Domain Events waren beispielsweise ursprünglich nicht in DDD vorgesehen
- DDD inspiriert neue Entwicklungsansätze wie CQRS und Microservices

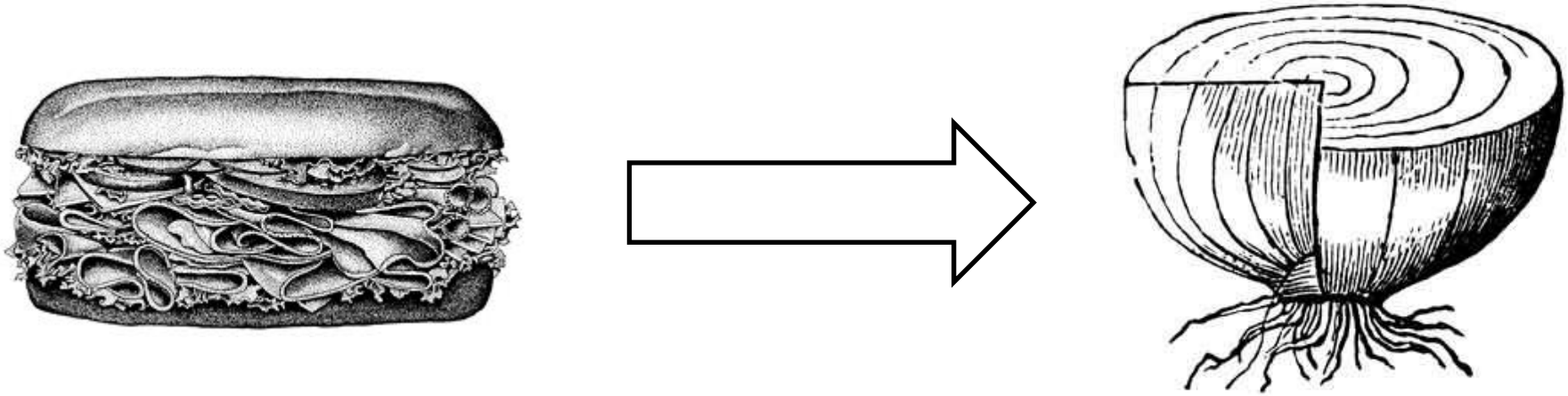
DDD: was Sie mitnehmen sollten

- Für die Verständlichkeit, Wartbarkeit und Evolvierbarkeit einer Anwendung ist es wichtig, dass Fachwissen der Problemdomäne sich im Domänenmodell wiederspiegelt
- DDD liefert Methoden und Werkzeuge, um das Fachwissen zu heben und in Code abzubilden
- Das wichtigste Element bei DDD ist die enge Zusammenarbeit mit dem Kunden
- Taktisches DDD ist eine wertvolle Erweiterung des Entwickler-Werkzeugkastens
- Taktisches DDD sollte pragmatisch, nicht dogmatisch angewendet werden

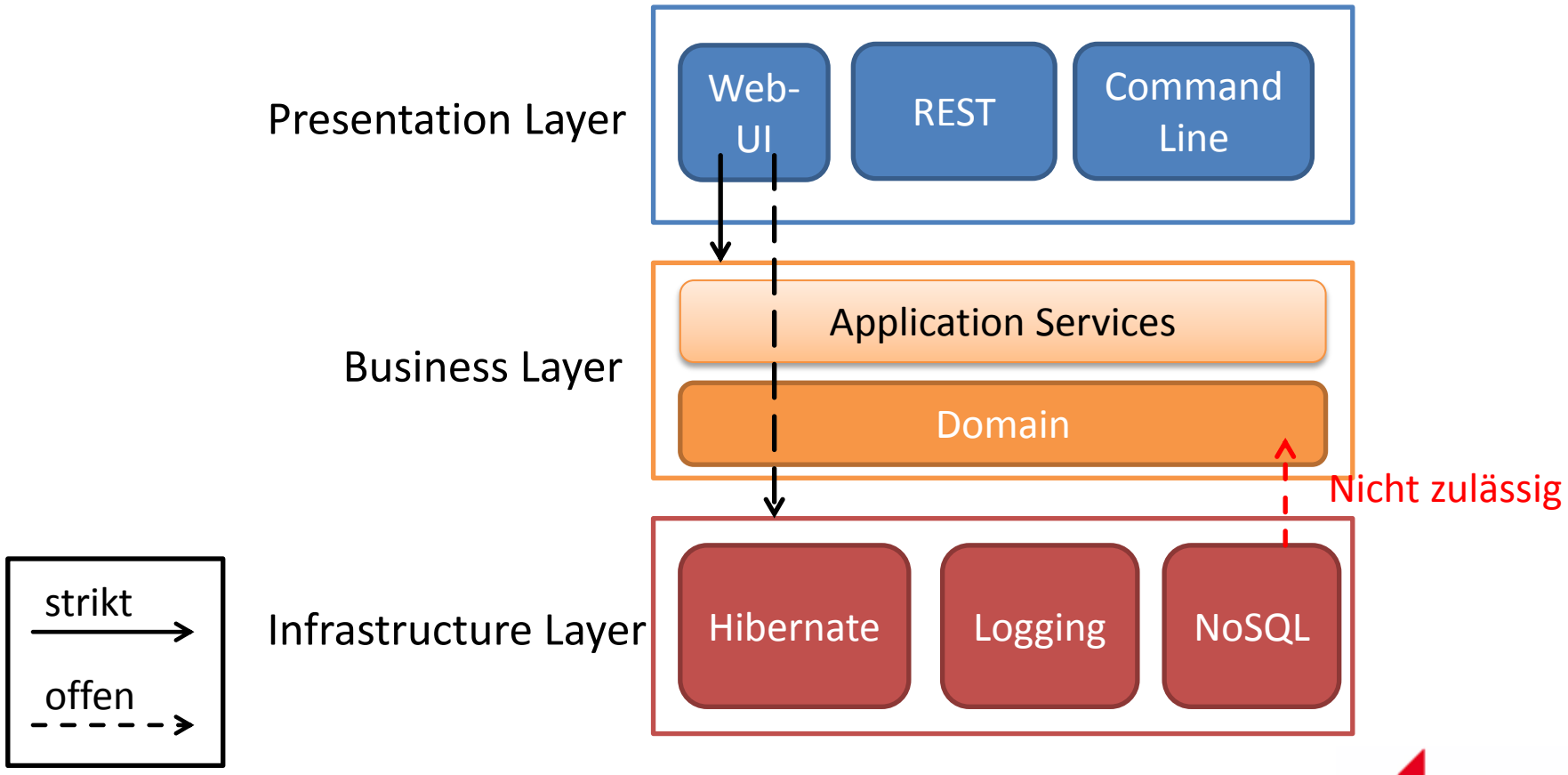
3. Onion Architecture

Alternative zur klassischen Schichtenarchitektur

3.1 Vom Sandwich zur Zwiebel: eine zeitgemäße Schichten-Architektur



Klassische Schichten-Architektur



Klassische Schichten-Architektur

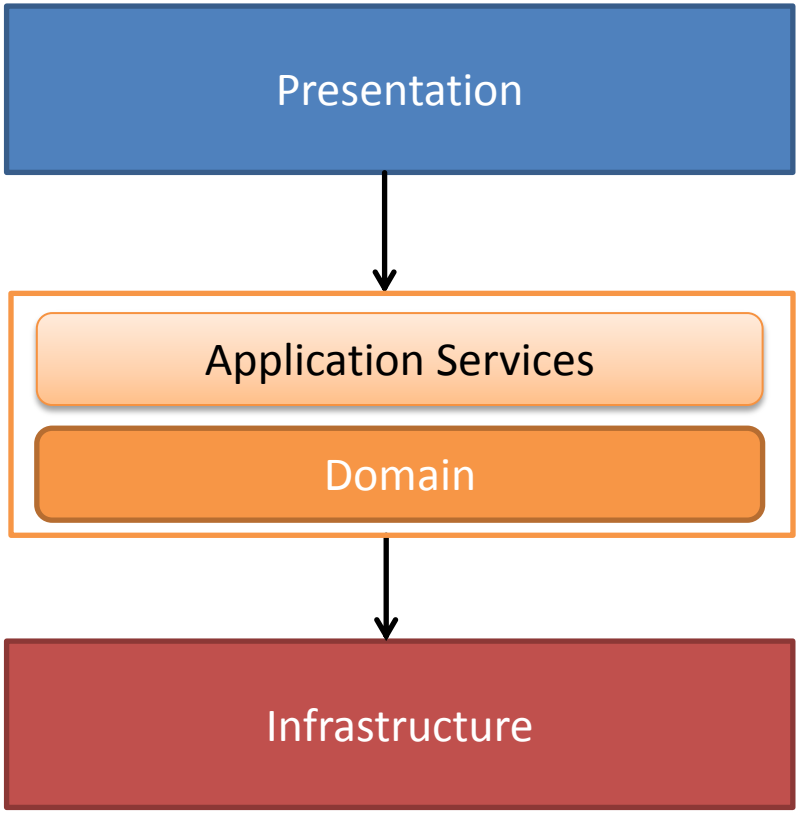
- Die Schichten-Architektur ist das „klassische“ Architekturmuster
- Eine Schicht darf nur mit den **unter ihr liegenden** Schichten kommunizieren
- Bei **strikter** Schichten-A. darf nur die **nächstniedrigere** Schicht aufgerufen werden
- Bei einer **offenen** Schichten-Architektur darf **jede beliebige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

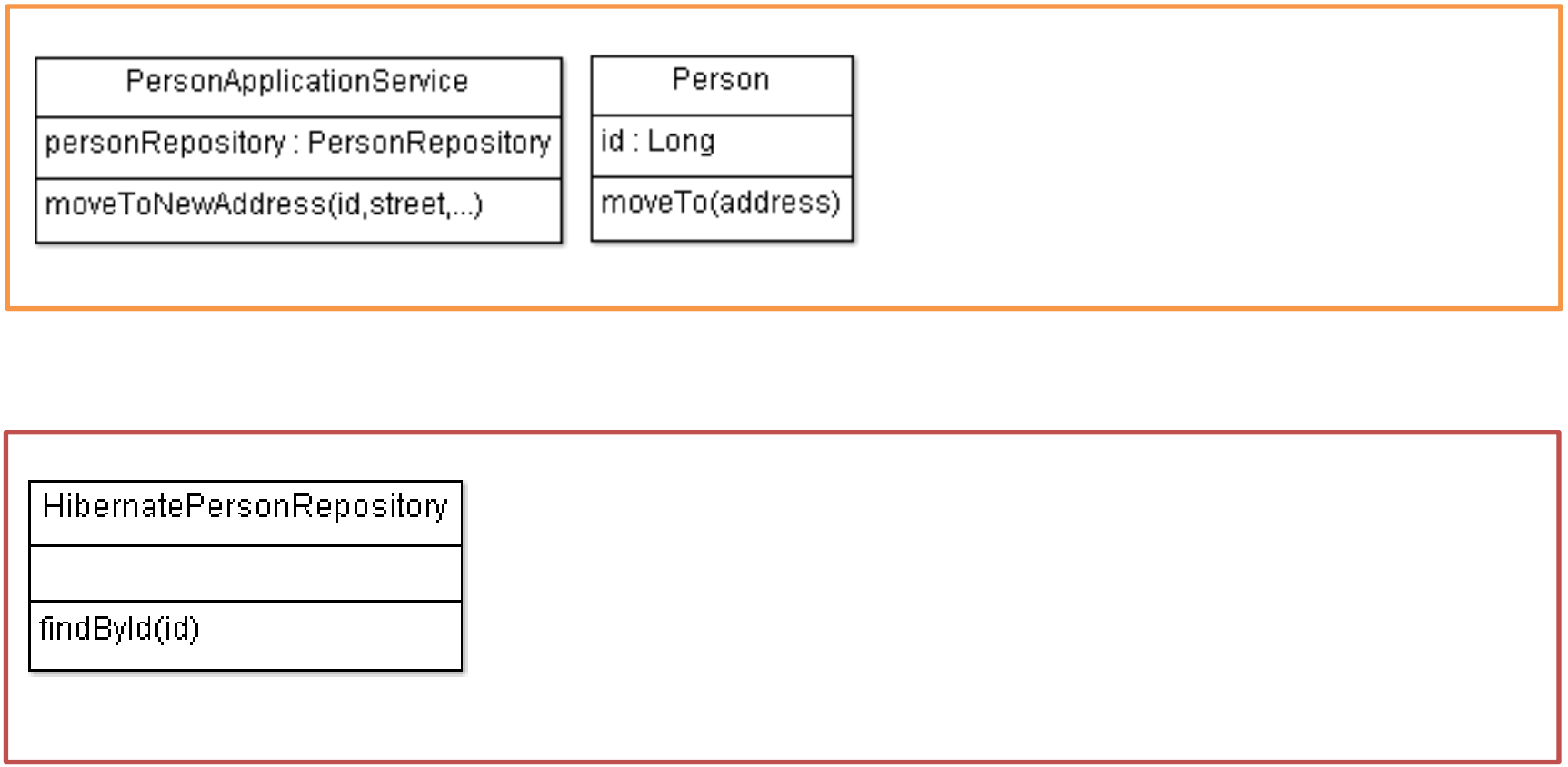
Klassische Schichten-Architektur

Frage: zu welchen Schichten gehören folgende Klassen?



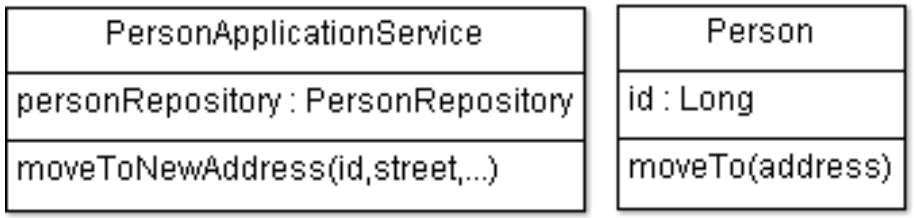
DHBW

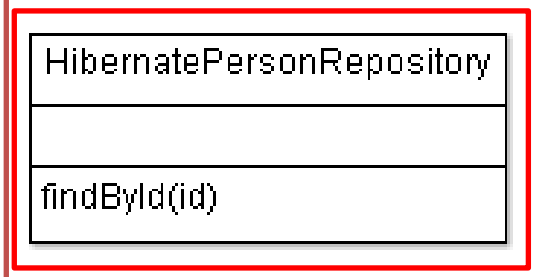
Klassische Schichten-Architektur



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

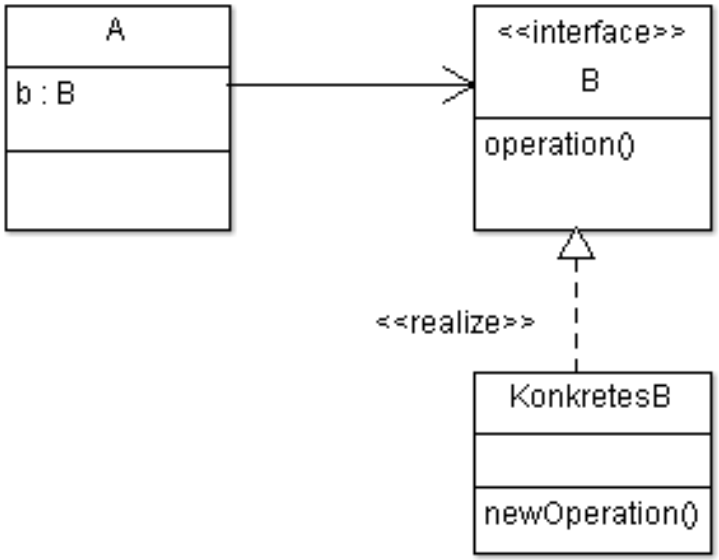
- Die Domänenschicht soll frei von technischen Details bleiben
- Die konkrete Implementierung „HibernatePersonRepository“ darf also nicht in die Domänenschicht
- Gleichzeitig gehört aber das Repository konzeptionell zu Domänenschicht, darf also eigentlich auch nicht in die Infrastruktur-Schicht
- Die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

Dependency Inversion Principle (DIP)

Definition:

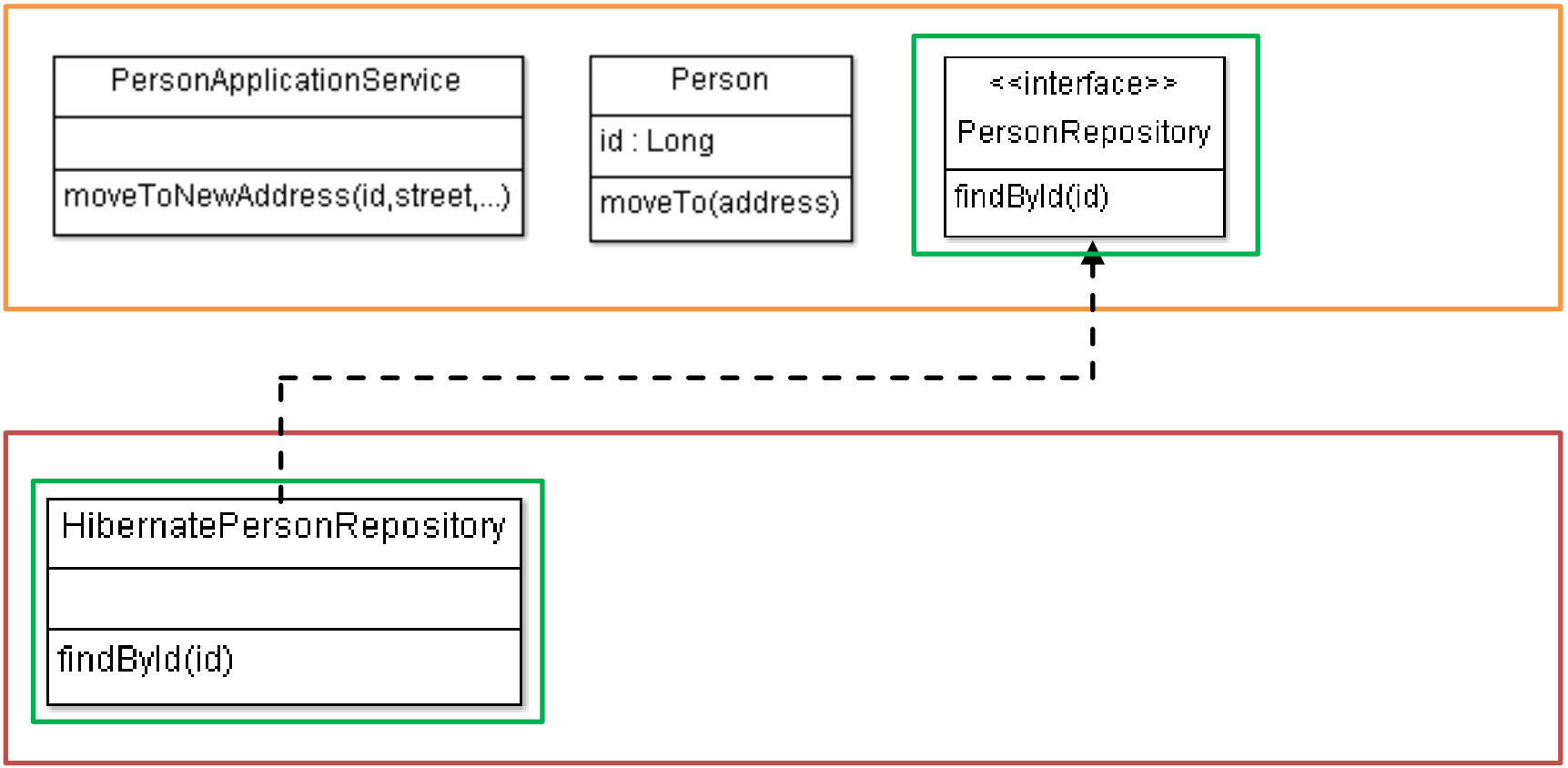
„A. *Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen.*

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



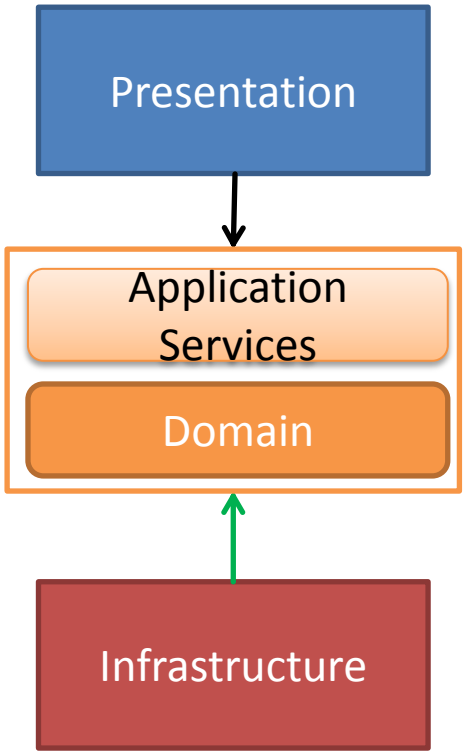
Dependency Inversion Principle (DIP)

Lösung: Definition eines Vertrages durch Dependency Inversion



Dependency Inversion Principle

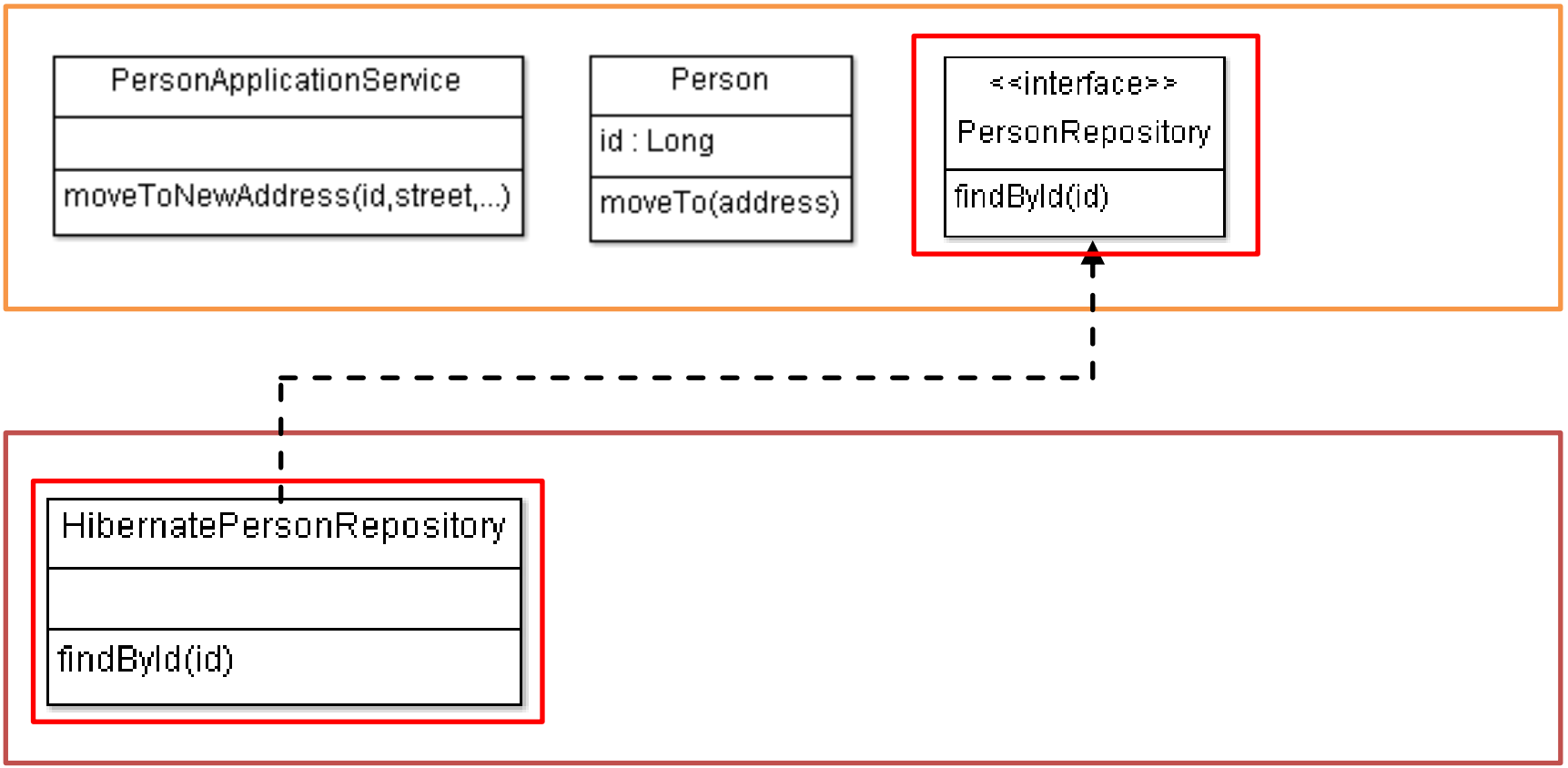
- Die Domänenschicht gibt durch ein Interface einen **Vertrag** vor, der beschreibt, welches Verhalten sie erwartet
- Die **konkrete Implementierung des Vertrages** erfolgt in der Infrastruktur-Schicht
- Die Domäne ist **damit nicht mehr abhängig von Details** (HibernatePersonRepository), sondern von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



DHBW

Problem II

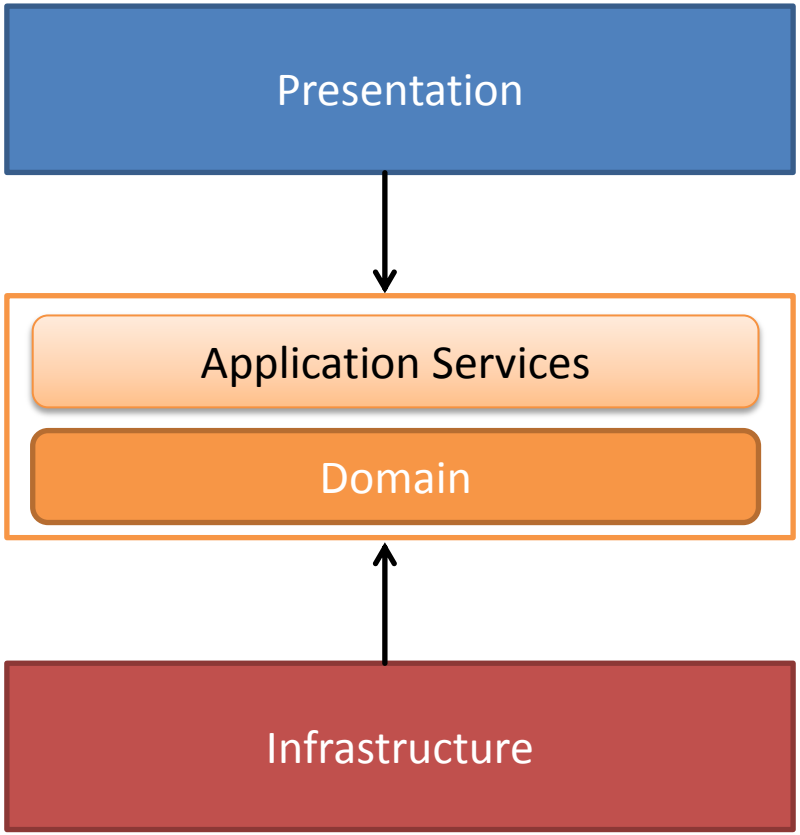
Infrastrukturschicht ist jetzt abhängig von Domänenschicht



Problem II

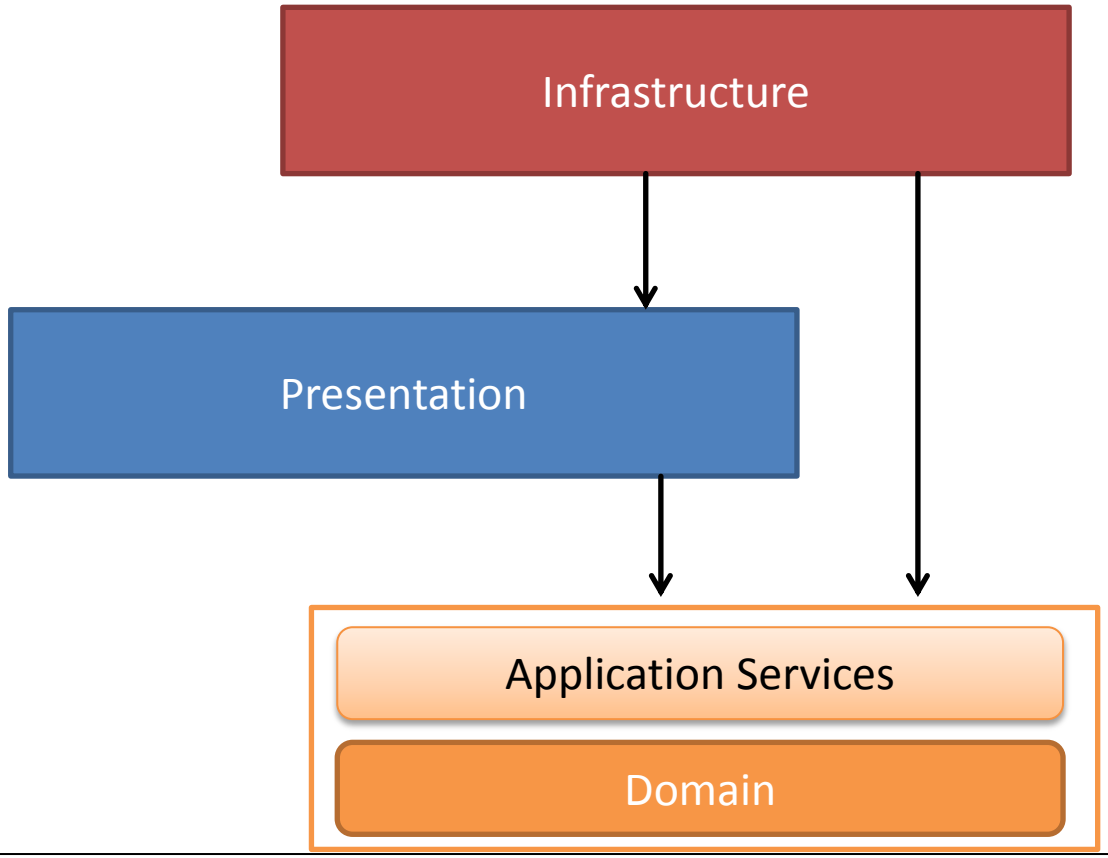
- Die Infrastrukturschicht ist abhängig vom in der Domänenschicht definierten PersonRepository
- Dies verletzt die Regeln der Schichtenarchitektur (Schicht darf nur von darunterliegenden Schichten abhängig sein)
- Die Lösung: das „Verschieben“ der Infrastruktur-Schicht

Umschichtung

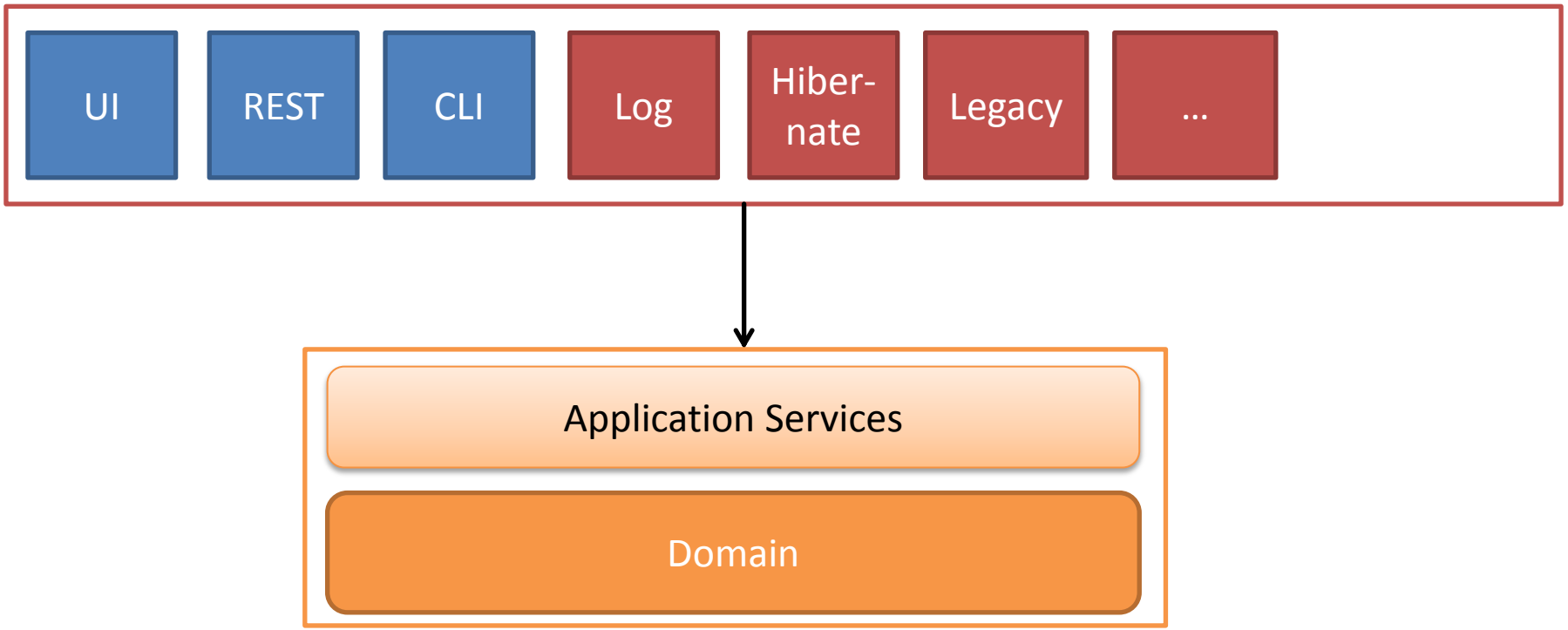


DHBW

Umschichtung



Umschichtung

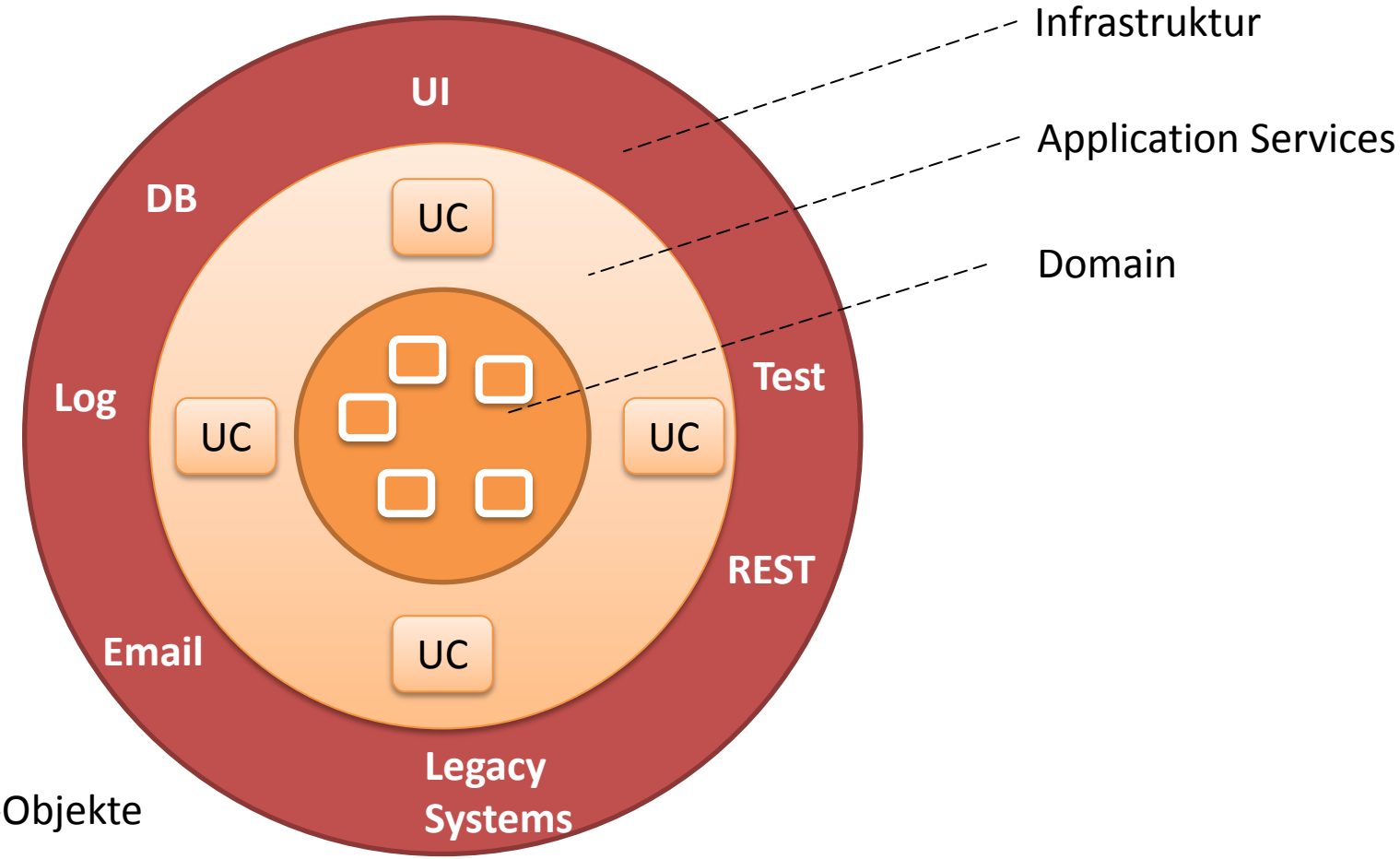


DHBW

Umschichtung

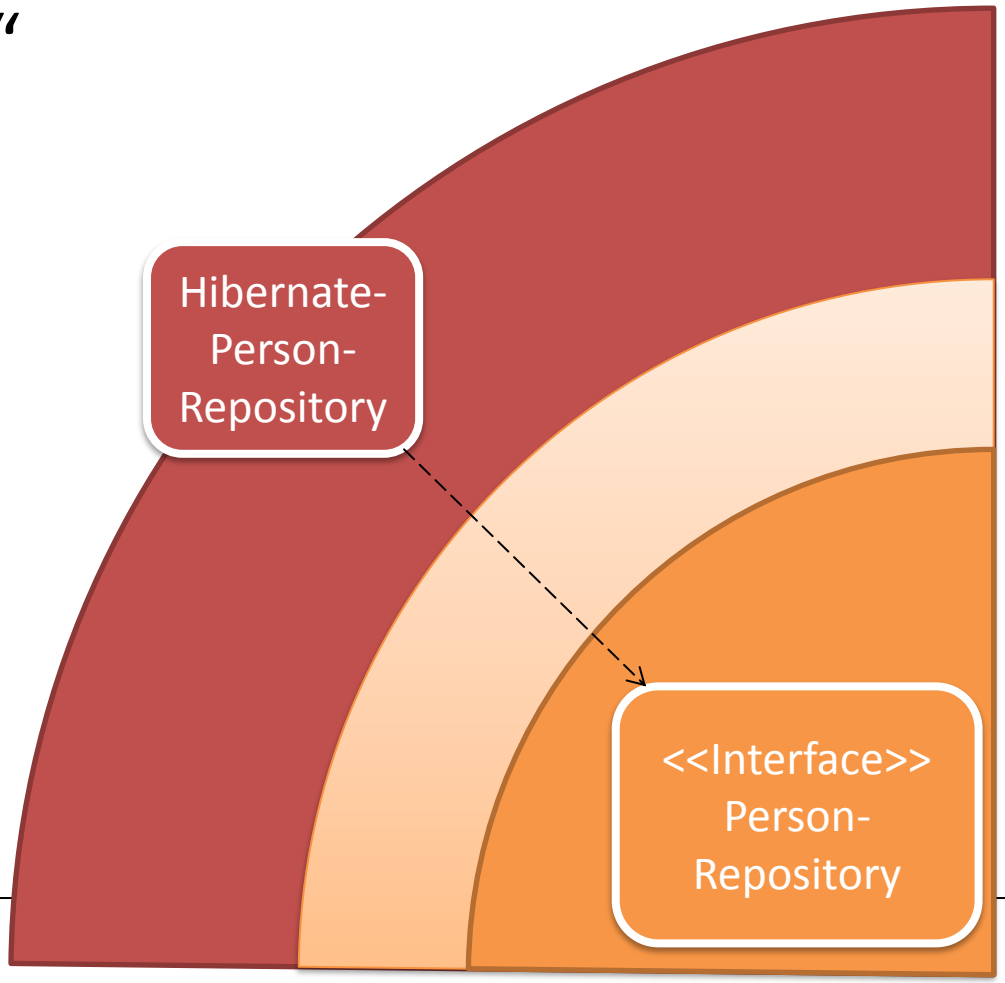
- Durch das Verschieben der Infrastruktur-Schicht darf diese konkrete Verträge für alle anderen Schichten implementieren, ohne die Regeln der Schichten-Architektur zu verletzen
- Die Präsentationsschicht kann dann auch als Teil der Infrastrukturschicht betrachtet werden
- Die Domänenschicht liegt ganz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

Domäne als isolierter Kern: Zwiebel statt Sandwich



Hauptmerkmale der „Onion Architecture“

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



Hauptmerkmale der „Onion Architecture“

- Schichten im Inneren sind 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)

- Die Domänenschicht bildet die anwendungsweiten Grundregeln und Funktionen ab
- Normalerweise ist bei der Abbildung eines bestimmten Anwendungsfalles aber mehr als ein Domänenobjekt involviert
- Beispiel: Ändern der Kundenadresse eines Auftrags benötigt Zugriff auf CustomerRepository und OrderRepository
- Die ASL implementiert diese Anwendungsfälle als sog. Application Services
- Die ASL bildet damit eine API für die Infrastrukturschicht, die bestimmt, welche Funktionen ausgeführt werden können

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 (so lange sich der konkrete Anwendungsfall nicht ändert)

Aufgaben eines Application Service

- Implementierung eines Anwendungsfalls
- Validierung, Übersetzung und Aufbereitung von Eingaben und Ausgaben
- Transaktionsverwaltung
- 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** einer Methode in **Domänenobjekte** (Beschaffung einer Entity über Repository, Konstruktion eines VO mit Hilfe einer Factory, String-Pfad zu File usw.)
- **Übersetzt** bei Bedarf das **Resultat** der Verarbeitungslogik für die Außenwelt (beispielsweise mit Data Transfer Objects)

Aufgaben eines Application Service:
Transaktionsverwaltung

- Wenn mit einer **relationalen Datenbank** gearbeitet wird, müssen Schreibvorgänge (create, update, delete) normalerweise **innerhalb einer Transaktion** durchgeführt werden
- Da ein Anwendungsfall sich oft **über mehrere Domänenobjekte** erstreckt, wäre die Transaktionsverwaltung innerhalb der Repositories zu **feingranular** – meistens soll der ganze Anwendungsfall entweder erfolgreich sein oder fehlschlagen und zurückgerollt werden

Aufgaben eines Application Service:
Transaktionsverwaltung

- Daher sollte die Transaktionsverwaltung innerhalb des jeweiligen Application Service stattfinden, nicht im Repository

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 werden – beispielsweise alle Artikel und alle Lagerbewegungen in einem bestimmten Zeitraum
- Ein Application Service kann diese Daten aggregieren und in einem speziellen Report-Objekt (DTO) zurückliefern

Aufgaben eines Application Service:
Reporting

- Das Generieren komplexer Berichte über die Domänenschicht ist oft teuer und ineffizient; aus der DB müssten alle Daten geladen und in Objekte gewandelt werden – nur damit Zahlen aufsummiert werden können
- In diesem Fall kann das Domänenmodell einen Vertrag für einen Infrastrukturservice vorgeben, der direkten Zugriff auf die Datenquelle erlaubt (beispielsweise vermöge nativer Queries) und die Daten dann wiederum in einem Report-Objekt ausliefern

Aufgaben eines Application Service:
Security

- Meist existieren in einer Applikation Anwendungsfälle, die nur Benutzern mit bestimmten Rechten zugänglich sind
- Da die bereitgestellten Anwendungsfälle durch Application Services implementiert werden, macht es Sinn, dort auch Authentifizierung und Autorisierung abzubilden
- Die 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

Wie stark soll die Domäne von der Außenwelt abgeschottet sein?

- Isolierung des Domänenmodells bietet Schutz vor ungewollten Veränderungen
- Isolierung erhöht die Komplexität und den Aufwand durch zusätzliche Indirektion / Mappings
- ->Aufwand muss gerechtfertigt sein

Zwei Hilfestellungen für die Isolierung der Domänenschicht

- Data Transfer Objects (DTO)
- Commands

Exkurs: Data Transfer Object (DTO)

- Einfaches POJO mit dem Zweck, Daten zu übertragen
- Enthält keine Geschäftslogik, sondern höchstens Get/Set-Methoden
- Attribute sind primitive Datentypen oder wiederum DTO

Exkurs: Command Objects

- https://sourcemaking.com/design_patterns/command

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

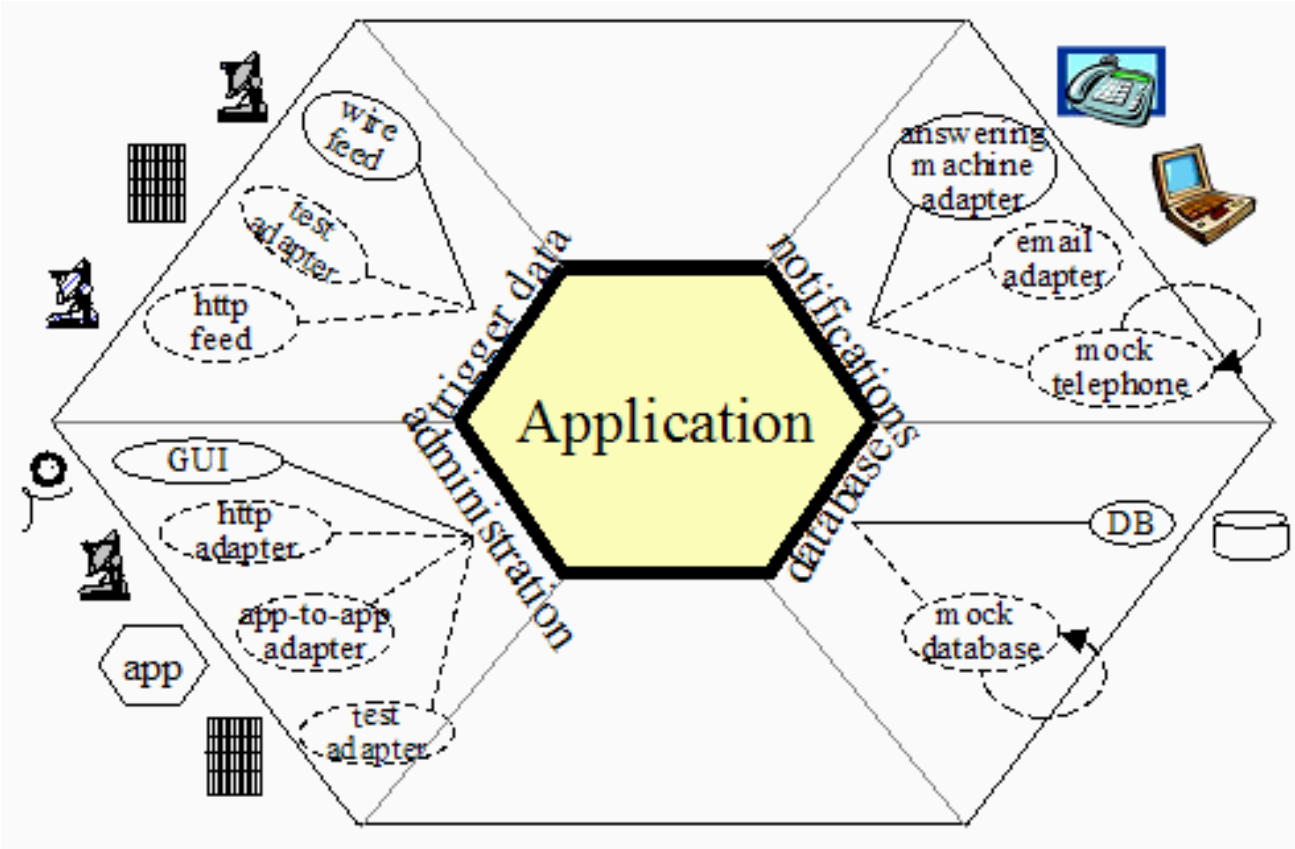
Varianten der Onion Architecture

Aktuell existieren mehrere Varianten der „Onion Architecture“, unter anderem:

- Hexagonale Architektur / Ports-and-Adapters
- Clean Architecture

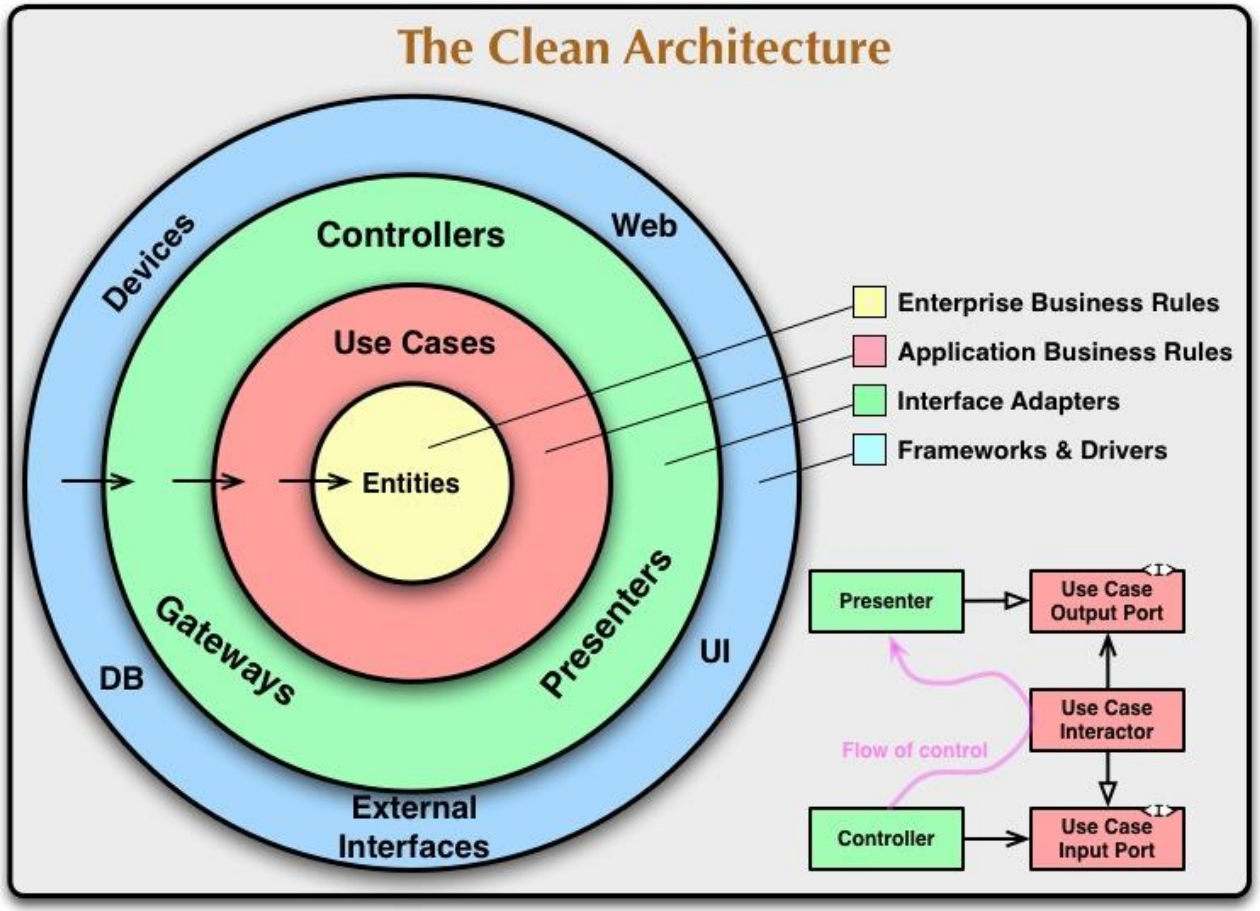
Tatsächlich wurde die „Onion Architecture“ für diese Vorlesung nur exemplarisch gewählt, weil sie (meiner Meinung nach) am allgemeinsten gehalten ist (und weil ich ein tolles Schwarz-Weiß-Bild einer Zwiebel gefunden habe). Zwar unterscheiden sich die genannten Architekturstile im Detail, das Grundprinzip (Abhängigkeiten zeigen von Außen nach Innen) ist aber immer dasselbe.

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



By Alistair Cockburn

Clean Architecture



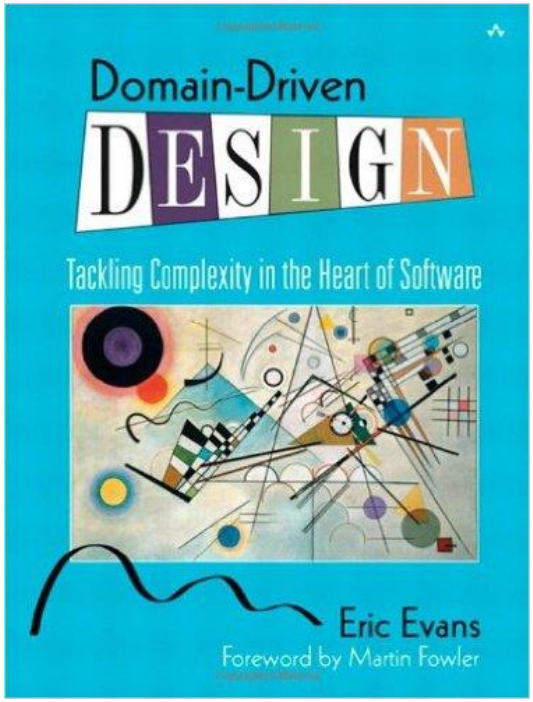
By Robert Martin

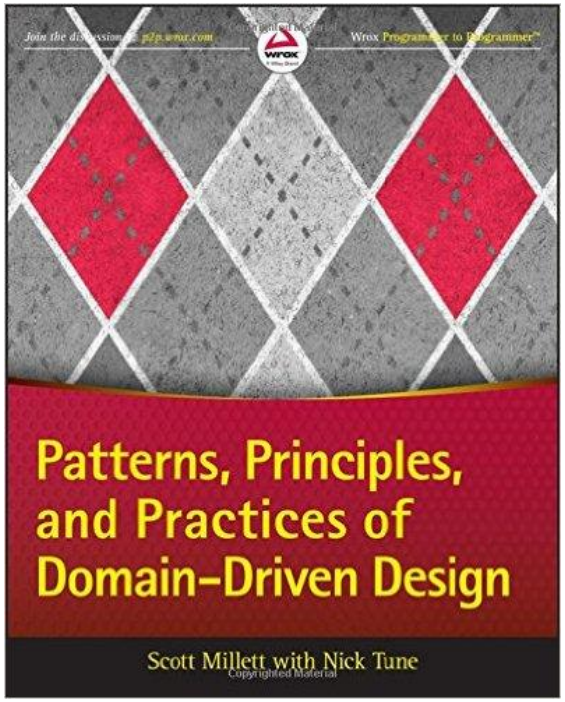
3.3 Fazit und Ausblick

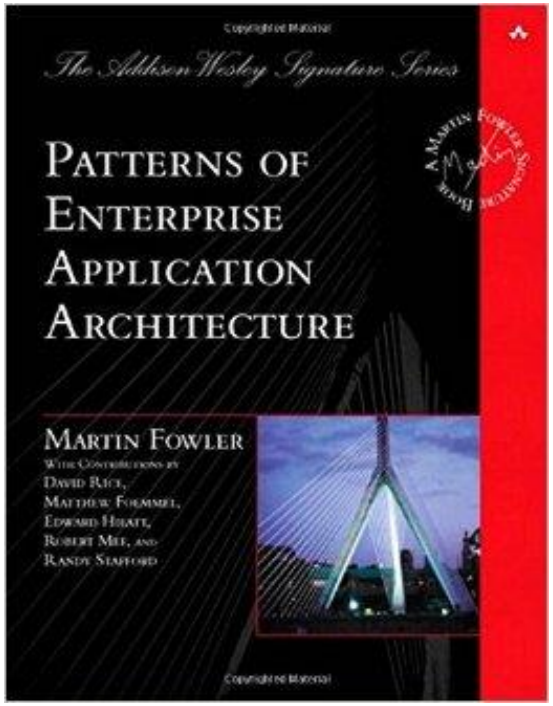
Kritik

- 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

- http://dddcommunity.org/learning-ddd/what_is_ddd/
- Bass, L.; Clements, P. & Kazman, R. (2013), *Software Architecture in Practice* , Addison-Wesley Longman Publishing Co., Inc. , Boston
- 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)
- 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
- 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
- Fowler, M. Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., Inc., 2002
- Steve Freeman and Nat Pryce. 2009. *Growing Object-Oriented Software, Guided by Tests* (1st ed.). Addison-Wesley Professional.
- Vaughn Vernon. Implementing Domain-Driven Design. Addison-Wesley Professional, 1. edition, February 2013.
- Millett, S. Patterns, Principles and Practices of Domain-Driven Design, *Wiley,* **2015**