

Domain Driven Design

Eine Sprache, sie alle zu finden

Was ist Domain Driven Design?

- Eine Herangehensweise an die Modellierung von Software
- Das Design der Software wird maßgeblich von der Fachlichkeit (der Anwendungsdomäne) bestimmt
- Das Domänenmodell ist die Grundlage für den Entwurf und die Umsetzung der Software

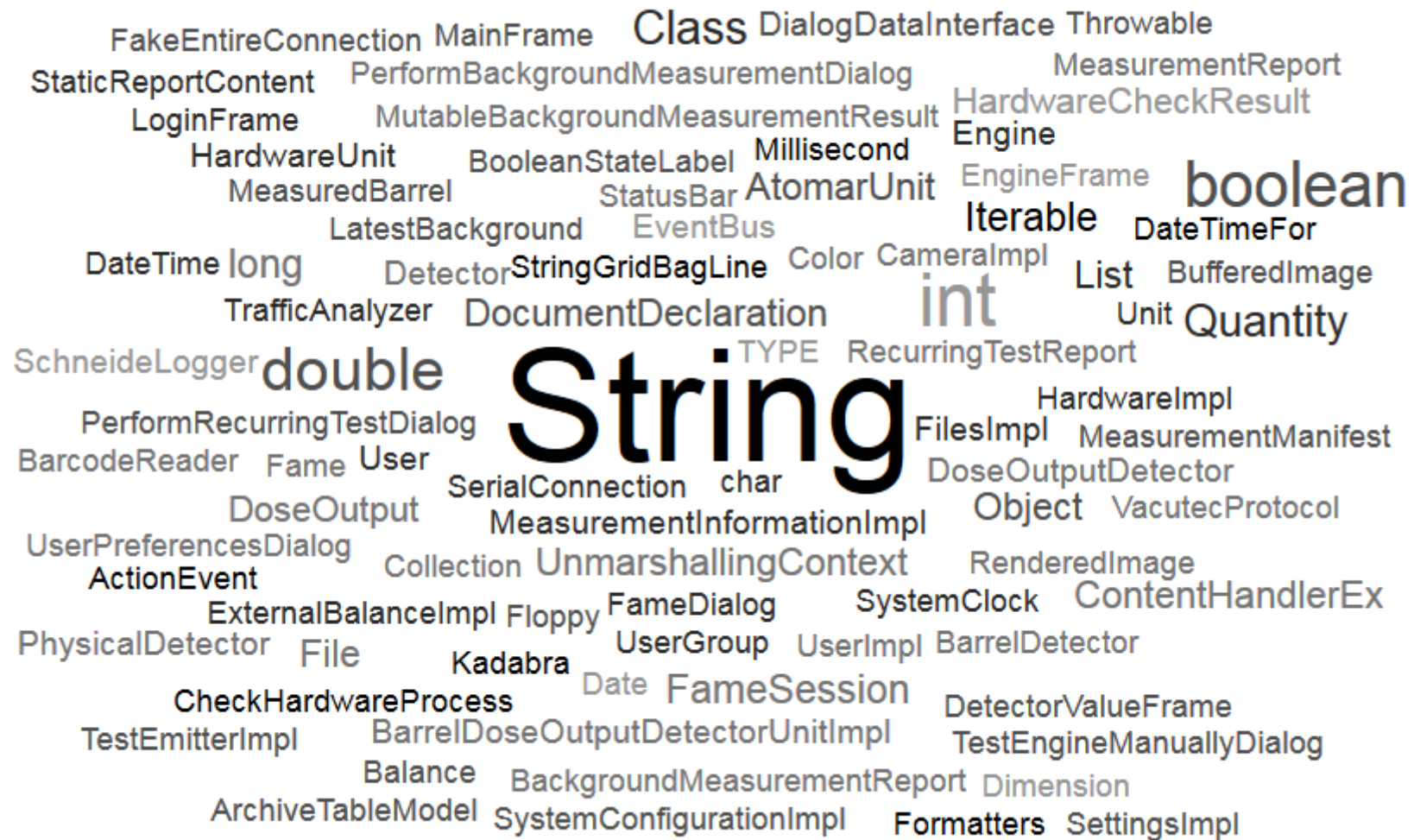
Grundannahmen für DDD

- Sinn der Software ist die Unterstützung bei einer Aufgabenstellung in der jeweiligen Anwendungsdomäne
- Je kleiner der „Übersetzungsaufwand“ von der Domäne zum Code ist, desto leichter fällt die Umsetzung der Aufgabenstellung
- Ein Programm in der Domäne des Bibliothekswesen muss nicht wie eine Bibliothek strukturiert sein, sollte aber deren Konzepte enthalten

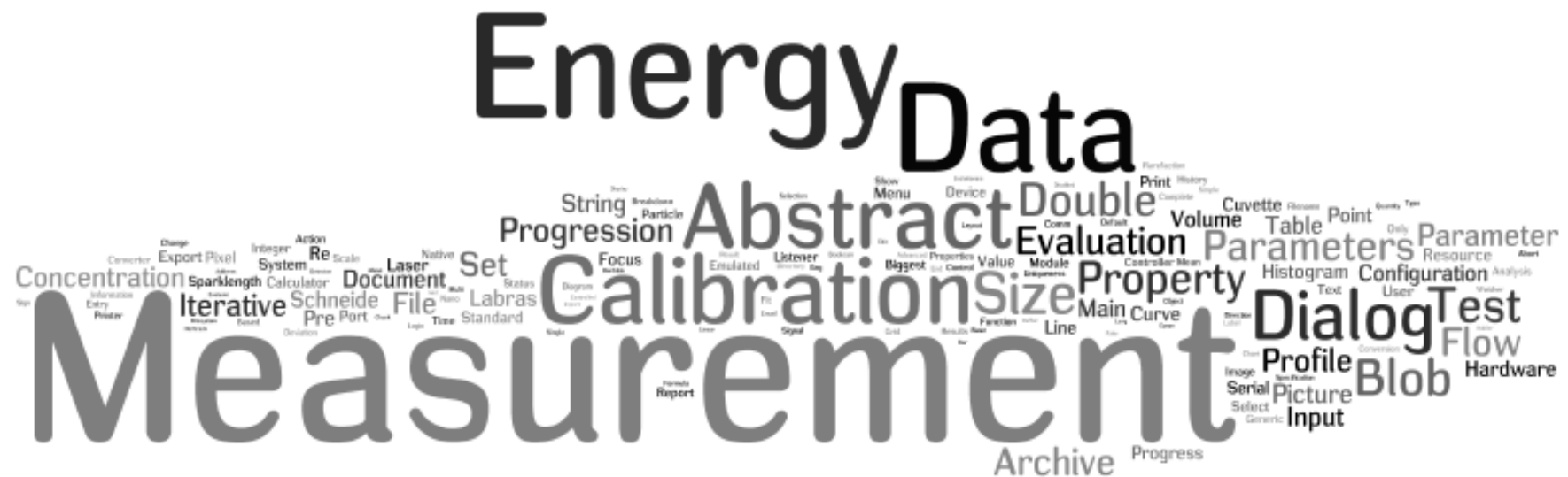
Begriffsextraktion aus Anwendung

- Die deklarierten Typen aus einem Java-Programm als Wortwolke (Word Cloud)
- Wenn der Code domänenannah geschrieben ist, werden die Hauptkonzepte der Domäne im Typsystem erkennbar
 - Manchmal überraschend: Die Hauptkonzepte sind nicht immer die häufigsten Konzepte

Beispiel eines Projekts



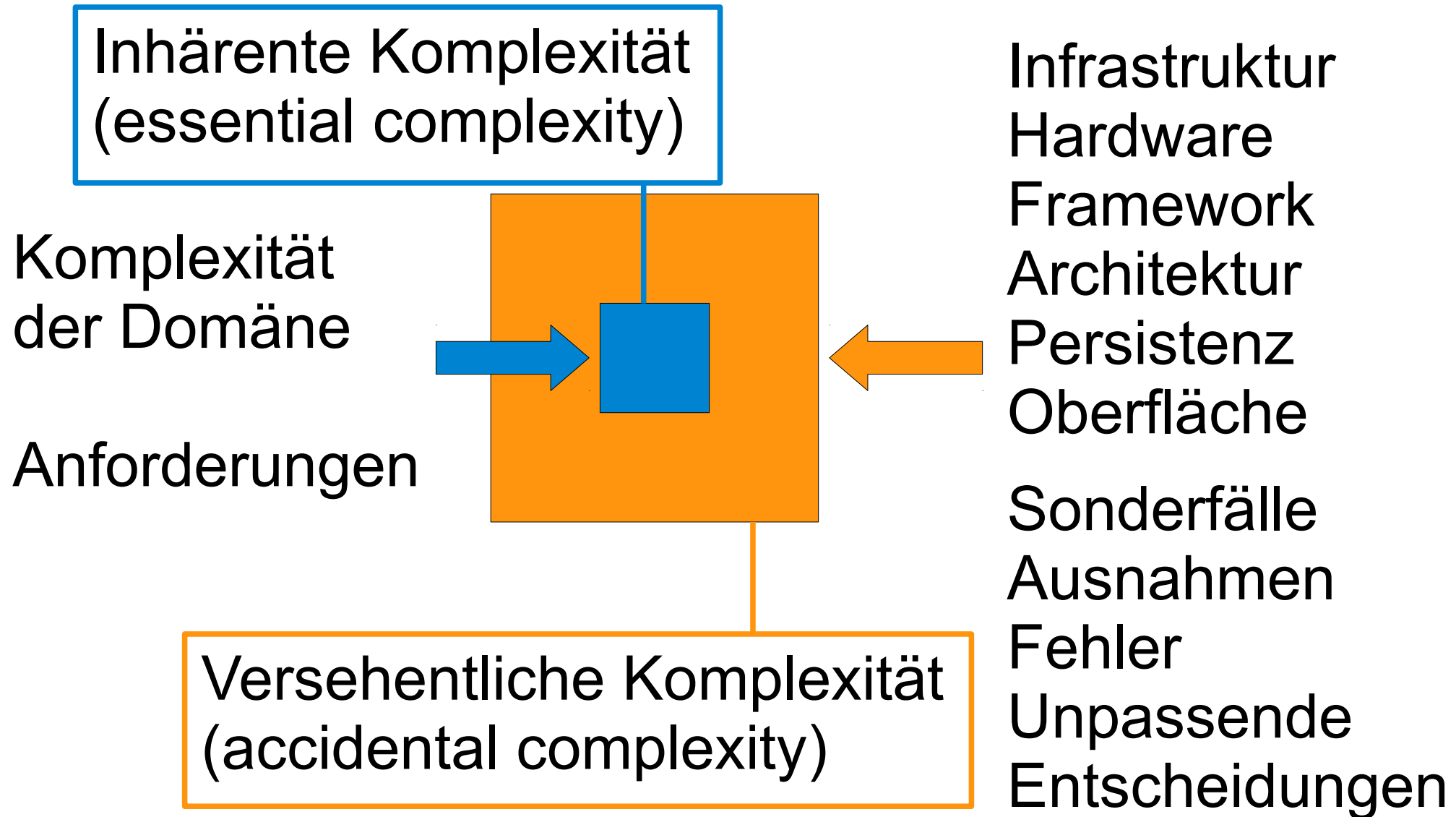
Beispiel eines anderen Projekts



Was ist das Problem?

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

Systemkomplexität



Systemkomplexität

- Die Komplexität der Domäne und der Anforderungen ist nicht wegdiskutierbar, sondern fest gegeben (inhärente Komplexität).
 - Damit sollte sich ein Entwickler eigentlich beschäftigen
- Die zusätzliche Komplexität der technischen Umsetzung (Pure Fabrication) kann nicht komplett eliminiert werden, sie ist ein notwendiges Übel
 - Ziel: Verhindern, dass die Komplexität der Domäne negativ beeinflusst wird

Was passiert sonst?

- Es entsteht ein „Big Ball of Mud“
- „Code that does something useful, but without explaining how“ - Eric Evans
- Eigenschaften:
 - Sourcecode gibt Intention nicht preis (was/warum)
 - Technische Belange belasten die Fachlogik
 - Fachlogik über die gesamte Anwendung verteilt



Big Ball of Mud

- Fester Begriff in der Softwarearchitekten-Szene
- Angstgegner (Berufsehre steht auf dem Spiel!)
- Muss aber nicht schlecht sein
 - „Lieber kontrolliertes Chaos an einer Stelle als überall unkontrollierte Ordnungswut“ - Lindner
 - Small Balls of Mud ist eine moderne Architektur (Microservices)
- Wichtig ist, dass der Ball in seinem „Gebiet“ bleibt und sich nicht ausbreitet
 - Kontrolliertes Risiko

Wie hilft Domain Driven Design?

- Durch Reduktion des Übersetzungsaufwands
 - Gleiche Begriffe in Domäne und Sourcecode
 - Klare Modellierung der Fachlichkeit
 - Möglichst identische Konzepte
- Durch Beschreiben nützlicher Methoden und Muster
 - Strategisches DDD: Analyse, Dokumentation und Abgrenzung der Domäne
 - Taktisches DDD: Umsetzung der Erkenntnisse in Sourcecode

Ubiquitous Language

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

Verständnisprobleme

- Domänenexperten verstehen die Sprache der Entwickler meistens nur sehr begrenzt
 - Wenn ein Entwickler in seiner Sprache über das Domänenmodell spricht, hängt er den Experten ab
- Entwickler verstehen die Sprache der Domänenexperten meistens nur sehr begrenzt
 - Wenn ein Domänenexperte in seiner Sprache über die Domäne spricht, verliert er den Entwickler schnell, weil dieser immer „übersetzen“ will/muss
- Keine der beiden Sprachen eignet sich alleine für das Projekt

Verständnisprobleme im Code

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

Beispiel für die Kluft



UserOffice soll Proposal-PDFs verschieben können, so dass sie neu generiert werden



Edit



Comment

Assign

More ▾

Reopen Issue

Details

Type:	New Feature	Status:	Resolved (View Workflow)
Priority:	Major	Resolution:	Fixed
Affects Version/s:	None	Fix Version/s:	
Component/s:	Administration , User Office		
Labels:			

Description

Damit die manuelle Datenbankänderung "Co-Prosherschaft von Hand akzeptieren" überhaupt vom UserOffice sinnvoll bearbeitet werden kann, braucht das UserOffice die möglichkeit, Proposal-PDFs aus dem Weg zu räumen (verschieben), so dass sie danach bei Request neu generiert werden.

Wenn es Probleme bei PDF-Generierung gibt, hat das UO so auch die Möglichkeit, ohne unser Zutun eine Neugenerierung anzustoßen (evtl. nach Korrektur von Datenbankeinträgen).

Diese Funktion soll im Administrationsbereich zu Verfügung gestellt werden.

„Proposals neu generieren können“

👤 (Inactive) added a comment - 26.06.2015 14:15

Die Funktionalität ist implementiert, siehe `ProposalPdfRemovalController`.

[illegible]

- Konzept des Kunden: Proposal neu generieren
- Technisch: Proposal löschen, wird automatisch neu generiert
 - Muss aber in der Sprache des Kunden bleiben!

Beispiel für Mehrdeutigkeit

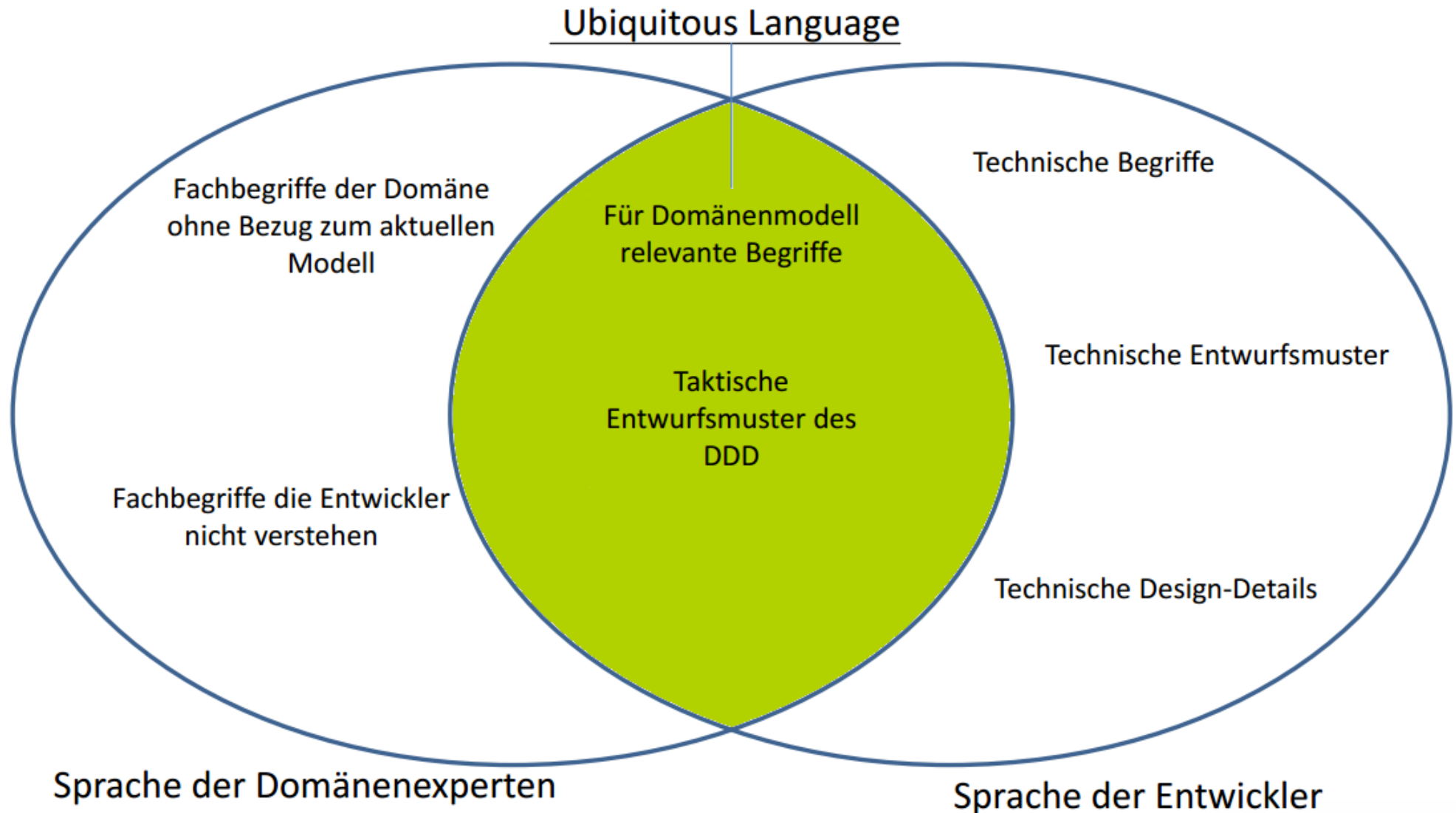
Eine Bibliothek bietet im Internet einen Service mit Verkauf von Nachdrucken alter Zeitschriften an. Dann kann der Begriff „Artikel“ schnell mehrere Bedeutungen haben:

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

Daraus ergeben sich uneindeutige Anforderungen:

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

Die gemeinsame Projektsprache



Gemeinsame Projektsprache finden

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

Artefakte mit Projektsprache

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

Grenzen der Projektsprache

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

Sichtbarkeitsstufen der Umsetzung

- Stufe 0: Inline
 - Stufe 0+: Inline mit Kommentar
- Stufe 1: Eigene Methode
- Stufe 2: Eigene Klasse
 - Stufe 2+: Eigener Typ im Domänenmodell
- Stufe 3: Eigenes Aggregat
- Stufe 4: Eigenes Package/Modul
- Stufe 5: Eigene Anwendung (Service)

Beispiel: Die Anforderung (AN-17)

- In einem Webshop können Artikel in den Warenkorb gelegt und dann gekauft werden.
- Bei der Vorschau des Warenkorbs und beim Kaufvorgang müssen auf die Nettopreise der Artikel noch die passenden Steuern, vorrangig die Mehrwertsteuer, aufgerechnet werden.
- Der so entstandene Bruttopreis soll neben dem Nettopreis angezeigt werden.

Sourcecode bisher

```
public class ShowShoppingCart {  
    public ShoppingCartRenderModel render(Iterable<Product> inCart) {  
        final ShoppingCartRenderModel result = new ShoppingCartRenderModel();  
        for (Product each : inCart) {  
            result.addProductLine(  
                each.description(),  
                each.nettoPrice());  
        }  
        return result;  
    }  
}
```

Stufe 0: Inline

```
public class ShowShoppingCart {  
    public ShoppingCartRenderModel render(Iterable<Product> inCart) {  
        final ShoppingCartRenderModel result = new ShoppingCartRenderModel();  
        for (Product each : inCart) {  
            final Euro bruttoPrice = each.nettoPrice().multiplyWith(1.19D);  
            result.addProductLine(  
                each.description(),  
                each.nettoPrice(),  
                bruttoPrice);  
        }  
        return result;  
    }  
}
```

Stufe 0+: Inline mit Kommentar

```
public class ShowShoppingCart {  
    public ShoppingCartRenderModel render(Iterable<Product> inCart) {  
        final ShoppingCartRenderModel result = new ShoppingCartRenderModel();  
        for (Product each : inCart) {  
            // AN-17: calculating the brutto price from the netto price  
            final Euro bruttoPrice = each.nettoPrice().multiplyWith(1.19D);  
            result.addProductLine(  
                each.description(),  
                each.nettoPrice(),  
                bruttoPrice);  
        }  
        return result;  
    }  
}
```

Stufe 0++: Inline mit Kommentaren

```
public class ShowShoppingCart {  
    public ShoppingCartRenderModel render(Iterable<Product> inCart) {  
        final ShoppingCartRenderModel result = new ShoppingCartRenderModel();  
        for (Product each : inCart) {  
            // AN-17: calculating the brutto price from the netto price  
            // TODO: take different tax factors into account  
            final Euro bruttoPrice = each.nettoPrice().multiplyWith(1.19D);  
            result.addProductLine(  
                each.description(),  
                each.nettoPrice(),  
                bruttoPrice);  
        }  
        return result;  
    }  
}
```

Stufe 1: Eigene Methode

```
public class ShowShoppingCart {  
    public ShoppingCartRenderModel render(Iterable<Product> inCart) {  
        final ShoppingCartRenderModel result = new ShoppingCartRenderModel();  
        for (Product each : inCart) {  
            result.addProductLine(  
                each.description(),  
                each.nettoPrice(),  
                bruttoPriceFor(each));  
        }  
        return result;  
    }  
  
    /**  
     * AN-17: Calculates the brutto price for the given product.  
     */  
    private Euro bruttoPriceFor(Product product) {  
        final BigDecimal taxFactor = <gets the right tax factor from somewhere>  
        return product.nettoPrice().multiplyWith(taxFactor);  
    }  
}
```

Stufe 2: Eigene Klasse

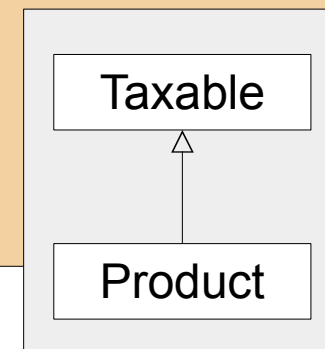
```
public class ShowShoppingCart {  
    public ShoppingCartRenderModel render(Iterable<Product> inCart) {  
        final ShoppingCartRenderModel result = new ShoppingCartRenderModel();  
        for (Product each : inCart) {  
            result.addProductLine(  
                each.description(),  
                each.nettoPrice(),  
                CalculateTax.forProduct(each));  
        }  
        return result;  
    }  
}
```

```
/**  
 * AN-17: Calculates the value added tax (VAT) for the given product.  
 */  
public class CalculateTax {  
    public static Euro forProduct(Product product) {  
        [...]  
    }  
}
```

Stufe 2+: Eigener Typ in Domäne

```
public class ShowShoppingCart {  
    public ShoppingCartRenderModel render(Iterable<Product> inCart) {  
        final ShoppingCartRenderModel result = new ShoppingCartRenderModel();  
        for (Product each : inCart) {  
            result.addProductLine(  
                each.description(),  
                each.nettoPrice(),  
                ValueAddedTax.calculateFor(each));  
        }  
        return result;  
    }  
}
```

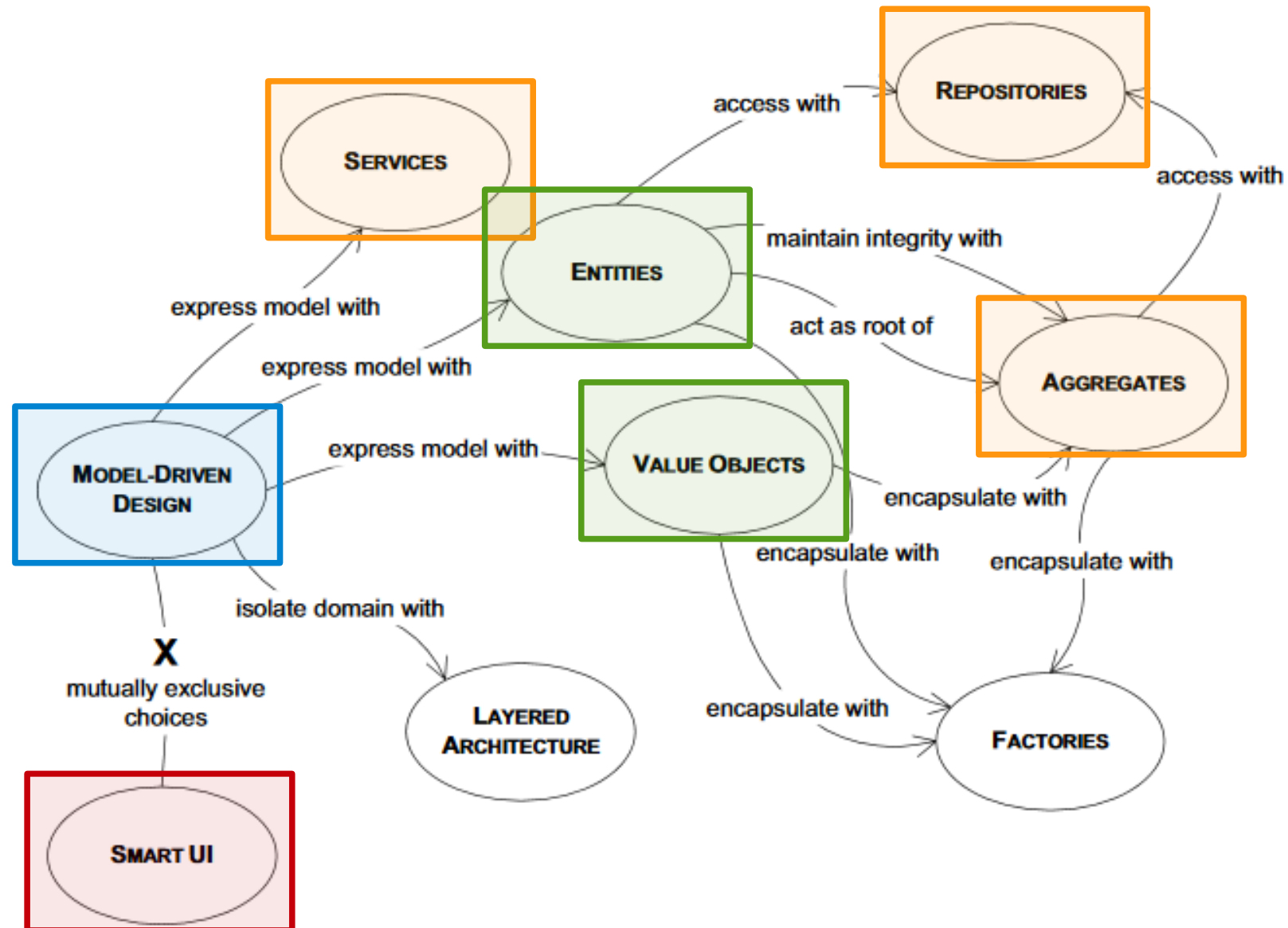
```
/**  
 * AN-17: Calculates the value added tax (VAT) for the given Taxable.  
 */  
public class ValueAddedTax {  
    public static Euro calculateFor(Taxable item) {  
        [...]  
    }  
}
```



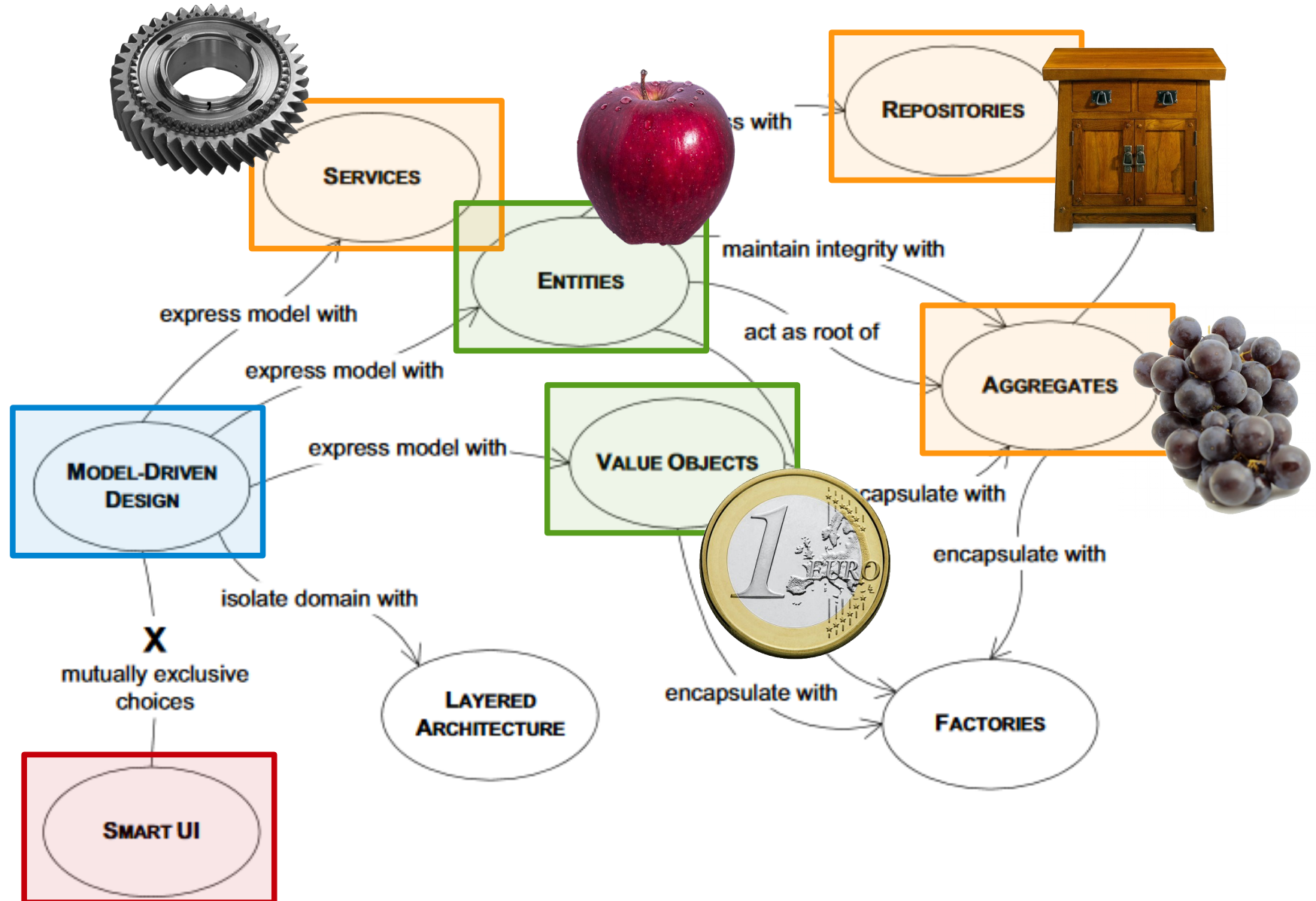
Passende Umsetzungssichtbarkeit

- Die Stufe der Umsetzungssichtbarkeit muss zur Wichtigkeit der Anforderung passen
 - Wie sprechen die Domänenexperten darüber?
 - Ist die Funktionalität abgrenzbar?
 - Soll die Anforderung isoliert getestet werden?
- Die Sichtbarkeit kann häufig leicht erhöht werden, aber manchmal nur schwer verringert
 - Lieber niedrig einsteigen und später erhöhen
- Dauerhaft zu niedrige Sichtbarkeit ist schädlich

Die Grundbausteine eines Modells
























Die Grundbausteine eines Modells



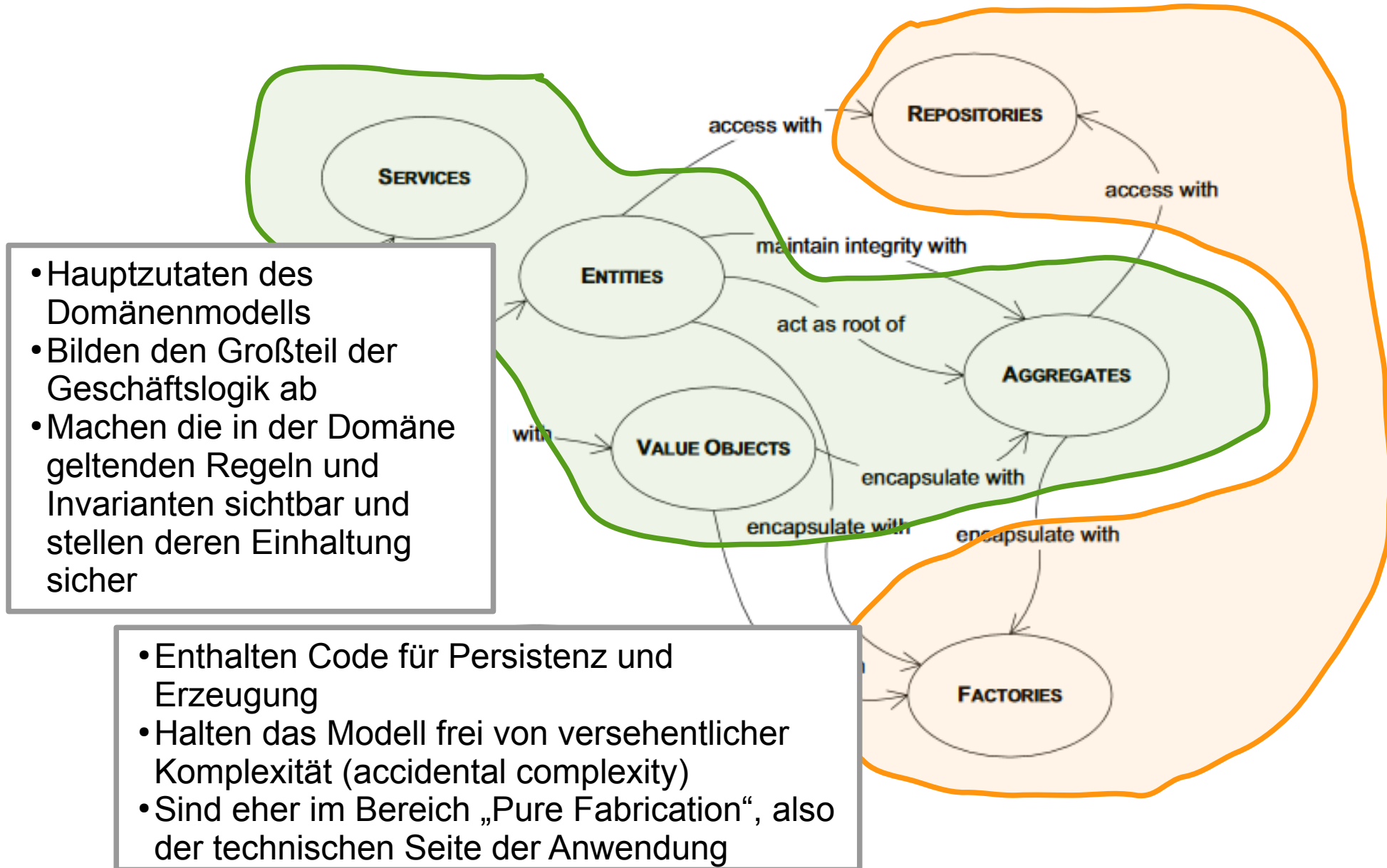
Smart UI - Anwendungen

- Viele Anwendungen sind nicht komplex genug, um von einem DDD-Modell zu profitieren
 - CRUD – Create, Read, Update, Delete
 - Datenbank- und Spreadsheet-Oberflächen
- Solche Anwendungen nennen wir „Smart UI“
 - Das ist kein abwertender Begriff
- Diese Anwendungen sollten nicht mit einem zusätzlichen Modell künstlich komplexer gemacht werden
 - Modelle müssen einen konkreten Nutzen haben

Beispiel Smart UI

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

Übersicht DDD Grundbausteine





Grundbaustein Value Objects

- Value Objects (VO) sind einfach Objekte ohne eigene Identität
- VO ändern ihre Eigenschaften/Werte nach dem Erzeugen nicht mehr (immutable)
- VO bilden das Konzept „Wert“, gerne mit dem Zusatz „zu einem bestimmten Zeitpunkt“ ab
- Zwei VO sind also gleich, wenn sie die selben/gleichen Werte haben
- Beispiel: Gewicht



Beispiel Value Objects: Gewicht

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



Grundbaustein Entities

- Entities sind Objekte, die aufgrund ihrer Identität modelliert werden
 - Sie bilden eine kontinuierliche Existenz ab
- Sie werden nicht durch ihre Werte definiert
- Die Werte sind über die Zeit veränderlich
- Jede Entity hat einen eigenen Lebenszyklus
- Zwei Entities sind verschieden, wenn sie unterschiedliche Identitäten (IDs) haben
 - Selbst wenn sie die gleichen Werte haben



Beispiel Entities: Person

- Jede Person hat eine eigene Identität
- Jede Person hat einen eigenen Namen
 - Die Identität ist unabhängig vom Namen
- Jede Person hat ein eigenes Gewicht
 - Das sich mit der Zeit ändern kann, ohne dass man gleich seine Identität verliert
- Zwei Personen mit gleichem Namen und gleichem Gewicht sind trotzdem verschieden
- Jede Person lebt seine individuelle Zeitspanne



Value Objects vs. Entities

Value Object

- Keine Identität (in der Domäne)
- Unveränderlich (Immutable)
- Kein Lebenszyklus
- Verschieden bei verschiedenen Eigenschaften
- Gibt Werten in der Domäne eine Semantik

Entity

- eindeutige Identität in der Domäne
- Hat veränderliche Eigenschaften
- Eigener Lebenszyklus
- Verschieden bei verschiedenen Identitäten
- Repräsentiert Ding in der Domäne



Zurück zu den Value Objects

- Value Objects (VO) sind einfach Objekte ohne eigene Identität
- VO bilden Konzepte ab. Dabei ist Semantik viel wichtiger als Struktur
 - Ein Preis besteht aus Betrag und Währung
 - Ein Rabatt besteht aus Betrag und Währung
 - Ist daher ein Rabatt nur ein besonderer Preis?
 - Nicht für die Domänenexperten!
 - Daher lieber zwei unabhängige VO
 - „Don't repeat yourself“ ist für die Domäne kein Ziel



Immutable ist gut

- Ein unveränderbares (immutable) Objekt hat viele Vorteile:
 - Gültig erzeugte Objekte bleiben gültig
 - Frei von Seiteneffekten
 - Einhalten von Regeln der Domäne wird einfach
 - Sehr leicht zu testen
 - Gleichheit ist entweder immer oder nie gegeben
 - Gleichheit basiert auf den Werten der Objekte



Gleichheit feststellen in Java

- Um zwei Objekte auf Wertegleichheit (Equality) zu überprüfen, wird in Java die equals()-Methode verwendet
- Wer equals() überschreibt, muss auch hashCode() passend(!) überschreiben
- Das ist eine Kunst mit klaren Regeln:

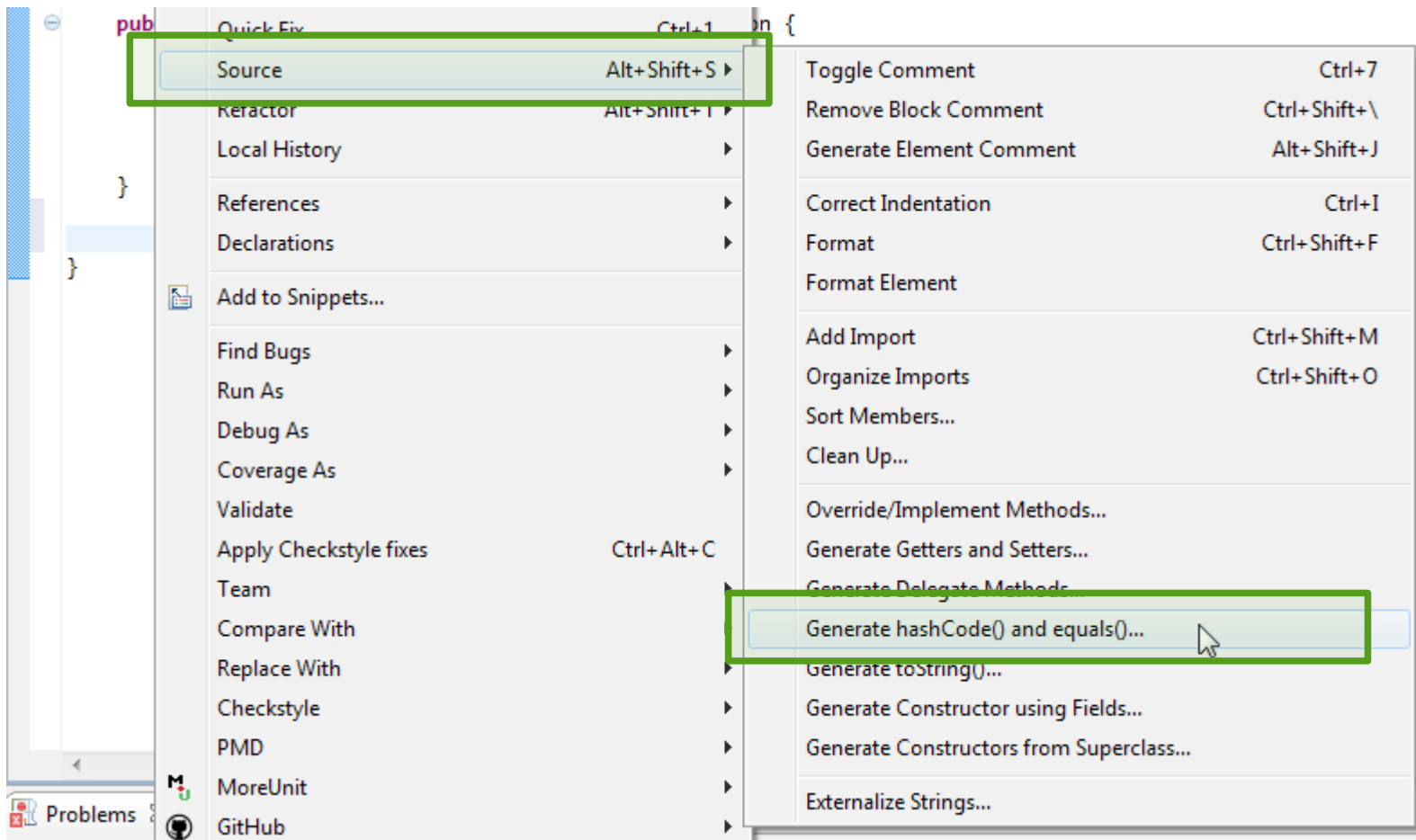
<https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html#equals-java.lang.Object->

<https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html#hashCode-->



Tipp: IDE helfen lassen

- Jede gute IDE kann die Methoden für euch generieren:





Value Objects erkennen

- VO beschreiben oder messen eine Sache
- Beispiele:
 - Geldbeträge, Datum, Farben, E-Mail-Adressen, Telefonnummern, physikalische Werte (Temperatur, Gewicht, Druck)
- VO können Zusammenfassungen anderer Objekte sein (oft wieder VO):
 - Geldbetrag: Betrag und Währung (132 €)
 - physikalischer Wert: Betrag und Einheit (78 kg)
 - Adresse: Straße, Hausnummer, PLZ und Ort



Value Objects implementieren

- 1. Alle Felder sind „blank final“ deklariert
- 2. Nur Konstruktor stellt gültigen Zustand ein
- 3. equals() und hashCode() sind überschrieben
- 4. Keine Setter oder andere Methoden, durch die Felder geändert werden
- 5. Methoden liefern als Rückgabe nur
 - a. Unveränderliche (immutable) Objekte
 - b. Defensive Kopien (beispielsweise bei Collections)
- 6. Klasse ist final deklariert (keine Vererbung)



6

Codebeispiel Value Object

```
public final class EuroCent {  
    1 private final int amount; 1  
  
    public EuroCent(int amount) {  
        super();  
        if (amount < 0) {  
            throw new IllegalArgumentException(  
                "Cannot create negative monetary amount: " + amount);  
        }  
        this.amount = amount; 2  
    }  
  
    public int inCent() {  
        return this.amount; 5a  
    }  
  
    public EuroCent add(EuroCent cents) {  
        return new EuroCent(this.amount + cents.amount); 5a  
    }  
  
    @Override  
    public int hashCode() { [...] } 3  
  
    @Override  
    public boolean equals(Object obj) { [...] }  
}
```



Mehr zu den Entities

- Entities bilden identifizierbare und identitätstragende „Etwasse“ aus der Domäne ab
- Jede Entity hat ihren eigenen Lebenszyklus
 - Die Eigenschaften der Entity können sich ändern
- Auch Entities kümmern sich darum, die Regeln der Domäne zu forcieren
 - Der Konstruktor erzeugt nur gültige Instanzen
 - Spätere Änderungen der Eigenschaften dürfen die Entity nicht in einen ungültigen Zustand versetzen
 - „Gültiger Zustand“ ist durch die Domäne bestimmt



Grundlage des Domänenmodells

- Entities sind zusammen mit den Value Objects die Basis jeden Domänenmodells
- Nicht alle Entities sind physikalisch manifestierte „Etwasse“
 - In einer Bankanwendung sind Kunden, Konten und Verträge Entities und real vorhanden
 - Kontenbewegungen sind ebenfalls Entities, aber vermutlich nicht mehr real vorhanden
- Entities tauchen in der Sprache der Domänenexperten irgendwann auf (meist als Substantiv)



Identität von Entities

- Die Identität einer Entity in der Domäne ist nicht die Objektidentität (Speicheradresse) in der Programmiersprache
 - Oft ist sogar die Verwendung von equals() und hashCode() für die Domänenidentität schwierig
- Zwei Entities können sogar die gleiche Identität (in der Domäne) haben, obwohl sie Instanzen von unterschiedlichen Klassen/Typen in der Software sind
- Die Identität einer Entity ist oft in der Domäne wichtig. Sie ist immer für die Entwicklung wichtig.



Beispiel für Identität von Entities

- Logistik im zweiten Weltkrieg
 - Jede Kiste hat eine eindeutige Nummer – für den Heeresteil, dem sie gehört
 - Jedes Schiff hat bis zu drei Nummern: Eine von der Marine, eine für das Heer und eine für private Expeditionen
 - Alle Nummern waren 6- oder 7-stellig
- In einem Lagerhaus konnten drei Kisten „123456“ stehen (Marine, Heer, Luftwaffe)
 - Die Kiste 123456 des Heeres soll auf das Schiff 7654321 verladen werden ...



Eigenschaften von Identitäten

- Mindestens drei Arten von Identität:
 - Kombination von Eigenschaften
 - Surrogatschlüssel
 - Natürlicher Schlüssel
- Jede Art und Vorgehensweise, Identitäten zu vergeben, hat Vorteile und Nachteile
 - Ohne klare Gründe erstmal die Vorgehensweise der Domäne verwenden
- Es können durchaus verschiedene Vorgehensweisen in einer Anwendung verwendet werden



Kombination von Eigenschaften

- Identifikation einer Ausgabe einer Tageszeitung:
 - Name der Zeitung
 - Ausgabeort
 - Erscheinungsdatum
- Was ist mit Sonderausgaben?
- Was passiert bei Änderung eines Namens?
 - z. B. Ausgabeort: Chemnitz vs. Karl-Marx-Stadt
 - z. B. Name der Zeitung: Märkische Volksstimme in Märkische Allgemeine



Surrogatschlüssel

- Von der Anwendung (intern) selbst generierte Identifizierer
 - Eigenes, zeichenkettenbasierendes Format
 - Inkrementierender Zähler
 - UUID: Universally Unique Identifier (auch: GUID)
 - Aufbau in RFC 4122 beschrieben
 - Basierte auf MAC-Adresse und Uhrzeit.
 - Mittlerweile nur noch eine große Zufallszahl (Kollision möglich?)

```
f4f6bb41-727b-4524-82da-a181ecb95cf1
```

```
final UUID id = UUID.randomUUID();
```




Surrogatschlüssel

- Vorteile:
 - Jederzeit generierbar (keine externe Abhängigkeit)
 - Als UUID anwendungsübergreifend eindeutig
- Nachteile:
 - Nicht sprechend
 - Keine Bedeutung in der Domäne
 - Generierung wird eventuell Engpass bei Hochlastsystemen



Natürlicher Schlüssel

- Ein bereits vorhandener Identifikator aus der Domäne
- Beispiele:



ISBN



Personalausweis-
Nummer

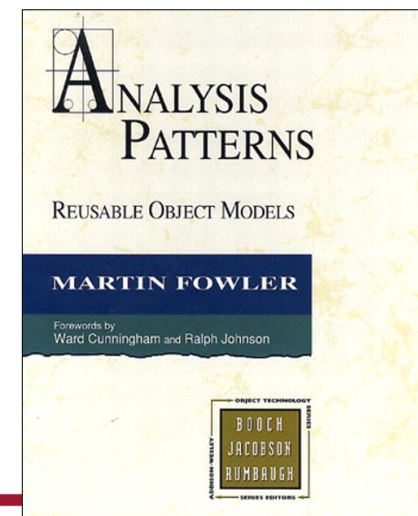


KFZ-Kennzeichen

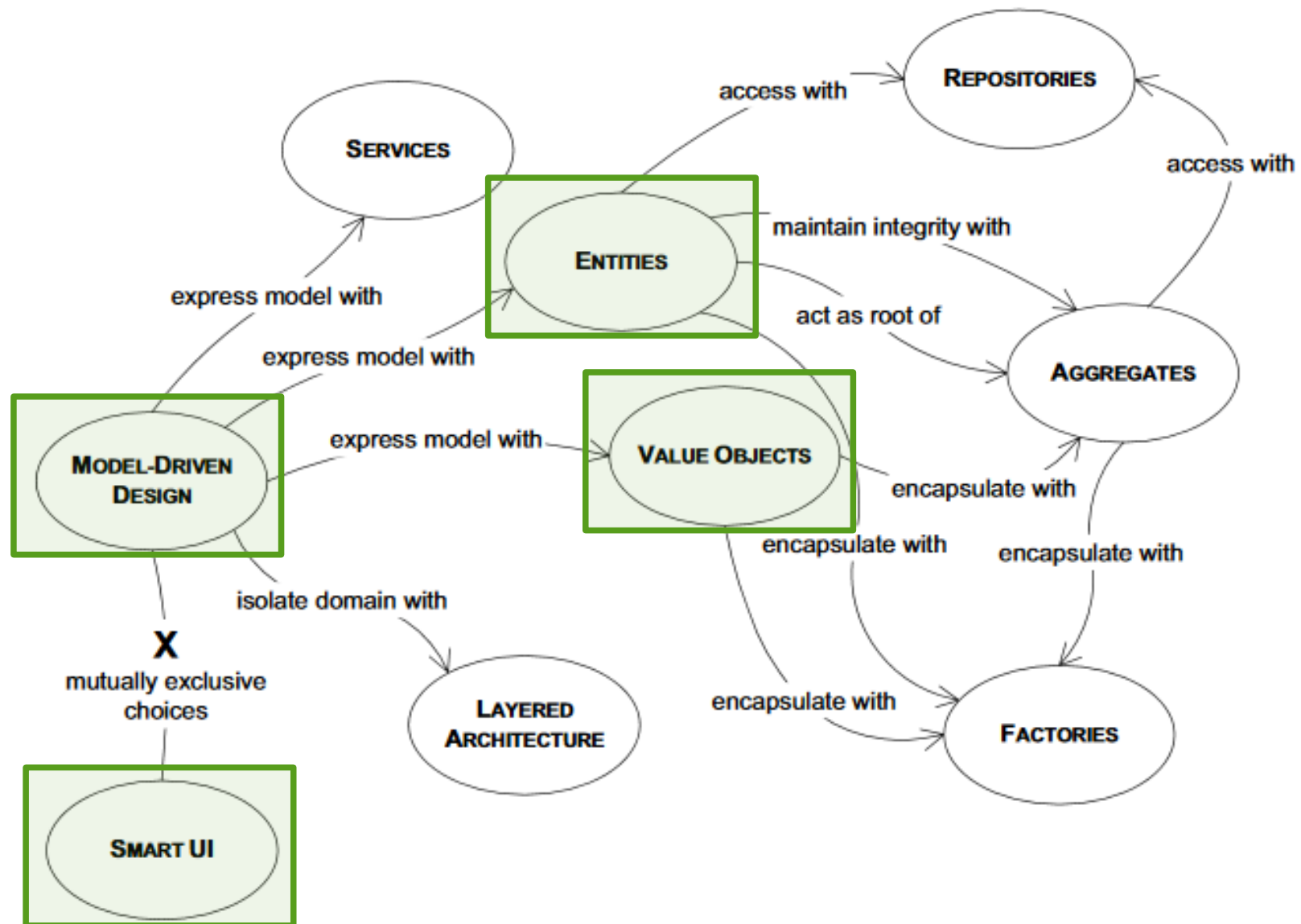


Natürlicher Schlüssel

- Vorteile:
 - Sehr aussagekräftig (Benutzer des Systems können ihn intuitiv zuordnen)
 - Keine Duplikate innerhalb eines Kontextes
- Nachteile:
 - Zwangsläufig fremdbestimmt
 - Wird sich das Format der MAC-Adresse eines Gerätes wirklich niemals ändern?
 - Keinerlei Garantie auf Duplikatfreiheit (Eindeutigkeit) außerhalb des Kontextes
 - Neu ab 1939: „Kombinierte Operationen“



Zusammenfassung Teil 1



Zusammenfassung Teil 1

- Es gibt viele Anwendungen, die nicht komplex genug sind, um ein Model-Driven Design zu benötigen oder zu rechtfertigen
 - Diese Anwendungen nennt man Smart UI-Anwendungen
- Ein DDD-Modells besteht grundlegend aus
 - Value Objects – unveränderlichen Werten ohne Lebenszyklus oder Identität
 - Entities – veränderlichen Etwassen mit Identität und individuellem Lebenszyklus



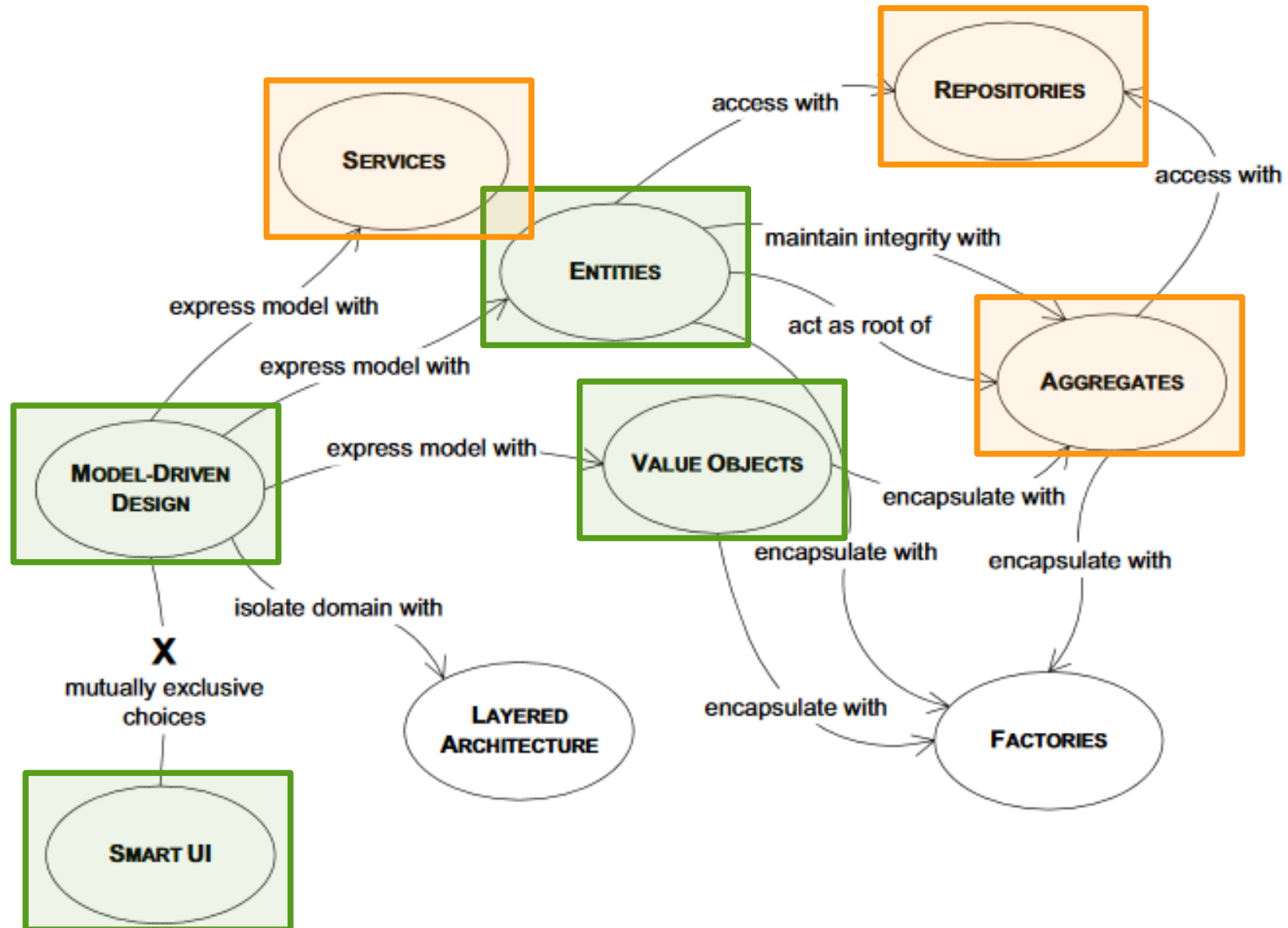
Softwareschneiderei

Domain Driven Design

Eine Sprache, sie alle zu finden
(Teil 2)



Agenda Teil 2





DDD Services

- Das Problem an DDD: Die Begriffe sind vielfach überladen und umgedeutet
- Ein DDD Service ist kein Service aus der Serviceorientierten Architektur (SOA)
- Ein DDD Service ist auch kein Service in Sinne der Microservices
- Ein DDD Service sollte eher Domain Service heißen und ist ein Hilfskonstrukt, um keine „unechten“ Entities oder Value Objects im Modell zu haben



Zweck von Domain Services

- In einem Domain Service landet komplexes Verhalten, Vorgänge oder Regeln der Domäne, die sich nicht einer bestimmten Entity oder einem VO zuordnen lassen

In der Clean Architecture: Use Case

- Der Domain Service definiert einen Funktionsvertrag für einen externen Dienst, damit das Domänenmodell frei von technischer Komplexität bleibt

In der Clean Architecture: Interface für Adapter



Use Case Domain Service (Beispiel)

- Um den aktuellen Zustand eines gelagerten Buches zu bestimmen, benötigt man
 - Das Buch (Entity)
 - Alle Zustandsbewertungen (Entities)
 - Enthält u.a. die Schadensklasse (Value Object)
 - Alle erfolgten Restaurierungsmaßnahmen (Entities)
- Das Verhalten kann nicht eindeutig einer dieser Entities (oder VO) zugeordnet werden
 - In einen Domain Service auslagern
 - Passender Name?

BookStateDetermination

Buchzustandsbestimmung



Use Case Domain Service (Beispiel)

```
public class Buchzustandsbestimmung {  
    public Buchzustandsbestimmung() {  
        super();  
    }  
  
    public Buchzustand anhandVon(  
        final Buch buch,  
        final Iterable<Zustandsbewertung> bewertungen,  
        final Iterable<Restaurierungsmaßnahme> restaurierungen) {  
        final Buchzustand result =  
            Komplizierte, normgerechte Berechnung  
        return result;  
    }  
}
```

- Sprechende Schnittstelle
- Erfüllt das Single Responsibility Principle
- Sehr einfach zu testen
 - Unit Test – keine Abhängigkeit von Datenbank, etc.



Adapter Interface Domain Service

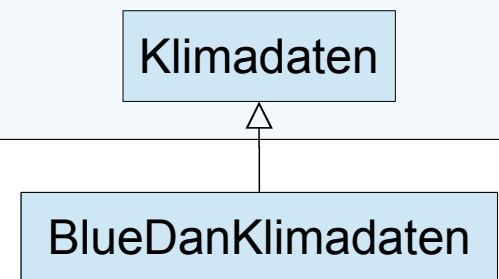
- Die Klimadaten in der Nähe eines gelagerten Buches können von externen Sensoren mit Speicher abgefragt werden
 - Wie genau das passiert, ist für das Domänenmodell völlig unerheblich
 - es soll frei von genau diesen technischen Details sein
- In der Domäne ist nur wichtig:
 - „Die **Klimadaten** jedes **Lagerorts** können für jeden **Zeitraum** von einer externen Datenquelle bezogen werden“
 - Passender Name?

Klimadaten



Adapter Interface DS (Beispiel)

```
public interface Klimadaten {  
    Optional<Iterable<Klima>> für(  
        final Lagerort ort,  
        final Zeitraum intervall);  
}
```



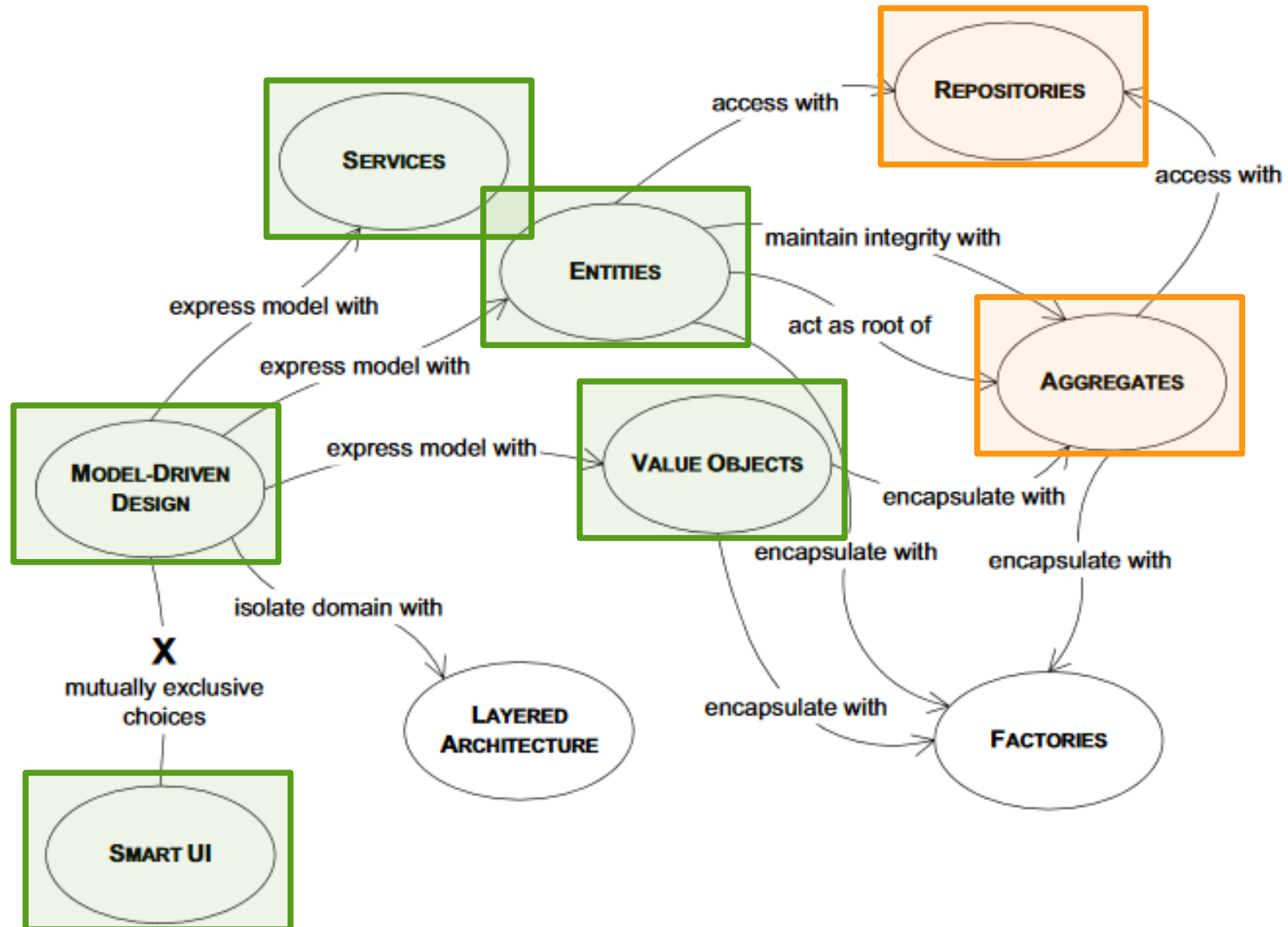
- Sprechende Schnittstelle
- Definiert im Kern der Anwendung
 - Bestimmt die technischen Anforderungen
- Implementiert in der Peripherie (als Plugin)
- Leicht durch Pseudo-Implementierung zu ersetzen
 - Für die Entwicklung ohne Hardware, Vorführungen



Eigenschaften von Domain Services

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

Abschluss Domain Services





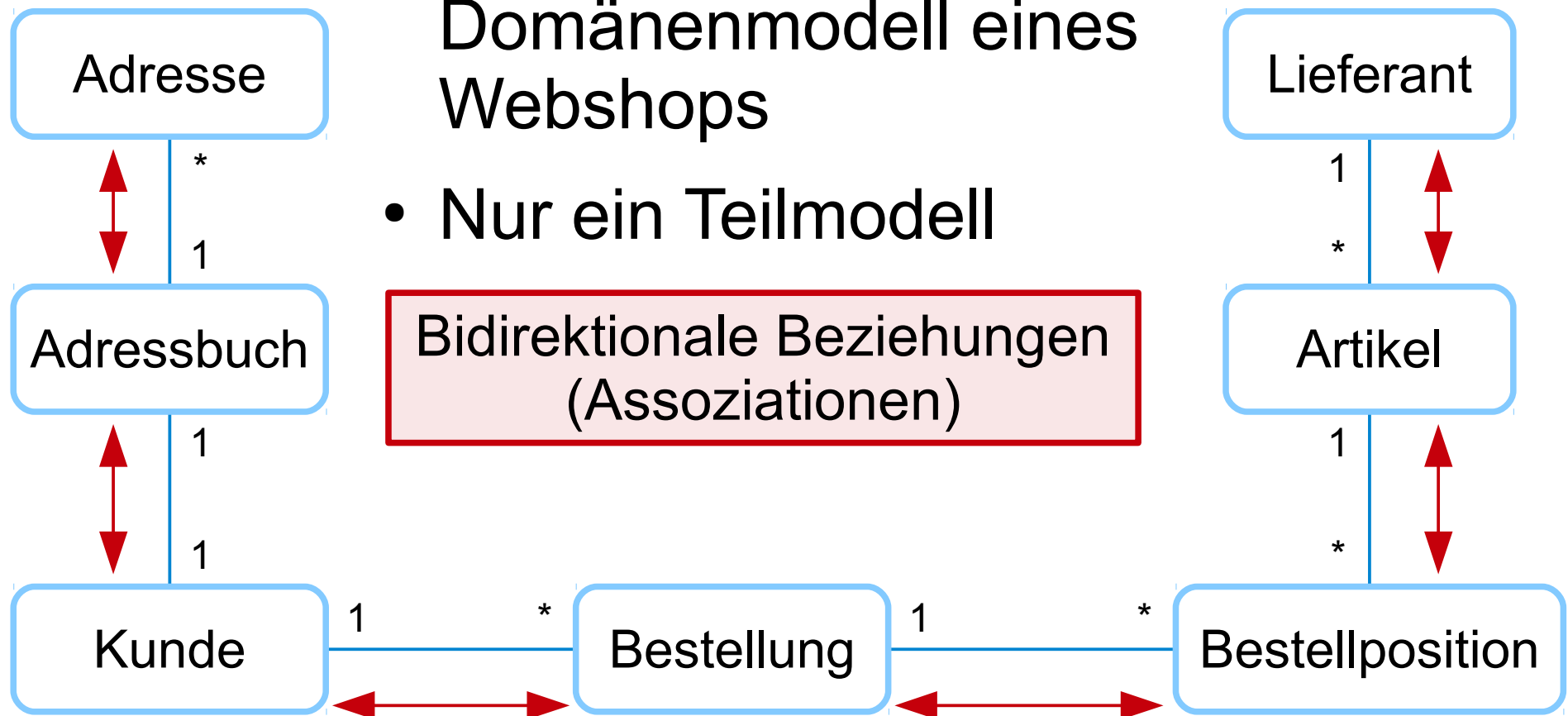
DDD Aggregates

- Wenn die Domäne maßstabsgetreu modelliert wird, findet man viele Entities und VO, die große Objektgraphen mit oft bidirektionalen Abhängigkeiten bilden
- Das wird schnell ungemütlich:
 - Wahrscheinlichkeit nicht eingehaltener Regeln steigt
 - Verstärkt Kollisionen beim gleichzeitigen Bearbeiten
 - Performance-Einbußen durch Warten auf Sperrenfreigabe
 - Lange Wartezeiten beim Laden und Speichern



Zu viele Abhängigkeiten

- Ausschnitt aus dem Domänenmodell eines Webshops
- Nur ein Teilmodell



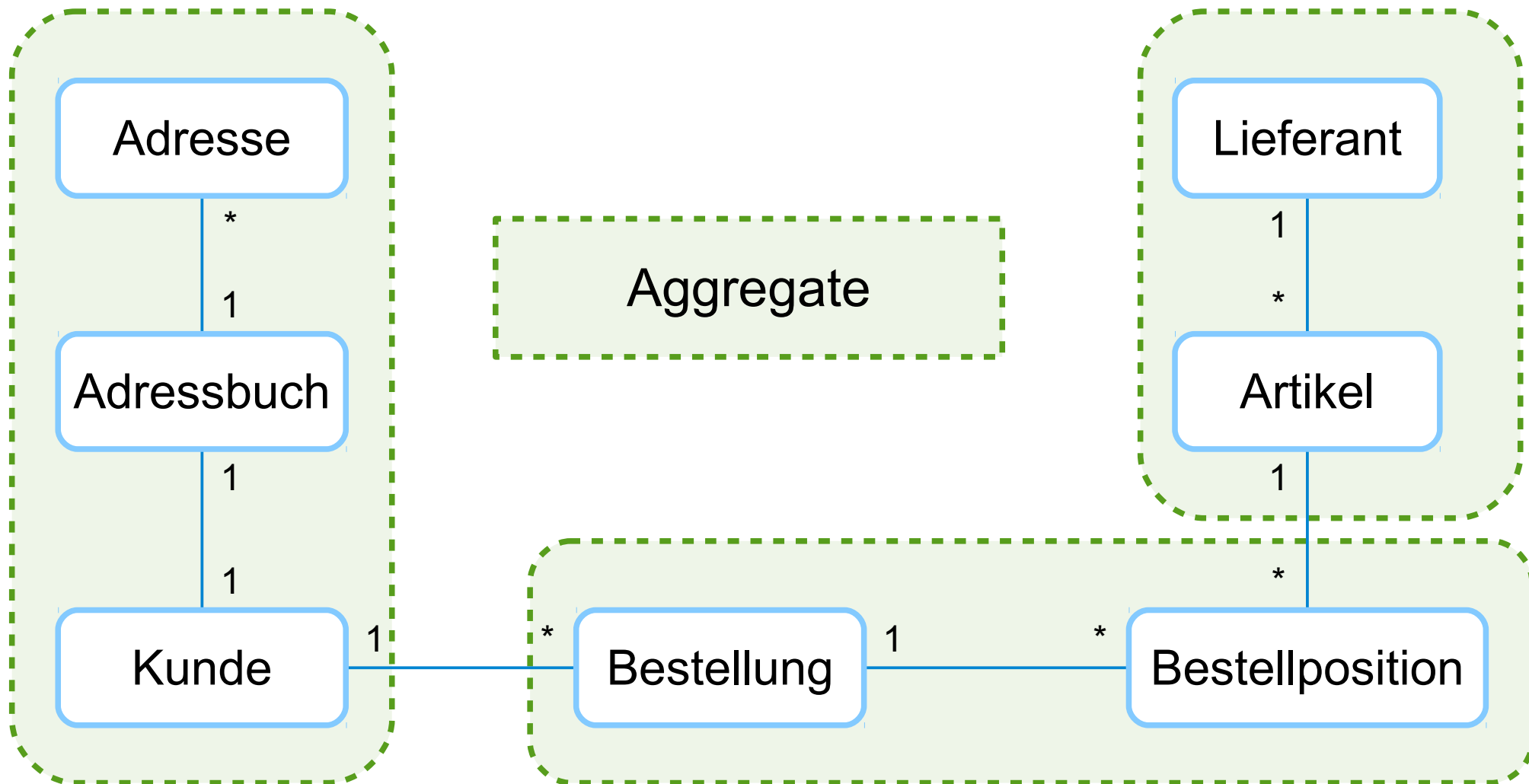


Aggregate zur Reduktion

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

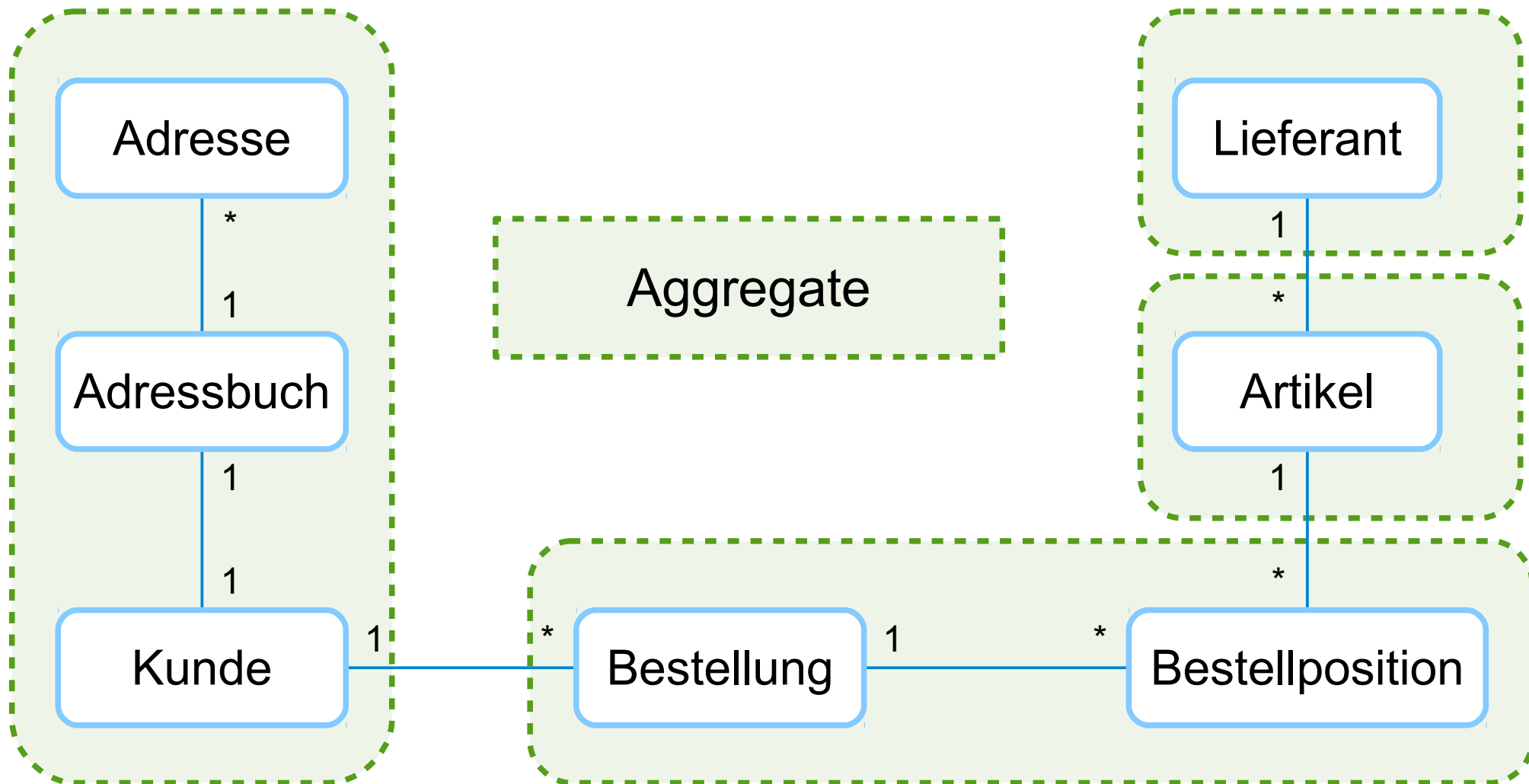


Aggregate im Beispiel



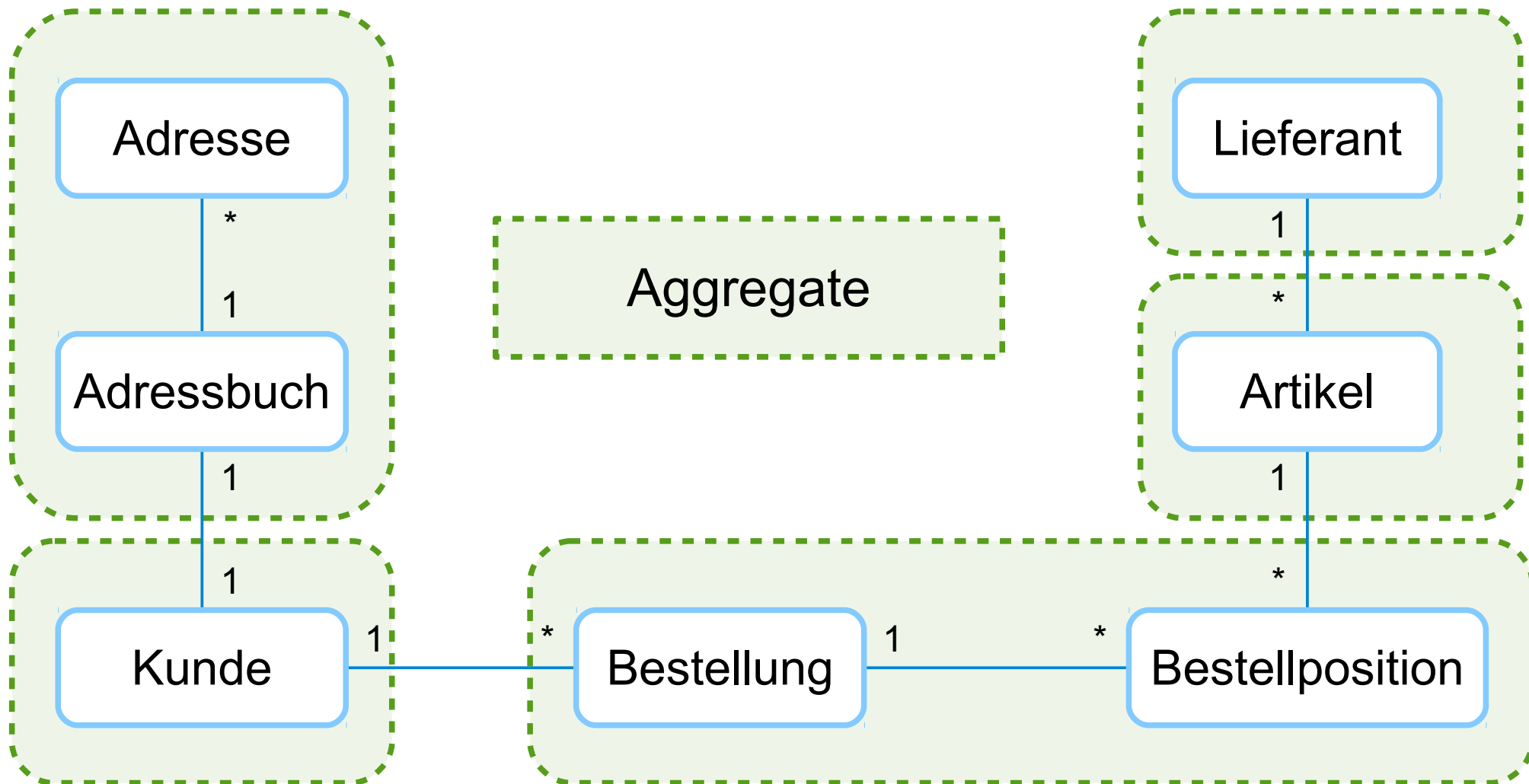


Aggregate im Beispiel – Variante 2





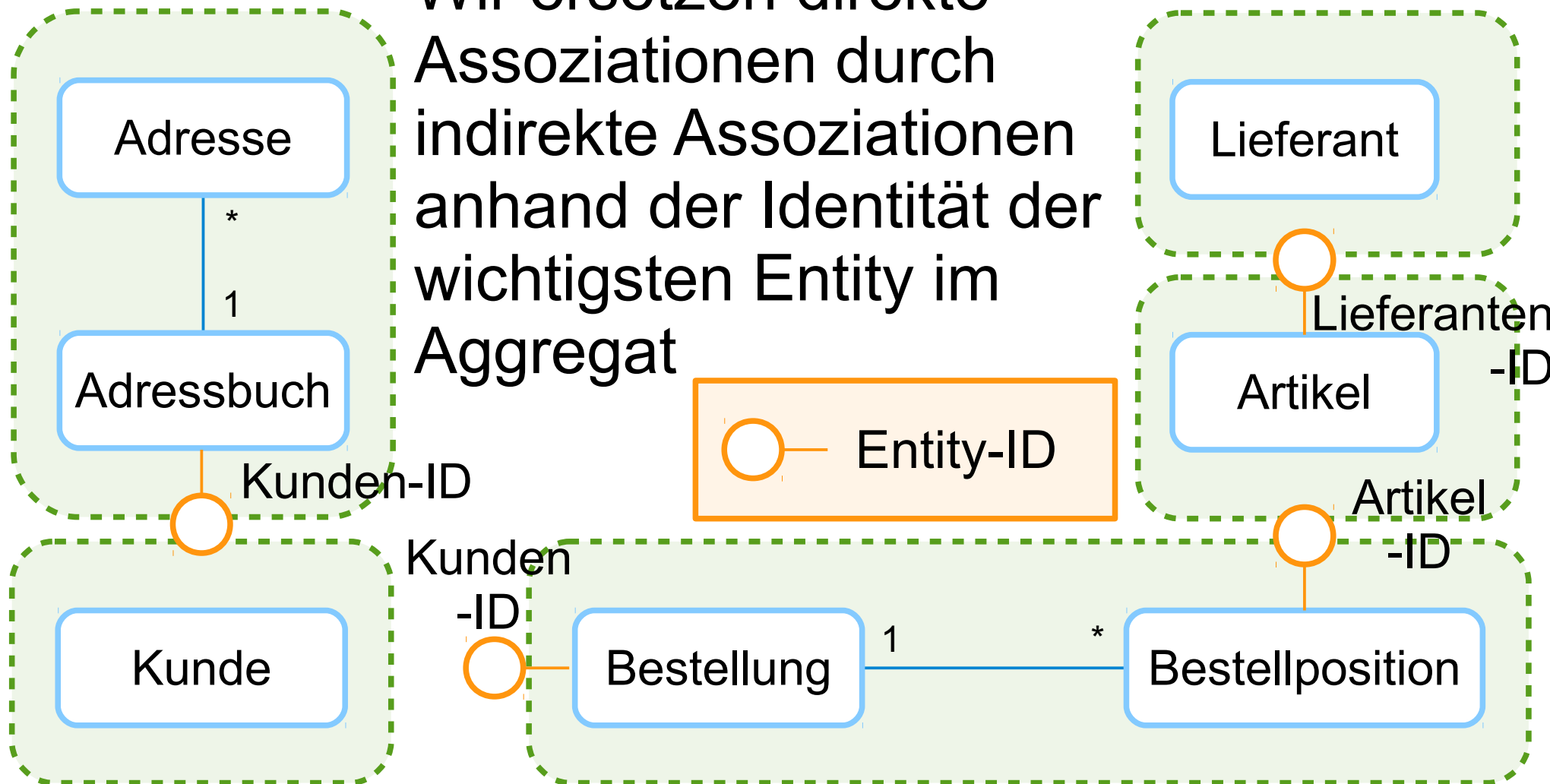
Aggregate im Beispiel – Variante 3





Assoziationen lösen

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





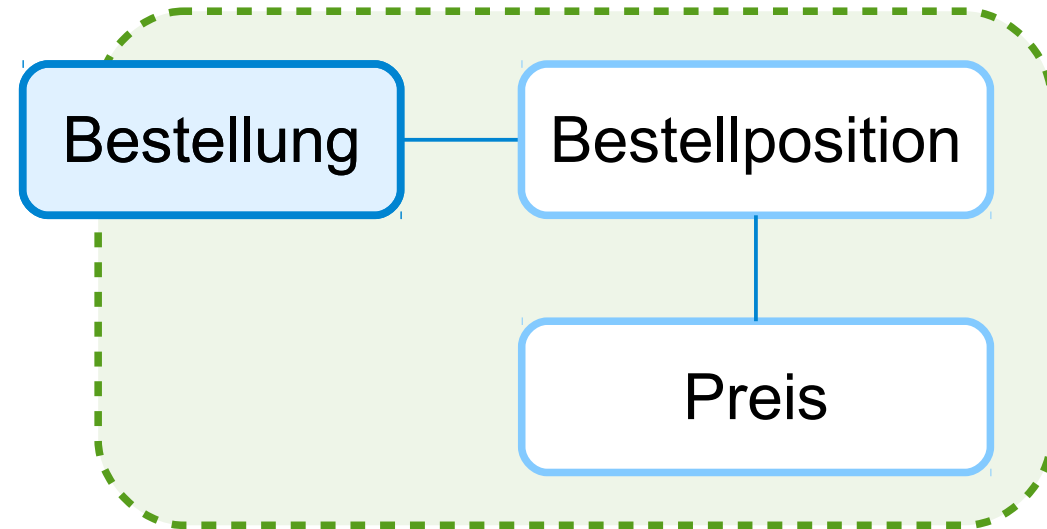
Aggregate Root Entity

- In jedem Aggregat übernimmt eine Entity die Rolle der Aggregate Root Entity bzw. des Aggregat Root (AR)
- Alle Zugriffe auf das Aggregat müssen über das AR erfolgen
 - Auch Zugriffe auf die inneren Elemente des Aggregat
- Langfristige direkte Referenzen auf innere Elemente sind nicht erlaubt
 - Nur temporäre Referenzen während einer Berechnung



Aggregate Root

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





Aggregate und Außenwelt

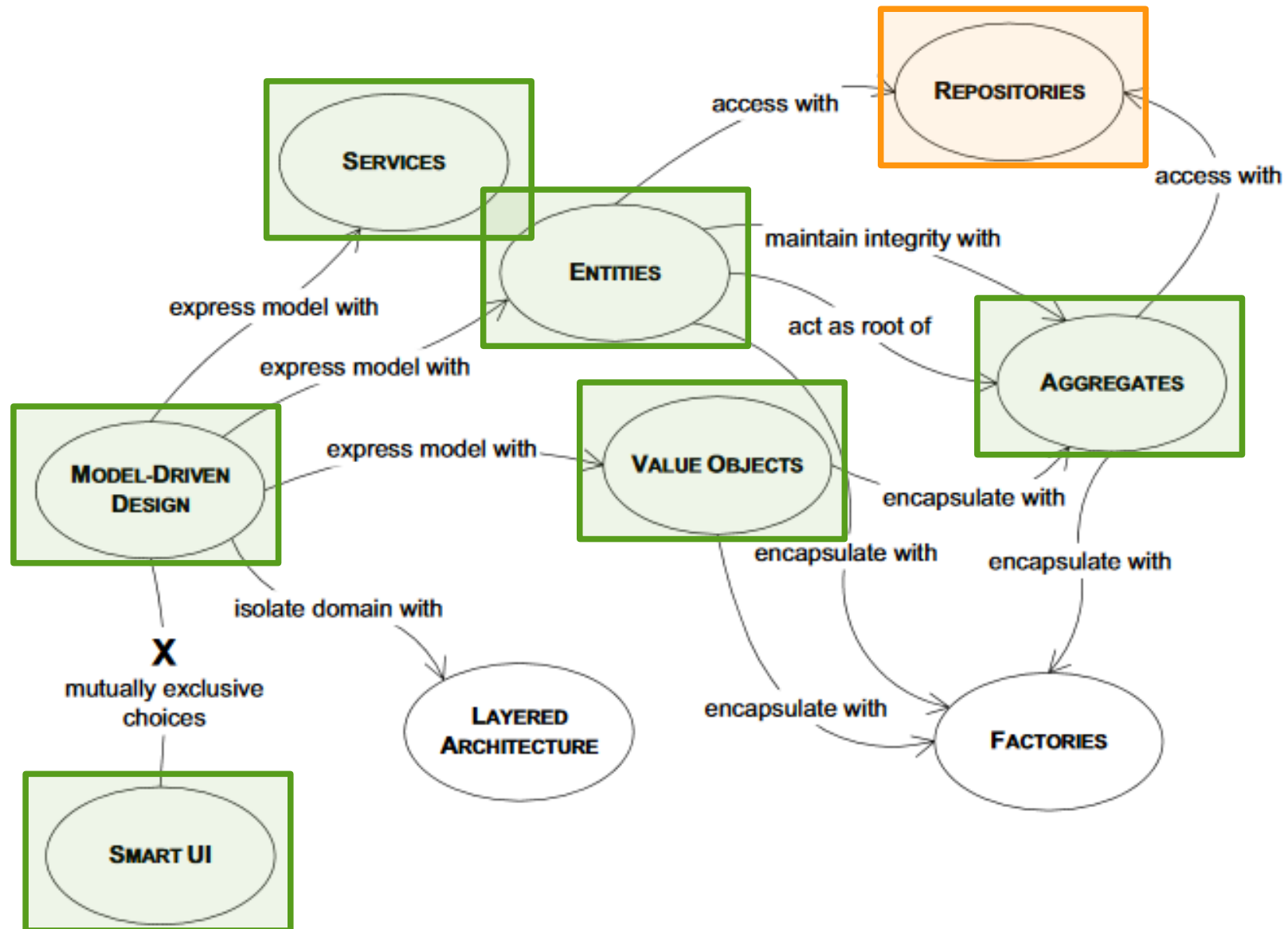
- Zugriff auf Aggregat nur über Aggregate Root
- Wenn das AR Referenzen auf innere Objekte herausgeben muss, sollten das immer defensive Kopien oder Immutable-Dekorierer sein
- Das Aggregat sorgt dafür, dass sein Zustand immer den Domänenregeln entspricht
 - Alle Änderungen gehen über den AR und sind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



Zusammenfassung Aggregate

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

Abschluss Aggregates





DDD Repositories

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



DDD Repositories

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



DDD Repositories

- Die Methoden des Repository-Interface werden in der Sprache der Domäne benannt
- Der Klassenname kann „Pure Fabrication“ sein

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



DDD Repositories

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

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



DDD Repositories

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

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

- Gerne auch Abfragen, die nur Metadaten zurückgeben
 - Sind in der Speicherschicht effizienter umzusetzen

```
int aktuellerBuchbestand();
```




Zusatznutzen von Repositories

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

```
public BuchId nächsteId();
```

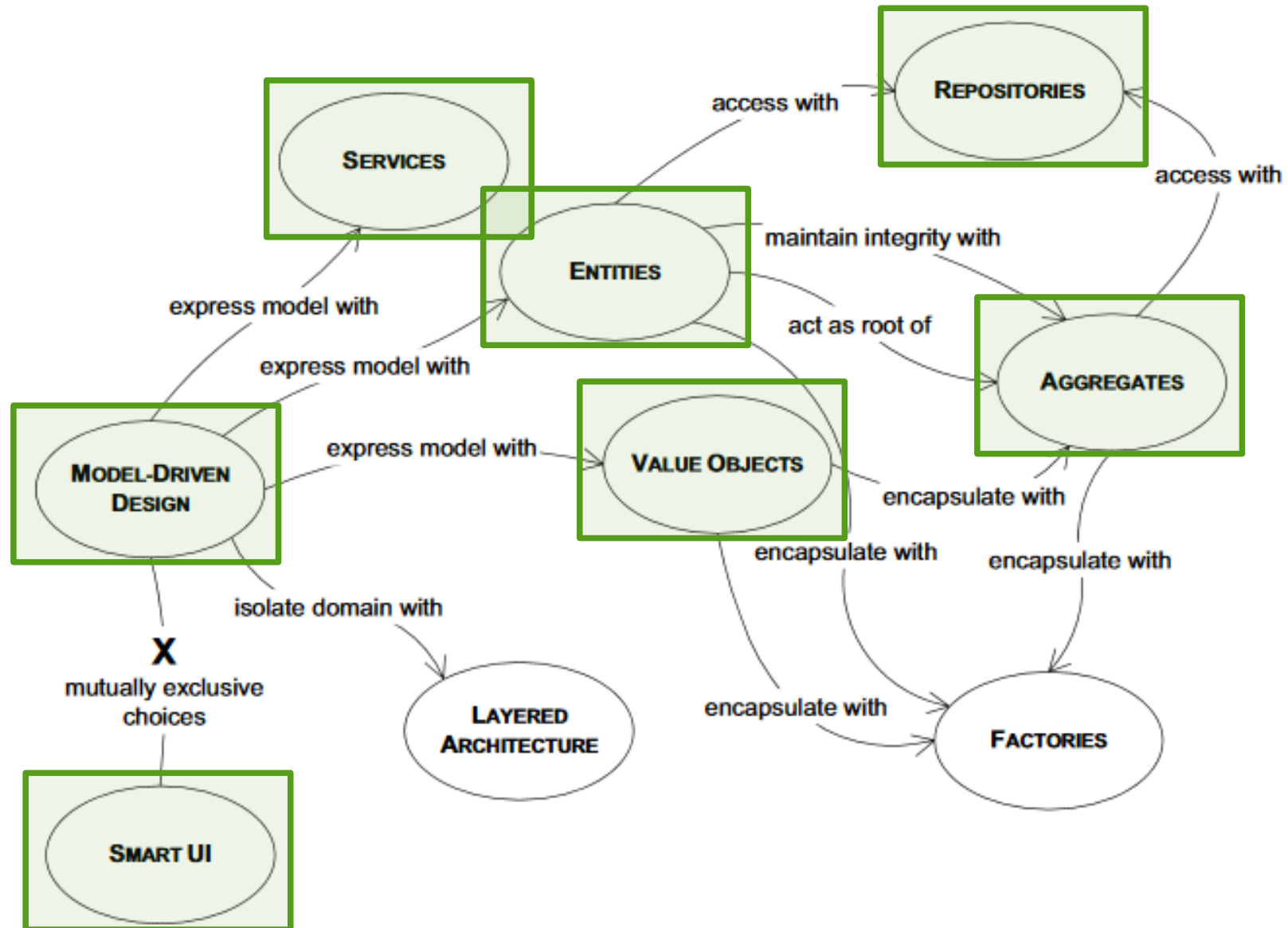
- Kann zusätzliche speicherseitige Prüffelder bei Veränderungen setzen („zuletztGeändertAm“)
- Kann für Unit Tests leicht gemockt werden
- Kann für Integrationstests ohne Datenbank durch eine In-Memory-Implementierung ersetzt werden



Zusammenfassung Repositories

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

Abschluss Repositories



Zusammenfassung Teil 2

- Um die Entities und Value Objects technisch sauber implementieren zu können, benötigen wir unterstützende Strukturen
- **Domain Services** enthalten Use Cases oder entkoppeln von Drittsystemen
- **Aggregates** gruppieren Entities und vereinfachen die Beziehungen zwischen den Gruppen
 - **Aggregate Roots** stellen die primären Objekte dar
- **Repositories** entkoppeln die Persistenz und machen die Aggregate Roots einfach auffindbar

Sichtbarkeitsstufen der Umsetzung

- Stufe 0: Inline
 - Stufe 0+: Inline mit Kommentar
- Stufe 1: Eigene Methode
- Stufe 2: Eigene Klasse
 - Stufe 2+: Eigener Typ im Domänenmodell
- Stufe 3: Eigenes Aggregat
- Stufe 4: Eigenes Package/Modul
- Stufe 5: Eigene Anwendung (Service)

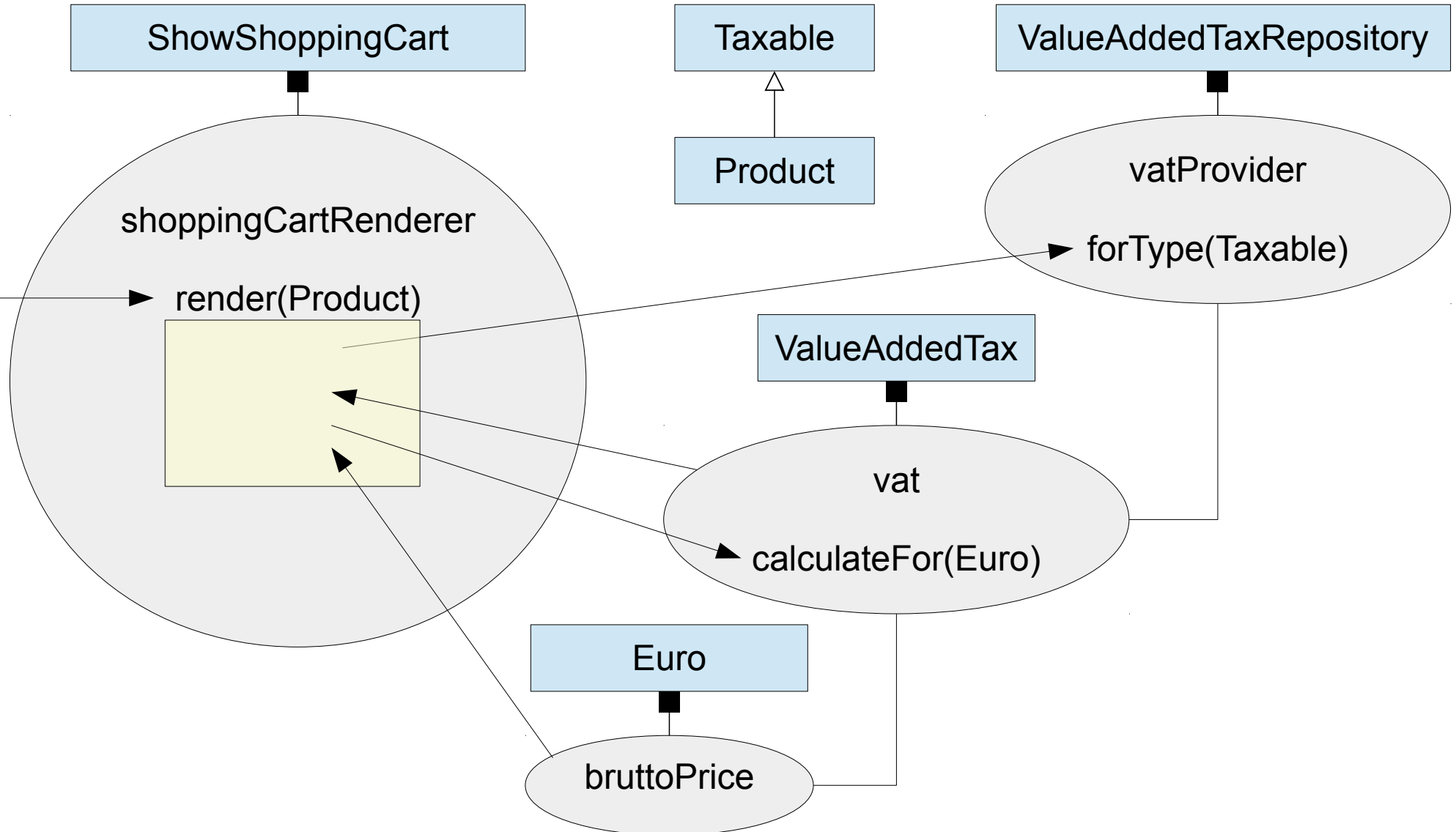
Stufe 3: Eigenes Aggregat

```
public class ShowShoppingCart {  
    public ShoppingCartRenderModel render(Iterable<Product> inCart) {  
        final ShoppingCartRenderModel result = new ShoppingCartRenderModel();  
        final ValueAddedTaxRepository vatProvider = new ValueAddedTaxRepository();  
        for (Product each : inCart) {  
            final ValueAddedTax vat = vatProvider.forType(each);  
            final Euro bruttoPrice = vat.calculateFor(each.nettoPrice());  
            result.addProductLine(  
                each.description(),  
                each.nettoPrice(),  
                bruttoPrice);  
        }  
        return result;  
    }  
}
```

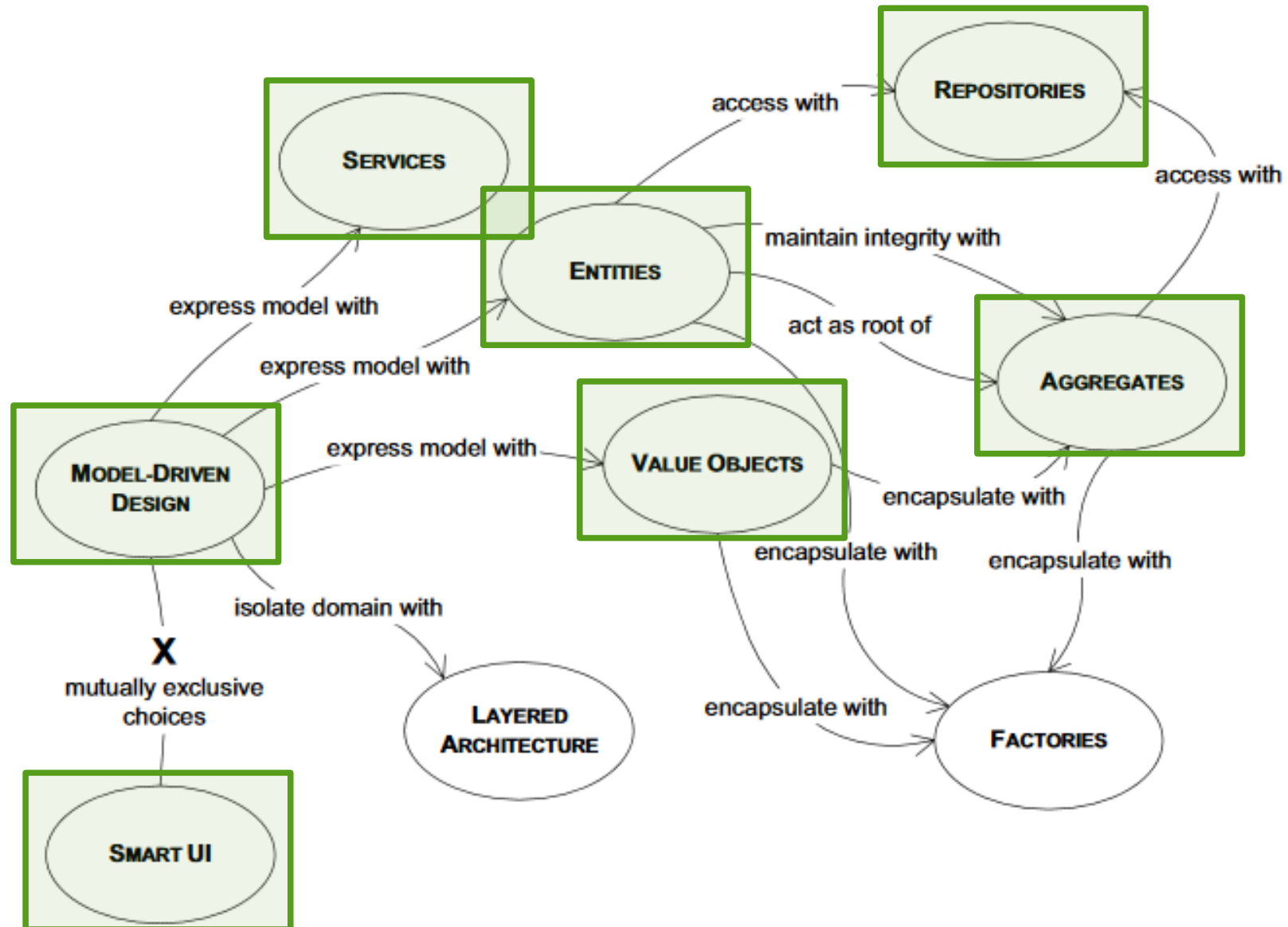
```
/**  
 * AN-17: Calculates the brutto price (netto price with value added tax (VAT))  
 * for the given netto price, using the associated VAT category.  
 */  
public class ValueAddedTax {  
    public Euro calculateFor(Euro nettoPrice) { [...] }  
}
```

```
public class ValueAddedTaxRepository {  
    /**  
     * Loads a ValueAddedTax for the given taxable item, using it as a VAT category.  
     */  
    public ValueAddedTax forType(Taxable category) { [...] }  
}
```

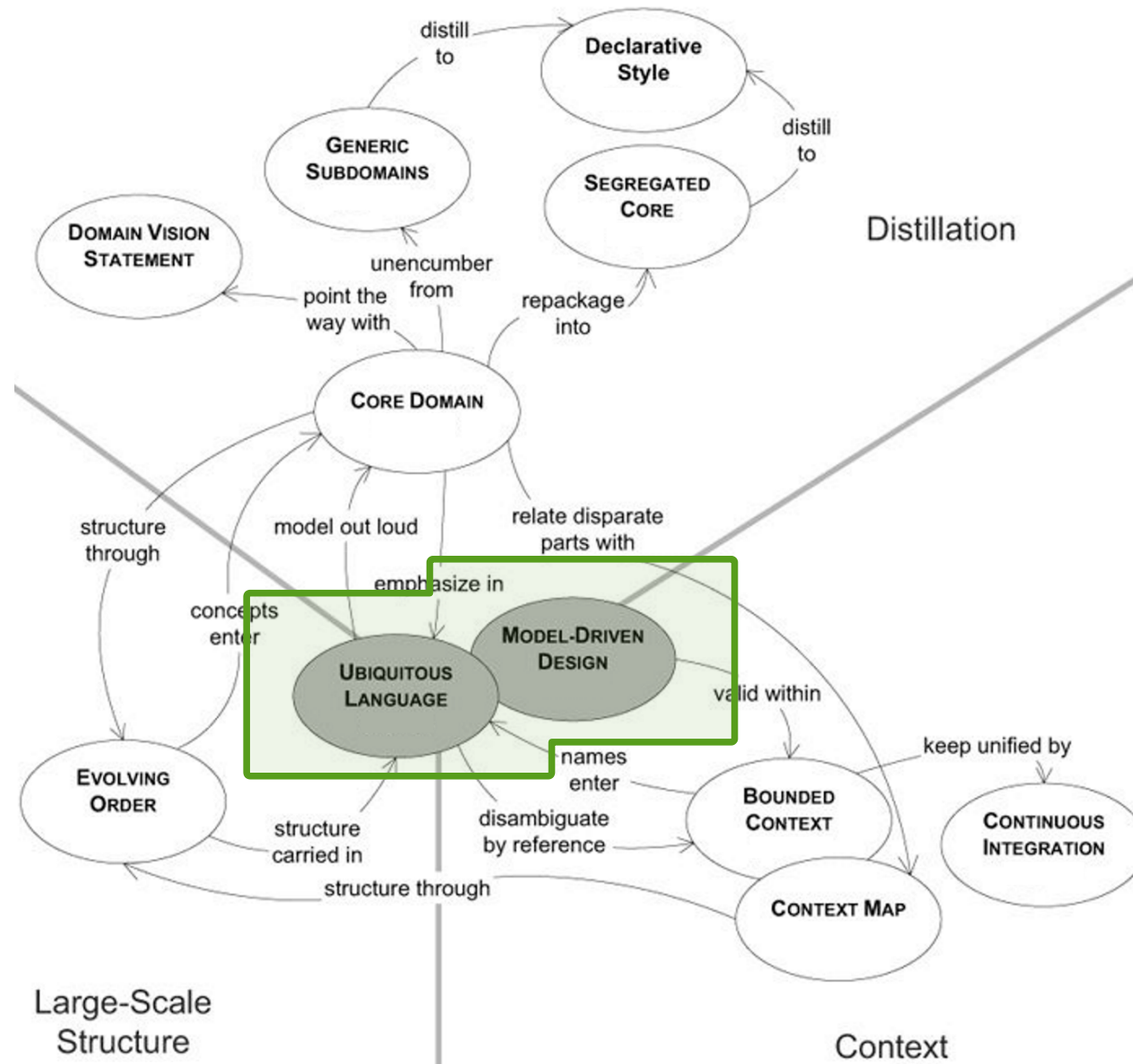
Ablauf auf Stufe 3 (Aggregat)



Zusammenfassung Teil 2



Aus- und Überblick





Domain Events

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



Beispiel für Domain Events

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



Domain Events als Stream

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



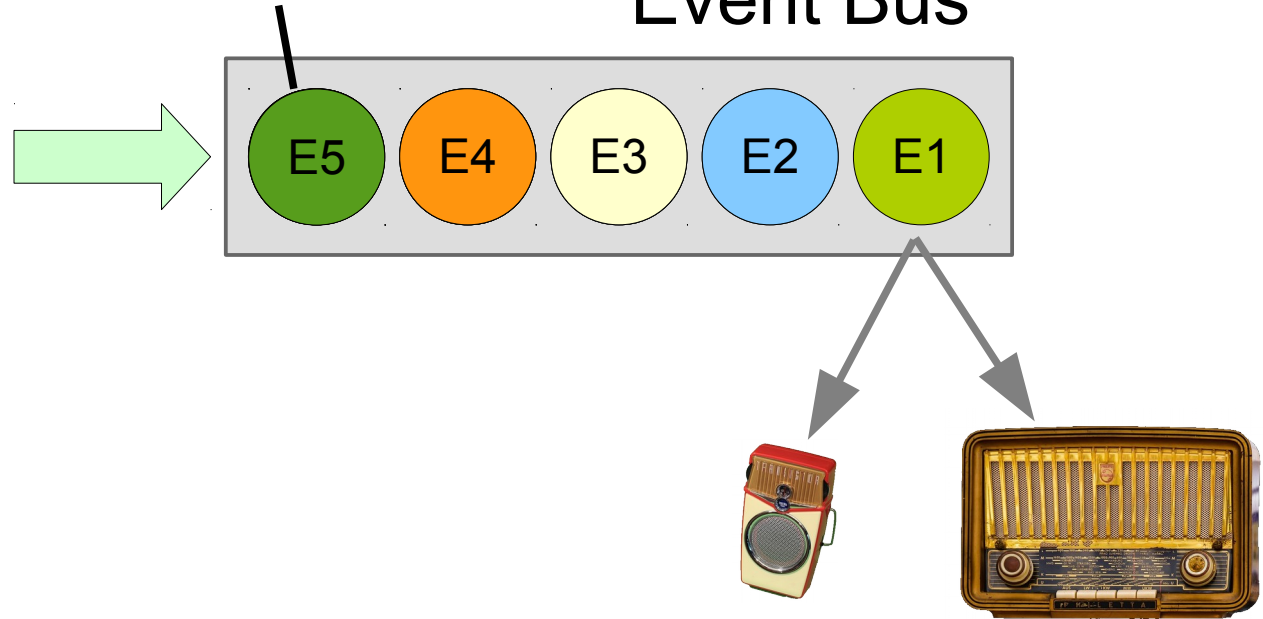
Domain Events und Event Bus

Event Source



(Domain)
Event

Event Bus



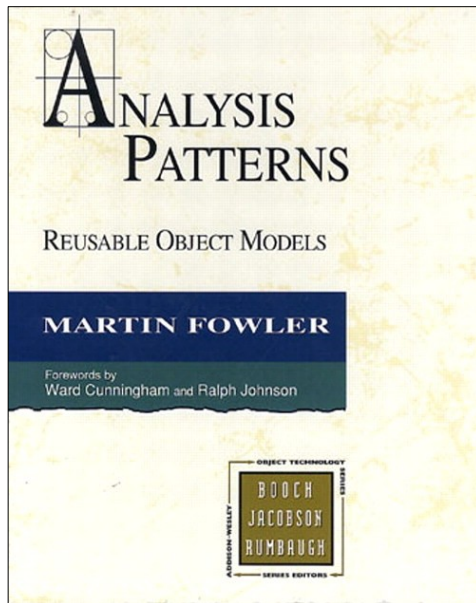
Subscribers



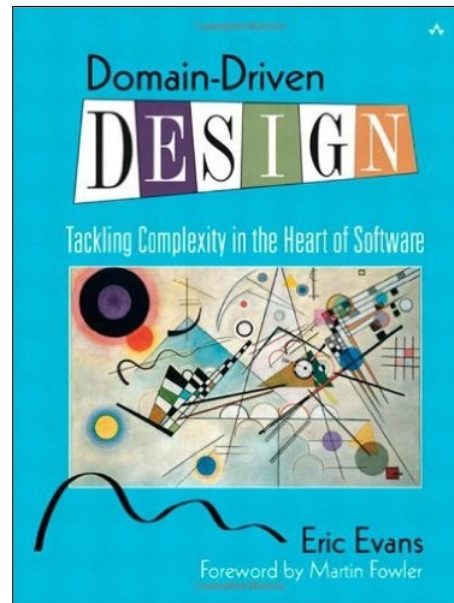
Zusammenfassung Domain Events

- Änderungen in der Domäne als „Event“ posten
- Jedes Event enthält die relevanten Neuigkeiten
 - Mindestens Was, Wann, Wer, Von wem erstellt
- Durch den Ereignisstrom werden sehr mächtige Architekturen möglich
 - Vollständige Entkopplung von Teilsystemen
 - Vollständige Rekonstruktion des Anwendungszustands (Event Sourcing)
- Vergleichbare Konzepte
 - Versionskontrolle (Commits), Kirks Sternentagebuch

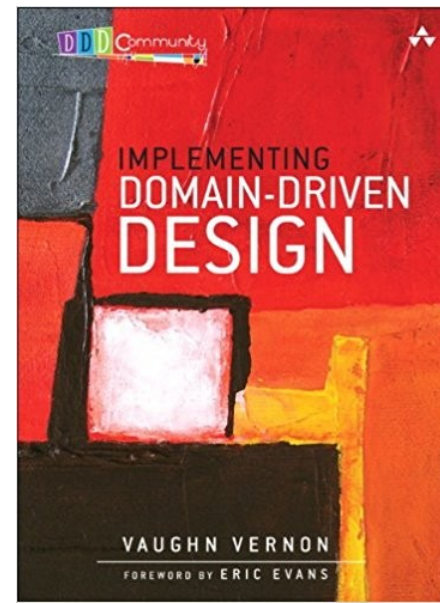
Domain Driven Design: Weiterführende Literatur



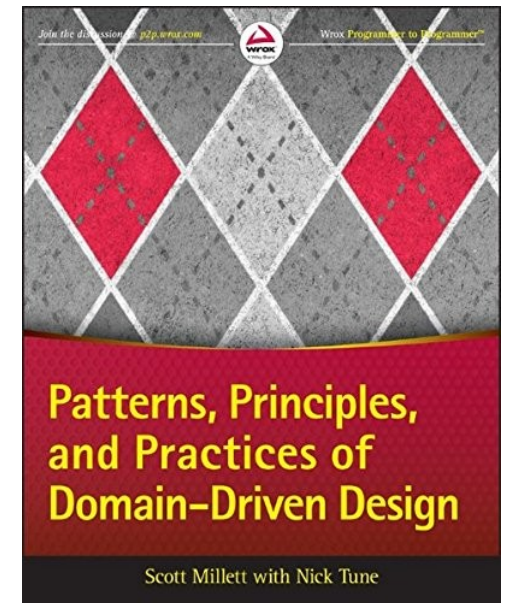
1996



2003



2013



2015

Gute Vorträge von Eric Evans

- What I've learned since the book (70 min)

<https://www.infoq.com/presentations/ddd-eric-evans>

- DDD and Microservices (55 min)

<https://www.infoq.com/presentations/ddd-microservices-2016>

- Good Design Is Imperfect Design (45 min)

<https://www.infoq.com/presentations/ddd-imperfect-design>

- Recovering the Ability to Design when Surrounded by Messy Legacy Systems (60 min)

<https://www.infoq.com/presentations/Strategy-Messy-Legacy-Systems>