

# Domain-Driven Design

# Agenda

- Vorstellung
- Organisatorische Rahmenbedingungen / Zeitplanung
- Inhalt
  - Definition
  - Domäne, Modell und Ubiquitous Language
  - Der Weg zum Modell
  - Vom Modell zur Implementierung
  - Das Modell in der Anwendungsarchitektur
  - Modelle schneiden und voneinander abgrenzen
  - Lokale Modellkonsistenz wahren
- Übungen

# Zeitplanung

## Tag 1:

09:00 – 10:15	Vortrag
10:30 – 12:00	Vortrag
12:00 – 13:00	Mittagspause
13:00 – 14:00	Vortrag
14:15 – 15:15	Vortrag
15:30 – 16:30	Vortrag

## Ziele:

- Überblick über Domain Driven Design
- Verstehen der Grundkonzepte
- Grundbegriffe zuordnen können
- Mitreden können

## Tag 2:

09:00 – 10:15	Vortrag
10:30 – 12:00	Vortrag
12:00 – 13:00	Mittagspause
13:00 – 14:00	Vortrag
14:15 – 15:15	Übung
15:30 – 16:30	Übung

## Nicht Ziel:

- Selbstständige Umsetzung der Methoden und Tools

- Domain-Driven Design (DDD) ist eine Herangehensweise an die Modellierung komplexer objektorientierter Software.
- Die Modellierung der Software wird dabei maßgeblich von den umzusetzenden Fachlichkeiten der Anwendungsdomäne beeinflusst.
- Domain-Driven Design ist nicht nur eine Technik oder Methode. Es ist viel mehr eine Denkweise und Priorisierung zur Steigerung der Produktivität von Softwareprojekten im Umfeld komplexer fachlicher Zusammenhänge
- Der Begriff „Domain-Driven Design“ wurde von Eric Evans in seinem gleichnamigen Buch geprägt.

Domain-Driven Design basiert auf folgenden zwei Annahmen:

- Der Schwerpunkt des Softwaredesigns liegt auf der Fachlichkeit und der Fachlogik.
- Der Entwurf komplexer fachlicher Zusammenhänge sollte auf einem Modell der Anwendungsdomäne, dem Domänenmodell basieren.

Domain-Driven Design ist an keinen bestimmten Softwareentwicklungsprozess gebunden, orientiert sich aber an agiler Softwareentwicklung. Insbesondere setzt es iterative Softwareentwicklung und eine enge Zusammenarbeit zwischen Entwicklern und Fachexperten voraus

- DDD hilft beim Entwerfen und Implementieren von hochwertiger Software, das sowohl auf strategischer wie auch auf taktischer Ebene
- Die Organisation wird am meisten von Softwaremodellen profitieren, die ausdrücklich ihre Kernkompetenzen reflektiert
- Wird verwendet um effektive Softwareentwürfe und -implementierungen abzuliefern, wie sie in der heutigen Geschäftswelt gebraucht werden
- DDD gehört in das Softwareentwicklungsteam

## DDD ist schwergewichtig?

- Kompliziert und schwergewichtig?
- Besteht es aus einer Reihe von fortgeschrittenen Techniken
- DDD ist mächtig und man muss viel lernen
- Deshalb kann es entmutigend sein, DDD ohne Hilfe einzusetzen
- Die gute Nachricht ist: DDD muss nicht wehtun. DDD hilft, die Komplexität einfacher zu beherrschen



- Oft wird über gutes und schlechtes Design gesprochen
- Viele Entwicklungsteams verschwenden keinen Gedanken an Design
- Stattdessen wird oft „Taskboard-Lotterie“ gespielt
- Die heftigste Auswirkung hat dieses nicht vorhandene Design auf das Business
- Der Rest wird Programmier-Heldentaten überlassen
- Das Ergebnis ist selten so gut, wie es sein könnte.

## Gängige Geschäftsprobleme

- Softwareentwicklung wird als Kostenstelle und nicht als Profitcenter betrachtet
- Entwickler sind besessen von Technologie und davon, Probleme mit Technologie zu lösen, statt sie sorgfältig zu durchdenken
- Der Datenbank wird zu hohe Priorität zugesprochen
- Entwickler geben sich nicht genug Mühe, die Objekte und Operationen so zu benennen, dass die Namen zu den fachlichen Aufgaben passen

- Schätzungen über den Projektfortschritt werden zu oft verlangt
- Entwickler platzieren Fachlogik in falschen Komponenten
- Die Entwickler schaffen falsche Abstraktionen
- Services sind untereinander stark gekoppelt

*Die Frage, ob Design nötig oder bezahlbar ist, trifft nicht den Punkt. Design ist unumgänglich. Die Alternative zu gutem Design ist schlechtes Design, nicht überhaupt kein Design.*

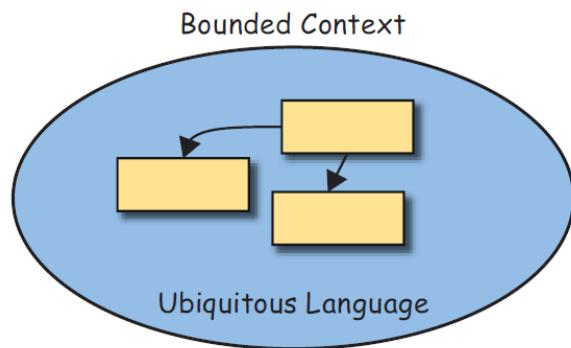
*A Practical Introduction von Douglas Martin [Martin 1990]*

## Effektives Design

*Die meisten Menschen machen den Fehler zu denken, dass es bei Design nur darum geht, wie es aussieht. Die Leute denken, es ist diese Fassade – dass man den Designern einen Kasten übergibt und sagt: »Macht den hübsch!« Das ist nicht unser Verständnis von Design. Es geht nicht nur darum, wie etwas aussieht und sich anfühlt. Design ist, wie etwas funktioniert.*

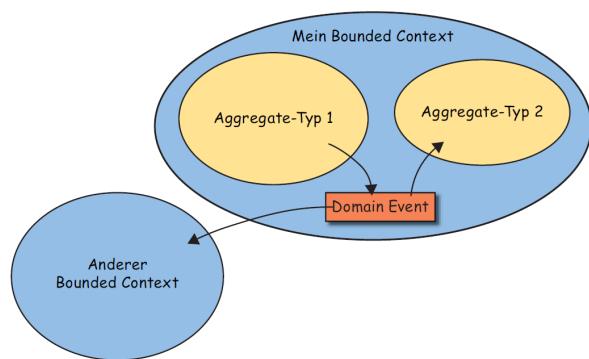
*Steve Jobs*

- Mit strategischem Design setzt man die groben Pinselstriche
- Teilt Arbeit anhand von Prioritäten auf
- Strategische Entwurfsmuster *Bounded Context*
- Wie *Ubiquitous Language* innerhalb eines expliziten *Bounded Context* entwickelt wird
- Wichtig ist es Entwickler und Domänenexperte mit einzubeziehen
- *Subdomains* helfen uns, mit der grenzenlosen Komplexität von Altsystemen umzugehen



## Taktisches Design

- Taktisches Design ist der dünne Pinselstrich
- *Entities* und *Value Objects* zusammenfassen – im Muster *Aggregate*
- Ziel ist, die Fachlichkeit so ausdrucksstark wie möglich zu modellieren
- Ereignisse werden an andere weitergegeben. An lokale *Bounded Context* oder fremde entfernte *Bounded Contexts*

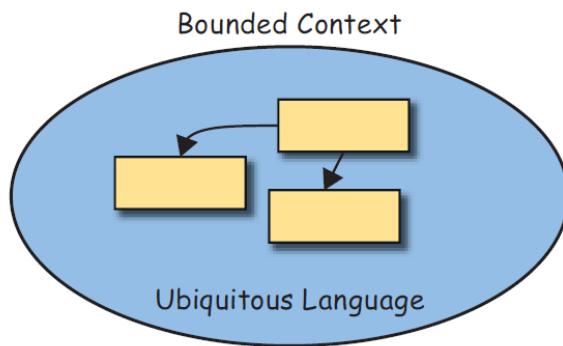


- Hilft, das Wissen zu vertiefen
- Lernprozess erfolgt durch Konversation und Ausprobieren
- Neu erworbene Wissen wird sich über das ganze Team verteilen
- Ziel ist es, so schnell wie möglich zu lernen und zu vertiefen



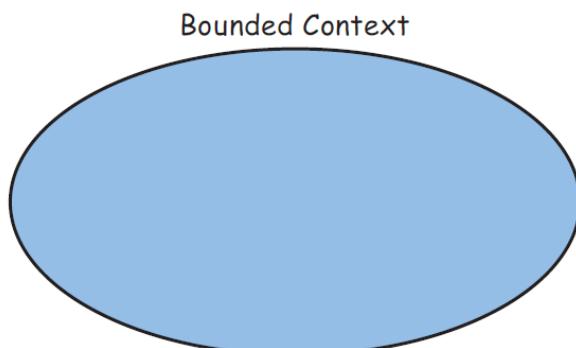
## STRATEGISCHES DESIGN MIT BOUNDED CONTEXTS UND DER UBIQUITOUS LANGUAGE

- *Bounded Contexts*
- *Ubiquitous Language*
- *Ubiquitous Language* in einem bestimmten *Bounded Context* modellieren



## Bounded Context

- *Bounded Context* eine semantische kontextuelle Grenze
- Innerhalb der Grenze hat jede Komponente des Softwaremodells eine bestimmte Bedeutung
- Komponenten sind kontextspezifisch und semantisch
- Problemraum als *Core Domain*
- *Bounded Context* als Herzstück Ihrer Unternehmensstrategie
- Integration mit anderen *Bounded Contexts*
- Beispiel: Order, Delivery and Billing



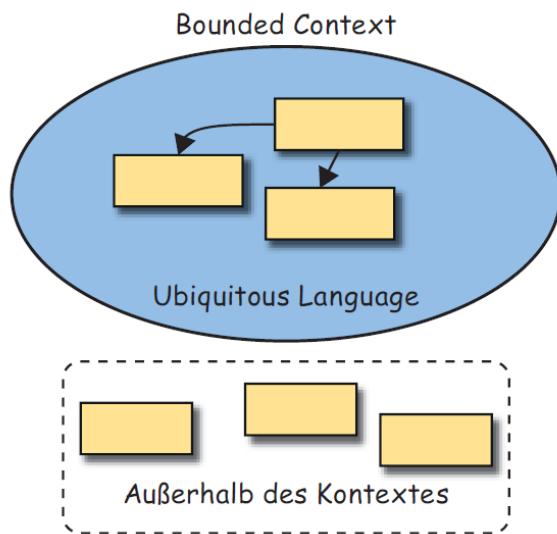
- Softwaremodell innerhalb der Kontextgrenze spiegelt eine Sprache wider
- Wird von jedem Mitglied des Teams gesprochen
- Gesprochen als auch im implementiert
- Äußerst präzise sein
  - Strikt
  - Genau
  - Durchgängig und
  - Knapp
- Beispiel: Länder und Sprachgrenzen



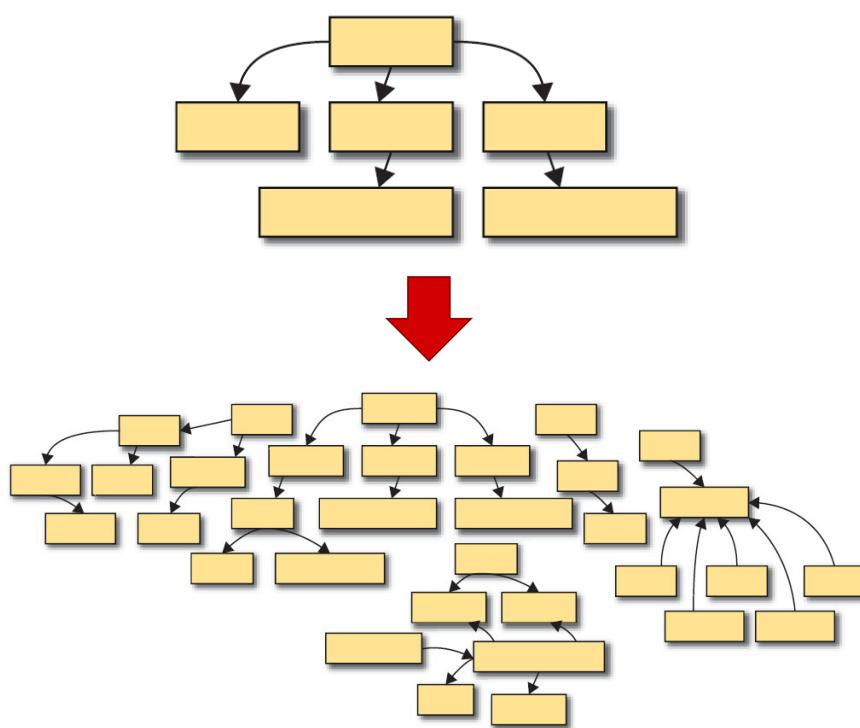
## Bounded Contexts, Teams und Quellcode-Repositories

- Ein Team ist einem *Bounded Context* zur Bearbeitung zugewiesen
- Ein getrenntes Quellcode-Repository für jeden *Bounded Context*
- Es ist möglich, dass ein Team an mehreren *Bounded Contexts* arbeitet
- **Nicht** mehrere Teams an demselben *Bounded Context*
- Grenzen Sie Quelltext und Datenbankschema für jeden *Bounded Context* sauber von denen der anderen ab
  
- Dadurch werden unerwünschte Überraschungen vermieden
- Quelltext und Datenbank gehören immer einem Team
- Schnittstellen werden zwischen *Bounded Context* benutzt

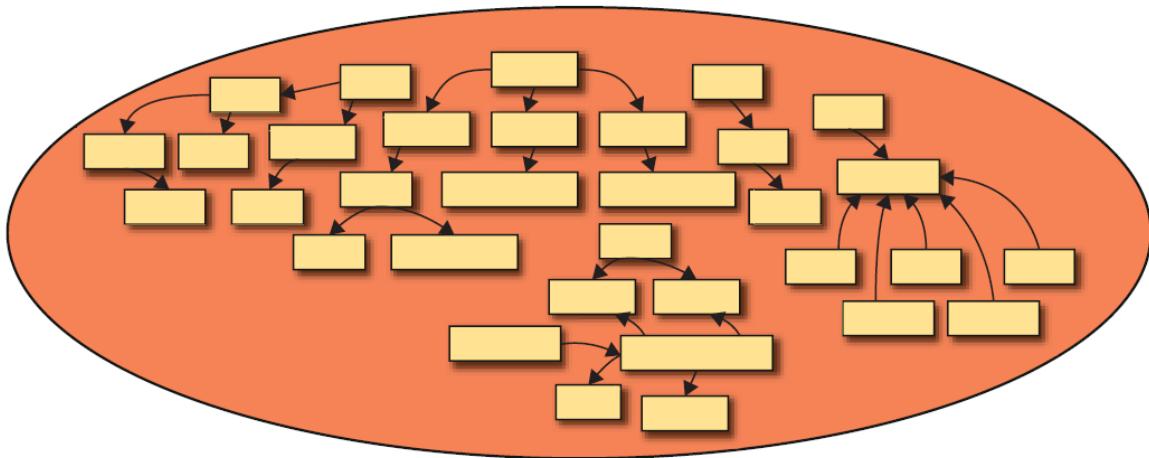
- In natürlichen Sprachen entwickelt sich Terminologie über die Zeit und über die Grenzen hinweg.
- Wörter können unterschiedliche Nuancen von Bedeutung annehmen.
- Definitionen sogar unterschiedlich (sei es geringfügig oder erheblich) von den Komponenten, die ein anderes Team modelliert.
- Das ist völlig in Ordnung!



## Gründe für Bounded Contexts

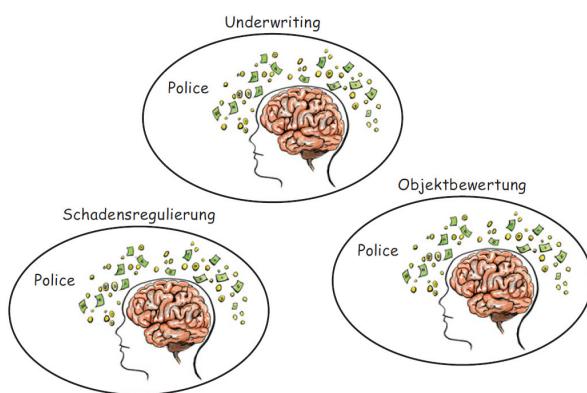


- *Big Ball of Mud*
- *Monolith*
- *Historisch gewachsen*

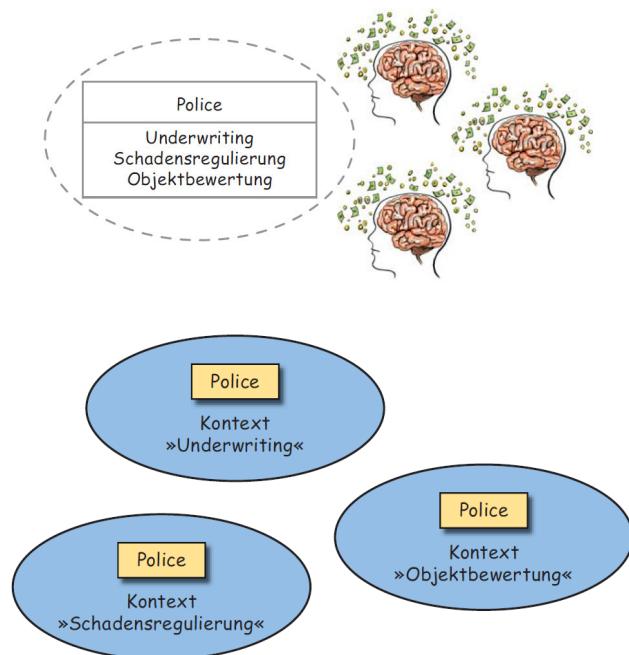


## Domain Experts und Geschäftstreiber

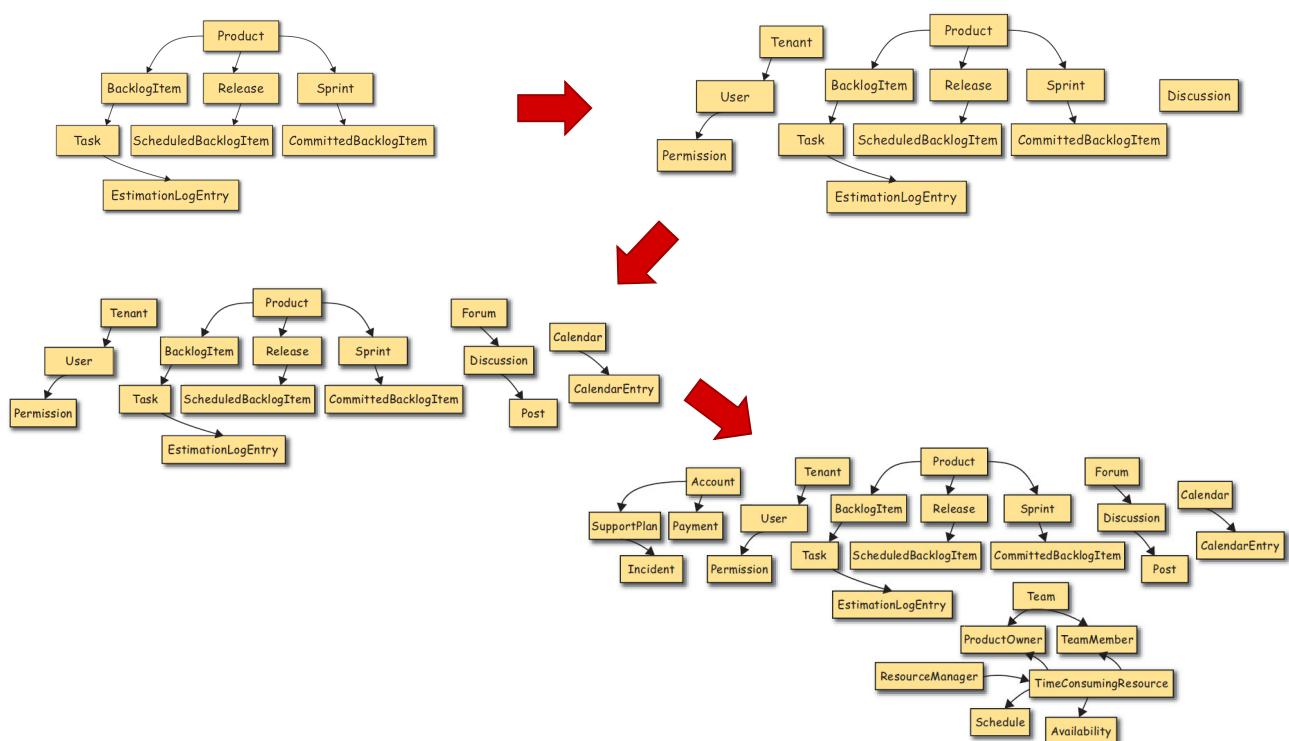
- Die Aufteilung der Organisation in Abteilungen oder Arbeitsgruppen kann eine gute Indikation dafür sein, wo Modellgrenzen gezogen werden sollten
- Trend, Personen nach Projekten aufzuteilen
- Gruppierungen nach Funktionen unter einem hierarchischen Management weniger populär
- Es ist zu empfehlen, über die Aufteilung nachdenken

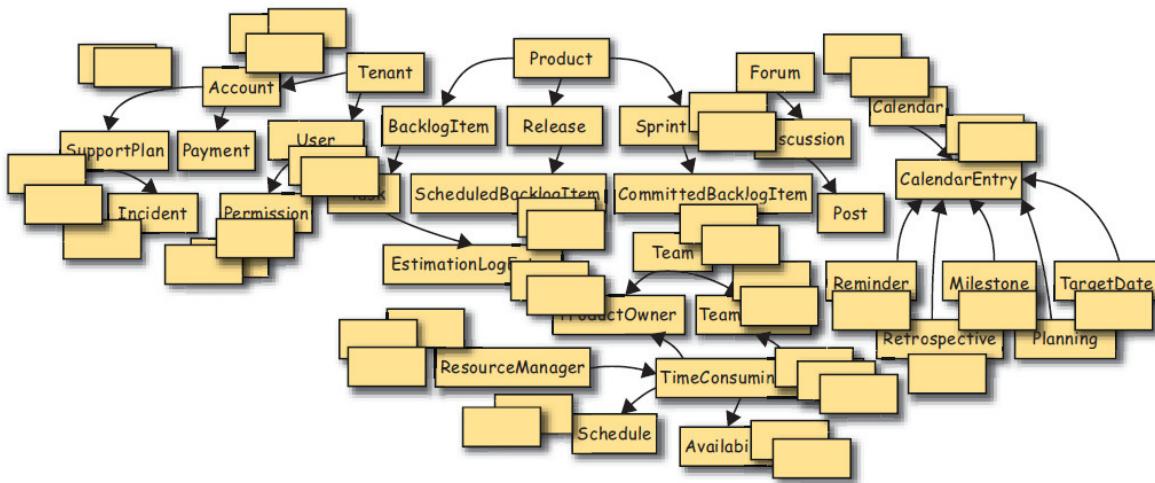


- **Police im Underwriting**
  - Bewertung des Risikos
  
- **Police in der Objektbewertung**
  - Immobilien, die versichert werden sollen, zu untersuchen und Gutachten dafür zu erstellen
  
- **Police in der Schadensregulierung**
  - Zahlungsantrag an den Versicherten aufgrund von Versicherungsbedingungen



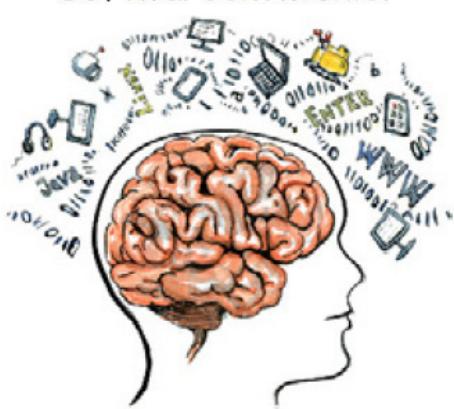
## Beispiel





Grundlegendes strategisches Design ist notwendig

Softwareentwickler

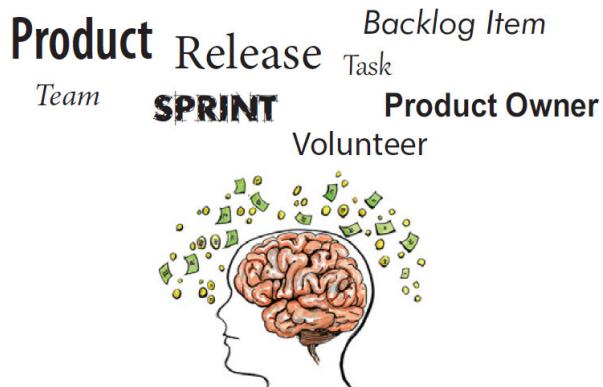


Domain Experts



Die Konzepte, die die strenge Anwendung des »Nur-Kern-Filters« überstehen, sind Teil der *Ubiquitous Language* des Teams.

- Product Owner im Sinne von Scrum und einem *Domain Expert* im Sinne von DDD ist
- Product Owner typischerweise stärker auf die Verwaltung und das Priorisieren des Product Backlog fokussiert
- *Domain Expert* typischerweise stärker in der Kernkompetenz des Unternehmens
- Stellen Sie sicher, dass Sie einen echten *Domain Expert* im Team haben

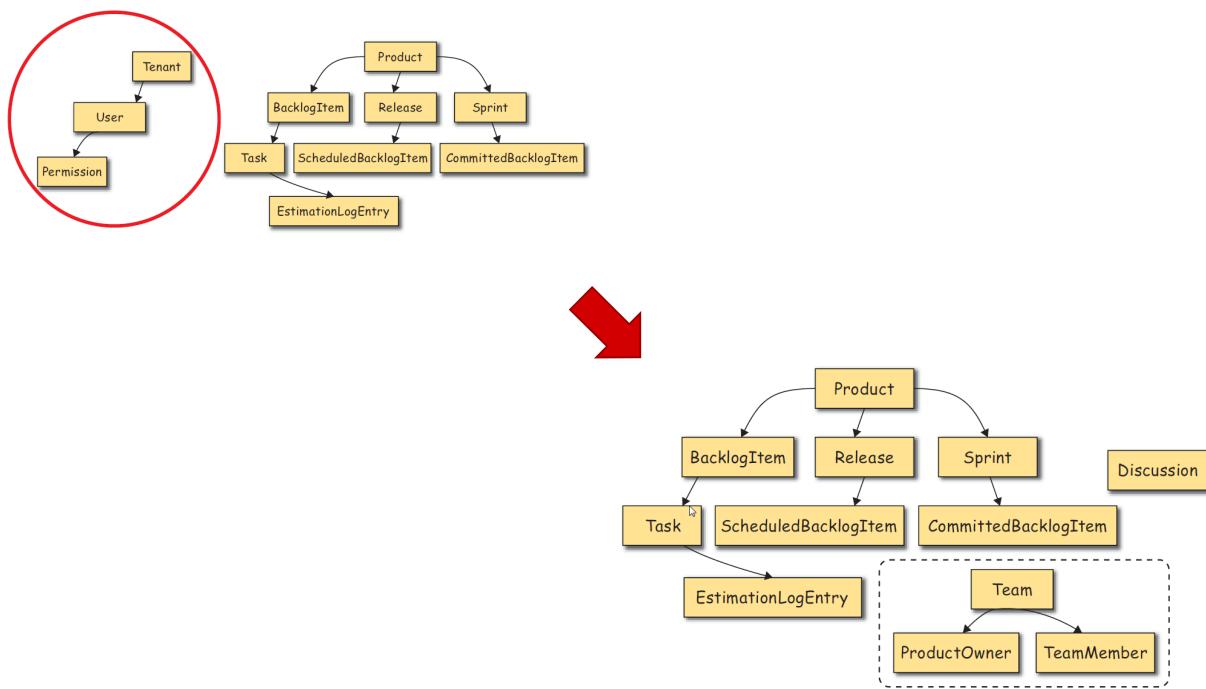


## Entwickler

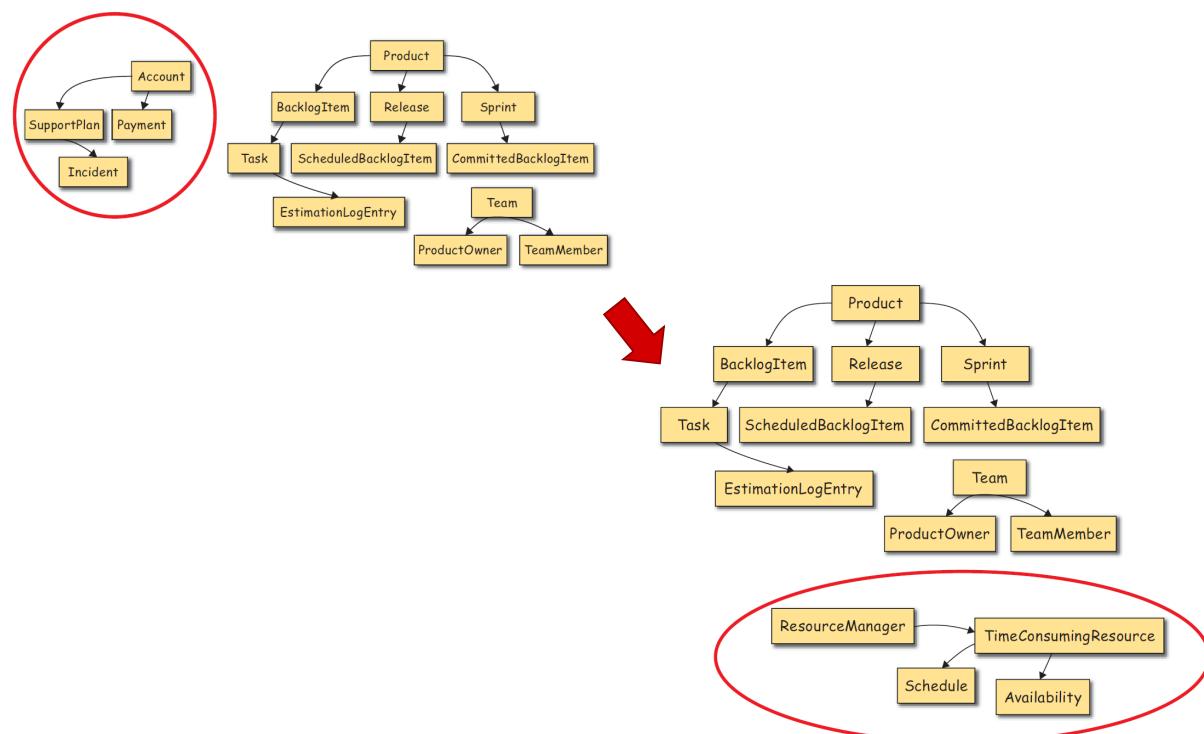
- Entwickler sind auf Softwareentwicklung fokussiert
- Entwickler, die in einem DDD-Projekt arbeiten, dürfen nicht so technikzentriert sein
- Sollten unangebrachte Reduktion auf Technik zurückweisen
- Fokussieren Sie auf die fachliche Komplexität, nicht auf die technische

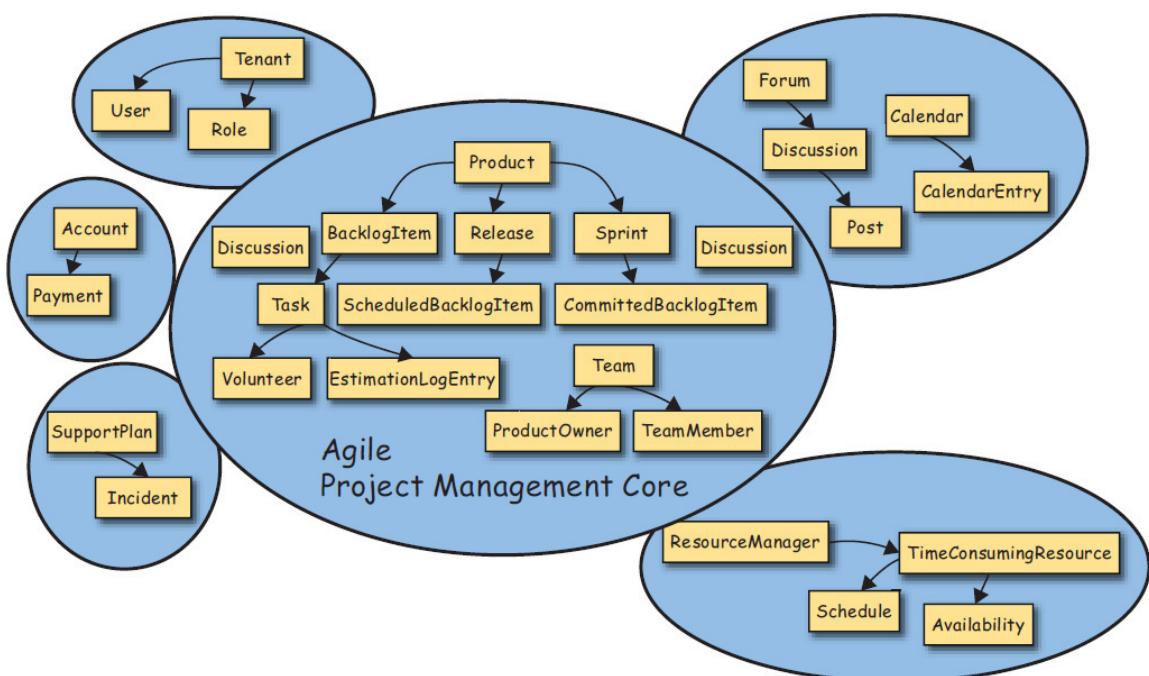
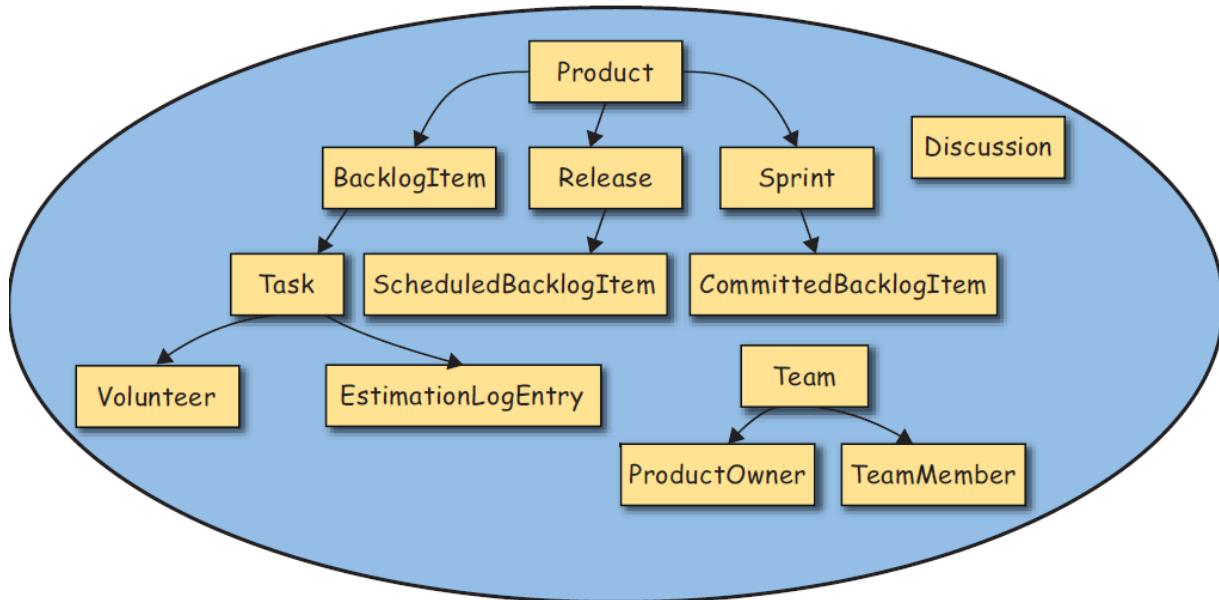
```
while (a < b) { 0xFB249E7
    a += c;      C# Scala Java
}           JavaScript PHP BPM
          10110001101010110001 BPEL
```





# Infragestellen und Vereinheitlichen





- Oft ist es sinnvoll, ein paar *Event-Storming*-Sessions durchzuführen, wenn man an Szenarios arbeitet
- Diese können dabei helfen, schnell zu verstehen, an welchen Szenarios man arbeiten sollte
- Umgekehrt gibt das Entwickeln von konkreten Szenarios Ihnen eine bessere Vorstellung davon, in welche Richtung Sie Ihre *Event-Storming*-Sessions lenken sollten
- DDD-Werkzeuge wie *Event Storming* und Szenarios arbeiten gut zusammen

## Szenario

### Szenario:

- Der Product Owner ordnet ein Backlog Item einem Sprint zu
- Gegeben ist ein Backlog Item, das für das nächste Release geplant ist
- Und der Product Owner des Backlog Item
- Und ein Sprint, dem zugeordnet werden kann
- Und ein Quorum von Team Members, die dem Backlog Item zustimmen
- Wenn der Product Owner das Backlog Item dem Sprint zuordnet
- Dann wird das Backlog Item dem Sprint zugeordnet
- Und das Ereignis “Backlog Item Committed“ wird erzeugt

```
[Test]
public void ShouldCommitBacklogItemToSprint()
{
    // Given
    var backlogItem = BacklogItemScheduledForRelease();
    var productOwner = ProductOwnerOf(backlogItem);
    var sprint = SprintForCommitment();
    var quorum = QuorumOfTeamApproval(backlogItem, sprint);

    // When
    backlogItem.CommitTo(sprint, productOwner, quorum);

    // Then
    Assert.IsTrue(backlogItem.IsCommitted());

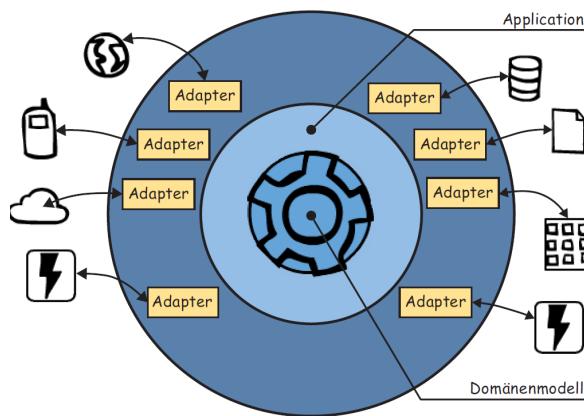
    var backlogItemCommitted =
        backlogItem.Events.OfType<BacklogItemCommitted>().SingleOrDefault();

    Assert.IsNotNull(backlogItemCommitted);
```

## Auf lange Sicht

- Lernen und Wissenserwerb findet über einen langen Zeitraum statt
- Manchmal erfolgen das beste Lernen und der beste Wissenserwerb erst in der Phase der sogenannten „Wartung“
- Teams machen einen Fehler, wenn sie glauben, dass Innovation dort endet
- Schlimmste, was einer *Core Domain* passieren kann, ist, dass das Label „Wartungsphase“ aufgeklebt wird

- Obwohl Technologie quer über Ihre Architektur verteilt sein wird, sollte das Domänenmodell frei von Technologie sein
- Das ist einer der Gründe, warum Transaktionen von den Application Services und nicht vom Domänenmodell verwaltet werden



Neben *Ports and Adapters* kann man DDD mit jeder der folgenden Architekturen oder Architekturmuster (und anderen) umsetzen

- Ereignisgetriebene Architektur
- Command Query Responsibility Segregation
- Reactive und Aktorenmodell
- Representational State Transfer
- Serviceorientierte Architektur
- Microservices
- Cloud Computing

- Einige Autoren betrachten einen Microservice als etwas viel Kleineres als einen *Bounded Context*
- Nach dieser Definition modelliert ein Microservice nur ein Konzept und verwaltet eine schlanke Menge von Daten
- Ein Beispiel eines solchen Microservice ist Product und ein anderes ist BacklogItem
- Wenn dies die Granularität ist, die man für einen Microservice als angemessen betrachtet, muss man verstehen, dass diese ein einem größeren logischen *Bounded Context* liegen werden

## Zusammenfassung

In diesem Teil haben Sie Folgendes kennengelernt:

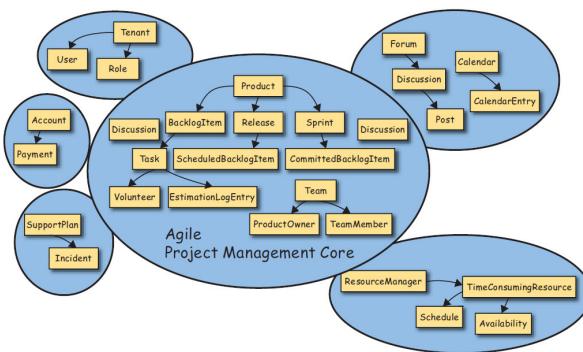
- Einige der gefährlichsten Fallstricke, wenn man zu viel in ein Modell steckt und einen *Big Ball of Mud* erzeugt
- Wie strategisches Design mit DDD angewendet wird
- Wie *Bounded Contexts* und *Ubiquitous Language* verwendet werden
- Wie man seine Annahmen infrage stellt und mentale Modelle vereinheitlicht
- Wie man eine *Ubiquitous Language* entwickelt
- Einiges über Architekturkomponenten, die man innerhalb eines *Bounded Context* findet
- Dass DDD nicht zu schwierig ist, um es selbst in die Praxis umzusetzen

# 3

## STRATEGISCHES DESIGN MIT SUBDOMAINS ODER CONTEXT MAPPING

## Strategisches Design mit Subdomains

- Einer der *Bounded Contexts* wird die *Core Domain* (dt.: *Kerndomäne*) sein
- Verschiedene Modelle nach ihrer spezifischen *Ubiquitous Language* aufzuteilen
- *Agile Project Management Core* ist sowohl ein sauberer *Bounded Context* als auch eine saubere *Subdomain*
- In einigen Situationen kann es mehrere *Subdomains* in einem *Bounded Context* geben



- Eine *Subdomain* ein Teil Ihrer das ganze Unternehmen umfassenden Geschäftsdomäne
- *Subdomain* als Repräsentation eines einzigen logischen Domänenmodells vorstellen
- Geschäftsdomänen sind zu groß und zu komplex, um sie als Ganzes zu betrachten
- Deshalb beschäftigen wir uns normalerweise nur mit den *Subdomains*, die wir innerhalb eines einzelnen Projektes brauchen
- Eine *Subdomain* bietet eine Lösung für ein Kerngebiet Ihres Geschäfts
- Wenn DDD richtig eingesetzt wurde, um eine *Subdomain* zu entwickeln, dann sollte sie als wohlgeformter *Bounded Context* umgesetzt worden sein.

## Arten von Subdomains

### *Core Domain*

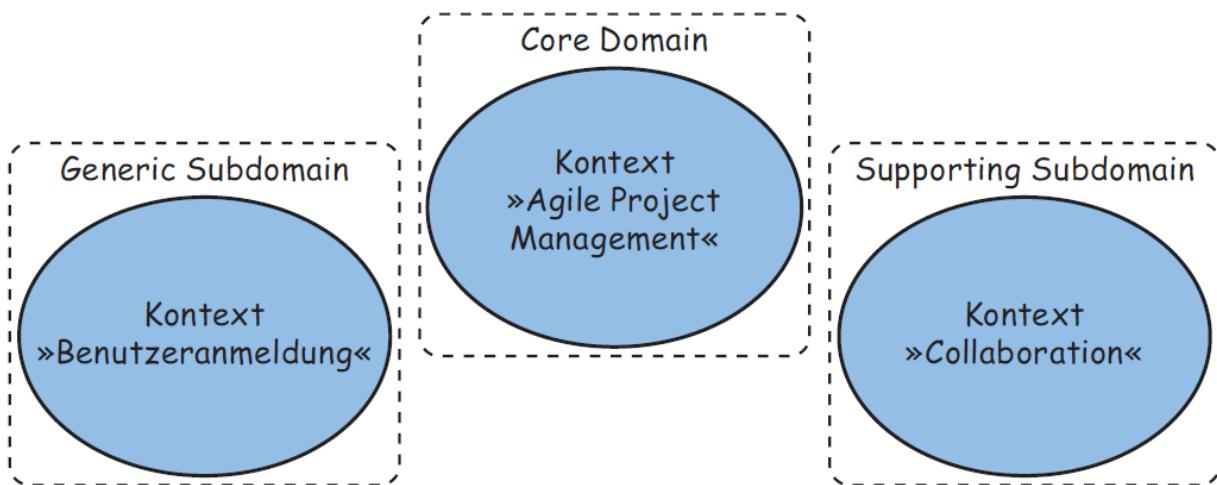
- Bei der *Core Domain* muss man sich für eine strategische Investition in ein einzelnes, wohldefiniertes Domänenmodell entscheiden.
- In diesem Bereich muss die Organisation am großzügigsten in Software investieren.

### *Supporting Subdomain*

- Für eine *Supporting Subdomain* müssen das Modell und die Software individuell entwickelt werden, weil es dafür keine Standardlösung gibt
- *Supporting Subdomains* sind wichtige Softwaremodelle, denn ohne sie kann die *Core Domain* keinen Erfolg haben

### *Generic Subdomain*

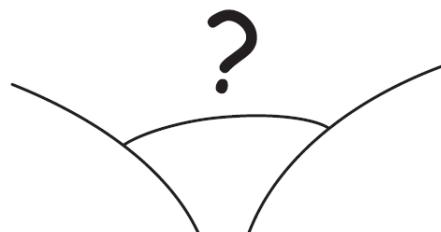
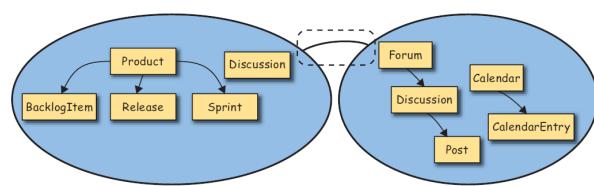
- Lösungen für *Generic Subdomains* kann man wahrscheinlich von der Stange kaufen
- Alternativ kann man sie per Outsourcing oder auch intern entwickeln lassen.



Wenn man DDD verwendet, soll ein *Bounded Context* sich eins zu eins (1:1) auf eine einzelne *Subdomain* abbilden lassen

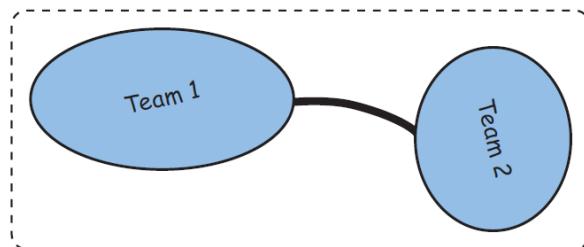
## Strategisches Design mit Context Mapping

- Das *Context Mapping* ist in der Abbildung mit dem gestrichelten Kasten hervorgehoben
- Der Strich zeigt an, dass die beiden *Bounded Contexts* in irgendeiner Weise aufeinander abgebildet werden
- Dynamik zwischen den beiden Teams und eine Integration zwischen den beiden *Bounded Contexts*



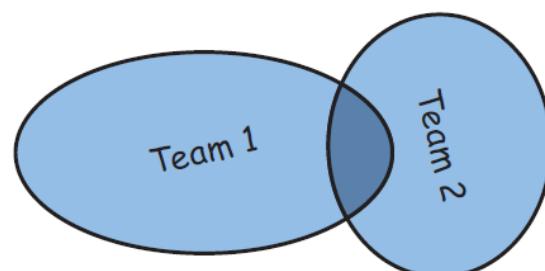
Arten von Mappings

- Besteht zwischen zwei Teams, die jeweils für einen *Bounded Context* verantwortlich sind
- Einführen um zwei Teams mit voneinander abhängigen Zielen zu koordinieren
- Beiden Teams haben zusammen Erfolg (oder Misserfolg)
- *Partnership* sollte nur so lange bestehen, wie sie einen Vorteil bietet

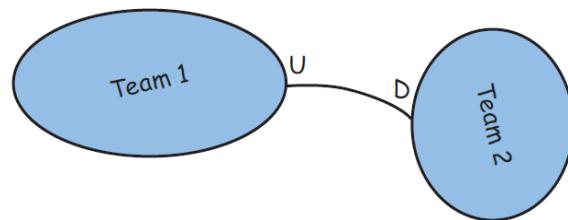


# Shared Kernel

- Schnittmenge von zwei *Bounded Contexts* dargestellt, beschreibt die Beziehung zwischen zwei (oder mehr) Teams, die sich ein kleines, aber gemeinsames Modell teilen
- *Shared Kernel* ist oft schwer zu erzeugen und schwierig zu pflegen, weil man eine offene Kommunikation zwischen den Teams und eine dauerhafte Einigkeit darüber erreichen muss
- Teilung:
  - Pflege von Code, Build und Test
  - Modell



- Beziehung zwischen zwei *Bounded Contexts*
- *Supplier* vorgeschaltet (upstream)
- *Customer* nachgeschaltet (downstream)
- Beziehung hängt vom *Supplier* ab, weil er zur Verfügung stellen muss, was der *Customer* braucht
- *Supplier bestimmt*, was der *Customer* bekommen wird und wann
- Praktische Beziehung

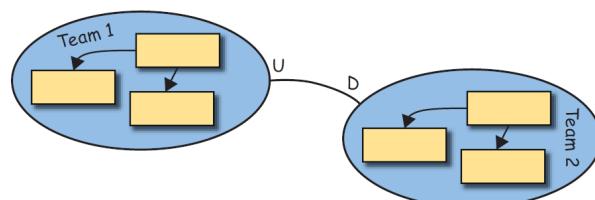


## Conformist

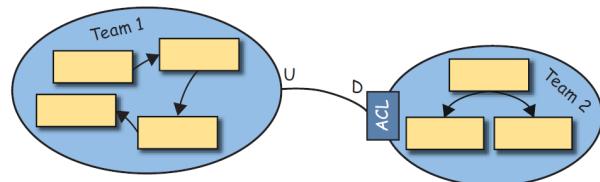
- besteht, wenn es ein Upstream- und ein Downstream-Team gibt, und das Upstream-Team keine Motivation hat, die Anforderungen des Downstream-Teams zu unterstützen

Beispiel:

- Wenn Sie Amazon-Partner werden wollen, werden Sie sich dem Amazon-Modell anpassen müssen, um sich zu integrieren

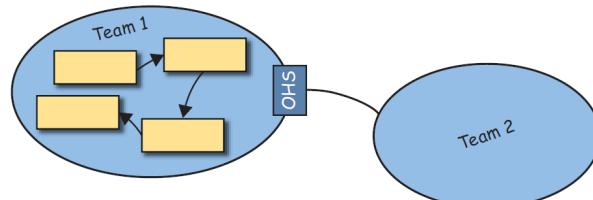


- Ein *Anticorruption Layer* (ACL), ist die defensivste Art einer *Context-Mapping*-Beziehung
- Downstream-Team eine Übersetzungsschicht zwischen seiner *Ubiquitous Language* (Modell) und der *Ubiquitous Language* (Modell) des Upstream-Teams
- Möglichkeit, Integration Modellkonzepte zu bauen, die exakt die eigenen Anforderungen befriedigen und von fremden Konzepten isoliert sind

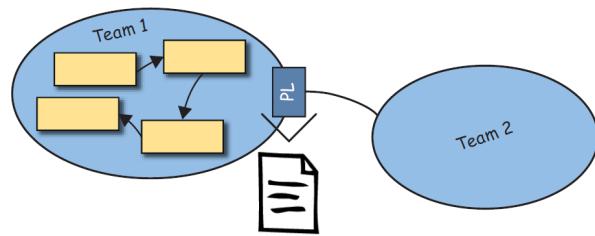


## Open Host Service

- Ein Protokoll oder eine Schnittstelle, das den Zugriff zu einem *Bounded Context* in Form einer Menge von Services bietet
- Das Protokoll ist „offen“, dass jeder relativ leicht mit ihm interagieren kann
- Wohldokumentierte Services
- Angenehmer, ein *Conformist* dieses Modell zu sein als ein *Conformist* von vielen Altsystemen

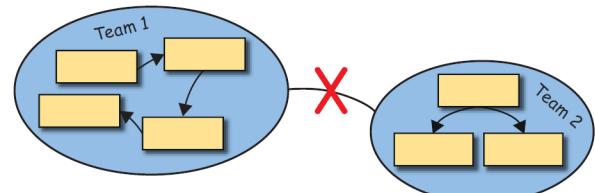


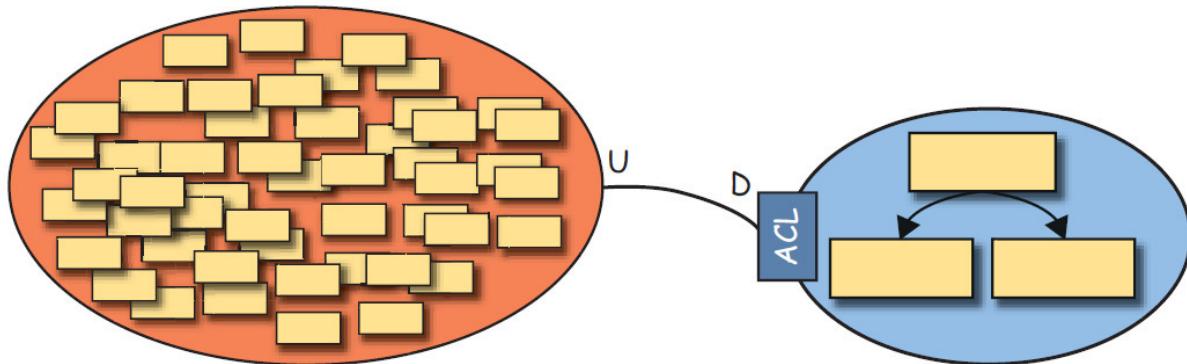
- Wohldokumentierte Sprache zum Informationsaustausch
- Konsumenten, die sowohl lesen als auch schreiben, können sich darauf verlassen, dass ihre Integration korrekt ist
- *Published Language* kann in XML Schema, JSON Schema oder einem platzsparenden Format wie Protobuf oder Avro definiert werden
- Oft bietet ein *Open Host Service* eine *Published Language* als Schnittstelle



## Separate Ways

- Beschreibt eine Situation, in der sich kein signifikanter Nutzen einer Integration erkennen lässt
- Gesuchte Funktionalität von keiner *Ubiquitous Language* komplett angeboten
- Eigene spezialisierte Lösung wird erstellt





## Context Mapping richtig nutzen

Welche Art von Schnittstellen sollen angeboten werden um den Zugriff auf den Bounded Context zu ermöglichen?

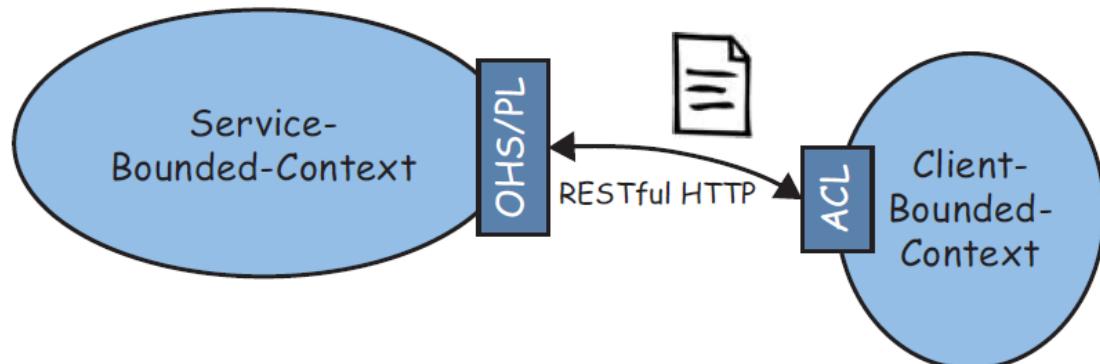
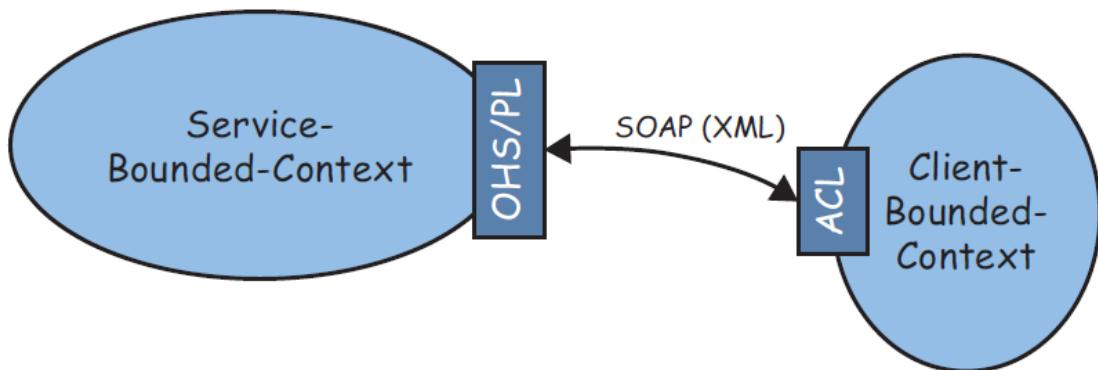
Das hängt davon ab, was das Team anbietet:

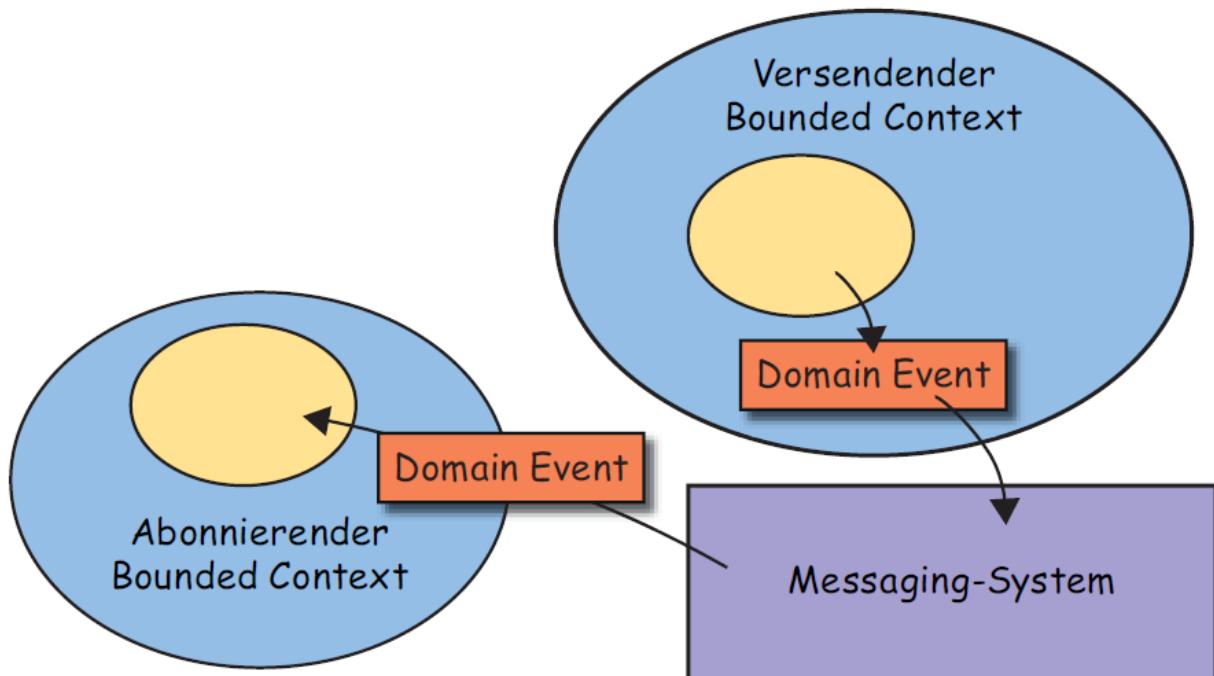
- RPC über SOAP
- RESTful API
- Messaging-System

Im schlechtesten Fall:

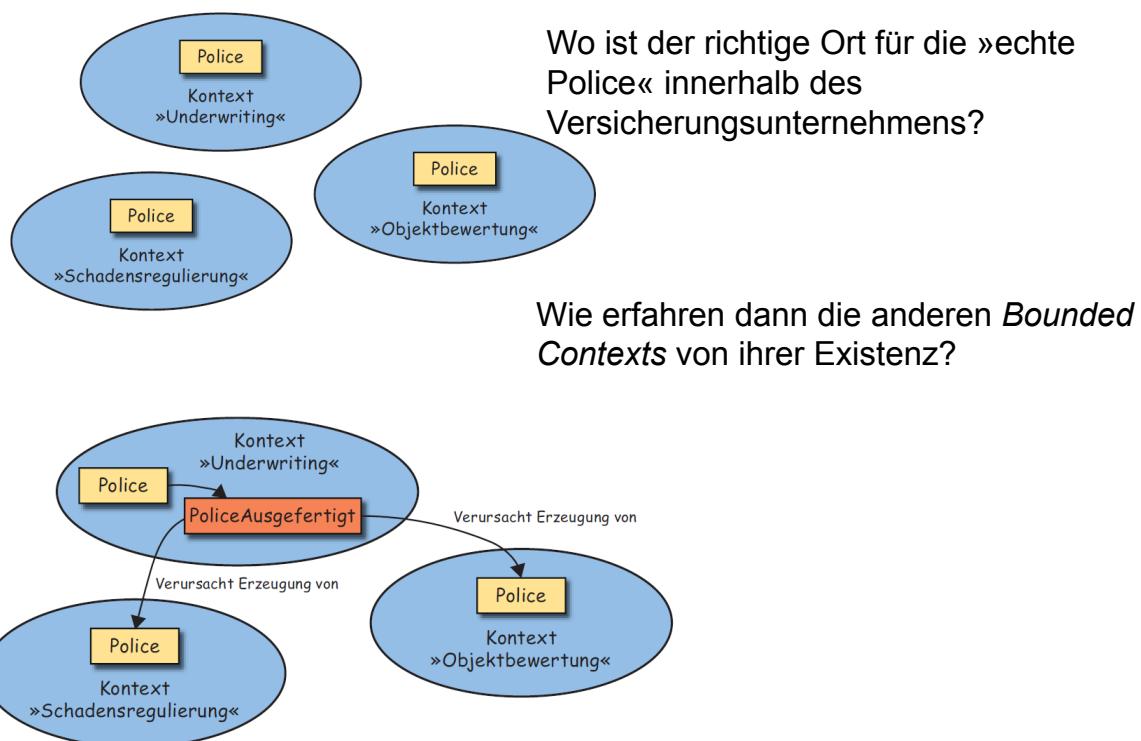
- Datenbankintegration
- Dateisystemintegration

Hier muss Ihr Modell durch einen ACL geschützt sein.

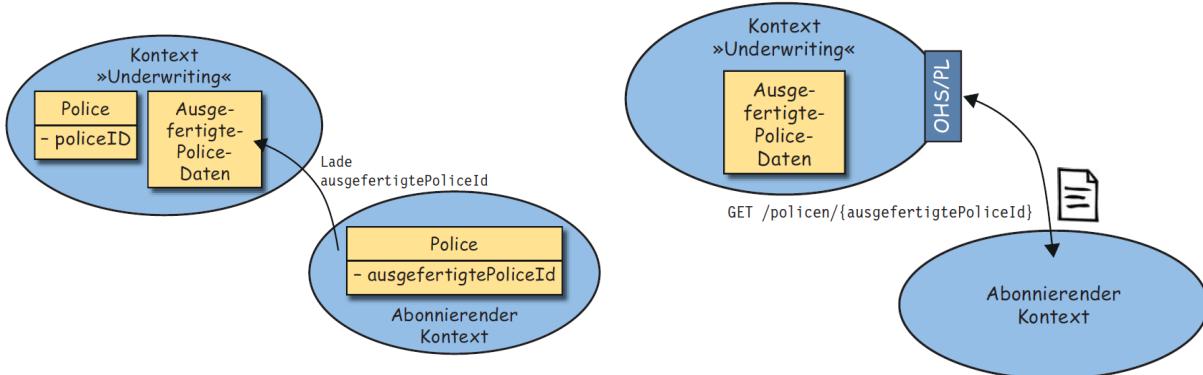




## Context Mapping am Beispiel



## Abwägung zwischen Anreichern und Nachladen



Wann das eine oder das andere in Betracht ziehen?

## Zusammenfassung

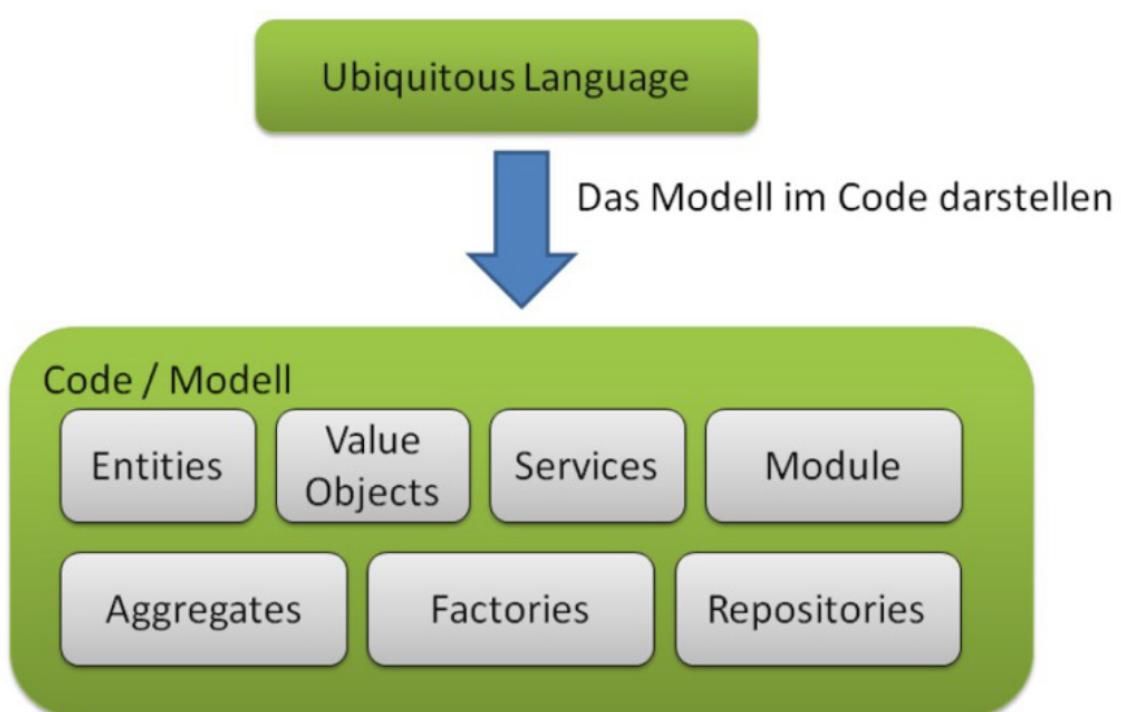
In diesem Teil haben Sie Folgendes kennengelernt:

- Wie die verschiedenen Arten von *Context-Mapping*-Beziehungen, wie *Partnership*, *Customer-Supplier* und *Anticorruption Layer*, aussehen
- Wie man *Context-Mapping*-Integration mit RPC, RESTful HTTP und Messaging verwendet
- Wie *Domain Events* mit Messaging funktionieren
- Eine Grundlage, auf der Sie Ihre *Context-Mapping*-Erfahrung ausbauen können

4

## TAKTISCHES DESIGN MIT AGGREGATES

Domänen Objekte



- Die wichtigste Unterscheidung die Evans zwischen Modellelementen macht, ist ob ein Element eine Identität hat oder nicht
- Objekte müssen beispielsweise identifizierbar sein, um sie von einander unterscheiden zu können oder zu einem späteren Zeitpunkt wieder zu finden um weitere Operationen mit diesen durchführen zu können (Personen, Events, Konto, ...)
- Andere Objekte stellen nur die Repräsentation einer Eigenschaft dar (Farben, Tags, ...). Diese sind definiert durch alle Eigenschaften

## Services

- Nicht an das Objekt gebundene Funktionen oder Handling von mehreren Objekten

Beispiel:

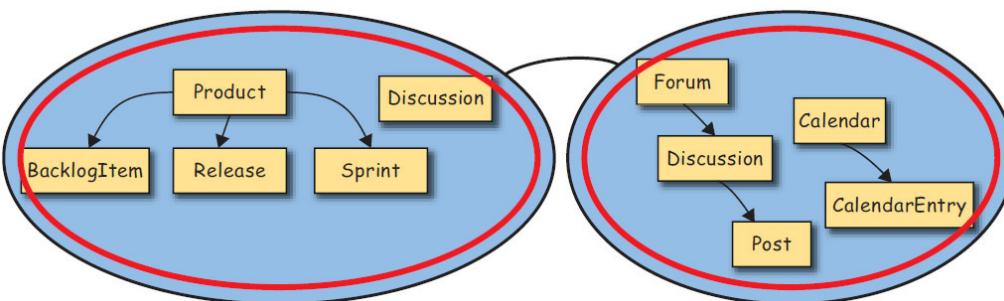
- Geokoordinaten-Ermittlung für Adresse oder
- Überweisung zwischen zwei Konten

- Technische Details (der Persistenz) sollen nicht in die UL eindringen
- Dafür wurden Repositories erschaffen



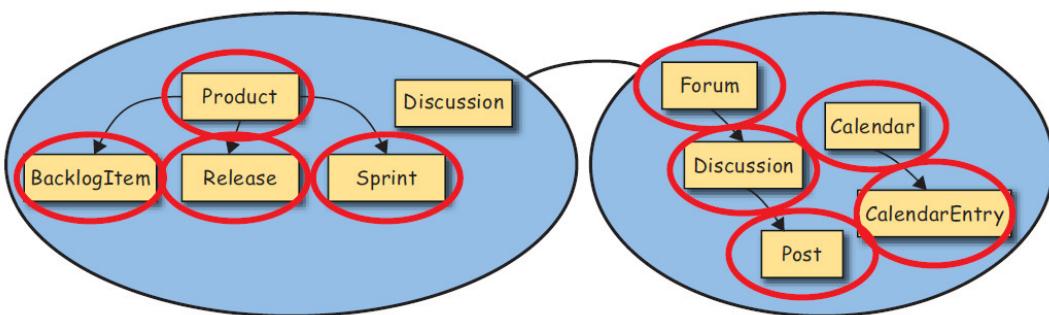
## Taktisches Design mit Aggregates

- Bisher haben wir strategisches Design mit *Bounded Contexts*, *Subdomains* und *Context Maps* diskutiert
- Die *Supporting Subdomain* stellt durch die *Context-Mapping*-Integration ein Werkzeug für die Zusammenarbeit dar
- Sie heißt deshalb *Collaboration*
- Was ist aber mit den Konzepten, die innerhalb eines *Bounded Context* existieren?

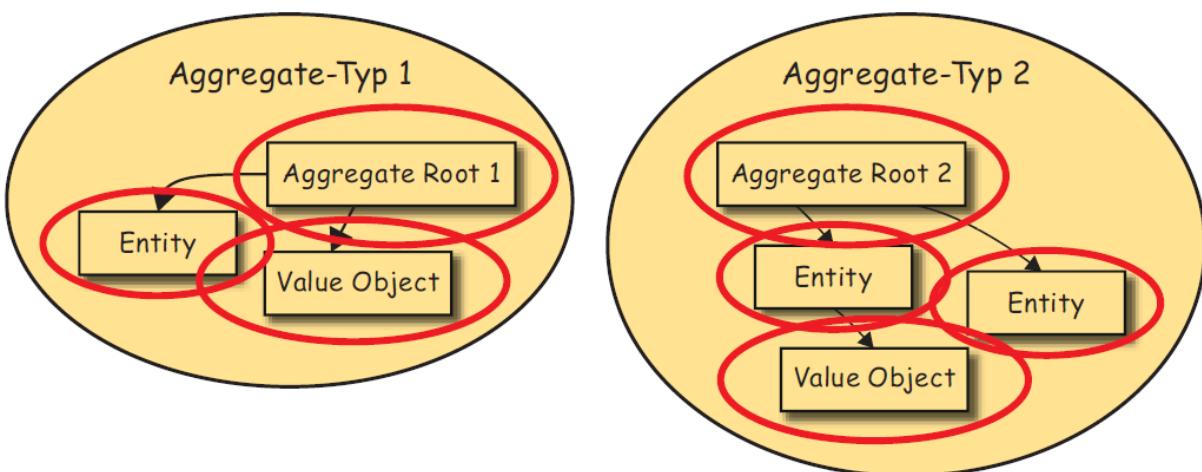


# Was ist ein Aggregate?

- Jedes der eingekreisten Konzepte, die in diesen beiden *Bounded Contexts* zu sehen sind, ist ein *Aggregate*
- Jedes *Aggregate* besteht aus einer oder mehreren *Entities*, wobei eine *Entity* als *Aggregate Root* bezeichnet wird
- *Aggregates* können auch *Value Objects* enthalten



## Aggregates



## Was ist eine Entity?

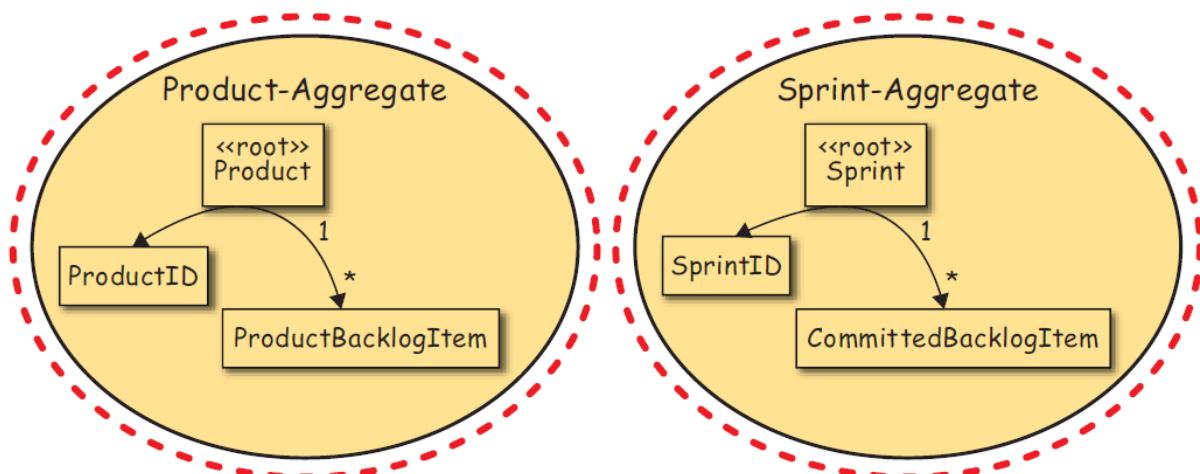
- Eine *Entity* (dt.: *Entität, Ding*) modelliert ein individuelles Ding
- Jede *Entity* hat eine eindeutige Identität
- Meistens wird eine *Entity* veränderlich sein
- Die Hauptsache, die eine *Entity* von anderen Modellierungswerkzeugen unterscheidet, ist ihre Einzigartigkeit

## Was ist ein Value Object?

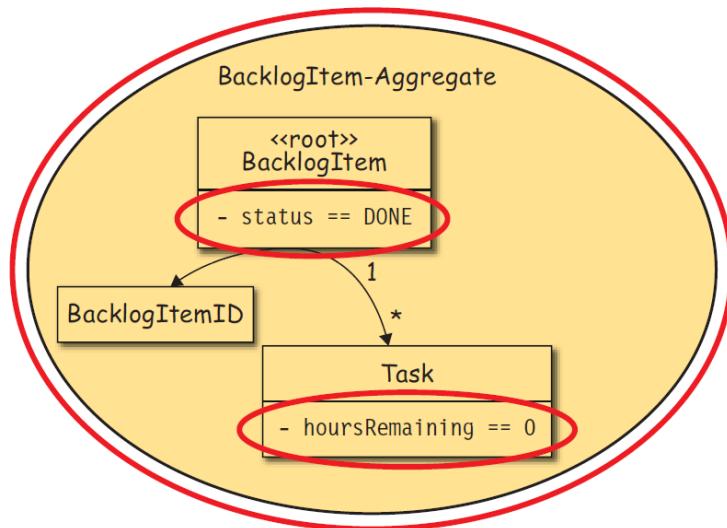
- Ein *Value Object* (dt.: *Wertobjekt, Fachwert*), oder einfach ein *Value* (dt.: *Wert*), modelliert ein unveränderliches konzeptionelles Ganzes
- Im Gegensatz zu einer *Entity* besitzt es keine eindeutige Identität
- Darüber hinaus ist ein *Value Object* kein Ding, sondern wird häufig verwendet, um eine *Entity* zu beschreiben, zu quantifizieren oder zu messen

- Schütze fachliche Invarianten innerhalb von *Aggregate*-Grenzen
- Entwirf kleine *Aggregates*
- Referenziere andere *Aggregates* nur über ihre Identität
- Aktualisiere andere *Aggregates* unter Verwendung von Eventual Consistency

## Regel 1: Schütze fachliche Invarianten innerhalb von Aggregate-Grenzen

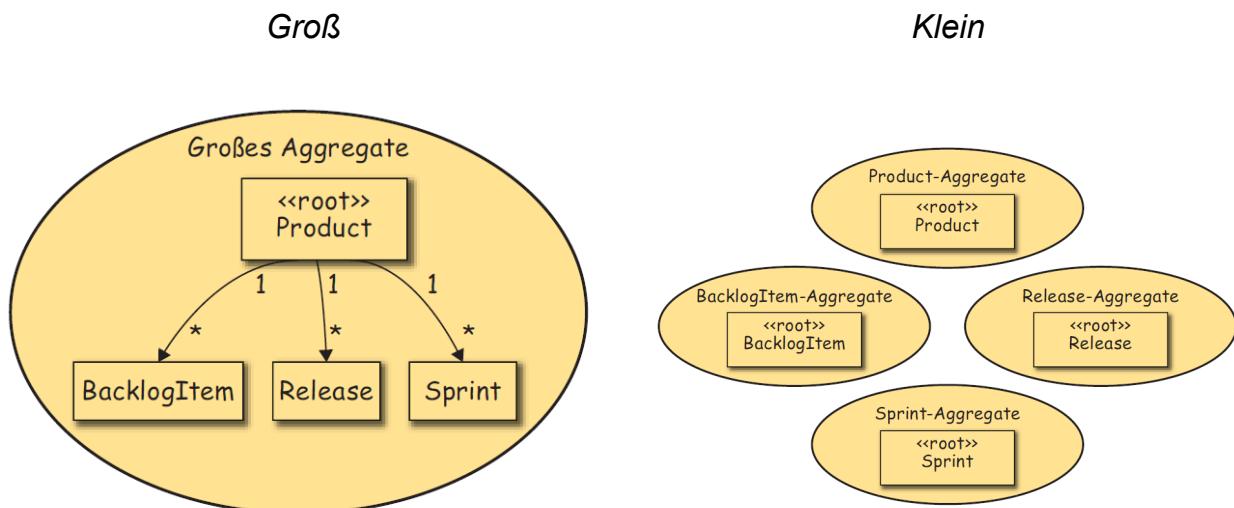


Regel 1 bedeutet, dass der Fachbereich die Zusammensetzung der *Aggregates* auf der Grundlage dessen bestimmen soll, was konsistent sein muss, wenn der Commit einer Transaktion ausgeführt wird.



„Wenn alle Exemplare von *Task* im Feld *hoursRemaining* einen Wert von 0 haben, muss der *status* im *BacklogItem* auf *DONE* gesetzt werden.“

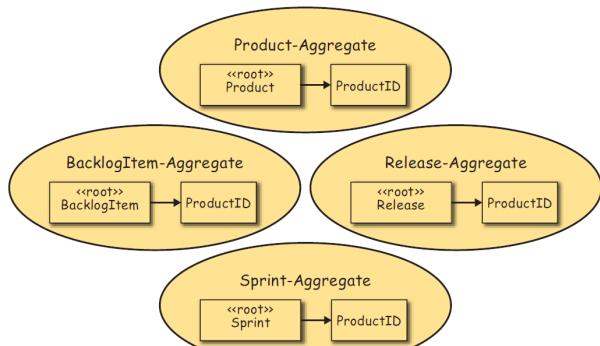
## Regel 2: Entwirf kleine Aggregates



*Folgt dem Single Responsibility Principle*

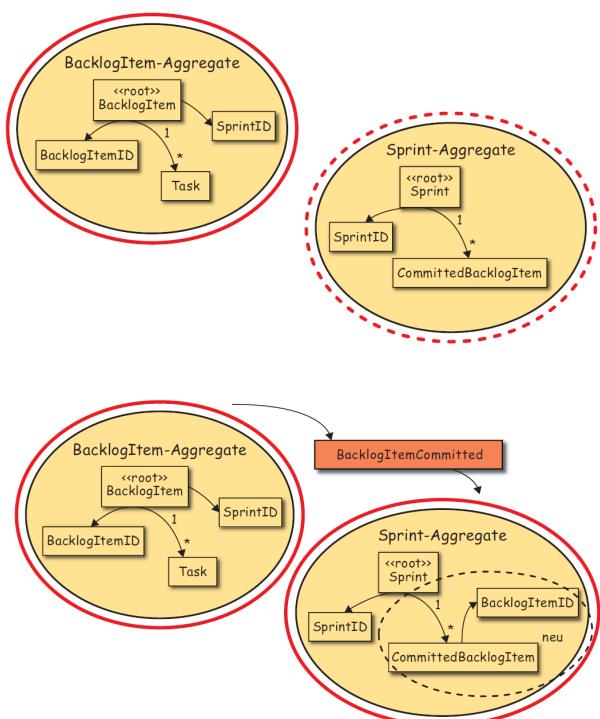
Regel 3: Referenziere andere Aggregates nur über ihre Identität

- In diesem Beispiel sehen wir, dass BacklogItem, Release und Sprint alle Product referenzieren, indem sie eine ProductId enthalten
  - Dies hilft, Aggregates klein zu halten
  - Schützt vor der Versuchung, mehrere Aggregates in der gleichen Transaktion zu ändern

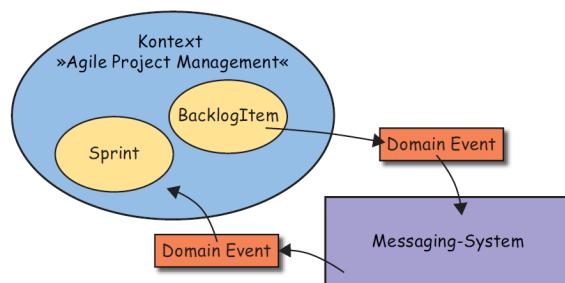


Regel 4: Aktualisiere andere Aggregates unter Verwendung von Eventual Consistency

- Wie stellen wir sicher, dass der Sprint auch mit der BacklogItemId des neu zugewiesenen BacklogItem aktualisiert wird?
  - Event BacklogItemCommitted erzeugt
  - Sprint wird geändert um BacklogItemId und BacklogItem
  - Sprint enthält BacklogItemId in CommittedBacklogItem-Entity



- *Domain Events* werden von einem *Aggregate* veröffentlicht und von einem sich dafür interessierenden *Bounded Context* abonniert
- Das Messaging-System liefert die *Domain Events* an Interessenten über einen Subscription-Mechanismus aus
- Der interessierte *Bounded Context* kann derselbe sein, der auch das *Domain Event* veröffentlicht hat, oder es können unterschiedliche *Bounded Contexts* sein



## Aggregates modellieren

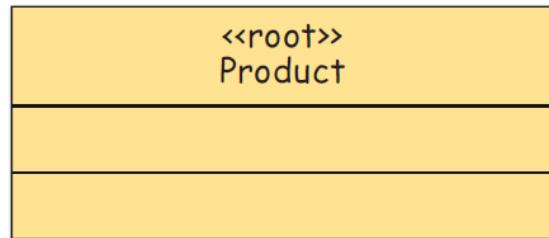
- Es gibt einige Fallstricke, die auf Sie warten, wenn Sie an Ihrem Domänenmodell arbeiten und Ihre *Aggregates* implementieren
- Ein großer, fieser Fallstrick ist das *Anemic Domain Model*
  - Alle *Aggregates* haben nur öffentliche Getters und Setters und keine wirkliche Geschäftslogik
  - Das passiert häufig, wenn während der Modellierung der Fokus mehr auf das Technische als das Fachliche gelegt wird
- Achten Sie auch auf Geschäftslogik, die in die *Application Services* oberhalb Ihres Domänenmodells durchsickert
- Platzieren Sie Ihre Geschäftslogik in Ihrem Domänenmodell, sonst leiden Sie später unter Bugs

- Setzt man auf funktionale Programmierung, ändern sich die Regeln erheblich
- *Anemic Domain Model* ist es bei funktionaler Programmierung eher die Norm
- Funktionale Programmierung fördert die Trennung von Daten und Verhalten
- Funktionen ändern nicht die Daten, die sie als Parameter erhalten; stattdessen geben sie neue Werte zurück
- Diese neuen Werte können der neue Zustand eines *Aggregate* sein oder ein *Domain Event*, das einen Übergang im Zustand des *Aggregate* darstellt
- Bei funktionalen Sprache und einen funktionalen Ansatz für DDD sind einige Leitlinien nicht anwendbar

MODEL-DRIVEN DESIGN has limited applicability using languages such as C, because there is no modeling paradigm that corresponds to a purely *procedural* language.

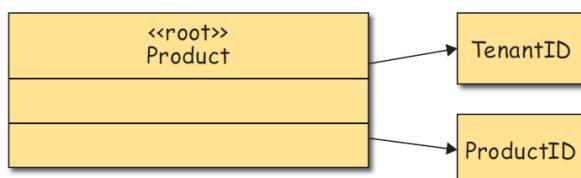
Eric Avens

- *Root Entity* Product
- Diese Basisklasse kümmert sich nur um Standard-*Entity*-Dinge



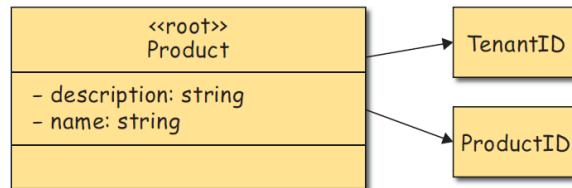
```
public class Product : Entity
{
    ...
}
```

- Jede Aggregate *Root Entity* muss eine global eindeutige Identität haben
- Die TenantId identifiziert die *Root Entity* innerhalb einer gegebenen abonnierenden Organisation
- Diese zweite Identität unterscheidet ein Product von allen anderen innerhalb des gleichen Tenant



```
public class Product : Entity
{
    private ProductID productID;
    private TenantID tenantID;
}
```

- Als Nächstes erfassen Sie alle wesentlichen Attribute oder Felder, die für die Suche nach dem *Aggregate* erforderlich sind.

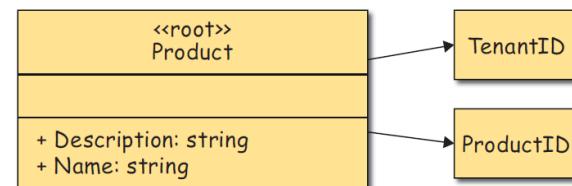


```

public class Product : Entity
{
    private string description;
    private string name;
    private ProductID productID;
    private TenantID tenantID;
}
  
```

# Aggregates modellieren

- Wenn Sie Setter veröffentlichen, kann dies schnell zu Anämie führen, weil die Logik zum Setzen von Werten in Product außerhalb des Modells implementiert würde.
- Denken Sie scharf nach, bevor Sie so etwas tun



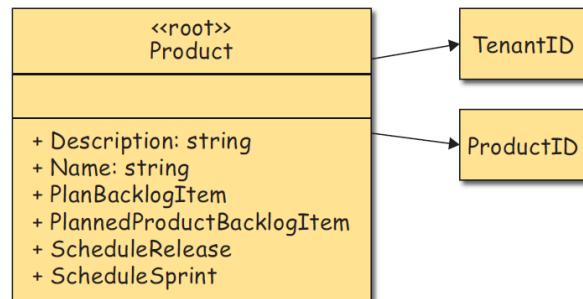
```

public class Product : Entity
{
    ...
    public string Description
    { get; private set; }

    public string Name
    { get; private set; }
}
  
```

neue Methoden:

- PlanBacklogItem(),
- PlannedProductBacklogItem()
- Schedule-Release() und
- ScheduleSprint().



```

public class Product : Entity
{
    ...
    public void PlannedProductBacklogItem(...)
    {
        ...
    }
}
  
```

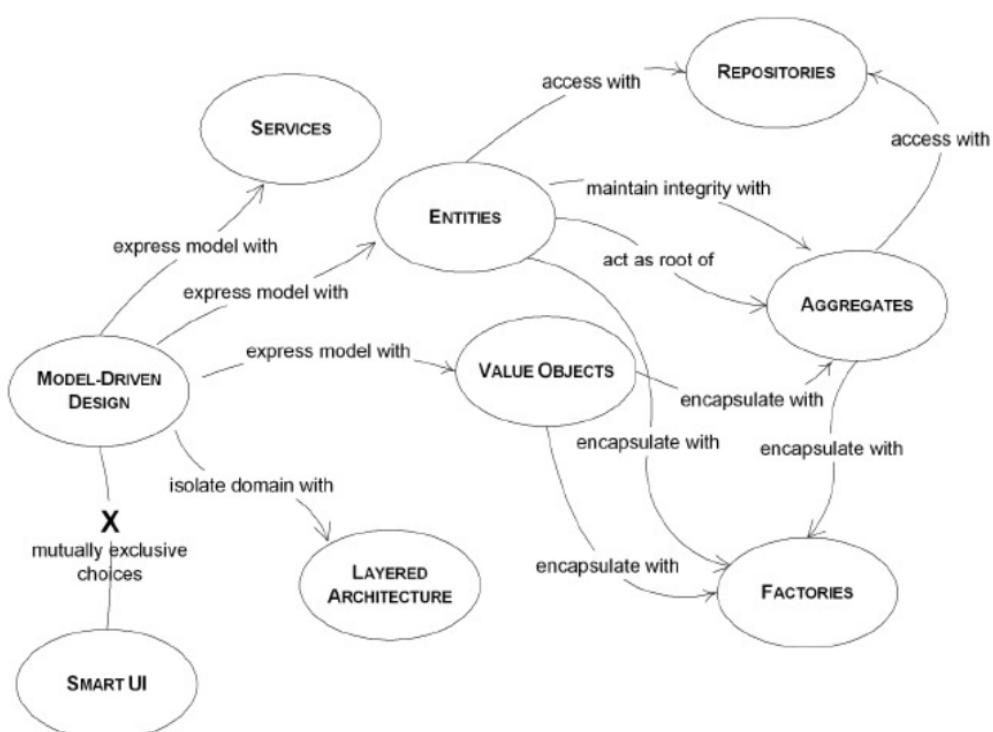
## Abstraktionen mit Bedacht wählen

Probleme bei falscher Abstraktion:

- Die Sprache des Softwaremodells stimmt nicht mit dem mentalen Modell der *Domain Experts* überein
- Das Abstraktionsniveau ist zu hoch, und Sie werden in große Schwierigkeiten geraten, wenn Sie beginnen, die Details der einzelnen Typen zu modellieren
- Dies führt zum Programmieren von Spezialfällen in jeder der Klassen und zu einer komplexen Klassenhierarchie
- Sie haben viel mehr Code, als Sie benötigen
- Oft wird die Sprache der falschen Abstraktionen ihren Weg sogar in die Benutzeroberfläche finden
- Sie werden nie in der Lage sein, alle zukünftigen Bedürfnisse von vornherein abzudecken

- Sie sollten Ihre *Aggregates* auch als abgeschlossene Kapseln für Unit Tests entwerfen
- Unit-Testing unterscheidet sich von der Validierung fachlicher Spezifikationen
- Wir wollen alle Operationen abklopfen, um die Korrektheit, Qualität und Stabilität jedes einzelnen *Aggregate* zu gewährleisten
- Unit-Test-Framework einsetzen
- Werden mit in seinem Quellcode-Repository gespeichert

# Zusammenfassung

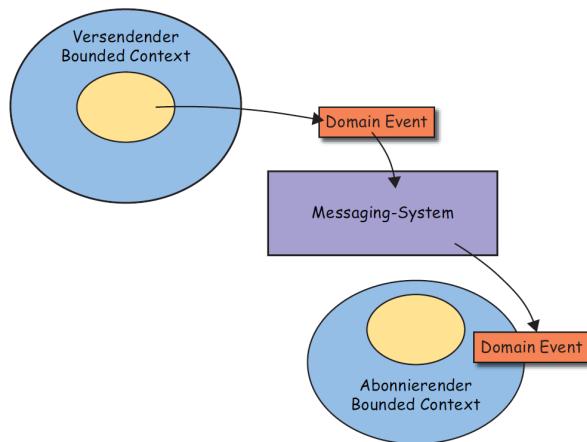


In diesem Teil haben Sie Folgendes kennengelernt:

- Was das Muster *Aggregate* ist und warum Sie es verwenden sollten
- Wie wichtig es ist, beim Design die Konsistenzgrenzen im Auge zu behalten
- Wie die unterschiedlichen Teile eines *Aggregate* aussehen
- Wie die vier Daumenregeln für effektives *Aggregate*-Design lauten
- Wie man das Verhalten eines *Aggregate* modelliert
- Wie wichtig es ist, für Entwürfe das richtige Abstraktionslevel zu wählen

## TAKTISCHES DESIGN MIT DOMAIN EVENTS

- Sue sendet eine Nachricht:  
»Ich habe meine Brieftasche verloren!«
- Gary antwortet darauf:  
»Das ist ja furchtbar!«
- Sue sendet eine Nachricht:  
»Keine Sorge, ich habe meine Brieftasche wiedergefunden!«
- Gary antwortet:  
»Das ist großartig!«
- Wie sieht es jedoch auf verteilte Knoten aus?



## Domain Events entwerfen, implementieren und verwenden

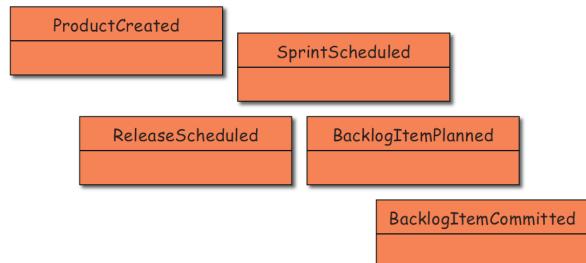
- Minimale Schnittstelle ansehen, die jedes *Domain Event* implementieren sollte
- Auftreten von Datum und Uhrzeit des Domain Events
- *Domain-Event-Typen* implementieren diese Schnittstelle



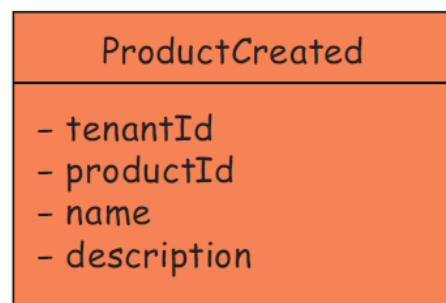
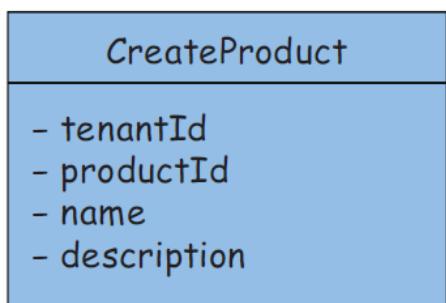
```

public interface DomainEvent
{
    public Date OccurredOn
    {
        get;
    }
}
  
```

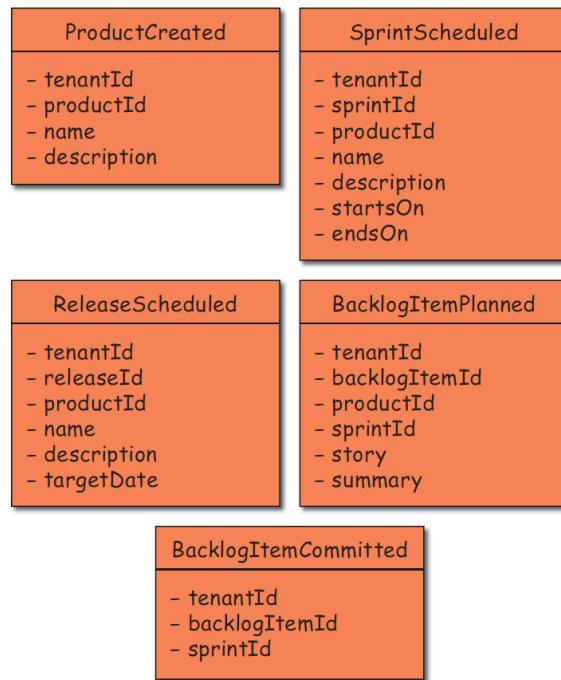
- Namen Ihrer *Domain-Event*-Typen sollten ein vergangenes Ereignis ausdrücken
- ProductCreated, zum Beispiel, drückt aus, dass ein Scrum-Produkt erstellt wurde
- Aber welche Eigenschaften sollte ein *Domain Event* enthalten?



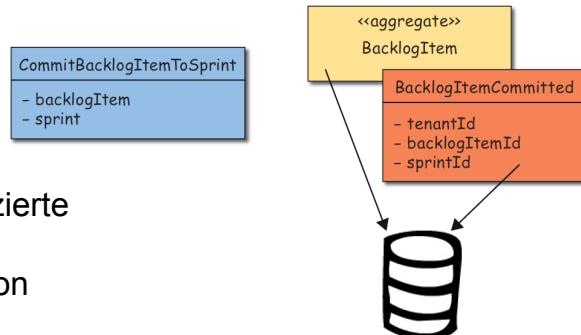
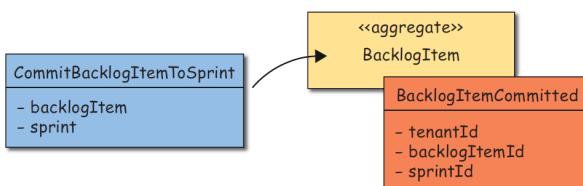
- Jeder dieser Eigenschaften ist wesentlich für die Erstellung eines Product
- Daher sollte das *Domain Event* ProductCreated alle Eigenschaften enthalten, die mit dem *Command* geliefert wurden, das es erstellt hat



- Beispiele geben Ihnen eine gute Vorstellung von den Eigenschaften
- Wenn beispielsweise ein BacklogItem an einen Sprint gebunden wird, dann wird das *Domain Event* BacklogItemCommitted instanziert und veröffentlicht
- *Domain Event* mit zusätzlichen Daten angereichert
- keine weiteren Anfragen vom Empfänger nötig

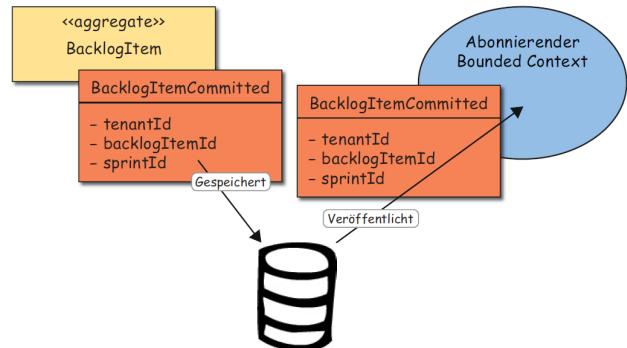


# Domain Events entwerfen, implementieren und verwenden



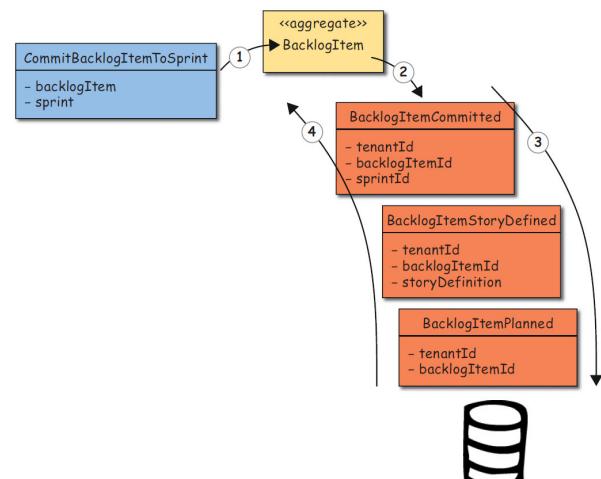
Entscheidend ist, dass das modifizierte Aggregate und das *Domain Event* zusammen in derselben Transaktion gespeichert werden

- Sobald Ihr *Domain Event* im Event Store gespeichert ist, kann es für alle Interessenten veröffentlicht werden
- Diese können sich innerhalb Ihres eigenen *Bounded Context* und in externen *Bounded Contexts* befinden



## Event Sourcing

- *Event Sourcing* lässt sich als das Speichern aller *Domain Events* beschreiben, die für eine *Aggregate*-Instanz aufgetreten sind
- Anstatt den *Aggregate*-Status als Ganzes festzuhalten, werden alle einzelnen *Domain Events* gespeichert
- Alle *Domain Events*, die eine *Aggregate*-Instanz betreffen, bilden in der Reihenfolge, in der sie ursprünglich aufgetreten sind, seinen Event Stream



## MIGRATION EINER LEGACY APPLIKATION

## Inhalt

- Ausgangssituation
- Verfahren und Vorgehen zur Migration
- Auswahl der Anwendung
- Die Migration nach Services
- Ergebnisse und nächste Schritte
- Fragen und Antworten

- IT Bereich „Beratung und Vertrieb“ einer Großbank
- Zeigt Vorteile und die Verwendbarkeit des DDD Ansatzes
- Zeigt die nächsten Schritte zur Einführung eines DDD
- Ziele entstanden auf Basis der Erfahrungen aus der bestehenden Kundenbeziehung
- Fachlich motivierte Ziele wurden auf Basis der vorliegenden Erfahrungen angenommen
- Ziele im Hinblick den IT Nutzen waren:
  - Eine reaktive Anwendung die möglichst kurze Time-To-Market Zyklen für neue bzw. geänderte Funktionalität erlaubt
  - Ein effektiver Betrieb bei dem Aufwand und Kosten in Bezug auf die benötigte Funktionalität minimiert werden

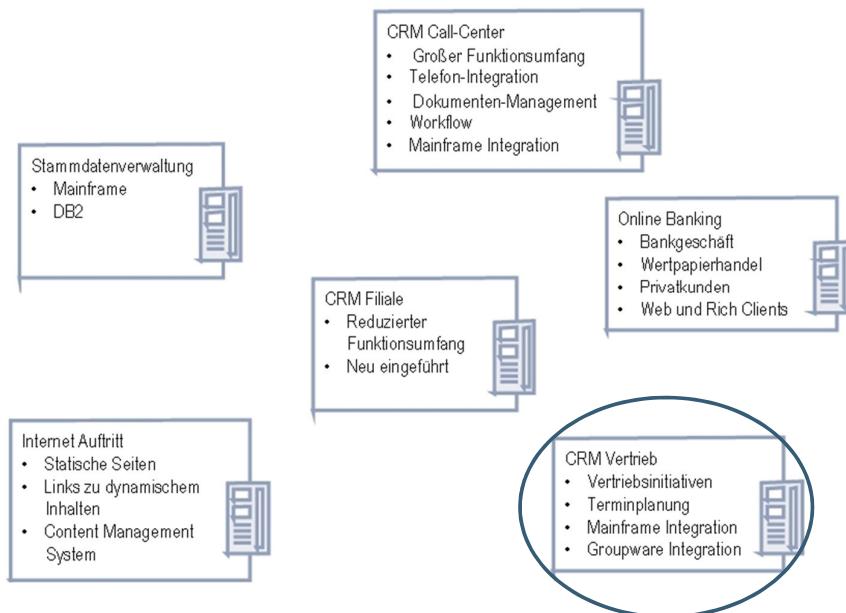
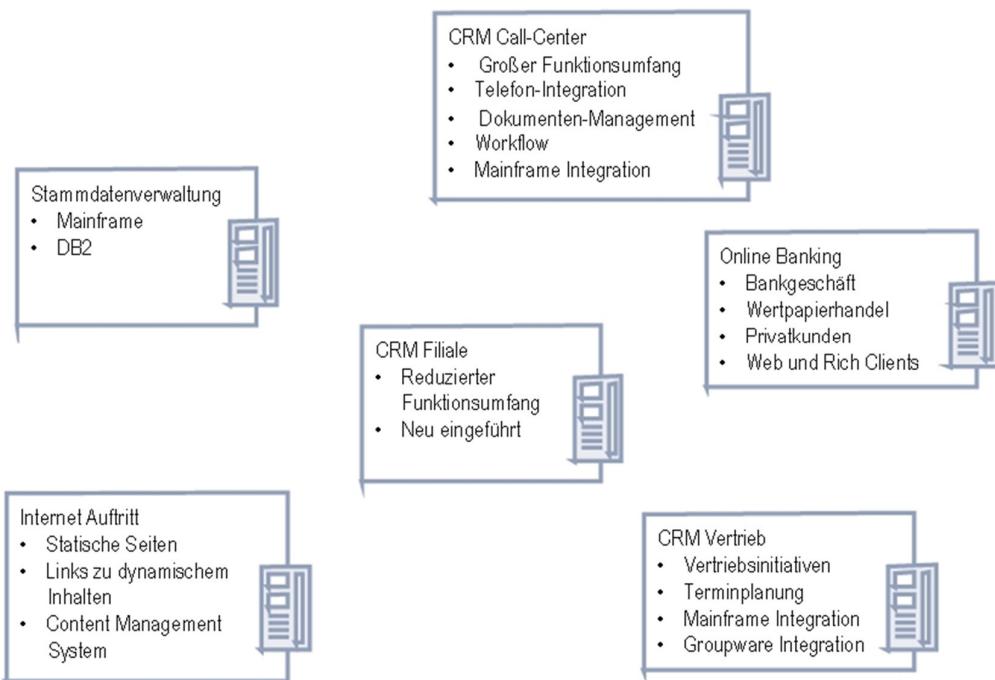
## Was ist dieses Mal anders?

- Technische Standards sind reifer geworden
- Offene Standards werden angenommen
- Werkzeuge sind weiter entwickelt
- Unternehmensziele als treibende Kräfte
- Anwendungen werden nicht mehr für EINEN Anwendungsfall gebaut

- Bottom-Up
  - Vorteile
  - Liefert schneller Ergebnisse
  - Auf Abteilungsebene nutzbar
  - Berücksichtigt vorhandenes
  - Nachteile
  - Reduzierte Flexibilität
  - Reduzierte Standardisierung
  - DDD Vorteile eingeschränkt
- Top-Down
  - Vorteile
  - Fachlicher Bezug
  - Standardisierung
  - Bessere Qualität
  - Verbesserte Wiederverwendung
  - Nachteile
  - Längere Dauer
  - Größerer Initialaufwand
  - Erfordert viel Disziplin und Wissen

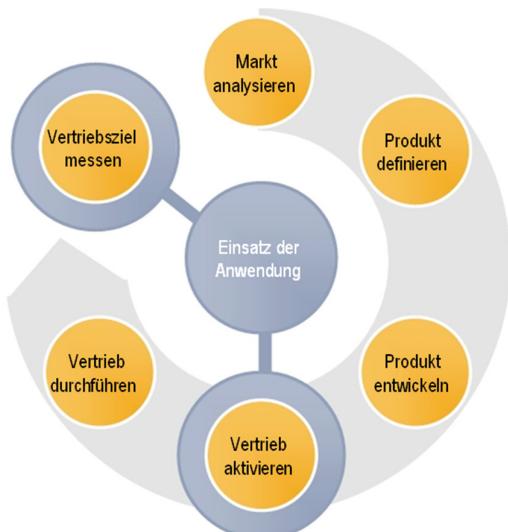
## Auswahl der Anwendung

- Kriterien zur Auswahl geeigneter Anwendungen
  - Anwendung kann als Service-Provider agieren
  - Anwendung befindet sich in einem „stabilen“ Umfeld
  - Anwendung enthält Technologien mittlere Komplexität
  - Anwendung ist nicht „unternehmenskritisch“
  - Potentielle Dienste können wieder verwendet werden



- Eigenschaften der Anwendung CRM Vertrieb
  - Enthält bereits Fragmente von Geschäftprozessen
  - Anwendung hat Auswirkung auf Vertriebsergebnis
  - Verfügt über viele Schnittstellen
  - Keine Technologien mit hoher Komplexität

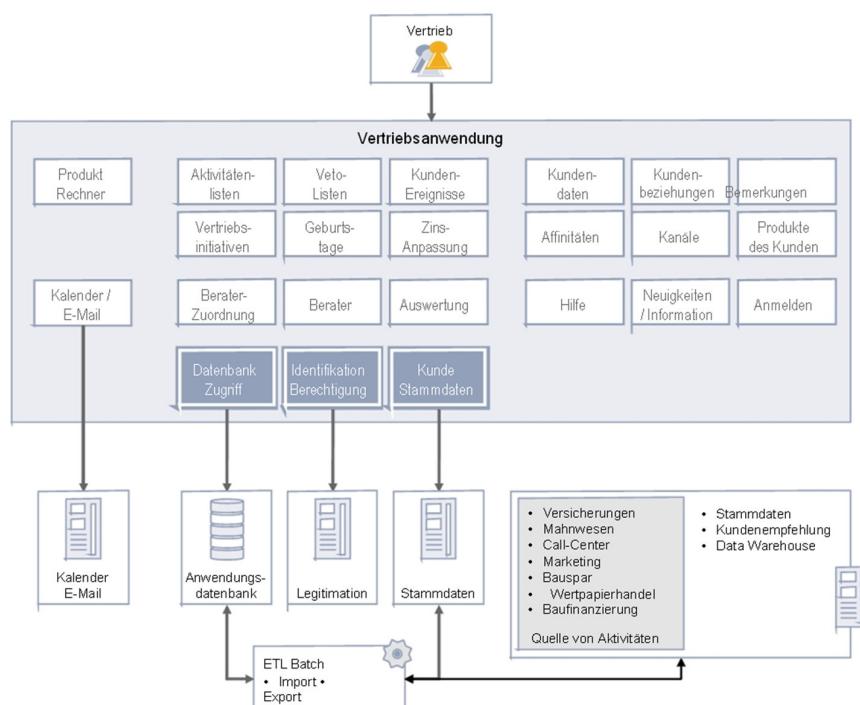
## Migration

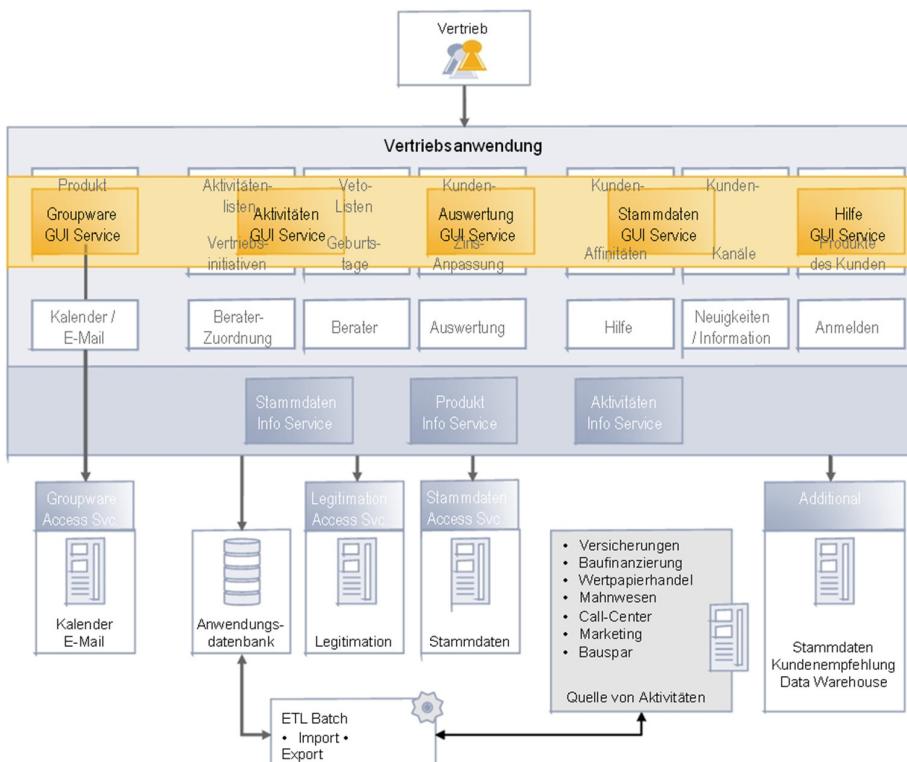
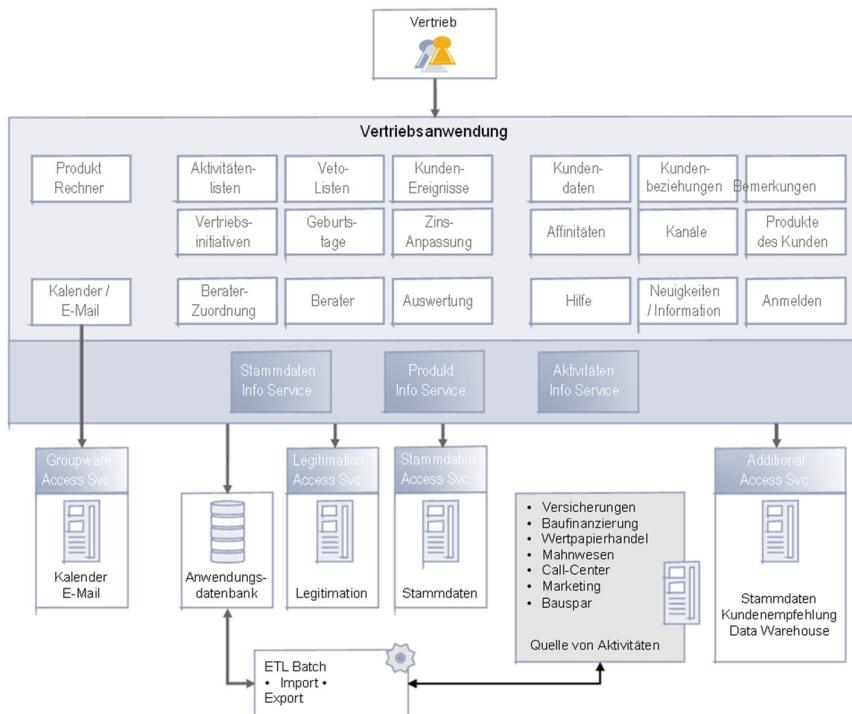


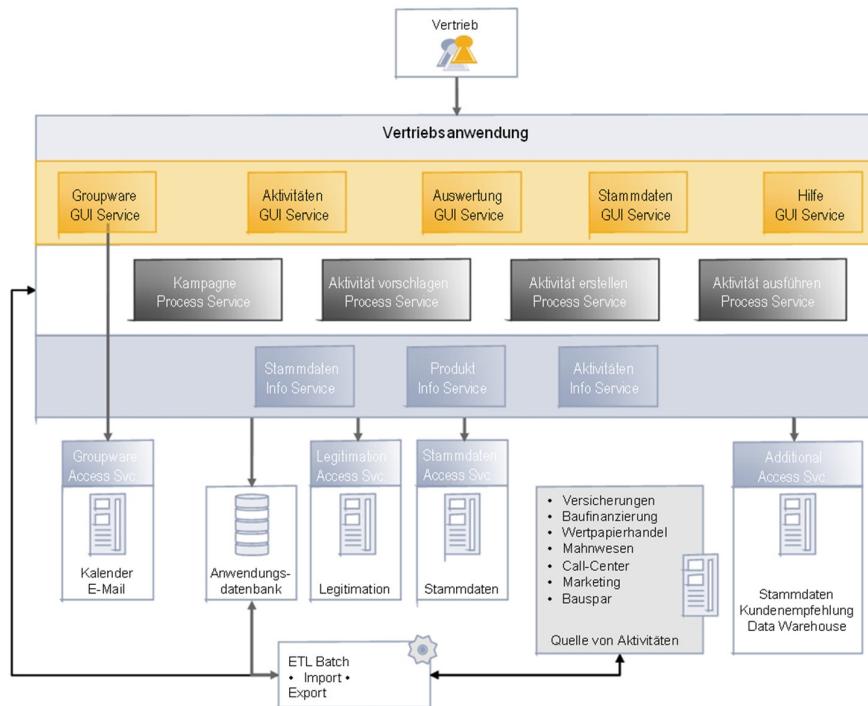
- Aufgaben der Anwendung
  - Verwaltung von Aktivitäten
  - Definition von Vertriebszielen
  - Überwachung der Zielerreichung
  - Gesprächsplanung

- Bestehende Anwendung
  - Front-End Schicht
    - JEE Web Anwendung
    - Kundenspezifisches Framework für Präsentation und Datenanbindung
  - Back-End Schicht
    - DB2 Datenbank für Daten der Anwendung
    - Lotus Domino für Messaging und Groupware Funktionalität
    - ETL Produkt auf für Datenintegration zwischen Bestandssystemen und DB2 im Batch
    - Mainframe-Anwendung für Authentisierung und Autorisierung von Mitarbeitern
    - Mainframe-Anwendung für Kunden-Stammdaten
    - Vielzahl unterschiedlicher Bestandssysteme

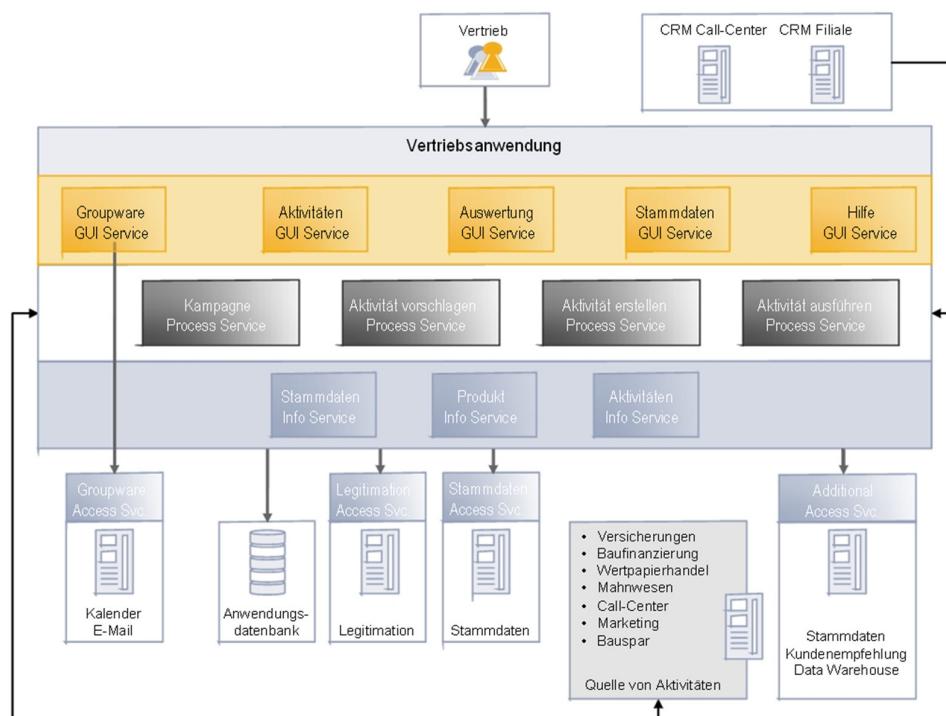
## Bestehende Anwendung







## Letzter Schritt



- Anzahl der Schnittstellen ist gesunken (in diesem Beispiel)
- Ressourcenbedarf für doppelte Datenhaltung reduziert
- Bedarf nach einem ETL Werkzeug entfällt
- Vorhandene Bestandssysteme werden weiter genutzt
- Betriebskosten werden reduziert und effizienter eingesetzt
- Entwicklung einer Referenzarchitektur für den Unternehmensbereich

## Ergebnisse / Unternehmensnutzen

- Kürzere Time-To-Market Zeiten
- Größere Flexibilität durch Entkopplung von Präsentation, Prozessen und Bestandssystemen
- Mehr Effizienz durch wieder verwendbare Dienste der Bestandssysteme
- Möglichkeit zur Aktivierung weiterer Vertriebskanäle
- Bessere Kontrollmöglichkeiten für den Vertriebsprozess

- Top-Down Analyse und Design weiterer Geschäftsprozesse
- Detaillierung der Migrationsstrategie
- Wirtschaftlichkeitsbetrachtung bei anderen Systemen
- Detaillierung der organisatorischen Rahmenbedingungen

## IBM nennt fünf effiziente und effektive Einstiegspunkte in die Arbeit mit Services

- Mitarbeiter
  - Der Schwerpunkt liegt hierbei auf der Steigerung der Produktivität durch Zusammenarbeit und Interaktion mit Anwendungs- und Informationsservices, die die Geschäftsprozesse unterstützen.
- Prozesse
  - Werkzeuge und Services zur Optimierung des Managements von Geschäftsprozessen (Business Process Management, BPM) für die kontinuierliche Entwicklung innovativer Lösungen.
- Informationen
  - Bereitstellung des Zugriffs auf komplexe, heterogene Datenquellen innerhalb des Unternehmens, indem Informationen als Service zur Verfügung gestellt werden.
- Konnektivität
  - Verknüpfung von Mitarbeitern, Prozessen und Informationen im Unternehmen mit einem nahtlosen Nachrichten- und Informationsfluss, und das unabhängig vom Standort und Zeitpunkt sowie unter Verwendung beliebiger Werkzeuge.
- Wiederverwendung
  - Kernpunkte sind hierbei, kontinuierlichen Nutzen aus bereits getätigten Investitionen zu ziehen, Services zu identifizieren, die ausgelagert werden können, und neue Services für die Bereiche zu entwickeln, die bisher nicht über das Produktpotfolio abgedeckt sind.

- Think BIG, Start small (denke groß, fange klein an)
- Fachabteilung einbinden
- Bestandsaufnahme
- Erste Services einbinden
- Registry installieren
- Governance regeln
- Sicherheit planen
- Messaging Infrastructure einrichten
- Service Management einrichten
- Services orchestrieren

---

## Fragen

... und Antworten

---

© Integrata Cegos GmbH

Integrata Cegos GmbH  
Zettachring 4  
70567 Stuttgart

**Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.**

---