DDD in der Praxis

KARLSRUHER ENTWICKLERTAG 03. Juni 2024

embarc

0

Alexander Kaserbacher

Alexander.Kaserbacher@embarc.de
linkedin.com/in/alexksbr
@alexksbr
@alexksbr@mastodon.online

embarc

1

Dr. Felix Kammerlander

Felix.Kammerlander@embarc.de
linkedin.com/in/felix-kammerlander
@f_kammerlander
@f_kammerlander@mastodon.social

2

00. Agenda

Was euch erwartet

01 Wichtige Begriffe
02 Event Storming
03 Workflows
04 Integrationsmuster
05 Taktisches DDD
06 Konsistenz

01.

DDD - Wichtige Begriffe

4

Was ist ...

Domain-driven Design?

5

Softwareprojekte scheitern … regelmäßig

- Zu spät, zu teuer, erfüllen die Anforderungen nicht
- Zentrales Problem: Fehlgeschlagene Kommunikation
  - Unklare Anforderungen
  - Unklare Projektziele
  - Ineffiziente Koordination von Aufwänden zwischen Teams

6

Domain-driven Design

- Bietet Werkzeuge und Praktiken um effiziente Kommunikation zu ermöglichen
- Strategisches DDD
  - Business-Domänen und -Strategie
  - High-level Design-Entscheidungen
    - Zerlegung des Systems in Komponenten und deren Integration
- Taktisches DDD
  - Muster zur Abbildung der Business-Domäne in Code

7

DDD ist …

Alignment von Software Design und Business-Domänen

8

DDD ist …

(Business) Domain-driven (Software) Design

9

Wichtige Begriffe

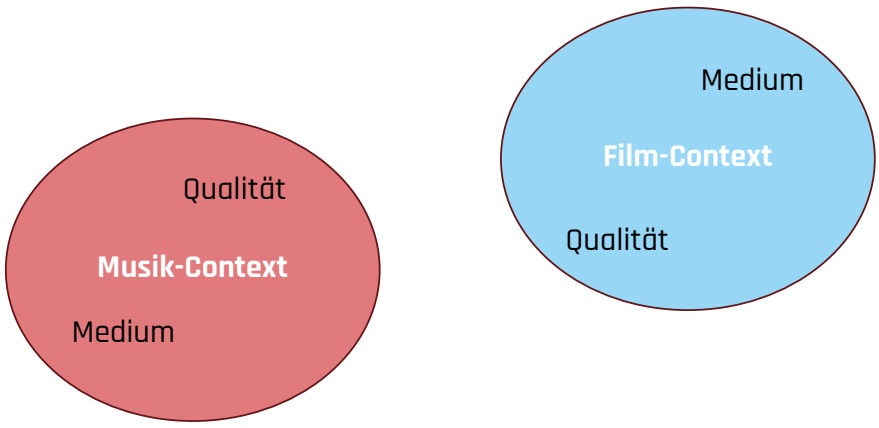
10

11

Bounded Context

- Bilden Modellgrenzen, insbesondere also **Grenzen der ubiquituous language**
- Bounded Contexts **werden entworfen**, sind also Ergebnis expliziter Entscheidungen
  - Maximale Größe durch Inkonsistenzen im Modell
- Ein Bounded Context wird von einem Team verwaltet

Bounded Context



Redundante Datenhaltung

- Kein Begriff von DDD
- Zunächst nicht intuitive, aber häufige Konsequenz bei der Anwendung von DDD
- Daten liegen inhaltlich gesehen mehrfach vor, d.h. aber nicht, dass die Daten in Form und Verständnis identisch sind

14

Wartbarkeits-probleme

15

Verschiedene Probleme mit Wartbarkeit

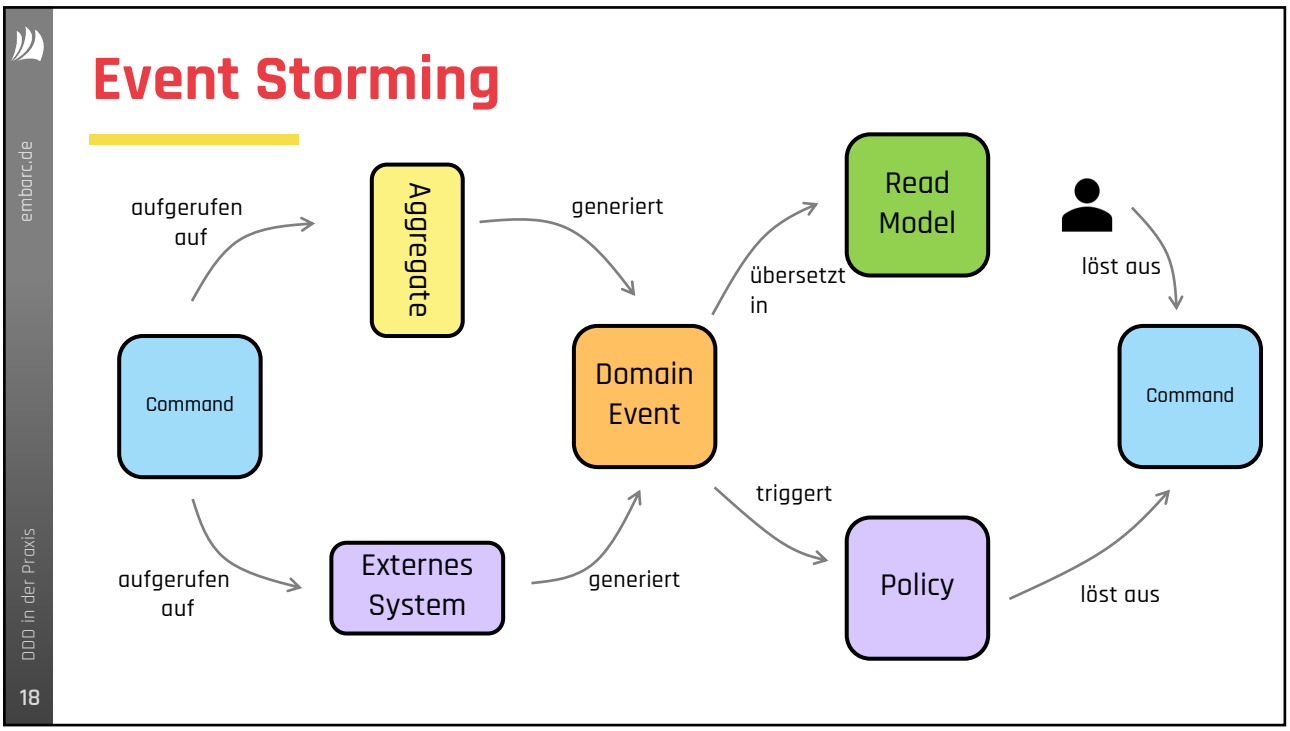
Problem	Adressiert durch
Abhängigkeit von anderen Teams	Bounded Contexts, Redundante Datenhaltung
Schlecht lesbarer Code	Ubiquitous language, Bounded Contexts, Redundante Datenhaltung
Zu große Module	Bounded Contexts
Komplexe Queries und Transformationen	Redundante Datenhaltung
Inkonsistente Modellierung der Domäne	Bounded Contexts, Ubiquitous language

16

02.

Event Storming

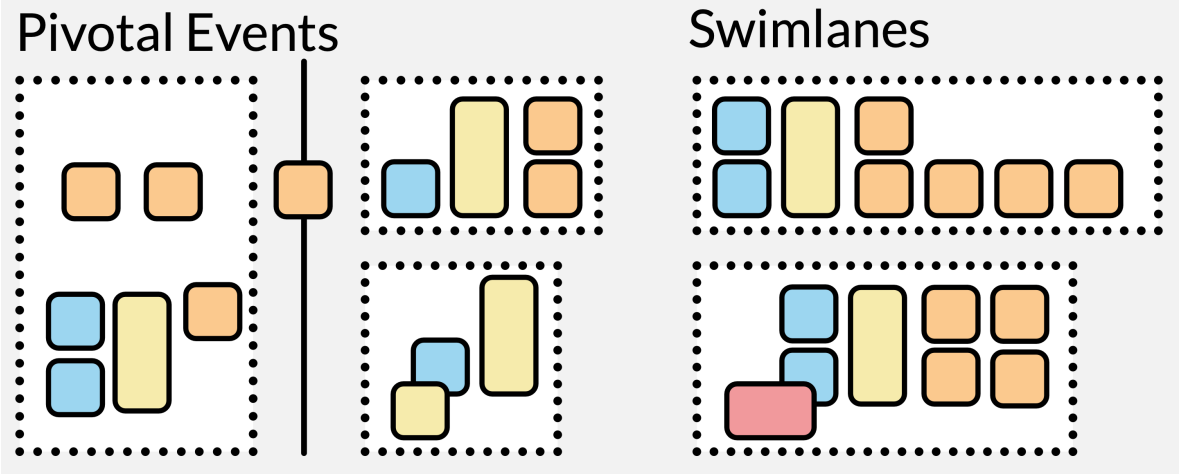
17


embarc.de
DDD in der Praxis
18
Event Storming
aufgerufen auf
Aggregate
generiert
Read Model
übersetzt in
löst aus
Command
Domain Event
Command
triggert
aufgerufen auf
Externes System
generiert
Policy
löst aus

18

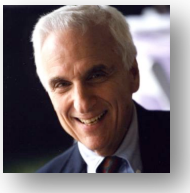
19

Bounded Contexts aus Event Storming



20

Conway's Law - Ein Einflussfaktor auf Softwarearchitektur



"Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations"

Melvin Conway, 1968

- Die Struktur in einer Softwarearchitektur und die Struktur der Organisation sind oftmals kongruent. Dabei können Kräfte in beide Richtungen wirken.

"The organization of the software and the organization of the software team will be congruent ... If you have four groups working on a compiler, you'll get a 4-pass compiler"

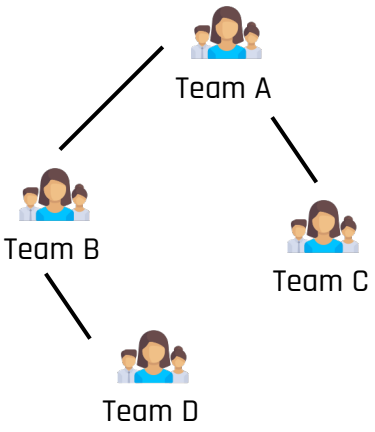
Eric S. Raymond

21

Inverse Conway Maneuver

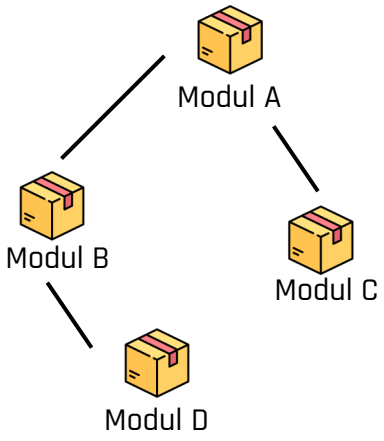
Kommunikationsstruktur

Zuerst von Organisation ausgehen
Rahmen für Modularisierung schaffen



Modulstruktur

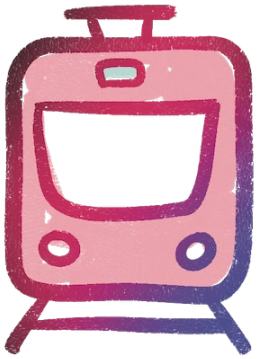
Module und Strukturierung ergeben sich (nach Conway's Law) aus Kommunikationsstruktur



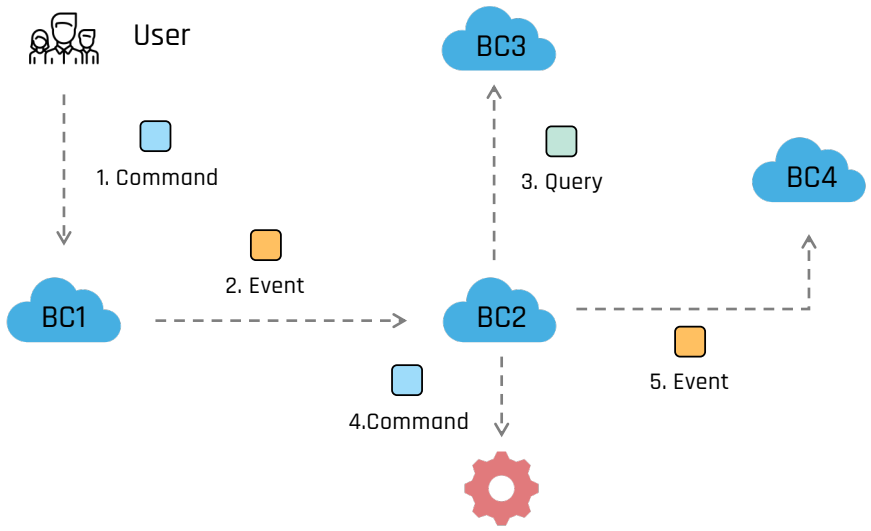
22

03.

Workflows



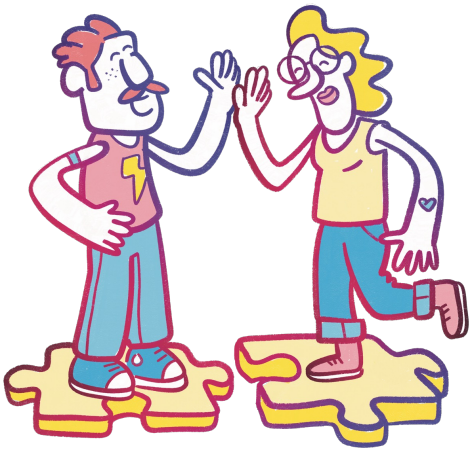
Domain Message Flow Modeling



24

04.

Integrationsmuster

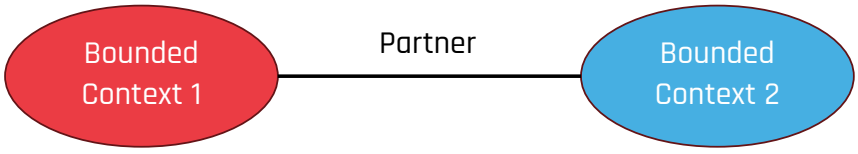


25

Integrationsmuster

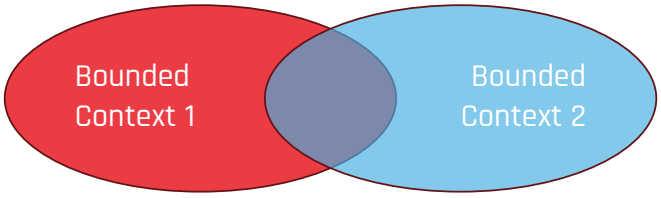
- Bounded Contexts sind nicht unabhängig voneinander:
  - Sie sollen sich **unabhängig entwickeln** lassen
  - Aber sie müssen sich **integrieren**
- Zentrale Fragestellungen zur Integration
  - Wie sehen die Verträge aus?
  - Wie werden Änderungen vorgenommen?
  - Welche Sprache wird für die Integration verwendet?
- Integrationsmuster
  - Muster zur Definition von Beziehungen zwischen Bounded Contexts
  - Kollaborationsgetrieben

Kooperation: Partnership



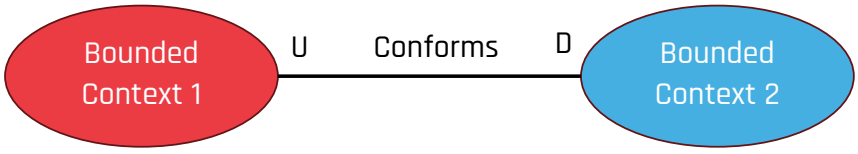
- Ad-hoc-Koordination in beide Richtungen
- Gemeinsame Sprache, wo notwendig

Kooperation: Shared Kernel



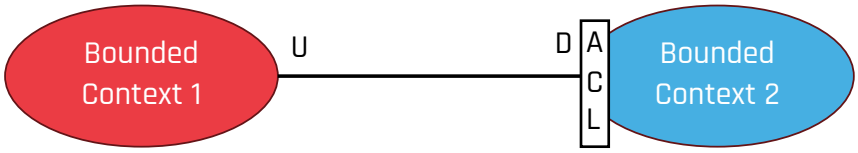
- Geteiltes Modell
- Änderungen haben direkte Auswirkungen auf alle betroffenen Bounded Contexts

Customer – Supplier: Conformist



- Macht liegt Upstream
- Konsumenten akzeptieren das Upstream-Modell und passen sich an

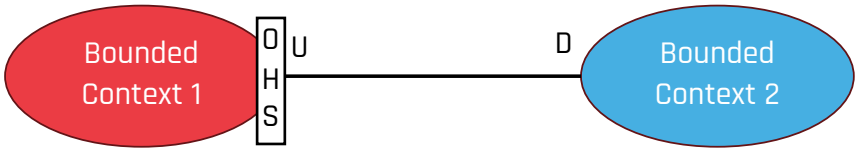
Customer – Supplier: Anti Corruption Layer



- Macht liegt Upstream
- Downstream übersetzt das Upstream-Modell in ein für sich passendes Modell

30

Customer – Supplier: Open Host Service



- Macht liegt Downstream
- Upstream schützt Konsumenten vor internen Änderungen
- Protokoll, das für Konsumenten passt: Published Language

31

Keine Integration: Separate Ways

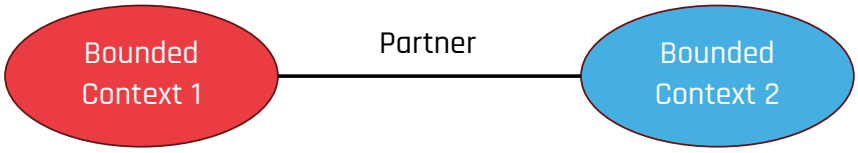
- Keine Kollaboration zwischen Bounded Contexts

32

Integrationsmuster auswählen

33

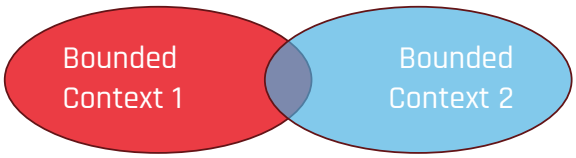
Kooperation: Partnership



- Benötigt etablierte Kollaborationspraktiken, viel Commitment und Synchronisation

34

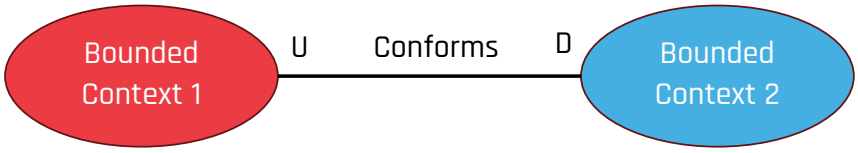
Kooperation: Shared Kernel



- Wenn Kosten der Duplizierung höher sind, als die der Integration
- Falls Partnership nicht möglich ist
- Integration zweier Contexts, die von einem Team entwickelt werden, wo aber eine Partnership die Kontextgrenzen verwischen könnte
- Temporär zur Ablöse von Legacy Systemen

35

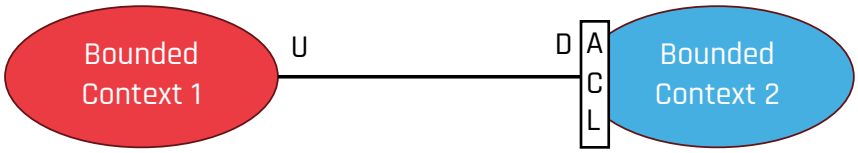
Customer – Supplier: Conformist



- Akzeptanz von Industriestandards
- Anbindung externer Service Provider
- Modell ist passend oder gut genug für den Konsumenten

36

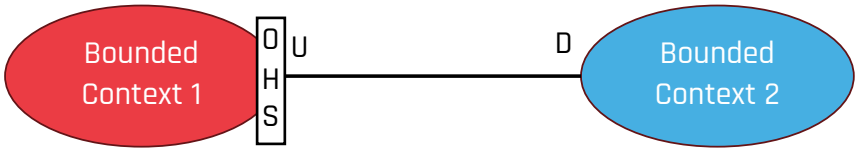
Customer – Supplier: Anti Corruption Layer



- Falls Downstream eine Core Subdomain behandelt, die geschützt werden muss
- Falls Upstream ineffizient oder unangenehm ist, z.B. Legacy-Systeme
- Falls Upstream das Modell häufig ändert

37

Customer – Supplier: Open Host Service



- Falls Upstream seine Konsumenten vor Änderungen schützen möchte
- Falls Versionierung eine große Rolle spielt

38

Keine Integration: Separate Ways

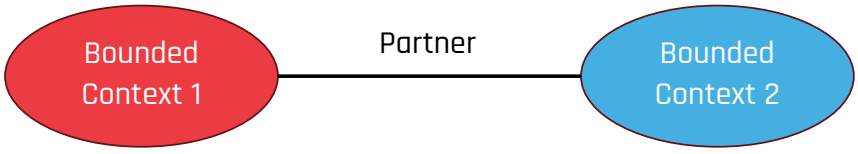
- Bei Kommunikationsproblemen
- Falls Teile jeweils einfacher separat eingebaut werden können, als diese zentral zur Verfügung zu stellen
- Bei Core Subdomains vermeiden
- Zu große Unterschiede für Conformist oder ACL

39

Integrationsmuster effektiv umsetzen

40

Kooperation: Partnership

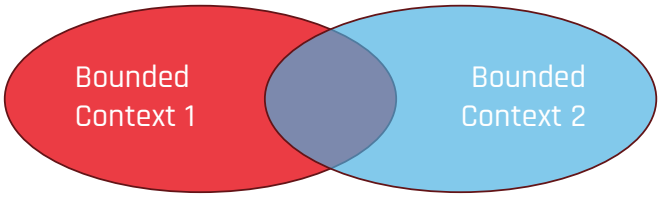


- Nähe in Geographie, Sprache, Zeitzone
- Probleme in Context 1 führen unweigerlich zu Problemen in Context 2 und umgekehrt: Hoher Grad an CI von Vorteil

41

41

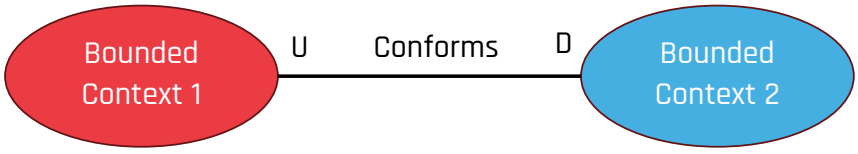
Kooperation: Shared Kernel



- Änderungen am Shared Kernel haben direkte Auswirkungen auf alle verwendenden Contexts
- Geteilte Dateien in einem Mono-Repo oder Verweis auf ein separates Projekt
- CI-Trigger bei Änderungen im Shared Kernel für alle verwendenen Contexts

42

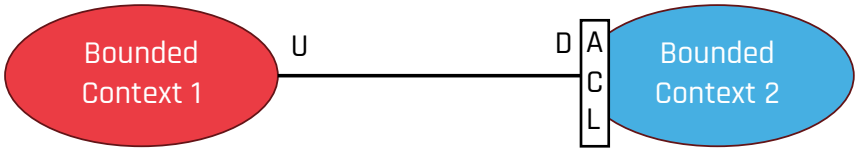
Customer – Supplier: Conformist



- Betroffene Domain in Conformists möglichst klein und wenig komplex halten

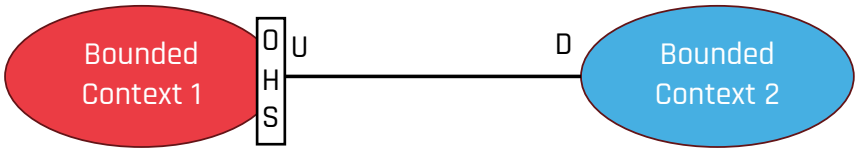
43

Customer – Supplier: Anti Corruption Layer



- Facade und Adapter Pattern
- Mapper-Bibliotheken
- Ggf. in einem API Gateway zentralisiert

Customer – Supplier: Open Host Service



- Facade und Adapter Pattern
- Mapper-Bibliotheken
- Ggf. in einem API Gateway zentralisiert
- OpenAPI, automatische Generierung von Clients
- Semantische Versionierung

05.

Taktisches DDD

Komplexe Business-Logik umsetzen

46

embarc.de

DDD in der Praxis

47

Domain Model

- Modellierung auf Code-Ebene zur Behandlung komplexer Geschäftslogik
- Kein einfaches CRUD:
  - Komplizierte Statusübergänge
  - Validierungen
  - Invarianten, d.h. Regeln, die zu jeder Zeit eingehalten werden müssen
- Verwendet die folgenden Muster als Bausteine (tactical DDD patterns)

47

Value Objects

- Objekt kann durch seine Werte identifiziert werden
- Haben keine ID
- Nicht veränderbar, d.h. immutable
- Beschreiben häufig Eigenschaften von Objekten

```
record Color
{
    2 Verweise
    public int Red { get; init; }
    2 Verweise
    public int Green { get; init; }
    2 Verweise
    public int Blue { get; init; }

    0 Verweise
    public Color MixWith(Color other){
        return new other with {
            Red = Math.Min(other.Red + Red, 255),
            Green = Math.Min(other.Green + Green, 255),
            Blue = Math.Min(other.Blue + Blue, 255)
        };
    }
}
```

Learning Domain-Driven Design, O'Reilly, 2021

48

Entities

- Objekt hat eine eigene Identität
- Gleiche Eigenschaften bedeuten nicht, dass die Objekte gleich sind
- Unterscheidung durch eindeutige ID
- Veränderbar
- Sind Bausteine für Aggregates

```
class User
{
    1 Verweis
    public System.Guid UserId { get; }

    0 Verweise
    public string UserName { get; set; }
    0 Verweise
    public string Password { get; set; }
    0 Verweise
    public Adress Adress { get; set; }

    0 Verweise
    public User() {
        UserId = System.Guid.NewGuid();
    }
}
```

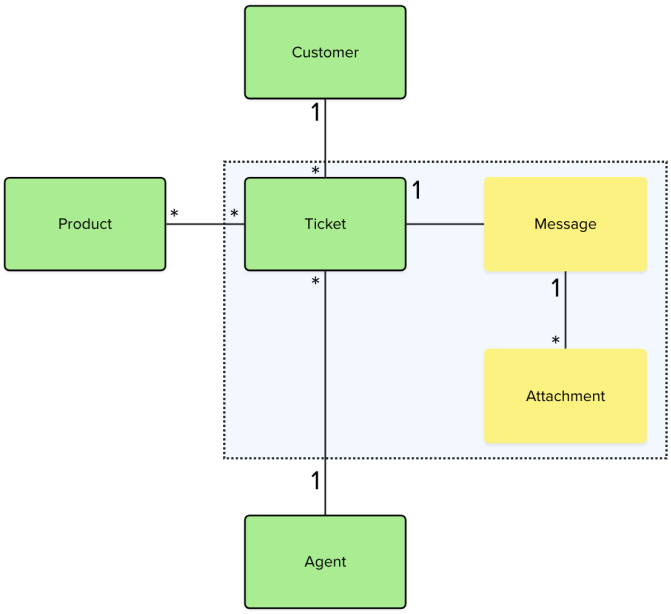

49

Aggregates

- **Hierarchie** von Entities und Value Objects
- Haben **Konsistenzgrenzen für starke Konsistenz**
- Sämtliche Daten innerhalb der Konsistenzgrenze werden **transaktionell** verarbeitet
- Änderungen am Aggregate und Referenzen zum Aggregate nur über eine öffentliche Schnittstelle in der **Aggregate Root**
- Aggregate außerhalb werden **über Ids referenziert**

50

Aggregates



Learning Domain-Driven Design, O'Reilly, 2021

51

06.

Konsistenz

52

ACID – Konsistenz in Softwaresystemen

Atomic:

- ✓ Eine Sequenz von Daten-Operationen wird entweder **ganz oder gar nicht** ausgeführt.

Consistent:

- ✓ Eine Sequenz von Daten-Operationen hinterlässt nach Beendigung einen konsistenten Datenzustand, (**Integritätsbedingungen** bleiben erfüllt)

Isolated:

- ✓ Nebenläufig in Ausführung befindliche Daten-Operationen **beeinflussen** sich nicht gegenseitig.

Durable:

- ✓ Daten sind nach dem erfolgreichen Abschluss einer Transaktion garantiert **dauerhaft** gespeichert.

53

53

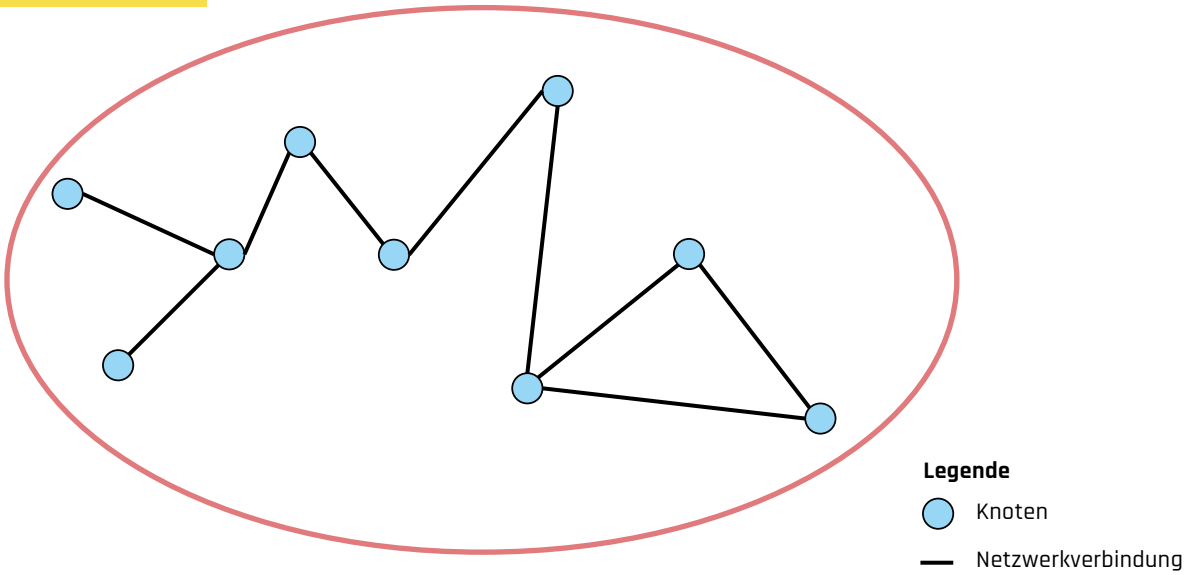
Fallacies of Distributed Computing

- Das Netzwerk ist ausfallsicher.
- Die Latenzzeit ist gleich Null.
- Der Datendurchsatz ist unendlich.
- Das Netzwerk ist sicher.
- Die Netzwerktopologie wird sich nicht ändern.
- Es gibt immer nur einen Netzwerkadministrator.
- Die Kosten des Datentransports können mit Null angesetzt werden.
- Das Netzwerk ist homogen.

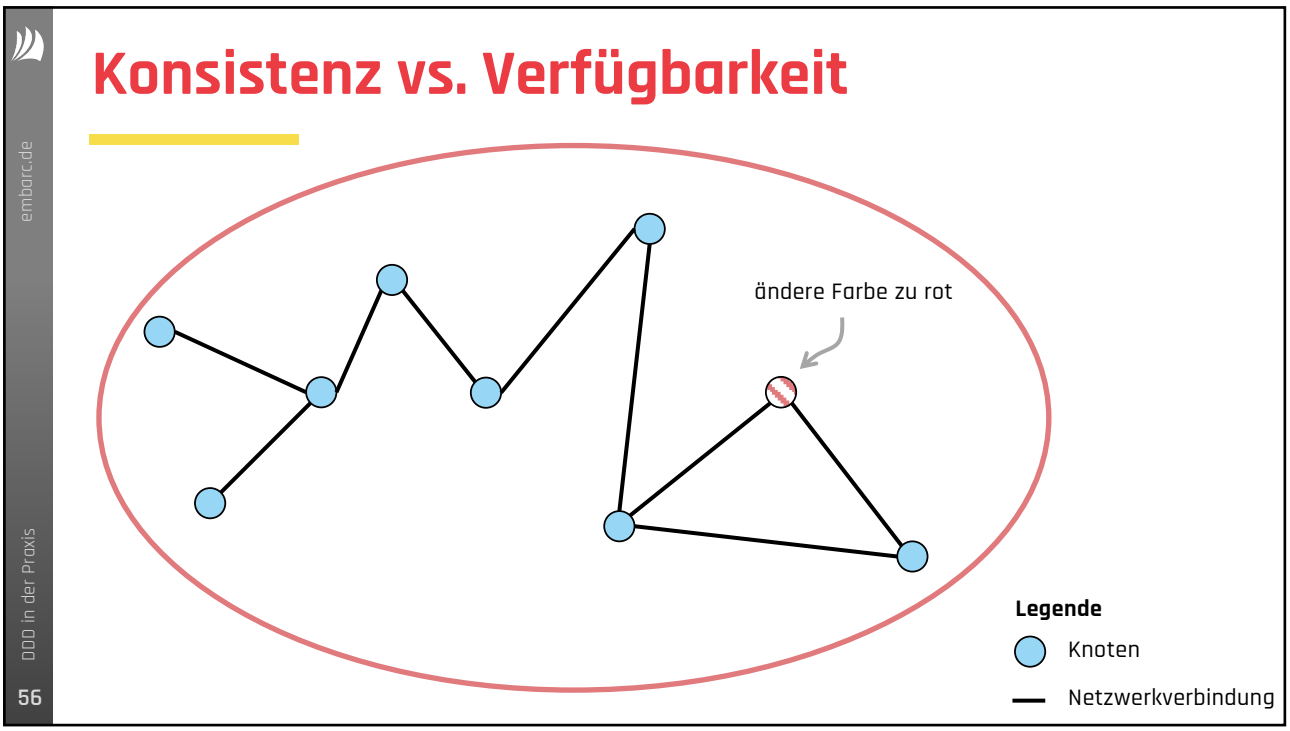
Fallacies == Trugschlüsse / Irrtümer

54

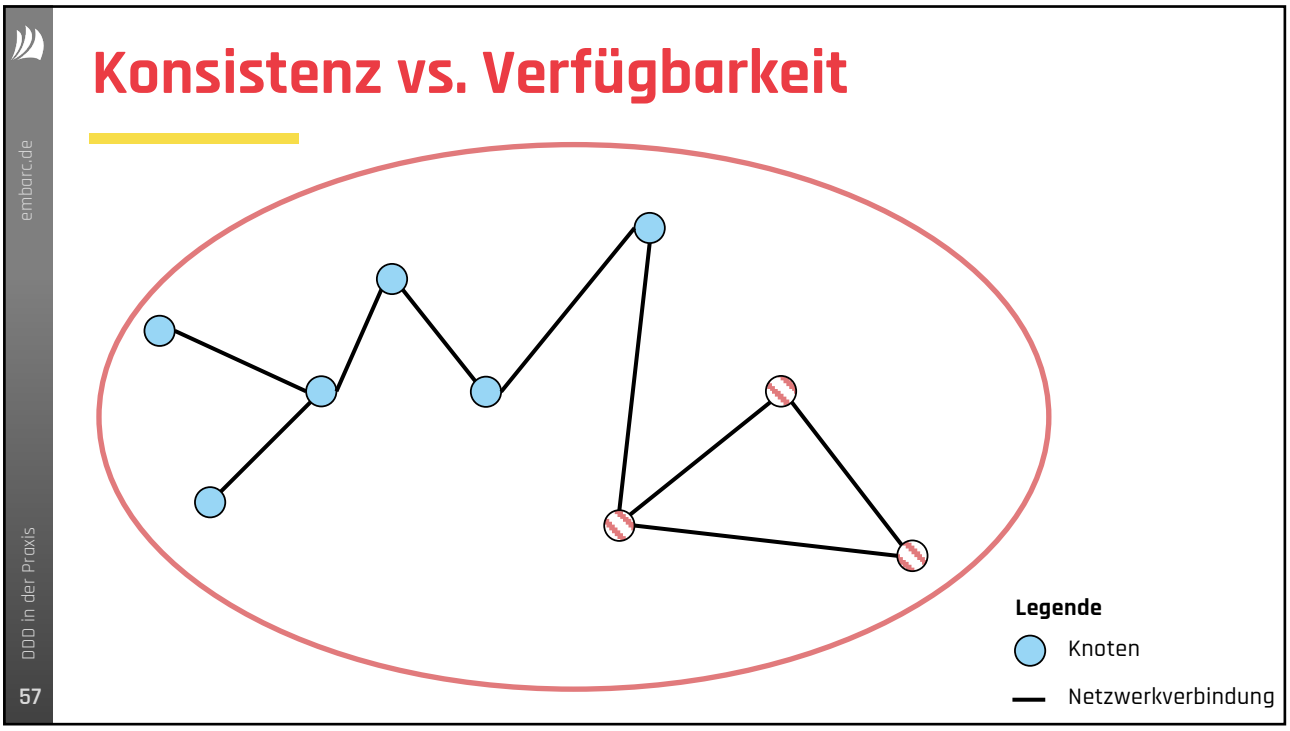
Konsistenz vs. Verfügbarkeit



55

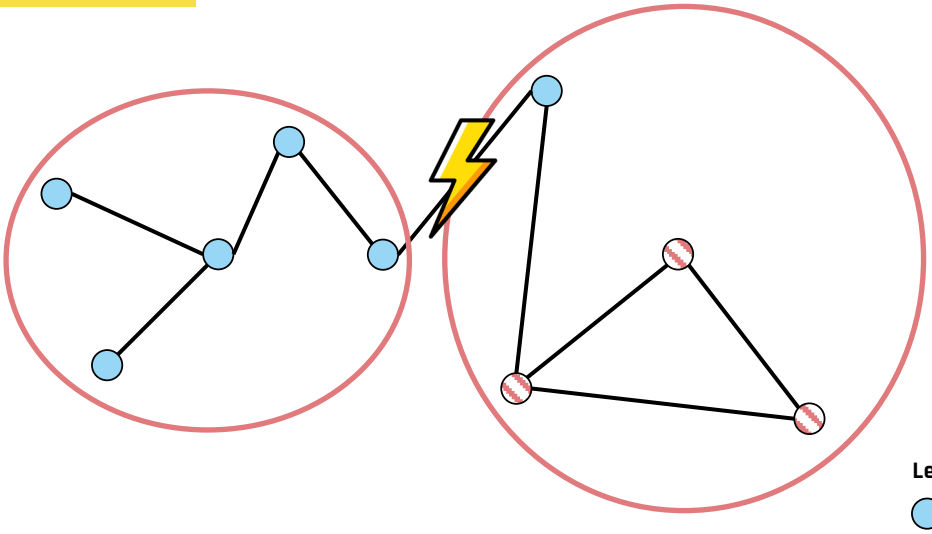

Konsistenz vs. Verfügbarkeit
embarc.de
DDD in der Praxis
56
ändere Farbe zu rot
Legende
Knoten
Netzwerkverbindung

56


Konsistenz vs. Verfügbarkeit
embarc.de
DDD in der Praxis
57
Legende
Knoten
Netzwerkverbindung

57

Konsistenz vs. Verfügbarkeit



Legende

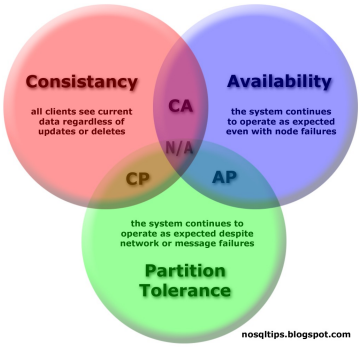
Knoten

Netzwerkverbindung

58

CAP-Theorem (2002)

„Pick two!“

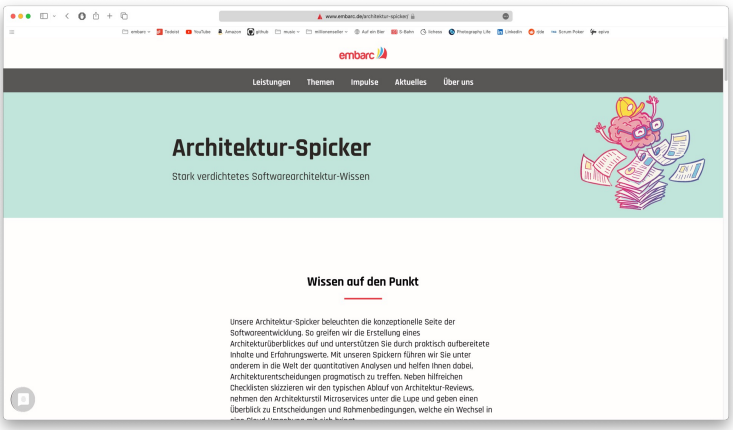


In verteilen Systemen sind die drei Anforderungen nach Konsistenz (Consistency), Verfügbarkeit (Availability), Ausfalltoleranz (Partition Tolerance) nicht vollständig vereinbar und nur maximal zwei von dreien sind erreichbar.

→ Eventual Consistency

59

Architektur-Spicker



embarc.de/architektur-spicker

60

Vielen Dank!

Was probiert ihr bei euch „zu Hause“ aus?

fk@embarc.de
linkedin.com/in/felix-kammerlander/

ak@embarc.de
linkedin.com/in/alexksbr/

61