CLOUD-NATIVE ARCHITEKTUREN

Unit 10:

Domain Driven Design

Stand: 11.03.21

TECHNISCHE HOCHSCHULE LÜBECK

1

INHALTSVERZEICHNIS

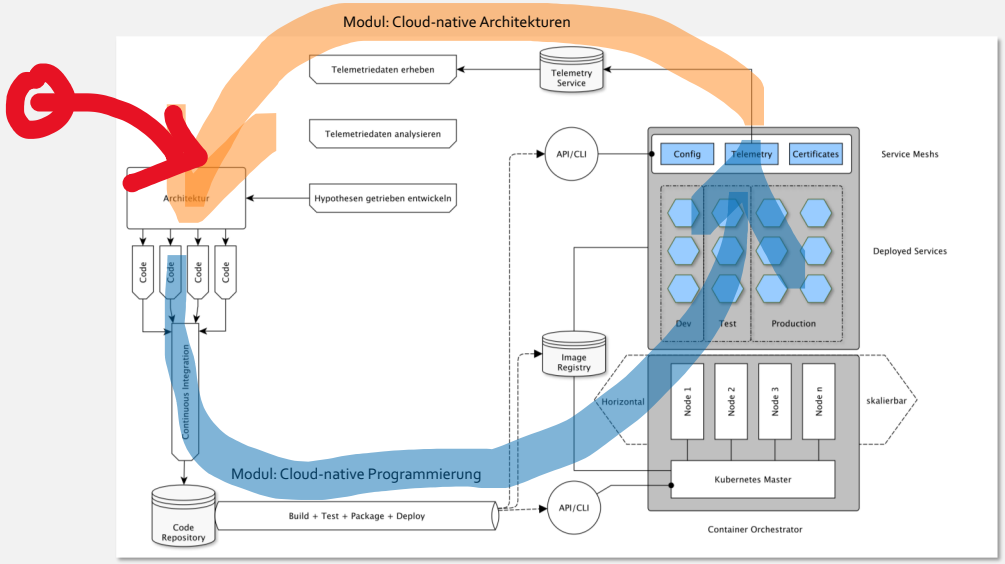
Überblick über Units und Themen dieses Moduls

TECHNISCHE HOCHSCHULE LÜBECK

2

DOMAIN DRIVEN DESIGN

Wo sind wir nun?



TECHNISCHE HOCHSCHULE LÜBECK

Prinzipien des Flow
(durch Automatisierung und Plattform Entwicklung beschleunigen)

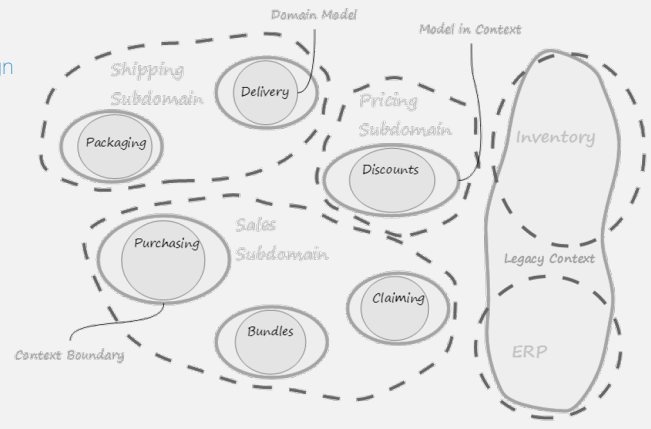
Prinzipien des Feedbacks
(durch Beobachtung Systeme verstehen und optimieren)

INHALTE

Domain Driven Design
- Was ist das?
- Effektives Software Design
- Strategisches Design
- Taktisches Design

Strategisches Design
- Subdomains
- Bounded Context + Ubiquitous Language
- Context Mapping

Taktisches Design
- Business Logic Pattern
- Architectural Pattern



TECHNISCHE HOCHSCHULE LÜBECK

DOMAIN DRIVEN DESIGN

Auf der Suche nach einer Methodik zur Entwicklung Cloud-nativer Architekturen

TECHNISCHE HOCHSCHULE LÜBECK

Wir erinnern uns an die Unit 07. Services in Cloud-nativen Systemen sollten u.a. folgenden Prinzipien folgen und folgende Eigenschaften haben:

- Dekomposition mittels Services
- Evolutionäres Design und Entwicklung des Gesamtsystems
- Lose Kopplung von Services (Standardformate, z.B. JSON/XML, Standarddatentypen, Messaging)
- Hohe Kohäsion innerhalb von Services (Single-Responsibility Prinzip)
- Services sollten unabhängig von einander austausch- und aktualisierbar sein
- Conways Law => Services sollten als langlebende Produkte und nicht Projekte als verstanden wissen (You build it, you run it)

Noch mehr als die Programmierung einzelner Services ist die Entwicklung einer tragfähigen Architektur eine „Kunst".

Im Zusammenhang mit Microservice Architekturen wird aber immer wieder eine Methodik erwähnt, die (mit einer höheren Wahrscheinlichkeit) Architekturen erzeugt, die gut auf die in Unit 7 genannten Erfordernisse abgestimmt sind.

Insbesondere jüngeren SW-Architekten sei daher die Methodik des Domain Driven Design (DDD) ans Herz gelegt, obwohl DDD weder auf Cloud-native Systeme oder Microservice Architekturen festgelegt ist und auch in anderen Kontexten der Entwicklung tragfähiger SW-Architekturen eingesetzt werden kann.

DOMAIN DRIVEN DESIGN

Fachlichkeit auf Basis eines Domänenmodells

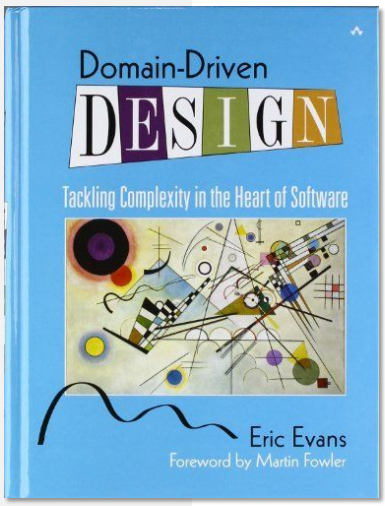
TECHNISCHE HOCHSCHULE LÜBECK

Domain-driven Design (DDD) ist eine Methodik zur Modellierung komplexer Software. Der Begriff „Domain-driven Design" wurde 2003 von Eric Evans in seinem gleichnamigen Buch geprägt (also deutlich bevor es den Begriff Cloud-native und entsprechende Systeme überhaupt gab).

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.

DDD orientiert sich an agiler Softwareentwicklung, ist aber SW-Entwicklungsprozess-agnostisch. Eine iterative Softwareentwicklung und eine enge Zusammenarbeit zwischen Entwicklern und Domänen-Experten ist jedoch erforderlich.



DOMAIN DRIVEN DESIGN

Effektives Design

- Domain-driven Design (DDD) strebt dabei nicht nach einem perfekten sondern nach einem effektiven Design.
- Effektives Design erfüllt die Anforderungen einer Firma/einer Organisation/einer Einrichtung bis zu einem Grad, der notwendig ist, damit man sich mittels digitalisierter Prozesse von Mitbewerbern/in Benchmarks abheben kann.
- Effektives Design zwingt Organisationen dazu, zu erkennen, worin sie sich hervortun müssen und sich auf diese Bereiche zu fokussieren.
- Und nur in diesen Bereichen wird effektives Design eingesetzt, um ein Domänen-konformes Softwaremodell zu entwickeln und fortzuschreiben.
- Andere nicht Domänen-spezifische Bereiche versucht man durch Standardsoftware oder mittels externer Services abzudecken.

„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

TECHNISCHE HOCHSCHULE LÜBECK

DOMAIN DRIVEN DESIGN

Fokus auf die Fachlichkeit

- Software dient zur Umsetzung von Geschäftsanforderungen (nicht umgekehrt)
- Fachlichkeit ist langlebiger als Technologien und bringt somit Stabilität in Softwarearchitekturen
- Fachlichkeit ist der gemeinsame Nenner aller am Entwicklungsprozess Beteiligten
- Domänenwissen begleitet die Softwareentwicklung, d.h. Softwarentwicklung startet und endet mit dem Wissen von Domänen-Experten

Bei DDD geht es um

- Sprache
- Zusammenarbeit
- Methoden und Verfahren
- Änderung von Denkmustern

TECHNISCHE HOCHSCHULE LÜBECK

DOMAIN DRIVEN DESIGN

Wie findet man methodisch raus was wichtig ist? => Strategisches Design

Strategisches Design

- Was ist wichtig für die Zielerreichung (das Geschäft)?
- Wie teilt man Arbeit anhand von Prioritäten auf?
- Wo fasst man Dinge zusammen?

Vorgehensweisen

- Domain Matter Experts mittels einer gemeinsamen Sprache einbeziehen (Ubiquitous Language)
- Komplexe Domänenmodelle aufteilen (mittels Bounded Contexts und Subdomains)
- Subdomains mittels Context Mapping und definierter Beziehungen integrieren

Daraus werden letztlich Services definiert und gem. Conways Law Teams gebildet

TECHNISCHE HOCHSCHULE LÜBECK

Nicht erschrecken: *DDD hat Ähnlichkeiten mit militärischen Vorgehensweisen.*

Strategisches Design schneidet Services anhand von Fachlichkeit.

DOMAIN DRIVEN DESIGN

Wie bricht man methodisch herunter was wichtig ist? => Taktisches Design

Taktisches Design

- „Der dünne Pinselstrich, um die feinen Details des Domänenmodells zu zeichnen" (Vaughn Vernon)
- Ausarbeitung des Domänenmodells in wertschöpfenden Bereichen

Vorgehensweisen

- Arbeiten im inneren eines Bounded Contexts
- Entities und Value Objekts in Aggregaten zusammenfassen
- Service Interaktionen über die Grenzen von Bounded Contexts mittels Domain Events modellieren und realisieren

Daraus werden letztlich Schnittstellen zwischen Services definiert

TECHNISCHE HOCHSCHULE LÜBECK

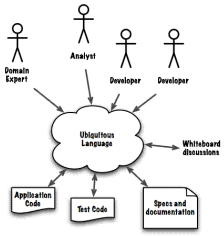
Nicht erschrecken: *DDD hat Ähnlichkeiten mit militärischen Vorgehensweisen.*

Taktisches Design arbeitet einzelne Services (Bounded Contexts) im Detail aus.

DOMAIN DRIVEN DESIGN

Ziele

- DDD beschreibt einen Prozess
- DDD liefert unterstützende Verfahren/Techniken
- DDD ist eine Philosophie
- Ausrichtung der Software-Entwicklung an der Fachlichkeit



Fachlichkeit und SW korrespondieren nachvollziehbar

- Durch die Methode
- In welchem SW-Baustein ist diese Fachlichkeit umgesetzt?
- Für welche Fachlichkeit ist dieser Baustein zuständig?

SW-Lösung skaliert mit der Systemgröße

- Integrationskomplexität einer fachlichen Änderung darf nicht mit der Größe eines Systems wachsen
- Wachsende Systeme bleiben wartbar

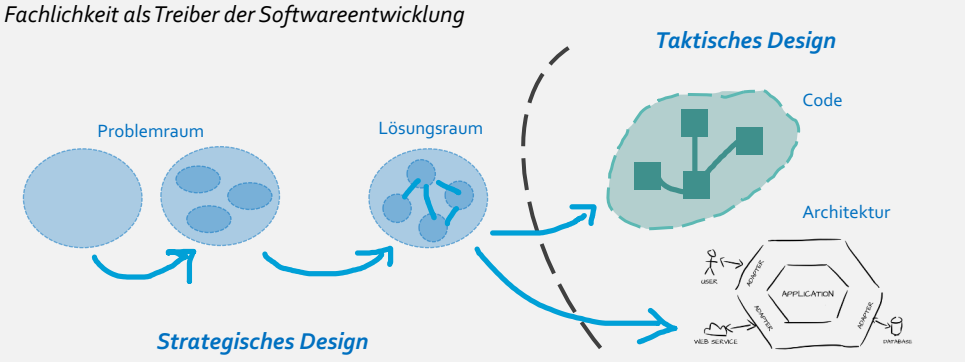
Zusammenarbeit steht im Mittelpunkt

- Fachexperten und Entwicklung arbeiten Hand in Hand
- Fachexperten und Entwicklung kommunizieren technologiefrei
- Die Zusammenarbeit von Fachexperten und Entwicklung vereint unterschiedliche Fähigkeiten

TECHNISCHE HOCHSCHULE LÜBECK

DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



Fachliche Kriterien bestimmen das Vorgehen

Allgemeingültige Sprache

Fachliche Durchgängigkeit

TECHNISCHE HOCHSCHULE LÜBECK

DOMAIN DRIVEN DESIGN

DDD is overrated

Ein Wort der „Warnung"

„There is a life beyond DDD. Not every good design needs to be Domain-Driven (though I can accept it should always be driven by the domain, just not necessarily in the DDD sense). ***You can design good systems even if you're not a DDD expert."***

Stefan Tilkov
Blog Post

Siehe auch: Stefan Tilkov, **GOTO 2019:** Good Enough Architectures, https://www.youtube.com/watch?v=PzEox3szeRc

TECHNISCHE HOCHSCHULE LÜBECK

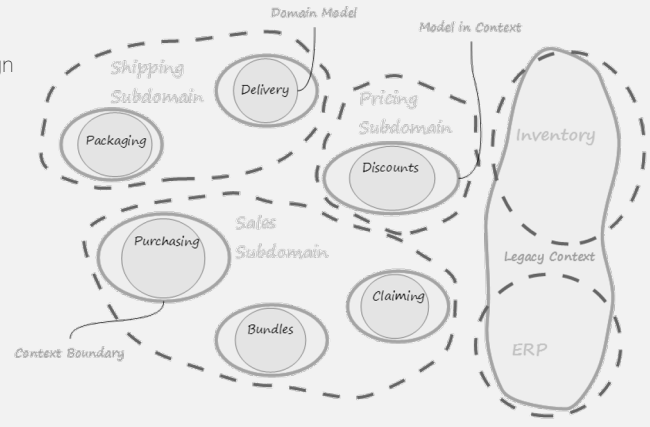
Stefan Tilkov
Geschäftsführer und Principal Consultant bei INNOQ; Strategische Beratung im Umfeld von Software-Architekturen

INHALTE

Domain Driven Design
- Was ist das?
- Effektives Software Design
- Strategisches Design
- Taktisches Design

Strategisches Design
- Subdomains
- Bounded Context + Ubiquitous Language
- Context Mapping

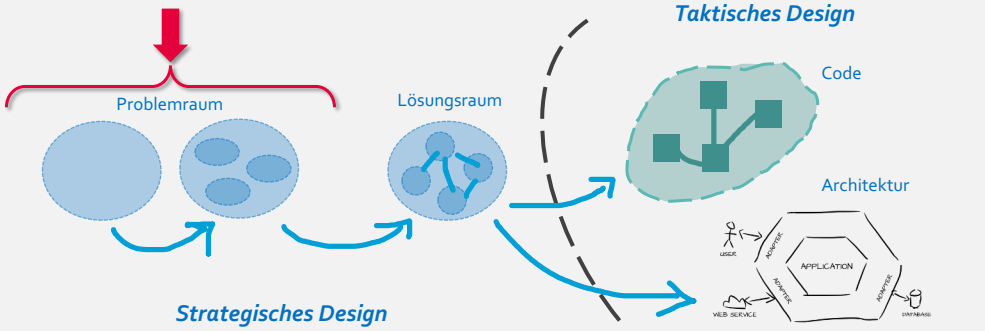
Taktisches Design
- Business Logic Pattern
- Architectural Pattern



TECHNISCHE HOCHSCHULE LÜBECK

DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



Fachliche Kriterien bestimmen das Vorgehen

Allgemeingültige Sprache

Fachliche Durchgängigkeit

TECHNISCHE HOCHSCHULE LÜBECK

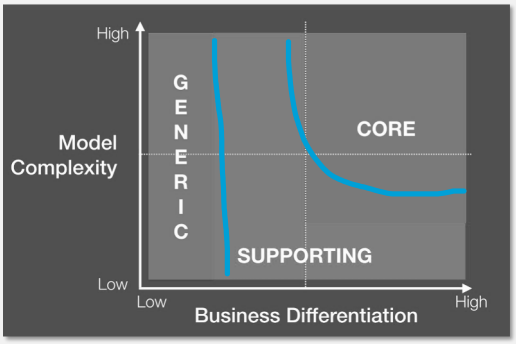
STRATEGISCHES DESIGN

Was ist eine Domäne?

Unter einer Domäne versteht man im allgemeinen einen bestimmten Tätigkeits- oder Wissensbereich.

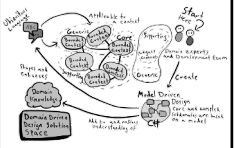
Das Domänenkonzept ist sehr breit und abstrakt. Um es konkreter und greifbarer zu machen, ist es sinnvoll, es in kleinere Teile aufzuteilen, die Subdomains genannt werden.

Solche Subdomains werden daher häufig in drei Kategorien unterteilt.



Alle Subdomänen, unabhängig von der Kategorie, sind wichtig für Gesamtlösungen. Fehlt eine Domäne wird die Lösung unvollständig sein. Die genannten Domänen erfordern jedoch einen unterschiedlichen Aufwand und können auch unterschiedliche Anforderungen an Qualität und Vollständigkeit haben.

TECHNISCHE HOCHSCHULE LÜBECK



- *Core Domain (Kerndomäne)*
- *Supporting Subdomain*
- *Generic Subdomain*

STRATEGISCHES DESIGN

Analyse von Business Domains

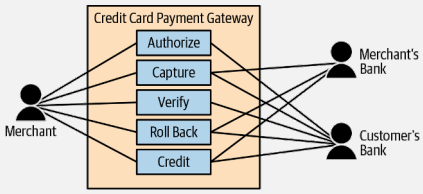
TECHNISCHE HOCHSCHULE LÜBECK

Ein Geschäftsfeld (Business Domain) definiert den gesamten Tätigkeitsbereich einer Organisation oder eines Unternehmens. Im Allgemeinen ist es die Dienstleistung, die Einrichtungen Kunden anbieten. Ein Beispiel:

- DHL => Paketdienst
- Starbucks => Kaffee
- Karstadt => Einzelhandel

Ein Unternehmen kann in mehreren Geschäftsbereichen tätig sein. Zum Beispiel bietet Amazon sowohl Einzelhandels- als auch Cloud-Computing-Dienste an.

Unternehmen wandeln ggf. ihre Geschäftsbereiche auch im Verlaufe der Zeit. Nokia war bspw. im Laufe seiner Geschichte in so unterschiedlichen Bereichen wie Holzverarbeitung, Gummiherstellung, Telekommunikation und Mobile Devices Geräte tätig.



Um die Ziele seiner Geschäftsdomäne zu erreichen, muss ein Unternehmen in mehreren Subdomänen operieren. Eine Subdomäne ist ein feingranularer Bereich der Geschäftsaktivität. Alle Subdomänen zusammen ergeben die Geschäftsdomäne des Unternehmens.

Oft korrelieren Subdomains mit den Abteilungen oder anderen Organisationseinheiten des Unternehmens. Zum Beispiel könnte ein Online-Handelsshop Subdomains wie Katalogmanagement, Werbung, Buchhaltung, Support, Kundenbeziehungen, Lieferantenbeziehungen und andere enthalten.

Aus technischer Sicht ähneln Subdomänen Sets von zusammenhängenden Anwendungsfällen. Vgl. Beispiel Kreditkartenzahlung.

STRATEGISCHES DESIGN

Was ist eine Core Subdomain? - Kerndomäne?

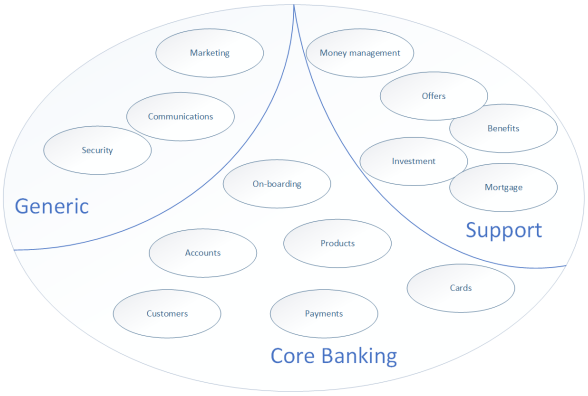
TECHNISCHE HOCHSCHULE LÜBECK

Eine **Kerndomäne** unterscheidet ein Unternehmen von seinen Konkurrenten. Dabei kann es sich um die Erfindung neuer Produkte oder Dienstleistungen handeln oder um die Optimierung bestehender Prozesse und damit um die Reduzierung von Kosten.

Eine einfach zu implementierende Kern-Subdomäne kann nur einen kurzlebigen Wettbewerbsvorteil bieten. Daher sind Kern-Subdomänen naturgemäß komplex. Üblichweise gibt es hohe Eintrittsbarrieren, die es Wettbewerbern schwer machen Lösungen des Unternehmens gleichwertig zu kopieren oder zu imitieren.

Üblicherweise investiert man im DDD System- und Softwareentwicklungsaufwände primär in die Kerndomäne. Kern-Subdomains müssen meist intern implementiert werden. Sie können nicht gekauft oder übernommen werden; andernfalls wären Wettbewerber einfach in der Lage, dasselbe zu tun.

Es wäre auch unklug, die Implementierung einer Core-Subdomain auszulagern.



STRATEGISCHES DESIGN

Was ist eine Supporting Subdomain?

TECHNISCHE HOCHSCHULE LÜBECK

Im Gegensatz zu den Kern-Subdomains bieten **unterstützende Subdomains** keinen Wettbewerbsvorteil für Unternehmen. Sie unterstützen lediglich das Kerngeschäft des Unternehmens.

Zu den Kern-Subdomänen eines Online-Werbeunternehmen gehören bspw. die Anpassung der Anzeigen an die Besucher, die Optimierung der Wirksamkeit der Anzeigen und die Minimierung der Kosten für die Werbeplätze. Hierfür muss das Unternehmen u.a. seine kreativen Materialien katalogisieren. Die Art und Weise, wie das Unternehmen seine Kreativmaterialien verwaltet, hat jedoch keinen nennenswerten Einfluss auf seine Gewinne. In diesem Bereich gibt es nichts zu erfinden oder zu optimieren.

Komplexität und Änderungsgeschwindigkeit

Der auffälligste Unterschied zwischen Kern- und unterstützenden Subdomains ist die Komplexität der Geschäftslogik. Unterstützende Subdomänen sind meist einfach. Ihre Geschäftslogik basiert häufig auf einfachen ETL-Operationen und sogenannten CRUD-Schnittstellen.

Im Gegensatz zu den Kern-Subdomänen ändern sich die unterstützenden Subdomänen auch nicht oft. Sie bieten keinen Wettbewerbsvorteil für das Unternehmen, und daher gibt es keinen geschäftlichen Wert, sie im Laufe der Zeit weiterzuentwickeln.

Implementierungsüberlegungen

Mangels Wettbewerbsvorteil ist es sinnvoll, unterstützende Subdomains nicht selbst zu implementieren. Anders als bei generischen Subdomains gibt es jedoch häufig keine vorgefertigten Lösungen. Sodass auch unterstützende Subdomains oft selbst implementiert werden müssen.

In diesem Bereich sind jedoch oft einfache Anwendungsentwicklungs-Framework ausreichend.

Diese Einfachheit der Geschäftslogik macht Supporting Subdomains zu einem naheliegenden Kandidaten für das Outsourcing.

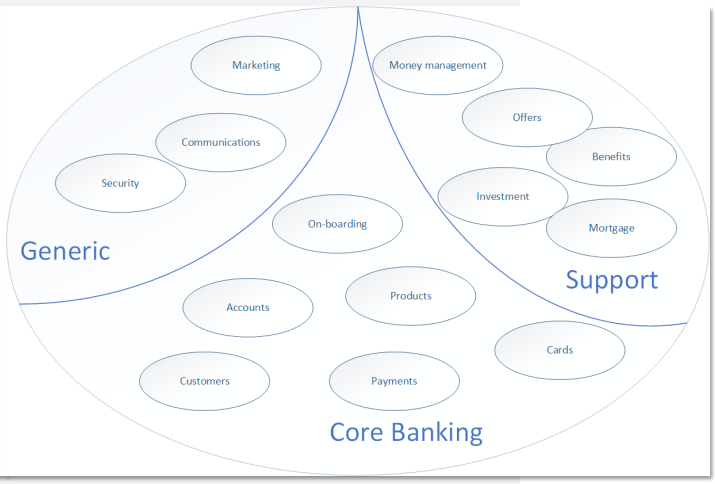
STRATEGISCHES DESIGN

Was ist eine Generic Subdomain?

TECHNISCHE HOCHSCHULE LÜBECK

Auch **generische Subdomains** sind in der Regel komplex und schwer zu implementieren. Allerdings bieten generische Subdomains keinen Wettbewerbsvorteil für das Unternehmen. Hier gibt es bereits bewährte und weit verbreitete Implementierungen, die von vielen Unternehmen genutzt werden. Generische Subdomains sind Geschäftsaktivitäten, die alle Unternehmen auf die gleiche Weise durchführen.
Üblicherweise versucht man für generische Subdomains Standardsoftware und –systeme zu verwenden. Ein typisches Beispiel wäre die Identitätsverwaltung mittels Microsofts Active Directory.

Anmerkung: *Ein und dieselbe Subdomain kann durchaus in verschiedene Kategorien fallen, je nachdem, was die Organisation tut. Für ein Unternehmen, das sich auf Identitätsmanagement spezialisiert hat, ist Identitätsmanagement eine Kerndomäne. Für ein Unternehmen, das sich hingeghen auf Customer Relationship Management spezialisiert hat, ist das Identitätsmanagement jedoch ein allgemeiner (technologisch austauschbarer) Teilbereich.*



STRATEGISCHES DESIGN

Kategorien von Domänen

TECHNISCHE HOCHSCHULE LÜBECK

	Core Subdomain	Supporting Subdomain	Generic Subdomain
Alleinstellungsmerkmal / Geschäftskritisch	ja	nein	nein
Unternehmensspezifisch	ja	ja	nein
Technische Komplexität (Einstiegshürde)	hoch	niedrig	mittel/ hoch
Technische Rate of Change	hoch	niedrig	niedrig
Existieren Lösungen / Produkte / Services	nein	teilweise	ja
Entwicklung In-House sinnvoll	ja	mögl.	nein
Out-Sourcing der Entwicklung ratsam	nein	mögl.	nein
Existieren COTS Produkte / Managed Services?	nein	teilweise	ja

BEISPIEL

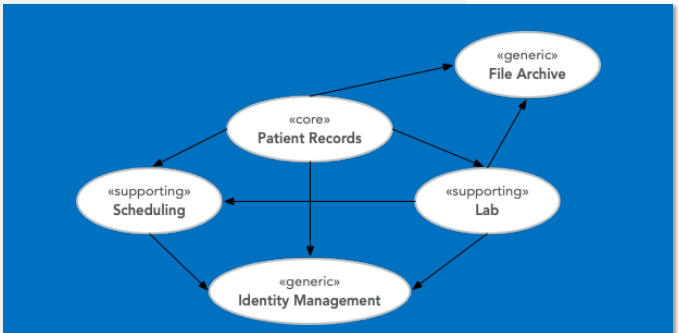
EMR-System (Electronic Medical Records)

TECHNISCHE HOCHSCHULE LÜBECK

Nehmen wir an, wir bauen ein EMR-System (Electronic Medical Records) für kleinere Kliniken für das die folgenden Subdomänen identifiziert worden sind:

- **Patient Records** für die Verwaltung der Patientenakten (persönliche Informationen, Anamnese, etc.).
- **Lab** für die Bestellung von Labortests und die Verwaltung der Testergebnisse.
- **Scheduling** für die Planung von Terminen.
- **File Archive** für die Speicherung und Verwaltung von Dateien, die den Patientenakten beigefügt sind (z. B. verschiedene Dokumente, Röntgenbilder, gescannte Papierdokumente).
- **Identitätsmanagement**, um sicherzustellen, dass die richtigen Personen Zugriff auf die richtigen Informationen haben.

Wie würden wir nun diese Subdomänen klassifizieren? Die offensichtlichsten sind das Dateiarchiv und das Identitätsmanagement, die eindeutig generische Subdomänen sind. Aber was ist mit den anderen? Das hängt davon ab, was dieses bestimmte EMR-System von den anderen auf dem Markt abheben soll.

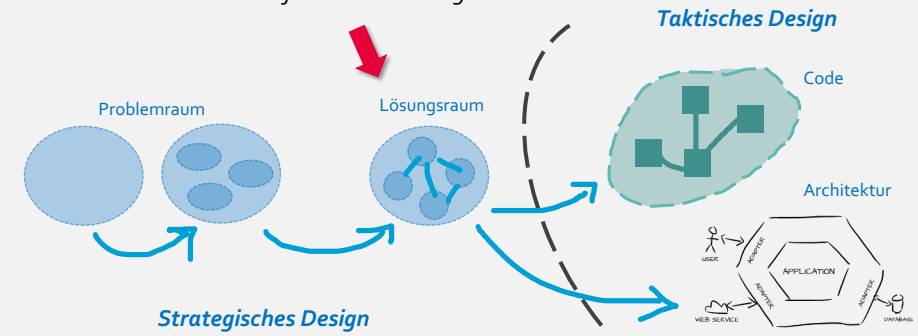


Bei einem EMR-System, kann man ziemlich sicher davon ausgehen, dass Patient Records eine Kerndomäne ist. Wenn allerdings ein EMR-System ebenfalls zum Ziel hat, Kliniken durch clevere und innovative Terminplanung effizienter zu machen, dann ist Scheduling ggf. auch eine Kerndomäne. Andernfalls ist es eine unterstützende Subdomäne, die durch eine bestehende Terminplanungs-Engine gut abdeckbar ist. Ähnlich bei der Subdomäne Labor: Sollte ein wesentlicher Teil unseres Business Case eine nahtlose Integration zwischen Patientenakten und Labor sein, dann ist das Labor höchstwahrscheinlich eine Kerndomäne. Andernfalls ist es eine unterstützende Subdomäne.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



Fachliche Kriterien bestimmen das Vorgehen

Allgemeingültige Sprache

Fachliche Durchgängigkeit

TECHNISCHE HOCHSCHULE LÜBECK

VOM PROBLEMRAUM ZUM LÖSUNGSRAUM

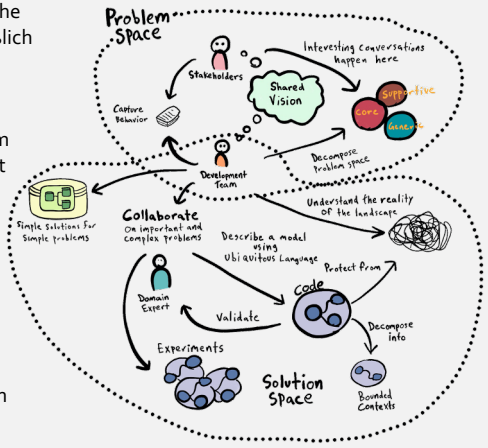
Ubiquitous Language

Manchmal wird die Domäne auch als "Problemraum" bezeichnet. Das kommt daher, dass die Domäne fachliche Probleme definiert, die eine Software lösen soll (schließlich gibt es einen Grund, warum die Software überhaupt entwickelt werden soll).

Vaughn Vernon unterscheidet daher einen Problemraum und einen Lösungsraum. Der Problemraum konzentriert sich auf die fachlichen Probleme, die wir zu lösen versuchen. Die genannten Subdomänen gehören in diesen Raum.

Der Lösungsraum konzentriert sich darauf, wie die Probleme technisch gelöst werden sollen. Er ist konkreter, technischer und enthält mehr Details.

Die Schwierigkeit ist wie man die Beziehungen zwischen dem fachlichen Problemraum und den Software-technischen Lösungsraum erhält.



TECHNISCHE HOCHSCHULE LÜBECK

Übliche SW-Entwicklungs-methodiken basieren meist implizit auf den folgenden Übersetzungen:

- *Domänenwissen in Analysemodell*
- *Analysemodell in Anforderungen*
- *Anforderungen in Systementwurf*
- *Systementwurf in Quellcode*

Bei jedem dieser Übergänge können Übersetzungsfehler passieren.

KODIFIZIERUNG VON DOMÄNENWISSEN

Ubiquitous Language

TECHNISCHE HOCHSCHULE LÜBECK

In vielen Softwareentwicklungsmodellen gibt es Tätigkeiten in denen Domänenwissen in eine ingenieursfreundliche Form "übersetzt" wird. Diese Artefakte werden meist als Analysemodell bezeichnet und umfassen eine Beschreibung der Anforderungen an das System und nicht ein Verständnis der dahinter liegenden Geschäftsdomäne.

Für den Wissensaustausch von Fachdomäne zur Entwicklung ist dies jedoch gefährlich. Bei jeder Übersetzung gehen Informationen verloren; in diesem Fall geht Domänenwissen, das für die Lösung von Geschäftsproblemen unerlässlich ist, auf dem Weg zu den Softwareingenieuren verloren.

Seit der Softwarekrise in den 1960ern haben viele Untersuchungen das Scheitern von Software-Projekten untersucht. Fast alle haben gezeigt, dass u.a. Kommunikation für den Projekterfolg unerlässlich ist.

Fast alle Softwareentwicklungsmodelle versuchen daher Domänenwissen von Domänenexperten an die Ingenieure weiterzugeben. Meist geschieht dies durch Vermittlerrollen, die Wissen von der Fachdomäne an die Entwicklung „durchreichen" und übersetzen: System-/Business-Analysten, Product Owner, Projektmanager, usw..

Das impliziert aber nicht automatisch eine **effektive**, d.h. auf Verständnis beruhende, Kommunikation.

Domain-driven Design versucht vor diesem Hintergrund das Wissen von Domänenexperten zu Softwareingenieuren mittels einer allgegenwärtigen – und gemeinsam entwickelten – Sprache zu bringen, der sogenannten Ubiquitous Language.

STRATEGISCHES DESIGN

Konzepte im Problemraum - Ubiquitous Language

TECHNISCHE HOCHSCHULE LÜBECK

Gemeinsame Sprache ist Schlüssel zum gemeinsamen Verständnis

- Aufwändiger aber notwendiger Schritt
- Klärung der fachlichen Begriffe
- Klarheit der gemeinsamen Sprache ist Indikator für die Tiefe des Verständnisses des Problemraums.
- Unklare Begriffe deuten auf tieferliegende und verdeckte (ggf. unverstandene) Konzepte hin.
- Hierzu müssen implizite Annahmen explizit gemacht werden um eine eindeutige Bedeutung innerhalb der Ubiquituous Language festlegen zu können.

Ubiquitär = „allumfassend, überall vorhanden, allgegenwärtig"

- Jeder im Projekt muss dieselbe Sprache sprechen und verstehen können
- Alle relevanten Konzepte müssen in der Sprache beschrieben werden können
- Die Sprache wird evolutionär weiterentwickelt und ist in allen Phasen der Entwicklung präsent (auch im Code!)

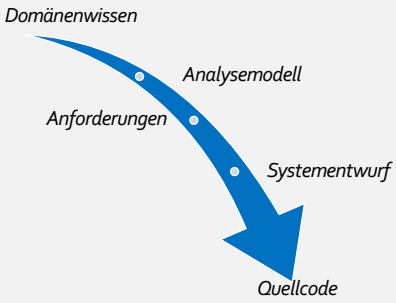
Eines der wichtigsten Konzepte in DDD

- Grundlage für gemeinsames Verständnis
- Grundlage für durchgängiges Verständnis
- Indikator für Durchdringung und Beschreibungsstabilität der Domäne

UBIQUITOUS LANGUAGE

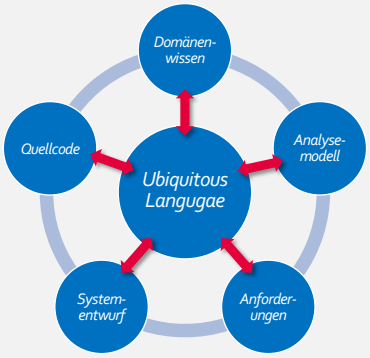
TECHNISCHE HOCHSCHULE LÜBECK

Non - DDD



Anstatt das Domänenwissen ständig zu übersetzen, fordert DDD die Kultivierung einer einzigen Sprache zur Beschreibung der Geschäftsdomäne: die allgegenwärtige Sprache.

DDD



Durch die durchgängige Verwendung der allgegenwärtigen Sprache und ihrer Begriffe kann ein gemeinsames Verständnis zwischen allen Projektbeteiligten kultiviert werden. Die allgegenwärtige Sprache sollte daher nur aus Begriffen bestehen, die sich auf die Geschäftsdomäne beziehen.

Ziel ist es nicht, Domänenexperten etwas über SW-Entwicklung (z.B. Singletons und abstrakte Fabriken) beizubringen.

Der Zweck der allgegenwärtigen Sprache ist es, das Verständnis und die mentalen Modelle der Geschäftsdomäne der Domänenexperten in leicht verständliche Begriffe für die Entwicklung zu fassen.

UBIQUITOUS LANGUAGE

TECHNISCHE HOCHSCHULE LÜBECK

So

Angenommen, wir arbeiten an einem System zur Verwaltung von Online Werbekampagnen. Folgende Aussagen sind in der Sprache von Marketing-Domänenexperten formuliert.

- Eine Werbekampagne kann verschiedene kreative Materialien anzeigen.
- Eine Kampagne kann nur veröffentlicht werden, wenn mindestens eine ihrer Platzierungen aktiv ist.
- Verkaufsprovisionen werden für genehmigte Transaktionen abgerechnet.

So nicht

Die folgenden Aussagen sind rein technischer Natur und werden für Domänenexperten vermutlich unklar sein. Sie gehören nicht in eine Ubqiquitous Language.

- Der Anzeigen-iframe zeigt eine HTML-Datei an.
- Eine Kampagne kann nur veröffentlicht werden, wenn sie mindestens einen zugehörigen Datensatz in der Tabelle active-placements hat.
- Die Verkaufsprovisionen basieren auf korrelierten Datensätzen aus den Tabellen transactions und approved sales.

DDD == Fachlichkeit, Fachlichkeit, Fachlichkeit

UBIQUITOUS LANGUAGE

Mehrdeutige und synonyme Begriffe

TECHNISCHE HOCHSCHULE LÜBECK

Mehrdeutige Begriffe

Der englische Begriff "Policy" hat mehrere Bedeutungen: Er kann eine gesetzliche Vorschrift oder einen Versicherungsvertrag bedeuten. Die genaue Bedeutung ergibt sich meist aus dem Kontext. Problematisch sind Kontexte in denen bspw. Versicherungsvertäge aufgrund gesetzlicher Vorschriften erforderlich sind. Welches Konzept meint Policy nun? Mit Versicherungen haben wir eine Domäne in der beide Konzepte zeitgleich vorkommen.

Ubiquitäre Sprache verlangt für solche Fälle eine einzige Bedeutung für jeden Begriff, und daher sollte "policy" explizit mit den beiden Begriffen "Regulierungsvorschrift" oder "Versicherungsvertrag" modelliert werden (aber nie Policy genannt werden).

Synonyme Begriffe

In DDD können zwei Begriffe nicht austauschbar verwendet werden. Zum Beispiel verwenden viele Systeme den Begriff "Benutzer". Der Begriff "Benutzer" wird im normalen Sprachgebrauch jedoch häufig austauschbar verwendet: z. B. "Benutzer", "Besucher", "Administrator", "Konto" usw.

Synonyme Begriffe bezeichnen meist unterschiedliche Feinkonzepte. Bspw. beziehen sich die Begriffe "Besucher" und "Konto" technisch gesehen auf die Benutzer des Systems; in den meisten Systemen stellen jedoch nicht registrierte und registrierte Benutzer unterschiedliche Rollen dar und haben unterschiedliche Verhaltensweisen. Daten von "Besuchern" werden meist zu Analysezwecken verwendet, während "Konten" das System und seine Funktionen tatsächlich nutzen.

Es ist vorzuziehen, jeden Begriff explizit und in seinem spezifischen Kontext zu verwenden. Ein Verständnis der Unterschiede zwischen den verwendeten Begriffen ermöglicht die Erstellung einfacherer und klarerer Modelle und Implementierungen der Entitäten der Geschäftsdomäne.

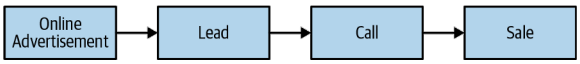
BOUNDED CONTEXTS

Begriffs-Komplexitäten beherrschen

TECHNISCHE HOCHSCHULE LÜBECK

Wie wir gesehen haben, ist es für den Erfolg eines Projekts wichtig, eine Ubiquitous Language zu entwickeln, die klar und konsistent sein und die mentalen Modelle der Domänenexperten widerspiegeln muss.

Es ist aber nicht ungewöhnlich, dass mentale Modelle von Domänenexperten selbst nur in einem spezifischen Kontext konsistent sind. Erweitert man den Kontext entstehen Inkonsistenten mit anderen Kontexten.



Angenommen, wir arbeiten für ein Telemarketing-Unternehmen. Die Marketingabteilung des Unternehmens generiert Leads durch Online-Anzeigen. Die Vertriebsabteilung ist dafür zuständig, potenzielle Kunden zum Kauf der Produkte oder Dienstleistungen zu bewegen. Beide Abteilungen haben ein unterschiedliches Verständnis eines Leads.

Marketing-Abteilung

Im Marketing stellt ein Lead eine Benachrichtigung dar, dass jemand an einem der Produkte interessiert ist. Das Ereignis des Erhalts der Kontaktdaten des potenziellen Kunden wird als Lead bezeichnet.

Vertriebsabteilung

Im Vertrieb repräsentiert ein Lead den gesamten Lebenszyklus eines Verkaufsprozesses. Er ist kein bloßes Ereignis, sondern ein lang andauernder Prozess.

Insbesondere Enterprise Architeture Management Ansätze haben oft den Anspruch eine globale Ubiquitous Language für ein ganzes Unternehmen zu entwickeln. Häufig scheitern diese Ansätze jedoch.

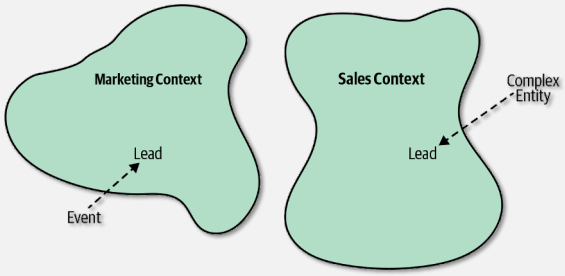
Der Autor hat als EA-Architektekt versucht die Bundeswehr bei der Entwicklung solcher Modelle für die vernetzte Operationsführung zu unterstützen (d.h. Heer, Luftwaffe und Marine). Die Ansätze scheiterten häufig schon in frühen Phasen an eigentlich gut verstanden gemeinten Begrifflichkeiten wie Operation, Einheit, Gruppe, Task-Force, usw.

BOUNDED CONTEXT

Was ist ein Bounded Context?

Die Lösung zum Umgang mit solchen Problemen ist letztlich trivial. Man hat nicht zum Ziel eine einzige allumfassende Ubiquititious Language zu entwickeln, sondern mehrere kleinere!

Man teilt hierzu die allgegenwärtige Sprache in mehrere kleinere Sprachen auf und ordnet dann jede Sprache dem expliziten Kontext zu, in dem sie angewendet werden kann - ihrem begrenzten Kontext (Bounded Kontext).



In unserem Beispiel können wir zwei begrenzte Kontexte identifizieren: Marketing und Vertrieb. Der Begriff "Lead" existiert in beiden Bounded Contexts.

Solange ein „Lead" in jedem begrenzten Kontext nur eine einzige Bedeutung hat, bleiben die Marketing und Vertriebs-Ubiquitous Languages konsistent und folgt den mentalen Modellen der Domänenexperten.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

TECHNISCHE HOCHSCHULE LÜBECK

Würde man nicht so vorgehen, wäre es erforderlich ein riesiges Modell zu konzipieren und abzustimmen.

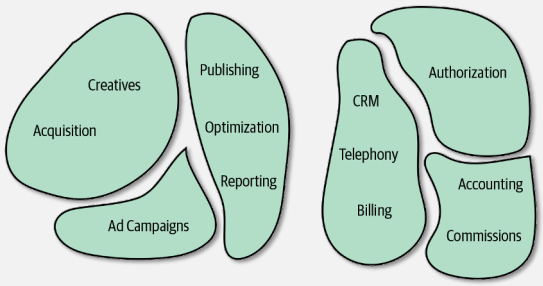
Bis man damit fertig ist, wäre das Geschäftsfeld vermutlich schon von einem Wettbewerber besetzt worden.

Insbesondere EAM-Ansätze scheitern immer wieder an dieser Stelle.

BOUNDED CONTEXT

Umfang eines Bounded Contexts

Die Größe eines Bounded Contexts ist für sich genommen kein entscheidender Faktor. Modelle sollten nie per se groß oder klein sein. Modelle müssen klar sein und einem Zweck folgen. Je umfassender die Grenze einer Ubiquitous Language ist, desto schwieriger wird es, sie konsistent zu halten. Daher hängt die Entscheidung, wie groß ein Bounded Context sein soll, von der spezifischen Problemdomäne ab.



- Es kann von Vorteil für die Modellierung sein, eine große Ubiquitous Languages in kleinere, besser handhabbare Problemdomänen aufzuteilen.
- Das Streben nach kleinen Bounded Contexts wird aber auch mehr Integrations-Overhead erzeugen, wie wir noch sehen werden.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

TECHNISCHE HOCHSCHULE LÜBECK

BOUNDED CONTEXT

Bounded Contexts vs. Subdomains

Subdomains

Um das Geschäft eines Unternehmens zu verstehen, müssen wir seine Geschäftsdomäne analysieren. Gemäß DDD beinhaltet die Analysephase die Identifizierung der verschiedenen Arten von Subdomains (Core, Supporting, Generic).

"Identifikation" ist hier das Schlüsselwort. Die Subdomänen sind bereits vorhanden; sie sind ein Teil des Geschäfts. **Durch Analyse entdecken wir Subdomänen, die bereits vorhanden sind.** Wir designen sie nicht!

Diese Subdomains helfen uns aber dabei Schwerpunkte zu setzen und begrenze Mittel primär in den Core-Subdomains einzusetzen (z.B. zur Entwicklung einer Ubiquitous Language in einem Bounded Context).

Bounded Contexts

Bounded Contexts hingegen werden entworfen. Die Wahl der Grenzen von Modellen ist eine strategische Design-Entscheidung. Wir entscheiden, wie wir die Geschäftsdomäne in kleinere, überschaubare Problemdomänen unterteilen.

TECHNISCHE HOCHSCHULE LÜBECK

Wir haben gesehen, dass eine Geschäftsdomäne aus mehreren Subdomains besteht. Wir haben uns ferner mit der Zerlegung einer Geschäftsdomäne in eine Reihe von feingranulareren Problemdomänen (Bounded Contexts) beschäftigt. Beide Methoden erscheinen irgendwie redundant. Doch ist das so?

CONTEXT MAPPING

Relationen zwischen Bounded Contexts

Modelle in verschiedenen Bounded Contexts können unabhängig voneinander entwickelt und implementiert werden. Bounded Contexts sind also nicht isoliert, da ein System kann nicht aus unabhängigen Komponenten aufgebaut werden kann; die Komponenten müssen miteinander interagieren, um die übergreifenden Systemziele zu erreichen.

Dies gilt natürlich auch für Bounded Contexts. Obwohl sich ihre Implementierungen unabhängig voneinander entwickeln können sollen, müssen sie miteinander integriert werden. Infolgedessen wird es immer Berührungspunkte geben, die als Verträge bezeichnet werden.

Beziehungen zwischen Bounded Contexts

Zwei Bounded Contexts verwenden per Definition unterschiedliche Ubibiquituous Languages. Welche Sprache soll für die Integration also verwendet werden? Diese Integrationsbelange sollten beim Entwurf der Lösung berücksichtigt werden.

Hierfür gibt es Domain-Driven Design's Patterns zur Handhabe von Beziehungen und Integrationen zwischen Bounded Contexts. Diese Pattern werden meist in drei Gruppen unterteilt, die jeweils eine Art der Teamzusammenarbeit repräsentieren:

- Kooperation
- Customer-Supplier
- Separate Ways

TECHNISCHE HOCHSCHULE LÜBECK

CONTEXT MAPPING

Cooperation Patterns

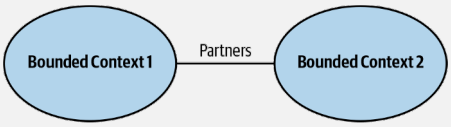
Kooperationsmuster beziehen sich auf Bounded Contexts, die von Teams mit gut etablierter Kommunikation implementiert werden.

Diese Anforderung ist automatisch für Bounded Contexts erfüllt, die von demselben Team implementiert werden.

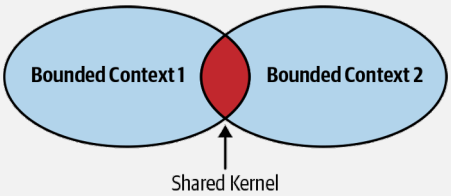
Diese Pattern sind aber auch für Teams mit abhängigen Zielen geeignet, bei denen der Erfolg des einen Teams von dem des anderen abhängt und umgekehrt.

Auch hier ist das Hauptkriterium die Qualität der Kommunikation und Zusammenarbeit der Teams.

Partnership



Shared Kernel



Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

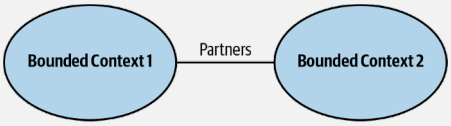
TECHNISCHE HOCHSCHULE LÜBECK

CONTEXT MAPPING

Cooperation Patterns – Partnership

Im Partnerschaftsmodell wird die Integration zwischen Bounded Context ad hoc – also anlassbezogen -- koordiniert. Ein Team kann ein zweites über eine Änderung in der API benachrichtigen, und das zweite Team wird kooperieren und erforderliche Anpassung unmittelbar vornehmen.

Die Koordination der Integration erfolgt hier in beide Richtungen. Kein Team diktiert die Sprache, die für die Definition der Verträge verwendet wird. Die Teams können Unterschiede gemeinsam bewerten und am besten geeignete Lösung wählen. Außerdem kooperieren beide Seiten bei der Lösung von Integrationsproblemen. Keines der Teams ist daran interessiert, das andere zu blockieren.



Gut eingespielte Praktiken der Zusammenarbeit, ein hohes Maß an Engagement und häufige Synchronisationen zwischen den Teams sind für dieses Integrationsmuster erforderlich.

Dieses Muster ist meist nicht für geografisch verteilte Teams geeignet, da es zu Synchronisations- und Kommunikationsproblemen führen kann.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

TECHNISCHE HOCHSCHULE LÜBECK

CONTEXT MAPPING

Cooperation Patterns – Shared Kernel

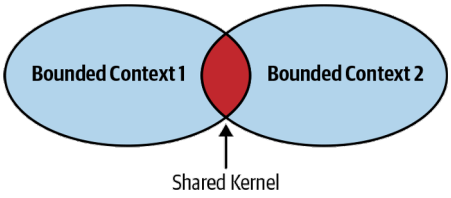
TECHNISCHE HOCHSCHULE LÜBECK

Der gemeinsame Kernel ist eine formalere Art, einen Vertrag zwischen mehreren Bounded Contexts ohne Machtgefälle zu definieren. Anstelle von Ad-hoc-Integrationen wird hier der Vertrag explizit in einer kompilierten Bibliothek - dem gemeinsamen Kernel - definiert.

Gemeinsam genutzte und entwickelte Libraries definieren somit die Integrationsmethoden und die Sprache, die von beiden Bounded Contexts verwendet werden.

Shared Kernel, widersprechen in gewisser Weise einem Kernprinzip von gebundenen Kontexten: Nur ein Team soll einen Bounded Context besitzen. Der gemeinsam genutzte Shared Kernelt ist allerdings im gemeinsamen Besitz mehrerer Teams.

Der Schlüssel zur Implementierung des Shared-Kernel-Musters besteht darin, den Umfang des Shared-Kernels klein zu halten und nur auf den Integrationsvertrag zu beschränken.



Der Shared Kernel wird von mehreren Bounded Contexts sowohl referenziert als auch besessen. Jedem Team steht es frei, den Shared Kernel zu ändern. Eine Änderung des Vertrags kann jedoch den Build des anderen Teams unterbrechen; daher erfordert auch dieses Muster ein hohes Maß an Synchronisation zwischen den Teams.

Shared Kernel können eine disziplinierende Voraussetzung für die Integration von Bounded Contexts sein, die vom gleichen Team betrieben und implementiert werden. In einem solchen Fall können Ad-hoc-Integrationen der Bounded Contexts die Grenzen der Kontexte im Laufe der Zeit „verwaschen". Ein Shared Kernel kann hier für die explizite Definition des Integrationsvertrags verwendet werden. In diesen Szenarien wird auch das Prinzip des Kontext-Besitzes durch ein Team nicht gebrochen - beide Bounded Kontexts werden vom selben Team implementiert.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

CONTEXT MAPPING

Customer-Supplier Patterns

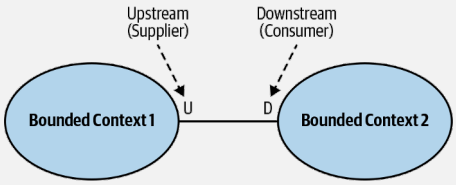
TECHNISCHE HOCHSCHULE LÜBECK

Die zweite Gruppe von Kooperationsmustern, sind die sogenannten Customer-Supplier-Pattern.

Anders als im Fall der Kooperation können beide Teams (Upstream und Downstream) unabhängig voneinander erfolgreich sein.

Daher gibt es in vielen Fällen ein Machtungleichgewicht: Entweder das vorgelagerte oder das nachgelagerte Team kann den Integrationsvertrag diktieren.

DDD sieht die folgenden drei Muster vor, solche Machtungleichheiten zu adressieren.



- Conformist Pattern
- Anti-Corruption Layer
- Open-Host Service

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

CONTEXT MAPPING

Customer-Supplier Patterns – Conformist Pattern

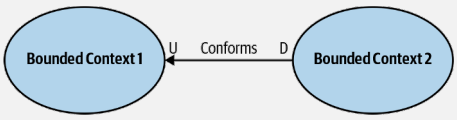
TECHNISCHE HOCHSCHULE LÜBECK

In einigen Fällen ist das Kräfteverhältnis zugunsten des Upstream Teams, das keine wirkliche Motivation hat, die Bedürfnisse seiner Kunden zu unterstützen.

Stattdessen stellt es nur den Integrationsvertrag zur Verfügung, der nach seinem eigenen Modell definiert ist - take it or leave it.

Solche Machtungleichgewichte existieren häufig bei unternehmensexternen Service Providern, können aber auch durch interne Organisationspolitik verursacht werden.

Wenn das Downstream Team das Modell des Upstream Teams akzeptieren kann, wird die Beziehung zwischen den Bounded Contexts als konformistisch bezeichnet.



Das Downstream Team ist konform mit dem Modell des Upstream Teams.

Die Entscheidung des Downstream-Teams, einen Teil seiner Autonomie aufzugeben, kann durchaus sinnvoll sein. Zum Beispiel kann der vom Upstream-Team offengelegte Vertrag ein branchenübliches, gut etabliertes Modell sein.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

CONTEXT MAPPING

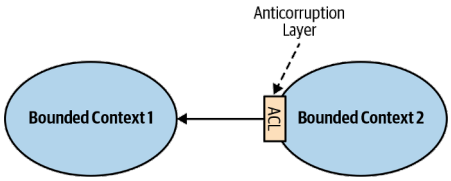
Customer-Supplier Patterns – Anti Corruption Layer

TECHNISCHE HOCHSCHULE LÜBECK

Das nächste Muster behandelt Fälle, in denen ein Consumer nicht bereit ist, das Modell des Supplier vorbehaltlos zu akzeptieren.

Wie im Fall des konformistischen Musters ist das Machtgleichgewicht in dieser Beziehung immer noch zugunsten des Upstream Services verschoben. In diesem Fall ist der Downstream Bounded Context jedoch nicht bereit, sich anzupassen.

Stattdessen wird das Modell des Upstream Bounded Contexts über eine Antikorruptionsschicht in ein Modell übersetzt, das auf die Bedürfnisse des eigenen Bounded Contexts zugeschnitten ist.



Das Anticorruption Layer Pattern adressiert Szenarien, in denen es nicht wünschenswert oder den Aufwand wert ist, dem Modell des Suppliers vollständig zu entsprechen

- Wenn der nachgelagerte Bounded Context eine Core-Subdomäne enthält – also sehr spezifisch ist. Das Modell einer Core-Subdomäne erfordert besondere Aufmerksamkeit, und die Einhaltung des Modells des Suppliers könnte die Modellierung der eigenen Problemdomäne behindern.
- Wenn das vorgelagerte Modell schlecht oder unpassend ist und ggf. über Jahrzehnte gewachsen ist. Dies ist häufig bei der Integration von Altsystemen der Fall.
- Wenn sich der Vertrag des Suppliers häufig ändert und der Consumer sein Modell vor solchen häufigen Änderungen schützen möchte. Mit einer Antikorruptionsschicht wirken sich die Änderungen im Modell des Lieferanten nur auf den Übersetzungsmechanismus aus.
- Wenn das Modell des Suppliers sehr groß ist und nur Teile dieses Modells benötigt werden.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

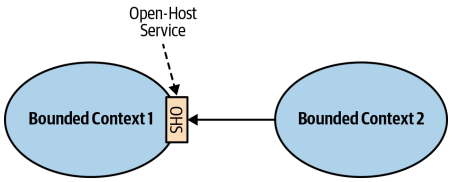
CONTEXT MAPPING

Customer-Supplier Patterns – Open Host Service

TECHNISCHE HOCHSCHULE LÜBECK

Dieses Muster befasst sich mit dem Fall, dass die Macht in Richtung der Consuming Services verzerrt ist. Der Supplier ist daran interessiert, seine Consumer zu schützen und den bestmöglichen Service zu bieten.

Um die Verbraucher vor Änderungen an seiner Implementierung zu schützen, entkoppelt der Upstream-Supplier sein Implementierungsmodell von der öffentlichen Schnittstelle. Diese Entkopplung ermöglicht es dem Supplier, sein Implementierungsmodell und sein öffentliches Modell mit unterschiedlicher Geschwindigkeit weiterzuentwickeln



Die öffentliche Schnittstelle des Anbieters ist also nicht dazu gedacht, seiner eigenen Ubiquitous Language zu entsprechen, sondern soll für die Consumer ein bequemes Protokoll bereitstellen, das in einer integrationsorientierten Sprache ausgedrückt ist. Daher wird das öffentliche Protokoll als "veröffentlichte Sprache" (Published Language) bezeichnet.

In gewissem Sinne ist das Open-Host-Service-Muster eine Umkehrung des Musters der Antikorruptionsschicht: Anstelle des Consumers implementiert der Supplier die Übersetzung seines internen Modells.

Es ist von entscheidender Bedeutung, dass die veröffentlichte Schnittstelle gut dokumentiert und für die Consumer bequem ist. Swagger und verwandte Lösungen sind eine gute Option für die Bereitstellung einer solchen Dokumentation. Diese Lösungen werden gerne bei der Dokumentation von REST-basierten APIs eingesetzt und ermöglichen auch auch die Generierung des Modells für die veröffentlichte Sprache, die der Dienstanbieter implementieren muss.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

CONTEXT MAPPING

Separate Ways Pattern

TECHNISCHE HOCHSCHULE LÜBECK

Die letzte Möglichkeit der Zusammenarbeit ist gar nicht zusammenzuarbeiten. Dieses Muster kann aus verschiedenen Gründen auftreten. Insbesondere aber in Fällen, in denen die Teams nicht bereit oder in der Lage sind, zusammenzuarbeiten. Wir werden uns hier ein paar davon ansehen.

Niemals bei Core Domains!

Das Separate Ways Pattern sollte bei der Integration von Core-Subdomains auf alle Fälle vermieden werden. Eine doppelte Implementierung solcher Subdomains würde die Strategie des Unternehmens, diese möglichst effektiv und optimiert zu implementieren, konterkarieren.

(1) Probleme mit der Kommunikation

Ein häufiger Grund für die Vermeidung von Zusammenarbeit sind Kommunikationsschwierigkeiten, die durch die Größe der Organisation oder interne politische Probleme verursacht werden. In diesen Fällen kann es kostengünstiger sein, Funktionalität in mehreren Bounded Contexts zu duplizieren.

(2) Generic Subdomains (Libraries)

Wenn die fragliche Subdomäne generisch ist, kann es, wenn die generische Lösung einfach zu integrieren ist, kostengünstiger sein, sie in jedem der Bounded Contexts lokal zu integrieren. Ein Beispiel ist ein Logging-Framework; es würde wenig Sinn machen, es als Service bereitzustellen. Die Duplizierung der Funktionalität in mehreren Bounded Contexts ist letztlich meist weniger kostspielig als die Zusammenarbeit.

(3) Modellunterschiede

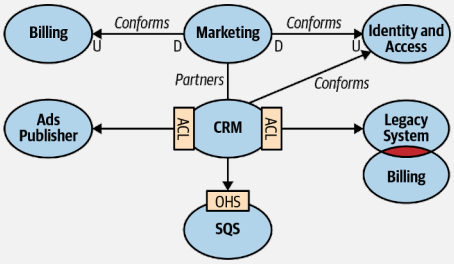
Wenn Modelle von Bounded Contexts so unterschiedlich sind, dass eine konforme Beziehung nicht möglich ist, und die Implementierung einer Antikorruptionsschicht teurer wäre als die Duplizierung der Funktionalität. Auch in einem solchen Fall ist es für die Teams kostengünstiger, getrennte Wege zu gehen.

CONTEXT MAPPING

Context Maps

Context Maps sind eine visuelle Top-Level Darstellung aller Bounded Contexts eines Systems. Diese erstaunlich einfache visuelle Notation gibt bereits wertvolle strategische Einblicke auf mehreren Ebenen:

- **High-Level-Design:** Eine Context Map bietet einen Überblick über die Komponenten und Modelle des Systems.
- Es werden **Kommunikationsmuster** zwischen den Teams dargestellt und welche Teams wie zusammenarbeiten.
- **Organisatorische Fragen:** Context Maps liefern Einblick in organisatorische Fragen. Was bedeutet es, wenn Downstream Consumer eines bestimmten Upstream Teams alle auf Antikorruptionsschichten zurückgreifen oder wenn sich alle Separate Ways auf dasselbe Team konzentrieren?



Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

TECHNISCHE HOCHSCHULE LÜBECK

INHALTE

Domain Driven Design

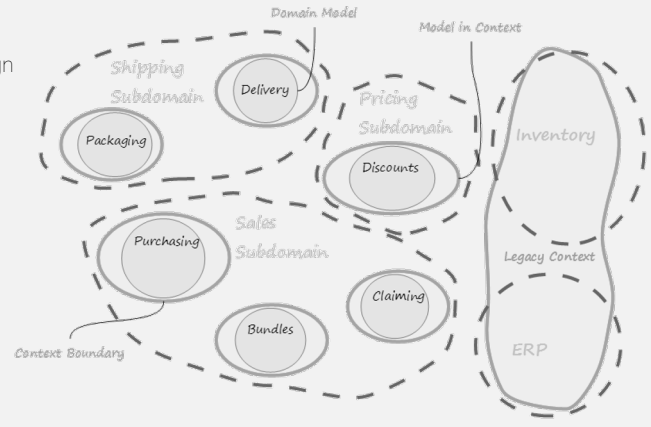
- Was ist das?
- Effektives Software Design
- Strategisches Design
- Taktisches Design

Strategisches Design

- Subdomains
- Bounded Context + Ubiquitous Language
- Context Mapping

Taktisches Design

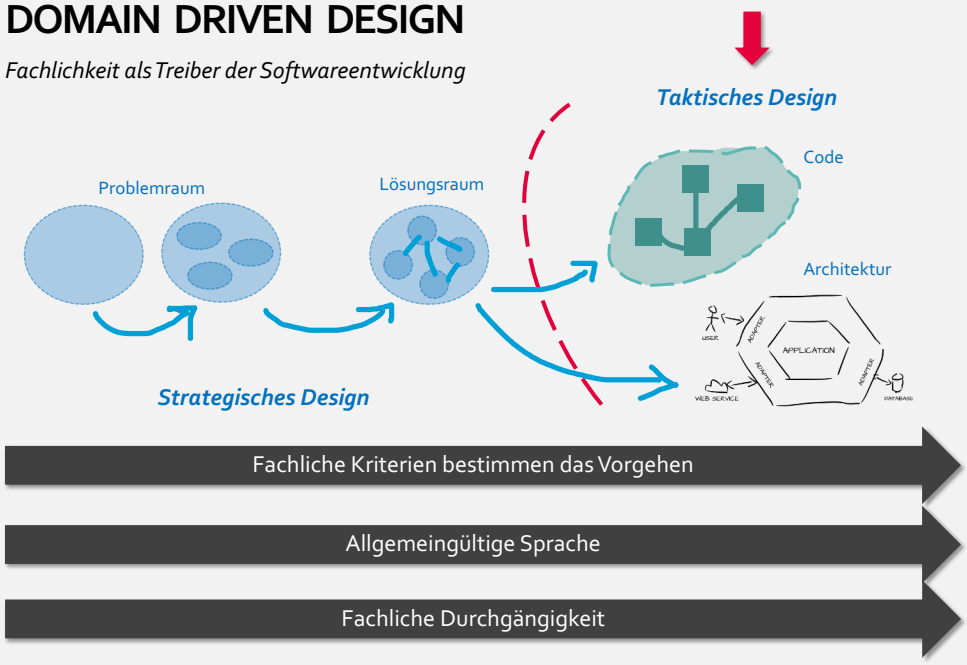
- Business Logic Pattern
- Architectural Pattern



TECHNISCHE HOCHSCHULE LÜBECK

DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



Fachliche Kriterien bestimmen das Vorgehen

Allgemeingültige Sprache

Fachliche Durchgängigkeit

TECHNISCHE HOCHSCHULE LÜBECK

TACTICAL DESIGN

Bislang haben wir uns vor allem mit strategischen Design-Entscheidungen befasst. Insbesondere den Prinzipien für die Unterteilung von Geschäftsdomänen in Komponenten und die Modellierung der Interaktionen zwischen ihnen.

Im Tactical Design geht es, um die Implementierung dieser Komponenten.

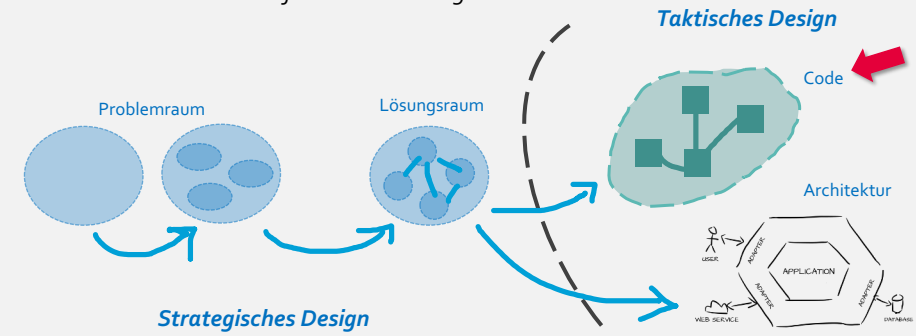
Einige der hier besprochenen Muster würden ein ganzes Modul rechtfertigen. Einige dieser Pattern haben Sie bereits in SW-Technik Vorlesungen gehört. Daher wird das Material in diesem Teil auf einem rein Überblick gebenden Niveau aufbereitet, mit dem Ziel, für das Tactical Design hilfreiche Entwurfsmuster zu erwähnen und auf die Unterschiede und geeignete Einsatzgebiete einzugehen.

1. Geschäftslogik implementieren
2. Geeignete Architekturmuster
3. Interaktionen zwischen Bounded Contexts

TECHNISCHE HOCHSCHULE LÜBECK

DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



Fachliche Kriterien bestimmen das Vorgehen

Allgemeingültige Sprache

Fachliche Durchgängigkeit

TECHNISCHE HOCHSCHULE LÜBECK

TACTICAL DESIGN

Implementierungsmuster für die Geschäftslogik

Wie wir gesehen haben, sind nicht alle Subdomänen gleich geschaffen. Verschiedene Subdomänen haben unterschiedliche Ebenen der strategischen Bedeutung und Komplexität. Wir werden daher vier verschiedene Arten der Implementierung von Geschäftslogik in Code untersuchen. Jedes Muster eignet sich für einen anderen Grad an Komplexität in der Geschäftsdomäne.

Supporting Subdomains	**Core Subdomains**
Transaction Script	Domain Model (Value Objects + Aggregates)
Active Record	Event Sourcing (Domain Events)

Hinweis:
Aggregate und Value Objects sind nur zwei der taktischen Muster, die DDD bietet für Domain Models anbietet. Weitere verfügbare Muster sind das Domänenservice-, das Repository-, das Saga- und das Factory-Muster auf deren Feinheiten wir hier jedoch nicht eingehen.

TECHNISCHE HOCHSCHULE LÜBECK

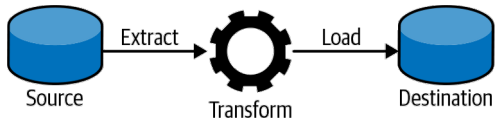
PATTERN FÜR GESCHÄFTSLOGIK

Supporting Subdomains: Transaction Script

TECHNISCHE HOCHSCHULE LÜBECK

Extract – Transform - Load

Dieses Muster eignet sich gut für die einfachsten Problemdomänen, in denen die Geschäftslogik ETL-Operationen (Extract-Transform-Load) ähnelt, d. h. jede Operation extrahiert Daten aus einer Quelle, wendet Transformationslogik an, um sie in eine andere Form umzuwandeln, und lädt das Ergebnis in den Zielspeicher.



Die Geschäftslogik wird durch einfache Prozeduren realisiert.

Jede Prozedur wird als einfaches, unkompliziertes prozedurales Skript implementiert. Sie kann eine dünne Abstraktionsschicht für die Integration mit Speichermechanismen verwenden, ist aber auch frei, direkt auf die Datenbanken zuzugreifen.

Die einzige Anforderung, die Prozeduren erfüllen müssen, ist das transaktionale Verhalten. Jede Operation sollte entweder erfolgreich sein oder fehlschlagen. Wenn ein Prozess aus irgendeinem Grund fehlschlägt, sollte das System in einem konsistenten Zustand bleiben - daher der Name des Musters, Transaktionsskript.

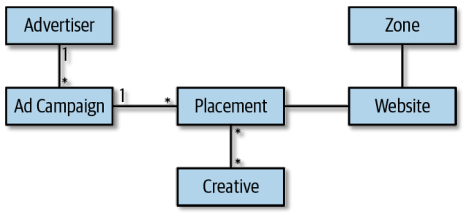
Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

PATTERN FÜR GESCHÄFTSLOGIK

Supporting Subdomains: Active Record

TECHNISCHE HOCHSCHULE LÜBECK

Wie das vorherige Muster unterstützt auch dieses Muster Fälle, in denen die Geschäftslogik einfach ist. Hier kann die Geschäftslogik jedoch auf komplexeren Datenstrukturen operieren. Zum Beispiel können wir anstelle von flachen Datensätzen kompliziertere Objektbäume und Hierarchien haben



Der Betrieb auf solchen Datenstrukturen über ein einfaches Transaktionsskript würde zu viel sich wiederholendem Code führen.

Daher verwendet dieses Muster dedizierte Objekte, um komplizierte Datenstrukturen zu repräsentieren: aktive Datensätze. Neben der Datenstruktur implementieren diese Objekte auch Datenzugriffsmethoden zum Erstellen, Lesen, Aktualisieren und Löschen von Datensätzen - die sogenannten CRUD-Operationen. Folglich sind die aktiven Datensatzobjekte von einem objektrelationalen Mapping (ORM) oder einem anderen Datenzugriffs-Framework abhängig. Der Name des Musters leitet sich von der Tatsache ab, dass jedes Record "aktiv" ist und die erforderliche Datenzugriffslogik implementiert.

Anders als beim Transaction Script Pattern erfolgt der Zugriff nicht direkt auf einer Datenbank, sondern über die Active Records.

```
public class CreateUser {
    public void Execute(userDetails) {
        try {
            DB.StartTransaction();

            var user = new User();
            user.Name = userDetails.Name;
            user.Email = userDetails.Email;
            user.Save();

            DB.Commit();
        } catch {
            DB.Rollback();
        }
    }
}
```

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Domain Model

TECHNISCHE HOCHSCHULE LÜBECK

Nach der Definition von Martin Fowler ist ein Domänenmodell ein Objektmodell einer Domäne, das sowohl Verhalten als auch Zustand (Daten) enthält. Die taktischen Pattern von DDD liefern u.a. die folgenden Bausteine für solche Objektmodelle:

- Value Object
- Aggregate
- Domain Event

Alle diese Muster haben eines gemeinsam. Die Geschäftslogik steht an erster Stelle und vor allem in direktem Bezug zur Ubiquitous Language.

Dieses Muster ermöglicht es dem Code, die Ubiquitous Language zu "sprechen" und den mentalen Modellen der Domänenexperten konzeptionell sehr eng zu folgen.

Value Objects
Value Objekts sind Objekte, die durch ihre Werte eindeutig identifiziert werden können und sich nicht ändern (Immutables). Ein triviales Beispiel für ein Value Objekt ist das String-Objekt in Java- und .NET-Stapeln. Enveränderlich, und alle seine Operationen führen zu einer neuen Instanz eines Strings.

Aggregates
Ein Aggregat ist eine Entität, die eine Hierarchie von Objekten darstellt. Im Gegensatz zu einem Wertobjekt kann eine Entität nicht allein durch ihren Wert identifiziert werden. Das heißt, jedes Aggregat benötigt ein ID-Feld, um identifiziert zu werden. Aggregate sind (anders als Value Objects) Mutables, d.h. sie können ihren Zustand ändern.

Domain Events
Ein Domain Event ist eine Nachricht, die ein bedeutendes Ereignis beschreibt, das in der Geschäftsdomäne aufgetreten ist. Zum Beispiel: Auftrag bezahlt, Lager wieder aufgefüllt, Werbekampagne veröffentlicht, usw. Solche Events sind ein Teil der öffentlichen Schnittstelle eines Aggregats. Ein Aggregat veröffentlicht über sein Public Interface seine Domänenereignisse.

```
class Color {
  int red;
  int green;
  int blue
}
```

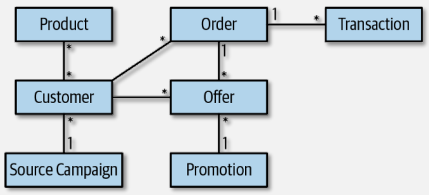
Auf folgenden Slides

PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Domain Model (Aggregates)

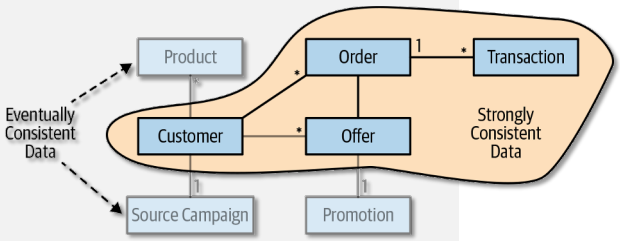
TECHNISCHE HOCHSCHULE LÜBECK

Aggregates
Ein Aggregat ist eine Entität, die eine Hierarchie von Objekten darstellt. Im Gegensatz zu einem Wertobjekt kann eine Entität nicht allein durch ihren Wert identifiziert werden. Das heißt, jedes Aggregat benötigt ein ID-Feld, um identifiziert zu werden. Aggregate sind (anders als Value Objects) Mutables, d.h. sie können ihren Zustand ändern.



Daher ist es entscheidend, die Konsistenz von Aggregat-Zuständen zu schützen. Das Aggregatmuster zieht dazu eine klare Grenze zwischen dem Aggregat und seinem äußeren Bereich. Nur die Geschäftslogik des Aggregats darf seinen Zustand verändern. Alle Prozesse oder Objekte außerhalb des Aggregats dürfen nur seinen Zustand lesen oder seine öffentlichen Methoden ausführen.

Die öffentlichen Methoden des Aggregats sind für die Validierung der Eingabe und die Durchsetzung aller Geschäftsregeln und Invarianten verantwortlich. Diese strenge Abgrenzung stellt auch sicher, dass die gesamte Geschäftslogik in Bezug auf das Aggregat an einer Stelle implementiert wird - im Aggregat selbst.



DDD sieht vor, dass der Entwurf eines Systems von seiner Fachlichkeit getrieben sein sollte. Aggregate bilden da natürlich keine Ausnahme. Ein Aggregat ist daher selten flacher Datensatz, meist ist eine Aggregate eine Hierarchie zusammengehöriger Objekte.

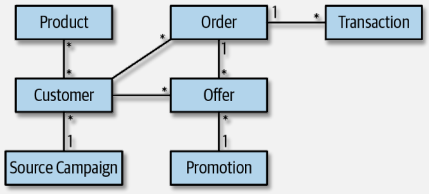
Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Domain Model (Aggregates)

TECHNISCHE HOCHSCHULE LÜBECK

Aggregates
Ein Aggregat ist eine Entität, die eine Hierarchie von Objekten darstellt. Im Gegensatz zu einem Wertobjekt kann eine Entität nicht allein durch ihren Wert identifiziert werden. Das heißt, jedes Aggregat benötigt ein ID-Feld, um identifiziert zu werden. Aggregate sind (anders als Value Objects) Mutables, d.h. sie können ihren Zustand ändern.



Daher ist es entscheidend, die Konsistenz von Aggregat-Zuständen zu schützen. Das Aggregatmuster zieht dazu eine klare Grenze zwischen dem Aggregat und seinem äußeren Bereich. Nur die Geschäftslogik des Aggregats darf seinen Zustand verändern. Alle Prozesse oder Objekte außerhalb des Aggregats dürfen nur seinen Zustand lesen oder seine öffentlichen Methoden ausführen.

Die öffentlichen Methoden des Aggregats sind für die Validierung der Eingabe und die Durchsetzung aller Geschäftsregeln und Invarianten verantwortlich. Diese strenge Abgrenzung stellt auch sicher, dass die gesamte Geschäftslogik in Bezug auf das Aggregat an einer Stelle implementiert wird - im Aggregat selbst.

DDD sieht vor, dass der Entwurf eines Systems von seiner Fachlichkeit getrieben sein sollte. Aggregate bilden da natürlich keine Ausnahme. Ein Aggregat ist daher selten flacher Datensatz, meist ist eine Aggregate eine Hierarchie zusammengehöriger Objekte und damit zusammenhängenden Problemen wie bspw. Transaktionen.

Aggregate bilden Transaktionsgrenzen
Da der Zustand von Aggregaten nur durch ihre eigene Geschäftslogik geändert werden kann, fungiert die Aggregatgrenze in DDD auch immer als Transaktionsgrenze. Alle Änderungen am Zustand des Aggregats sollten transaktional als eine atomare Operation übertragen werden. Eine Transaktion darf sich also nur auf ein Aggregat beziehen.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

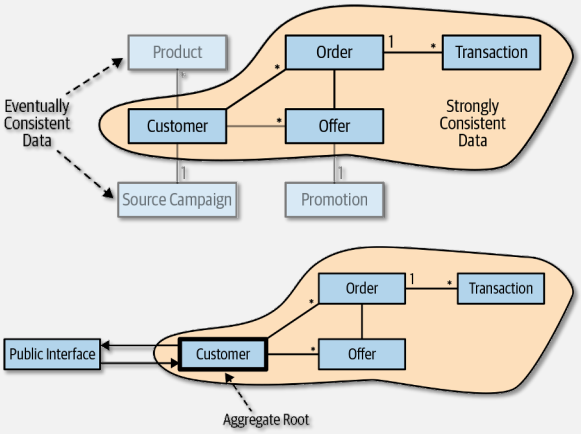
PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Domain Model (Aggregate als Transaktionsgrenzen)

TECHNISCHE HOCHSCHULE LÜBECK

Da alle in einem Aggregat enthaltenen Objekte dieselbe Transaktionsgrenze haben, können Leistungs- und Skalierbarkeitsprobleme auftreten, wenn Aggregate zu groß werden.

Man kann dies nutzen, um Grenzen von Aggregaten anhand eines Strict- und Eventual-Consistency Kriteriums zu schneiden. Die Konsistenz der Daten kann somit eine praktische Heuristik für den Entwurf der Aggregatgrenzen sein. Nur die Informationen, die für die Implementierung der Geschäftslogik des Aggregats streng konsistent sein müssen, sollten sich innerhalb der Grenzen des Aggregats befinden. Objekte, die eventuell konsistent sein können, sollten nicht zum Aggregat gehören und nur über ihre ID referenziert werden.



Regel 1:
Schütze fachliche Invarianten innerhalb von Aggregatgrenzen

Regel 2:
Entwirf kleine Aggregate

Regel 3:
Referenziere andere Aggregate nur über ihre Identität

Regel 4:
Aktualisiere andere Aggregate unter Verwendung von Eventual Consistency

Daumenregel:
Aggregate so klein wie möglich halten und nur Objekte aufnehmen, die von der Geschäftsdomäne in einem stark konsistenten Zustand benötigt werden.

Aggregatwurzel:
Da ein Aggregat meist eine Hierarchie von Objekten darstellt, sollte aus Gründen des Zustandsschutz von Aggregaten nur eines von ihnen als öffentliche Schnittstelle des Aggregats bestimmt werden - die Wurzel des Aggregats

PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Domain Model (Domain Events)

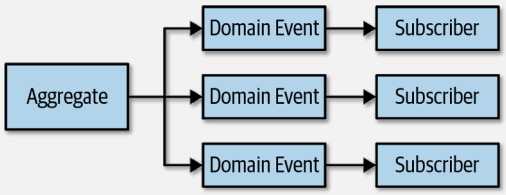
TECHNISCHE HOCHSCHULE LÜBECK

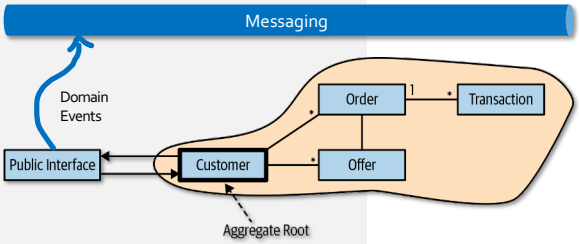
Domain Events
Ein Domain Event ist eine Nachricht, die ein bedeutendes Ereignis beschreibt, das in der Geschäftsdomäne aufgetreten ist. Zum Beispiel: Auftrag bezahlt, Lager wieder aufgefüllt, Werbekampagne veröffentlicht, usw.. Solche Events sind ein Teil der öffentlichen Schnittstelle eines Aggregats. Ein Aggregat veröffentlicht über sein Public Interface seine Domänenereignisse.

Domain Events sind ein Teil der öffentlichen Schnittstelle eines Aggregats. Ein Aggregat veröffentlicht seine Ereignisse. Andere Prozesse, Aggregate oder sogar externe Systeme können die Domänenereignisse abonnieren und als Reaktion darauf ihre eigene Logik ausführen.

Die hierzu passenden Interaktionspattern haben wir in **Unit 07** als **Pub/Sub** und **Queueing** kennengelernt.

Vgl. auch **Smart Endpoints / Dump Pipes**





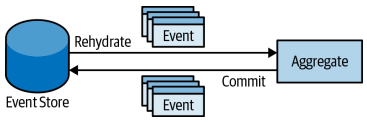
Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

PATTERN FÜR GESCHÄFTSLOGIK

Core Subdomains: Event Sourcing

TECHNISCHE HOCHSCHULE LÜBECK

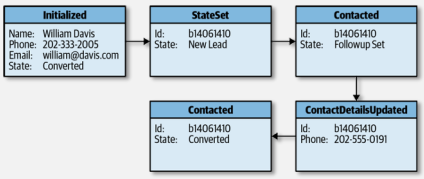
Das Event-Sourcing-Pattern verwendet Domain Events, um die Dimension der Zeit in das Domänenmodell einzuführen. Jede Änderung des Systemzustands sollte als Domänenereignis ausgedrückt und aufgezeichnet werden.



Die Datenbank, die die Domänenereignisse des Systems speichert, ist der einzige stark konsistente Speicher - die „Single Source of Truth" des Systems. Jede Operation an einem Aggregat, das von Ereignissen gespeist wird, folgt diesem Skript:

- Laden der Domänenereignisse des Aggregats.
- Erstellung der Zustandsdarstellung.
- Ausführen der Geschäftslogik und Erzeugen neuer Domain-Events.
- Übertragen der neuen Domain-Events in den Event Store.

Event-Sourcing wird bspw. in der Finanzindustrie zur Darstellung von Änderungen in einem Ledger verwendet. Ein Ledger ist ein Append-Only-Log das den Verlauf von Transaktionen protokolliert. Ein aktueller Zustand (z. B. der Kontostand) kann immer durch ein Replaying der Ledger-Datensätze abgeleitet werden.



Replaying Time Machine
Da Domain Events verwendet werden können, um den aktuellen Zustand eines Aggregats wieder-herzustellen, können sie auch für die Wiederherstellung aller vergangenen Zustände des Aggregats verwendet werden.

Tiefer Einblick
Event Sourcing bietet tiefe Einblicke in den Zustand und das Verhalten des Systems. Außerdem ermöglicht das flexible Modell die Umwandlung der Ereignisse in verschiedene Zustandsdarstellungen - auch solche, die ursprünglich nicht geplant waren.

Audit log
Die persistierten Domain-Events stellen ein stark konsistentes Audit-Protokoll von allem dar, was mit den Zuständen der Aggregate passiert ist. Gesetze verpflichten einige Geschäftsdomänen, solche Audit-Protokolle zu implementieren (z.B. Finanzindustrie). Event Sourcing bietet dies von Haus aus.

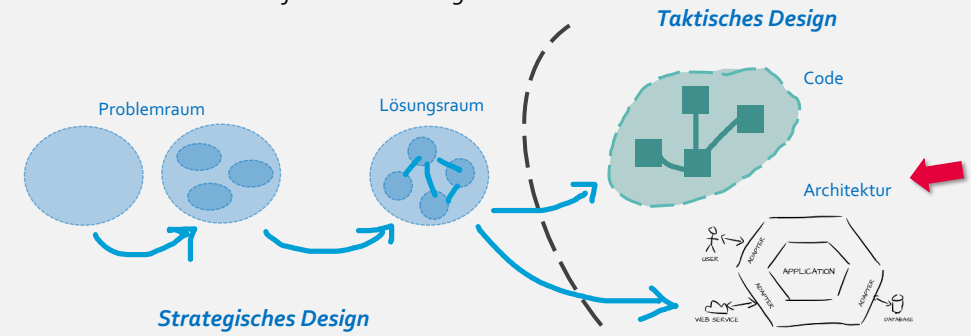
Event Souroing ist besonders praktisch für Systeme, die Geld oder monetäre Transaktionen verwalten. Es erlaubt, die Entscheidungen, die das System getroffen hat, und den Geldfluss zwischen Konten leicht nachzuvollziehen.

Im Vergleich zu einer zustandsbasierten Darstellung erfordert das Event-Sourcing-Modell mehr Aufwand bei der Modellierung. Allerdings ist dieses Muster insbesondere in den blauen Szenarien erwägenswert.

Randnotiz:
Blockchains sind letztlich auch nichts anderes als P2P-basierte Append-Only-Logs!

DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



Fachliche Kriterien bestimmen das Vorgehen

Allgemeingültige Sprache

Fachliche Durchgängigkeit

TECHNISCHE HOCHSCHULE LÜBECK

TACTICAL DESIGN

Architektur-Pattern

Welches Architekturmuster verwendet werden soll, ist eine wichtige taktische Designentscheidung. Das richtige Muster unterstützt die Umsetzung der funktionalen und nicht-funktionalen Anforderungen des Systems.

Wir werden uns hierzu drei häufig anzutreffende Architekturmuster und geeignete Anwendungsfälle hierfür ansehen.

1. Layered Architecture
2. Ports & Adaptors
3. CQRS (Command-Query-Responsibility-Segregation)

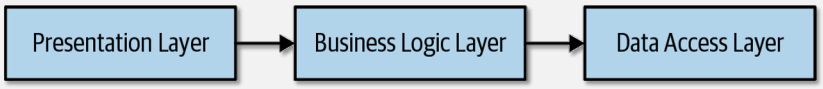
OK! Ähh, was für ein Boot?

- *Speedboot?*
- *Schlauchboot?*
- *Kanu?*
- *Floß?*
- *...*

TECHNISCHE HOCHSCHULE LÜBECK

TACTICAL DESIGN

Layered Architecture



Präsentiert und definiert die Benutzeroberfläche

Implementiert die Geschäftslogik

Zugriff auf Persistenz-mechanismen (DBen und andere Infrastruktur-komponenten)

Durch die Abhängigkeit zwischen der Geschäftslogik und den Datenzugriffsschichten passt dieses Architekturmuster gut zu einem System, dessen Geschäftslogik mit dem Active-Record-Pattern implementiert wurde.

*Das Pattern ist daher häufig in Bounded Contexts für **Supporting Domains** anzutreffen!*

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

TECHNISCHE HOCHSCHULE LÜBECK

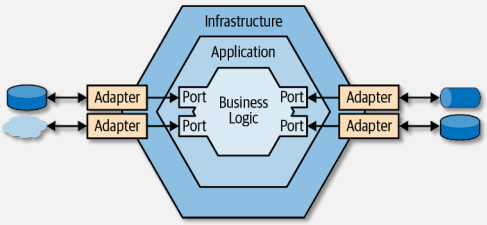
Ein Klassiker!
Vielleicht sogar DER Klassiker!

TACTICAL DESIGN

Ports & Adaptors (auch Hexagonale Architektur)

Das Ports & Adapter-Pattern ist dem Layered Architecture Pattern insofern ähnlich, als das die Codebasis des Systems ebenfalls nach technologischen Gesichtspunkten zerlegt wird. Sie unterscheidet sich jedoch hinsichtlich:

- **Terminologie:** Es werden die Benutzeroberfläche des Systems, der Code für den Datenzugriff und alle anderen infrastrukturellen Belange einfach als "Infrastruktur" bezeichnet. Dadurch wird insbesondere die Machine-to-Machine-Interaktion konzeptionell besser berücksichtigt.
- **Abhängigkeiten:** Anstatt von technologischen Belangen dominiert (UI-Technologien + Persistenz-Technologien) zu sein, nimmt die Geschäftslogik in der Ports & Adapters-Architektur die zentrale Rolle ein. Sie ist nicht direkt von einer der Infrastrukturkomponenten des Systems abhängig.
- **Anwendungsschicht:** Die Anwendungsschicht implementiert eine Fassade für die öffentlichen Schnittstellen des Systems. Sie beschreibt alle vom System angebotenen Operationen und orchestriert die Geschäftslogik des Systems, um diese auszuführen.



Integration von Infrastrukturkomponenten

Ziel der Ports & Adapter-Architektur ist es, die Geschäftslogik des Systems von den Infrastrukturkomponenten zu entkoppeln.

Anstatt die Infrastrukturkomponenten direkt zu referenzieren und aufzurufen, definiert die Geschäftslogikschicht "Ports", die von der Infrastrukturschicht implementiert werden müssen. Die Infrastrukturschicht implementiert "Adapter" der Ports für die Arbeit mit verschiedenen Technologien. Die Anwendungsschicht liefert die Adapter für die Ports der Geschäftslogik durch Dependency Injection.

Die Entkopplung der Geschäftslogik von allen technologischen Belangen macht das Ports & Pattern vor allem für Geschäftslogiken, die auf dem Domain Model Pattern implementiert werden, interessant.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019
Siehe auch: Alistair Cockburn - https://alistair.cockburn.us/hexagonal-architecture

TECHNISCHE HOCHSCHULE LÜBECK

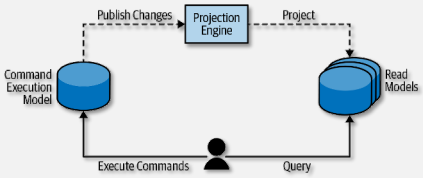
*Das Pattern ist daher häufig in Bounded Contexts für **Core Subdomains** anzutreffen!*

TACTICAL DESIGN

CQRS (Command-Query Responsibility Segregation)

TECHNISCHE HOCHSCHULE LÜBECK

CQRS basiert auf einem einzigem Modell zur Ausführung von Operationen, die den Zustand des Systems verändern (Systembefehle) . Gem. dem CQRS-Pattern ist dieses Command Execution Modell ist das einzige Modell, das stark konsistente Daten repräsentiert (Single Source of Truth).



Das System kann beliebig viele Modelle mittels Projektionen erzeugen, die Darstellung von Daten für Benutzer oder andere Systeme erforderlich sind. Diese Modelle sind Read-only. Keine der Operationen des Systems kann die Daten der gelesenen Modelle direkt ändern.

Polyglotte Persistenz

In vielen Fällen sind unterschiedliche Modelle für unterschiedliche Anforderungen des Systems erforderlich. Bspw. gibt es optimierte Datenmodelle für Online-Transaktionsverarbeitung (OLTP), Online-Analytical Processing (OLAP) und Suche.

Ein weiterer Grund ist die in Microservice-Architekturen häufig anzutreffende polyglotte Persistenz, bei der mehrere Datenbanken für unterschiedliche Anforderungen an den Datenzugriff verwendet werden. Ein einzelnes System könnte zum Beispiel einen Dokumentendatenbank (document store) als operative Datenbank, einen Column Store für Analysen/Berichte und eine Suchmaschine für die Implementierung von Suchfunktionen verwenden.

Modell-Trennung

In der CQRS-Architektur sind die Zuständigkeiten der Modelle des Systems nach ihrem Schreib-Lese-Charakter getrennt. Ein Command kann nur auf dem stark konsistenten Befehlsausführungsmodell operieren. Eine Query kann keinen der Systemzustände direkt verändern - weder die Lesemodelle noch das Befehlsausführungsmodell.

Use Cases

CQRS eignet sich insbesondere für ereignisgesteuerte Domänenmodelle.

Reines Event-Sourcing lässt es nicht zu, Datensätze basierend auf den Zuständen der Aggregate abzufragen, aber CQRS kann dies ermöglichen, indem die Zustände in abfragbare Datenbanken projiziert werden.

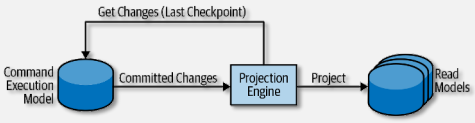
*CQRS ist daher häufig in Bounded Contexts mit **Event Sourcing** von **Core Subdomains** anzutreffen!*

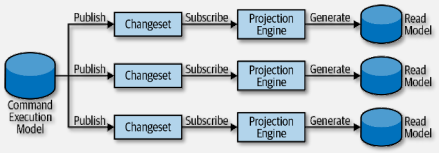
Siehe auch: Greg Young, CQRS Documents - https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

TACTICAL DESIGN

CQRS (Projections)

TECHNISCHE HOCHSCHULE LÜBECK





SYNCHRONE PROJEKTIONEN

Synchrone Projektionen funktionieren grundsätzlich nach folgendem Verfahren:

- Die Projektions-Engine fragt die OLTP-Datenbank nach Datensätzen ab, die nach dem letzten verarbeiteten Checkpoint (oft ein Zeitstempel) hinzugefügt oder aktualisiert wurden.
- Die Projektions-Engine verwendet die aktualisierten Daten, um die Read Modells des Systems neu zu generieren/aktualisieren.
- Die Projektions-Engine speichert den Checkpoint des zuletzt verarbeiteten Datensatzes. Dieser Wert wird bei der nächsten Iteration verwendet, um Datensätze abzurufen, die nach dem letzten verarbeiteten Datensatz hinzugefügt oder geändert wurden.

ASYNCHRONE PROJEKTIONEN

Bei asynchronen Projektionen veröffentlicht das Command Execution Model alle bestätigten Änderungen mittels einesPub/Sub Messaging Bus. Die Projektions-Engines des Systems können die veröffentlichten Nachrichten abonnieren und sie zur Projektion der gelesenen Modelle verwenden.

Trotz der offensichtlichen Skalierungs- und Leistungsvorteile der asynchronen Projektionsmethode ist sie anfälliger für die Fallacies of Distributed Computing. Wenn die Nachrichten nicht in der richtigen Reihenfolge verarbeitet oder dupliziert werden, werden inkonsistente Daten in die gelesenen Modelle projiziert.

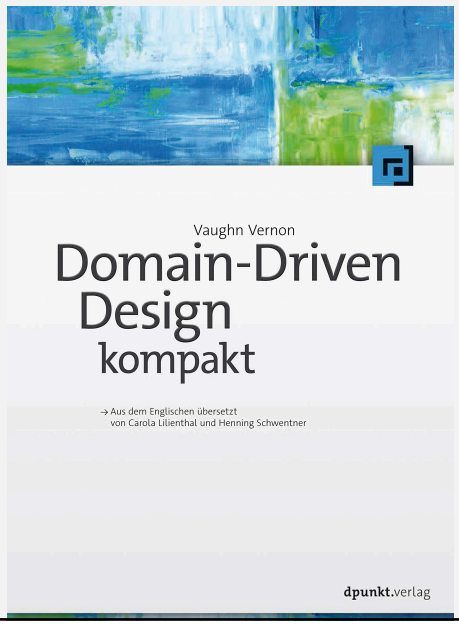
Diese Methode macht es auch schwieriger, neue Projektionen hinzuzufügen oder bestehende zu regenerieren.

Projektionen lassen sich grundsätzlich synchron oder asynchron erzeugen.

Tipp:
Aus diesen Gründen ist es ratsam, immer als Basis eine synchrone Projektion zu nutzen und nur bei Bedarf eine zusätzliche asynchrone Projektion darauf zu implementieren.

Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

ZUM NACHLESEN



1. DDD
2. Strategisches Design mit Bounded Contexts und der Ubiquitous Language
3. Strategisches Design mit Subdomains
4. Strategisches Design mit Context Mapping
5. Taktisches Design mit Aggregates
6. Taktisches Design mit Domain Events

TECHNISCHE HOCHSCHULE LÜBECK

ZUM NACHLESEN

What is Domain Driven Design?

I. Strategic Design

1. Analysing Business Domains
2. Discovering Domain Knowledge
3. Managing Complexity with Bounded Contexts
4. Context Mapping

II. Tactical Design

5. Business Logic Implementation Patterns
6. Architectural Patterns
7. Integration of Bounded Contexts

TECHNISCHE HOCHSCHULE LÜBECK

ZUM NACHLESEN

Paper + Textbooks + URLs

- *Martin Fowler. 2002. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., USA.*
- *Vaughn Vernon. 2013. Implementing Domain-Driven Design (1st. ed.). Addison-Wesley Professional.*
- *Greg Young. CQRS Documents - https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf*
- *Martin Fowler. 2014. CQRS, https://martinfowler.com/bliki/CQRS.html*
- *Alistair Cockburn. 2005. Hexagonal Architecture. https://alistair.cockburn.us/hexagonal-architecture*

TECHNISCHE HOCHSCHULE LÜBECK

KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de