CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Domain Driven Design

(3) Strategisches Design

Bounded Contexts + Context Mapping

PROF. DR.
NANE KRATZKE

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

KAPITEL 14

Domain Driven Design

14.1 Fachlichkeit, Fachlichkeit, Fachlichkeit

14.2 Strategisches Design

- Subdomänen
- Ubiquitous Language
- Bounded Context
- Context Mapping

14.3 Taktisches Design

- Oft genutzte Pattern für Geschäftslogik
  (ETL, Active-Record, Domain Model, Event-Sourcing)
- Oft genutzte Architektur-Pattern
  (Layered Architecture, Ports & Adaptor, CQRS)

14.4 Zusammenfassung

INHALTE

Domain Driven Design

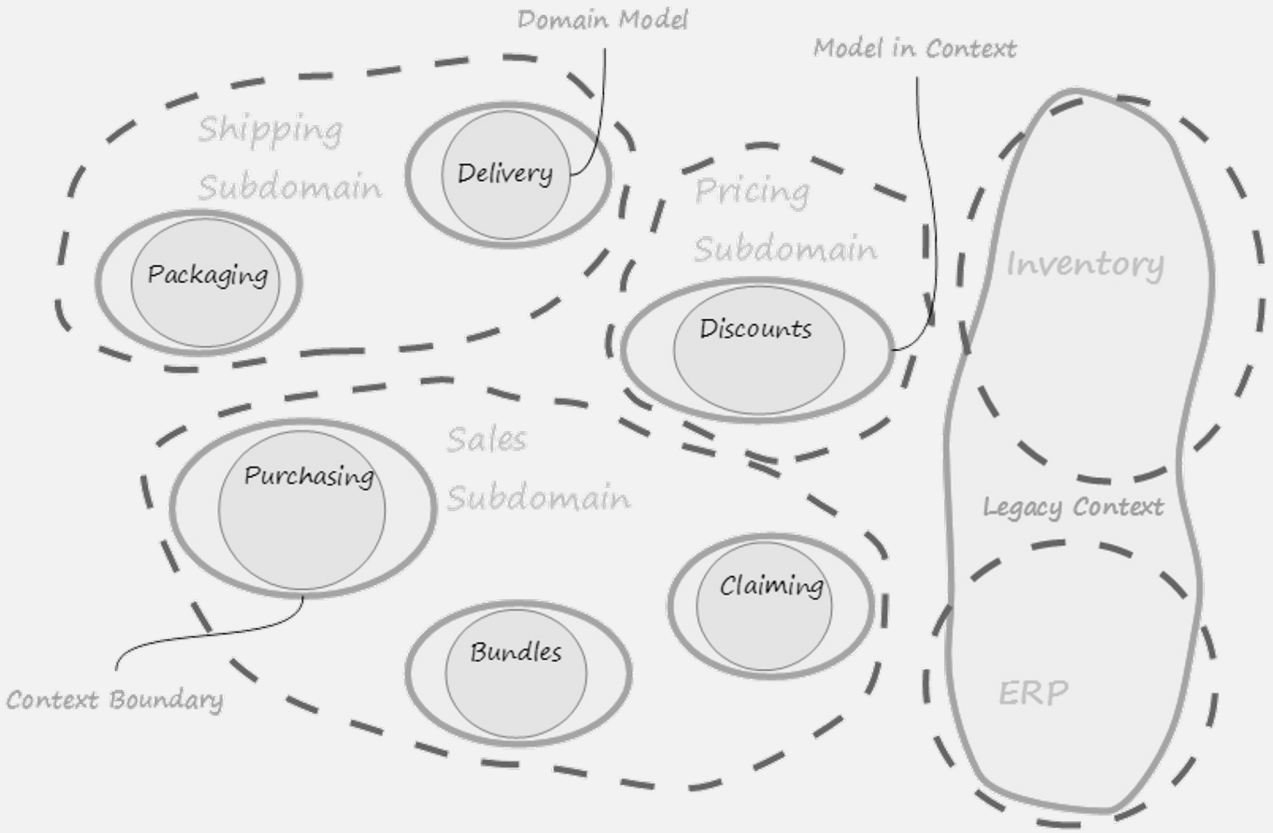
- Was ist das?
- Effektives Software Design
- Strategisches Design
- Taktisches Design

Strategisches Design

- Subdomains
- Ubiquitous Language
- **Bounded Context**
- **Context Mapping**

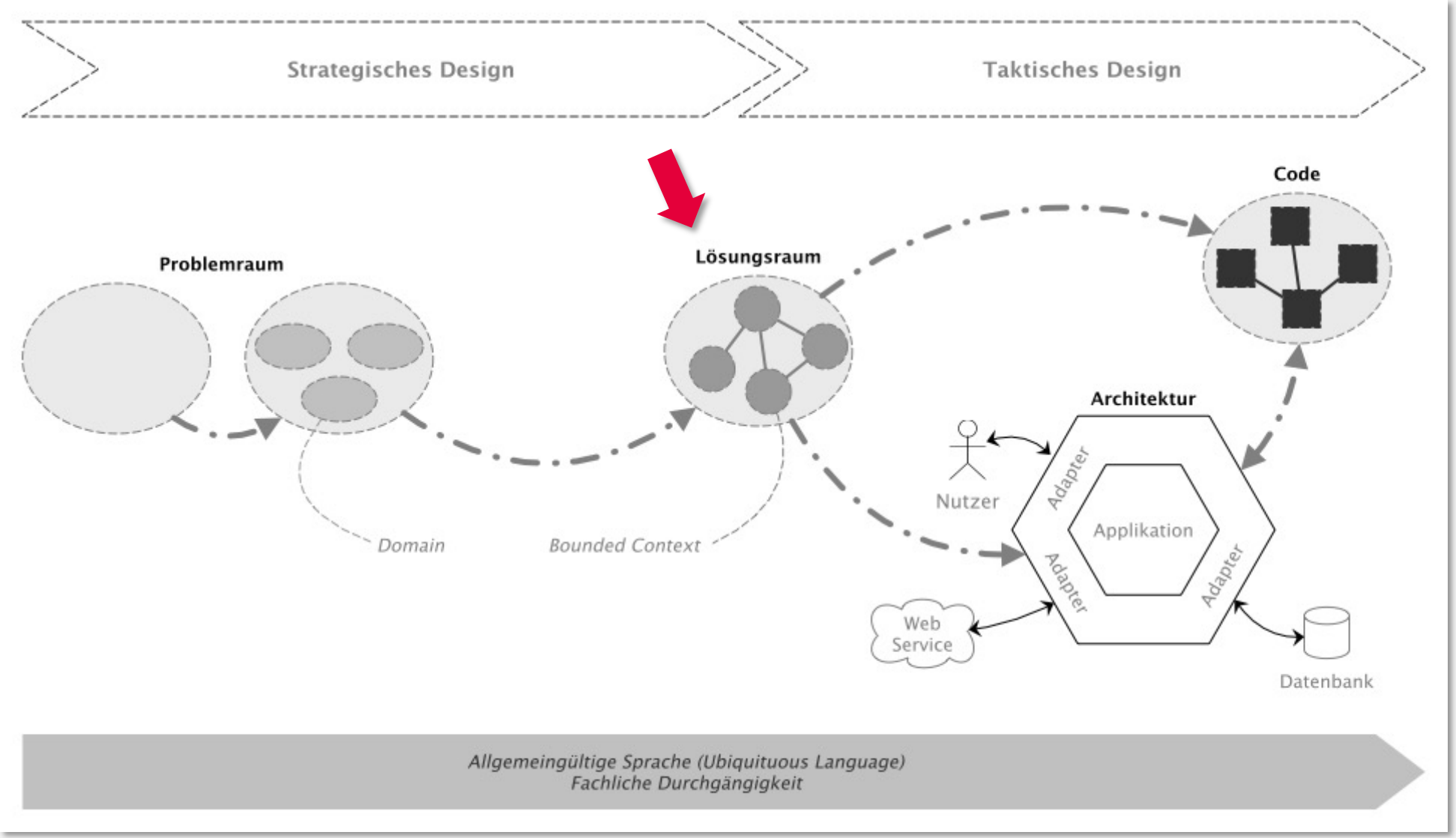
Taktisches Design

- Business Logic Pattern
- Architectural Pattern



DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung

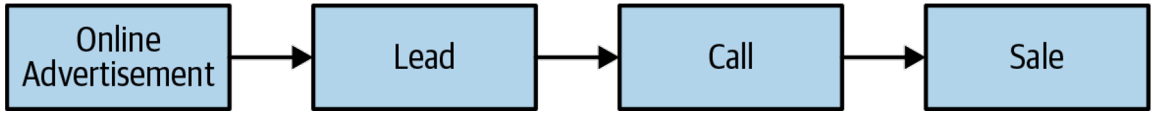


BOUNDED CONTEXTS

Begriffs-Komplexitäten beherrschen

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.

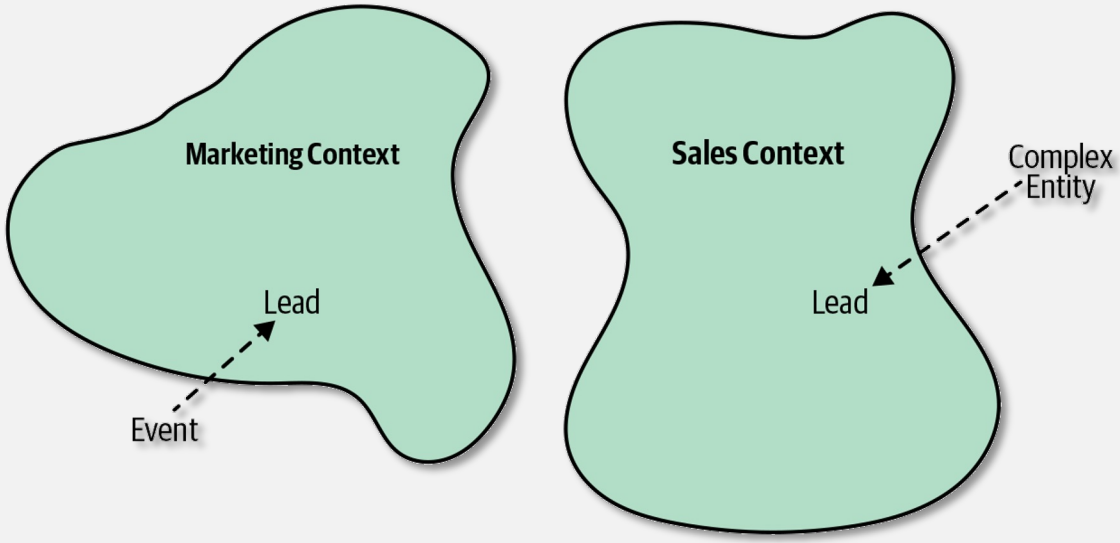
Insbesondere Enterprise Architecture Management Ansätze haben oft den Anspruch eine globale „Ubiquitous Language“ (ohne diese so zu nennen) für ein ganzes Unternehmen zu entwickeln. Häufig scheitern diese Ansätze jedoch.

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

CLOUD NATIVE COMPUTING

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

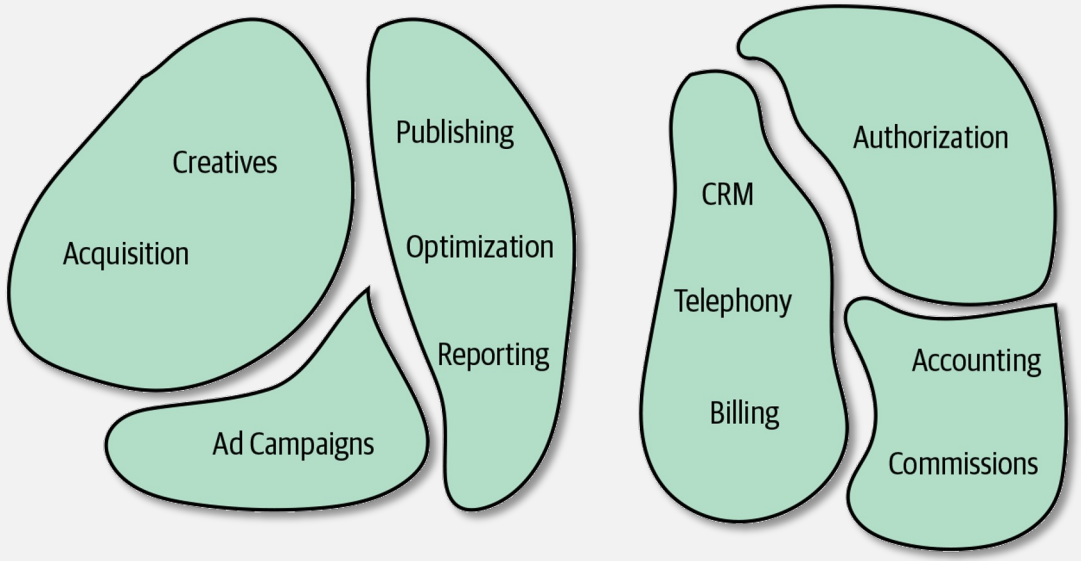
Bis man damit fertig ist, wäre das Geschäftsfeld vermutlich schon von einem Wettbewerber besetzt worden.

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 Language 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

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.

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 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 Ubiquitous 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:

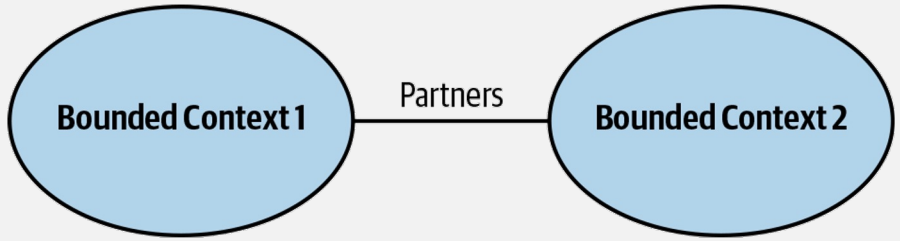
- Kooperation
- Customer-Supplier
- Separate Ways

CONTEXT MAPPING

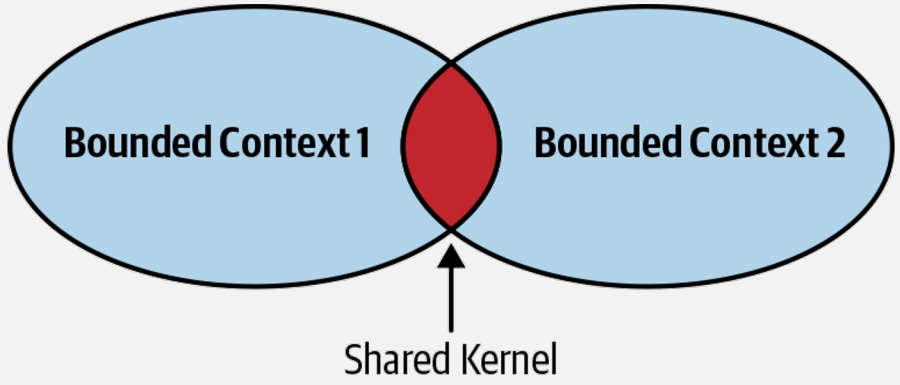
Cooperation Patterns

- 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



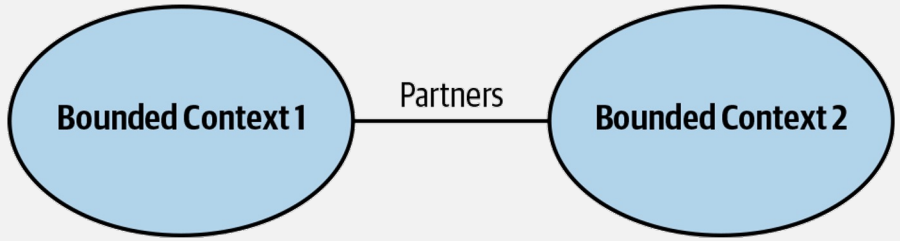
Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

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

CONTEXT MAPPING

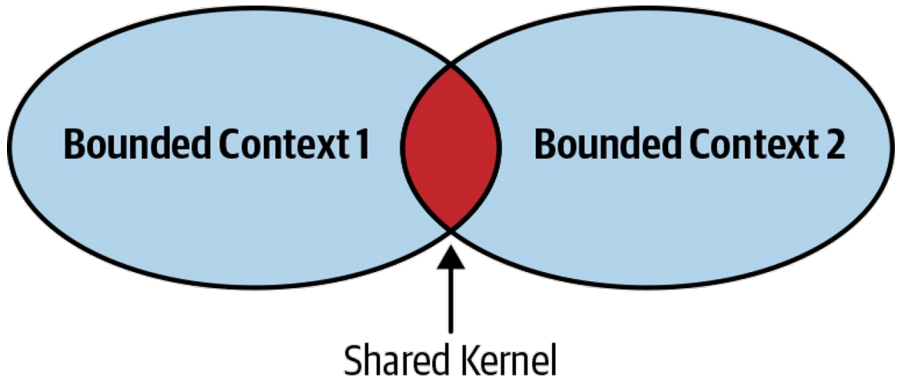
Cooperation Patterns – Shared Kernel

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 Kernel 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.

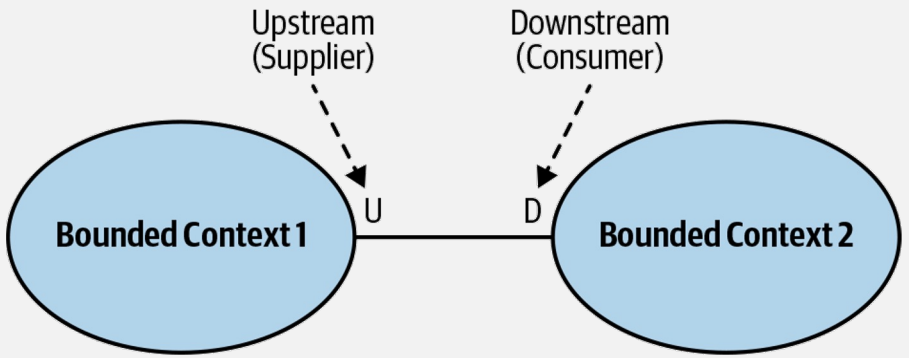
Shared Kernel können eine disziplinierende Voraussetzung für die Integration von Bounded Contexts sein, die vom gleichen Team betrieben und implementiert werden. In einem solchen Fall können Ad-hoc-Integrationen der Bounded Contexts die Grenzen der Kontexte im Laufe der Zeit „verwaschen". Ein Shared 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

- 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.



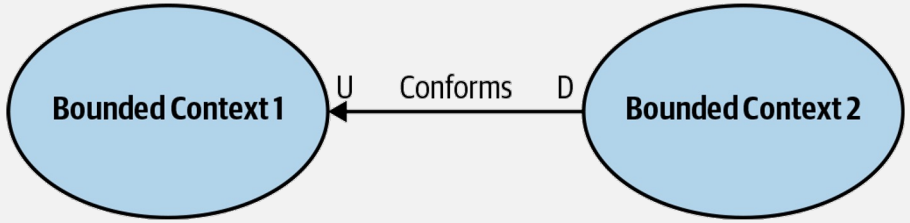
- Conformist Pattern
- Anti-Corruption Layer
- Open-Host Service

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

CONTEXT MAPPING

Customer-Supplier Patterns – Conformist Pattern

- 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

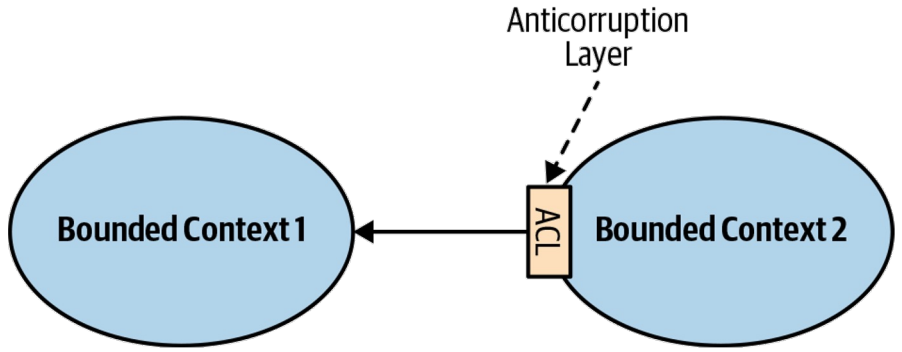
CONTEXT MAPPING

Customer-Supplier Patterns – Anti Corruption Layer

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.

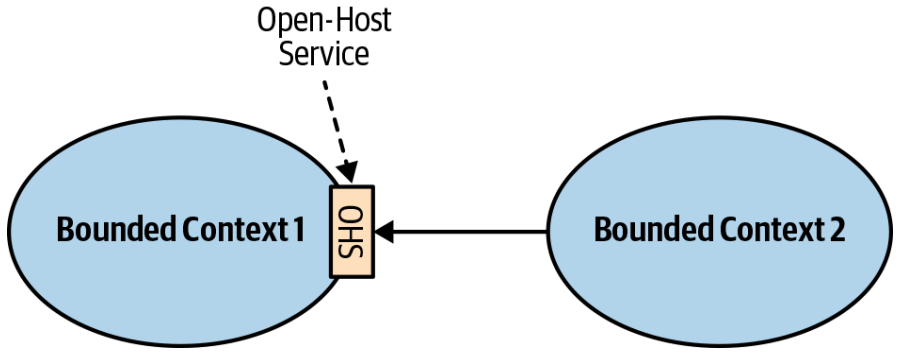
Bildquelle: Vladik Khononov, *What Is Domain-Driven Design?* O'Reilly Media, 2019

CONTEXT MAPPING

Customer-Supplier Patterns – Open Host Service

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 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

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.

Niemals bei Core Domains!

Das Separate Ways Pattern sollte bei der Integration von Core-Subdomains auf alle Fälle vermieden werden. Eine doppelte Implementierung solcher Bounded Contexts 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

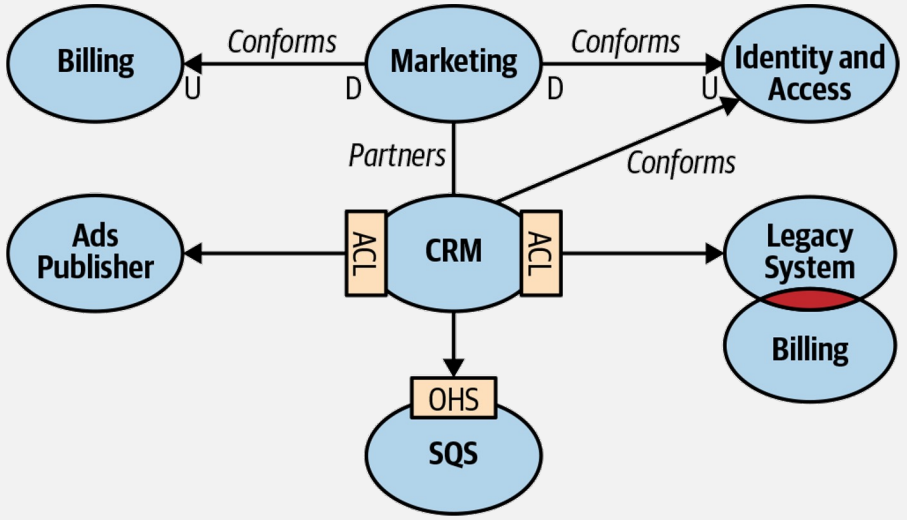
Wenn Modelle von Bounded Contexts so unterschiedlich sind, dass eine konforme Beziehung nicht möglich ist, und die Implementierung einer Antikorruptionsschicht teurer wäre als die Duplizierung der Funktionalität. Auch in einem solchen Fall ist es für die Teams kostengünstiger, getrennte Wege zu gehen.

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

AUSBLICK

Domain Driven Design

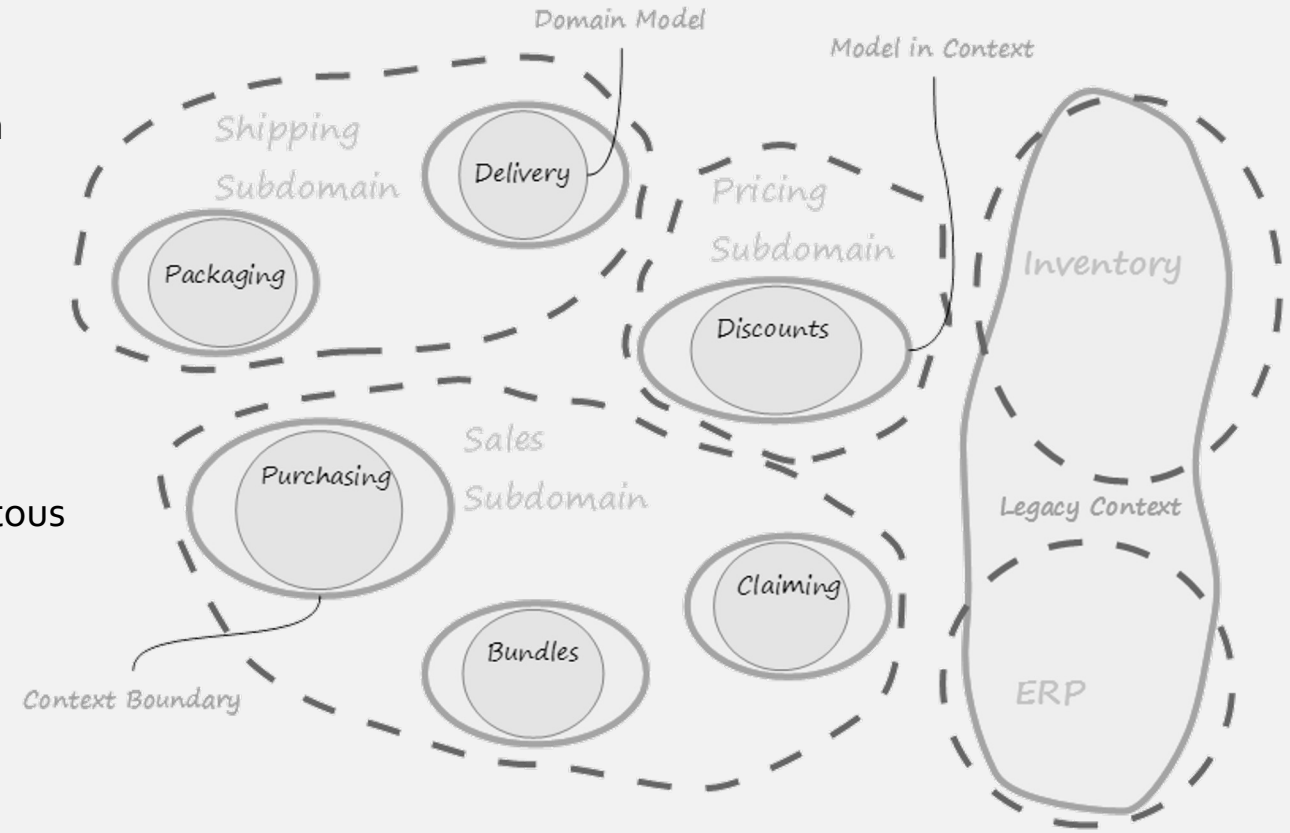
- Was ist das?
- Effektives Software Design
- Strategisches Design
- Taktisches Design

Strategisches Design

- Subdomains
- Bounded Context + Ubiquitous Language
- Context Mapping

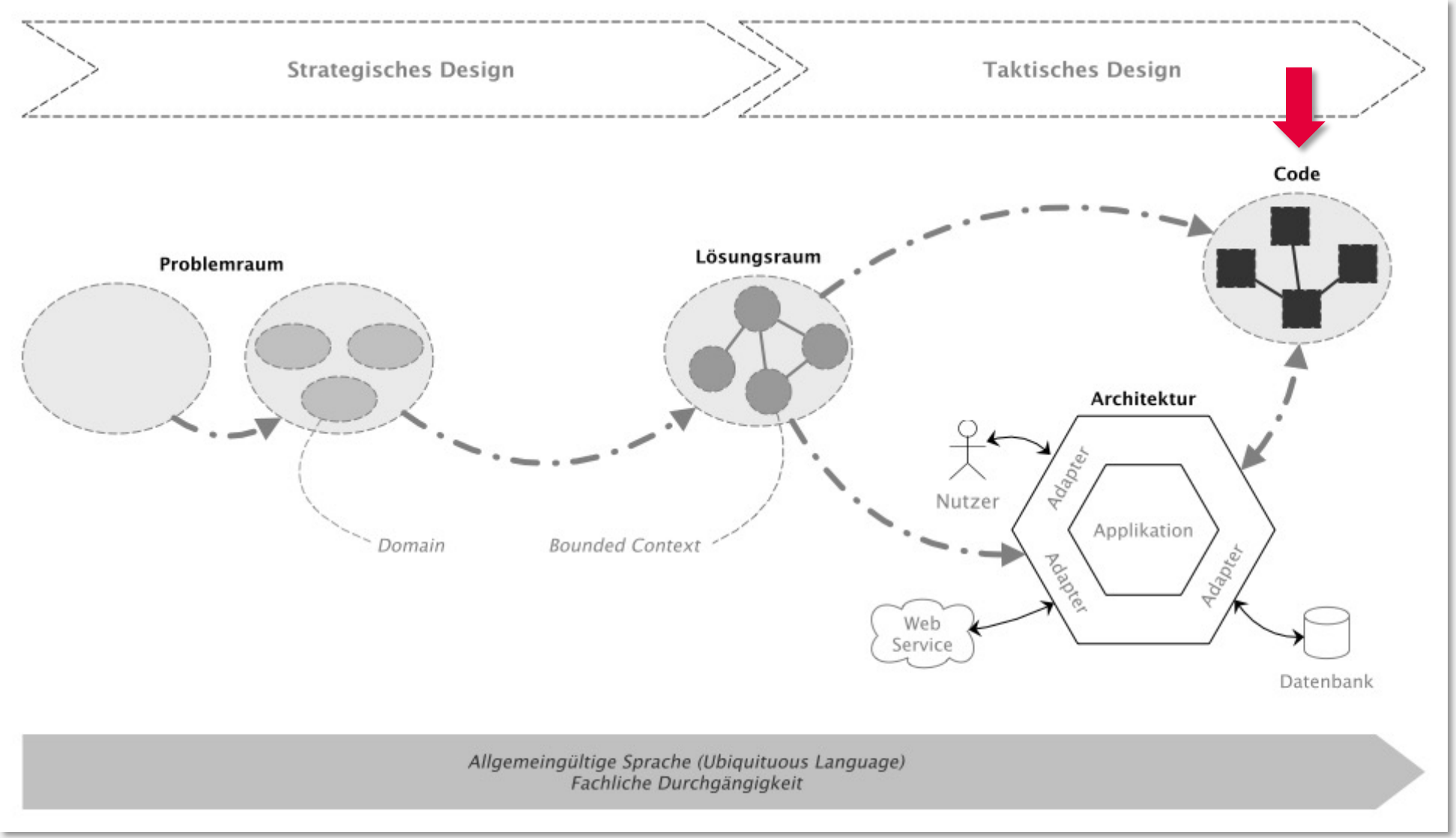
Taktisches Design

- Business Logic Pattern
- Architectural Pattern



DOMAIN DRIVEN DESIGN

Fachlichkeit als Treiber der Softwareentwicklung



CLOUD NATIVE COMPUTING

KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de