# Whitepaper
# Domain Driven Design

## Inhalt

**Domain Driven Design – Was ist das und warum macht man das?**
von Dr. Ute Heimann

In der Blogserie über Microservice-Architekturen[1] wurde das Thema Domain Driven Design angesprochen, um natürliche Grenzen für Microservices zu finden. Mit dem zunehmenden Fokus auf Microservices erlebt das Domain Driven Design eine Aufmerksamkeit, die es im Jahr 2004 bei der Veröffentlichung des Buches von Eric Evans *Tackling Complexity in the Heart of Software*, nicht einmal ansatzweise bekam. Und das Beste: Domain Driven Design ist viel mehr als eine methodische Hilfestellung zur Entwicklung von Microservices.

Stattdessen beschreibt es einen Philosophieansatz zur Softwareentwicklung, der statt technologischen Entscheidungen, die Fachlichkeit ins Zentrum der Implementierung von komplexen Softwareprojekten rückt.

Was soll damit erreicht werden? Zum einen geht es darum, Komplexität an der Stelle einzufangen, an der sie verankert ist. Undurchschaubare Geschäftsprozesse lassen sich nicht besser verstehen und in Software gießen, wenn man das eine oder andere technische Framework verwendet. Stattdessen werden im Domain Driven Design fachliche Sachverhalte bewusst von technologischen Entscheidungen entkoppelt. Technologische Entscheidungen werden so spät wie möglich und möglichst so getroffen, dass sie nicht das Design der Fachlichkeit beeinflussen.
Zum anderen wirkt die Fokussierung auf die Fachlichkeit als Stabilisierungsfaktor für die gesamte Software. Der langlebigste und wichtigste Teil eines Systems ist die fachliche Aufgabenstellung (nicht zu verwechseln mit subjektiv geäußerten fachlichen Anforderungen). Software muss sich auf diese Aufgabenstellung konzentrieren und erst in zweiter Linie technologische Fragenstellungen lösen.

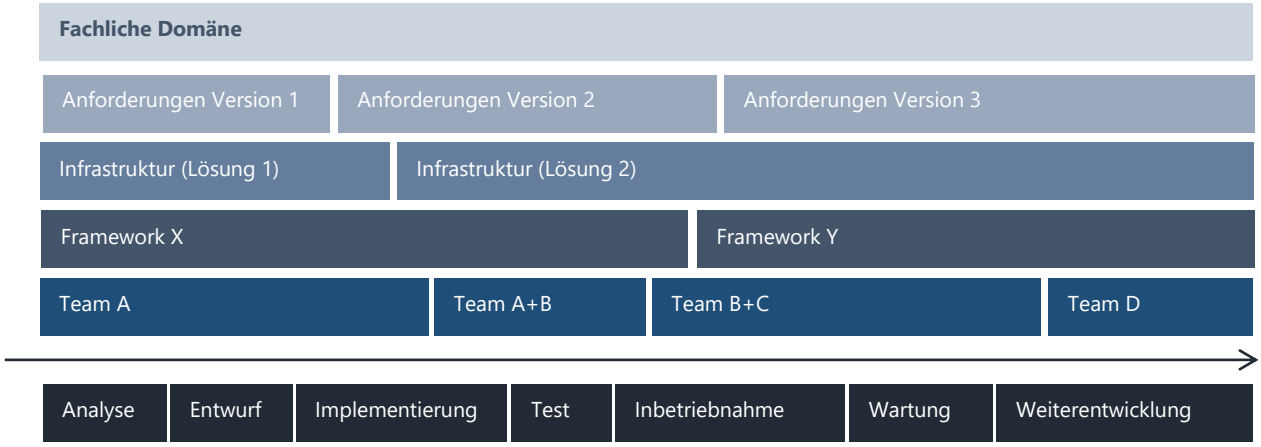



*Abbildung 1: Die fachliche Domäne als stabilstes Element der Softwareentwicklung*

Domain Driven Design rückt die Fachlichkeit sowohl beim Designprozess, als auch beim Ergebnis des Designs in den Mittelpunkt.

Zum einen wird eine Methodik bereitgestellt, bei der Fachexperten und Entwicklung gemeinsam die Fachlichkeit designen. Fachlichkeit wird zum Gegenstand enger, beidseitiger Zusammenarbeit. Des Weiteren wird ein Verfahren angeboten, welches die Modularisierung von Software auf Basis der Fachlichkeit stringent ermöglicht.
Komplexität verlangt außerdem nach Organisation und Abstimmung. Domain Driven Design liefert Möglichkeiten zur Aufteilung von umfangreichen Systemen in abgeschlossene Einheiten (Bounded Contexts), die besonders gut in eine Microservice-Architektur passen. Außerdem definiert es Methoden zur Reflektion der Beziehungen und Kommunikation der Teile untereinander.

Trotz aller Euphorie: Domain Driven Design hat (genau wie die Einführung einer Microservice-Architektur) einen Preis und zwar unter Umständen einen höheren Entwicklungsaufwand und insbesondere eine sehr intensive Einarbeitung in fachliche Themen. Ob die Vorteile für Domain Driven Design oder Teile davon überwiegen, muss in jedem Projekt individuell geklärt werden. Die reflektierte Auseinandersetzung mit den mehr oder weniger komplexen Bestandteilen einer Domäne und ihrem Einfluss auf das zu erstellende System schadet aber mit Sicherheit keinem Projekt.

**Das Domänenmodell – Dreh- und Angelpunkt der Entwicklung**
von Dr. Ute Heimann

Die Erstellung eines lebendigen, mitwachsenden Domänenmodells steht im Zentrum des Domain Driven Design. Bedeutet das, neben der Entwicklung auch noch parallel ein Modell pflegen zu müssen? Als zusätzliche Dokumentation sozusagen? Und wie soll dieses Modell überhaupt aussehen? Brauche ich nur ein Domänenmodell um von den Vorteilen des Domain Driven Design zu profitieren? Und – was ist überhaupt die „Domäne“?

Das Wichtigste zuerst: Das Domänenmodell ist kein Selbstzweck. Es ist die Verbindung zwischen den fachlichen Anforderungen und der Implementierung des Systems. Eine der wichtigsten Aufgaben eines Systems ist es, die Fachlichkeit widerzuspiegeln. Denn wer braucht schon ein schön designtes System, das sich aber von der realen Fachlichkeit, für die es gebaut wird, immer weiter entfernt?
Die Domäne beschreibt die Fachlichkeit mit ihren individuellen Aufgabenstellungen und Prozessen, Abhängigkeiten, Bezeichnungen und komplexen Zusammenhängen. Die Domäne ist zum einen der Auslöser der Entwicklung, sie ist aber auch die größte Konstante über den gesamten Lebenszyklus der Software hinweg. Das Domain Driven Design stellt deshalb die Domäne ins Zentrum und an den Anfang der Software-Entwicklung. Und ja: Das heißt auch, dass sich der Entwickler mit der zunächst unbekannten Domäne beschäftigt und die fachlichen Herausforderungen versteht, bevor er technische Fragestellungen angeht.

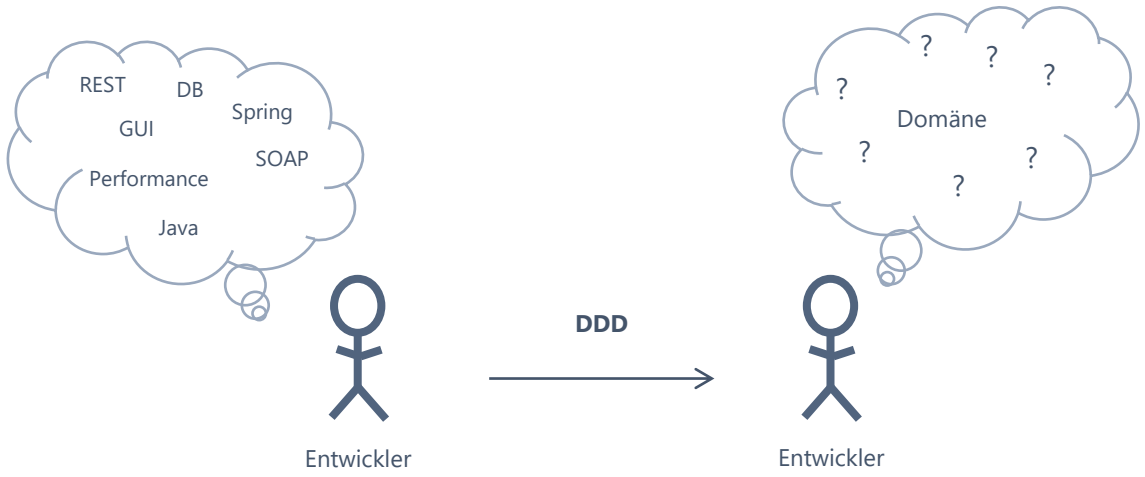



*Abbildung 2: Die neue Rolle des Entwicklers im DDD*

Die fachlichen Zusammenhänge zu fassen und als Basis für die Implementierung aufzubereiten, ist Aufgabe des Domänenmodells. Dabei ist es wichtig zu verstehen, dass ein Modell immer nur einen bestimmten Ausschnitt der Wirklichkeit abbildet und auch das nur aus einer bestimmten Perspektive. Das Domänenmodell erhebt also nicht den Anspruch, jedes Detail der Fachdomäne zu beschreiben. Es muss aber genau die für die Implementierung wichtigen Artefakte und Prozesse abstrahieren und mit dem Fokus der späteren Umsetzbarkeit abbilden.

Das Domänenmodell ist keine Einbahnstraße von den Anforderungen zur Implementierung. Auch wenn der primäre Nutzen natürlich die Modellierung komplexer Zusammenhänge ist, dient es auch als Resonanzfläche der entwickelten Artefakte. Konkret bedeutet das: Wenn ein Entwickler merkt, dass sich im Modell festgeschriebene Beziehungen und Sachverhalte so nicht im Code abbilden lassen oder ungünstig formuliert sind, wird nicht nur der Code, sondern auch das Modell verändert. Und zwar so, dass Code und Modell zu jeder Zeit synchron bleiben.
Es genügt also nicht, ein Domänenmodell zu erstellen und den Code darauf aufzubauen. Das Domänenmodell lebt mit der Software und umgekehrt. Nur so ist der Mehrwert durch Domain Driven Design auch in Zukunft sichergestellt. Genauso wenig ist das Domänenmodell nur eine neue

Dokumentationsart. Es wird nicht nachgezogen, nachdem die Implementierung bereits stattgefunden hat, sondern bestimmt Designentscheidungen und dient als Basis für die gesamte Kommunikation im Team.

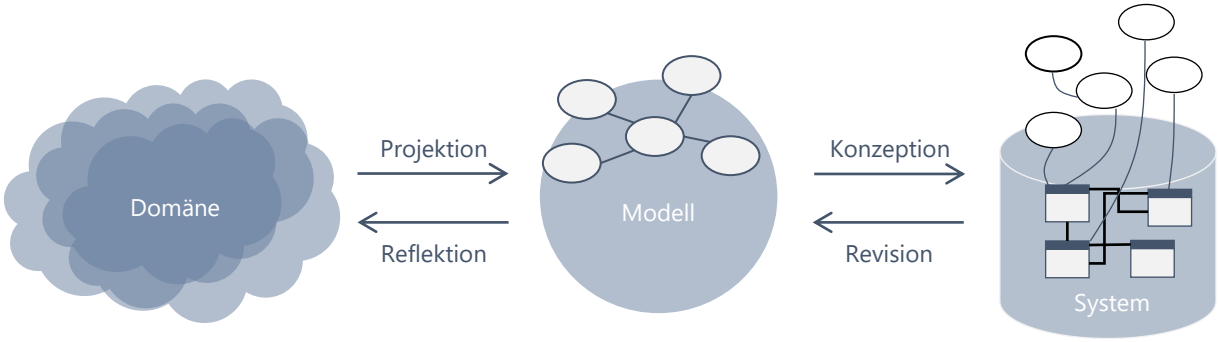



*Abbildung 3: Das Domänenmodell als Verbindung von Implementierung und Fachlichkeit*

**Die ubiquitäre Sprache – Der rote Faden im Softwaresystem**
von Dr. Ute Heimann

„To create a supple, knowledge-rich design calls for a versatile, shared team language" schreibt Eric Evans. Um Software gut zu designen, braucht das Team eine gemeinsame sprachliche Basis. Selbstverständlich nimmt kein Entwickler an, er würde den Kunden oder seinen Stellvertreter im Projekt nicht verstehen oder sich nicht verständlich ausdrücken. Trotzdem kommt es in Software-Projekten immer wieder zu Fehlinterpretationen und irreführenden Bezeichnungen. Spätestens nach einigen Jahren im Gebrauch weiß keiner mehr, was mit einem bestimmten Methodennamen genau gemeint war, es kommt zu unsachgemäßer Wiederverwendung und Erweiterungen – die Software erodiert.

Sprachliche Schwierigkeiten in der Entwicklung stabiler Software und Missverständnisse in Anforderungen und Umsetzung werden weder von technikfernen Fachexperten, noch von Code-fixierten Entwicklern verschuldet. Sie sind eine natürliche Herausforderung der Zusammenarbeit. Der folgerichtige Schritt wäre, ein Glossar anzulegen, um damit die Abkürzungen, Synonyme und kontrovers verwendete Begriffe besser zu verstehen. Wenn nötig, können sogar neue Begriffe erfunden werden, die einen Sachverhalt eindeutig machen oder vereinfachen.

Der Ansatz der ubiquitären Sprache im Domain Driven Design geht noch weiter. Alle an der Erstellung des Systems Beteiligten sollten die festgelegten Ausdrücke aktiv in einer gemeinsamen Sprache etablieren – Fachexperten, Entwickler, Business Analysten, Tester usw. Diese Sprache wird verwendet, um Anforderungen zu diskutieren und zu formulieren, um Testfälle zu erstellen und ja – auch in der Implementierung zur Bezeichnung von Klassen und Services.

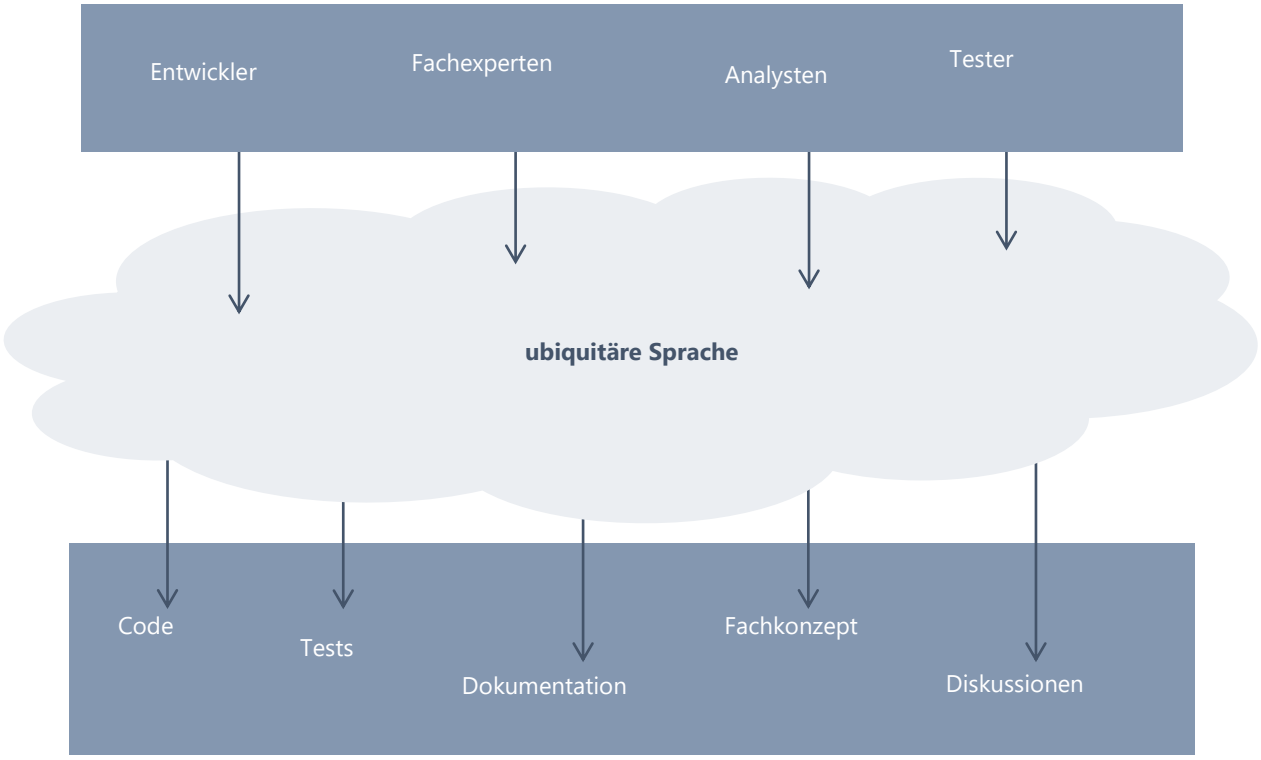



*Abbildung 4: Entstehung und Verwendung der ubiquitären Sprache*

Zwei Fragen stellen sich: Woher kommt diese Sprache und wie gelingt die konsequente Verwendung? Außerdem: Was heißt eigentlich „ubiquitär"?

„Ubiquitär" bedeutet „allumfassend", oder „überall vorhanden". Beides ist wichtig: Die Sprache muss alle relevanten Sachverhalte beschreiben und sie muss von allen im Team gesprochen und verstanden werden.

Die ubiquitäre Sprache fällt nicht von Himmel und sie wird auch nicht in Form eines Wörterbuches von der Fachabteilung herausgegeben. Stattdessen entsteht sie durch Diskussionen und Erläuterungen von allen Beteiligten. Im Fokus der ubiquitären Sprache steht das Domänenverständnis. Fachbegriffe und Prozesse werden hinterfragt und ihre Bedeutung genau erklärt und abgegrenzt. Sobald sich alle Beteiligten über die Bedeutung eines Begriffes verständigt haben, wird er ab sofort immer und von allen genau dieser Festlegung entsprechend verwendet. Dabei ist zu beachten, dass ein und derselbe Ausdruck in verschiedenen Kontexten unterschiedliche Bedeutungen haben kann. In diesem Fall darf der Ausdruck nur genutzt werden, wenn klar ist in welchem Kontext dies geschieht.

Die Aufgabe des Entwicklers ist nicht nur zu verstehen, was Fachexperten meinen, sondern auch zu abstrahieren und das Gesamtbild im Blick zu behalten. Zusätzlich bringt er sein eigenes technisches Verständnis ein. Denn auch wenn die Domäne im Fokus steht – umgesetzt werden soll ein IT-System.

Ein Beispiel:

*Fachexperte: Für uns arbeiten Berater in Kundenprojekten. Wir möchten ein System erstellen, das unsere Mitarbeiter und Freelancer verwaltet und die Kommunikationswege speichert.*

*Entwickler: Wir brauchen also eine Datenbank für Mitarbeiter und für Kontaktarten. Kein Problem.*

*Fachexperte: Nein, die Kontakte werden bei uns im CRM verwaltet. Und insbesondere möchten wir auch die Freelancer im System speichern.*

Was ist passiert? Der Entwickler hat sofort Freelancer und Mitarbeiter zu einer Klasse zusammengefasst und diese auch wieder mit Mitarbeiter bezeichnet – ob ein Mitarbeiter nun festangestellt ist oder nicht, ist schließlich nur ein Attribut. Außerdem hat er „Kommunikationswege" durch „Kontaktarten" ersetzt. Hätte er seine Überlegungen mit dem Fachexperten besprochen, dann wäre bestimmt herausgekommen, dass ein Kontakt für den Fachexperten ein Kundenkontakt ist und vielleicht würde der Experte einsehen, dass man Freelancer und Festangestellte unter dem Begriff Mitarbeiter sammeln kann. Vermutlich würde es sogar eine Diskussion darüber geben, wie groß die Unterschiede zwischen Freelancern und Festangestellten sind und ob sie wirklich in eine gemeinsame Klasse gehören.

Nehmen wir an, die Etablierung einer ubiquitären Sprache gelingt gut. Die Begriffe und festgestellten Zusammenhänge sind in einem Domänenmodell gut dokumentiert. Was für Auswirkungen hat das auf die Entwicklung? Wo ist der Unterschied zu einem reinen Glossar?

Zunächst einmal verschieben sich viele Entscheidungen auf einen früheren Zeitpunkt im Entwicklungsprozess, weil sie durch fachliche Argumente getroffen werden müssen. Im Beispiel von oben wäre es möglich, dass Freelancer und Festangestellte nichts miteinander zu tun haben, außer dass beide (genau wie Kunden) über Kommunikationswege erreicht werden können. Die Frage, ob sie beide von derselben Klasse erben, sollte dann aus fachlicher Perspektive diskutiert werden. Für Festangestellte macht es zum Beispiel keinen Sinn, umfangreiche Informationen über Verträge, Stundensätze und Beschäftigungszeiträume zu hinterlegen. Umgekehrt braucht ein Freelancer keine Urlaubsverwaltung. Ob die Unterschiede gravierend genug sind, in verschiedenen Kontexten implementiert zu werden, sollte keine technische Entscheidung sein.
Zum anderen ist die Sprache des Modells die Basis für die Implementierung. Sie verwendet dieselben Begriffe wie das Modell und umfasst genau die dort beschriebenen Operationen. Wird während der Umsetzung festgestellt, dass die Formulierung ungünstig ist, so wird auch das Modell geändert. Das Modell spiegelt also zu jedem Zeitpunkt die Implementierung und umgekehrt.

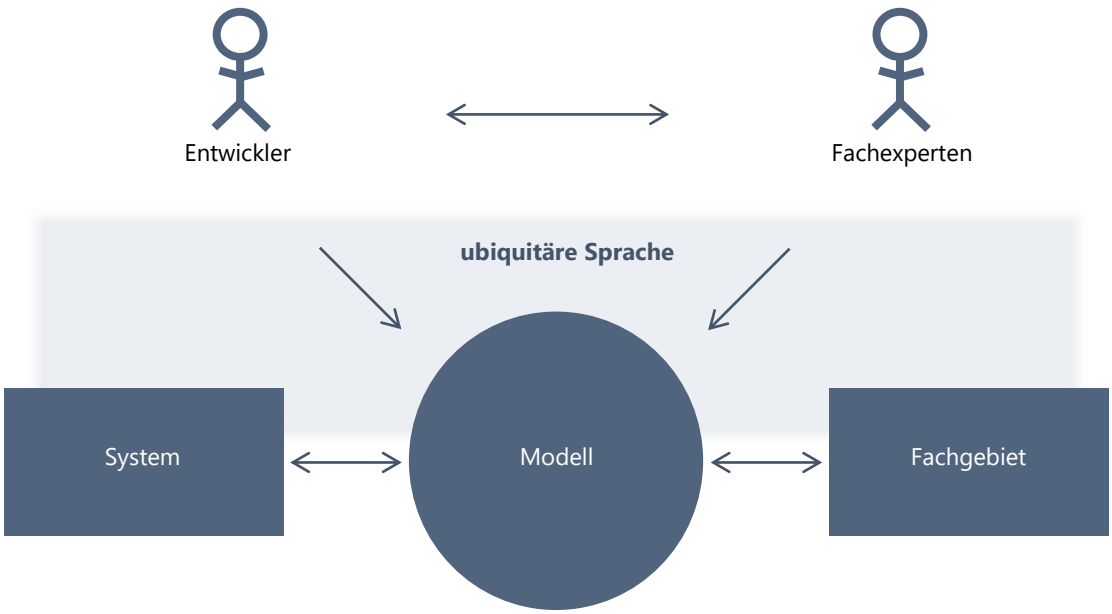



*Abbildung 5: Die ubiquitäre Sprache als Kommunikationsbasis im gesamten Prozess der Systementwicklung*

Was ist der Vorteil der konsequenten Verwendung der ubiquitären Sprache?

Missverständnisse werden vermieden, auch bei komplexen Systemen und insbesondere bei Änderungen und spät geäußerten Anforderungen. Die Art der Umsetzung von fachlichen Entscheidungen wird nicht mehr alleine von Entwicklern getroffen und Probleme kommen früher ans Licht, nämlich schon bei der Modellerstellung. Die Domäne rückt ins Zentrum der Entwicklung und garantiert die Stabilität des Systems über wechselnde Entwicklerteams, Umsetzungs-paradigmen und technische Infrastruktur hinaus.

**Strategic Design – Stabile Software durch gute Beziehungen**
von Dr. Ute Heimann

In den letzten Abschnitten wurde festgestellt, wie wichtig ein Domänenmodell und eine domänenbezogene Teamsprache für die Erstellung langlebiger und stabiler Software sind. Wie schafft man es aber, die Entwicklung so zu organisieren, dass große und komplexe Projekte nicht in kleinteiligen Modellen und einer unübersichtlichen Sprache versanden?

Domain Driven Design liefert darauf eine klare Antwort: Es müssen Zuständigkeiten und Beziehungen definiert werden. Nicht alle Teile eines Systems können oder müssen in einem großen

Modell erfasst werden. Wichtiger ist es, Teile, sogenannte Bounded Contexts, zu identifizieren, ihre Grenzen genau abzustecken und ihre Beziehungen zu anderen Modellteilen transparent zu machen.

Ein Bounded Context ist ein Teil des Gesamtsystems, der klar vom Rest des Systems getrennt und von einem Team entwickelt werden kann. Insbesondere hat innerhalb eines Bounded Context jeder Begriff der ubiquitären Sprache genau eine definierte Bedeutung. Zwischen Bounded Contexts gibt es im Allgemeinen keine Wiederverwendung von Code-Teilen, keine geteilten Datenbank-Tabellen usw.

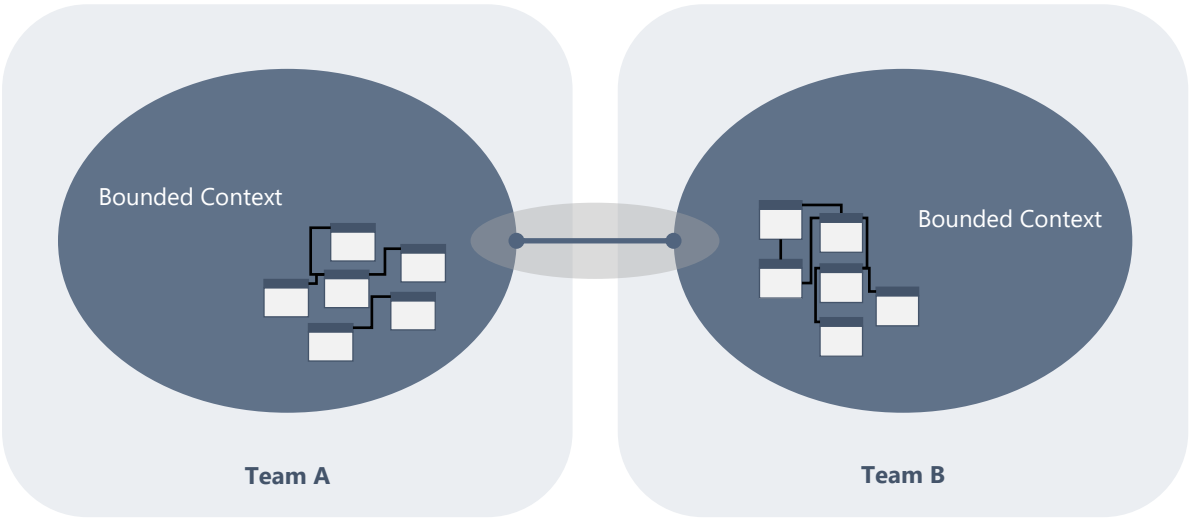



*Abbildung 6: Organisation von Teams und ihren Beziehungen anhand von Bounded Contexts*

Damit wird in Kauf genommen, dass unter Umständen aus unterschiedlichen Perspektiven derselbe Sachverhalt mehrfach entwickelt wird. Die Entscheidung für die Grenze eines Bounded Context wird also bewusst unter Berücksichtigung möglicher Mehrkosten durch erhöhten Entwicklungsaufwand getroffen und sorgfältig durchdacht. Sind die Kosten zu hoch, macht es vielleicht Sinn, Kontexte zusammen zu entwickeln, oder die Beziehung der Kontexte entsprechend zu gestalten.
Allein die Tatsache, dass dem Verhältnis von Bounded Contexts zueinander ein großer Stellenwert beigemessen wird, hilft Probleme zu vermeiden. Zusätzlich schlägt Eric Evans in seinem Buch *Tackling Complexity in the Heart of Software* verschiedene Patterns vor, mit denen sich die Kommunikation zwischen Bounded Contexts formalisieren lässt.

Wenn zwei Kontexte zwar nicht zusammen entwickelt werden sollen, sich aber auch nicht streng voneinander trennen lassen, sondern Überschneidungen aufweisen, kann das *Shared-Kernel-Pattern* verwendet werden. Hier wird genau definiert, welche Teile des Codes oder der Infrastruktur gemeinsam genutzt werden. Besonders wichtig sind gemeinsame Tests und eine gemeinsame ubiquitäre Sprache, um fehlerhafte Verwendung des Codes zu vermeiden.

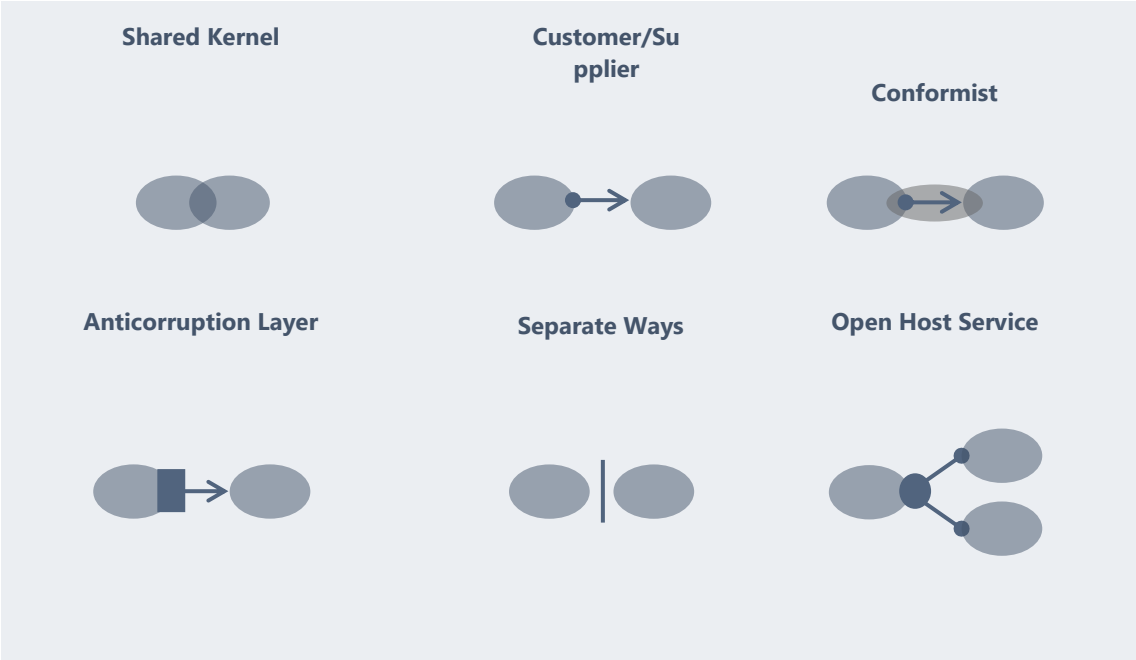



*Abbildung 7: Strategic Design Patterns*

Das setzt voraus, dass alle Bounded Contexts, die Anteil am geteilten Code haben, gleichberechtigt darauf zugreifen und Änderungen anstoßen können. Wenn dies z.B. aufgrund der Priorisierung von Projekten nicht gegeben ist, kommt ein Team in die Rolle von einem anderen Team abhängig zu sein. Das führt schnell zu Frustrationen und Verzögerungen, wenn nicht klar definiert wird, welche Rechte und Pflichten beide Teams haben. Wenn es eine zentrale Steuerungseinheit, zum Beispiel einen Abteilungs- oder Projektleiter, gibt, können beide Teams in einer *Customer-Supplier*-Beziehung entwickeln. Das *Supplier*-Team entwickelt unabhängig ein eigenes Domänenmodell, muss aber dem *Customer*-Team die benötigten Schnittstellen oder Informationen zur Verfügung stellen. Das *Customer*-Team passt sich an die vom *Supplier*-Team vorgegebene Struktur an, hat aber ein Veto-Recht bei Entscheidungen, die seine Arbeit behindern würden. Wichtig sind hier regelmäßige gemeinsame Integrationstests. So kann sich das *Supplier*-Team darauf verlassen, dass alle Änderungen vom *Customer*-Team berücksichtigt werden.

Wenn das *Supplier*-Team nicht mit dem *Customer*-Team zusammenarbeiten kann, ist dieses Pattern nicht anwendbar und würde die Entwicklung auf Seiten des Customer-Teams stark beeinträchtigen. Stattdessen empfiehlt Eric Evans das *Conformist*-Pattern, bei dem ein Team dem Design eines anderen Bounded Context folgt. Strukturentscheidungen werden damit zugunsten einer Kooperation abgegeben. Sinnvoll ist so ein Vorgehen auch bei ausgereiften Legacy-Systemen, die zwar nicht beeinflusst werden können, deren Struktur aber die Domäne gut erfasst.

Bestehende Systeme, die nicht geeignet sind, um Designentscheidungen vorzugeben, können mit einem *Anticorruption Layer* angebunden werden. Dieser Layer ist eine Art Übersetzungsschicht, die einem Bounded Context ermöglicht, ein eigenes Modell mit einer eigenen ubiquitären Sprache zu entwickeln, aber trotzdem auf einen anderen Kontext zuzugreifen.

Wichtig ist in jedem Fall die Abwägung von Vor- und Nachteilen eines bestimmten Patterns in der konkreten Situation. Wenn eine Verbindung von zwei Bounded Contexts hohe Kosten verursacht, sollte überlegt werden, ob sie wirklich nötig ist, oder ob beide Teams nach dem Pattern *Separate Ways* arbeiten. Das bedeutet, dass keine Wiederverwendung von Code-Teilen oder Infrastruktur stattfindet, dass jedes Team sein eigenes Modell erstellt und dass die Teile sich nicht gegenseitig beeinflussen.

Wenn ein Team zwar unabhängig von anderen Modellen entwickeln soll, aber mehrere weitere Bounded Contexts auf Funktionalitäten oder Daten zugreifen müssen, besteht die Möglichkeit, *Open Host Services* anzubieten. Benötigte Services werden so allgemein wie möglich formuliert und den anderen Teams zur Verfügung gestellt.

Wichtiger als die Frage, welches der vorgestellten Patterns im Einzelfall verwendet werden sollte, ist das grundsätzliche Bewusstsein dafür, dass es sinnvoll ist, komplexe Domänen in Bounded Contexts abzubilden und deren Beziehungen sorgfältig zu überdenken. Zwischen den beiden Extremen, dass jedes kleine Projekt unabhängig vom Gesamtsystem entwickelt oder dass Wiederverwendung und vollständige Integrität um jeden Preis durchgesetzt wird, gibt es noch viele weitere Möglichkeiten, die auch in Kombination miteinander eingesetzt werden können.

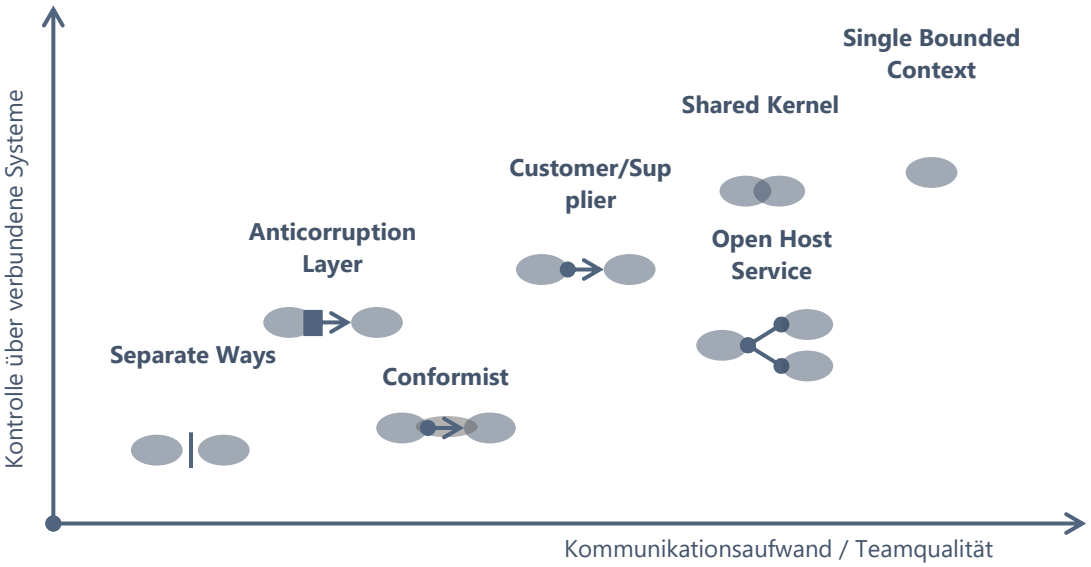



*Abbildung 8: Abwägung von Strategic Design Patterns*

Das spart nicht nur Zeit bei der Entwicklung und in der Kommunikation, sondern beseitigt auch Frustrationsquellen, wie unklare hierarchische Beziehungen und unterschiedlich verstandene Abhängigkeiten. Die Visualisierung der Bounded Contexts und ihrer Beziehungen in einer Context Map schafft außerdem einen Überblick über das Gesamtsystem und vereinfacht Weiterentwicklung und Wartung.

Strategic Design hilft also nicht nur, Softwareentwicklung besser zu strukturieren und für alle Beteiligten nachvollziehbar zu priorisieren, sondern liefert auch konkrete Lösungsansätze für Implementierung und Projektplanung. Zusammen mit der Verwendung einer ubiquitären Sprache und der Erstellung eines Domänenmodells ist Strategic Design die Basis für eine nachhaltige und stabile Softwareentwicklung – insbesondere für komplexe Systeme.

**Domain Driven Design – Taktisches Design**
von Christoph Schmidt-Casdorff

Das strategische Design liefert die Struktur des Lösungsraums und damit des Softwaresystems und ist Grundlage für das weitere Softwaredesign.

Im taktischen Design wird ein *Bounded Context* explizit und en Detail modelliert. Taktisches Design setzt also auf den Ergebnissen des strategischen Designs auf.

Da taktisches Design aufwendig ist (siehe unten), kommt es nicht für alle *Bounded Contexts* zur Anwendung. In der Regel werden diejenigen *Bounded Contexts* modelliert, welche von besonderer geschäftskritischer Bedeutung sind (*Core Domäne*).

Ziel des taktischen Designs ist es, alle fachlichen Konzepte des *Bounded Contexts* offenzulegen. Es werden ausschließlich fachliche Aspekte modelliert und keinerlei technologische Aspekte betrachtet. Während das bei Domänen-Objekten kein Problem darstellen sollte, ist es deutlich schwieriger fachliche Konzepte zu modellieren, welche sich hinter Assoziationen zwischen Domänen-Objekten, fachlichen Regeln oder Prozeduren und Verfahren verbergen. (Schlagwort*: Mache Implizites explizit*).

Die Diskussionen über die richtige Modellierung decken so oft tiefer liegende fachliche Konzepte auf. Oft sind es kleine Unstimmigkeiten, welche auf tiefer liegende bis nicht erkannte Konzepte führen.

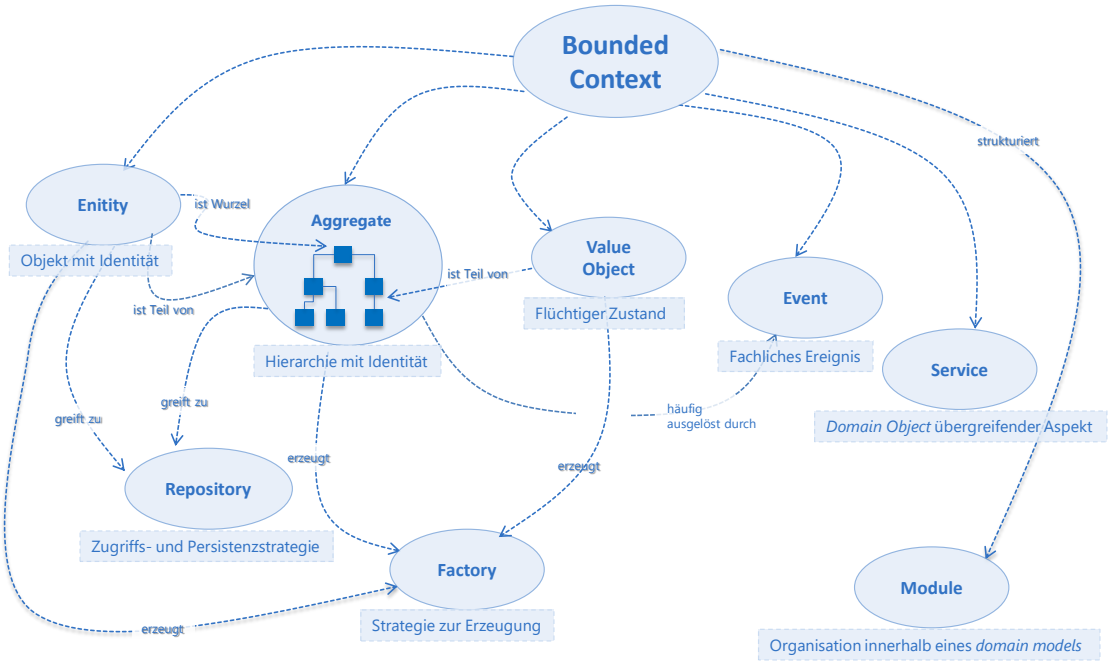
Dieses Vorgehen stützt sich darauf, dass die Konzepte immer wieder diskutiert und so geklärt werden. Daher ist auch in dieser Phase eine intensive und evolutionäre Zusammenarbeit zwischen Entwicklung und Experten unabdinglich.

Taktisches Design verfeinert das Domänenmodell spezieller *Bounded Contexts* und liefert ein Designmodell der Fachlichkeit dieses *Bounded Context*. Dieses Modell basiert auf objektorientiertem Design und beschreibt alle fachlich relevanten Objekte des *Bounded Contexts*.

Die Bezeichnungen von Objekten, Methoden und Assoziationen nutzen die ubiquitäre Sprache, sodass sich die Konzepte im Modell möglichst leicht wiederfinden lassen. In dieser Phase ist es denkbar (häufig sogar die Regel), dass die ubiquitäre Sprache angepasst und erweitert wird.

DDD führt die Pattern des *model driven modelling* ein. Diese liefern klare Vorgehensweise zur Bewertung und Darstellung von Modellobjekten. So besitzen *Entities* eine Identität und beschreiben Objekte mit kontinuierlichem Lebenszyklus (also auch mit zugehöriger Persistenz). *Value Objects* beschreiben im Gegensatz dazu flüchtige Zustände. *Aggregate* sind eine Aggregation/Hierarchie von *Entities* und *Value Objects*. Sie besitzen ebenso wie *Entities* eine Identität.

Abbildung 1 gibt einen Überblick über die Pattern. Eine detaillierte Darstellung der Pattern findet sich in Eric Evans[2] und Vaughn Vernon[3].




Nach *Domain-Driven Design: Tackling Complexity in the Heart of Software; Eric Evans*

*Abbildung 9: Taktisches Design (inspiriert durch Domain Driven Design: Tackling Complexity in the Heart of Software; Eric Evans)*

Aber was ist dieses Designmodell eines *Bounded Context* genau und wie wird es repräsentiert?

DDD grenzt sich in der Phase des taktischen Designs explizit von einem Vorgehen ab, bei dem Analysemodelle durch Experten vorgegeben und diese dann durch die Entwicklung umgesetzt werden. Bei einem solchen Vorgehen ist die schnelle Umsetzung von Feedbacks aus der Entwicklung nur schwer möglich und es kommt immer wieder zu Missverständnissen bei der Überführung des Analysemodells in Code.

Zur Beziehung zwischen Analysemodellen und DDD siehe [Evans, Kap. *Model Driven Design*)

Tatsächlich wird aber jedes Modell in letzter Konsequenz durch den implementierenden Code repräsentiert. Warum nicht den Code als Modell nutzen[4] ("*A Model Expressed in Software*"). Das Designmodell eines *Bounded Context* wird daher durch Code repräsentiert. Um als Basis für die Diskussion zwischen Experten und Entwicklung zu dienen, muss es von technologischen Aspekte weitestgehend befreit (so frei wie nur möglich)[5] und konsequent in der ubiquitären Sprache formuliert sein. Folgendes Zitat beleuchtet dies nochmals:

> *"Later you also add tests that verify the correctness of the new domain object from every possible (and practical) angle. At this point you are interested in the correctness of the expression of a domain concept that is embodied in the new domain object.*
> *Reading the demonstrative client-like test code must reveal the proper expressiveness using the Ubiquitous Language. Domain experts who are non-technical should be able to with the help of a developer read the code well enough to get a clear impression that the model has achieved the goal of the team. This implies that test data must be realistic, and support and enhance the desired expressiveness. Otherwise, domain experts cannot make a complete judgment about the implementation."* *[Domain-Driven Design Distilled; Vaughn Vernon, Chap 7; Timebox Modelling]*

Auch im taktischen Design gilt die Fokussierung auf die Fachlichkeit. Daher kann ein solches fachlich zentriertes Domänenmodell durch Code repräsentiert werden und trotzdem Diskussionsbasis sein. Es kann gar nicht genug betont werden, dass dieses Designmodell sich in enger Zusammenarbeit zwischen Entwicklung und Experten entwickelt.

Um den Experten die Möglichkeit zu gegeben, das Designmodell besser zu verstehen, bieten sich kollaborative Verfahren wie *behaviour driven development*[6] (BDD) an. BDD basiert auf Szenarien, welche beispielhaft die Arbeitsweise des Codes beschreiben. Diese Szenarien werden ganz bewusst aus Sicht der Experten formuliert (natürlich in ubiquitärer Sprache).

**Domain Driven Design – Im Mittelpunkt steht die Fachlichkeit**
von Christoph Schmidt-Casdorff

An der Entstehung und auch Pflege von Software sind viele unterschiedliche Personen mit unterschiedlichen Rollen und vor allem unterschiedlichen Sichten auf die Software und deren Fachlichkeit beteiligt. Aus dem Blickwinkel der Entwicklung ist Software das lauffähige Programm, aus dem Blickwinkel z.B. der Fachexperten ist die Software ein Spiegel ihrer Sicht auf die Fachlichkeit. Erst wenn alle Beteiligten am Softwareentwicklungsprozess ihre Sicht der Software und dem Softwareentwicklungsprozess wiederfinden, ist dieser Prozess stabil.

Als Konsequenz dieser Erkenntnis[7] rückt DDD ganz bewusst und konsequent die Fachlichkeit in den Mittelpunkt des Softwaredesigns. Software richtet sich nach fachlichen Gegebenheiten aus.

Im Mittelpunkt steht das Domänenmodell, welches in unterschiedlichen Phasen des Entwicklungsprozesses entsteht und schrittweise verfeinert wird.

Zentrale Domänenmodelle neigen dazu, die Verbindung zum Code zu verlieren, durch den sie letztlich umgesetzt werden. DDD hat die Gefahr erkannt, die sich ergibt, wenn diese beiden „Wahrheiten" auseinanderdriften. Das Ziel von DDD ist daher, die Verbindung eines Modells der Domäne einerseits sowie dem Design und Code andererseits in allen Phasen des Software-Entwicklungsprozess konsistent zu halten.

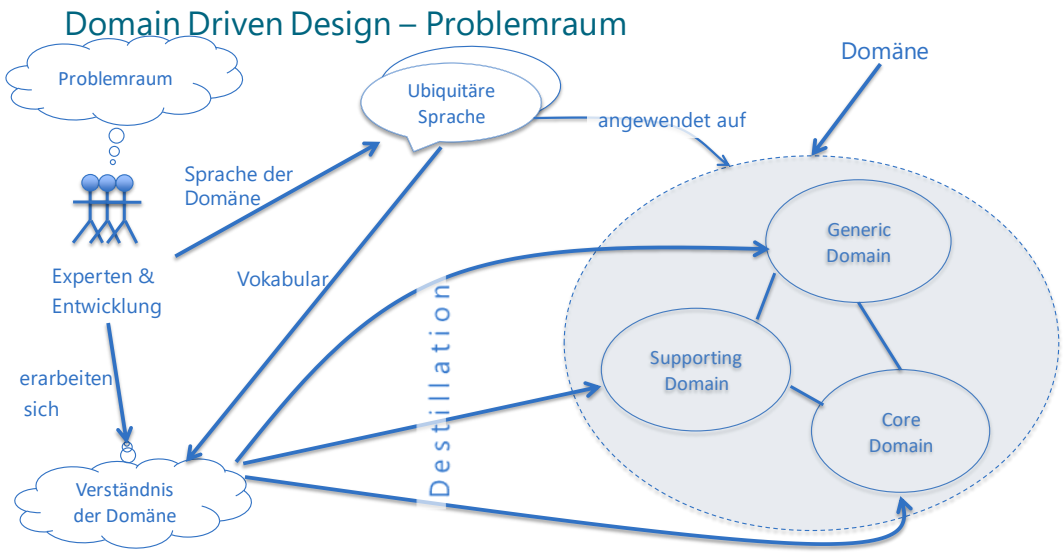
Methodisch betont DDD stark die enge Zusammenarbeit zwischen Experten und Entwicklung, um die Konsistenz zwischen Modell und Code auch tatsächlich gewährleisten zu können.

**Strategisches Design im Problemraum**

Im strategischen Design werden die strukturellen Entscheidungen des Systems getroffen. Im ersten Schritt wird der Problemraum analysiert. Der Problemraum beschreibt die Gesamtheit der Anforderungen.

Um diese Anforderungen zu strukturieren, wird im strategischen Design[8] der Problemraum in Subdomänen partitioniert (*Destillation*). Es entstehen getrennte Subdomänen, welche eine gemeinsame Sicht von Experten und Entwicklung auf das Gesamtsystem widerspiegelt.
Dieser Schritt ist sehr wichtig, bestimmt er doch die Struktur des entstehenden Systems (siehe Abbildung 1).

Ein weiteres wichtiges Resultat dieser Designphase ist die ubiquitäre Sprache[9].
Mit dieser ubiquitären Sprache existiert eine für Experten und Entwicklung gemeinsame Sprache, welche in alle Phasen und Artefakten zum Tragen kommt.




Nach https://leanpub.com/Practicing-DDD, Scott Millett, Leanpub anpub

*Abbildung 10: Domain Driven Design im Problemraum (inspiriert durch https://leanpub.com/Practicing-DDD)*

### Strategisches Design im Lösungsraum

Ist das Domänenmodell stabil, so wird auf dessen Basis der Problemraum in den Lösungsraum überführt. Diese Überführung lässt *Bounded Contexts* und die Bewertung der Zusammenarbeit dieser *Bounded Contexts* (*Collaboration Pattern, Context Map*) entstehen (siehe Abbildung 2).

Ein *Bounded Context* korrespondiert zu Subdomänen. Allerdings gilt leider nicht die einfache Formel, dass ein *Bounded Context* genau einer Subdomäne entspricht. Eine gute Beschreibung der Probleme, die bei der Überführung von Domänen/Subdomänen in *Bounded Contexts* zu bewältigen sind, findet sich in D. Esposito & A. Saltarello[10].

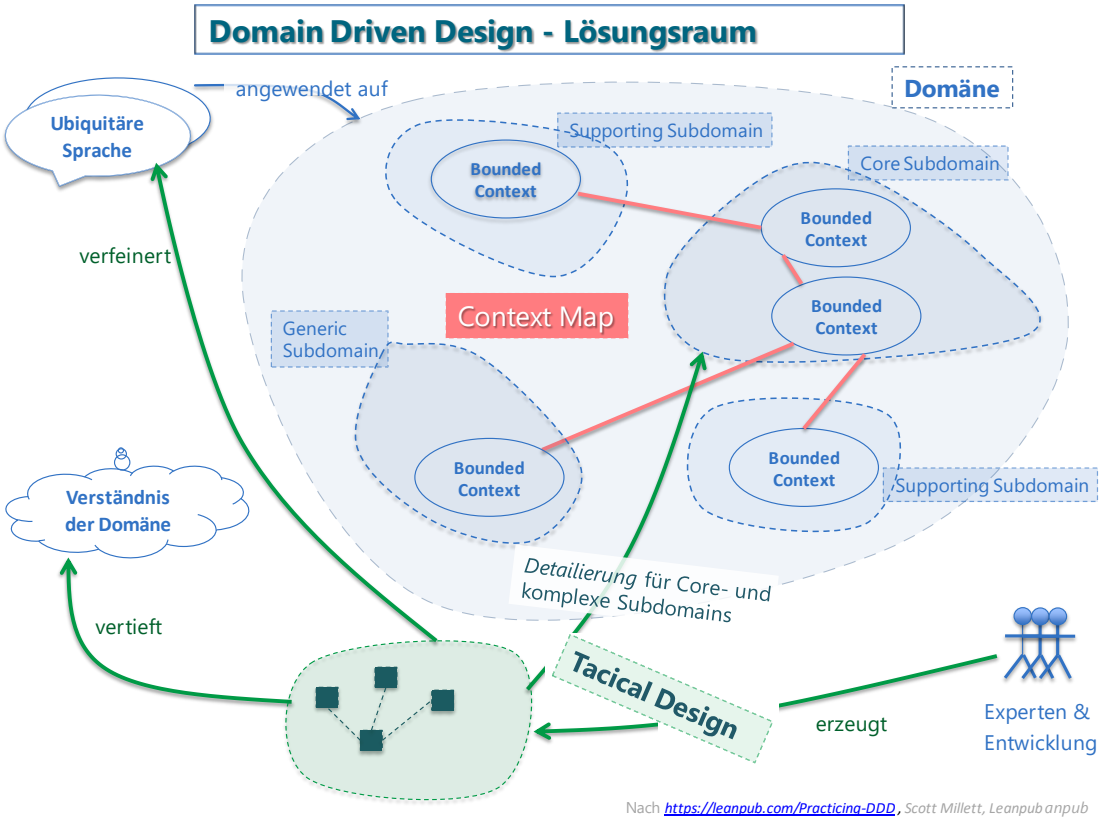



*Abbildung 11: Domain Driven Design im Lösungsraum (inspiriert durch https://leanpub.com/Practicing-DDD)*

Mit den *Bounded Contexts* liegt eine Grundlage für die Gesamtstruktur eines Softwaresystems vor. Diese Struktur ist durch fachliche Aspekte bestimmt. Die Struktur der Software ist daher in allen Phasen und von allen Beteiligten des Softwareentwicklungsprozesses nachzuvollziehen und wiederzuerkennen.

**Taktisches Design**

Für ausgewählte *Bounded Contexts* wird ein explizites und detailliertes Softwaredesign durchgeführt[11]. Das Modell basiert auf Code, ist aber frei von technologischen Aspekten. Dieses Modell ist Basis und Diskussionsgrundlage der Zusammenarbeit zwischen Experten und Entwicklung. Diese steht auch in dieser Phase im Mittelpunkt.

**Auswirkung**

Methodischer Mittelpunkt des DDDs ist die Zusammenarbeit zwischen Entwicklung und Fachexperten. In allen Phasen des DDDs wird Wert auf kollaboratives und evolutionäres Arbeiten gelegt, also schrittweise Verfeinerung mit gegenseitiger Unterstützung. Alle Ergebnisse des DDD (Subdomänen, *Bounded Contexts*, taktisches Design) gehören zur gemeinsamen Sicht von Fachexperten und Entwicklung auf die Fachlichkeit und werden gemeinsam entwickelt und fortgeschrieben.

Der Einsatz von DDD bedingt daher nicht nur neue Designkonzepte wie *Bounded Contexts,* sondern auch eventuell die Anpassung der Arbeitsweisen. DDD unterstützt agiles Vorgehen und integriert sich gut in agile Prozesse wie Scrum[12]. Während sich das taktische Design in Sprints integrieren lässt, ist strategisches Design im Allgemeinen (je nach Projekt-/Domänengröße) eine projektübergreifende Querschnittsaufgabe. Daher kann sie im Kontext von agilen Methoden mit der Aufgabe der *Architektur* verglichen werden[13].

Des Weiteren sind *Bounded Contexts* prädestiniert, um Teamstrukturen abzubilden (*Context Map*). Im DDD stellen sich Teams gemäß eines Bounded Context auf, nicht gemäß der Organisationsstrukturen. Auch aus diesem Grund fügt sich DDD gut in agile Prozesse.

**Wie geht es los?**

DDD kann ein wichtiger Teil des Entwicklungsprozesses sein. Allerdings ist DDD nicht für jedes Projekt oder System geeignet. Der Aufwand, den DDD mit sich bringt, muss gerechtfertigt sein[14].

Wie für alle komplexen Methoden ist auch für DDD eine schrittweise Einführung sinnvoll. Der einzuschlagende Weg hängt von einigen Rahmenbedingungen ab:

- Wie weit ist die agile Vorgehensweise bereits etabliert?
- Und ist die Zusammenarbeit zwischen Experten und Entwicklung bereits agil?

Im Mittelpunkt der Einführung sollten die Prinzipien und Grundhaltungen von DDD stehen, insbesondere die intensive Zusammenarbeit zwischen Experten und Entwicklung. Beide Seiten müssen sich aufeinander zubewegen und werden liebgewonnene Arbeitsweisen und Haltungen ändern müssen.

In Millet[15] sind einige der wichtigsten Hürden bei der Einführung beschrieben und Kriterien benannt, unter welchen Bedingungen DDD sinnvoll erscheint.

Es lohnt sich auf jeden Fall, sich mit DDD zu beschäftigen. Stellt DDD doch Konzepte und Methoden bereit, um im weiten Niemandsland zwischen Fachlichkeit und Software einen nachvollziehbaren Weg zu bahnen und damit die Stabilität des Softwaresystems zu verbessern.

**Referenzen**

[1] https://www.iks-gmbh.com/link/microservices-1
[2] Domain-Driven Design: Tackling Complexity in the Heart of Software; Eric Evans, Kap. 5+6
[3] Implementing Domain-Driven Design; Vaughn Vernon, 5-12
[4] Domain-Driven Design: Tackling Complexity in the Heart of Software; Eric Evans
[5] https://www.infoq.com/articles/ddd-in-practice
[6] https://www.okgrow.com/posts/what-is-bdd
[7] https://www.iks-gmbh.com/link/ddd-teil-1
[8] https://www.iks-gmbh.com/link/ddd-teil-2
[9] https://www.iks-gmbh.com/link/ddd-teil-3
[10] https://www.microsoftpressstore.com/articles/article.aspx?p=2248811
[11] https://blog.iks-gmbh.com/taktisches-design-domain-driven-design-teil-5
[12] Domain-Driven Design Distilled; Vaughn Vernon, Kap. 7
[13] https://allesagil.net/2011/06/27/scrum-und-architektur
[14] https://leanpub.com/Practicing-DDD Kap. 9-10
[15] https://leanpub.com/Practicing-DDD Kap. 9-10

**Kontakt:**
IKS Gesellschaft für Informations- und Kommunikationssysteme mbH
Siemensstraße 27
40721 Hilden

Telefon +49 2103 – 5872-0
info@iks-gmbh.com
www.iks-gmbh.com