
INNOQ
Modernes Domain-driven Design
Vom Geschäftsmodell bis zum System – agil und soziotechnisch
Michael Plöd

Modernes Domain-driven Design

Vom Geschäftsmodell bis zum System - agil und soziotechnisch

Michael Plöd

innoQ Deutschland GmbH
Krischerstraße 100 · 40789 Monheim am Rhein · Germany
Phone +49 2173 33660 · www.INNOQ.com

Layout: Tammo van Lessen with XeLaTeX
Design: Murat Akgöz
Typesetting: André Deuerling

Modernes Domain-driven Design – Vom Geschäftsmodell bis zum System - agil und soziotechnisch
Published by innoQ Deutschland GmbH
1. Auflage · November 2025

Inhaltsverzeichnis

Notizen

1 Einleitung

Domain-Driven Design (DDD) ist seit der Veröffentlichung von Eric Evans' Buch im Jahr 2003 ein prägender Ansatz für die Entwicklung komplexer Softwaresysteme. Doch seither hat sich viel getan – sowohl innerhalb der DDD-Community als auch in angrenzenden Disziplinen wie agiler Produktentwicklung, Organisationsdesign oder moderner Softwarearchitektur.

Dieser Primer ist für all jene gedacht, die sich in diesem Spannungsfeld bewegen:
- **Software-Entwickler:innen und Architekt:innen**, die nach praktikablen Mustern für robuste Systeme suchen.
- **Agile Coaches, Product Owner und Business Analysts**, die verstehen wollen, wie sich DDD als Brücke zwischen Business und Technik einsetzen lässt.
- **Engineering Leads und Führungskräfte**, die an der Schnittstelle zwischen Organisation, Architektur und Zusammenarbeit agieren.

1.1 Warum ein umfassender Überblick?

In den letzten Jahren sind neue Methoden und Ansätze entstanden, die den ursprünglichen Kern von DDD ergänzen und erweitern:
- **Collaborative Modelling**-Techniken wie EventStorming, Domain Storytelling oder Event Modelling haben gezeigt, wie wertvoll es ist, Business und Technik an einen Tisch zu bringen.
- **Strategisches DDD** hat massiv an Bedeutung gewonnen und wurde durch Werkzeuge wie die Bounded Context Design Canvas und von Konzepten aus *Team Topologies* aufgegriffen.
- **Taktisches DDD** wird heute in einem breiteren Architekturkontext gedacht – von klassischen Monolithen über Microservices bis hin zu Event-driven Architectures und Cloud-Native-Systemen.

Parallel dazu hat sich die Arbeitswelt verändert: Agile Methoden sind Mainstream geworden, Produktorganisationen arbeiten stärker cross-funktional, und Themen wie *Soziotechnische Architekturen* oder *Fast Flow Prinzipien* rücken in den Vordergrund.

1.2 Überblick über die Inhalte des Primers

Dieser Primer ist in mehrere Teile gegliedert, die jeweils unterschiedliche Facetten von modernem Domain-Driven Design beleuchten:

- **DDD als Arbeitsweise und Einstellung**
  Eine Einführung in die zentralen Ideen von Domain-Driven Design. Hier wird deutlich, dass DDD nicht nur ein technisches Pattern-Set ist, sondern eine Denk- und Arbeitsweise, die stark mit Prinzipien agiler Entwicklung korreliert: enge Zusammenarbeit, iteratives Vorgehen und Fokus auf Wertschöpfung. Missverständnisse werden aufgeklärt und erste Leitplanken für den Einsatz in modernen Produkt- und Organisationsumfeldern gesetzt.

- **Orientierungsansätze im DDD-Umfeld** Um von einer Haltung und Denkweise zu konkreten Praktiken zu gelangen, benötigen Teams Orientierung. Dieses Kapitel zeigt, wie DDD durch zwei komplementäre Ansätze greifbar wird: Der Model Exploration Whirlpool von Eric Evans verdeutlicht den iterativen Charakter von Modellierung als Lernprozess, während der DDD Starter Modelling Process der DDD Crew eine pragmatische Schrittfolge für den Einstieg bereitstellt. Gemeinsam helfen sie, Unsicherheit zu strukturieren und den Weg zu den folgenden Methoden des Primers vorzubereiten.

- **Collaborative Modelling**
  In diesem Teil geht es um die Methoden, wie Personen mit Fach- und Software-Engineering Wissen gemeinsam an einem Modell arbeiten. EventStorming, Domain Storytelling und Event Modelling helfen dabei, ein geteiltes Verständnis zu schaffen und implizites Wissen sichtbar zu machen. Diese Techniken fördern offene Kommunikation, erleichtern Requirements Engineering im agilen Umfeld und bilden die Basis für tragfähige Modelle.

- **Strategisches DDD**
  Hier steht die Strukturierung komplexer Domänen im Vordergrund. Konzepte wie Bounded Contexts, Subdomains und Context Maps helfen, Klarheit über Verantwortlichkeiten und Schnittstellen zu gewinnen. Ergänzt durch Werkzeuge wie die Bounded Context Design Canvas oder Core Domain Charts und Kon-

zepte aus *Team Topologies* wird gezeigt, wie man fachliche und organisatorische Grenzen so gestaltet, dass Wertströme optimal unterstützt werden.

- **Taktisches DDD**
  Dieser Teil widmet sich den konkreten Patterns und Architekturmöglichkeiten, die aus den fachlichen Modellen abgeleitet werden. Aggregates, Value Objects, Entities, Repositories und Domain Events bilden das Handwerkszeug für robuste Implementierungen. Ergänzt wird dies durch Architekturansätze von Monolithen über Microservices bis hin zu Event-driven Architectures sowie Muster wie Ports & Adapters, CQRS und Event Sourcing.

- **Abschluss: Do's and Don'ts im Domain-Driven Design**
  Domain-Driven Design ist kein Dogma, sondern eine Haltung, die Denken, Zusammenarbeit und Architekturentscheidungen prägt. Dennoch gibt es typische Stolperfallen, die fast jedes Team irgendwann erlebt und ebenso typische Erfolgsfaktoren, die über das Gelingen entscheiden. Dieses Abschlusskapitel fasst die wichtigsten Do's and Don'ts zusammen. Nicht als Checkliste, sondern als Orientierung für die Praxis.

1.3 Ziel dieses Primers

Dieser Primer soll einen **kompakten, aber umfassenden Überblick** über modernes Domain-Driven Design geben:
- Er ordnet die Vielzahl an Konzepten und Methoden ein.
- Er zeigt, wie DDD heute in Organisationen eingesetzt wird – nicht nur im Code, sondern auch in der Zusammenarbeit.
- Er benennt Stolperfallen und Missverständnisse, die in der Praxis immer wieder auftreten.

Mein Ziel ist nicht, jedes Detail auszuleuchten. Vielmehr möchte ich Orientierung geben und deutlich machen, wie sich Domain-Driven Design in den letzten Jahren weiterentwickelt hat. Wer tiefer einsteigen will, findet am Ende des Buches Hinweise auf weiterführende Ressourcen.

2 DDD als Arbeitsweise und Einstellung

2.1 Einleitung

Domain-Driven Design (DDD) wird häufig zunächst als eine Sammlung von Methoden, Mustern und Techniken verstanden. In der öffentlichen Diskussion stehen dabei oftmals **Artefakte** im Vordergrund – wie etwa *Bounded Contexts* oder *Aggregates*. Andere Architekturmuster wie *hexagonale Architektur* oder *Microservices* werden häufig im selben Atemzug genannt, sind jedoch nicht per se DDD-spezifische Artefakte, sondern eher verwandte Ansätze, die sich gut mit DDD kombinieren lassen.

Der eigentliche Kern von DDD liegt nicht primär in diesen Artefakten, sondern in dem **Weg**, der zu ihnen führt: *Wie identifiziere ich Bounded Contexts? Wie schneide ich Domänen? Warum entscheide ich mich für ein bestimmtes Architekturmuster – oder bewusst dagegen?* Genau dieser Prozess des Verstehens, Diskutierens und Entscheidens macht DDD aus. **Der Weg ist das Ziel.**

In diesem Kapitel geht es deshalb darum, zu zeigen, dass DDD nicht nur ein Werkzeugkasten für Architekt:innen und Entwickler:innen ist, sondern eine **Arbeitsweise und Einstellung**, die Zusammenarbeit, Denkweisen und Entscheidungsprozesse prägt. Dabei wird auch die enge Verbindung zu agilen Prinzipien sichtbar.

2.2 DDD als Arbeitsweise verstehen

DDD ist keine Methodik, die man einmal einführt und dann wie ein starres Regelwerk anwendet. Vielmehr handelt es sich um eine **kontinuierliche Praxis**, die sich auf das Verständnis der Fachdomäne konzentriert und dieses Verständnis zum Fundament der technischen Gestaltung macht. Wesentliche Merkmale dieser Arbeitsweise sind:

- **Iteratives Vorgehen**: Wissen über die Domäne wächst mit der Zeit. DDD betont, dass Modelle und Implementierungen gemeinsam mit diesem Verständnis reifen.
- **Enge Zusammenarbeit**: Fachliche Expert:innen und Entwickler:innen arbeiten nicht getrennt, sondern in einem fortlaufenden gemeinsamen Dialog.
- **Experimentieren und Lernen**: Hypothesen über die Domäne werden früh getestet, angepasst oder verworfen. Dabei geht es auch darum, implizite mentale Modelle der Beteiligten in **explizites, geteiltes Wissen** zu überführen. Alberto Brandolini, der Erfinder von EventStorming, hat es treffend formuliert: Was geht eigentlich in Produktion – das Wissen der Fachseite oder die Hypothesen und Annahmen, die in den Köpfen der Entwickler:innen über dieses Wissen existieren? Genau dieser Prozess des Sichtbarmachens und Abgleichens macht den Kern von Lernen im DDD-Kontext aus.

Diese Aspekte machen DDD anschlussfähig an agile Werte, wie sie etwa im **Agilen Manifest** beschrieben sind: Individuen und Interaktionen stehen über Prozessen und Werkzeugen; Reagieren auf Veränderung ist wichtiger als das Befolgen eines Plans.

2.3 Einstellung: Die Haltung hinter DDD

Neben der Arbeitsweise betont DDD eine spezifische **Einstellung**, die sich auf mehrere Dimensionen auswirkt:

1. **Respekt vor der Fachlichkeit**: DDD stellt die Domäne in den Mittelpunkt. Es geht nicht darum, Technologie um ihrer selbst willen einzusetzen, sondern die Geschäfts- und Anwendungslogik präzise zu verstehen und abzubilden.
2. **Gemeinsame Sprache** (**Ubiquitous Language**): Die Sprache der Fachlichkeit wird zur Sprache der Implementierung. Dies erfordert Offenheit, Empathie und den Willen, Brücken zwischen unterschiedlichen Disziplinen zu schlagen.
3. **Langfristige Wertorientierung**: Statt kurzfristige Optimierungen in den Vordergrund zu stellen, verfolgt DDD das Ziel, nachhaltige Modelle und Architekturen zu entwickeln.
4. **Akzeptanz von Komplexität**: DDD ist kein Werkzeug, um Komplexität zu eliminieren, sondern um sie **sichtbar und beherrschbar** zu machen.

5. **Kontinuierliches Lernen**: DDD lebt davon, dass Modelle nicht von Anfang an perfekt sein können. Iteration 1 kann daher oft nur den Anspruch erfüllen, „stets sehr bemüht“ gewesen zu sein. Erst durch fortlaufendes Lernen, Reflektieren und gemeinsames Weiterentwickeln entsteht ein wirklich tragfähiges Modell. Diese Haltung verbindet DDD eng mit agilen Prinzipien, in denen iteratives Vorgehen und kontinuierliche Verbesserung zentral sind.

2.4 Die Korrelation zu Agilität

DDD lässt sich nicht isoliert von agilen Prinzipien betrachten. Vielmehr verstärken sich beide Ansätze gegenseitig. Besonders im **Agilen Manifest** lassen sich zahlreiche Parallelen zu DDD finden:

- **Business People and Developers must work daily together throughout the project**: Dieses Prinzip wird in DDD konkret umgesetzt, indem Domänenexpert:innen und Entwickler:innen gemeinsam Modelle entwickeln und kontinuierlich anpassen. Ubiquitous Language und kollaborative Workshops sind direkte Ausdrucksformen dieses Prinzips.
- **Feedbackzyklen**: Agile Methoden setzen auf kurze Iterationen, in denen Feedback eingeholt wird. DDD konkretisiert diesen Ansatz, indem es Feedback nicht nur auf technischer Ebene, sondern auch auf der Ebene der Modelle einfordert.
- **Kollaboration**: Agilität betont funktionsübergreifende Teams. DDD macht diese Kollaboration konkret, indem es die Fachdomäne zum verbindenden Element erklärt.
- **Anpassungsfähigkeit**: Agilität will auf Veränderungen reagieren können. DDD bietet Mechanismen wie **Bounded Contexts** und **Context Maps**, um diese Veränderungen strukturiert in die Architektur einfließen zu lassen.

2.5 Praktische Implikationen

DDD als Arbeitsweise und Einstellung zeigt sich im Alltag in zahlreichen Facetten:

2.5.1 Workshops und kollaborative Modellierung

Formate wie **Event Storming** oder **Domain Storytelling** verdeutlichen, dass DDD stark auf visuelle, gemeinschaftliche und interaktive Methoden setzt. Dabei entstehen Modelle, die sowohl für Fach- als auch Technik-Expert:innen verständlich sind.

2.5.2 Architekturentscheidungen

Die Haltung von DDD führt dazu, dass Architektur nicht isoliert am „Reißbrett" entsteht, sondern aus der Fachlichkeit herauswächst. Entscheidungen wie die Aufteilung in Bounded Contexts sind das Resultat fachlicher Abgrenzungen.

2.5.3 Priorisierung und Wertschöpfung

DDD schärft den Blick darauf, welche Teile einer Domäne **strategisch relevant** sind. Statt alles gleich zu behandeln, unterscheidet DDD zwischen Kern-Domäne, unterstützenden Subdomänen und generischen Subdomänen. Daraus leiten sich Investitionsentscheidungen ab.

2.5.4 Kontinuierliches Refactoring

Die Einstellung hinter DDD akzeptiert, dass Modelle nie perfekt sind. Stattdessen gilt es, sie kontinuierlich zu verbessern, sobald neues Wissen vorliegt.

2.6 Herausforderungen in der Umsetzung

DDD als Haltung zu verinnerlichen ist nicht trivial. Eine der größten Hürden besteht im **Silodenken**: Wenn Fachbereiche und IT strikt getrennt agieren, fehlt die Basis für eine gemeinsame Sprache und für den notwendigen kontinuierlichen Dialog.

Ebenso herausfordernd ist der allgegenwärtige **Zeitdruck**. In Projekten mit engen Deadlines wird das sorgfältige Modellieren oft als „Luxus" betrachtet, obwohl es langfristig entscheidend für die Qualität und Nachhaltigkeit der Systeme ist.

Darüber hinaus gibt es zahlreiche **Missverständnisse** in der Praxis. Häufig begegnet man dogmatischen Ansichten zu Domain-Driven Design. So entsteht etwa der Eindruck, man müsse zwangsläufig bei einer Microservice-Architektur landen, wenn man DDD ernsthaft betreibt. Andere vertreten die Meinung, DDD verlange zwingend nach einer hexagonalen Architektur oder führe automatisch zu eventgetriebenen Systemen. Diese Sichtweisen sind verkürzt und falsch. DDD schreibt keine spezifische Zielarchitektur vor, sondern stellt vielmehr Denkwerkzeuge und Praktiken bereit, um für eine konkrete Domäne und deren Kontext die passenden Entscheidungen zu treffen. Genau darin liegt die eigentliche Stärke von DDD: Es geht um den Weg der Erkenntnis und die bewusste Auswahl, nicht um vorgefertigte dogmatische Lösungen.

Ein weiterer Punkt, der oft übersehen wird, ist der **Umfang der Adoption von Domain-Driven Design**. In seiner Vollausbaustufe ist DDD zeitintensiv und damit auch kostenintensiv. Es wäre ineffizient, diese Herangehensweise auf jede Subdomäne oder jeden Teil eines Systems gleichermaßen anzuwenden. Stattdessen propagiert DDD sehr deutlich, dass die Methoden und Praktiken dort ihren größten Nutzen entfalten, wo es um **Core-Domains** geht – also jene Bereiche, die für den strategischen, wirtschaftlichen oder geschäftlichen Erfolg mittel- bis langfristig entscheidend sind. In unterstützenden oder generischen Subdomänen reicht es hingegen oft, pragmatischere Ansätze zu wählen.

2.7 Chancen und Mehrwert

Die Chancen und Mehrwerte von Domain-Driven Design liegen weniger in allgemeinen Schlagworten wie Kommunikation oder Nachhaltigkeit, sondern in ganz konkreten Effekten, die durch die enge Verbindung von fachlichem Verständnis und technischer Umsetzung entstehen.

Ein zentraler Mehrwert sind **besser wartbare Systeme**. Weil die Struktur des Codes unmittelbar aus der fachlichen Denke entspringt, wird die Software klarer, nachvollziehbarer und damit robuster. Fachliche Änderungen können leichter in die technische Welt übertragen werden, ohne dass Brüche entstehen. Dies verringert das Risiko, dass Systeme unbeherrschbar werden.

Ein weiterer großer Vorteil liegt im **Alignment**: DDD fördert die Ausrichtung von fachlichen Grenzen mit den vertikalen Modulen der Software-Architektur. Bounded Contexts werden so nicht nur ein architektonisches Konstrukt, sondern helfen auch, Deployment-Grenzen sauber zu ziehen und Verantwortlichkeiten klar zu schneiden. Dieses Alignment lässt sich auch auf organisatorische Aspekte übertragen. Betrachtet man etwa die Ideen aus *Team Topologies*, dann sind Bounded Contexts ein hervorragender Startpunkt für das Zuschneiden von Teams. So entstehen Teams, deren Verantwortung mit den fachlichen Grenzen übereinstimmt und die autonom arbeiten können.

DDD schafft also nicht nur technisch saubere Architekturen, sondern trägt auch dazu bei, Organisation, Architektur und Domäne in Einklang zu bringen. Diese Verbindung von Technik und Organisation ist ein spezifischer und strategisch relevanter Mehrwert, der DDD von vielen anderen Ansätzen abhebt.

2.8 Fazit

DDD als Arbeitsweise und Einstellung ist weit mehr als eine Methodensammlung. Es verbindet technisches Handwerk mit einer Haltung, die **Zusammenarbeit, Respekt vor der Fachlichkeit und die bewusste Auseinandersetzung mit Komplexität** ins Zentrum stellt. Im Zusammenspiel mit agilen Prinzipien und den Werten des **Agilen Manifestos** entsteht eine kraftvolle Grundlage, um komplexe Softwaresysteme nachhaltig zu gestalten.

3 Orientierungsansätze im DDD-Umfeld

Im ersten Teil dieses Primers haben wir gesehen, dass DDD weit mehr ist als ein Werkzeugkasten aus Mustern und Artefakten. Es ist ein Denkansatz, der Zusammenarbeit, Respekt vor Fachlichkeit und kontinuierliches Lernen in den Mittelpunkt stellt. Doch damit stellt sich für viele Teams sofort die praktische Frage: *Wie genau sollen wir starten?*

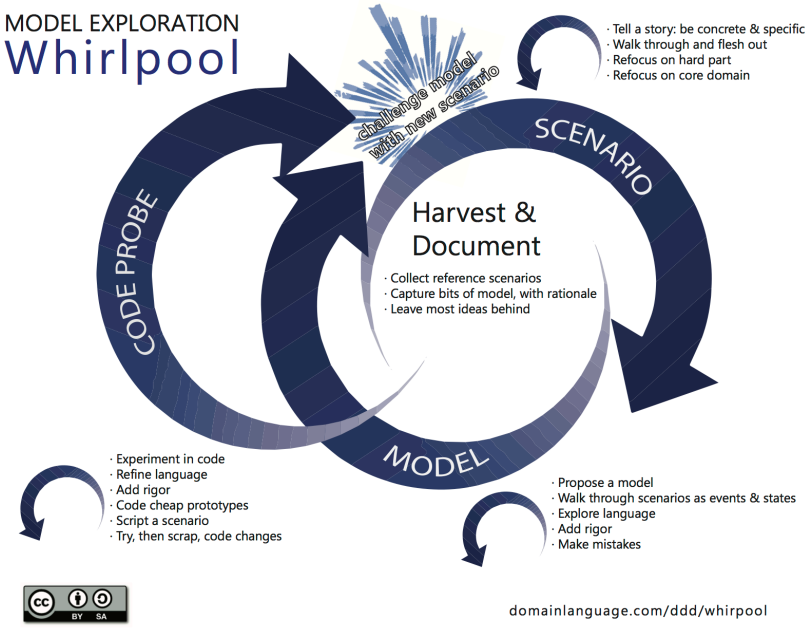
An diesem Punkt kommen Orientierungsansätze ins Spiel. Sie helfen, die abstrakte Haltung greifbar zu machen und in konkrete erste Schritte zu übersetzen. Dabei ist wichtig, dass DDD keine starre Methodik vorgibt, die von A bis Z durchlaufen wird. Vielmehr geht es darum, Wege aufzuzeigen, die Teams Orientierung geben, ohne sie in ein Korsett zu zwingen.

In diesem Kapitel stellen wir zwei Ansätze vor, die in der DDD-Community besondere Bedeutung erlangt haben: den **Model Exploration Whirlpool** von Eric Evans und den **DDD Starter Modelling Process** der DDD Crew. Sie sind auf unterschiedliche Weise hilfreich: Der Whirlpool beschreibt die Denk- und Lernbewegungen, die beim Modellieren natürlicherweise entstehen, während der Starter Process eine pragmatische Schrittfolge anbietet, die sich besonders für den Einstieg eignet. Zusammen bilden sie einen wertvollen Bezugsrahmen, um DDD im Projektalltag anzuwenden.

3.1 Modellieren als Kreislauf: Der Model Exploration Whirlpool

Eric Evans, der Begründer von Domain-Driven Design, hat früh erkannt, dass das Modellieren von Fachlichkeit kein linearer Prozess ist. Anforderungen zu sammeln, ein Modell zu entwerfen und dieses dann unverändert umzusetzen – so funktioniert es in der Realität nicht. Stattdessen bewegen sich Teams in Schleifen, verfeinern Ideen, verwerfen Hypothesen und lernen dabei Schritt für Schritt, die Domäne besser zu verstehen.

Um diesen Prozess zu beschreiben, hat Evans die Metapher des **Model Exploration Whirlpool** geprägt. Der Whirlpool steht für ein ständiges Kreisen zwischen verschiedenen Tätigkeiten: Fachgespräche führen, Modelle aufstellen, Prototypen umsetzen, Grenzen neu ziehen und Hypothesen überprüfen. Es gibt keinen festen Anfang und kein endgültiges Ende. Der Strom des Whirlpools reißt die Beteiligten immer wieder hinein und sorgt dafür, dass das Modell mit jeder Drehung ein Stück reifer wird.



(Bild Quelle: https://domainlanguage.com/ddd/whirlpool)

Diese Metapher verdeutlicht mehrere zentrale Einsichten:

- **Iteratives Lernen ist unvermeidbar.** Ein Modell, das beim ersten Versuch „fertig“ ist, existiert in der Praxis nicht.
- **Fachlichkeit und Technik bedingen einander.** Erkenntnisse aus der Implementierung fließen zurück ins Modell, ebenso wie fachliche Diskussionen unmittelbare Auswirkungen auf den Code haben.

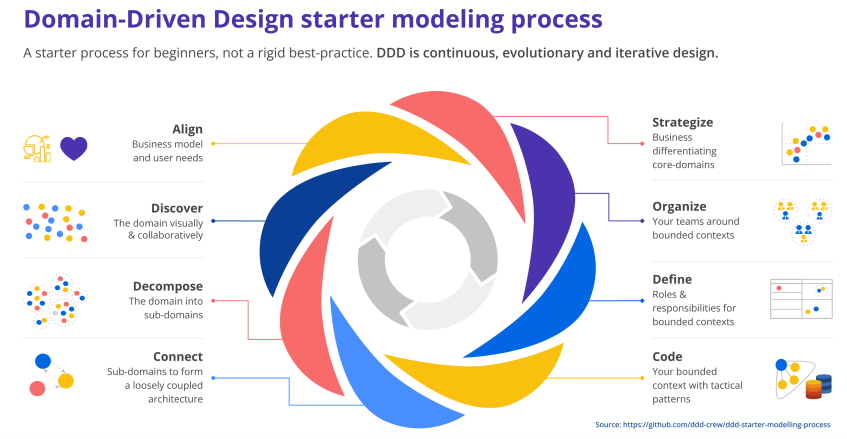
- **Unsicherheit ist Teil des Prozesses.** Anstatt sich von vagen oder widersprüchlichen Anforderungen entmutigen zu lassen, akzeptiert der Whirlpool diese Unsicherheit als Normalzustand.

Für Teams ist der Model Exploration Whirlpool eine Einladung, **Gelassenheit im Umgang mit Komplexität** zu entwickeln. Er vermittelt die Sicherheit, dass es nicht darum geht, von Anfang an die perfekte Lösung zu finden, sondern dass Lernen und Anpassung integraler Bestandteil von DDD sind.

3.2 Schritt für Schritt starten: Der DDD Starter Modelling Process

Während der Whirlpool eine hilfreiche Denkmetapher liefert, brauchen Teams in der Praxis oft etwas Handfesteres. Gerade für den Einstieg ist es wichtig, einen roten Faden zu haben, der Orientierung gibt. Hier setzt der **DDD Starter Modelling Process** der DDD Crew an.

Dieser Prozess bietet eine **leichtgewichtige Abfolge von Schritten**, die es auch weniger erfahrenen Teams erlaubt, die ersten Modellierungssitzungen zu gestalten. Er ist nicht als strenges Regelwerk zu verstehen, sondern als pragmatische Sammlung von Empfehlungen, die sich in vielen Projekten bewährt haben.



(Bild Quelle: https://github.com/ddd-crew/ddd-starter-modelling-process)

Typische Elemente dieses Prozesses sind:

- **Ein gemeinsames Verständnis der Domäne schaffen.** Häufig geschieht dies in Form eines Explorationsworkshops, etwa durch EventStorming oder ähnliche Techniken. Ziel ist es, implizites Wissen sichtbar zu machen und eine gemeinsame Sprache aufzubauen.
- **Erste Subdomänen und Bounded Contexts identifizieren.** Statt die gesamte Komplexität auf einmal zu erfassen, werden Teilbereiche abgegrenzt und mit groben Verantwortlichkeiten versehen.
- **Die Ubiquitous Language entwickeln.** Begriffe, die in den Diskussionen auftauchen, werden präzisiert und in den Modellen wie im Code konsequent verwendet.
- **Schrittweise verfeinern.** Der Prozess ermutigt dazu, mit einfachen Visualisierungen oder Prototypen zu beginnen und diese nach Bedarf zu erweitern.

Besonders wertvoll am Starter Modelling Process ist seine **Pragmatik**: Er ist klar genug, um Orientierung zu geben, aber offen genug, um an unterschiedliche Kontexte angepasst zu werden. Teams, die bislang keine Erfahrung mit DDD haben, können sich daran entlanghangeln und erste Erfolge erzielen, ohne im Detail zu wissen, wie das gesamte Puzzle am Ende aussieht.

3.3 Whirlpool und Starter Process im Zusammenspiel

Auf den ersten Blick könnten Whirlpool und Starter Process wie Gegensätze wirken: Der eine beschreibt ein offenes, nichtlineares Kreisen, der andere eine Schrittfolge. In Wahrheit ergänzen sie sich jedoch ideal.

Der **Model Exploration Whirlpool** liefert die **Philosophie**: Modellieren ist kein geradliniger Ablauf, sondern ein iteratives Lernspiel, das nie abgeschlossen ist. Er sensibilisiert Teams dafür, dass Rücksprünge und Schleifen kein Fehler, sondern der Normalfall sind.

Der **DDD Starter Modelling Process** liefert die **Praxis**: Er gibt Teams einen konkreten Ablaufplan, an dem sie sich orientieren können – gerade in den ersten Sitzungen, wenn Unsicherheit groß ist.

Zusammengenommen entsteht so ein **doppelter Bezugsrahmen**: Der Whirlpool erklärt, warum DDD so funktioniert, wie es funktioniert. Der Starter Process zeigt, wie man im Alltag trotzdem ins Handeln kommt.

3.4 Ausblick: Unser Bezugspunkt im weiteren Primer

Im weiteren Verlauf dieses Primers – bei den Kapiteln zu **Collaborative Modelling**, zu **strategischem DDD** und zum **taktischen DDD** – werden wir uns auf den **DDD Starter Modelling Process** beziehen. Er dient uns als roter Faden, weil er gut geeignet ist, Methoden wie EventStorming, Domain Storytelling oder Context Mapping in eine nachvollziehbare Abfolge zu bringen.

Gleichzeitig bleibt der Model Exploration Whirlpool ein wichtiger Hintergrundgedanke. Denn egal, welche Technik oder welches Werkzeug eingesetzt wird: Das Denken in iterativen Lernzyklen ist die Grundlage dafür, dass DDD seine volle Wirkung entfalten kann.

3.5 Fazit

Mit diesem Kapitel schlagen wir die Brücke vom Fundament hin zu den konkreten Praktiken. Wir haben zwei Ansätze kennengelernt, die unterschiedliche, aber komplementäre Perspektiven bieten. Der Model Exploration Whirlpool erinnert uns daran, dass DDD ein fortlaufender Lernprozess ist. Der DDD Starter Modelling Process gibt uns ein handhabbares Vorgehen, um in diesen Prozess einzusteigen.

Zusammen bilden sie den Bezugsrahmen, mit dem wir in den kommenden Kapiteln weiterarbeiten werden – sei es bei der kollaborativen Modellierung, bei strategischen Überlegungen oder beim taktischen Design. Wer DDD verstehen und anwenden will, braucht beides: das Vertrauen in den Wirbel des Lernens und den Mut, mit klaren ersten Schritten loszugehen.

4 Collaborative Modelling

4.1 Einführung & Prinzipien

Im ersten Teil dieses Primers hast Du gesehen, dass Domain-Driven Design (DDD) keine reine Methodensammlung ist, sondern eine Arbeitsweise und Einstellung, die Zusammenarbeit, Respekt vor Fachlichkeit und kontinuierliches Lernen in den Mittelpunkt stellt. Genau hier setzt die vergleichsweise junge Disziplin des **Collaborative Modelling** an: Sie macht diese Prinzipien praktisch erlebbar in dem sie hoch interaktiv ist und keine nennenswerten Einstiegshürden hat.

Collaborative Modelling beschreibt eine Familie von Methoden, in denen Menschen mit unterschiedlichem Hintergrund – Fachexpert:innen, Entwickler:innen, Product Owner, Agile Coaches oder Business Analysts – **gemeinsam fachliche Modelle entwickeln**, statt sie in Silos zu erarbeiten und dann in Form von Dokumenten bildlich betrachtet „über Zäune werfen“. Die Leitidee ist, implizites Wissen sichtbar zu machen und eine gemeinsame Sprache aufzubauen. Damit werden Missverständnisse frühzeitig erkannt und reduziert, Risiken gesenkt und die Grundlage für tragfähige Systeme geschaffen.

Wesentliche Prinzipien sind:

- **Gemeinsam statt getrennt:** Modelle entstehen nicht durch Übergaben, sondern in direkter Zusammenarbeit. Alle Beteiligten, von Fachexpert:innen bis zu Entwickler:innen, arbeiten zusammen, um ein einheitliches und klares Verständnis der Problemstellung zu bekommen.
- **Visuell statt textlastig:** Durch einfache Visualisierungen auf Whiteboards, Sticky Notes oder digitalen Boards wird Wissen greifbar und überprüfbar. Dies erleichtert die Kommunikation, macht komplexe Zusammenhänge sichtbar und fördert die aktive Teilnahme aller Teammitglieder.
- **Explorativ statt deterministisch:** Modellieren ist ein laufender Lernprozess, eine Konversation. Hypothesen dürfen ausprobiert und verworfen werden. Das Modell wird nicht in einem einzigen Schritt perfektioniert, sondern in

Zyklen entwickelt. Durch kontinuierliches Feedback und Anpassungen wird das Modell schrittweise verbessert und an neue Erkenntnisse angepasst.

- **Fokus auf Wertschöpfung:** Die Modellierung ist kein Selbstzweck, sondern konzentriert sich konsequent darauf, ein konkretes Problem zu lösen und einen messbaren Geschäftswert zu schaffen. Es geht nicht darum, jedes Detail zu dokumentieren, sondern die entscheidenden Elemente für Produktziele und Bedarfe von Nutzer:innen herauszuarbeiten.

Man könnte Collaborative Modelling als eine **moderierende Business Analyse für agile Umfelder** beschreiben. Statt Anforderungen in dicken Dokumenten festzuhalten, entstehen lebendige Modelle in kurzen Iterationszyklen, die von allen Beteiligten getragen und verstanden werden. Dies macht die Methoden nicht nur für Software-Architekt:innen relevant, sondern auch für Rollen wie Agile Coaches, Product Owner oder Requirements Engineers.

Im Folgenden betrachten wir ausgewählte Methoden im Kontext der Phasen „Align" und „Discover" des im vorherigen Kapitel erwähnten **DDD Starter Modelling Processes**.

4.2 Phase Align: Business Model Canvas und Value Proposition Canvas

Die erste Phase des DDD Starter Modelling Processes trägt den Namen **Align**. Bevor es darum geht, eine Domäne in der Tiefe zu erkunden, muss Klarheit über die übergeordneten Ziele herrschen: *Wozu existiert unser Produkt? Welche Kund:innen adressieren wir? Welchen Wert wollen wir schaffen?*

4.2.1 Business Model Canvas

Das **Business Model Canvas** (**BMC**) ist ein Werkzeug, das diese Fragen systematisch beantwortet. Auf einer einzigen Seite werden zentrale Aspekte wie Kundensegmente, Wertangebote, Kanäle, Einnahmequellen und Kostenstruktur sichtbar gemacht.

Facilitation-Tipps:
- Arbeite iterativ: erst grob ausfüllen, dann verfeinern.
- Achte auf Verständlichkeit – jede:r im Raum sollte die Begriffe nachvollziehen können.
- Nutze Farben oder Symbole, um Unsicherheiten oder offene Fragen zu markieren.

Das BMC ist besonders wertvoll, um **eine geteilte Geschäftslogik** zu entwickeln. In DDD-Sprache: Es schafft das Fundament, auf dem Subdomänen und Bounded Contexts später geschnitten werden.

4.2.2 Value Proposition Canvas

Das **Value Proposition Canvas** (**VPC**) vertieft einen Teilbereich des BMC: die Passung zwischen Kundensegment und Wertangebot. Es stellt die Frage: *Welche Jobs haben unsere Kund:innen, welche Schmerzen erleben sie, welche Gewinne erhoffen sie sich – und wie adressiert unser Produkt diese?*

Facilitation-Tipps:
- Beginne mit den Kund:innen, nicht mit dem Produkt.
- Nutze konkrete Beispiele, um Abstraktion zu vermeiden.
- Lass unterschiedliche Perspektiven zu: Marketing, Technik und Fachlichkeit haben oft verschiedene Sichtweisen.

Im Kontext von DDD hilft das VPC, Grundlagen für eine Diskussion um **strategische Prioritäten** zu schaffen: Welche Teile der Domäne sind Kern-Domänen, weil sie direkten Kundennutzen erzeugen? Wo reicht Standardsoftware? Wir werden diesen Aspekt, später im „Strategize“ Teil des Folgekapitels rund um Strategisches Domain-Driven Design vertiegen.

4.3 Phase Discover: Big Picture EventStorming

Nachdem Ziele und Wertversprechen geklärt sind, geht es in der Phase **Discover** darum, die Domain im Detail zu verstehen. Eine der wirkungsvollsten Methoden dafür ist das **Big Picture EventStorming**.

4.3.1 Beschreibung

EventStorming basiert auf der Idee, dass sich Fachlichkeit am besten durch **Domain Events** beschreiben lässt – Dinge, die in der Realität geschehen und für das Geschäft relevant sind („Customer placed an order"). Auf einem langen Papier- oder digitalen Whiteboard werden diese Events als Post-its chronologisch und thematisch angeordnet. Durch diesen visuellen, kollaborativen Ansatz entsteht in kürzester Zeit ein geteiltes Bild der Domain.

4.3.2 Teilnehmerkreis

Ein Big Picture EventStorming entfaltet seinen Nutzen nur, wenn **alle relevanten Perspektiven** vertreten sind:

- Domain Experts, die das tägliche Geschäft kennen.
- Developers und Architects, die technische Machbarkeit und Integration bedenken.
- Product Owner, Business Analysts und Agile Coaches, die für Priorisierung, Wertschöpfung und Moderation sorgen.
- Vertreter:innen angrenzender Abteilungen oder externer Partner, wenn ihre Systeme oder Prozesse direkt involviert sind.

Ziel ist es, ein möglichst vollständiges Bild der Value Streams zu gewinnen – Silodenken wird so überwunden.

4.3.3 Elemente

Im Big Picture EventStorming werden wenige zentrale **Elementtypen** verwendet, die durch Farben und Symbole unterschieden werden:

Element	Farbe	Beschreibung
Domain Events	orange	fachlich relevante Events, die den Zustand verändern
Commands	blau	Aktionen, die ein Event auslösen
Actors / Roles	gelb	Personen oder Systeme, die Commands ausführen
External Systems	pink	Systeme oder Organisationen außerhalb des eigenen Einflussbereichs
Hot Spots	rot	Unsicherheiten, Widersprüche oder Konflikte, die untersucht werden

4.3.4 Phasen des Ablaufs

Ein Big Picture EventStorming folgt typischerweise mehreren Phasen:

1. **Chaotic Exploration** – Alle Beteiligten notieren Domain Events auf Sticky Notes. Es geht um Quantität, nicht um Ordnung. Ziel: implizites Wissen auf den Tisch bringen.

2. **Enforcing the Timeline** – Die Events werden in eine zeitliche Abfolge gebracht. Hier entstehen erste Diskussionen über Reihenfolgen, Abhängigkeiten und Prozesslogik.

3. **Adding Structure** – Pivotal Events werden markiert, Swimlanes eingeführt, External Systems ergänzt. Das Modell gewinnt Struktur und Tiefe. Eine besondere Rolle im Big Picture EventStorming spielen **Pivotal Events**. Sie markieren besonders wichtige Domain Events in der Timeline, an denen sich der Zustand der Domain grundlegend verändert.
4. **Identifying Hot Spots** – Offene Fragen und Konflikte werden sichtbar gemacht. Diese Punkte sind besonders wertvoll, weil sie Risiken oder Innovationschancen markieren.

5. **Refinement & Exploration** – Je nach Zielsetzung können weitere Elemente wie Commands oder User Roles ergänzt werden. Damit entwickelt sich das Modell hin zu einer Grundlage für Bounded Contexts und Architekturdiskussionen.

4.3.5 Facilitation-Tipps

Als Moderator:in solltest du zunächst für einen großen, physischen oder digitalen Raum sorgen – Platzmangel ist der größte Feind. Gleich zu Beginn gilt es zu erklären, dass es nicht um „richtig oder falsch" geht, sondern um Exploration. Achte darauf, dass wirklich alle Beteiligten aktiv beitragen, indem du auch stillere Rollen gezielt einbindest. Farben helfen dir, die verschiedenen Elemente klar unterscheidbar zu machen. Schließlich ist es wichtig, die Geschwindigkeit hochzuhalten: Details können später immer noch verfeinert werden, entscheidend ist das gemeinsame Momentum im Workshop.

4.3.6 Nutzen

EventStorming eignet sich in der Discover-Phase besonders, weil es **schnell Tiefe bringt** und **implizites Wissen sichtbar macht**. Teams erkennen Prozessbrüche, Widersprüche und Potenziale für Verbesserungen. Gleichzeitig entsteht ein Rohmaterial, aus dem später Bounded Contexts, Ubiquitous Language und technische Modelle entwickelt werden können.

4.4 Phase Discover: Domain Storytelling

Nachdem durch Big Picture EventStorming ein breites Bild der Domain sichtbar geworden ist, erlaubt **Domain Storytelling** ein tieferes Eintauchen in konkrete Abläufe. Die Methode rückt **Akteure und ihre Interaktionen** in den Mittelpunkt und macht implizite Arbeitsweisen durch erzählte Geschichten sichtbar. Für viele Teams ist das besonders wertvoll, da Geschichten leichter zugänglich sind als abstrakte Modelle. Wer miterlebt, wie ein Domain Expert Schritt für Schritt eine Aufgabe beschreibt, bekommt sofort ein Gefühl dafür, wie die Arbeit in der Realität funktioniert.

4.4.1 Beschreibung

Beim Domain Storytelling erzählen Domain Experts Geschichten aus ihrem Arbeitsalltag. Moderator:innen zeichnen diese Geschichten live mit einfachen Symbolen: Akteure werden als Piktogramme dargestellt, ihre Aktivitäten als Pfeile mit kurzen Beschreibungen. Das Ergebnis ist eine visuelle Narration, die zeigt, wie Personen und Systeme zusammenarbeiten, um bestimmte Aufgaben zu erledigen. Die Methode ist leicht verständlich, weil sie die natürliche Art aufgreift, wie Menschen Wissen weitergeben – durch Geschichten. Anstatt trockene Prozessdokumentationen zu lesen, sehen die Beteiligten sofort, wie eine typische Situation abläuft und welche Rollen, Work Objects und Aktivitäten dabei eine Rolle spielen.

4.4.2 Teilnehmerkreis

Für ein gelungenes Domain Storytelling braucht es die richtigen Menschen im Raum. Unverzichtbar sind **Domain Experts**, die authentische Geschichten aus der Praxis einbringen können. Dazu kommen **Developers und Architects**, die beim Zuhören technische Fragen stellen und so implizite Annahmen sichtbar machen. Auch **Product Owner, Business Analysts und Agile Coaches** sind wichtige Teilnehmer:innen, weil sie die Brücke zur Produktvision und Wertschöpfung schlagen und die Moderation unterstützen können. Oft genügt es, mit einem kleinen, fokussierten Kreis zu starten, denn mehrere Storytelling-Sessions mit unterschiedlichen Beteiligten lassen sich später leicht kombinieren. Wichtig ist, dass diejenigen im Raum sind, die wirklich wissen, wie die Arbeit im Alltag aussieht.

4.4.3 Elemente

Die wichtigsten Bausteine im Domain Storytelling sind einfach gehalten:
- **Actors** repräsentieren Personen oder Rollen, die handeln.
- **Work Objects** sind Dinge, die im Prozess bearbeitet oder bewegt werden, wie eine Bestellung oder ein Dokument.
- **Activities** beschreiben Handlungen, die von den Actors durchgeführt werden.
- Mehrere Activities ergeben eine **Story**, also eine Sequenz, die ein Szenario aus der Realität abbildet.

Mit diesen wenigen Elementen lassen sich auch komplexe Abläufe leicht nachvollziehbar darstellen.

4.4.4 Ablaufphasen

Ein Domain Storytelling Workshop beginnt in der Regel damit, dass die Domain Experts eine **Story erzählen**, also einen typischen Ablauf aus ihrem Arbeitsalltag schildern. Währenddessen **visualisieren** die Moderator:innen die Actors, Work Objects und Activities auf einem Whiteboard oder in einem digitalen Tool. Sobald eine erste Version steht, folgt eine kurze **Reflexion**: Die Beteiligten prüfen, ob die Story korrekt wiedergegeben ist. Danach werden **Varianten und Sonderfälle** ergänzt, damit auch Ausnahmen und alternative Abläufe sichtbar werden. Mit der Zeit entstehen mehrere Stories, die zusammen ein umfassendes Bild der Domain ergeben. Dieser iterative Aufbau macht es leicht, klein anzufangen und das Modell Schritt für Schritt zu erweitern.

4.4.5 Facilitation-Tipps

Als Moderator:in solltest du die Stories in der Sprache der Domain Experts aufzeichnen, ohne Begriffe zu übersetzen oder zu abstrahieren – hier beginnt die Ubiquitous Language. Nutze einfache Symbole und halte das Tempo hoch, damit der Fluss der Erzählung nicht unterbrochen wird. Ermutige die Beteiligten, konkrete Beispiele zu nennen, denn diese machen die Story greifbarer. Achte darauf, dass verschiedene Perspektiven zu Wort kommen, indem du gezielt nachfragst, wie andere Rollen den Prozess erleben. Wenn mehrere Stories gesammelt sind, lohnt es sich, Gemeinsamkeiten und Unterschiede hervorzuheben, um Muster und wiederkehrende Strukturen sichtbar zu machen.

4.4.6 Nutzen

Domain Storytelling eignet sich in der Discover-Phase, um **konkrete Arbeitsprozesse** transparent zu machen und ein gemeinsames Verständnis aufzubauen. Es unterstützt dabei, implizite Regeln, Rollen und Abhängigkeiten sichtbar zu machen. Dadurch lassen sich auch Missverständnisse aufdecken, die im Alltag oft

unter der Oberfläche bleiben. Für Teams bietet die Methode eine hervorragende Grundlage, um Anforderungen klarer zu formulieren, Bounded Contexts zu definieren und die Ubiquitous Language zu entwickeln. Wer eine Session miterlebt hat, versteht nicht nur die Theorie, sondern hat ein greifbares Bild davon, wie die Domain tatsächlich funktioniert – und genau das ist der entscheidende Mehrwert dieser Methode.

4.5 Fazit

Collaborative Modelling macht die im Fundament beschriebenen Prinzipien – enge Zusammenarbeit, gemeinsame Sprache, kontinuierliches Lernen – in der Praxis erlebbar. Ob durch Business Model Canvas, Value Proposition Canvas, EventStorming oder Domain Storytelling: In allen Fällen entsteht ein Raum, in dem Business und Technik **gemeinsam denken**.

Für die Zielgruppen dieses Primers – von Software-Entwickler:innen über Agile Coaches bis hin zu Product Ownern und Business Analysts – bietet Collaborative Modelling konkrete Werkzeuge, um Brücken zu bauen. Am Ende geht es nicht nur um Modelle, sondern um **Alignment, Verständlichkeit und tragfähige Entscheidungen**.

5 Strategisches Domain-Driven Design

Strategisches Domain-Driven Design (DDD) beschäftigt sich mit den großen Linien: Wie lassen sich komplexe fachliche Domänen so strukturieren, dass daraus tragfähige technische Systeme und zugleich passende organisatorische Strukturen entstehen? Während das taktische DDD die Detailarbeit am Modell leistet, liefert das strategische DDD den Bezugsrahmen, in dem diese Detailarbeit überhaupt Sinn ergibt. Es geht darum, Klarheit zu schaffen: Welche Teile einer Domäne sind wirklich von strategischer Bedeutung? Wo lohnt sich Investition in Qualität und Eigenentwicklung? Und wie verhindern wir, dass Architektur oder Organisation an den tatsächlichen Bedürfnissen der Fachlichkeit vorbeigehen?

5.1 Problemraum und Lösungsraum

Ein wichtiges Fundament des strategischen DDD ist die Trennung von Problem- und Lösungsraum. Im Problemraum stellen wir die fachliche Frage: *Was soll eigentlich erreicht werden?* Hier bewegen wir uns vollständig auf der Ebene von Bedarfen von Nutzer:innen, Geschäftsmodellen und Rahmenbedingungen. Technologie spielt an dieser Stelle keine Rolle. Erst im Lösungsraum geht es um das *Wie*: Welche Architekturen, Systeme und Modelle setzen die fachlichen Zwecke um?

Domains und Subdomains gehören ausschließlich in den Problemraum. Sie beschreiben, was eine Organisation tut, worin ihre Expertise liegt und wie sie Wert schafft. Alle weiteren Konzepte des DDD – Bounded Contexts, Aggregates, Entities, Value Objects, Context Maps oder die Ubiquitous Language – liegen im Lösungsraum. Sie beschreiben, wie die Zwecke der Subdomains fachlich und technisch umgesetzt werden.

Diese Unterscheidung verhindert vorschnelle Technologieentscheidungen und schafft einen frühen Fokus auf die Fachlichkeit. Ein Praxisbeispiel: Wer eine Subdomain *Praxismanagement* identifiziert, beschreibt damit den Zweck, Abläufe in einer Arztpraxis zu strukturieren und Patienteninformationen zu verwalten.

Ob dies später als App, Webanwendung oder als Bestandteil eines größeren Systems umgesetzt wird, ist zunächst irrelevant. Erst wenn der fachliche Zweck verstanden ist, lohnt es sich, in den Lösungsraum zu wechseln.

5.2 Domains und Subdomains

Domains sind die Tätigkeits- und Kompetenzbereiche einer Organisation. Sie sind das, was sich ein Unternehmen „auf die Fahnen schreibt": der Anspruch an den Markt, verbunden mit einem klaren Zweck, spezifischem Fachwissen und bestimmten Arbeitsweisen. Eine Domain markiert den Bereich, in dem eine Organisation über Know-how verfügt und in dem sie bestimmte Abläufe sowie Arbeitsweisen etabliert hat.

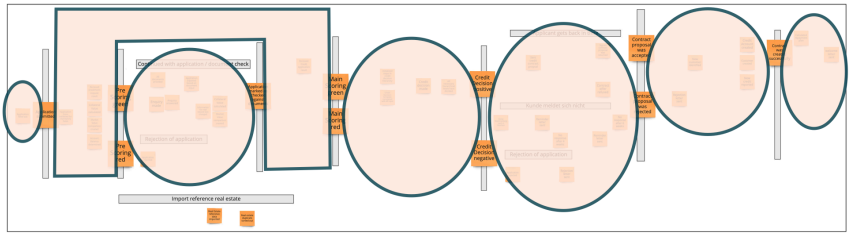
In großen Unternehmen existieren oft mehrere solcher Domains nebeneinander. Ein Automobilkonzern bewegt sich beispielsweise in der Domain *Fahrzeugentwicklung*, gleichzeitig aber auch in der Domain *Finanzdienstleistungen*. Beide Domains unterscheiden sich in Anspruch, Fachwissen und Arbeitsweisen grundlegend, auch wenn sie an vielen Stellen ineinandergreifen. Gleiches gilt beispielsweise auch für das regulatorische Umfeld. Die Domain „Finanzdienstleistungen" ist BaFIN reguliert während beider *Fahrzeugentwicklung* Rahmenbedingungen der StVO oder der Arbeitssicherheit eine Rolle spielen.

Innerhalb einer Domain werden Subdomains geschnitten, die jeweils einen klaren Zweck erfüllen. Das Schneiden von Subdomains ist eine anspruchsvolle Aufgabe, weil es nicht um technische Abgrenzungen geht, sondern um das Herausarbeiten kohärenter fachlicher Bereiche. Eine Subdomain sollte sich immer auf eine präzise formulierte Aufgabe konzentrieren. Eine Faustregel lautet: Wenn man den Zweck einer Subdomain nicht in einem einzigen Satz ohne viele „und" oder „oder" beschreiben kann, ist sie zu breit geschnitten und innen wahrscheinlich nicht kohäsiv.

Hilfreich ist dabei, auf Prozesse zu schauen: **Pivotal Events**, also Schlüsselmomente wie *Antrag eingereicht* oder *Dokument geprüft*, markieren häufig Übergänge zwischen Subdomains.

Pivotal Events

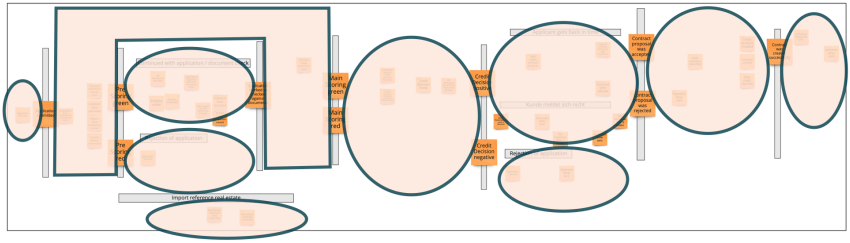
Heuristik: Ein Pivotal Event liegt wahrscheinlich an der Grenze einer Subdomain



Auch **Swimlanes**, die verschiedene Bahnen oder Abzweigungen im Prozess sichtbar machen, helfen bei der Abgrenzung. Sie sind häufig ein Indikator, dass an diesen Stellen etwas unterschiedliches passiert.

Swimlanes

Heuristic: Swimlanes sind Indikatoren für eine Zerteilung eines Subdomain Kandidaten

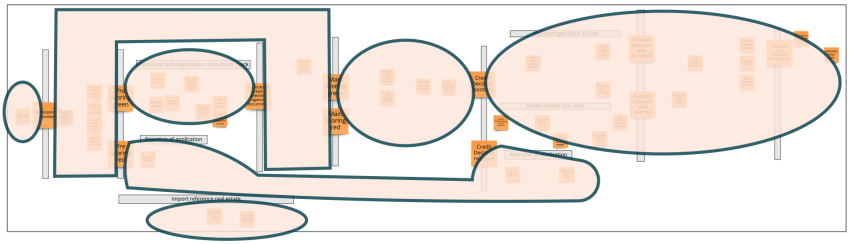


Dieses Unterschiedliche ist wichtig für das Thema Kohäsion. Wir wollen in einer Subdomain einen hohen grad an funktionaler Kohäsion erreichen. Daher sollte

man immer sehr kritisch darauf schauen, ob Funktionalitäten, die in einer Subdomain gebündelt werden sollen, wirklich eine sehr hohe Zusammengehörigkeit auf funktionaler Ebene haben.

Kohäsion

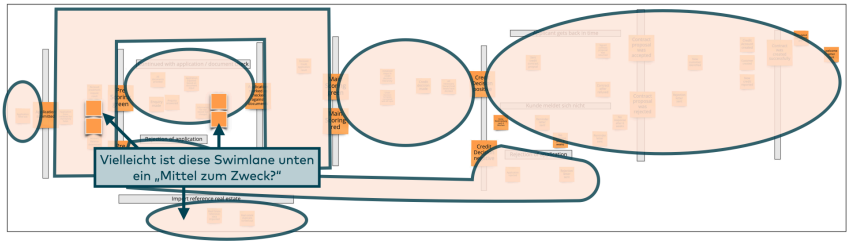
Heuristic: Domain Events in einer Subdomain weisen einen hohen Grad an funktionaler Kohäsion auf



Ein Mittel um den Grad an funktionaler Kohäsion auf den Prüfstand zu stellen ist, den fachlichen Zweck eier Subdomain auszuformulieren. Je weniger „und“ „oder“ „sowie“ in diesem Text auftauchen desto höher ist der Grad an funktionaler Kohäsion. Eine sehr lesenswerte Publikation zu diesem Thema ist das „Structured Design“ Paper von Stevens, Meyers und Constantine. Wir werden das Thema „fachlicher Zweck“ auch noch einmal im Bezug auf Bounded Contexts aufgreifen.

Purpose - Zweck

Heuristic: Jede Subdomain hat einen klar beschreibbaren Zweck



Subdomains stehen in enger Korrelation zueinander. Sie bilden gemeinsam den Problemraum ab, sind also inhaltlich miteinander verbunden, ohne jedoch ihre Eigenständigkeit zu verlieren. Ziel ist, dass diese Korrelation schlank und klar bleibt. Lose Kopplung entsteht nicht erst im Code, sondern schon auf dieser Ebene: Je kohärenter Subdomains geschnitten sind, desto weniger Abhängigkeiten benötigen sie untereinander.

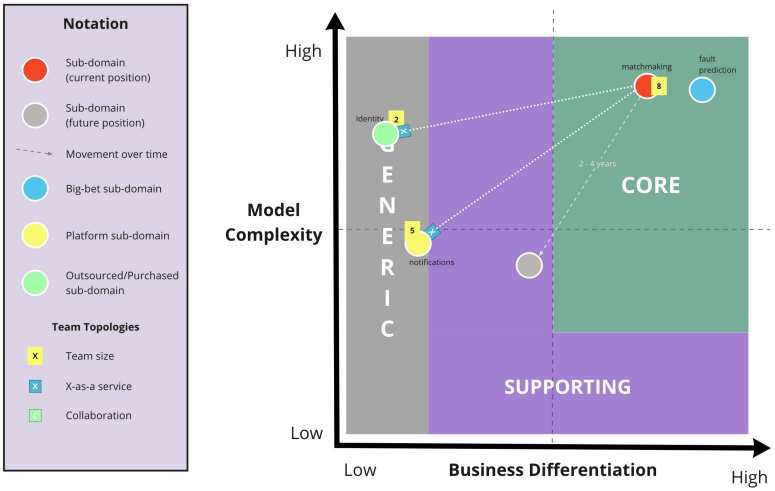
Im Domain-Driven Design Starter Modeling Process findet das Zerlegen einer Domain in Subdomains in der Phase „Decompose“ statt.

5.3 Strategische Klassifikation

Nicht alle Subdomains sind gleich wichtig. Strategisches DDD unterscheidet zwischen Core-, Supporting- und Generic-Subdomains. Core-Domains sind jene Bereiche, in denen eine Organisation sich vom Wettbewerb differenziert. Hier liegt die höchste strategische Bedeutung, weshalb keine Kompromisse bei Qualität und Ownership gemacht werden sollten. Supporting-Subdomains sind wichtig, aber nicht differenzierend. Hier sind Kompromisse, etwa in Form von Customizing, möglich. Generic-Subdomains schließlich stellen Commodity-Funktionalität be-

reit, die besser durch Standardsoftware abgedeckt wird als durch Eigenentwicklung.

Diese Unterscheidung ist kein Selbstzweck, sondern steuert Investitionsentscheidungen. Core-Domains gehören ins Zentrum der eigenen Entwicklung, während Generic-Subdomains oft kostengünstiger mit Produkten von der Stange abgebildet werden können. Ein Modeunternehmen, das seinen E-Commerce zunächst als Commodity betrachtete, später professionalisierte und schließlich als Core-Domain insourcing-basiert entwickelte, konnte dadurch innerhalb weniger Wochen neue Vertriebsmodelle einführen und agil auf Marktveränderungen reagieren.



(Bild Quelle: https://github.com/ddd-crew/core-domain-charts)

Im Domain-Driven Design Starter Modeling Process findet die strategische Klassifikation von Subdomains in der Phase „Strategize“ statt.

5.4 Bounded Contexts

Im Lösungsraum werden Subdomains durch Bounded Contexts umgesetzt. Ein Bounded Context definiert eine Grenze, innerhalb derer ein Modell in konsistenter Sprache existiert, welches auf einen bestimmten Zweck (den der Subdomain) zugeschnitten ist. Damit ist er die Brücke zwischen Fachlichkeit und Technik. Entscheidend ist dabei das Prinzip **Modellspezialisierung vor Generalisierung**.

Viele Organisationen haben die leidvolle Erfahrung gemacht, ein universelles Modell erzwingen zu wollen, das alle Eventualitäten abdeckt. Heraus kommt ein überladenes System ohne Fokus: etwa ein „Geschäftspartner"-Service, der in einer Versicherung gleichermaßen Endkunden, Werkstätten und Therapeut:innen abbilden soll und dadurch hunderte Attribute aufnimmt. Das Resultat ist ein meist Daten-getriebenes Modell, das niemand mehr versteht und das keine Aufgabe wirklich gut erfüllt.

DDD empfiehlt stattdessen spezialisierte Modelle, die klar auf ihren Zweck zugeschnitten sind. Ein Teilnehmer an einer Konferenz ist im Kontext *Check-in* kein „Customer", sondern erhält einen *Badge*. Im Kontext *Ticketverkauf* steht derselbe Mensch für eine finanzielle Transaktion, im Kontext *Catering* für eine Hochrechnung des Essensbedarfs. Jedes Modell ist spezifisch für seinen Zweck – und gerade dadurch wertvoll.

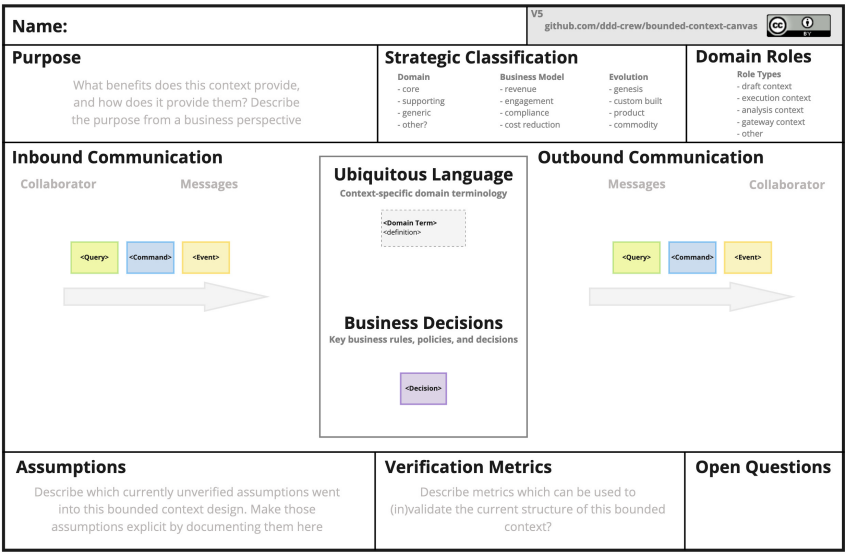
Werkzeuge wie die **Bounded Context Design Canvas** unterstützen diesen Übergang vom Problem- in den Lösungsraum. Sie helfen, für jede Subdomain die relevanten Begriffe, Regeln und Kommunikationsflüsse zu erfassen und eine konsistente Sprache zu entwickeln. In der Praxis entstehen daraus oft „lebende Dokumentationen", die sowohl Architekt:innen als auch Fachexpert:innen Orientierung bieten.

5.5 Arbeiten mit der Bounded Context Design Canvas

Die **Bounded Context Design Canvas** ist eines der zentralen Werkzeuge im strategischen Domain-Driven Design. Sie hilft, den Übergang vom Problemraum in den Lösungsraum strukturiert und nachvollziehbar zu gestalten. Während viele

DDD-Workshops mit einer großen Menge an Erkenntnissen über Domains und Subdomains enden, stellt die Canvas sicher, dass dieses Wissen nicht einfach verpufft, sondern in ein greifbares, lebendiges Artefakt überführt wird.

Im Kern dient sie dazu, für jede identifizierte Subdomain ein klar umrissenes, gemeinsam verstandenes Modell zu entwickeln – mit einer eigenen Sprache, eigenen Regeln und klaren Kommunikationsgrenzen. Jede Canvas steht dabei für einen möglichen oder tatsächlichen **Bounded Context**.



(Bild Quelle: https://github.com/ddd-crew/bounded-context-canvas)

5.5.1 Vom Problemraum in die Canvas – die Decompose-Phase

Nach der **Decompose-Phase** des *DDD Starter Modelling Processes* liegt das Augenmerk darauf, die zuvor identifizierten **Subdomains** konkret auszuarbeiten. Hierbei werden die grundlegenden Eckpfeiler auf die Canvas übertragen: der **Name**, der **Purpose** (also der fachliche Zweck) und die ersten Ideen für die

Ubiquitous Language – also die Begriffe, mit denen die beteiligten Personen über diesen Teil der Domäne sprechen.

Bereits in diesem frühen Stadium lohnt es sich, erste **fachliche Regeln** (**Business Rules**) zu notieren. Sie sind Ausdruck des spezifischen Verhaltens, das in der Subdomain erwartet wird, und geben dem entstehenden Modell Struktur. Gerade hier zeigt sich der Wert kollaborativer Modellierung: Fachleute und Entwickler:innen sitzen gemeinsam vor der Canvas und gleichen ihre Sprache ab. Aus „Kunde" wird vielleicht „Teilnehmer", aus „Ticket" ein „Badge", und plötzlich entsteht Klarheit über Bedeutungen, die zuvor implizit und damit potenziell missverständlich waren.

Für jede Subdomain wird eine eigene Canvas angelegt. Diese Trennung ist wichtig, weil sie Fokus und Kohärenz sicherstellt. Eine gemeinsame Canvas für mehrere Subdomains führt fast immer dazu, dass Grenzen verwischen und Modelle unkontrolliert zusammenwachsen. In dieser frühen Phase geht es nicht um Präzision, sondern um Orientierung. Ziel ist ein Artefakt, das Diskussionen anregt und als Kommunikationsanker dient.

5.5.2 Kommunikation sichtbar machen – die Connect-Phase

Nachdem Subdomains identifiziert und ihre Zwecke beschrieben sind, folgt die **Connect-Phase**. In dieser Phase werden die Beziehungen zwischen den Subdomains betrachtet. Die Canvas hilft hier, die **Inbound- und Outbound-Kommunikation** systematisch zu erfassen.

Inbound-Kommunikation beschreibt, welche Ereignisse, Befehle oder Anfragen eine Subdomain aus ihrer Umgebung empfängt. Outbound-Kommunikation hingegen zeigt, welche Signale, Nachrichten oder Ergebnisse sie selbst aussendet. Diese Beziehungen werden auf der Canvas festgehalten, um die Interaktion zwischen den Subdomains transparent zu machen.

Oft werden hier **Domain Events** sichtbar, die das Rückgrat einer losen Kopplung bilden. Anstatt dass eine Subdomain direkt eine andere aufruft, veröffentlicht sie ein Ereignis – etwa „Termin bestätigt" oder „Rechnung erstellt" –, auf das andere

Subdomains reagieren können. Diese Denkweise verschiebt die Perspektive von synchroner Kontrolle hin zu asynchroner Zusammenarbeit.

Die Canvas dient in dieser Phase nicht nur der Dokumentation, sondern auch als Werkzeug zur Reflexion: Stimmen die Kommunikationsbeziehungen mit dem intendierten Zweck der Subdomain überein? Wird sie durch zu viele Abhängigkeiten in ihrer Autonomie eingeschränkt? Wenn solche Fragen sichtbar werden, ist die Canvas kein statisches Formular, sondern ein Spiegel des gemeinsamen Verständnisses.

5.5.3 Strategische Entscheidungen treffen – die Strategize-Phase

In der **Strategize-Phase** geht es darum, die Subdomains strategisch einzuordnen. Die Bounded Context Design Canvas bietet hierfür das Feld der **Strategic Classification**, in dem festgehalten wird, ob eine Subdomain zu den **Core**, **Supporting** oder **Generic Domains** gehört.

Diese Einordnung ist mehr als ein Etikett. Sie beeinflusst, wie viel Energie, Budget und Aufmerksamkeit eine Subdomain in der weiteren Entwicklung erhält. Core-Subdomains – also jene, über die sich die Organisation differenziert – verlangen nach intensiver Pflege, hoher Qualität und meist nach interner Entwicklungskompetenz. Generic-Subdomains hingegen sollten bewusst vereinfacht und, wo möglich, durch Standardlösungen abgedeckt werden.

Durch das sichtbare Festhalten dieser Klassifikation auf der Canvas wird die Diskussion über Prioritäten greifbar. Das Team kann gemeinsam entscheiden, wo es sich lohnt, tief in die Modellierung zu investieren, und wo „gut genug" wirklich gut genug ist.

5.5.4 Sprache und Regeln schärfen – ein kontinuierlicher Prozess

Während aller Phasen wird immer wieder an der **Ubiquitous Language** und an den **Business Rules** geschärft. Diese beiden Felder bilden das Herzstück jeder

Canvas. Sprache und Regeln sind das Fundament, auf dem Verständigung und Modellkonsistenz aufbauen.

Neue Begriffe werden eingetragen, alte hinterfragt, Definitionen verfeinert. Gleichzeitig werden fachliche Regeln konkretisiert oder ergänzt. So entsteht über die Zeit ein geteiltes Verständnis, das sowohl für die Fachdomäne als auch für die technische Umsetzung tragfähig ist.

Besonders wertvoll ist dieser Prozess dann, wenn verschiedene Disziplinen beteiligt sind: Produktmanagement, Entwicklung, UX, vielleicht sogar Support oder Vertrieb. Die Canvas wirkt hier wie ein Katalysator – sie zwingt alle Beteiligten, implizite Annahmen explizit zu machen.

5.5.5 Arbeiten mit Unsicherheit – offene Fragen und Hypothesen

Kein Modell entsteht im luftleeren Raum. Oft bleiben Fragen offen oder Annahmen unbestätigt. Die Canvas bietet dafür eigene Felder: **Open Questions**, **Assumptions** und **Verification Metrics**.

In *Open Questions* werden Unklarheiten dokumentiert – fachlich, organisatorisch oder technisch. *Assumptions* halten Hypothesen fest, etwa „Die Terminplanung ist immer an eine Ärztin gekoppelt". Diese Annahmen bilden die Grundlage für gezieltes Lernen. *Verification Metrics* schließlich definieren, woran sich später überprüfen lässt, ob eine Annahme zutraf. So wird Lernen systematisch in den Modellierungsprozess integriert, anstatt nur nebenbei zu passieren.

Diese drei Felder fördern eine reflektierte Haltung im DDD-Kontext: Es geht nicht darum, von Anfang an alles zu wissen, sondern darum, bewusst mit Unsicherheit umzugehen.

5.5.6 Vom Canvas zur Umsetzung – Verfeinerung im Lösungsraum

Wenn die Arbeit im Problemraum gereift ist, beginnt die Umsetzung im Lösungsraum. Hier zeigt sich ein weiteres Merkmal der Canvas: Sie ist kein einmaliges Ergebnis, sondern ein lebendes Artefakt.

Im Lösungsraum werden die Inhalte der Canvas kontinuierlich verfeinert. Begriffe werden präziser, Regeln klarer, Kommunikationsflüsse genauer modelliert. Gleichzeitig kann es vorkommen, dass eine Subdomain im Laufe der Arbeit in mehrere **Bounded Contexts** aufgeteilt wird – jede mit einer eigenen Canvas. Dieser Split ist kein Fehler, sondern Ausdruck wachsenden Verständnisses. Was anfangs als eine Einheit erschien, entpuppt sich später als mehrere klarer definierte Kontexte mit jeweils eigener Sprache und Verantwortung.

Gerade in diesem kontinuierlichen Prozess liegt die Stärke der Bounded Context Design Canvas: Sie verbindet die explorative Offenheit des Problemraums mit der strukturierten Präzision des Lösungsraums. Sie ist Denkwerkzeug, Kommunikationsmittel und Dokumentation zugleich – und damit ein zentrales Bindeglied im strategischen DDD.

5.6 Soziotechnisches Alignment

Strategisches DDD endet nicht bei Bounded Contexts, sondern reicht in die Organisation hinein. Mit **Context Maps** lassen sich Beziehungen zwischen Teams und Bounded Contexts visualisieren, die zugleich technische und organisatorische Abhängigkeiten abbilden. Hier wird deutlich, dass klare Kontextgrenzen nicht nur die Architektur stabiler machen, sondern auch die Grundlage für autonome Teams bilden.

Konzepte aus den **Team Topologies** verstärken diesen Ansatz: Teams sollten entlang von Bounded Contexts geschnitten werden, sodass Ownership, Autonomie und minimale Koordination möglich sind. Wer Subdomains und Kontexte sauber schneidet, schafft damit nicht nur eine robuste Softwarearchitektur, sondern legt auch die Basis für eine Organisation, die Flow ermöglicht und Wertströme optimal unterstützt.

5.6.1 Context Maps im Strategischen DDD

Context Maps sind das Werkzeug, um Beziehungen zwischen Teams und ihren Bounded Contexts sichtbar zu machen – fachlich, technisch und organisatorisch. Während Bounded Contexts die Grenzen innerhalb des Lösungssystems beschreiben, zeigt die Context Map, wie Teams zueinander in Beziehung stehen, wer von wem abhängig ist und welche Art von Zusammenarbeit oder Kopplung besteht.

Sie macht damit implizite Dynamiken explizit: Machtverhältnisse zwischen Teams, die Ausbreitung von Modellen und Governance-Strukturen. Ein sauber modellierter Code hilft wenig, wenn Teams in Abhängigkeiten gefangen sind oder Modelle unkontrolliert durch die Landschaft propagieren. Die Context Map bringt Ordnung in dieses Geflecht.

Team-Beziehungen im Context Maps

Context Maps betrachten nicht nur technische Schnittstellen, sondern vor allem soziale und organisatorische Beziehungen. Drei Grundtypen bilden das Fundament:

Upstream und Downstream: Das Upstream-Team gestaltet das Modell und prägt die Sprache. Das Downstream-Team konsumiert diese Ergebnisse und ist von Stabilität und Qualität des Upstream abhängig. Wie in einem Flusssystem liegt die Kontrolle „flussaufwärts“. Das Verständnis dieser Richtung ist zentral für jede Context Map.

Free: Teams agieren unabhängig voneinander. Es gibt keine direkte Kopplung, weder technisch noch organisatorisch. Diese Freiheit ist wertvoll, wenn die Zusammenarbeit nur lose oder temporär erforderlich ist, birgt aber das Risiko inkonsistenter Modelle.

Mutually Dependent: Zwei Teams hängen gegenseitig voneinander ab. Entscheidungen müssen abgestimmt werden, Änderungen erfordern Synchronisation. Solche Beziehungen sind besonders sensibel, weil sie leicht zu Koordinationsengpässen führen. Sie erfordern explizite Kommunikations- und Release-Mechanismen.

Diese drei Beziehungstypen bestimmen, wie eng Teams zusammenarbeiten, wie stark sie voneinander abhängig sind und welche Integrationsmuster sinnvoll sind. Die Context Map dient hier als Spiegel der realen Organisationsdynamik – nicht als Wunschbild.

Die Patterns der Context Map

1. Open-Host Service (**OHS**) Ein Upstream-Context bietet eine stabile, dokumentierte Schnittstelle für viele Konsumenten an – eine Art öffentliches Gateway. Technisch kann das eine API, ein Event-Stream oder ein Message-Format sein. Es ist neutral und erweiterbar, ideal für viele Downstream-Systeme.

2. Anticorruption Layer (**ACL**) Das Downstream-System schützt sich durch eine Übersetzungsschicht, die das externe Modell in ein internes überführt. Diese Schicht isoliert Altlasten und macht evolutionäres Design möglich. Das ACL ist das wichtigste Entkopplungsmuster für Integrationen.

3. Conformist (**CF**) Das Downstream-Team übernimmt das Modell des Upstream-Systems unverändert – aus Zwang, Bequemlichkeit oder Überzeugung. Ein Conformist vermeidet Übersetzungslogik, verliert aber Autonomie. In gewachsenen Systemlandschaften oft ein Symptom enger Kopplung.

4. Shared Kernel (**SK**) Zwei Kontexte teilen einen Teil des Modells oder sogar Artefakte (z. B. eine Library oder Datenbank). Das spart Integrationsaufwand, schafft aber hohe Kopplung. Akzeptabel, wenn der gemeinsame Teil klein und stabil ist; toxisch, wenn unterschiedliche Teams (oder Lieferanten) daran ziehen.

5. Partnership Teams, die einen Shared Kernel besitzen, sind gegenseitig abhängig und sollten eine Partnerschaft bilden. Sie planen, entwickeln und integrieren gemeinsam. Erfolg oder Scheitern sind wechselseitig. Partnerschaften erfordern Vertrauen, Transparenz und häufig koordinierte Releases.

6. Customer/Supplier Development (**CUS-SUP**) Das Upstream-Team ist Lieferant, das Downstream-Team Kunde. Anforderungen werden gemeinsam abgestimmt, es gibt aber eine Hierarchie. Dieses Muster hilft, Einfluss gezielt zu gestalten: Wer darf was fordern, wer liefert was wann?Abweichungen davon sind

aufschlussreich: „vehemente Kunden“ oder „übervorsichtige Lieferanten“ zeigen Governance-Probleme.

7. Published Language (**PL**) Die gemeinsame Sprache, in der Systeme Informationen austauschen. Sie ist dokumentiert, von mehreren Parteien getragen und entkoppelt interne Modelle von Schnittstellen. In Kombination mit einem OHS bildet sie oft den Standard für Integration.

8. Separate Ways (**SW**) Manchmal ist die beste Integration gar keine. Wenn der Aufwand einer Kopplung den Nutzen übersteigt, ist es besser, Teams unabhängig arbeiten zu lassen oder Prozesse organisatorisch zu lösen. „Separate Ways“ steht für bewusste Trennung – oft temporär, manchmal dauerhaft.

9. Big Ball of Mud (**BBOM**) Bezeichnet chaotische, unstrukturierte Systeme ohne klare Modellgrenzen. Sie sind zu groß, zu diffus und zu riskant für tiefgreifende Eingriffe. Wichtig ist, sie zu isolieren, damit ihr Modell sich nicht weiterverbreitet. Anticorruption Layers sind hier Pflicht, Conformists eine Warnung.

Nutzen der Context Map

Context Maps sind mehr als Architekturdiagramme – sie sind Werkzeuge zur Organisationsdiagnose. Sie helfen, drei zentrale Perspektiven sichtbar zu machen:

Macht und Einfluss: Wer kontrolliert Modelle und Schnittstellen? Wo entstehen Abhängigkeiten?

Modellpropagation: Welche Modelle verbreiten sich unkontrolliert durch Copy-Paste oder Shared Kernels?

Governance: Wie wird über Änderungen entschieden? Gibt es klare Verantwortlichkeiten oder versteckte Veto-Strukturen?

Mit diesen Einsichten können Teams bewusste Entscheidungen treffen: Welche Beziehungen wollen wir pflegen? Wo müssen wir Grenzen ziehen? Wo hilft technisches Entkoppeln – und wo organisatorisches?

Praktische Anwendung und Visualisierung

Context Maps lassen sich auf Whiteboards, in Miro oder als Diagramme modellieren. Typischerweise werden Teams und ihre Bounded Contexts als Kreise dargestellt, Beziehungstypen als Linien mit Pattern-Labels versehen (z. B. Customer/Supplier, ACL, Partnership, SW).

Ein Beispiel: Ein Upstream-Team bietet über einen Open-Host Service (OHS) ein Modell an, das ein Downstream-Team über einen Anticorruption Layer (ACL) integriert. Oder zwei Teams gehen „Separate Ways“ – dokumentiert, akzeptiert, stabil.

Diese grafische Sprache macht auch politische und organisatorische Flüsse sichtbar – und das ist der wahre Wert von Context Maps im strategischen DDD.

Ressourcen und Werkzeuge

Ein hervorragender Einstiegspunkt ist das Open-Source-Projekt Context Mapping von der DDD Crew (https://github.com/ddd-crew/context-mapping). Es bietet eine leichtgewichtige Notation, viele Visualisierungsideen und ein Miro Starter Kit, mit dem sich eigene Maps direkt modellieren lassen. Die dort verfügbaren Beispiele zeigen, wie sich technische und organisatorische Beziehungen gleichermaßen darstellen lassen.

Für Teams, die Context Maps automatisiert oder codebasiert pflegen möchten, bietet sich außerdem das Tool Context Mapper (https://contextmapper.org/) an. Es erlaubt, Bounded Contexts und ihre Beziehungen in einer domänenspezifischen Sprache zu beschreiben und daraus Diagramme oder Architekturdokumentation zu generieren. So entsteht eine Brücke zwischen konzeptionellem Design und dokumentierter Architektur.

5.6.2 Beziehungen zwischen DDD und Team Topologies

Domain-Driven Design (DDD) und Team Topologies ergänzen sich auf natürliche Weise. Beide Ansätze beschäftigen sich mit der Frage, wie komplexe Systeme gestaltet und weiterentwickelt werden können – DDD über die Linse der Fachlichkeit, Team Topologies über die Linse der Organisation. Während DDD hilft,

fachliche Grenzen im Softwaremodell zu erkennen, zeigt Team Topologies, wie diese Grenzen in der Teamstruktur und in Kommunikationswegen abgebildet werden sollten.

Gemeinsames Ziel: Soziotechnisches Alignment

Sowohl DDD als auch Team Topologies streben ein soziotechnisches Alignment an – also die enge Kopplung zwischen fachlicher Struktur, technischer Architektur und organisatorischer Aufstellung. In einem gut ausgerichteten System reflektieren die Teamgrenzen die Bounded Contexts, die im strategischen DDD definiert wurden. Das Ziel: Teams besitzen End-to-End-Verantwortung für klar abgegrenzte fachliche Bereiche und können unabhängig voneinander Wert liefern.

DDD beschreibt, was getrennt werden sollte – die fachlichen Domänengrenzen –, während Team Topologies beschreibt, wie diese Trennung organisatorisch und kommunikativ umgesetzt werden kann. Zusammen ermöglichen sie es, architektonische Klarheit und organisatorische Autonomie zu vereinen.

Teams entlang von Bounded Contexts

In einem idealen Setup ist jedes Team einem Bounded Context zugeordnet. Dieser Kontext definiert das fachliche Modell, die Sprache und die Verantwortlichkeiten des Teams. Dadurch entstehen autonome Einheiten, die innerhalb ihres Kontexts schnell und selbstbestimmt agieren können, ohne auf zentrale Abstimmung angewiesen zu sein.

Die vier Teamtypen aus Team Topologies – Stream-aligned, Enabling, Complicated Subsystem und Platform – lassen sich direkt auf DDD-Konzepte abbilden:

Stream-aligned Teams arbeiten entlang eines Wertstroms, der meist einem oder mehreren Bounded Contexts entspricht. Sie besitzen volle Verantwortung für die Features, die aus dieser Domäne entstehen.

Enabling Teams helfen Stream-aligned Teams, neue Praktiken zu übernehmen – beispielsweise kollaborative Modellierung oder Event Storming. Sie fördern die DDD-Adoption in der Organisation und können auch gut bei der täglichen Architektur- und Entwicklungsarbeit unterstützen.

Complicated Subsystem Teams kümmern sich um technisch anspruchsvolle Komponenten, die zwar nicht zentral für die Fachdomäne sind, aber hohe technische Expertise verlangen. Sie arbeiten oft an supporting oder generic Subdomains.

Platform Teams stellen gemeinsame Infrastruktur und Services bereit, die anderen Teams die Arbeit erleichtern, ohne ihre Autonomie einzuschränken.

Diese Zuordnung sorgt dafür, dass die aus DDD abgeleiteten Grenzen nicht nur theoretisch bleiben, sondern organisational wirksam werden.

Kommunikationsströme und Team Interaktion Modes

Team Topologies beschreibt mit den Team Interaction Modes drei grundlegende Arten, wie Teams zusammenarbeiten. Diese Modi helfen, die Kommunikationswege bewusst zu gestalten und Abhängigkeiten gezielt zu steuern, anstatt sie dem Zufall zu überlassen.

Collaboration bezeichnet eine intensive, zeitlich begrenzte Zusammenarbeit zwischen Teams, die einem klaren Zweck dient. Sie findet statt, wenn ein Problem neu oder komplex ist und mehrere Perspektiven benötigt werden, etwa bei der Einführung einer neuen Technologie oder der Entwicklung eines neuen Domänenmodells. Collaboration ist auf Lernen und gemeinsame Exploration ausgerichtet.

X-as-a-Service steht für eine stabile, serviceorientierte Beziehung: Ein Team bietet einen klar umrissenen Service oder eine Plattformfunktion an, die andere Teams konsumieren können. Diese Form der Interaktion reduziert kognitive Last, weil Konsumententeams sich auf ihre eigene Domäne konzentrieren können, während der Serviceanbieter Stabilität und Zuverlässigkeit liefert.

Facilitating beschreibt eine unterstützende Zusammenarbeit, bei der ein Team (meist ein Enabling Team) anderen Teams hilft, neue Fähigkeiten, Praktiken oder Technologien zu übernehmen. Der Fokus liegt nicht auf dem Ergebnis, sondern auf der Befähigung – das Ziel ist, das unterstützte Team zu stärken und anschließend wieder in Autonomie zu entlassen.

Diese drei Interaktionsmodi bilden die Grundlage für bewusste Kommunikationsarchitektur. Sie ermöglichen, dass Teams je nach Situation unterschiedlich stark zusammenarbeiten – temporär, dauerhaft oder beratend. Erst in Kombination mit den Prinzipien des DDD entsteht daraus eine vollständige Sicht: Die Patterns der Context Map beschreiben die fachliche und technische Kopplung, während die Interaction Modes die soziale und organisatorische Ebene abbilden.

Fracture Planes und Strukturierung

Ein zentrales Verbindungselement zwischen DDD und Team Topologies sind die sogenannten Fracture Planes. Sie beschreiben mögliche Trennungslinien innerhalb einer Software- oder Organisationsarchitektur – entlang derer Teams und Systeme sinnvoll voneinander abgegrenzt werden können. Während DDD diese Linien aus fachlicher Perspektive identifiziert, bietet Team Topologies damit eine Sprache für organisatorische Umsetzung.

Fracture Planes können entlang verschiedener Dimensionen verlaufen: fachlich (z. B. Subdomains), nach Änderungsfrequenz, nach regulatorischen Anforderungen, nach Technologie oder sogar nach organisatorischen Verantwortlichkeiten. Besonders wertvoll ist die Kombination mit Bounded Contexts – denn sie liefern eine natürliche Fracture Plane, an der sich Teams orientieren können.

Wo Bounded Contexts entstehen, sollten auch Teamgrenzen verlaufen. So entstehen Teams, die innerhalb ihrer Fracture Plane autonom agieren und ihre Verantwortung verstehen, ohne von anderen Teilen des Systems abhängig zu sein. Dadurch wird das Alignment zwischen Domänenstruktur und Organisation verstärkt.

Grenzen und Dynamik

DDD und Team Topologies teilen auch die Erkenntnis, dass Grenzen nicht statisch sind. Fachliche Domänen verändern sich, Organisationen wachsen, und neue technische Möglichkeiten entstehen. Entsprechend müssen auch Bounded Contexts und Teamzuschnitte regelmäßig überprüft und angepasst werden. Was heute eine Core Domain ist, kann morgen unterstützend oder generisch werden – und umgekehrt.

Der Schlüssel liegt darin, Veränderungen früh zu erkennen und aktiv zu gestalten. Team Topologies bietet dafür Werkzeuge wie Team API und Evolutionary Change, während DDD mit Context Maps und Strategic Classification zeigt, wo Anpassungen fachlich sinnvoll sind. Zusammen ermöglichen sie eine Organisation, die anpassungsfähig bleibt, ohne ihre Kohärenz zu verlieren.

Domain-Driven Design und Team Topologies greifen ineinander. DDD schafft den fachlichen und technischen Bezugsrahmen, Team Topologies sorgt dafür, dass die Organisation diese Struktur leben kann. Wenn beide Perspektiven zusammenspielen, kann eine Delivery-Organisation entstehen, in der Architektur, Fachlichkeit und Wertschöpfung aufeinander abgestimmt sind – und Teams so arbeiten, dass sie langfristig Wirkung und einen schnellen Arbeitsfluss entfalten können.

Der soziotechnische Aspekt wird im DDD Starter Modeling Process in der Phase „Organize“ adressiert.

5.7 Fazit

Strategisches DDD ist mehr als ein Methodenset – es ist ein Denkrahmen für den Umgang mit fachlicher Komplexität. Es zwingt uns, zuerst die fachlichen Zwecke im Problemraum klar zu verstehen, bevor wir technische Entscheidungen im Lösungsraum treffen. Domains sind dabei die Tätigkeits- und Kompetenzbereiche einer Organisation: das, was sie sich auf die Fahnen schreibt, wo sie spezielles Fachwissen und charakteristische Arbeitsabläufe entwickelt hat. Innerhalb dieser Domains werden Subdomains geschnitten, die hohe Kohäsion haben und deren Korrelationen schlank gehalten werden. Bounded Contexts im Lösungsraum sorgen für spezialisierte Modelle, statt in generische Datenhalden abzugleiten. Die Klassifikation in Core-, Supporting- und Generic-Subdomains hilft, Investitionen gezielt zu steuern. Und schließlich verbindet das soziotechnische Alignment die fachliche und technische Struktur mit der Organisation selbst.

Damit wird strategisches DDD zur Brücke zwischen Business, Technik und Organisation – und schafft die Voraussetzung für Systeme, die nicht nur heute funktionieren, sondern auch morgen noch tragfähig sind.

6 Taktisches Domain-Driven Design

Strategisches Domain-Driven Design beschreibt, wie eine Organisation ihre fachliche Landschaft verstehen, aufteilen und strukturieren kann. Es liefert Sprache, Werkzeuge und Muster, um Orientierung zu schaffen – eine gemeinsame Karte der Domäne. Doch diese Karte allein genügt nicht. Erst auf der taktischen Ebene beginnt das eigentliche Handwerk: die konkrete Gestaltung dessen, was innerhalb eines Bounded Context geschieht. Hier wird Fachlichkeit nicht mehr analysiert, sondern gebaut. Taktisches Domain-Driven Design ist die Kunst, aus Konzepten lebendige Strukturen zu formen – Architektur, Modelle und Sprache so miteinander zu verweben, dass sie ein stabiles Ganzes ergeben.

Wer DDD ernst nimmt, erkennt, dass strategische Einsichten erst dann Wirkung entfalten, wenn sie in konkrete technische Formen überführt werden. Grenzen zwischen Kontexten sind nur sinnvoll, wenn das, was sich innerhalb dieser Grenzen befindet, klar, verständlich und konsistent ist. Diese innere Klarheit ist Aufgabe des taktischen Designs. Es ist die Ebene, auf der Teams entscheiden, welche Objekte Verantwortung tragen, wie Konsistenz aufrechterhalten wird, welche Regeln dauerhaft gelten und wie Systeme sich verändern dürfen, ohne auseinanderzufallen.

Taktisches DDD ist damit kein untergeordnetes Detail, sondern die praktische Grammatik der Ubiquitous Language. Es übersetzt Erkenntnis in Struktur, Sprache in Code – und Code wieder in neue Einsichten. Der eigentliche Wert entsteht, wenn diese Rückkopplung bewusst gestaltet wird: Fachleute und Entwickler:innen verstehen dieselben Begriffe, sehen dieselben Zusammenhänge und korrigieren dieselben Missverständnisse.

Wo das strategische DDD Landkarten zeichnet, beschreibt das taktische DDD die Architektur der Orte. Es arbeitet in feinen Schichten: Entitäten, Werte, Aggregate, Ereignisse, Services und Fabriken bilden die Bausteine, mit denen sich Fachlichkeit präzise ausdrücken lässt. Dazu kommen Architekturmuster, die die Domäne in den Mittelpunkt stellen – etwa die hexagonale Architektur oder Event

Sourcing. Und schließlich Methoden wie Design-Level EventStorming, die Fach- und Technikwelt miteinander ins Gespräch bringen.

Taktisches Domain-Driven Design versteht Software nicht als statisches Gebilde, sondern als System in Bewegung. Ein System, das gelernt hat, sich anzupassen. Im Kern geht es nicht um Abstraktion, sondern um Präzision: um die Fähigkeit, Bedeutung so klar zu modellieren, dass sie auch in Code erkennbar bleibt.

6.1 Tactical Patterns

Das Herzstück des taktischen DDD bilden eine Reihe bewährter Bausteine – die sogenannten *tactical patterns*. Sie bilden gemeinsam eine Sprache, mit der man Fachlichkeit ausdrücken und implementieren kann. Diese Patterns sind nicht isoliert zu verstehen, sondern als Elemente eines zusammenhängenden Vokabulars: Entitäten, Wertobjekte, Aggregate, Domain Events, Repositories, Services und Factories.

Sie alle haben dasselbe Ziel: **Kohärenz in der Modellierung**. Sie geben der Software eine Struktur, die das Fachmodell respektiert und es gleichzeitig in den technischen Rahmen der Architektur einbettet.

6.1.1 Entity - Identität und Kontinuität

In einer Fachdomäne gibt es Begriffe, die über die Zeit hinweg bestehen. Sie ändern Zustand, aber nicht Identität. Genau das ist die Rolle der Entity. Eine Entity verkörpert die Kontinuität eines fachlichen Objekts. Sie hat einen Namen und eine Bedeutung innerhalb der Ubiquitous Language, und sie definiert sich nicht durch ihre Daten, sondern durch ihre Existenz.

In klassischer Objektorientierung war eine Entity meist ein Objekt mit einer ID. Im DDD-Kontext ist sie mehr: Sie ist eine Einheit von Verhalten und Zustand, eine Trägerin von Regeln. Ihre Methoden beschreiben fachliche Absichten. Wenn eine Entity eine Operation durchführt, ist dies immer ein Teil eines Prozesses – nicht eine bloße Mutation.

Eine Entity ist auch ein Ort der Verantwortung. Sie trägt Invarianten in sich, die nicht verletzt werden dürfen. Sie kennt die Bedingungen, unter denen sie gültig ist, und sie bewacht ihre Integrität. In einem gut gestalteten Modell sind Entities selbstprüfend: Sie können nicht in einen ungültigen Zustand versetzt werden, ohne dass eine fachliche Regel bewusst umgangen wird. Dadurch werden Fehler sichtbar und korrigierbar.

Die Gestaltung einer Entity beginnt immer mit der Sprache. Welche Verben und Substantive beschreiben ihr Verhalten? Welche Entscheidungen trifft sie selbst, welche delegiert sie? Solche Fragen sind nicht nur technisch, sondern semantisch. Entities sind daher nicht einfach Klassen, sondern Definitionen von Verantwortung – und Verantwortung ist das zentralste Gestaltungsmittel im taktischen DDD.

6.1.2 Value Object - Bedeutung und Ausdruck

Value Objects repräsentieren Konzepte, bei denen nicht die Identität, sondern der Wert entscheidend ist. Sie sind die präzise Art, Fachlichkeit auf feinster Ebene auszudrücken. Ein Value Object existiert nur durch seine Eigenschaften und das Verhalten, das daraus folgt.

Die entscheidende Eigenschaft ist Immutability. Ein Value Object ändert sich nicht, es wird ersetzt. Dieser scheinbar technische Grundsatz ist ein tiefes fachliches Statement: Bedeutung ändert sich nicht rückwirkend. Wenn ein Preis, eine Menge oder ein Zeitraum sich ändert, entsteht ein neuer Wert, nicht ein veränderter alter. Das bewahrt die Kohärenz der Domäne über die Zeit.

Value Objects sind gleich, wenn ihre Eigenschaften gleich sind – nicht, weil sie dasselbe Objekt sind. Sie werden anhand ihrer Bedeutung verglichen. Das macht sie zu verlässlichen Bausteinen für Berechnungen, Validierungen und Kommunikation zwischen Aggregaten.

Darüber hinaus sind Value Objects Träger von Sprache. Ein System, das in Begriffen der Domäne spricht, ist nachhaltiger und lesbarer als eines, das sich auf primitive Typen stützt. Eine Operation wie betrag.addiere(andererBetrag) ist

nicht nur technisch, sondern semantisch verständlich. Die Syntax spiegelt die Sprache der Fachlichkeit wider.

Value Objects sind damit das Gegenteil von Beliebigkeit. Sie sind präzise Begriffe, die Bedeutung konservieren und Fehler auf semantischer Ebene verhindern. Sie machen den Code ausdrucksstark und zugleich robust.

6.1.3 Aggregate - Konsistenzgrenzen und Invarianten

Aggregates bündeln Entities und Value Objects zu Einheiten fachlicher Konsistenz. Sie definieren, wo Transaktionen enden und wo Kohärenz gewährleistet sein muss. Eine Aggregate Root ist der einzige Einstiegspunkt, über den interner Zustand verändert werden darf.

Das Gestalten von Aggregates ist eine Übung in Balance. Sind sie zu groß, verliert man Agilität; sind sie zu klein, verliert man Bedeutung. Der Schnitt entlang fachlicher Invarianten ist entscheidend. Diese Invarianten sind die Regeln, die immer gelten müssen – nicht manchmal, nicht eventuell. Wenn zwei Daten oder Verhalten gemeinsam gültig sein müssen, gehören sie ins gleiche Aggregate. Wenn sie unabhängig variieren, nicht.

Aggregates bilden in der Praxis häufig die schwierigsten, aber auch wichtigsten Designentscheidungen im taktischen DDD. Die Kunst besteht darin, Konsistenzgrenzen zu finden, die fachlich notwendig, aber technisch tragfähig sind. Aggregates sind kein rein strukturelles Pattern, sondern Ausdruck fachlicher Verantwortung. Ihre Aufgabe ist es, Integrität zu wahren, während sie zugleich Autonomie ermöglichen.

Das zentrale Kriterium ist dabei die Invariante. Eine Invariante beschreibt, was unter keinen Umständen verletzt werden darf. In einem Banksystem etwa könnte gelten: „Ein Konto darf nie einen negativen Saldo aufweisen, es sei denn, es verfügt über eine genehmigte Überziehung.“ In diesem Satz steckt alles, was man braucht, um ein Aggregate zu schneiden: das Konto, der Saldo, die Regel. Diese drei Elemente bilden eine semantische Einheit, die in einem Transaktionskontext konsistent bleiben muss.

Aggregates sind somit keine Sammlung von Entitäten, sondern Abbildungen von fachlicher Kohärenz. Die Aggregate Root ist der Wächter dieser Kohärenz. Sie entscheidet, welche Operationen erlaubt sind, und vermittelt zwischen innerem Zustand und äußerem Verhalten.

Mit zunehmender Systemgröße wird die Beziehung zwischen Aggregaten komplexer. Kommunikation über Grenzen hinweg erfolgt dann meist über Domain Events. In solchen Fällen wird die strikte Konsistenz eines Monolithen gegen eine flexible Form der Integrität eingetauscht. Das bedeutet, dass Änderungen nicht überall gleichzeitig sichtbar sind, sondern sich über Ereignisse verbreiten. Dieses Prinzip der eventual consistency ist kein Mangel, sondern eine bewusste architektonische Entscheidung. Es erlaubt Systeme, verteilt und dennoch zuverlässig zu sein.

Sara Pellegrinis Konzept der **Dynamic Consistency Boundaries** (**DCB**) ergänzt diese Sicht. Es erkennt an, dass Konsistenz kein absoluter, sondern ein kontextueller Wert ist. Ein System kann situativ entscheiden, wie viel Kohärenz es benötigt. So entsteht ein Modell, das flexibel mit Last, Verfügbarkeit und fachlicher Bedeutung umgeht. Anstatt ein einziges, universelles Muster zu erzwingen, erlaubt DCB eine Vielfalt von Konsistenzstrategien – von strikter Isolation bis zu weicher, zeitversetzter Synchronisierung.

Im Kern bedeutet das: Aggregates sind keine Grenzen der Software, sondern Grenzen des Verständnisses. Sie definieren, was ein Team als untrennbar begreift. Und weil sich dieses Verständnis mit der Domäne verändert, ist auch das Design von Aggregaten evolutionär. In einem lebendigen System werden Aggregate neu geschnitten, Invarianten verschoben, Ereignisse anders gruppiert. Taktisches DDD begreift das nicht als Schwäche, sondern als Ausdruck von Reife.

6.1.4 Domain Event - Das Gedächtnis der Domäne

Domain Events sind die elementarste Form der Kommunikation in einem DDD-System. Sie sind die Sprache, mit der sich Fachlichkeit über Zeit hinweg ausdrückt. Ein Domain Event ist keine Benachrichtigung und kein Signal – es ist eine Aussage

über eine Tatsache, die in der Domäne eingetreten ist. Diese Formulierung ist entscheidend: Ein Event beschreibt etwas, das passiert ist, nicht etwas, das passieren soll.

Ein Domain Event trägt drei wesentliche Eigenschaften. Es ist vergangenheitsorientiert, weil es nur über bereits eingetretene Ereignisse spricht. Es ist unveränderlich, weil eine Tatsache nicht rückgängig gemacht werden kann. Und es ist bedeutungstragend, weil es in der Sprache der Fachlichkeit formuliert ist.

Durch Domain Events wird Zeit zu einem ersten Bürger im System. Anstatt Zustand in Tabellen oder Objekten zu fixieren, erzählt man eine Geschichte aus Ereignissen. Jede neue Tatsache reiht sich in diese Chronik ein. Das ermöglicht Nachvollziehbarkeit, Auditing, Reproduktion und – vielleicht am wichtigsten – Verständnis.

Ereignisse sind auch das Mittel, mit dem Aggregate, Bounded Contexts und ganze Systeme lose gekoppelt werden. Wenn ein Event auftritt, können andere Komponenten darauf reagieren, ohne den Urheber zu kennen. Das schafft Unabhängigkeit und ermöglicht Evolution. In dieser Form werden Domain Events zur Infrastruktur des Wissens: Sie verbinden Vergangenheit, Gegenwart und Zukunft der Fachlichkeit.

In eventgetriebenen Architekturen bilden Domain Events die Grundlage für asynchrone Kommunikation. Sie sind zugleich technisch (etwa als Nachrichten) und semantisch (als Konzepte). Ihre größte Stärke liegt jedoch nicht in der Integration, sondern in der Klarheit. Jedes Domain Event zwingt dazu, Bedeutung explizit zu machen. Man kann nichts publizieren, das man nicht verstanden hat.

6.1.5 Repository - Die Abstraktion des Datenzugriffs

Ein Repository ist die Brücke zwischen Domänenmodell und Persistenz. Sein Zweck ist es, das Speichern und Laden von Aggregaten so zu abstrahieren, dass die Fachlogik davon unberührt bleibt. Der Fachcode denkt in Aggregaten, nicht in Tabellen.

Ein gutes Repository verhält sich so, als besäße es eine Sammlung von Objekten im Speicher. Methoden wie findById, save oder delete sind fachlich formuliert und

folgen der Ubiquitous Language. Die Implementierung – ob über eine relationale Datenbank, ein Dokumenten-Store oder ein Event-Log – bleibt austauschbar.

Damit ist das Repository ein Muster der Verantwortungstrennung. Es ermöglicht, dass sich die Domäne ausschließlich mit Verhalten beschäftigt, während Persistenz ein technisches Detail bleibt. In testbaren Architekturen wird das Repository typischerweise durch Schnittstellen abstrahiert, die im Domain- oder Application-Layer definiert und in der Infrastruktur implementiert werden.

Ein wichtiger Aspekt ist, dass Repositories immer mit Aggregaten arbeiten, nie mit Teilobjekten. Sie garantieren die Konsistenz der Transaktion und verhindern, dass interne Strukturen nach außen dringen. Das Repository ist damit mehr als ein Datenzugriffsmuster – es ist ein strukturelles Bindeglied zwischen Modell und Architektur.

6.1.6 Service - Orchestrierung

Nicht jede Regel hat ein natürliches Zuhause in einer Entity oder einem Aggregate. Manche Operationen betreffen mehrere Aggregate, andere sind berechnender oder koordinierender Natur. Für solche Fälle existiert der Service.

Im DDD unterscheidet man zwischen Domain Services und Application Services. Domain Services enthalten reine Fachlogik, die keinem konkreten Objekt zugeordnet werden kann. Sie modellieren Prozesse, Berechnungen oder Prüfungen, die im semantischen Raum der Domäne liegen. Application Services hingegen orchestrieren Use Cases. Sie sprechen mit Repositories, starten Transaktionen, rufen Aggregates auf und steuern den Ablauf eines Anwendungsfalls.

Die Trennung ist nicht nur technischer Natur, sondern Ausdruck von Verantwortungsbewusstsein. Domain Services dürfen keine technischen Abhängigkeiten besitzen. Application Services dürfen keine Fachlogik enthalten. So bleibt die Domäne rein und die Anwendung handlungsfähig.

Services verkörpern eine Haltung zur Modularität. Sie stehen dort, wo Koordination notwendig ist, aber keine Eigentümerschaft. Ihre Existenz ist oft ein Hinweis auf Grenzen – sie zeigen, wo Fachlichkeit miteinander in Kontakt tritt, ohne

sich zu verschmelzen. Dadurch werden Systeme verständlicher und Änderungen nachvollziehbarer.

6.1.7 Factory - Externalisierung von Erzeugungslogik

Factories sind die Mechanismen, mit denen neue Aggregate oder Entities korrekt erzeugt werden. Sie fassen Erstellungslogik zusammen, prüfen Voraussetzungen und garantieren, dass ein Objekt in einem gültigen Zustand startet.

Eine Factory ist nicht einfach ein Konstruktionswerkzeug, sondern ein Instrument der Integrität. Sie bewahrt vor fehlerhaften Zuständen und stellt sicher, dass alle Invarianten beim Entstehen geprüft werden. Besonders bei komplexen Aggregaten oder bei Event Sourcing, wo eine Reihe von Ereignissen initial erzeugt werden müssen, wird die Factory zur formalen Startbedingung des Systems.

Ihre Existenz befreit das Modell von Nebensächlichkeiten. Wenn Objekte nicht wissen müssen, wie sie entstehen, können sie sich ganz auf ihr Verhalten konzentrieren. Factories unterstützen damit ein Prinzip, das sich durch das gesamte DDD zieht: jede Verantwortung an den Ort zu legen, wo sie semantisch hingehört.

6.2 Architekturmuster

Architektur im Sinne des taktischen DDD ist kein technischer Selbstzweck. Sie ist das sichtbare Ergebnis einer Haltung zur Fachlichkeit. Jede architektonische Entscheidung ist letztlich eine Aussage darüber, wie viel man von der Domäne verstanden hat – und wie konsequent man bereit ist, dieses Verständnis in Code zu übersetzen. Taktisches Domain-Driven Design betrachtet Architektur als sprachliches Phänomen. Strukturen entstehen nicht aus Technologie, sondern aus Semantik. Wenn Fachlichkeit der Ursprung ist, wird Architektur zur Form der Bedeutung.

Architektur ist in diesem Sinne nicht starr. Sie ist ein Gespräch über Stabilität und Wandel. Manchmal braucht man klare Grenzen und Schichten, manchmal fließende Übergänge und Schnittstellen. Die Stärke des DDD liegt darin, dass es

nicht eine Architektur vorschreibt, sondern ein Kriterium: Die gewählte Struktur muss den Fluss der Fachlichkeit unterstützen.

6.2.1 Layered Architecture

Die klassische Schichtarchitektur ist das vertraute Bild vieler Systeme. Sie teilt Software in Ebenen: Präsentation, Anwendung, Domäne und Infrastruktur. Jede Ebene kennt nur die direkt darunterliegende. Daten fließen nach unten, Steuerung nach oben. Diese Ordnung hat einen unbestreitbaren Vorteil: Sie ist leicht zu verstehen.

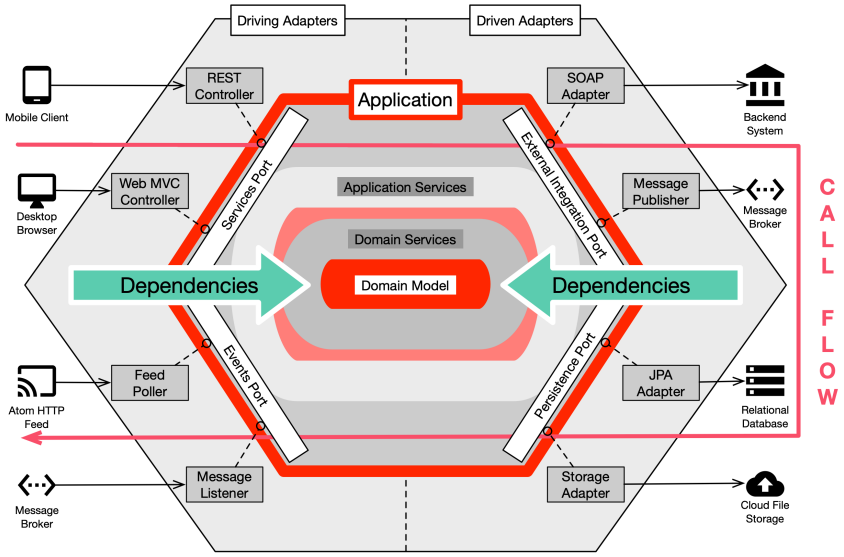
Für Teams, die mit DDD beginnen, bietet dieses Modell oft eine solide Basis. Es schafft Disziplin, zwingt zu Trennung von Verantwortlichkeiten und erleichtert das Testen in klar abgegrenzten Bereichen. Doch genau diese Einfachheit kann auch zur Begrenzung werden. Wenn die Domäne wächst, entstehen feine Abhängigkeiten zwischen den Schichten. Fachlogik sickert in den Application-Layer, technische Details schleichen sich in die Domäne. Die Grenzen verwischen, bis Schichten nur noch als Projektordner existieren.

Das Problem liegt weniger in der Architektur als in ihrer impliziten Hierarchie. Die Schichtenarchitektur setzt die Technik an die Basis und die Fachlichkeit in die Mitte. Dadurch entsteht eine subtile Abhängigkeit der Sprache von der Infrastruktur. Was zunächst wie eine organisatorische Ordnung wirkt, ist in Wahrheit eine semantische Verzerrung: Die Domäne ist nicht das Fundament, sondern das Zentrum. Wenn die Fachlichkeit sich ändert, muss sich alles darum herum bewegen können. In der klassischen Schichtenarchitektur ist das selten der Fall.

Dennoch bleibt sie in bestimmten Kontexten nützlich. In Systemen, die überschaubar, stabil oder stark datenzentriert sind, kann eine Layered Architecture vollkommen ausreichen. Ihre Schwäche zeigt sich erst, wenn Systeme zu leben beginnen, wenn sich Modelle weiterentwickeln und Teams unabhängig agieren sollen. Dann wird das Schichtenmodell zu eng für das, was DDD ermöglichen will: fachlich getriebene Evolution.

6.2.2 Hexagonal Architecture

Die hexagonale Architektur, von Alistair Cockburn als Ports and Adapters bekannt gemacht, entstand aus dem Wunsch, diese Begrenzung zu überwinden. Sie kehrt das Verhältnis zwischen Fachlichkeit und Technik um: Alle Abhängigkeiten müssen nach innen Zeigen



Im Zentrum des Hexagons steht das Domain Model. Es ist der Kern des Verständnisses, die reine Fachlichkeit, unberührt von Frameworks und Datenbankzugriffen. Im Domain Model findet man meist Entities, Value Objects, Aggregates und interne Domain Events. Das Domain Model wird im Bereich des Domain Layers meist ergänzt durch Domain Services.

Darum liegt der Application Layer. Er orchestriert die Use Cases. Er beinhaltet Application Services und Ports, er koordiniert Abläufe, führt Transaktionen aus und verwaltet Policies. Der Application Layer ist die Schicht der Handlung – er verbindet das, was die Domäne bedeutet, mit dem, was das System tut.

An der Peripherie befinden sich die Adapter. Sie implementieren die Ports, über die das System mit der Außenwelt kommuniziert. Ein Adapter kann eine Be-

nutzeroberfläche sein, ein REST-Endpunkt, eine Nachricht in einem Message Broker oder ein Datenbank-Treiber. Diese Struktur macht eine einfache, aber tiefgreifende Aussage: Alle Abhängigkeiten zeigen nach innen. Alles Wissen über Technik, Frameworks oder Infrastruktur bleibt außerhalb des Kerns.

Das Hexagon ist damit keine rein technische Form, sondern ein semantisches Versprechen. Es schützt die Bedeutung vor der Erosion durch Technologie. Systeme, die so gebaut sind, können sich langfristig verändern, weil ihre Essenz stabil bleibt. Fachlogik kann getestet werden, ohne Infrastruktur zu starten; Infrastruktur kann ersetzt werden, ohne die Domäne zu berühren.

Doch auch hier gilt: Die Hexagonale Architektur ist ein Werkzeug, kein Dogma. Nicht jedes System braucht diese Trennungstiefe. In kleinen Projekten kann sie zu aufwendig wirken, in hochgradig explorativen Phasen sogar hinderlich. Entscheidend ist, dass das Team versteht, warum es sich für oder gegen diese Form entscheidet. Im DDD gilt: Architektur muss zum Reifegrad der Organisation passen. Das Hexagon zeigt eine Richtung – es zwingt zu Sprache, zu Klarheit und zu bewussten Grenzen. Aber es verlangt auch Disziplin, und Disziplin entsteht nur, wenn man die Vorteile spürt.

6.2.3 Event Sourcing

Während klassische Architekturen den aktuellen Zustand eines Systems speichern, behandelt Event Sourcing den Zustand als Ergebnis seiner Geschichte. Jede Veränderung in der Domäne wird als Ereignis gespeichert, und der aktuelle Zustand ergibt sich durch das Abspielen dieser Ereignisse. Damit wird nicht mehr die Gegenwart, sondern der Verlauf der Zeit zum primären Speicher.

Dieser Perspektivwechsel ist radikal. Er macht die zeitliche Dimension explizit und stellt Fachlichkeit in einen historischen Kontext. Ein System, das Ereignisse aufzeichnet, weiß nicht nur, was jetzt gilt, sondern auch, warum. Diese Nachvollziehbarkeit schafft nicht nur Auditierbarkeit, sondern ermöglicht Lernen: Man kann die Vergangenheit analysieren, Fehler rekonstruieren und Alternativen simulieren.

Event Sourcing zwingt zu Präzision. Jede fachliche Entscheidung muss als Ereignis beschrieben werden, in der Sprache der Domäne. „Antrag wurde eingereicht“, „Scoring wurde berechnet“, „Vertrag wurde freigegeben“. Dadurch entsteht ein Modell, das die Kommunikation in der Organisation widerspiegelt. Domain Events werden zu historischen Dokumenten, nicht zu technischen Signalen.

Doch Event Sourcing ist nicht trivial. Es verlangt sorgfältige Versionierung, gutes Verständnis für Evolutionsstrategien und eine klare Trennung von Command- und Query-Seite. Man muss lernen, Geschichte als Quelle der Wahrheit zu begreifen, und akzeptieren, dass sich Wahrheit im Nachhinein nicht ändern lässt. Ein Event kann ergänzt, aber nie gelöscht werden. Systeme mit Event Sourcing müssen also mit ihrer Vergangenheit leben – im Guten wie im Schlechten.

In Kombination mit Dynamic Consistency Boundaries entsteht hier eine mächtige Struktur: Konsistenz wird flexibel, Geschichte bleibt vollständig. Systeme werden nicht mehr nur auf der Ebene des Zustands resilient, sondern auf der Ebene der Zeit. Wer die zeitliche Dimension als Modellierungsmittel begreift, kann Software entwerfen, die Veränderung nicht fürchtet, sondern integriert.

6.2.4 CQRS

Command Query Responsibility Segregation (CQRS) ist die logische Ergänzung von Event Sourcing, aber auch unabhängig davon ein wertvolles Prinzip. Es besagt, dass Schreiben und Lesen zwei verschiedene Akte mit unterschiedlichen Zielen sind – und daher auch unterschiedliche Modelle verdienen.

Die Command-Seite kümmert sich um die Veränderung der Welt. Sie nimmt Befehle entgegen, validiert Regeln, erzeugt Domain Events und speichert sie. Die Query-Seite dagegen stellt Informationen bereit. Sie braucht keine Transaktionen, keine Invarianten, sondern Geschwindigkeit und Ausdruckskraft. Durch diese Trennung kann jede Seite für ihren Zweck optimiert werden.

Der eigentliche Wert von CQRS liegt nicht in der Technik, sondern im Denken. Es trennt Zustandsveränderung von Wissensabfrage. Diese Unterscheidung zwingt

dazu, die Domäne als System aus Perspektiven zu verstehen. Das Schreiben dient der Kausalität, das Lesen der Erkenntnis.

In komplexen Systemen führt das zu einer neuen Form von Klarheit. Teams, die CQRS praktizieren, lernen, dass dieselbe Domäne unterschiedlich gesehen werden darf – solange die Bedeutungen konsistent bleiben. Die Schreibseite bewahrt Integrität, die Leseseite bietet Interpretationen. Das ist nicht nur technisch, sondern epistemologisch: Es beschreibt, wie Wissen entsteht und geteilt wird.

CQRS kann mit oder ohne Event Sourcing existieren. In eventbasierten Systemen entstehen Projektionen aus Ereignissen. In klassischen Systemen erzeugen Commands Zustände, die über optimierte Read-Modelle abgefragt werden. In beiden Fällen gilt: CQRS ist kein Architektur-Pattern, das man einführt – es ist eine Haltung zur Trennung von Verantwortung.

Diese Haltung passt tief zum Geist des DDD: Trenne, was unterschiedliche Bedeutung hat, und verbinde nur, was gemeinsam verstanden wird. So entsteht nicht Symmetrie durch Gleichmacherei, sondern durch Klarheit.

6.3 Deployment-Optionen

DDD ist älter als der Begriff „Microservice“. Als Eric Evans 2003 sein Buch veröffentlichte, dachte niemand in Services, sondern in Kontexten. Domain-Driven Design entstand lange vor Microservices, Containern oder Cloud-Umgebungen. Es war nie auf verteilte Systeme angewiesen, um wirksam zu sein. Im Gegenteil: Die frühen DDD-Modelle lebten in modularen, monolithischen Anwendungen – gut strukturiert, sprachlich sauber und fachlich kohärent. DDD ist ein Denkrahmen, kein Infrastrukturkonzept. Die Aufgabe besteht nicht darin, Systeme zu zerteilen, sondern Grenzen zu gestalten. Wie diese Grenzen deployt werden, ist eine zweite, nicht die erste Frage.

Deshalb gilt: **DDD braucht keine Microservices.**

In der Praxis gilt eine einfache Heuristik: Modell zuerst, Deployment später. Wer zu früh verteilt, verteilt nur Missverständnisse.

6.3.1 Monolith

Das Wort „Monolith“ ist zu Unrecht negativ besetzt. Es suggeriert Starrheit, Undurchdringlichkeit und technische Schuld. Doch im Sinne des DDD ist ein Monolith nichts anderes als ein System, das in einem Prozessraum läuft. Ob dieser Prozess wohlgeordnet oder chaotisch ist, entscheidet nicht die Deployment-Form, sondern das Design.

Ein sauber geschnittener Monolith kann eine der klarsten Ausdrucksformen des Domain-Driven Designs sein. Wenn die internen Module entlang fachlicher Grenzen strukturiert sind, wenn die Kommunikation zwischen ihnen über explizite Schnittstellen erfolgt und wenn jedes Modul als Bounded Context verstanden wird, dann entsteht etwas, das man heute als Modulith bezeichnet.

Der Modulith besteht aus mehreren Kontexten, die innerhalb eines Deployments leben, aber klar voneinander getrennt sind. Jeder Kontext kann unabhängig getestet, entwickelt und verändert werden, ohne den Rest zu stören. Das Deployment ist gemeinsam, aber die Architektur ist dezentral. So entsteht ein System, das sowohl die Einfachheit gemeinsamer Prozesse als auch die Klarheit getrennter Modelle vereint.

Ein solcher Aufbau ist oft die beste Wahl für Organisationen, die in stabilen Geschäftsmodellen arbeiten oder sich in einer Phase des Lernens befinden. Er erlaubt schnelle Refactorings, einfache Deployments und gemeinsame Datenhaltung, ohne auf die Prinzipien des DDD zu verzichten. Vor allem aber bietet er eine Plattform für Evolution: Wenn die Fachlichkeit wächst, können einzelne Module sich verselbstständigen – in eigene Deployments, eigene Teams, eigene Lebenszyklen.

Der Monolith ist dann kein Widerspruch zu Microservices, sondern deren logischer Ursprung.

6.3.2 Microservices - verteilte Grenzen

Die Idee des Microservice hat die Softwarearchitektur in den letzten Jahren verändert. Sie verspricht Unabhängigkeit, Geschwindigkeit und Teamautonomie. Doch sie bringt auch neue Formen der Komplexität mit sich. Microservices sind aus

DDD Perspektive nichts anderes als physisch getrennte Bounded Contexts. Sie machen sichtbar, was DDD schon lange beschreibt: dass Grenzen nicht nur gedacht, sondern gelebt werden müssen.

Wenn jedes Team für einen Kontext verantwortlich ist, entsteht echte Ownership. Die Sprache innerhalb des Services kann sich unabhängig entwickeln. Entscheidungen, die früher zentral getroffen wurden, wandern dorthin, wo das Wissen ist. Diese Autonomie ist der größte Gewinn – aber auch die größte Herausforderung. Denn verteilte Systeme sind nie nur technische Artefakte; sie sind Organisation in Code gegossen.

Wo der Modulith interne Schnittstellen nutzt, kommunizieren Microservices über Netzwerke. Das macht Fehler sichtbar, aber auch teuer. Daten müssen getrennt werden, Konsistenz wird lose, Integrationen müssen orchestriert werden. Der Preis der Unabhängigkeit ist erhöhte Komplexität. DDD hilft hier, weil es diese Trennung nicht als technische Forderung, sondern als fachliche Notwendigkeit begreift. Ein Microservice sollte nie entstehen, weil man Services möchte, sondern weil ein Kontext eine eigene Sprache, ein eigenes Modell und ein eigenes Tempo braucht.

Im besten Fall ist der Übergang organisch: Aus einem Modulith wachsen Microservices hervor, wenn Grenzen stabil geworden sind. Wenn Teams ihre Kontexte kennen, wenn Kommunikation verstanden ist und wenn Modelle unabhängig voneinander evolvieren können, dann kann die physische Trennung stattfinden, ohne die semantische Integrität zu gefährden.

Das ist der eigentliche Sinn des Prinzips „Model first, deploy later“. Erst wenn man die Sprache beherrscht, lohnt es sich, sie zu verteilen.

6.3.3 Self-contained Systems – die Evolution der Verantwortung

Zwischen Monolith und Microservice liegt ein Ansatz, der beide Perspektiven vereint: das Konzept der Self-contained Systems (SCS). Es wurde vor allem im deutschsprachigen Raum geprägt und beschreibt eine Architektur, in der jedes

System eigenständig alle Schichten enthält – von der UI bis zur Persistenz –, aber über definierte Integrationspunkte mit anderen Systemen verbunden ist.

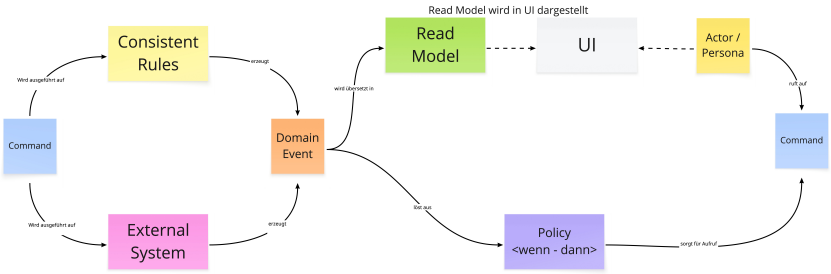
SCS folgt denselben Ideen wie DDD, aber mit einem pragmatischeren Blick. Anstatt Services nach technischen Kriterien zu trennen, werden Systeme entlang fachlicher Domänen geschnitten. Jedes System ist in sich geschlossen, unabhängig deploybar und dennoch Teil eines größeren Ganzen. Es besitzt ein eigenes Modell, eine eigene Oberfläche und eine eigene Datenhaltung.

Im Vergleich zu Microservices sind SCS oft gröber geschnitten, aber stabiler. Sie bilden die organisatorische Realität besser ab, weil sie komplette Funktionsbereiche umfassen. Ein SCS ist damit weniger ein Service als eine kleine Anwendung im Systemverbund.

In der Praxis sind Self-contained Systems eine natürliche Weiterentwicklung des Modulithen. Ein Modul, das sich in der Praxis als stabiler Kontext erwiesen hat, kann sich zu einem eigenen System entwickeln, ohne sein Modell zu verlieren. So entsteht eine Architektur, die evolutionär wächst, nicht revolutionär. Sie bleibt anschlussfähig, weil sie ihre Grenzen aus der Fachlichkeit ableitet, nicht aus der Technologie.

6.4 Design-Level EventStorming - Vom Modell zur Implementierung

Das Design-Level EventStorming ist die Brücke zwischen strategischem Denken und technischer Umsetzung. Es übersetzt die Erkenntnisse aus dem Big-Picture-EventStorming und der Kontextabgrenzung in ein konkretes, implementierbares Modell innerhalb eines Bounded Contexts. Bildlich gesprochen zeichnet das Big Picture die Landkarte und das Design Level entwirft den Bauplan eines Gebäudes. Sein Ziel ist es, das Verhalten eines Systems so zu verstehen, dass es unmittelbar in Code übersetzt werden kann – und zwar so, dass dieser Code die Sprache der Fachlichkeit spricht.



6.4.1 Vorbereitung – Der Scope und die Bühne

Ein Design-Level-Workshop beginnt immer mit einem klaren Fokus. Im Gegensatz zum Big Picture, das eine gesamte Domäne untersucht, konzentriert man sich hier auf einen begrenzten Fluss von Ereignissen - eine Abfolge von Entscheidungen und Reaktionen innerhalb eines Bounded Context. Dieser Scope kann ein einzelner Use Case sein, ein zentrales Ereignis oder ein Teilprozess, der fachlich bedeutsam ist.

Der Raum, in dem gearbeitet wird – physisch oder digital –, ist ein zentrales Werkzeug. Alle Teilnehmer:innen müssen die gleiche Fläche sehen und gleichzeitig darauf wirken können. Orangefarbene Zettel markieren Domain Events, gelbe Akteure, blaue Commands, violette Policies, grüne Aggregates, hell pinke External Systems und grüne Read Models. Diese Farben sind mehr als nur visuelle Unterscheidung – sie bilden die Syntax der gemeinsamen Sprache.

Die moderierende Person führt das Team in kleinen Schritten vom ersten Ereignis bis zur vollständigen Ereigniskette. Die Devise lautet: „Lass die Domäne sprechen, nicht die Technik.“ Jedes Event wird in der Vergangenheitsform formuliert, jedes Command als Handlung, die zu einem Event führen könnte. So entsteht nach und nach ein sichtbarer Fluss von Ursache und Wirkung.

6.4.2 Startpunkt auswählen

Jedes Design-Level EventStorming beginnt mit einer bewussten Eingrenzung. Der Scope sollte klein genug sein, um innerhalb eines Workshops in die Tiefe zu gehen, aber groß genug, um eine zusammenhängende fachliche Geschichte zu erzählen. Typische Startpunkte sind eine Subdomain, ein Bounded Context oder der Bereich zwischen zwei zentralen Domain Events aus einem Big-Picture-Workshop.

Der Startpunkt wird auf dem Board sichtbar gemacht – meist ein zentrales Ereignis oder eine Entscheidung. Er markiert die Grenze der Exploration: Von hier aus wollen wir verstehen, wie wir von einem fachlichen Zustand zum nächsten gelangen. Das Ziel dieser ersten Phase ist Klarheit, nicht Vollständigkeit.

6.4.3 Fachliche Regeln identifizieren

Nachdem der Scope gesetzt ist, geht es um die fachliche Substanz. Welche Regeln bestimmen, wie sich das System verhält? Diese Regeln – Consistent Rules – sind das Herz des Workshops. Sie werden auf breiten gelben Stickies gesammelt, meist frei und ohne Ordnung.

Jede Regel beschreibt eine Bedingung, ein Kriterium oder eine Entscheidung, die für die Fachlichkeit wesentlich ist. Sie wird in Alltagssprache formuliert, ohne Rückgriff auf technische Begriffe. Dabei gilt: Lieber zu viele Regeln sammeln als zu früh strukturieren. Das Board soll Denken sichtbar machen, nicht Ordnung erzwingen.

Schon an dieser Stelle lässt sich ein erster Bezug zu Test-Driven Development herstellen. Jede Regel, die auf einem gelben Sticky notiert ist, kann als Testhypothese betrachtet werden. In der frühen Implementierung kann daraus ein Unit Test entstehen, der überprüft, ob diese Regel im Code erfüllt wird. So wird aus einem Workshop-Artefakt ein Baustein fachlicher Qualitätssicherung.

6.4.4 Fachliche Regeln gruppieren – Konsistenzgrenzen sichtbar machen

Wenn genügend Regeln gesammelt sind, beginnt die strukturierende Phase. Nun werden die Regeln gruppiert – nach dem Kriterium, welche von ihnen immer gemeinsam konsistent bleiben müssen. Diese Gruppen repräsentieren fachliche Konsistenzgrenzen. Sie sind das, was im taktischen DDD später als Aggregates bezeichnet wird, hier jedoch bewusster als Consistent Rule Sets beschrieben wird.

Das Gruppieren ist kein mechanischer Schritt, sondern eine tiefgehende Diskussion. Fachleute und Entwickler:innen gleichen ihre mentalen Modelle ab: Welche Regeln gehören aus fachlicher Sicht zusammen? Welche hängen voneinander ab? Welche dürfen unabhängig voneinander existieren?

In diesem Moment verschiebt sich die Perspektive von losen Regeln zu Systemlogik. Die entstehenden Gruppen sind semantische Einheiten.Sie bilden den Kern dessen, was später im Domain Layer als Aggregate Roots, Entities und Value Objects implementiert wird. Ihre Namen sind entscheidend – sie werden zu Elementen der Ubiquitous Language und sollten später im Code unverändert auftauchen.

6.4.5 Commands und Events hinzufügen

Sobald die Regelcluster stehen, werden sie mit Interaktion verbunden. Die Frage lautet: Wie wird diese Regel ausgelöst, und was passiert, wenn sie ausgeführt wird?

Die Antworten führen zu Commands und Domain Events. Ein Command ist eine Handlung – ein Aufruf, der eine Regel aktiviert. Ein Event ist die Beobachtung des Ergebnisses. Commands werden auf blaue, Events auf orangefarbene Stickies geschrieben und direkt an die entsprechenden Regelgruppen geheftet.

So entsteht das Rückgrat des Modells: eine Abfolge von Ursachen und Wirkungen, eine Geschichte, die die Fachlogik in Bewegung zeigt.

Für Test-Driven Development bedeutet das: Jede Kombination aus Command und Event wird zu einem Testfall. Ein Test beschreibt den erwarteten Effekt eines fachlichen Aufrufs: „Wenn Command X ausgeführt wird, entsteht Event Y.“Die Testfälle dokumentieren das Verhalten der Aggregates – und damit das Verhalten des Systems.

6.4.6 Policies und Read Models – Das System wird vollständig

Im letzten Schritt wird das System verbunden. Jetzt geht es um den Fluss der Reaktionen – darum, wie einzelne Teile der Domäne miteinander interagieren. Hier kommen Policies und Read Models ins Spiel.

Policies beschreiben orchestrierenden Regeln, die auf Ereignisse reagieren. Sie verbinden Events mit neuen Commands in einer wenn (Event), dann (Commands) Semantik. Wenn ein bestimmtes Event eintritt, wird automatisch eine fachliche Reaktion ausgelöst. Diese Reaktion kann synchron oder asynchron erfolgen, intern oder übergreifend, aber sie ist immer Ausdruck einer Geschäftsregel. In der späteren Architektur sind Policies häufig Kandidaten für Application- oder Domain Services.

Read Models dagegen sind die Perspektive der Auswertung. Sie dienen nicht der Steuerung, sondern der Beobachtung. Ein Read Model projiziert Domain Events auf eine für Benutzer oder externe Systeme verständliche Form – eine Abbildung, ein Bericht, eine Anzeige. Sie können auch für CQRS-basierte Architekturen eine spannende Perspektive spielen.

Am Ende dieser Phase ist das Board ein vollständiges, fachlich geschlossenes Abbild des Kontextes. Alle Aggregates, Commands, Events, Policies und Read Models sind miteinander verbunden. Das System hat eine erkennbare Topologie – ein Netzwerk aus Bedeutung, Verantwortung und Kommunikation.

6.4.7 Ergebnis – Ein System, das sich selbst erklärt

Ein gelungenes Design-Level EventStorming endet nicht mit einem bunten Board, sondern mit einem geteilten Verständnis. Die Teilnehmenden haben nicht nur

Regeln gesammelt, sondern gelernt, wie diese Regeln in Architektur, Code und Sprache übersetzt werden. Die entstehenden Systeme sind dadurch verständlicher, testbarer und erweiterbarer. Sie basieren nicht auf Annahmen, sondern auf überprüfter Bedeutung. In dieser Kombination aus Workshop, Test und Architektur zeigt sich die eigentliche Stärke des taktischen Domain-Driven Designs: Es verbindet Denken, Handeln und Entwickeln zu einer kohärenten Praxis.

6.5 Zusammenfassung und Ausblick

Das taktische Domain-Driven Design ist die Kunst, Fachlichkeit in Struktur zu übersetzen. Es vereint die Bausteine des Modells mit Architekturprinzipien und praktischen Methoden zu einem kohärenten Ganzen. Die Patterns – Entities, Value Objects, Aggregates, Domain Events, Services, Repositories und Factories – sind das alphabetische Vokabular dieser Sprache. Architekturen wie Layered, Hexagonal, CQRS oder Event Sourcing sind ihre Grammatik, die Deployment-Optionen ihre Ausdrucksformen.

Wer DDD auf taktischer Ebene praktiziert, baut Software, die in der Sprache der Fachlichkeit spricht. Das Ziel ist kein Code, der bloß funktioniert, sondern ein System, das verstanden wird, sich mit der Domäne entwickelt und Veränderungen als Teil seiner Natur akzeptiert. Tactical DDD ist damit nicht das Ende der Architektur, sondern ihr lebendiger Kern: die ständige Übersetzung von Wissen in Design, von Design in Code und von Code zurück in gemeinsames Verständnis.

7 Abschluss: Do's and Don'ts im Domain-Driven Design

Domain-Driven Design ist kein Dogma, sondern eine Haltung, die Denken, Zusammenarbeit und Architekturentscheidungen prägt. Dennoch gibt es typische Stolperfallen, die fast jedes Team irgendwann erlebt – und ebenso typische Erfolgsfaktoren, die über das Gelingen entscheiden. Dieses Abschlusskapitel fasst die wichtigsten Do's and Don'ts zusammen – nicht als Checkliste, sondern als Orientierung für die Praxis.

7.1 Fachlichkeit zuerst, Technik später

DDD beginnt immer im **Problemraum**. Wer vorschnell in Frameworks, Technologien oder Architekturdiagramme springt, verliert leicht den Blick für das eigentliche „Warum".

Do

- Starte mit der Domäne: Kund:innen, Geschäftsprozesse, Ziele, Wertversprechen.
- Arbeite mit Workshops wie *EventStorming* oder *Domain Storytelling*, um gemeinsam zu verstehen, was wirklich relevant ist.
- Verwende Werkzeuge wie das *Bounded Context Design Canvas*, um fachliche und technische Grenzen bewusst zu ziehen.

Don't

- Lass dich nicht von Hype-Technologien oder „Best Practices" treiben.
- Schneide keine Bounded Contexts entlang technischer Schichten (z. B. „Frontend-Kontext", „Backend-Kontext").
- Verwechsle „Code-Struktur" mit „Domänenstruktur".

7.2 Gemeinsame Sprache ernst nehmen

Die **Ubiquitous Language** ist der zentrale Anker zwischen Business und Technik. Sie entsteht in Gesprächen, nicht in Code-Generatoren.

Do

- Nutze die Fachsprache der Expert:innen konsequent im Code, in Tests und in der Dokumentation.
- Halte Begriffe sichtbar – etwa auf der Canvas oder in Glossaren.
- Hinterfrage doppeldeutige oder überladene Begriffe („Kunde“, „Antrag“, „Fall“ …).

Don’t

- Übersetze Fachbegriffe automatisch in technische Synonyme („CustomerDTO“, „ClientEntity“).
- Verwende mehrere konkurrierende Sprachen in einem Bounded Context.
- Lass Entscheidungen über Begriffe unkommentiert – sie sind Architekturentscheidungen.

7.3 Modelle kollaborativ entwickeln

Gute Modelle entstehen **im Dialog**, nicht in Einzelarbeit.

Do

- Plane regelmäßige, kurze Modellierungs-Sessions – nicht nur einmalige Workshops.
- Visualisiere: Whiteboard, Miro, Sticky Notes.

- Halte Modelle lebendig – sie sind Werkzeuge zum Denken, keine Artefakte zum Archivieren.
- Kombiniere *Big Picture EventStorming* für Überblick mit *Design Level EventStorming* für Detailarbeit.

Don't

- Übergib Modelle als „fertig" von einer Rolle zur nächsten.
- Lass Meetings zu Vorträgen verkommen – jede:r soll kleben, schreiben, verschieben dürfen.
- Verstecke Modelle in PDF-Anlagen oder Tool-Silos.

7.4 Kleine, kohärente Grenzen ziehen

Ein Bounded Context ist so groß wie das, was in einer konsistenten Sprache beschrieben werden kann.

Do

- Schneide Kontexte entlang fachlicher Kohärenz – nicht entlang Teams, Technologien oder Datenbanken.
- Halte die **Kohäsion hoch** und die **Kopplung niedrig**.
- Prüfe bei zu vielen Regeln oder Begriffen, ob der Kontext zu groß geworden ist.

Don't

- Erzeuge „Megakontexte", die alles enthalten und viel generalisieren.
- Zentralisiere Datenmodelle („Geschäftspartner", „Master Data") ohne klaren Zweck.

- Kopple Kontexte über gemeinsame Datenbanken – nutze stattdessen *Published Language* oder *Events*.

7.5 Iterativ denken und lernen

DDD ist ein Lernprozess. Modelle reifen, Grenzen verschieben sich, Sprache verändert sich.

Do

- Arbeite in kleinen Evolutionen statt in Big-Bang-Refactorings.
- Nutze TDD, um Regeln schrittweise in Code zu überführen.
- Dokumentiere offene Fragen und Annahmen („Assumptions“, „Open Questions“) sichtbar auf der Canvas.
- Reflektiere regelmäßig: Stimmen unsere Modelle noch mit der Realität überein?

Don’t

- Suche nach „dem perfekten Modell sofort“ - die erste Iteration kann immer nur die Qualitätsgüte „sie waren stets sehr bemüht“ haben.
- Warte mit der Implementierung, bis alles klar scheint – Lernen passiert durch Tun.
- Sieh Refactoring als Fehlerkorrektur – es ist zentraler Bestandteil von Lernen.

7.6 Fachliche Resilienz gestalten

Software ist nie fehlerfrei – aber sie kann **resilient** sein, wenn Fachlogik bewusst entkoppelt wird.

Do

- Implementiere „Graceful Degradation“: Systeme sollen auch bei Teilausfällen wertvolle Antworten liefern.
- Entkoppel Subdomains über Domain Events und Policies statt synchrone Aufrufketten.
- Übe, wie sich Ausfälle oder verspätete Events auf Geschäftsprozesse auswirken.

Don’t

- Baue starre Abhängigkeiten zwischen Kontexten („Der ruft den, der ruft den“).
- Verstecke Fehler hinter technischen Retries – verstehe ihre fachliche Bedeutung.

7.7 Architekturen dem Problem anpassen

Muster sind Werkzeuge, keine Gesetze.

Do

- Wähle Architekturstile (Monolith, Hexagon, Microservice, Event-Driven) nach Kontext.
- In Core Domains – hohe Qualität, Inhouse-Entwicklung, testgetriebene Modelle.
- In Generic Domains – Standardsoftware, SaaS oder Low-Code reicht oft.

Don’t

- Verwende die hexagonale Architektur, weil sie „modern klingt“.

- Strebe Microservices an, wenn Teamgröße oder Fachlogik es nicht rechtfertigen.
- Missbrauche Patterns als Selbstzweck.

7.8 Verantwortung sichtbar machen

DDD ist soziotechnisch: Architektur und Organisation müssen zusammenpassen.

Do

- Aligniere Teams entlang von Bounded Contexts (Team Topologies: Stream-aligned Teams).
- Mache Upstream/Downstream-Beziehungen explizit (Context Map).
- Kläre Ownership, bevor du Schnittstellen definierst.

Don't

- Verstecke Team-Abhängigkeiten hinter technischen APIs.
- Erwarte, dass Architektur allein organisatorische Probleme löst.
- Ignoriere Kommunikationswege – sie prägen Software genauso wie Code.

7.9 Pragmatismus bewahren

DDD verlangt Disziplin – aber auch Augenmaß.

Do

- Nutze DDD dort, wo Komplexität und Wertschöpfung hoch sind (Core Domain).

- Bleibe leichtgewichtig in Supporting- und Generic-Bereichen.
- Verwende DDD-Vokabular als Denkwerkzeug, nicht als Label.

Don't

- Betreibe „Full DDD" überall.
- Diskutiere über Pattern-Orthodoxie statt über Geschäftsnutzen.
- Verwechsele *Komplexität erzeugen* mit *Komplexität beherrschen*.

7.10 Fazit

Domain-Driven Design ist kein Rezeptbuch, sondern ein kontinuierlicher Dialog zwischen Fachlichkeit und Technik. Wer die hier beschriebenen Do's und Don'ts beherzigt, schafft dafür den Boden: eine Sprache, die verbindet – Modelle, die Sinn stiften – und Architekturen, die Veränderung zulassen.

Am Ende gilt: **DDD ist kein Ziel, sondern eine Praxis.** Wer sie lebt, lernt, wie aus Modellen Wert entsteht – nicht durch Dogma, sondern durch Verständnis, Zusammenarbeit und stetige Verbesserung.

8 Quellen und Referenzen

DDD Crew auf GitHub: https://www.github.com/ddd-crew

Brandolini, Alberto (2015): Introducing Event Storming, Leanpub https://leanpub.com/introducing_eventstorming

Dahan, Udi: Clarified CQRS http://udidahan.com/2009/12/09/clarified-cqrs/

Domain Storytelling Homepage: http://www.domainstorytelling.org/

Evans, Eric (2003): Domain-driven Design, Addison Wesley

Evans, Eric (2016): Model Exploration Whirlpool https://domainlanguage.com/ddd/whirlpool/

Khononov, Vlad (2021): Learning Domain-Driven Design, O'Reilly

Manifesto for Agile Software Development https://agilemanifesto.org/

Patton, Jeff (2014): User Story Mapping: Discover the Whole Story, Build the Right Product, O'Reilly and Associates

Principles behind the Agile Manifesto https://agilemanifesto.org/principles.html

Stevens, Meyers, Constantine (1974): Structured Design https://dl.acm.org/doi/10.5555/1241515.1241533

Vaughn, Vernon (2013): Implementing Domain-driven Design, Addison Wesley

Notizen

Der Autor

Michael Plöd

Michael ist ein Fellow bei INNOQ. Seine derzeitigen Schwerpunkte sind Domain-driven Design, Team Topologies, sozio-technische Architekturen und die Transformation von IT-Organisationen in Richtung Collaboration und lose gekoppelte Teams. Michael ist Autor des Buches „Hands-on Domain-driven Design - by example" auf Leanpub, Übersetzer des Buches Team Topologies ins Deutsche für O'Reilly, Contributor / Committer für DDD-Crew auf GitHub und ein regelmäßiger Speaker auf nationalen und internationalen Konferenzen.