Java™ magazin
Java | Architektur | Software-Innovation

THREAT MODELING

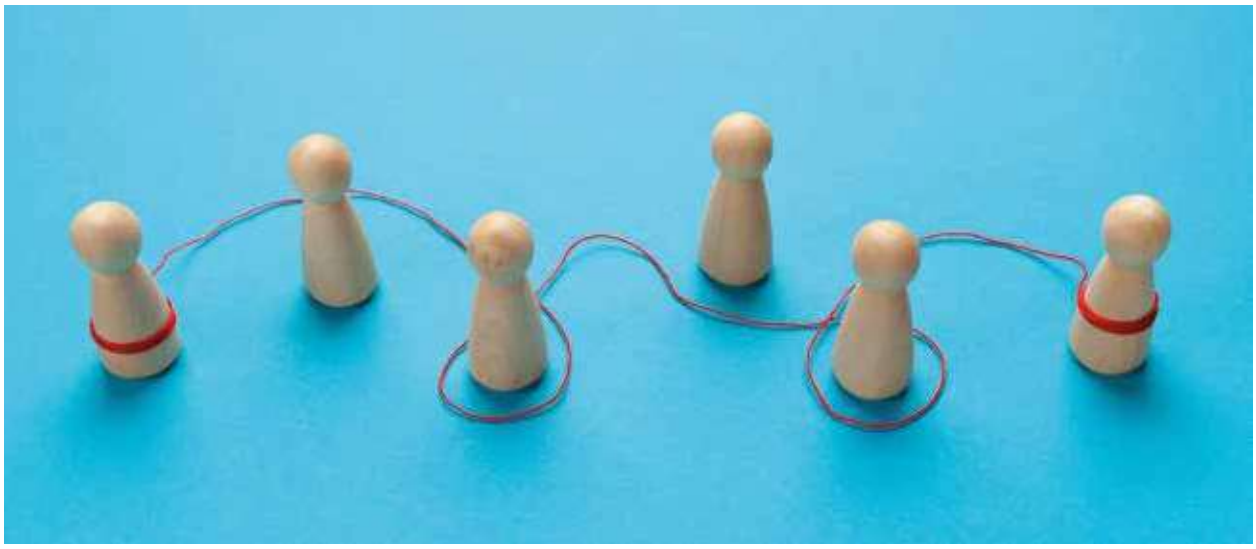
Keine Angst vorm Angreifer

Ausgabe 1.2026
Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 01

entwickler.de

Coverbild erstellt mit ChatGPT (DALL-E)



Modell und Sprache in DDD – Teil 2

Sprachphilosophie, Wittgenstein und DDD

Zwanzig Jahre nach der Veröffentlichung von Eric Evans Buch „Domain-Driven Design: Tackling Complexity in the Heart of Software“ ist DDD ein beliebter und bekannter Ansatz in der professionellen Softwareentwicklung. Trotzdem haben Teams in der Praxis oft mit der Komplexität von Sprache und Kommunikation zu kämpfen. Zudem ist das von Evans vorgeschlagene Konzept der Ubiquitous Language nicht einfach in die Tat umzusetzen –warum ist das so?

von Martin Nitsch

Domain-Driven Design (DDD) ist weder der erste noch der letzte Versuch, das komplexe Zusammenspiel von Organisation, Kommunikation und Design in der Softwareentwicklung zu durchdringen und zu verbessern.

Bereits 1968 zog der Amerikaner Melvin E. Conway, ganz im Geist dieser Jahre, die Schlussfolgerung, dass eine andere Managementphilosophie für das Design von Softwaresystemen gebraucht werde. Denn im Fazit des Artikels, der die Quelle für das bekannte Conway's Law ist, heißt es: „Ways must be found to reward design managers for keeping their organizations lean and flexible. There is need for a philosophy of system design management which is not based on the assumption that adding manpower simply adds to productivity. The development of such a philosophy promises to unearth basic questions about value of resources and techniques of communication which will need to be answered before our system-building technology can proceed with confidence“ [1].

Seine Forderung nach einer neuen Managementphilosophie stand vor dem Hintergrund der aufkommenden Softwarekrise, die im selben Jahr auch auf einer NATO-Konferenz im bayrischen Garmisch offiziell thematisiert wurde [2].

In den Jahrzehnten nach Conway entwickelten sich verschiedene Ansätze, die seine Forderung aufgriffen:

Artikelserie

- *In den 1970er-Jahren:* strukturierte Programmierung und Top-down-Design
- *In den 1980er-Jahren:* objektorientierte Programmierung und frühe iterative Entwicklungsmodelle
- *In den 1990er-Jahren:* die Patterns-Bewegung (initiiert durch die „Gang of Four")
- *In den späten 1990er-Jahren:* Extreme Programming (XP) und andere agile Methoden
- *In den frühen 2000er-Jahren:* Domain-Driven Design (DDD)

Sowohl XP wie auch DDD lassen sich als Versuche sehen, dem von Conway identifizierten Bedarf nach einer anderen Philosophie des Designs von Systemen zu begegnen. Im Vergleich zu Kent Beck, einem der Begründer von XP, der explizit auf die Kommunikationsstrukturen und Organisationsformen eingeht, fokussiert sich Eric Evans jedoch stärker auf die gemeinsame Sprache und Wissensrepräsentation. Evans nennt zwar Conway in seinem Buch nicht als Inspirationsquelle, greift er aber dessen zentrale Themen auf.

Um die Verbindung zu verstehen, lohnt ein Blick auf Conways Hauptargument. Das Herzstück seiner Überlegungen bildet die These, die später als Conway's Law bekannt wurde: „The basic thesis of this article is that organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations" [1]. Mit anderen Worten: Die Architektur der Software spiegelt die Kommunikationsstruktur der Organisation wider, die sie entwickelt.

Conway beobachtete zudem, dass große Systeme in ihrer Entwicklung oft chaotisch werden, was gemäß seiner These mit der Organisation selbst zu tun haben muss. Selbst ausgebildeter Physiker, führte Conway die entstehende Unordnung jedoch nicht auf naturwissenschaftliche Prinzipien wie Entropie zurück, sondern auf konkrete Managementprobleme:

- die Tendenz von Entwicklungsteams, aus Angst vor dem Scheitern zu viele Ressourcen einzufordern
- das Prestigedenken von Managern, das zu aufgeblähten Teams führt
- die Fehlannahme, dass mehr Personal automatisch zu höherer Produktivität führt

Nicht etwa unveränderliche Gesetze der Thermodynamik, sondern schlichtweg Fehlentscheidungen des Managements sind demnach Ursache für die Misere. Diese Perspektive wirkt deutlich weniger schicksalsergeben. Die Zukunft der Softwareentwicklung sollte, wenn es nach Conway geht, „lean and flexible" sein. Das konventionelle Management hatte für ihn ausgedient.

Domain-Driven Design knüpft an Conways Ideen an, indem es Sprache und Kommunikation im Kontext von Organisation adressiert. Das Konzept der Sprachbarrieren liefert eine Erklärung, warum die etablierten Kommunikationsstrukturen einer Organisation hinderlich für das Design eines Softwaresystems sein können. Die von Evans geforderte Ubiquitous Language zielt darauf ab, Sprachbarrieren zwischen Fachexperten und Entwicklern zu überwinden. Sie ist eine Antwort auf Conways Beobachtung, dass Kommunikationsstrukturen die Systemarchitektur bestimmen: Mit Hilfe der Ubiquitous Language soll gleichzeitig die Kommunikationsstruktur der Organisation verändert werden, um bessere Voraussetzungen für das Design der Software zu schaffen. Das Konzept der Bounded Contexts erkennt an, dass verschiedene Teile einer Organisation unterschiedliche Modelle und Sprachen verwenden. Statt diese Unterschiede zu ignorieren, will DDD sie explizit machen und ihre Beziehungen bewusst managen.

Ein wesentlicher Unterschied zwischen Conway und Evans liegt in ihren Perspektiven und Zielgruppen. Conway richtete seine Kritik gegen konventionelle Managementansätze und forderte eine Alternative zu diesen, ohne einen konstruktiven Vorschlag zu machen. Evans hingegen richtet sich vorrangig an Entwickler, Architekten und Designer. Sein Buch bietet eine Designphilosophie, die die Fachdomäne ins Zentrum stellt. Evans' Ansatz liest sich weniger kritisch gegenüber dem Management und ist mehr darauf bedacht, Praktiken zu etablieren, die auch in bestehenden Organisationsstrukturen funktionieren können. Eine Kombination von DDD mit agilen Managementansätzen wie Scrum kann jedoch als umfassendere Antwort auf Conways ursprüngliche Forderung betrachtet werden, die sowohl die Design- als auch die Managementaspekte komplexer Softwareprojekte adressiert.

Der Designprozess im Domain-Driven Design

Zunächst einmal ist der Designprozess im Domain-Driven Design als Kreislauf der Kreativität zu sehen. Denn DDD beschreibt einen iterativen, kreativen Erkenntnisprozess, der nicht linear verläuft, sondern sich in Zyklen entwickelt. Die Methodik beginnt mit einer explorativen Phase und einer strategischen Analyse der Fachdomäne. Diese geht in eine Phase der Destillation über. Das initiale Modell wird iterativ verfeinert.

Eine Besonderheit des Ansatzes ist, dass die Verfeinerung des Modells gleichzeitig auf konzeptioneller Ebene und auf Ebene des Quellcodes durch Refactoring stattfindet. Dadurch ergibt sich ein Wechselspiel und ein Kreislauf im Designprozess. Evans betont explizit, dass keine Trennung zwischen einem abstrakten Analysemodell und einem technischen Implementierungsmodell existieren sollte [3].

Keine Schnapsidee – Destillation von Modellen

Die Methodik von DDD dient maßgeblich dazu, das vielfältige und oft implizite Wissen der Domänenexperten systematisch zu „destillieren". Evans wählt diesen Begriff bewusst in Anlehnung an chemische Trennverfahren, bei denen Substanzen zur Gewinnung reiner Essenzen separiert und veredelt werden: „Distillation is the process of separating the components of a mixture to extract the

essence in a form that makes it more valuable and useful. A model is a distillation of knowledge“ [3], [4]. Der Destillationsprozess soll zwei zentrale Fragen klären:

1. Welches Wissen sollte Teil des Designs werden? (Trennung)
2. Wie kann dieses Wissen in eine Form gebracht werden, die für das Softwaredesign optimal nutzbar ist? (Veredelung)

Bevor Wissen destilliert werden kann, muss es zunächst durch gezielte Exploration gesammelt werden.

Die Destillation findet im DDD auf unterschiedlichen Ebenen statt. Strategische Destillation zielt darauf ab, innerhalb der Domäne den Kernbereich (Core Domain) von generischen Subdomänen zu trennen. Diese Trennung ist eine architektonische Entscheidung von langfristiger Bedeutung, da sie später nur schwer zu revidieren ist. Bei generischen Subdomänen kann zudem eine Make-or-Buy-Entscheidung relevant werden. Innerhalb der Core Domain werden dann durch konzeptuelle Destillation fachliche Konzepte weiter destilliert, um ein „Deep Model“ zu gewinnen: „A deep model distills the most essential aspects of a domain into simple elements that can be combined to solve the important problems of the application“ [3].

Die Methode zur Destillation eines tiefen Modells umfasst das Explizieren impliziter Konzepte und kontinuierliches Refactoring des Codedesigns. Diese beiden Aktivitäten sind eng miteinander verflochten und beeinflussen sich gegenseitig. Refactoring dient dabei nicht nur der technischen Verbesserung, sondern ist ein erkenntnisgeleiteter Prozess, der tieferes Domänenwissen fördern soll.

Es gibt eine aufschlussreiche Parallele zu „Knowledge Distillation“ im Bereich der künstlichen Intelligenz. Hinton et al. beschreiben im Paper „Distilling the Knowledge in a Neural Network“ (2015) einen Prozess, bei dem Wissen aus großen, rechenintensiven „Expertenmodellen“ in kleinere, anwendungstaugliche Modelle übertragen wird: „Once the cumbersome [expert] model has been trained, we can then use a different kind of training, which we call ‚distillation‘ to transfer the knowledge from the cumbersome model to a small model that is more suitable for deployment“ [5]. Aus dem Wissen eines oder mehrerer Expertenmodelle wird das Wissen für ein kleineres Modell destilliert, das für praktische Zwecke besser geeignet ist [5].

In beiden Kontexten geht es um die Übertragung und Verdichtung von Wissen. Während jedoch die KI-Destillation auf mathematisch definierten Verfahren (mittels Cross-Entropie als Kostenfunktion) basiert, ist die Modelldestillation im DDD ein kreativer, kommunikativer Prozess, bei dem die Fähigkeiten der Beteiligten entscheidend sind.

Soft Skills für Software

Evans erklärt seine Methodik anhand einiger ausführlicher didaktischer Beispiele und Anekdoten, die der Veranschaulichung dienen. Dennoch bleiben die Ausführungen eher skizzenhaft und aphoristisch. Ein Grund mag sein, dass praktische Erfahrung sich nicht leicht versprachlichen lässt. Außerdem findet thematisch ein Übergang zu Soft Skills und Kultur statt. Das sind Themen, die keineswegs spezifisch für DDD sind und schnell den Rahmen sprengen würden. Dennoch ist es hilfreich, sich die zentralen Arbeitsweisen von Evans Vorgehen bewusst zu machen.

Die Destillation im DDD ist ein kontinuierlicher kommunikativer Verstehensprozess, der einen intensiven Dialog zwischen Fachexperten und Softwareentwicklern erfordert. Dieser Prozess verlangt spezifische Fähigkeiten, die über rein technische Kompetenzen hinausgehen.

Hör zu! – die Kunst des achtsamen Zuhörens

Evans betont die fundamentale Bedeutung des achtsamen Zuhörens [3]. Entwickler müssen feine Unterschiede und Nuancen in der Sprache der Domänenexperten erkennen und deren Bedeutung für das Design entschlüsseln können. Die Fertigkeit erfordert nicht nur Konzentration, sondern auch linguistisches Verständnis und die Fähigkeit, Ambiguitäten zu erkennen.

Aber wie lassen sich Zen und Achtsamkeit im Trubel eines vollen Meetingkalenders bewahren? Auf die praktischen Herausforderungen des Zuhörens im modernen Arbeitsalltag geht Evans nicht tiefer ein, aber sie sind zwingend zu reflektieren, wenn das Unterfangen in der Praxis gelingen soll.

There’s too much confusion, I can’t get no relief

Für eine erfolgreiche Destillation ist es essenziell, Ungereimtheiten sowohl im Design der Software als auch in den Aussagen der Experten zu identifizieren und offen zu thematisieren [3]. Domänenexperten verfügen über spezialisiertes Wissen, sind aber nicht unfehlbar. Widersprüche müssen gezielt thematisiert werden, um Verwirrung im Designprozess zu vermeiden. Doch wie viel Verständnis für Hermeneutik und Interpretation hat das Management, das mit Druck auf Kosten und Ergebnisse schaut? Diese Aufgabe erfordert also nicht nur logisches Denken, sondern auch diplomatisches Geschick und die Fähigkeit, konstruktive Kritik zu formulieren.

Lies doch mal! – keine Zeit

Evans empfiehlt ausdrücklich die Integration von Fachliteratur in den Entwicklungsprozess. In einem Bereich wie der Softwareentwicklung, in dem Forschung und Praxis eng verknüpft sind (wie das Aufkommen von KI unterstreicht), ist es unerlässlich, sich mit vorhandenem Wissen auseinanderzusetzen, anstatt bei null zu beginnen. Das erfordert jedoch eine Lernkultur, die Wissenserwerb explizit als Teil der Arbeit anerkennt und entsprechende Zeiträume dafür vorsieht.

Ich hatte immer den Verdacht, dass zu wenig gelesen wird. Vielleicht hat das banale Gründe. Nicht jeder liest gern. Alle fühlen sich verpflichtet, bereits Experten zu sein und ausgereifte Ergebnisse zu produzieren, auch wenn sie dann am Impostor-Syndrom leiden müssen.

Literaturstudien kosten Zeit und sind scheinbar ein Eingeständnis der eigenen Unwissenheit. Lernen steht nicht im Vordergrund der Arbeit: Wer hat schon die Ruhe im Arbeitsalltag, es sich mit Buch und Kaffee im Ohrensessel gemütlich zu machen? Lesen ist Privatvergnügen oder berufliche Weiterbildung, die bitte abseits des Arbeitsalltags stattfindet.

So bleibt das Lernen allerdings oft unfruchtbar. Und Evans hat recht: Wenn wir in der Praxis informierter sein wollen, sollten wir mehr und im Kontext unserer Arbeit lesen und lernen. [3]

Anything goes? – mutiges Experimentieren

Ausprobiert wird bekanntlich viel und mitunter auch etwas dabei gelernt, aber gezielte, erkenntnisorientierte Experimente sind selten, weil ungleich schwerer in der Praxis zu gestalten. Du brauchst eine passende Lernkultur. Irrtümer und Fehler dürfen nicht abgestraft werden. Die Reflexion des Gelernten ist entscheidend. Doch diese erfordert Zeit und Ruhe sowie Offenheit. Nicht zuletzt braucht es fundiertes Methodenverständnis, um Experimente erfolgreich durchzuführen und ihre Ergebnisse zu interpretieren. Das alles fällt vielen Unternehmen bekanntlich eher schwer.

Evans' Bewertung von Trial and Error ist nicht falsch, liest sich vor diesem Hintergrund aber recht optimistisch: „All these changes of direction are not just thrashing. Each change embeds deeper insight into the model. […] There is really no choice, anyway. Experimentation is the way to find out what works and what doesn't."

Die Destillation von Domänenwissen stellt eine zentrale Aktivität im Domain-Driven Design dar. Sie umfasst sowohl technische als auch kommunikative Aspekte und verlangt ein kontinuierliches Wechselspiel zwischen Exploration, Modellierung und Implementierung. Neben den Fähigkeiten der Beteiligten braucht es eine unterstützende Organisationskultur, die Zeit für Lernen, Experimentieren und kontinuierliche Verbesserung einräumt. Nur so kann das volle Potenzial des Domain-Driven Designs ausgeschöpft werden.

Sprachphilosophische Werkzeugkiste

Die bisher dargestellten DDD-Methoden bieten eine solide Grundlage, behandeln die Explizierung impliziter Konzepte jedoch nur oberflächlich. Die Sprachphilosophie bietet eine reichhaltige „Werkzeugkiste" von Konzepten mit direkten Parallelen zum Softwaredesign.

Eine philosophische Verbindung zu DDD hatte bereits 2009 David Laribee in einem Artikel für Microsoft hergestellt [6]: „You can relate Plato's theory of Forms to DDD. Much of its guidance helps us get closer to the ideal model over time. The path to the Form you want to describe with your code is scattered about in the minds of domain experts, the desires of stakeholders, and the requirements of industries in which we're working."

Für Laribee gleicht die Softwareentwicklung Platons Höhlengleichnis: Entwickler streben nach einem idealen Abbild der Realität, sind aber durch Zeit, Kosten und Programmiersprachen gefangen. Ein perfektes Modell sei unmöglich, aber schrittweise Approximation an die „wahre Form" der Applikation möglich: „The software you create is not the true model. It is only a manifestation – a shadow, if you will – of the application Form you set out to achieve."

So inspirierend die Verbindung von Philosophie und DDD ist, die Laribee aufzeigt, hat sein Rückgriff auf Platons Philosophie auch seine Schwachpunkte. Nach Platons Ideenlehre sind die Formen kontextunabhängige, universelle Wahrheiten; Begriffe und Modelle sind ihre schemenhaften Abbilder der Realität. DDD macht mit Bounded Contexts hingegen die Kontextgebundenheit von Bedeutung zum Prinzip.

Ausgangsbasis für die Destillation ist der Sprachgebrauch der Domänenexperten: Was ein Begriff wie „Kunde" bedeutet, entscheidet der konkrete Gebrauch im jeweiligen Kontext. Statt einer Abbildtheorie der Bedeutung, die Begriffe und Modelle als Abbilder universeller Wahrheiten ansieht, ist eine Gebrauchstheorie der Bedeutung in der Praxis sinnvoller – Anforderungen verändern sich, Bedeutungen werden fortlaufend ausgehandelt. Es gibt nicht ein einzig wahres, perfektes Modell, sondern es ist zwischen verschiedenen Modellen mit praktischen Vor- und Nachteilen abzuwägen. Viele Domänenkonzepte bilden Netze überlappender Ähnlichkeiten und lassen sich nicht durch starre Definitionen unveränderlicher Formen beschreiben. Auch wenn Destillation nach den essenziellen Bestandteilen und klaren Formen des Designs sucht, sind die zentralen Konzepte der Domäne selbst oft nicht durch notwendige und hinreichende Bedingungen definierbar.

Trotz dieser Schwachpunkte in Laribees Beispiel halte ich die Verbindung zur Philosophie für nicht unberechtigt. Ich würde jedoch argumentieren, dass es andere Beiträge gibt, die einen engeren Bezug zu DDD haben.

Wittgensteins Sprachspiele

Viele der Einsichten und Praktiken, für die DDD argumentiert, ähneln meines Erachtens in bemerkenswerter Weise den Gedanken des Philosophen Ludwig Wittgenstein (1889-1951). Das Konzept der Sprachbarrieren erinnert zum Beispiel an eine berühmte Aussage aus Wittgensteins „Tractatus Logico-Philosophicus": „Die Grenzen meiner Sprache sind die Grenzen meiner Welt." Im Kontext des Software-Engineerings lassen sich drei Arten von Grenzen differenzieren: erstens technische Limitierungen, wie etwa die Beschränkungen einer Programmiersprache und ihres Paradigmas. In „The Pragmatic Engineer" wird etwa festgestellt, dass Programmiersprachen beeinflussen, wie wir über ein Problem nachdenken, das wir lösen wollen [4]. Zweitens gibt es interpersonelle Sprachbarrieren, Kommunikationshindernisse zwischen Personen, wie zum Beispiel verschiedener fachlicher oder sprachlicher Hintergrund. Und drittens stoßen wir an konzeptuelle Grenzen, d. h. Grenzen der eigenen Sprache im Verhältnis zur Welt, die als Koordinatensystem für unser Denken wirken.

Abb. 1: Hase oder Ente?

Wichtige Beiträge, die Wittgenstein für die Sprachphilosophie geliefert hat, sind das Konzept der Sprachspiele und seine Gebrauchstheorie der Bedeutung. Sprachspiele sind eingebettet in soziale Praktiken und Lebensformen. Sprachspiele ergeben sich aus „Spielzügen" innerhalb eines bestimmten Gebrauchs von Sprache.

Den Kerngedanken seiner Gebrauchstheorie der Bedeutung hat Wittgenstein in seinem Hauptwerk „Philosophische Untersuchungen" so erklärt: „Man kann für eine große Klasse von Fällen die Benutzung des Wortes ‚Bedeutung' – wenn auch nicht für alle Fälle seiner Benutzung – dieses Wort so erklären: Die Bedeutung eines Wortes ist sein Gebrauch in der Sprache" [7].

Auch in der Softwareentwicklung haben wir es mit verschiedenen Sprachspielen zu tun und die Gebrauchstheorie der Bedeutung erklärt zum Beispiel, warum Pair Programming zu einer gemeinsamen Sprache im Team beiträgt und eine organisatorische Trennung von Fachteam und Entwicklungsteam die Entwicklung einer gemeinsamen Sprache hemmt. Eine gemeinsame Sprache entsteht nur durch gemeinsame Praktiken. Wer eine gemeinsame Sprache fördern will, muss gemeinsame Praktiken fördern.

Eine weitere, interessante Verbindung besteht zum Problem des Aspektwechsels, das Wittgenstein anhand des Beispiels des Hase-Ente-Kopfes behandelt (**Abb. 1**). „Man kann ihn als Hasenkopf, oder als Entenkopf sehen. Und ich muss zwischen dem ‚stetigen Sehen' eines Aspekts und dem ‚Aufleuchten' eines Aspekts unterscheiden" [7].

Wittgensteins H-E-Kopf illustriert, wie Wahrnehmung und Interpretation kontextabhängig sind – ein Prinzip, das beim Duck Typing augenfällig wird. Scherzhaft heißt es zum Duck Typing: „If it walks like a duck and quacks like a duck, then it must be a duck." Die Formulierung bringt ähnlich gut auf den Punkt, was wir als Phänomen beim H-E-Kopf beobachten. Während die Hasen-Ente zeigt, wie dasselbe Bild im Wechsel unterschiedlich wahrgenommen werden kann, wechselt beim Duck Typing der Wert einer Variable in verschiedenen Kontexten seinen Typ. In beiden Fällen finden Aspektwechsel statt. Diese Parallele ist besonders relevant für DDD, das mit Bounded Contexts einen Rahmen schaffen möchte, um die kontextabhängige Natur von Begriffen und Konzepten explizit zu managen – ähnlich wie einige Programmiersprachen versuchen, das Duck Typing durch strenge Typisierung zu begrenzen und durch Interfaces beherrschbarer zu machen.

Wittgenstein argumentierte außerdem, dass viele Begriffe nicht durch präzise Definitionen mit notwendigen und hinreichenden Bedingungen erfasst werden können, sondern nur durch ein Netzwerk überlappender Ähnlichkeiten – etwa wie Ähnlichkeiten zwischen Familienmitgliedern – beschrieben werden können: „Betrachte z. B. einmal die Vorgänge, die wir ‚Spiele' nennen. Ich meine Brettspiele, Kartenspiele, Ballspiele, Kampfspiele usw. Was ist allen diesen gemeinsam? – Sag nicht: ‚Es muß ihnen etwas gemeinsam sein, sonst hießen sie nicht Spiele' – sondern schau, ob ihnen allen etwas gemeinsam ist. – Denn wenn du sie anschaust, wirst du zwar nicht etwas sehen, was allen gemeinsam wäre, aber du wirst Ähnlichkeiten, Verwandtschaften, sehen, und zwar eine ganze Reihe" [7].

Beispiel: Eine Bank nutzt für ihre KYC-Prozesse den Begriff „Geschäftspartner", um damit verschiedene Entitäten zu benennen, die im KYC-Prozess eine Rolle spielen (Unternehmen, Personen, Schiffe …). Eine Softwarelösung hat versucht, das Konzept als eine Klasse zu modellieren. Es gibt jedoch keinen gemeinsamen Kern von Eigenschaften, die alle Geschäftspartner gemeinsam haben, die im Rahmen des KYC für die Bank eine Rolle spielen. Der Ansatz begünstigte eine verworrene Businesslogik innerhalb der Anwendung.

Neben den logischen Parallelen seiner Philosophie zum Domain-Driven Design gibt es auch eine eher zufällige, biografische Verbindung. Bevor Wittgenstein einer der wichtigsten Sprachphilosophen des 20. Jahrhunderts wurde, hatte er bereits eine Ingenieurlaufbahn hinter sich, während der er Propellerantriebe für Flugzeuge entwickelt hatte. Später lehrte er am Trinity College in Cambridge, wo auch Alan Turing zu seinen Hörern zählte.

Carnaps Methode der Explikation

Rudolf Carnap (1891–1970), ein weiterer bedeutender Sprachphilosoph, der bei Wuppertal geboren wurde, bietet mit seiner Methode der Explikation ein wertvolles Werkzeug für Domain-Driven Design. Was versteht Carnap unter Explikation? In Kurzfassung: Die Explikation ersetzt einen unpräzisen oder vagen Begriff durch einen präzisen, logisch wohldefinierten Begriff, das Explikat. Nach Carnap sollte ein gutes Explikat vier Kriterien erfüllen:

- Es ist exakt (präzise definiert).
- Es ist einfach (so unkompliziert wie möglich).
- Es ist fruchtbar (ermöglicht weitere Erkenntnisse).
- Es ist dem ursprünglichen Begriff hinreichend ähnlich und damit anschlussfähig.

In seiner Schrift „Meaning and Necessity“ veranschaulicht Carnap diese Methode anhand einer Analyse des Wahrheitsbegriffs [8]. Ein Beispiel mit Praxisbezug zum Software-Engineering ist die Explikation des Begriffs des wirtschaftlichen Eigentümers aus der EU-AML-Direktive 6 zum Zweck der Entwicklung einer digitalen Lösung. Der Begriff des wirtschaftlichen Eigentümers wird aus den Gesetzestexten ins Domänenmodell übernommen und als Relation von Pfaden zwischen zwei Knoten innerhalb eines Property-Graphen modelliert, der dem fachlichen Konzept einer Beteiligungsstruktur entsprechen soll. Die Ermittlung des wirtschaftlichen Eigentümers lässt sich so explizieren als Berechnung dieser Relation von Pfaden anhand von speziellen Graphen-Algorithmen.

Fazit

Um sein Wissen und seine Skills in DDD zu vertiefen, kann sich der Blick über den Tellerrand hinaus lohnen. Das Wissen liegt nicht auf der Straße, so viel ist sicher, aber die gut gefütterte KI kann zur Rettung kommen, um sich zunächst überblicksartig aufzuschlauen. Die englische Redewendung „world of knowledge at your fingertips“ unterstreicht in meinen Augen treffend: Es war noch nie so einfach. Der kurze Ausflug in die Philosophie sollte illustrieren, dass es oft überraschende Verbindungen zwischen verschiedenen Wissensgebieten gibt und die sprachlich-kommunikative Dimension von DDD nicht unterschätzt werden sollte.

Martin Nitsch berät seit mehreren Jahren als Architekt und Software-Engineer Kunden bei der Entwicklung von komplexen Softwaresystemen. Optimierung von Risiken, Time-to-Market und Mehrwert für den Nutzer stehen für ihn im Fokus. Er hat außerdem eine akademische Ausbildung in Mathematik und Philosophie. In seiner Freizeit begeistert er sich für das Segeln.

Links & Literatur

[1] Conway, M. E.: „How Do Committees Invent?“; Datamation, 1968

[2] Naur, P.; Randell, B.: „Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968“; NATO, 1969

[3] Evans, E.: „Domain-Driven Design. Tackling Complexity in the Heart of Software“; Addison-Wesley, 2004

[4] Thomas, D.; Hunt, A.: „The Pragmatic Programmer. Your journey to mastery“; Addison-Wesley, 2020

[5] Hinton, G.: „Distilling Knowledge in a Neural Network“; 2015

[6] Laribee, D.: „Best Practice – An Introduction To Domain-Driven Design“: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design [Zugriff am 18 Juli 2025].

[7] Wittgenstein, L.: „Philosophische Untersuchungen“; Suhrkamp, 1980

[8] Carnap, R.: „Meaning and Necessity“; University of Chicago Press, 1947