Neue Serie: Continuous Architecture S. 37

Neue Serie: Modell und Sprache in DDD S. 42

Neue Serie: Transaktionskosten S. 60

Java™ magazin

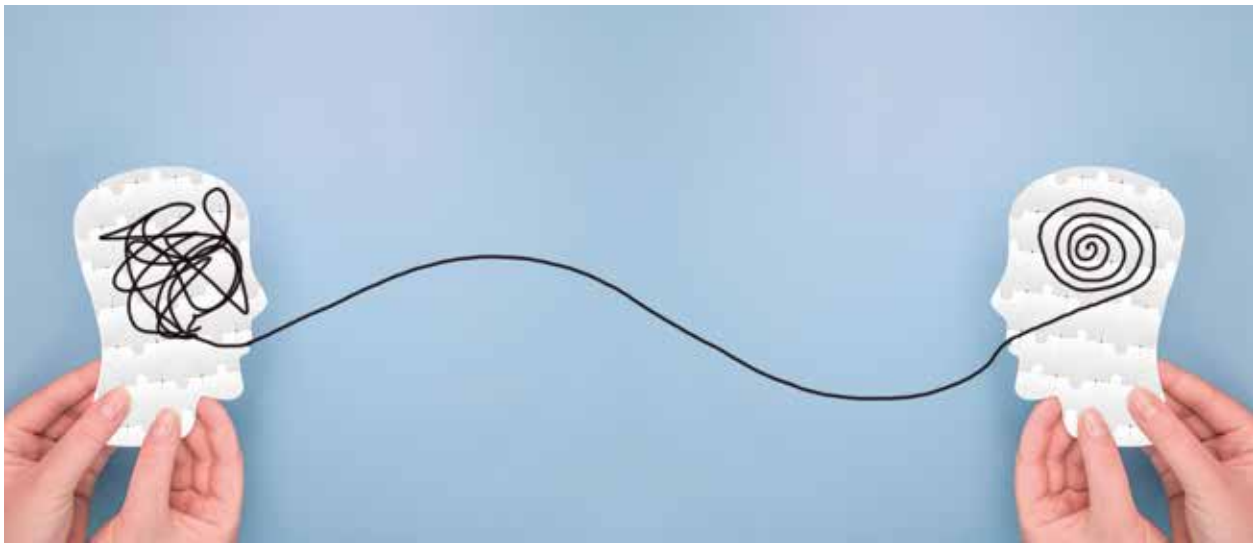
Java | Architektur | Software-Innovation

AGILE

Ein Sehnsuchtsort

entwickler.de

Ausgabe 12.2024



Modell und Sprache in DDD – Teil 1

Die Tücken der Terminologie

Zwanzig Jahre nach der Veröffentlichung von Eric Evans Buch „Domain-Driven Design: Tackling Complexity in the Heart of Software“ ist Domain-Driven Design 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, und das von Evans vorgeschlagene Konzept der Ubiquitous Language ist nicht einfach in die Tat umzusetzen. Warum ist das so?

von Martin Nitsch

Domain-Driven Design (DDD) ist ein methodischer Ansatz für das Softwaredesign, der das Ziel hat, die Komplexität der Software beherrschbar zu machen. Für die Softwareentwicklung spielen verschiedene Arten von Komplexität eine Rolle. DDD legt den Fokus vor allem auf Domain Complexity. Diese liegt in der Aufgabe oder dem fachlichen Problem begründet, das die Software lösen soll, und ist zum Beispiel von technischer Komplexität zu unterscheiden, die aus der Wahl der eingesetzten Technologien resultiert.

Zwei verschiedene Erkenntnispfade sind für DDD von zentraler Bedeutung. Der erste beruft sich auf die lang bestehende und oft bewährte Praxis, Modelle für praktische und wissenschaftliche Zwecke zu verwenden. So ist beispielsweise eine Karte ein Modell, das zur Navigation verwendet wird. Ein Modell hat erklärende und vorhersagende Funktionen und dient oft auch

Artikelserie

Teil 1: Die Tücken der Terminologie
Teil 2: Sprachphilosophie, Wittgenstein und DDD
Teil 3: XP vs. DDD

einem praktischen Zweck. Modelle lassen sich iterativ und inkrementell verbessern.

Der zweite ist ein Pfad der Reflexion über Sprache und Kommunikation und verbindet DDD mit Erkenntnissen zu praktischen Problemen der Kommunikation und des Sprachgebrauchs. Wir bewegen uns hier auf den Gebieten der Kommunikationspsychologie, Linguistik, Informationstheorie und Philosophie. Ein Ausgangspunkt ist die Idee, dass jede Kommunikation mit Transaktionskosten verbunden ist und eine gewisse Wahrscheinlichkeit hat, „noise" oder Missverständnisse zu produzieren. Das Modell der vier Seiten einer Nachricht oder Sprachspiele wie „Stille Post" erklären verschiedene Aspekte von Missverständnissen.

Wir haben es also inhaltlich mit zwei unterschiedlichen Denkansätzen zu tun, die Evans beide zu berücksichtigen versucht, wenn er DDD als effektiven Ansatz zur Bewältigung von Komplexität in der Softwareentwicklung vorschlägt. Beide sind in DDD eng miteinander verwoben. Evans legt allerdings den Schwerpunkt seines Buchs auf die Erstellung eines Modells der Domäne. Reflexion über Kommunikation und Sprache findet durch sein Konzept der Ubiquitous Language Eingang, spielt insgesamt aber eine untergeordnete Rolle.

Das Domänenmodell

Evans beginnt sein Buch mit einer Vision des Domänenmodells und dessen Bedeutung für das Softwaredesign. Seine Definition des Modells lautet: „A rigorously organized and selective abstraction of expert knowledge" [1]. Dessen Bedeutung fasst er mit mehreren Kernaussagen zusammen:

- „The model and the heart of design shape each other."
- „The model is the backbone of a language used by all team members."
- „The model is distilled knowledge."
- „A domain model can be the core of a common language for a software project." [1]

Im Fall einer komplexen Domäne soll das Modell helfen, das Design der Software zu verbessern und die Komplexität der Domäne beherrschbar zu machen. Das Softwaredesign soll im Zusammenspiel mit einem Modell stattfinden. Die Kommunikation im Designprozess soll sich auf das Modell berufen können. Das Modell soll eine effektive Kommunikation unterstützen und es ermöglichen, eine gemeinsame Sprache zu finden.

Das Argument der Sprachbarrieren

Um die Nutzung von DDD zu motivieren, argumentiert Evans in einem ersten Schritt, dass angesichts einer komplexen Fachlichkeit Sprachbarrieren im Team vorprogrammiert seien. Sein Argument des „Linguistic Divide" lässt sich kurz so zusammenfassen: Softwareentwicklung ist ein komplexer Prozess, an dem verschiedene Experten einer Organisation beteiligt sind. Der Erfolg hängt in hohem Maß von einer effektiven Zusammenarbeit ab. Doch leider gibt es in vielen Organisationen Sprachbarrieren, die eine effektive Zusammenarbeit gefährden, da sie das gegenseitige Verständnis erschweren.

In einer Organisation findet Arbeitsteilung statt. In der Regel gibt es eine Aufteilung in zwei Gruppen von Spezialisten, nämlich in Experten mit Fachwissen auf der einen und Experten mit technischem Wissen auf der anderen Seite. Spezialwissen geht oft mit Fachausdrücken und einer besonderen Terminologie einher. Menschen aus anderen Gruppen sind mit diesen In-Group-Dialekten nicht vertraut. Infolgedessen besteht zwischen diesen Gruppen eine Sprachbarriere, die viel Raum für Missverständnisse lässt.

Argumente gegen Übersetzungen

Ein zweiter Schritt in Evans' Argumentation stützt sich auf ein Konzept von Übersetzung, das eine Kommunikationsform zwischen Experten beschreiben soll. Es dient der Unterscheidung zwischen zwei Möglichkeiten, wie Experten in einer Organisation kommunizieren können:

- per Übersetzung zwischen ihren fragmentierten, spezialisierten Fachsprachen oder
- in einer gemeinsamen Sprache, die sie zusammen etablieren und praktizieren

Diese beiden Formen der Kommunikation hält Evans nicht für gleich effektiv, um ein gegenseitiges Verständnis zu erzielen.

Typische Beispiele für Übersetzungen im Alltagsverständnis sind die literarischer Texte oder die Arbeit von Dolmetschern und Übersetzungen in Untertiteln. Nehmen wir das Beispiel einer deutschen Übersetzung von „À la recherche du temps perdu" von Marcel Proust. Da ich der französischen Sprache nicht mächtig genug bin, kann ich den von Proust selbst verfassten französischen Originaltext nicht lesen. Ich kann beschließen, eine Übersetzung des Buches zu kaufen, die von einem Literaturübersetzer angefertigt wurde, der über die entsprechende sprachliche Kompetenz verfügt. Oder ich kann beschließen, mein Französisch zu vertiefen. Aber ich kann nicht davon ausgehen, von heute auf morgen Proust im französischen Wortlaut selbst lesen zu können. Eine gute Übersetzung, die dem ursprünglichen Sinn des Textes so nahe zu kommen versucht wie nur möglich, hilft mir, den Originaltext sinngemäß zu verstehen. Doch die Lektüre von Übersetzungen wird mich nicht mit der Zeit befähigen, die von Proust selbst verfassten französischen Originaltexte zu lesen und zu verstehen. Und klar, wir alle kennen schlechte Übersetzungen in Untertiteln von Filmen und Serien.

Auch Experten, die sich einer Sprachbarriere gegenübersehen, werden laut Evans versuchen, einander zu „übersetzen". Ein Spezialist sagt etwas. Der andere versucht, zu verstehen, was der Spezialist gesagt hat, indem er den Wortbeitrag für sich „übersetzt". Der Austausch basiert jedoch gerade nicht auf der sprachlichen Kompetenz des Übersetzers, sondern ist in hohem Maße auf

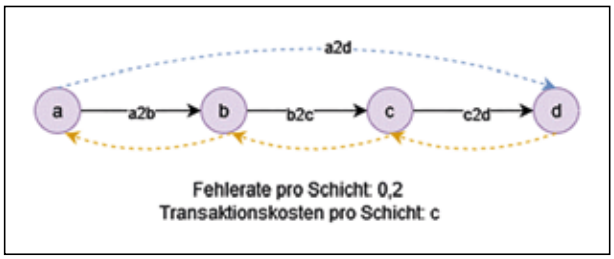


Abb. 1: Fehlerquoten in Übersetzungen

Annahmen, Rückfragen und raten angewiesen. Für eine gute Übersetzung fehlt den Beteiligten wichtiges Hintergrundwissen, das sich nur schwer explizieren lässt. Niemand hat eine klare Vorstellung davon, was kommuniziert werden muss, um ein gegenseitiges Verständnis zu erreichen. Überall lauern „unknown Unknowns".

Die Beispiele für Übersetzung, die zu unserem Alltagsbegriff passen, halten wir insgesamt für unproblematisch. Im Großen und Ganzen können wir auf die sprachliche Kompetenz von professionellen Übersetzern und Dolmetschern vertrauen. Schlechte Übersetzungen von Untertiteln stören uns hin und wieder vielleicht ein wenig, aber viele Leute finden sie auch amüsant. Exakte Übersetzungen sind nicht immer möglich, aber bei der alltäglichen Kommunikation kommt es nicht auf eine große Genauigkeit an.

Im Gegensatz dazu stuft Evans die Übersetzungen zwischen Experten im Designprozess von Software als ein ernst zu nehmendes Problem ein. Solche Übersetzungen beruhen in der Regel nicht in gleichem Maße auf sprachlicher Kompetenz wie in den genannten Fällen. Gleichzeitig erfordert die Arbeit eine hohe Genauigkeit bei der Spezifikation. Mit Missverständnissen können Risiken und Kosten verbunden sein.

Im klassischen Ansatz wird versucht, dieses Problem zu lösen, indem kompetente Übersetzer gefunden werden. Business Analysts oder Requirements Engineers sollen zwischen den beiden Expertengruppen vermitteln. Evans hält diese Entscheidung für bedenklich. Weil es zu wenige fähige Übersetzer gebe, die sowohl das fachliche als auch das technische Wissen haben, sieht er hier den eklatanten Nachteil eines „Information Bottleneck". Die Übersetzungen sind oft ungenau und ziehen deshalb Missverständnisse nach sich. Wie wir sehen werden, können sich Übersetzungsfehler sogar ansammeln, wenn im Prozess gezielt Übersetzer eingesetzt werden.

Evans führt seine Argumente nicht im Detail aus, was einigen Raum für Interpretationen offenlässt. Ein wesentlicher Aspekt scheint jedoch zu sein, dass Übersetzungen einer gemeinsamen Sprache und einem geteilten Verständnis von Fachlichkeit, Design und Software entgegenwirken.

Zum Beispiel schreibt er an einer Stelle von „translation muddles model concepts" und etwas später: „The effort of translation prevents the interplay of knowledge and ideas that lead to deep model insights" [1]. Zugleich aber bilden für ihn die Begriffe des Modells das Rückgrat einer gemeinsamen Sprache. Eine vollständige Version eines seiner Argumente könnte also lauten:

1. „Translation muddles model concepts",
2. „Model concepts are the backbone of a common language",
3. [therefore, translation hinders the creation of a common language] (Rekonstruktion des Autors)

Doch wie lassen sich dann die Begriffe des Modells, die das Rückgrat einer gemeinsamen Sprache bilden, klären und definieren, wenn man „lost in translation" ist und noch keine gemeinsame Sprache gefunden hat?

Eine andere Art der Übersetzung, auf die Evans nicht so stark eingeht, ist die von Spezifikationen in Quellcode. Der Gedanke an eine solche Übersetzung schwingt sicherlich in seiner Beobachtung mit, dass die „Terminology of day-to-day discussions is disconnected from the terminology embedded in the code". Solange Spezifikation und Quellcode in verschiedenen Sprachen formuliert werden, braucht es eine Übersetzung. Gegen diese und ihre Fehleranfälligkeit kann Evans also nicht argumentieren. Sie lässt sich kaum eliminieren. Die Idee, dass der Quellcode die Terminologie und Konzepte der Fachsprache nutzen sollte, ist dabei logisch zu unterscheiden von der Idee, dass der Austausch auf Basis einer gemeinsamen Sprache geschehen sollte, deren Terminologie durch ein Modell der fachlichen Domäne geklärt wird.

Zwar argumentiert Evans gegen Übersetzungen von Experten, weil diese zu verworrenen Begrifflichkeiten beitrügen; doch Übersetzungen sind nicht der einzige Grund für Sprachverwirrung. Evans führt Vagheit als weiteren Grund dafür an. Vagheit stellt jedoch kein kommunikatives Versagen dar, sondern ist ein Merkmal natürlicher Sprachen.

Indirekte Kommunikation

In der Literatur zu DDD, aber auch in der Agile-Community finden sich Argumente gegen indirekte Kommunikation und es werden Praktiken favorisiert, die indirektem Austausch oder dessen Folgen entgegenwirken sollen („Let the developer talk with the user", User Stories, Demo ...). Indirekte Kommunikation kann durch die Komplexität der Domäne bedingt sein, weil nur viele Experten gemeinsam über das nötige Wissen zur Domäne verfügen. Sie verfestigt sich jedoch am stärksten auf der Organisationsebene. Ein Modell wird indirekte Kommunikation in Organisationen nicht beseitigen. Denn es ist leicht, ein theoretisches Modell der Domäne zu erstellen, das in der Kommunikation zwischen den Experten, mit den Stakeholdern und den Nutzern in der Praxis nicht genutzt wird.

Indirekte Kommunikation fördert Übersetzungen. Bestimmte Strukturen innerhalb von Organisationen fördern indirekte Kommunikation. In der Folge können sich Fehler in der Kommunikation leicht anhäufen. In einem Unternehmen könnten zum Beispiel Fachexperten mit Business Analysts, Business Analysts mit Softwarearchi-

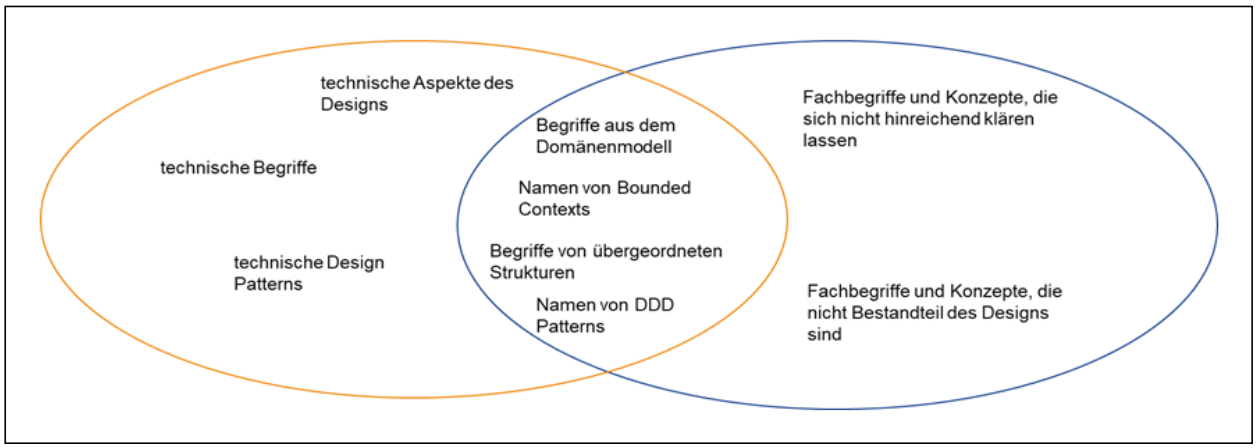


Abb. 2: Das Vokabular der Ubiquitous Language

tekten und Softwarearchitekten mit Entwicklern kommunizieren. In einer solchen Umgebung gibt es drei Ebenen der Übersetzung *a2b*, *b2c* und *c2d* wie **Abbildung 1** zeigt (schwarze Pfeile). Nehmen wir an, die verschiedenen Ebenen sind stark voneinander getrennt und übersetzen unidirektional und unabhängig, und sie haben eine konstante Fehlerquote von *0,2* pro Ebene, d. h., in jeder fünften Transaktion tritt ein Fehler auf, z. B. ein Missverständnis, und feste, einheitliche Transaktionskosten pro Ebene *c*. In einem solchen Fall haben sie eine Gesamtfehlerquote von $1\text{-}0{,}8^3$, also etwa 50 Prozent, und Gesamttransaktionskosten von *3c*.

Um die Gesamtfehlerrate zu verringern, könnte man eine Rückwärtskommunikation pro Schicht zulassen und damit die Transaktionskosten erhöhen (orange Pfeile). Nimmt man zum Beispiel an, dass zwei zusätzliche Transaktionen erforderlich sind, um die Fehlerrate zu halbieren, und führt daher einen Multiplikator von *3* pro Schicht für die Transaktionskosten *c* pro Schicht ein, erhält man *9c* an Gesamttransaktionskosten für drei Schichten und die Gesamtfehlerquote wäre $1\text{-}0{,}9^3$, also etwa 27 Prozent. Im Gegensatz dazu könnte man bei nur einer Übersetzungsschicht standardmäßig eine Fehlerquote von *0,2* und Kosten von *c* haben und eine Fehlerquote von nur *0,1* bei Kosten von *3c* erreichen (blauer Pfeil).

Ob sich Kommunikationsfehler wirksam reduzieren lassen, hängt natürlich stark von der Natur der Fehlerraten und Transaktionskosten pro Ebene ab. Zum Beispiel könnte in einem Netzwerk mit mehreren Ebenen die Lösung einiger Missverständnisse eine rekursive Rückwärtsauflösung erfordern, sodass die Lösung eines Missverständnisses zwischen *c2d* von der Rückwärtskommunikation *b2c* und so weiter abhängt und somit zu einem größeren Anstieg der Transaktionskosten führen könnte. Andererseits mag eine direkte Kommunikation zum Beispiel aufgrund der Größe einer Organisation nicht immer praktikabel sein, sodass die Einführung mehrerer Ebenen erforderlich ist. Wenn jedoch die direkte Kommunikation eine Option ist und der Erwartungswert für die Fehlerquote und die Transaktionskosten für eine direkte Übersetzung *a2d* nicht höher ist als der Erwartungswert für die Gesamtfehlerquote und die Transaktionskosten für die ursprünglichen Schichten *a2b*, *b2c*, *c2d*, dann scheint die Eliminierung empfehlenswert zu sein.

Eine gemeinsame Sprache

Ein gewisses Maß an Übersetzung mag zu Beginn jeder Zusammenarbeit zwischen Spezialisten notwendig sein. Langfristig soll das Team jedoch nicht auf Basis von Übersetzungen kommunizieren, sondern eine gemeinsame Sprache etablieren und praktizieren.

Es gibt ein einfaches Argument, das wir zur Unterstützung seines Vorschlags anführen können. Wie wir aus praktischer Erfahrung wissen, ist die Wahrscheinlichkeit von Missverständnissen zwischen zwei Personen, die unterschiedliche Sprachen sprechen, größer als zwischen zwei Personen, die eine gemeinsame Sprache, z. B. Englisch, beherrschen. Das Gleiche dürfte auch für die Fachsprachen der Spezialisten gelten.

Doch was macht eine gemeinsame Sprache aus? Eine gemeinsame Sprache besteht aus gemeinsamen

- Regeln,
- Begriffen,
- Bedeutungen,
- Gegenständen/Artefakten.

Die gemeinsame Domänensprache wird von Evans Ubiquitous Language genannt. Diese Sprache sollte sowohl von Fachleuten als auch von technischen Experten verstanden werden. Das Vokabular der Ubiquitous Language soll die Terminologie des Domänenmodells enthalten. Namen von Klassen und prominenten Operationen, einige Patterns, Prinzipien der High-Level-Architektur sowie Schlüsselkonzepte der Domäne sollen ebenfalls enthalten sein [1]. Domänenkonzepte, die nicht ausreichend geklärt werden können oder die für den Zweck der Anwendung nicht relevant sind, sind nicht Teil der Sprache (**Abb. 2**).

Die Dinge werden jedoch komplizierter, wenn man beginnt, über die Details der Ubiquitous Language nachzudenken. Es folgt ein kurzer Ausblick auf zwei Probleme, die wir im nächsten Teil der Artikelserie näher beleuchten werden.

Allgegenwärtigkeit

Die gemeinsame Sprache soll im Team allgegenwärtig sein. Ein Problem, das sich aus der geforderten Allgegenwärtigkeit der Sprache ergibt, hat mit der Beziehung zwischen der Ubiquitous Language und dem Quellcode zu tun. Eine Domänensprache ist laut Evans nur dann allgegenwärtig, wenn sie auch in den Code eingebettet ist.

Es ist an sich einfach, Klassen auf der Grundlage der Domänensprache zu benennen und auf diese Weise Domänensprache in den Quellcode einzubetten. Die Einbettung garantiert jedoch nicht, dass die Domänensprache eine gemeinsame Sprache darstellt. Damit Klassennamen tatsächlich Teil einer gemeinsamen Sprache sein können und jeder effektiv über sie sprechen kann, müsste jeder wissen, auf welche Klassen sie sich beziehen. Aber Klassen finden sich nur im Quellcode. Fachexperten kennen in der Regel den Quellcode nicht. Sie werden in den meisten Fällen nicht wissen, worauf sich die „Namen von Klassen“ beziehen. Wenn man Fachexperten nicht beibringt, den Quellcode zu lesen, wie können dann Namen von Klassen Teil einer gemeinsamen Sprache sein?

Sprache-Modell-Bindung

Ein weiteres Problem ist, wie die Bindung von Sprache und Modell sichergestellt werden kann. Da sich Sprache und Modell weiterentwickeln können, ist im Einzelfall nicht klar, ob sie nicht Gefahr laufen, sich getrennt zu entwickeln. Da die Ubiquitous Language in den Quellcode eingebettet sein soll, muss insbesondere die Bindung zwischen dem Quellcode und dem Domänenmodell bestehen bleiben. Auch der Quellcode kann mit der Zeit größeren Veränderungen unterworfen sein.

Wie findet man heraus, ob Sprache und Modell noch zusammenpassen? Eine einfache Methode, das zu prüfen, wäre wünschenswert. Andernfalls wird man nicht in der Lage sein, eine Bindung langfristig aufrechtzuerhalten. Automatisierte Tests könnten ein Weg sein, eine Bindung zwischen Quellcode und Modell sicherzustellen. Die Tests prüfen die Kriterien für die Akzeptabilität des Quellcodes aus der Perspektive des Modells.

Das Idealbild von Evans ist die bidirektionale Bindung zwischen Domänenmodell und Quellcode. Änderungen am Quellcode sollen Änderungen am Modell nach sich ziehen können. Aus organisatorischer Sicht scheint das nur möglich zu sein, wenn der Softwareentwurfs- und -entwicklungsprozess nicht vollständig von oben nach unten diktiert wird, sondern Rückkopplungsschleifen zulässt.

Fazit

Domain-Driven Design ist also nicht auf die leichte Schulter zu nehmen. Es wird oft als eine Reihe von Methoden und Techniken für die Modellierung behandelt und damit betont, dass es sich um einen Ansatz der modellbasierten Softwareentwicklung handelt. Obwohl es nicht falsch ist, DDD auf diese Weise zu charakterisieren, ist die Vorstellung, es sei im Kern ein Ansatz der modellbasierten Softwareentwicklung, etwas unglücklich, da es der Problemdarstellung von Evans nicht gerecht wird.

Eine starke Fokussierung auf Modellierung, ohne über Sprache und Kommunikation zu reflektieren, überschätzt die Bedeutung von Modellen. Denn Modelle entstehen nicht aus dem Nichts. Ein Modell ist ein Artefakt, dessen Erstellung das Ergebnis einer gemeinsamen Anstrengung des Teams ist. Idealerweise beschreibt es einen gemeinsamen Wissensstand und hat eine klar definierte Terminologie. Sprache, Begriffe und Konzepte müssen also geklärt werden, um ein Modell überhaupt erstellen zu können. Modelle können zwar helfen, die Komplexität der realen Welt zu bewältigen, aber ihre Verwendung trägt nicht per se dazu bei, die Komplexität von Sprache und Kommunikation zu bewältigen.

DDD ist kein Selbstläufer und erfordert eine intensive Auseinandersetzung mit der Methode, um sie zu erlernen und zu erkennen, wo die Fallstricke bei der Umsetzung liegen. Mit einem Workshop zu Domain Storytelling und Event Storming wird es jedenfalls nicht getan sein.

Angesichts der Bekanntheit und Popularität von DDD rate ich daher zu nüchterner Vorsicht. DDD kann eine gute Wahl sein, wenn die Fachlichkeit recht komplex ist. Regelmäßige Kommunikation und kontinuierliches Lernen sind in einem solchen Fall möglicherweise nicht effektiv genug, wenn dem Team ein gemeinsames, tieferes fachliches und methodisches Verständnis fehlt. Die Verwendung des DDD-Ansatzes könnte dann hilfreich sein, um Fachlichkeit und Methodik zu klären. Wenn die Komplexität der Fachlichkeit jedoch gering ist, ist der DDD-Ansatz aufgrund seines eher hohen Aufwands wahrscheinlich keine gute Wahl. Es ist also in jedem Fall wichtig, zunächst zu einer Einschätzung der fachlichen Komplexität zu kommen und dann reflektiert zu entscheiden, ob sich der Ansatz lohnt.

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