iX DEVELOPER
Sommer 2020
14,90 €
Österreich 16,40 €
Schweiz 27.90 CHF
Luxemburg 17,10 €
www.ix.de

Moderne Softwarearchitektur

Bessere Software, bessere Systeme

Softwarearchitektur

Domain-driven Design: Best Practices
Microservices Design Patterns

Qualitätssicherung

Shift Left – Secure by Design
The Art of Software Reviews

Cloud

Der Weg hin zu Cloud-native
Container-Orchestrierung im Wandel

Trends

WebAssembly und Project Fugu
Ethik, KI und Quanten-Computing

Rheinwerk Konferenz für

KOTLIN

12.–13. Oktober 2020
Live im KOMED / Köln

Zwei Tage Kotlin, zwei Tage volles Programm!

Tauschen Sie sich mit den Kotlin-Profis aus und holen Sie sich wertvolles Wissen für Ihre Projekte. Ganztägige **Intensivworkshops** und **Vorträge** von führenden Experten machen Sie fit in Kotlin – live vor Ort oder bequem zuhause mit dem **Video-Ticket**.

Mit den internationalen Kotlin-Koryphäen **Annyce Davis**, **Shelby Cohen** und **Katie Levy** sowie **Sebastian Aigner** von JetBrains.

Acht Gründe, dabei zu sein!

- Zwei Tage mit den Kotlin-Profis
- 19 Vorträge zu allen Kotlin-Themen
- Vier Intensivworkshops
- Alle Vorträge auf Video
- Leckeres Essen, tolle Location
- 100 Prozent Rheinwerk
- Viel Platz für alle Teilnehmer
- **Hygiene- und Sicherheitskonzept**

Jetzt Early-Bird-Ticket-sichern!

www.rheinwerk-kkon.de

Unsere Sponsoren und Partner:

MATHEMA · heise Developer · REWE digital · grandcentrix Ein Vodafone Unternehmen · JetBrains · t:n digital pioneers

Von Erfahrung profitieren

Zugegeben – ich bin kein Softwarearchitekt. Aber ich verfolge seit geraumer Zeit aufmerksam die zentrale Rolle der Softwareentwicklung aus unterschiedlichsten Perspektiven und bekomme so die fortwährend neuen Anforderungen mit, die sich zu den weiterhin wichtigen traditionellen Aufgaben von Softwarearchitekten gesellen. Um auf dem Laufenden zu bleiben, dient mir der SoftwareArchitekTOUR-Podcast auf *heise Developer* als wichtigste Quelle, der seit 2009 „on air" ist und den ich mittlerweile glücklicherweise wieder Monat für Monat produzieren darf (nachdem er über Jahre hinweg fast einzuschlafen schien). Dafür danke ich den Podcastern – Carola Lilienthal, Sandra Parsick, Stefan Tilkov, Michael Stal, Gernot Starke, Christian Weyer und Eberhard Wolff – aufrichtig.

Ihre Erfahrung ist ein Schatz, den es immer wieder aufs Neue zu heben gilt, und ich kann es nur empfehlen, dass Sie, liebe Leserinnen und Leser dieses Sonderhefts, in den Podcast reinhören. Das gilt selbst für die älteren Episoden, denn viele der schon vor Jahren produzierten Folgen sind auch heute noch zeitgemäß. Ansonsten beleuchten die Podcaster selbstverständlich auch aktuelle oder sogar zukunftsgerichtete Themen. Wer als Softwarearchitekt oder Softwarearchitektin dem Podcast folgt, ist sozusagen auf dem Laufenden und kann vom Wissen, von den Erfahrungen und Ratschlägen unserer Expertinnen und Experten profitieren.

Mit einem ähnlichen Anspruch ist die Redaktion von *heise Developer* beim Sonderheft „Moderne Softwarearchitektur" vorgegangen. Wir haben für Sie einen Mix aus *iX*- und *heise Developer*-Artikeln zu neuen Technologien und Trends konfektioniert, uns war es aber auch wichtig, weniger attraktive und doch sehr wichtige Bereiche der Softwarearchitektur, wie Software-Reviews, mit aufzunehmen, die andernorts gut und gerne mangels fehlendem Sexappeal unberücksichtigt bleiben.

Wir hoffen, dass Ihnen die breite Themenmischung einen runden Überblick zum Status quo der Softwarearchitektur bietet und vielfältige Anregungen liefert, die komplexen Aufgaben in unterschiedlichsten Gebieten zu meistern. Denn wie ein schönes und stabiles Haus einen guten Architekten braucht, steht hinter stabiler Software ein guter Softwarearchitekt, auch wenn er in der Welt des Codes weniger sichtbar ist als die Gestalter prächtiger Bauwerke.

iX und *heise Developer* wünschen Ihnen viel Spaß bei der Lektüre. Bleiben Sie gesund!

ALEXANDER NEUMANN

Softwarearchitektur

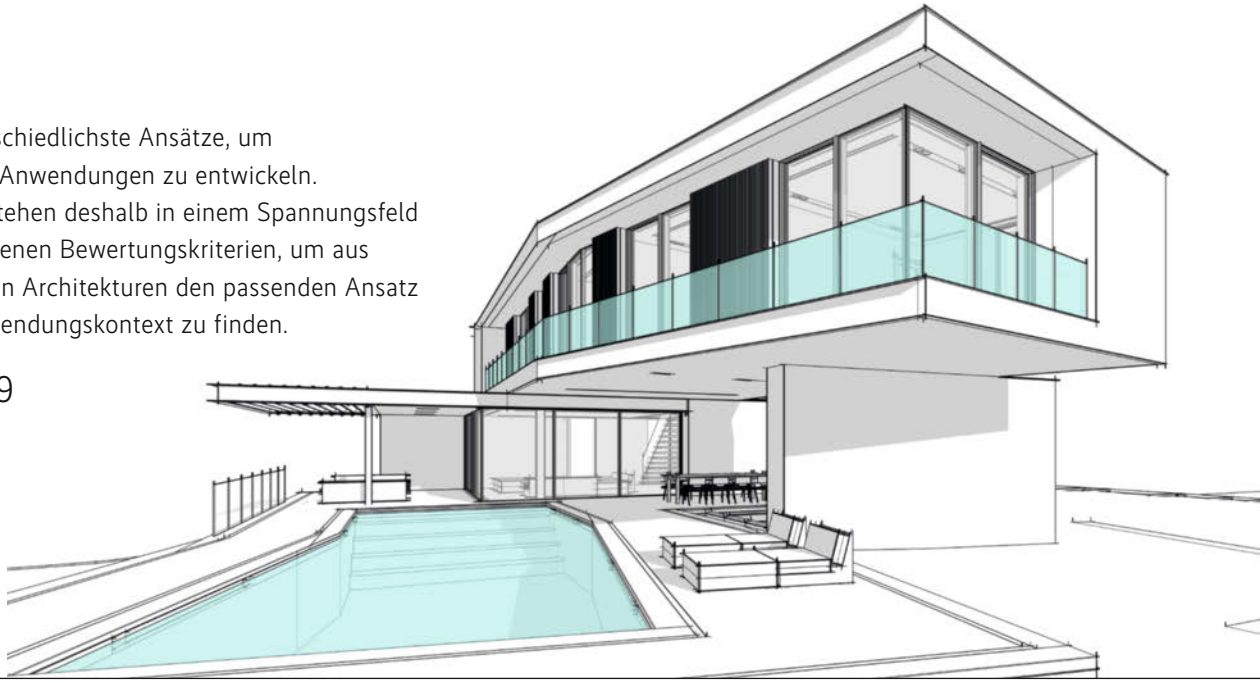
Domain-driven Design ist keineswegs ein neues Verfahren, Software zu modellieren. Aber insbesondere die komplexen Herausforderungen durch Microservices verdeutlichen den Wert dieser Herangehensweise. Und zur Bewältigung der Komplexität von Microservices-Architekturen mag sogar eine erneute Beschäftigung mit Design Patterns hilfreich sein.

ab Seite 7

Cloud

Es gibt unterschiedlichste Ansätze, um Cloud-native Anwendungen zu entwickeln. Architekten stehen deshalb in einem Spannungsfeld aus verschiedenen Bewertungskriterien, um aus den möglichen Architekturen den passenden Ansatz für ihren Anwendungskontext zu finden.

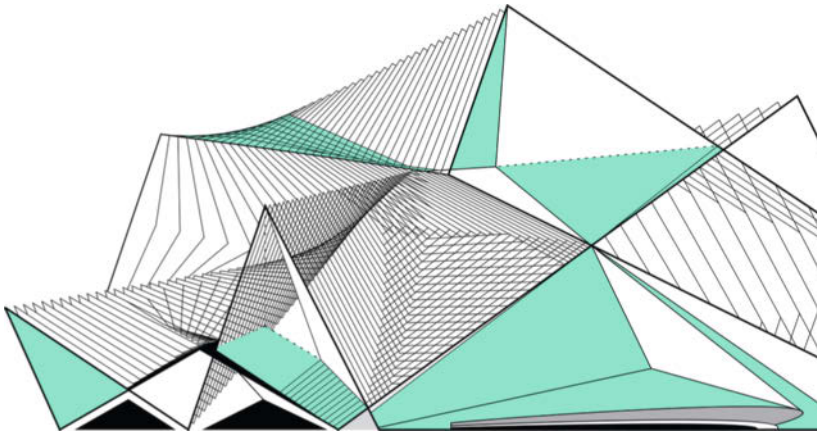
ab Seite 29

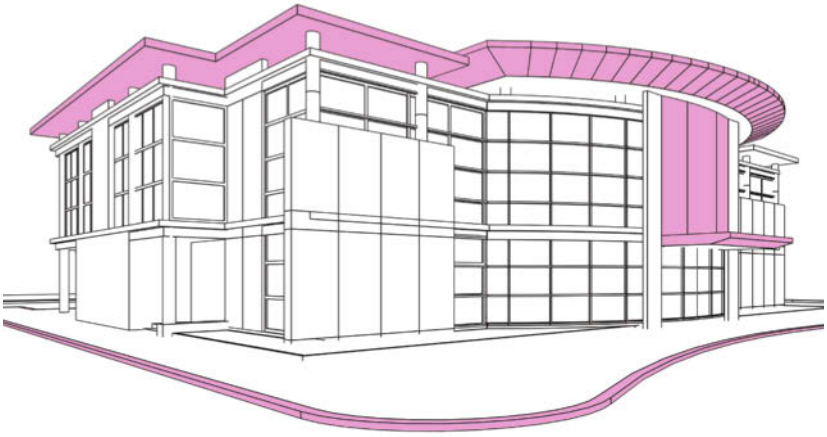


Qualitätssicherung

Requirements Engineering, Testautomatisierung, Security, Software-Reviews, Dokumentation und Performance-Analyse sind allesamt etablierte Bereiche des Softwarelebenszyklus. Dieses Sonderheft beleuchtet unter anderem jüngere Entwicklungen wie Shift Left oder Documentation as Code.

ab Seite 57





Web und Mobile

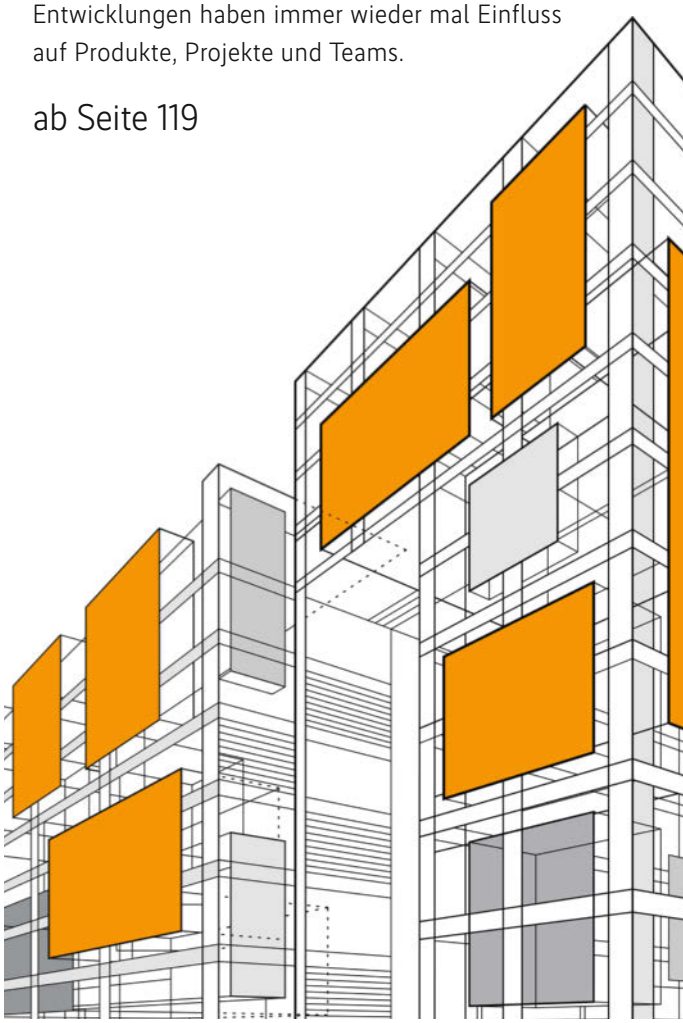
WebAssembly will eine Plattform zur Ausführung von vorkompiliertem Bytecode sein, Project Fugu die Lücke zwischen nativen Apps und Progressive Web Apps schließen und deklarativen UI-Frameworks gehört offenbar die App-Zukunft. Zeit, sich jetzt hierzu schlau zu machen.

ab Seite 101

Diverses

Softwarearchitekten müssen sich beileibe nicht nur mit System-Design, Aspekten der Softwarequalität oder angesagten neuen Technologien auseinandersetzen, sie sollten auch Zukunftstechniken wie Machine Learning oder Quanten-Computing frühzeitig auf dem Radar haben. Selbst soziale oder gesellschaftliche Entwicklungen haben immer wieder mal Einfluss auf Produkte, Projekte und Teams.

ab Seite 119

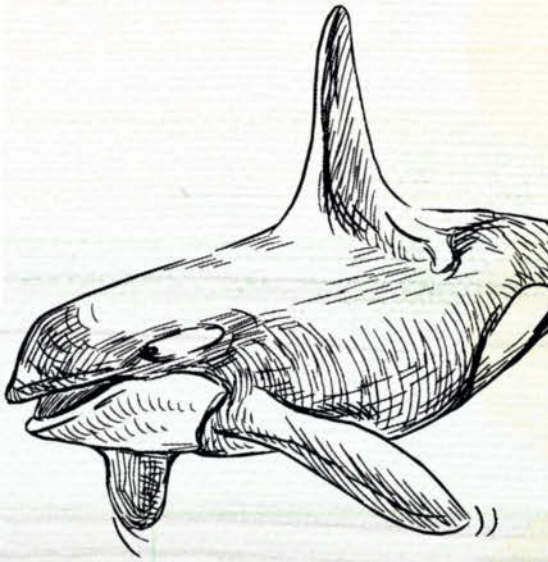


Diverses

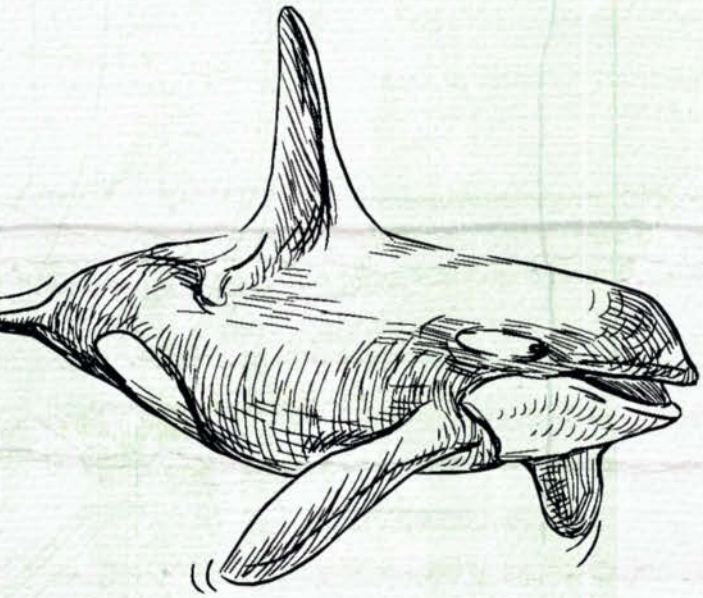
Sonstiges

enterJS

28.9. – 1.10.2020
Online



Die Online-Konferenz für Enterprise JavaScript



- Moderne Softwarearchitektur
- Frameworks & Tools
- Testing & Security
- Frontend & Backend
- JavaScript Deep Dives

Early-Bird bis zum
7. September 2020

Silbersponsor
ACCSO
ACCELERATED SOLUTIONS

Bronzesponsor
Maiborn Wolff
Mensch IT

Veranstalter
iX heise Developer dpunkt.verlag

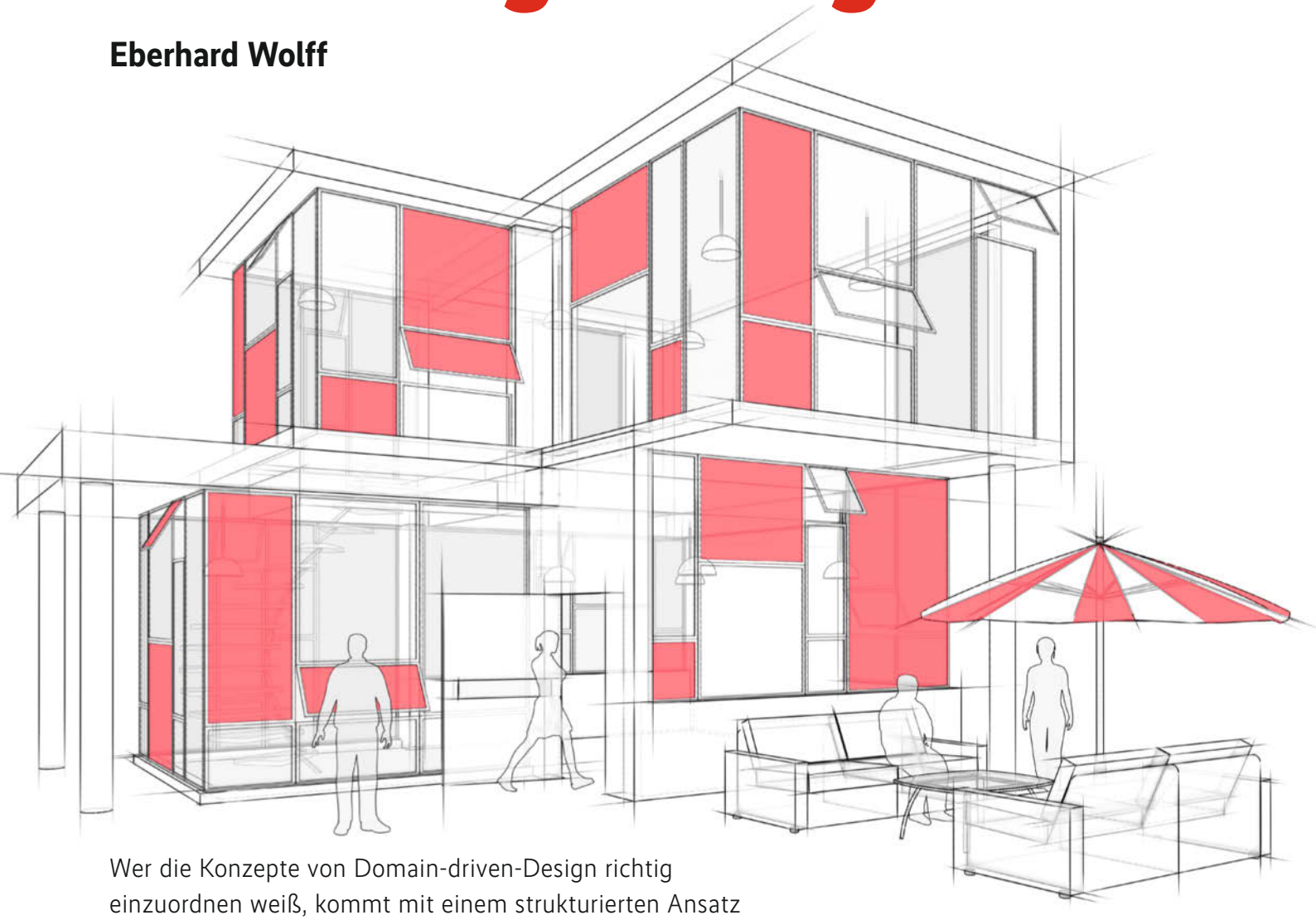
Softwarearchitektur

„Der Unterschied zwischen guter und schlechter Architektur ist die Zeit, die man dafür aufwendet.“ (Sir David Chipperfield)

Große Systeme mit DDD entwerfen

Beziehungsmanagement

Eberhard Wolff



Wer die Konzepte von Domain-driven-Design richtig einzuordnen weiß, kommt mit einem strukturierten Ansatz recht einfach zu guten Ergebnissen.

Domain-driven Design [1] teilt große Systeme in Bounded Contexts auf, von denen jeder ein eigenes Domänenmodell hat, um einen Teil der Fachlichkeit weitgehend unabhängig zu erbringen. In einem System für eine Bibliothek könnte es einen Bounded Context für das Ausleihen von Büchern und einen anderen für die Suche geben. Das eine Domänenmodell hätte dann die Informationen für die Leihe wie die Anzahl der Exemplare eines Buchs. Beim Domänenmodell für die Suche spielt die Anzahl keine Rolle, dafür sind Autor oder Titel wichtige Informationen.

Beziehungen zwischen Bounded Contexts

Domain-driven Design bietet Ansätze, um das benötigte Integrieren der Bounded Contexts durchzuführen (s. Abb.). Am Anfang steht die Analyse, in welcher Beziehung Bounded Contexts zueinanderstehen. Die Arten der Beziehung sind folgende:

- Free bedeutet, dass die Arbeit an einem Bounded Context wenig Auswirkungen auf die Arbeit an anderen Bounded Contexts hat,
- bei Upstream-Downstream ist der Erfolg des Teams, das den Bounded Context im Downstream bearbeitet, von dem Team abhängig, das den im Upstream in Arbeit hat,
- Mutually Dependent bedeutet, dass die Teams beide Systeme nur zusammen erfolgreich liefern können.

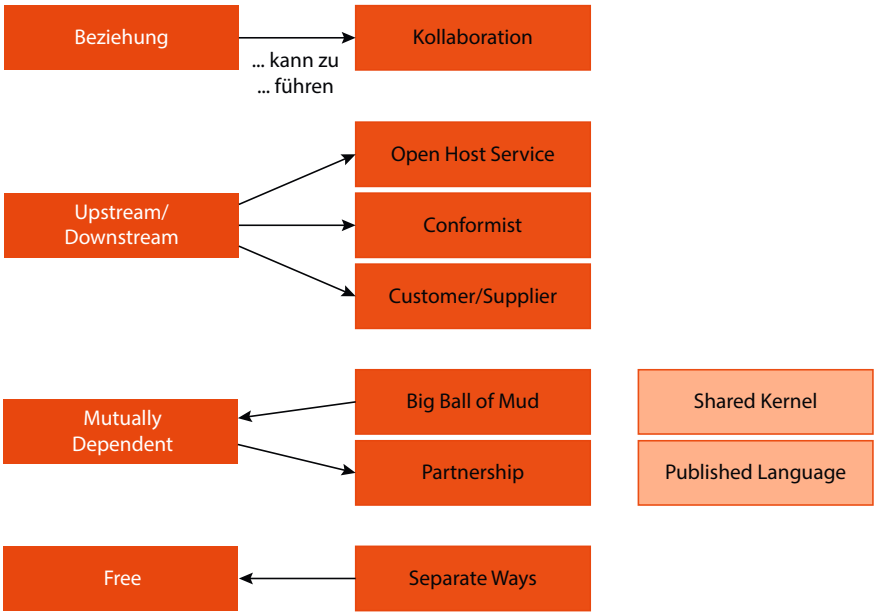
Welche Beziehung vorliegt, lässt sich fast immer durch die Analyse der Bounded Contexts ermitteln. Es ist also meistens keine Designentscheidung, sondern ergibt sich aus der Aufteilung.

Zurück zum Beispiel der Bibliothek: Deren Besucher suchen zunächst Bücher und leihen sie anschließend aus. Bei „Leihe“ und „Suche“ kommt Free somit nicht in Frage, da es eine Mög-

lichkeit geben muss, gefundene Bücher auszuleihen. Daher haben die beiden Bounded Contexts eine Beziehung. Der Erfolg der „Leihe“ ist wahrscheinlich gegeben, wenn sich Bücher ausleihen lassen. Der Erfolg der „Suche“ hingegen bedeutet vermutlich, dass das Finden der Bücher erfolgreich ist. Besucher können ein Buch nur leihen, wenn sie es vorher gefunden haben. „Leihe“ ist damit Downstream zu „Suche“ und umgekehrt „Suche“ Upstream zu „Leihe“.

Mutually Dependent ist die Beziehung nicht: Zum einen geht sie nur in eine Richtung und ist daher nicht gegenseitig (mutually) und zum anderen gibt es in der „Suche“ und „Leihe“ genügend Bereiche und Features, die sich unabhängig voneinander entwickeln lassen. Die Ergänzung der Suche um neue Kriterien hat beispielsweise keine Auswirkungen auf die Leihe.

Primär schlagen sich die Beziehungen in der Organisation und Koordination zwischen den Teams nieder und nicht in den Softwareartefakten. Dennoch sind es zusätzlich Ergebnisse der Relationen auf Softwareebene: Zwei Bounded Contexts mit einer Schnittstelle können kaum Free sein.



DDD bietet mehrere Ansätze, über die sich Bounded Contexts integrieren lassen.

Auf der Organisationsebene sind die Beziehungen ein direktes Ergebnis der Softwareartefakte. Wenn ein Team die Verantwortung für einen Downstream Bounded Context übernimmt, bekommt es die passende Downstream-Rolle. Es ist genau genommen nicht das Team selbst, das die Beziehung hat, sondern der Bounded Context gibt die Beziehung an das Team weiter.

Kollaborationen: Organisation designen

Die oben genannten Beziehungen ergeben sich aus der Aufteilung in Bounded Contexts: „Leihe“ ist unabdingbar Downstream von „Suche“. Das hat jedoch Nachteile, da das Verleihen von Büchern die Kernaufgabe einer Bücherei ist. Wenn andere Bereiche die zentrale Aufgabe negativ beeinflussen können, sind Gegenmaßnahmen angebracht, um die Prioritäten in dem Projekt richtig setzen zu können.

Die Upstream-Downstream-Beziehung lässt sich nicht wegdiskutieren, da sie objektiv vorhanden ist. Dennoch existieren unterschiedliche Wege, auf das Verhältnis Upstream-Downstream in der Organisation zu reagieren:

- Customer-Supplier-Kollaboration
- Conformist-Kollaboration
- Anticorruption Layer
- Open-Host-Service

Bei einer Customer-Supplier-Kollaboration kann das Downstream- beim Upstream-Team Features einfordern, die letzteres im Rahmen seiner Möglichkeiten umsetzt. Bei dieser Art der Zu-

sammenarbeit könnte „Leihe“ bei der „Suche“ neue Features anfordern. Damit hat „Leihe“ tatsächlich eine höhere Priorität als „Suche“.

Anders bei der Conformist-Kollaboration: In dem Fall übernimmt das Downstream-Team das Datenmodell des Upstream-Teams und hat kein Mitspracherecht. Das erleichtert zwar die Integration, liefert aber das Downstream-Team dem Upstream-Team aus. Die Entscheidung zwischen Customer-Supplier und Conformist ist eine Architekturentscheidung. Sie hängt von der Priorisierung ab: Ist das Downstream-Team politisch mächtig oder bearbeitet ein wichtiges Thema, bekommt es eher eine Customer-Rolle. Sonst findet es sich vielleicht in einer Conformist-Rolle wieder.

Somit lässt sich die Upstream-Downstream-Beziehung mit einer Architekturentscheidung an den Stellen kompensieren, an denen es sinnvoll erscheint. Teams könnten die „Leihe“ in eine Conformist-Rolle zwingen, wenn die Priorisierung nicht hoch genug ist.

Eine weitere Option ist der Anticorruption Layer: Ähnlich wie bei Conformist kann das Downstream-Team nicht in die Belange des Upstream-Teams eingreifen, hat jedoch in dem eigenen System eine Schicht eingezogen, durch die es sich von dem Domänenmodell des Upstream-Teams entkoppelt. Auf der Ebene der Zusammenarbeit entspricht das dem Conformist, aber es kommt eine weitere Schicht zur Übersetzung der Modelle beim Downstream Bounded Context hinzu.

Bei einem Open-Host-Service gibt es mehrere Downstream-Teams, die gemeinsam eine Schnittstelle des Upstream-Teams nutzen. Als Vorschrift für die Interaktion gilt: Das Upstream-Team ist dafür zuständig, die Schnittstelle weiterzuentwickeln. Anforderungen der Downstream-Teams können in die Weiterentwicklung einfließen, aber das Upstream-Team kann Anforderungen ablehnen und stattdessen einem Downstream-Team eine eigene Schnittstelle anbieten.

Auf zweierlei Weise voneinander abhängig sein

Analog zu den unterschiedlichen Ansätzen für Upstream-Downstream-Relation lässt sich auf Organisationsebene auf Beziehungen, die Mutually Dependent sind, unterschiedlich reagieren:

- Partnership
- Big Ball of Mud

Bei einer Mutually-Dependent-Beziehung kommt fast ausschließlich eine Partnership für die Integration in Frage. Der Begriff hört sich zunächst positiv an, steht aber für eine äußerst enge Abstimmung, die typischerweise zu erheblichen Reibungsverlusten führt. Ein anderes Vorgehen für den Mutually-Dependent-Fall ist der sogenannte Big Ball of Mud, der allerdings kein geschickter Umgang mit wechselseitig abhängigen Teams ist. Der Begriff bezeichnet ein System, das mehrere Bounded Contexts umfasst, aber dermaßen schlecht strukturiert ist, dass praktisch überhaupt keine Struktur zu erkennen ist. Aufgrund der schlechten Struktur existieren viele Abhängigkeiten. Daraus folgt, dass sich Teams, die gemeinsam an einem Big Ball of Mud arbeiten, eng koordinieren müssen und daher Mutually Dependent sind. Kaum ein Team entscheidet sich bewusst für einen Big Ball of Mud, sondern er entsteht, wenn eine sinnvolle Strukturierung des Systems nicht angegangen wird.

Die Free-Beziehung führt zu vollständiger Unabhängigkeit und daher zu keiner Zusammenarbeit. Daher erscheint es kaum sinnvoll, überhaupt eine Beziehung zwischen Free und einer Kollaboration zu diskutieren. Dennoch gibt es die Separate-Way-Kollaboration: Wenn Teams zwei Bounded Contexts zwar integrieren können, der Aufwand dafür aber in keinem vernünftigen Verhältnis zum Nutzen steht, können sie sich eine Integration sparen. Voraussichtlich kaum genutzte Schnittstellen kosten gegebenenfalls in der Implementierung mehr Geld, als sie später an Ersparnis einbringen. Durch die Auflösung sind die Teams, die an den beiden Bounded Contexts arbeiten, in einer Free-Beziehung.

Weitere Modelle für die Zusammenarbeit

Darüber hinaus existieren Kollaborationen, die nicht unbedingt Beziehungen beeinflussen:

- Bei einem Shared Kernel teilen sich die Bounded Contexts den Code, die Akzeptanztests und die Datenbank. Änderungen am Shared Kernel erfordern eine Koordination. Um eine ausreichende Unabhängigkeit der Teams zu erreichen, sollte der Shared Kernel klein sein, um eine Mutually-Depend-Beziehung zu verhindern. Upstream-Downstream ergibt sich ebenfalls nicht zwingend, da die beiden Teams gemeinsam an dem Shared Kernel arbeiten sollen.
- Eine Published Language ist ein gemeinsames Datenmodell, das nur zur Kommunikation dient. Es kann sich um einen öffentlich publizierten Standard handeln oder um ein in einem Wiki erklärtes Format. Offensichtlich haben die Bounded Contexts dabei eine Beziehung und müssen sich abstimmen, aber ähnlich wie bei Shared Kernel ist die Beziehung nicht eng, weil sie nur ein Datenmodell für die Kommunikation betrifft. Änderungen am Modell lassen sich daher von Anpassungen am internen Modell entkoppeln.

Gute Beziehungen

Gerade für die Strukturierung großer Systeme ist Domain-driven Design nützlich. DDD beschreibt auf der einen Seite Beziehungen, die Teams eingehen, wenn sie sich um einzelne Bounded Contexts kümmern müssen. Da sich die Relationen aus der Aufteilung des Systems ergeben, lassen sie sich nicht direkt beeinflussen. Daher schlägt DDD einige Kollaborationen vor, um mit den Beziehungen weitgehend gewinnbringend umzugehen. Die Begriffe „Beziehung“ und „Kollaboration“ finden sich übrigens nicht in der DDD-Literatur. Sie helfen aber, die beiden Mechanismen voneinander zu unterscheiden. Einen Überblick bietet die Abbildung, und weitere Informationen finden sich bei Eric Evans [1]. (ane@ix.de)

Quellen

Eric Evans; Domain-driven Design Referenz (ddd-referenz.de)

Eberhard Wolff

arbeitet als Fellow bei INNOQ. Er ist seit mehr als 15 Jahren als Architekt und Berater tätig – oft an der Schnittstelle zwischen Business und Technologie. Sein technologischer Schwerpunkt liegt auf modernen Architektur-Ansätzen – Cloud, Continuous Delivery, DevOps, Microservices oder NoSQL spielen oft eine Rolle.

Herbstcampus

Java 2020

Die Online-Konferenz zum Status quo moderner Java-Entwicklung

1. bis 3. September 2020

Frühbucherrabatt
bis zum 10. August

Die „Java 2020"-Edition des Herbstcampus bietet Ihnen einen kompakten Überblick zum Status quo der Java-Entwicklung und hilft Ihnen, Ihre Java-Anwendungen zukunftssicher zu gestalten.

Das können Sie lernen:

- ☑ (Wir haben ein) Neues JDK – was nun?
- ☑ Kommerzielle Anbieter oder eine freie Version?
- ☑ Wie die GraalVM richtig einsetzen?
- ☑ Eclipse MicroProfile oder Jakarta EE?
- ☑ Ist Quarkus wirklich das „Supersonic Subatomic Java"?
- ☑ Micronaut oder andere Microframeworks

www.herbstcampus.de

Goldsponsor

adesso business. people. technology.

Silbersponsoren

codecentric

e.solutions

Bronzesponsoren

ISO GRUPPE IT | Systems · Solutions · Services

NÜRNBERGER VERSICHERUNG

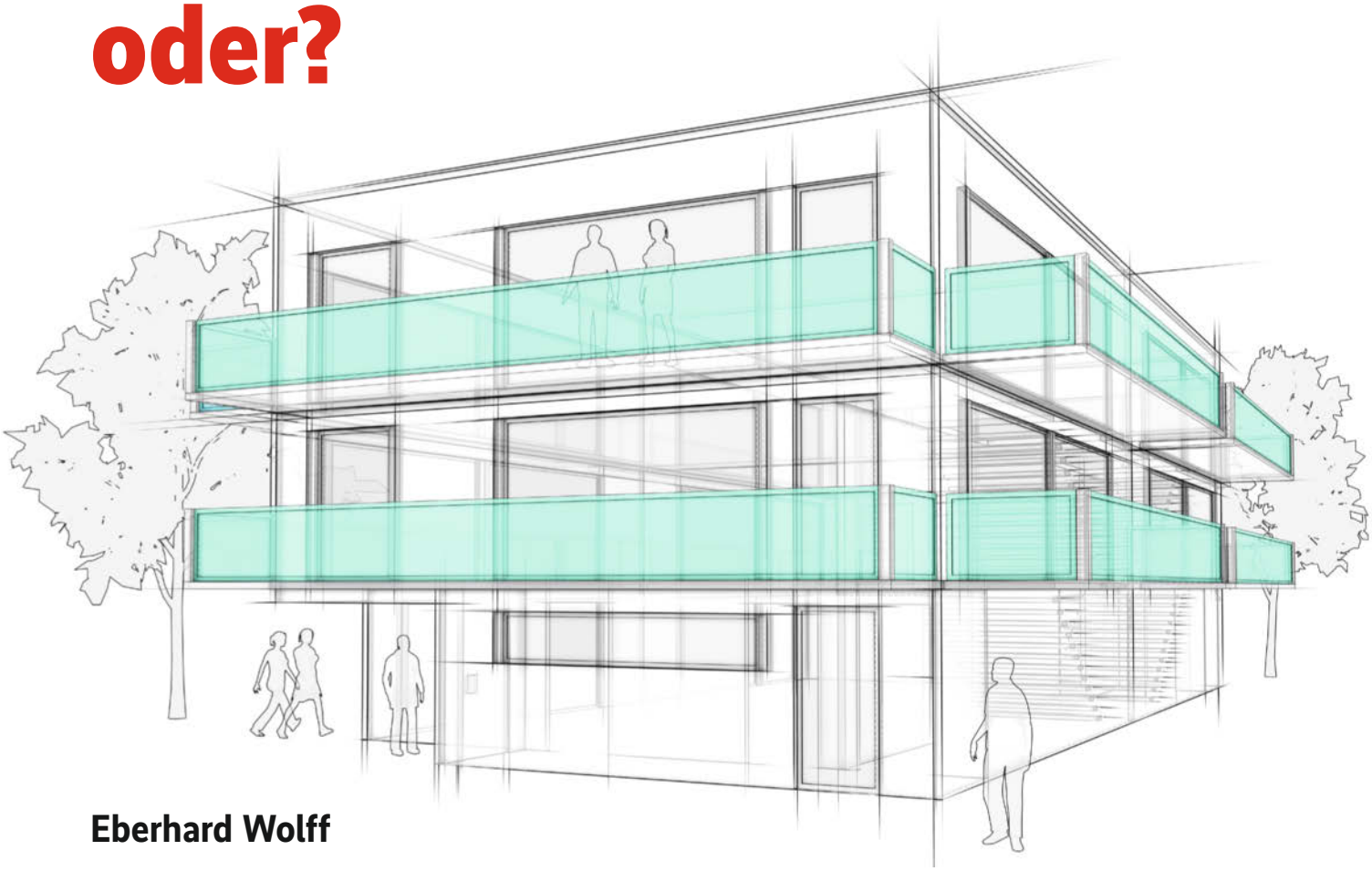
Veranstalter

iX heise Developer

dpunkt.verlag

Domain-driven Design und Bounded Context

Eigentlich ganz einfach, oder?



Eberhard Wolff

Die Konzepte von DDD und Bounded Kontext sind in der Praxis komplizierter, als es auf den ersten Blick erscheint.

Domain-driven Design (DDD) ist einer der wichtigsten Ansätze, um fachliche Funktionalitäten in einem System zu organisieren und abzubilden. Konzepte wie Strategic Design und Bounded Context sind ausgesprochen hilfreich, um grobgranulare Module wie Microservices zu schneiden. Die Konzepte sind im praktischen Einsatz oft komplizierter als gedacht.

Der Begriff „Bounded Context“ sagt klar, dass es um einen begrenzten Bereich geht. Ein Bounded Context begrenzt unterschiedliche Dinge:

- Der Bounded Context definiert den Einsatzbereich eines Domänenmodells. Es umfasst die Geschäftslogik für eine bestimmte Fachlichkeit. Als Beispiel beschreibt ein Domänenmodell die Buchung von S-Bahn-Fahrkarten und ein weiteres die Suche nach S-Bahn-Verbindungen. Da die beiden Fachlichkeiten wenig miteinander zu tun haben, sind es zwei getrennte Modelle. Für die Fahrkarten sind die Tarife relevant und für die Verbindung die Zeit, das Fahrziel und der Startpunkt der Reise.
- Außerdem soll der Bounded Context den Gültigkeitsbereich einer sogenannten Ubiquitous Language festlegen. Die Bezeichnung lässt sich mit allgegenwärtiger Sprache übersetzen und beschreibt eine Menge von Begriffen, die Domänenexperten verwenden und die für Softwareartefakte wie Code oder Datenbank-Schemata genutzt werden sollen. Beispielsweise erschließt sich die Bedeutung des Begriffs „Taktverstärker“ nicht sofort, den die Münchener Verkehrsbetriebe für zusätzliche S-Bahn-Züge zu den Stoßzeiten verwenden. Ein System für die Münchner S-Bahn sollte daher einen Taktverstärker als Klassennamen im Code und als Name einer Tabelle im Datenbank-Schema verwenden. Das Beispiel zeigt, dass die Ubiquitous Language ohne Wissen über den fachlichen Kontext nicht zu verstehen ist.
- Ein Bounded Context sollte in den Zuständigkeitsbereich genau eines Teams fallen. Ein Team kann für mehr als ein Bounded Context zuständig sein, aber umgekehrt sollten nicht mehrere Teams an einem Bounded Context arbeiten, da das Vorgehen zu viel Abstimmung erfordert.

Somit ist ein Bounded Context ein Gültigkeitsbereich eines Domänenmodells, einer Ubiquitous Language und die Basis für die Organisation des Projekts.

In der Praxis

Als Beispiel für ein System nach dem DDD-Gedanken soll eine Bücherei dienen. Zwei wesentliche Funktionalitäten einer Bibliothek sind das Suchen und das Ausleihen von Büchern. Sie unterscheiden sich stark genug, dass zwei unterschiedliche Bounded Contexts sinnvoll erscheinen. Für die Funktionalitäten müssen jeweils Geschäftslogik und Daten über Domänenobjekte wie Bücher oder Leser bereitstehen (s. Abb. 1).

Interessant ist die Entität „Buch" in den beiden Kontexten: Im Bounded Context „Ausleihe" ist ein „Buch" ein konkretes Exemplar, das man in die Hand nehmen kann. Für eine ISBN existieren beliebig viele solcher Bücher.

Für die Suche ist ein „Buch" hingegen eine Sammlung von Daten wie Autor, Titel oder Schlagwörter. Ein solches Buch kann mehrere ISBNs haben, da das gedruckte Buch und das E-Book desselben Titels unterschiedliche ISBNs erhalten. Das Buch in der Suche und in der Leihe sind somit vollständig unterschiedliche Konzepte, die andere Daten haben und eine andere Identität, wie das Verhältnis zur ISBN belegt.

Das Buch hat in der Leihe und in der Suche zudem jeweils einen unterschiedlichen Lebenszyklus: Ein zusätzliches Exemplar erfordert im Kontext der Leihe einen Vermerk, aber in der Suche ändert sich nichts, denn der Sucheintrag existiert ab oder sogar vor dem ersten Exemplar. Bei der Aufnahme eines Buchs in die Suche muss es in der Leihe noch nicht verfügbar sein. Das Anlegen eines neuen Buchs ist daher nur beim Betrachten in einem bestimmten Bounded Context sinnvoll.

Die inhaltliche Trennung eröffnet die Chance auf mehrere kleine, spezialisierte Domänenmodelle statt übermäßig komplexer großer Domänenmodelle. Das hilft nicht nur bei objektorientierten Modellen, sondern lässt sich ebenso für Datenbankmodelle anwenden, die häufig ebenso viel zu komplex sind.

Nicht nur für die Entwicklung, sondern auch für die Analyse der Domäne ergeben sich Vorteile: Der Begriff „Buch" ist in der Bibliothek offensichtlich nicht eindeutig definierbar. Durch die Bounded Contexts lässt sich in der Analyse damit umgehen, dass „Buch" in unterschiedlichen Bereichen anders definiert sein kann. Innerhalb eines Bounded Context gibt eine eindeutige Definition von „Buch", die aber nicht darüber hinaus gilt [1].

Wenn mehrere Domänenmodelle Informationen über das Buch enthalten, wirkt das wie redundante Datenhaltung, aber wegen der unterschiedlichen Fachlichkeiten haben die Domänenmodelle tatsächlich kaum redundante Daten. Informationen wie die Anzahl der Exemplare gehören eindeutig zur Leihe, Informationen wie die Schlagwörter hingegen eindeutig zur Suche. Es kann sicher Informationen geben, die in beiden Bounded Context relevant sind, aber weil die beiden Bounded Contexts unterschiedliche Domänenmodelle haben und sogar unterschiedliche Ubiquitous Languages gelten, sind solche redundanten Informationen die Ausnahme.

Spezialisierte, kleine Domänenmodelle sind ein wesentliches Konzept von Domain-driven Design. In der Praxis ist es allerdings manchmal nicht ganz einfach, dieses Konzept umzusetzen. Der Ausgangspunkt für den Entwurf sind die Funktionalitäten der Bounded Contexts. Sie müssen die dafür benötigten Daten und die Logik umfassen. Das entspricht objektorientiertem Design, bei dem die Umgangsformen und Methoden einer Klasse im Mittelpunkt stehen und erst in der Folge die Daten eine Rolle spielen.

Viele Entwürfe gehen jedoch von den Daten aus. Das führt für die beispielhafte Bibliothek zu einem Problem, weil es in beiden Bounded Contexts „Leihe" und „Suche" das „Buch" gibt, es aber in jedem Bounded Context etwas anderes bedeutet. Eine Modellierung nach den Daten führt kaum zu einer Aufteilung in sinnvolle Bounded Contexts, sondern eher zu übermäßig komplexen Modellen. Wichtig ist, die Daten als Folge der Funktionalitäten zu modellieren.

Keine grünen Wiesen mehr

Es gibt kaum Projekte, bei denen Systeme auf der grünen Wiese entstehen. Meistens gibt es andere IT-Systeme, die es zu integrieren oder abzulösen gilt. Beispielsweise könnte in der Bibliothek ein Buchsystem existieren, das alle Informationen zu Büchern umfasst und das in einem Widerspruch zu der erläuterten, idealen Modellierung steht:

- Das Buchsystem hat ein Modell mit allen Daten eines Buchs. Wie erwähnt ist eine Aufteilung in mehrere Modelle mit spezialisierten Modellierungen des Buchs beispielsweise für Suche oder Leihe wesentlich sinnvoller.
- Ein System, das lediglich Daten verwalten soll, implementiert keine Funktionalitäten, die Grundlage für die Aufteilung nach Bounded Context ist. Systeme, die nur Daten verwalten, bilden die wesentliche Domänenlogik nicht ab.

Das Buchsystem erfüllt dennoch ein Merkmal eines Bounded Context: Es hat ein eigenes Domänenmodell. Es verwaltet alle Informationen über Bücher.

Ein anderes Merkmal eines Bounded Context erfüllt das Buchsystem nicht: Es setzt keine Ubiquitous Language um. Weil das System alle Informationen zu einem Buch verwaltet, setzt es sie weder für die Leihe noch die für die Suche um. Das Datenmodell hat vermutlich eine Klasse „Buch", aber es muss die Daten für Leihe, Suche und gegebenenfalls einige weitere Funktionalitäten implementieren. Daher kann die Klasse „Buch" im Buchsystem keiner der beiden Ubiquitous Languages entsprechen und somit keinem der Begriffe der Domänenexperten.

Es gibt also Domänenmodelle, die keine Ubiquitous Language abbilden. Im Beispiel müsste ein solches Modell alle Informationen für Bücher verwalten. Dabei geht es nicht nur um die Anzahl der Attribute, sondern auch darum, die Konzepte „Buch" in Leihe und Suche abzubilden. Da sie grundverschieden sind, entsteht ein kompliziertes und wegen der Mehrdeutigkeiten verwirrendes Modell. Zudem ändert sich bei jeder fachlichen Änderung an der Leihe oder der Suche der Bücherservice, wenn sich etwas an der Modellierung der Bücher ändert. Auf die Weise kann er zu einem Änderungsschwerpunkt werden, den es bei jeder fachlichen Änderung anzupassen gilt.

Die spezialisierten Modelle sind auf jeden Fall kleiner, einfacher und weniger verwirrend. Sie versprechen zudem, dass Än-

Bounded Context für Suche und Leihe (Abb. 1)

Datenreplikation kann gegebenenfalls später in die umgekehrte Richtung erfolgen

Buchsystem und Bounded Context für Leihe (Abb. 2)

derungen mit einer höheren Wahrscheinlichkeit nur ein Domänenmodell umfassen. Eines der zentralen Versprechen von Bounded Contexts ist, dass mehrere kleinere Domänenmodelle oft ein besserer Weg sind, um mit der Komplexität klarzukommen.

Beim Umgang mit dem Buchsystem sollten die Verantwortlichen somit eine Ablösung erwägen, um die Vorteile von Domain-driven Design vollständig umzusetzen. Das ist jedoch eventuell aufwendig, da das Domänenmodell in dem Buchsystem komplex ist. Außerdem kann das System Abhängigkeiten zu vielen anderen Systemen haben. Schließlich kommt in einer Bibliothek wohl kaum ein IT-System ohne Informationen über Bücher aus, die sich das jeweilige System vom Buchsystem holt.

Wegen der Komplexität der Domänenmodelle kann es dennoch eine gute Idee sein, von solchen Systemen weg zu migrieren oder sie zumindest nicht weiter wachsen zu lassen. Dazu lässt sich ein neues System mit einem Bounded Context wie „Leihe“ oder „Suche“ implementieren, das die benötigten Daten aus dem Buchsystem repliziert. Das Buchsystem bleibt zunächst für die Daten führend.

Wenn der neue Bounded Context später für die Informationen führend ist, lässt sich die Richtung der Replikation umkehren. Dann erfolgt beispielsweise die Verwaltung der Information über die Anzahl der vorhandenen Exemplare im Bounded Context „Leihe“ und das Buchsystem hat nur eine Kopie der Informationen. Gegebenenfalls ist die Anzahl der Exemplare nur für die Leihe relevant. In dem Fall lässt sich auf das Replizieren der Daten verzichten. Weil die benötigten Daten eine Folge der im Bounded Context implementierten Funktionalitäten sind, ist es sogar recht wahrscheinlich, dass nur ein Bounded Context die Information braucht.

Wenn es dennoch andere Systeme gibt, die an der Anzahl der Exemplare interessiert sind, wenden sie sich nicht mehr an das Buchsystem, sondern den Bounded Context „Leihe“. Nach der Umsetzung verschwinden diese Informationen aus dem Buchsystem, dessen Komplexität dadurch sinkt. Nach und nach lässt sich das Buchsystem eliminieren.

Dabei handelt es sich um einen langwierigen Prozess, da das Ablösen eines solchen Abhängigkeitsschwerpunkts nur mit erheblichem Aufwand realisierbar ist. Weit vor der endgültigen Ablösung ergibt sich der Vorteil, dass die Leihe nun ein eigenes kleines Domänenmodell hat, das wesentlich einfacher zu verstehen und weiterzuentwickeln ist. Das kann sofort zu einem Produktivitätsvorteil führen, um neue Anforderungen in dem Bounded Context „Leihe“ umzusetzen. Es kann durchaus sein, dass der Aufwand für eine vollständige Migration keine ausreichende Produktivitätssteigerung zur Folge hat und es daher nie zur vollständigen Ablösung des Buchsystems kommt.

Strategic Design

Bisher standen nur einzelne Bounded Contexts im Fokus, aber ein Bounded Context kann nicht in Isolation existieren. Wenn Leser nach einem Buch suchen, sollte sich idealerweise ein Leihvorgang anschließen. Dazu ist eine Schnittstelle notwendig, mit der die Suche der Leihe Informationen übergeben kann, sodass Büchereibesucher das richtige Buch ausleihen können.

Domain-driven Design behandelt Beziehungen zwischen Bounded Contexts im Strategic Design. In diesem Bereich existieren unterschiedliche Patterns: Bei Customer/Supplier kann das Team, das für „Leihe“ verantwortlich ist, dem Team, das für „Suche“ verantwortlich ist, Anforderungen geben, die letzteres Team umsetzen muss.

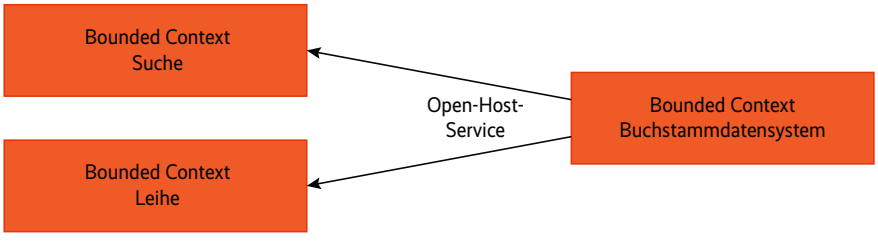
Beim Open-Host-Service würde hingegen die „Suche“ anderen Systemen eine generische Schnittstelle anbieten, die auch die „Leihe“ nutzen kann. Der Open-Host-Service hält dadurch die Anzahl der Schnittstellen gering. Er hat organisatorische Auswirkung: Wer den Service verantwortet, nimmt zwar Anforderungen anderer Teams entgegen, aber am Ende entscheidet das Team, ob es die Anforderungen umsetzt oder die Anforderungen in der generischen Schnittstelle nicht berücksichtigt, weil sie zu speziell sind. Das letzte Wort liegt dazu bei dem Team, das den Open-Host-Service verantwortet.

Strategic Design scheint Beziehungen zwischen Teams festzulegen. In Wirklichkeit handelt es sich jedoch um Beziehungen zwischen Kontexten: Das Team, das einen Bounded Context verantwortet, übernimmt die damit verbundenen Rechte und Pflichten. Das für einen Bounded Context verantwortliche Team muss nicht konstant sein, da die Änderungsschwerpunkte über den Verlauf eines Projekts nicht gleich bleiben. Um dennoch die Teams gleichmäßig auszulasten, müssen sie Bounded Contexts tauschen.

Beispielsweise kann anfangs je ein Team das Buchsystem, die Leihe und die Suche verantworten. Das Buchsystem ist ein Open-Host-Service, sodass das „Suche“-Team und das „Leihe“-Team Anforderungen stellen können, die das „Buchsystem“-Team umsetzt, wenn sie in den Plan des Teams passen. Letzteres Team könnte nun die Verantwortung für das „Buchsystem“ an das „Leihe“-Team übergeben. Das Buchsystem bleibt dabei ein Open-Host-Service. Das „Leihe“-Team muss somit die Anforderungen von „Suche“ an das „Buchsystem“ entgegennehmen, bewerten und gegebenenfalls im „Buchsystem“ umsetzen, weil es die Pflichten mit der Übernahme des „Buchsystems“ übernommen hat.

Kartografierte Verflechtungen

Die Beziehungen zwischen den Bounded Contexts wie in den Abbildungen 2 und 3 stellen eine Context Map dar, wie sie oft zur Darstellung des Sollzustand eines Systems dient. In einem IT-System oder einer IT-Landschaft gibt es jedoch bereits Domänenmodelle, zwischen denen Beziehungen existieren. Eine Context Map lässt sich zur Analyse des Ist-Zustands nutzen. Damit sind Bounded Contexts wie ein Buchsystem die Regel: Sie haben zwar ein Domänenmodell, aber entsprechen keiner Ubi-



Buchsystem, Suche und Leihe und die Beziehungen (Abb. 3)

quitous Language, weil sie ohne Rücksicht auf die Erkenntnisse von DDD entstanden sind. Bei einem Sollmodell entsprechen mehr Bounded Contexts dem DDD-Ideal, wenn eine Migration in Richtung DDD geplant ist.

Bei den Beziehungen zwischen den Bounded Contexts können andere Patterns auftreten als im Domain-driven Design vorgesehen. Beispielsweise kann das Buchsystem zwar eine generische API anbieten, aber das „Buchsystem"-Team hat nicht das letzte Wort, welche Änderungen tatsächlich in die API fließen. Ein Grund dafür könnte sein, dass ein anderes Team politisch mächtiger ist und seine Anforderungen auf jeden Fall durchsetzen kann. Die Abhängigkeiten und Schnittstellen auf der Software-Ebene sind dann genau so wie bei einem Open-Host-Service, aber die Beziehungen auf Team-Ebene unterscheiden sich.

Obwohl die Unterschiede als Detail erscheinen mögen, haben sie potenziell erhebliche Konsequenzen. Wenn das Team nicht die Möglichkeit hat, die Entwicklung der API zu steuern, kann es passieren, dass die API irgendwann schwer zu verstehen und zu benutzen ist, weil sie keinem eindeutigen Konzept mehr folgt. Weil Abweichungen von den DDD-Pattern zu Herausforderungen führen können, ist es wichtig, die Beziehungen präzise zu dokumentieren und Abweichungen gegebenenfalls zu untersuchen. Es kommt durchaus vor, dass Abweichungen sinnvoll sind oder nicht zu Problemen führen. In dem Fall ist es nicht unbedingt notwendig, die Beziehungen zu ändern.

Drei Ebenen

Auf drei unterschiedlichen Ebenen existieren Bounded Contexts: dem Datenmodell, der Ubiquitous Language und den Teams. Bounded Contexts haben spezialisierte Datenmodelle. In jedem der Datenmodelle kann es gleich benannte Domänenobjekte wie „Buch" geben, die aber meistens keine redundanten Daten enthalten, sondern unterschiedliche fachliche Aspekte eines solchen Domänenobjekts modellieren. Die Aufteilung in mehrere kleine Datenmodelle ist ein wesentlicher Grund, warum DDD-Entwürfe einfacher sein können. Gleichzeitig ist es oft eine Hürde, zu akzeptieren, dass unterschiedliche Datenmodelle Informationen über ein Domänenobjekt wie ein Buch haben.

Die Ubiquitous Language stellt eine Sammlung von Begriffen dar, die Domänenexperten nutzen und sich in der Software wiederfinden sollten. Für die Leihe und die Suche ist das Buch etwas anderes. Mit einem Bounded Context können mehrdeutige Begriffe eindeutig werden. Ein Buchsystem durchbricht dieses Design, weil es alle Informationen zu dem Buch sammelt und beide Ubiquitous Languages umfassen muss. Es zeigt somit, dass ein Bounded Context mit einem Domänenmodell nicht unbedingt ebenfalls einer für eine Ubiquitous Language sein muss.

Ein Bounded Context definiert die Rechte und Pflichten des Teams, das dafür verantwortlich ist. Wenn ein anderes Team ihn übernimmt, erhält es damit auch die Rechte und Pflichten. Weil ein Bounded Context demnach verschiedene Aspekte umfasst, erscheint es sinnvoll, Bounded Contexts für Domänenmodelle und Bounded Contexts für die Ubiquitous Languages zu unterscheiden – auch wenn Domain-driven Design das nicht direkt vorsieht.

Schließlich ist es wichtig, die Aufteilung in Bounded Contexts zu analysieren und den Ist-Zustand klar von einem möglichen Sollzustand abzugrenzen. In bereits vorhandenen Legacy-Systemen existieren auf jeden Fall Bounded Contexts für Domänenmodelle, aber solche für Ubiquitous Languages finden sich gegebenenfalls nicht wieder, weil Systeme Daten aus mehreren Ubiquitous Languages verwalten. Gerade diese Erkenntnisse sind interessant, um Systeme zu verbessern sowie weiterzuentwickeln. (ane@ix.de)

Literatur

Eric Evans; Domain-driven Design Referenz (ddd-referenz.de)

Eberhard Wolff

arbeitet als Fellow bei INNOQ. Er ist seit mehr als 15 Jahren als Architekt und Berater tätig – oft an der Schnittstelle zwischen Business und Technologie. Sein technologischer Schwerpunkt liegt auf modernen Architektur-Ansätzen – Cloud, Continuous Delivery, DevOps, Microservices oder NoSQL spielen oft eine Rolle.