KAPITEL 3

Die Komplexität einer Domain im Griff behalten

Wie Sie im vorherigen Kapitel gesehen haben, ist es für den Erfolg eines Projekts von entscheidender Bedeutung, dass Sie eine Ubiquitous Language aufbauen, die Sie für die Kommunikation zwischen allen Beteiligten nutzen können – von der Softwareentwicklung bis zu den Domänenexperten. Die Sprache sollte die mentalen Modelle der Domänenexperten zu den inneren Abläufen und den zugrunde liegenden Prinzipien der Fachdomäne widerspiegeln.

Da es unser Ziel ist, die Ubiquitous Language zu nutzen, um Entscheidungen rund um das Softwaredesign zu unterstützen, muss die Sprache klar und konsistent sein. Sie sollte frei von Mehrdeutigkeiten, impliziten Annahmen und belanglosen Details sein. Aber auf Firmenebene können die mentalen Modelle der Domänenexperten selbst inkonsistent sein. Verschiedene Domänenexperten können unterschiedliche Modelle derselben Fachdomäne nutzen. Schauen wir uns ein Beispiel an.

Inkonsistente Modelle

Kehren wir zurück zu dem Beispiel einer Telemarketing-Firma aus Kapitel 2. Die Marketingabteilung des Unternehmens sorgt durch Onlinewerbung für Leads. Die Sales-Abteilung ist dann dafür verantwortlich, die potenziellen Kunden dazu zu bringen, die Produkte oder Services der Firma zu kaufen. Diesen Ablauf zeigt Abbildung 3-1.

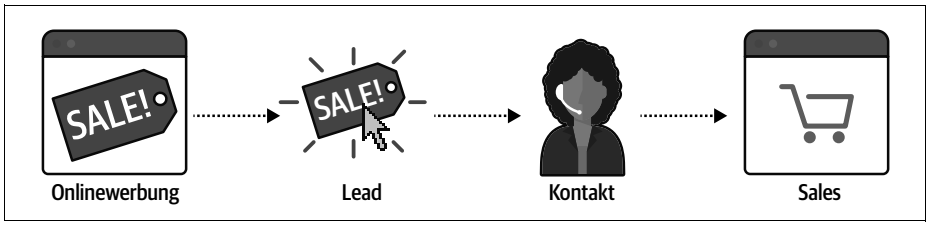

Abbildung 3-1: Beispiel der Fachdomäne einer Telemarketing-Firma

Schauen wir uns die Sprache der Domänenexperten an, machen wir eine interessante Beobachtung. Der Begriff *Lead* hat für die Marketing- und für die Sales-Abteilung unterschiedliche Bedeutungen:

Marketingabteilung
: Für die Marketingabteilung ist ein Lead der Hinweis darauf, dass jemand an einem ihrer Produkte interessiert ist. Das Ereignis, die Kontaktdaten des potenziellen Kunden zu erhalten, wird als Lead bezeichnet.

Sales-Abteilung
: Im Kontext der Sales-Abteilung ist ein Lead eine viel komplexere Entität. Sie steht für den gesamten Lebenszyklus des Verkaufsprozesses. Also ist es nicht mehr ein reines Ereignis, sondern ein länger laufender Prozess.

Wie formulieren wir eine Ubiquitous Language für diese Telemarketing-Firma?

Einerseits wissen wir, dass die Ubiquitous Language konsistent sein muss – jeder Begriff sollte nur eine Bedeutung haben. Andererseits muss die Ubiquitous Language auch die mentalen Modelle der Domänenexperten widerspiegeln. In diesem Fall ist das mentale Modell des *Lead* zwischen den Domänenexperten der Sales- und der Marketingabteilung inkonsistent.

Diese Mehrdeutigkeit ist bei einer direkten Kommunikation zwischen zwei Menschen gar nicht so schlimm. Unterhaltungen zwischen Personen aus verschiedenen Abteilungen können zwar etwas schwieriger werden, aber Menschen fällt es nicht so schwer, sich die exakte Bedeutung des Inhalts zu erschließen.

Weitaus schwieriger ist es aber, solch ein divergentes Modell der Fachdomäne in Software umzusetzen. Quellcode kommt mit Mehrdeutigkeiten nicht sehr gut klar. Wollen wir das kompliziertere Modell der Sales-Abteilung für das Marketing nutzen, würde das für zusätzliche unnötige Komplexität sorgen – viel mehr Details und Verhalten, als das Marketingteam für das Optimieren von Werbekampagnen brauchen. Würden wir hingegen versuchen, das Modell für Sales an der Sichtweise des Marketings auszurichten, würde es nicht zu den Anforderungen der Sales-Subdomain passen, weil es zu einfach wäre für das Managen und Optimieren des Verkaufsprozesses. Im ersten Fall hätten wir eine zu weit gefasste, im zweiten Fall eine zu wenig entwickelte Lösung.

Wie kommen wir nun aus dieser Zwickmühle heraus?

Die klassische Lösung für dieses Problem ist, ein einzelnes Modell zu entwerfen, das für alle Arten von Problemen genutzt werden kann. Solche Modelle führen zu umfangreichen *Entity Relationship Diagrams* (ERDs), die ganze Bürowände bedecken. Ist Abbildung 3-2 ein effektives Modell?

Wie das Sprichwort sagt: »Von allem etwas, aber nichts richtig.« Solche Modelle sollen alles abdecken, sind aber letztendlich nirgendwo effektiv. Egal was Sie tun – Sie sehen sich immer einer Komplexität gegenüber: die Komplexität, überflüssige Details auszufiltern, die Komplexität, das zu finden, was Sie brauchen, und vor allem die Komplexität, die Daten in einem konsistenten Zustand zu halten.

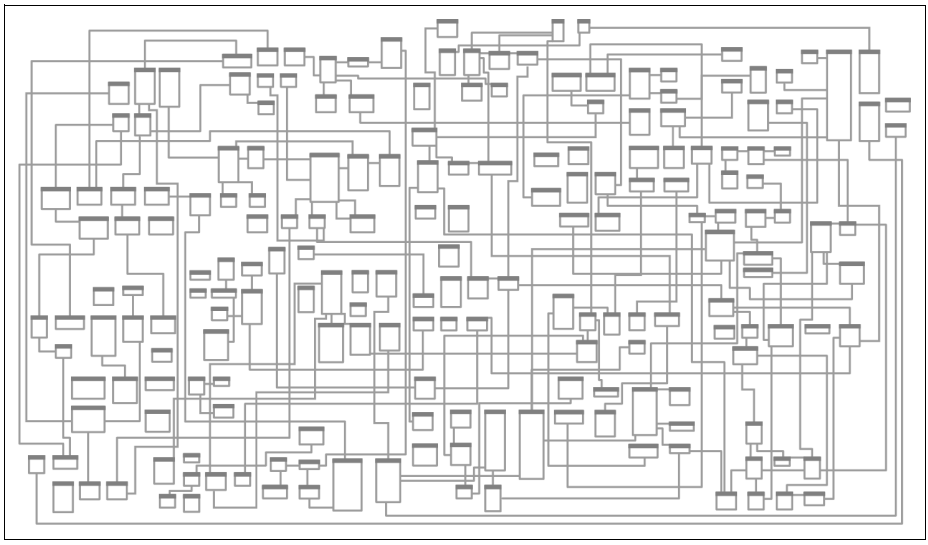

Abbildung 3-2: Unternehmensweites Entity Relationship Diagram

Eine andere Lösung bestünde darin, den problematischen Begriff mit einer Definition des Kontexts zu ergänzen: »Marketing-Lead« und »Sales-Lead«. Das würde ein Implementieren der beiden Modelle in Code ermöglichen. Aber dieser Ansatz hat zwei große Nachteile. Erstens sorgt er für kognitive Last. Wann sollte welches Modell verwendet werden? Je ähnlicher sich die Implementierungen der miteinander im Konflikt stehenden Modelle sind, desto leichter macht man einen Fehler. Und zweitens würde die Implementierung des Modells nicht mit der Ubiquitous Language übereinstimmen. Niemand würde die Präfixe in Unterhaltungen verwenden. Menschen brauchen diese zusätzliche Information nicht – sie können sich auf den Kontext des Gesprächs verlassen.

Wenden wir uns dem Pattern im Domain-Driven Design zu, mit dem solche Szenarien angegangen werden – dem Bounded Context Pattern.

Was ist ein Bounded Context?

Die Lösung im Domain-Driven Design ist trivial: Teilen Sie die Ubiquitous Language in mehrere kleinere Sprachen auf und weisen Sie jede davon dem expliziten Kontext zu, in dem sie angewendet werden kann – ihrem *Bounded Context* (dem gebundenen oder begrenzten Kontext).

Im vorherigen Beispiel können wir zwei Bounded Contexts erkennen: Marketing und Sales. Der Begriff *Lead* existiert in beiden Kontexten, wie in Abbildung 3-3 zu sehen ist. Solange er in jedem Bounded Context nur eine einzige Bedeutung besitzt, ist auch jede feingranulare Ubiquitous Language konsistent und orientiert sich an den mentalen Modellen der Domänenexperten.

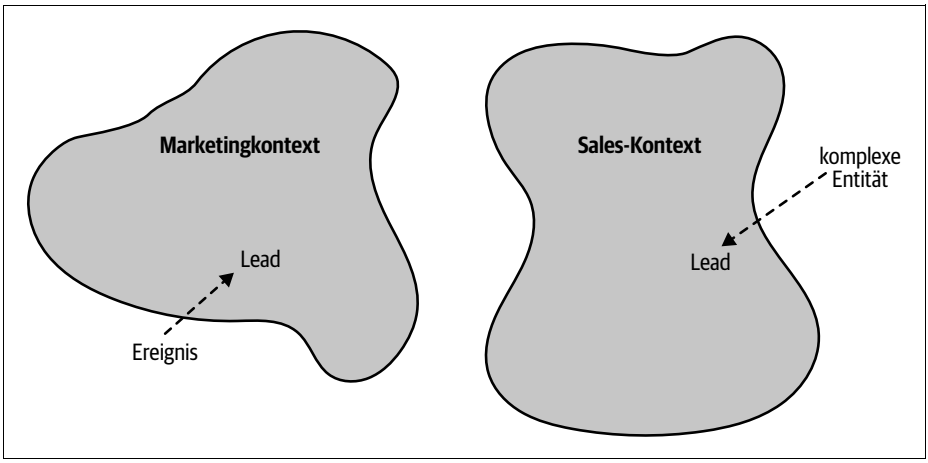

Abbildung 3-3: Inkonsistenzen in der Ubiquitous Language angehen, indem man sie in Bounded Contexts aufteilt

In gewissem Maße sind Terminologiekonflikte und implizite Kontexte ein inhärenter Teil jedes Unternehmens mit einer gewissen Größe. Mit dem Bounded Context Pattern werden die Kontexte als expliziter und integraler Teil der Fachdomäne modelliert.

Modellgrenzen

Wie schon im vorherigen Kapitel besprochen, ist ein Modell keine Kopie der Realität, sondern ein Konstrukt, das uns dabei hilft, ein komplexes System besser zu verstehen. Das Problem, das es lösen soll, ist ein inhärenter Teil eines Modells – sein Zweck. Ein Modell kann nicht ohne eine Begrenzung existieren – es wird sich sonst immer weiter ausbreiten, bis es eine Kopie der realen Welt sein würde. Das macht das Definieren der Grenzen eines Modells – seine Bounded Contexts – zu einem intrinsischen Teil des Modellierungsprozesses.

Kehren wir zu dem Beispiel zurück, Karten als Modelle zu betrachten. Wir haben gesehen, dass jede Karte ihren spezifischen Kontext besitzt – Luftraum, Meer, Boden, U-Bahn und so weiter. Eine Karte ist nur im Rahmen ihres spezifischen Zwecks nützlich.

So wie eine U-Bahn-Karte für die nautische Navigation nutzlos ist, kann eine Ubiquitous Language in einem Bounded Context für den Rahmen eines anderen Bounded Context vollkommen irrelevant sein. Bounded Contexts definieren die Anwendbarkeit einer Ubiquitous Language und des durch sie repräsentierten Modells. Sie erlauben das Definieren unterschiedlicher Modelle für unterschiedliche Problemdomänen. Mit anderen Worten: Bounded Contexts sind die Konsistenzgrenzen von Ubiquitous Languages. Die Terminologie, Prinzipien und Business-Regeln einer Sprache sind nur innerhalb ihres Bounded Context konsistent.

Verbesserte Ubiquitous Language

Bounded Contexts ermöglichen uns, die Definition einer Ubiquitous Language zu vervollständigen. Eine Ubiquitous Language ist nicht dahin gehend »ubiquitär«, dass sie in der gesamten Organisation »allgegenwärtig« eingesetzt und angewendet werden sollte. Eine Ubiquitous Language ist also nicht universell.

Stattdessen ist eine Ubiquitous Language nur innerhalb der Grenzen ihres Bounded Context ubiquitär. Die Sprache ist darauf fokussiert, nur das Modell zu beschreiben, das durch den Bounded Context umschlossen wird. Da ein Modell ohne ein Problem, das es lösen soll, nicht existieren kann, lässt sich eine Ubiquitous Language nicht ohne einen expliziten Kontext ihrer Anwendbarkeit definieren oder einsetzen.

Gültigkeitsbereich eines Bounded Context

Das Beispiel zu Beginn dieses Kapitels hat eine inhärente Grenze der Fachdomäne aufgezeigt. Verschiedene Domänenexperten nutzten miteinander in Konflikt stehende mentale Modelle der gleichen Business Entity. Um die Fachdomäne zu modellieren, mussten wir das Modell aufteilen und einen strikten Anwendbarkeitskontext für jedes feingranulare Modell definieren – dessen Bounded Context.

Die Konsistenz der Ubiquitous Language hilft nur dabei, die äußersten Grenzen dieser Sprache zu definieren. Sie kann nicht größer sein, weil es dort dann inkonsistente Modelle und Begriffe geben wird. Aber wir können die Modelle trotzdem weiter in kleinere Bounded Contexts unterteilen, wie in Abbildung 3-4 gezeigt.

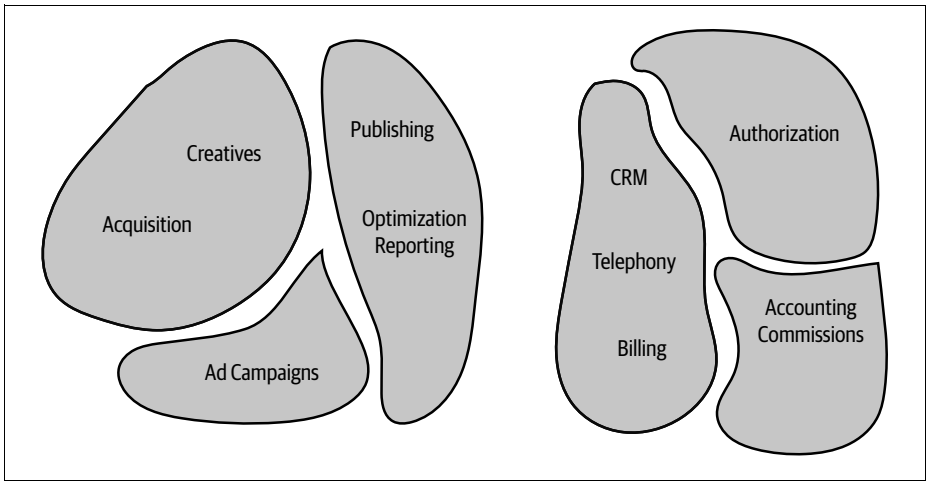

Abbildung 3-4: Kleinere Bounded Contexts

Das Definieren des Gültigkeitsbereichs einer Ubiquitous Language – ihres Bounded Context – ist eine strategische Designentscheidung. Grenzen können weit sein und den inhärenten Kontexten der Fachdomäne folgen, oder eng und die Fachdomäne weiter in kleinere Problemdomänen unterteilen.

Die Größe eines Bounded Context selbst ist kein entscheidender Faktor. Modelle müssen nicht notwendigerweise groß oder klein sein. Sie müssen nützlich sein. Je weiter gefasst die Grenze der Ubiquitous Language ist, desto schwerer ist es, die Sprache konsistent zu halten. Es kann hilfreich sein, eine große Ubiquitous Language in kleinere, besser handhabbare Problemdomänen zu unterteilen, aber es kann auch nach hinten losgehen. Je kleiner sie sind, desto mehr Integrations-Overhead bringt das Design mit sich.

Daher sollte die Entscheidung für die Größe Ihrer Bounded Contexts von der spezifischen Problemdomäne abhängen. Manchmal wird der Einsatz einer weiten Grenze klarer sein, während es in einer anderen Situation sinnvoller sein mag, sie weiter zu unterteilen.

Zu den Gründen, kleiner geschnittene Bounded Contexts aus einem größeren zu extrahieren, gehört das Schaffen neuer Entwicklungsteams oder der Umgang mit nicht funktionalen Anforderungen des Systems – wenn Sie zum Beispiel die Entwicklungslebenszyklen mancher der Komponenten, die ursprünglich in einem einzelnen Bounded Context untergebracht waren, voneinander trennen müssen. Ein weiterer häufig vorkommender Grund ist die Möglichkeit, eine Funktionalität unabhängig vom Rest des Bounded Context skalieren zu können.

Sorgen Sie daher dafür, dass Ihre Modelle nützlich bleiben, und stimmen Sie die Größen der Bounded Contexts auf Ihre Business-Anforderungen und organisatorischen Gegebenheiten ab. Achten Sie darauf, eine kohärente Funktionalität nicht in mehrere Bounded Contexts zu unterteilen. Das behindert das unabhängige Weiterentwickeln jedes einzelnen Kontexts. Stattdessen werden die gleichen Business-Anforderungen und -Änderungen simultan alle Bounded Contexts betreffen und ein gleichzeitiges Ausliefern der Änderungen erfordern. Um solch eine ineffiziente Dekomposition zu verhindern, greifen Sie auf die Faustregel aus Kapitel 1 zurück, mit der Sie Subdomains finden: Suchen Sie nach Gruppen kohärenter Use Cases, die mit den gleichen Daten arbeiten, und vermeiden Sie hier ein Aufteilen in mehrere Bounded Contexts.

Wir werden das fortlaufende Optimieren der Grenzen von Bounded Contexts noch weiter in Kapitel 8 und Kapitel 10 besprechen.

Bounded Contexts versus Subdomains

In Kapitel 2 haben wir gesehen, dass eine Fachdomäne aus mehreren Subdomains besteht. In diesem Kapitel haben wir uns bisher mit der Idee beschäftigt, eine Fachdomäne in eine Reihe von kleineren Problemdomänen oder Bounded Contexts zu

unterteilen. Auf den ersten Blick scheinen die beiden Methoden zum Aufteilen von Fachdomänen redundant zu sein. Aber das ist nicht der Fall. Schauen wir uns an, warum wir beide Abgrenzungen benötigen.

Subdomains

Um die Business-Strategie einer Firma zu verstehen, müssen wir ihre Fachdomäne analysieren. Nach der Methode von Domain-Driven Design gehört es zur Analysephase, die verschiedenen Subdomains (Core, Supporting und Generic) zu ermitteln. So funktioniert die Organisation, und so plant sie ihre Wettbewerbsstrategie.

Wie Sie in Kapitel 1 gelernt haben, setzt sich eine Subdomain aus einer Gruppe von miteinander in Verbindung stehenden Use Cases zusammen. Die Use Cases werden durch die Fachdomäne und die Systemanforderungen definiert. In der Softwareentwicklung definieren wir nicht die Anforderungen – das ist die Aufgabe des Business. Stattdessen analysieren wir die Fachdomäne, um die Subdomains zu ermitteln.

Bounded Contexts

Bounded Contexts wiederum werden designt. Die Wahl der Modellgrenzen ist eine strategische Designentscheidung. Wir legen fest, wie die Fachdomäne in kleinere, besser handhabbare Problemdomänen unterteilt wird.

Die Verbindung zwischen Subdomains und Bounded Contexts

Theoretisch könnte ein einzelnes Modell die gesamte Fachdomäne umfassen (das wäre aber impraktikabel). Diese Strategie könnte für ein kleines System funktionieren, wie es Abbildung 3-5 zeigt.

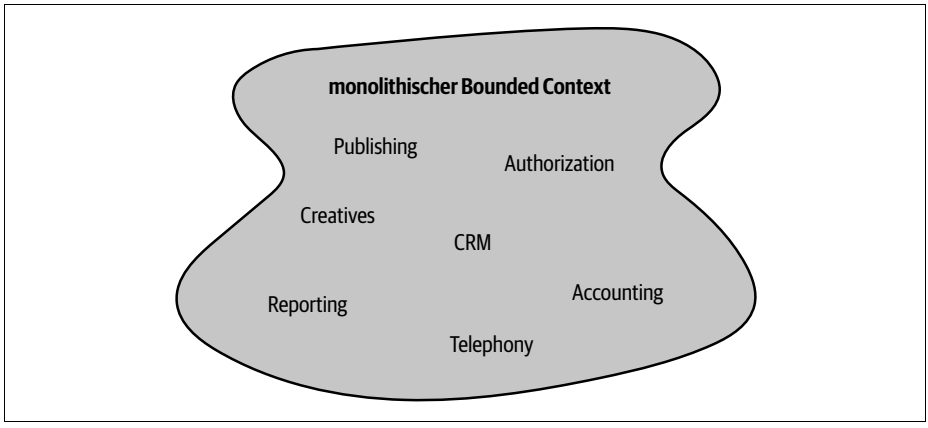

Abbildung 3-5: Monolithischer Bounded Context

Tauchen miteinander in Konflikt stehende Modelle auf, können wir uns an den mentalen Modellen der Domänenexperten orientieren und das System in Bounded Contexts unterteilen, wie in Abbildung 3-6 zu sehen ist.

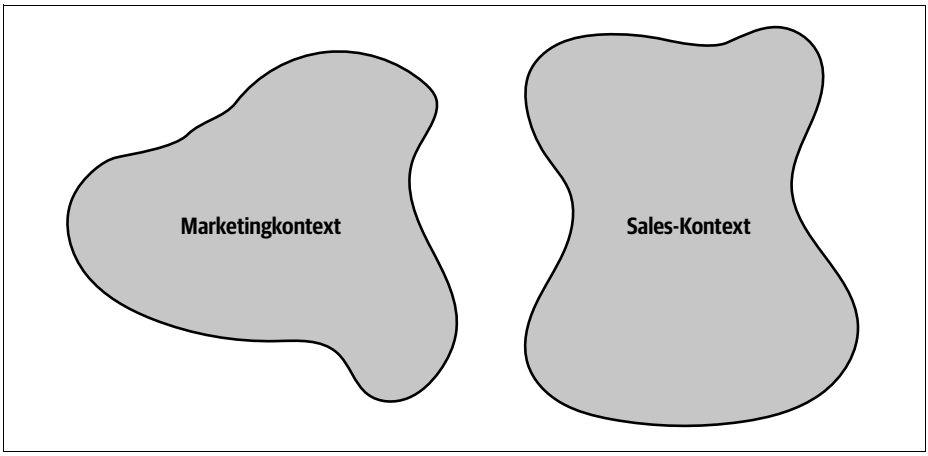

Abbildung 3-6: Bounded Contexts werden durch die Konsistenz der Ubiquitous Language definiert.

Sind die Modelle immer noch groß und nur schwer zu betreuen, können wir sie in noch kleinere Bounded Contexts unterteilen – zum Beispiel durch einen Bounded Context für jede Subdomain (siehe Abbildung 3-7).

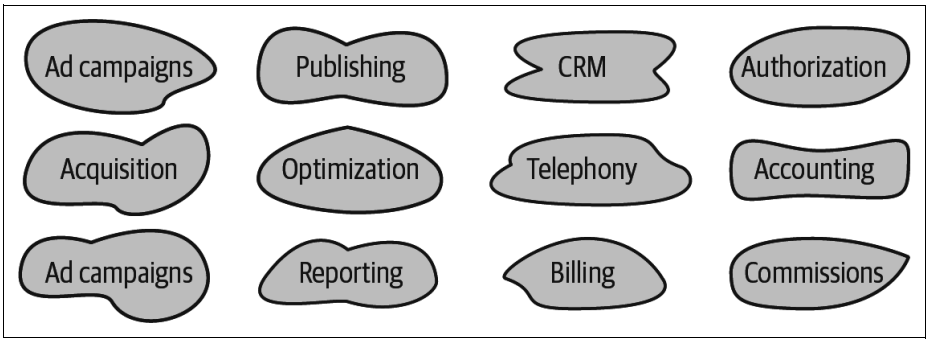

Abbildung 3-7: Bounded Contexts, die auf die Grenzen der Subdomains abgestimmt sind

Das Ganze ist aber auf jeden Fall eine Designentscheidung. Wir designen diese Grenzen als Teil der Lösung.

Eine Eins-zu-eins-Beziehung zwischen Bounded Contexts und Subdomains ist in manchen Szenarien ausgesprochen vernünftig. In anderen können aber andere Aufteilungsstrategien passender sein.

Sie müssen sich bewusst machen, dass Subdomains ermittelt und Bounded Contexts designt werden.[1] Die Subdomains werden durch die Business-Strategie definiert. Aber wir können die Softwarelösung und ihre Bounded Contexts so entwerfen, dass wir den spezifischen Kontext und die Beschränkungen des Projekts berücksichtigen.

Schließlich haben Sie in Kapitel 1 gelernt, dass ein Modell dazu gedacht ist, ein spezifisches Problem zu lösen. In manchen Fällen kann es von Vorteil sein, mehrere Modelle des gleichen Konzepts parallel einzusetzen, um verschiedene Probleme zu lösen. Beschränken Sie das Design auf Eins-zu-eins-Beziehungen zwischen Bounded Contexts, würden Sie diese Flexibilität verlieren und uns dazu zwingen, ein einzelnes Modell einer Subdomain in ihrem Bounded Context einzusetzen.

Grenzen

Ruth Malan schrieb, dass es beim Architekturdesign inhärent um Grenzen geht:

> Architekturdesign ist Systemdesign. Systemdesign ist kontextabhängiges Design – es geht inhärent um Grenzen (was ist drin, was draußen, was umfasst es, was bewegt sich dazwischen?) und um Kompromisse. Es formt das, was draußen ist, genauso wie das, was drinnen ist.[2]

Das Bounded Context Pattern ist das Werkzeug von Domain-Driven Design zum Definieren von physischen und Zuständigkeitsgrenzen.

Physische Grenzen

Bounded Contexts dienen nicht nur als Modellgrenzen, sondern auch als physische Grenzen der Systeme, auf denen sie implementiert werden. Jeder Bounded Context sollte als einzelner Service oder einzelnes Projekt implementiert werden – also unabhängig von anderen Bounded Contexts implementiert, weiterentwickelt und versioniert werden.

Klare physische Grenzen zwischen den Bounded Contexts ermöglichen uns, jeden Bounded Context mit dem Technologie-Stack zu implementieren, der am besten dafür geeignet ist.

Wie weiter oben besprochen, kann ein Bounded Context mehrere Subdomains enthalten. In solch einem Fall ist der Bounded Context eine physische Grenze, während jede seiner Subdomains eine logische Grenze bedeutet. Logische Grenzen haben in den verschiedenen Programmiersprachen auch verschiedene Namen: Namensräume, Module oder Pakete.

---

1 Es gibt dabei eine erwähnenswerte Ausnahme. Abhängig von der Organisation, in der Sie arbeiten, haben Sie vielleicht zwei Rollen und sind sowohl für die Software- als auch für die Business-Entwicklung zuständig. Dann haben Sie die Möglichkeit, auf Softwaredesign (Bounded Contexts) und Business-Strategie (Subdomains) einzuwirken. Daher konzentrieren wir uns hier im Rahmen unserer Diskussion nur auf die Softwareentwicklung.

2 Bredemeyer Consulting, »What Is Software Architecture«, *https://www.bredemeyer.com/who.htm*, gelesen am 22. September 2021.

Zuständigkeitsgrenzen

Studien zeigen, dass gute Zäune für gute Nachbarschaft sorgen. In Softwareprojekten können wir die Modellgrenzen – Bounded Contexts – für eine friedliche Koexistenz von Teams nutzen. Das Aufteilen von Arbeit zwischen Teams ist eine weitere strategische Entscheidung, die mithilfe des Bounded Context Pattern umgesetzt werden kann.

Ein Bounded Context sollte von nur einem Team implementiert, weiterentwickelt und gewartet werden. Es können nicht zwei Teams am gleichen Bounded Context arbeiten. Diese Aufteilung eliminiert implizite Annahmen, die Teams eventuell über die Modelle anderer treffen. Stattdessen müssen sie Kommunikationsprotokolle für das Integrieren ihrer Modelle und Systeme explizit definieren.

Es ist wichtig, darauf hinzuweisen, dass die Beziehung zwischen Teams und Bounded Contexts unidirektional ist: Ein Bounded Context sollte von nur einem Team betreut werden. Aber ein einzelnes Team kann für mehrere Bounded Contexts zuständig sein, wie in Abbildung 3-8 gezeigt.

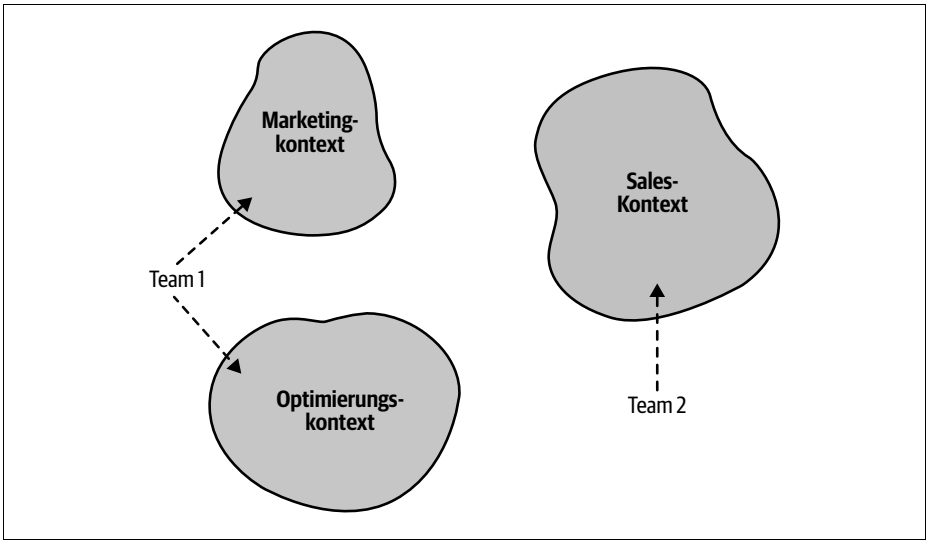

Abbildung 3-8: Team 1 arbeitet an den Bounded Contexts für Marketing und Optimization, während Team 2 am Bounded Context für Sales arbeitet.

Bounded Contexts in der Realität

In einem meiner Kurse zum Domain-Driven Design stellte ein Teilnehmer einmal fest: »Sie haben gesagt, dass es bei DDD um das Ineinklangbringen des Softwaredesigns mit den Fachdomänen geht. Aber wo sind die Bounded Contexts in der Realität? In Fachdomänen gibt es keine Bounded Contexts.«

Es stimmt – Bounded Contexts sind nicht so offensichtlich wie Fachdomänen oder Subdomains, aber es gibt sie genau so, wie es die mentalen Modelle der Domänenexperten gibt. Sie müssen nur bewusst darauf achten, wie Domänenexperten über die verschiedenen Business Entities und die zugehörigen Prozesse nachdenken.

Ich möchte dieses Kapitel damit schließen, dass es nicht nur Bounded Contexts gibt, wenn wir Fachdomänen in Software modellieren, sondern dass die Idee, unterschiedliche Modelle in unterschiedlichen Kontexten zu verwenden, in der Realität ganz allgemein verbreitet ist.

Semantic Domains

Man kann sagen, dass die Bounded Contexts von Domain-Driven Design auf der lexikografischen Idee der Semantic Domains (*https://oreil.ly/ugv75*) basieren. Eine *Semantic Domain* ist als Bedeutungsbereich und die dafür verwendeten Wörter definiert. So haben beispielsweise die Wörter *Monitor*, *Port* und *Prozessor* in den Semantic Domains der Software- und der Hardwareentwicklung unterschiedliche Bedeutungen.

Ein recht spezielles Beispiel für unterschiedliche Semantic Domains ist die Bedeutung des Worts *Tomate*.

Laut botanischer Definition ist eine Frucht der Weg für eine Pflanze, ihre Samen zu verteilen. Eine Frucht sollte an der Blüte der Pflanze wachsen und mindestens einen Samen enthalten. Gemüse ist andererseits der allgemeine Begriff für alle anderen essbaren Teile einer Pflanze: Wurzeln, Stamm und Blätter. Laut dieser Definition *ist die Tomate eine Frucht*.

Diese Definition ist aber im Kontext der Kulinarik nicht sehr nützlich. Dort werden Früchte und Gemüse anhand ihrer Geschmacksprofile definiert. Eine Frucht hat eine weiche Textur, ist entweder süß oder sauer und kann roh genossen werden, während Gemüse eine festere Textur besitzt, langweiliger schmeckt und oft gekocht werden muss. Laut dieser Definition *ist die Tomate ein Gemüse*.

Im *Bounded Context der Botanik* ist die Tomate also eine Frucht, während sie im *Bounded Context der Kulinarik* ein Gemüse ist. Damit sind wir aber noch nicht am Ende.

Im Jahr 1883 führten die USA ein Steuer von 10 % auf importiertes Gemüse, nicht aber auf Früchte ein. Die botanische Definition der Tomate als Frucht erlaubte das Importieren von Tomaten in die USA, ohne eine Steuer zahlen zu müssen. Um dieses Schlupfloch zu schließen, entschied der United States Supreme Court im Jahr 1893, dass die Tomate ein Gemüse sei. Daher ist die Tomate im *Bounded Context der Besteuerung* ein Gemüse.

Und mein Freund Romeu Moura sagt, dass die Tomate im *Bounded Context von Theatervorführungen* ein Feedback-Mechanismus sei.

Wissenschaft

Der Historiker Yuval Noah Harari schreibt: »Die Wissenschaft ist sich im Allgemeinen darin einig, dass keine Theorie zu 100 % korrekt ist. Daher ist die eigentliche Wissensprüfung nicht die Wahrheit, sondern die Nützlichkeit.« Mit anderen Worten: Keine wissenschaftliche Theorie ist in allen Fällen korrekt. Unterschiedliche Theorien sind in unterschiedlichen Kontexten nützlich.

Diese Idee lässt sich anhand der unterschiedlichen Modelle der Gravitation zeigen, die von Sir Isaac Newton und Albert Einstein vorgestellt wurden. Nach Newtons Bewegungsgesetzen sind Raum und Zeit absolut. Sie bilden die Bühne, auf der die Bewegung von Objekten stattfindet. In Einsteins Relativitätstheorie sind Raum und Zeit nicht mehr länger absolut, sondern sie unterscheiden sich je nach Beobachter.

Auch wenn diese zwei Modelle widersprüchlich gesehen werden können, sind beide in ihren passenden (gebundenen) Kontexten nützlich.

Einen Kühlschrank kaufen

Schauen wir uns abschließend noch ein bodenständigeres Beispiel von realen Bounded Contexts an. Was sehen Sie in Abbildung 3-9?

Abbildung 3-9: Ein Stück Pappe

Ist das nur ein Stück Pappe? Nein, es ist ein Modell. Ein Modell für den Kühlschrank Siemens KG86NAI31L. Suchen Sie im Internet danach, wird Ihnen auffallen, dass die Pappe nun wirklich nicht wie dieser Kühlschrank aussieht. Sie hat keine Tür, und selbst die Farbe ist anders.

Das mag zwar stimmen, ist aber nicht entscheidend. Wie wir besprochen haben, soll ein Modell eine Entität aus der Realität nicht kopieren. Stattdessen sollte es einen Zweck haben – ein Problem darstellen, das es lösen soll. Daher ist die korrekte Frage bezüglich der Pappe, bei welchem Problem dieses Modell hilft.

In unserer Wohnung haben wir keine normale Tür zur Küche. Die Pappe wurde so zurechtgeschnitten, dass sie die Breite und Tiefe des Kühlschranks besitzt. Das zu lösende Problem: Es soll geprüft werden, ob der Kühlschrank durch die Küchentür passt (siehe Abbildung 3-10).

Abbildung 3-10: Das Pappmodell im Durchgang zur Küche

Obwohl die Pappe auch nicht annähernd wie der Kühlschrank aussieht, hat sie sich als außerordentlich nützlich erwiesen, als wir entscheiden mussten, ob wir dieses Modell kaufen oder auf ein kleineres zurückgreifen. Es hätte sicherlich viel Spaß gemacht, ein 3-D-Modell des Kühlschranks zu bauen. Aber hätte es das Problem effizienter gelöst als die Pappe? Nein. Wenn die Pappe passt, würde auch das 3-D-Modell passen und umgekehrt. Auf die Softwareentwicklung bezogen, würde das Bauen des 3-D-Modells einem massiven Overengineering entsprechen.

Aber was ist mit der Höhe des Kühlschranks? Was, wenn die Grundfläche zwar passen würde, der Kühlschrank aber zu hoch für die Tür wäre? Würde das den Zusammenbau eines 3-D-Modells rechtfertigen? Nein. Das Problem ließe sich viel schneller und einfacher mit einem einfachen Maßband lösen, mit dem man die Höhe der Tür misst. Was ist das Messen mit einem Maßband in diesem Fall? Ein anderes einfaches Modell.

Also haben wir nun zwei Modelle desselben Kühlschranks. Das Verwenden von zwei Modellen, die jeweils für ihre spezifische Aufgabe optimiert sind, spiegelt den DDD-Ansatz zum Modellieren von Fachdomänen wider. Jedes Modell hat seinen

exakt definierten Bounded Context: Die Pappe stellt sicher, dass die Grundfläche des Kühlschranks durch die Küchentür passt, die Messung mit dem Maßband prüft, dass der Kühlschrank nicht zu hoch ist. *Ein Modell sollte überflüssige Informationen weglassen, die für die aktuelle Aufgabe nicht relevant sind.* Zudem ist es nicht nötig, ein komplexes Modell für alle Probleme zu konstruieren, wenn mehrere, viel einfachere Modelle das jeweilige Problem effektiv lösen können.

Ein paar Tage, nachdem ich diese Story auf Twitter gepostet hatte (*https://oreil.ly/rqnEy*), erhielt ich einen Reply, in dem stand, dass ich statt einer Pappe auch ein Smartphone mit LIDAR-Scanner und einer Augmented-Reality-(AR-)App hätte nutzen können. Schauen wir uns diesen Vorschlag aus Sicht von Domain-Driven Design an.

Der Autor des Kommentars sagt, dass dies ein Problem ist, das andere schon gelöst haben, und dass die Lösung fertig einsetzbar zur Verfügung steht. Unnötig, darauf hinzuweisen, dass sowohl die Scanning-Technologie wie auch die AR-Anwendung komplex sind. Im DDD-Jargon macht das das Problem (prüfen, ob der Kühlschrank durch die Küchentür passt) zu einer Generic Subdomain.

Zusammenfassung

Immer dann, wenn wir über einen inhärenten Konflikt in den mentalen Modellen der Domänenexperten stolpern, müssen wir die Ubiquitous Language in mehrere Bounded Contexts aufteilen. Eine Ubiquitous Language sollte innerhalb des Gültigkeitsbereichs ihres Bounded Context konsistent sein. Aber über Bounded Contexts hinweg können die gleichen Begriffe durchaus unterschiedliche Bedeutungen haben.

Während Subdomains »gefunden« werden, werden Bounded Contexts designt. Das Aufteilen der Domäne in Bounded Contexts ist eine strategische Designentscheidung.

Ein Bounded Context und seine Ubiquitous Language können von einem Team implementiert und gewartet werden. Es können keine zwei Teams am gleichen Bounded Context arbeiten. Aber ein Team kann für mehrere Bounded Contexts verantwortlich sein.

Bounded Contexts unterteilen ein System in physische Komponenten – Services, Subsysteme und so weiter. Der Lebenszyklus jedes Bounded Context ist vom Rest entkoppelt. Jeder Bounded Context kann sich unabhängig vom Rest des Systems weiterentwickeln. Aber die Bounded Contexts müssen zusammenarbeiten, um ein System zu schaffen. Manche der Änderungen werden unvermeidlich andere Bounded Contexts beeinflussen. Im nächsten Kapitel werden wir über die verschiedenen Patterns für die Integration der Bounded Contexts sprechen, die genutzt werden können, um die Kontexte vor kaskadierenden Änderungen zu schützen.

Übungen

1. Worin besteht der Unterschied zwischen Subdomains und Bounded Contexts?
   a. Subdomains sind designt, Bounded Contexts werden erkannt.
   b. Bounded Contexts sind designt, Subdomains werden erkannt.
   c. Bounded Contexts und Subdomains sind im Grunde das Gleiche.
   d. Keine der drei Antworten ist richtig.
2. Ein Bounded Context ist eine Grenze für:
   a. ein Modell
   b. einen Lebenszyklus
   c. eine Zuständigkeit
   d. alles drei
3. Welche der folgenden Aussagen ist bezüglich der Größe eines Bounded Context korrekt?
   a. Je kleiner der Bounded Context, desto flexibler ist das System.
   b. Bounded Contexts sollten immer mit den Grenzen von Subdomains abgestimmt sein.
   c. Je größer der Bounded Context ist, desto besser.
   d. Es kommt darauf an.
4. Welche der folgenden Aussagen ist bezüglich der Zuständigkeit von Teams für Bounded Contexts korrekt?
   a. Mehrere Teams können am selben Bounded Context arbeiten.
   b. Ein einzelnes Team kann für mehrere Bounded Contexts zuständig sein.
   c. Ein Bounded Context kann nur von genau einem Team betreut werden.
   d. B und C sind korrekt.
5. Schauen Sie sich das WolfDesk-Beispiel im Vorwort an und versuchen Sie, Funktionalitäten des Systems zu erkennen, für die eventuell unterschiedliche Modelle eines Support-Tickets erforderlich sein könnten.
6. Versuchen Sie, neben den Beispielen aus diesem Kapitel selbst Beispiele für reale Bounded Contexts zu finden.