UMSETZUNG KOMPLEXER PROBLEMSTELLUNGEN ALS SOFTWAREMODELL MITHILFE VON DOMAIN-DRIVEN DESIGN

FABIAN PARZEFALL

Hochschule für angewandte Wissenschaften München

VORWORT

Softwareentwickler werden mit der Herausforderung konfrontiert, Anwendungen für Fachgebiete zu entwickeln, die von Experten über Jahre hinweg studiert wurden. Das Verständnis, welches innerhalb des Entwicklungszyklus einer Anwendung entwickelt werden kann, ist damit stark begrenzt.

Aus diesem Problem haben Softwaredesigner durch die Durchführung erfolgreicher, aber auch fehlgeschlagener Projekte Methodiken entwickelt, um ihren Teams den Prozess von der Analyse bis zur Umsetzung eines Expertenproblems als Anwendung zu erleichtern. Bereits Fowler beschreibt in seinem Buch [2] den Einsatz eines Domänenmodelles zur Kapselung der Kernlogik innerhalb einer Anwendung. Nicht zwangsläufig darauf aufbauend, aber einen ähnlichen Standpunkt vertretend behandelt Evans in seinem Buch [1] ausführlich den Prozess, welcher ein Team zur Entwicklung eines solchen Domänenmodelles durchlaufen muss.

Diese Arbeit beschäftigt sich mit Aspekten und Entwurfsmustern, welche dem von Evans formulierten Domain-Driven Design zugrunde liegen.

INHALTSVERZEICHNIS

Teil I

EINFÜHRUNG

Softwareentwickler investieren oft viel Zeit in das Lösen technischer Probleme einer Anwendung. Die wichtigeren Interessen der Domänenexperten können währenddessen in den Hintergrund geraten. Domain Driven Design ist ein Ansatz diesen Konflikt zu lösen, indem die Modellierung domänenspezfischer Aspekte vor die Modellierung technischer Aspekte gestellt wird.

1 GRUNDLAGEN DOMAIN-DRIVEN DESIGNS

Domain-Driven Design ist eine Vorgehensweise zur Modellierung komplexer Problemdomänen. Der Begriff Domain-Driven Design wurde stark von Eric Evans in seinem Buch [1] geprägt. In diesem Buch beschreibt er die Vorgehensweisen und Entwurfsmuster zur Entwicklung objektorientierter und wartbarer Modelle, gemischt mit praktischen Beispielen.

1.1 DDD IST KEINE ARCHITEKTUR

Domain-Driven Design stellt keine Anleitung zur Strukturierung gesamter Anwendungen bereit. Vielmehr ist es der Versuch, eine gemeinsame Vorgehensweise für Softwareentwickler und Domänenexperten aufzustellen, eine Problemdomäne zu formulieren und als Softwaremodell umzusetzen. Eine Problemdomäne ist ein eingrenzbarer Fachbereich, dessen Verständnis zur Umsetzung als Software für die Entwickler erforderlich ist. Die Domänenexperten verfügen über das nötige Fachwissen des Problems, welches an die Entwickler möglichst unverfälscht vermittelt werden muss.

Ziel ist es, die Problemdomäne in einem Modell so realitätsnah wie möglich, aber nicht mit mehr Komplexität als nötig, abzubilden. Mit Realitätsnähe ist gemeint, dass die verwendeten Klassen- und Methodennamen der natürlichen Sprache der Domänenexperten entsprechen. Zudem soll sämtliche Businesslogik ausschließlich innerhalb dieses Modelles verankert sein und sich nicht in anderen Teilen der Anwendung befinden. Ein Domänenexperte ohne Erfahrung im Bereich der Softwareentwicklung sollte letztendlich in der Lage sein, die umgesetzten Anforderungen innerhalb des Programmcodes (mit Hilfe eines Entwicklers) nachvollziehen zu können.

Das führt gleichzeitig zu der Implikation, dass das Modell frei von sämtlichen technischen Aspekten sein muss. Da das Modell meistens innerhalb einer Datenbank abgebildet werden soll, werden Object Relation Mapper eingesetzt. ORMs übernehmen die Aufgabe des Lesens und transaktionalen Schreibens von Enitäten in relationalen Datenbankensystemen. Eine Umsetzung Domain-Driven Designs ist auch ohne ORMs möglich, mit diesen ist es aber um einiges einfacher, technische Aspekte vom Modell zu trennen. Insgesamt ist Domain-Driven Design an keine bestimmte Architektur gebunden.

1.2 VOM ANEMIC ZUM RICH DOMAIN MODEL

Ein häufiger Fehler, den unerfahrene Entwickler bei der Entwicklung eines Domänenmodelles begehen, ist die Umsetzung eines Anemic Domain Models. Einem Anemic Domain Model sieht man auf den ersten Blick nicht unbedingt an, dass es die Domäne nur oberflächlich abbildet. Die Entitäten und Beziehungen können gut ausgearbeitet sein, dementsprechend aussagekräftig wirkt ein Objektdiagramm.

Dieses Modell weist jedoch keinerlei Verhalten auf. Die Klassen bestehen zu einem großen Teil aus Getter- und Settermethoden, Businesslogik fehlt gänzlich. Die eigentliche Logik wird innerhalb einer auf dem Domänenmodell aufbauenden Service-Schicht umgesetzt. [3]

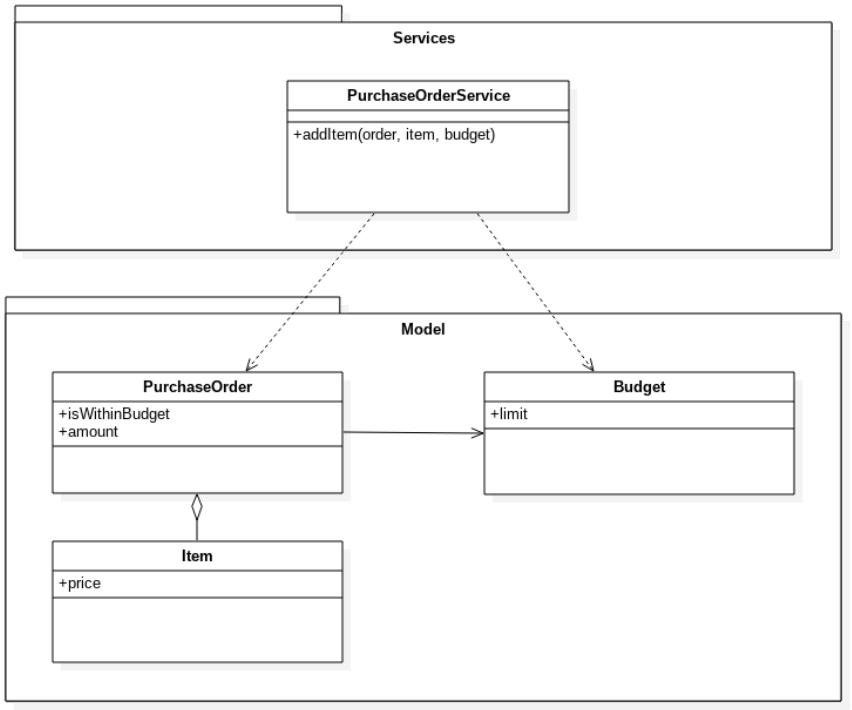

Abbildung 1: Beispiel eines Anemic Domain Models. Sämtliche Logik befindet sich innerhalb des PurchaseOrderServices.

Das Beispiel aus Abbildung 1 stellt ein vereinfachtes Klassendiagramm eines Anemic Domain Models dar. Eine Purchase Order besteht aus mehreren Items. Außerdem ist eine PurchaseOrder einem Budget zugeordnet. Wird ein Item hinzugefügt, so berechnet die Methode addItem den Amount der Order neu und überprüft, ob diese noch im Budget liegt.

Das Problem hier ist, dass das Design eher einem prozeduralen Paradigma entspricht, als einem objektorientierten. Wird ein Item hinzugefügt, so gehen sämtliche Zustandsänderungen von der addItem-Methode aus, das Modell ist lediglich ein Container für Daten. Wird außerdem das Budget reduziert, ist die Order unter Umständen nicht

mehr gültig. Ist eine solche Struktur jedoch gewünscht, empfiehlt Fowler den Einsatz von *Transaction Scripts* [2]. Ansonsten sollte die Umstrukturierung zu einem Rich Domain Model vollzogen werden (vgl. Abbildung 2). [3]

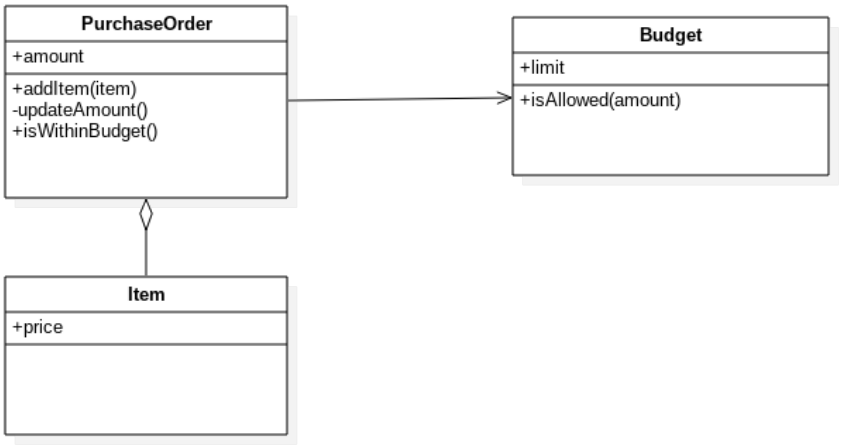

Abbildung 2: Das Modell aus Abbildung 1 mit der gleichen Funktionalität als Rich Domain Model.

Im Rich Domain Model wird die Logik innerhalb des Modelles umgesetzt. In diesem können Operationen aussagekräftiger abgebildet werden, da hier einzelne Objekte miteinander interagieren und nicht von außen manipuliert werden. Das Rich Domain Model schließt jedoch den Einsatz einer übergeordneten Serviceschicht nicht aus. Nur soll in dieser kein Verhalten abgebildet werden, sondern diese koordiniert und delegiert ausschließlich Aufgaben an die entsprechenden Domänenobjekte. [1]

1.3 ENTWICKLUNG EINER UBIQUITÄREN SPRACHE

Die Formulierung eines Domänenmodelles ist ein kollaborativer Prozess zwischen Experten und Entwicklern. Dieser Prozess setzt ein gemeinsames Verständnis des Problems voraus. Experten pflegen in der Regel eine aus ihrem Alltag gebildete Sicht auf das Problem. Entwickler hingegen tendieren dazu, eigene Abstraktionen auf Basis der für sie unbekannten Domäne zu bilden.

Wird kein Wert auf eine gemeinsame Kommunikationsgrundlage gelegt, führt es dazu, dass der ohnehin schon schwierige Wissensaustausch von Missverständnissen geprägt wird. Experten verstehen die technischen Begrifflichkeiten der Entwickler nicht und auf Seite der Entwickler hat jeder ein eigenes mentales Modell der Domäne. Unterschiedliche Entwickler beginnen unterschiedliche Wörter für die gleichen Dinge zu verwenden. Entwickler übersetzen ihre eigenen Begrifflichkeiten für andere Entwickler und Experten.

Wird das Projekt auf diese Weise entwickelt, dann erfüllt das resultierende Modell womöglich letztendlich alle Anforderungen, es spiegelt aber nicht die Ideen der Experten wider. Das Modell ist in sich selbst nicht konsistent und einzelne Teile passen nicht richtig zueinander. Änderungen und Erweiterungen gestalten sich deshalb als zeitintensiv und fehleranfällig.

Aus diesem Grund wird bei Domain-Driven Design großer Wert auf die Verwendung einer ubiquitären Sprache gelegt. [1] Diese muss als Grundlage sämtlicher Diskussionen dienen, sowohl mit Experten als auch zwischen Entwicklern. Experten sollten Entwickler darauf hinweisen, sobald diese Begriffe verwenden, welche unnatürlich für ihren allgemeinen Sprachgebrauch sind. Auch Entwickler müssen sich bemühen, untereinander immer die ubiquitäre Sprache zu verwenden und dürfen nicht auf ein bequemeres, technisches Jargon zurückfallen.

Es ist klar, dass die ubiquitäre Sprache sich im Laufe der Entwicklung wandeln muss. Diese wird in Zusammenarbeit mit den Experten gebildet. Solange sich aber jeder daran hält und das Modell anhand dieser Sprache geformt wird, so wird das Modell selbst Teil der Sprache und trägt eine Bedeutung, die jeder versteht.

1.4 AGILE ENTWICKLUNG

Die ständige Kommunikation zwischen Entwicklern und Experten ist ein essentieller Bestandteil des Domain-Driven Designs. Die Gestaltung des Domänenmodells ist ein fortlaufender Prozess. Daher sind, zumindest während der Ausarbeitung des Modelles, agile Entwicklungsmodelle besser geeignet als lineare Entwicklungsmodelle (wie das Wasserfallmodell).

Agile Entwicklung bedeutet aber nicht, dass zwangsläufig Scrum eingesetzt werden muss, sondern dass das Modell auf Basis regelmäßigen Feedbacks Stück für Stück verfeinert wird. Währenddessen wird auch die ubiquitäre Sprache weiterentwickelt und Missverständnisse werden beseitigt, was wiederum zu einem höherwertigerem Modell führt.

Teil II

DAS DOMÄNENMODELL IN DER PRAXIS

In diesem Teil werden elementare Entwurfsmuster für die praktische Umsetzung des Domänenmodelles erläutert. Außerdem wird auf Probleme eingegangen, welche durch den Einsatz relationaler Datenbanksysteme entstehen. Zum Schluss wird diskutiert, wie besonders umfangreiche Modelle behandelt werden können.

2 ELEMENTE DER DOMÄNE

In diesem Kapitel werden die Bausteine und deren Zusammenhänge innerhalb eines Domänenmodells erklärt: die Unterschiede zwischen Entitäten und Wertobjekte, deren Lebenszyklus in Aggregaten und der Einsatz von Services für Operationen, die nicht eindeutig einem Objekt zugeordnet werden kann.

2.1 ENTITÄTEN

Entitäten sind Objekte, welche primär über ihre Identität identifiziert und referenziert werden. [1] Wichtig ist, dass zwei Objekte aufgrund ihrer Identität und nicht ihrer Eigenschaften unterschieden werden.

Wird ein Fahrer aufgrund zu schnellen Fahrens geblitzt, so wird in den Datenbanken nach dem Kennzeichen gesucht, um es dem Autobesitzer zuordnen zu können. Das Fahrzeug wird also über seine Identität gesucht. Wird nach den Kriterien, dass das Auto ein silberner VW Golf sei und der Fahrer schwarzhaarig sei, gesucht, dann ist die Wahrscheinlichkeit einer Verwechslung nicht zu vernachlässigen. In diesem Fall würde ein falsch beschuldigter Fahrer über das Kennzeichen, also der Identität seines Autos nachweisen, dass das Auto auf dem geblitzten Foto nicht ihm gehöre. In diesem Fall ist es folglich wichtig, dass sowohl der Fahrer als auch das Auto über ihre Identität und nicht ihre äußerlichen Eigenschaften ausfindig gemacht werden.

Aber nicht immer ist eine Zuordnung einer Identität zu einem Objekt notwendig oder sinnvoll. In einem Softwaresystem zur Reservierung von Kinoplätzen hat jeder Platz eine eindeutige Sitznummer. Ein Kunde erhebt den Anspruch, dass er, wenn er ein Ticket für einen bestimmten Platz erworben hat, auf diesem auch sitzen darf. Für den Kunden ist die Position des Sitzplatzes von Bedeutung (ansonsten sind schließlich alle Stühle gleich), für das System ist aber ausschließlich die Sitznummer der Reservierung relevant. Hier wird der Stuhl als Objekt mit Identität behandelt.

In einem anderen Beispiel geht es um ein System für die Buchung von Tickets für Konferenzen. Bei diesem können keine Plätze reserviert werden, die Wahl des Sitzplatzes wird dem Besucher vor Ort überlassen. Der Konferenzsaal hat aber nur eine begrenzte Anzahl an Sitzplätzen, die auch alle durchnummeriert sind. Die Anforderung an das System ist jedoch nur, dass der Saal nicht überbucht wird. Damit ist die Nummer der Sitzplätze für das System irrelevant, da es nur die Anzahl kennen muss. Jeden Sitz nun einzeln zu verwalten wäre

nicht nur unnötig, sondern es würde nicht dem Modell des Systems entsprechen. Ein Sitz wird hier ohne Identität behandelt.

2.1.1 *Identitäten*

Identitäten entstehen nicht aus dem Nichts, irgendjemand muss eine Zuweisung vornehmen. Im Falle der Autos wird über die Zulassung mit dem Kennzeichen eine Identität vergeben. Diese ist ausreichend zum Auffinden des derzeitigen Besitzers eines Autos. Kennzeichen können sich jedoch ändern oder das Auto wechseln. Der Gesetzgeber aber möchte, dass Autos über deren gesamte Lebenszeiten herstellerübergreifend eindeutig identifizierbar sind. Aus diesem Grund vergeben alle Autohersteller ihren Autos eine eindeutige Nummer (Fahrzeug Identifizierungsnummer [6]), welche an einer unauffälligen Stelle innerhalb des Autos vermerkt wird.

In Softwaresystemen reicht es oftmals aus, eine fortlaufende Nummer zu generieren, wenn der Nutzer ohnehin an dieser kein Interesse hegt. Eine Software zur Verwaltung von beliebigen Dokumenten muss jedes Dokument eindeutig identifizieren können. Der Nutzer will bei der Suche aber nicht einen zufälligen Bezeichner eingeben, sondern nach den Attributen des Dokumentes suchen. Hier hat der Nutzer also kein Interesse an der Identität des Dokumentes.

Bei der Verfolgung von Postpaketen dagegen ist die Identität des Paketes von Bedeutung für den Nutzer. Dem Paket wird eine Nummer zugewiesen, über welche der Empfänger das Paket verfolgen kann. Um das Raten von Nummern einzuschränken sind fortlaufende Nummer nun nicht unbedingt erwünscht. Eindeutigkeit ist aber nach wie vor eine strikte Anforderung.

Identitäten werden stets entweder vom Entwickler definiert oder gehen aus der Domäne hervor. Je nach Domäne können unterschiedliche Anforderungen an die Form der Identitfier gestellt werden. Eindeutigkeit ist immer eine davon.

Moderne Datenbanksysteme wie MySQL [5] oder SQL Server [4] stellen Features bereit, um IDs selbstständig beim Einfügen von Einträgen zu generieren. Dies ist oftmals die einfachste Lösung, da die Eindeutigkeit dieser Werte auch in nebenläufigen Szenarien garantiert ist.

Unter Umständen muss aber die Erzeugung der IDs von der Anwendung selbst übernommen werden. Die oben genannten Fahrzeug Identifizierungsnummern bestehen aus drei Teilen: einem Hersteller Code, einem Identifikator für die Baureihe und den Motortyp und einer fortlaufenden Nummer. [6] Dies ist offensichtlich eine Anforderung an die Domäne und sollte dementsprechend dort implementiert werden und nicht innerhalb des Datenbanksystems.

2.1.2 *Erzeugung und Zugriff*

Um mit Entitäten arbeiten zu können ist eine Referenz auf ebenjene notwendig. Um an eine solche Referenz zu kommen, gibt es zwei Möglichkeiten: die Entität wird neu erzeugt oder eine bestehende wird zum Beispiel innerhalb einer Datenbank materialisiert. Für diese Anwendungsfälle gibt es zwei Entwurfsmuster: Factories und Repositories.

2.1.2.1 *Factories*

In objektorientierten Programmiersprachen werden Objekte in der Regel über ihren Konstruktor erzeugt. In Fällen, bei denen die Konstruktion eines Objektes sehr kompliziert ist oder zusätzliche Logik erfordert, kann die Erzeugung über Factories gekapselt werden. [1]

In Unterabschnitt 2.1.1 wurde die Generierung der Fahrzeug Identifizierungsnummer (FIN) als Teil der Problemdomäne identifiziert. Ein Fahrzeug kann sich diese Nummer nicht selbst zuweisen und erzwingt deshalb eine Übergabe als Konstruktorparameter.

Der Client müsste nun selbst die FIN generieren und bei der Erzeugung des Fahrzeuges angeben. Damit würden Aspekte des Domänenmodell nach außen gegeben werden, genau das soll aber vermieden werden. An dieser Stelle kann eine dedizierte Factory-Klasse eingesetzt werden. In Abbildung 3 wird bei der Erzeugung nur der Typ angegeben. Die Factory stellt sicher, dass das erzeugte Fahrzeug über eine valide und eindeutige FIN verfügt.

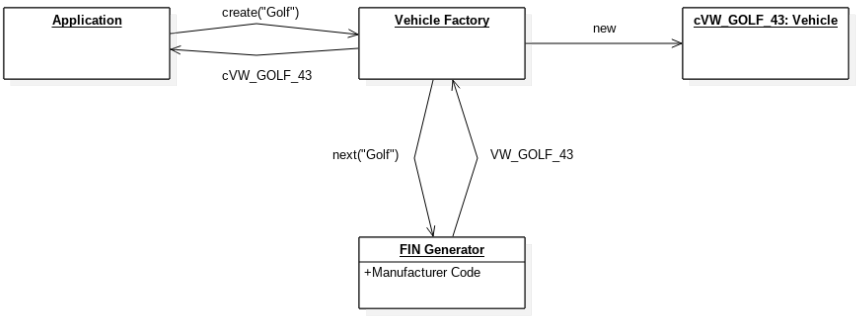

Abbildung 3: Einsatz einer Factory um Details der Erzeugung von Domänenobjekten vor dem Client zu verstecken. (Die gewählten FINs sind exemplarisch und entsprechen nicht dem Standard)

Eine Factory muss nicht zwangsläufig eine eigene Klasse sein, wenn die erzeugten Objekte sinnvoll von einem anderen Objekt erzeugt werden können. Ein einfaches Beispiel ist ein System zu Abwicklung von Bestellungen. Eine Bestellung muss immer von einem Kunden getätigt werden, welcher unter Umständen besondere Vergünstigungen bekommen kann. Hier könnte das Kundenobjekt selbst eine neue,

leere Bestellung erzeugen, welche die Information über potentielle Vergünstigungen bereits enthält.

2.1.2.2 *Repositories*

Um mit einem Objekt arbeiten zu können, ist eine Referenz auf dieses erforderlich. Es gibt drei Möglichkeiten, um eine Referenz auf ein Objekt zu erhalten. Die erste ist die Erzeugung des Objektes. Die zweite ist der Zugriff über eine Assoziation von einem anderen Objekt (siehe Abschnitt 2.3). Und die dritte ist die Materialisierung eines bestehenden Objektes aus einer Datenbank.

Ein bestehendes Objekt aus einer Datenbank zu laden ist oberflächlich betrachtet ein triviales Problem: die Datenbank selbst hat keinerlei Restriktionen, welche Objekte geladen werden können. Mithilfe einer `SELECT`-Abfrage kann jedes beliebige Objekt aus der Datenbank materialisiert werden.

Der Datenbankzugriff ist offensichtlich auch keine Aufgabe der Domäne, da es sich dabei um Interaktionen mit der Infrastruktur handelt. Sind die Entwickler selbst dafür verantwortlich, die Objekte aus der Datenbank abzurufen und abzuspeichern, also SQL Queries zu formulieren, auf die Datenbank zuzugreifen, das Ergebnis als Objekt zu materialisieren und die einzelnen Felder letztendlich wieder auszulesen und in der Datenbank abzuspeichern, dann entsteht die Tendenz, dass das Modell nur noch als Ansammlung Daten haltender Objekte gesehen wird. In dieser Situation besteht die Gefahr, dass für scheinbar triviale Operationen der Umweg über die Materialisierung des Modells ausgelassen wird und die Daten direkt in der Datenbank manipuliert werden. [1] Passiert dies, dann hat das Domänenmodell keinen Grund mehr zu existieren.

Aus diesem Grund sollte der Zugriff auf die Datenbank in Repositories gekapselt werden. Repositories verstecken vor dem Aufrufer sämtliche Aspekte des Datenbankzugriffes und präsentieren sich nach außen als In-memory Menge aller Objekte eines bestimmten Types. Die zugrunde liegende Speicherungstechnologie muss auch nicht zwangsläufig eine relationale Datenbank sein. Es könnten auch XML-Dateien innerhalb des Dateisystems mithilfe eines Repositories gekapselt werden. Nach außen mach das keinen Unterschied, solange Objekte hinzugefügt und entfernt werden können, da das Repository im Hintergrund alle notwendigen Zugriffsoperationen abbildet.

Neben den Operationen des Hinzufügens und Entfernens von Objekten stellen Repositories noch weitere Methoden zum Suchen von Objekten nach bestimmten Kriterien bereit. Die einfachste Methode zum Suchen ist, jeden benötigten Query per Hand im Code zu hinterlegen. Für größere Projekte mit vielen verschiedenen Queries kann es wirtschaftlich sein, eine allgemeine Methode zu entwickeln, welche als Parameter eine Spezifikation über die gewünschten Objekte entgegennimmt.

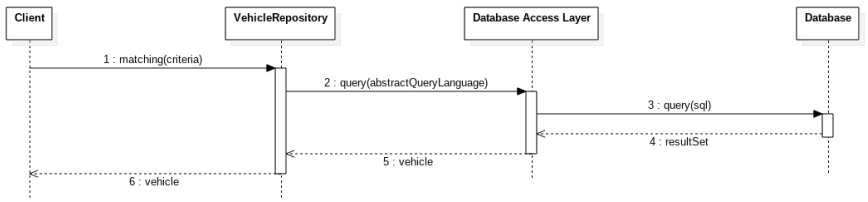

Abbildung 4: Abruf einer Entität mit einer Spezifikation aus einer Datenbank.

In Abbildung 4 wird der Abruf einer Entität aus einer Datenbank exemplarisch dargestellt. Der Client kennt ausschließlich das Repository und spezifiziert das Fahrzeug, welches abgerufen werden soll. Repositories greifen üblicherweise nicht direkt auf die Datenbanken zu, sondern generieren einen Query in einer abstrakten Querysprache und übergeben diesen an die Datenbankzugriffsschicht. Diese Schicht generiert aus dem abstrakten Query einen SQL Query und erzeugt aus dem Ergebnis die gewünschten Entitäten.

2.2 WERTOBJEKTE

In Abschnitt 2.1 wurde bereits erwähnt, dass es nicht sinnvoll ist, jedes Objekt aus der Realität als Entität abzubilden, wenn dies nicht erforderlich ist. Dennoch müssen diese Objekte, die nicht über eine Identität verfügen, sondern sich durch ihre Charakteristiken voneinander unterscheiden, abgebildet werden.

Ein Auto hat vier Reifen, zwei links und zwei rechts. Die Profile der Reifen auf der linke Seite ist andersrum angebracht, als die Profile auf der rechten Seite. Dies ist notwendig, da die Profile so gestaltet sind, dass das Wasser optimal aus diesen abfließen kann. Bringt ein Automechaniker also Reifen an das Auto an, dann achtet er darauf, dass er die Reifen abhängig von der Profilrichtung an der korrekten Seite anbringt. Ob an diesem Auto im Herbst des vorigen Jahres genau die gleichen Reifen angebracht waren ist für ihn nicht relevant. Die einzelnen Reifen werden ausschließlich anhand ihrer Charakteristiken unterschieden. Der Kunde identifiziert beim Abholen seines Fahrzeuges dieses aber über das Kennzeichen, also der Identität, seines Autos.

Objekte, die über keine Identität verfügen, werden Wertobjekte genannt. [1] Im vorigen Beispiel ist das Auto eine Entität, da es über seine Identität, dem Kennzeichen, von anderen Autos unterschieden wird. Die Reifen dagegen sind einzelne Wertobjekte, da sie sich nur durch ihre Charakteristiken unterscheiden. Welcher Reifen an welchem Auto angebracht wird ist nicht von Bedeutung.

Wertobjekte sollten, wenn sie von mehreren Objekte referenziert werden, unveränderlich sein. Ein Geldbetrag, bestehend aus einer De-

zimalzahl und einem Währungssymbol, ändert seinen eigenen Betrag nicht. Das gleiche gilt für die Autoreifen. Ein Autoreifen könnte in obigem Modell die Eigenschaften Profilrichtung, Sommer- oder Winterreifen und einem Abnutzungsgrad (neu, gebraucht, abgenutzt) bestehen. Da ein Autoreifen nicht als Individuum betrachtet wird, ist es unsinnig Zuständsübergänge für diesen zu definieren.

2.3 ASSOZIATIONEN

Haben zwei Objekte innerhalb des Modells eine Assoziation, dann muss diese in der Software als auflösbare Beziehung abgebildet werden. Eine Assoziation kann durch eine Referenz oder eine Collection repräsentiert werden. Es könnte aber auch eine Methode sein, welche über einen Datenbankzugriff die Assoziationen auflöst.

Um das Modell so einfach wie möglich zu halten, sollten so viele Assoziationen wie möglich eliminiert werden. Bidirektionale Beziehungen, also `A` verweist auf `B` und `B` verweist auf `A`, sollten grundsätzlich vermieden werden. Diese sind technisch aufwändig in der Implementation, da das Anlegen und Entfernen von solchen Assoziationen ein zweistufiger Prozess ist. Wenn aufgrund eines Fehlers eine Beziehung nur auf einer Seite verändert wird befindet sich das Modell in einem ungültigen Zustand.

Bei der Ausarbeitung des Modelles haben viele Objekte intuitiv bidirektionale Beziehungen untereinander. Eine Software zur Verwaltung von E-Mail Verteilern hat eine Beziehung zwischen Verteiler und Empfänger. Ein Verteiler ist mit `n` Empfängern assoziiert und ein Empfänger empfängt E-Mails von `m` Verteilern. Für den Versand von E-Mails ist jedoch ausschließlich die Assoziation von Verteiler zu Empfänger interessant. Mit diesem Wissen über die Problemdomäne kann das Modell vereinfacht werden. (vergleiche Abbildung 5)

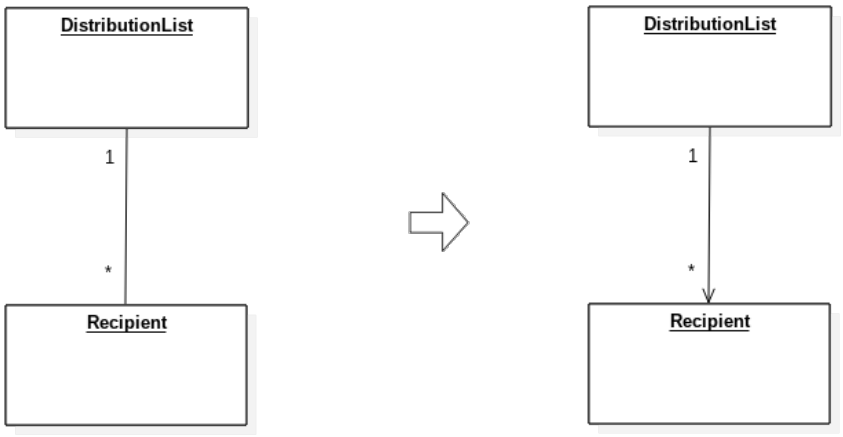

Abbildung 5: Vereinfachung der Assoziationen des Modelles durch die Anwendung von neuen Erkenntnissen über die Domäne.

2.4 AGGREGATE

Manche Objekte müssen referenziert werden, damit sie im Domänenkontext sinnvoll eingeordnet werden können. In der Kontakte App eines Smartphones ist es möglich, bei einem Kontakt eine Adresse zu hinterlegen. Ist dies bei einem Kontakt der Fall, so hält das Objekt hinter dem Kontakt eine Referenz auf ein Adressobjekt. Im Kontext der gegebenen Domäne kann eine Adresse nicht selbstständig, sondern nur an einem Kontakt existieren. Wird der Kontakt gelöscht, dann muss auch die zugehörige Adresse gelöscht werden. Damit ist die Adresse dem Kontakt untergeordnet.

In diesem Beispiel wurde durch das Binden der Adresse an einen Kontakt ein Aggregat mit dem Kontakt als Root gebildet. Allgemein ist ein Aggregat ein Cluster von Objekten, welche als logische Einheit behandelt werden. Der Aggregatsroot ist das einzige Objekt, welches von Objekten außerhalb des Clusters referenziert werden kann. Objekte innerhalb des Aggregates können dagegen Referenzen aufeinander und auf andere Aggregate Roots halten. Entitäten innerhalb des Aggregates haben folglich keine globale, sondern nur eine lokale, auf das Aggregat beschränkte Identität. [1]

Weil der Aggregate Root das einzige Objekt ist, das außerhalb des Aggregates referenziert werden kann, können Objekte innerhalb des Aggregates nicht verändert werden, ohne dass der Root Kenntnis davon nimmt. Damit kann über den Root sichergestellt werden, dass das gesamte Aggregat zu jedem Zeitpunkt konsistent ist.

Repositories können ausschließlich Aggregate Roots in ihrer Signatur bereitstellen, da diese die einzigen global referenzierbaren Objekte sind. Das Ergebnis ist eine drastische Vereinfachung des Domänemodelles, da die Anzahl der direkt modifizierbaren Objekte geringer ist und diese nach jeder Operation garantiert konsistent sind. Besteht jedoch uneingeschränkter Zugang zu jedem Objekt der Domäne, dann besteht die Gefahr, dass irgendein untergeordnetes Objekt so abgeändert wird, dass Businessregeln verletzt werden. Auch beim Entfernen von Objekten sind Aggregate hilfreich, da einfach bei der Entfernung des Roots schlicht alle Objekte innerhalb des Aggregates mit dem Root entfernt werden.

In Abbildung 6 ist ein einfaches Beispiel eines Modelles für die Verwaltung von Konferenzräumen abgebildet. Ein Konferenzraum hat eine unbegrenzte Anzahl an Reservierungen mit der Einschränkung, dass sich einzelne Reservierungen nicht überschneiden dürfen. Der Konferenzraum und die Reservierung bilden zusammen ein Aggregat mit dem Raum als Root. Außerdem hat jede Reservierung einen Besitzer. Dies ist erlaubt, da Person auch ein Aggregate Root ist.

Erstellt oder ändert eine Person eine Reservierung, dann kann dies ausschließlich über das ConferenceRoom Objekt geschehen. Es gibt auch kein Repository, über welches direkt Reservierungen angelegt

werden können. Deshalb kann der Root sicherstellen, dass sich keine Reservierungen überschneiden und befindet sich damit immer in einem gültigen Zustand.

Wird ein Konferenzraum gelöscht, weil der Raum zum Beispiel für einen anderen Zweck benötigt wird, dann ist aufgrund der Definition des Aggregates klar, dass sämtliche Reservierungen nicht mehr benötigt werden und gelöscht werden können, alle Personen jedoch im System verbleiben. Diesen könnte bei der Löschung eine Benachrichtigung zugestellt werden, dass ihre Reservierung nicht mehr gültig sei.

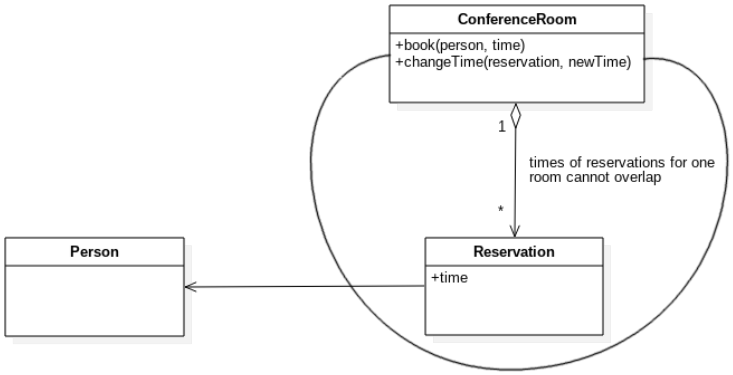

Abbildung 6: Einsatz eines Aggregates um Zugriffe innerhalb des Modells zu kontrollieren.

2.5 SERVICES

Nicht jede Operation kann sinnvoll als Methode einer Entität oder eines Wertobjektes umgesetzt werden. Um dieses Problem zu lösen können Services innerhalb des Modelles eingeführt werden. Es besteht jedoch die Gefahr durch den übermäßigen Einsatz von Services die Domänenobjekte ihres Verhaltens zu berauben. Auch müssen Domänenservices frei von Aspekten der Anwendung gehalten werden. [1]

Die Registrierung neuer Nutzer in einem Onlineshop erfordert einige Schritte. Es muss die Gültigkeit der Daten überprüft werden, anschließend soll eine E-Mail zur Verifikation seiner Adresse an den Nutzer geschickt werden. Diese Operation findet innerhalb einer Transaktion statt und kann nicht auf bestehenden Domänenobjekten abgebildet werden, da diese erst noch erzeugt werden müssen. Deshalb ist es sinnvoll die Registrierung innerhalb eines Services umzusetzen.

Hier entsteht jedoch die Frage, ob der Versand einer E-Mail eine Aufgabe der Domäne sein kann. Die tatsächliche Abwicklung des Ver-

sandes über SMTP muss als Service in der Infrastrukturschicht bereitgestellt werden. Aber auch das Konzept der E-Mail ist nicht zwangsläufig in der Domäne verankert. Eine Anwendung könnte auch eine Verifikation per SMS anbieten. Diese Aspekte der Benutzerinteraktion sind auf Anwendungsebene zu lösen. Die Überprüfung der Daten und das Anlegen eines nicht verifizierten Nutzers sind dagegen Aspekte der Domäne. [1]

In diesem Beispiel können die benötigten Services in drei Kategorien eingeteilt werden:

ANWENDUNG Services, welche Operationen auf Anwendungsebene durchführen. Verantwortlich für die Weitergabe der Nachricht an den Domänenservice und das Versenden der Bestätigungsemail über einen Infrastruktur Service.

DOMÄNE Abbildung von Operationen in der Domäne. Hier die Validierung der eingegeben Daten und das Anlegen des Benutzers.

INFRASTRUKTUR Interaktion mit Systemen, welche außerhalb der Anwendung liegen. In diesem Fall der Versand von E-Mails über SMTP.

2.6 BOUNDED CONTEXT

In größeren Projekten können innerhalb einer Anwendung mehrere Modelle nebeneinander existieren. Eine solche Situation tritt auf, wenn mehrere Teams Teile der Anwendung entwickeln, welche über größtenteils voneinander unabhängige Businessregeln verfügen. Oft gibt es dennoch Entitäten, welche in mehreren Modellen gefunden werden können. Hierbei besteht die Gefahr, dass die unterschiedlichen Teams nicht über eine gemeinsame ubiquitäre Sprachen verfügen, da die Entitäten im Kontext des jeweiligen Teams eine andere Bedeutung haben. In diesem Fall schafft die Definition eindeutiger Grenzen und Interaktionen zwischen den einzelnen Teilmodellen diskrete Kontexte, welche auf die ubiquitäre Sprache des jeweiligen Teams zugeschnitten sind. [1]

Abbildung 7 zeigt ein vereinfachtes Modell, in welchem die Abgrenzung bereits durchgeführt wurde. Auf Vertriebsseite ist interessant, welcher Mitarbeiter verantwortlich für die Betreuung des Kunden ist und wie viele Verträge mit welchen Produkten abgeschlossen wurden. In den Produkten ist beispielsweise verankert, dass bestimmte Produkte im Rahmen eines Vertrages nicht miteinander kombinierbar sind.

Auf Support Seite stehen die Tickets im Zentrum mit Verweisen auf den Kunden, die betroffenen Produkte, dem zugewiesen Mitarbeiter und der Dokumentation, welche Aktionen im Rahmen dieses Tickets bereits durchgeführt wurden. Die genauen Details des Vertrages oder

mögliche Produktkombinationen sind in diesem Kontext nicht erforderlich, die Auffindung Tickets ähnlicher Probleme des gleichen Kunden oder Produktes ist es dagegen schon.

An das Modell werden abhängig vom Kontext verschiedene Anforderungen gestellt, deshalb liegt die Trennung naheliegend. Dennoch müssen die Modelle miteinander interagieren: im Support-Kontext sind die genauen Vertragsdetails nicht von Relevanz, dennoch wird die Information, ob ein aktiver Vertrag vorliegt, benötigt. Außerdem müssen die Produkte des Vertriebs mit den Produkten des Supports übereinstimmen und folglich im Modell assoziiert werden.

Durch die Definition von Bounded Contexts können die einzelnen Teams ihr Modell nach ihren Vorstellungen gestalten ohne andere Teams zu behindern, solange die Schnittstellen zwischen den Modellen gepflegt werden. Um die Interoperabilität der einzelnen Modelle während der Entwicklung zu gewährleisten, können automatisierte Tests innerhalb einer Continous Integration durchgeführt werden.

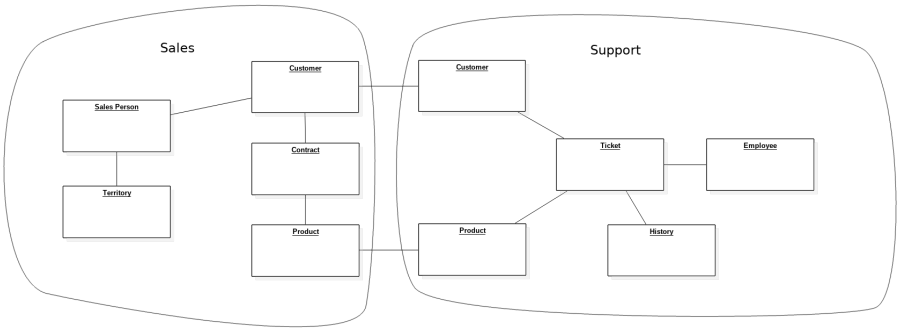

Abbildung 7: Ein Modell im Kontext des Vertriebes auf der linken Seite und auf der rechten Seite im Kontext des Supports.

3 FAZIT

In dieser Arbeit wurde der Einsatz Domain-Driven Designs nur oberflächlich angeschnitten. Es wurden die wichtigsten Elemente des Prozesses erklärt und einige Entwurfsmuster vorgestellt, um generelle Probleme innerhalb eines Modelles einheitlich angehen zu können.

In der Praxis gibt es oftmals Hürden, welche die Entwicklung des perfekte Modelles verhindern. Gründe hierfür können unter anderem technische Limitationen sein. Aber auch dann, wenn Anforderungen zu spät erkannt und Missverständnisse zu spät beseitigt werden und daraus resultierende Änderungen aus Gründen des Aufwandes nicht mehr sauber in das Modell integriert werden können, leidet die Qualität des Modelles.

Mit Domain-Driven Design wird keine Schritt-für-Schritt Anleitung geliefert, sondern nur Methodiken, mit denen diese Probleme frühzeitig erkannt und gelöst werden sollen. Dabei muss jedes Team seinen eigenen Stil finden und kann sich bei Bedarf auf dieser Methodiken bedienen. Wenn der vom Team adaptierte Prozess jedoch funktioniert und das Projekt es nicht erfordert, dann besteht keine Notwendigkeit zwanghaft diese Methoden einzusetzen.

Teil III

APPENDIX

LITERATUR

[1] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003.

[2] Martin Fowler. *Patterns of Enterprise Application Architecture*. Pearson Education, 2002.

[3] Martin Fowler. *Anemic Domain Model*. Website. Nov. 2003. URL: https://www.martinfowler.com/bliki/AnemicDomainModel.html.

[4] *IDENTITY (Property) (Transact-SQL) | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql-identity-property.

[5] *MySQL 5.7 Reference Manual :: 4.6.9 Using AUTO_INCREMENT*. URL: https://dev.mysql.com/doc/refman/5.7/en/example-auto-increment.html.

[6] *VERORDNUNG (EU) Nr. 19/2011 DER KOMMISSION vom 11. Januar 2011 über die Typgenehmigung des gesetzlich vorgeschriebenen Fabrikschilds und der Fahrzeug-Identifizierungsnummer für Kraftfahrzeuge und Kraftfahrzeuganhänger zur Durchführung der Verordnung (EG) Nr. 661/2009 des Europäischen Parlaments und des Rates über die Typgenehmigung von Kraftfahrzeugen, Kraftfahrzeuganhängern und von Systemen, Bauteilen und selbstständigen technischen Einheiten für diese Fahrzeuge hinsichtlich ihrer allgemeinen Sicherheit*. Amtsblatt der Europäischen Union. Jan. 2011. URL: http://eur-lex.europa.eu/legal-content/DE/TXT/HTML/?uri=CELEX:32011R0019&from=EN.

DECLARATION

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, 2017

Fabian Parzefall