Moderne Ansätze zur Umgestaltung eines monolithischen Systems

Bachelorarbeit

von

Wilhelm Alexander Maximilian Thelen

aus

Köln

vorgelegt am

Lehrstuhl für Softwaretechnik und Programmiersprachen
Prof. Dr. Michael Leuschel
Heinrich-Heine-Universität Düsseldorf

September 2019

Betreuer:
Dr. Jens Bendisposto
Christian Meter

Vorwort

Diese Bachelorarbeit ist für mich keine normale Prüfungsleistung im Rahmen eines Studiums: Sie behandelt ein Thema, in das ich gern Herzensblut reinstecke. Ein komplexes Problem zu analysieren, bis ins kleinste Detail zu verstehen, in seine atomaren Bestandteile zu zerlegen und diese dann wieder in kleinere, simple Probleme zusammenzubauen, um letztendlich das gesamte Problem zu erfassen und sukzessive zu lösen, ist mein Hobby. Diesen Weg möglichst schön zu gestalten, ist meine Leidenschaft, die durch Perfektionismus angetrieben wird. Denn erst durch diese Schönheit wird komplex zu simpel und kompliziert zu einfach. Die Leidenschaft dazu findet in der Softwareentwicklung ihr Maximum. Drei Monate sind für das ausgewählte Thema in Gänze allerdings zu wenig und deswegen hoffe ich, dass man auf dieser Arbeit aufbauen kann und den Weg weitergeht.

Ich bin ehrlich. Diese Bachelorarbeit wurde effektiv innerhalb von zwei Wochen zusammengeschrieben. Die meiste Zeit habe ich in Recherche mit sehr vielen Notizen investiert. Noch nie habe ich so viel in meinem Leben gelesen und gelernt wie in dem letzten halben Jahr. Unzählige Bücher habe ich verschlungen, viele Talks bis tief in die Nacht geguckt und Artikel durchgearbeitet. 16 Stunden pro Tag wurde von mir Wissen angesammelt. Es hat ungemein Spaß gemacht. Keiner konnte mich verstehen.

Dem Leser wird auffallen, dass diese Ausarbeitung nicht wie jede andere Abschlussarbeit eines Bachelors einer Wissenschaft aufgebaut ist. Dies liegt vor allem in der Natur der Sache: Softwaredesign und -architektur ist mehr Kunst als Wissenschaft. Ich habe aber dennoch, so gut es geht, versucht, dem wissenschaftlichen Charakter nicht zu kurz zu kommen und in den Anfangskapiteln der Theorie den Vorrang gegeben. Außerdem habe ich mich wissentlich dagegen entschieden, Standardkapitel wie „Grundlagen“ oder „Anwendung“ einzuführen und von anderen Kapiteln abzugrenzen. Vor allem, weil das Modul „Programmierpraktikum“ damals noch keine Grundlagen zu Softwaredesign vermittelte, hätten diese viel Platz eingenommen. Diese Arbeit soll sich vielmehr wie eine Geschichte lesen und den Weg zu einer guten Architektur anhand eines Problembeispiels ebnen und wiedergeben. Dadurch können Gedankengänge, Lernprozesse und die einzelnen Stadien während dieses anhaltenden Schleifprozesses besser nachvollzogen werden. Denn nichts anderes ist der Kern der Softwareentwicklung: Ein andauernder, künstlerischer Prozess des Abwägens, Überdenkens und Evaluierens.

Ehrlich gesagt, ist mir die Note egal, solange sie einer Promotion nicht im Weg steht (*zwinker*). Ich bin einfach glücklich, letztendlich das gefunden zu haben, was mir Spaß macht: Softwaredesign.

Zusammenfassung

Softwareprojekte erfordern viel Hingabe und Leidenschaft, um sie zu stemmen. Je größer das Projekt und je komplexer die Logik wird, desto wichtiger wird Struktur und sauberer Code. Dazu kommt, dass viel zu oft Entwickler sich in technischen Finessen verlieren und das Problem, der eigentliche Grund, warum die Software geschrieben werden soll, in den Hintergrund tritt. Das Verständnis des Problems scheitert vor allem mangels Analyse und Zusammenarbeit mit den Menschen, die dieses Problem schon kennen oder gelöst haben wollen. Ordnung geht verloren und die geschriebene Software wird eine große Matschkugel (Big Ball of Mud) voller Spaghetti-Code. Domänengetriebene Gestaltung (Domain-driven Design, DDD) rückt die Kollaboration mit diesen Menschen in den Vordergrund, um nicht nur einen tiefen Einblick in das Problem und seine Lösung zu bekommen, sondern auch um Analysemodell und Codemodell miteinander zu verschmelzen. Auf jeder Ebene der Domänenerkundung bietet dieses Vorgehensweisen, Muster, Regeln und Prinzipien, um Komplexität zu zerlegen und im Team zu bewältigen. Daneben gibt es weitere bewährte Stile und Ansätze, die die Ordnung und Struktur von Software fördern.

Die angesprochenen Aspekte werden in der vorliegenden Arbeit behandelt, wobei der Fokus auf Handhabung komplexer Geschäftslogik mittels DDD liegt. Zum besseren Verständnis wird dies an einem Problem aus der universitären Lehre, dem Übungsbetrieb, und seiner derzeitigen Softwarelösung, dem Abgabesystem der Heinrich-Heine-Universität Düsseldorf (AUAS), erläutert. In dem ersten Kapitel wird anhand dessen die Problemstellung motiviert. Im zweiten Kapitel wird der Begriff der Softwarearchitektur, ihr Nutzen und Qualitätsanforderungen erklärt. Im dritten Kapitel wird eine theoretische Grundlage geebnet, wie komplexe Probleme analysiert werden sollten und welche Hilfsmittel einer kollaborativen, explorativen Analyse zur Verfügung stehen. Hauptsächlich werden die Muster und Prinzipien von DDD herausgearbeitet und anhand von Beispielen eingeführt. Zudem wird auf die Wichtigkeit von Modellen und deren Qualität eingegangen. Im vierten Kapitel wird sich der Implementierung einiger Muster gewidmet, die einem, richtig angewandt, zu sauberem, strukturiertem und ausdrucksstarken Code helfen, der tiefes Verständnis des Problems zeigt. Im fünften Kapitel wird das AUAS kurz analysiert, um anschließend das Problem mit den bis dahin vorgestellten Methoden neu anzugehen. Die Ausarbeitung schließt mit dem Fazit im sechsten Kapitel ab.

Schlussendlich sollte klar werden, dass jeder dieser Aspekte, wie andere Dinge in der Soft-

wareentwicklung auch, keine „Silver Bullet“ darstellt. Jedes Problem erfordert eine auf es zugeschnittene Herangehensweise und Architektur. Zu viel von einer Sache macht das Problem komplexer, als es ist. Gute Software zu schreiben bleibt auch in Zukunft hart.

Danksagung

Hier sollte normalerweise eine ausführliche Liste an Menschen stehen, denen ich danke. Natürlich fallen darunter Prof. Dr. Michael Leuschel und meine beiden Betreuer Dr. Jens Bendisposto und Christian Meter, denen ich es überhaupt zu verdanken habe, an dem Projekt zur Erstellung eines Systems für die Verwaltung der Lehre teilnehmen zu dürfen und mit dieser Bachelorarbeit die ersten Ideen und Erkenntnisse sammeln zu können. Die Mittagspausen, die wegen Besprechungen verschoben oder sogar ausgefallen sind, mache ich wieder gut. Das ist versprochen.

Die üblichen Personen, die einen im Leben und Studium zur Seite stehen und Kraft geben, hatte ich bis vor knapp einem Jahr noch nicht. So bescheuert es klingt, danke ich meinem Körper, der diese Zeit trotz vieler schwerer äußerer und innerer Rückschläge nicht aufgegeben hat und meines Erachtens nach einer Maschine gleicht; auch, wenn ich jetzt merke, dass das letzte Jahrzehnt so langsam seinen Tribut fordert. Ich hoffe, dass bald Ruhe einkehrt. Ich wünsche es meiner Schwester und mir so sehr.

Und ich danke meiner Nina für die letzten 9 Monate und die Zeit, die noch kommen wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In diesem kurzen Kapitel wird geklärt, was diese Ausarbeitung behandelt und warum dies wichtig ist.

1.1 Motivation

Tag für Tag steht der Mensch vor Problemen mit unterschiedlichster Merkmalen und in allerlei Variationen. Jedes will gelöst werden. Bevor die Lösung des Problems geerntet werden kann, geht das Verständnis der Aufgabe dem Bearbeiten voraus: Bei alltäglichen Problemen eher implizit, doch bei komplexeren Aufgaben durchaus viel Raum einnehmend, ist die Analyse der gegebenen Sachverhalte einschließlich der Aufgabenstellung mitsamt Umständen der Grundpfeiler der Problembewältigung. Fast schon selbstverständlich wird sich danach der Modellierung erster Lösungsideen gewidmet, um sich dann an Hilfsmitteln zu bedienen, die für die Bewältigung der vorliegenden Aufgabe nützlich sind und Arbeit abnehmen.

Besteht ein Problem, das mit Unterstützung eines Computers gelöst werden kann, so wird Software als Hilfsmittel genommen: Entweder wird ein bereits vorhandenes Programm verwendet oder selbst Code entwickelt, der für das vordefinierte Aufgabenspektrum von Nutzen ist. Software, die nicht nur dem Nutzer Freude bei der Anwendung bereit, sondern auch den Programmierer bei der Weiterentwicklung keine Steine in den Weg legt, gibt es aber leider selten. Unsauberer, unstrukturierter Code machen Wartbarkeit zu einem kostbaren Unterfangen, woran letztendlich auch die Funktionalität leidet, da nicht mehr agil entwickelt werden

kann. Trotz aktuellster Technologien hat man eine schwer zu wartende große Matschkugel (Big Ball of Mud) vor sich, da man sich in Spaghetti-Code verfängt. Fehlendes Verständnis des Problems ist meist der Anfang allen Übels, sodass es auf die Dauer nicht klar von anderen Aufgaben der Software wie Datenpersistenz getrennt werden kann.

1.2 Problemstellung

Ziel dieser Ausarbeitung ist es, Ansätze zur Analyse und Modellierung eines Problems zu geben, um es letztendlich mit seiner Lösung in seinen Anfängen in Software klar strukturiert und simpel umzusetzen, sodass die Anforderungen an Software durch eine sinnvolle Architektur erfüllt werden können. Entsprechend folgen nach einer Begriffserklärung der Softwarearchitektur Methoden zur Analyse, durch die ein tiefes Verständnis des Problems erlangt wird. Anschließend werden Möglichkeiten, Muster und Prinzipien aufgezeigt, wie das Modell in Code übersetzt werden kann. Währenddessen werden eingeführte Sachverhalte durch beispielhaftes Anwenden auf eine Fallstudie veranschaulicht. Auf diese wird näher in einem Kapitel eingegangen, indem zuerst die bestehende Architektur der Software analysiert wird, um dann erste Ideen zur Umgestaltung auf Basis der neu gewonnen Ansätze zu implementieren. Die Fallstudie behandelt eine in die Jahre gekommene Software zur Unterstützung der universitären Lehre, die aufgrund mangelnder Qualität sehr viele Einschränkungen in der Anwendung vorgibt. Um den wichtigsten Anforderungen an Software gerecht zu werden, werden wir uns ein wenig von den normierten Qualitätsanforderungen lösen und uns in erster Linie an den klassischen Grundsäulen der Architektur nach Vitruv, angewendet auf Software, orientieren, um all ihrem Nutzen einschließlich desjenigen für Entwickler gerecht zu werden. Denn letztendlich sollte ein Architekt „nicht nur seine Musik lesen, wie es ein Ingenieur tut, sondern sie auch spielen“[1] können.

[1] Zitat aus dem Film „The House That Jack Built“ von Lars von Trier

Kapitel 2

Softwarearchitektur

In diesem Kapitel wird eine Definition für Softwarearchitektur gegeben und ihr Nutzen dargelegt.

2.1 Software und ihre Entwicklung

Software kommt aus dem Englischen und bedeutet wörtlich übersetzt „weiche Ware“. Sie wird als Gegenteil von Hardware, der „harten Ware“, auf der sie ausgeführt wird, angesehen. Hardware ist steif und starr. Sie ist materiell, man kann sie anfassen. Es bedarf an Handwerk, sie zu ändern. Software dagegen ist reines Gedankengut. Gedanken sind biegsam und leicht. Sie können einem Kaninchenbau gleichen, einem Labyrinth, einem Wollknäuel, aber auch in manchen Situationen einem klaren Gebilde. Sie sind immateriell. Der geschriebende Code ist nur eine Ausdrucksform dieses Werks.

Softwareentwicklung ist wohl eines der kreativsten Dinge auf der Welt. Nirgendwo sonst ist der Mensch seiner Phantasie so nah und kann so viel erschaffen. Nirgendwo sonst kann man so sehr Gott spielen. Nirgendwo sonst erntet man genau das, was man sät. Das Endergebnis ist so rein und unverfälscht wie kein anderes. Es ist eine Sache, die alleine durch Gedanken entstanden ist: Ein Algorithmus - nicht nur in Form von Programmcode, sondern auch in Form der Schritte die ein Nutzer - egal, ob Anwender, Entwickler oder Tester - des Programms anwenden muss, um sein Problem zu lösen. Letztendlich sind es also Gedanken, die sich in Algorithmen ordnen und manifestieren, um als Hilfsmittel zu dienen. Jedoch bedingt

Ordnung eine gewisse Struktur. Dazu kommt, dass Software noch mehr Zuständigkeiten hat, als die reine Lösung des Problems. Daten sollen unter anderem gespeichert oder mit Nutzern ausgetauscht werden. Ansonsten kann die Aufgabe unter Umständen nicht ordentlich bewältigt werden. Software neigt zur Komplexität. Es braucht deswegen Strukturen und Regeln, die das Miteinander dieser und deren Komponenten koordinieren. Gebraucht wird ein Plan, der Analyse, Modell und Design vereint. Dieser spiegelt sich in der Architektur wieder.

2.2 Architektur und ihr Nutzen

Architektur kommt aus dem Griechischen bzw. Lateinischen und bedeutet „Kunst des Bauens“. Damit kann sowohl der Prozess des Bauens, als auch das Ergebnis dessen gemeint sein. Die Kunst drückt dabei aus, dass dieser Prozess auf Kreativität, Erfahrung, Wahrnehmung, Vorstellung und Intuition basiert, also durch Wechselwirkungen aller Umstände bestimmt ist. Folglich beschreibt die Architektur in ihren Ursprüngen das planvolle Entwerfen, Gestalten und Konstruieren von Bauwerken im Hinblick auf die Wechselbeziehung zwischen Mensch, Raum und Zeit. Letzteres kann nach Vitruv, dem bekanntesten römischen Architekten (50 v. Chr.), durch drei Prinzipien erreicht werden (vergleiche [Pol+65]): utilitas (Nützlichkeit), firmitas (Stabilität) und venustas (Schönheit). Jede dieser Säulen ist gleichwichtig.

Was bedeutet das im Rahmen der Softwareentwicklung? Eine Software ist ein Bauwerk aus Gedanken. Ihre Architektur gibt nach Ralph Johnson das unter Entwicklern geteilte Verständnis der Software wieder. Dieses beinhaltet die Menge aller Strukturen, die Softwareelemente, Beziehungen, sowie Eigenschaften und Einschränkungen von beiden. Softwarearchitektur legt den Fokus auf die wichtigen Dinge, was auch immer das ist (siehe [Fow03]). Computerprogramme sollten dem Menschen im Hier und Jetzt zur Lösung eines Problems dienen. Dies kann softwaretechnisch von verschiedenen Sichtweisen aus betrachtet werden, welche die zugrundeliegenden Szenarien des Problems umschließen (siehe [Kru95]). Allgemein lässt sich aber die Eignung zur Problemlösung durch zwei Anforderungen hochhalten: Erstens sollte die Software natürlich ihrem geforderten Verhalten entsprechen, also dem eigentlichen Anwender dienen und die an sie gestellte Aufgabe lösen (funktionelle Anforderung). Zweitens sollte die Software auch für die Entwickler und Tester in dem Sinne nützlich sein, dass mit ihr leicht zu arbeiten ist (qualitative Anforderungen). Sie sollte also leicht zu handhaben sein. Dies ist vor allem für die Stabilität, die Widerstandsfähigkeit gegen äußere Einflüsse, wie beispielsweise der Technologiefortschritt, wichtig. Die Schwierigkeit einer Änderung sollte

nur auf die Modifikation an sich bezogen sein und nicht durch die vorliegende Struktur des schon Erarbeiteten schwieriger gemacht werden. Wird eine neue Anforderung implementiert, sollten die Änderungen bestmöglichst lokal bleiben und sich nicht durch das ganze System ziehen. Software sollte leicht zu verwenden und anzupassen sein. Programme werden entwickelt, um Arbeit abzunehmen, nicht um sie zu schaffen: Die Einarbeitungszeit soll nicht nur für Anwender, sondern auch für Entwickler am besten null sein. Beiden Parteien soll intuitiv klar werden, was sie als nächsten Schritt zur Lösung ihres Problems in der Anwendung bzw. im Code machen müssen. Die Schönheit und Nützlichkeit des Programms muss deswegen sowohl außen in der UI und ihrer Anwendung, als auch innen im Code und seiner Wartbarkeit liegen. Die Person, die mit dem Programm zu tun hat, soll am besten gar nicht merken, dass es sich um Software handelt. Es muss zu jedem Zeitpunkt klar sein, was war, ist und eventuell sein wird. Software wird genutzt, weil man es muss, nicht weil man es will. Die Leichtigkeit und Freiheit im Interagieren mit Software sollte deswegen einen hohen Stellenwert einnehmen. Schönheit wird maßgeblich durch die Freiheit des eigenen Handelns bestimmt und ist eine statische Eigenschaft (siehe [Sch16]).

Diese Kriterien wurden in noch differenzierter Weise mit anderen Qualitätsanforderungen unter anderem Bedienbarkeit und Modifzierbarkeit in der ISO-Norm 9126[1], abgelöst durch die ISO-Norm 25000[2], gebündelt. Sie geben die Möglichkeit die Qualität der Software zu beurteilen.

[1] `https://de.wikipedia.org/wiki/ISO/IEC_9126`
[2] `https://de.wikipedia.org/wiki/ISO/IEC_25000`

Kapitel 3

Analyse und Modellierung

Bei der Entwicklung von Software sollte der Schwerpunkt darauf gelegt werden, was sie ausmacht: Das Problem und seine Lösung. Die Aufmerksamkeit sollte auf der Sache liegen, für die die Software geschrieben wird. Dies ist der Kern, der isoliert betrachtet wird. Möglichkeiten zur Datenverwaltung oder Nutzerinteraktion sind Implementierungsdetails, die sich schnell ändern könnten, aber nicht das eigentliche Problem, was sich in seinen Grundzügen nicht so schnell ändert, betreffen. Um das Problem zu lösen, bedarf es einem tiefen Verständnis. Wege zu dem beschriebenen Verständnis werden in den folgenden Abschnitten, die sich an den entsprechenden Kapiteln in [Eva14] und [Eva03], sowie [Ver16] und [Ver13] orientieren, neben unterstützenden Hilfsmitteln und Konzepten dargelegt.

3.1 Fokus auf Fachbereich

Der Fachbereich, in dem sich das Problem und sein Kontext, wie auch Wissen, Einfluss und Aktivität spezifisch einordnet, Problemraum genannt, wird auch als Domäne (Domain) bezeichnet. Die Domäne sollte die Softwarearchitektur und -entwicklung treiben (Domain-driven Architecture) und in dieser, zentral stehend, das Herzstück und Gehirn darstellen (Domain-centric Architecture), welches nie aus dem Fokus tritt. Auf ihr sollte das größte Augenmerk, vor allem im Hinblick auf das Verständnis dieser, liegen. Domain-driven Design ist ein Prozess, bestehend aus strategischen und taktischen Mustern, die dieser Art von Softwareentwicklung zu Gute kommen, um Software letztendlich an die Domäne anzugleichen. Die Domäme kollaborativ und experimentell zu erkunden, ist das Herzstück dieses

Musterkatalogs. Denn nicht immer ist sie von Anfang in ihrer Gänze offensichtlich.

Beispiel 1. Der Fachbereich, in dem sich die Fallstudie einordnet, ist die universitäre Lehre, eingeschränkt auf den Übungsbetrieb einschließlich Verwaltung und Bearbeitung von Übungsgruppen und -aufgaben, die die Studenten lösen, um anschließend durch eine Korrektur mitgeteilt zu bekommen, ob sie einen Teil der Prüfungsleistung absolviert haben.

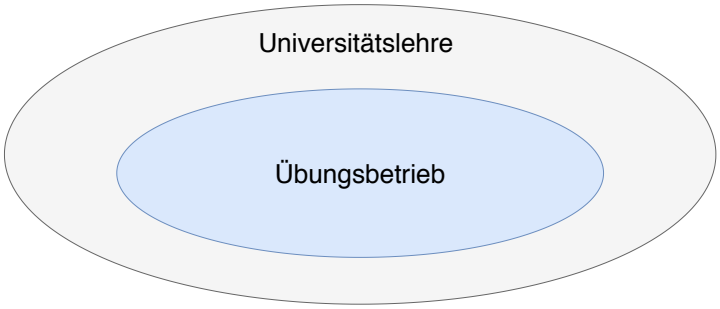

Abbildung 1: Domäne (blau) des Problembeispiels

3.2 Zerlegung des Fachbereichs

Die Domäne besteht in natürlicher Weise aus kleineren integralen Komponenten, den Subdomänen (Subdomains), auch Teildomänen genannt. Diese müssen einschließlich ihrer Grenzen zuerst erkundet werden, um einen ersten Eindruck ihrer Größe und Wichtigkeit zu erhalten: Der Fokus sollte auf den Kernbereichen der Domäne (Core Subdomains), dem Grund, warum die Software geschrieben werden soll, liegen. Generische Subdomänen (Generic Subdomains), die einem in vielen Fachbereichen gleichermaßen begegnen, und Subdomänen, die zwar zur Lösung des spezifischen Problems gebraucht werden, aber nicht den Kern darstellen (Supporting Subdomains), sollten nur die nötigste Aufmerksamkeit bekommen. Generell folgen Subdomänen dem Gesetz von Conway[1]. Demnach sind die Grenzen der Subdomänen vor allem durch die Kommunikationsstrukturen innerhalb des Fachbereichs bestimmt. Sie entsprechen also nicht zwangsläufig den Organisationsstrukturen, sondern spiegeln höchstwahrscheinlich die Kommunikationsstrukturen wider, in die sich die Domänenexperten als Personengruppen bzw. Teams einordnen lassen (siehe [Gor13]). Diese können selbst nochmal unterteilt werden. Isolation von Subdomänen erlaubt, wie allgemein das Prinzip der losen

[1] `https://de.wikipedia.org/wiki/Gesetz_von_Conway`

Kopplung auch, eine einfache abgeschottete Modifikation dieser, ohne einen Dominoeffekt auszulösen, der sich auf das gesamte System ausbreitet.

Beispiel 2. Die wichtigsten Aspekte beim Übungsbetrieb sind für das Studium das Nachhalten des Bestehens von Prüfungsleistungen. In dem Fall stellt der Student mit seinen Leistungen, die auf den Ergebnissen der Korrektur von Lösungen basieren, die Kerndomäne und unterstützende Domäne dar. Übungsverwaltung und Aufgabenbereitstellung sind zwar unerlässlich für einen ordentlichen Betrieb, aber nicht der Kern des Geschäftsproblematik.

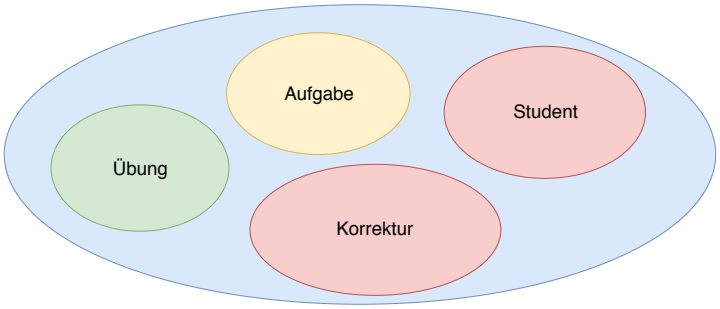

Abbildung 2: Subdomänen der Problemdomäne: Core (rot), Supporting (gelb) und Generic Domains (grün)

3.3 Verständnis durch kollaboratives Erkunden

Bei der Analyse der Domäne und ihrer Subdomänen sollten Informationen über allerlei Wege angereichert und gebündelt werden. Überlicherweise beginnt dies mit der Auflistung von Anforderungen im Beisein von Interessensvertretern (Stakeholders). Um ein bloßes Abarbeiten von Listenpunkten zu vermeiden, sollten zuerst interessante Bereiche der Domäne entdeckt werden. Dazu gehören sowohl problematische und ertragsreiche Areale, als auch Anwendungsfälle, in die diese Bereiche verwickelt sind. Währenddessen werden nicht nur bestehende Modelle, Diagramme, Dokumentation, Software und das womöglich existierende und zu ersetzende ältere System (Legacy System) miteinbezogen, sondern auch der gesamte Ablauf angeschaut, wenn keine Software zur Hilfe genommen werden kann. Vor allem aber Kollaboration mit Menschen, die in der Domäne arbeiten und Fachwissen besitzen, sogenannte Domänenexperten (Domain Experts), die für das Verständnis des Fachbereichs und seiner Aufteilung wichtig sind, ist die bedeutendste Informations- und Verständnisquelle. Ausdrucksstarke Fragen (Powerful Questions), beispielsweise was passieren würde, wenn

das System nicht in Software umgesetzt wird, regen das Gespräch an und werden vor allem durch Impact Mapping, einer Technik, die graphisch darstellt, wie die Hauptauswirkungen der Anforderungen erreicht werden, hervorgebracht werden (siehe [BA12]). Der Business Model Canvas mit seinen neun Grundblöcken ist eine weitere Möglichkeit zu erfahren, was für die Domäne wichtig ist. Nur durch diese Art von Zusammenarbeit bekommen sowohl Entwickler, als auch die Domänenexperten einen wirklich tiefen Einblick und Verständnis in die Domäne, ihre Aufteilung in Subdomänen und die ursprüngliche Intention, die das Wie hinter dem Was ist und hinter Anforderungen steckt. Verborgene wichtige Teile der Domäne müssen bewusst erkundet werden (Deliberate Discovery). Die Zusammenarbeit sollte auf ein Maximum ausgeweitet werden und nicht nur auf wöchentliche Treffen eingeschränkt sein. Durch diese Intensität werden implizite Konzepte leichter explizit festgehalten. Eine Möglichkeit, die Domäne geleitet zu erkunden, bietet Event Storming, ein Seminar, bei dem die Ereignisse, die in der Domäne stattfinden, im Mittelpunkt stehen. Ein sehr gutes Vorgehen, den Problemraum sukzessive zu analysieren, stellt [Ric18b] dar. Er schlägt vor, zuerst Spezifikationen, Anforderungen und gewünschtes Verhalten einzufangen, um dann Szenarien und Anwendungsfälle aufzuschreiben, die diese Aspekte kapseln. Danach werden die Sätze analysiert und Substantive rausgeschrieben, die oft vorkommen und demnach Domänenkonzepte ausdrücken. Nachdem diese dann semantisch gruppiert sind, wird daraus ein High-Level-View der Domäne entworfen, der dann auf tieferen Ebenen immer mehr zerlegt wird. Michael Plöd betont in einem seiner Talks (siehe [Plö18]) die Wichtigkeit, nicht nur diesen Top-Down-Approach zu gehen, sondern abwechselnd auch bei feingranulareren Domänenobjekten anzufangen und immer wieder aus einer anderen Sichtweise Domänenkonzepte zu formen.

3.4 Modelle und deren Qualität

Die Masse an Informationen könnte bei der Zusammenarbeit so gewaltig sein, dass relevante Informationen extrahiert werden müssen, was Knowledge Crunching genannt wird. Knowledge Crunching führt dazu, dass die Domäne abstrahiert wird (siehe [MT15]). Die Domäne als Ganzes zu vereinfachen, ist unter Umständen zu komplex. Außerdem könnte es sein, dass zu viele Dinge berücksichtigt werden müssen und zu viele Abhängigkeiten bestehen, sodass jede Änderung eine globale Synchronisierung erfordert, was einer agilen Zusammenarbeit im Weg steht. Folglich sollte besser jede Subdomäne in ein passendes Modell simplifiziert

werden. Ein Modell besteht aus Strukturen von Abstraktionen, sodass das eigentliche, durch Zusammenwirken aller Subdomänen lösbare Problem noch bewältigt werden kann. Es muss frei von technischen Details sein, damit sich Geschäftslogik anhand von Regeln und Richtlinien, die das Problem vorgibt, einfach entwickeln kann.

Bei der Konzeption eines Modells sollte nach [Fow96] immer wieder vor Augen gehalten werden, dass es ein menschliches Artefakt aus dem Prozess der Analyse des Problems und seiner Lösung ist. Es ist ein Produkt aus Interpretation und Meinungen. Es gibt folglich kein kanonisches Modell, kein Richtig oder Falsch, sondern nur ein weniger oder mehr nützlicheres Modell im Hinblick auf das Problem und seinen Kontext. Die Wahl des Modells und die Qualitätsanforderungen beeinflussen sich gegenseitig. Letztere sollten nur im Rahmen ihres Nutzens eingehalten werden. Es ist besser, nützliche Abstraktionen zu verwenden, als dem Missverständnis nachzugehen, das Modell der Wirklichkeit, so gut es geht, anzupassen. Es nützt auch nichts, ein flexibleres Modell zu nehmen, was wahrscheinlich nicht in Zukunft und erst recht nicht für die derzeitige Lösung des Problems gebraucht wird, wenn es zu viel Komplexität mit sich bringt. Gesucht wird das simpelste und dennoch nützlichste Modell, mit der die Aufgabe gelöst werden kann. Dies erfordert Zeit. Simplizität zu finden, kann vor allem in Anbetracht des ständigen Abwägens der Vor- und Nachteile eines konzeptionellen Elementes schwierig sein. Auf der anderen Seite soll das Modell den tiefen Einblick in die Domäne widerspiegeln, während es gleichzeitig so angepasst wird, dass es natürlicher in Software umgesetzt werden kann (siehe [Eva+19]). Analysemodell und Codemodell decken sich nach diesem Vorgehen und werden in Synergie gehalten, was als modellgetriebene Entwurf (Model-Driven Design, MDD) bezeichnet wird. MDD und DDD stoßen sich demnach gegenseitig an: Codemodellierung fördert Analysemodellierung, was wiederum ersteres unterstützt. Eine Änderung des einen bedingt die Änderung des anderen. Sie sind komplementär zueinander, was der Zusammenarbeit zwischen Entwickler und Domänenexperten weiterhin hilft.

Folglich repräsentieren Analyse und Modellierung einen fortlaufenden Prozess immer tieferen Verständnisses, der Iterationen durchläuft. Neben der Einführung von neuen Anforderungen sollte sich ein Modell vor allem dadurch weiterentwickeln. Eric Evans postuliert, dass man mindestens drei Modelle entwickeln soll, damit bei der vierten Konzeption genügend Verständnis vorhanden ist. Ein gutes, nützliches Modell ist ein Modell, was flexibel ist, Konzepte einschließlich Richtlinien und Regeln der Domäne ausdrucksstark vermittelt und als Abstraktion von einem Teil der Domäne das Design der Logik lenkt. Sogar verbesserungswürdige Schwachstellen und Eigenarten des ursprünglichen Arbeitsablaufs können dadurch

aufgedeckt und behoben werden. Anforderungen sind nicht in Stein gemeißelt und sind durch ein gutes Design leicht anpassbar.

3.5 Hilfsmittel zur Veranschaulichung

Programmiersprachen sollten während der Modellierung mit Vorsicht eingesetzt werden. Sie sind nützlich, um prototypenhaft den Einsatz von Teilen des Modells zu sehen, wie sich Interaktion mit diesen anfühlt und um neue Fragen und Einblicke in die Domäne zu erlangen. Genauso wie bei dem konzeptionellem Modell, sollte nicht an einer Modellierung festgehalten werden, sondern die Bereitschaft bestehen, diese auch verwerfen zu können (throw-away Code). Mit der Zeit kristallisiert sich heraus, welche Teile von Codemodellen nützlich sind. Es besteht aber die Gefahr, dass sich bei der Programmierung auf Sprachfunktionalitäten beschränkt wird, sich von diesen leiten lässt und die Kollaboration mit Domänenexperten bestimmen. Darüber hinaus verstehen die meisten Domänenexperten Code und dessen Abarbeitung gar nicht oder nur begrenzt. Simplizität kann dabei helfen. Auf der anderen Seite könnte eine Herangehensweise, bei der man sich nur auf Analyse ohne Feedback durch Prototypen fokussiert, in einem Zustand enden, der keinen Fortschritt erzielt (Analysis Paralysis, siehe [Mak19]) und zu unverständlich für die Domänenexperten wird. Experimentieren in der Domäne sollte einen genauso hohen Stellenwert haben wie das Erkunden der Domäne. Folglich dient eine gute Mischung aus schneller Prototypenenwicklung und Grundwerkzeugen wie Skizzen und simplen Graphen der schnellen Konzeption, Veranschaulichung und Bewertung des Modells. Komplexe Graphen und Diagramme könnten zu viel Verständnis in der Handhabung erfordern, um kurzfristig etwas zu vermitteln. UML kann zu formal sein und konzentriert sich zu sehr auf die Implementierung (siehe [Fow96]). Klasse-Verantwortung-Kollaboration-Karten (Class-Responsibility-Collaboration Cards, CRCC) sind dafür laut [MT15] besser geeignet. Auch diese Entwürfe werden durch ständige Kommunikation und Interaktion zwischen Entwicklern und Domänenexperten erarbeitet. Die Sicht auf das eigentliche Problem wird nach und nach eindeutiger und einheitlicher. Verhaltensgetriebene Entwicklung (Behavior-Driven Development, BDD, siehe [Nor06]), die auf testgetriebener Entwicklung (Test-Driven Development, TDD) basiert, unterstützt DDD, indem Szenarien anhand von Given-When-Then Aussagen festgehalten werden, um das Verhalten des Systems einzufangen, sodass die Modellentwicklung darauf aufbauen kann.

3.6 Sprache im Kontext

Anwendungsfälle und Arbeitsabläufe werden klarer formuliert, indem beide Parteien nicht nur generelles Wissen austauschen, sondern sich auch über Wortwahl und deren Bedeutung einig werden, sodass sie in die Modelle einfließen können. Die Domäne und der tiefe Einblick in diese gibt die Sprache vor. Sie ist in der Regel frei von technologischen Begrifflichkeiten der Softwareentwicklung, die die Sprache der Domänenexperten verschmutzt. Dies wird erheblich einfacher, wenn das Augenmerk darauf gelegt wird, wie das Problem in einer Zeit ohne Computer, UI und Datenbanken gelöst wurde, sofern es möglich ist. Der Fachbereich gibt zwar Fachsprache und deren Begriffe für die Kommunikation, Modellierung und Implementierung vor, jedoch müssen diese auch geteilt, erklärt und gefiltert werden, um Missinterpretation und irrelevanten Informationen vorzubeugen. Sind bedeutsame Wörter zu generisch, sollten neue passende Wörter in Kollaboration geschaffen werden. Außerdem kann es auch passieren, dass man sich bei der Benennung von Konzepten, beispielsweise bei Klassennamen bei CRCCs, aufgrund noch zu geringem Einblicks nicht sicher ist. Es wird dann dazu tendiert, überladene oder mit alten Bedeutungen behaftete Wörter oder Fachbegriffe der Domäne, die die Analyse aufgrund ihrer schon vorhandenen Bedeutung einschränken, zu verwenden oder technische Suffixe dranzuhängen, was vermieden werden sollte. Diese Wörter sollten bei Verwendung in irgendeiner Weise, beispielsweise farblich, markiert werden, um die Unsicherheit zu kennzeichnen und die explizite Namensgebung so weit wie möglich nach hinten zu schieben. Allerdings kann es dann noch immer vorkommen, dass Altlast an Bedeutung die Analyse stört. Greg Young empfiehlt deswegen, Quatschwörter bzw. Geschwafel dafür einzuführen. Alles in allem sollte ein Glossar angelegt werden, was sämtliche Begriffe, die genutzt werden, und ihre Bedeutungen zu jeder Zeit festhält. Nach und nach entwickeln sich dadurch Modelle, in denen jeweils eine gemeinsam entwickelte und genutzte Sprache (Ubiquitous Language, UL) zu finden ist. Der größte Vorteil dieser von Entwicklern und Domänenexperten gemeinsam genutzten Sprache ist, dass nicht mehr zwischen technischem Modell und geschäftlichem Analysemodell übersetzt werden muss. Umgekehrt kann ohne geteilte Sprache kein geteiltes Modell entstehen. Sie ist allgegenwärtig und überlebt sogar die Software. Die Chance, dass das Verständnis der Domäne verloren geht, wird minimiert. Es kann allerdings vorkommen, dass das gleiche Wort oder sogar die gleiche physikalische Sache verschiedene Konzepte für Experten aus verschiedenen Subdomänen repräsentiert und folglich unterschiedliche Bedeutungen in unterschiedlichen Kontexten hat. Dies in einem Modell oder gar in einer Komponente zu vereinen, führt zu Missverständnissen und widerspricht codetechnisch dem SRP. Das ist dann ein Indiz dafür, dass mehrere begrenzte

Kontexte (bounded Contexts) vorliegen, die jeweils ein eigenes Modell abschirmen sollten, dass den spezifischen Nutzen erfüllt. Ein gibt vor, wofür das Modell verwendet wird, wo es konsistent sein muss und was relevant und unwichtig ist. Er fördert Autonomie. So, wie die Subdomänen die Anwendbarkeit der Domäne abgrenzen, limitieren die begrenzten Kontexte, die linguistische Grenzen darstellen, die Anwendbarkeit und Verantwortung der jeweiligen Modelle der Subdomänen (siehe [Gor13]). Mehrdeutigkeit und Irrelevanz wird so vermieden. Andere Möglichkeiten, begrenzte Kontexte zu finden, sind, sich die Geschäftsbereiche oder Personengruppen anzugucken. Es ist nicht sinnvoll, diese Bereiche nachzumodellieren. Technisch gesehen sollte man auch alte und externe Systeme in je einem Kontext betrachten. Allerdings ist die unterschiedliche Bedeutung von einem Wort das ausschlaggebendste Kriterium für mehrere begrenzte Kontexte. Greg Young praktiziert dafür das Context Game, in dem sich alle Beteiligten in Gruppen anhand Geschäftsbereichen bzw. Subdomänen zusammenfinden und dann wichtige Begrifflichkeiten oder Sachverhalte in der Sichtweise jeder dieser Gruppen beschrieben wird. Optimalerweise besteht eine 1:1-Assoziation zwischen Subdomäne und begrenztem Kontext. Aufgrund der Tatsache, dass nicht jede Softwareentwicklung domänengesteuert ist und Subdomänen auch technische Details, wie beispielsweise Autorisierung, enthalten können, die einen eigenen Kontext erfordern, bleibt diese Bijektion meistens ein Wunschdenken.

Beispiel 3. In der Domäne der Fallstudie finden sich neben Studenten auch Angestellte. Beiden lassen sich außerdem noch Rollen, wie die des Korrektors, zuweisen. Anstatt die Domäne im Lösungsraum nur durch die Bereiche des Studenten und des Angestellten aufzuteilen, lässt sich auch feingranularer in Kontexten anhand Arbeitsablaufen gliedern (siehe folgende Abbildung). Eine andere Möglichkeit wäre, die Kontexte anhand Dingen abzugrenzen, mit denen interagiert wird, wie beispielsweise Übung, Aufgabe, Lösung und Korrektur.

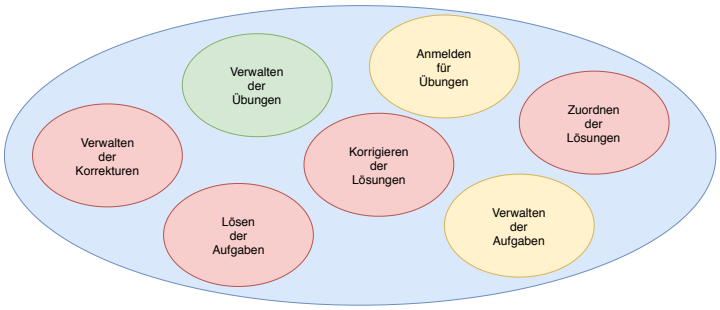

Abbildung 3: Begrenzte Kontexte der Fallstudie (nach Schwerpunkt markiert)

3.7 Integration von Kontexten

Modelle der Subdomänen haben zwar in ihrem Kontext eine explizite Bedeutung, sind aber isoliert nur wenig wert. Jedes Team sollte einen Überblick über das System haben und wissen, wie die Grenzen und Beziehungen zwischen seinem Kontext und anderen Kontexten sind. So, wie die Subdomänen im Problemraum zusammen die Domäne darstellen, müssen die begrenzten Kontexte im Lösungsraum zusammenarbeiten, um das Verhalten des gesamten Systems zu garantieren. Die bisher angesprochenen strategischen Muster von DDD - Subdomäne und begrenzter Kontext mit seiner allgegenwärtigen Sprache - zerlegen den Problem- und Lösungraum. Für das Zusammenfügen der Kontexte und ihrer Modelle stellt DDD ein weiteres strategisches Vorgehen bereit: Die Kontextlandkarte (Context Map) enthält alle begrenzten Kontexte und stellt deren integrative Übersetzungen im aktuellen Zustand dar, die gebraucht werden, um die Kollaboration zwischen ihnen zu gewährleisten. Sie zeigt die Hierarchie und Machtverhältnisse zwischen begrenzten Kontexten. Es ist kein Diagramm, was technische Aspekte wie Aufrufe widerspiegelt, sondern den Modellfluss repräsentiert. DDD stellt eine Reihe von Mustern bereit, mit denen diese Beziehungen charakterisiert werden können, die nach Michael Plöd (siehe [Plö18]) in drei Kategorien unterteilt werden: Flussaufwärts (Upstream), Flussabwärts (Downstream) und dazwischen (In-Between). Die Terminologie der Kategorisierung richtet sich dabei nach den Machteinflüssen innerhalb einer Beziehung zwischen zwei begrenzten Kontexten. Neben unausgewogenen Beziehungen, in der beteiligten Kontexte nicht gleichberechtigt sind (Upstream und Downstream), gibt es auch Muster für ausgewogene Machtverhältnisse (In-Between). Wichtig zu wissen ist, dass diese Muster als Charakterisierungen von Beziehungen zwischen zwei Kontexten dienen und nicht für einen Kontext an sich gelten. Die so entstehenden hierarchischen Modellflüsse und -verschmutzungen können folglich transitive Auswirkungen haben. Durch das explizite Festhalten dieses Wissens können sich die Teams der Kontexte besser aufeinander abstimmen, sodass integrative Probleme besser erkannt und gelöst werden.

3.7.1 Upstream

Ein Kontext, der als flussaufwärts gekennzeichnet ist, ist größtenteils frei von Einflüssen des anderen an der Beziehung teilnehmenden Kontextes bzw. seines Modells. Sein Erfolg wird maßgeblich durch eigene Leistung bestimmt, während der andere begrenzte Kontext

von ihm beeinflusst wird. Flussaufwärts liegende Systeme sind meistens offen angebotene Dienste (Open Host Services). Diese stellen eine Schnittstelle und Funktionen bereit, mit denen andere Kontexte kommunizieren können, um Informationen aus dem Upstream-Modell anzufragen.

3.7.2 Downstream

Der dem Upstream-Kontext gegenüberliegende und als flussabwärts (Downstream) gekennzeichnete, konsumierende begrenzte Kontext kann konform mit dem angebotenen Modell sein (Conformist), sodass er sich dem, was bereitgestellt wird, beugt und in sein Modell aufnimmt. Will der Downstream-Kontext sein Modell vor äußeren Einflüssen schützen, umgibt er sich selbst mit einer Antikorruptionsschicht (Anticorruption Layer), die externe Informationen anpasst und in das interne Modell übersetzt. Vor allem in Situationen, in denen ein Open Host Service ohne seinen Client nur wenig dem gesamten System nützt, bekommt letzterer ein Mitspracherecht bei der Modellierung. Die Beziehung gleicht dann einem Kunde-Lieferant-Verhältnis (Customer-Supplier).

3.7.3 In-Between

Nicht jede Beziehung muss unausgeglichen bezüglich der Machtverhältnisse sein, was die In-Between-Muster wiedergeben: Eine Partnerschaft (Partnership) kann eingegangen werden, wenn beide Kontexte gleichermaßen voneinander abhängig sind und der jeweils eine auf den Erfolg des jeweils anderen angewiesen ist. Im besten Falle sollten dann Analyse, Modellierung und Implementierung kooperativ geschehen. Mindestens aber in der Planung sollten sich beide Parteien koordinieren. Gleichen die Modelle der Kontexte sich in einigen Aspekten, so kann ein begrenzter Teil zusammen konzipiert werden. Dieser sogenannte geteilte Kern (Shared Kernel) muss in jeder Schicht des Entwurfs klar gekennzeichnet sein und möglichst klein gehalten werden. Externe Domänenereignisse fallen in diesen Bereich. Sollen die Modelle doch strikt getrennt voneinander entwickelt, betrachtet und gleichzeitig kein Kontext in der Beziehung benachteiligt werden, so eignet sich die Einführung einer von beiden Kontexten entwickelten und veröffentlichen Sprache (Published Language), die dann als indirektives Austauschformat dient. Natürlicherweise gibt es auch Kontexte, die sich nicht

untereinander beeinflussen, sodass ihre Entwicklung getrennte Wege (Separated Ways) gehen kann.

3.8 Struktur innerhalb von Modellen

Jede Strategie besteht bei genauerem Hinsehen aus vielen Taktiken. Während die strategischen Muster von DDD der Einteilung, Abgrenzung und Integration von Kontexten dienen und selbst bei einfacheren Aufgaben Mehrwert besitzen, sind die taktischen Muster dazu da, das Innere von komplexen Kontexten und ihrer Modelle zu organisieren und strukturieren. Es sind die Objekte, ihre Eigenschaften, ihr Verhalten und ihr Zusammenwirken innerhalb des Kontextes, die das Modell ausmachen. Das Paradigma der objektorientierten Programmierung eignet sich am besten für diese Kapselung und Interaktion. Entitäten, die durch Werteobjekte beschrieben werden, Aggregate und Module stellen verschiedene Ebenen der Kohäsion dar und werden von Domänendiensten, Repositories und Fabriken unterstützt, um die gesamte Geschäftslogik auszuführen. Diese und weitere Domänenobjekte ergeben sich aus Knowledge Crunching und anderen Techniken zur Domänenerkundung wie beispielsweise Event Storming und werden in der UL modelliert. Sie und andere nützliche Muster wie Lesemodelle werden in den folgenden Abschnitten beschrieben.

3.8.1 Werteobjekte

Es gibt Objekte in der Domäne, die dazu dienen, andere Dinge zu beschreiben. Sie sind identitätslos in dem Sinne, das zwei Objekte mit dem gleichen Informationsgehalt als gleich betrachtet werden. Falls es Domänenkonzepte erfordern, müssen diese sogenannten Werteobjekte (Value Objects) dazu alle notwendigen Informationen vereinen, um sie jeweils kohäsiv darzustellen. Ein Werteobjekt definiert selbst, was es heißt, ein Repräsentant von sich zu sein, weswegen es ausdrucksstark in der Sprache der Domänenexperten benannt werden sollte. Werteobjekte besitzen keinen Lebenszyklus, sodass jede Änderung von ihnen in der Erstellung eines neuen Objektes mit angepassten Daten resultiert. Komplexität, die aufgrund eines Lebenszyklus eintreten kann, fällt so weg. Diese wird an Entitäten übertragen.

Beispiel 4. Der Name und die Matrikelnummer eines Studenten sind Werteobjekte.

3.8.2 Entitäten

Manche Domänenobjekte werden als verschieden angesehen, obwohl sie die gleichen Informationen beinhalten, oder benötigen doch einen Lebenszyklus, sodass Änderungen ihrer Daten nicht in einer Erstellung eines neuen Objekts enden. Diese Domänenobjekte werden Entitäten genannt. Ihre wichtigsten Eigenschaften sind ihre Identität und Kontinuität. Dies wird über eine globale, eindeutige Eigenschaft, die natürlich sein kann oder künstlich durch die Einführung einer ID realisiert. Zwei Repräsentanten desselben Entitätstyps sind genau dann gleich, wenn ihre Werte in genau diesen Daten übereinstimmen.

Die in einer Entität enthaltenen Daten werden durch Werteobjekte, andere Entitäten oder Sammlungen von diesen gekapselt. Allerdings sollte der Fokus bei der Modellierung von Entitäten neben der Identität nur auf ihrem Verhalten liegen. Erst dieses macht ihre Bedeutung aus und ist genau wie das Ausmaß der Identität kontextabhängig, sodass eine Entität in einem begrenzten Kontext eine andere Bedeutung hat als in einem anderen Kontext. Es sollte nicht versuchen werden, die wirkliche Welt nachzumodellieren und „Haben“-Assoziationen darzustellen. Nur die Daten, die für das Verhalten und die Kontinuität der Entität wichtig sind, sollten in ihr gekapselt werden, um das Modell so simpel wie möglich zu halten. Bei der Modellierung der Entität sollte diese aber zuerst mit Attributen der Kontinuität versehen, ehe weitere Attribute für die Erhaltung der Konsistenz hinzufügt werden.

Genau wie die Werteobjekte vorgeben und überprüfen, was Instanzen von ihnen gültig macht, validiert auch jede Entität ihre Repräsentanten. Invarianten, also immer einzuhaltende Geschäftsregeln, müssen bei jeder Zustandsänderung überprüft werden, sodass ein invalider Status nie auftreten kann. Aufgrund der höheren Komplexität von Entitäten im Gegensatz zu Werteobjekten ist dies in den meisten Fällen mit mehr Aufwand verbunden.

Nicht nur das Verhalten des Domänenobjekts, sondern auch die Entscheidung, ob es als Entität oder Werteobjekt modelliert wird, ist nicht global eindeutig, sondern hängt vom begrenzten Kontext ab. Generell gilt aber, dass, wenn von einem Domänenobjekt der Lebenszyklus eingefangen werden soll, es als Entität modelliert wird. Insbesondere ist dies der Fall, wenn dieses Objekt keine Assoziation zu einer im Verhalten übergeordneten Entität hat, aber Kontinuität trotz lebenserhaltender Zustandsänderung erforderlich ist.

Beispiel 5. Eine Aufgabe, die von Angestellten erstellt wird, ist im Kontext der Aufgabenverwaltung eine Entität. Sie kann für Studenten in einem gewissen Bearbeitungszeitraum, der

ein Werteobjekt darstellt, veröffentlicht oder korrigiert werden. Letzteres ist dazu gedacht, sie nicht zu einer semantisch neuen Aufgabe zu machen, sondern nur Rechtschreib- oder Logikfehler der Aufgabenstellung auszubessern. Außerdem enthält sie in diesem Kontext auch die Punktzahl, während die Punktzahl für Verhalten der veröffentlichen Aufgabe im Kontext der Aufgabenbearbeitung nicht nötig ist, was ein Beispiel für die kontextabhängige Bedeutung ist.

3.8.3 Aggregate

Wie Entitäten untereinander und mit Werteobjekten verknüpft sind, lässt sich in Objektgraphen darstellen. Aber nicht alle Assoziationen sind zur Einhaltung von Konsistenz wichtig. Diejenigen Domänenobjekte, deren Verknüpfungen wichtig für die Konsistenz sind, werden entlang von ihnen überspannenden Konsistenzregeln zu Gruppen, sogenannten Aggregaten, zusammengefasst und nicht nach der Wirklichkeit oder dem Datenmodell gruppiert. Diese Regeln werden auch wie bei den Entitäten und den zugehörigen Werteobjekten vom Verhalten und von Invarianten vorgegeben und nicht nach der Wirklichkeit und Besitzbeziehungen modelliert. Die Aggregatgrenze ist folglich gleichzeitig die Konsistenz- und Transaktionsgrenze. Für die Einhaltung der Invarianten innerhalb eines Aggregats wird eine Entität ausgewiesen, die auch zur Kommunikation mit der Außenwelt dient. Diese wird als Aggregatwurzel (Aggregate Root) bezeichnet und ist für Klienten der Eintrittspunkt in das Aggregat. Nur über sie kann eine Änderung des Aggregats und seiner Entitäten mitsamt der beschreibenden Werteobjekte stattfinden. Folglich ist ein Aggregat die kleinste zustandsbehaftete Einheit im Produktionssystem, mit der der Klient arbeiten kann. Die Aggregatwurzel muss dazu eine globale Identität besitzen. Für die restlichen Entitäten innerhalb eines Aggregats reicht eine auf das Aggregat bezogene lokale ID. Um die Transaktionsgrenze zu wahren, referenzieren sich Aggregate untereinander mit ihren globalen IDs.

Die Größe eines Aggregats sollte möglichst kleingehalten werden, um Komplexität zu vermeiden. Ist ein Aggregat zu groß, kann der Mehrbenutzerbetrieb beeinflusst werden, da zu viele Nutzer gleichzeitig mit dem Aggregat interagieren (Hot Aggregate, siehe [Cla14]), sodass beispielsweise Wettlaufsituationen entstehen oder Sperrverfahren die Benutzerfreundlichkeit einschränken. Angefangen bei einer Entität, wird nur dann das Aggregat sukzessive um weitere Entitäten erweitert, wenn es für die Einhaltung von Invarianten und Verhalten nötig ist. Bidirektionale Assoziationen, die dabei entstehen, sollten bestmöglichst reduziert

werden, da sie meistens gar nicht direkt erforderlich sind (siehe [Dah09]). Unidirektionale Verknüpfungen reichen in den meisten Fällen aus und machen die Handhabung eindeutiger. Auch Assoziationen zwischen Entitäten in Form von Listen können, sofern es die Invarianten erlauben, durch geschickte Modellierung eliminiert werden, indem man jedem Listenelement eine Referenz zur Liste gibt, anstatt die Liste mit allen Referenzen zu den Elementen zu versehen, was wiederum Parallelitätsprobleme begünstigt.

Nicht jede Konsistenz muss sofort gegeben sein. In manchen Situationen reicht es auch, wenn garantiert wird, dass Invarianten nicht sofort, aber schlussendlich eingehalten werden. Falls beispielsweise das Aggregat in Bezug auf Komplexität zu groß wird, könnte das Abschwächen der transaktionellen Konsistenz zu dieser zeitlich verzögerten Konsistenz (Eventual Consistency) sinnvoll sein. Dies wird mit Domänenereignissen (siehe unten) erreicht. Ein tieferer Einblick in die Domäne verrät meistens, dass transaktionelle Konsistenz überbewertet ist (siehe [You10a]).

Beispiel 6. Im Kontext der Aufgabenbearbeitung gäbe es nur Probleme, wenn die Aufgabe mit Referenzen zu den Lösungsentwürfen versehen wird. Die Wahrscheinlichkeit, dass Lösungen bei entsprechender Anzahl an Studenten in der Übung verloren gehen, da sie zeitgleich abgegeben werden, sollte komplett wegmodelliert werden. Anstattdessen referenziert jede Lösung die entspreche Aufgabe von sich aus.

3.8.4 Depots

Entitäten müssen, vor allem da sie einen Lebenszyklus besitzen, auch für späteres Handeln abgelegt und hervorgerufen werden können. Da Aggregate aufgrund ihrer darstellenden Transaktionsgrenze die kleinste Einheit bilden, mit der Domänenlogik ausgeführt wird, sind es diese, die im Ganzen gespeichert und geladen werden müssen. Jeder Aggregattyp besitzt dafür ein für ihn zugeschnittenes Depot (Repository), das für das Sammeln und Bereitstellen von Repräsentanten eines Aggregattyps dient. Es ist zustandsfrei und stellt in der ubiquitären Sprache ausgedrücktes Verhalten bereit (siehe [Cal10]), durch das man Aggregate archivieren und mit bestimmten Eigenschaften suchen kann. Folglich kann das Depot auch einen Teil der Geschäftslogik übernehmen: Wenn das gewünschte Verhalten und der gesuchte Zustand beim ganzen auszuführenden Anwendungsfall eindeutig ist, kann das Depot schon Filterfunktionalitäten übernehmen, um fehlschlagendes Verhalten zu vermeiden. Globale IDs für das entsprechende Aggregat zu erstellen, ist ebenfalls durch das Depot möglich.

Beispiel 7. Das Depot, was im Kontext der Aufgabenbearbeitung die Aufgaben bereitstellt, kann direkt nach nur veröffentlichten und vom Studenten bearbeitbaren Aufgaben filtern. Denn hauptsächlich die Aufgaben mit diesem Zustand sind für den Studenten in diesem Kontext wichtig.

3.8.5 Fabriken

Ist die Erstellung eines Aggregats, einer Entität oder eines Werteobjekts zu komplex von der Logik her oder umfasst zu viele Abhängigkeiten, kann ein Objekt eingeführt werden, welches diese Aufgabe übernimmt und die Einhaltung von Invarianten während der Instanziierung garantiert. Dieses Objekt wird „Fabrik“ genannt. Ist eine Fabrik ausgewiesen, sollte zur Einhaltung der Kohäsion nur dieses mit der Objekterstellung beauftragt werden. Das Objekt selbst ist dann nur noch für sein Verhalten zuständig.

Beispiel 8. Lösungen tauchen in der ubiquitären Sprache nicht aus dem Nichts auf, sondern werden anhand von Aufgaben entworfen. Es macht also Sinn, der Aufgabe im Bearbeitungskontext, das Verhalten einer Lösungsfabrik zuzusprechen. Dabei können schon wichtige Invarianten, die Informationen einer Aufgabe enthalten, überprüft werden.

3.8.6 Domänendienste

Passt eine Funktionalität nicht richtig zu einer Entität oder einem Werteobjekt, kann ein zustandsfreien Dienst, ein sogenannter Domänendienst (Domain Service), modelliert werden, der dieses Verhalten implementiert. Vor allem Aggregat oder Kontext übergreifende Aufgaben sind so einfacher zu realisieren. Allerdings sollte die Einführung jedes Domänendienstes kritisch begutachtet werden, da die Gefahr besteht, Verhalten, was eigentlich Entitäten oder Werteobjekten zusteht, in Dienste auszulagern, sodass letztere im Extremfall zu puren Datencontainern werden und eine Verhaltensanämie entsteht. Folglich ergeben die Domänendienste zusammen eine dünne Schicht, die die restlichen Domänenobjekte umspannt. Außerdem sollten sie nur in wenigen Fällen Ereignisse (siehe unten) publizieren müssen, da letztendlich die Aggregate die Einheiten mit exekutivem Verhalten sind.

Beispiel 9. Die Zuordnung einer Menge von Lösungen passt weder zur Übungsentität, noch stellt sie ein Verhalten einer Aufgabe oder Lösung dar. Für sie könnte sich demnach ein Do-

mänendienst eignen. Nichtsdestotrotz sollte dieses Vorgehen vermerkt werden, um es später einem vielleicht bis jetzt noch unentdecktem Domänenkonzept, welches eine Entität darstellt, zuzuordnen, falls dies sich besser in die ubiquitäre Sprache fügt.

3.8.7 Module

Module fassen Domänenkonzepte zusammen, die miteinander in Verbindung stehen. Sie helfen dabei, das Modell zu verstehen, ohne sich direkt mit feingranularen Konzepten wie Entitäten oder Werteobjekten auseinanderzusetzen. Typischerweise umschließen sie ein Aggregat, also einen konsistent zu haltenden Verbund aus Entitäten und Werteobjekten. Kontexte werden durch Module gegliedert. Ob ein Domänenkonzept durch ein Modul oder einen begrenzten Kontext repräsentiert wird, kann manchmal schwierig zu bestimmen sein. Ist die ubiquitäre Sprache, die als Kontextgrenze dient, noch nicht ausgereift, sollte es als Modul realisiert und der Kontext aufgrund nicht klarer Grenzen vergrößert werden.

3.8.8 Domänenereignisse

Der Aufruf von Verhalten auf Aggregaten resultiert in geschehenen Ereignissen. Dies kann unter anderem durch Kommandos oder anderen Ereignisse ausgelöst werden. Manche Ereignisse der Domäne und ihr Resultat sind wichtig für die Subdomäne, in der sie auftreten, für die Benachrichtigung von Dritten oder das Zusammenspiel mit anderen Subdomänen oder anwendungsbezogenen Aufgaben. Diese Domänenereignisse (Domain Events) sollten demnach nicht nur für eine zeitlich verzögerte Konsistenz zwischen Aggregaten innerhalb eines begrenzten Kontextes, sondern auch für die Integration von Kontexten und Funktionalitäten der Anwendung fixiert werden. Entsprechend wird zwischen internen und externen Domänenereignissen (Internal Events, External Events) unterschieden. Letztere werden auch Integrationsevents (Integration Events) genannt. Kollaborierende Teams unterschiedlicher Kontexte sollten dafür mindestens in einer Customer-Supplier Beziehung stehen oder bestenfalls einen Shared Kernel aushandeln. Bei letzterem ist Vorsicht geboten, da zu viele und zu informationsreiche externe Domänenereignisse diesen aufblähen. Ereignisse, die für Anwendungsfunktionalitäten wie beispielsweise E-Mail-Versand interessant sind, werden als Anwendungsevents (Application Events) bezeichnet. Domänenereignisse werden auch dazu genutzt, Lesemodelle zu aktualisieren oder - auf feingranularer Ebene - eine logbasierte Da-

tenspeicherung, die Event Sourcing heißt, zu implementieren. Sie werden in der ubiquitären Sprache entwickelt. Da sie aber Geschehenes festhalten, werden sie in der Vergangenheitsform benannt.

Beispiel 10. Einer der wichtigsten Ereignisse in der Domäne ist das Veröffentlichen der Aufgaben zu Beginn des Bearbeitungszeitraums. Eine passende Bezeichnung wäre demnach „AufgabeVeroeffentlicht“. Ein Mindestmaß an Informationen, unter anderem die ID des Ereignis publizierenden Aggregats, sollte dem Ereignis mitgegeben werden.

3.8.9 Lesemodelle

Die bisher angesprochenen Muster vom taktischen DDD modellieren die Verhaltensseite. Sie sind zur Bearbeitung von Anfragen gedacht. Eine Anfrage selbst basiert auf Informationen, die dem Nutzer zur Entscheidungsfindung angezeigt werden. Diese Daten decken sich meistens nicht mit den Daten, die durch den Verhaltensaufruf konsistent gehalten werden müssen und folglich in einem Aggregat zu finden sind. Sie sind ein Konglomerat von Teilen verschiedenster Aggregate, manchmal sogar über mehrere begrenzte Kontexte verteilt. Ein Lesemodell bzw. Datenmodell (Read Model, View, Projection), das kein Verhalten aufweist, dient als reine Projektion von genau diesen Daten. Dieses kann durch ein extra dafür angelegtes Depot bereitgestellt werden. Lesemodell und Domänenmodell sind voneinander abhängig und müssen deswegen in enger Zusammenarbeit konzipiert werden. Sie ermöglichen das Architekturmuster CQRS (siehe unten).

Kapitel 4

Implementierung von Konzepten

Mit jedem Analysemodell sollte in Code experimentiert werden, um ein noch tieferes Verständnis zu bekommen. Schon beim Prototypenentwurf empfiehlt es sich, je nach Komplexität und Wichtigkeit der dazugehörigen Subdomäne und ihrem Verhalten unterschiedliche Muster zur Implementierung zu nehmen. Neben diesen werden auch Prinzipien und typische Vorgehensweisen zur Implementierung von Geschäftslogik in diesem Kapitel besprochen. Dieses Kapitel richtet sich hauptsächlich nach [Ver13] und [MT15].

4.1 DDD in Code

Für die strategischen und taktischen Muster von DDD gibt es bewährte, technische Muster, die angepasst helfen, die Hilfsmittel von DDD zu implementieren. Vor allem objektorientierte Programmiersprachen stellen verschiedene Funktionalitäten und Sprachkonstrukte dafür bereit.

4.1.1 Integration von Kontexten

Es gibt verschiedene technische Möglichkeiten, Kontexte zu integrieren. Die am schnellsten und altmodischsten erscheinenden Techniken sind Integration per Datenbank oder Dateisystem. Jeder begrenzte Kontext muss seine Daten, die Kontinuität erfordern, sowieso persistieren. Ein anderer Kontext kann diese Daten dann sofort abrufen und mit ihnen weiterarbeiten.

Folglich ist ein gewisses Maß an loser Kopplung zwischen Kontexten durch dieses Zwischenglied garantiert. Sogar die Integrationssprache ist auf Makroarchitekturebene festgehalten (bspw. SQL bei relationalen Datenbanken). Allerdings kann es mit zunehmender Auslastung zu Parallelitätsproblemen aufgrund von Sperrverfahren oder Wettlaufsituation kommen, die letztendlich die Benutzerfreundlichkeit einschränken. Außerdem wird die Integration zunehmend komplexer, wenn die Modelle oder sogar die zu wünschenden Datenbankschemata aufgrund von Weiterenwicklung divergieren. Ein anderer Nachteil ist die Kopplung an die Datenbank: Stellt die Datenbank ihren Dienst aufgrund eines Problems ein, kann keiner der an ihr hängenden Modelle für sich selbst weiterarbeiten, geschweige denn mit einem anderen kommunizieren. Integration per Dateisystem behebt zwar die Sperrprobleme einer Datenbank, bietet aber zusätzlich noch das Problem der fehlenden Abfragesprache.

Messaging und direkte Kommunikation via HTTP (per REST oder RPC) bieten bessere Alternativen, indem beispielsweise Integrationsereignisse zwischen Kontexten ausgetauscht werden. Der Nachteil ist auch hier, dass sie mehr infrastrukturelle Komplexität mit sich bringen. Näheres dazu findet man in in [HW03] und [MT15].

4.1.2 Kohäsion und ihre Zugänglichkeit

Konzepte wie begrenzte Kontexte, Module, Aggregate oder Entitäten stellen verschiedene Ebenen der Kohäsion dar. Begrenzte Kontexte können als autonome Komponenten wie beispielsweise selbst enthaltene Systeme (Self-Contained Systems) oder Mikrodienste (Microservices) in Form von Laufzeitkomponenten wie zum Beispiel JARs in Java eingesetzt oder aber als Namensräume bzw. Pakete oder ähnlichen Sprachelementen innerhalb eines Monolithen implementiert werden. Physische Aufteilung bringt den Vorteil, dass Konzepte schwieriger vermischt werden. Auch Aggregate können als einzelne, selbstständige Einheiten in beispielsweise einer JVM bereitgestellt werden. Punkte, die dafür abgewägt werden müssen, sind unter anderem Autonomie und Skalierbarkeit. Außerdem lassen sich auch Module im Sinne von DDD am besten als Module im Sinne der Programmiersprache implementieren. Kohäsion, Kapselung und Information Hiding bei Aggregaten und Entitäten, sowie Werteobjekten lässt sich neben der groben Einordnung in Module feingranularer durch Einschränkung der Sichtbarkeit und Gültigkeit regeln, zu denen auch Schnittstellen beitragen (siehe [See11]). Java bietet im Gegensatz zu C# den Vorteil von paketsichtbaren Sprachkonzepten. Dadurch können öffentliche Schnittstellen feinabgestimmt werden, während im

Paket selbst mehr Informationen zugänglich sind. Felder und Eigenschaften sollten generell klassenprivat sein. Letztendlich soll dem Klienten, der auch der Entwickler sein kann, nur das öffentlich zugänglich gemacht werden, was er in seiner Rolle ausführen darf. Beispielsweise sollte die Erstellung von Objekten nur bei einem guten Grund über mehrere Wege möglich sein.

4.1.3 Werteobjekte und Entitäten

Werteobjekte müssen zur besseren Handhabung unveränderlich sein, sodass ein Funktionsaufruf zur Veränderung der Daten in einer Instanziierung eines neuen Objekts vom gleichen Typ resultiert. Anstatt sich allerdings mit vorgegebenen Typen der Programmiersprache zufrieden zu geben, deren Wertebereiche nicht dem Wertebereich des Objektes der ubiquitären Sprache entsprechen, werden, da Werteobjekte Domänenkonzepte sind, neue ausdrucksstarke Klassen und somit Datentypen geschaffen, die sich selbst validieren, um folglich nur gültige Objekte darzustellen. Dabei kann das Werteobjekt auch spezialisiert oder mit anderen kombiniert werden. Ersteres kann man durch Wrappen als Mikrotypen implementieren, um neben noch expliziterer Terminologie Unterstützung zur Kompilierungszeit zu erhalten, wenn mehrere Mikrotypen eines übergeordneten Typs in einer Methode gebraucht werden. Der Nachteil davon ist, dass sie Komplexität in Form einer weiteren Indirektion einführen.

Zur Realisierung der Identifikation bei Entitäten hat sich der Einsatz von GUID bzw. UUID am bewährtesten gezeigt, da die Eindeutigkeit direkt gegeben ist. Als Alternative kann man eine Zeichenkette oder Klasse verwenden, die aus verschiedenen Daten der Entität eine eindeutige Identität zusammenstellt. Eine Zahlenfolge eignet sich nur bedingt, da der Zählerstand verfolgt werden muss. Allerdings entfällt dabei auch ein Übersetzen in einen künstlichen Schlüssel eines relationalen Datenbankmodells, was man bei den beiden ersten Varianten implementieren muss, sofern kein Zweit- bzw. Ersatzschlüssel für die Datenbankschicht eingeführt wird. Um einer Verhaltensanämie vorzubeugen, sollte man klassische Setter vermeiden und Getter auf ein Minimum beschränken, da es sonst sein kann, dass Geschäftslogik verstreut ist und in andere Schichten vordringt und nicht mehr in der Entität kohäsiv vorliegt und gekapselt ist. Möchte man dennoch viele Daten von der Entität der Umgebung mitteilen, so eignet sich das Mementomuster. Nachteil hierbei ist wieder die Einführung von neuen Klassen und Abhängigkeiten. Die Zustände einer Entität sind auf mehrere Arten implementierbar. Von einer einfachen Zuweisung des Zustands als Eigenschaft der Entität in Form

eines Werteobjekts durch einen Enum bzw. Enumerationsklassen (siehe [Bog08]) oder ähnliche, machtvollere Konstrukte wie einer versiegelten Klasse, algebraische Datentypen oder dem Besuchermuster (siehe [See18]), über das Statusmuster (siehe [Gam+94]), bis hin zu der Einführung von ausdrucksstarken Klassen zum expliziten Festhalten und Ausweisen von zustandsbehafteten Entitäten (siehe [MT15, Kapitel 16]) mitsamt spezifischem Verhalten kann jede Komplexität feinabgestimmt behandelt werden. Während ein Enum alle Zustände einer Klasse zur Verfügung stellt und demnach nur für Entitäten mit wenigen Zuständen übersichtlich zu gebrauchen ist, da sonst die Ordnung in der Entität aufgrund vieler If-Anweisungen leidet, wird das für die jeweiligen Zustände spezifische Verhalten beim Statusmuster in verschiedene Klassen separiert und bei der letzten Variante sogar segregiert. Können nicht in allen Zuständen alle Schnittstellen aufgerufen werden, so resultiert dies beim Statusmuster, laut Greg Young (siehe [You10b]) in vielen unnützen Methoden, in denen nur eine Exception geworfen wird. Nicht nur viel Boilerplate-Code wird geschrieben, sondern auch das Liskovsche Substitutionsprinzip (Liskov Substitution Principle, LSP) wird verletzt. Öffentlich zugängliches Verhalten wird so stark von Status zu Status abgeändert, dass dem Klienten des Codes sogar unnütze und ungültige Schnittstellen in gewissen Zuständen zur Verfügung stehen. Auch wenn die Methodensignatur anhand der ubiquitären Sprache modelliert wurde, muss umgebender Code miteinbezogen werden, um das Verhalten des Aufrufs allein anhand vom Code zu verstehen. Jeder Zustand einer komplexen Entität sollte demnach in eine extra dafür angelegte Klasse gekapselt werden, die nur die Schnittstellen bereitstellt, die in diesem Status aufgerufen werden können und gegebenenfalls in einen anderen Zustand übergehen. Gleichzeitig werden Konzepte der gemeinsamen Sprache expliziter. Dazu kommt, dass nicht jeder Kontext alle Zustände eines ähnlichen Konzepts benötigt.

Die Grundeigenschaften von Werteobjekten und Entitäten können jeweils durch einen Layer Supertype in Form von abstrakten Basisklassen repräsentiert werden, was dem DRY-Prinzip (Don't Repeat Yourself) zu Gute kommt. Eine Möglichkeit ist es, die Basisklasse aller Entitäten typisiert mit einer Identität zu versehen und gleichzeitig die Operationen zur Gleichheit anzupassen, sodass alle erbenden Klassen, die die Entitäten modellieren, nur noch Geschäftslogik beinhalten. Die Gleichheit bei der Oberklasse für Werteobjekte kann entweder durch Reflektion oder der Deklaration einer abstrakten und somit von Unterklassen bereitzustellenden Methode, die die für die Gleichheit einzubeziehenden Werte sammelt, implementiert werden. Grundsätzlich kann noch ein Schritt weitergegangen werden und noch eine weitere übergeordnetere Klasse erstellt werden, die eine vertragsbasierte Programmierung ermöglicht und ihren Unterklassen Methoden zur Prüfung von Vorbedingungen, Nachbedingungen

und Invarianten bereitstellt. Wird diese Prüfung vor allem bei Entitäten zu komplex, sollte bei Zustandsüberprüfungen die Verwendung von dem Spezifikationsmuster in Betracht gezogen werden, um dies dennoch sauber darstellen zu können. Das Domänenobjekt selbst wird dann von der genauen Überprüfung der Invarianten befreit, indem sie die Validierung an die Spezifikation delegiert.

4.1.4 Aggregate, Depots, Domänenereignisse

Die Instanziierung von Aggregaten sollte entsprechend der ubiquitären Sprachen stattfinden. Ein Aggregat fällt nicht vom Himmel, sondern entsteht immer aus einem anderen Kontext heraus. Dies kann durch das Fabrikmethodenmuster implementiert werden. SRP wird dadurch nur in den wenigsten Fällen verletzt, da das Objekt, das die Methode enthält meistens als Fabrik für mehrere Aggregate dient oder die Nähe nur ubiquitären Sprache überwiegt.

Inkonsistenzen zwischen Aggregaten müssen durch Domänenereignisse behoben werden, die extra dafür bereitgestellte Aktionen in die Wege leiten, was bei Anwendungsfällen, die aus mehreren Transaktionen bestehen, durch Prozessmanager bzw. Sagas (siehe [Ric18a]) koordiniert wird. Muster wie Mediator oder Beobachter oder sogar eigens für die Kommunikation gefundene Messaging-Muster sind zwangsläufig nötig, falls in diesen Fällen eine Entkopplung garantieren werden soll. Folglich müssen wichtige, abgeschlossene Methodenaufrufe in Domänenereignissen festgehalten werden. Aggregatwurzeln sollten diese Funktion übernehmen. Eine Möglichkeit ist es, eine statische Klasse zu implementieren, die sich sowohl um Ereignisregistrierung, als auch um Ereignispublizierung kümmert. Nachteil ist, dass das Domänenmodell dann direkt von der Infrastrukturschicht abhängig ist. Eine andere Möglichkeit besteht darin, diese Klasse in jede Methode mit öffentlich zugänglichem Verhalten zu injizieren, was sich allerdings im Hinblick auf die ubiquitäre Sprache unnatürlich anfühlt. Eine bessere Variante ist es, dass das Aggregat Domänenereignisse in einer Liste speichert, die vom zuständigen Depot abgerufen und veröffentlicht werden kann. Entweder geschieht dies durch den Rückgabewert der Methode, was unter Umständen aber CQS widerspricht, oder auch für Aggregate wird eine Basisklasse erstellt, die die Oberklasse der Entitäten erweitert, entsprechend der Kategorie der Domänenereignisse typisiert ist und Methoden zur Verwaltung der Ereignisliste mit Nähe zur ubiquitären Sprache bereitstellt. Externe Domänenereignisse sollten, vor allem weil sie nur zu verschickende Datencontainer (Data Transfer Objects) sind, sprachenagnostisch implementiert werden. Protokolle und Datenformate, die

durch eine Menge von Schlüssel-Werte ausgedrückt und bestenfalls von Schemata festgehalten werden, eignen sich dazu am besten (siehe JSON, Avro, Protocol Buffers, Cap'n Proto, etc.). Für die internen Domänenereignisse und der Publizierung können beispielsweise Framework spezifische Techniken (bspw. Application Events in Spring) genutzt werden, die durch vom Framework ermöglichte Inversion of Control leicht zu handhaben sind.

Für weitere Informationen siehe [Cla18].

4.2 Komplexe Verhaltenslogik oder simple Datenverwaltung

Nicht jeder Kontext erfordert ein Domänenmodell, welches komplexes Verhalten bereitstellt. Es gibt auch Kontexte, die eher daten- als verhaltensorientiert orientiert sind. Bei diesen würde ein Domänenmodell nach nur Ressourcen verschwenden und das Problem komplexer werden lassen, als es ist. In [Fow02] werden für die Lösung dieses Sachverhalts zwei Muster vorgestellt. Sollen nur wenig Schritte und Logik ausgeführt werden und man will größtenteils vom Datenbankmodell unabhängig bleiben, eignet sich das Transaktionsskriptmuster (Transaction Script) zur Implementierung. Bei diesem wird pro Transaktionstyp eine Klasse angelegt, die sämtlichen Code einschließlich infrastrukturellen Anliegen zur Ausführung eines Anwendungsfalls in einer Methode enthält. Vorteil dieses Musters ist es, dass man bei simplen Datenanwendungen schnell zum Ziel kommt. Wächst allerdings die Komplexität des Projekts und der Geschäftslogik, findet man sich bald in einer Klassen- und Methodenexplosion wieder. Ein anderes Muster zur einfachen Datenverwaltung ist das Tabellenmodul (Table Module). Bei diesem wird pro Datenbanktabelle eine Klasse angelegt, deren instanziiertes Objekt alle Zeilen in einer Tabelle verwaltet. Folglich wird auch hier Geschäftslogik und Datenbankkommunikation in einer Klasse vereint. Vorteil dieses Musters ist, dass die objektrelationale Unverträglichkeit (object-relational impedance mismatch) wegfällt, die bei dem Domänenmodell in aller Regel vorkommt. Dadurch muss nicht mehr zwischen Datenbankmodell und Codemodell übersetzen. Der Nachteil ist, wie bei dem Transaktionsskript auch, dass sobald die Geschäftslogik komplexer wird, beide Modelle auseinandergehen. Dies wiegt beim Tabellenmodul allerdings noch stärker, da ein Transaktionsskript besser mit Unterschiedlichkeiten in den Modellen umgehen kann.

4.3 SOLID

In diesem Abschnitt werden diese Prinzipien nicht erklärt, sondern ihr Nutzen zur Architektur zusammengetragen. Für eine genaue Beschreibung empfiehlt sich [Mar17]. Viele Prinzipien werden genauso wie Muster einfach nur stumpf angewandt. Das Single-Responsibility-Prinzip ist zwar dazu da Dinge, die sich zu verschiedenen Zeitpunkten und aus unterschiedlichen Gründen ändern, zu trennen, jedoch ist die dadurch bedingte Implementierung einer neuen Klasse eine weitere Gefahr für die Übersichtlichkeit auf Klassenebene. Dies muss abgewägt werden. Feingranularität, die auch durch das Interface-Segregation-Prinzip hervorgebracht wird, ist nicht das Ziel einer guten Architektur, wenn dadurch die Einfachheit und das Verständnis leidet. SRP sollte nur angewandt werden, wenn Symptome vorhanden sind. Andererseits bieten diese zwei Prinzipien gute Leitlinien, um auf Kontext- und Aggregate-Ebene Grenzen zu ziehen. Alles in allem sollten Prinzipien nicht blind gefolgt werden, sofern das Verständnis der Domäne dadurch verschwimmt.

4.4 Testing

Die testgetriebene Entwicklung (test-driven development, TDD) ermöglicht nicht nur Stabilität des Codes, sondern hilft auch dabei die Domäne aus einer anderen Sicht zu sehen. Dadurch, dass der Entwickler des Codes für das Domänenkonzept sich in die Lage des Klienten versetzt, werden Domänenkonzepte klarer und die ubiquitäre Sprache und ihre Anwendung auf Harmonie überprüft. Man erkennt, wie simpel und natürlich sich der Code für den Nutzer anfühlt. Die Agilität dieses Vorgehens hilft dabei, Analyse- und Codemodell gegeneinander abzustimmen, da ein TDD-Zyklus nur zur Implementierung kleiner Konzepte gedacht ist. Sowohl Unit-, als auch Integrationstests tragen zum Domänenverständnis bei.

4.5 CQRS

Das Domänenmodell kümmert sich um die Abarbeitung von Befehlen. Dazu aggregiert es Daten in Einheiten, die für das Ausführen des gewünschten Verhaltens nötig sind. Anwender treffen aber die Entscheidung, welche Befehle sie ausführen, meistens auf einem anderen

Datensatz, der in dem Domänenmodell auf mehrere Aggregate und vielleicht sogar mehrere Kontexte verteilt ist. Die angesprochenen Lesemodelle haben ein weiteres Muster ergeben: Command-Query-Responsibility-Segregation (CQRS) hebt das auf Methodenebene wirkende Comman-Query-Separation (CQS) auf die architekturelle Ebene. Das Domänenmodell ist für Schreibzugriffe verantwortlich und das Lesemodell ist für Leseoperationen zuständig. Dabei sind die Lesemodelle auf die Schnittstelle zum Anwender aufgrund der Denormalisierung abgepasst und insofern optimiert, dass schnelle Datenbankzugriffe erfolgen können. Die Konsistenz von beiden Modellen muss nur unidirektional erfolgen: Eine Änderung im Domänenmodell muss über Domänenereignisse eine Änderung im Lesemodell auslösen. Dies ist bei Integrationsevents meistens mit einer zeitlichen Verzögerung (Eventual Consistency) versehen. Für mehr Informationen siehe [Bet+13].

4.6 Eingliederung in die Anwendung

Obwohl es in dieser Ausarbeitung nur um die Modellierung der Logikschicht geht, sei trotzdem noch etwas zur Eingliederung in die Anwendung gesagt. Mit Architekturstilen wie Hexagonale (auch Ports-Adapters genannt), Clean oder Onion erreicht man eine domänenzentrale Architektur Domain-Centric Architecture, indem sie sämtliche Geschäftslogik in das zentral liegende Domänenmodell kapseln, das keine Abhängigkeiten mehr zu anderen Schichten aufweist. Infrastruktur, wie zum Beispiel Persistenz, und Presentation werden als äußere Details betrachtet, die durch Vorgaben des Domänenmodells in Form von Schnittstellen des Programmiersprache von dieser abhängig sind und nicht mehr umgekehrt. Das Modell, das das eigentliche Problem und dessen Lösung umfasst, wird isoliert. Grobere, Domänenkonzepte umspannende Dienste sollten, falls sie überhaupt gebraucht werden, die äußerste schmale Schicht des Modells bilden. Umgeben wird es von den Diensten der Anwendungen (Application Services), die Anwendungsfälle ausführen. Ihre Aufgabe besteht nur darin, als direkter Klient der Depots für das gewünschte Verhalten passende Aggregate zu laden, an die die Aufgabe weitergegeben wird. Sämtliche Geschäftslogik wird dann von diesen ausgeführt. Anschließend werden die geladenen Aggregate wieder gespeichert und das Ergebnis als Domänenereignisse für andere begrenzte Kontexte oder Aggregate, veröffentlicht. Eine gute Zusammenfassung bietet [Gra17].

Kapitel 5

Fallstudie

Der Alltag in einer Universität ist komplex und umfasst ebenso viele Bereiche wie Personengruppen. Ein wichtiger Bereich einer Hochschule ist die Lehre. Studenten absolvieren Studien- und Prüfungsleistungen, zu denen unter anderem Vorlesungen und Übungen gehören, die von Lehrkräften organisiert werden. Die Koordination von Veranstaltungen, das Nachverfolgen von Prüfungsleistungen und die Einteilung von Personal sind neben anderen Dingen essentiell für einen ordentlich laufenden Betrieb. Viele Aufgaben davon können durch Software gelöst oder unterstützt werden. Für einen Teilbereich des Betriebs existiert schon eine mehr oder weniger gute Softwarelösung: Das automatische Abgabesystem des Lehrstuhls der Informatik an der Heinrich-Heine-Universität Düsseldorf, kurz AUAS, verwaltet mitunter Übungsgruppen, Aufgabenblätter, deren von Teilnehmern eingereichte Lösungen und daraus resultierende Korrekturen einschließlich Prüfungszulassungen. Bevor dieses System im Kern partiell um- bzw. neugestaltet wird, wird kurz seine Architektur in einigen Punkten analysiert.

5.1 Analyse des alten Systems

Das AUAS ist in seiner Essenz ein Monolith. Ein Monolith ist ein Softwaresystem, das die Lösung des vordefinierten Problems in einer einzigen Anwendung umfasst. Er ist in sich geschlossen: Kommunikation mit Nutzern, alle Teilaufgaben und sämtliche Datenbankzugriffe sind in einem Programm vereint. Die Kohäsion ist, von außen betrachtet, von Natur aus hoch, da keine Funktionalität ausgelagert ist und sämtliche Geschäftslogik miteinander

kommunizieren kann. Alles wird auf einen Schlag verteilt (Single Unit of Deployment) und ausgeführt. Zudem waren in den ersten Versionen alle Laufzeitkomponenten auf einer physischen Plattform zu finden (Single-Tier). Alles ist kein Nachteil, solange die innere Struktur einfach zu warten ist und allen anderen Qualitätsanforderungen genügt. Im Gegenteil: Ein gut strukturierter Monolith ist leicht zu handhaben. Wenn allerdings die Funktionalität wächst, der Monolith scheinbar ein riesiges Ungetüm von Codezeilen wird, oder einzelne Teilaufgaben aufgrund hoher Nachfrage skaliert werden müssen, kann die Handhabung einschließlich effizienter Ressourcennutzung leiden. Ist jedoch auch die innere Ordnung aufgrund unstrukturiertem „Spaghetti"-Code und enger Verdrahtung nicht gegeben, scheitert es schon bei der anfänglichen Wartbarkeit. Die Stabilität leidet. Eine schnelle Reaktion auf wandelnde äußere Umstände ist nicht mehr gegeben. Die Produktivität sinkt. Agilität sucht man vergebens. Ohne zu wissen, hat man das verbreitetste architekturelle Gestaltungsmuster angewandt, das es gibt: Die Software wurde zu einer großen Matschkugel (Big Ball of Mud), „einem planlos strukturierten, sich ausbreitenden, unsauberen, durch Klebeband und Draht zusammengehaltenen Spaghetti-Code-Dschungel". Der Code macht zwar bestenfalls das, was er soll, aber ohne direkt zu erklären wie. Die schon gegebene, natürliche Komplexität des Problems wurde mit unbeabsichtigter Komplexität, die mit Anwendung von Technologie einhergeht, verzahnt.

Taucht man tiefer in den Code des jetzigen AUAS ein, so findet man jene Art von Komplexität: Man wird von einem Geflecht von Sprunganweisungen in Form von IF- und SWITCH-Ausdrücken erschlagen. Anliegen wie Datenbankzugriffe, Geschäftslogik und Nutzerinteraktion werden ineinander verschachtelt. Der anfängliche Wunsch nach einer Aufteilung der unterschiedlichen Anliegen und Interessen (Separation of Concerns) konnte nicht umgesetzt werden: Dem Architekturstil einer richtigen Schichtenarchitektur wurde nicht nachgegangen. Klassischerweise werden entsprechend den oben angesprochenen Anliegen drei Schichten eingeführt. Jede Schicht kapselt ihr Anliegen. Kommunikation der Schichten untereinander erfolgt in einer expliziten Hierarchie: Wenn jede Schicht nur auf die direkt unter ihr liegende Schicht zugreifen kann, nennt man dies strikte Schichtenarchitektur. Entschärft man diese Regeln, sodass höhere Schichten auf alle unter ihnen liegenden Schichten zugreifen kann, spricht man entsprechend von einer lockeren Schichtenarchitektur. Beim AUAS wurde versucht, die Schichten nach dem üblichen Verfahren anzuordnen: Die UI-Schicht greift auf die Geschäftslogik zu, welche wiederum Datenbankoperationen ausführt. Allerdings erkennt man bei genauerem Hingucken, dass die Schichten nicht klar voneinander getrennt wurden. Geschäftsobjekte stellen nicht nur Schnittstellen zum Laden und Speichern von sich selbst

bereit (Active Record), sondern auch Implementierungsdetails sind in diesen Objekten verankert. Manchmal werden sogar Transaktionsskripte (Transaction Scripts) oder Tabellenmodule (Table Modules) verwendet. Meistens ist das ein Artefakt dessen, dass zuerst die Datenbank mit ihren Tabellen bspw. anhand eines ER-Modells entwickelt wurde (bottom-up, database-first approach, data-driven-design). Wirkliche Geschäftslogik geht dann eher verloren, sodass man letztendlich bei einem System endet, was nur die möglichen Datenbankoperationen der Datenbank widerspiegelt (CRUD-Anwendung), was an sich nicht verkehrt ist, für ein System mit komplexer Geschäftslogik aber unzufriedenstellend ist. Schon allein die Namen der Dateien lassen auf ein CRUD-System schließen. Außerdem neigt dieser Ansatz auch dazu, dass Persistenz Ignoranz nicht eingehalten wird. Beim jetzigen AUAS wurde nicht nur ein relationales Datenbankmanagementsystem festlegt, sondern auch der SQL-Dialekt tief in die Software eingegliedert: Die Klasse oMySQL ist die Basisklasse, ja sogar der Layer Supertype der gesamten Geschäftslogik. Ein Austausch des Dialekts, geschweige denn des ganzen DBS ist nur schwer möglich. Zudem findet man HTML-Code, also Code für eine ganz spezielle UI in der Schicht der Geschäftslogik. Die anfänglich gut gemeinte Idee, das Model-View-Controller (MVC) Muster zu verwenden, um unter anderem das Single Responsibility Principle (SRP) einzuhalten, wurde falsch umgesetzt. Darüber hinaus gibt es für diese Art von komplexen Programmen bessere Vorgehensweisen, die die Daten, die in der UI angezeigt werden sollen, in einem eigenen Objekt kapseln, wie beispielsweise Model-View-Presenter-Viewmodel (MVPVM).

Dadurch, dass die Geschäftsobjekte viele ihrer eigentlichen Fachlichkeit übersteigende Funktionalitäten besitzen, wurde die eigentliche Geschäftslogik in andere Klassen ausgelagert. Dabei stellen die wichtigsten Objekte größtenteils nur noch ihre Daten per simplen Akzessoren bereit, was in dieser Konstellation gegen das Prinzip „Tell, don't ask“ verstößt und wieder die Stärken der Kapselung vernachlässigt. Das Modell wurde schon allein aufgrund von Designentscheidungen in eine Verhaltensanämie getrieben. Anstatt die Geschäftslogik in verschiedene voneinander unabhängige Bereiche aufzuteilen, musste die Kapselung leiden, um Herr über die Komplexität eines einziges monolithischen ER-Modells für die ganze Software zu werden. In seiner Essenz ist der aktuelle Code prozedural. Die Stärken der Objektorientierung wurden nahezu gar nicht angewandt. Die meisten Objekte sind nur noch Datencontainer, die sich auf MySQL ausruhen. Alles in allem wurden entweder nicht passende Muster und Entwurfsstile vereint oder passende Muster falsch bzw. nicht konsequent umgesetzt, was letztendlich in einer unschönen, instabilen Architektur mit eingeschränktem Nutzen endete.

5.2 Erste Schritte zum neuen System

Erste Schritte zur Implementierung, die auf den vorherigen Kapiteln mit ihren Beispielen aufbauen, findet man im Quellcode im GitLab der Informatik der HHU.

Kapitel 6

Fazit

Gute Software zu schreiben ist hart. Auch wenn es bei einfachen Softwareprojekten reicht, keine ordentliche Struktur und gute Architektur zu haben und zu pflegen, kann dies doch für die Zukunft von Nutzen sein. Schnell kann ein kleines Projekt großen Andrang finden und die Entwickler verbringen mehr Zeit damit, sich in dem Spaghetti-Code zurechtzufinden. Je komplexer das Problem und seine Logik wird, desto wichtiger ist auch das Verständnis des Fachbereichs. Denn nur so kann schöner Code entstehen. Domain-driven Design hilft mit seinen Mustern und Prinzipien, die Komplexität vor allem durch Kollaboration und der Entwicklung einer ubiquitären Sprache zu beherrschen, sodass ein flexibles Design entsteht und sich Analyse- und Codemodell decken. Andere Stile und Muster, wie CQRS geben aufgrund weniger Nachteile einen ungeheuren Mehrwert für leseintensive Anwendungen, bei denen das Daten- und Domänenmodell divergieren. Ist dies nicht der Fall, so bringt man mit diesen Mitteln nur mehr Komplexität rein. Wie immer bewahrheitet sich die Aussage, dass es keine „Silver Bullet“ gibt. Jeder Entscheidung geht ein Abwägen der Vor- und Nachteile des nächsten Schrittes voraus.

Literatur

Was wäre die Menschheit nur ohne ihr über all die Jahrtausende angesammeltes Wissen?! Folgend werden vor allem die Werke aufgelistet, die diese Arbeit ermöglicht haben und denen ich mein Wissen zu verdanken habe.

[BA12] Marjory Bisset und Gojko Adzic. *Impact Mapping*. Provoking Thoughts, 2012.

[Bet+13] Dominic Betts u. a. *Exploring CQRS and Event Sourcing*. Microsoft patterns & practices, 2013.

[Bog08] Jimmy Bogard. *Enumeration classes*. Aug. 2008. URL: https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/ (besucht am 11. 09. 2019).

[Cal10] Phil Calçado. *How to write a Repository*. Dez. 2010. URL: https://philcalcado.com/2010/12/23/how_to_write_a_repository.html (besucht am 11. 09. 2019).

[Cla14] Jef Claes. *Splitting hot aggregates*. Nov. 2014. URL: https://www.jefclaes.be/2014/11/splitting-hot-aggregates.html (besucht am 11. 09. 2019).

[Cla18] Paulo Clavijo. *DDD – Aggregate Roots and Domain Events publication*. Mai 2018. URL: https://paucls.wordpress.com/2018/05/31/ddd-aggregate-roots-and-domain-events-publication/ (besucht am 11. 09. 2019).

[Dah09] Udi Dahan. *DDD & Many to Many Object Relational Mapping*. Jan. 2009. URL: http://udidahan.com/2009/01/24/ddd-many-to-many-object-relational-mapping/ (besucht am 11. 09. 2019).

[Eva03] Eric Evans. *Domain-Driven Design*. Addison Wesley, 2003.

[Eva14] Eric Evans. *Domain-Driven Design Reference*. Dog Ear Publishing, 2014.

[Eva+19] Eric Evans u. a. *Domain-driven Design Referenz*. InnoQ, 2019.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[Fow03] Martin Fowler. „Who needs an architect?“ In: *IEEE Software* 5 (2003), S. 11–13.

[Fow96] Martin Fowler. *Analysis Patterns*. Addison Wesley, 1996.

[Gam+94] Erich Gamma u. a. *Design Patterns*. Prentice Hall, 1994.

[Gor13] Lev Gorodinski. *Sub-domains and Bounded Contexts in Domain-Driven Design (DDD)*. Apr. 2013. URL: http://gorodinski.com/blog/2013/04/29/sub-domains-and-bounded-contexts-in-domain-driven-design-ddd/ (besucht am 11. 09. 2019).

[Gra17] Herberto Graca. *DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together*. Nov. 2017. URL: https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/ (besucht am 11. 09. 2019).

[HW03] Gregor Hohpe und Bobby Woolf. *Enterprise Integration Patterns*. Addison Wesley, 2003.

[Kru95] Philippe Kruchten. „The 4+1 View Model of Architecture“. In: *IEEE Software* 12.6 (1995), S. 42–50.

[Mak19] Source Making. *Analysis Paralysis*. 2019. URL: https://sourcemaking.com/antipatterns/analysis-paralysis (besucht am 11. 09. 2019).

[Mar17] Robert C. Martin. *Clean Architecture*. Prentice Hall, 2017.

[MT15] Scott Millett und Nick Tune. *Patterns, Principles, and Practices of Domain-Driven Design*. Wrox, 2015.

[Nor06] Dan North. *Introducing BDD*. März 2006. URL: https://dannorth.net/introducing-bdd/ (besucht am 11. 09. 2019).

[Plö18] Michael Plöed. *Microservices lieben Domain Driven Design*. Nov. 2018. URL: https://www.youtube.com/watch?v=Ewy3NC97Qdk (besucht am 11. 09. 2019).

[Pol+65] Vitruvius Pollio u. a. *Des Vitruvius zehn Bücher über Architektur*. Bd. 110. Krais & Hoffmann, 1865.

[Ric18a] Chris Richardson. *Managing data consistency in a microservice architecture using Sagas*. März 2018. URL: https://www.youtube.com/watch?v=7dy5WPSv2DQ (besucht am 11. 09. 2019).

[Ric18b] Chris Richardson. *Microservices Patterns*. Manning, 2018.

[Sch16] Friedrich Schiller. *Über Anmut und Würde*. CreateSpace Independent Publishing Platform, 2016.

[See11] Mark Seemann. *Interfaces are access modifiers*. Feb. 2011. URL: https://blog.ploeh.dk/2011/02/28/Interfacesareaccessmodifiers/ (besucht am 11. 09. 2019).

[See18] Mark Seemann. *Visitor as a sum type*. Juni 2018. URL: https://blog.ploeh.dk/2018/06/25/visitor-as-a-sum-type/ (besucht am 11. 09. 2019).

[Ver13] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison Wesley, 2013.

[Ver16] Vaughn Vernon. *Domain-Driven Design Distilled*. Addison-Wesley Professional, 2016.

[You10a] Greg Young. *Eventual Consistency and Set Validation*. Aug. 2010. URL: http://codebetter.com/gregyoung/2010/08/12/eventual-consistency-and-set-validation/ (besucht am 11. 09. 2019).

[You10b] Greg Young. *State Pattern Misuse*. März 2010. URL: http://codebetter.com/gregyoung/2010/03/09/state-pattern-misuse/ (besucht am 11. 09. 2019).

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 12. September 2019 Wilhelm Alexander Maximilian Thelen