Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Ein flexibles Datenmodell für prozessorientierte Geschäftsanwendungen

Stefan Schmid

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer:	Dr. rer. nat. Ivan Bogicevic, Jochen Günzel (AEB GmbH), Frederik Niehus (AEB GmbH)
Begonnen am:	24. Oktober 2016
Beendet am:	8. Mai 2017
CR-Nummer:	D.2.8

Kurzfassung

Das mittelständische IT-Unternehmen AEB GmbH[1] aus Stuttgart bietet IT-Lösungen für Geschäftsprozesse im Bereich Logistik, Außenwirtschaft und Risikomanagement an und möchte mit einer neuen Softwareplatform neue Weg gehen, wie Geschäftsprozesse eines Kunden flexibel, individuell und agil in einem IT-System implementiert werden können. Ein wesentliches Merkmal soll dabei die leichte Integration von bestehenden und zukünftigen (Micro-)Services sein.

Dazu werden unter dem Projektnamen nEXt Architekturkonzepte erarbeitet, Technologien evaluiert und prototypisch implementiert. Die besondere Herausforderung besteht dabei in dem Ziel der Flexibilisierung von Geschäftsprozessdefinitionen (Flex-Process), von Datenmodellen (Flex-Model) und von darauf aufbauenden Benutzerinteraktionen (Flex-App)
Insbesondere aus der Flexibilisierung des Daten-/Objektmodells zur Laufzeit ergeben sich Problemstellungen wie beispielsweise die Sicherstellung der Konsistenz zwischen Geschäftslogik und Objektmodell oder die sinnvolle Nutzung neuer Assoziationen zwischen Geschäftsobjekten zur Laufzeit.

In dieser Masterarbeit wird ein Lösungsansatz für ein flexibles Datenmodell erarbeitet, prototypisch implementiert und schlussendlich gegenüber verschiedenen realen Szenarien evaluiert. Der Lösungsansatz stützt sich maßgeblich auf den Einsatz in einer neuen, cloudbasierten und prozessorientierten Plattform für Geschäftsanwendungen. Diese Plattform soll Neuentwicklungen deutlich vereinfachen und bestehende Services leicht integrieren können.
Das flexible Datenmodell wird dabei in Form einer eigenen Cloud-Komponente namens Model Manager konzipiert und soll die Datenbank für Clients abstrahieren. Die Metamodellierung spielt dabei zur Definition der Datenmodelle eine tragende Rolle.

[1] https://www.aeb.com/intl/index.php

Inhaltsverzeichnis

1 Einleitung **11**
- 1.1 Motivation und Umfeld der Arbeit . . . 11
- 1.2 Ziel . . . 12
- 1.3 Gliederung . . . 12

2 Aktueller Technologiestand **15**
- 2.1 Metamodellierung . . . 15
- 2.2 Domain-Driven Design (DDD) . . . 21
- 2.3 Web APIs . . . 28
- 2.4 NoSQL Datenbanken . . . 35
- 2.5 Graphcool . . . 37
- 2.6 Zusammenfassung . . . 40

3 Anforderungen **41**
- 3.1 Initiale Fragestellungen . . . 41
- 3.2 (Meta-) Modellierung . . . 41
- 3.3 Zusammenfassung . . . 46

4 Konzept zur Umsetzung **47**
- 4.1 Model Manager . . . 47

5 Implementierung eines Prototypen **55**
- 5.1 Lauffähiges Beispiel . . . 63
- 5.2 Zusammenfassung und Ausblick . . . 65

6 Evaluierung des Prototypen **67**
- 6.1 Performance . . . 67
- 6.2 Szenarien . . . 69
- 6.3 Zusammenfassung . . . 72
- 6.4 Bewertung der Evaluierungsergebnisse . . . 74

7 Fazit und Ausblick **75**
- 7.1 Fazit . . . 75
- 7.2 Ausblick . . . 76

Literaturverzeichnis **77**

Abbildungsverzeichnis

2.1 Beispiel zur sprachlichen Metamodellierung [AK03] . . . 17
2.2 Beispiel zur ontologischen Metamodellierung [AK03] . . . 18
2.3 Prinzip der Metamodellierung - nach [AK03][BGK+06] . . . 19
2.4 Prinzip der automatisierten Modellmigration nach [Spr03] . . . 20
2.5 Domain-Driven Design - Schichten (engl.: Layer) - nach [Eva04][MA07] . . . 23
2.6 Domain-Driven Design - Zusammenhang zwischen Domänen (engl.: Domains), Subdomänen (engl.: Subdomains) und Kontextgrenzen (engl.: Bounded Contexts) nach [Ver13] . . . 24
2.7 Domain-Driven Design - Patterns und Abhängigkeiten im Domänenmodell nach [MA07][Eva04] . . . 26
2.8 Navigation durch eine serverseitige Objektstruktur mittels REST über HTTP-GET-Anfragen [Bay02] . . . 29
2.9 Beispielhafte GraphQL-Kommunikation zwischen Client und Server Quelle: http://sangria-graphql.org/getting-started/ . . . 32
2.10 Relationale Datenmodelle im Vergleich zu NoSQL-Datenmodellen [Ara16] . . . 35
2.11 Logo des Datenbanksystems ArangoDB Quelle: https://www.arangodb.com/ . . . 36
2.12 Definition eines (flexiblen) Datenmodells im GraphCool-Framework . . . 38

3.1 Zusammenhang zwischen konkreten Daten, Datenmodell und Metamodell . . . 42
3.2 Das Metamodell zur Definition des Datenmodells. . . . 43

4.1 Grundlegend geplante Architektur - Quelle: AEB GmbH . . . 48
4.2 Stark vereinfachter Zugriff auf Daten innerhalb der Cloud . . . 49
4.3 Bestandteile und Aufgaben eines Model Managers (Quelle: AEB GmbH) . . . 50
4.4 Architektur des ersten Model Manager Prototypen . . . 51

5.1 Beispielhaftes Datenmodell . . . 55
5.2 Klassenstruktur des Model Manager Prototypen als UML-Diagramm . . . 59
5.3 Schematische Zusammenhänge bei der Implementierung eines GraphQL-Schemas mit der Referenzimplementierung . . . 61
5.4 Darstellung des Startvorgangs einer Model Manager Instanz . . . 63
5.5 Anlegen eines neuen Users über das GraphiQL-Interface des Model Managers 64
5.6 Abfragen des angelegten Users aus Abbildung 5.5 über das GraphiQL-Interface des Model Managers . . . 64

6.1 Laufzeiten für die Erzeugung verschieden großer Dokumente mit dem Model Manager 68
6.2 Faktor der Laufzeitänderung im Verhältnis zum Faktor der Dateigrößenänderung 68

Verzeichnis der Listings

2.1 HTTP-GET-Request zur Abfrage des Warenkorbs [Bay02] 29
2.2 HTTP-Result zur Warenkorb-Abfrage aus Listing 2.1 [Bay02] 30
2.3 HTTP-PUT-Request zur Anlage eines neuen Artikels [Bay02] 30
2.4 HTTP-Result zum angelegten Artikel aus Listing 2.3 [Bay02] 30
2.5 HTTP-POST-Request zur Bestellung von Artikel 961 [Bay02] 31
2.6 Einfache GraphQL-Query [Fac17] 33
2.7 JSON-Rückgabe der einfachen GraphQL-Query aus Listing 2.6 [Fac17] 33
2.8 Beispiel eines GraphQL Object Type [Fac17] 34
2.9 GraphQL Query und Mutation Type [Fac17] 34

5.1 Einfaches Datenmodell als JSON-Definition 56
5.2 Definition von NPM-Abhängigkeiten in der Datei „package.json“ 57
5.3 Generierung eines GraphQL-Schemas mithilfe der GraphQL JavaScript Referenzimplementierung 60
5.4 Beispielhafte Implementierung einer Filteranfrage für RootEntities 62

1 Einleitung

Bei dieser Masterarbeit handelt es sich um eine externe Arbeit in Zusammenarbeit mit der Firma AEB GmbH[2] und der Universität Stuttgart[3] im Bereich Softwaretechnik[4].
In den folgenden Kapiteln 1.1 und 1.2 wird zum einen die Motivation der Arbeit in Zusammenhang mit dem Projektumfeld erläutert und zum anderen werden die erwarteten Ziele spezifiziert.

1.1 Motivation und Umfeld der Arbeit

Das Unternehmen AEB möchte in einem internen Projekt namens nEXt eine neue Softwareplattform entwickeln, die die bisherigen AEB-Services leicht integrieren und in der Cloud betrieben werden kann. Anders als bei den bisherigen Softwareplattformen soll kein alles umfassender Standard geschaffen werden, jedoch mehrere Standards zur Umsetzung neuer Anwendungen. Dadurch soll erreicht werden, Kunden-individuellere Geschäftsprozess-Lösungen anbieten zu können. Dieser Wunsch kann dadurch erfüllt werden, dass die neue Softwareplattform leichtgewichtig genug ist, um in kurzer Zeit von einem Prototypen zu einer funktionierenden Anwendung zu gelangen.
Die neue Plattform soll zudem die bestehende Systemwelt durchgängig integrieren und eine agile Entwicklung neuer Anwendungen unterstützen können.

nEXt als Projekt hat dabei den Anspruch, viele neuartige Technologien und Konzepte hinsichtlich ihrer Tauglichkeit für eine flexible Geschäftsprozess-Schicht mit moderner User Experience (UX)[5] zu prüfen. Es sollen auf der grünen Wiese, frei von altem Denken, frische

[2] https://www.aeb.com/intl/index.php
[3] https://www.uni-stuttgart.de/
[4] http://www.iste.uni-stuttgart.de/se.html
[5] https://www.gruenderszene.de/lexikon/begriffe/user-experience

Ideen entwickelt und erforscht werden, um schlussendlich eine optimale Grundlage für die Zukunft des Unternehmens zu schaffen.
Diese Masterarbeit gliedert sich in das große nEXt-Projekt ein, indem ein flexibles Datenmodell konzipiert und prototypisch implementiert werden soll. Aktuell sind die Datenmodelle der Kundenlösungen relativ statisch, auch aufgrund der relationalen Datenbanken, denn eine Erweiterung des Modells zieht immer einen Neustart des Systems, zusammen mit einem Datenbankabgleich, nach sich. Darum werden Änderungen meist erst mit der Auslieferung einer neuen Softwareversion beim Kundensystem eingespielt. Das zu entwickelnde flexible Datenmodell soll im Gegensatz dazu zur Laufzeit und ohne Neustart des Systems angepasst werden können. Kombiniert mit der Vorstellung von nEXt, in Zukunft Lösungen deutlich schneller erweitern zu können, soll der gesamte Entwicklungsprozess somit schlanker und agiler werden.

1.2 Ziel

Das finale Ziel dieser Arbeit ist es, einen lauffähigen Prototypen zu implementieren und zu evaluieren. Dazu müssen erst die Anforderungen im Voraus spezifiziert werden, um einen Schritt später ein konkretes Konzept zur Umsetzung des flexiblen Datenmodells zu entwickeln. Der lauffähige Prototyp soll zum Schluss evaluiert werden, um die Performance und Flexibilitätsgrade zur Laufzeit zu messen. Im Falle positiver Ergebnisse, besteht explizit der Wunsch, den Ansatz weiter zu verfolgen, um in absehbarer Zeit, auf dem Weg zur Produktivität, eventuell erste Demo-Anwendungen darauf zu implementieren.

1.3 Gliederung

Die Arbeit gliedert sich in folgende, thematisch aufbauende Kapitel:

Kapitel 2 – Aktueller Technologiestand: Hier werden die Grundlagen der Arbeit in Form des aktuellen Technologiestands beschrieben. Dazu gehören Metamodellierung, die Web-APIs REST und GraphQL, die Konzepte des Domain-Driven Design, sowie ausgewählte Grundlagen zu NoSQL-Datenbanken.
Die im weiteren Verlauf entwickelten Konzepte und Implementierungen bauen zum Großteil auf dem hier beschriebenen Technologiestand auf.

Kapitel 3 – Anforderungen: Hier werden die Anforderungen an ein flexibles Datenmodell auch im Sinne des AEB-Kontext erläutert. Die Anforderungen stützen sich zu einem gewissen Teil auf eine Vision der AEB, wie in Zukunft neue Software im Unternehmen entwickelt werden kann und bilden die Basis für das Konzept mit prototypischer Implementierung.

Kapitel 4 – Konzept zur Umsetzung: Hier wird ein Konzept mit mehreren Stufen erarbeitet, um die bereits beschriebenen Anforderungen im Bezug auf das flexible Datenmodell möglichst gut in die Praxis umzusetzen. Dazu gehören Technologie-/Schnittstellenbetrachtungen als auch die Abgrenzung zu ähnlichen existierenden Lösungsansätzen.

Kapitel 5 – Implementierung eines Prototypen: Dieses Kapitel befasst sich mit der Implementierung des erarbeiteten Konzepts in Form eines Prototypen. Behandelt werden vor allem die Technologieentscheidungen basierend auf der gewählten Architektur, die einzelnen Programmkomponenten und etwaige Probleme.

Kapitel 6 – Evaluierung des Prototypen: Im Anschluss an die Implementierung des Prototypen wird das Ergebnis anhand realer Szenarien in der Praxis und auf Basis der Anforderungen evaluiert. Somit können mögliche Einschränkungen und Probleme herausgearbeitet werden. Es wird hier versucht, möglichst viele Szenarien abzudecken und nach Möglichkeit jeweils Verbesserungsvorschläge für die Zukunft zu erkennen.

Kapitel 7 – Fazit und Ausblick: Am Ende der Arbeit werden die vorangegangenen Themen nochmals kritisch reflektiert und zusammengefasst. Ebenso wird ein kurzer Ausblick geliefert, wie sich die Arbeit in Zukunft in die neue Softwareplattform der AEB GmbH eingliedern könnte und welche Rolle sie spielen wird.

2 Aktueller Technologiestand

Hier werden die relevanten Grundlagen der Arbeit anhand des aktuellen Technologiestands beschrieben. Diese bilden die konzeptionelle Basis zur Entwicklung des Prototypen im weiteren Verlauf.

Thematisch werden erst die Metamodellierung als Bestandteil des Model-Driven Design, gefolgt von den Konzepten des Domain-Driven Design näher betrachtet. Diese ordnen sich eher unter dem fachlichen Kontext der Arbeit ein.
Anschließend werden technische Grundlagen in Form von Web-API-Technologien, wie REST oder GraphQL, sowie NoSQL-Datenbanken erläutert.

2.1 Metamodellierung

Jonathan Sprinkle nahm in seinem IEEE Artikel im Jahr 2004 [Spr04] Bezug auf die Art und Weise der Softwareentwicklung von den Anfängen bis hin zum sogenannten Model-Driven Development. Er beschreibt die Anfänge damit, dass Wissenschaftler den Computer anfangs vor allem aus Eigeninteresse entwickelten und daher keinen Fokus auf umfangreiche Abstraktionen der Hardware legten. Folglich war es einem Außenstehenden kaum möglich, den Sinn eines Programms direkt ohne Expertenhilfe zu erfassen, da das komplette Verhalten mehr oder weniger direkt in Maschinencode und nicht besonders gut menschenlesbar geschrieben war. Durch die stärkere Verbreitung in den folgenden Jahren, wurde nach und nach die Hardware für den Softwareentwickler weg abstrahiert. Es wurden sogenannte Hochsprachen entwickelt, die per Compiler automatisch in Maschinencode übersetzt und dann ausgeführt werden können. Geht man wieder einen Schritt weiter, so landet man schnell bei Modellen, welche die Software, z.B. implementiert in einer Hochsprache, beschreiben. [Spr04]

Metamodellierung im Sinne der Softwareentwicklung ist eine Technik, welche aus dem Model-Driven Development (MDD) hervorgeht. In den letzten Jahren wurde auf diesem Gebiet einiges an Forschungsarbeit betrieben, um vor allem die Qualität und Wartbarkeit sehr großer Softwaresysteme deutlich zu verbessern. [AK03][BGK+06][MC09][Bar09][AMST09][ABK+09]

Balasubramanian et al. [BGK+06] beschreibt Model-Driven Development als eine Entwicklungsmethodik, welche vor allem große Softwaresysteme verbessern soll. Die Idee besteht darin, das Verhalten des Systems auf einer höheren Abstraktionsebene zu definieren.
Allgemein gesprochen kann eine Software durch ein Modell spezifiziert werden, welches die eigentliche Software abstrahiert. Dadurch ist es mit deutlich weniger technischem Wissen möglich, ein Programm zu verstehen oder neu zu entwickeln. [AK03][BGK+06]

Atkinson und Kuhne griffen die Metamodellierung 2003 in ihrem IEEE-Artikel „Model-driven development: a metamodeling foundation“ [AK03] auf. Sie unterscheiden beispielsweise zwischen sprachlicher (engl.: linguistic) und ontologischer (engl.: ontological) Metamodellierung mit den zwei Beispielen zur Rasse/Abstammung von Hunden aus Abbildung 2.1 und 2.2. Modelliert werden sollen die Abhängigkeiten eines Hundes namens „Lassie“, welcher von der Züchtung „Collie“ abstammt.

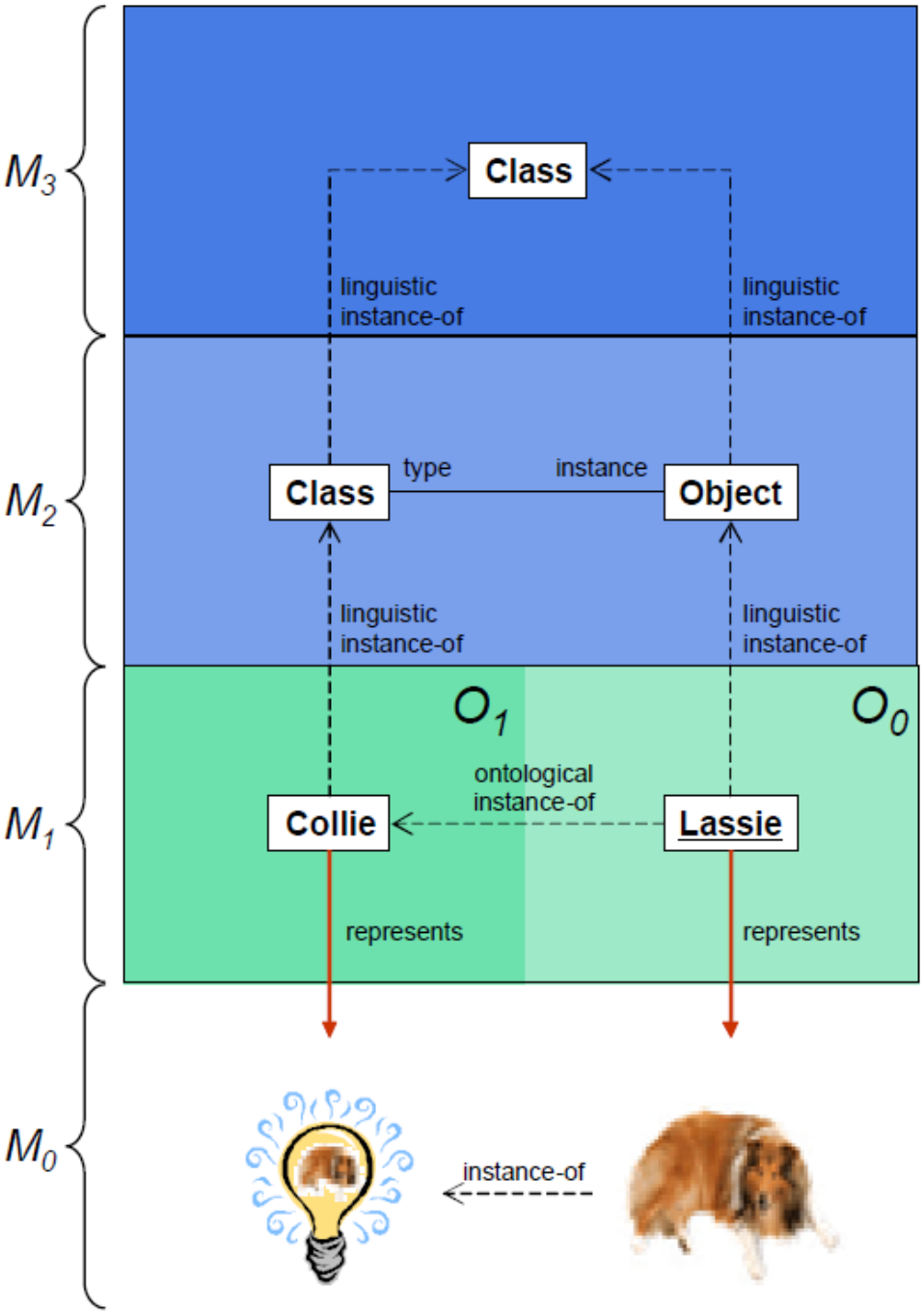

Abbildung 2.1: Beispiel zur sprachlichen Metamodellierung [AK03]

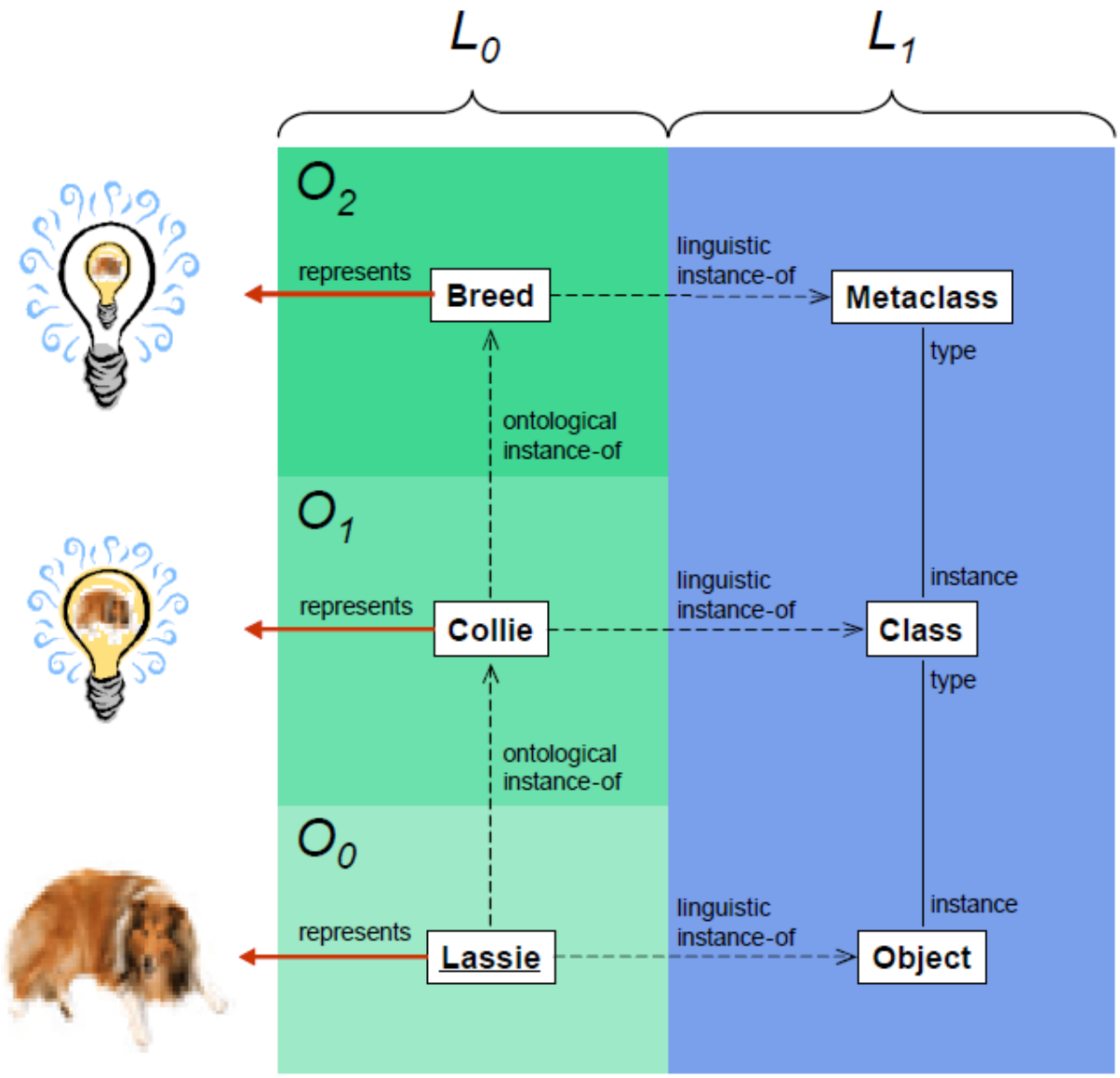

Abbildung 2.2: Beispiel zur ontologischen Metamodellierung [AK03]

Die sprachliche Metamodellierung aus Abbildung 2.1 ist in mehrere übereinander geschichtete Abstraktionsebenen eingeteilt. Die Rede ist hier auch von den Metaebenen M_0 bis M_3. Die Idee besteht darin, dass jede Metaebene durch die darüber liegende beschrieben wird. So wird beispielsweise der eindeutige Hund in Ebene M_0 durch den Namen „Lassie“ eine Abstraktionsebene höher beschrieben. Innerhalb dieser Ebene M_1 wiederum stellt „Lassie“ eine Instanz der Züchtung „Collie“ dar, bzw stammt von dieser ab.
Sowohl „Lassie“ als auch „Collie“ sind per Definition wiederum Instanzen der darüber liegenden

Metaebene M_2. Diese definiert allgemein gesprochen Objekttypen als Klassen und Instanzen von Klassen als Objekte. Somit entspricht der Name „Lassie“ einem Objekt und die Züchtung „Collie“ einer Klasse. Sowohl die Klasse als auch das Objekt aus Metaebene M_2 stellen Instanzen einer allgemeinen, abstrakten Klasse aus Metaebene M_3 dar. [AK03]

Die sprachliche Metamodellierung aus Abbildung 2.1 bietet schon einige Modellierungsmöglichkeiten, doch hat sie nach Atkinson und Kuhne auch eine entscheidende Schwachstelle. In der Praxis ist es sicherlich nötig, mehrere Hunderassen analog zu „Collie“ als Typen zu definieren. Diese neuen Typen benötigen jedoch eine gemeinsame Metaklasse, welche alle Züchtungen auch als Züchtung spezifiziert und genau hier kommt die ontologische Metamodellierung aus Abbildung 2.2 ins Spiel.
Nun stammt der Hund „Lassie“ immer noch von der Züchtung „Collie“ ab, jedoch ist diese nun eine Instanz einer Züchtung (engl.: Breed). [AK03]

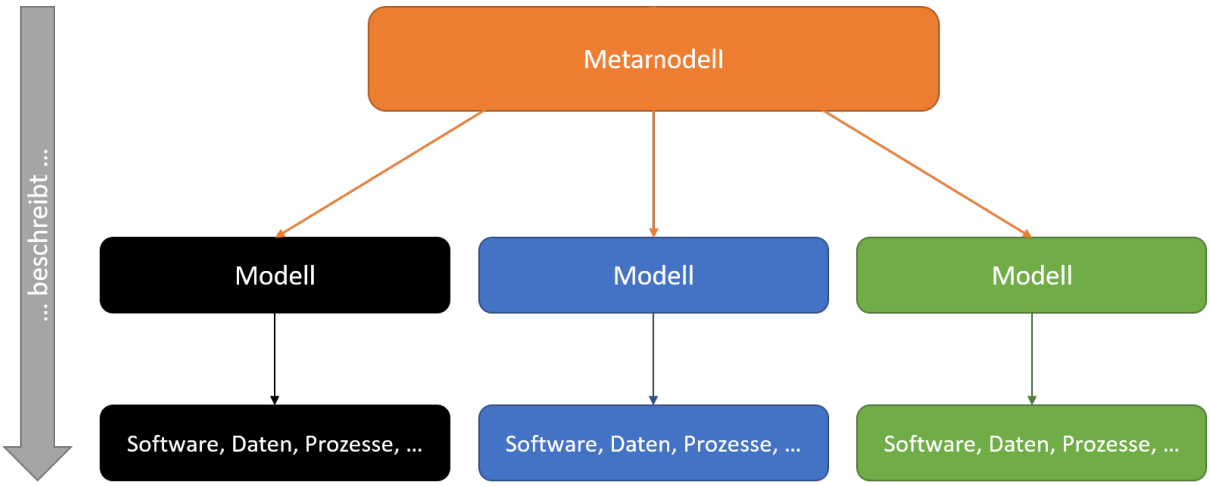

Abbildung 2.3: Prinzip der Metamodellierung - nach [AK03][BGK+06]

In Abbildung 2.3 ist der Zusammenhang zwischen einer „Realität“ in Form von beispielsweise Software, Daten oder Prozessen zum beschreibenden Modell und Metamodell nochmals abstrakter und allgemeiner, als in den vorangegangenen Beispielen aus Abbildung 2.1 und 2.2, dargestellt.
Das Metamodell auf oberster Ebene entspricht den Bereichen M_2, M_3 aus Abbildung 2.1 und dem Bereich L_1 aus Abbildung 2.2. Ein Metamodell beschreibt, wie Modelle aussehen dürfen und aus welchen Typen sie bestehen. Die Typen jedes davon instanziierten Modells werden dabei einem (Meta-)Typ auf der darüber liegenden Abstraktionsebene zugeordnet.

Die Realität in Form einer Software etc. wird durch ein Modell abstrahiert, bzw. beschrieben. Dieses definiert die vorkommenden Typen in der zu beschreibenden Software auf unterster Ebene und deren Abhängigkeiten zueinander.
Zusammenfassend lässt sich sagen, dass ein Modell eine konkrete Instanz des Metamodells darstellt und das reale System wiederum eine Instanz des beschreibenden Modells.[Spr03]

2.1.1 Automatisierte Modellmigration

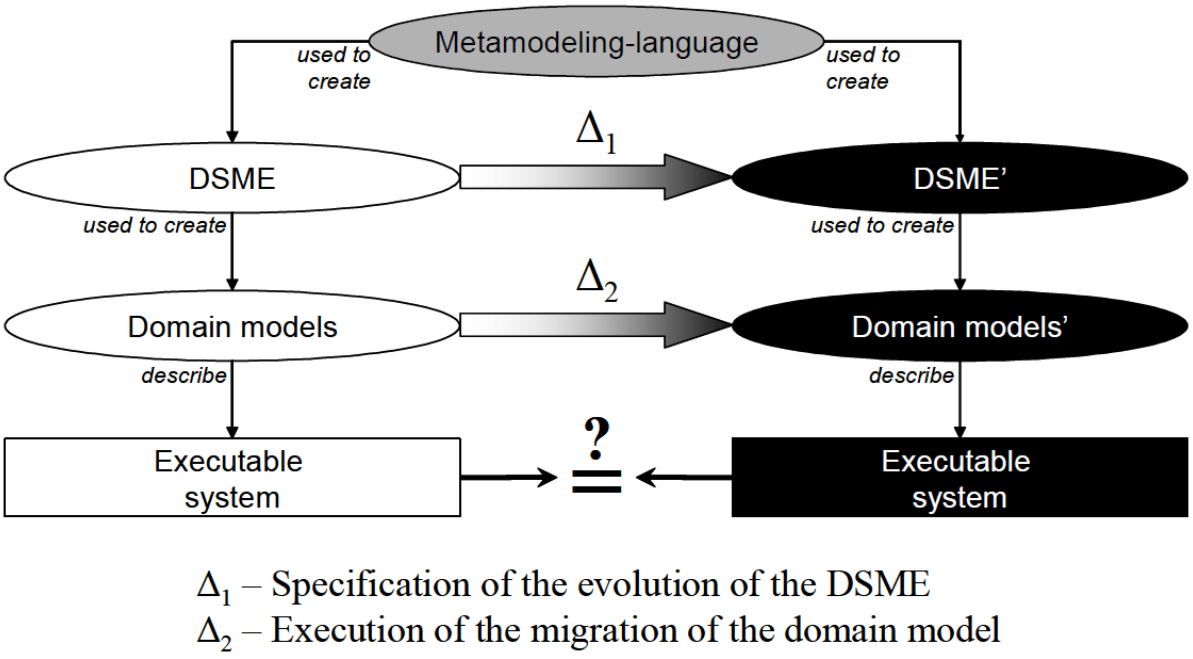

Abbildung 2.4: Prinzip der automatisierten Modellmigration nach [Spr03]

Durch die Abhängigkeiten zwischen Metamodell und Modell, wie in Abbildung 2.3 dargestellt, ergeben sich neue Fragestellungen im Sinne der Konsistenz zwischen Modell und Metamodell: Was passiert zum Beispiel mit bestehenden (alten) Datenmodellen, wenn sich das Metamodell ändert?
Wird ein bestehendes Metamodell zur Laufzeit verändert, so kann dies je nach Art der Änderung zu inkonsistenten Datenmodellen führen [RPKP09].

Dieser Frage widmete Jonathan Sprinkle im Jahr 2003 seine Dissertation [Spr03] in den USA. Sprinkle entwickelt darin ein Framework zur Metamodellierung, welches automatisiert

die abhängigen Modelle der fachlichen Domäne(n) bezüglich den Metamodelländerungen konsistent halten kann. Die Domänenmodelle können hier beispielhaft in einer üblichen UML-Klassen-Notation mit Assoziationen vorliegen.
In Abbildung 2.4 ist der Transformationsprozess beispielhaft dargestellt. Das Metamodell definiert hier die Sprache, um sogenannte Domain-Specific Modeling Environments (DSME) zu definieren. Diese stellen Entwicklungsumgebungen dar, um die wirklichen Domänenmodelle zu definieren. Der Prozess sieht nun vor, dass bei Anpassung eines DSME (gekennzeichnet durch Δ_1) automatisch die abhängigen Domänenmodelle transformiert werden (gekennzeichnet durch Δ_2) sodass die Beziehungen zwischen System, Domänenmodell, DSME und Metamodell der DSME auch nach Anpassungen immer konsistent gehalten werden. [Spr03]

Markus Herrmannsdoerfer und Daniel Ratiu zeigten 2009 [HR09] jedoch auch die Grenzen automatisierter Modellmigration auf. Sie legten die Probleme anhand von modellspezifischen Anpassungen offen, die während des Transformationsprozesses Informationen benötigen und somit eine vollautomatische Migration verhindern können.

2.2 Domain-Driven Design (DDD)

Domain-Driven Design ist eine Herangehens- oder Denkweise zur Modellierung komplexer, objektorientierter Software und gliedert sich unter dem Überbegriff Model-Driven Design, bei dem das Modell im Vordergrund steht, ein.
Die Methodik und der Begriff wurden federführend von Eric Evans in seinem Buch [Eva04] bereits im Jahr 2004 entwickelt. Evans selbst reflektierte 2009 seine fünf Jahre zuvor entwickelten Konzepte im Sinne der praktischen Anwendung [Eva09] und schrieb im Jahr 2015 nochmals eine Zusammenfassung seiner DDD Patterns und Definitionen in Form einer DDD Referenz [Eva15].
Darauf aufbauend haben Abel Avram und Floyd Marinescu im Jahr 2007 [MA07] nochmals die wichtigsten Kernpunkte herausgearbeitet. Vaughn Vernon hat ungefähr 10 Jahre später, nämlich 2012 und 2013, erneut die Konzepte Evans' aufgegriffen, indem er Vorträge zu einem sinnvollen Aggregat-Design hielt [Ver12] und zum anderen ein Buch veröffentlichte, in dem er vor allem die konkrete Umsetzung/Implementierung der Domain-Driven Design Konzepte in den Mittelpunkt stellt [Ver13]. Ein sehr praktisches Anwendungsbeispiel des Domain-Driven Design präsentierten auch Rauch und Borrmann mit ihrem Vortrag beim VKSI Entwicklertag

2014 in Karlsruhe [VKS14].

Die folgenden Konzepte werden anhand von Evans [Eva04] [Eva15], Avram und Marinescu [MA07], Vaughn Vernon [Ver13] und dem praktisch orientierten Entwicklervortrag von Rauch und Borrmann [VKS14] beschrieben.

Domain-Driven Design zielt im Kern darauf ab, die Fachlichkeit/Fachlogik beim Softwaredesign in den Vordergrund zu stellen, um den Domänenkontext bestmöglich und effizient in einer Anwendung umzusetzen.
Das Ziel des Domain-Driven Design ist es, ein domänenspezifisches Projekt zu entwickeln, welches sowohl auf dem Fachkontext, als auch dem technischen Entwicklerkontext aufsetzt. Durch diesen Ansatz wird eine intensive Kommunikation zwischen fachlichen und technischen Experten notwendig, was in der Praxis bei herkömmlichen Projektorganisationen oftmals zu kurz kommt. Ein bedeutender Grund dafür ist, dass es von beiden Seiten aus durch die unterschiedliche Sprache oftmals sehr mühsam ist, bis man ein gemeinsames Verständnis erreicht hat.

Das Hauptaugenmerk beim Domain-Driven Design liegt deshalb auf der Einführung einer einheitlichen, sogenannten ubiquitären Sprache. Sie wird definiert über das Domänenmodell, genauer gesagt über dessen Bestandteile oder Abhängigkeiten, und ist der Schlüssel zu einer deutlich effizienteren Kommunikation zwischen Entwicklern und Fach-/Domänenexperten.

2.2.1 Schichtenarchitektur

Das Domain-Driven Design sieht insgesamt vier konzeptionelle Schichten (engl.: Layer) für die Implementierung einer Anwendung vor. Wie in Abbildung 2.5 dargestellt, werden diese hierarchisch übereinander gestellt, mit der Einschränkung, dass Zugriffe jeweils nur auf tiefer liegende Schichten erfolgen dürfen. Die Entwickler werden somit bei der Implementierung gezwungen, fachlich zusammenhängenden Code in exakt nur der zugeordneten Schicht zu implementieren. Das vereinfacht wiederum Änderungen, da kein verteilter Code aus anderen Schichten gesucht werden muss. [MA07]

Die Präsentationsschicht als äußerste Hülle ist zuständig für die Anzeige von Informationen für den Benutzer und die Koordination der Interaktionen mit diesem. Sie kommuniziert

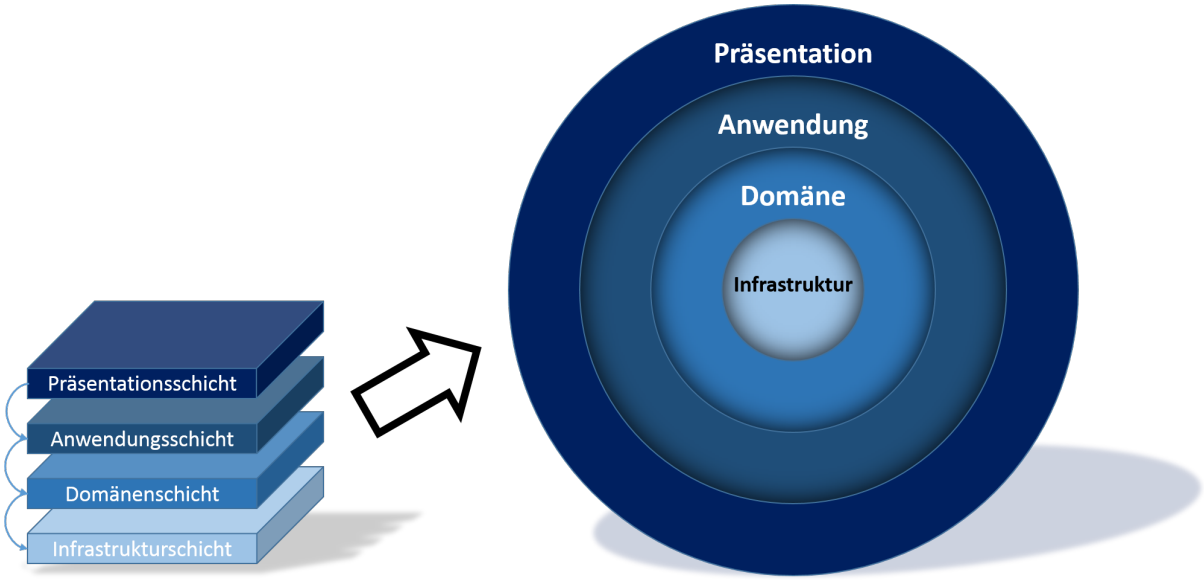

Abbildung 2.5: Domain-Driven Design - Schichten (engl.: Layer) - nach [Eva04][MA07]

eine Ebene tiefer mit der Anwendungsschicht, welche noch keine Geschäftslogik enthält, jedoch die Anwendung und die damit verbundenen Prozesse verwaltet.
Die Anwendungsschicht kommuniziert wiederum eine Ebene tiefer mit der elementarsten Schicht im Domain-Driven Design, nämlich der Domäne. Darin wird die komplette Geschäftslogik implementiert, das Datenmodell definiert und die Verwaltung der Geschäftsobjekte vorgenommen. Die Domäne steht in ständigem Kontakt mit der Schicht für Infrastruktur zur Persistierung der Datenobjekte. [MA07]

2.2.2 Domänen, Subdomänen und Kontextgrenzen [Ver13]

Vaughn Vernon beschreibt in seinem Buch [Ver13], wie in Abbildung 2.6 dargestellt, den Zusammenhang von Domänen (engl.: Domains) zu Subdomänen (engl.: Subdomain) und sogenannten Kontextgrenzen (engl.: Bounded Contexts). Eine alles umfassende Domäne entspricht der gesamten, fachlichen Geschäftsdomäne (engl.: Business Domain) und kennzeichnet die äußere Hülle in Abbildung 2.6. Sie wird fachlich unterteilt in Subdomänen, welche die gesamte Fachlichkeit weiter unterteilen (gestrichelte Linien).
Die Subdomänen werden über Kontextgrenzen klassifiziert, bzw. zusammengefasst (durchgängige Linien). Sie können zudem in Relation zu anderen Kontextgrenzen oder Subdomänen stehen.

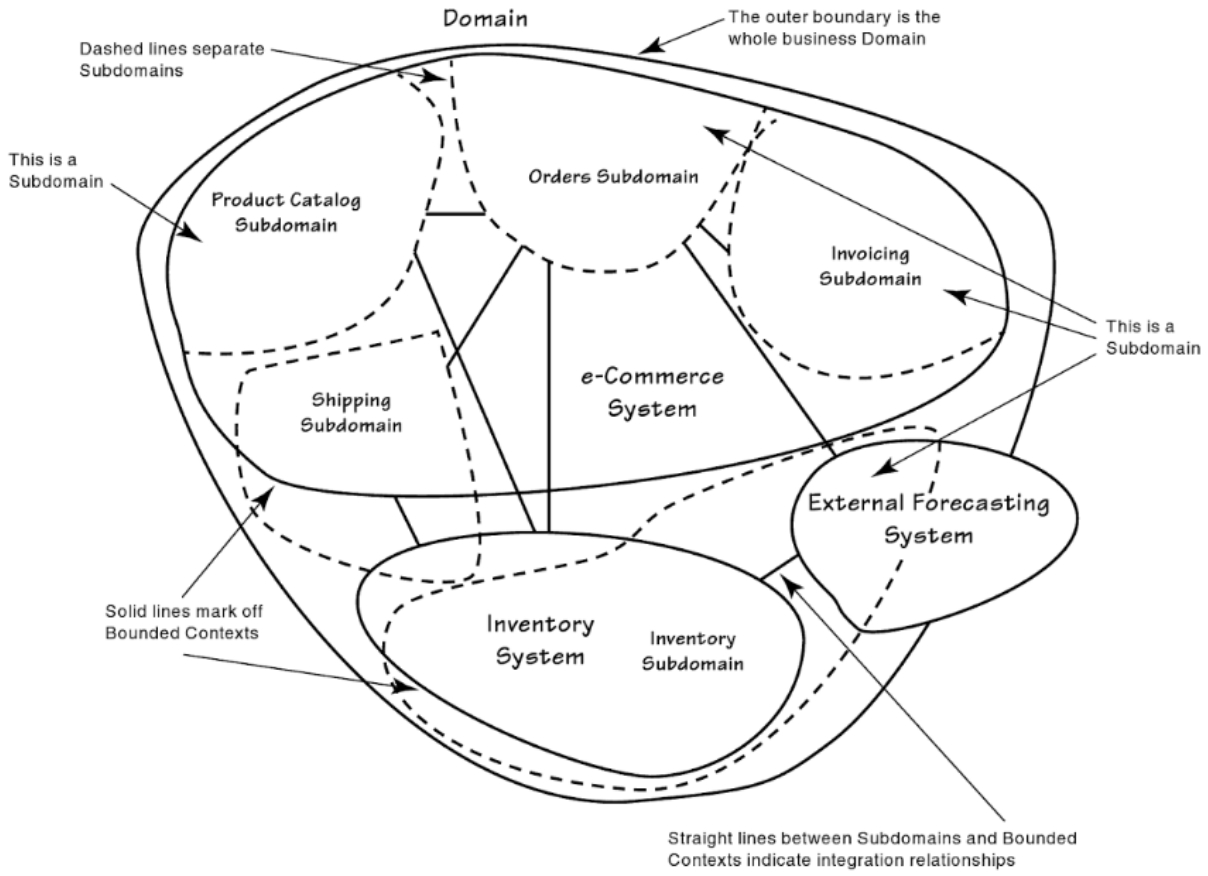

Abbildung 2.6: Domain-Driven Design - Zusammenhang zwischen Domänen (engl.: Domains), Subdomänen (engl.: Subdomains) und Kontextgrenzen (engl.: Bounded Contexts) nach [Ver13]

Angewandt auf das Beispiel aus Abbildung 2.6, soll eine Art Online-Shop modelliert werden. Dieser Shop besteht aus drei Teilsystemen, welche jeweils einen eigenen fachlichen Kontext abgrenzen, also in Form von Kontextgrenzen innerhalb der (Haupt-)Domäne modelliert werden. Diese sind beispielhaft das e-Commerce System, das Lagersystem (engl.: Inventory System) und das externe Vorhersagensystem (engl.: External Forecasting System), welches zum Großteil, jedoch nicht komplett der (Haupt-)Domäne zuzuordnen ist.
Der Kontext des e-Commerce Systems lässt sich fachlich wiederum in 4 Subdomänen einteilen, nämlich den Produktkatalog, die Bestellungen, die Rechnungserstellung und Teile des Versands. Diese Subdomänen stehen zusätzlich in Relationen zueinander, denn eine Bestellung darf unter anderem nur valide Produkte aus dem Produktkatalog beinhalten oder eine Rechnung nur auf Basis einer Bestellung ausgestellt werden.
Die fachliche Lager-Subdomäne erstreckt sich sowohl über den Lager-, als auch den

Vorhersagen-Kontext. Das Vorhersagensystem benötigt zum Beispiel zur Preisbildung anhand des aktuellen Angebots und der Nachfrage, Zugriff sowohl auf die Bestellungen, als auch den Lagerbestand. Da die Bestell-Subdomäne fachlich jedoch dem e-Commerce System zuzuordnen ist, wird sie entsprechend per Relation in den Vorhersagen-Kontext integriert. [Ver13]

2.2.3 Bestandteile eines Domänenmodells

Wie eingangs schon erwähnt, stützt sich das Domain-Driven Design maßgeblich auf die Fachlichkeit der Software, kombiniert mit einer einheitlichen, sogenannten ubiquitären Sprache. Genau mithilfe dieser Sprache wird im weiteren Verlauf das Modell der Domäne oder einzelner Subdomänen definiert und weiter spezifiziert.
Ein Domänenmodell kann als eine Art Datenmodell der Domäne interpretiert werden und weist eine gewisse Ähnlichkeit zu objektorientierten Paradigmen, z.B. in Form von UML-Notationen auf. [MA07]

Die Abbildung 2.7 definiert die Patterns und Abhängigkeiten, die zur Definition eines Domänenmodells im Sinne des Domain-Driven Design nötig sind und legt zugleich die einheitliche, ubiquitäre Sprache fest. Die Grafik kann folglich als Metamodell eines darauf aufbauenden Domänenmodells interpretiert werden.
Ein Modell wird ganz grundsätzlich beschrieben durch Entitäten, Wertobjekte und Serviceobjekte. Die elementaren Bestandteile und Abhängigkeiten des Modells aus Abbildung 2.7 werden in den folgenden Kapiteln schrittweise näher erläutert.

Entitäten (engl.: Entities)

Eine äußerst wichtige Rolle bei der Domänenmodellierung spielen die sogenannten Entitäten. Sie kennzeichnen Objekte, die über den gesamten Lebenszyklus eindeutig identifizierbar, bzw. immer von anderen Objekten gleichen Typs zu unterscheiden sind. Die Identität von Entitäten darf sich zudem nie verändern.

Betrachtet man beispielhaft eine Gruppe aus Menschen, so hat jeder bestimmte Eigenschaften wie zum Beispiel Vorname, Nachname und Geburtsdatum. Alle Menschen der gewählten Gruppe sind eindeutig voneinander zu unterscheiden, auch wenn sie rein von den 3 beispielhaften

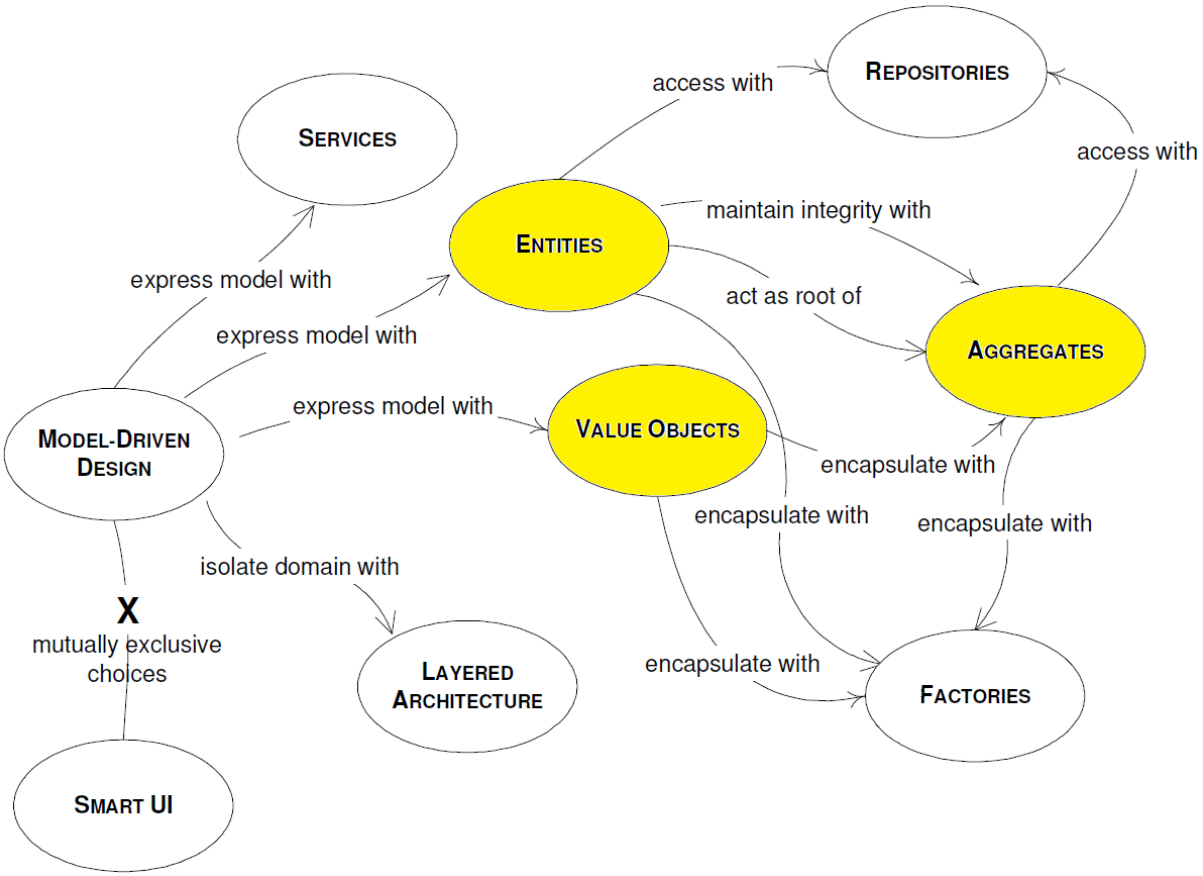

Abbildung 2.7: Domain-Driven Design - Patterns und Abhängigkeiten im Domänenmodell nach [MA07][Eva04]

Eigenschaften (Vorname, Nachname, Geburtsdatum) identisch wären. Diese Erkenntnis ist ein eindeutiger Indikator dafür, dass ein Mensch als Entität modelliert werden muss und jeder instanziierte Mensch somit eine eindeutige und unveränderliche ID bekommen sollte.
Aufgrund ihrer elementaren Rolle, sollten die Entitäten aus dem fachlichen Kontext bereits zu Beginn der Modellierung ermittelt werden. [Eva04][MA07]

Wertobjekte (engl.: Value Objects)

Im Gegensatz zu Entitäten, besitzen Wertobjekte keine Identität, d.h. sie müssen nicht von anderen unterschieden werden können. Der einzig relevante Teil sind die Attribute oder Eigenschaften. Wertobjekte sind üblicherweise unveränderlich, können also nur angelegt und wieder gelöscht werden. Dies erleichtert das Teilen von Wertobjekten innerhalb des Domänenkontext.

Betrachtet man wieder die Entität Mensch, so hat diese eine Adresse, die sich der Einfachheit halber nur aus Straße und Ort zusammensetzt. Solch eine Adresse lässt sich nicht komplett als einfaches Attribut speichern, da sie selbst aus zwei Werten, nämlich Straße und Ort besteht. Eine beispielhafte Adresse <Musterstraße, Musterstadt> hat keine Identität im Sinne eines menschlichen Individuums, jedoch sind ihre Werte von Bedeutung. Innerhalb der Domäne kann folglich die beispielhafte Adresse <Musterstraße, Musterstadt> als unveränderliches Wertobjekt modelliert und problemlos von mehreren Menschen gleichzeitig als Wohnort genutzt werden. Zieht ein Mensch um, so verändert er nicht sein Wertobjekt Adresse, da dies per Definition verboten ist, sondern erstellt ein neues, bzw. nutzt ein anderes aus der Domäne. [Eva04][MA07]

Aggregate (engl.: Aggregates)

Zusätzlich zu Entitäten und Wertobjekten existiert die Möglichkeit, Aggregate zu definieren. Diese haben einen abgeschlossenen Kontext und es besteht die Möglichkeit, Entitäten und Wertobjekt darin zu schachteln. Jedes Aggregat muss genau eine Entität als Wurzel-Entität (engl.: Root Entity oder Aggregate Root) besitzen, welches als einziges in der Lage ist, nach außen hin zu kommunizieren und dessen ID global definiert ist. Innerhalb eines Aggregats kann die Wurzel-Entität in Beziehung zu anderen Entitäten oder Wertobjekten stehen. Alle sonstigen Entitäten innerhalb eines Aggregats haben keine globale ID mehr, sondern nur noch eine lokale, denn sie können nicht nach außen kommunizieren und nicht von außen referenziert/assoziiert werden. Der einzig mögliche Einstiegspunkt ist die Wurzel-Entität. [Eva04][MA07]

Serviceobjekte (engl.: Services)

Zusätzlich zur Datenmodelldefinition der Domäne mit seinen Entitäten, Wertobjekten und Aggregaten, besteht die Möglichkeit, Geschäftslogik in spezielle Serviceobjekte auszulagern. Dies ist vor allem dann sinnvoll, wenn beispielsweise eine Transaktion zwischen zwei Entitäten stattfinden soll. Dann ist nämlich nicht eindeutig ersichtlich, in welcher Entität die entsprechende Geschäftslogik implementiert sein muss. [Eva04][MA07]

Des Weiteren können Serviceobjekte dazu genutzt werden, um externe Schnittstellen in

Form von REST oder GraphQL zu implementieren. Generell gilt, dass Serviceobjekte ausschließlich den Code enthalten dürfen, welcher fachlich keiner Entität und keinem Wertobjekt zugeordnet werden kann. [Eva04][VKS14]

2.3 Web APIs

In diesem Kapitel werden zwei populäre Web-APIs in Form der herkömmlichen REST-, als auch der deutlich jüngeren GraphQL-Schnittstelle beschrieben.

2.3.1 REST

Thomas Bayer beschreibt 2002 den REpresentational State Transfer, kurz REST, als einen Architekturstil, um Web-Services umzusetzen. [Bay02]
Das Architekturkonzept von REST stammt jedoch von Roy Thomas Fielding aus dem Jahr 2000. Er beschäftigte sich in seiner Dissertation an der University of California mit Architekturstilen für Netzwerk-basierte Softwarearchitekturen. Dabei spielt REST als Schnittstellen-Technologie eine tragende Rolle um die Kommunikation zwischen Client(s) und Server(n) zu realisieren.

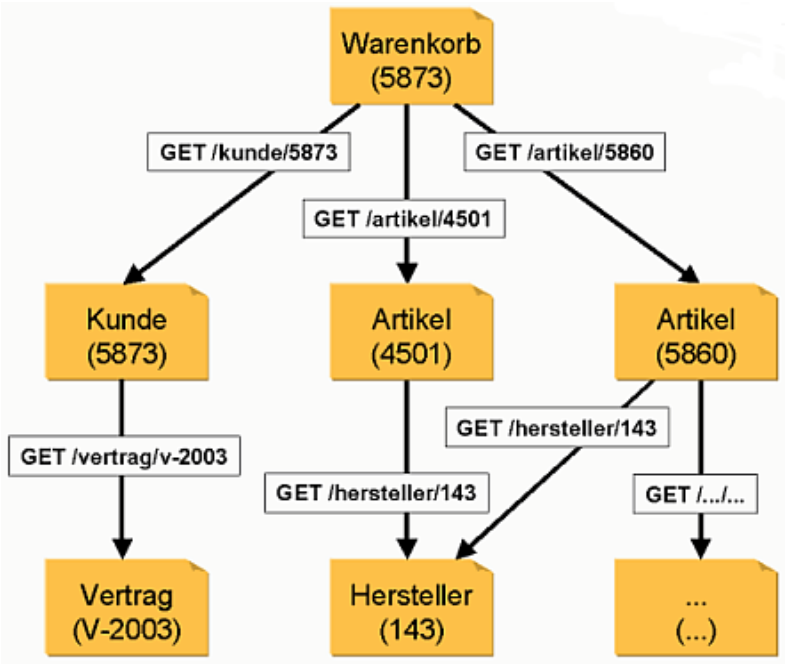

Abbildung 2.8: Navigation durch eine serverseitige Objektstruktur mittels REST über HTTP-GET-Anfragen [Bay02]

In Abbildung 2.8 ist eine beispielhafte Navigation ausgehend von einem Client durch die serverseitige Objektstruktur dargestellt. Jedes serverseitige Objekt stellt dabei einen eigenen REST-Endpunkt dar, der einfach über eine URL, z.B. mithilfe von HTTP-GET erreicht werden kann. Im dargestellten Onlineshop-Beispiel gibt es serverseitig genau fünf Objekttypen, nämlich Warenkorb, Kunde, Vertrag, Artikel und Hersteller. Will der Kunde mit Nummer 5873 nun als Client agieren und seinen Warenkorb aufrufen, so müssen im Hintergrund mehrere REST-Anfragen durchgeführt werden. Zuerst muss über den Endpunkt `GET /kunde/5873` das entsprechende Kundenobjekt vom Server holen. Anschließend geht es analog mit dem Vertrag Nummer V-2003 über den entsprechenden Endpunkt `GET /vertrag/v-2003` weiter. Analog dazu können die Artikel, welche im Warenkorb liegen und die zugehörigen Hersteller ebenfalls jeweils über einen eigenen Endpunkt geholt werden.
Folglich muss der Client im Beispiel aus Abbildung 2.8 mindestens sieben Mal Daten abfragen, bis er den Warenkorb, wie er auf Serverseite gespeichert ist, vollständig aufgebaut hat. Dies passiert jedoch in der Regel implizit im Hintergrund. [Bay02] [Fie00]

Listing 2.1 HTTP-GET-Request zur Abfrage des Warenkorbs [Bay02]

```
GET /warenkorb/5873
```

Listing 2.2 HTTP-Result zur Warenkorb-Abfrage aus Listing 2.1 [Bay02]

```
HTTP/1.1 200 OK Content-Type: text/xml
<?xml version="1.0"?>
<warenkorb xmlns:xlink="http://www.w3.org/1999/xlink">
<kunde xlink:href="http://shop.oio.de/kunde/5873">
      5873
</kunde>
<position nr="1" menge="5">
      <artikel xlink:href="http://shop.oio.de/artikel/4501" nr="4501">
            <beschreibung>
                  Dauerlutscher
            </beschreibung>
      </artikel>
</position>
<position nr="2" menge="2">
      <artikel xlink:href="http://shop.oio.de/artikel/5860" nr="5860">
            <beschreibung>
                  Earl Grey Tea
            </beschreibung>
      </artikel>
</position>
</warenkorb>
```

Listing 2.3 HTTP-PUT-Request zur Anlage eines neuen Artikels [Bay02]

```
PUT /artikel
<artikel>
      <beschreibung-kurz>
            Rooibusch Tee
      </beschreibung-kurz>
      <beschreibung>
            Feiner namibischer Rooibusch Tee
      </beschreibung>
      <preis>
            2,80
      </preis>
      <einheit>
            100g
      </einheit>
</artikel>
```

Listing 2.4 HTTP-Result zum angelegten Artikel aus Listing 2.3 [Bay02]

```
HTTP/1.1 201 OK
Content-Type: text/xml;
Content-Length: 30
http://shop.oio.de/artikel/6005
```

Listing 2.5 HTTP-POST-Request zur Bestellung von Artikel 961 [Bay02]

```
POST /warenkorb/5873
artikelnummer=961
```

Die Listings 2.1 bis 2.5 demonstrieren das grundlegende Potential von REST. Über den entsprechenden URL-Endpunkt wird in Listing 2.1 der Warenkorb für den Kunden mit der Nummer 5873 per HTTP-GET-Request abgefragt. Das HTTP-Result aus Listing 2.2 liefert in diesem Beispiel den kompletten Warenkorb mit allen enthaltenen Artikeln als XML-Struktur zurück. Denkbar wäre genauso eine JSON-Datenstruktur anstelle von XML.
REST bietet nicht nur die Möglichkeit, Daten abzufragen, sondern auch mittel PUT neue Datensätze anzulegen oder per POST vorhandene zu verändern. Dabei werden ebenfalls die entsprechenden REST-Endpunkte angesprochen. Somit wird zur Anlage eines neuen Artikels dessen XML-Struktur, wie in Listing 2.3 im HTTP-Body mitgeliefert. Der beispielhaft betrachtete Server liefert dann in Listing 2.4 die entsprechende Artikelnummer samt Endpunkt zurück. Zu guter Letzt wird der neu angelegte Artikel mit der Nummer 961 vom Benutzer in den Warenkorb gelegt, indem per HTTP-POST auf den Warenkorb-Endpunkt die Artikelnummer übergeben wird (siehe Listing 2.5). [Bay02]

2.3.2 GraphQL

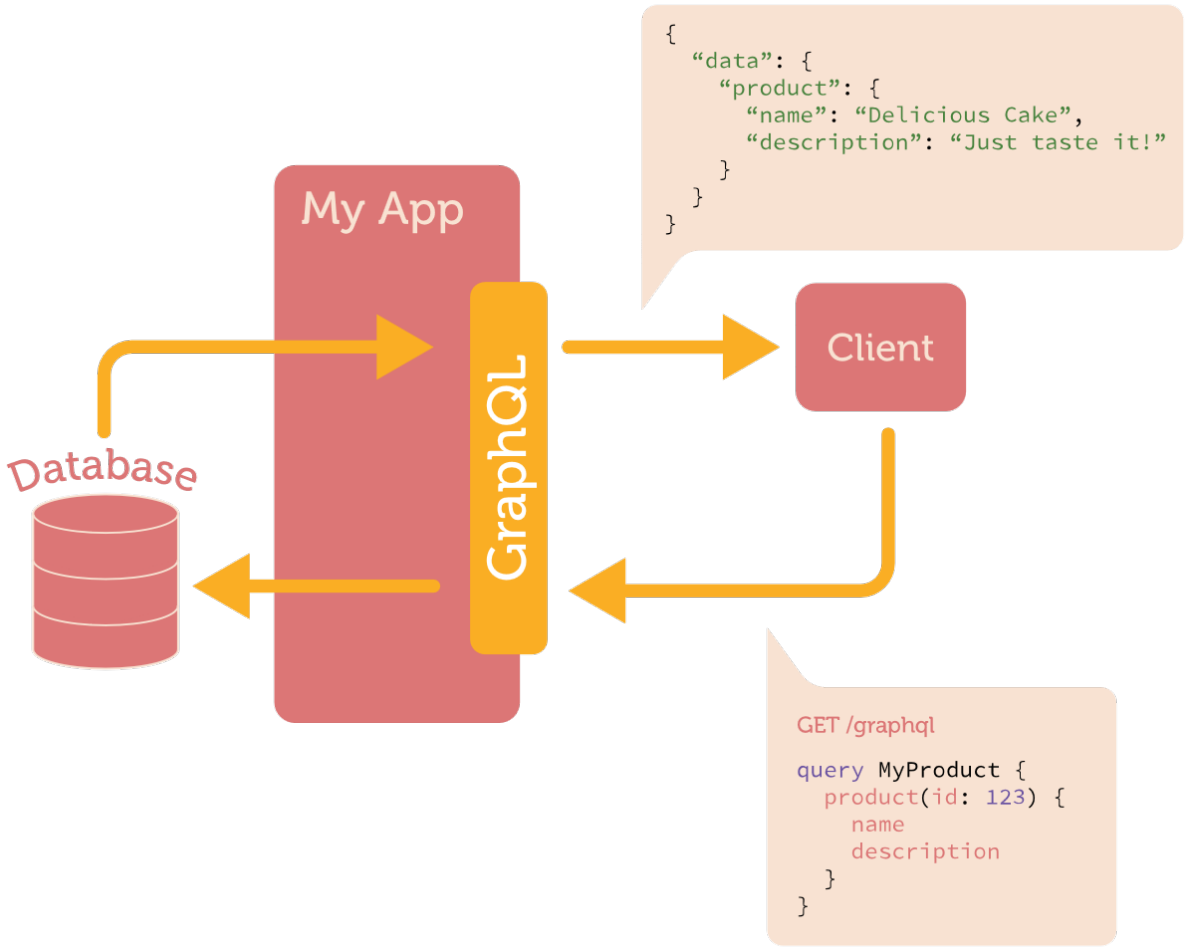

Abbildung 2.9: Beispielhafte GraphQL-Kommunikation zwischen Client und Server
Quelle: http://sangria-graphql.org/getting-started/

Im Gegensatz zum bereits im Jahr 2000 entwickelten Konzept von REST-Endpunkten, entwickelte Facebook im Jahre 2012 im Rahmen eines großen Refactorings eine neue API-Abfragesprache namens GraphQL[1].
Im Gegensatz zu REST gibt es in der Regel nur einen einzigen GraphQL-Endpunkt, über diesen im HTTP-Body die GraphQL-Anfragen in Form von Queries oder Mutations mitgegeben werden können. [Fac17]

[1] http://graphql.org/

Dieser Worklflow ist in Abbildung 2.9 dargestellt, indem die Datenbank über eine App-Komponente wegabstrahiert ist und ein Client die Daten nur noch mittels GraphQL Query im HTTP-Body an die App-Komponente abfragen kann.

Listing 2.6 Einfache GraphQL-Query [Fac17]

```
query {
      hero {
            name
            appearsIn
      }
}
```

In Listing 2.6 ist eine einfache GraphQL-Query abgedruckt. Angefragt werden alle „hero“-Objekte und jeweils sollen nur die Attribute „name“ und „appearsIn“ aufgelistet werden. Auch wenn ein „hero“ noch mehr Attribute hätte, werden ausschließlich die in der Query explizit definierten zurück gegeben (siehe Listing 2.7). Dies unterscheidet unter anderem GraphQL maßgeblich vom REST-Konzept, bei dem immer alle Attribute standardmäßig geliefert werden.

Listing 2.7 JSON-Rückgabe der einfachen GraphQL-Query aus Listing 2.6 [Fac17]

```
{
      "data": {
            "hero": {
                  "name": "R2-D2",
                  "appearsIn": [
                        "NEWHOPE",
                        "EMPIRE",
                        "JEDI"
                  ]
            }
      }
}
```

GraphQL trennt streng die Schemadefinition, auf Basis derer die API bereitgestellt wird, von der tatsächlichen Implementierung, wie die Daten geholt werden. Die tatsächliche Implementierung der Queries und Mutations auf Backend-Seite ist im Normalfall dem Entwickler selbst überlassen, was z.B. prinzipiell alle Möglichkeiten bezüglich Technologien oder Datenbanken auf Serverseite erlaubt. [Fac17]

Schema und Typen

Die mit Abstand wichtigste Rolle im GraphQL-Kontext spielen die Schemata und Typen.
Ein GraphQL-Schema besteht ganz grundsätzlich aus Objekttypen (engl.: object types) mit Feldern (engl.: fields). Jedes Feld hat wiederum einen Typ. [Fac17]

Listing 2.8 Beispiel eines GraphQL Object Type [Fac17]

```
type Character {
      name: String!
      appearsIn: [Episode]!
}
```

Betrachtet man den beispielhaften Objekttyp „Character“ aus Listing 2.8, so hat dieser per Definition genau zwei Felder, nämlich „name“ und „appearsIn“. Das Feld „name“ ist vom skalaren Typ String und darf aufgrund des Ausrufezeichens nicht leer sein. Das Feld „appearsIn“ dagegen ist eine Liste aus „Episode“-Objekten, die gegebenenfalls selbst wiederum Objekttypen mit Feldern sein könnten.
Zusätzlich zu Objekttypen existieren zwei spezielle Typen im GraphQL-Schema, nämlich Query und Mutation. [Fac17]

Listing 2.9 GraphQL Query und Mutation Type [Fac17]

```
schema {
      query: Query
      mutation: Mutation
}
```

Laut Spezifikation muss jedes GraphQL-Schema ein „query“ -Feld wie in Listing 2.9 besitzen. Dieses definiert den Einstiegspunkt für GraphQL-Abfragen, wie bereits in Listing 2.6 skizziert. Optional dazu kann ein „mutation“ -Feld analog als Einstiegspunkt existieren, um Datenänderungen im Sinne von Hinzufügen (engl.: add), Löschen (engl.: delete) oder Verändern (engl.: modify) auf den definierten Objekttypen und Feldern durchzuführen.
Wie bereits oben erwähnt, unterliegt die tatsächliche Server-seitige Implementierung alleine dem Entwickler. [Fac17]

2.4 NoSQL Datenbanken

Dieses Kapitel befasst sich mit NoSQL-Datenbanktechnologien und im speziellen der Multi-Modell-Datenbank ArangoDB[1]. NoSQL steht allgemein für „Not only SQL“[2] und damit im direkten Gegensatz zu herkömmlichen, relationalen Datenbanksystemen, mit einigen signifikanten Unterschieden und Einsatzbereichen.

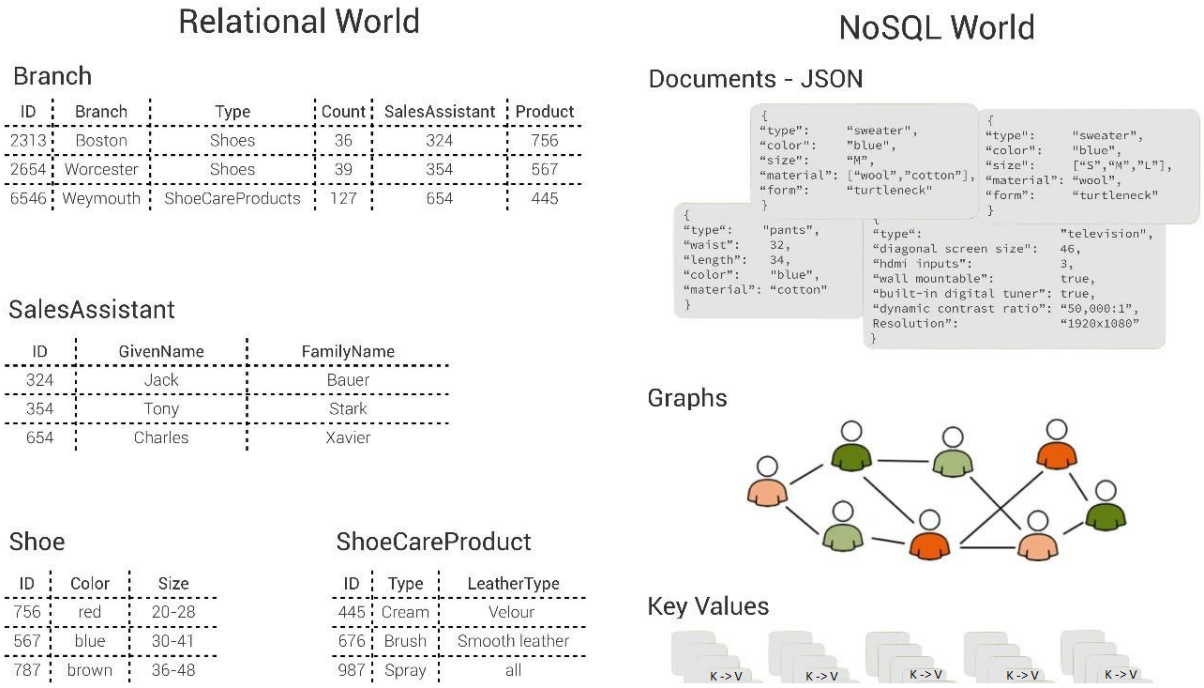

Abbildung 2.10: Relationale Datenmodelle im Vergleich zu NoSQL-Datenmodellen [Ara16]

Die ArangoDB GmbH hat die Unterschiede zwischen der relationalen, sowie der NoSQL-Datenbankwelt grafisch in Abbildung 2.10 zusammengefasst.

Relationale Datenbanksysteme speichern Daten üblicherweise in Tabellen. Je Datensatz wird eine Zeile in der Tabelle der zugehörigen Entität angelegt mit den Attributen als Spalten. Dies impliziert auch, dass ein Datensatz, welcher nicht alle vorhandenen Spalten befüllt, diese trotzdem als leeres Feld mit führt.

[1] https://www.arangodb.com/
[2] https://www.heise.de/ct/artikel/NoSQL-im-Ueberblick-1012483.html

Jeder Datensatz besitzt zudem einen Primärschlüssel in Form der ID, welcher tabellenübergreifend eindeutig sein muss und von anderen Datensätzen, in Form eines Fremdschlüssels, referenziert werden kann. [Ara16]

Im Gegensatz dazu existieren diverse NoSQL-Datenmodelle, angefangen mit strukturierten JSON-Dokumenten, über Graphen, bis hin zu Key-Value-Modellen. [Ara16] [BCAT14][BCAT14][sol17b]

2.4.1 ArangoDB

ArangoDB[1] ist eine Multi-Modell-Datenbank, welche vom jungen Unternehmen ArangoDB GmbH in Köln entwickelt wird. ArangoDB wird Jahr für Jahr populärer, wie im Datenbank-Ranking[2] der solid IT GmbH [sol17a] zu sehen ist. Das Ranking wird zwar immer noch von den erfahrenen, relationalen Datenbanksystemen wir Oracle, MySQL, MSSQL etc. dominiert, doch von April 2016 bis April 2017 arbeitete sich die ArangoDB immerhin von Platz 94 auf aktuell Platz 80 vor. Auch andere NoSQL-Datenbanken steigen aufgrund der immer stärker werdenden Web-Technologien weiter nach oben und nehmen einen bedeutenden Gegenpart zu herkömmlichen Systemen ein.

Abbildung 2.11: Logo des Datenbanksystems ArangoDB
Quelle: https://www.arangodb.com/

Die wohl wichtigste Eigenschaft von ArangoDB ist, dass sie eine native Multi-Modell-Datenbank darstellt. Die ArangoDB GmbH selbst definiert 2016 [Ara16] diese Eigenschaft als Kombination aus drei verschiedenen Datenmodellen. So können Daten entweder als Key-Value-Paare, Dokumente oder Graphen abgespeichert werden, allesamt verschiedene

[1] https://www.arangodb.com/
[2] https://db-engines.com/de/ranking

NoSQL-Technologien. Diese können über eine einheitliche, deklarative Abfragesprache erreicht werden, der es möglich ist, auch mehrere Datenmodelle gleichzeitig zu nutzen.
Abhängig vom Anwendungsfall können die drei Datenmodelloptionen entweder einzeln oder kombiniert eingesetzt werden. [Ara16]

Lucas Dohmen widmete sowohl seine Bachelorthesis 2012 [DKC12], als auch seine Masterarbeit 2014 [DEH14] - jeweils in Zusammenarbeit mit der ArangoDB GmbH - der Erforschung technischer Möglichkeiten des Datenbanksystems. Er beschäftigte sich sowohl mit effizienten Graph-Algorithmen innerhalb der ArangoDB, als auch der Implementierung eines deklarativen Frameworks zur Generierung von REST-basierten Web-APIs.
Zum Zeitpunkt der Forschungsarbeit 2012, konnte ArangoDB bei den Ressourcen noch mit der reinen Graph-Datenbank Neo4J[1] mithalten, doch die Performance lies damals noch Raum nach oben. Zusammen mit der umfangreichen Testsuite, die damals entwickelt wurde, konnten diese Probleme inzwischen aber behoben werden. [DKC12]
Im Jahr 2014 wurde ganz im Sinne des Domain-Driven Design ein Framework zur Definition von REST-APIs, mithilfe von Zustandsdiagrammen, entwickelt. Diese höhere Abstraktion der API-Definition führte vor allem zu einer deutlich einfacheren Kommunikation zwischen Entwicklern und fachlichen Domänenexperten. Auch gelang es, die definierten Modelle, inklusive Authentifizierung, automatisiert in eine lauffähige REST-API mit Anbindung ans Datenbank-Backend zu übersetzen. [DEH14]

2.5 Graphcool

In Abbildung 2.12 ist eine mögliche Datenmodelldefinition am Beispiel von Graphcool[2], einer noch relativ jungen Backend Plattform innerhalb der Cloud, dargestellt.
Graphcool bietet dem Domänenexperten über die „Console“ die Möglichkeit, ein einfaches Datenmodell einer Anwendung zu definieren und zur Laufzeit anzupassen. Ein Graphcool-Datenschema wird grundsätzlich spezifiziert durch sogenannte Modelle und Relationen. Hierbei darf der Modellbegriff jedoch nicht mit dem gesamten Datenmodell verwechselt werden. Ein Modell im Graphcool-Sinne entspricht mehr einer einzigen Entität aus dem Domain-Driven Design.

[1] https://neo4j.com/product/
[2] https://www.graph.cool/

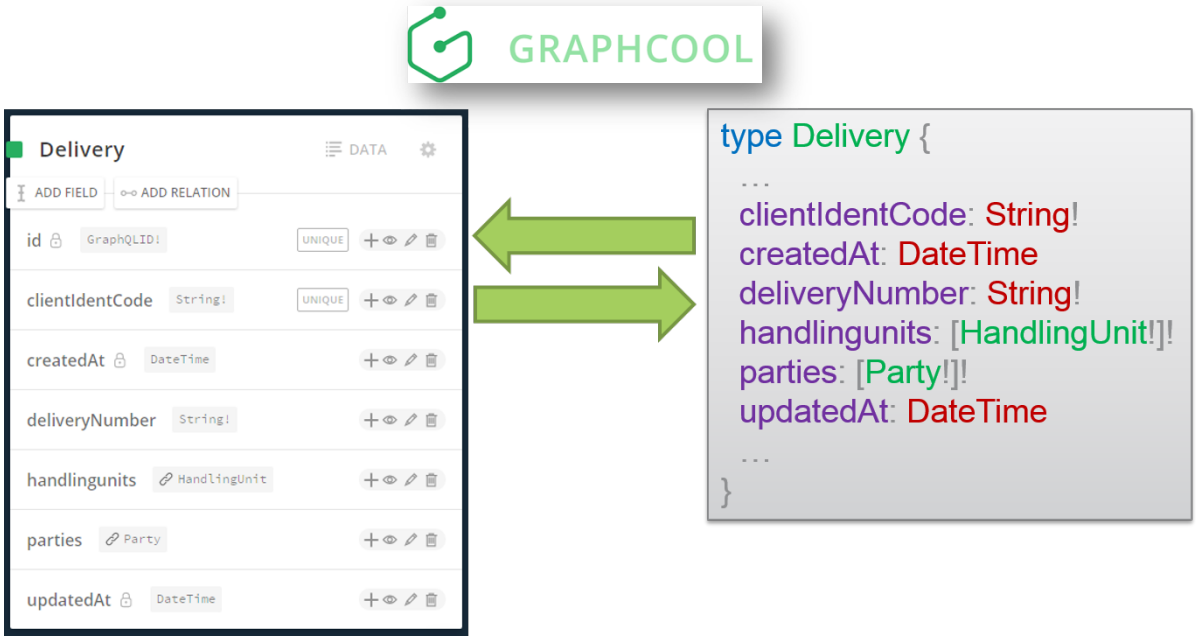

Abbildung 2.12: Definition eines (flexiblen) Datenmodells im GraphCool-Framework

Diese Modelle einzelner Entitäten können an der Oberfläche analog zu Abbildung 2.12 erstellt und über Relationen in Zusammenhang gesetzt werden. Sie bestehen zudem aus Feldern, wahlweise skalaren Typs (string, int, boolean, float, datetime, enum) oder einer Liste daraus. Zusätzlich kann für jedes Feld eine Restriktion angegeben werden, durch welche der tatsächliche Wert später nicht leer sein darf.
Im Gegensatz zu den frei definierbaren Eigenschaften und Feldern eines Modells, existieren zudem drei Standardfelder, deren Werte zur Laufzeit automatisch vom System verwaltet werden. Somit wird sowohl der Zeitpunkt der initialen Instanziierung eines Modells und der letzten Änderung dieser Instanz automatisch gespeichert. Um ein Modell im Sinne einer Entität eindeutig identifizieren zu können, wird zur Laufzeit automatisch eine eindeutige und unveränderliche ID vergeben.
Das benutzerdefinierte Datenmodell (siehe linke Seite aus Abbildung 2.12) lässt sich zusätzlich als JSON-Struktur (siehe rechte Seite aus Abbildung 2.12) textuell vom GraphCool-Framework exportieren und repräsentiert exakt die linke Seite. [GRA17a]

Das Graphcool Backend bietet über die reine Datenmodell-Funktionalität hinaus noch einige weitere Funktionen. So kann über das „Permission System“ beispielsweise der Zugriff auf Modelle für bestimmte Benutzer oder Benutzergruppen eingeschränkt werden. ALs logische

Konsequenz daraus werden zudem „Permanent Authentication Tokens“ zur Benutzerauthentifizierung unterstützt. [GRA17a]
Eine dritte nützliche Zusatzfunktion stellen die sogenannten „Mutation Callbacks“ dar. Sie erlauben es, auf bestimmte Queries oder Mutations Aktionen auszuführen. Dies ermöglicht in gewisser Weise die Implementierung von Geschäftslogik innerhalb des Graphcool-Backends. Mutation Callbacks bestehen aus einem „Trigger“ und einem „Handler“. Der Trigger legt fest, auf welche Aktionen reagiert werden soll und ein Handler stellt einen externen Datenverarbeitungsdienst in Form von Webtask[1] oder AWS Lambda[2] dar, der über eine URL nach Auslösen des Triggers angesprochen wird. Dadurch kann zum Beispiel das Ausführen von Geschäftslogik realisiert werden. [GRA17a][GRA17b]

Nach der Speicherung des Datenmodells innerhalb der Graphcool Console, wird daraus generisch eine GraphQL-API generiert, um wie üblich über Queries und Mutations auf die realen Daten zuzugreifen und sie gegebenenfalls zu verändern.

Nach den Modellierungsmöglichkeiten mit Graphcool, werden nun die Technologien zur Umsetzung dieser Plattform näher betrachtet. Die komplette Graphcool-Plattform wird aktuell auf dem Amazon Datencenter „eu-west-1“[3] in Irland gehostet und kann zum Zeitpunkt der Arbeit nicht in einem eigenen, lokalen Rechenzentrum betrieben werden.
Zur Datenhaltung wird ein Cluster aus Amazon Aurora Datenbanken[4] eingesetzt. Es handelt sich dabei um eine konventionelle, relationale Datenbank, die kompatibel zu den Open-Source-Datenbanken MySQL[5] und PostgreSQL[6] ist. Zu Speicherung hochgeladener Dateien wird dagegen der Objektspeicher Amazon S3[7] eingesetzt.
Wie auch schon zur Implementierung von Mutation Callbacks, wird die Geschäftslogik von Graphcool über einen AWS Lambda[8] Dienst erledigt.
Der Frontend Code für die Weboberfläche wird auf Netfly[9] gehostet.
Zusammenfassend lässt sich sagen, dass Graphcool für den Betrieb der Plattform nahezu

[1] https://webtask.io/
[2] https://aws.amazon.com/de/lambda/
[3] http://docs.aws.amazon.com/general/latest/gr/rande.html
[4] https://aws.amazon.com/de/rds/aurora/
[5] https://www.mysql.de/
[6] https://www.postgresql.org/
[7] https://aws.amazon.com/de/s3/
[8] http://docs.aws.amazon.com/lambda/latest/dg/welcome.html
[9] https://www.netlify.com/

vollständig auf externe Amazon Dienste setzt und den lokalen Betrieb momentan oder in absehbarer Zukunft nicht vorsieht. [GRA17c]

Die Oberfläche von Graphcool, im Voraus bereits als „Console“ beschrieben, basiert technisch gesehen auf React[1] (JavaScript-Bibliothek zur Implementierung einer Benutzeroberfläche (UI)) kombiniert mit Relay[2] (JavaScript-Framework zur Implementierung datengetriebener Anwendungen auf Basis von GraphQL). Der Quellcode wurde durchgängig mit Typescript[3], also typisiertem Javascript, geschrieben.
Das für diese Masterarbeit deutlich interessantere Backend stützt sich technologisch auf die Programmiersprache Scala[4], kombiniert mit Sangria[5], einer GraphQL-Implementierung auf Basis von Scala. [GRA17c]

2.6 Zusammenfassung

In diesem Kapitel wurden einige Grundlagen für die weitere Konzeption und Implementierung beschrieben. Das Domain-Driven Design bietet zahlreiche Konzepte und Patterns, um fachliche Domänen in Form eines Softwareprodukts zu entwickeln. Ein zentraler Bestandteil ist die ubiquitäre Sprache, die eine durch gemeinsames Begriffsverständnis eine leichte Kommunikation zwischen Fachexperten und Entwicklern unterstützt. Zur Definition solch einer einheitlichen Sprache kann das Konzept der Metamodellierung beitragen, welche diese durch eine höhere Abstraktionsebene definiert.
Um dem Anspruch einer Cloud-fähigen Komponente gerecht zu werden, wurden zusätzlich die Web-Schnittstellen REST und GraphQL betrachtet. Auch NoSQL-Datenbanken, als Gegensatz zu relationalen Datenbanken, werden im Umfeld der Cloud immer populärer.
Schlussendlich wurde der Fokus auf das Backend Graphcool gelegt, welches als Web-Komponente die Definition und den Zugriff auf ein einfaches Datenmodell unterstützt. Graphcool ist vor allem als eines der wenigen Backends mit GraphQL-Schnittstelle besonders interessant, um den Einsatz in der Praxis zu veranschaulichen.

[1] https://code.facebook.com/projects/176988925806765/react/
[2] https://facebook.github.io/relay/
[3] https://www.typescriptlang.org/
[4] https://www.scala-lang.org/index.html
[5] http://sangria-graphql.org/

3 Anforderungen

Dieses Kapitel beschreibt die Anforderungen der konkreten Konzeption und Umsetzung, welche im Unternehmenskontext über mehrere Iterationen während der Arbeit entwickelt und abgestimmt wurden.
Auf Grundlage der theoretischen Betrachtungen werden in den folgenden Kapiteln Konzepte zur praktischen Umsetzung entwickelt, welche wiederum in Form eines implementierten Prototypen abgebildet werden. Dieser initiale Prototyp dient im Anschluss dazu, die Betrachtungen anhand realer Szenarien zu evaluieren und mögliche Schwachstellen, vor allem im Sinne der Flexibilität, offenzulegen.

3.1 Initiale Fragestellungen

Betrachtet man das übergeordnete Thema der Arbeit, nämlich ein flexibles Datenmodell zu entwickeln, so stellen sich zu Beginn einige elementare Fragen.
Zunächst einmal muss es möglich sein, ein flexibles Datenmodell innerhalb eines komplexen Softwaresystems zu definieren oder beschreiben.
Aus dieser Erkenntnis heraus muss zum Beispiel ermittelt werden, wie ein Datenmodell überhaupt definiert wird, oder wie hoch die Flexibilität an bestimmten Stellen zur Laufzeit sein darf.

3.2 (Meta-) Modellierung

Einen maßgeblichen Bestandteil des Domain-Driven Design stellt, wie eingangs erwähnt, die einheitliche, ubiquitäre Sprache der fachlichen Domäne dar. Dadurch kann fachlich und technisch gleichermaßen über Datenmodelle mit gleichem Verständnis diskutiert werden. Das

Metamodell zur Definition dieser ubiquitären Sprache im speziellen Unternehmenskontext dieser Arbeit wird nachfolgend in Abbildung 3.2 mit Anlehnung an die Domain-Driven Design Abhängigkeiten aus Abbildung 2.7 entwickelt.

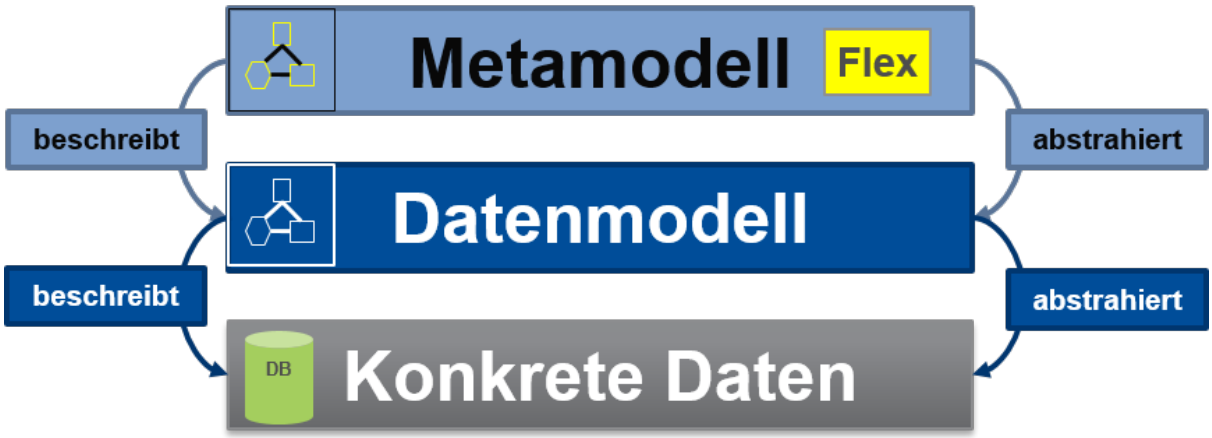

Abbildung 3.1: Zusammenhang zwischen konkreten Daten, Datenmodell und Metamodell

Der Zusammenhang zwischen den konkreten Daten und den abstrahierenden Modellen, vor allem in Bezug auf die geforderte flexible Datenmodell-Komponente, ist in Abbildung 3.1 dargestellt.
Das mittige Datenmodell abstrahiert und beschreibt somit die Struktur und Abhängigkeiten der darunter liegenden Daten in der Datenbank. Es spielt eine zentrale Rolle für diese Arbeit, gerade auch in Bezug auf die darin mögliche Flexibilität und Anpassung zur Laufzeit.
Wie ebenfalls zu Beginn erläutert, sollten auch Datenmodelle eine feste Struktur besitzen, welche auf einer Abstraktionsebene darüber in Form eines Metamodells definiert werden muss. Dieses Metamodell definiert gleichzeitig die mögliche Flexibilität innerhalb des Datenmodells zur Laufzeit und beschreibt analog die Bausteine und Abhängigkeiten, aus denen dieses besteht. Der Flexibilität konkreter Datensätze ist in der Regel keine Grenze gesetzt, da auf der Datenbank grundsätzlich beliebig Datensätze angelegt, gelöscht oder verändert werden können.

Wie eingangs erwähnt wird im Sinne des Domain-Driven Design ein Metamodell in Abbildung 3.2 als ubiquitäre Sprache zur späteren Definition der Datenmodelle im Unternehmen definiert. Die verwendeten Begrifflichkeiten wurden zusammen mit den AEB-seitigen Betreuern der Masterarbeit, Herr Günzel und Herr Niehus, als auch weiteren Teamkollegen

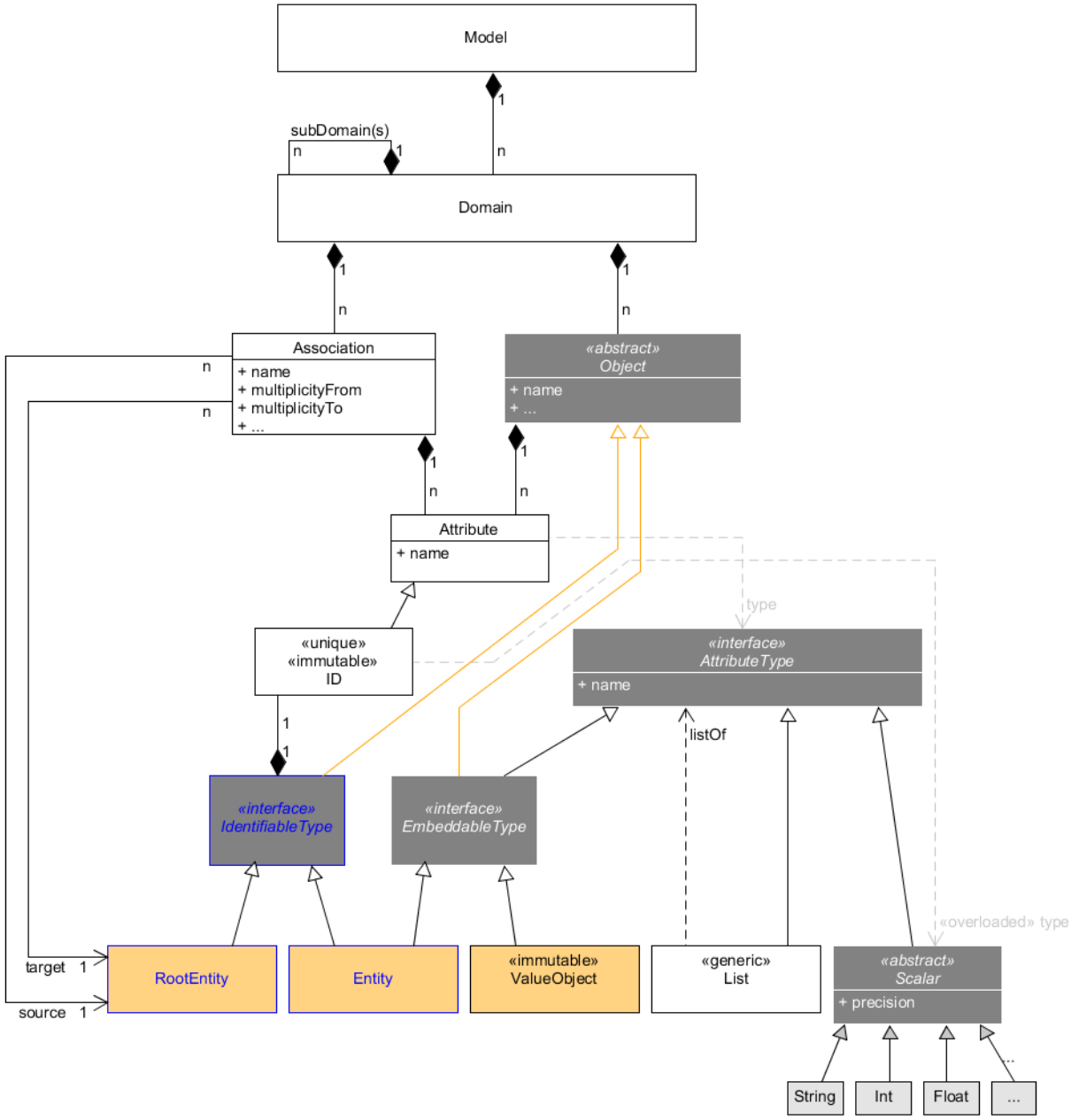

Abbildung 3.2: Das Metamodell zur Definition des Datenmodells.

abgestimmt und ausdiskutiert. Dieses elementare Modell soll einen einheitlichen Wortschatz zur Kommunikation von Datenmodellen im Unternehmen und eventuell auch mit Kunden definieren.

Ein Modell (engl. Model) im Sinne eines Datenmodells aus Abbildung 3.2 besteht initial einmal aus beliebig vielen fachlichen Domänen (engl. Domain). Diese können zudem in

Unterdomänen (engl. Subdomain) aufgeteilt werden.
Jede Domäne, bzw. Unterdomäne definiert sich aus Objekten (engl.: Object) und Assoziationen (Association), die Objekte miteinander verbinden können. Das Datenmodell einer Domäne kann vereinfacht auch als Domänenmodell bezeichnet werden.
Sowohl Objekte, als auch Assoziationen haben einen Namen, wobei eine Assoziation zusätzlich noch eine Multiplizität (z.B. 1:n, n:1, n:m etc.) besitzt. Beide können eine beliebige Anzahl an Attributen besitzen, die über einen lokal eindeutigen Namen identifiziert werden können.

Objekte und Attribute müssen im weiteren Verlauf mit Blick auf Entitäten, Aggregate oder Wertobjekte aus dem Domain-Driven Design noch granularer aufgeteilt werden. Objekte werden in sogenannte identifizierbare IdentifiableType und einbettbare EmbeddableType Typen unterschieden werden. Identifizierbare Objekttypen besitzen eine eindeutige, unveränderliche ID und können als RootEntity oder Entity im Daten-/Domänenmodell instanziiert werden. Eine Entität unterscheidet sich von der RootEntity genau dadurch, dass sie gleichzeitig als Attribut eingebettet werden kann, d.h. dass die ID auch nur lokal im eingebetteten Kontext Gültigkeit hat. Der dritte und letzte Objekttyp ist ganz nach dem Domain-Driven Design das ValueObject, welches eingebettet werden kann und zur Laufzeit unveränderlich ist, d.h. nur einmal angelegt oder wieder gelöscht werden kann.
Löst man analog dazu ein Attribut nach Typen auf, so fällt schnell auf, dass die Objekte, welche zum EmbeddableType gehören, aufgrund ihrer Eigenschaft, eingeschachtelt werden zu können, gleichzeitig ein Attributtyp (engl.: AttributeType) sein müssen. Wie weiter oben bereits festgestellt wurde, sind allein Attribute Bestandteil von Objekten und somit die einzige Möglichkeit, andere Objekte einzuschachteln.
Der wohl einfachste und gleichzeitig bekannteste Attributtyp ist der Scalar, welcher alle bekannten skalaren Datentypen, wie String, Int, Float etc. umfasst. Ebenfalls häufig notwendig in der Datenmodellierung sind Attributlisten, im Metamodell als generische List spezifiziert. Jede instanziierte Liste hat als inneren Typ der Elemente genau einen Attributtyp um keine Listen mit unterschiedlichen Objekten zu erlauben.

Zu guter letzt sind Assoziationen nur auf oberster Ebene und nicht zwischen eingeschachtelten Objekten möglich. Folglich kann eine Assoziation nur zwischen RootEntity und RootEntity, eines als source und eines als target, existieren

Zusammenfassend kann zum Metamodell aus Abbildung 3.2 gesagt werden, dass es gän-

gige Datenmodelle, wie sie in UML definiert werden können, weitestgehend unterstützt, als auch eingebettete Typen.
Als Anwendungsbeispiel könnte zum Beispiel eine Sendung als RootEntity modelliert werden. Diese hat diverse Felder, wie eine Sendungsnummer oder einen eingeschachtelten Empfänger, welcher als ValueObject mit weiteren Attributen wie Name, Adresse etc. modelliert werden könnte, da er üblicherweise in mehreren Sendungen auftauchen wird.

3.2.1 Flexibilität im Datenmodell

Eine bedeutende Anforderung an diese Arbeit stellt die Flexibilität des Datenmodells zur Laufzeit dar. Betrachtet man die Zusammenhänge zwischen den konkreten Daten, dem Datenmodell und dem Metamodell aus Abbildung 3.1, so wird die Flexibilität des Datenmodells im Metamodell definiert.
Die Flexibilität der konkreten Daten muss nicht analog dazu im Datenmodell definiert werden, da die Daten in der Regel beliebig direkt in der Datenbank angepasst werden könnten, wenn man explizit am Datenmodell vorbei navigiert. Dieser Fall erübrigt sich im Folgenden, da das Datenmodell die zentrale Rolle spielen soll und dort die Anpassungen zur Laufzeit durchgeführt werden.

Die zentrale Rolle spielen die Objekte und Abhängigkeiten aus Abbildung 3.2, wobei im Sinne möglichst großer Flexibilität initial keine Einschränkungen definiert werden. Dies bedeutet, dass grundsätzlich alle Objekte, Assoziation oder Attribute im Datenmodell zur Laufzeit angelegt, gelöscht oder modifiziert werden können. Dies beinhaltet im ersten Schritt auch die Anpassung von Attributtypen.

Sicherlich führt diese Freiheit später im Prototypen zu einigen Problemen. Die Evaluation am Schluss der Arbeit spielt darum eine entscheidende Rolle, um für möglichst viele, reale Szenarien zu prüfen, inwieweit das Datenmodell an bestimmten Stellen flexibel sein darf oder auch nicht. Interessante Fragen sind zum Beispiel, was passiert, wenn ein neues Pflichtfeld aufgenommen wird oder wenn ein vorhandenes Feld gelöscht wird.
Die Evaluation versucht daher, möglichst detailliert festzustellen, wo die Grenzen der Flexibilität im Datenmodell liegen.

3.3 Zusammenfassung

Die Analyse der Anforderungen für die AEB, kombiniert mit den Konzepten der Metamodellierung und des Domain-Driven Design führt zu einem Metamodell. Dieses beschreibt die Bestandteile und Abhängigkeiten der realen Datenmodelle und definiert die Anforderungen an den Prototypen, welcher in den folgenden Kapiteln konzipiert und implementiert wird.

Im ersten Schritt sollen keine Einschränkungen der Flexibilität im Datenmodell vorgenommen werden, um möglichst viele reale Szenarien in der Evaluation ausführlich prüfen zu können und dann zu entscheiden, wo eventuell die Grenzen liegen.

Graphcool ist auf der einen Seite ein vielversprechender Ansatz für eine Datenmodell-Komponente, bietet jedoch nur einen Teil der Funktionalität aus dem Metamodell. Die größte Einschränkungen sind die fehlende Schachtelung von Objekten, was zu einem bestimmten Teil an der relationalen Persistenzlösung liegt, als auch der ausschließliche Betrieb in der Amazon Cloud. Viele Firmen lehnen in Bezug auf die Geheimhaltung ihrer Daten grundsätzlich externe Cloud-Lösungen ab und möchten unter Umständen ihren eigenen Server lokal betreiben.
Zudem ist nur der Code für die UI des Backends in Form der Console[1] als Open-Source verfügbar, nicht jedoch der des Backends selbst. Somit sind firmeneigene Anpassungen und Erweiterungen sicherlich schwer möglich.
Bezüglich der Flexibilität unterstützt Graphcool momentan vor allem additive Änderungen zur Laufzeit, indem eine neue Entität mit Attributen definiert werden kann. Auch das Löschen von Attributen oder Entitäten stellt im Datenmodell kein Problem dar. Die Auswirkungen auf Client-Systeme mit GraphQL-Anbindung muss dabei gesondert betrachtet werden. Während das Umbenennen von Feldern noch möglich ist, führt eine Typänderung in der Regel zum Verlust der gesamten Attributwerte. Ebenso werden die Attributwerte beim Löschen des Attributs im Datenmodell direkt entfernt. Wird das Attribut danach identisch wieder angelegt, sind immer alle Altdaten gelöscht. Trotz der vereinzelten Probleme, ist es möglich, Datenmodelländerungen zur Laufzeit durchzuführen. Ebenfalls wird immer sofort die generische GraphQL-Schnittstelle automatisch aktualisiert.

[1] https://github.com/graphcool/console

4 Konzept zur Umsetzung

Nach der Analyse der Anforderungen im vorherigen Kapitel 3, werden nun Visionen und daraus Konzepte entwickelt, wie ein flexibel anzupassendes Datenmodell in Form eines Prototypen umgesetzt werden kann. Dabei spielt die Vision der AEB, wie in Zukunft Software im Unternehmen entwickelt werden soll, eine zentrale Bedeutung.

4.1 Model Manager

Die vorangehenden Kapitel und das Thema der Arbeit, machen mehr als deutlich, dass das Datenmodell mit seinen Flexibilitätsaspekten die zentrale Rolle für die zu entwickelnden Konzepte spielen wird.
Angefangen mit der Vision der AEB von zukünftiger, agiler Softwareentwicklung, wird ein Architekturkonzept eines modellzentrischen Ansatzes entwickelt und gegenüber dem Konzept der objektorientierten Datenbanken abgegrenzt.

4.1.1 Vision

Im Unternehmensprojekt namens nEXt soll der Aufbau einer zukunftsweisenden Software-Geschäftspross-Schicht, die die bestehenden und zukünftige AEB Services integriert und das Versprechen einer durchgängigen Suite für Logistik, Außenwirtschaft und Riskmanagement einlöst, erforscht, implementiert und evaluiert werden. Wenn diese neue Plattform existiert, sollen schrittweise zuerst die Neusysteme darauf aufbauen und im Laufe der kommenden Jahre die alten Plattformen abgeschafft oder auch mit nEXt kombiniert werden.

Eines der Ziele für die neue Plattform, ist die deutlich stärkere Fokussierung auf eine Implementierung von Geschäftsprozessen mit einer leichten Integration bestehender Services

innerhalb der Cloud. Der Trend geht dabei weg von einem Software-Monolith hin zu einer leichtgewichtigen Architektur, in der sich vieles für den Kunden sehr effizient anpassen lässt. Je Prozesschritt soll es zukünftig eine kleine Applikation geben, die im Idealfall aus einem „App-Store“ einfach integriert werden kann.
Die neue nEXt-Architektur soll zudem eine agile, iterative Entwicklung unterstützen, in der es z.B. einfach möglich ist, mit dem Kunden einen lauffähigen Prototypen zusammen zu klicken. Sie soll zudem mit einem modularen Aufbau offen für Erweiterungen oder Weiterentwicklungen sein und auf moderne Technologien setzen. Kombiniert mit agiler Softwareentwicklung, entsteht so eine langfristige und nachhaltige Plattform für prozessorientierte Geschäftsanwendungen. (Quelle: AEB GmbH)

Aus dieser Vision der AEB heraus, ergeben sich mehrere Aufgabenbereiche. Zum einen die App-Entwicklung auf Client-Seite, kombiniert mit modernen UI-Technologien. Zusätzlich zu dieser Frontend-Entwicklung ist im Sinne einer Cloud-Architektur immer ein Backend nötig, um auf Daten, oder unter Umständen auch Geschäftslogik zugreifen zu können.

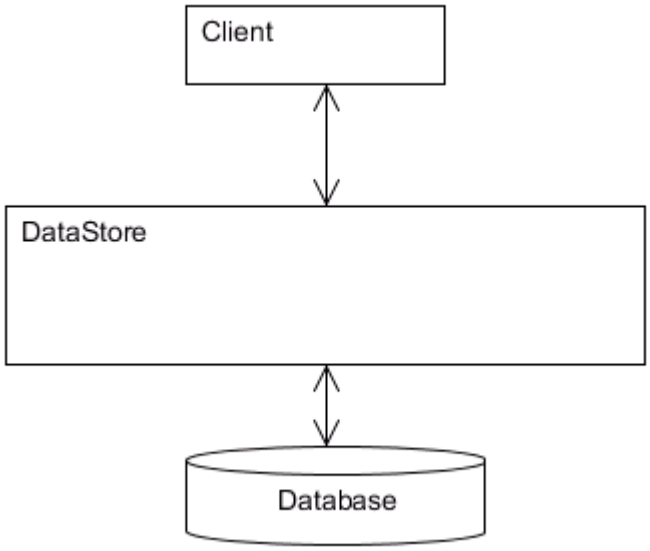

Abbildung 4.1: Grundlegend geplante Architektur - Quelle: AEB GmbH

Abbildung 4.1 zeigt das grundsätzlich angestrebte Architekturkonzept, um von Client-Anwendungen auf Daten zuzugreifen. Ziel ist es, die Datenbank für die Clients über eine DataStore-Komponente zu abstrahieren. Diese soll die Kommunikation, sowohl mit der Daten-

bank, als auch mit den Apps auf Client-Seite regeln. Ein Client hat nie die Möglichkeit, die Datenbank direkt und ohne die Zwischenkomponente anzusprechen.
Der große Vorteil dieses Architekturkonzepts ist, dass die verwendete Datenbanktechnologie absolut keine Rolle für die Implementierung der Clients spielt. Die einzig abhängige Komponente ist die Schnittstelle von der Datenbank zur DataStore-Komponente, die für jede Datenbanktechnologie entsprechend angepasst werden muss. Die Schnittstelle zwischen Client und DataStore wird aber zu keiner Zeit beeinflusst.
Ein weiterer Hintergund für diesen Wunsch ist die Tatsache, dass Datenmodelle aktuell statisch zwischen Anwendung und Datenbank übereinstimmen müssen. Das hat wiederum zur Folge, dass eine Änderung am Datenmodell einer Anwendung (z.B. die Hinzunahme eines Feldes x zur Entität y) immer einen Abgleich mit der Datenbank nach sich zieht, was nur im heruntergefahrenen System funktioniert.

4.1.2 Architekturkonzept

In Abbildung 4.2 ist nochmals das grundlegende Problem dargestellt. Innerhalb der Cloud-, bzw. im speziellen der nEXt-Landschaft, möchten mehrere Client-Apps auf Daten aus der Datenbank zugreifen, um Geschäftslogik auszuführen.

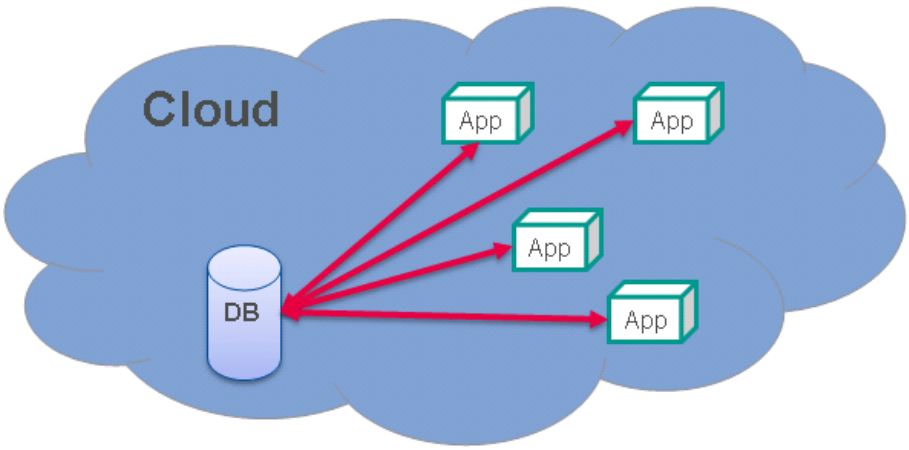

Abbildung 4.2: Stark vereinfachter Zugriff auf Daten innerhalb der Cloud

Wie bereits in Abbildung 4.1 dargestellt und in der Vision beschrieben, soll die Datenbank für die Apps in Form einer extra Backend-Komponente abstrahiert werden. Das flexible Datenmodell soll darin die Hauptrolle spielen und darum auch die Komponente im Folgenden sprechender

als Model Manager bezeichnet werden, um diesen Anspruch zu unterstreichen.
Der Model Manager soll als einzige Komponente die Oberhand über das Datenmodell der fachlichen Domäne erhalten. Er soll Daten-Anfragen über eine Schnittstelle entgegen nehmen können und sie passend über eine zweite Schnittstelle an die Datenbank weiterleiten.

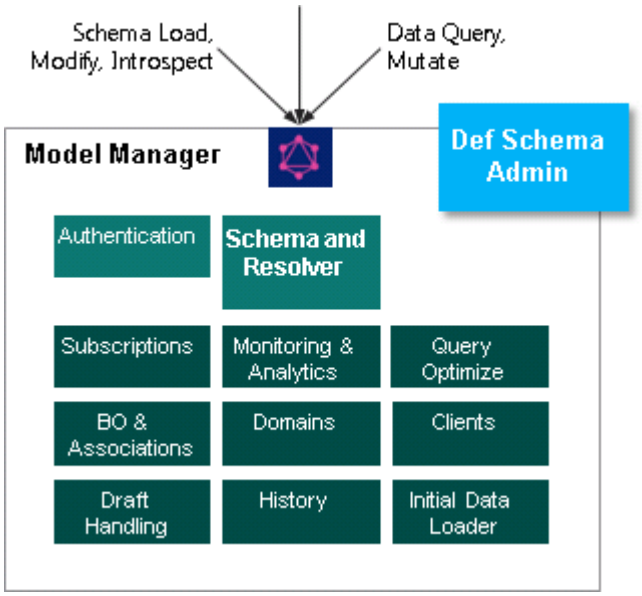

Abbildung 4.3: Bestandteile und Aufgaben eines Model Managers (Quelle: AEB GmbH)

Der Model Manager soll als eigenständige Komponente innerhalb der Domäne existieren und grundsätzlich Datenabfragen über das Datenmodell unterstützen. Eine weitere wichtige Aufgabe stellt die Möglichkeit dar, das Modell zur Laufzeit anzupassen.
Die Abbildung 4.3 fasst die einzelnen Bestandteile eines Model Managers im Sinne der AEB zusammen. Dabei fällt auf, dass die reine Modellverwaltung zusätzliche Fähigkeiten impliziert, dazu gleich mehr.
Die einzige Kommunikationsschnittstelle zum Model Manager definiert die API im oberen Bereich. Sie ermöglicht es ein Datenmodell, bzw. Schema zu laden, zu bearbeiten oder per Introspektion im Sinne von GraphQL abzufragen. Daten können ebenfalls über diese Schnittstelle per Query abgefragt und per Mutation modifiziert werden.
Die elementarste Komponente im Model Manager stellt Schema and Resolver dar. Sie hat die Aufgabe, das Datenmodell alleine zu verwalten und Datenanfragen der obigen Schnittstelle automatisiert an die Datenbank aufzulösen. Authentication spielt vor allem später eine wichtige Rolle, wenn z.B. mehrere Clients auf dem gleichen Model Manager arbeiten, jedoch unterschiedliche Zugriffsberechtigungen auf Daten besitzen.

Spätere Zusatzfunktionen des Model Managers könnten zudem Subscriptions sein, d.h. bestimmte Clients werden bei bestimmten Datenänderungen benachrichtigt. Das Auflösen der Anfragen impliziert direkt die Query Optimize Komponente, um die Daten möglichst effizient aus der Datenbank abzufragen. Auch beispielsweise eine History für Daten- und Modelländerungen kann in Zukunft notwendiger Bestandteil des Model Managers werden.

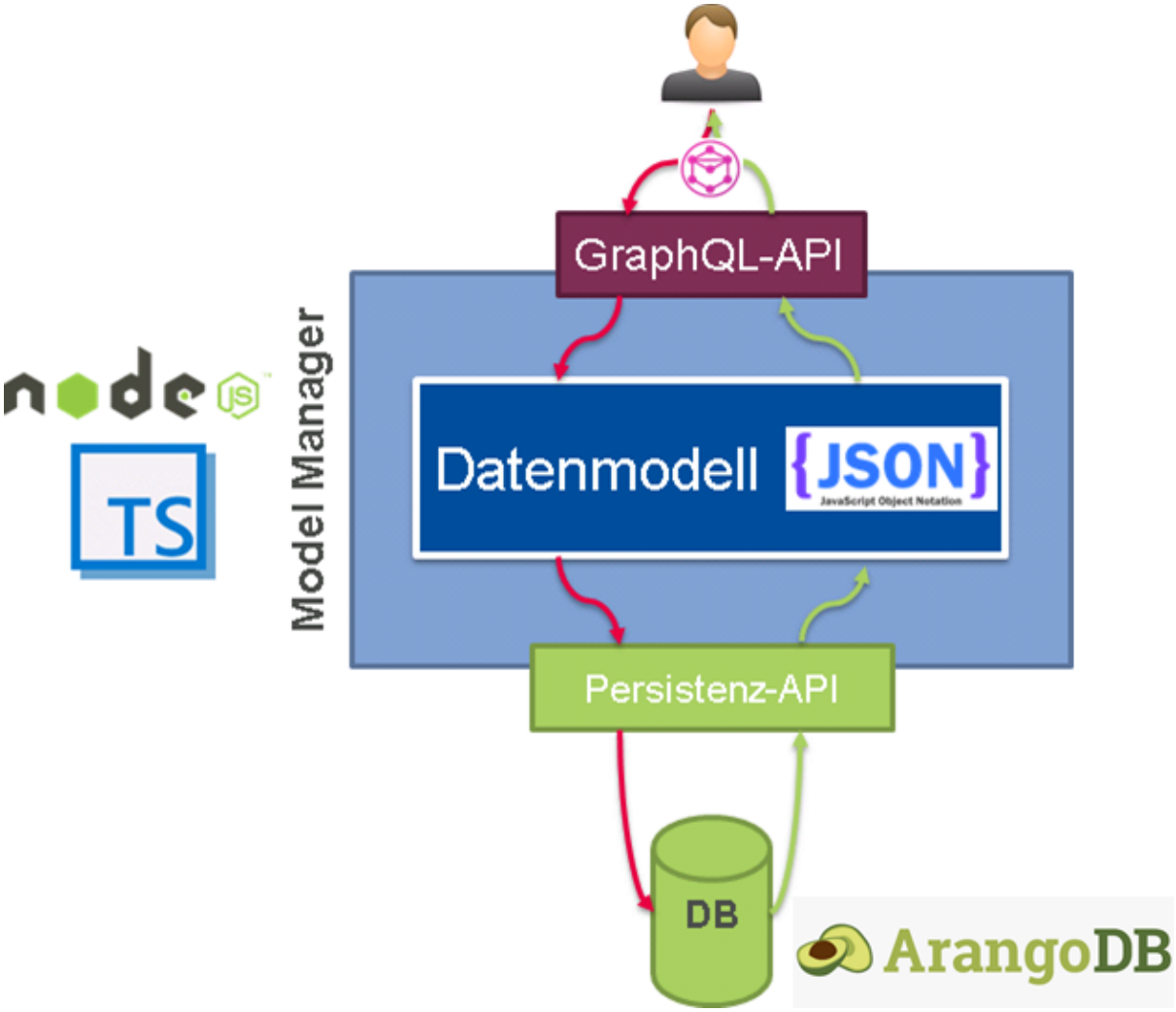

Abbildung 4.4: Architektur des ersten Model Manager Prototypen

Das Datenmodell der fachlichen Domäne wird nach dem Metamodell aus Abbildung 3.2 modelliert, bzw. stellt eine konkrete Instanz dieses Metamodells dar. Mit dem Blick zurück auf das Domain-Driven Design, sollte es möglich sein, dass durch die ubiquitäre Sprache sowohl technische Experten, als auch fachliche Domänenexperten ein Datenmodell einer Domäne definieren und anpassen können.

Einfache Operationen, wie die Hinzunahme eines neuen Feldes innerhalb einer beispielhaften RootEntity zur Laufzeit, muss in Zukunft keine Entwicklerarbeit mehr sein, sondern könnte direkt von den fachlichen Domänenexperten, welche sowieso im ständigen Austausch mit dem Kunden stehen, durchgeführt werden. Dies kann am ehesten dadurch gewährleistet werden, dass der Model Manager eine Möglichkeit bietet, das Datenmodell strukturiert und konform zum Metamodell zu verändern. Hierfür bietet sich sowohl die eXtensible Markup Language, kurz XML[1], als auch die JavaScript Object Notation, kurz JSON[2] an. Datenmodelle mit JSON lassen sich deutlich schlanker definieren, was die Lesbarkeit etwas verbessert. Darum werden die Datenmodelle im ersten Entwurf des Model Managers als JSON-Strukturen definiert.
In Abbildung 4.4 ist die Architektur des im weiteren Verlauf implementierten, ersten Prototypen eines Model Managers zu sehen. Diese wurde selbständig auf Basis der Anforderungen und Visionen der AEB im Rahmen der Arbeit entwickelt und mit Teamkollegen abgestimmt. Der Model Manager als Komponente beherbergt das Datenmodell in Form der beschriebenen JSON-Struktur und stellt für den Benutzer in Form von z.B. Client-Apps, eine GraphQL-Schnittstelle bereit. GraphQL wurde als API-Abfragesprache anstelle einer REST-Architektur gewählt, da mit einer einzigen GQL-Abfrage eine komplette, benutzerdefinierte Datenstruktur, exakt wie sie im Client benötigt wird, geholt werden kann. Auch die Fähigkeit, dass jede GraphQL-API einem GraphQL-Schema unterliegt kommt dem Model Manager Ansatz zugute. Zur Laufzeit kann das selbst definierte JSON-Datenmodell somit generisch in ein GraphQL-Schema umgewandelt und dieses wiederum in Form einer GraphQL-API nach außen veröffentlicht werden. Wie auch schon in Abbildung 4.3 erkannt, müssen die GQL-Anfragen der obigen API schlussendlich gegenüber der Datenbank aufgelöst werden. Als Persistenzlösung wurde dabei die bereits beschriebene Multi-Modell-Datenbank ArangoDB gewählt. Sie ermöglicht es durch den klassischen Dokument-orientierten Speicher, die geschachtelten Objekte einfach als JSON-Strukturen abzulegen. Somit wäre jede RootEntity, wie sie im Metamodell definiert wurde, ein eigenes Dokument innerhalb der entsprechenden „Collection“. Betrachtet man zum Beispiel die RootEntity Sendung, so würde eine Instanz davon in der „SendungCollection“ gespeichert werden. Alle anderen Objekttypen aus dem Metamodell, wie Entity und ValueObjekt, würden nie als eigenständiges Dokument, sondern immer nur eingeschachtelt in einer RootEntity gespeichert werden. Ein weiterer Grund für die Wahl der ArangoDB ist die Möglichkeit, Dokumente über Kanten zu verbinden und somit Assoziationen ausgelagert in je einer eigenen „EdgeCollection“ zu speichern, ohne die verknüpften Dokumente anpassen zu müssen. Somit

[1] https://www.w3schools.com/xml/xml_whatis.asp
[2] http://www.json.org/json-de.html

wird der Attributname für den Zugriff auf die Assoziation innerhalb das Source- und Target-RootEntity auch genau in dieser definiert. Zur Laufzeit werden die entsprechenden, extern definierten Assoziationsfelder, in den RootEntities generisch im GraphQL-Schema angelegt.
Die Implementierung des Model Managers wird komplett in Typescript[1], also typisiertem Javascript geschrieben. Der Code soll auf einem NodeJS[2]-Server mit der ts-node[3]-Erweiterung laufen, welche automatisiert in Echtzeit die Konvertierung von Typescript auf ausführbares JavaScript im Arbeitsspeicher vornimmt. NodeJS ist eine Plattform-unabhängige Javascript-Laufzeitumgebung und lässt sich somit identisch auf Linux/Unix, Windows oder Mac OS betreiben.

4.1.3 Unterschiede zu objektorientierten Datenbanken

Dieses Kapitel befasst sich mit den konzeptionellen Unterschieden objektorientierter Datenbanken, wie z.B. db4o[4], ObjectStore[5], ObjectDB[6] zum bereits vorgestellten Konzept eines Model Managers.

Die sogenannten „object-oriented database management system(s)“ (OODBMS) werden als Datnbanksysteme definiert, welche Daten direkt als Objekte speichern können. [MSOP86][Mon16][Tec17]
Diese Eigenschaft deckt sich einigermaßen mit der des Model Managers, wenn man eine Sendung beispielsweise als JSON-Dokument definiert. Dieses Dokument könnte in einer objektorientierten Datenbank als Objekt gespeichert werden. Trotzdem haben sich laut aktuellem Datenbank-Ranking der solid IT GmbH [sol17a] die Objektdatenbanken nie wirklich durchgesetzt. Die Datenbank db40 ist als aktuell oberster Vertreter dieser Technologie gerade einmal hinter Platz 100 zu finden.

Konzeptionell unterscheiden sich die objektorientierten Datenbanken deutlich von der Idee des vorgestellten Model Managers. Üblicherweise spart man sich als Entwickler mit OODBMS das Mapping zwischen Daten aus der Datenbank und Klassen/Objekten in der darüber liegenden

[1] https://www.typescriptlang.org/
[2] https://nodejs.org/de/
[3] https://github.com/TypeStrong/ts-node
[4] http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/Db4o
[5] ObjectStore
[6] http://www.objectdb.com/

Software. Dies wird üblicherweise dadurch erreicht, siehe ObjectDB [Obj10], dass die objektorientierte Klassendefinition das Datenmodell definiert. Dieses wird also von Entwicklern direkt im Code spezifiziert. Anders als beim entkoppelten Model Manager Ansatz, ist es somit einem fachlichen Domänenexperten nicht möglich, selbst das Datenmodell zu erweitern, ohne dazu einen Entwickler zu benötigen. Des Weiteren würde je nach Architektur die Hoheit über das Datenmodell im nEXt-System auch nicht mehr zentral beim Model Manager liegen, sondern eventuell verstreut über die Client-Apps, da diese jeweils auf einer eigenen Objektdatenbank, ohne eine abstrahierende Komponente, arbeiten würden.

Die grundsätzlichen Eigenschaften der Datenmodell-Definition grenzen den Model Manager und objektorientierte Datanbanksysteme deutlich voneinander ab, sodass OODBMS nicht zur Vision der definierenden Domänenexperten passen.

5 Implementierung eines Prototypen

Dieses Kapitel befasst sich mit der Implementierung der erarbeiteten Konzepte und Architekturen des Model Managers. Bei der Implementierung des ersten Model Manager Prototypen, wurde ich von meinem Entwicklerkollegen Jan Melcher in der AEB tatkräftig unterstützt. Anders wäre in der relativ kurzen Zeit nur deutlich weniger Funktionalität möglich gewesen.

Um das Datenmodell in Form einer JSON-Notation zu definieren, muss zu Beginn festgelegt werden, wie diese Notation aussehen soll. Beispielhaft wird dies anhand der Modellierung des Diagramms in Abbildung 5.1 erläutert.

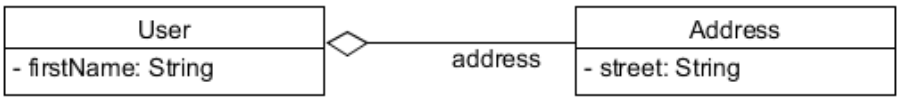

Abbildung 5.1: Beispielhaftes Datenmodell

Definiert wird ein Objekt „User“ mit einem Attribut „firstName“ vom Typ string. Dieses soll zudem unveränderlich und verpflichtend sein (nicht in Abbildung 5.1 eingezeichnet). Beim zweiten Attribut handelt es sich um die Adresse des Users, die vereinfacht nur ein einziges Attribut namens „street“ vom Typ String besitzt.
Die Repräsentation des Datenmodells aus Abbildung 5.1 als mögliche JSON-Definition für die Model Manager Komponente ist in Listing 5.1 abgedruckt. Mit Blick zurück auf das Metamodell aus Abbildung 3.2, wird man einen User üblicherweise als RootEntity definieren. Die Adresse hingegen könnte als eingeschachteltes Attribut im User definiert werden. Dafür wird beispielhaft das unveränderliche ValueObject gewählt, welches die Straße als Attribut mit führt.
Betrachtet man die JSON-Definition aus Listing 5.1, so existiert je Objekttyp aus dem Metamodell ein eigener Root-Knoten. Jeweils darunter werden konkrete Instanzen dieses Objekttyps mit ihren Attributen definiert. Jedes Attribut hat dabei einen „type“-Knoten, um den Attributtyp zu

spezifizieren. Analog zum Metamodell kann dieser ein skalarer Typ sein, ein eingeschachteltes Objekt oder auch eine Liste aus Attributtypen.

Listing 5.1 Einfaches Datenmodell als JSON-Definition

```
{
      "rootEntities": [
            {
                  "name": "User",
                  "attributes": {
                        "firstName": {
                              "type": "string",
                              "required": true,
                              "immutable": true
                        }
                        "address": {
                              "type": {
                                    "embedded": "Address"
                              }
                        }
                  }
            }
      ]
      "valueObjects": [
            {
                  "name": "Address",
                  "attributes": {
                        "street": {
                              "type": "string"
                        }
                  }
            }
      ]
}
```

Das sogenannte JSON Schema[1] wird in Kombination dazu eingesetzt, um die Struktur von JSON-Datenmodellen zu definieren und diese anhand der abstrakten Definition zu validieren. Das JSON Schema kann als Repräsentation des zuvor definierten Metamodells verstanden werden, sodass der Model Manager ausschließlich dazu konforme Datenmodell erlaubt.

[1] http://json-schema.org/

Listing 5.2 Definition von NPM-Abhängigkeiten in der Datei „package.json“

```
{
      "name": "@aeb/model-manager",
      "version": "0.1.0",
      "description": "A generic GraphQL server...",
      "repository": {
            "type": "git",
            "url": "..."
      },
      "scripts": {
            "start": "npm run start:live",
            "start:live": "ts-node index.ts",
            "start:idea": "ts-node %NODE_DEBUG_OPTION% index.ts",
            ...
      },
      "devDependencies": {
            "ts-node": "^2.0.0",
      ...
      },
      "dependencies": {
            "typescript": "^2.1.5",
            "jsonschema": "^1.1.1",
            "graphql": "^0.9.1",
            "body-parser": "^1.16.0",
            "arangojs": "^5.6.0",
            ...
      }
}
```

Wie bereits erwähnt, soll die Komponente als TypeScript-Projekt auf einem NodeJS-Server umgesetzt werden. Typescript bietet die Möglichkeit, auf sämtliche Javascript-Bibliotheken zuzugreifen und diese einzubinden. Zur Implementierung der GraphQL-API kann somit die GraphQL-Javascript-Referenzimplementierung[1] aus GitHub verwendet werden. Für die ArangoDB wird ebenfalls eine Javascript-Bibliothek angeboten, sodass sowohl die Persistenz-API, als auch die Client-API vollständig mit Typescript über externe Bibliotheken kompatibel sind. Als Javascript-Laufzeitumgebung, stellt NodeJS zudem den sogenannten „Node Package Manager“, kurz NPM[2], zur Verfügung, welcher alle Abhängigkeiten bezüglich verwendeter Code-Bibliotheken automatisch verwaltet. Diese müssen dabei in der Datei `package.json`, mit Angabe der minimalen Version, vom Entwickler definiert werden. Über die Konsole können diese anschließend über `npm install` vollständig automatisiert heruntergeladen werden.

[1] https://github.com/graphql/graphql-js
[2] https://www.npmjs.com/

In Listing 5.2 ist ein Auszug der wichtigsten Bibliotheksdefinitionen für das Model Manager Projekt abgedruckt. Diese befinden sich unter den Knoten „devDependencies“ und „dependencies“, jeweils mit der minimal kompatiblen Versionsnummer versehen. Als Bestandteil von „scripts“ können ausführbare NPM-Befehle durch Hinterlegen eines Kommandozeilenbefehls definiert werden. Das bedeutet, dass durch die Ausführung von `npm start:idea` implizit der hinterlegte Kommandozeilenbefehl `ts-node %NODE_DEBUG_OPTION% index.ts` ausgeführt werden kann.
Zu Beginn der Datei stehen noch allgemeine Informationen, wie unter anderem der Projektname, die Version, oder Informationen zum Code-Repository.

TypeScript unterstützt im Gegensatz zu Javascript eine objektorientierte Programmierung, also unter anderem die Definition von Klassen oder Interfaces. In Abbildung 5.2 ist die grundlegende Klassenstruktur des Model Manager Prototypen exemplarisch mit den Abhängigkeiten untereinander dargestellt.
Im oberen Teil wird ein allgemeines Datenbank-Interface namens `DatabaseAdapter` definiert. Dies legt alle nötigen Methoden zur Erzeugung, Löschung oder Aktualisierung von RootEntities etc. an. Wird eine neue Datenbanktechnologie angebunden, muss analog zur ArangoDB ein Adapter geschrieben werden, der davon erbt und Datenbank-spezifisch die definierten Methodenrümpfe aus-implementiert. Unter der Klasse `ArangoDBAdapter` versteht sich die Schnittstelle vom Model Manager zur Datenbank ArangoDB. Alle ArangoDB-spezifischen Implementierungen sind dabei durch entsprechende Methodenaufrufe weg gekapselt. Diese Architektur gewährleistet, dass zu einem beliebigen Zeitpunkt die ArangoDB durch eine beliebige Datenbank ersetzt werden könnte, sogar anderer Datenbanktechnologie. Der einzige Programmieraufwand steckt dann in der speziellen Aus-Implementierung des neuen Datenbank-Adapters.
Die Klasse `ModelDefinition` ist zuständig für das Einlesen des JSON-Datenmodells. Die Modelldateien werden dabei über den integrierten Javascript JSON Parser[1] geparsed, gegenüber dem bereits beschriebenen JSON Schema validiert und anschließend in entsprechende Objekte gespeichert. Die dafür nötigen Typklassen zu den im Metamodell definierten Objekttypen sind `RootEntityDefinition`, `ValueObjectDefinition` und `AssociationDefinition`. Die Instanz der Klasse ModelDefinition hält nach erfolgreichem Einlesevorgang je ein Array aus RootEntities,

[1] https://www.w3schools.com/js/js_json_parse.asp

ValueObjects und Associations vor. Dies repräsentiert das JSON-definierte Datenmodell zur Laufzeit im Code des Model Managers.

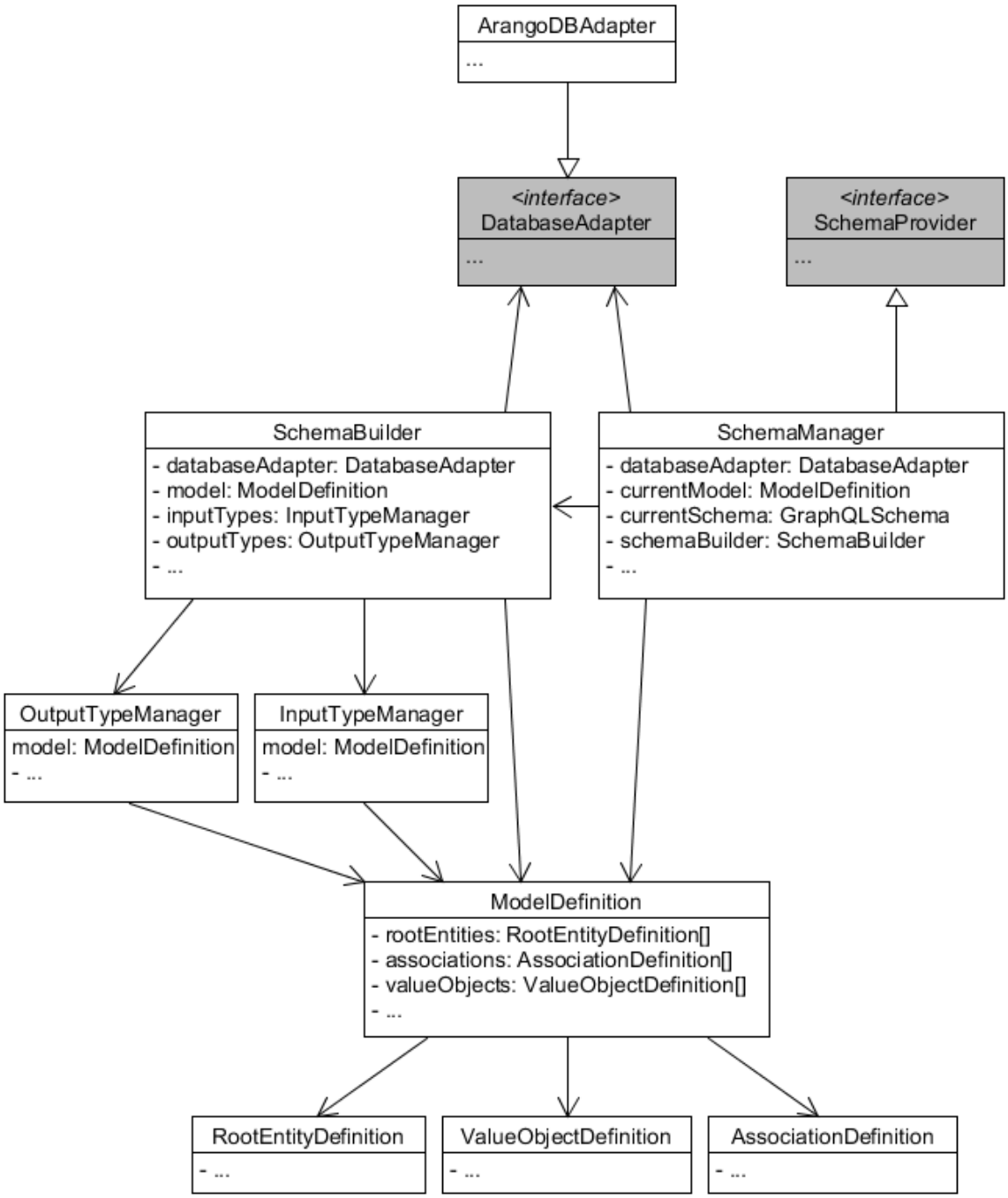

Abbildung 5.2: Klassenstruktur des Model Manager Prototypen als UML-Diagramm

Die nächste Stufe hat die Aufgabe, das eingelesene Datenmodell in ein GraphQL-Schema zu überführen, um die Client-API bereitstellen zu können. Die Klasse SchemaBuilder erzeugt dieses notwendige GraphQL-Schema auf Basis des eingelesenen Datenmodells mit seinen 3 verschiedenen Objekttypen. Der SchemaBuilder nutzt hierzu die GraphQL-JavaScript-Referenzimplementierung. Diese bietet die eingangs vorgestellten GraphQL-Typen bereits als Klassen an und reduziert somit den Entwicklungsaufwand.

Listing 5.3 Generierung eines GraphQL-Schemas mithilfe der GraphQL JavaScript Referenzimplementierung

```
...
return new GraphQLSchema({
      query: new GraphQLObjectType({
            name: 'query',
            description: 'The root query type',
            fields: this.createQueries(dataModel),
      }),
      mutation: new GraphQLObjectType({
            name: "mutation",
            description: "The root mutation type",
            fields: this.createMutations(dataModel)
      })
});
...
```

Wie in Listing 5.3 zu sehen ist, kann mit Hilfe der Referenzimplementierung vergleichsweise einfach ein GraphQL-Schema erzeugt werden. Die vordefinierten Klassen „GraphQLSchema“ und „GraphQLObjectType“ müssen dazu geeignet instanziiert werden. Das komplette GraphQL-Schema, welches nach außen veröffentlicht werden soll, wird im Objekt der Klasse GraphQLSchema zusammen gebaut. Diese sieht bereits die Standardfelder „query“ und „mutation“ vor, wobei mutation optional ist. Es könnte auch eine reine Abfrage-API ohne Möglichkeiten zur Datenmodifikation gebaut werden.
Beide Felder des Schemas müssen ein GraphQLObjectType sein, welcher neben dem Namen einen Beschreibungstext und weitere Felder besitzt. Ein Feld wird als „GraphQLFieldConfig“ definiert. Es liefert einen „GraphQLOutputType“ zurück und spezifiziert optional reine resolve-Methode über den „GraphQLFieldResolver“, um zu beschreiben, wie das Feld aus der Datenbank geholt werden soll. Hier wird der entsprechende Datenbankadapter mit beispielsweise einer Query aufgerufen.

Die schematischen Zusammenhänge der GraphQL-Klassen innerhalb eines Schemas sind nochmals ungefähr in Abbildung 5.3 zusammengefasst.

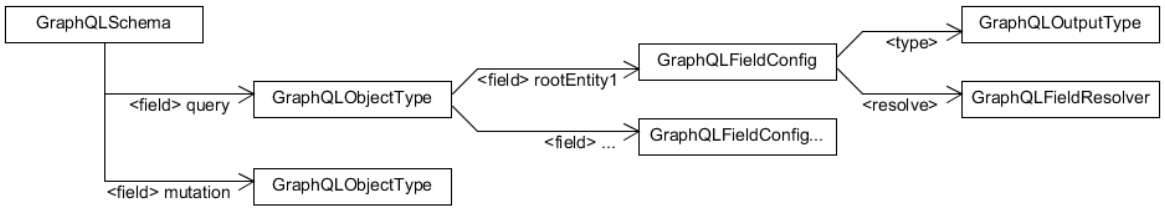

Abbildung 5.3: Schematische Zusammenhänge bei der Implementierung eines GraphQL-Schemas mit der Referenzimplementierung

Der OutputTypeManager und der InputTypeManager aus dem Klassendiagramm in Abbildung 5.2 sind zuständig für die Generierung und Verwaltung aller nötigen Ein- und Ausgabe-Typen auf Basis der eingelesenen RootEntities, ValueObjects und Associations. Diese werden als normale GraphQLObjectType-Objekte mit allen im Datenmodell definierten Feldern erzeugt.
Eine Anfrage nach allen Usern muss schlussendlich eine Liste aus User-Typen zurückgeben können. In Listing 5.4 wird beispielhaft die Definition einer Filter-Query im GraphQL-Schema gezeigt. Zunächt wird ein Filter-Feld als Argument erzeugtm it dem entsprechenden Eingabetyp für das aktuelle RootEntity. Wie bei GraphQL üblich, kann zusätzlich wieder ein Beschreibungstext vergeben werden.
Schlussendlich wird eine GraphQLFieldConfig erzeugt, mit Definition des Ausgabetyps und der Auflösung gegenüber der Datenbank. Für jedes Feld muss händisch beschrieben werden, wie die Daten über den DatabaseAdapter bei entsprechender GQL-Anfrage geholt oder geschrieben werden sollen. Im Beispielcode aus Listing 5.4, wird ein QUery-Objekt mit Filterinformationen generiert und der query-Methode des DatabaseAdapter übergeben, welcher im ANschluss die Kommunikation mit der ArangoDB steuert. Die Rückgabe der Anfrage des DatabaseAdapter muss in der Struktur dem Ausgabetyp für das RootEntity entsprechen.

Das letzte Modul, der SchemaManager, überwacht die JSON-Definitionen des Datenmodells und stößt bei Änderungen eine Neugenerierung des GraphQL-Schemas über den SchemaBuilder an. War die Generierung erfolgreich, so wird die alte GraphQL API direkt ersetzt, ohne dass ein Client dies mitbekommt. Diese Funktionalität wird realisiert über die Dateisystem-Operationen

Listing 5.4 Beispielhafte Implementierung einer Filteranfrage für RootEntities

```
...
let args: GraphQLFieldConfigArgumentMap = {
      filter: {
            type: this.getRootEntityInputType(rootEntity),
            description: 'Find ${decapitalize(rootEntity.pluralName)} where some
                fields equal given values'
      }
};

return {
      description: "Provides a filter query to all " +
          decapitalize(rootEntity.pluralName) + ' in the data base',
      type: makeNonNull(new
          GraphQLList(makeNonNull(this.getOutputType(rootEntity.name)))),
      args: args,
      resolve: (source, args, context, info) => {
            const query = buildRootEntityFromCollectionQuery({
                  rootEntity,
                  filterArgs: args['filter'] || {},
                  model: this.model,
                  ...
            });
            return this.databaseAdapter.query(query);
      }
};
...
```

von NodeJS, genauer einem asynchronen fs.watch[1] auf das Datenmodellverzeichnis. Dadurch werden Datenmodellanpassungen zur Laufzeit des Systems möglich, ohne den NodeJS-Server neustarten zu müssen.

Zu Testzwecken wurde der Model Manager um das Open-Source Projekt GraphiQL[2], einer Browseroberfläche zur händischen Ausführung von GraphQL-Anfragen und Introspektion des GraphQL-Schemas, erweitert. Dies ermöglicht, direkt im Browser GraphQL-Queries und Mutations inklusive Syntax-Prüfung zu schreiben und auszuführen. Ein integrierter „Document Explorer“ bietet die Möglichkeit, von den Root-Feldern query und mutation aus tiefer in das GraphQL-Schema einzusteigen. Es werden sogar die Beschreibungstexte zu allen Feldern angezeigt.

[1] https://nodejs.org/docs/latest/api/fs.html
[2] https://github.com/graphql/graphiql

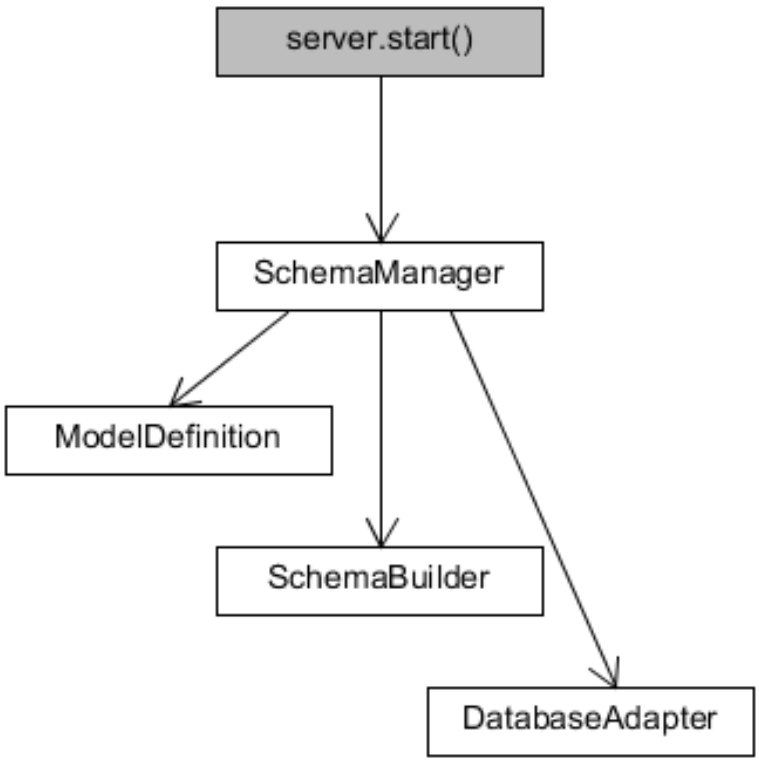

Abbildung 5.4: Darstellung des Startvorgangs einer Model Manager Instanz

In Abbildung 5.4 ist des Startprozess des Model Manager Prototyps schematisch dargestellt. Durch den Start des Servers wird ein `SchemaManager` erzeugt. Dieser organisiert nacheinander das Einlesen des JSON-Schemas, die Generierung des generischen GraphQL-Schemas und die Erzeugung des Datenbankadapters. Dazu erzeugt er zunächst Objekte der spezialisierten Klassen `ModelDefinition`, `SchemaBuilder` und `DatabaseAdapter`, um die entsprechenden Methoden sequentiell anzustoßen.
Nicht in der Abbildung dargestellt, startet er zusätzlich einen Thread, um Änderungen am Datenmodell zu überwachen.

5.1 Lauffähiges Beispiel

In den Abbildungen 5.5 und 5.6 sind beispielhafte GQL-Anfragen an den lauffähigen ModelManager-Prototypen dargestellt.

Abbildung 5.5: Anlegen eines neuen Users über das GraphiQL-Interface des Model Managers

Abbildung 5.6: Abfragen des angelegten Users aus Abbildung 5.5 über das GraphiQL-Interface des Model Managers

In Abbildung 5.5 wird zunächst der User „Stefan Schmid“ angelegt, indem ein entsprechender Input-Typ übergeben wird. Als einziges Attribut soll die ID des DOkuments in der Datenbank zurückgegeben werden. Der Model Manager liefert hierfür als Resultat genau die angeforderte Struktur, nämlich je gefundenem User lediglich das Feld „id“.
Einen Schritt später in Abbildung 5.6 wird dieser angelegte User anhand einer Filterung nach dem Vornamen wieder abgefragt. Analog zur vorherigen Mutation, wird wieder nur das Feld „id“ verlangt und zurückgegeben.

5.2 Zusammenfassung und Ausblick

Der initiale Prototyp des Model Managers funktioniert zuverlässig mit der ArangoDB im Hintergrund. Er unterstützt schon RootEntities, wie sie im Metamodell definiert wurden, mit skalaren Attributtypen und ist, wie in den Abbildungen 5.5 und 5.6 zu sehen, in der Lage das benutzerdefinierte Datenmodell in Form einer generischen GraphQL-API bereitzustellen. Diese erzeugt für jedes definierte RootEntity automatisch eine Filter-Query unterhalb des Root-Query-Typs. Je Filter-Query werden nur die Attribute der entsprechenden RootEntity zur Filterung angeboten. Über den GraphQLOutputType wird genau spezifiziert, welche Felder vom Resultat abgefragt werden können.
Analog dazu werden generische CRUD-Mutations für alle RootEntities erzeugt. Dies beinhaltet eine Add-, Delete- und Modify-Mutation je RootEntity. Über den Input-Typ können jeweils Attribute mitgegeben werden. Mein Entwicklerkollege Jan Melcher hat zusätzlich zur RootEntity-Funktionalität die Unterstützung für eingeschachtelte Objekte implementiert, mit Assoziationen begonnen und Paging inkl. Sortierung in GraphQL Queries eingebaut.
Der Ansatz zur Implementierung eines Model Managers hat gezeigt, dass dieses Vorhaben mit einer NoSQL-Datenbank im Hintergrund und einer GraphQL-API als Abfragemöglichkeit insgesamt gut funktioniert. Die Unterstützung für JavaScript Bibliotheken ist zudem sehr umfangreich und erleichtert die Weiterentwicklung. Die Implementierung von Assoziationen mit Feldern als Arango-Kanten konnte im Umfang der Masterarbeit nicht umgesetzt werden. Trotzdem sind die initialen Schritte dafür gemacht und die Weiterentwicklung im Team ist gesichert.

Die Implementierungsarbeit des Model Managers teilt sich in die Masterarbeit und Unterstützung durch das nEXt-Entwicklerteam auf. Im Rahmen der Masterarbeit wurden vor allem die GraphQL-Typen und die zugehörigen Datenbankoperationen für skalare Attributtypen selbständig implementiert. Mit Unterstützung des Teams wurden die Attributtypen um eingeschachtelte ValueObjects und die momentan noch unvollständigen Assoziationen erweitert. Möglich sind derzeit nur weniger umfangreiche Referenzen auf RootEntities durch einfache Speicherung der referenzierten Dokument-ID aus der Datenbank als Attribut.

6 Evaluierung des Prototypen

Dieses Kapitel befasst sich mit der Evaluierung des ersten implementierten Model Manager Prototypen. Dabei wird das Augenmerk sowohl auf die Performance von Datenabfragen und Modellaktualisierungen, als auch verschiedene Szenarien für Daten- und Modelländerung gelegt, um die Korrektheit von Operationen zu überprüfen.

6.1 Performance

Zur Messung der Laufzeiten für Model Manager Anfragen wurde von Jan Melcher eine Testsuite erstellt und über Continuous Integration (CI) als Pipeline im Gitlab-Repository[2] durchgeführt, in der auf zufällig generierten Daten gemessene GraphQL-Anfragen gleichzeitig abgefeuert werden können. Die Benchmarks sind dabei in mehrere Kategorien gestaffelt, nämlich Dokumente mit 100 Byte, 10 Kilobyte, 1 Megabyte, 10 Megabyte und 100 Megabyte. Die ermittelten Resultate wurden selbständig im Rahmen der Arbeit ausgewertet und bewertet.
In der folgenden Tabelle sind die erhaltenen Laufzeiten für das Anlegen bestimmter Dokumentgrößen aufgelistet:

Angelegte Dokumentgröße	100 Byte	10 KByte	1 MByte	10 MByte	100 MByte
Abgerundete Laufzeit	10 ms	23 ms	417 ms	3336 ms	30032 ms

Es fällt dabei auf, dass der Sprung von 100 Byte großen Dokumenten auf 10 Kilobyte große Dokumente sich in den Laufzeiten nur gut mit Faktor 2 anstatt 100 niederschlägt. Dies wird auch in den Diagrammen aus Abbildung 6.1 und 6.2 ersichtlich. Sobald die Dateigrößen in den Megabyte-Bereich steigen, nähert sich der Änderungsfaktor der Laufzeit linear dem Änderungsfaktor der Dokumentgrößenänderung an (siehe Abbildung 6.2).

[2] https://about.gitlab.com/features/gitlab-ci-cd/

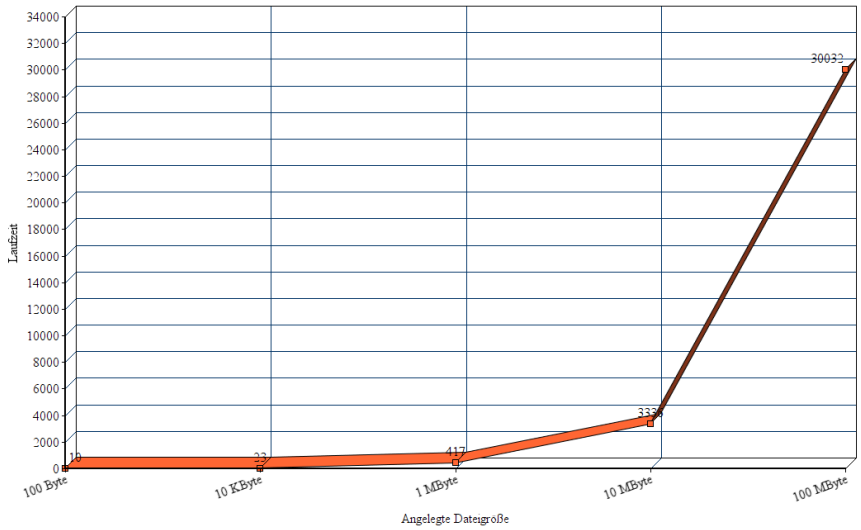

Abbildung 6.1: Laufzeiten für die Erzeugung verschieden großer Dokumente mit dem Model Manager

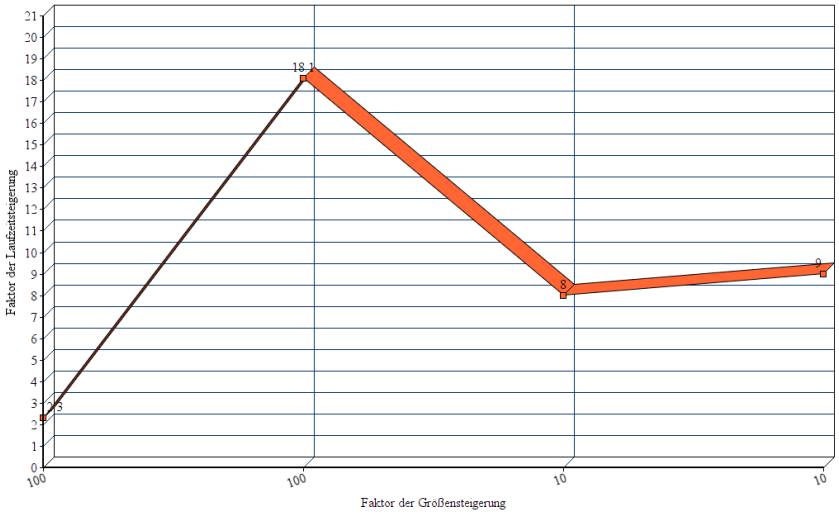

Abbildung 6.2: Faktor der Laufzeitänderung im Verhältnis zum Faktor der Dateigrößenänderung

Bei sehr großen Dokumenten führt eine Verzehnfachung der Größe somit auch zu einer ungefähren Verzehnfachung der Laufzeit. Dieses Verhalten ist für den Betrieb stets gut kalkulierbar, zumal für kleine Dokumente, welche den Großteil ausmachen, deutlich schnellere Laufzeiten erzielt werden.

Zur Evaluierung wurden nicht nur Benchmarks für das Anlegen von verschieden großen Dokumenten durchgeführt, sondern zudem auch unterschiedliche Abfrageszenarien betrachtet. Um ein Dokument der Größe 100 Byte aus der Datenbank zu holen, wurde die Abfrage auf verschieden großen Dokument-Collections von 1.000 bis 1.000.000 Elementen durchgeführt. Es hat sich gezeigt, dass dieser Vorgang bei 100 Byte großen Dokumenten unabhängig von der Collection-Größe ist. Die Laufzeit bewegt sich hier jeweils im Rahmen von 4 ms.
Analog dazu wurde das Abfragen großer Dokumente mit 10 KByte, 1 MByte und 10 MByte aus einer Collection mit 10 Elementen gemessen. Wie schon bei der ersten Messung aus Abbildung 6.2 nähert sich der Änderungsfaktor der Laufzeit dem Änderungsfaktor der Dokumentgröße ab dem MByte-Bereich linear an. Somit dauert das Abfragen eines 10 MByte-Dokuments etwa zehn mal so lange wie das Abfragen eines 1 MByte-Dokuments.
Geht man wieder einen Schritt weiter und betrachtet den Fall, nur je zwei zufällige Felder eines Dokuments abzufragen, so bleibt die Laufzeit sogar konstant bei etwa 3 ms, egal ob die Dokumente 10 KByte, 1 MByte oder 10 MByte groß sind. Selektive Abfragen sind somit in nahezu konstanter Zeit sehr effizient möglich.

Die Auswertung der Benchmark-Ergebnisse führte unterm Strich zu einem positiven Resultat, da in keinem der Fälle mehr als ein linearer Zusammenhang zwischen Dokumentgröße und Laufzeit ermittelt werden konnte. Für kleine bis mittlere Dokumente liegen die Laufzeiten von Abfragen oftmals sogar deutlich darunter.

6.2 Szenarien

Hier werden verschiedene Szenarien für Datenmodellanpassungen betrachtet, um diese auf etwaige Probleme und nötige Nachbesserungen zu untersuchen. Grob betrachtet lassen sich Datenmodelländerungen in drei Kategorien klassifizieren. Entweder werden neue Objekte/Attribute angelegt, bestehende gelöscht oder ebenfalls bestehende verändert.
Bei den Betrachtungen muss immer im Hinterkopf beachtet werden, dass die Clients mit ihren

(festen) GraphQL-Anfragen, im Gegensatz zum Datenmodell nicht verändert werden. Dadurch liegt der Fokus vor allem auf der Robustheit der Flexibilität des Model Managers gegenüber den Anwendungen, die ihn ansprechen.

6.2.1 Erweiterung des Datenmodells

Im ersten Schritt werden additive Änderungen am Datenmodell betrachtet. Dies beinhaltet die Hinzunahme einer neuen RootEntity, Entity oder eines ValueObject, sowie neuer Attribute innerhalb bestehender Objekttypen.
Die Hinzunahme von Objekten im Sinne des Metamodells zur Laufzeit, gestaltet sich grundsätzlich als unkritisch, da sich die bisherigen GraphQL-Anfragen noch nicht auf die neuen Objekte beziehen können. Die Definition von Attributen innerhalb dieser neuen Objekte gestaltet sich ebenfalls als unkritisch, da auch sie zu diesem Zeitpunkt noch nirgends verwendet wurden.
Die letzte Möglichkeit einer additiven Änderung beschreibt die Hinzunahme eines neuen Attributs zu einem bestehenden Objekt. Dies ist ebenso wie oben unkritisch, solange nur ein Attribut mit Namen, Typ und optional dem „immutable“-Tag definiert wird. Interessanter wird es, wenn ein neues Pflicht-Attribut, also mit gesetztem „required“-Tag, zu einem bestehenden Objekt in Form einer RootEntity, Entity oder eines ValueObject hinzugefügt wird. Zunächst funktionieren dafür noch die alten GraphQL-Queries, da sie nicht versuchen, das neue Attribut abzufragen. Problematisch wird es aber schon für neue Queries, die versuchen das neue Pflicht-Attribut auf alten Datensätzen abzufragen, in denen es noch nicht existiert. Ebenso laufen die alten Add-Mutations für das betroffene RootEntity mit neuem Pflicht-Attribut auf einen Fehler, da sie dieses zukünftig als zwingenden Übergabeparameter voraussetzen.
Ein Lösungsansatz für diese Probleme wäre, z.B. den Gültigkeitsbereich eines neuen Pflicht-Attributs nicht auf bereits bestehende Daten zu erstrecken. Man könnte dies durch konsequente Versionierung der Daten und Datenmodelle erreichen. Ein Datensatz X würde dann beispielsweise die Versionsnummer 1.0 des Datenmodells tragen, im Zuge dessen er erzeugt wurde. Ändert sich dieses Datenmodell, so erhält es exemplarisch die Versionsnummer 1.1, der Datensatz X behält aber die alte Versionsnummer 1.0 und setzt somit das neu definierte Pflicht-Attribut nicht voraus. Zur Umsetzung dieses Konzepts müssten die Clients dem Model Manager zusätzlich zur GraphQL-Query ebenfalls eine Versionsnummer mitgeben, um die alten Queries und Mutations weiterhin zu erlauben. Die GraphQL-API würde nur das jeweils aktuelle Schema in Form von generischen Queries und Mutations nach außen veröffentlichen.

6.2.2 Reduzierung des Datenmodells

Im zweiten Schritt werden subtraktive Änderungen am Datenmodell betrachtet. Dabei können entweder ganze Objekte oder Attribute entfernt werden, als auch nur Eigenschaften von Attributen. Die explizite Löschung von RootEntities zieht das Problem nach sich, dass alle Queries und Mutations, die darauf basieren, verschwinden und damit bei Anfrage von den alten Clients ins Leere laufen. Hier könnte zumindest über ein Versionierungskonzept wie oben Abhilfe geschaffen werden, indem alten Apps der Zugriff auf die alten Daten weiterhin über die alten Queries und Mutations der letzten Datenmodell-Version gestattet wird. Fachlich gesehen erstreckt sich der Gültigkeitsbereich einer Löschung eines RootEntity nämlich ab der entsprechenden Datenmodellanpassung in die Zukunft hinein. Angelegte Daten aus der Vergangenheit bleiben trotzdem bestehen.
Die Löschung einer Entity oder eines ValueObject zieht immer auch die Löschung der betroffenen Attribute nach sich, da ansonsten die Modell-Validierung einen Fehler wirft. Entities und ValueObjects besitzen keine eigenen Queries und Mutations, sondern können nur, wie im Metamodell definiert, als Attributtypen in einem Objekt eingeschachtelt werden. Dieses Szenario kann somit unter dem Gesichtspunkt einer Attribut-Löschung mit betrachtet werden. Wird ein Attribut gelöscht, so betrifft dies alle GraphQL-Queries und Mutations auf Seite der Client-Apps, welche dieses Attribut abfragen, filtern oder als Argument einer Mutation übergeben möchten. Auch hier wäre wieder ein Versionierungskonzept denkbar, da die alten Datensätze auf einem alten Modellstand basieren und somit unter Umständen auch dessen Fachlichkeit unterstützen sollten.
Ein anderer Lösungsansatz wäre, die zu löschenden Objekte oder Attribute lediglich als „deprecated“[1] zu markieren. Dadurch könnten sie weiterhin abgefragt werden, nur den Clients würde im Idealfall eine Warnung mitgegeben werden, dass sie nicht mehr aktuell sind.

6.2.3 Anpassung bestehender Typen des Datenmodells

Die dritte und letzte Betrachtung bezieht sich auf Änderungen bestehender Typen des Datenmodells. Hierbei kommen viele Möglichkeiten in Betracht. Denkbar wären z.B. Transformationen zwischen Objekttypen, z.B. von Entity auf RootEntity oder von RootEntity auf Entity. Die GraphQL-API würde sich drastisch verändern, da nur Queries und Mutations auf Basis von

[1] http://whatis.techtarget.com/definition/deprecated

RootEntities angeboten werden. wird eines dieser RootEntities zu einem Entity, ist es nicht mehr zugreifbar. Analog dazu geschieht es in die andere Richtung, da zudem die eingeschachtelten Attributtypen der Entity gelöscht werden müssen. Das Szenario, dass Objekttypen zur Laufzeit verändert werden, sollte aufgrund der immensen Probleme, die dadurch entstehen, generell verboten werden.
Attributtypen im Datenmodell können sich zur Laufzeit ebenfalls verändern. Es wären Wechsel zwischen skalaren Datentypen denkbar. Während alle skalaren Datentypen auch als String repräsentiert werden könnten, stellt der Wechsel von String auf Int etc. möglicherweise ein Problem dar. Die GraphQL-Schnittstelle läuft auf einen Fehler, sobald ein Integer angefordert wird, in der Datenbank jedoch ein Name aus Buchstaben steht. In die Andere Richtung werden alle anderen skalaren Typen problemlos als String-Repräsentation zurückgegeben. Andere Typ-Modifikationen, z.B. von Skalar auf eingebettete Strukturen, wie Entities, ValueObjects oder Listen scheitern im Prototypen ebenso, da sie nicht kompatibel sind.
Eine letzte Betrachtung gilt der Anpassung von den Attributeigenschaften „required“ und „immutable“. WIrd ein gewöhnliches Attribut plötzlich verpflichtend, so ergeben sich die gleichen Probleme wie bei additiven Änderungen. Die Umkehrrichtung ist unproblematisch. Wird ein gewöhnliches Attribut plötzlich unveränderlich, so hat dies alleine Einfluss auf die Update-Mutation der entsprechenden RootEntity. Alle anderen Queries und Mutations funktionieren weiterhin. AUch hier ist die Gegenrichtung völlig unproblematisch, da nur die besagte Update-Mutation um eine Option erweitert werden würde.

6.3 Zusammenfassung

Zum Abschluss der Evaluierung, werden die Ergebnisse bezüglich der Performance und Flexibilität des Model Managers nochmals zusammengefasst und allgemeiner betrachtet.
Die Performance-Analyse des Model Managers fiel durchweg positiv aus. Die erst bei sehr großen Dokumenten maximal linearen Zusammenhänge schaffen eine gewisse Planungssicherheit für einen späteren Produktiveinsatz. Üblicherweise bewegen sich die Laufzeiten für Queries und Mutations durchweg im Millisekunden-Bereich, was ausreichend schnell ist, wenn später jeder Kunde eventuell sogar seinen eigenen Model Manager erhält.
Die Flexibilität des Datenmodells wurde für den Prototypen, wohl wissend, dass es stellenweise Probleme geben wird, nicht im Voraus eingeschränkt. Dadurch war es in der Evaluierung möglich, alle denkbaren Szenarien bezüglich Datenmodelländerungen auszuprobieren. Generell

hat sich gezeigt, dass additive Änderungen gegenüber subtraktiven oder modifikativen eher als unkritisch zu betrachten sind. Einzige Ausnahme ist dabei die Einführung eines neuen Pflicht-Attributs für ein bestehendes Objekt.Hier stellt sich bei der Datenmodell-Anpassung auch immer die Frage, welchen Gültigkeitsbereich eine Änderung für die Daten haben soll. Durch die Verwendung einer dokumentenorientierten NoSQL-Datenbank im Hintergrund, muss nicht analog zu SQL ein neues Attribut als Spalte oder Eintrag in allen existierenden Dokumenten angelegt werden. Die Altdaten bleiben ungeachtet dem Datenmodell bestehen. Die Löschung von Objekten oder Attributen aus dem Datenmodell kann einfach dadurch entschärft werden, dass die entsprechenden Teile als „deprecated“ markiert werden könnten und somit bei Zugriff eine entsprechende Warnung an der Client zurückgegeben wird, ohne direkt auf einen Fehler zu laufen.
Modifikative Anpassungen im Sinne von Typänderungen für Objekte führen zu immensen Problemen für alte Clients. Deutlich Gutmütiger sind Typänderungen für Attribute innerhalb des skalaren Kontexts. Alle skalaren Typen können als String dargestellt werden, während String, die nur aus Zahlen bestehen z.B. auch als Integer repräsentiert werden könnten.

Vergleicht man den Model Manager mit dem zu Beginn vorgestellten Backend GraphCool, so ergeben sich bei Modellanpassungen einige Unterschiede. GraphCool basiert auf SQL-Datenbanken und entfernt direkt die zugehörige Spalte, wenn ein Attribut auf der Console gelöscht wird. Nimmt man es anschließend identisch wieder auf, so sind alle vorherigen Daten gelöscht. Das gilt sogar für skalare Typwechsel von Attributen. Dieses Verhalten schränkt die Flexibilität im Modell deutlich gegenüber dem Model Manager ein. Hier bleiben die Daten bei Modelländerungen aufgrund der Schemalosigkeit der Datenbank vollständig unangetastet. Eine bestehende Schwachstelle des Model Managers ist hingegen das Umbenennen von Attributen oder Objekten. Währen bei GraphCool die Daten erhalten bleiben, werden im Model Manager die Altdaten nicht mit umgezogen. Dies wäre eine denkbare Erweiterung für die Zukunft.

6.4 Bewertung der Evaluierungsergebnisse

Die Evaluation wurde bis auf die reine Durchführung der Benchmarks selbständig erarbeitet. Im Rahmen der Flexibilität wurden eigenständig Szenarien für Datenmodell-Anpassungen entwickelt und prototypisch selbst durchgeführt. Je Szenario wurden anschließend die Probleme für Client-Apps theoretisch betrachtet. Realistischer wäre hier eine Betrachtung des Verhaltens anhand von Demo-Anwendungen, aufbauend auf dem Model Manager Prototyp gewesen. Der unvollständige Implementierungsstand und unternehmerische Planungen bezüglich erster Demo-Anwendungen, erlaubten es im Sinne der Masterarbeit nicht, eine Client-App darauf zu testen.
Nichtsdestotrotz können die betrachteten Szenarien, vor allem additiver Anpassungen als sehr realistisch und später am häufigsten vorkommend betrachtet werden. Wie bereits in der Vision der AEB beschrieben, soll ein Fokus auf einfacher Erweiterung liegen und genau unter diesen Gesichtspunkt fallen additive Szenarien. Zur Laufzeit werden überwiegend Anpassungen, wie „Firma X benötigt innerhalb der Sendung eine weitere Nummer als extra Integer-Attribut“, verlangt.

Die Performance-Betrachtungen sind durchweg positiv zu bewerten und betrachten eher deutlich größere Datenmengen als im Produktivbetrieb anfallen würden. Dadurch werden die Worst-Case-Szenarien definitiv abgedeckt und selbst diese sind als akzeptabel zu bewerten.

7 Fazit und Ausblick

Diese Masterarbeit zeigt die schrittweise Konzeption und Implementierung eines Ansatzes, um ein flexibles Datenmodell umzusetzen. Auf Basis der ermittelten und festgelegten Anforderungen für die AEB wurden Konzepte zur Umsetzung entwickelt und anschließend in einem lauffähigen Prototypen implementiert. Die Ergebnisse wurden abschließend ausführlich evaluiert, um den Erfolg des Konzepts in der Praxis zu testen.

7.1 Fazit

In dieser Ausarbeitung wurden zu Beginn aktuelle Technologien und Konzepte erläutert, die später als Basis für einen Prototypen dienten. Das Datenmodell wurde im Sinne des Domain-Driven Design selbständig über ein Metamodell definiert. Dieses kennzeichnet die Anforderungen an das Datenmodell, welche im Model Manager konzipiert wurden und spielt eine tragende Rolle in der gesamten Umsetzung.
Der Prototyp des Model Managers wurde als Cloud-fähige Komponente mit einer GraphQL-Schnittstelle zu den Client-Apps mit Unterstützung des nEXt-Entwicklerteams implementiert. Ein Datenmodell kann dabei als JSON-Struktur spezifiziert werden. Dieses wird zur Laufzeit in ein GraphQL-Schema konvertiert, welches generische CRUD[2]-Queries und Mutations auf den RootEntities des Datenmodells unterstützt. Diese werden automatisch über den Datenbankadapter zur Persistenzlösung in Form von ArangoDB durchgereicht. Der Funktionsumfang des ersten Prototypen implementiert noch nicht vollständig das Metamodell, bietet aber eine gute Basis für eine Weiterentwicklung durch sinnvolle und abgekoppelte Code-Strukturen. In erster Linie fehlen noch Assoziationen zwischen RootEntities.
Die Evaluation des Model Managers wurde sowohl aus Performance- als auch aus Flexibilitätssicht durchgeführt. Die Performance-Benchmarks wurden dabei von einem Kollegen

[2] https://glossar.hs-augsburg.de/CRUD

durchgeführt, aber im Rahmen der Masterarbeit selbständig bewertet. Es hat sich gezeigt, dass die Laufzeit im schlimmsten Fall linear zur Dokumentgröße in der Datenbank skaliert. In der Praxis wird man jedoch eher deutlich darunter liegen. Die Analyse der Flexibilität hat gezeigt, dass in erster Linie additive Anpassungen unproblematisch sind, solange es sich nicht um neue Pflicht-Attribute handelt. Dies stellt bereits einen großen Mehrwert im zukünftigen Produktiveinsatz dar, da Fachexperten das Modell ohne einen Softwareentwickler zumindest gefahrlos erweitern können. Mit gewissen Einschränkungen sind auch subtraktive oder modifikative Anpassungen möglich.

7.2 Ausblick

Die Evaluierung hat gezeigt, dass der Model Manager Ansatz, auch im Sinne der Performance, durchaus tauglich für eine produktive Komponente ist. Darum soll der Prototyp schrittweise weiter ausgebaut werden, solange bis er alle Anforderungen aus dem Metamodell erfüllt. Die Implementierung auf Basis von TypeScript war ebenfalls erfolgreich, da ein großes Ökosystem an JavaScript-Bibliotheken zur Verfügung steht. Um aus Sicht der Entwicklung eine möglichst hohe Durchgängigkeit von Technologien zu erreichen, wird aktuell der Einsatz von TypeScript vom Backend bis zu den Clients erprobt. Zudem wird versucht, alle nEXt-Komponenten über GraphQL-Schnittstellen miteinander zu verbinden.
Angedacht für die Zukunft ist zudem eine Konfigurationsschnittstelle für das JSON-Datenmodell im Model Manager. Diese würde die Umsetzung einer Konfigurationsoberfläche für das Datenmodell, welches aktuell direkt in einer JSON-Datei geschrieben wird, ermöglichen. Die Oberfläche könnte dabei, zusätzlich zur rein optischen Darstellung, direkt eine Validierung der Änderungen anhand des „JSON Schema“ durchführen und somit den Anwender unterstützen. Möglich wäre dabei auch eine Einteilung der Fachexperten in bestimmte Rechtegruppen, z.B. nur zur Durchführung additiver Änderungen.

Der Model Manager wird in der aktuellen nEXt-Architektur zukünftig eine tragende Rolle als Backend-Komponente spielen und zusammen mit den Apps und einer Prozess-Engine die neue Software-Plattform definieren. GraphQL soll zudem als Schnittstelle zwischen allen Komponenten ausprobiert werden. In naher Zukunft sollen dann erste testweise Client-Apps über GraphQL mit dem Model Manager kommunizieren, um den Praxiseinsatz besser evaluieren zu können.

Literaturverzeichnis

[ABK+09] Altmanninger, Kerstin ; Brosch, Petra ; Kappel, Gerti ; Langer, Philip ; Seidl, Martina ; Wieland, Konrad ; Wimmer, Manuel: Why model versioning research is needed. An experience report. In: *Proceedings of the MoDSE-MCCM 2009 Workshop@ MoDELS* Bd. 9, 2009

[AK03] Atkinson, Colin ; Kuhne, Thomas: Model-driven development: a metamodeling foundation. In: *IEEE software* 20 (2003), Nr. 5, S. 36–41

[AMST09] Arendt, Thorsten ; Mantz, Florian ; Schneider, Lars ; Taentzer, Gabriele: Model refactoring in Eclipse by LTK, EWL, and EMF refactor: a case study. In: *Model-Driven Software Evolution, Workshop Models and Evolution*, 2009

[Ara16] ArangoDB GmbH: What is a multi-model database and why use it. (2016). https://www.arangodb.com/wp-content/uploads/2017/01/ArangoDB-White-paper-What-is-a-multi-model-database-and-why-use-it.pdf

[Bar09] Bartelt, Christian: Inconsistency Analysis at Integration of Evolving Domain Specific Models based on OWL. In: *Int Conf Model Driven Eng Lang Syst*, Citeseer, 2009, S. 27–37

[Bay02] Bayer, Thomas: REST Web Services - Eine Einführung. (2002). http://www.oio.de/public/xml/rest-webservices.pdf

[BCAT14] Bugiotti, Francesca ; Cabibbo, Luca ; Atzeni, Paolo ; Torlone, Riccardo: Database design for NoSQL systems. In: *International Conference on Conceptual Modeling*, Springer, 2014, S. 223–231

[BGK+06] Balasubramanian, Krishnakumar ; Gokhale, Aniruddha ; Karsai, Gabor ; Sztipanovits, Janos ; Neema, Sandeep: Developing applications using model-driven design environments. In: *Computer* 39 (2006), Nr. 2, S. 33–40

[DEH14] Dohmen, Lucas ; Edlich, Ing S. ; Hackstein, Michael: *A Declarative Web Framework for the Server-side Extension of the Multi Model Database ArangoDB*, Masters thesis, RWTH Aachen, Aachen, Diss., 2014

[DKC12] Dohmen, Lucas ; Klamma, PDR ; Celler, Frank: *Algorithms for large networks in the nosql database arangodb*, Bachelors thesis, RWTH Aachen, Aachen, Diss., 2012

[Eva04] Evans, Eric: *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. – ISBN 978–0–321–12521–7

[Eva09] Evans, Eric: *Eric Evans: What I've learned about DDD since the book*. https://dddcommunity.org/library/evans_2009_1/. Version: 2009

[Eva15] Evans, Eric: *Domain-Driven Design Reference*. 2015 http://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf

[Fac17] Facebook Open Source: *GraphQL: A query language for APIs*. http://graphql.org/. Version: 2017

[Fie00] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, University of California, Diss., 2000. http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf

[GRA17a] GRAPHCOOL: *Console*. https://www.graph.cool/docs/reference/platform/console-uh8shohxie/. Version: April 2017

[GRA17b] GRAPHCOOL: *Mutation callbacks*. https://www.graph.cool/docs/reference/platform/mutation-callbacks-ahlohd8ohn/. Version: April 2017

[GRA17c] GRAPHCOOL: *What technology is Graphcool using?* https://www.graph.cool/docs/faq/graphcool-technology-ul6ue9gait/. Version: April 2017

[HR09] Herrmannsdoerfer, Markus ; Ratiu, Daniel: Limitations of automating model migration in response to metamodel adaptation. In: *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2009, S. 205–219

[MA07] Marinescu, Floyd ; Avram, Abel: *Domain-Driven Design Quickly*. Lulu.com, 2007. – ISBN 978–1–4116–0925–9

[MC09] Meneses, Rafael ; Casallas, Rubby: A strategy for synchronizing and updating models after source code changes in Model-Driven Development. In: *Models and Evolution* (2009), S. 16

[Mon16] MongoDB: *JSON DB vs ODBMS*. https://www.mongodb.com/post/437029788/json-db-vs-odbms. Version: 2016

[MSOP86] Maier, David ; Stein, Jacob ; Otis, Allen ; Purdy, Alan: Development of an Object-Oriented DBMS. (1986), April. http://digitalcommons.ohsu.edu/cgi/viewcontent.cgi?article=1143&context=csetech

[Obj10] ObjectDB: *JPA Persistable Types (Entity class, Embeddable class, ...)*. http://www.objectdb.com/java/jpa/entity/types. Version: 2010

[RPKP09] Rose, Louis M. ; Paige, Richard F. ; Kolovos, Dimitrios S. ; Polack, Fiona A.: An analysis of approaches to model migration. In: *Proc. Joint MoDSE-MCCM Workshop*, Citeseer, 2009, S. 6–15

[sol17a] solid IT gmbh: *DB-Engines Ranking - die Rangliste der populärsten Datenbankmanagementsysteme*. http://db-engines.com/de/ranking. Version: März 2017

[sol17b] solid IT gmbh: *Schemafreiheit einer Datenbank*. http://db-engines.com/de/article/Schemafreiheit. Version: 2017

[Spr03] Sprinkle, Jonathan M.: *Metamodel driven model migration*, Citeseer, Diss., 2003

[Spr04] Sprinkle, Jonathan: Model-integrated computing. In: *IEEE potentials* 23 (2004), Nr. 1, S. 28–30

[Tec17] TechTarget: *object-oriented database management system (OODBMS or ODBMS)*. http://searchoracle.techtarget.com/definition/object-oriented-database-management-system. Version: 2017

[Ver12] Vernon, Vaughn: *Effective Aggregate Design Part III - DDD Denver Meetup*. https://vimeo.com/36884903. Version: Februar 2012

[Ver13] Vernon, Vaughn: *Implementing Domain-Driven Design*. Addison-Wesley, 2013. – ISBN 978–0–13–303988–7

[VKS14] VKSI Karlsruhe: *Entwicklertag 2014: Rauch, Borrmann - Einführung in Domain-Driven Design*. https://www.youtube.com/watch?v=1C9aQlz7N2I. Version: 2014

Alle URLs wurden zuletzt am 25. 04. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift