TYPO3 CMS

Datenmodifikation & Event Sourcing

M a s t e r a r b e i t

an der Hochschule für Angewandte Wissenschaften Hof

Fakultät Informatik

Studiengang Internet-Web Science

Hof, 1. November 2016

vorgelegt bei

Prof. Dr. Jürgen Heym

Alfons-Goppel-Platz 1

95028 Hof

vorgelegt von

Dipl.-Inf. (FH) Oliver Hader

Schaumbergstraße 2

95032 Hof

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde nach meiner besten Kenntnis bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hof, den 1. November 2016

Oliver Hader

Vorwort

Vor fast zehn Jahren habe ich meine Diplomarbeit zum Thema „Inline Relational Record Editing“ angefertigt. Die Implementierung dazu wurde mit der Version 4.1 in den Kernbestand von TYPO3 aufgenommen und hat mir persönlich einen ganz neuen Weg innerhalb des TYPO3 Universums ermöglicht – ich wurde in das Team der Kernentwickler eingeladen, war ein Jahr später Release Manager für TYPO3 Version 4.3 und zusätzlich von 2010 bis 2014 Leiter dieses großartigen Teams der TYPO3 Kernentwickler, bis ich erneut ein Studium begonnen habe, dessen Abschluss diese Arbeit darstellt. Das Thema bezieht sich dabei erneut auf das TYPO3 Projekt.

Da ich vor zehn Jahren noch nicht begreifen konnte, welche Auswirkungen TYPO3 auf meinen Werdegang haben wird und ich es damals versäumt habe, meinen Dank auszusprechen, möchte ich dies an dieser Stelle nachholen. Die damaligen Kernentwickler Ingmar Schlecht und Sebastian Kurfürst haben durch die Überprüfung meines Quellcodes während eines Code-Sprints in Hof erst ermöglicht, dass meine Arbeit in das TYPO3 Projekt aufgenommen werden konnte. Michael Stucki war zu diesem Zeitpunkt mein Vorgänger als Leiter des Kernentwicklerteams und hat entsprechend zu meiner persönlichen Geschichte beigetragen. Deshalb möchte ich mich für das damalige Engagement und das in mich gelegte Vertrauen besonders bedanken.

Die aktuelle Arbeit wurde mit ähnlich überwältigender Unterstützung ausgearbeitet – gleich mehrere TYPO3 Kernentwickler haben durch ihre Bereitschaft und Offenheit sehr wichtige Aspekte zu dieser Masterarbeit beigetragen. Mein herzlicher Dank gilt deshalb Benjamin Mack, Mathias Schreiber, Thomas Maroschik, Susanne Moog, Christian Kuhn und Frank Nägler für die gemeinsame Korrekturphase und die Aufgeschlossenheit, sich bereits im Vorfeld dieses Themenkomplexes gedanklich anzunehmen, sowie mich durch zahlreiche Diskussionen und Ideen zu fordern, zu motivieren und zu fördern.

Zum Schluss möchte ich mich noch bei allen bedanken, die mich auf meinem Weg in den letzten Jahren und Jahrzehnten begleitet haben und mich dabei immer wieder unterstütz haben – allen voran gilt dieser Dank meiner Familie.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AOP ..... *Aspektorientierte Programmierung*

API ..... *Application Programming Interface*

CAP ..... *Consistency, Availability, Partition Tolerance*

CMS ..... *Content Management System*

CQRS ..... Command Query Responsibility Segregation

CQS ..... *Command Query Separation*

CRUD ..... *Create, Read, Update, Delete*

CSS ..... *Cascading Style Sheets*

DBAL ..... *Database Abstraction Layer*

DDD ..... *Domain Driven Design*

DOM ..... *Document Object Model*

DTO ..... *Data Transfer Object*

FAL ..... *File Abstraction Layer*

HTML ..... *Hypertext Markup Language*

HTTP ..... *Hypertext Transfer Protocol*

IBAN ..... *International Bank Account Number*

MVC ..... *Model View Controller*

ORM ..... *Object Relational Mapping*

REST ..... *Representational State Transfer*

SFTP ..... *Secure File Transfer Protocol*

SQL ..... *Structured Query Language*

TCA ..... *Table Configuration Array*

TCE ..... *TYPO3 Core Engine*

UID/UUID ..... *Unique Identifier/Universally Unique Identifier*

WebDAV ..... *Web-based Distributed Authoring and Versioning*

XML ..... *Extensible Markup Language*

XSD ..... *XML Schema Definition*

1 Einführung & Motivation

TYPO3 hat sich vor allem im europäisch-deutschsprachigen Raum als verlässliches Content-Management-System für eine umfangreiche Bandbreite an Anwendungsfällen etabliert – so findet das System gleichermaßen bei regionalen Vereinen, wie auch bei börsennotierten Unternehmen Anklang. In der Geschichte von TYPO3 wurden seit 1997 zahlreiche Kernkomponenten sukzessive modernisiert, bereinigt und erweitert. In diesem Zusammenhang ist die Aufnahme von Extbase[1] im Jahr 2009 als MVC- und ORM-Schicht (Hader, 2009), sowie die komplette Umstrukturierung der Klassenhierarchien für die Einführung von PHP-Namespaces[2] während der Entwicklung von TYPO3 6.0 im Jahr 2012 wichtig und nennenswert (Hummel, 2012).

1.1 Ausgangssituation

Umbauarbeiten bei den in Entwicklerkreisen bekannten TCE[3]-Konstrukten gab es lediglich für TYPO3 CMS 7 hinsichtlich der Prozessabläufe der im Backend verwendeten Formularmasken (Mack, 2015). Die Komponenten zur Entgegennahme und Verwaltung für die persistente Speicherung von Benutzereingaben wurde dagegen letztmalig während des Entwicklungszeitraums von TYPO3 4.0 im Jahr 2005 im großen Stil angepasst. Zum damaligen Zeitpunkt wurden TYPO3 Workspaces integriert, eine Möglichkeit, Änderungen an Inhalten in einer separaten Arbeitsumgebung vorzubereiten und nach einer frei definierbaren redaktionellen Freigabe für alle Nutzer zu publizieren. Die Anforderung an Workspaces haben sich spätestens mit der Integration des File Abstraction Layers[4] in TYPO3 6.0 verschärft – die zu handhabenden Daten waren plötzlich im großen Umfang komplexer Natur und nicht mehr lediglich in einer flachen Hierarchie voneinander abhängig.

---

[1] Extbase wurde als Backport des DDD-Frameworks „TYPO3 Flow“ experimentell integriert

[2] eindeutige Namensräume für Software-Pakete, damit bessere Wiederverwertbarkeit

[3] TCE steht für TYPO3 Core Engine, das ursprüngliche „Herzstück“ des Systems

[4] datenbankgestützte Abstraktion von Dateien und deren Relationen und Verwendung

In der Kombination von Workspaces, Mehrsprachigkeit, Gruppenberechtigungen und komplexen Eltern-Kind-Relationen wird der Datenhandhabung von TYPO3 einiges abverlangt – zahlreiche Fehlerberichte und erstellte automatisierte Testfälle belegen die Anstrengungen, die bereits in diese Komponente investiert wurden und vermutlich noch investiert werden müssen[5].

1.2 Zielsetzung

Aus den benannten Unzulänglichkeiten in der Datenhandhabung lässt sich auch die Forschungsaufgabe dieser Arbeit ableiten.

Zunächst sollen moderne Paradigmen und Entwicklungsmuster der Software-Entwicklung analysiert werden – im speziellen liegt der Fokus auf dem Einsatz von Domain-Driven Design und Event Sourcing. Im weiteren Verlauf werden die vorhandenen Abläufe der Datenverarbeitung innerhalb von TYPO3 im Detail untersucht und hinsichtlich Optimierungsmöglichkeiten durch den Einsatz von Event Sourcing aus dem vorhergehenden Schritt bewertet.

In der Umsetzungsphase werden die gewonnenen Erkenntnisse in konkrete Änderungen, sowie neu zu schaffende Funktionalitäten für TYPO3 überführt und konkret ausgearbeitet. Diese Arbeit fokussiert sich dabei auf die Frage, ob der Einsatz von Event Sourcing Paradigmen vorhandene Unzulänglichkeiten der Datenhandhabung reduzieren oder gar beseitigen kann. Das generelle Ziel ist dabei, zentrale Komponenten zu etablieren, welche Abläufe der Datenverarbeitung vereinheitlichen und gleichzeitig dazu beitragen, Prozesse stabiler, verlässlicher und ausfallsicherer zu gestalten. Insbesondere soll analysiert werden, ob die technologische Komplexität von Workspaces durch den Einsatz von Event Sourcing reduziert werden kann.

Diese Arbeit richtet sich an erfahrene Softwareentwickler, welche die Kernkonzepte von TYPO3 CMS 7.6 verstanden haben und ebenso anwenden können.

[5] vgl. TYPO3 Code Review. https://review.typo3.org/#/q/topic:workspaces (04.10.2016)

2 Terminologien

Im Folgenden werden Entwurfsmuster für Software-Applikationen betrachtet, die für die eingangs definierten Ziele sinnvoll erscheinen. An dieser Stelle werden zunächst nur allgemeine Grundannahmen erläutert, eine Verknüpfung und Bewertung hinsichtlich der konkreten Herausforderungen mit TYPO3 sind im nachfolgenden Kapitel zu finden.

2.1 Domain-Driven Design

Hauptsächlich wurde der Begriff „Domain-Driven Design“ durch Eric Evans in seinem gleichbenannten Buch geprägt. Dort wird die Herangehensweise beschrieben, um sinnvolle Überführungen von realen Begebenheiten in verlässliche und erweiterbare Software-Projekte umsetzen zu können.

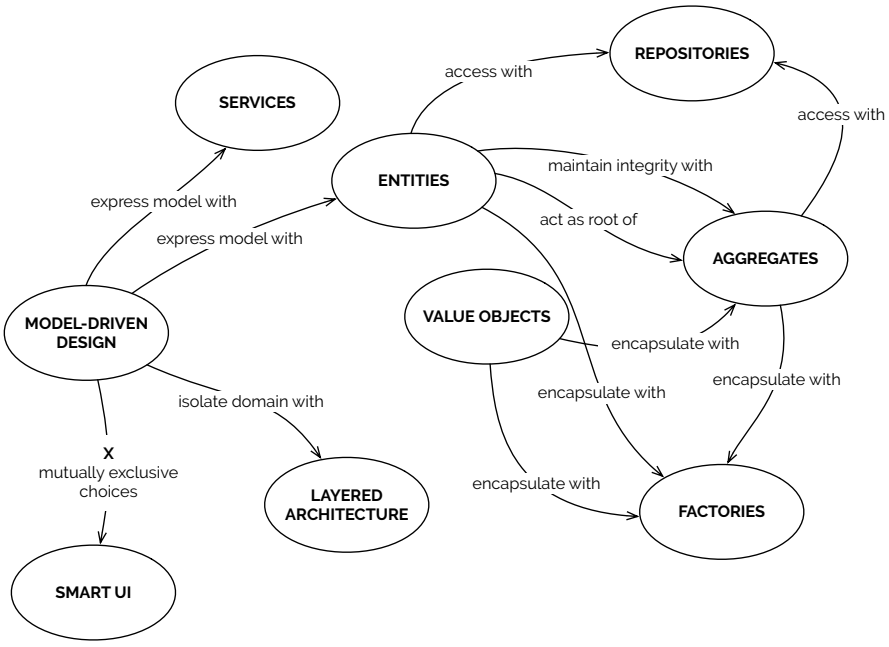

Abbildung 1: Model-Driven Design Übersicht (Evans, 2003, S. 66)

2.1.1 Ubiquitous Language

Das Hauptaugenmerk wird im Vorfeld auf die Vereinbarung auf eine einheitliche Sprache gelegt, der „Ubiquitous Language“. Diese soll dazu beitragen, dass alle Beteiligten, vom Anwender bis zum Softwareentwickler, einheitliche Charakteristiken hinsichtlich der Anwendung erarbeiten. Technische Zusammenhänge können dann mit

diesem gemeinsamen, einheitlichen Vokabular diskutiert und abgebildet werden (Evans, 2003, S. 23ff.).

Bei unterschiedlichen Anwendungskontexten mit ähnlichem Vokabular führt dies jedoch zu neuen Herausforderungen. Beispielsweise benötigt das „Abheben“ von Bargeld an einem Geldautomaten am Flughafen eine stärkere Abgrenzung zum „Abheben“ des Flugzeugs, um diese abweichende Bedeutung des gleichen Begriffs zu vermeiden (Nijhof, 2013, S. 77).

2.1.2 Architekturschichten

Um die Komplexität von Software-Projekten einzudämmen empfiehlt es sich, Komponenten in insgesamt vier Architekturschichten anzulegen – Benutzerschnittstelle, Anwendungslogik, Datenmodellierung und Infrastrukturschicht. Jede einzelne dieser Schichten steht für sich alleine und ist von den jeweils anderen losgelöst. Die Kommunikation und Interaktion erfolgt lediglich über definierte Schnittstellen (Evans, 2003, S. 68ff.).

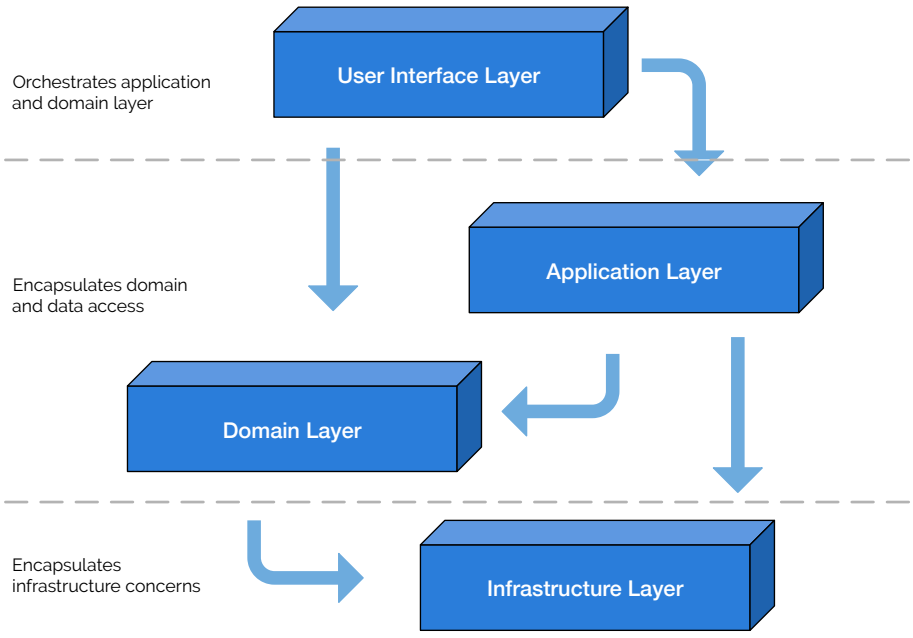

Abbildung 2: Architekturschichten (Buenosvinos, et al., 2016, S. 12)

Die Benutzerschnittstelle ist lediglich dafür verantwortlich, den Zustand des Systems an den Anwender bzw. einen anderen Automaten, sowie in der Umkehrung Eingaben und

Aktionen an die Anwendung zu senden. Die Applikationsschicht ist dagegen für alle ablaufrelevanten Anwendungsprozesse verantwortlich und bildet die übergeordnete Kontrollinstanz der gesamten Anwendung – spezifische Regeln finden sich allerdings in den Datenmodellen wieder, welche komplexe Zusammenhänge der Wirklichkeit in entsprechende Domain Models überführen. Innerhalb dieser Modelle werden somit auch die konkreten und relevanten Entscheidungen hinsichtlich des Verhaltens der Anwendung getroffen.

Allgemeine technische Aufgaben, in Bezug auf die Verarbeitung von Daten, wie beispielsweise die persistente Speicherung oder Weiterleitung an einen entfernten Web-Service, werden in einer separaten Infrastrukturschicht erledigt.

2.1.3 Entitäten & Wertobjekte

Als Entitäten (*Entities*) werden Objekte bezeichnet, die für sich eigenständig existieren und ihre Identität beibehalten, auch wenn sich Eigenschaften ändern – beispielsweise liegt weiterhin die gleiche Identität bei einer Person vor, auch wenn diese einen anderen Namen angenommen hat oder den Arbeitgeber gewechselt hat. Der referenzielle Charakter, also unterschiedliche Entitäten miteinander in Abhängigkeit setzen zu können, stellt in diesem Zusammenhang den wichtigsten Aspekt dar.

Wertobjekte (*Value Objects*) werden dagegen lediglich über ein oder mehrere Attribute definiert und haben somit keine individuelle Identität. Weiterhin werden diese als unveränderbar (*immutable*) modelliert und sind damit ohne ungewünschte Nebeneffekte verteilbar und wiederverwendbar. Geldbeträge oder Datumsangaben sind typischerweise Wertobjekte – die Validierung der Eigenschaften findet dabei direkt in den Modellen statt, bei einem Datum kann so beispielsweise sichergestellt werden, dass der „38. Oktember 2027,13“ keine gültige Kombination aus den benötigten Eigenschaften Tag, Monat und Jahr darstellt. Somit steht die Bedeutung der Eigenschaftswerte bei diesen Objekten im Vordergrund.

Die Entscheidung, Entitäten oder Wertobjekte einzusetzen, hängt jedoch stark vom vorliegenden Anwendungskontext ab – so würde bei den meisten Bestellportalen

Geldbeträge als Wertobjekt umgesetzt werden, bei einer Anwendung der Europäischen Zentralbank könnten in Kombination mit der Seriennummer eines Geldscheins jedoch Entitäten zum Einsatz kommen (Buenosvinos, et al., 2016, S. 45).

```
1. $date = \DateTimeImmutable::createFromFormat(
2.     \DateTime::W3C,
3.     '1981-11-27T10:04:00+01:00'
4. );
```

Programm 1: unveränderlicher Datumswert als Wertobjekt (eigene Darstellung)

2.1.4 Transferobjekte

Im Gegensatz zu Entitäten und Wertobjekten werden Transferobjekte (*Data Transfer Objects, DTO*) nur zur Objektkapselung von Eigenschaften und zum Austausch dieser Objekte verwendet. DTOs besitzen demnach keine fachspezifische Domänenlogik. Würde anstelle eines Transferobjekts eine Entität von der Benutzerschnittstelle empfangen und verarbeitet werden, bestünde das Risiko von unbeabsichtigten Änderungen im Domain Model hinsichtlich abhängiger Datenstrukturen (Buenosvinos, et al., 2016, S. 16). Natürlich müssen DTOs auch innerhalb der Applikation vor der eigentlichen Weiterverarbeitung dennoch auf Plausibilität und Gültigkeit der übertragenen Werte validiert werden.

2.1.5 Aggregate

Als Aggregate werden Entitäten bezeichnet, die wiederum andere zugehörige Entitäten und Wertobjekte zu einer Einheit zusammenfassen und diese hinsichtlich des Anwendungskontexts von anderen Objekten oder Aggregaten abgrenzen. Diese Zusammenfassung ist nicht mit einer losen Sammlung zu verwechseln, welche als wesentlich leichtgewichtiger und ohne bestimmte Anwendungslogik erkennbar wäre (Buenosvinos, et al., 2016, S. 210ff.). Innerhalb eines Aggregates kann ein sogenanntes Aggregate Root definiert werden, welches als Schnittstelle zwischen Elementen außerhalb des Aggregats zu den aggregierten Objekten darstellt – eine direkte Interaktion mit Objekten innerhalb des Aggregats ist bei diesem Entwurfsmuster nicht erlaubt und muss zwingend über das Aggregate Root durchgeführt werden. Es muss

jedoch nicht zwingend ein Aggregate Root vorhanden sein (Evans, 2003, S. 169-173). Assoziationen zwischen Objekten gestalten sich dabei als 1:1 Einzelbeziehungen, 1:n Eltern-Kind-Relationen und m:n Mehrfachrelationen.

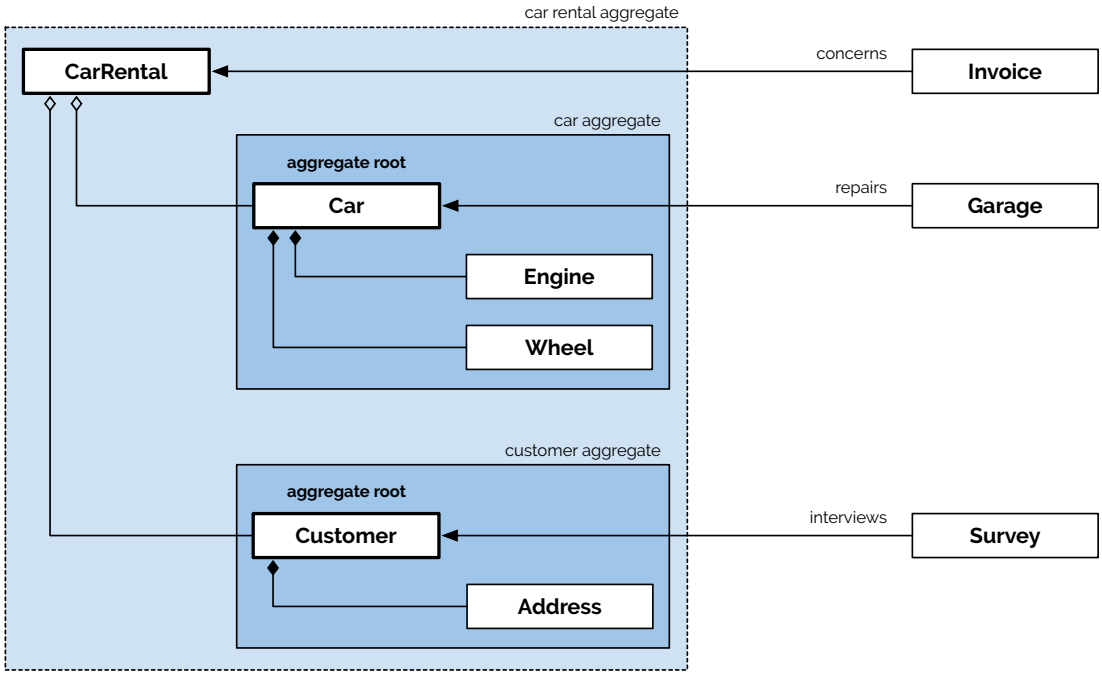

Abbildung 3: Aggregate versus Aggregate Root (eigene Darstellung)

Im gezeigten Beispiel wird die Abgrenzung der Begriffe „Aggregate“ und „Aggregate Root“ am Beispiel einer Autovermietung verdeutlicht. Autos besitzen in der Regel einen Motor und mindestens vier Räder. Die Verwendung dieser Komponenten würde ohne das entsprechende Fahrzeug wenig Sinn ergeben – deswegen stellt ein Auto (*Car*) das Aggregate Root in diesem Aggregat, bestehend aus Motor (*Engine*) und Rädern (*Wheels*). Der Kunde (*Customer*) bildet ebenfalls das Aggregate Root im Hinblick auf die Verwendung seiner Meldeadresse.

Die Vorgänge, welche für die Anmietung eines Autos stattfinden, sind in einem separaten Objekt modelliert, welche eine bestimmte Station zur Autovermietung (*Car Rental*) abbilden. Die beiden Entitäten, die des Autos und des Kunden, sind der Anmietstation direkt zugeordnet, welches somit zumindest ein Aggregat bildet. Würde das Car Rental Objekt zusätzlich noch ein Aggregate Root darstellen, wäre der Zugriff auf die beiden Unterelemente nur noch über diese übergeordnete Instanz erlaubt –

Reparaturarbeiten einer Werkstatt (*Garage*) und Kundenumfragen (*Survey*) müssten somit diesen Umweg über die Autovermietung in Kauf nehmen. Um den direkten Zugriff dennoch zu erlauben, ist Car Rental im Beispiel kein Aggregate Root.

2.1.6 Bounded Context

Eine Kontextgrenze (*Bounded Context*) bezieht sich auf Eigenschaften und Objekte eines eigenständig abgrenzbaren Bereichs der Anwendungsdomäne. Unterschiedliche Bereiche können eigene Begriffe entwickeln, jedoch auch Begriffe in ihrem spezifischen Kontext wiederverwenden, allerdings nur mit den relevanten Eigenschaften. Eine Anwendung besteht aus mindestens einem Bounded Context, der alle Objekte beinhaltet, falls es keinen weiteren Kontext gibt. Dieses Verfahren eignet sich dann, wenn Konzepte zu umfangreich werden, um alle Möglichkeiten darin abbilden zu können. Durch die Aufspaltung eines Objekts in unterschiedliche Bereiche, werden auch Abhängigkeiten zu anderen Komponenten eingegrenzt und reduziert (Evans, 2003, S. 328).

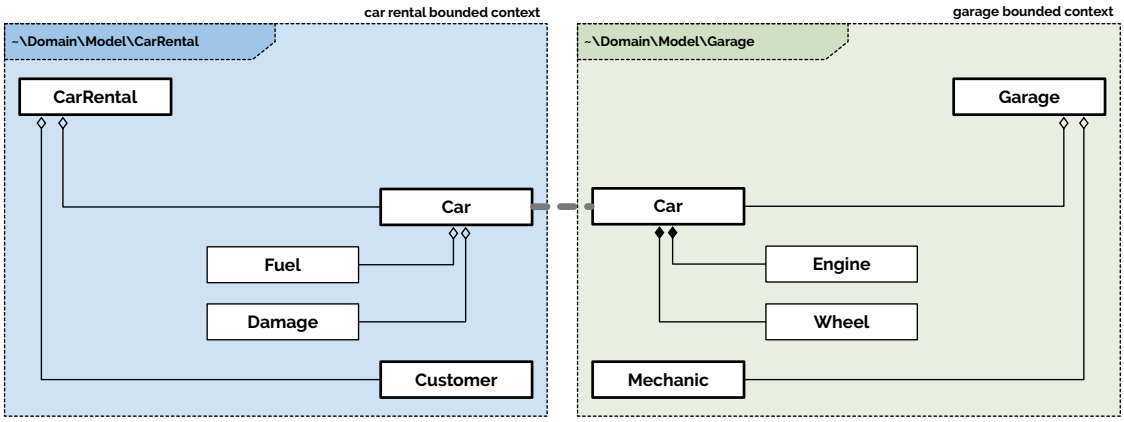

Abbildung 4: Aufteilung in separate Kontextgrenzen (eigene Darstellung)

Die gezeigte Abbildung bezieht sich erneut auf das Beispiel der Autovermietung aus dem vorherigen Abschnitt – bereits dort wurde gezeigt, dass ein bestimmtes Auto einerseits für die reine Vermietung vorhanden sein muss und andererseits der Reparaturservice regelmäßig Wartungsarbeiten am Fahrzeug durchführt. Durch die Aufteilung in zwei Anwendungsbereiche für die Anmietung und die Werkstatt werden die jeweiligen Eigenschaften eines Autos für die Bedürfnisse des jeweiligen Bereichs aufgeteilt – für

die Werkstatt ist es unerheblich, wer zuletzt mit dem Auto gefahren ist und für die Vermietung ist es nicht interessant, wann der Motor zuletzt geprüft wurde. In der Praxis ließe sich die Aufteilung zum Beispiel durch unterschiedliche Namensräume umsetzen.

Die Kommunikation zwischen unterschiedlichen Kontextgrenzen erfolgt über Ereignisse – somit gibt es keine festen Abhängigkeiten, jedoch die Möglichkeit auf Vorkommnisse aus einem anderen Bounded Context reagieren zu können (Betts, et al., 2013, S. 281).

2.1.7 Ereignisse

Anstatt in komplexen Anwendungen Zustände von Komponente zu Komponente weiterzureichen, werden dynamische Vorgänge über fachliche Ereignisse (*Domain Events*) abgebildet. Diese Ereignisse werden wiederum als Objekte modelliert, welche die relevanten Zustandsänderungen beschreiben und von beliebigen Prozessen verarbeitet werden können. Über das Beobachter-Muster (*Observer*) – konkret in den Ausprägungen „Publish-Subscribe" oder „Signal-Slot" – können auch nachträglich integrierte Erweiterungen auf bestehende Ereignisse reagieren (Evans, 2003, S. 72). Durch die optionale und zusätzliche Übermittlung an entfernte Systeme – beispielsweise durch den Einsatz einer Warteschlange (*Message Queue*) – kann die Prozessverarbeitung flexibel erweitert werden. Somit erfüllen Anwendungen die Anforderungen an Wartbarkeit und Skalierbarkeit.

2.1.8 Speicherung

Die persistente Speicherung des Zustands von Datenmodellen, samt aller Werte und Assoziationen, erfolgt über sogenannte Repositories. Diese sind in der Lage Domain Models zu serialisieren und abzuspeichern, umgekehrt werden aus gespeicherten Informationen wieder Entitäten und Wertobjekte abgeleitet (Fowler, 2002, S. 332ff.). Neben den notwendigen CRUD-Funktionalitäten finden sich in Repositories auch häufig spezifische Methoden zur Abfrage und Eingrenzung der Resultate. Die Speicherung ist dabei nicht an ein konkretes System gebunden. Über eine zusätzliche

Abstraktionsebene kann so zur Laufzeit beispielsweise in MySQL oder NoSQL[6] Datenbanken, wie auch direkt im Dateisystem persistiert werden.

Typischerweise existiert für jedes Aggregate Root der Anwendungsdomäne ein entsprechend ausgeprägtes Repository Objekt (Buenosvinos, et al., 2016, S. 262).

```
1. $car = new Car($engine, $wheels);
2. $repository = new CarRepository();
3. $repository->add($car);
```

Programm 2: Beispiel zur Speicherung eines neuen Autos (eigene Darstellung)

```
1. $car = $repository->findByEngineIdentificationNumber(
2.    new EngineIdentificationNumber('123456789')
3. );
4. $car->replaceEngine($newEngine);
5. $repository->update($car);
```

Programm 3: Beispiel zur Speicherung eines Motoraustauschs (eigene Darstellung)

Die beiden Programmbeispiele zeigen die Interaktionen eines Autos mit dem zugehörigen Repository. Änderungen innerhalb des Aggregats werden ausschließlich über das Aggregate Root durchgeführt – im Beispiel wird dies durch den Austausch des Motors veranschaulicht.

2.2 Objektrelationale Abbildung

Die objektrelationale Abbildung (*Object-Relational Mapping, ORM*) verbindet die Paradigmen der objektorientierten Modellierung mit der persistenten Speicherung in relationalen Datenbanken. Bei der objektorientierten Programmierung werden Informationen in Objekten vorgehalten, während Datenbanken Informationen in Tabellen und deren Datensätzen speichern. Über zusätzliche Abbildungsvorschriften, der Metadaten-Abbildung, ist die ORM-Schicht in der Lage einerseits Objekte in Datensätze zu überführen und andererseits wiederum Entitäten oder Wertobjekte aus Informationen der Datenbank zu erzeugen. In der Regel wird das Verfahren über

[6] NoSQL steht für „Not only SQL" und geht somit über die schemaabhängige Speicherung hinaus

Repositories umgesetzt, welche auf einer unterliegenden Anwendungsschicht entweder direkt auf die Datenbank zugreifen, oder dies optional über zusätzliche ORM-Techniken für den konkreten Datenzugriff abstrahieren (Fowler, 2002, S. 278ff., 306ff, 322ff.).

2.3 Command Query Responsibility Segregation

2.3.1 Command Query Separation

Der Begriff „Command Query Separation“ (*CQS*) wurde durch Bertrand Meyer während den Arbeiten an seiner objektorientierten Programmiersprache „Eiffel“ begründet. Das Prinzip basiert auf einfachen Maßnahmen und zielt darauf ab, Objekte hinsichtlich der Methoden in zwei Anwendungsbereiche aufzuteilen. Einerseits gibt es Abfragen (*Queries*), welche lediglich den Zustand eines Datenmodels liefern, ohne dabei Änderungen vorzunehmen – man spricht von effektfreier Verarbeitung. Andererseits werden Befehle (*Commands*) verwendet, welche ausschließlich den Systemzustand von Objekten abändern, dabei jedoch aber keine Resultate zurückliefern. Häufig werden auch die Begriffe „Modifier” oder „Mutator” für Befehle verwendet (Meyer, 2012, S. 22; Meyer, 1997).

Da eine Objektmethode für nur jeweils einen dieser Anwendungsbereiche implementiert wird, lässt sich die Komplexität hinsichtlich Nebeneffekten insgesamt reduzieren. Es gibt jedoch auch Anwendungsfälle, bei denen diese Aufteilung nicht strikt eingehalten werden kann – beispielsweise bei der Abfrage des obersten Elements eines Stapelspeichers[7] (Fowler, 2005).

2.3.2 Command Query Responsibility Segregation

Greg Young verfolgt einen noch strikteren Ansatz, indem er die durch Evans aufgestellten Paradigmen des Domain-Driven Designs mit Meyers Gedanken zu CQS kombiniert und daraus Bedingungen für die komplette Architektur eines Software-Systems ableitet. Somit unterteilt sich die Anwendungsdomäne in den modifizierenden

[7] bei der Anwendung des Pop-Befehls auf einen Stapelspeicher (*Stack*) wird das oberste Element entfernt

Zugriff (*Write Model & Commands*) und den abfragenden Zugriff (*Read Model & Queries*). Die Seite, auf der Befehle verarbeitet werden, entspricht weitgehend den bekannten Domänenmodellen, samt Attributen, Verhaltensweisen und Regelwerken. Der Lesezugriff ist dagegen für das tatsächliche Einsatzgebiet individualisiert – daraus resultiert, dass bei Abfragen Objekte zum Einsatz kommen, welche auf das notwendige Maß an Informationsgehalt für den jeweiligen Anwendungskontext spezialisiert und reduziert sind (Nijhof, 2013, S. 5).

Die Informationen des Write Models werden dabei ebenfalls abgetrennt vom Read Model gespeichert. Durch Projektionen (siehe Abschnitt 2.5) werden Änderungen des schreibenden Zugriffs für die konkreten Anforderungen des Lesezugriffs bereitgestellt. Dadurch ergeben sich neue Möglichkeiten hinsichtlich der Skalierbarkeit – so könnte der rein darstellende Bereich einer Website auf mehrere Systeme verteilt werden, um so einem möglichen Ansturm an Besuchern Stand zu halten.

Die folgende Abbildung visualisiert den Kreislauf im Zusammenspiel von Read Model, Write Model, Speicherung und der Interaktion mit einer Benutzerschnittstelle.

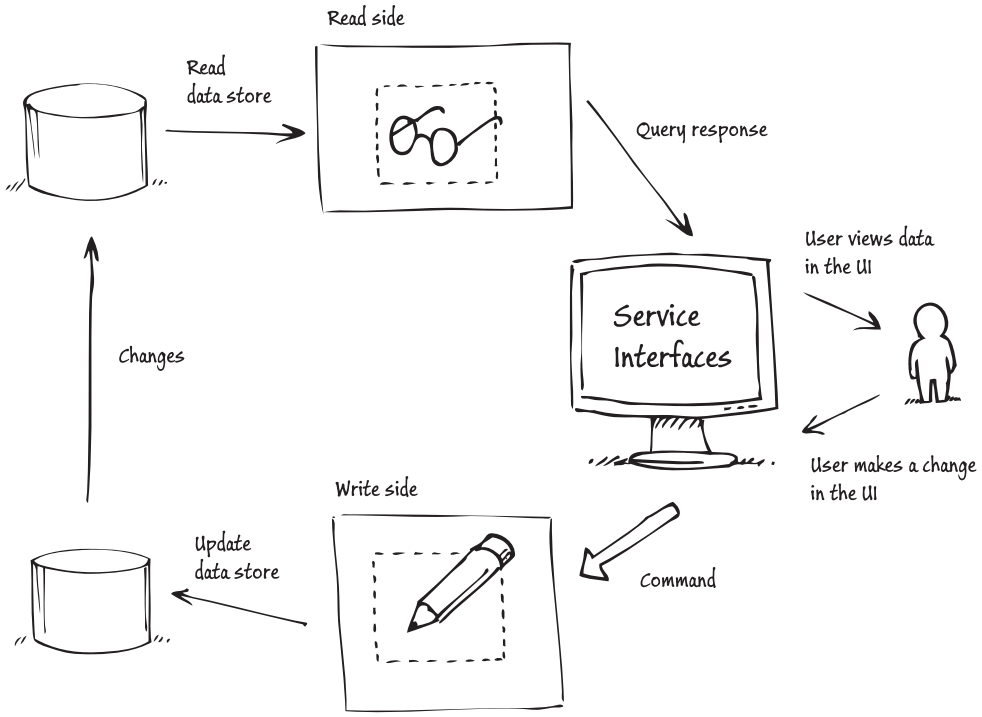

Abbildung 5: Anwendungsbeispiel des CQRS Paradigmas (Betts, et al., 2013, S. 225)

2.3.3 Commands

In diesem Zusammenhang werden Befehle als Nachrichten an den Domain Layer der Anwendung zur weiteren Verarbeitung betrachtet. Dabei werden Befehle (*Commands*) nicht direkt an bestimmte Objekte übermittelt, sondern über einen Command Bus vom konkreten Empfänger gelöst – somit ist die Verarbeitung lose gekoppelt und kann durch eine alternative Implementierung ausgetauscht werden.

Befehle sind jeweils für genau einen Zweck als Objekt implementiert, die relevanten Informationen für Domänenobjekte werden als Eigenschaften innerhalb dieser Instanzen transportiert.

Befehle beinhalten eine bestimmte Absicht, die ebenso verstanden und ausgeführt werden sollen. In der Namensgebung rücken technische Aspekte in den Hintergrund – anstatt also bei einer Reparaturwerkstatt „den Motor zu definieren“ (*SetEngineCommand*), sollte in diesem Beispiel besser „ein defekter Motor durch einen neuen getauscht werden“ (*ReplaceBrokenEngineCommand*).

2.4 Event Sourcing

Das Analysemuster „Event Sourcing“ hat in der ursprünglichen Definition (Fowler, 2005) lediglich den Stellenwert eines alternativen Ansatzes zum Aufzeichnen und Abspielen von Ereignissen innerhalb des Domain Models. Wenn Anwender Aktionen ausführen, beispielsweise zum Erzeugen oder Abändern von Daten, werden diese im regulären CRUD-Ansatz als direktes Resultat abgespeichert. Unterschiedliche und mehrmalige Modifikationen lassen sich aus diesem Zustand nicht mehr ableiten – Zwischenschritte müssten separat erzeugt und abgespeichert werden. Event Sourcing kehrt den Prozessfluss um und stellt nicht mehr die in der Datenbank gespeicherten Ergebnisse in den Mittelpunkt, sondern verschiebt diesen Fokus auf die ursprünglich stattgefundenen Ereignisse. Die spätere korrekte und sequenzielle Anwendung dieser Vorkommnisse ergibt wieder den eigentlichen Zustand.

Greg Young verfeinerte diese Idee unter dem gleichzeitigen Einsatz des CQRS-Musters und konkretisierte die daraus resultierenden Prozesse und Implikationen für Software-

Applikationen (Young, 2016, S. 50ff.). Das Verfahren von Ereignisanwendung und Speicherung wird auch als Projektion bezeichnet und kann individuell für den jeweiligen Anwendungsbereich definiert werden (siehe Abschnitt 2.5).

2.4.1 Event Store

Der Event Store bildet das Herzstück des Event Sourcing Paradigmas, hier werden alle stattgefundenen Ereignisse in serialisierter Form persistent gespeichert. Da diese Ereignisse die einzig verlässliche Quelle für den Zustand von Objekten des Domänenmodells zu einem bestimmten Zeitpunkt darstellen, muss sichergestellt werden, dass Ereignisse verlässlich, dauerhaft und unveränderbar vorgehalten werden können. Die tatsächliche Art der Speicherung ist dabei austauschbar.

Es gibt einige Mindestanforderungen hinsichtlich der Informationen, die in einem Event Store abgespeichert werden müssen – Erweiterungen und Konkretisierungen sind natürlich nach Anwendungskontext möglich (Verraes, 2014, S. Folie 50).

- Stream Name: Rückschluss auf das ursprüngliche Aggregat des Ereignisses
- Zeitpunkt: Zeitpunkt des Ereignisses, meist mit Mikrosekunden und Zeitzone
- Event ID: eindeutige Identifikation des Ereignisses, meist als UUID
- Event Typ: Rückschluss auf die Art des eingetretenen Ereignisses
- Revision: Version des Ereignisses in Hinblick auf das ursprüngliche Aggregat
- Daten: serialisierte Form der relevanten Attribute des Ereignisses
- Metadaten: optionale periphere Informationen, die zum Ereignis geführt haben
- Kategorien: optionale Kennzeichnung von Gültigkeitsbereichen eines Ereignisses

Die tatsächliche Speicherung dieser Informationen kann in mehreren geeigneten Systemen stattfinden – so ist der Einsatz von relationalen und dokumentenorientierten Datenbanken, Key-Value-Stores oder auch des Dateisystems möglich.

Bei der individuellen Implementierung eines Event Stores muss allerdings sichergestellt werden, dass Ereignisse in der korrekten Reihenfolge gespeichert und gelesen werden können – der Umgang mit konkurrierenden Schreibvorgängen führt dabei zu neuen

konzeptionellen Herausforderungen in einer Anwendung. Aus diesem Grund gibt es spezielle Event Stores, die diesen Anforderungen nachkommen.

Das Projekt „GetEventStore“, entwickelt mit der Beteiligung von Greg Young, kommt beispielsweise den Anforderungen an einen Event Store nach. Die Interaktion erfolgt über eine RESTful HTTP API, einzelne Versionen eines Event Streams sind jeweils über eine eigene URL adressierbar. Weiterhin unterstützt GetEventStore die spezifische Auswahl von Ereignissen über bestimmte Kategorienamen (Event Store LLP, 2016).

2.4.2 Event Stream

Event Streams stellen innerhalb der Applikation eine generische Komponente bereit, um auf Ereignisse eines Aggregats in der korrekten Reihenfolge unter Einsatz eines Iterators zugreifen zu können. Über diesen Iterator werden Events chronologisch vollständig oder bis zu einem gewissen Zeitpunkt auf das eigentliche Domain Model angewandt. Da sich im Laufe der Zeit viele Ereignisse ansammeln, ist es ratsam, einzelne Events zu einer Schattenkopie (*Snapshot*) zusammenzufassen. Ein Snapshot, welcher den Zustand des kompletten Aggregates speichert, kann beispielsweise nach jeweils tausend Ereignissen angelegt werden.

2.4.3 Herausforderungen

Auch wenn die Vorteile des Einsatzes von Event Sourcing auf den ersten Blick einleuchtend wirken, verbergen sich dahinter auch einige Herausforderungen und Risiken. Für den Fall, dass das Abspielen von Ereignissen bei den betroffenen Aggregaten Benachrichtigungen an externe Systeme auslöst – zum Beispiel den Versand einer E-Mail aufgrund einer neu angelegten Bestellung – würden unerwünschte Begebenheiten auftreten. In diesem konkreten Fall wurden Kunden bereits in der Vergangenheit informiert und würden sich wundern, Tage oder Monate später erneut informiert zu werden. Hier wird empfohlen, diese Art von Effekten zunächst zu sammeln und erst nachdem alle Ereignisse verarbeitet wurden entsprechende Handlungen durchzuführen – am Ende kann auch entschieden werden, ob dies im jeweiligen Anwendungs- und Ereigniskontext tatsächlich relevant ist oder ignoriert werden kann (Nijhof, 2013, S. 21).

Eine weitere Herausforderung stellt die Abhängigkeit von externen Datenquellen dar. Werden bei Abrechnungssystemen unterschiedliche Währungen verwendet, so müssen diese vorher anhand des aktuellen Umrechnungskurses aufbereitet werden. Beim nachträglichen Ausführen der Ereigniskette hat sich der Wechselkurs wahrscheinlich verändert und würde zu falschen Berechnungen führen – entweder zum Nachteil des Anbieters oder des Kunden. Deshalb ist es notwendig, externe Informationen zum Zeitpunkt des Ereignisses in das Datenmodell zu integrieren, auch wenn sich die eigentliche Quelle außerhalb der Anwendungsdomäne befindet (Fowler, 2005).

2.5 Materialized View

Der Begriff „Materialized View" stammt ursprünglich aus dem Bereich der Datenbanksysteme (Schneede, 2011). Um die Abfragegeschwindigkeit zu steigern werden benötigte Kombinationen unterschiedlicher Aggregate in einer separaten, denormalisierten Datensammlung zusammengefasst. Die Ermittlung und Bereitstellung dieser Informationen wird dabei bereits nach dem Eintreten einer Modifikation während des schreibenden Zugriffs und nicht erst bei der Abfrage durchgeführt.

Rückblickend auf das Beispiel der Autovermietung (siehe Abschnitt 2.1.5, Abbildung 3) kann somit eine Übersicht erzeugt werden, welcher Kunde ein bestimmtes Fahrzeug gemietet hat und ob die Ausleihgebühr bereits bezahlt wurde, ohne dabei bei jeder Aktualisierung in der Benutzeransicht die beteiligten Aggregate auflösen zu müssen.

Im Gegensatz zu einer reinen Eingrenzung auf Datenbankebene, werden Materialized Views innerhalb des Domain Models bereits von der Anwendung selbst erzeugt. In den folgenden Abhandlungen und auch der späteren Implementierung ist der Begriff Projektion prinzipiell gleichbedeutend mit dieser Verfahrensweise.

2.6 Eventual Consistency

Bei der Verwendung von CQRS und Event Sourcing ist es essentiell, dass Ereignisse verlässlich gespeichert werden können, schließlich bilden diese die Grundlage für die Projektionen, welche wiederum in der Benutzerschicht dargestellt werden. Es bleibt

jedoch die Frage, zu welchem Zeitpunkt die Read Models auf allen beteiligten Subsystemen aktualisiert sein müssen.

Durch Eric Brewer wurde das CAP-Theorem aufgestellt, welches die Ziele bei verteilten Datenbanksystemen in die drei Bereiche Konsistenz (*Consistency*), Verfügbarkeit (*Availability*) und Partitionstoleranz (*Partition tolerance*) auf Netzwerkebene einordnet. Es ist jeweils immer nur möglich, maximal zwei dieser Aspekte einzuhalten, niemals aber alle gleichzeitig (Newman, 2015, S. 232-236).

Bei der Projektion von Ereignissen ergibt sich auf Applikationsebene die gleiche Problemstellung. Im Folgenden werden die unterschiedlichen Aspekte erklärt und gegeneinander abgewogen.

2.6.1 Vernachlässigung der Verfügbarkeit

Bei Vernachlässigung der Verfügbarkeit wären zwar alle Informationen konsistent und das System wäre auch bei Störungen oder Verzögerungen im Datenaustausch funktionsfähig. Allerdings könnte eben der Fall eintreten, dass Systeme nicht verfügbar sind – daraus würde ebenfalls resultieren, dass kein Datenaustausch mehr mit anderen Systemen durchgeführt wird und die Konsistenzbedingung nicht erfüllt werden kann.

Banken nehmen es in Kauf, dass der Geldautomat oder Auszugsdrucker nicht einsatzbereit ist, dafür aber Kontobuchungen intern konsistent und zeitnah verarbeitet werden.

2.6.2 Vernachlässigung der Partitionstoleranz

Hier stehen Verfügbarkeit und Konsistenz im Vordergrund. Verzögerungen des Datenabgleichs werden jedoch in Kauf genommen. Im Internet-Zeitalter kann das Vorliegen von Netzwerkverzögerungen allerdings als gegeben angesehen werden.

MySQL Master-Slave-Cluster arbeiten nach diesem Prinzip – die Replikation zwischen Deutschland und Hongkong kann dabei stark auseinanderdriften.

2.6.3 Vernachlässigung der Konsistenz

Vernachlässigt man jedoch die Konsistenz, so wären Systeme vorranging verfügbar und Partitionen können ausgeglichen werden. Die fehlende Konsistenz bedeutet konkret, dass möglicherweise veraltete Daten geliefert werden – jedoch soll diese Inkonsistenz in der kürzest möglichen Zeit wieder beseitigt werden. Dies bedeutet, dass natürlich ein gewisser Grad an Konsistenz beabsichtigt ist, jedoch mit leichten Einschränkungen – dies stellt auch die konzeptionelle Grundlage für Eventual Consistency dar (Vogels, 2008).

Beispielsweise beruhen Gruppenchats bei Nachrichtendiensten wie Facebook oder WhatsApp auf diesem Prinzip. Dafür ist es vernachlässigbar, ob alle Teilnehmer wirklich gleichzeitig die gleiche Nachricht zugestellt bekommen, oder mit Verzögerung.

2.7 Event Storming

Diese Bezeichnung „Event Storming“ lässt einen Zusammenhang mit Event Sourcing vermuten – dies ist korrekt, jedoch handelt es sich um einen interaktiven Team-Prozess, anstatt um ein weiteres Entwurfsmuster. Somit wird diese Methode eher der Ausgangsanforderung des Domain-Driven Designs gerecht, unter Experten der Anwendungsdomäne und Entwicklern gemeinsames Wissen anzuhäufen und für diese Vorgänge eine einheitliche Sprache, sowie ein gemeinsames Verständnis zu entwickeln.

Der Ablauf ist im Entwicklerteam mit einfachen Mitteln umzusetzen, erfordert aber auch ein gewisses Maß an Disziplin und Moderation. Mittels farbiger Haftnotizen visualisieren die Teilnehmer Abläufe und Abhängigkeiten eines Softwareprozesses.

Zu Beginn werden die zu erwartenden Ereignisse (*Events*) dargestellt. Es empfiehlt sich dabei, chronologisch rückwärts zu arbeiten, da es wichtiger ist, den konkreten Endzustand zu benennen, als einen beliebigen Anfangszustand zu wählen. Zwischen den einzelnen Events werden im nächsten Schritt, jeweils mit unterschiedlichen Farben, Befehle (*Commands*) und Bedingungen bzw. Regeln (*Conditions & Rules*) gesetzt. Daraus ergeben sich die Anforderungen an die Benutzerschnittstelle, über welche üblicherweise Eingaben getätigt werden. Weiterhin wird es so ermöglicht, spezifische und genaue Prozesse zu definieren (Verraes, 2013).

Abbildung 6: Event Storming mit Mathias Verraes und TYPO3 Entwicklern (eigene Darstellung)

Im nachfolgenden Beispiel wird der Prozess einer Geldauszahlung eines Bankkontos unter dem Einsatz von Event Storming veranschaulicht. Bevor das Ereignis (gelb) „Geld ausgezahlt“ stattfinden kann, muss ein entsprechender Befehl[8] (blau) geäußert werden. Natürlich hängt der Erfolg dieser Transaktion von einigen Bedingungen (rot) ab – so muss das Bankkonto tatsächlich existieren und darf nicht etwa gekündigt worden sein, weiterhin muss der auszuzahlende Geldbetrag auch tatsächlich verfügbar sein, da sonst das Konto einen Negativbetrag aufweisen würde.

Abbildung 7: Bankauszahlung mittels Event Storming (eigene Darstellung)

Änderungen während dieser Phase sind vergleichsweise kostengünstig - Ablaufprozesse können durch das Erstellen, Umhängen oder Vernichten von Haftnotizen einfach angepasst werden. Zu einem späteren Zeitpunkt, während der Implementierung oder Integration in Zielsysteme, sind Änderungen vergleichsweise gravierender zu bewerten.

[8] in der Realität würde aus Gründen der Höflichkeit wohl eher ein Wunsch geäußert werden

Event Storming funktioniert jedoch nur bei aktiver Beteiligung aller Teammitglieder und eines stattfindenden Kommunikations- und Normierungsprozesses – ein neutraler Moderator oder Katalysator kann hier ebenfalls unterstützend einwirken.

2.8 Zusammenfassung

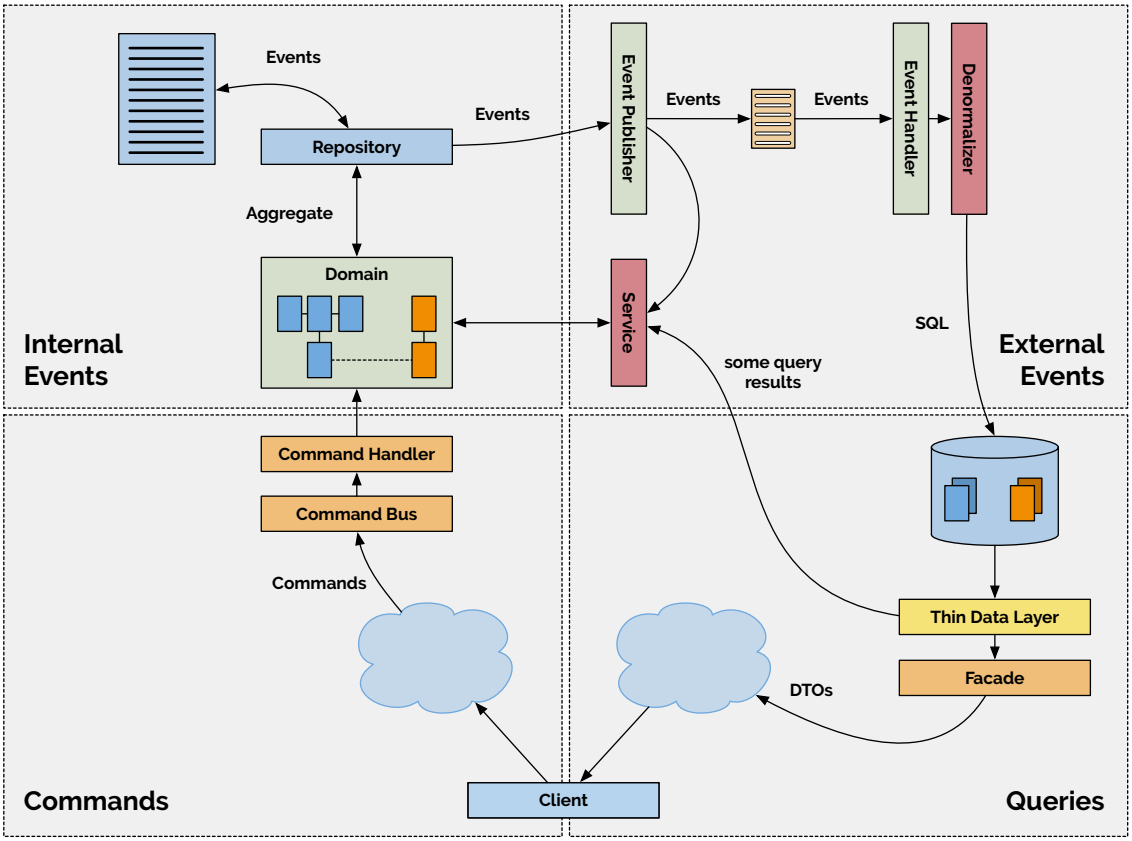

Abbildung 8: CQRS à la Greg Young, modifizierte Darstellung (Nijhof, 2013, S. 6)

Die Abbildung zeigt eine grafische Übersicht der in diesem Kapitel genannten Komponenten, deren Abgrenzung in separate Schichten, sowie den gerichteten Verlauf von Nachrichten im Gesamtsystem – von der Benutzerschnittstelle, über die Interpretation von Befehlen, der Verarbeitung von internen und externen Ereignissen, zurück zur Benutzerschnittstelle. Dieser Ablauf wird in Kapitel 4 wieder aufgegriffen und für die Anwendung in TYPO3 konkretisiert.

3 Analyse

3.1 Architektur

An dieser Stelle werden vorhandene Konstrukte innerhalb von TYPO3 ermittelt, hinsichtlich ihrer Aufgaben analysiert und mit den Architekturschichten des Domain-Driven Designs verglichen.

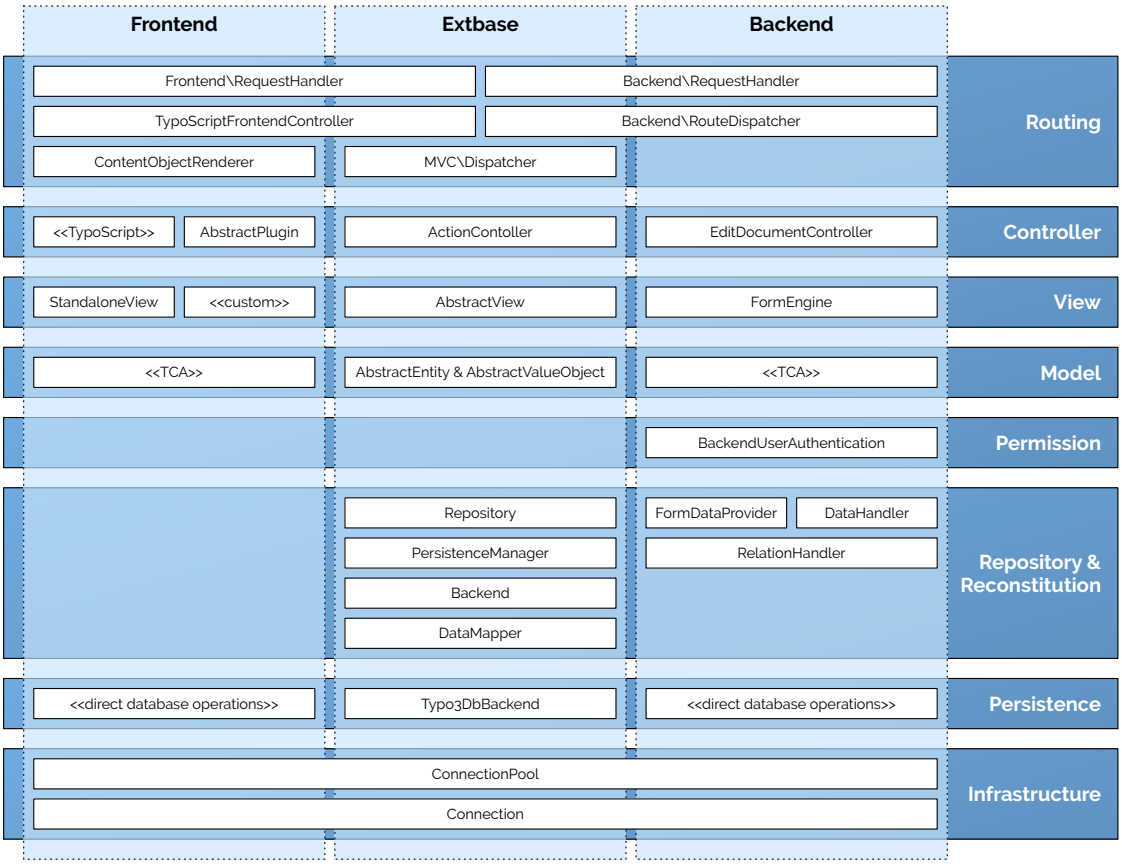

Abbildung 9: TYPO3 Architekturübersicht (eigene Darstellung)

3.1.1 Frontend

Der Haupteinstiegspunkt im Frontend trägt zwar den Beinamen Controller, ist jedoch nicht als solcher im klassischen MVC-Ansatz zu verstehen. Viel mehr werden hier TypoScript Konfigurationen für die Kombination aus Seite, Sprachparameter und Domainnamen geladen und ausgeführt.

Die eigentliche Ablaufsteuerung ist somit entweder generisch über TypoScript definiert oder kann in individuellen Plug-ins[9] abgearbeitet werden. Alternativ kann auch auf das MVC-Framework Extbase verzweigt werden (siehe Abschnitt 3.1.3). Bei der ausschließlichen Umsetzung mittels TypoScript kann nur lesend auf Daten zugegriffen werden – Konstrukte zum Erzeugen, Modifizieren oder Löschen von Daten sind hier nicht vorhanden.

Für die Repräsentation der Ergebnisse kommt entweder eine spezielle Variante der Fluid Templating Engine zum Einsatz oder aber auch ganz rudimentäres Markup, welches innerhalb der Applikation erstellt wird – eine Schichtentrennung zwischen Controller und View muss also nicht zwingend vorhanden sein.

Konkrete Domänenmodelle gibt es nicht wirklich – die Basis zum Ermitteln von Informationen stellt TCA bereit, als eine Art generische Metadaten-Modell. Lese- und Schreibvorgänge werden direkt über mehr oder weniger abstrahierte SQL-Abfragen[10] an die Datenbank weitergegeben.

Im Frontend gibt es keine Schicht, die spezifisch Zugriffsberechtigungen steuert. Zwar können Inhalte auf Seiten- oder Elementebene für eine bestimmte Benutzergruppe ein- oder ausgeblendet werden – eine Steuerung hinsichtlich der Eigenschafen liegt aber nicht vor. Eigenständig entwickelte Plug-ins können somit beliebige Daten, ohne jegliche Prüfung, lesen und auch modifizieren.

3.1.2 Backend

In der Verwaltungsansicht von TYPO3, dem Backend, werden Kontrollinstanzen direkt ausgeführt. In der gezeigten Übersicht steht dabei das Datenmanagement auf der Ebene von Inhaltselementen im Vordergrund – natürlich gibt es noch andere Anwendungsfälle und Programmsteuerungen, die hier jedoch nicht näher betrachtet werden. Durch das

[9] leitet zwar von der Oberklasse *Abstract Plugin* ab, ist aber in der internen Ablaufsteuerung flexibel

[10] Doctrine DBAL wird erst mit TYPO3 CMS 8 eingeführt, Anweisungen können bereits als Abstraktion über den *Query Builder* vorliegen oder, wie bei den meisten Erweiterungen von Drittanbietern, noch in reinem SQL hinterlegt sein

vorausgehende Routing, einer Umleitung an konkrete Software-Komponenten auf der Basis von URL-Parametern, können alternativ Implementierungen auf der Basis von Extbase ausgeführt werden – diese Verzweigung ist im Frontend-Kontext in ähnlicher Form vorhanden.

Für den vorliegenden Anwendungsfall wurde die Formularverarbeitung (*Form Engine*) im Sinne einer Präsentationsschicht interpretiert. Darzustellende Daten werden innerhalb dieses Moduls gesammelt und hierarchisch angeordnet und in einem separaten Schritt ausgegeben. Auch wenn an dieser Stelle keine strikte Trennung zwischen Steuerung und Ausgabe vorliegt, sind diese beiden Aspekte bereits bis zu einem gewissen Grad voneinander getrennt (*Separation of Concerns*).

Ebenso wie im Frontend-Bereich, gibt es keine konkrete Datenmodellierung – die generische Grundlage für Modelle wird hier ebenfalls durch TCA definiert. Ermittelte Daten werden dann über Array-Datentypen weitergereicht. Da es sich dabei um flache Hierarchien handelt, müssen Assoziationen separat für den jeweiligen Anwendungskontext (Workspaces und Lokalisierungen, siehe Abschnitt 3.3) ermittelt werden.

Für die Speicherung ist das Objekt *Data Handler* verantwortlich. Diese Komponente wird mit Array-Daten aufgerufen, welche wiederum auf der Ebene von Inhaltselementen keine weiteren Hierarchieinformationen besitzen. Assoziationen werden über die zusätzliche Komponente *Relation Handler* somit erneut angepasst und innerhalb des Objekts wieder direkt gespeichert. Da in dieser Schicht ausschließlich mit vollständigen Wertemengen verfahren wird, müssen die tatsächlich geänderten Werte und Relationen erst aufwendig ermittelt und für den jeweiligen Zielkontext erneut interpretiert werden.

Die konsequente Verwendung von Doctrine DBAL im Backend sorgt dafür, dass zumindest die Art der Speicherung austauschbar ist – jedoch werden die Anweisungen dennoch unmittelbar persistiert. Ein gesamter Blick auf mögliche Datenmodelle und Hierarchien wird im *Data Handler* nicht ermöglicht.

3.1.3 Extbase

Extbase nimmt als MVC-Framework innerhalb von TYPO3 eine besondere Rolle ein, welches sowohl für Frontend- als auch Backend-Anwendungen genutzt werden kann. Extbase wurde 2009 teilweise als Backport des PHP-Frameworks „TYPO3 Flow“ in TYPO3 CMS integriert. Der Fokus lag dabei jedoch weitgehend auf der MVC-Schicht, weitere Konzepte von TYPO3 Flow wie beispielsweise aspektorientierte Programmierung (*AOP*) wurden nicht übernommen. Somit sind die Konzepte des Domain-Driven Designs in Extbase zwar ähnlich zu TYPO3 Flow, allerdings in dieser Hinsicht nicht vollständig.

Plug-ins und Module leiten dazu jeweils von der abstrakten Klasse *Action Controller* ab – Anweisungen der Benutzerschicht können somit nachvollziehbarer in separate Aktionen (*Actions*) aufgeteilt werden.

Da Extbase unter Verwendung der Paradigmen des Domain-Driven Designs entwickelt wurde, existieren Domänenmodelle, welche den Zugriff auf Datenbankinhalte über eine interne ORM-Schicht kapseln. Anwendungslogik wird jedoch meist im Controller selbst ausgeführt oder in Service-Klassen ausgelagert.

Nach interner Aufbereitung werden die von Repositories bezogenen und in eigenständige Entitäten oder Wertobjekte überführten Daten an die Präsentationsschicht weitergereicht – meist wird diese durch eine Sammlung von Fluid Templates umgesetzt. Individuelle Formatierungen, wie beispielsweise ein Datumsobjekt in eine lesbare Tag-Monat-Jahr Angabe zu überführen, werden über sogenannte *View Helper* erreicht (Rau, et al., 2010, S. 188).

Die Transformation von Datenbankinhalten zu Domänenobjekten wird über eine weitgehend automatisierte Feldzuordnung[11] realisiert. Über eine Abstraktionsschicht wird das vorhandene TCA einer Datenbanktabelle in ein Meta-Model überführt – daraus lassen sich neben Attributen auch Assoziationen und deren Kardinalität ableiten.

[11] durch Namenskonvention; das Datenbankfeld *first_value* wird in das Attribut *firstValue* überführt

Eine Schicht, welche den Datenzugriff für Benutzer steuert und reguliert, wird jedoch auch hier nicht durch das Framework bereitgestellt. Die Abfrage von nicht autorisierten Informationen, sowie das Abändern von Attributen müssen somit individuell in der Anwendungslogik umgesetzt werden. Das ursprünglich für Extbase herangezogene Projekt TYPO3 Flow implementiert im Gegensatz dazu mittels aspektorientierter Programmierung ein eigenes Security-Framework, welches den Anforderungen einer Zugriffsbeschränkung nachkommen würde.

Die tatsächliche Speicherung von Assoziationen stellt in Extbase eine eigene Implementierung dar und unterscheidet sich somit von den Speichervorgängen des im TYPO3 Backend eingesetzten *Data Handlers*. Zum größten Teil ist das Speicherverhalten ähnlich, bei der Interpretation von 1:n Relationen gibt es jedoch einige Abweichungen zur ursprünglichen Intention dieser Speicherform (Hader, 2007).

Um Änderungen an Objekten feststellen und bei Bedarf wieder abspeichern zu können, verwendet Extbase das Arbeitspaket-Entwurfsmuster (*Unit of Work*) (Fowler, 2002, S. 184). Entitäten, die über ein Repository bezogen wurden, werden über den Persistence Manager für die komplette Laufzeit der HTTP-Anfrage im Speicher gehalten, da der Persistence Manager das Singleton-Muster implementiert. Der Zustand des Arbeitspakets umfasst dabei neue, zu löschende, sowie modifizierte und nicht modifizierte Objekte. In den ersten Extbase Versionen wurden Zustandsänderungen implizit und automatisch am Ende des HTTP-Prozesses gespeichert, inzwischen muss der Speichervorgang explizit angestoßen werden. Bei einer großen Anzahl an zu verarbeitenden Objekten stellt der Einsatz dieses Entwurfsmusters gesteigerte Ansprüche an den zugeteilten Arbeitsspeicher.

3.2 Datenmodelle

3.2.1 Metadaten

Für alle Anwendungsfälle innerhalb TYPO3 bildet das Table Configuration Array (*TCA*) individuell für jede Datenbanktabelle die Grundlage für Informationen über vorhandene Attribute und Relationen zu anderen in der Datenbank abgelegten Tabellen.

Die Interpretation dieser Konfiguration wird dabei meistens individuell bei der Verwendung vorgenommen – eine Kapselung in einen globalen Metadaten-Service gibt es derzeit noch nicht. Daraus resultieren abweichende oder unvollständige Interpretationen, welche letztendlich, je nach Anwendungskontext, zu unterschiedlichen Verhaltensweisen führen kann.

Mittels TCA wird auch definiert, welche Werte ein Feld annehmen kann, so kann beispielsweise zwischen numerischen, alphanumerischen Werten oder Datumswerten variiert werden.

3.2.2 Relationen

Prinzipiell unterstützt TYPO3 drei Arten von Relationstypen – 1:1, 1:n und m:n Relationen. Die ersten beiden Relationstypen können dabei entweder serialisiert am Aggregat oder in normalisierter Form am aggregierten Gegenstück gespeichert werden. 1:n und m:n Relationen können zusätzlich noch die Speicherung in einer Zwischentabelle verwenden.

Nur bei Relationen der Kardinalität m:n ist es möglich, durch eine separate Einstellung, echte bidirektionale asymmetrische Relationen zu erzeugen – bei den verbleibenden Szenarien kennt jeweils immer nur eine Seite der Relation das Gegenstück. Um bei Bedarf aufwendige Ermittlungen zu reduzieren wird innerhalb von TYPO3 eine separate Datenbanktabelle geführt, welche Zuordnungen in beide Richtungen vorhält, der sogenannte *Reference Index* (siehe auch Abschnitt 3.4.5).

Das folgende Beispiel veranschaulicht die resultierende Speicherung in den Datenbanktabellen für 1:1 bzw. 1:n Relationen als serialisierte oder normalisierte Form der Speicherung.

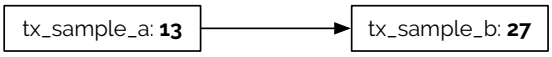

Abbildung 10: 1:1/1:n Relation

:tx_sample_a		
uid	**...**	**reference**
13	...	27

Abbildung 11: Referenzspeicherung am Aggregat (eigene Darstellung)

:tx_sample_b		
uid	**...**	**parent_id**
27	...	13

Abbildung 12: normalisierte Speicherung am aggregierten Element (eigene Darstellung)

Das untenstehende Beispiel zeigt das Datenbankresultat für die Speicherung von m:n Relationen bei serialisierter Speicherung – hier wird der Tabellenname des aggregierten Elements dem Primärschlüssel als Präfix vorangestellt – sowie die Verwendung einer Zwischentabelle.

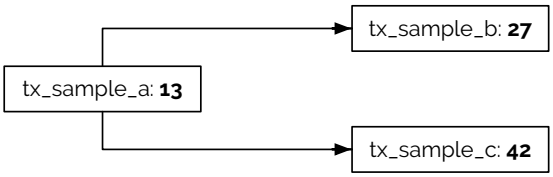

Abbildung 13: m:n Relationen

:tx_sample_a		
uid	**...**	**reference**
13	...	tx_sample_b_27, tx_sample_c_42

Abbildung 14: serialisierte Speicherung am Aggregat (eigene Darstellung)

:tx_sample_a_b_mm				
uid	**uid_local**	**uid_foreign**	**tablenames**	**sorting**
1	13	27	tx_sample_b	1
2	13	42	tx_sample_c	2

Abbildung 15: Speicherung mittels Zwischentabelle (eigene Darstellung)

3.2.3 FlexForms

Um verschachtelte Informationen, hauptsächlich zu Konfigurationszwecken, an Datensätzen hinterlegen zu können, ohne dazu das eigentliche Schema der Datenbanktabelle anpassen zu müssen, wurden „FlexForms“ in TYPO3 integriert. Dieses schemalose Konzept wurde jedoch soweit ausgebaut, dass damit ebenfalls eigene Hierarchien erstellt und Assoziationen zu anderen Datensätzen bewerkstelligt werden

können. Die Speicherung von FlexForms erfolgt in XML, jedoch gibt es keinerlei Methoden, um die syntaktische Korrektheit mittels XML Schema Definitionen (*XSD*) validieren zu können. Referenzierte Datensätze, die innerhalb einer FlexForm serialisiert wurden, müssten theoretisch beim Löschen dieser Datensätze entsprechend automatisch bereinigt werden. Tatsächlich bleiben aber diese verwaisten Informationen jedoch weiterhin gespeichert.

3.2.4 Domain Models

Die im Kernumfang systemweit verwendbaren Domain Models finden sich hauptsächlich in der Systemerweiterung Extbase und in der Dateiabstraktionsschicht (*File Abstraction Layer, FAL*). Da diese Modelle ausschließlich zum Datentransfer und zur Kapselung verwendet werden, bestehen diese nur aus Eigenschaften, entsprechenden Zugriffsmethoden (*Getter & Setter*) und enthalten keinerlei Applikationslogik. Die Anwendungslogik ist dagegen auf verschiedene Bestandteile des Systems verteilt – beispielsweise finden sich Komponenten des File Abstractions Layers in der Klasse *Content Object Renderer* für die Frontend-Ausgabe, oder in *Extended File Utility* für die Handhabung von Dateien im Backend. Im *Data Handler* und *Relation Handler* sind zusätzlich Aspekte der Dateiabstraktionsschicht durch die implizite Verwendung von 1:n Relationen zu finden.

3.3 Datenverarbeitung

3.3.1 Übersetzung & Lokalisierung

Zur Umsetzung von mehrsprachigen Funktionalitäten auf Datensatzebene gibt es in TYPO3 eine Fallunterscheidung, die auch hinsichtlich des technischen Vokabulars eine Herausforderung darstellt – Übersetzung und Lokalisierung. Es existiert eine Standardsprache, die jedoch nicht näher im Anwendungskontext spezifiziert ist – somit kann in einem Bereich der Website Deutsch als Standardsprache verwendet werden, in einem anderen Abschnitt könnte dies jedoch auch englische Inhalte betreffen. Daher ist es wichtig, dass die Standardsprache für eine TYPO3 Installation bzw. einen Seitenbaum

konsequent beibehalten wird – bei großen internationalen Projekt wird hier meistens Englisch als Standardsprache verwendet.

- Übersetzung (*Translation*): Ein Datensatz der Standardsprache wird in eine Zielsprache übersetzt, zusätzlich wird eine Relation zwischen den beiden Datensätzen hergestellt, welche auf das Element der Standardsprache zeigt.
- Lokalisierung (*Localization*): Ein Datensatz einer beliebigen Sprache wird als inhaltliche Vorlage für die Übersetzung in einen anderen Datensatz verwendet. Eine Verknüpfung der beiden Datensätze findet jedoch nicht statt – im TYPO3 Umfeld ist auch der Begriff einer Sprachkopie zu finden.

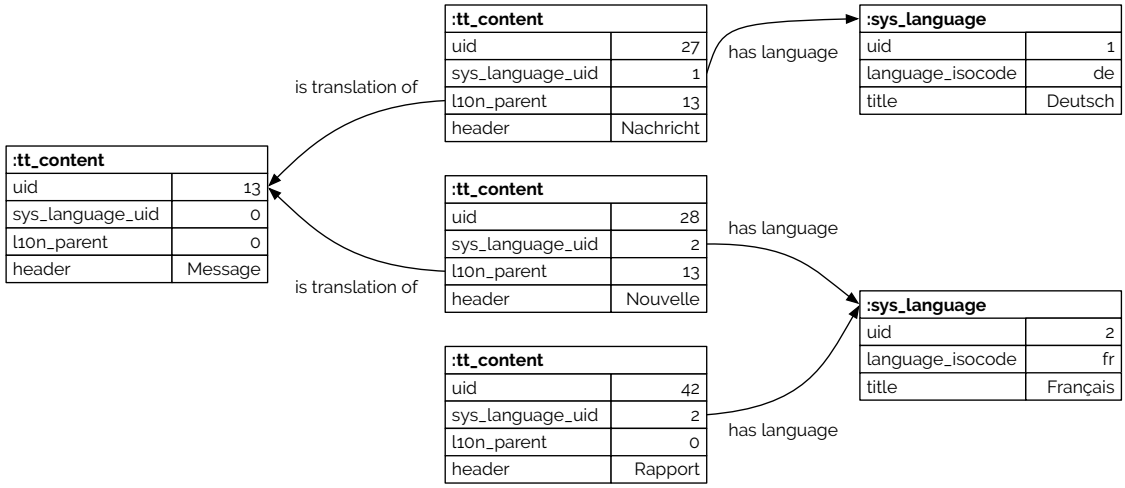

Abbildung 16: Übersetzungs- & Lokalisierungsreferenzen (eigene Darstellung)

Die vorherige Abbildung zeigt die Relationen für deutsche und französische Übersetzungen, sowie ein weiteres losgelöstes Inhaltselement im Französischen.

Für die Ermittlung übersetzter Datensätze wird zunächst das Element der Standardsprache ermittelt und schließlich für die zu verwendende Sprache überlagert (*Language Overlay*). Diese Überlagerung lässt sich auf Feldebene mittels TCA noch individuell festlegen – so ist es möglich einen Wert aus der Originalsprache in der Anzeige der Übersetzung zu verwenden. Die Bedeutung eines Elements ergibt sich somit erst aus der Analyse der Kombination der beiden Datenbankfelder.

3.3.2 Workspaces

Der Begriff „Workspaces“ bezeichnet innerhalb von TYPO3 die Möglichkeit, Modifikationen in einer separaten Arbeitsumgebung vorzubereiten, ohne im öffentlichen Bereich der Website dargestellt zu werden. Die Inhalte, die in einer Arbeitsumgebung erzeugt oder modifiziert werden, haben bis zur Publizierung nur Gültigkeit in diesem Anwendungskontext. Der Personenkreis ist dabei auf die beteiligten Redakteure begrenzt. Die Erzeugung eines Vorschaulinks ermöglicht jedoch auch außenstehenden Personen, den Bearbeitungsstand im Frontend zu betrachten.

Um Änderungen innerhalb eines Workspaces zu kennzeichnen und zu referenzieren, werden weitere Felder in der Datenbanktabelle benötigt. Beispielsweise ist die modifizierte Version mit dem ursprünglichen Element über eine implizite Referenz[12] verknüpft, ein weiteres Feld[13] gibt Aufschluss, ob ein Datensatz entweder komplett neu erzeugt, gelöscht oder verschoben wurde – für diese Szenarien werden zusätzliche Schattendatensätze als Platzhalter angelegt, welche wiederum von konkreten Workspace Versionen referenziert werden. Die Inhalte eines Workspaces befinden sich dabei stets in der gleichen Datenbank und Tabelle, wie auch die im Produktivsystem öffentlich darzustellenden Inhalte.

Das folgende Beispiel kombiniert das Übersetzungskonzept mit Modifikationen innerhalb eines bestimmten Workspaces. Ein Standarddatensatz wird dabei ins Deutsche übersetzt – da es die Übersetzung vorher nicht gab, wird neben dem konkreten neuen Inhaltselement ein Platzhalter („New Placeholder”) angelegt. Der Titel des Originalelements wird zu einem späteren Zeitpunkt noch abgeändert, was in einem neuen Versionseintrag resultiert.

---

[12] *t3ver_oid* zeigt auf den ursprünglichen Datensatz

[13] *t3ver_state* repräsentiert den Zustand (-1: neue Version; 1: Platzhalter für neue Version; 2: Platzhalter für gelöschten Datensatz; 3: Platzhalter für Verschiebung; 4: Indikator für verschobenen Datensatz)

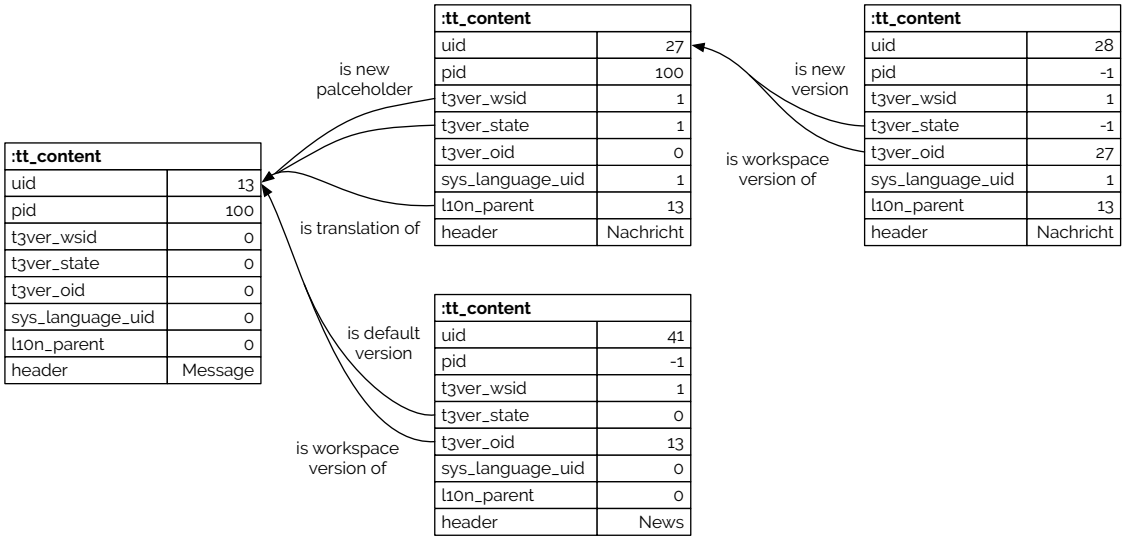

Abbildung 17: Übersetzungs- & Lokalisierungsreferenzen mittels Workspaces (eigene Darstellung)

Die Voransicht im Frontend erfolgt ähnlich zu der Anwendung von Übersetzungen in Lokalisierungen im vorherigen Abschnitt durch die Überlagerung von Feldwerten. Für die Ansicht in der Standardsprache wird der Originaldatensatz ermittelt und anschließend durch eine mögliche Workspace Version überlagert – im vorherigen Beispiel würde also „News“ anstatt „Message“ als Titel angezeigt werden. Soll die Ausgabe für eine konkrete Sprache erfolgen wird ebenfalls das Standardelement bezogen, zunächst mit der Workspace-Modifikation überlagert und anschließend durch das Workspace Overlay der Übersetzung ergänzt.

Für die Ansicht im Frontend ist der für die Übersetzung angelegte Platzhalter irrelevant. Dieser hätte nur Bedeutung, wenn übersetzte Datensätze – ohne den Umweg über den Originaldatensatz – direkt abgefragt werden.

Um die Verwendungsart eines Inhaltselements zu ermitteln müssen im kombinierten Kontext von Workspace und Sprache eine Vielzahl[14] von Eigenschaften ermittelt und interpretiert werden.

[14] notwendig sind insgesamt acht Felder: *uid, pid, t3ver_wsid, t3ver_state, t3ver_oid, t3ver_move_id, sys_language_uid* und *l10n_parent*

3.3.3 Hooks & Signal-Slot

Die Begriffe „Hook“ und „Signal-Slot“ werden innerhalb von TYPO3 variabel im Sinne der Verhaltensmuster Schablonenmethode (*Template Method*) oder auch Interceptor verwendet. Während Schablonenmethoden nach ihrer Ausführung wieder zurück zum eigentlichen Programmablauf zurückkehren, kann ein Interceptor dafür sorgen, dass die ursprüngliche Ausführung komplett übersprungen wird.

Innerhalb des *Data Handlers* gibt es zahlreiche Hooks, die es entweder erlauben, die zu speichernden Informationen abzuändern oder auch zusätzliche eigene Datenmodifikationen vorzunehmen. Durch diese Möglichkeit können sich weitere Fehlerquellen ergeben, welche sich negativ auf die Integrität und Datenkonsistenz auswirken können.

3.3.4 Data Handler & Relation Handler

Formulareingaben im Backend werden als flache Array-Strukturen für jeweils einen Datensatz an den *Data Handler* übergeben. Neu zu erzeugende Datensätze werden mit einem speziellen Zufallswert gekennzeichnet – dieser Wert wird auch für zu erstellende Referenzen verwendet und muss somit zu einem späteren Zeitpunkt in den echten ganzzahligen Primärschlüssel einer Entität transformiert werden.

Referenzielle Abhängigkeiten werden im *Data Handler* erst dann verarbeitet, wenn das referenzierende Eltern-Element verarbeitet wird. Da keine Hierarchiestruktur ermittelt wird, ist es somit möglich, dass das Kind-Element vor dem eigentlichen Eltern-Element verarbeitet wird – fehlende Daten werden für die beteiligten Elemente teilweise temporär gesammelt und am Ende der kompletten Verarbeitung auf die betroffenen Elemente erneut angewandt. Die Speicherung eines Datensatzes besteht aus mehreren Einfüge- und Aktualisierungsoperationen – durch die genannten Einschränkungen der referentiellen Integrität, ist es wahrscheinlich, dass Entitäten und deren Abhängigkeiten kurzzeitig in einem unvollständigen und inkonsistenten Zustand in der Datenbank abgespeichert sind.

Spezielle Aktionen werden zusätzlich durch bereits vorhanden Möglichkeiten innerhalb des *Data Handlers* abgebildet. Dies führt zwar zur Wiederverwertung von Code, schafft jedoch auch komplexe Rekursionen, die im Fehlerfall nur schwer zu analysieren sind. Beispielsweise wird die Aktion, einen Datensatz zu übersetzen intern als Datensatzkopie umgesetzt – am Ende dieser Verarbeitung werden sprachrelevante Datenbankfelder erneut modifiziert. Die Kopie eines Datensatzes wird wiederum durch die Erzeugung eines neuen Elements mit den Eigenschaftswerten des ursprünglichen Datensatzes durchgeführt.

Der Einsatz von Hooks ermöglicht es, während der Verarbeitung auf bestimmte Vorkommnisse zu reagieren. Jedoch ist es damit auch möglich, Informationen zu verändern und damit den ursprünglichen Prozessfluss zu manipulieren.

Das nachfolgende Sequenzdiagramm visualisiert die genannten Prozesse, die bei der Übersetzung eines Inhaltselements ausgeführt werden. Der *Data Handler* instanziiert und ruft sich dabei selbst mehrmals auf. Lese- und Speichervorhänge von Referenzen werden an den *Relation Handler* delegiert. Bedingt durch die verschachtelte Verarbeitungsstruktur werden mehrmals ähnliche Informationen mit der Datenbank ausgetauscht. Das Beispiel zeigt nur einen von vielen Teilaspekten des Data Handlers, die Komplexität ist beim Einsatz von Workspaces noch wesentlich umfangreicher.

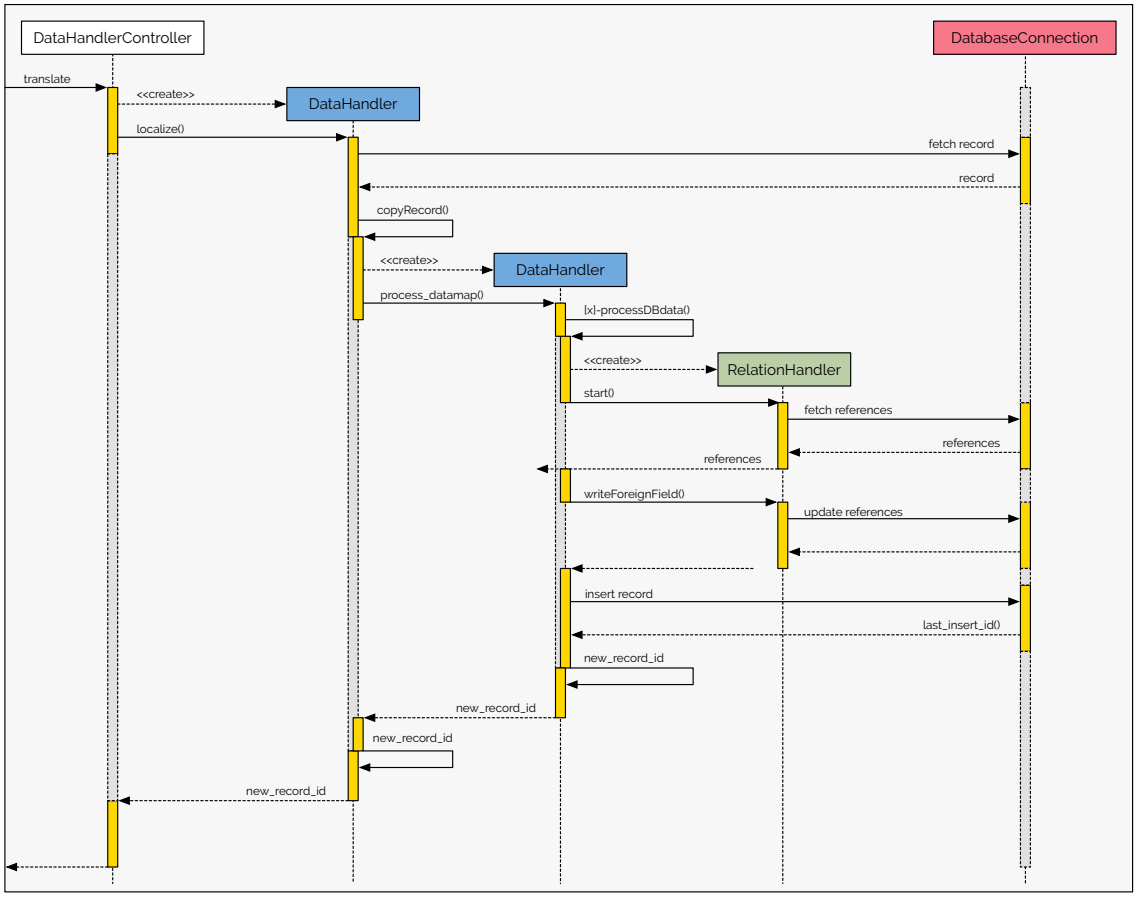

Abbildung 18: Sequenzdiagramm zur Übersetzung eines Inhaltselements (eigene Darstellung)

3.4 Datenspeicherung

3.4.1 Transaktionslose Speicherung

Die Datenspeicherung von TYPO3 beruht auf der optimistischen und teilweise naiven Annahme, dass geschriebene Datenbankwerte korrekt sind und direkt für die Erzeugung von weiteren Assoziationen verwendet werden können. Auf Datenbankebene gibt es keine Prozesse, welche in Datenbanktransaktionen gekapselt und somit gegenüber Nebeneffekten gesichert wären.

Alle TCA-Tabellen besitzen jeweils ein Identifikationsfeld (*uid*), welches automatisch inkrementiert wird – nach dem Erzeugen von Datensätzen wird der datenbankseitig generierte Feldwert erneut ausgelesen und für Referenzen zu anderen Datensätzen herangezogen. Bei der Persistenz von m:n Mehrfachrelationen über eine Zwischentabelle kann optional ebenfalls ein Identifikationsfeld vorliegen – allerdings

werden in den meisten Fällen[15] nur die Werte der Primärschlüssel, der für die Assoziation relevanten Datensätze, zur Identifikation verwendet. Bei gleichzeitigen Datenbankoperationen auf gleiche Entitäten ist die Wahrscheinlichkeit für das Auftreten von Problemen bei konkurrierenden Prozessen (*Race Conditions*) gegeben. Das folgende Beispiel zeigt den aktuell in TYPO3 implementierten Verlauf bei der Aktualisierung von Mehrfachreferenzen über eine Zwischentabelle – hier versuchen zwei unterschiedliche Prozess fast zeitgleich unterschiedliche Verknüpfungen zu setzen.

Sequenz	Prozess A	Prozess B
#1	`DELETE FROM tx_sample_a_b_mm WHERE uid_local=13 AND uid_foreign=27 AND tablenames='tx_sample_b';`	
#2		`DELETE FROM tx_sample_a_b_mm WHERE uid_local=13 AND uid_foreign=27 AND tablenames='tx_sample_b';`
#3	`INSERT INTO tx_sample_a_b_mm (uid_local,uid_foreign,tablenames) VALUES (13,27,'tx_sample_b');`	
#4		`INSERT INTO tx_sample_a_b_mm (uid_local,uid_foreign,tablenames) VALUES (13,42,'tx_sample_c');`

Abbildung 19: Nebeneffekte bei konkurrierenden Datenbankaktionen (eigene Darstellung)

In der Datenbank wären nun zwei Relationen vorhanden, obwohl beide Prozesse jeweils nur eine Relation beabsichtigt hätten. Im nächsten Beispiel wird das gleiche Szenario unter dem Einsatz von Datenbanktransaktionen durchgeführt. Für den Einsatz unter MySQL ist es jedoch notwendig, als Speicherart nicht MyISAM, sondern beispielsweise InnoDB[16] zu verwenden – unter MyISAM können die Anforderungen der referentiellen Integrität für Transaktionen nicht angewandt werden.

[15] die Definition obliegt dem Entwickler – beispielsweise ist im Quellcode von TYPO3 CMS 8.3 dieses Feld für Kategoriezuweisungen beliebiger Elemente in der Tabelle *sys_category_record_mm* nicht gesetzt

[16] ohne explizite Angabe der Speicherart auf Tabellen-Ebene wird ab MySQL Version 5.5.5 InnoDB als Standard verwendet, in den Vorgängerversionen war dies jedoch MyISAM

Sequenz	Prozess A	Prozess B
#1	`START TRANSACTION;`	
#2		`START TRANSACTION;`
#3	`DELETE FROM tx_sample_a_b_mm WHERE uid_local=13 AND uid_foreign=27 AND tablenames='tx_sample_b';`	**<<element locked>>**
#4		`DELETE FROM tx_sample_a_b_mm WHERE uid_local=13 AND uid_foreign=27 AND tablenames='tx_sample_b';`
#5	`INSERT INTO tx_sample_a_b_mm (uid_local,uid_foreign,tablenames) VALUES (13,27,'tx_sample_b');`	
#6	`COMMIT;`	**<<element lock released>>**
#7 (war #4)		`DELETE FROM tx_sample_a_b_mm WHERE uid_local=13 AND uid_foreign=27 AND tablenames='tx_sample_b';`
#8		`INSERT INTO tx_sample_a_b_mm (uid_local,uid_foreign,tablenames) VALUES (13,42,'tx_sample_c');`
#9		`COMMIT;`

Abbildung 20: Nebeneffektfreie Datenbankaktionen durch Transaktionen (eigene Darstellung)

In Sequenz #3 wird durch die Modifikation eine Sperre auf den Datensatz erzeugt. Die Anweisung aus Sequenz #4 wird somit solange blockiert bis die Verarbeitung des ersten Prozesses durch das den Commit-Befehl in Sequenz #6 abgeschlossen wurde. Zwar lassen sich durch den Einsatz von Transaktionen gleichzeitige Zugriffe auf Datenbankebene nicht verhindern, jedoch kann sichergestellt werden, dass keine Nebeneffekte auftreten – in diesen Fall überschreibt Prozess B die Relationen.

3.4.2 Eindeutigkeit von Entitäten

Das für TCA-Datenbanktabellen verwendete Identifikationsfeld ist als fortlaufender, ganzzahliger Wert definiert. Werden Entitäten aus einem TYPO3 System in eine andere TYPO3 Installation importiert, müssen diese Werte entsprechend neu vergeben werden – dies gilt natürlich auch für alle Relationen, welche bei den zu importierten Daten angelegt sind. Eine systemübergreifende Einzigartigkeit liegt somit nicht vor, wäre aber gerade für den interoperablen Einsatz auf verteilten Systemen wünschenswert und vorteilhaft. Dies könnte beispielsweise durch den Einsatz von UUIDs[17] umgesetzt

[17] gemäß UUID Version 4 in RFC 4122, siehe https://tools.ietf.org/html/rfc4122

werden, welche pseudo-zufällige Zahlen im Wertebereich bis 2^{122} erzeugen – innerhalb der Anwendung würden diese Werte hexadezimal repräsentiert werden und über ein Wertobjekt auf syntaktische Korrektheit geprüft werden können. Die Wahrscheinlichkeit, dass bei zwei betreuten TYPO3 Systemen die gleiche UUID vergeben wird ist hinreichend gering.

3.4.3 Objektrelationale Abbildung

Die in TYPO3 vorhandene objektrelationale Abbildung kann nur über Applikationen verwendet werden, welche auf der Basis des Extbase-Frameworks umgesetzt wurden. Starke Namenskonventionen leiten aus Klassennamen einer Entität das zugehörige Repository ab und verknüpfen dies automatisch mit einer entsprechend mittels TCA konfigurierten Datenbanktabelle – beispielhaft sind im Folgenden die implizit zusammengehörigen Komponenten Entität, Repository und Tabelle aufgeführt:

- *H4ck3r31\BankAccountExample\Domain\Model**Account***
- *H4ck3r31\BankAccountExample\Domain\Repository**Account**Repository*
- *tx_bankaccountexample_domain_model_**account***

Durch den Einsatz dieser ORM-Schicht können Datensätze in entsprechende Entitäten überführt werden und schließlich wiederum in der Datenbank in serialisierter Form abgespeichert werden. Für den Einsatz der objektrelationalen Abbildung in Extbase muss jedoch zwingend das entsprechende Table Configuration Array für ein bestimmtes Datenmodell vorhanden sein.

Ein eigenständiges Abfrage Objekt (*Query*) erlaubt es Anfragen abstrakt ohne den Einsatz von SQL erzeugen zu können. Über zusätzliche Einstellungen (*Query Settings*) können dabei die Resultate sortiert werden oder, für TYPO3 spezifische, Sichtbarkeitseinstellungen definiert werden. Abhängige Referenzen werden dabei automatisch aufgelöst (Rau, et al., 2010, S. 148-157). Die erwähnten Sichtbarkeitseinstellungen lassen sich jedoch für automatisch bereitgestellte Kind-

Elemente nicht mehr anpassen – die Abläufe der Selektion werden in diesem Fall nicht über Repositories ausgeführt, sondern innerhalb der ORM-Schicht interpretiert[18].

3.4.4 Zwischenspeicher

Um wiederkehrende rechen- und abfrageintensive Operationen in TYPO3 zu reduzieren, wurden für verschiedene Anwendungsszenarien Zwischenspeicher (*Caches*) geschaffen. Über das integrierte Caching Framework werden Zustände in relationalen Datenbanken, Key-Value-Stores oder auch dem Dateisystem gespeichert. Diese Caches sind zum größten Teil darauf ausgelegt, zur Anwendungslaufzeit neu erzeugt werden zu können, falls diese gelöscht wurden.

Die folgenden Zwischenspeicher sind relevant für die performante Datenverarbeitung im Frontend und Backend von TYPO3.

- *cache_pages*: Speicherung des im Frontend auszugebenden Markups für eine bestimmte Kombination aus Sprach- und Benutzergruppenkontext
- *cache_pagesection*: Speicherung der analysierten und ausgewerteten TypoScript Funktionalitäten und dynamischen Bedingungen für einen bestimmten Sprach-oder Benutzerkontext im Frontend
- *cache_rootline*: Speicherung von vererbten Konfigurationen einer bestimmten Seite in der Hierarchie des Seitenbaums (*Rootline*)
- *extbase_datamapfactory_datamap*: Speicherung der für Extbase relevanten Konfigurationen der aus TCA abgeleiteten Metamodelle
- *extbase_reflection*: Speicherung der Resultate der PHP Class Reflection, wie z.B. Datentypen von Domain Model Eigenschaften oder individuelle Annotationswerte, die für Dependency Injection benötigt werden
- *workspaces_cache*: Speicherung der Unterschiede von Versionen im Vergleich zu Originaldatensätzen für einen bestimmten Redakteur bei der Verwendung des Workspace Moduls im Backend

[18] durch einen Trick, kann dies umgangen werden, http://stackoverflow.com/questions/39993664/typo3-extbase-how-to-get-disabled-related-object-without-raw-sql-query/39994951#39994951

Der Einsatz von Caches beschleunigt zwar in den meisten Fällen die Verarbeitungszeit, ist jedoch auch ein Indiz dafür, dass die Anwendungslogik an diesen Stellen zu umfangreich, zu komplex oder auch zu rechenintensiv ist. Zusätzliche Probleme treten bei der Invalidierung dieser Zwischenspeicher auf. Ein bekanntes Phänomen ist in diesem Zusammenhang, ein unverändertes Seitenmenü im Frontend, obwohl eine dort aufgeführte Seite beispielsweise umbenannt wurde – als Ausweg bleibt dem Redakteur nur, den Seiten-Cache manuell zu leeren.

3.4.5 Projektionen

Innerhalb von TYPO3 gibt es bereits einige Konstrukte, die den Charakter einer Projektion haben (siehe Abschnitt 2.5), jedoch explizit ohne den Einsatz von Domain Events erstellt werden. Die im Folgenden betrachteten Komponenten werden weitgehend aus dem Umfeld des *Data Handlers* (siehe Abschnitt 3.3.3) heraus erzeugt.

In den Datenbanktabellen *sys_history* und *sys_log* werden Zustandsmodifikationen für die Ausgabe eines Änderungsprotokolls erfasst. Da der ursprüngliche und beabsichtigte Zustand bei jedem Vorgang gespeichert werden, ist es bereits möglich, Änderungen bis zu einem bestimmten Zeitpunkt rückgängig zu machen. Dieses Verfahren funktioniert jedoch nicht vollständig bei verschachtelten Datenstrukturen und Assoziationen.

Der Referenzindex speichert den Bezug von bestimmten Entitäten zueinander für einen bestimmten Anwendungsfall im Live- oder Workspace-Kontext in der Datenbanktabelle *sys_refindex*. Da dieser Index jedoch nicht in allen Situationen verlässlich aktualisiert wird, gibt es ein separates Skript, welches die Inhalte unter allen Datensätzen hinsichtlich existierender Assoziationen erneut ermittelt und abspeichert. Diese Abweichungen treten vor allem dann auf, wenn mittels Hooks im Backend zusätzliche Daten modifiziert werden oder im Frontend neue Datensätze erzeugt werden, ohne dabei explizit eine Aktualisierung des Referenzindex anzustoßen.

Bei Anwendungen, die auf der Basis von Extbase umgesetzt werden, ist eine entsprechende Anweisung, den Referenzindex bei Speichervorgängen automatisch zu aktualisieren standardmäßig deaktiviert. Zwar könnten Entwickler von TYPO3

Erweiterungen diese Einstellung individuell für ihre Anwendungsdomäne aktivieren, eine Stichprobe hat jedoch ergeben, dass dem in der Realität nicht so ist[19].

Im aktuellen Zustand kann der Referenzindex also nicht als verlässliche Quelle verwendet werden – soll eine Datei über die Benutzeransicht im Backend gelöscht werden, wird jedoch genau diese Information herangezogen um verbleibende Referenzen zu ermitteln.

3.4.6 Datenbankabstraktion

Während der Ausarbeitung dieser Arbeit wurden im Kern von TYPO3 CMS 8 alle Datenbankzugriffe auf die neu integrierte Datenbankabstraktionsschicht „Doctrine DBAL“ umgestellt. Durch diesen Schritt wurden MySQL-spezifische Anweisungen angepasst – somit ist es möglich, TYPO3 plattformübergreifend mit PostgreSQL, Oracle oder auch SQLite einzusetzen. Datenbankanweisungen werden über den sogenannten *Query Builder* erzeugt und für die gewählte Datenbankplattform übersetzt.

```
1. $connectionPool = GeneralUtility::makeInstance(
2.     ConnectionPool::class
3. );
4. $queryBuilder = $connectionPool
5.     ->getQueryBuilderForTable('tt_content');
6. $statement = $queryBuilder
7.     ->select('*')
8.     ->from('tt_content')
9.     ->where(
10.        $queryBuilder->expr()->eq('pid', $pidParameter),
11.        $queryBuilder->expr()->eq('colPos', $colPosParameter)
12.    )
13.    ->orderBy('sorting')
14.    ->execute();
15. foreach ($statement as $row) {
16.    // ...
17. }
```

Programm 4: Beispielhafte Verwendung des DBAL Query Builders (eigene Darstellung)

[19] lediglich elf Erweiterungen, die auch im TYPO3 Extension Repository verfügbar sind, haben die Einstellung *persistence.updateReferenceIndex* aktiviert

Zusätzlich wurden Restriktionen für Datenbankaktionen integriert, welche automatisch beim Abrufen von Informationen ausgeführt werden. Damit ist die spezifische Abfragelogik für einen kombinierten Anwendungskontext aus Workspace und Sprache erstmals in einer entsprechenden Komponente gekapselt. Diese implizierte Ausführung kann alternativ zur Laufzeit für bestimmte Vorgänge angepasst oder deaktiviert werden.

3.5 Konfiguration

Dieser Abschnitt betrachtet Konfigurationsmöglichkeiten, die Verhaltensweisen und Freigabeoptionen des Gesamtsystems bzw. eines bestimmten Anwendungskontexts betreffen, jedoch innerhalb von TYPO3 als eigenständige Datenbanktabellen und somit Entitäten behandelt werden.

Es gilt zu entscheiden, ob sich die Bedeutung einer dieser Komponenten primär auf das Verhalten und den Funktionsumfang des Gesamtsystems bezieht, oder andererseits die individuelle Verwendung im Sinne einer Entität gegeben ist. Daraus kann abgeleitet werden, ob eine Komponente eher den Charakter eines eigenständigen Datensatzes oder einer statischen Einstellungsdirektive hat.

Durch die Extraktion von geeigneten Tabellen kann die Komplexität des Systems hinsichtlich referentieller Abhängigkeiten auf der Datenbankebene reduziert werden.

	Sprache möglich	Version möglich	Extraktion möglich	Hinweise & Bemerkungen
Benutzergruppen	nein	nein	ja	ähnlich in Frontend & Backend
Datenspeicher	nein	nein	ja	eindeutiger Bezeichner notwendig
TypoScript	nein	ja	bedingt	Zuordnung als Seiteneigenschaft
Backend Layouts	nein	ja	bedingt	Einschränkung für Redakteure
Domains	nein	nein	ja	Zuordnung als Seiteneigenschaft
Sprachen	nein	nein	ja	definiert Anwendungskontext
Workspaces	nein	nein	ja	definiert Anwendungskontext

Abbildung 21: Extraktion von Einstellungen als statische Konfiguration (eigene Darstellung)

3.5.1 Benutzergruppen

Benutzergruppen definieren Zugriffsberechtigungen auf Inhalte und Anwendungsbereiche. Unabhängig, ob Berechtigungen aus einer Datei eingelesen werden oder über die Datenbank bereitgestellt werden, soll das System die Bedeutung in beiden Fällen universell ermitteln können.

Beispielhaft wird angenommen, dass ein Benutzer in der Vergangenheit Inhalte löschen konnte, diese Berechtigungen aber inzwischen entzogen wurden. Im Hinblick auf Event Sourcing stellt diese Änderung jedoch keine Einschränkung dar, da Ereignisse in der Vergangenheit tatsächlich eingetreten sind und in der Gegenwart keine erneute Validierung der Berechtigungen stattfinden darf. Daraus resultiert, dass Benutzergruppen nicht als Entitäten angesehen werden müssen.

3.5.2 Dateispeicher

Im Rahmen des File Abstraction Layers definieren Dateispeicher über die Datenbanktabelle *sys_file_storage* den Speicherort von Dateien und konfigurieren zusätzliche Parameter, wie beispielsweise Benutzer- und Passwortdaten für den Zugriff mittels SFTP oder WebDAV auf entfernte Systeme. Die relevanten Einstellungen können ebenfalls statisch, außerhalb der Datenbank hinterlegt werden. Aktuell wird die Information, ob ein entfernter Dienst erreichbar ist in dieser Entität abgelegt – da sich dieser externe Zustand ändern kann, hat diese Information jedoch eher die Charakteristik eines Zwischenspeichers.

3.5.3 TypoScript

TypoScript definiert im Frontend den Ablauf der Ausgabeprozesse. Aktuell erfolgt die Zuweisung von Konfigurationen, die bereits jetzt statisch im Dateisystem gespeichert sein können, über einen Datensatz und Zuweisung dessen zu einem bestimmten Ast des Seitenbaums. Das aktuelle Konstrukt als Datenbanktabelle kann bis auf den Aspekt der Assoziation reduziert werden – damit kann diese Information auch direkt als Eigenschaft einer Seite gespeichert werden.

Es ist möglich, TypoScript für einen bestimmten Workspace-Kontext zu definieren – dieser Umstand lässt sich jedoch ebenfalls durch eine neue Vererbungsstruktur innerhalb der statischen Konfiguration lösen.

3.5.4 Backend Layouts

Dieses Konstrukt definiert die Bedeutung und Verfügbarkeit der Spaltenansicht des Seitenmoduls innerhalb des TYPO3 Backends. Die Verwendung der Datenbank bringt implizit lediglich den Vorteil, Einstellungen komfortabler über einen grafischen Assistenten der Backend Eingabemasken durchführen zu können. Da die Speicherung alternativ bereits statisch im Dateisystem erfolgen kann, gibt es keine weitere technische Notwendigkeit an der Speicherung in Datensätzen festzuhalten.

Aktuell ist es auch normalen Redakteuren erlaubt, die als Datensatz abgelegten Layouts zu modifizieren, oder neue anzulegen. Bei einer Extraktion als statische Konfiguration, muss jedoch vorab die konkrete Auswirkung für Redakteure ermittelt werden, da dies den Wegfall einer Funktionalität bedeutet würde.

3.5.5 Domains

Das Anlegen von Domainnamen, auf die eine Website bzw. ein bestimmter Seitenast reagieren soll, hängt stark vom Kontext der Ausführung und der Konfiguration des Webservers ab. Dieser Bereich ist somit eher als Aspekt der Softwareverteilung zu sehen, da sich Domainnamen auf einer Entwicklungsumgebung von denen der Produktivumgebung unterscheiden werden.

Die Konfiguration von Domainnamen kann somit in eine separate statische Datei ausgelagert werden.

3.5.6 Sprachen

Sprachen sind typische Wertobjekte, bestehend aus einem eindeutigem Bezeichner, dem Sprachcode. Die Repräsentation der Sprache und einer Flagge kann statisch erfolgen. Sprachen haben also den Charakter einer Konfiguration auf Systemebene.

Aktuell wird die relevante Zuordnung zu einer Sprache über einen zweistelligen Sprachcode[20] vorgenommen – zum Beispiel „en“ für Englisch oder „de“ für Deutsch. Eine

[20] gemäß ISO-639-1 alpha-2, siehe https://en.wikipedia.org/wiki/ISO_639-1

bessere Aussagekraft lässt sich jedoch durch die Einbeziehung der Sprachregion[21] (*Locales*) erreichen – beispielweise „en-GB“ für Britisches Englisch oder „de-AT“ für die in die in Österreich angewandte Sprache.

Für die Ausgabe im Frontend existiert eine Priorisierung und Zurückschaltung – beispielsweise sollen Inhalte in Frankokanadisch oder Französisch angezeigt werden, sind diese nicht vorhanden wird als Ersatz Englisch verwendet. Die Konfiguration dafür erfolgt aktuell bereits statisch mittels TypoScript als Liste numerischer Werte – durch die Verwendung von Locales ändert sich dies beispielsweise in die Definition der Verkettung „fr-CA,fr-FR,en-US“.

In der aktuellen TYPO3 Version werden Sprachen als Entitäten verwendet – die Zuweisung erfolgt durch die Referenz auf den Primärschlüssel der Datensätze. Jedoch ist es möglich, einem Inhaltselement die Standardsprache (Zahlenwert „0“) oder die Verwendung in allen Sprachen (Zahlenwert „-1“) zuzuweisen – da es hierzu keinen entsprechenden Datensatz gibt, ist eine eigenständige und konditionale Behandlung dieser beiden Ausnahmewerte notwendig.

3.5.7 Workspaces

Workspaces definieren einen abgetrennten Bereich und Anwendungskontext auf Systemebene – deswegen liegt nahe, dass diese Einstellung ebenfalls statisch außerhalb der TYPO3 Datenbank erfolgen kann. Die Zuweisung von Berechtigungen innerhalb eines Workspaces an Redakteure erfolgt aktuell jedoch wahlweise über Benutzergruppen oder direkt Entitäten der Benutzer. Für eine Extraktion muss also eine allgemeingültige Definition geschaffen werden, die Eigenschaften bezeichnen kann, oder den Namen eine Benutzerrolle zuweist – unabhängig von Primärschlüsseln einer Datenbanktabelle.

[21] gemäß RFC 5646, siehe https://tools.ietf.org/html/rfc5646

3.6 Testszenarien

Hinsichtlich des Workspaces Teilprojekts während der Entwicklungsphase von TYPO3 CMS 6.2 wurden im Vorfeld funktionale Testfälle definiert. Mit diesen Testfällen ist es möglich, das zu erwartende Verhalten zu überprüfen und mögliche Seiteneffekte bei Änderungen der Implementierung zu minimieren. Diese Tests prüfen dabei weniger den konkreten Ablauf von einzelnen Modulen der Anwendung, sondern stellen sicher, dass für bestimmte Eingabeparameter des Gesamtprozesses ein definierter Endzustand in der Datenbank vorhanden ist. Innerhalb des TYPO3 Kerns sind im Speziellen für die Handhabung von Datenstrukturen folgende Testszenarien vorhanden:

- Erzeugen, Modifizieren, Löschen von Inhaltselementen
- Erzeugen, Modifizieren, Löschen von Seiten und Seitenhierarchien
- Erstellen, Entfernen und Sortieren von Referenzen auf Dateien (1:n)
- Erstellen, Entfernen und Sortieren von Referenzen auf Kategorien (m:n)
- Übersetzen und Kopieren der vorgenannten Entitäten und Strukturen
- Versionsverwaltung mittels Workspaces für die vorgenannten Aspekte

Die Tatsache, dass vor allem für das Anwendungsumfeld des *Data Handlers* Prüfmechanismen verfügbar sind, erlaubt eine weitgehende Sicherstellung, dass die Anreicherung mit Event Sourcing Methoden noch immer zu den gleichen Zuständen in der Datenbank führen.

3.7 Zusammenfassung

Währen der Analysephase wurde deutlich, dass TYPO3 mit Extbase bereits zahlreiche Konzepte des Domain-Driven Designs unterstützt, dabei jedoch eine individuelle Interpretation verfolgt. Die Zusammenhänge zwischen Datenmodell, Repository und Datenbanktabelle unterliegen einer stark bindenden Namenskonvention.

Datenverarbeitungsprozesse können im TYPO3 Backend über Hooks großzügig individuell behandelt werden – jedoch erlauben die vorhandenen Strukturen auch, manipulativ Einfluss auf die ursprüngliche Information und Intention zu nehmen. Vorkehrungen zur Reduzierung von Nebeneffekten bei der Datenverarbeitung werden vom System nicht angeboten – dadurch können auch inkonsistente Zustände, wie beispielsweise bei Workspace Versionen oder dem Referenzindex auftreten. Insgesamt ist das Konzept, Datensätze mit Versionen und Übersetzungen zu überlagern, unter subjektiver Betrachtung als kompliziert einzustufen und beinhaltet das Risiko entwicklerseitig falsch eingesetzt oder komplett vernachlässigt zu werden.

Im Hinblick auf Datenstrukturen, die Rolle von Entitäten und deren Abhängigkeiten untereinander, wurden einige Optimierungsmaßnahmen aufgedeckt. Unabhängig von der Anwendung von Event Sourcing stellt beispielsweise die Extraktion der Datenbanktabellen für Sprachen, Domains und Workspace-Definitionen eine Erleichterung für das gesamte System dar.

4 Implementierung

4.1 Transfer & Abgrenzung

In Bezug auf die Umsetzung und Interpretation von Domain-Driven Design Paradigmen sind keine Grenzen gesetzt. Die im Rahmen der Arbeit anzufertigende Implementierung konzentriert sich dabei jedoch nur auf einige Aspekte und hat dabei lediglich den Anspruch, einen funktionierenden Demonstrationsprototypen zu erschaffen. Es ist zu erwarten, dass das Resultat zunächst als Diskussionsgrundlage innerhalb der TYPO3 Community verwendet wird und dass durch weitere Iterationen die neuen Konzepte verfeinert oder auch teilweise verworfen werden.

Für die Funktionalität eines Prototyps werden folgende Teilbereiche definiert:

- Anfertigung von generischen Komponenten, die den TYPO3 Kern und Drittanbietererweiterungen in die Lage versetzen, Event Sourcing einzusetzen
- Überführung der Anwendungslogik im TYPO3 Backend zur Verwaltung von Datensätzen (*Data Handler*) in geeignete Befehle und Ereignisse unter dem Einsatz von generischen Datenstrukturen, sowie Projektion von Ereignissen für eine kontextabhängige Darstellung im Frontend
- Ausarbeitung einer abgegrenzten Anwendungsdomäne auf Basis des Extbase MVC-Frameworks, um die zukünftige Bedeutung vorhandener Komponenten dieses Pakets analysieren und bewerten zu können

Die Implementierung ist als Interpretation der Konzepte des Domain-Driven Designs hinsichtlich TYPO3 und den dort vorhandenen Funktionsfähigkeiten zu verstehen.

4.2 Allgemeine Komponenten

Während der Implementierungsphase wurde zunächst die generische Umsetzung für den TYPO3 Kern und die Verarbeitung über das Backend, sowie zusätzlich eine spezifische Ausarbeitung im Frontend auf der Basis von Extbase geschaffen. Dieser Abschnitt betrachtet Komponenten, die in beiden Varianten in ähnlicher Form vorhanden sind.

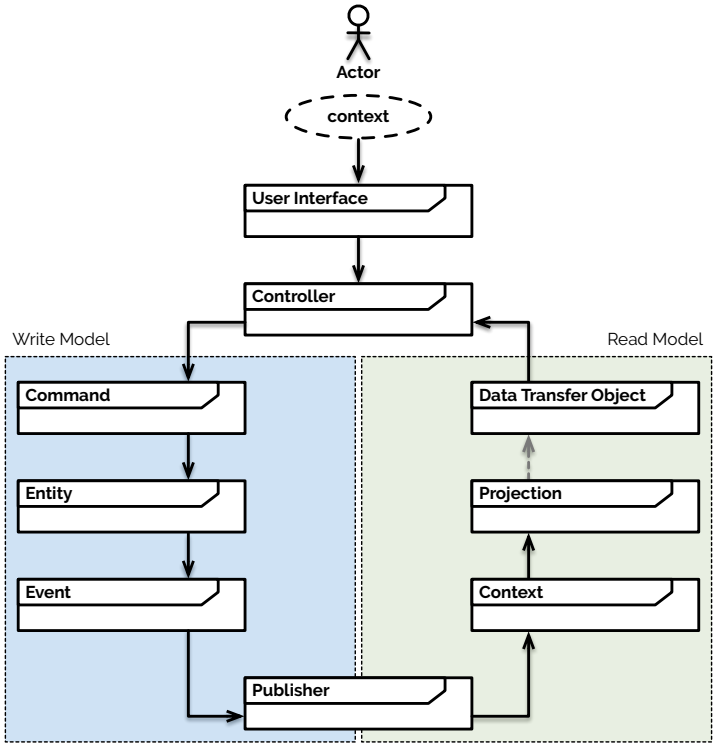

Abbildung 22: Übersicht allgemeiner Schichten (eigene Darstellung)

Die Abbildung zeigt die im Allgemeinen bei der Verarbeitung beteiligten Schichten. Das „Write Model“ auf der linken Seite zeigt die Interaktion mittels Befehlen und Ereignissen, sowie die automatische Überführung in entsprechende Projektionen. Das „Read Model“ auf der rechten Seite verwendet dann lediglich die projizierten Informationen unter Einsatz von Datentransferobjekten. Somit ist die Schreibrichtung von der Leserichtung getrennt.

4.2.1 Commands

Befehle beziehen sich auf Absichten, die von Domain Models verstanden werden sollen und beinhalten somit die notwendigen Informationen, damit diese auch ausgeführt werden können. Ob ein Befehl tatsächlich verarbeitet werden kann, ist erst zu einem späteren Zeitpunkt relevant – beispielsweise kann die Ausführung verweigert werden, falls Zugriffsberechtigungen nicht ausreichend sind. Befehle können auch als eine Art Benachrichtigung verstanden werden und sind als separate Objekte modelliert, welche das *Domain Command* Interface implementieren – dadurch erhalten Objekte die Bedeutung, innerhalb einer Applikation als Kommandos interpretiert zu werden.

Neben den Objekteigenschaften sind in der Klasse nur Methoden zum Abfragen der Werte (*Getter*) vorhanden, die Modifikation außerhalb der Objektinstanz ist ausdrücklich nicht erlaubt.

```
18. class ChangeAddressCommand implements DomainCommand
19. {
20.     private $customerId;
21.     private $address;
22.     public function __construct($customerId, Address $address)
23.     {
24.         $this->customerId = $clientId;
25.         $this->address = $address;
26.     }
27.     public function getCustomerId() { /* ... */ }
28.     public function getAddress() { /* ... */ }
29. }
```

Programm 5: Beispiel für einen generischen Befehl (eigene Darstellung)

Das vorangehende Beispiel verwendet einen generischen Befehl, der primär auf die technischen Aspekte eingeht, nämlich eine Adresse abzuändern – die echte Intention dahinter, im Sinne der Abbildung der Realität gemäß Domain-Driven Design, bleibt dabei jedoch verborgen. Das folgende Beispiel verwendet die gleichen Objekteigenschaften, dafür jedoch einen aussagekräftigeren Namen. Innerhalb des Domain Models ist somit klar, dass ein Kunde den Wohnsitz verlagert und deswegen die Adresse ändern möchte.

```
1.  class CustomerIsMoving implements DomainCommand
2.  {
3.      private $customerId;
4.      private $address;
5.      public function __construct($customerId, Address $address)
6.      {
7.          $this->customerId = $clientId;
8.          $this->address = $address;
9.      }
10.     public function getCustomerId() { /* ... */ }
11.     public function getAddress() { /* ... */ }
12. }
```

Programm 6: Beispiel für einen aussagekräftigen Befehl (eigene Darstellung)

Befehle werden von einer Kontrollinstanz erzeugt, dort mit den relevanten Attributen versehen und schließlich an den *Command Bus* zur weiteren Verarbeitung delegiert.

4.2.2 Command Bus

Die Aufgabe des *Command Bus* ist es, Befehle und deren Verarbeitung lose zu koppeln. So ist die verarbeitende Schicht nicht fest in der Logik eines Controllers verankert, sondern kann im sogenannten *Command Handler* an zentraler Stelle ausgetauscht werden. Der *Command Bus* stellt dazu eine TYPO3-spezifische Fassade bereit, welche wiederum eine externe Bibliothek[22] erweitert.

```
1. CommandBus::provide()->addHandler(
2.     CustomerIsMovingCommandHandler::instance(),
3.     CustomerIsMoving::class
4. );
```

Programm 7: Separate Zuweisung eines Befehls an einen Command Handler (eigene Darstellung)

Die Verarbeitung selbst kann wahlweise in einer separaten Klasse für einen bestimmten Befehl erfolgen, oder auch in einem Paket (*Bundle*) zentralisiert werden – Letzteres ist für die Wiederverwertbarkeit von ähnlichen Ablaufschritten von Vorteil.

```
1. CommandBus::provide()->addHandlerBundle(
2.     CustomerCommandHandlerBundle::instance(), [
3.         CustomerIsMoving::class,
4.         CustomerAddsBankAccount::class,
5.     ]
6. );
```

Programm 8: Gesammelte Zuweisung an ein Command Handler Bundle (eigene Darstellung)

4.2.3 Domain Model

Innerhalb eines Command Handlers wird eine betroffene Entität über ein entsprechendes Event Repository durch das Abspielen aller bisherigen Ereignisse erzeugt. Im Anschluss wird der Befehl an die erstellte Entität zur Verarbeitung weitergeleitet – erst dann kommen Anwendungslogik und Regelwerke zum Tragen.

[22] Composer Paket „league/tactician“, siehe https://github.com/thephpleague/tactician

```
1. class CustomerIsMovingCommandHandler implements CommandHandler
2. {
3.     public function handle(CustomerIsMoving $command)
4.     {
5.         $repository = CustomerEventRepository::instance();
6.         $customer = $repository->findByCustomerId(
7.             $command->getClientId()
8.         );
9.         $customer->moveTo(
10.            $command->getAddress()
11.        );
12.        $repository->commit($customer);
13.    }
14. }
```

Programm 9: Verarbeitung eines Befehls durch einen Command Handler (eigene Darstellung)

Falls der Befehl erfolgreich angewandt werden konnte, wird innerhalb des Objekts ein zugehöriges Ereignis erzeugt und zunächst intern aufgezeichnet, ohne es bereits zur Speicherung weiterzureichen. Da mehrere Events bei der Verarbeitung eines Befehls auftreten können, empfiehlt es sich, diese zunächst zu sammeln und erst im Anschluss an ein *Event Repository* zu übergeben – dort werden nur die Ereignisse einer Entität gespeichert nicht aber die Entität selbst[23].

Aktuell werden gleiche Merkmale (*Traits*) von Befehlen und Ereignissen über das gleichnamige PHP-Konstrukt ausgelagert. Somit können konkrete Implementierungen Methoden wiederverwerten, die bei der Verarbeitung von Commands und Events benötigt werden. Dieses Sprachkonstrukt wird jedoch in Entwicklerkreisen stark kontrovers diskutiert, da es die Paradigmen der objektorientierten Programmierung und des Domain-Driven Designs aufweicht.

[23] das sollte nicht mit dem Aufruf *Repository::add($entity)* bei Extbase verwechselt werden – *EventRepository::commit($entity)* bezieht sich ausschließlich auf neue Ereignisse einer Entität

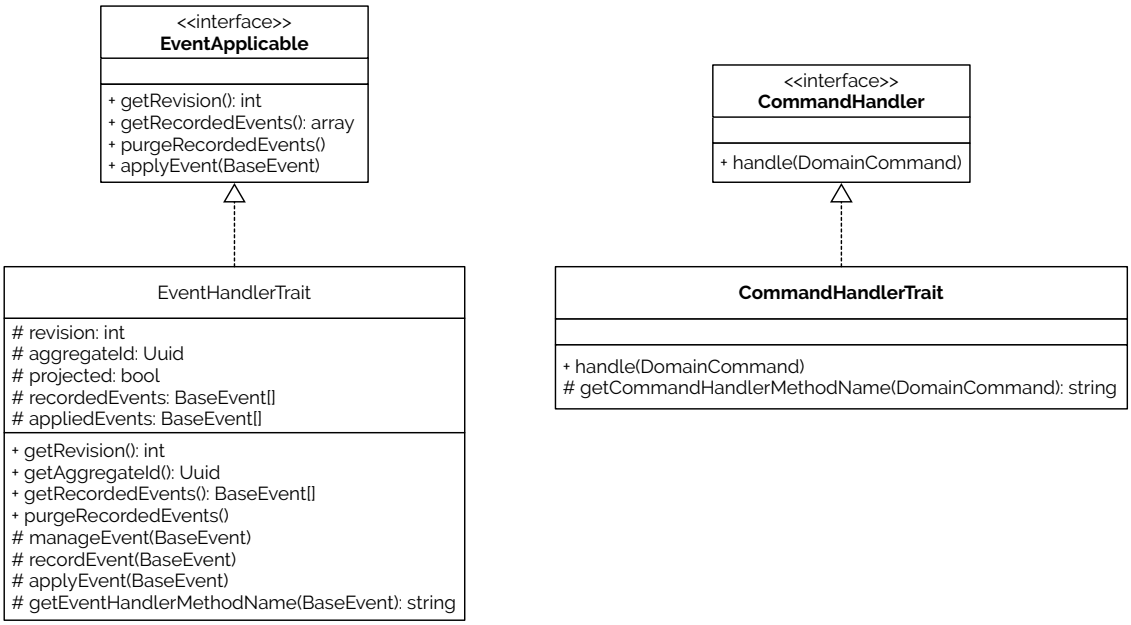

Abbildung 23: Traits für Commands und Events (eigene Darstellung)

4.2.4 Events

Ereignisse beziehen sich auf Änderungen, die im Hinblick auf ein Domain Model tatsächlich ausgeführt wurden und welche auch zu einem späteren Zeitpunkt erneut angewandt werden können. Um ein Objekt semantisch als Event auszuzeichnen, muss das Interface *Domain Event* implementiert werden – im vorliegenden Kontext muss zusätzlich das *Base Event* Interface verwendet werden, welches grundlegende Voraussetzungen für die Serialisierung und Deserialisierung, sowie hinsichtlich der Speicherung in einem Event Store definiert.

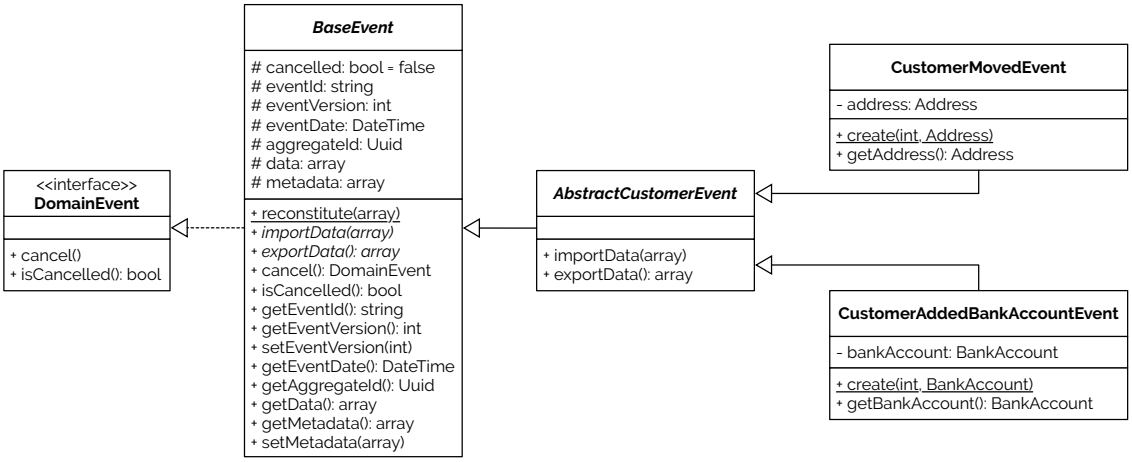

Abbildung 24: Klassendiagramm der Event Hierarchie (eigene Darstellung)

Die Abbildung zeigt die Vererbungshierarchie von Events – die abstrakte Ereignisklasse für Kunden-Objekte ist optional und kapselt Funktionalitäten, die zum Export bzw. Import von Daten hinsichtlich der Serialisierung benötigt werden.

```
1.  class Customer {
2.      // ...
3.      public function moveTo(Address $address)
4.      {
5.          if ($this->address->equals($address)) {
6.              return;
7.          }
8.          $this->manageEvent(
9.              CustomerMovedEvent::create($this->id, $address)
10.         );
11.     }
12.     // ...
```

Programm 10: Anwenden von Regeln und Aufzeichnung eines Ereignisses (eigene Darstellung)

Innerhalb der Kunden-Entität wird die angeblich neue Adresse des Befehls mit der aktuell gespeicherten Adresse des Kunden verglichen – sind die Werte identisch, liegt keine Adressänderung vor und die Verarbeitung ist beendet. Sind die Adressdaten dagegen unterschiedlich, wird ein entsprechendes Ereignis aufgezeichnet und auf das aktuelle Objekt angewandt – die gezeigte Methode erledigt dies implizit[24].

Das folgende Beispiel zeigt das Anwenden eines Ereignisses innerhalb einer bestehenden Entität. Der Zugriff von außen erfolgt über eine einheitliche Methode zum Ausführen des Ereignisses – diese Methode ist ebenfalls in der Schnittstelle implementiert, gegen die Klassen implementieren müssen, die Ereignisse verarbeiten.

[24] die Methode *manageEvent()* ruft intern die Methoden *recordEvent()* und *applyEvent()* auf

```
1. class Customer {
2.     // ...
3.     public function applyEvent(BaseEvent $event)
4.     {
5.         if ($event instanceof CustomerMovedEvent) {
6.             $this->applyCustomerMovedEvent($event);
7.         }
8.         // ...
9.     }
10.    private function applyCustomerMovedEvent(
11.        CustomerMovedEvent $event
12.    ) {
13.        $this->address = $event->getAddress();
14.    }
15.    // ...
16. }
```

Programm 11: Anwenden eines Ereignisses auf eine Entität (eigne Darstellung)

4.2.5 Event Repository

Das *Event Repository* wird durch den *Command Handler* aufgerufen, um zunächst den aktuellen Zustand einer Entität aus dem *Event Store* abzuleiten, um darauf einen Befehl anzuwenden, sowie im weiteren Verlauf neue Ereignisse wieder abzuspeichern.

Das *Event Repository* ist damit im Allgemeinen verantwortlich für den Zugriff auf speicherbare Ereignisse eines bestimmten Entitätstyps. In der vorliegenden Implementierung stößt diese Komponente zusätzlich konfigurierte Projektionen für bestimmte Events an.

Die in der Entität neu gesammelten Ereignisse werden sequentiell an den Event Store weitergeleitet – erst dort erfolgt die tatsächliche persistente Speicherung. Alle Ereignisse einer bestimmten Entität werden unter einem eindeutigen Namen abgelegt, dieser besteht im folgenden Beispiel aus dem Typ der Entität (*Customer*), sowie der UUID der vorliegenden Objektinstanz.

```php
class CustomerEventRepository implements EventRepository
{
    // ...
    public function commit(Customer $customer)
    {
        foreach ($customer->getRecordedEvents() as $event) {
            $this->commitEvent($event);
        }
        ProjectionManager::provide()->projectEvents(
            $customer->getRecordedEvents()
        );
        $customer->purgeRecordedEvents();
    }
    public function commitEvent(BaseEvent $event)
    {
        $streamName = 'Customer/' . $event->getAggregateId();
        EventStorePool::provide()->getDefault()
            ->attach($streamName, $event);
    }
    // ...
```

Programm 12: Persistente Speicherung von Events mittels Event Store (eigene Darstellung)

Nachdem neue Ereignisse im Event Store abgespeichert wurden, löst das Event Repository den Projektionsvorgang aus. Erst zu diesem Zeitpunkt wird der Zustand der Entität für den Lesezugriff aktualisiert (siehe Abschnitt 4.2.7).

```php
class CustomerEventRepository implements EventRepository
{
    // ...
    public function findByCustomerId($customerId) {
        $streamName = 'Customer/' . (string)$customerId;
        $eventSelector = EventSelector::instance()
            ->setStreamName($streamName);
        $customer = Saga::create($eventSelector)
            ->tell(Customer::instance());
        return $customer;
    }
    // ...
}
```

Programm 13: Zustand einer Entität aus den gespeicherten Ereignissen ableiten (eigene Darstellung)

Bei der Rekonstitution wird der Name des Event Streams in einer Klasse gekapselt, welche die Selektion von bestimmten Ereignissen (*Event Selector*) vereinheitlicht. Das Konzept dieses Selektors wurde zwar definiert, aber in der Implementierungsphase nur

an vereinzelten Stellen eingesetzt. Diese Komponente ist ähnlich aufgebaut wie CSS-Selektoren, zur Ermittlung von DOM-Elementen, und besteht aus dem Namen des Event Streams, den Namen von Kategorien und den zu selektierenden Ereignisarten. Dabei muss mindestens einer dieser drei Aspekte angegeben sein.

```
1. $eventSelector = EventSelector::create(
2.     '$Customer/*.Customer.Address[CustomerMovedEvent]'
3. );
```

Programm 14: Beispielhafte Verwendung des Ereignisselektors (eigene Darstellung)

Das angeführte Beispiel würde aus dem *Event Store* alle Umzugsereignisse von beliebigen Kundenentitäten ermitteln, welche der Kundenkategorie oder Adresskategorie zugeordnet wurden.

```
EventSelector =     StreamPart   { CategoryPart } { EventPart }
                  | [ StreamPart ]   CategoryPart   { EventPart }
                  | [ StreamPart ] { CategoryPart }   EventPart
StreamPart    = "$" Literals [ "/*" ]
CategoryPart  = "." Literals { "." Literals }
EventPart     = "[" Literals { "," Literals } "]"
Literals      = ALPHA { ALPHA | DIGIT | ":" | ";"  | "-" | "_" }
```

Abbildung 25: Ereignisselektor als angereicherte Backus-Naur-Form (eigene Darstellung)

4.2.6 Event Store

Da der *Event Store* für die persistente Speicherung und Abfrage von Ereignissen zuständig ist, kann dieser als das Herzstück aller Event Sourcing Abläufe betrachtet werden. Über eine Sammlung (*Event Store Bundle*), welche unterschiedliche Event Store Instanzen in einem Iterator aggregiert, können Events gleichzeitig durch mehrere Speicherformen abgelegt werden. Aktuell sind die Speichervarianten MySQL und GetEventStore durch separate Treiber implementiert.

Bei der Abfrage von Ereignissen werden Resultate in einem Event Stream gekapselt – dieses Objekt implementiert das *Iterator* Interface. Somit wird vermieden, dass schon vor der eigentlichen Verarbeitung alle Daten geladen und zu Event Objekten konvertiert werden müssen – diese Vorgehensweise hat positive Auswirkungen auf die Performance und den Speicherverbrauch.

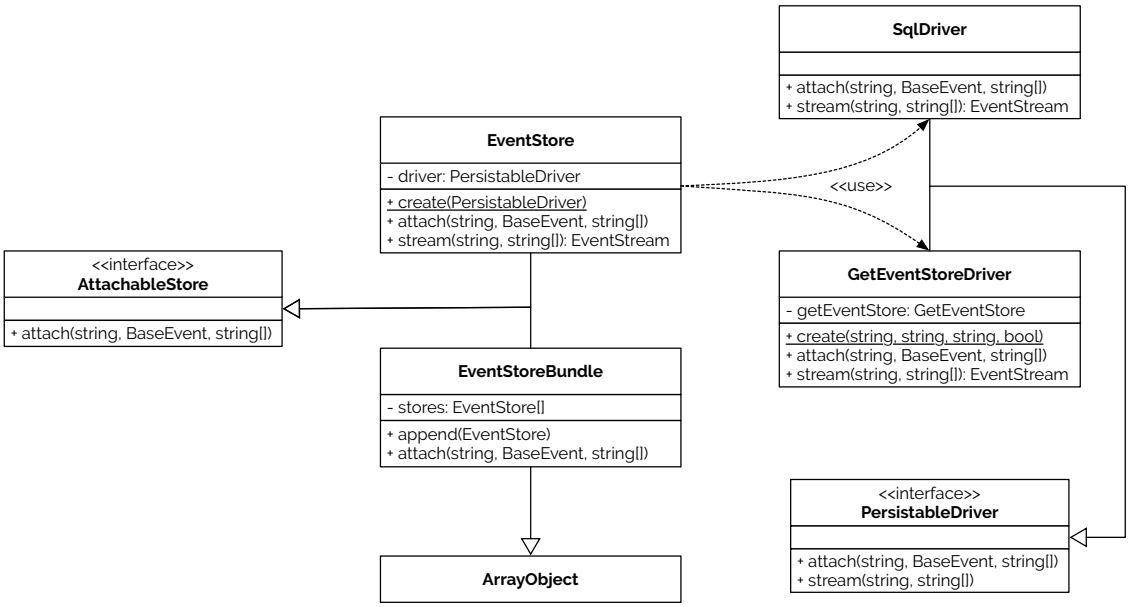

Abbildung 26: Klassendiagramm der Event Store Schicht (eigene Darstellung)

Event Stores werden bei Ausführungsbeginn der Applikation (*Bootstrap*) konfiguriert und registriert. Die nachfolgenden Beispiele veranschaulichen dies für die beiden vorhandenen Treiber.

```
1. EventStorePool::provide()
2.     ->enrolStore('sql')
3.     ->concerning('*')
4.     ->setStore(
5.         EventStore::create(
6.             SqlDriver::instance()
7.         )
8.     );
```

Programm 15: Registrierung eines MySQL Event Store (eigene Darstellung)

```
1. EventStorePool::provide()
2.     ->enrolStore('geteventstore.com')
3.     ->concerning('*')
4.     ->setStore(
5.         EventStore::create(
6.             GetEventStoreDriver::create('http://127.0.0.1:2113')
7.         )
8.     );
```

Programm 16: Registrierung eines GetEventStore Event Stores (eigene Darstellung)

Zur Bewertung der Performance werden die beiden vorhanden Treiber miteinander verglichen. Für die Messung der Verarbeitungszeit werden zwei funktionsbezogene Testfälle implementiert, die sich jedoch nur durch die Art der Speicherung unterscheiden und sonst identisch ablaufen. Im Rahmen der Tests werden insgesamt jeweils 5.000 Events erzeugt, entsprechend persistiert und anschließend wieder durch den Event Store eingelesen.

Im Vorfeld werden die Speichervorgänge der MySQL Datenbank auf Optimierungsmöglichkeiten analysiert. In der Ausgangssituation sind keinerlei Indizes auf Tabellenspalten hinterlegt. Während der Speicherung von Events wird die Versionsnummer für gleichnamige Streams automatisch inkrementiert. Die hierzu verwendete Abfrage liefert bei der Analyse folgende Resultate.

```
1. EXPLAIN SELECT event_version
2. FROM sys_event_store
3. WHERE event_stream = 'SqlPerformanceTest/0f0012cd-2a64-4e3a-...'
4. ORDER BY event_version DESC LIMIT 1;
```

Programm 17: Abfrageanalyse einer lesenden MySQL Anweisung (eigene Darstellung)

Id	1
select_type	SIMPLE
Table	sys_event_store
type	ALL
possible_keys	NULL
key	NULL
key_len	NULL
ref	NULL
rows	5006
Extra	Using where; Using filesort

Abbildung 27: Index-Analyse nicht optimierter Event Store Abfragen (eigene Darstellung)

Unter der Verwendung eines Primärschlüssels für die Tupel *event_stream* und *event_version* ergibt sich folgendes Resultat der Abfrageanalyse. Anstatt die Abfragebedingung auf alle Einträge anzuwenden, kann über den Primärschlüssel das relevante Ergebnis direkt ermittelt werden.

id	1
select_type	SIMPLE
table	sys_event_store
type	ALL
possible_keys	PRIMARY
key	PRIMARY
key_len	386
ref	const
rows	1
Extra	Using where; Using index

Abbildung 28: Index-Analyse optimierter Event Store Abfragen (eigene Darstellung)

Die Gegenüberstellung zeigt einen Geschwindigkeitsvorteil der optimierten MySQL Speicherart gegenüber GetEventStore – diese Abweichung ist hier weitgehend auf die Latenz der vielen HTTP-Anfragen zurückzuführen.

Durchlauf	**MySQL**		**MySQL (optimiert)**		**GetEventStore**	
	write	*read*	*write*	*read*	*write*	*read*
#1	24,95 s	0,38 s	5,93 s	0,41 s	10,36 s	6,34 s
#2	24,06 s	0,42 s	5,88 s	0,39 s	10,18 s	6,38 s
#3	23,91 s	0,41 s	6,22 s	0,38 s	10,26 s	6,25 s
#4	23,83 s	0,39 s	6,12 s	0,41 s	10,51 s	6,32 s
#5	23,84 s	0,42 s	5,90 s	0,39 s	10,28 s	6,20 s
Durchschnitt	**24,12 s**	**0,40 s**	**6,01 s**	**0,40 s**	**10,32 s**	**6,30 s**
Durchschnitt gesamt	**24,52 s**		**6,41 s**		**16,72 s**	

Abbildung 29: Performance-Vergleich von Event Store Treibern (eigene Darstellung)

4.2.7 Projektionen

Projektionen werden für ein bestimmtes Ereignis aufgerufen, mit der Absicht daraus eine abfragefähige Datenbasis zu erstellen. Die Repräsentation der Information ist dabei frei und nicht an eine vollständige Transformation eines Objekts des Write Models gebunden. Weiterhin ist die Art der Speicherung flexibel wählbar – neben der Persistenz in einer Datenbank, ist also auch eine Projektion in das Dateisystem denkbar, beispielsweise direkt im HTML-Format.

Bei der Implementierung von Projektionen sollen die tatsächlich benötigten Informationen berücksichtigt werden und eine optimale Variante im Sinne des Materialized View Paradigmas angestrebt werden. Bei der Interaktion mit einer

Speicherart ist ein separates Repository empfehlenswert, um Vorgänge zu kapseln und austauschbar zu gestalten. Die Instanziierung von Datentransferobjekten des Read Models kann ebenfalls durch dieses Repository erfolgen.

```
1. ProjectionManager::provide()->registerProjection(
2.     new CustomerMovedProjection()
3. );
```

Programm 18: Registrierung einer Projektion (eigene Darstellung)

Projektionen werden global registriert – die Information, welche Ereignisse verarbeitet werden können, wird dabei von der Projektion selbst bereitgestellt. Auch ist es möglich, dass unterschiedliche Projektionen auf ein gemeinsames Event reagieren – über die Information, dass ein bestimmtes Event abgebrochen wurde, kann den nachfolgenden Projektionen mitgeteilt werden, dieses Ereignis zu ignorieren.

```
1. class CustomerMovedProjection implements Projection
2. {
3.     public function listensTo()
4.     {
5.         return [CustomerMovedEvent::class];
6.     }
7.     public function project(CustomerMovedEvent $event)
8.     {
9.         $address = $event->getAddress();
10.        $tableName = 'tx_customer_address';
11.        $connection = ConnectionPool::instance()
12.            ->getConnectionForTable($tableName);
13.        $connection->update(
14.            $tableName,
15.            [
16.                'country' => $address->getCountry(),
17.                'street' => $address->getStreet(),
18.                'city' => $address->getCity(),
19.                'zip' => $address->getZip(),
20.            ],
21.            [
22.                'id' => $event->getAggregateId(),
23.            ]
24.        );
25.    }
26. }
```

Programm 19: Projektion eines Ereignisses zur Adressänderung (eigene Darstellung)

Im vorhergehenden Beispiel wird eine Adressänderung direkt in eine Datenbanktabelle projiziert. Die Implementierung geht dabei davon aus, dass ein entsprechender Datensatz vorhanden ist. Zusätzlich ist es möglich, dass eine Projektionsklasse auf mehrere Ereignisse reagiert.

Mit der Ausführung der im Beispiel dargestellten Projektion wird der eigentliche Zustand der Entität eines Kunden tatsächlich für den Lesezugriff aktualisiert. Damit endet die in den vorhergehenden Abschnitten beleuchtete Prozesskette aus Befehlen, Ereignissen und Projektionen.

4.3 Generische Datenverarbeitung

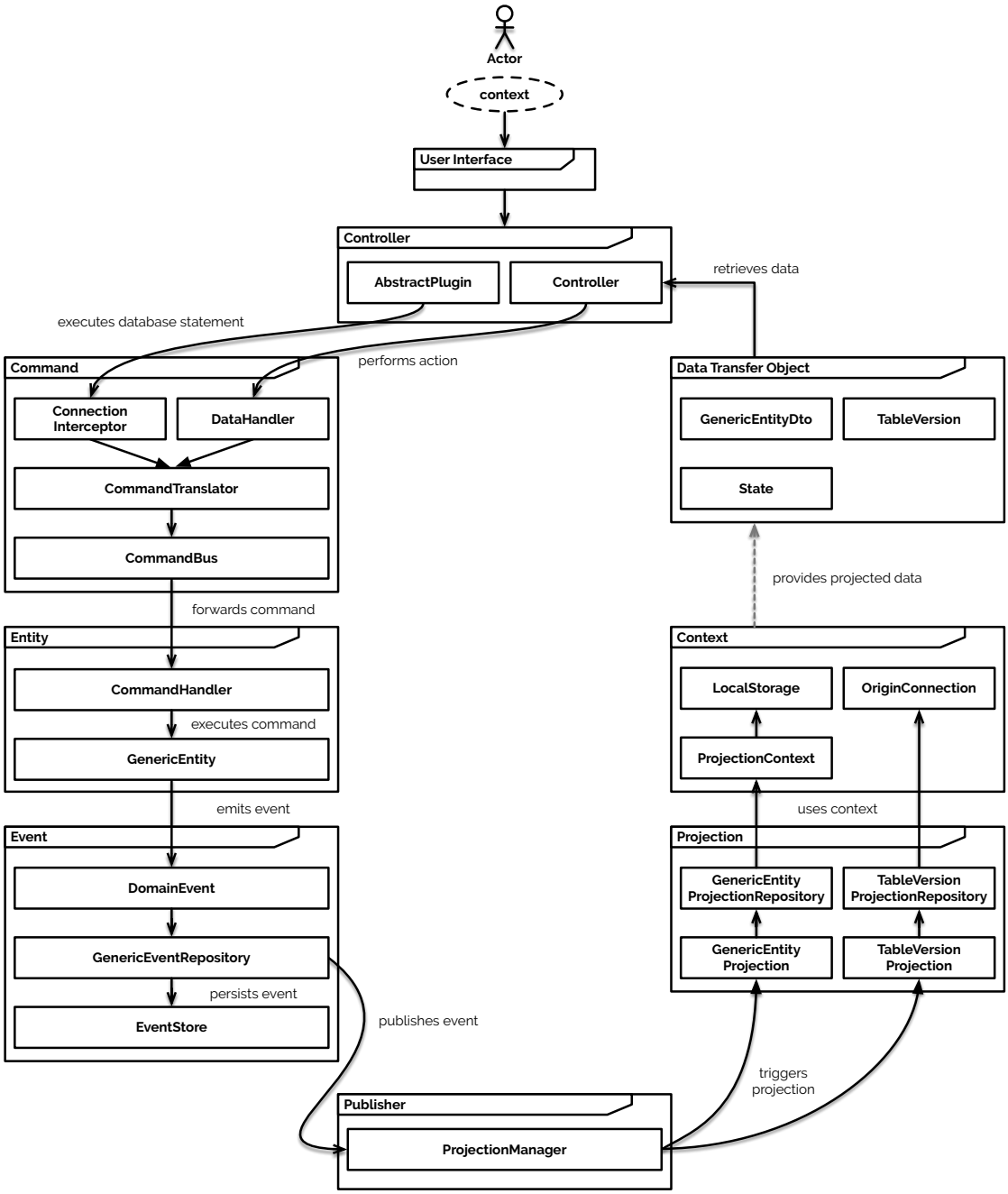

Abbildung 30: Übersicht der Komponenten generischer Datenstrukturen (eigene Darstellung)

Die Herausforderung bei der Handhabung von generischen Datenstrukturen, die nicht im Domain Model verfügbar sind und deren Zuständigkeiten sich über mehrere Schichten und Pakete verteilen, besteht darin, die Prinzipien von Event Sourcing im Kern zu integrieren und dennoch für alle beteiligten Ausgabekomponente kompatibel zu

bleiben. Weiterhin sind Mechanismen zu schaffen, welche die direkte Manipulation von Informationen auf Datenbankebene entweder untersagen oder alternativ in eine geeignete Ereignisstruktur transformieren.

4.3.1 Domain Model

Die konkrete Bedeutung des generischen Domain Models ergibt sich erst aus der Kombination aller Eigenschaften zur Laufzeit – dieses Modell kann sich beispielsweise auf ein Inhaltselement oder auch auf eine Kategorie beziehen. Jedoch gilt es, diesen Widerspruch zugunsten einer allgemeinen Umsetzung zu akzeptieren und die resultierenden Konsequenzen in Kauf zu nehmen. Die folgende Darstellung zeigt die generische Verteilung der Attribute – *Generic Entity* stellt dabei die zentrale Klasse dar, welche bei dieser Implementierungsart beliebige Datensätze abbilden kann.

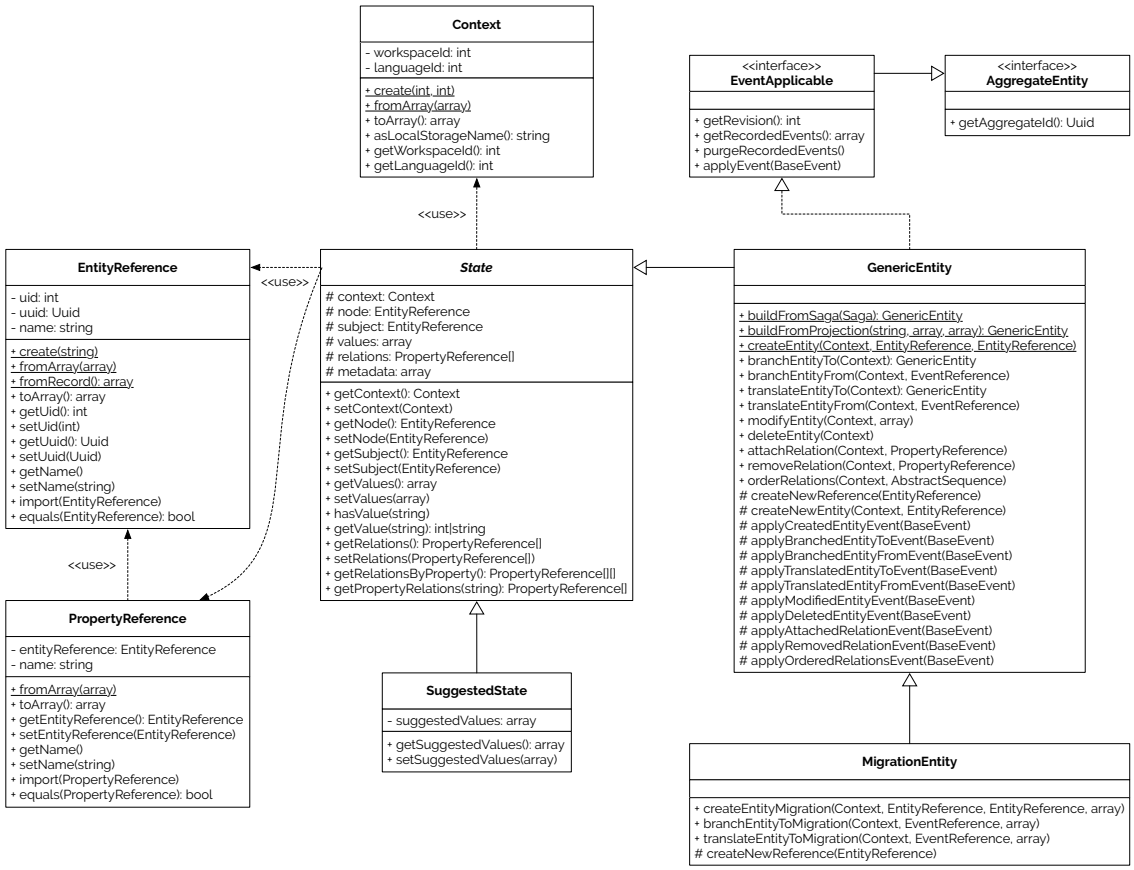

Abbildung 31: GenericEntity Klassendiagramm (eigene Darstellung)

Dem allgemeinen Zustand einer Entität (*State*) ist jeweils immer eine Seite (*Node*) zugeordnet und besteht ferner aus atomaren Werten (*Values*) und allgemeinen Relationen (*Relations*) zu anderen Entitäten. Eine Kombination aus Datenbanktabelle, UID und UUID sind die Attribute, die eine Entität in diesem Zusammenhang identifizieren. Der UUID-Wert wird dabei einmalig automatisch vergeben und ebenfalls in der Datenbank abgespeichert, damit ist jederzeit eine Transformation zwischen der alten UID und der neuen UUID möglich – beispielsweise wird das im Frontend eingesetzte TypoScript noch immer auf ganzzahlige Werte der UID aufbauen.

Ein vorgeschlagener Zustand (*Suggested State*) findet Anwendung, wenn ein Benutzer über die Eingabeformulare des Backends Datenänderungen einreicht – die Übermittlung erfolgt dabei immer als komplette Datenstruktur mit allen Werten und allen Relationen. Erst bei der Verarbeitung durch den *Data Handler* wird ermittelt, welche Werte sich tatsächlich verändert haben.

Innerhalb der generischen Entität (*Generic Entity*) finden sich die für CQRS und Event Sourcing notwendigen Verarbeitungsschritte durch Befehle und Ereignisse. Für die einmalige Erzeugung von Ereignissen während des Migrationsprozesses werden zusätzlich spezifische Verhaltensweisen in einer abgeleiteten Klasse (*Migration Entity*) implementiert (siehe Abschnitt 4.3.3).

4.3.2 Commands & Events

Hinsichtlich der durch einen Redakteur ausführbaren Aktionen, wurde im generischen TYPO3-Kontext folgendes Vokabular für Befehle und gleichnamige Ereignisse als Grundlage ermittelt – alle Aspekte sind dabei jeweils an einen bestimmten Kontext gebunden, einer Kombination aus Workspace und Sprache.

- *create*: Erzeugen eines Datensatzes
- *branch to*: Ableiten einer Workspace Version von der aktuellen Entität
- *branch from*: Erzeugen einer Workspace Version von einer anderen Entität
- *translate to*: Ableiten einer Übersetzung von der aktuellen Entität
- *translate from*: Erzeugen einer Übersetzung von einer anderen Entität

- *modify*: Abändern der atomaren Werte der aktuellen Entität
- *delete*: Löschen einer Entität
- *attach relation*: Hinzufügen einer Relation zur aktuellen Entität
- *remove relation*: Entfernen einer Relation der aktuellen Entität
- *order relations*: Ändern der Reihenfolge von Relationen der aktuellen Entität

4.3.3 Initialisierungsprozess

Um das Event Sourcing Konzept innerhalb einer TYPO3 Installation erstmalig einführen zu können, ist es notwendig einen Ausgangszustand von gespeicherten Ereignissen herzustellen. Dazu werden alle TCA-Datensätze analysiert und die Events erzeugt, die zum gespeicherten Zustand geführt haben. Dieser Vorgang ist eine Momentaufnahme zum jeweiligen Zeitpunkt und lässt die tatsächlichen Abläufe in der Vergangenheit teilweise unberücksichtigt – beispielsweise ist daraus das mehrmalige Abändern des Titels eines Inhaltselements nicht ableitbar.

Hauptsächlich liegt in diesem Schritt die Aufteilung zwischen einfachen Werten und Assoziationen, sowie die Unterscheidung nach Workspace und Sprache im Fokus.

Als Resultat wird der Event Store mit Ereignissen angereichert, die wiederum dazu verwendet werden können, den eingangs verwendeten Datenbankeintrag herzustellen.

4.3.4 Data Handler

Der *Data Handler* stellt die Kernkomponente im TYPO3 Backend dar, die für die persistente Speicherung von redaktionellen Änderungen verantwortlich ist. Über einen Hook wird die ursprüngliche Verarbeitung dahingehend angepasst, dass Modifikationen für konfigurierte TCA-Datenbanktabellen abgefangen und in Befehle konvertiert werden – ergo wird dem *Data Handler* somit die Zuständigkeit für bestimmte Tabellen entzogen. Für nicht konfigurierte Datenbanktabellen, beispielsweise von separaten Extensions, erfolgt die Verarbeitung ohne Einschränkungen wie bisher.

```
1. $GLOBALS['TCA']['tt_content']['ctrl']['eventSourcing'] = [
2.     'listenEvents' => true,
3.     'recordEvents' => true,
4.     'projectEvents' => true,
5. ];
```

Programm 20: Aktivierung von Event Sourcing für die Tabelle tt_content (eigene Darstellung)

Über die abgebildete Konfiguration mittels TCA lässt sich definieren, welche Prozesse hinsichtlich Ereignissen für eine bestimmte Datenbanktabelle ausgeführt werden sollen.

- *listenEvents*: nicht ereignisabhängige Aktionen[25] sollen in entsprechende Ereignisse und Befehle überführt werden – individuelle Command Handler sind dann in der Lage, diese weiter zu verarbeiten
- *recordEvents*: generierte Befehle werden an die generische Entität übergeben und dort für den Event Store aufgezeichnet
- *projectEvents*: generierte Ereignisse werden an die generische Projektion weitergeleitet, um den Zustand der Datenbanktabellen zu aktualisieren

Der neue Transformationsschritt des *Data Handlers* ermittelt zunächst die Unterschiede von Werten und Relationen zwischen dem im Event Repository hinterlegten Zustand (*Generic Entity*) und dem durch den Redakteur gewünschten Zustand (*Suggested State*). Mögliche Differenzen werden in entsprechende Befehle transformiert und zur Verarbeitung an die generische Entität weitergereicht – dort werden schließlich Ereignisse erzeugt und durch den Command Handler an das Event Repository übergeben (siehe Abschnitt 4.2 bzgl. der Erläuterungen des allgemeinen Event Sourcing Zyklus).

Das für generische Entitäten zuständige Event Repository stößt nach der Speicherung der Ereignisse im *Event Store* unmittelbar die Projektion an – damit resultieren Ereignisse direkt in eine Aktualisierung der Datenbanktabellen.

[25] ereignisunabhängige Aktionen beziehen sich dabei auf ursprüngliche Eingabewerte an den DataHandler

4.3.5 Local Storage

Bevor die erzeugten Ereignisse an die Projektionsschicht weitergeleitet werden, ist hinsichtlich des Anwendungskontexts des Gesamtsystems noch eine Erweiterung der Persistenzschicht vorzunehmen. Jeder möglichen Kombination aus einem Workspace und einer Sprache wird eine neu geschaffene lokale Speichervariante (*Local Storage*) zugeordnet, welche in der prototypischen Implementierung eine separate Speicherung im lokalen Dateisystem auf der Basis von SQLite darstellt. Damit ist es möglich, dynamisch Speicherorte zu generieren, die ebenfalls über SQL abgefragt werden können. Der Einsatz der Datenbankabstraktionsschicht sorgt dafür, dass Local Storages transparent in die Datenbankprozesse von TYPO3 integriert werden können. In der Konsequenz erhält so jeder Kontext, bestehend aus Workspace und Sprache, eine eigene individuelle Datenbank.

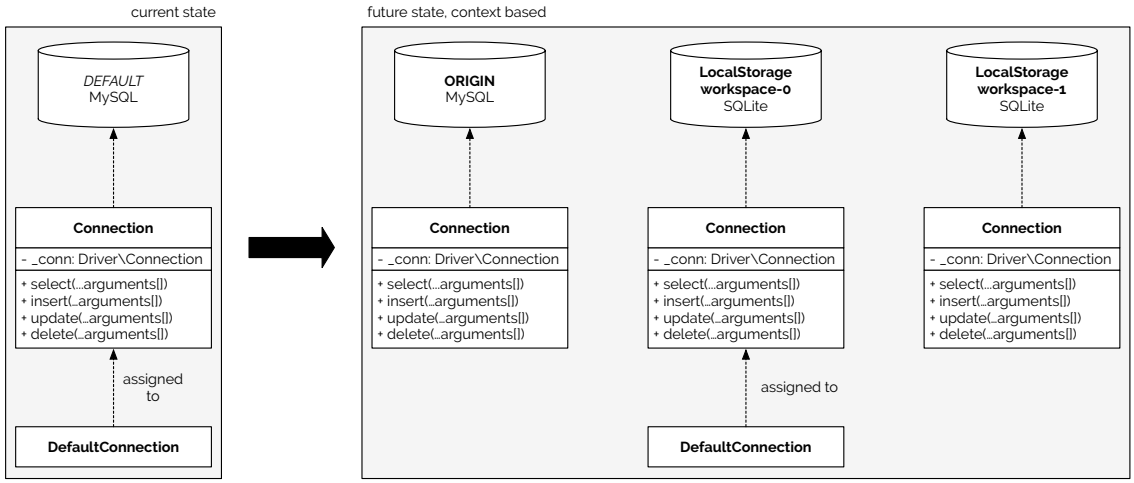

Abbildung 32: Überführung in kontextabhängige Speicherformen (eigene Darstellung)

Während der Initialisierungsphase jeder HTTP-Anfrage an die TYPO3 Anwendung wird somit die eigentliche echte Datenbank durch eine kontextabhängige lokale Speichervariante ausgetauscht. Projektionen sind dafür verantwortlich, Informationen für den jeweiligen Kontext zu schreiben – direkte Manipulationen in der Datenbank werden transparent[26] innerhalb der ausgewählten *Local Storage* Instanz umgesetzt.

[26] unter der Einschränkung, dass keine DBMS-spezifischen Anweisungen verwendet werden, die durch SQLite nicht umgesetzt werden können – der Einsatz von Doctrine DBAL löst dieses Problem jedoch auf

Standardmäßig wird in TYPO3 nur eine Datenbankverbindung definiert (*Default Connection*). Nach Anwendung des Local Storage Konzepts wird für die Auswahl der Standardverbindung die jeweils gültige lokale Speicherform zugewiesen. Jedoch kann in Anwendungen die ursprüngliche Datenbankverbindung (nun *Origin Connection*, vormals *Default Connection*) weiterhin direkt angesprochen werden – dies ist beispielsweise erforderlich, um zentrale Daten des Gesamtsystems, wie Benutzer oder Benutzersitzungen zu verwalten.

Diese Methode lässt sich zwar auch für Sprachen einsetzen, jedoch sind dazu weitere konzeptionelle Änderungen innerhalb TYPO3 notwendig, weswegen die Implementierung von Local Storages aktuell nur Workspace-Varianten unterstützt.

Zur Bewertung der Performance wird eine vollständig im Cache ablegbare Seite im TYPO3 Frontend herangezogen, welche insgesamt 500-mal per HTTP aufgerufen wird, mit jeweils zehn gleichzeitigen Verbindungen. Die Messung wird jeweils für die Speicherformen MySQL und SQLite durchgeführt – die Messwerte zeigen dabei jedoch ähnliche Ergebnisse und keine signifikante Veränderung der Performance.

Durchlauf	**MySQL (Origin)**		**SQLite (LocalStorage)**	
	ø(t)/Req.	*Req./t*	*ø(t)/Req.*	*Req./t*
#1	12,675 ms	78,90 #/s	12,692 ms	78,79 #/s
#2	13,078 ms	76,47 #/s	13,631 ms	73,36 #/s
#3	13,565 ms	73,72 #/s	13,074 ms	76,49 #/s
Durchschnitt	**13,106 ms**	**76,36 #/s**	**13,132 ms**	**76,21 #/s**

Abbildung 33: Vergleich der Web-Server Performance für MySQL und SQLite (eigene Darstellung)

Die Abbildung visualisiert den Vergleich der durchschnittlichen Dauer einer HTTP-Anfrage und der daraus resultierenden maximalen Anzahl an Anfragen, die der Web-Server innerhalb einer Sekunde verarbeiten kann.

4.3.6 Projektionen

Unter Einsatz der im vorhergehenden Abschnitt erläuterten lokalen Speicherformen, werden Ereignisse für den jeweiligen Kontext in die entsprechenden Datenbanktabellen projiziert. Dadurch werden Informationen zwar redundant in mehreren Datenbanken vorgehalten und müssen auch durch eine eigene Logik synchron gehalten werden,

jedoch werden bereits die tatsächlich auszugebenden Inhalte abgespeichert. Für eine Voransicht eines bestimmten Workspaces entfällt somit also die Prüfung, ob Datensätze noch durch weitere Werte von Workspace Versionen überlagert werden müssen. Dieses Resultat entspricht der Idee von Materialized Views (siehe Abschnitt 2.5).

Eine weitere Projektion im Workspace Umfeld sammelt die Anzahl der Versionen für eine bestimmte Datenbanktabelle auf einer bestimmten Seite. Diese Information ist notwendig, um im Seitenbaum des TYPO3 Backend vorliegende Änderungen für den Redakteur farblich Kennzeichnen zu können. In aktuellen TYPO3 Versionen wird diese Information jeweils immer zur Laufzeit ermittelt, was bei einer umfangreichen Seitenstruktur, vielen Datenbanktabellen und mehreren Workspaces zu wahrnehmbaren Einschränkungen der Performance führt.

4.3.7 Testszenarien

Die folgenden Testszenarien wurden während der Implementierungsphase definiert:

- *DataHandlerTranslatorTest*: Es wird geprüft, ob der Aufruf des *Data Handlers* durch eine Schablonenmethode abgefangen wird und dadurch entsprechende Befehle abgeleitet werden können.
- *DataHandlerTest*: Diese Tests erweitern die bereits verfügbaren Testfälle (siehe Abschnitt 3.6) und überprüfen die korrekte Verarbeitung der im *Data Handler* erzeugten Befehle, die Behandlung von Ereignissen bis hin zur Projektion in der ursprünglichen Datenbanktabelle.
- *ConnectionPoolTest*: Dieser Test überprüft, ob ein Kontext, bestehend aus Workspace und Sprache, zu einer *Local Storage* transformiert werden kann und die zu erzeugende SQLite Datenbank tatsächlich im Dateisystem erzeugt wird.
- *RelationMapTest*: Hier wird geprüft, ob vorhandene TCA-Hierarchien hinsichtlich möglicher Referenzen korrekt ermittelt und aufgelöst werden können.
- *EventInitializationUpdateTest*: Dieser Test überprüft, ob der im Install Tool neu geschaffene Upgrade Wizard zur Erzeugung von Events anhand des aktuellen Zustands fehlerfrei ausgeführt werden kann.

- *EventSelectorTest*: Hier wird die korrekte Funktionsweise der Syntaxanalyse und Zerteilung gemäß der vorhandenen angereicherten Backus-Naur-Form des Ereignisselektors sichergestellt (siehe Abschnitt 4.2.5)
- *SortingComparisonServiceTest*: Dieser Testfall stellt sicher, dass der Vergleich von zwei Arrays mit ähnlichen Elementen ermitteln kann, welche Elemente hinzugefügt, weggenommen oder umsortiert wurden.

4.3.8 Konsequenzen

Für konfigurierte Datenbanktabellen werden zukünftig Befehle und Ereignisse erzeugt, die ursprüngliche Verarbeitung durch den *Data Handler* wird somit unterbrochen. In der Konsequenz werden *Data Handler* Hooks nicht mehr ausgeführt und Extensions können auf den Ablauf der Datenverarbeitung keinen unmittelbaren Einfluss mehr nehmen. Im Hinblick auf das aufgestellte Ziel, Nebeneffekte und Fehlerquellen zu vermeiden ist diese Auswirkung jedoch beabsichtigt.

Anstatt Hooks zu verwenden, können Extensions zukünftig in eigenen Projektionen auf Ereignisse reagieren und entsprechende Daten im Sinne von Materialized Views zusammentragen. Alternativ können Befehle auch in eigenständige und domänenspezifische Konstrukte überführt werden (siehe auch Abschnitt 4.4.5).

Zwar müssen dadurch Hooks an den neuen Ablauf angepasst werden, jedoch besteht der Vorteil darin, die ursprünglichen Daten unverändert im Event Store abzulegen und somit stets eine verlässliche Datenquelle verwenden zu können.

4.4 Extbase Bank Account Example

Im Gegensatz zur generischen Datenverarbeitung im vorherigen Abschnitt, wird hier eine besser spezifizierbare Anwendung betrachtet. Diese Umsetzung soll auch zeigen, wie Extbase Anwendungen durch den Einsatz von Event Sourcing Technologien profitieren können.

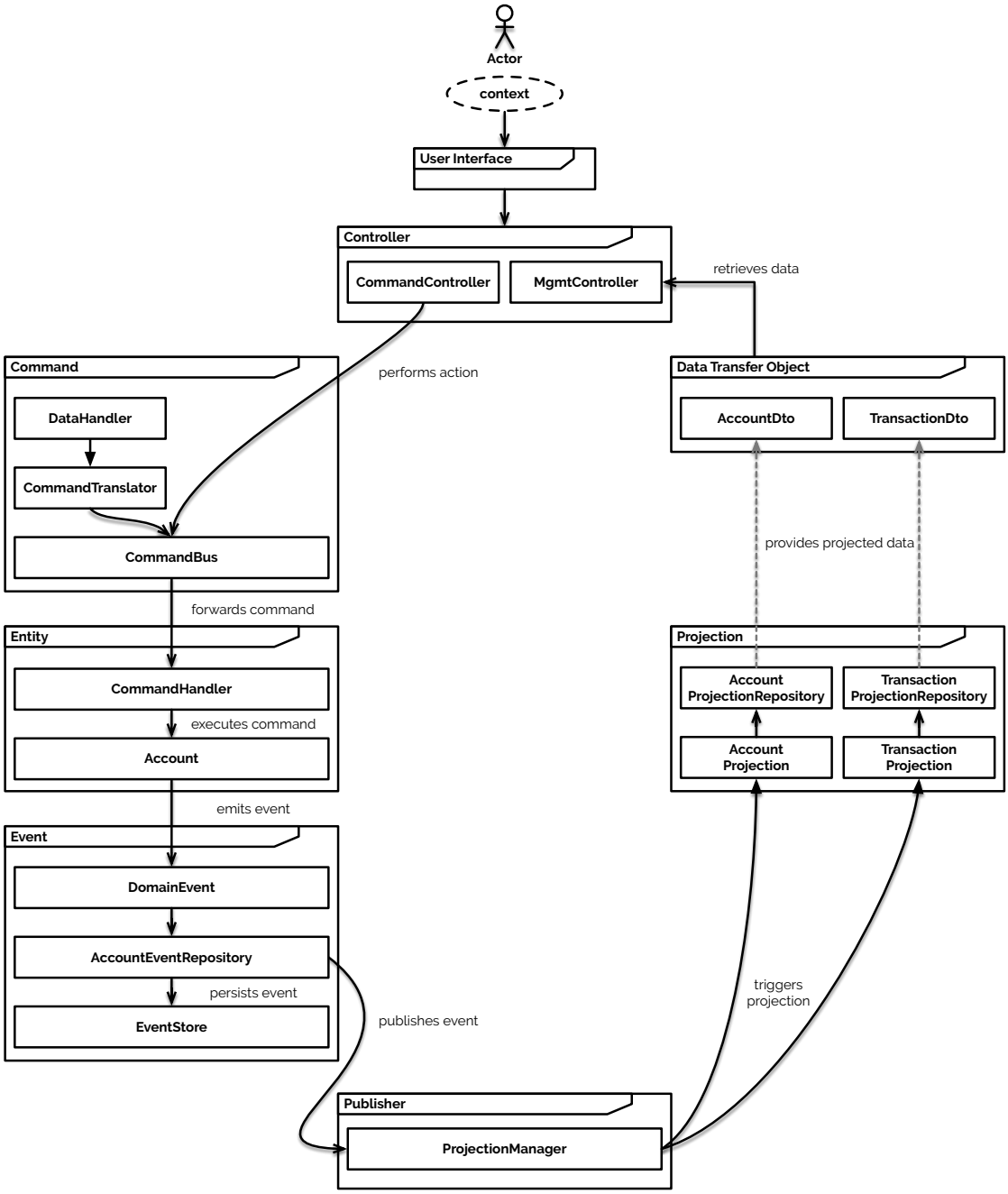

Abbildung 34: Übersicht der Komponenten des Bank Account Examples (eigene Darstellung)

Das Anwendungsbeispiel verwendet als Grundlage einige Aspekte aus dem Bankenumfeld. Kunden können bei einer Bank ein Konto eröffnen und darauf entweder Geld einzahlen oder abheben. Im diesem Beispiel wird jedoch kein Kredit eingeräumt, somit muss das Konto eine ausreichende Deckung hinsichtlich der auszuzahlenden Summe aufweisen. Der zeitliche Verlauf dieser Transaktionsvorgänge soll durch den Kontoinhaber ebenfalls ersichtlich sein. Für den Fall einer Eheschließung sieht die Umsetzung vor, dass der Kunde seinen bei der Bank hinterlegten Namen ändern kann.

Zusätzlich dazu wird eine Verknüpfung mit dem TYPO3 Backend implementiert. Dort soll es Kundenbetreuern der Bank möglich sein, Informationen zu modifizieren – jedoch soll es nur erlaubt sein, neue Transaktionen einzugeben, bestehende Buchungen dürfen auch über das Backend nicht verändert werden. An dieser Stelle werden also die generischen Konzepte aus den vorhergehenden Abhandlungen mit der spezifischen Modellierung des Bank Account Examples verschmolzen.

4.4.1 Domain Model

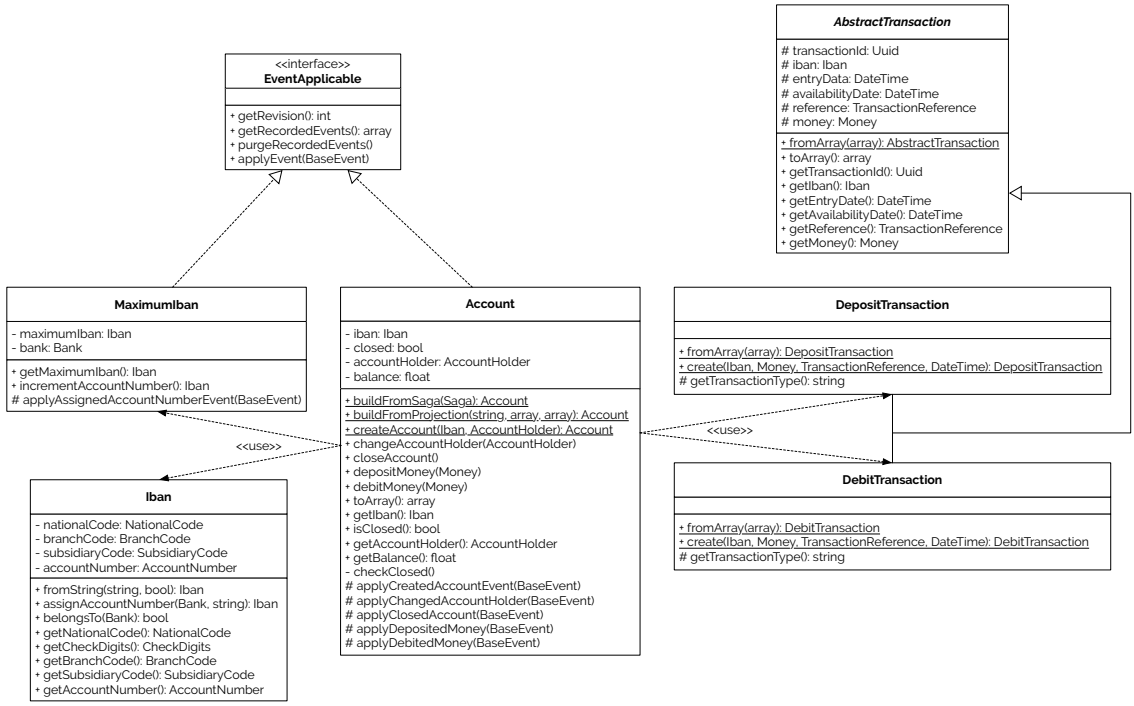

Abbildung 35: Übersicht relevanter Komponenten des Write Models (eigene Darstellung)

Das Domain Model ist in drei Bereiche aufgeteilt. Zum einen wird ein externer Web-Service verwendet, um gültige Bank- und IBAN-Definitionen für ein bestimmtes Land vorzuhalten. Daneben gibt es noch Konzepte für ein Bankkonto (*Account*) und daran gebundene Buchungen (*Transaction*) – das Bankkonto gilt in dieser Anwendung als Aggregate Root, welches alle Zugriffe und Aktivitäten bündelt. Die Definition einer IBAN, der Kontoinhaber sowie ein bestimmter Buchungsbetrag werden jeweils als Wertobjekte umgesetzt.

Beim Anlegen eines neuen Kontos wird über das Objekt *Maximum IBAN* automatisch eine neue Kontonummer vergeben. Aus vergangenen Ereignissen wird die jeweils höchste zugewiesene Kontonummer einer bestimmten Bank ermittelt – nach einem zusätzlichen Inkrement wird daraus schließlich eine neue und gültige IBAN erzeugt.

Die Ermittlung der Befehle und Ereignisse, die durch ein Bankkonto verarbeitet werden wurde unter Einsatz von Event Storming Methoden durchgeführt (siehe Abbildung 7 in Abschnitt 2.7)

4.4.2 Projektionen

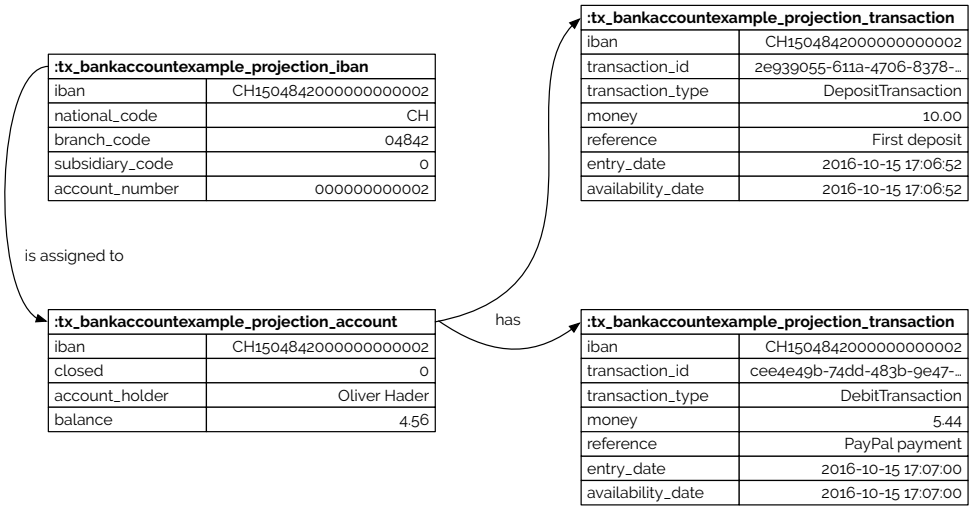

Abbildung 36: Materialized Views des Bank Account Examples (eigene Darstellung)

Die Projektionen sind auf die tatsächlich auszugebenden Ansichten der Anwendung abgestimmt. Die Ansicht der Banken wird dabei direkt aus den zwischengespeicherten Informationen der jeweiligen Web-Services der Nationalbanken abgebildet. Nach

Auswahl einer bestimmten Bank werden die Bankkonten für dieses Institut angezeigt. Im weiteren Verlauf können schließlich die Kontobewegungen eines selektierten Kontos dargestellt werden.

Für die Verwendung der Informationen im Backend werden weitere Projektionen in entsprechende Datenbanktabellen für Bankkonten und Kontobewegungen durchgeführt – diese Tabellen sind ebenfalls mittels TCA konfiguriert.

4.4.3 Datentransferobjekte

Die Interaktion mit der Benutzerschnittstelle hinsichtlich darstellbarer Informationen, wie auch zur Übermittlung von Benutzereingaben an die Anwendung, erfolgt ausschließlich über Datentransferobjekte. Die Eigenschaftswerte dieser Objekte werden über Projektionen bereitgestellt.

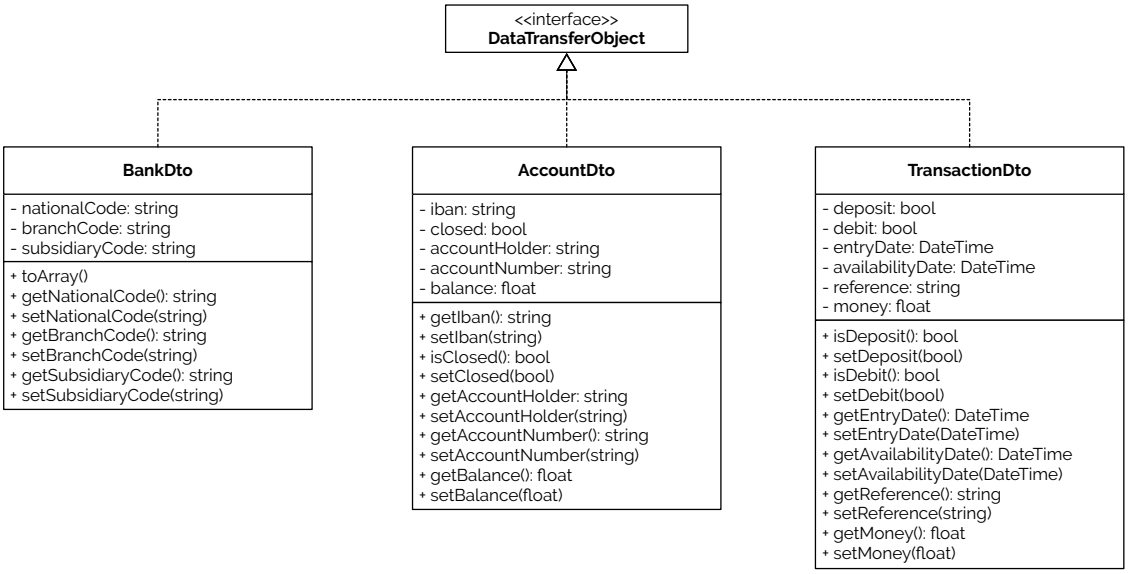

Abbildung 37: Datentransferobjekte des Bank Account Examples (eigene Darstellung)

4.4.4 Controller

Bei der Implementierung der Kontrollinstanzen wurde bereits eine Aufteilung für den lesenden Zugriff und schreibenden Zugriff berücksichtigt. Während der *Management Controller* lediglich projizierte Informationen über Datentransferobjekte bereitstellt, ist der *Command Controller* dafür zuständig Benutzereingaben, ebenfalls über

Datentransferobjekte, in entsprechende Befehle zu überführen und anschließend an den *Command Bus* weiterzuleiten.

4.4.5 Command Translation

Da das Bank Account Example Eingaben weitgehend im Frontend der Website verarbeitet, bleibt die eigentliche Aufgabe des Content Management Systems bisher unberücksichtigt. Die Verwaltung von Inhalten im TYPO3 Backend erzeugt generische Objekte, sowie zugehörige Befehle und Ereignisse. Um diese Anweisungen für die spezifische Anwendung der Kontoverwaltung zugänglich zu machen, werden generische Befehle in konkrete Befehle dieser Anwendung übersetzt.

```
1. $tcaAccountTable = TcaCommandManager::provide()
2.     ->for(static::TCA_TABLE_NAME_ACCOUNT)
3.     ->setDeniedPerDefault(true)
4.     ->setMapping([
5.         'closed' => 'closed',
6.         'balance' => 'balance',
7.         'iban' => 'iban',
8.         'account_holder' => 'accountHolder',
9.     ]);
10.$tcaAccountTable->onCreate()
11.    ->setAllowed(true)
12.    ->setFactoryName(AccountTcaCommandFactory::class)
13.    ->setProperties([
14.        'iban' => true,
15.        'account_holder' => true,
16.    ])
17.    ->forRelation('transactions')
18.        ->setAttachAllowed(true);
```

Programm 21: Transformation von generischen zu spezifischen Befehlen (eigene Darstellung)

Das vorangehende Beispiel konfiguriert die Transformation für das Erzeugen eines Datensatzes der Konto-Tabelle im Backend. Neben den Abbildungsvorschriften einzelner Eigenschaften wird ferner festgelegt, dass nur neue Transaktionen als Relation zulässig sind und, dass bei neuen Bankkonten lediglich IBAN und Kontoinhaber eingegeben werden können. Aus dieser Definition werden einerseits die Eingabefelder der Formularmasken des Backends erzeugt und andererseits generische Befehle in spezifischere Kommandos der Kontoverwaltung transformiert.

Durch die transformierten Befehle kommen auch die Anwendungsregeln der Anwendung zum Einsatz – universell für den Frontend- und Backend-Bereich. Resultierende Ereignisse werden wie erwartet persistiert und projiziert. So schließt sich der Kreis von generischer Verarbeitung und domänenspezifischer Implementierung.

4.5 Zusammenfassung

Die in diesem Kapitel durchgeführte Implementierung interpretiert die Entwurfsmuster CQRS und Event Sourcing für den Einsatz innerhalb von TYPO3. Unter anderem werden dadurch die theoretischen Ausführungen konkretisiert und verständlicher dargestellt. Durch die Bereitstellung dieser Technologien im Rahmen von TYPO3 als Open Source Software Projekt, wird es Extension-Entwicklern ermöglicht, individuell weitere Anpassungen und Verfeinerungen auf dieser Basis vorzunehmen zu können. Die Ausarbeitung wird dabei jedoch noch immer als Prototyp und Vorschlag verstanden und nicht als die endgültige und einzig gangbare Umsetzung für TYPO3. Die Implementierung eines allgemeinen Event Stores, sowie Schnittstellenbeschreibungen für Ereignisse, Befehle, Repositories und Projektionen ermöglichen weiteren Anwendungen, eigene und spezifische Erweiterungen auf dieser Basis anzufertigen. Zusätzlich liefert das Bank Account Example eine Konkretisierung für einen speziellen Anwendungsbereich im Frontend.

Im Rahmen von generischen Ereignissen ist es auch denkbar, dass daraus beispielsweise Integrationsmodule für die Technologien Apache Solr oder Elasticsearch entstehen, welche mittels Ereignisprojektionen den Suchindex befüllen.

Durch die Verlagerung des Fokus von Datensatzzuständen zu einzelnen chronologischen Ereignissen, welche dauerhaft in einem Event Store gespeichert werden, ist es nun nur noch wichtig, dass die Informationen der Events korrekt und vollständig sind. Fehler in der Projektionsschicht, die letztendlich für die Frontend-Ausgabe verantwortlich ist, lassen sich auch nachträglich korrigieren – die Zustände der Datensätze lassen sich dann jeweils erneut aus den aufgezeichneten Ereignissen generieren.

5 Ausblick

5.1 Event Sourcing in TYPO3

Für die komplette Einführung des Event Sourcing Paradigmas innerhalb von TYPO3 müssen zunächst noch weitere technologische Voraussetzungen erfüllt werden. Während der Implementierungsphase dieser Arbeit lag der Fokus nur auf einigen der wichtigsten Prozesse – weitere Aktionen, wie beispielsweise das Verschieben von Entitäten auf eine andere Seite wurden dabei noch nicht betrachtet.

Jedoch ist neben den Anforderungen an die Implementierung, die Akzeptanz dieser Technologie wesentlich wichtiger. Die Einführung von Extbase im Jahr 2009 hat gezeigt, dass die Ausarbeitung von Beispielen und Schulungsmaterialien dabei essenziell wichtig sind, um diese Technologie von der breiten Masse verwendet zu sehen. Erst einige Jahre später gab es einen ausreichenden Verbreitungsgrad bei Internetauftritten, woraus neue konkreteren Systemanforderungen abgeleitet werden konnten. Während einer noch nicht näher einschränkbaren Übergangszeit ist es deshalb notwendig, dass TYPO3 weiterhin wie gewohnt verwendet werden kann, jedoch bereits den Einsatz von Event Sourcing durch frühzeitige Anwender erlaubt und unterstützt.

5.2 Voraussetzungen

5.2.1 Performance-Steigerung

Da der Event Store in den Mittelpunkt der Datenspeicherung rückt, ergibt diese Tatsache neue Anforderungen hinsichtlich der Stabilität und vor allem auch Geschwindigkeit Unabhängig von den aktuell verfügbaren Speicherarten mittels MySQL oder GetEventStore wird eine sehr große Anzahl an Ereignissen aufgezeichnet werden, welche ebenfalls effizient und performant in Objekte des Domain Models überführt werden muss.

Es existieren Überlegungen, Events beispielsweise in Tausenderschritten oder auch größeren Mengen zu Schattenkopien (*Snapshots*) zusammenzufassen. Dadurch könnte die Anzahl der tatsächlich zu verarbeitenden Ereignisse stark reduziert werden – mit

einer positiven Auswirkung auf die gesamte Performance der Anwendung. Snapshots beinhalten den Zustand eines Aggregates bis zu einer bestimmten Version des Event Stores, alle nach der Anfertigung des Snapshots gespeicherten Ereignisse müssen auf diesen Zwischenzustand angewendet werden (Young, 2016, S. 36ff.).

Da diese Maßnahme im Resultat mit einem Cache zu vergleichen ist, existieren auch keine weiteren Anforderungen für die Speicherung der Zwischenstände – dies kann entweder als separater Stream im Event Store angelegt werden, oder auch über eine separate Projektion erfolgen.

5.2.2 Ereignisse für Binärdaten

Die Konzepte dieser Arbeit beziehen sich hauptsächlich auf Informationen, die einem gewissen Schema unterliegen – auch wenn sich dieses dynamisch ändern kann. Die Verwaltung von Änderungen auf Binärdaten wurde jedoch nicht näher betrachtet. Die Komponenten des File Abstraction Layers wäre zwar in der Lage, Änderungen der Metadaten einer Datei zu erfassen, jedoch bleiben die tatsächlichen Inhalte davon unberührt. Die Speicherung von großen Binärdaten innerhalb des Event Stores soll jedoch ebenfalls vermieden werden.

Die Versionskontrolle von größeren Dateien wird bereits mit Git Large File System (*LFS*) behandelt. Anstatt bei einer neuen Version die Binärdaten innerhalb eines Git Repositories abzulegen, werden definierte Dateitypen separat gespeichert und über deren kryptographischen Hashwert referenziert (Git LFS, 2015). Somit können Informationen eines Schemas und Dateien unabhängig voneinander gespeichert, aber dennoch im Sinne einer Version kombiniert werden.

Für TYPO3 kann dieses Verfahren eingesetzt werden, um hochgeladene Dateien in einem Verzeichnis außerhalb des Wurzelverzeichnisses des Web-Servers abzulegen. Ein neu zu schaffender Mechanismus des File Abstraction Layers verweist dann auf die jeweils aktuelle Version und ist in der Lage, diese Datei in ein öffentlich zugängliches Verzeichnis zu überführen.

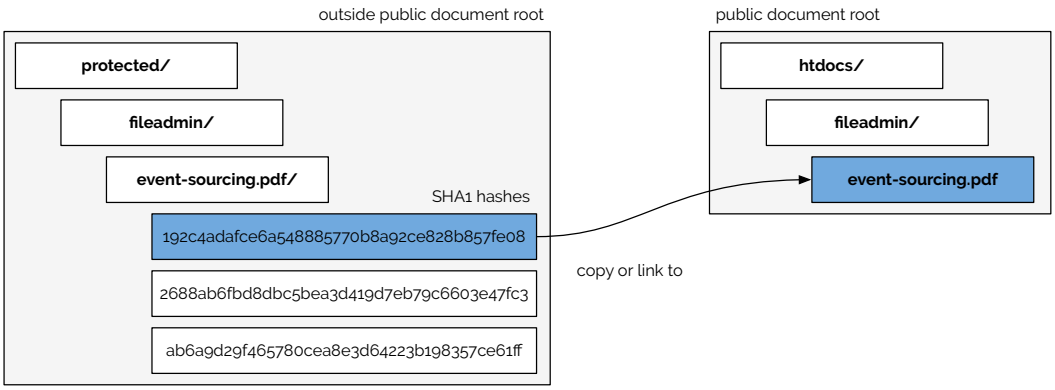

Abbildung 38: Projektion einer Dateiversion in einen konkreten Dateizustand (eigene Darstellung)

5.3 Unabhängige Projekte

Die folgenden Voraussetzungen müssen von TYPO3 erfüllt sein, bevor Event Sourcing ebenfalls in den Kernbestand aufgenommen werden kann. Bis zu einem gewissen Grad wurden prototypische Implementierungen bereits im Rahmen dieser Arbeit angefertigt. Hier gilt es, die Komponenten noch weiter generisch auszubauen. Unabhängig von Event Sourcing Methoden, stellen die meisten Aspekte unabhängige Projekte dar, die auch eigenständig zu einer Verbesserung der Abläufe in TYPO3 beitragen würden.

5.3.1 Vereinheitlichung

Innerhalb von TYPO3 werden Übersetzungen in der gleichen Datenbanktabelle abgelegt und durch Überlagerung für die Ausgabe transformiert. Die Übersetzung von Seiten stellt jedoch die einzige Ausnahme dar, bei der Übersetzungen in der separaten Databanktabelle *pages_language_overlay* abgespeichert werden. Durch die Migration dieser Anomalie in eine gemeinsame Tabelle, kann die Komplexität durch Entfernen von konditionalen Abläufen reduziert werden.

Im Vorfeld wurde die Möglichkeit analysiert, aktuell als Entitäten vorhandene Informationen als statische Konfiguration extrahieren zu können (siehe Abschnitt 3.5). Durch die Auflösung des referentiellen Charakters von Sprachen, Domains, Workspaces und Dateispeichern kann die gesamte Anwendung insgesamt vereinfacht, sowie spezielle Ausnahmeregelungen entfernt werden.

5.3.2 Meta Model Service

Der Zugriff auf das Table Configuration Array erfolgt in TYPO3 jeweils immer direkt. Somit werden Kombinationsmöglichkeiten der Konfigurationsparameter in unterschiedlichen Komponenten auch unterschiedlich interpretiert. Dieser Umstand kann durch eine gemeinsame Service-Klasse gelöst werden, in welcher die Bedeutung zentral interpretiert wird. Beispielsweise würde dann nur noch dieser Service ermitteln, ob ein Datenbankfeld als Mehrfachrelation oder als einfaches Texteingabefeld verwendet werden soll.

5.3.3 Form Engine Rules

Während der Kombination des Bank Account Examples als prototypische Extbase-Implementierung und der Verarbeitungsprozesse des Backends, wurde eine generische Konfiguration erschaffen um Formularfelder individuell aktivieren bzw. deaktivieren zu können (siehe Abschnitt 4.4.3). Damit ist es möglich, Berechtigungen auf der Ebene eines Eingabefeldes oder einer Relation zu definieren – die Darstellung ist bei neu anzulegenden Datensätzen unterschiedlich zur Modifikation eines bestehenden Elements. Diese Einschränkungen lassen sich ebenso auf das Erstellen, Entfernen oder Umsortieren von Referenzen einer Relation übertragen.

5.3.4 UUID Integration

Die Integration von eineindeutigen Werten stellt eine Basisanforderung für den systemübergreifenden Austausch von Informationen dar. Der vorhandene Export- und Importprozess von TYPO3 kann somit ebenfalls von der Umstellung auf UUIDs profitieren – hier wird aktuell die ganzzahlige UID erneut berechnet, falls ein bestimmter Wert beim Importvorgang bereits in Verwendung ist.

Dieser Schritt hat jedoch nicht nur zur Folge, dass alle anderen Komponenten, die innerhalb von TYPO3 mit Referenzen umgehen, entsprechend angepasst werden müssen – zusätzlich wäre der direkte Verweis auf einen Datensatz mittels TypoScript nur noch über die wesentliche längere UUID möglich.

6 Fazit

6.1 Extbase

Die Auseinandersetzung mit Prinzipien des Domain-Driven Designs hat nicht nur neue Möglichkeiten für das Verarbeiten durch Event Sourcing und CQRS aufgezeigt, auch können damit einige grundlegende Abläufe neu bewertet werden. Das innerhalb von TYPO3 vorhandene Extbase-Framework geht hinsichtlich der Datenmodelle und der Verwendung von Repositories einen sehr spezifischen Weg. Entitäten werden hier eher als Datentransferobjekte zur Abbildung der Datenbankinhalte verwendet – eine umfangreichere Verwendung zugunsten einer ausgeprägten Applikationslogik ist eher in eigenständigen Service-Objekten zu finden, die jedoch losgelöst vom eigentlich Domain Model zu betrachten sind.

Durch den geänderten Ablauf unter Einsatz des Event Sourcing Paradigmas, verlieren die aufgestellten Konventionen an Bedeutung. Die Datenspeicherung erfolgt nun abseits der existierenden Repository-Strukturen und die Validierung von Eingaben bezieht sich idealerweise nur noch auf Datentransferobjekte und nicht mehr auf echte Entitäten. Aufwendig geschaffene Mechanismen, um Abhängigkeiten erst bei Bedarf aufzulösen und zu laden (*Lazy Loading*) werden in dieser Form nicht mehr benötigt, da im Sinne von Materialized Views bereits nur tatsächlich benötigte Daten verarbeitet werden – das Lazy Loading Konzept kann somit zukünftig weitgehend vernachlässigt werden.

Es gilt jedoch im weiteren Verlauf der Event Sourcing Einführung zu betrachten, ob die vorhandene objektrelationale Abbildung damit an die neuen Bedürfnisse angepasst werden kann, alternativ auch durch ein externes Framework wie Doctrine ORM ausgetauscht wird, oder am Ende gänzlich überflüssig erscheint.

6.2 Komplexität

Das zugrundeliegende Thema dieser Arbeit führt neue Konzepte ein, die hinsichtlich ihrer Abläufe neue Komplexitäten schaffen. Vor allem die verteilte Verarbeitung von Befehlen, Ereignissen und Projektionen garantieren einen weiteren Anstieg der

Lernkurve für Entwickler. Allerdings ist die Absicht des Event Sourcing Paradigmas, generisch im Kern des Frameworks verankert zu werden, so dass im Idealfall die Notwendigkeit nicht mehr existiert, sich über diese Abläufe großartig Gedanken zu machen. Die wesentlich größere Herausforderung ist an die Akzeptanz dieser neuen Technologien gestellt, sowie die Aufgabe, bisherige Denkmuster und Erfahrungen an die neuen Begebenheiten zu adaptieren.

6.3 CRUD versus Event Sourcing

CQRS und Event Sourcing stellen im TYPO3 Umfeld neue Themenkomplexe dar, die ihren besonderen Reiz hinsichtlich innovativer, neuer Technologien verbreiten. Es sollte allerdings vermieden werden, nun jeden denkbaren Anwendungsfall für CQRS und Event Sourcing zurechtzubiegen und den Einsatz dieser Paradigmen zu erzwingen.

Anwendungen, die bisher rein auf CRUD basieren, haben natürlich noch immer eine Existenzberechtigung – falls es erforderlich und hilfreich erscheint, können diese Konstrukte jedoch mit den in dieser Arbeit dargestellten Prozessen in eine neue Software-Architektur überführt werden. Auch in diesem Zusammenhang gilt es, das richtige Werkzeug für eine konkrete Herausforderung zu finden.

6.4 Zusammenfassung

Die Aufteilung und Optimierung einer Anwendung für den ausschließlich modifizierenden oder ausschließlich abfragenden Zugriff resultiert zunächst darin, sich noch genauer mit den vorhandenen Prozessen auseinanderzusetzen. Diese Analyse hat dazu geführt, vermischte Abläufe und verteilte Abhängigkeiten gedanklich auflösen zu können und im ersten Schritt in entsprechende generische Befehle und Ereignisse zu überführen. Durch den zusätzlichen Einsatz einer generischen Projektion, welche von einem definierbaren Kontext abhängt, lassen sich die bisherigen Vermischungen noch weiter gegeneinander abtrennen. Die eingangs formulierte Forschungsfrage, ob der Einsatz von Event Sourcing Technologien dazu beitragen könne, die Komplexität der Datenspeicherung im Workspace-Kontext zu reduzieren, kann somit bejaht werden. In den nächsten Schritten sieht diese Reduzierung konkret vor, einen Großteil des

Programmcodes für die Ausnahmen im Workspace-Umfeld ohne Ersatz zu entfernen und damit Abläufe insgesamt zu vereinfachen.

Die zukünftige Verbreitung von Event Sourcing im TYPO3 Umfeld hängt jedoch auch von der Begeisterungsfähigkeit und Akzeptanz der Entwickler ab – Dokumentation, Beispiele und Vorträge bei den zahlreichen Veranstaltungen des TYPO3 Universums sind dafür jedoch eine geeignete Grundlage, um dieses Thema im Fokus zu halten und die Integration weiter voran zu treiben.

Abbildungsverzeichnis

Literaturverzeichnis

Betts, Dominic, et al. 2013. *Exploring CQRS and Event Sourcing.* Redmond : Microsoft patterns & practices, 2013. ISBN 978-1621140160.

Buenosvinos, Carlos, Soronellas, Christian und Akbary, Keyvan. 2016. *Domain-Driven Design in PHP.* Victoria : Leanpub, 2016. ISBN 978-0-9946084-1-3.

Evans, Eric. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* [E-Book] Boston, MA, US : Addison Wesley, 22. August 2003. ASIN B00794TAUG.

Event Store LLP. 2016. Documentation. *Event Store.* [Online] 27. August 2016. [Zitat vom: 6. Oktober 2016.] http://docs.geteventstore.com/.

Fowler, Martin. 2005. CommandQuerySeparation. *Martin Fowler - ThoughtWorks.* [Online] 5. Dezember 2005. [Zitat vom: 6. Oktober 2016.] http://martinfowler.com/bliki/CommandQuerySeparation.html.

—. 2005. Event Sourcing. *Martin Fowler - ThoughtWorks.* [Online] 12. Dezember 2005. [Zitat vom: 7. Oktober 2016.] http://martinfowler.com/eaaDev/EventSourcing.html.

—. 2002. *Patterns of Enterprise Application Architecture.* Boston : Addison Wesley, 2002. ISBN 978-0321127426.

Git LFS. 2015. Git LFS Specification. *Git extension for versioning large files.* [Online] 18. November 2015. [Zitat vom: 18. Oktober 2016.] https://github.com/github/git-lfs/blob/master/docs/spec.md.

Hader, Oliver. 2007. *TYPO3 - Inline Relational Record Editing.* Hof : Hochschule Hof, 2007. OPAC BV043190246.

—. 2009. TYPO3 4.3 Release Notes. *TYPO3 - The Enterprise Open Source CMS.* [Online] 30. November 2009. [Zitat vom: 5. Oktober 2016.] https://typo3.org/download/release-notes/typo3-43-release-notes/.

Hummel, Helmut. 2012. TYPO3 6.0 Release Notes. *TYPO3 - The Enterprise Open Source CMS.* [Online] 27. November 2012. [Zitat vom: 5. Oktober 2016.] https://typo3.org/download/release-notes/typo3-60-release-notes/.

Mack, Benjamin. 2015. TYPO3 CMS 7 LTS Release Notes. *TYPO3 - The Enterprise Open Source CMS.* [Online] 10. November 2015. [Zitat vom: 5. Oktober 2016.] https://typo3.org/download/release-notes/typo3-7-release-notes/.

Meyer, Bertrand. 2012. Eiffel: a language for software engineering. *LASER Summer School, ETH Zürich.* [Online] 4. September 2012. [Zitat vom: 6. Oktober 2016.] http://laser.inf.ethz.ch/2012/slides/Meyer/eiffel_laser_2012.pdf.

—. 1997. *Object-Oriented Software Construction.* 2. Auflage. Upper Saddle River : Prentice Hall, 1997. ISBN 978-0136291558.

Newman, Sam. 2015. *Building Microservices.* Sebastopol : O'Reilly and Associates, 2015. ISBN 978-1491950357.

Nijhof, Mark. 2013. *CQRS, The example.* Victoria : Leanpub, 2013. ISBN 978-1484102879.

Rau, Jochen und Kurfürst, Sebastian. 2010. *Zukunftssichere TYPO3-Extensions mit Extbase und Fluid.* 1. Auflage. Köln : O'Reilly Verlag, 2010. ISBN 978-3-89721-965-6.

Schneede, Frank. 2011. Einführung in Materialized Views. *Oracle DBA Community.* [Online] August 2011. [Zitat vom: 6. Oktober 2016.] http://www.oracle.com/webfolder/technetwork/de/community/dbadmin/tipps/mav_introduction/index.html.

Verraes, Mathias. 2013. Facilitating Event Storming. *Mathias Verraes.* [Online] 28. August 2013. [Zitat vom: 5. Oktober 2016.] http://verraes.net/2013/08/facilitating-event-storming/.

—. 2014. Practical Event Sourcing. *Mathias Verraes.* [Online] 18. September 2014. [Zitat vom: 6. Oktober 2016.] http://verraes.net/2014/03/practical-event-sourcing/.

Vogels, Werner. 2008. Eventually Consistent - Revisited. *All Things Distributed.* [Online] 22. Dezember 2008. [Zitat vom: 7. Oktober 2016.] http://www.allthingsdistributed.com/2008/12/eventually_consistent.html.

Young, Greg. 2016. CQRS Documents by Greg Young. *CQRS - Did you mean CARS?* [Online] 6. Novermber 2016. [Zitat vom: 13. Oktober 2016.] https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.

Anhang

Eigenständige Implementierungen

- Data Handling: beinhaltet die Basiskonzepte zur Einführung von Event Sourcing im TYPO3 Kern, https://github.com/TYPO3Incubator/data_handling
- Bank Account Example: Beispiel einer abgetrennten MVC Applikation auf der Basis von Extbase, https://github.com/TYPO3Incubator/bank_account_example
- Event Sourcing Bundle: stellt ein Installationspaket auf der Basis von Composer bereit, https://github.com/TYPO3Incubator/eventsourcing-bundle

Eingereichte Open Source Erweiterungen

TYPO3 CMS

- #77384: [TASK] Add functional tests for versioned MM references, https://review.typo3.org/#/c/49349/
- #77374: [BUGFIX] Opposite MM relation between both new entities not created, https://review.typo3.org/#/c/49337/
- #77352: [FOLLOWUP][TASK] Doctrine: Migrate queries in Extbase Typo3DbBackend, https://review.typo3.org/#/c/49355/
- #77507: [TASK] Doctrine: Migrate functional test cases in ext:core, https://review.typo3.org/#/c/49490/
- #78022: [TASK] Streamline DBAL connection invocation in RelationHandler, https://review.typo3.org/#/c/49989/
- #78137: [TASK] Streamline ConnectionPool invocation in Testbase, https://review.typo3.org/#/c/50085/
- #78045: [TASK] Implement DBAL inSet() for SQLite, https://review.typo3.org/#/c/50006/
- #78046: [TASK] Extract per-connection concerns from SchemaMigrator, https://review.typo3.org/#/c/50007/

- #78301: [BUGFIX] Use context specific table in PageRepository, https://review.typo3.org/#/c/50231/
- #78129: [BUGFIX] Statement::rowCount not reliable for SELECT queries, https://review.typo3.org/#/c/50083/
- #77375: [BUGFIX] MM references are not transformed to versioned entities, https://review.typo3.org/#/c/49338/

PHP Client for Event Store HTTP API

- #41: [BUGFIX] Invalid event reconstitution at EventStore::readEventBatch, https://github.com/FriendsOfOuro/geteventstore-php-core/pull/41
- #42: [TASK] Provide eventId in reconsituted feed event, https://github.com/FriendsOfOuro/geteventstore-php-core/pull/42
- #43: [TASK] Retrieve created version after writing to stream, https://github.com/FriendsOfOuro/geteventstore-php-core/pull/43

Bootstrap Package

- #384: [BUGFIX] Avoid specific DBMS literals in database queries, https://github.com/benjaminkott/bootstrap_package/pull/384

Inhalt des beigefügten Datenträgers

Dieser Arbeit liegt eine DVD bei, welche aus den nachfolgenden Daten besteht. Zur Ausführung der Applikation wird ein Apache Web-Server, der PHP Interpreter in der Version 7, sowie eine MySQL-fähige Datenbank benötigt.

- *application/www/* - Inhalte für das Wurzelverzeichnis des Webservers
- *application/database.sql* – Inhalte für die MySQL Datenbank *t3_es_local*
- *README.md* – Hinweise zur Installation & Passwortinformationen
- *TYPO3 – Datenmodifikation und Event Sourcing.pdf* – dieses Dokument