Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Refaktorisierung eines Produktes durch Anwendung von Domain-Driven Design

Tom Lausmann
Matrikelnummer: 4872596
t.lausmann@fu-berlin.de

Betreuer:	Prof. Dr. Lutz Prechelt
Eingereicht bei:	Prof. Dr. Lutz Prechelt
Zweitgutachter:	Prof. Dr. Jörn Eichler

Berlin, 7. Februar 2019

Zusammenfassung

BETA ist ein Produkt der GAMMA GmbH. Es dient zur schnellen Digitalisierung von Dokumenten durch den Einsatz von Künstlicher Intelligenz. BETA ist schnell aus seiner experimentellen Phase herausgewachsen und mit jedem neuen Kunden steigt die Komplexität des Produkts. Um dieser Komplexität eine Struktur zu verleihen, wird BETA mit der Hilfe von Domain-Driven Design refaktorisiert.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

7. Februar 2019

Tom Lausmann

Inhaltsverzeichnis

1 Einleitung

1.1 Domain-Driven Design

Domain-Driven Design ist eine Herangehensweise für die Modellierung von komplexer Software. Eric Evans hat diesen Begriff in seinem 2003 veröffentlichten Buch *Domain-Driven Design* geprägt. Die Techniken und Praktiken, die Domain-Driven Design ausmachen, werden hauptsächlich von der Fachdomäne beeinflusst, dessen Prozess abgebildet werden soll.

Das Ziel von Domain-Driven Design ist das finden einer sogenannten ubiquitären Sprache und das definieren von Bounded Contexts. Die *ubiquitäre Sprache* ist die allgegenwärtige Sprache, die von allen Beteiligten benutzt werden soll. Es soll vermieden werden, dass bestimmte Parteien Begriffe anders interpretieren als andere oder gar verschiedene Begriffe verwenden. Das heißt auch, dass sich die ubiquitäre Sprache im Quell-Code wiederfinden muss, da dieser die Fachlichkeit abbildet. Eine ubiquitäre Sprache wird innerhalb eines *Bounded Context* benutzt. Ein Bounded Context steckt eine fachliche Domäne ab. Innerhalb der Gesamt-Domäne kann es mehrere Teil-Domänen geben, die jeweils einen Bounded Context abbilden können. Ein beliebtes Beispiel ist hier ein Online-Shop, dessen Haupt-Aufgabe der Verkauf von Produkten ist. Hier könnte man zum Beispiel die Kontexte Verkauf, Bezahlung und Versand abbilden.

Nachdem die Bounded Contexts abgesteckt worden sind wird noch ermittelt, wie diese miteinander verbunden werden sollen, um den Gesamt-Prozess abzubilden. Dazu gibt es verschiedenste Möglichkeiten. Nur um ein Paar zu nennen, gibt es zum Beispiel den *Shared Kernel*, bei dem sich zwei Bounded Contexts einen Teil der Software-Komponenten Teilen. Dies erfordert ein enges Zusammenspiel zwischen den beiden beteiligten Teams, da Änderungen an den geteilten Komponenten beide Kontexte betreffen. Ein anderer Ansatz ist die Anwendung sogenannter *Domain-Events*. Dabei veröffentlicht ein Bounded Context über ein Messaging-System Ereignisse, die von anderen Bounded Contexts konsumiert werden können. Diese Methode erzeugt bei weitem die wenigste Kopplung zwischen den Bounded Contexts.

Domain-Driven Design gibt auch hier Leitlinien vor, wie diese Modellierung der Bounded Contexts aussehen kann und fasst diese Praktiken unter dem Begriff *taktisches Design* zusammen. Beispiele für diese Konzepte sind Aggregate, Entitäten, Value Objekte und Repositories. Diese seien hier nur erwähnt, da sie später näher erläutert werden. [4]

1.2 BETA

BETA ist ein Produkt der Firma GAMMA GmbH. Die Firma hat ihren Sitz in Berlin-Friedrichshain und beschäftigt sich hauptsächlich mit Lösungen für die Baufinanzierungs-Branche. Das älteste Produkt ist die DELTA, welches ein Produkt für Banken ist, um den Prozess der Kreditvergabe bei der Baufinanzierung abzubilden. Unter anderem hat dieses Produkt einen Bereich für das Management von Dokumenten, die Digitale Akte. Hier werden Dokumente gesammelt, die für die Kreditentscheidung relevant sind. Im Jahr 2015 kam die Überlegung auf, dass es sehr angenehm wäre, wenn sich

die geladenen Dokumente automatisch in die jeweiligen Bereiche [1] der Digitalen Akte einordnen würden. Daraus hat sich die Idee für BETA entwickelt.

BETA ist ein Werkzeug für die Digitalisierung und Verarbeitung von Dokumenten. Die an BETA übergebenen Dokumente werden in ihre einzelnen Seiten aufgeteilt. Die Seiten werden dann in Teildokumente gepackt und einem Dokumenten-Typ zugeordnet. Anschließend können die Teildokumente in einer Weboberfläche namens *DocumentMaster* korrigiert und bearbeitet werden. Zum Schluss erstellt BETA aus den Teildokumenten neue Dokumente, die dann an das jeweilige Zielsystem übertragen werden.

Um alle gewünschten Dateiformate verarbeiten zu können, stützt sich BETA auf verschiedene Dritt-Anbieter Werkzeuge, um die Dokumente auf ein für die Weiterverarbeitung geeignetes Bild-Format zu konvertieren.

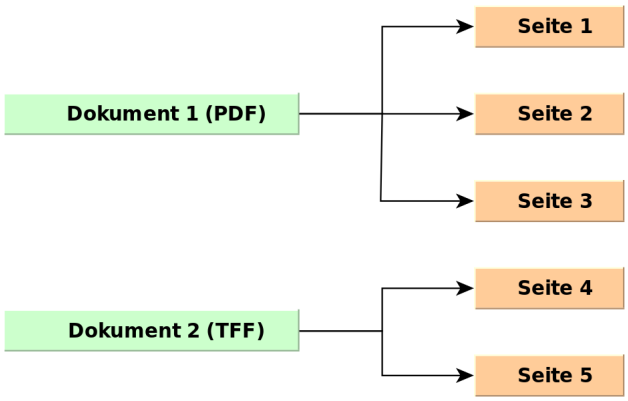

Abbildung 1: Extraktion der Seiten aus den übergebenen Dokumenten.

Anschließend wird der Text der Seiten durch ein OCR-Programm [2] ausgelesen. Die dadurch gewonnen Informationen dienen dann als Grundlage für die Segmentierung und Klassifizierung. Eine Ausnahme bei dem OCR-Vorgang bilden Dokumente, die bereits Text-Informationen enthalten, da diese stattdessen verwendet werden.

Bei der Segmentierung und Klassifizierung werden Text und Bild-Informationen genutzt, um zu bestimmen, welche Seiten zusammengehören und um was das für Teildokumente es sich handelt. Dabei werden verschiedene Modelle der Künstlichen Intelligenz verwendet. Die Ergebnisse der Analysen werden anschließend gespeichert.

Mit dem *DocumentMaster* können die Teildokumente nun manipuliert werden. Das Verschieben von Seiten in andere oder neue Teildokumente, ist mit diesem Werkzeug möglich. Zusätzlich können für einzelne Seiten Notizen angelegt und Markierung auf die Seiten gemalt werden.

Ist der Nutzer mit der Bearbeitung fertig, werden aus den Teildokumenten PDF-Dateien generiert. Die PDF-Dateien werden dann in einem bestimmten Format, das den Vorstellungen des Kunden entspricht, an das Zielsystem übertragen.

[1] Diese Bereiche kann man sich wie Schubladen vorstellen, die eine Beschriftung haben und bestimmte Dokumente enthalten sollen.

[2] OCR = Optical Character Rekognition. Die Extraktion von Textinformationen aus einem Bild

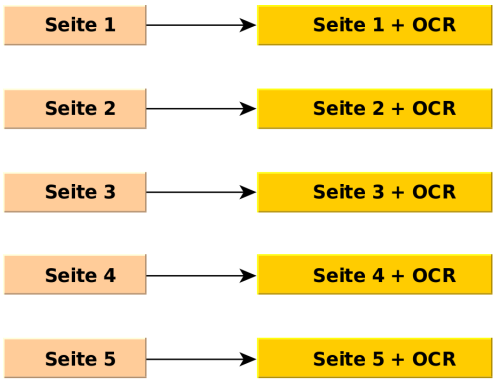

Abbildung 2: Anwendung von OCR auf die Seiten.

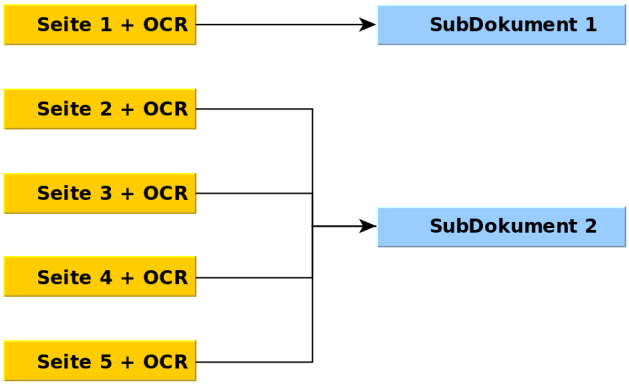

Abbildung 3: Verteilung der Seiten in Teildokumente.

Im Falle der DELTA, hat BETA noch eine weitere Aufgabe. Die Digitale Akte der DELTA hat sogenannte Spezifizierte Unterlagenanforderungen (SU). Diese sind Dokumente, die für die Kreditentscheidung Notwendig sind. Ein Beispiel für so eine SU ist *Personalausweis von Max Mustermann in Kopie*. BETA sucht nun die relevanten Teil-Dokumente, die die Dokumentenklasse *Personalausweis* besitzen und den Text *Max Mustermann* enthalten und ordnet diese der SU zu. So wird mit allen SUs der Digitalen Akte der DELTA verfahren. Die Zuordnung der Teildokumente zu den SUs kann ebenfalls im *DocumentMaster* bearbeitet werden. Am Ende erzeugt BETA für die SUs die Ergebnis-Dokumente und schickt sie an DELTA.

Ein neuer Kunde (Hier ALPHA genannt) möchte stattdessen einen anderen Prozess abbilden. Hier werden jeden Tag Dokumente durch einen Scanner in deren System geladen. Diese Dokumentenstapel sollen nach dem Scan automatisch zu BETA geschickt und analysiert werden. Die fertigen Analyse-Vorgänge sammeln sich in einer Web-Oberfläche, von der aus man den *DocumentMaster* aufrufen kann. Sind die Dokumente dort abgearbeitet, wird aus den Teildokumenten ein Archiv gebaut und zurück zum System von ALPHA geschickt.

BETA wird nun seit etwa drei Jahren entwickelt und ist seit etwa zwei Jahren im produktiven Einsatz. Es wird seither von drei bis vier Werksstudenten entwickelt.

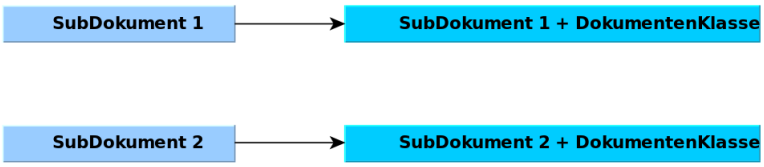

Abbildung 4: Den Teildokumenten wird eine Dokumentenklasse zugeordnet.

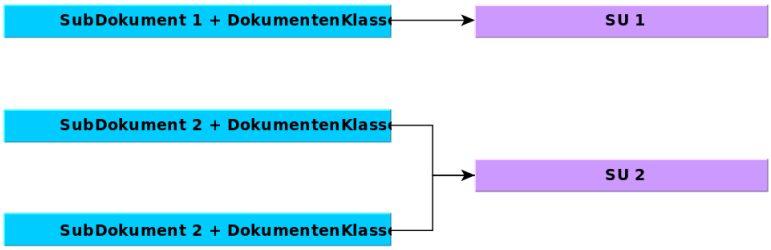

Abbildung 5: Zuordnung der Teildokumente zu den SUs.

BETA wird als HTTP-REST-Service angeboten und baut auf *Spring Boot* auf. Spring Boot ist ein Java-Framework, für die Erstellung von Microservices. Verwendete Technologien hier sind hier *Spring MVC* (Model-View-Controller Framework für die Abbildung von REST-Endpunkten), *Spring Data JPA* [3] und *Spring Security*[4]. Als OCR-Programm wird *Tesseract* von Google verwendet und für die Realisierung verschiedener Mustererkennungs-Probleme *Tensorflow* [5], welches ebenfalls von Google kommt. Außerdem werden diverse Bibliotheken und Tools von Drittanbietern verwendet, zum Beispiel für die Konvertierung von verschiedenen Dokumenten zu Bildern und die Erstellung von PDF-Dateien. Die meisten dieser Werkzeuge sind Kommandozeilenprogramme und werden von der Spring Boot Applikation über die Process-API von Java angesprochen. Das Projekt wird von einem *Jenking* Build-Server gebaut und in ein Docker-Image gepackt, welches dann auf den entsprechenden Servern geladen wird. Eom Docker-Images kann man sich wie eine Virtuelle Maschine vorstellen, die auf dem Zielsystem gestartet wird und alle Abhängigkeiten besitzt, die für die Ausführung des Programmes nötig sind. Das Projekt umfasst etwa fünfzehn-tausend Codezeilen und wird in einem Git-Repository verwaltet. Entwickelt wird nach Scrum in einem drei Wochen Sprint-Zyklus.

1.3 Motivation

Der Code von BETA ist mit der Zeit sehr komplex geworden. Am Anfang wurde sehr viel experimentiert und diese experimentellen Features dann als Basis für die weitere Entwicklungen genutzt. Der einzige Kunde war damals *DELTA* und das für eine relativ lange Zeit. Auch wenn versucht wurde, die fachlichen Aspekte von *DELTA* aus BETA rauszuhalten, haben sich immer wieder Kleinigkeiten eingeschlichen. Spätes-

[3] Persistenzframework für die Anbindung verschiedenster Datenbanken
[4] Framework, um die REST-Enpunkte mit einem Sicherheitsmechanismus zu versehen
[5] Ein Python-Framework für die Erstellung und Verwendung von KI-Modellen

tens bei der Implementierung der Anforderungen des neuen Kunden ALPHA ist uns das negativ aufgefallen. Es wurde versucht, ein eigenständiges Modul für den neuen Kunden zu entwickelt. Leider sind doch gewisse Sachen in das ursprüngliche BETA-Modell gerutscht, weil das zu diesem Zeitpunkt die technisch einfachste Lösung war. So haben sich die Fachlichkeiten mehrerer Kunden nach und nach miteinander vermischt und es sind unhandliche Konstrukte entstanden.

1.4 Ziel

Mit dieser Arbeit möchte ich die eben genannte Situation auflösen. Die vermischten Fachlichkeiten sollen auseinandergezogen und eine einheitliche Codestruktur geschaffen werden. Die Arbeit ist also die Dokumentation und Validierung der Refaktorisierungen von BETA. Dazu will ich die Konzepte von Domain-Driven Design benutzen. Ziel ist es, eine Modul-Struktur zu finden, die von außen schon erkennen lässt, wie BETA aufgebaut ist. Die Module selber sollen einheitlich strukturiert sein, sodass Software-Komponenten leichter zu finden und zu ändern sind.

Dazu erkläre ich bestimmte Konzepte von Domain-Driven Design genau und wende sie auf BETA an, sodass der Code die Eigenschaften des Konzeptes erfüllt. Dazu werde ich entsprechende Codebeispiele liefern. Zum Schluss bewerte ich den neuen Code auf seine Handhabbarkeit und überprüfe, ob sich die Komplexität verbessert, verschlechtert oder nur in einer anderen Form vorliegt.

Ich orientiere mich bei der Umsetzung hauptsächlich am Buch *Implementing Domain-Driven Design* von *Vaughn Vernon*, welches auf das Buch *Domain-driven design: tackling complexity in the heart of software* von *Eric Evans* aus dem Jahr 2003 aufbaut.

2 Die Refaktorisierung

2.1 Umstrukturierung der Module Teil 1

Eines meiner Ziele ist es, die gesamte Modulstruktur von BETA intuitiver zu gestalten. Dazu möchte ich mich an den Bounded Contexts von Domain-Driven Design orientieren. In den einzelnen Modulen soll sich die ubiquitäre Sprache der einzelnen Kunden von BETA wiederfinden.

Bounded Contexts sind die Teilsysteme eines Produkts. Sie können aus verschiedenen Gründen entstehen. Zwei Teilsysteme bedienen meist verschiedene Nutzergruppen, mit verschiedenen Aufgaben. Die Teilsysteme bilden ein Teil des Gesamt-Modells ab. Dieses Teil-Modell ist in sich abgeschlossen und mit einem anderen Teilsystem über bestimmte Mechanismen gekoppelt (Context-Mapping). Sie können von verschiedenen Teams entwickelt werden.

Context-Maps beschreiben die Beziehung zwischen den Bounded-Contexts. Diese können in verschiedenen Arten und Weisen dargestellt und implementiert werden.
Beispiel: Shared Kernel: Wenn zwei Bounded-Contexts einen Teil ihres Modells gleich Abbilden, kann es nützlich sein, wenn beide auf den gleichen Code zugreifen können, der diese Modellüberschneidung betrifft. Das heißt aber, dass die Teams, in Bezug auf den geteilten Code, voneinander abhängig sind. Deswegen sollte dieser Teil möglichst klein gehalten werden.

Die Ubiquitäre Sprache ist die Sprache, mit der das Model eines Bounded Context abgebildet ist. Sie sollte von allen Personen in allen Bereichen der Entwicklung benutzt werden. Also im Programm-Code, in Diagrammen, in E-Mails, ... und vor allem bei der verbalen Kommunikation. Synonyme, Überabstraktionen oder Übersetzungen sollten, wenn möglichen, vermieden werden, weil diese zu alternativen Modellen führen könne. Die ubiquitäre Sprache kann sich über Bounded-Contexts hinweg unterscheiden.[5]

In BETA würde ich jetzt drei Bounded-Contexts definieren. Einmal hätten wir den *Kern-Kontext*, der sich mit der grundlegenden Aufgabe von BETA beschäftigt: dem Segmentieren und Klassifizieren von Dokumenten. Das ist für alle Nutzer von BETA gleicht. Dazu kommen dann die Abbildungen der Prozesse der Kunden, die BETA benutzen. Das wären die der *DELTA* und von *ALPHA*. Der Grund ist, dass BETA für jeden Kunden zugeschnitten werden soll. Deshalb halte ich eine Einheitliche Schnittstelle für nicht sinnvoll. Dies entspricht dem *Customer/Supplier* Beziehungs-Typ, bei dem der Supplier die Schnittstellen-Wünsche des Customers abbildet. Da in den Kunden-Kontexten die Sprache der Kunden zur Implementierung verwendet werden kann, macht es das leichter, die gestellten Anforderungen zu verstehen und abzubilden. Neue Kunden würden ihren eigenen Bounded Context bekommen.

Die Beziehung zwischen Kunden-Kontext und Kern-Kontext würde dann dem *Partnership*-Prinzip entsprechen. Das heißt, die Kunden-Kontexte wären direkt vom Kern-Kontext abhängig. Dieser Beziehungstyp kann natürlich sehr riskant sein, weil nur eins der beiden Teilsysteme kaputt gehen darf, um das Gesamtsystem unbrauchbar zu machen. Da BETA aber von einem Team entwickelt wird, rückt dieser Nachteil eher in den Hintergrund. [5]

In BETA wird *Maven* eingesetzt, was ein Bau- und Abhängigkeiten Verwaltungs-

werkzeug für Java-Projekte ist. Es unterstützt die Verwaltung verschiedener ineinander Verschachtelter Module, aus denen einzelne Programm-Artefakte erzeugt werden können. Jeder genannte Bounded Context soll einem Maven-Modul entsprechen. Je nach Notwendigkeit können innerhalb dieser Module weitere Untermodule angelegt werden.

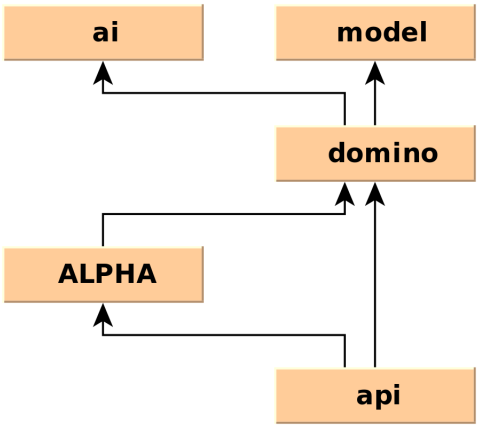

Abbildung 6: Alte Modulstruktur. Die Namen der Module sind unklug gewählt. Die KI-Algorithmen von BETA haben ihren eigenen Platz

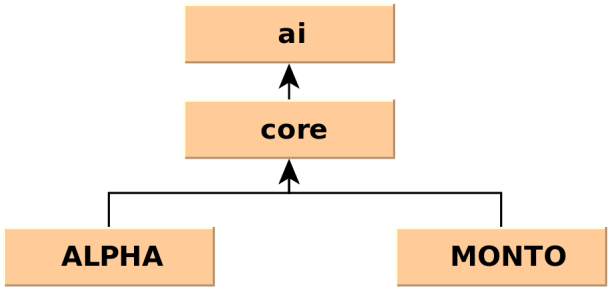

Abbildung 7: Neue (Ziel-) Modulstruktur. Die Module haben die Namen ihrer Kontexte.

Um dorthin zu gelangen müssen die entsprechenden Module angelegt, und die relevanten Code-Teile übertragen werden. Dies ist jedoch noch nicht ohne weiteres möglich. Das Modul *model* der aktuellen Modell-Struktur beinhaltet das komplette Modell aller Kunden. Fast die gesamte Logik von BETA ist im Modul *domino* enthalten. Damit ist sowol die Kern-Funktionalität gemeint, als auch Teile von der DELTA und von ALPHA. Die REST-Schnittstellen sind zwischen den Modulen *api* und *APLPHA* verteilt. Dass die Domäne der DELTA doch so viel Einfluss auf die Kern-Logik von BETA haben würde, war uns am Anfang nicht bewusst.

Die gewünschte Modul-Struktur lässt sich also so noch nicht realisieren. Es sind Refaktorisierungen innerhalb der aktuellen Module notwendig. Diese werden in den nächsten Abschnitten besprochen. Zum Schluss werde ich noch einmal auf dieses

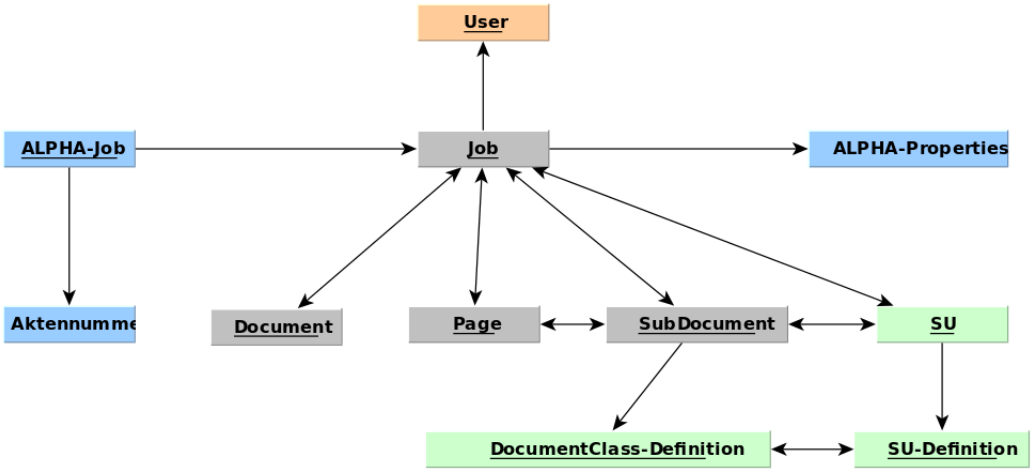

Abbildung 8: Entitäten vor der Refaktorisierung. Pfeile zeigen Abhängigkeiten, Unterstreichungen das Vorhandensein von DAO-Services für die Entität

Thema zurückkommen, wenn die Kopplung der relevanten Bestandteile aufgelöst worden ist.

2.2 Entfernen zyklischer Referenzen aus dem Model

Für das erste habe ich mir vorgenommen, zyklische Referenzen im schon vorhandenen Modell zu entfernen. Das heißt, dass ein Eltern-Objekt seine Kind-Objekte kennen soll, aber nicht anders herum. In BETA ist dies jedoch der Fall. Ein Beispiel dafür ist die Beziehung zwischen Subdokument und Seite. Ein Subdokument hat eine Liste von Seiten, aus die es besteht. Die Seiten wissen zusätzlich auch in welchen Subdokumenten sie vorhanden sind. Gerade im Hinblick auf Aggregate ist eine solche Beziehung zwischen Entitäten nicht gewollt.

Entitäten sind Objekte, die das Domänen-Modell reflektieren. Sie besitzen eine Identität und haben einen bestimmten Lebenszyklus, der zum Beispiel durch seinen Zustand modelliert werden kann. Sie sind in der Regel veränderlich. Ihr Zustand kann sich also im Laufe des Lebenszyklus ändern. Metaphorisch kann man sich Entitäten als die Tabellen in einer Relationalen Datenbank vorstellen. Sie können aber nicht nur einfache Datenhaltungs-Objekte sein, sondern bilden auch den von ihnen repräsentierte Fachlogik ab.

Value Objekte haben Gegensatz zu Entitäten keine Identität und keinen Lebenszyklus. Sie repräsentieren Werte und Eigenschaften der Entitäten. Man kann sie sich also, um die Metapher von eben fortzuführen, als Spalten der Tabellen vorstellen.

Aggregate sind Cluster von Entitäten und Value Objekten. Ein Aggregat besitzt eine Entität, die als Wurzel-Element dient und den gleichen Namen, wie das Aggregat besitzt. Sie wird **Aggregat-Wurzel** genannt. Der Zugriff auf die Entitäten eines Aggregats soll nur über ihre Aggregat-Wurzel geschehen. Die Aggregat-Wurzel ist dafür zuständig, den Zustand des Aggregats vor und nach einer Transaktion[6] konsis-

[6]Eine Operation auf ein Aggregat

tent zu halten. Dafür können Eigenschaften und Invarianten definiert werden, die von der Aggregat-Wurzel überwacht werden. Externe Objekte dürfen nur Referenzen zur Aggregat-Wurzel benutzen. Falls es doch einmal nötig sein sollte, interne Entitäten eines Aggregats zu verwenden, dann sollte dies nur für eine Operation geschehen.

Repositories dienen als Ablageort für Aggregate. Sie sind als Interface definiert und sollen dem Benutzer ein In-Memory Speicher simulieren. Sie haben die Eigenschaft, dass ein Aggregat, dass von einem Repository aufgenommen wird, im gleichen Zustand wieder geladen werden kann. [4]

Spring-Data-JPA bietet eine sehr komfortable Mechanik, um solche Repositories zu definieren.

```
public interface JobRepository extends JpaRepository<Job,
    Integer> {
  Job findByName(String name);
}
```

Lässt man ein Interface von *JpaRepository* erben, erhält man für die Angegebene Entität nicht einen Grundsatz von Standard-Lade und Speicher-Funktionen, sondern kann mit einer bestimmten Syntax auch eigene Methoden deklarieren. Je nachdem, was für eine Datenbank an die Applikation angeschlossen ist, stellt Spring-Data-JPA eine entsprechende Implementierung für das Repository Interface bereit. In BETA wird momentan eine MySQL Datenbank verwendet. Die Entitäten müssen deswegen zusätzlich mit Annotation versehen werden, die den Aufbau der korrespondierenden Datenbanktabellen beschreiben. [1]

In BETA ist die zentrale Entität der *Job*. Ein Job bildet den Analyseprozess von Dokumenten ab, die in ihre Teildokumente konvertiert werden. Der Job verwaltet die Dokumente, SUs, Teildokumente und deren Seiten. Die Kind-Entitäten besitzen momentan auch Repositories. Dies widerspricht der hier vorgestellten Definition von Repositories, die nur für Aggregat-Wurzeln existieren sollen.

Die Kind-Entitäten sind, wie schon angedeutet zyklisch mit ihren Eltern-Entitäten referenziert. Ein Job kennt alle seine Dokumente und ein Dokument seinen Job. Das ist bei den meisten Eltern-Kind-Beziehungen so.

```
// Eltern-Klasse
public class Job {
  ...
  @OneToMany (mappedBy = "job", cascade = CascadeType.ALL,
     orphranRemoval = true)
  private List<Document> documents;
  ...
}

// Kind-Klasse
public class Document {
  ...
  @ManyToOne (cascade = CascadeType.Refresh)
  private Job job;
  ...
}
```

Die ManyToOne Beziehungen, wie in Document wurden jetzt entfernt. Das *mappedBy* in der Annotation der Elternklassen ist deshalb nicht mehr nötig. Jedoch benötigt JPA nun eine Information, welche Spalte in der Kind-Entität für das Bilden der Beziehung nötig ist. Dies ist mit einer weiteren Annotation *@JoinColumn* möglich. Die neuen Beziehungsdeklarationen sehen so aus:

```
// Eltern-Klasse
public class Job {
  // ...
  @OneToMany (cascade = CascadeType.ALL, orphranRemoval = true)
  @JoinColumn (name = "job_id")
  private List<Document> documents;
  // ...
}

// Kind-Klasse
public class Document {
  // ...
}
```

Dies führte jedoch dazu, dass einige Methoden von Services, die Eltern-Objekte der Kind-Objekte nicht mehr auflösen konnten. Die Lösung dafür war, die entsprechenden Eltern-Objekte beim Methodenaufruf mitzugeben.

```
// vorher
public Path buildPdf(DocumentGroup documentGroup) {
  // ...
}

// nacher
public Path buildPdf(Job job, DocumentGroup documentGroup) {
  // ...
}
```

Die Entität *DocumentGroup* entspricht der oben beschriebenen *SU*. Diese heißt anders, weil bei der Entwicklung versucht wurde, das Konzept der SU zu generalisieren. Dies wird später behoben.

REST-Controller, die Operationen auf Kind-Objekte bereitstellen, müssen nun auch das entsprechende Elternobjekt laden, um weiter funktionieren zu können.

```
// vorher
@GetMapping("/jobs/{jobName}/documentgroups/{documentGroupId}")
public FileSystemResource buildPdf(
  @PathVariable int documentGroupId) {

  DocumentGroup documentGroup = documentGroupRepository.
     findById(documentGroupId);
  Job job = documentGroup.getJob();
  // ...
}

// nacher
@GetMapping("/jobs/{jobName}/documentgroups/{documentGroupId}")
public FileSystemResource buildPdf(
  @PathVariable String jobName,
  @PathVariable int documentGroupId) {

  Job job = jobRepository.findByName(jobName);
  DocumentGroup documentGroup = documentGroupRepository.
     findById(documentGroupId);
  // ...
}
```

Das Entfernen der zyklischen Abhängigkeiten war nicht sonderlich aufwendig. Es gab wenig Situationen, in denen zusätzlich zu der Entität auch die Eltern-Entität benutzt wurde. Das Datenbank-Schema ist bei der Umstellung gleichgeblieben. Das Risiko war hier gering. Aufwendig war die Umstellung der Controller, weil hier etliche Lade-Routinen hinzugefügt werden mussten.

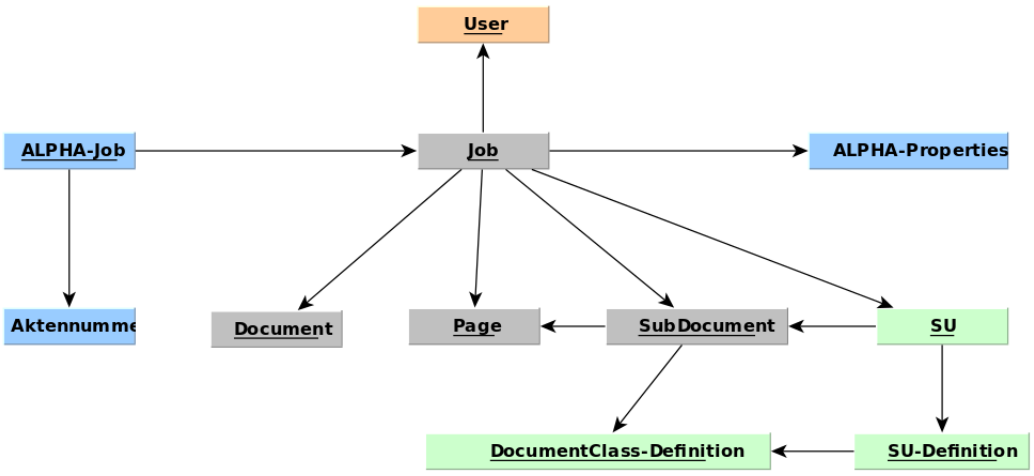

Abbildung 9: Entitäten nach Refaktorisierung der zyklischen Abhängigkeiten. Die Pfeile, die die Referenzen abbilden, zeigen nun nur noch in eine Richtig.

Für sich allein gestellt scheint diese Maßnahme den Vorteil eines schwächer gekoppelten Modells zu liefern. Jedoch führt das zu einem Anstieg der Komplexität in der Benutzung des Modells, weil in den Entitäten nicht mehr zurück navigiert werden

kann.

2.3 Aufteilung des Gesamtmodels in Aggregate

Als nächstes möchte ich das Gesamtmodell in Aggregate aufteilen, wie sie schon einmal beschrieben worden sind. Ein Beispiel für ein solches Aggregat ist hier der Job. Ein Job beschreibt den Arbeitsablauf von BETA. Er beinhaltet Dokumente, die Analysiert werden sollen, die Seiten dieser Dokumente und die Teildokumente, die aus den genannten Seiten bestehen. Dies ist die Zielvorstellung für den Job. Momentan enthält ein Job jedoch noch mehr: die SUs, die von der DELTA kommen können und manche Eigenschaften, die mit dem Kunden ALPHA zusammenhängen. Diese müssen aus Job raus und in ihre eigenen Aggregate gesteckt werden.

Eine **SU** gruppiert mehrere Teildokumente. Ein Teildokument besitzt eine Dokumentenklasse, die die Art eines Teil-Dokuments angibt. Auch SUs haben einen Typ, wie zum Beispiel *Legitimation*. Zur SU vom Typ *Legitimation* können Teildokumente mit der Dokumentenklasse *Personalausweis* und/oder *Reisepass* passen. Diese Beziehung zwischen SU-Typ und möglichen Dokumentenklassen ist vorgegeben und durch eine entsprechende Tabelle abgebildet. Dieses statische Mapping ist momentan über jedes Teildokument und jede SU zugänglich. Dies ist jedoch nur nötig, wenn für die im Job gesetzten SUs die passenden Teildokumente gefunden werden sollen. Die Zuordnung zwischen SU-Typen und Dokumentenklassen ist daher für mich in einem eigenen Aggregat mit dem Namen *SU-Definition* abzubilden. Die Referenzen der SUs auf die SU-Definitionen werden entfernt.

SUs sollen nicht im Job-Objekt gehalten werden. Der Job soll Teil des Kern-Kontextes werden, wohingegen die SU ein Konzept aus dem DELTA-Kontext ist. Deshalb soll es ein weiteres Aggregat geben, welches *DELTA-Job* heißen soll. Dies enthält die SUs und weitere DELTA relevante Informationen und eine Referenz auf das Job-Aggregat.

Informationen, die zu ALPHA gehören, werden auch aus Job gezogen und in ein eigenes *ALPHA-Job* Aggregat verschoben, welches wieder eine Referenz zu dem entsprechenden Job besitzt. Im Gegensatz zu DELTA ordnet ALPHA die Teildokumente sogenannten *Aktennummern* zu.

Auch werden Klassen umbenannt, die ein in der ubiquitären Sprache vorkommendes Konzept anders benennen. So zum Beispiel wird *DocumentGroup* zu *SU* umbenannt.

```
// vorher
public class Job {
  // ...
  @JoinColumn (name = "job_id")
  @OneToMany (cascade = CascadeType.ALL, orphranRemoval = true)
  private List<DocumentGroup> documentGroups;
  // ...
}

// nacher
public class DELTAJob {
  // ...
  @OneToOne
  private Job job;

  @JoinColumn (name = "DELTA_job_id")
  @OneToMany (mappedBy = "job", cascade = CascadeType.ALL,
     orphranRemoval = true)
  private List<DocumentGroup> documentGroups;
  // ...
}
```

Die Entitäten eines Aggregats sollen nur über die Aggregat-Wurzel angesprochen werden. Die zuvor beschriebenen Repositories für die Kind-Entitäten werden deswegen entfernt und im Job Methoden zur Verfügung gestellt, die diese Aufgabe stattdessen übernehmen. Das Speichern neuer Kind-Entitäten wird von der Aggregat-Wurzel übernommen. Auch kann die Aggregat-Wurzel jetzt die übergebenen Kind-Elemente validieren und somit die Korrektheit des Aggregats gewährleisten.

```
public class Job {
  // ...
  @JoinColumn (name = "job_id")
  @OneToMany (cascade = CascadeType.ALL, orphranRemoval = true)
  private List<SubDocument> subDocuments;

  public SubDocument getSubDocument(int subDocumentId) {
    return subDocuments.stream()
      .filter(s -> s.getId() == subDocumentId)
      .findAny()
      .orElseThrow(() -> new NoSuchElementException());
  }

  public void addSubDocument(SubDocument subDocument) {
    Validate.isNotNull(subDocument);
    // fachliche validierung, falls noetig ...
    subDocuments.add(subDocument)
  }
  // ...
}
```

Die Benutzung der Repositories für die Kind-Entitäten ist in den Controllern nicht mehr möglich und auch nicht mehr nötig. Dort muss der Code wieder angepasst wer-

den. Der Zugriff auf die Kind-Entitäten über die Repositories wird durch die Methode der Aggregat-Wurzel ersetzt.

```
// vorher
@GetMapping("/jobs/{jobName}/subdocuments/{subDocumentId}")
public FileSystemResource buildPdf(
  @PathVariable String jobName,
  @PathVariable int subDocumentId) {

  Job job = jobRepository.findByName(jobName);
  SubDocument subDocument = subDocumentRepository.findById(
     subDocumentId);
  // ...
}

// nacher
@GetMapping("/jobs/{jobName}/subdocuments/{subDocumentId}")
public FileSystemResource buildPdf(
  @PathVariable String jobName,
  @PathVariable int subDocumentId) {

  Job job = jobRepository.findByName(jobName);
  SubDocument subDocument = job.getSubDocument(subDocumentId);
  // ...
}
```

Diese Refaktorisierung war moderat Komplex. Wieder mussten fast alle Controller angepasst werden, um den Zugriff auf die Kind-Entitäten zu ändern. Die Controller sind jedoch von weniger Repositories als vorher abhängig. Externe fachliche Validierung jetzt im Controller nicht mehr nötig, weil die Aggregat-Wurzel dies Übernimmt. Dadurch übernimmt die Aggregat-Wurzel neue Aufgaben. In diesem Falle die Verwaltung ihrer Kind-Entitäten. Die Code-Komplexität ist meiner Meinung nach nicht gesunken, sondern ist in etwa gleichgeblieben, weil die eigentliche Komplexität mehr zu dem Aggregat gewandert ist.

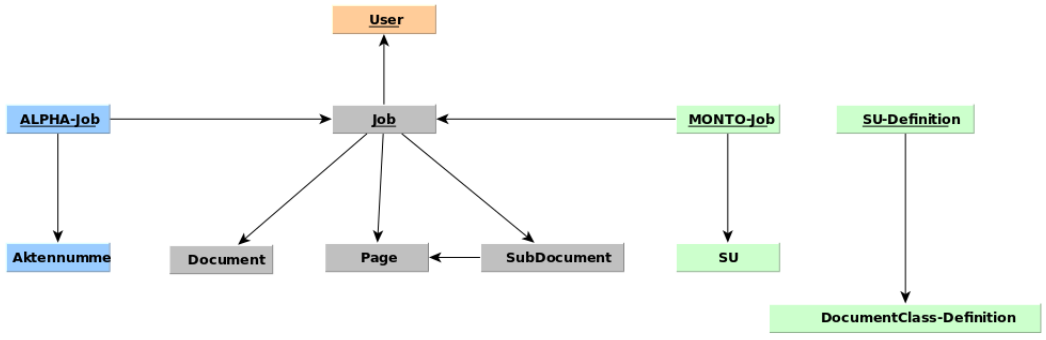

Abbildung 10: Entitäten nach Refaktorisierung der Aggregate. Zugriff auf Kind-Entitäten nur noch über die Aggregat-Wurzel (Unterstichen), Aggregate referenzieren nur Auf fremde Aggregat-Wurzeln

2.4 Definition von fachlichen Identitäten

Eine Entität besitzt eine **Identität**. Es gibt verschiedene Arten und Weisen, wie diese Modelliert werden kann.

Die Identität kann vom *Nutzer angegeben* werden. Dies kann jedoch zu Problemen führen. Angenommen ein Forum wird durch seinen Titel identifiziert. Da sich die Identität einer Entität in seinem Lebenszyklus optimaler Weise nicht ändert, können Rechtschreibfehler im Titel so nicht so einfach geändert werden. Deswegen muss bei solchen Identitäten gut darüber nachgedacht werden, was vom Nutzer als Identität benutzt wird.

Die Identität kann *von der Applikation generiert werden*. Dazu gibt es verschiedene Ansätze, wie die Verwendung einer UUID. Es kann von Vorteil sein, wenn die Identität etwas über die Entität aussagt. Dazu kann man zum Beispiel die Eigenschaften der Entität in die Identität kodifizieren (Beispiel *P-TL-15031995*, eine Person, mit den Initialen *TL*, die am 15.03.1995 geboren worden ist). Falls es mehrere verschiedene Entitäten mit einer solcher Identität gibt, kann zusätzlich noch ein zufälliges Element eingebettet werden.

Die Identität kann *durch die Persistenz* generiert werden. Datenbanken wie MySQL, PostgreSQL oder auch MongoDB bieten Mechanismen an, automatisch Identitäten zu erzeugen. Dies sind meist fortlaufende Nummern oder zufällige Zeichenketten. Diese Identitäten werden meist erst vergeben, wenn die Entität gespeichert wird und steht deswegen bei der Erstellung der Entität noch nicht zur Verfügung.

Die Identität kann durch *einen anderen Kontext vorgegeben* werden. Dabei ist die Entität durch das Context-Mapping mit der Entität eines anderen Bounded Context verbunden.

Jedes Aggregat soll **global identifizierbar** sein. Die Aggregat-Wurzel ist träget dieser Identität. Die Identität der Kind-Entitäten muss hingegen nur in Bezug auf ihr Aggregat einzigartig sein. Dies nennt man auch **lokale Identität**. [5]

Die Identität eines Jobs in BETA wird durch den Kunden vorgegeben. Die DELTA besteht aus sogenannten *Kundenprojekten*. Für jeden dieser Kundenprojekte soll ein Job in BETA existieren, der dessen Dokumente verarbeitet. Der korrespondierende DELTA-Job in BETA besitzt also die gleiche Identität wie die des Kundenprojekts. Gleiches gilt für den Kunden ALPHA. Sie Scannen ihre neuen Dokumente jeden Tag einmal ein und schicken sie an BETA. Für jeden Scan-Prozess, der eine Nummer vergeben bekommt, existiert ein ALPHA-Job, dessen Identität, der Nummer des Scans entspricht.

Jeder DELTA-Job und ALPHA-Job referenziert einen Job im Kern-Kontext. Dieser hat als Identität lediglich einen Namen. Die Kunden-Jobs vergeben diesen Namen selber anhand ihrer Identität. Da alle Job-Namen für die verschiedenen Nutzer von BETA gleichermaßen verwendet werden sollen, hat jeder Job den Nutzernamen des Nutzers als zusätzliche Komponente der Identität. Die Identität eines Jobs besteht also aus zwei Teilen. Da dies in Hibernate[7], was von Spring-Data-JPA verwendet wird, jedoch sehr unhandlich ist, bekommt Job hier zwei Identitäten. Einmal die fachliche Identität, die vom Kunden vorgegeben wird und eine technische Identität, die vom Datenbank-System generiert wird. Dies nennt man auch **Surrogate Identity** [5]. Für

[7] Implementierung einer Persistenzschnittstelle, um mit Datenbanken zu kommunizieren

eine bessere Handhabung der Job-Identität, werden ihre Komponenten in ein Value Objekt *JobId* gepackt.

Der Nutzer, dem der Job gehört, wird noch direkt referenziert. Dieser Nutzer wird aber primär für die Authentifizierung benutzt. Da dies eine ganz andere Aufgabe ist, wird diese Referenz entfernt und durch den Nutzernamen ersetzt.

```
@EqualsAndHashcode (of = "jobId")
public class Job {
  @Id
  @GeneratedValue
  @SuppressWarnings ("unused")
  private Long id;

  @Embedded
  private JobId jobId;
  //...
}
@Embeddable
@Getter
@NoArgsConstructor (access = AccessLevel.PROTECTED)
@AllArgsConstructor (access = AccessLevel.PRIVATE)
@EqualsAndHashcode
public class JobId implements Serializable {

  private String jobName;
  private String userName;

  public static JobId of(String jobName, String userName) {
    Validate.isTrue(StringUtils.isNotEmpty(jobName));
    Validate.isTrue(StringUtils.isNotEmpty(userName));

    return new JobId(jobName, userName);
  }
}
```

In diesem Codebeispiel wird **Lombok** verwendet. Das ist ein Java-Bibliothek, der aus Annotationen entsprechende Code-Konstrukte erzeugt. Aus *@Getter* werden zum Beispiel für alle Felder der Klasse eine Get-Methode erzeugt. Dies reduziert etwas den Ritual-Code[8] von Java. Die *@EqualsAndHashcode* Annotation ist hierbei besonders nützlich. Sie implementiert für alle gewählten Felder die *equals* und *hashCode* Methode. [2]

Dokumente sollen eine lokale Identität in Bezug auf den Job besitzen. Ein solches Dokumente besitzt einen Namen, einen Speicherort und einen Hochladezeitpunkt. Die Dokumente sollen in einer bestimmten Reihenfolge analysiert werden, die eventuell vom Hochladezeitpunkt abweicht. Deswegen existiert für das Dokument noch ein Sortier-Index. Diesen will ich als lokale Identität benutzen. Dieser Index wird jedoch nur indirekt vom Nutzer festgelegt und müsste deswegen generiert werden. Da der Job seine Dokumente verwaltet, macht es für mich Sinn, dass der Job diese Identität bestimmt. Bei dem hinzufügen des Dokuments zum Job, passt der Job dann den

[8] Codeteile, die immer wieder vor kommen, aber keinen mehrwert liefern

Index des Dokumentes an. Auch soll das Abfragen der Kind-Entitäten über die neue fachliche Identität geschehen.

```
public class Document {

  // technische ID
  @Id
  @GeneratedValue
  private int id;

  // fachliche ID
  private int sortIndex;

  // Getter, Setter...
}
@EqualsAndHashcode (of = "jobId")
public class Job {
  //...
  private List<Document> documents;

  public void aDomain-Driven Designocument(Document document) {
    Validate.isNotNull(document);
    document.setSortIndex(nextDocumentSortIndex());
    documents.add(document);
  }

  private Integer nextDocumentSortIndex() {
    return documents.stream()
      .map(Document::getSortIndex)
      .max(Comparator.naturalOrder())
      .map(x -> x + 1)
      .orElse(0);
  }

  public Document getDocument(int sortIndex) {
    return documents.stream()
      .filter(d -> d.getSortIndex() == sortIndex)
      .findAny()
      .orElseThrow(() -> new NoSuchElementException());
  }
  //...
}
```

Diese Umstellung betrifft im Java-Code nur die Bestandteile des Modells. Services und Controller mussten durch diese Refaktorisierung nicht angepasst werden. Jedoch hat sich das Datenbank-Modell stark geändert. Zusätzlich zu den technischen Identitäten existieren nun fachliche Identitäten, die nicht unter der Kontrolle des Datenbanksystems sind. Dies transferiert die Komplexität der Datenhaltung zu den Aggregaten. Zwar bedeutet dies mehr Aufwand im Quellcode, man ist jedoch unabhängiger davon, welche Datenbank-Architektur man benutzt und kann diese Einfacher austauschen, um zum Beispiel auf eine NoSQL Datenbank, wie *MongoDB*, umzusteigen. Trotzdem stellt das neue Datenbank-Schema ein Problem dar. Für die neuen Identitä-

ten müssen neue Spalten hinzugefügt werden müssen. Diese Spalten müssen jedoch mit Werten vorbelegt werden, die durch das Aggregat bestimmt werden. Das würde bedeuten, dass die Logik für das erstellen der fachlichen Identitäten in SQL nach programmiert werden müsste. Um dem aus dem Weg zu gehen, habe ich mir überlegt ein Migrations-Tool zu schreiben, welches die entstandenen Aggregate in JSON-Objekt[9] überträgt. Später, wenn das Datenbank-Schema aktualisiert worden ist, werden diese JSON-Objekte wieder eingelesen. Da das Managen der lokalen fachlichen Identitäten im Modell selber passiert, muss sich darüber beim Transfer keine Gedanken mehr gemacht werden.

Diese Refaktorisierung war relativ komplex. Das finden und implementieren der fachlichen Identitäten stellt sich nicht als all zu kompliziert heraus. Indes war die Überlegung, wie die alten Daten in das neue Schema migriert werden sollen eine komplexere Herausforderung. Das Erstellen der Migrations-Werkzeuge war zwar nicht sonderlich schwierig, aber sehr aufwendig, da zweimal in ein jeweils anderes Modell übertragen werden musste. Diese Werkzeuge mussten gut getestet werden, um eine saubere Migration der Bestandsdaten zu gewährleisten.

Vorher hat man auch von außen mit technischen Identitäten auf die einzelnen Kind-Entitäten zugegriffen. Nun geschieht das über die fachliche Identität. Das ist in meinen Augen ein großer Mehrwert.

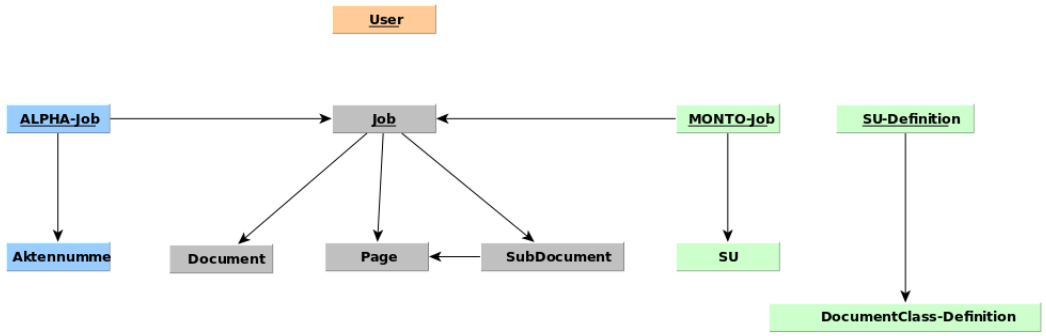

Abbildung 11: Entitäten nach Refaktorisierung der Identitäten. Job besitzt keinen Bezug auf User mehr.

Die Aggregate haben mittlerweile viel Verantwortung über ihre internen Objekte übernommen. Sie können die hinzuzufügenden Elemente fachlich validieren und übernehmen das teilweise das Identitätsmanagement.

[9] Datenformat, das aus Objekten, Schlüssel-Wert-Paaren und Listen besteht

2.5 Übertragung der Fachlogik zu Aggregaten und Domain-Services

Die letzte Aufgabe soll sein, auch die fachliche Logik zum Modell zu übertragen. Dies widerspricht der verbreiteten Ansicht, dass Daten und Logik strikt voneinander zu trennen sind.[5] Für Produkte, die sich auf die Verwaltung von vielen Daten konzentrieren, mag dies auch ein guter Ansatz sein. BETA ist jedoch auf die Abbildung eines Prozesses ausgelegt und dafür eignet sich die Kombination von Daten und Verhalten sehr gut. Dazu überträgt man die rein fachlichen Aspekte in entsprechende Methoden der Entitäten und Value Objekte. Ein Anwendungsbeispiel hier wäre folgender: ein Job kann erst analysiert werden, wenn er mindestens ein Dokument besitzt. Auch kann man einem Job keine neuen Dokumente hinzufügen, wenn er schon analysiert worden ist oder gerade analysiert wird. Ein Job hat deswegen mehrere Zustände: leer, bereit, in Analyse und fertig. Je nach Status soll der Job beim Hinzufügen eines Dokuments den Status auf bereit wechseln oder die Anmahne des Dokumentes verweigern.

```
public enum JobStatus {
  EMPTY,
  READY,
  IN_ANALYSIS,
  FINISHED
}

@EqualsAndHashcode (of = "jobId")
public class Job {
  //...
  private List<Document> documents;
  private JobStatus status;

  public void addDocument(Document document) {
    Validate.isNotNull(document);

    if (status != JobStatus.EMPTY && status != JobStatus.READY)
      throw new IllegalStateException("Dem job duerfen keine
          Dokumente hinzugefuegt werden, weil im falschen
          zustand ist.");

    document.setSortIndex(nextDocumentSortIndex());
    documents.add(document);
    state = JobState.READY;
  }

  public boolean canAnalyze() {
    return status == JobStatus.READY;
  }
  //...
}
```

Einen Unterschied bilden jedoch fachliche Aufgaben, die mehr als eine Entität betreffen, oder keiner Entitäten eindeutig zugeordnet werden kann. Dafür gibt es **Domain Services**. Ein Domain Service bildet genau eine Aufgabe ab. Er hat eine

genau definierte Schnittstelle und legt Vor- und Nachbedingungen fest, zu dessen Erfüllung er sich verpflichtet. Ein Beispiel dafür ist ein Service zur Erstellung von globalen fachlichen Identitätenfür ein Aggregat. Dieser Service garantiert, dass die erstellte Identität vorher nicht existiert hat, sodass das neue Aggregate eindeutig von den anderen Unterscheidbar sind.

Es gibt aber auch Aufgaben, die technischer Natur sind. Die Kommunikation mit externen Schnittstellen oder Ein- und Ausgabeoperationen fallen zum Beispiel darunter. Die Klassen des Domain-Modells sollen nur die fachlichen Aspekte abbilden. Technische Aspekte gehören nicht dazu. Diese Aufgaben übernehmen **technische Services** oder auch **Applikation-Services**. Diese sind dazu da, das fachliche Modell zu integrieren und für die Außenwelt zugänglich zu machen. Sie bilden meist eine Schicht um das Domain-Model. Die zuvor gezeigten Controller sind beispielsweise solche technischen Services, weil Sie die Schnittstelle nach außen bereitstellen und so den Zugang zum Domain-Model ermöglichen [5]

Es gibt auch die Möglichkeit, dass im fachlichen Modell ein Service benötigt wird, der eine technische Implementierung erfordert. Ein Beispiel sind die schon einmal erwähnten Repositories. Diese werden meist als Interface definiert. Da bei BETA *Spring-Data-JPA* verwendet wird, stellt das Framework die Implementierung bereit. Andernfalls müsste man dieses Interface selbst implementieren. Da es sich bei den Repositories um Schnittstellen für das Speichern und Laden von Daten handelt, sind die Implementierungen als technische Services anzusehen. Durch Techniken wie Dependency-Injection stehen diese Implementierungen dann den Klassen des Domain-Modells zur Verfügung. Das Domain-Model steht dabei im Zentrum und wird durch die technischen Services ummantelt. Dies wird auch Hexagonale Architektur genannt. Hier implementieren die Applikations-Services die technischen Komponenten des Produkts. Diese werden dann durch Adapter mit der Außenwelt verknüpft. [5]

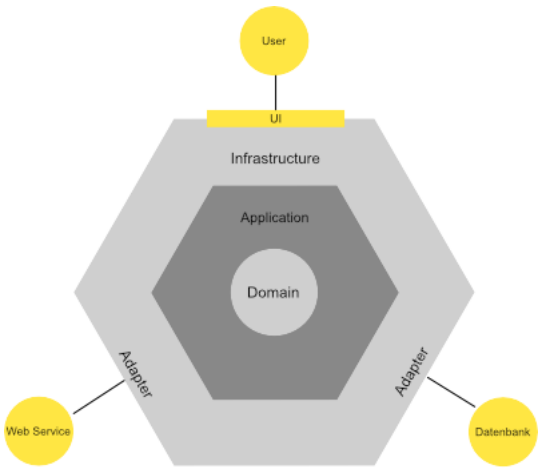

Abbildung 12: Hexagonale Architektur. Das Domain-Modell ist im Zentrum. Ports und Adapter greifen durch die Applikations-Schicht auf das Modell zu.

Vorher hatte BETA auch schon Services. Diese bildeten jedoch sowohl fachliche Logik als auch technische Aufgaben ab. Die Aufgabe besteht also darin, diese beiden Aspekte voneinander zu trennen. Die Fachlogik wandert in die Entitäten oder Domain-Services und die technischen Aufgaben in technische Services. Damit diese auch örtlich getrennt sind, habe ich mich an der Paketstruktur eines Domain-Driven Design Beispielprojekts orientiert, das unter anderem auch von Eric Evans, dem ursprünglichen Autor von *Domain-Driven Design* entworfen worden ist [3]

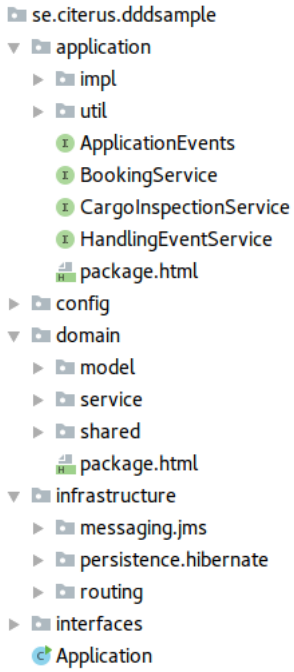

Abbildung 13: Java-Pakete eines Beispielprojektes für Domain-Driven Design.

Nun sind fachliche und technische Aufgaben voneinander getrennt. Die eigentliche Herausforderung war, den richtigen Ort für die fachliche Logik zu finden und abzuwägen, ob für eine bestimmte Aufgabe ein Service notwendig ist. Positiv herausgestellt hat sich, dass nun das Testen der fachlichen Bestandteile sehr einfach geworden ist, da sie nicht mehr von technischen Aspekten abhängen. Die Test-Klassen müssen nun nur noch sehr selten bestimmte Klassen mocken. Spring-Data-JPA stellt auch Test-Implementierungen für die Repositories bereit, weswegen man sich darüber auch keine Gedanken machen muss.

Die Klassen, die dem Domain-Modell angehören, sind nun sehr Feature-Reich geworden. Die Aggregate bilden nun die kompletten fachlichen Aspekte von BETA ab. Das hat aber auch zu folge, dass wenn die Aggregate komplexer geworden sind, und deren Sourcecode-Dateien sehr lang werden können. Deswegen sollten die Aggregate schmal gehalten werden, um diese interne Komplexität nicht allzu sehr ansteigen zu lassen.

Dieser Umbau hat sich sehr gelohnt. Nicht nur die technischen Services sind schmaler geworden, sie haben nun auch keine fachlichen Aufgaben mehr. Und wenn, dann sind diese durch ein Interface festgelegt. Ein Beispiel dafür ist, dass der Controller, der für einen Job die Dokumente annimmt, nun nicht mehr überprüfen muss,

ob der Job im richtigen Zustand ist und Dokumente annehmen darf.

```
@RestController
public class DocumentController {
  public void postDocument(
    @PathVariable String jobName,
    Principal pricipal,
    @RequestParam MultipartFile file) {

    JobId jobId = JobId.of(jobName, principal.getName());
    Job job = jobRepository.getJob(jobId);

    Document document = buildDocumentFromMultipartFile(file);

    // Hier wuerde Job eine Exception werfen, wenn das Dokument
        nicht angenommen werden darf
    job.addDocument(document);

    jobRepository.save(job);
  }
}
```

2.6 Umstrukturierung der Module Teil 2

Nun sind etliche Refaktorisierungen am Modell abgeschlossen. Sie hatten das Ziel, die Fachlichkeit hin zum Domain-Modell zu verfrachten und schwach voneinander abhängende Aggregate zu bilden. Nun kann erneut versucht werden, die neue Modulstruktur durchzusetzen.

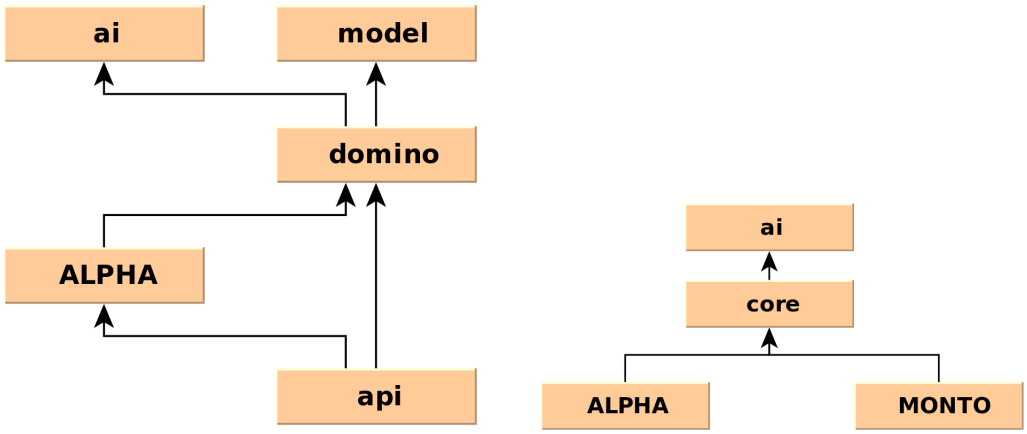

Abbildung 14: Alte Modulstruktur (links) und neue Modulstruktur (rechts).

Die Module *model* und *domino* fallen zusammen und bilden das Kern-Modul. Anschließend können die Aggregate und die fachlichen und technischen Services, die für den Kontext *DELTA* und *ALPHA* gedacht sind in die neuen Module verschoben werden. Da für das Context-Mapping zwischen dem Kern-Kontext und den Kunden-Kontexten *Partnership* gewählt wurde, hängen die beiden Kunden-Module direkt vom

Kern-Modul ab. Deshalb muss zwischen den Modulen keine weitere Abstraktionsschicht erfunden werden. Da auf diese Refaktorisierung hingearbeitet wurde, brauchen die entsprechenden Klassen nur noch verschoben werden. In den neuen Modulen wird die gleiche Paketstruktur verwendet, wie sie auch im Kern-Modul eingeführt wurde. Dadurch sind alle Module in ihrem Aufbau gleich.

3 Fazit

Insgesamt möchte ich behaupten, dass sich die Refaktorisierung gelohnt hat. Manche Refaktorisierungen fühlen sich für sich eher uneffektiv an, fügen sich aber gut in das Gesamtresultat ein. Das Entfernen der Rückreferenzen im Modell schienen die Abhängigkeiten etwas zu lockern, jedoch mussten in einigen stellen, die mehrere Elemente des Modells benutzen, doppelt auf die Repositories zugegriffen werden. Dies für sich allein hätte ich vermutlich so nicht umgesetzt. Im Zusammenspiel mit der darauffolgenden Maßnahme (dem Aufteilen des Modells in Aggregate) und den Zugriff auf das Modell, ausschließlich über die Aggregat-Wurzeln, ergab dies jedoch durchaus Sinn, weil die Rückreferenzen dadurch sowieso nicht mehr benötigt wurden.

Die Implementierung der neuen Identitäten hat den Code nicht einfacher gemacht. Jedoch haben sie nun eine fachliche Bedeutung, was die Benutzung und Wartung des Codes intuitiver gestaltet.

Die beste Maßnahme war jedoch die Übertragung der Fachlogik auf die Aggregate und Domain Services. Der Testaufwand der einzelnen Komponenten ist deutlich einfacher geworden. Durch die neue Paketstruktur, die das fachliche Modell und die technische Integration voneinander trennt, sollte es nun deutlich intuitiver sein, wo man nach bestimmten Teilen der Software suchen um, neue Anforderungen umzusetzen.

Ein negativer Aspekt ist, dass die Entitäten durch die Übernahme diverser fachlicher Aufgaben vergleichsweise groß geworden sind. Es mag Leute geben, die diese Herangehensweise eher vermeiden möchten. Ich glaube jedoch, dass das für BETA gut funktionieren kann.

Literaturverzeichnis

[1] http://spring.io/projects/spring-boot.

[2] https://projectlombok.org.

[3] Citerus. https://github.com/citerus/dddsample-core, Mar 2018.

[4] Eric Evans. *Domain-driven design reference: definitions and patterns summaries*. Dog Ear Publishing, 2015.

[5] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley, 2015.