Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Maximilian H. Zender

Domain-Driven Design mit Ruby on Rails

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Maximilian H. Zender

Domain-Driven Design mit Ruby on Rails

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 19. Oktober 2015

Maximilian H. Zender

Thema der Arbeit
Domain-Driven Design mit Ruby on Rails

Stichworte
Ruby, Ruby on Rails, Domain-Driven Design, Webanwendungen, Softwarearchitektur

Kurzzusammenfassung
Domain-Driven Design ist ein Ansatz zur Modellierung und Implementierung komplexer Software. Diese Bachelorarbeit untersucht die Vereinbarkeit von Domain-Driven Design mit Ruby on Rails, einem Framework für Webanwendungen. Dabei werden insbesondere Möglichkeiten für die Implementierung der Konzepte aus Domain-Driven Design in Verbindung mit Ruby on Rails aufgezeigt.

Maximilian H. Zender

Title of the paper
Domain-Driven Design with Ruby on Rails

Keywords
Ruby, Ruby on Rails, Domain-Driven Design, web applications, software architecture

Abstract
Domain-Driven Design is an approach to modelling and implementing complex software. This bachelor thesis examines the compatibility of Domain-Driven Design and Ruby on Rails, a web application framework. In particular, it illustrates possibilities for implementing the concepts of Domain-Driven Design in conjunction with Ruby on Rails.

Inhaltsverzeichnis

1 Einleitung

Ruby on Rails ist ein Framework für Webanwendungen und ermöglicht eine rasche Anwendungsentwicklung durch Prinzipien, wie *Konvention vor Konfiguration*, wonach die aufwendige Konfiguration der Anwendung zu vermeiden und stattdessen festgelegte Konventionen einzuhalten sind. Durch die lose Struktur von Anwendungen, die auf Ruby on Rails basieren, eignet es sich gut für die agile Softwareentwicklung, bei der sich auch in späteren Entwicklungsstadien Anforderungen aufgrund von neu gewonnenen Erkenntnissen ändern können.

Wächst aus der einstmals überschaubaren Anwendung allerdings eine komplexe heran, kann die Beibehaltung der losen Struktur zu Problemen führen: Wo es zu Beginn des Projekts von Vorteil war Zugriff auf beliebige Teile der Anwendung zu haben, fordert die dadurch entstandene enge Kopplung mit zunehmender Komplexität ihren Tribut. Die Vermischung von Verantwortlichkeiten verschlechtert die Wartbarkeit und Erweiterbarkeit der Anwendung.

Domain-Driven Design bietet eine Reihe von Konzepten und Vorgehensweisen, die bei der Entwicklung komplexer Anwendungen förderlich sind. Dazu gehören Methoden zur Verbesserung der Kommunikation innerhalb des Teams, Techniken zur Modellierung komplexer Domänen, und Entwurfsmuster, die dabei helfen Modelle in Code umzusetzen. Unter anderem durch die Einführung einer sogenannten *Domänenschicht*, die sämtliche Geschäftslogik von Infrastruktur- und Darstellungslogik kapselt, sollen Verantwortlichkeiten strikt getrennt und eine gute Wartbarkeit und Erweiterbarkeit von komplexen Anwendungen erreicht werden.

Im Rahmen dieser Bachelorarbeit soll untersucht werden, ob und inwieweit Ruby on Rails mit den Konzepten von Domain-Driven Design vereinbar ist. Dabei wird nur die Umsetzung eines bestehenden Modells in Code betrachtet. Der Nutzen bei der Modellierung von Domänen und die projektstrategischen Vorzüge von Domain-Driven Design sind nicht Gegenstand dieser Arbeit.

1.1 Struktur der Arbeit

Kapitel 2 erläutert die für das Verständnis der Arbeit benötigten Grundlagen. Es beginnt mit einer Einführung in Ruby, darauf folgt ein kurzer Überblick über Ruby on Rails und es schließt mit der Einführung in Domain-Driven Design.
Kapitel 3 beschreibt eine Anwendung, an deren Beispiel Probleme mit Ruby on Rails beschrieben werden, die mithilfe von Domain-Driven Design gelöst werden sollen.
Kapitel 4 erläutert zunächst die Implementierung der Domänenschicht und zeigt Lösungen für deren Isolation vom Rest der Anwendung. Es folgen Lösungmöglichkeiten für die Persistierung von Änderungen in der Domänenschicht in den Abschnitten 4.2 und 4.3.
Kapitel 5 bewertet die in Kapitel 4 gezeigte Lösung und die Vereinbarkeit von Domain-Driven Design und Ruby on Rails.

2 Grundlagen

2.1 Die Programmiersprache Ruby

Ruby ist eine objektorientierte *General Purpose Language* und wurde im Jahre 1993 von dem Japaner Yukihiro Matsumoto entwickelt. Ruby ist dynamisch typisiert und wird von einem Interpreter ausgeführt. Dieses Kapitel soll einen knappen Überblick über die Hauptmerkmale der Sprache bieten.

2.1.1 Klassen

In Ruby werden Klassen mit dem Schlüsselwort `class` gefolgt vom Klassennamen definiert, Methoden mit `def` gefolgt vom Methodennamen und etwaigen Parametern.

```ruby
class Person
  def greet(name)
    puts "Hello #{name}"
  end
end
```

Listing 2.1: Definition einer Klasse in Ruby.

Die Instanziierung von Objekten erfolgt dann durch den Aufruf der Methode `new` auf dem Klassennamen. Klammern sind in Ruby optional, solange durch ihr Fehlen keine Mehrdeutigkeiten entstehen:

```ruby
person = Person.new
person.greet("Frank") # Ausgabe: "Hello Frank"
person.greet "Frank"  # Ausgabe: "Hello Frank"
```

Listing 2.2: Instanziierung von Objekten und Aufruf von Methoden in Ruby.

Klassen werden mit sog. *Initializern* initialisiert. Diese tragen den Namen `initialize`. Die übergebenen Argumente können dann Instanzvariablen zugewiesen werden. Diese werden mit einem @ gekennzeichnet:

```ruby
class Person
  def initialize(name)
    @name = name
  end

  def introduce
    puts "My name is #{@name}"
  end
end

person = Person.new("John Doe")
person.introduce # Ausgabe: "My name is John Doe"
```

Listing 2.3: Zuweisung von Instanzvariablen im Initializer einer Klasse.

Vererbungsbeziehungen werden mit einer spitzen Klammer definiert:

```ruby
class VIP < Person
  def introduce
    puts "My name is #{@name} and I'm very important"
  end
end

vip = VIP.new("John Doe")
vip.introduce # Ausgabe: "My name is John Doe and I'm very important"
```

Listing 2.4: Umsetzung einer Vererbungsbeziehung in Ruby.

Klassenmethoden können deklariert werden indem dem Methodennamen das Schlüsselwort `self` vorangestellt wird:

```ruby
class Person
  def self.greet
    puts "Hello"
  end
end

Person.greet # Ausgabe: "Hello"
```

Listing 2.5: Deklaration von Klassenmethoden.

2.1.2 Datentypen

In Ruby existieren keine sogenannten *primitiven* Datentypen, wie in vielen anderen Sprachen. Alle Datentypen sind Objekte, auf denen Methoden aufgerufen werden können:

```ruby
"hello".length #=> 5
7.odd? #=> true
```

Listing 2.6: Alle Datentypen in Ruby sind Objekte.

Sie können sogar zur Laufzeit erweitert werden:

```ruby
class Integer
  def plus_one
    self + 1
  end
end

7.plus_one #=> 8
```

Listing 2.7: Erweiterung der Klasse Integer um die Methode `plus_one`.

Das Schlüsselwort `self` gibt die aktuelle Instanz der Klasse zurück.

2.1.3 Module

Module können zum Einen als *Mixins*[1] fungieren, die von mehreren Klassen benötigte Funktionalität bereitstellen, wie im folgenden Beispiel zu sehen ist.

[1] Ein *Mixin* ist eine Sammlung von Code (in diesem Beispiel Methoden), die von einer Klasse eingebunden und anschließend verwendet werden kann.

```ruby
module Communication
  def say(words)
    puts words
  end
end

class Person
  include Communication
end

person = Person.new
person.say("This is just an example") # Ausgabe: "This is just an example"
```

Listing 2.8: Module als Mixins in Ruby.

Zum Anderen können Module ganze Klassen oder weitere Module beinhalten und so als Namespaces dienen. Mit *::* kann dann auf Mitglieder eines Moduls zugegriffen werden:

```ruby
module Toolbox
  class Screwdriver
    def screw
      puts "Screwing the screw..."
    end
  end
end

screwdriver = Toolbox::Screwdriver.new
screwdriver.screw # Ausgabe: "Screwing the screw..."
```

Listing 2.9: Module als Namespaces in Ruby.

2.1.4 Arrays und Hashes

Zwei sehr häufig verwendete Datenstrukturen in Ruby sind *Arrays* und *Hashes*. Arrays werden mit eckigen Klammern initialisiert und können Elemente unterschiedlichen Typs enthalten. Der Zugriff auf einzelne Elemente erfolgt ebenfalls mittels eckiger Klammern. Listing 2.10 zeigt ein Beispiel.

```ruby
mixed_array = [42, "a string", 10.5]
puts mixed_array[1] # Ausgabe: "a string"
```

Listing 2.10: Arrays in Ruby.

Hashes unterscheiden sich von Arrays dadurch, dass statt Integers beliebige Objekte als Index für den Zugriff auf Elemente verwendet werden können. Außerdem haben Elemente in einem Hash keine garantierte Reihenfolge. Hashes in Ruby sind also das Äquivalent zu HashMaps in Java. Listing 2.11 zeigt die Verwendung von Hashes.

```ruby
translations = {
  "bear" => "Bär",
  :chicken => "Huhn"
}

puts translations["bear"] # Ausgabe: "Bär"
puts translations[:chicken] # Ausgabe: "Huhn"
```

Listing 2.11: Hashes in Ruby.

Bei der Zeichenkette `:chicken` aus dem Beispiel handelt es sich um ein sog. *Symbol*. Symbols unterscheiden sich prinzipiell nicht von Strings, allerdings werden Symbols im Gegensatz zu Strings zur Laufzeit jeweils nur einmal instanziiert und im Speicher gehalten. Listing 2.12 verdeutlicht dies.

```ruby
# Gleiche Strings, aber unterschiedliche Objekte
"test".object_id #=> 70285944608060
"test".object_id #=> 70285944554520

# Gleiche Symbols und das gleiche Objekt
:test.object_id #=> 350108
:test.object_id #=> 350108
```

Listing 2.12: Symbols in Ruby.

Symbols eignen sich somit besonders für den Einsatz als Indizes in Hashes (oder wo immer ein String als identifzierendes Objekt verwendet wird), weil nicht bei jeder Verwendung ein neues Objekt erzeugt und somit mehr Speicher alloziert werden muss.

2.1.5 Blöcke

Blöcke in Ruby enthalten Anweisungen, die zu einem späteren Zeitpunkt ausgeführt werden können. Blöcke werden meist bei einem Methodenaufruf übergeben und dann von der Methode bei Bedarf aufgerufen. Dieser Aufruf erfolgt durch das Schlüsselwort `yield`. Blöcke können durch die Verwendung von geschweiften Klammern oder mit den Schlüsselworten `do` und `end` erstellt werden. Sie können auch Argumente entgegennehmen. Listing 2.13 zeigt ein Beispiel.

```ruby
def hello_block
  yield("Hello")
end

hello_block { |an_argument| puts an_argument } # Ausgabe: "Hello"
hello_block do |an_argument|
  puts "Another block"
end # Ausgabe: "Another block"
```

Listing 2.13: Blöcke in Ruby.

2.2 Einführung in Ruby on Rails

Ruby on Rails (im Folgenden auch „Rails“ genannt) ist ein weit verbreitetes Framework für Webanwendungen basierend auf der Programmiersprache Ruby. Es wurde im Jahre 2003 von dem Dänen David Heinemeier Hansson entwickelt und basiert auf den folgenden zwei Prinzipien:

Don’t repeat yourself (DRY)

Die wiederholte Implementierung von bestehender Funktionalität ist zu vermeiden, da duplizierter Code den Wartungsaufwand wesentlich erhöht. Stattdessen soll auf existierenden Code zurückgegriffen werden [vgl. SH12, S. 12].

Konvention vor Konfiguration

Statt für jedes Detail der Anwendung eine Konfigurationsdatei zu erstellen, sollten gewisse Konventionen eingehalten werden. Dies findet sich in Rails beispielsweise in der Benennung der Klassen und Methoden wieder, je nachdem in welchem Teil der Anwendung sie sich befinden und welche Aufgabe sie übernehmen [vgl. SH12, S. 13].

Diese beiden Prinzipien sind mitunter Gründe für das hohe Tempo, in dem sich Rails-Anwendungen entwickeln lassen. Dieses Kapitel soll einen Eindruck davon vermitteln, wie Rails-Anwendungen strukturiert sind und wieso sich ihr Einsatz lohnt.

2.2.1 Model View Controller

Objekte innerhalb einer Rails-Anwendung werden gemäß des MVC-Entwurfsmusters („Model View Controller“) in drei Kategorien unterschieden: Model (Modell), View (Darstellung) und Controller (Steuerung). Jede dieser Kategorien hat einen genau definierten Aufgabenbereich, der im Folgenden jeweils genauer erläutert wird.

Models

Models beinhalten die Geschäftslogik[2] der Anwendung und sind verantwortlich für das Halten der Daten, die zur weiteren Verarbeitung benötigt, oder von Views dargestellt werden. Sie können von „ActiveRecord“, der in Rails enthaltenen ORM-Lösung[3], auf die Datenbank abgebildet und persistiert werden.

Views

Views enthalten die Darstellungslogik der Anwendung. Die darzustellenden Daten beziehen sie dabei aus den Models, die ihnen vom Controller zur Verfügung gestellt wurden. Die Ausgabe kann in verschiedenen Formaten erfolgen, geschieht aber meist in HTML[4].

Controllers

Controller sind die Bindeglieder zwischen Models und Views. Sie sollten keine Geschäftslogik enthalten, sondern lediglich Abläufe steuern: Die benötigen Models werden akquiriert und den Views zur Verfügung gestellt [vgl. SH12, S. 16]. Dabei dürfen Abhängigkeiten (dargestellt in Richtung der Pfeile) lediglich wie in Abbildung 2.1 gezeigt bestehen.

[2] *Geschäftslogik* beschreibt die Logik einer Anwendung, die die in der Realität existierenden Regeln der Domäne implementiert [siehe Fow02, S. 20].

[3] *Object-Relational Mapping* ist eine Technik, bei der Objektstrukturen auf Einträge in Tabellen von relationalen Datenbanken abgebildet werden.

[4] Die *Hypertext Markup Language* ist eine Sprache zur Strukturierung von Dokumenten. HTML-Dokumente werden von Browsern dargestellt.

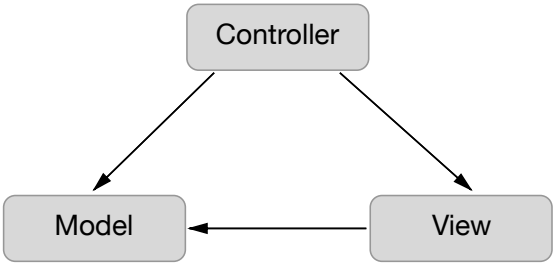

Abbildung 2.1: Erlaubte Assoziationen im MVC-Muster.

Offensichtlich muss der Controller Zugriff auf Models und Views haben, um seine Aufgabe erfüllen zu können. Zudem muss die View auf die ihr zur Verfügung gestellten Models zugreifen können, um die darzustellenden Daten zu erhalten. Weitere Assoziationen dürfen nur in indirekter Form existieren, wie zum Beispiel durch das *Observer-Pattern* [siehe GHJV94, S. 293].

2.2.2 REST und Routes

REST (*Representational state transfer*) ist ein Architekturstil, der spezifiziert, wie auf Ressourcen von Webanwendungen zugegriffen werden kann. Ressourcen sind Datensätze, die anhand einer URI[5] eindeutig identifiziert werden und auf denen Aktionen ausgeführt werden können. Bei HTTP[6]-basierten REST-konformen Schnittstellen hat sich folgende Semantik für den Zugriff auf Ressourcen und deren Manipulation durchgesetzt.

HTTP-Anfragemethode	Beschreibung
GET	Anforderung der gegebenen Ressource vom Server.
POST	Erstellung einer neuen Ressource mit den gegebenen Attributen.
PUT	Anpassung einer bestehenden Ressource mit den geg. Attributen.
DELETE	Löschen einer bestehenden Ressource.

Tabelle 2.1: Die gängigsten HTTP-Methoden und ihre Bedeutung in REST-Schnittstellen.

Routes (Routen) sind der Einstiegspunkt jeder Rails-Anwendung und unterstützen die Entwicklung REST-konformer Schnittstellen, indem sie einer HTTP-Anfrage mittels der gegebenen URI und der HTTP-Anfragemethode eine entsprechende Methode in einem Controller zuordnen, die dann aufgerufen wird, um die Antwort zu generieren. Dem Prinzip „Konvention vor

[5] Ein *Uniform Resource Identifier* ist eine Zeichenfolge anhand derer sich Ressourcen eindeutig identifizieren lassen.
[6] *Hypertext Transfer Protocol.* Mehr Informationen: http://tools.ietf.org/html/rfc7231

Konfiguration“ folgend, muss aber nicht manuell jede Route auf eine entsprechende Methode abgebildet werden. Stattdessen kann Rails den richtigen Controller und die aufzurufende Methode aus der HTTP-Anfrage ableiten. Ein Beispiel zur Verwendung von Routes wird im Abschnitt 2.2.4 gezeigt.

2.2.3 Wichtige Module

Der Kern von Rails besteht aus einer Reihe von Modulen, von denen die Module, die für ein grundlegendes Verständnis notwendig sind, hier kurz erläutert werden.

ActionDispatch

ActionDispatch bearbeitet HTTP-Anfragen und ist für das in Abschnitt 2.2.2 erläuterte Routing verantwortlich. Ein Beispiel für die Konfiguration von Routes ist im Abschnitt 2.2.4 zu finden.

ActionController

Das Modul ActionController und dessen Basisklasse `ActionController::Base` bieten grundlegende Funktionen, die von Controllern benötigt werden. Jeder Controller muss von dieser Basisklasse ableiten.

ActionView

ActionView ist verantwortlich für das Rendering der Views, die in verschiedenen Formaten vorhanden sein können und in der Regel HTML generieren.

ActiveRecord

ActiveRecord übernimmt das Object-Relational-Mapping von Ruby on Rails. Models können von der dazugehörigen Basisklasse `ActiveRecord::Base` ableiten, um auf eine Reihe von Funktionen zurückgreifen zu können. Gemäß dem Prinzip „Konvention for Konfiguration“, sucht ActiveRecord dann nach einer Tabelle in der Datenbank, benannt nach dem Plural des Namens des Models. Aus den Feldern der Tabelle werden dann die Attribute des Models abgeleitet und stehen als Methoden zur Verfügung. Auch Assoziationen (wie „1:1“, „1:n“ oder „n:m“) werden unterstützt.

Zu den Funktionen von ActiveRecord gehören auch die sogenannten *Migrations*, die eine schrittweise Erstellung und Erweiterung des Datenbankschemas erlauben. Unterstützt wird eine Reihe von Datenbanksystemen, unter anderem Oracle, MySQL und PostgreSQL.

2.2.4 Ein Beispiel

Angenommen die HAW Hamburg würde auf der Seite *http://www.haw-hamburg.de/events* eine Liste von aktuellen Veranstaltungen darstellen wollen. Einzelne Veranstaltungen sollen angezeigt, erstellt, verändert und gelöscht werden können.

Routes

Um Routes für die jeweiligen Aktionen zu definieren, muss die entsprechende Konfigurationsdatei angepasst werden. Alle Konfigurationsdateien einer Rails-Anwendung befinden sich im Ordner `config/`. In der Datei `config/routes.rb`, werden alle Routes der Anwendung definiert. Diese Datei sollte für dieses Beispiel wie in Listing 2.14 gezeigt aussehen.

```
1 Rails.application.routes.draw do
2   resources :events
3 end
```

Listing 2.14: Inhalt der Datei `config/routes.rb`

In Zeile 2 wird Rails mitgeteilt, REST-konforme Routes für Veranstaltungen zu registrieren. Dadurch entstehen die in Tabelle 2.2 gezeigten Routes. Der Ausdruck `:id` ist hier ein Platzhalter für das identifzierende Attribut der Veranstaltung (meist der Primärschlüssel der Ressource in der Datenbank).

Anfragemethode	Pfad	Controller	Methode	Aufgabe
GET	/events	EventsController	index	Anzeige einer Liste aller Veranstaltungen
GET	/events/new	EventsController	new	Anzeige eines Formulars zur Erstellung neuer Veranstaltungen
POST	/events	EventsController	create	Erstellen einer Veranstaltung
GET	/events/:id	EventsController	show	Anzeige einer bestimmten Veranstaltung

GET	/events/:id/edit	EventsController	edit	Anzeige eines Formulars zur Bearbeitung einer bestehenden Veranstaltung
PUT	/events/:id	EventsController	update	Verändern einer bestehenden Veranstaltung
DELETE	/events/:id	EventsController	destroy	Löschen einer bestehenden Veranstaltung

Tabelle 2.2: Generierte Routes durch die Anweisung `resources :events`.

Controller

Nun muss der `EventsController` implementiert werden, der verwendet werden soll, wenn eine der oben genannten Routes aufgerufen wird. Exemplarisch wird hier nur die Implementierung der `show`-Methode gezeigt. Controller, Models und Views befinden sich im Ordner `app/`, und dort wiederum jeweils in ihrem eigenen Ordner. Der `EventsController` muss sich also im Ordner `app/controllers/` befinden, damit Rails ihn verwendet. Eine weitere Konvention ist die Benennung des Dateinamens: Aus „camel case“ wird „snake case“. Die Datei muss in diesem Beispiel also `events_controller.rb` heißen.

```
1 class EventsController < ActionController::Base
2   def show
3     @event = Event.find(params[:id])
4   end
5 end
```

Listing 2.15: Inhalt der Datei `app/controllers/events_controller.rb`

In Listing 2.15 wird in Zeile 3 nach einer Veranstaltung mit der gegebenen ID gesucht. Diese wird dann von `find()` zurückgegeben und der Instanzvariable `@event` zugewiesen, damit sie später der View zur Verfügung steht.

Model

Die Implementierung des Models `Event` ist recht kompakt:

```ruby
class Event < ActiveRecord::Base
end
```

Listing 2.16: Inhalt der Datei `app/models/event.rb`

Zwar wurden in diesem Beispiel Dinge wie Fehlerbehandlungen und Validierungen der Kompaktheit halber außer Acht gelassen, aber tatsächlich ist der in Listing 2.16 gezeigte Code ausreichend, um das Basisverhalten eines Models herzustellen: Die Methode `find()` steht jetzt zur Verfügung, Attribute können gesetzt und das Model persistiert werden, und vieles mehr. Zusätzlich dazu muss allerdings eine Migration existieren, die das Tabellenschema für das Model `Event` anlegt, damit Rails aus diesem die Attribute des Models ableiten kann. Listing 2.17 zeigt eine solche Migration.

```ruby
class CreateEvents < ActiveRecord::Migration
  def change
    create_table :events do |table|
      table.string :title
      table.string :description
      table.timestamp :date
    end
  end
end
```

Listing 2.17: Inhalt der Datei `db/migrate/20150414162841_create_events.rb`

Alle vorhandenen Migrationen können mit dem Befehl `rake db:migrate` ausgeführt werden. Mittels `rake db:rollback` kann die jeweils letzte Migration wieder rückgänig gemacht werden. Das Ausführen der oberen Migration erzeugt in der Datenbank eine Tabelle mit dem Namen `events` und den Feldern `id` vom Typ Integer (wird implizit erzeugt), `title` vom Typ String, `description` vom Typ String und `date` vom Typ Timestamp. Diese Felder existieren nun auch als Attribute auf dem Model `Event`.

View

Views befinden sich im Ordner `app/views/` und werden nach ihrer zugehörigen Methode benannt: Wird auf dem `EventsController` die Methode `show` aufgerufen, sucht Rails nach der Datei `app/views/events/show.html.erb`. Die Dateiendung `.erb` signalisiert Rails, dass in der View die *eRuby*-Templatesprache verwendet wurde. Sie erlaubt die Einbettung von Ruby-

Code in Textdateien, oder in diesem Fall HTML.
Alle im Controller gesetzten Instanzvariablen stehen in der entsprechenden View ebenfalls zur Verfügung. Die View der `show`-Methode könnte zum Beispiel aussehen, wie in Listing 2.18 gezeigt.

```
<h1>Veranstaltungsdetails</h1>

<div>
  <p>Name: <%= @event.title %></p>
  <p>Datum: <%= @event.date %></p>
  <p>Beschreibung: <%= @event.description %></p>
</div>
```

Listing 2.18: Inhalt der Datei `app/views/events/show.html.erb`

Diese View wird dann *gerendert*, das heißt der Ruby-Code wird ausgeführt und durch dessen Ausgabe ersetzt, und mit dem Rest der Seite zusammengeführt. Die grobe Struktur der Seite wird in der Datei `app/views/layouts/application.html.erb` angegeben. Dort wird die View an der Position des Schlüsselwortes `yield` eingebunden:

```
<html>
  <head>
    <title>Veranstaltungen der HAW Hamburg</title>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Listing 2.19: Inhalt der Datei `app/views/layouts/application.html.erb`

Konventionen, wie die Benennung der View nach dem Methodennamen und das damit ermöglichte automatische Rendering, müssen natürlich nicht befolgt werden. Am Ende der Controller-Methode kann Rails auch dazu aufgefordert werden eine völlig andere Datei zu rendern oder JSON[7] auszugeben, was eine oft verwendete Funktion für die Implementierung von APIs[8] ist. Listing 2.20 zeigt, wie dies umgesetzt wird.

[7] *JavaScript Object Notation* ist ein schemaloses Datenformat zum Austausch von Daten zwischen Anwendungen
[8] Ein *Application Programming Interface* ist eine Schnittstelle, die anderen Anwendungen zur Verfügung gestellt wird, damit diese auf die Funktionalität der Anwendung zugreifen können, die die Schnittstelle bereitstellt.

```ruby
class EventsController < ActionController::Base
  def show
    event = Event.find(params[:id])
    render json: event
  end
end
```

Listing 2.20: Ausgabe der JSON-Repräsentation eines `Event`-Objekts

Die Vielzahl von Konventionen und Automatismen erlaubt zwar eine schnelle Entwicklung, sie kann allerdings auch problematisch sein: Bei Wissenslücken über die Verhaltensweisen von Rails oder die neuesten Änderungen am Framework, entstehen schnell unerwünschte Nebeneffekte oder unerwartete Ergebnisse. Auf dem neuesten Stand zu bleiben ist für die Entwicklung mit Ruby on Rails also unerlässlich.

2.3 Einführung in Domain-Driven Design

Der Begriff *Domain-Driven Design* wurde geprägt durch Eric Evans und die Veröffentlichung seines gleichnamigen Buches im Jahre 2003. In diesem beschreibt er Wege eine Problemdomäne so zu modellieren, dass das resultierende Modell bis zu dem Maße die Realität abbildet, in dem es der langfristigen Entwicklung der Software am dienlichsten ist [vgl. Eva03, S. 3]. Dies wird erreicht, indem möglichst viel Wissen über die Domäne akkumuliert wird, um anschließend die für die Funktionalität der Software relevanten Informationen zu extrahieren und zu präzisieren. Daraus lässt sich dann ein Modell entwickeln, das kontinuierlich in Folge von neuen Erkenntnissen über die Domäne oder hinzugekommenen Anforderungen an die Software überarbeitet wird. Dieses Modell nennt sich Domänenmodell. Um dieses Domänenmodell möglichst unverzerrt im Code abzubilden, zeigt Evans Möglichkeiten in Form von Entwurfsmustern und Konzepten auf, die im weiteren Verlauf dieses Kapitels vorgestellt werden.

2.3.1 Ubiquitäre Sprache

In jedem Softwareprojekt müssen sich Entwickler zunächst mit der Problemdomäne vertraut machen, bevor sie das weitere Vorgehen für die Entwicklung planen können. Dies erreichen sie mithilfe von Menschen, die sich mit der Domäne auskennen und bestenfalls auch in dieser arbeiten. Evans nennt diese Leute *Domänenexperten* (engl. *Domain Experts*) [vgl. Eva03, S. 24]. Ein wichtiger Prozess auf dem Weg zur Gestaltung des Domänenmodells ist die Entwicklung einer gemeinsamen Sprache, der sogenannten *ubiquitären Sprache* (engl. *Ubiquitous Language*).

Diese Sprache entsteht durch den Austausch zwischen Entwicklern und Domänenexperten und besteht zu Beginn häufig aus Begriffen, die aus der Problemdomäne stammen [vgl. Ver13, S. 21]. Die ubiquitäre Sprache soll aber nicht nur Entwickler beim Verständnis der Domäne unterstützen. Die Domänenexperten sollen ebenfalls mehr über Prozesse und Abläufe in ihrer eigenen Domäne lernen, da auch ihnen unter Umständen nicht alle Details von diesen bekannt sind [vgl. Ver13, S. 27]. So werden die Begriffe der ubiquitären Sprache durch den Einfluss der Entwickler mit der Zeit schärfer und präziser [vgl. Eva03, S. 34]. Die Begriffe der ubiquitären Sprache fließen in das Domänenmodell ein und finden sich dann auch im Code, zum Beispiel in Klassen- und Methodennamen, wieder.

2.3.2 Das Domänenmodell

Das Domänenmodell soll eine abstrakte und organisierte Repräsentation des Wissens der Domänenexperten sein. Evans drückt es folgendermaßen aus:

„*A domain model is not a particular diagram; it is the idea that the diagram is intended to convey. It is not just the knowledge in a domain expert's head; it is a rigorously organized and selective abstraction of that knowledge.*“ [Eva03, S. 3]

Das Modell muss also nicht besonders akkurat die Realität abbilden, sondern hauptsächlich eine Vorstellung oder Idee von den Abläufen und Zusammenhängen in der Domäne transportieren. Dabei kann UML[9] zum Einsatz kommen, muss aber nicht. Mit wachsender Größe eines UML-Diagramms steigt auch der Aufwand, den es braucht, um es zu verstehen. In diesem Fall macht es Sinn zum Zwecke der Vereinfachung nicht-standardisierte Darstellungsformen zu wählen, um die Verständlichkeit zu verbessern [vgl. Eva03, S. 36].

2.3.3 Model-Driven Design

Das aus der Erkundung der Domäne entstandene Modell eignet sich nicht immer für eine direkte Implementierung, da zwar die Abläufe und Zusammenhänge der Domäne berücksichtigt wurden, jedoch nicht, welche Rolle sie in der Software spielen. Es ist aber von großer Bedeutung, dass der Code und das dazugehörige Domänenmodell eng zusammenhängen, da das Modell ansonsten seinen Wert verliert [vgl. Eva03, S. 48]. Beim *Model-Driven Design* soll ein Modell gefunden werden, das zum einen die Domäne korrekt darstellt und sich zum anderen auch im Code gut abbilden lässt [vgl. Eva03, S. 49]. Dabei bringt der Code das Modell zum Ausdruck,

[9] Die *Unified Modeling Language* ist eine Modellierungssprache, die sich grafischen Elementen bedient, um Beziehungen und Abläufe in Bezug auf Software zu visualisieren.

sodass eine Änderung des Codes auch eine Änderung des Modells bedeuten kann [vgl. Eva03, S. 49]. *Model-Driven Design* ist ein zentrales Konzept von Domain-Driven Design.

2.3.4 Bounded Contexts

Bei größeren Softwareprojekten können durchaus mehrere Domänenmodelle existieren. Die Bedeutung oder Rolle der modellierten Bestandteile der Domäne kann sich je nachdem, in welchem Kontext sie sich befinden, ändern. Zum Beispiel kann ein Nutzer einer Video-Streaming-Plattform in einem Kontext als Konsument von Filmen gesehen werden, die er anschaut, auf eine Merkliste setzt oder bewertet. Übernimmt die Plattform auch die Abrechnung der Beiträge, hat der Nutzer in diesem Kontext eine ganz andere Rolle. Interessant ist in diesem Fall nicht mehr, für welche Filme sich der Nutzer interessiert, sondern beispielsweise welche Zahlungsarten er hinterlegt hat, ob er monatlich oder jährlich zahlt und auf welche Adresse die Rechnung ausgestellt wird.

Ein *Bounded Context* ist ein fest abgesteckter Rahmen, innerhalb dessen ein Domänenmodell gültig ist [vgl. Eva03, S. 336]. In dem oben genannten Beispiel existieren zwei *Bounded Contexts*, deren Namen zwar prinzipiell frei wählbar, aber unverzichtbar sind, um innerhalb des Teams über sie reden zu können [vgl. Eva03, S. 351]. Hier könnten sie zum Beispiel *Verbraucherkontext* und *Abrechnungskontext* heißen. Die ubiquitäre Sprache kann sich zwischen verschiedenen Bounded Contexts unterscheiden, da Begriffe durch den Wechsel des Kontexts ihre Bedeutung verändern können [vgl. Eva03, S. 336]. Ein *Bounded Context* muss jedoch nicht zwangsläufig ausschließlich ein Domänenmodell beinhalten, sondern kann darüber hinaus auch ein externes System repräsentieren [vgl. Ver13, S. 66].

2.3.5 Context Maps

Context Maps sollen einen Überblick über die existierenden Bounded Contexts und deren Beziehungen zueinander bieten, um dem Team eine Grundlage für die gemeinsame Kommunikation zu bieten. *Context Maps* können auch genutzt werden, um Abhängigkeiten zu erkennen, die nicht unbedingt nur durch den Code bedingt sein müssen, sondern auch unternehmensstrukturelle Ursachen haben können (zum Beispiel Abhängigkeit von einem Drittanbieter). So helfen sie auch bei der Analyse von Risiken.
Evans definiert in seinem Buch verschiedene Arten von Beziehungen und Umsetzungsmöglichkeiten für *Context Maps*, auf die hier nicht näher eingegangen wird [siehe Eva03, S. 352 - 396].

2.3.6 Schichtenarchitektur

Die Aufteilung von Code in verschiedene Schichten ist eine bewährte Technik in der Softwareentwicklung. Höherliegende Schichten greifen dabei auf Funktionalität von den darunterliegenden Schichten zu, aber nicht umgekehrt[10]. In einer Drei-Schichten-Architektur wird üblicherweise zwischen den folgenden Schichten unterschieden (von oben nach unten):

Schicht	Beschreibung
Präsentationsschicht	Übernimmt die Interaktion mit dem Nutzer der Software, also die Visualisierung der Daten und die Verarbeitung der Eingaben.
Domänenschicht	Enthält die Geschäftslogik der Anwendung. Abhängig von den empfangenen Befehlen aus der Präsentationsschicht werden Berechnungen basierend auf Eingaben und Daten aus der Datenhaltungsschicht vorgenommen und es wird gesteuert, welche Aktionen in dieser vollzogen werden.
Datenhaltungsschicht	Verantwortlich für die Kommunikation mit externen Systemen, die Daten bereitstellen oder andere Aufgaben ausführen. Häufig handelt es sich dabei um Datenbanken oder Message Broker, wie RabbitMQ[11].

Tabelle 2.3: Bestandteile einer Drei-Schichten-Architektur [vgl. Fow02, S. 19 ff.].

Die Verwendung einer Schichtenarchitektur bietet diverse Vorteile [aus Fow02, S. 17]:

- Die Schichten können einzeln und unabhängig von anderen Schichten betrachtet und verstanden werden.
- Einzelne Schichten können leicht ausgetauscht werden, solange die Funktionalität erhalten und die Schnittstelle gleich bleibt.
- Die Abhängigkeiten zwischen verschiedenen Schichten werden minimiert: Verändert sich eine Schicht, muss nur die jeweils direkt darüberliegende Schicht angepasst werden.
- Eine Schicht kann von mehreren darüberliegenden Schichten oder Diensten verwendet werden, Schichten sind also wiederverwendbar.

Laut Evans ist die Isolation der Geschäftslogik vom Rest der Anwendung durch eine Domänenschicht essenziell für das Gelingen von Model-Driven Design [vgl. Eva03, S. 75]. Um dies zu

[10] Indirekte Abhängigkeiten (wie zum Beispiel durch Callbacks) dürfen jedoch vorhanden sein [siehe Eva03, S. 73].
[11] Siehe https://www.rabbitmq.com/.

erreichen, führt er sogar noch eine weitere Schicht ein: Die Anwendungsschicht (engl. *Application Layer*). Sie liegt zwischen Präsentationsschicht und Domänenschicht und soll Abläufe und Aktionen koordinieren. Dabei delegiert sie jedwede Geschäftslogik an die Domänenschicht, enthält selbst also keine Geschäftslogik. Auch Fowler beschreibt diesen Ansatz, nennt die zusätzliche Schicht allerdings *Service Layer* [vgl. Fow03]. Beide Autoren sind sich einig, dass diese Schicht möglichst dünn gehalten werden muss [vgl. Eva03, S. 70] und im einfachsten Fall als Fassade oder Schnittstelle für die Domänenschicht fungiert [vgl. Fow02, S. 31 f.].

2.3.7 Bestandteile der Domänenschicht

Die Domänenschicht soll das Domänenmodell in Form von Code abbilden. Evans beschreibt dafür verschiedene Arten von Objekten, aus denen die Domänenschicht bestehen soll: *Entities*, *Value Objects* und *Services*. Mit *Modules* werden diese Objekte nach ihren Verantwortlichkeiten und Aufgaben gruppiert. Jedes der genannten Konzepte wird im Folgenden näher erläutert.

Entities

Bestimmte Objekte können nicht anhand ihrer Attribute identifiziert werden, sondern haben eine eigene Identität, die über die Zeit erhalten bleibt, während die Attribute sich verändern können. Ein oft gewähltes Beispiel ist das Konzept einer Person: Eine Person hat eine eigenständige Identität unabhängig von ihren Attributen, wie zum Beispiel der Haarfarbe, dem Namen, Geburtsdatum oder Geschlecht. Zwei Personen, die die gleichen genannten Attribute aufweisen, sind noch lange nicht dieselbe Person.

Ein weiteres Beispiel ist eine Banktransaktion: Zwei Transaktionen mit denselben Attributen (wie etwa Betrag, Kontonummer des Empfängers, Kontonummer des Senders und Zeitpunkt) sind nicht dieselbe Transaktion. Der Empfänger würde sonst eventuell um Geld betrogen.

Objekte, die eine eigene Identität und einen Lebenszyklus haben, nennt man **Entities**.

Da Entities nicht anhand ihrer Attribute von anderen Entities unterschieden werden können, benötigen sie ein zusätzliches eindeutig *identifizierendes* Attribut.

Value Objects

Im Gegensatz zu Entities existieren auch Objekte, die nur anhand ihrer Attribute unterschieden werden und keine eigene Identität benötigen. Ein Beispiel dafür ist der Name einer Person:

Haben zwei verschiedene Personen denselben Namen, sind die Objekte, die die Namen repräsentieren vollkommen austauschbar. Sie können also von verschiedenen Personen geteilt werden [vgl. Eva03, S. 100]. Die Unterscheidbarkeit zweier gleich geschriebener Namen hätte keinen Vorteil, würde aber einen höheren Verwaltungsaufwand bedeuten [vgl. Eva03, S. 98].

Objekte, die ausschließlich anhand ihrer Attribute unterschieden werden und keine eigene Identität benötigen, nennt man **Value Objects**.

Dass zwei Personen sich einen Namen teilen kann dann problematisch werden, wenn sich der Name von einer der beiden Personen ändert. Wird deswegen nun das Objekt, das den Namen repräsentiert, verändert, haben anschließend beide Personen den neuen Namen. Um solche Fehler zu verhindern, sollten Value Objects unveränderlich (engl. *immutable*) sein [vgl. Eva03, S. 100]. Ändert sich der Name einer Person, erhält sie einfach ein neues Value Object für den neuen Namen.
Unter bestimmten Umständen kann die Unveränderlichkeit allerdings zum Hindernis werden. Evans nennt dafür unter anderem folgende Beispiele [vgl. Eva03, S. 101]:

- Wenn der vom Value Object repräsentierte Wert sich oft verändert.
- Wenn die Erstellung beziehungsweise Löschung des Value Objects sehr aufwendig ist.
- Wenn die Ersetzung (im Gegensatz zu Veränderung) des Value Objects die Interaktion zwischen den beteiligten Objekten stört.

In diesen Fällen kann ein Value Object auch veränderlich sein, muss dann allerdings vor dem Zugriff von außen geschützt werden, da sonst Manipulationen daran vorgenommen werden können, die in einem ungültigen Zustand resultieren.

Value Objects können weitere Value Objects referenzieren, oder sogar auf Entities verweisen [vgl. Eva03, S. 98 f.]. Umgekehrt können und sollen auch Entities auf Value Objects verweisen.

Aggregates

Bestehen Entities aus mehreren Value Objects und/oder weiteren Entities, kann es schwierig werden die Konsistenz der Daten zu wahren. Angenommen es gäbe in einem Onlineshop eine Entity namens Artikel, die aus dem Value Object Preis und einer Liste von Value Objects namens Variation (zum Beispiel erhältliche Farben des Artikels) besteht. Jeder Artikel darf maximal 10 Variationen besitzen. Wird nun ein Artikel, der bereits 9 Variationen besitzt, von 2

Mitarbeitern des Onlineshops gleichzeitig bearbeitet und beide möchten eine neue Variation hinzufügen, sieht es für die Anwendung zunächst so aus, als wäre die Invariante (maximal 10 Variationen) erfüllt, da sie nichts von den Änderungen des jeweils anderen wissen. Persistieren aber beide ihre Änderungen in der Datenbank, besitzt der Artikel 11 Variationen und die Invariante ist nicht mehr erfüllt.

Aggregates markieren den Rahmen, innerhalb dessen Invarianten zu jedem Zeitpunkt erfüllt sein müssen [vgl. Eva03, S. 135]. Es werden nie einzelne Teile eines Aggregats aus der Datenbank gelesen oder in ihr persistiert, sondern immer nur das gesamte Aggregat. Des Weiteren darf je Transaktion nur ein Aggregat persistiert werden, sodass die Konsistenz der Abhängigkeiten des Aggregats mit *eventual consistency*[12] sichergestellt werden muss [vgl. Ver13, S. 287]. Dies ist allerdings eher als Richtlinie und nicht als unumstößliches Gesetz zu sehen: Durch *eventual consistency* können Probleme entstehen, die die User Experience erheblich zu stören vermögen, da Nutzer nicht immer sofort eine konsistente Sicht des Systems erhalten. In Fällen, in denen dies nicht akzeptabel ist, können auch mehrere Aggregate innerhalb einer Transaktion persistiert werden [vgl. MT15, S. 441].

Jedes Aggregat designiert eine Entity als sog. *Aggregatwurzel* (engl. *Aggregate root*), über die auf das Aggregat zugegriffen werden kann. Objekte außerhalb eines Aggregats dürfen keine Referenzen auf interne Objekte des Aggregats halten, aber durchaus über die Wurzel auf sie zugreifen [vgl. Eva03, S. 128 f.]. Aggregate können auch nur aus einer Wurzel bestehen.

Im oben genannten Beispiel eignet sich die Entity Artikel als Aggregatwurzel, während der Preis und die Liste der Varianten Teile des Aggregats sind, auf die nur durch die Aggregatwurzel zugegriffen werden darf.

Domain Events

Das Konzept von Domain Events wurde erst nach Veröffentlichung des Buchs von Eric Evans eingeführt [vgl. Ver13, S. 285 f.]. Domain Events repräsentieren Ereignisse, die in der Domäne geschehen und für die Domänenexperten bedeutsam sind [vgl. Ver13, S. 286]. Sie können verwendet werden, um nach dem Eintritt eines Ereignisses weitere Aktionen durchzuführen, oder um Kommunikation zwischen verschiedenen Bounded Contexts zu ermöglichen.

[12] *Eventual consistency* garantiert die Konsistenz der Daten nach dem Verstreichen eines genügend großen Zeitraums. Im Gegensatz zu *strong consistency*, welche garantiert, dass Daten zu jedem Zeitpunkt konsistent sind, sichert *eventual consistency* also nur zu, dass die Daten *irgendwann* konsistent sein werden.

In den späteren Abschnitten werden Domain Events verwendet, um *eventual consistency* zu implementieren. Domain Events haben einen Namen, der möglichst aus der ubiquitären Sprache stammen sollte, also zum Beispiel *user_created* für die erfolgte Erstellung eines Nutzers. Außerdem enthalten sie weitere Informationen zu dem Ereignis, wie zum Beispiel IDs von betroffenen Entities.

Services

Nicht alle Konzepte der Domäne lassen sich nur mit Entities und Value Objects sinnvoll abbilden. Manche Operationen stehen für sich allein und gehören in keines der existierenden Objekte, da diese sonst eventuell schwerer verständlich und schlechter refaktorierbar würden [vgl. Eva03, S. 104].

Services sind alleinstehende, zustandslose Operationen, die diese Lücke füllen:
„*A service is an operation offered as an interface that stands alone in the model, without encapsulating state, as entities and value objects do.*“ [Eva03, S. 105]

Sie arbeiten mit anderen Objekten und können diese auch verändern, dürfen aber selbst keine Zustandsinformationen halten [vgl. Eva03, S. 105 f.].
Je nach ihrer Aufgabe können sich Services in der Anwendungs-, Domänen- oder Datenhaltungsschicht befinden [siehe Eva03, S. 106 f.].

Module

Module dienen in erster Linie der Strukturierung der Anwendung und um Komplexität zu verbergen, die nicht zum Verständnis der Funktionalität erforderlich ist [vgl. Eva03, S. 109]. Evans betont dabei, dass in Domain-Driven Design mit Modulen nicht einfach Code aufgespalten wird, sondern Konzepte, die so beim Betrachten der Beziehung verschiedener Module zum Vorschein kommen [vgl. Eva03, S. 109]. Namen von Modulen fließen in die ubiquitäre Sprache mit ein und sollen sich mit dem Domänenmodell weiterentwickeln, um die neuesten Erkenntnisse des Teams zu reflektieren [vgl. Eva03, S. 110 f.].

2.3.8 Entwurfsmuster für die Domänenschicht

Factories

Die Erstellung von Objekten oder ganzen Aggregaten kann Komplexität beinhalten, die Wissen über die interne Struktur des Objekts oder Aggregats voraussetzt. Übernähme der Verwender

des Objekts diese Aufgabe, würde das Prinzip der Datenkapselung[13] verletzt. Übernähme das Objekt selbst diese Aufgabe, hätte es mehr als eine definierte Aufgabe und würde so das *Single-Responsibility-Prinzip*[14] verletzen.

In diesem Fall bietet sich die Einführung eines separat Objekts an, dessen einzige Aufgabe die Erstellung eines komplexen Objekts oder Aggregats ist. Objekte, deren einzige Aufgabe die Erstellung anderer Objekte ist, werden **Factories** genannt. Die Methoden einer Factory akzeptieren Parameter, die für die Erstellung benötigt werden und geben das erstellte und gültige Objekt oder Aggregat zurück [vgl. Eva03, S. 138 f.].

Repositories

Wurde der Zustand einer Entity durch eine Reihe von Operationen verändert, muss sie anschließend meist persistiert werden. Handelt es sich bei der Entity um eine Aggregatwurzel, muss das gesamte Aggregat persistiert werden. Umgekehrt muss es auch möglich sein, persistierte Entities oder ganze Aggregate zu einem späteren Zeipunkt wiederherzustellen. Dies sind allerdings keine Aufgaben, die innerhalb der Domänenschicht gelöst werden sollten, da diese lediglich Geschäftslogik enthalten darf.

Repositories vermitteln zwischen der Domänenschicht und der Datenhaltungsschicht, indem sie die nötige Logik zur Persistierung und Wiederherstellung von Domänenobjekten implementieren. Vaughn Vernon unterscheidet in seinem Buch zwischen zwei Arten von Repositories:

Collection-Oriented Repositories

Repositories diesert Art haben ähnliche Schnittstellen, wie typische *Collection*-Klassen (zum Beispiel Listen, Sets etc.). Der Verwender des Repositories soll nicht merken, dass es sich um einen Persistenzmechanismus handelt, sondern den Eindruck erhalten, die Daten befänden sich bereits im Hauptspeicher [vgl. Ver13, S. 402 ff.]. Voraussetzung dafür ist allerdings ein Mechanismus, der Änderungen an Objekten erkennt und diese automatisch persistiert: Müsste der Verwender explizit eine Operation zur Speicherung aufrufen, ginge die Illusion verloren.

[13] *Datenkapselung* in der objektorientierten Programmierung bezeichnet das Verbergen von Informationen über die interne Struktur eines Objektes (zum Beispiel private Methoden oder Attribute).

[14] Das *Single-Responsibility-Prinzip* besagt, dass eine Klasse nur eine fest definierte Verantwortlichkeit oder Aufgabe und somit nur „einen Grund zur Veränderung“ haben sollte [siehe Mar09, S. 138 f.].

Persistence-Oriented Repositories

Bei diesem Ansatz müssen Domänenobjekte explizit vom Verwender des Repositories gespeichert werden. Änderungen an Objekten müssen also nicht von der Anwendung erkannt werden. Dies vereinfacht insbesondere die Persistierung von Aggregaten stark [vgl. Ver13, S. 418].

3 Analyse

3.1 Die Beispielanwendung

Der Einsatz von Domain-Driven Design und dessen Vorteile werden am Beispiel eines digitalen Flohmarkts für Kinderbedarf gezeigt. Dabei handelt es sich um ein Projekt der Firma *mindmatters GmbH & Co. KG*, das vom Verlag *Gruner + Jahr GmbH & Co. KG* in Auftrag gegeben wurde. Dieser Abschnitt soll einen Überblick über die Funktionalität der Anwendung bieten, um die folgenden Abschnitte leichter verständlich zu machen.

Das System besteht aus einer App für das mobile Betriebssystem iOS und einer auf Ruby on Rails basierenden REST-Schnittstelle, mit der die App kommuniziert. Die App fungiert dabei als *Front-End* und übernimmt hauptsächlich Darstellungsaufgaben und die Steuerung von Abläufen. Die Rails-Anwendung dient als *Back-End* und beherbergt den Hauptteil der Anwendung, unter anderem die Geschäftslogik und die Datenhaltung.

Diese Arbeit beschränkt sich auf die Betrachtung des Back-Ends, da sich hier die für Domain-Driven Design relevanten Teile der Anwendung befinden.

3.1.1 Funktionale Anforderungen

Das System soll Eltern eine Plattform bieten, um Gebrauchsgegenstände für Kinder (zum Beispiel Kleidung, Spielzeug oder Möbel) zu verkaufen oder von anderen Eltern zu kaufen. Dazu können diese zunächst Artikel einstellen, die sie verkaufen möchten. Andere Nutzer können dann für diese Artikel Anfragen verschicken, um über den Preis zu verhandeln. Der Verkäufer kann dem Interessenten anschließend ein Angebot machen, gegebenenfalls auch für weitere Artikel, für die der Interessent Anfragen an den Verkäufer geschickt hat. Der Interessent kann dann das Angebot annehmen oder ablehnen (und eventuell weiterverhandeln).

Dieser Abschnitt enthält nicht alle existierenden funktionalen Anforderungen an die An-

wendung, sondern lediglich die zum Kern der Anwendung gehörenden Funktionen, die zum Verständnis der folgenden Kapitel und Abschnitte nötig sind.

Registrierung
Der Nutzer muss sich mit einer Eingabemaske registrieren können. Dabei werden der Vor- und Nachname, die E-Mail-Adresse und das gewünschte Passwort des Nutzers benötigt.

Anmeldung
Der Nutzer muss sich mittels der zuvor angegebenen Kombination aus E-Mail-Adresse und Passwort anmelden können.

Abmeldung
Der Nutzer muss sich wieder abmelden können.

Bestätigung der E-Mail-Adresse
Der Nutzer muss seine E-Mail-Adresse verifizieren können.

Artikel anlegen
Der Nutzer muss neue Artikel mit diversen Attributen (unter anderem Hersteller, Größe, Farbe und Preis) anlegen können, die anschließend in der Anwendung zum Verkauf bereitstehen, sofern der Nutzer seine E-Mail-Adresse bereits bestätigt hat. Ansonsten ist der Artikel zwar vorhanden, steht aber erst nach Bestätigung der E-Mail-Adresse zum Verkauf.

Artikel suchen
Der Nutzer muss zum Verkauf stehende Artikel mittels einer Suchmaske auffinden können. Dabei kann er gewünschte Attribute der Artikel (zum Beispiel Hersteller, Größe, Farbe oder Preis) angeben.

Artikel favorisieren
Der Nutzer muss gefundene Artikel favorisieren können, um sie später erneut zu betrachten.

Liste favorisierter Artikel
Der Nutzer muss eine Liste aller von ihm favorisierten Artikel aufrufen können, um einzelne Artikel erneut zu betrachten.

Nachrichten austauschen

Zwei Nutzer müssen zum Zwecke des Verhandelns in der Lage sein, Nachrichten miteinander auszutauschen.

Anfrage verschicken

Der Nutzer muss für Artikel, die ihn/sie interessieren Anfragen verschicken können. Eine Anfrage ist dabei eine spezielle Form von Nachricht.

Angebot machen

Der Nutzer muss für Artikel, die zum Verkauf stehen und für die er/sie eine Anfrage erhalten hat Angebote machen können. Wurden mehrere Artikel desselben Verkäufers von einem Interessenten angefragt, muss der Verkäufer auswählen können, für welche Artikel er ein Angebot machen möchte und wie hoch der Preis dafür sein soll. Ein Angebot ist dabei eine spezielle Form von Nachricht.

Angebot zurückziehen

Der Nutzer muss bereits gemachte Angebote zurückziehen können. Das Angebot verfällt dadurch, der Verkäufer kann aber ein neues Angebot machen.

Angebot annehmen

Der Nutzer muss erhaltene Angebote für Artikel annehmen können. Der Artikel steht anschließend nicht mehr zum Verkauf und alle bereits verschickten Anfragen und Angebote verfallen.

Angebot ablehnen

Der Nutzer muss in der Lage sein erhaltene Angebote für Artikel abzulehnen. Das Angebot verfällt dadurch. Der Verkäufer kann aber ein neues Angebot machen.

Verknüpfung mit PayPal

Der Nutzer muss sein Konto für spätere eventuelle Zahlungen mit dem Zahlungsanbieter PayPal verknüpfen können. Die Verknüpfung ist optional.

Zahlung mit PayPal

Sofern Käufer und Verkäufer ihre Konten mit PayPal verknüpft haben, kann der Käufer die Zahlung mittels PayPal in der Anwendung vornehmen.

3.1.2 Bisheriges Modell

Wie bereits erwähnt, handelt es sich beim Back-End des Systems um eine Rails-Anwendung. In diesem Abschnitt wird das zugrundeliegende Modell von dieser beschrieben, wobei Elemente,

die nicht zur Kernfunktionalität gehören, zugunsten der Übersichtlichkeit nicht gezeigt werden. Das Modell findet sich in Rails-Anwendungen hauptsächlich in den Model-Klassen wieder, da diese auch den Großteil der Geschäftslogik beinhalten.

Abbildung 3.1 zeigt die Beziehungen zwischen den wichtigsten Model-Klassen der Anwendung als ER-Diagramm[1]. Klassen, die nicht zum Verständnis der nachfolgenden Kapitel notwendig sind, werden hier nicht dargestellt.

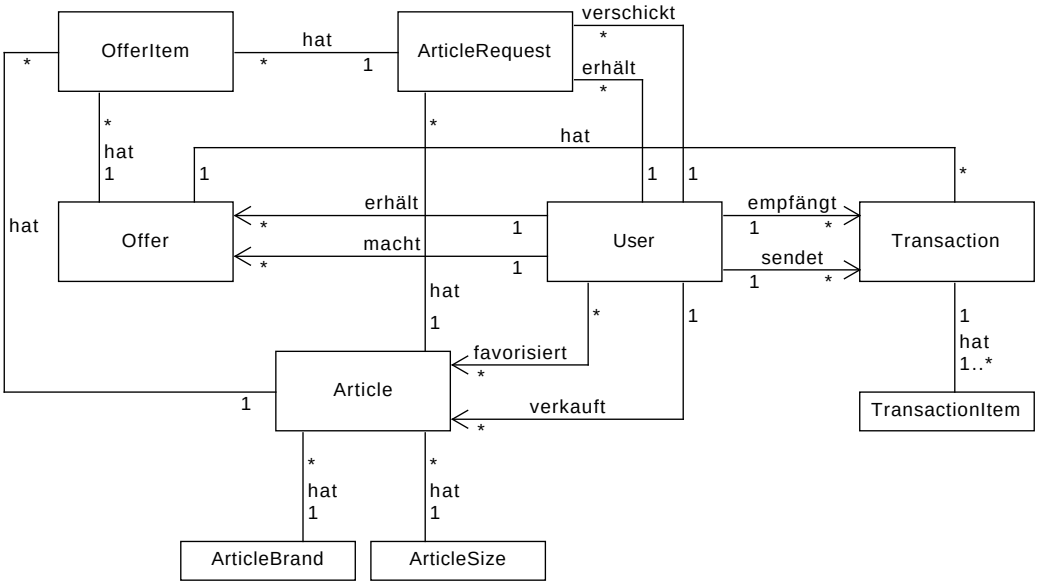

Abbildung 3.1: Vereinfachtes Modell der Anwendung als ER-Diagramm.

Offer

Angebote werden durch diese Klasse repräsentiert. Angebote sind einem (potenziellen) Käufer und einem Verkäufer (jeweils Objekte der Klasse `User`) zugeordnet. Die Informationen zu den im Angebot enthaltenen Artikeln und den jeweiligen verhandelten Preisen finden sich in Objekten der Klasse `OfferItem`. Im Laufe ihres Lebenszyklus nehmen die Angebote verschiedene Zustände an. Sie können zum Beispiel akzeptiert oder zurückgezogen werden, noch gänzlich unbeantwortet sein oder ungültig werden (wenn der Artikel nicht mehr zum Verkauf steht).

[1] *Entity-Relationship-Diagramme* visualisieren Entitäten und die vorhandenen Beziehungen zwischen diesen.

OfferItem

Objekte der Klasse `OfferItem` bilden die einzelnen Posten eines Angebots und werden eindeutig identifiziert durch den zugehörigen Artikel, die entsprechende Anfrage (`ArticleRequest`) und das Angebot. Ein `OfferItem` enthält auch die Information über den verhandelten Preis des Artikels.

Article

Artikel werden von Käufern favorisiert und von Verkäufern verkauft (jeweils `User`). Sie können über ein `OfferItem` Teil eines Angebots sein oder über ein `ArticleRequest` angefragt werden. Auch Artikel nehmen über die Zeit verschiedene Zustände an, zum Beispiel ob sie bereits verkauft wurden oder noch zum Verkauf stehen und ob sie nach dem Verkauf bezahlt wurden.

ArticleBrand

Jedem Artikel kann vom Nutzer ein Hersteller zugeordnet werden. Dieser kann entweder aus einer Liste von vorhandenen Herstellern ausgewählt, oder selbst angelegt werden.

ArticleSize

Jedem Artikel kann vom Nutzer eine Größe zugeordnet werden. Dieser kann aus einer Liste von vorhandenen Größen ausgewählt, aber nicht selbst angelegt werden.

ArticleRequest

Anfragen in Form von Objekten der Klasse `ArticleRequest` können von (potenziellen) Käufern verschickt und von Verkäufern empfangen werden. Sie sind einem Artikel zugeordnet und können von einem `OfferItem`, also Posten eines Angebots, referenziert werden. Eine Anfrage kann beantwortet, unbeantwortet, erfüllt oder ungültig sein (zum Beispiel wenn der Artikel in der zwischen Zeit schon an einen anderen Nutzer verkauft wurde).

Transaction

Transaktionen haben einen Sender (Käufer) und einen Empfänger (Verkäufer). Sie halten u.a. Informationen über die Versandkosten und sind einem Angebot zugeordnet. Die einzlnen Posten der Transaktion finden sich in den entsprechenden Objekten der Klasse `TransactionItem`. Transaktionen können eine Reihe von Zuständen annehmen, die vom Fortschritt des Zahlungsvorgangs abhängen.

TransactionItem

Ein `TransactionItem` repräsentiert einen Posten einer Transaktion. Es enthält den zu bezahlenden Preis und den Namen des dazugehörigen Artikels.

User

Diese Klasse repräsentiert einen Nutzer der Anwendung. Die Beziehungen zu den anderen Klassen sind durch seine Rolle bedingt: Als Käufer verschickt er Anfragen (`ArticleRequest`), erhält Angebote (`Offer`), favorisiert Artikel (`Article`) oder sendet Zahlungen (`Transaction`). Als Verkäufer erhält er Anfragen, macht Angebote, verkauft Artikel oder empfängt Zahlungen.

3.1.3 Ablauf einer Anfrage

Die meisten der oben genannten Klassen besitzen einen eigenen Controller, der Anfragen vom Client (wie etwa der iOS-App) bearbeitet und beantwortet. Dazu delegieren die Controller Aufgaben an betroffene Model-Klassen und gegebenenfalls an Service-Klassen für komplexere Aufgaben. Eine dieser Service-Klassen ist der sogenannte *StateTransitionsService*, der die Zustandsübergänge der Angebote, Artikel und Anfragen verwaltet. Abbildung 3.2 zeigt diesen Ablauf am Beispiel von der Annahme eines Angebots.

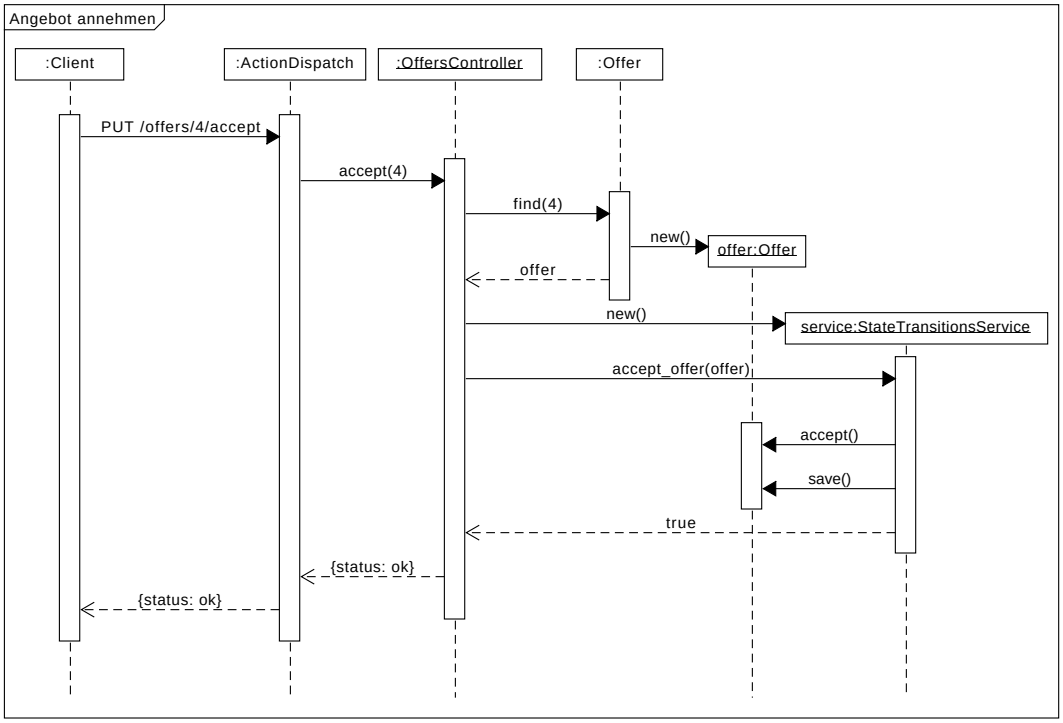

Abbildung 3.2: Sequenzdiagramm für die Annahme eines Angebots (vereinfacht).

Der Client schickt eine HTTP-Anfrage an die Rails-Anwendung, um das Angebot mit der ID 4 zu akzeptieren. Die Anwendung bestimmt dann anhand der angefragten URI den zu verwendenden Controller und die aufzurufende Methode. In diesem Beispiel ist dies der `OffersController` und die `accept()`-Methode. Der `OffersController` ruft dann die Methode `find()` auf der Model-Klasse `Offer` auf, um die entsprechende Instanz für das Angebot mit der ID 4 zu erhalten. Anschließend erstellt der Controller einen neuen `StateTransitionsService` und ruft auf diesem `accept_offer()` mit dem eben erhaltenen Angebot auf.
Der `StateTransitionsService` markiert das Angebot mittels Aufruf von `accept()` als akzeptiert und persistiert es durch den Aufruf von `save()`. An diesem Punkt markiert er außerdem alle anderen existierenden Angebote und Angebotsanfragen (`ArticleRequest`) für die im Angebot enthaltenen Artikel als ungültig und markiert den Artikel als verkauft (der Übersichtlichkeit halber nicht im Diagramm dargestellt).
Nachdem das Angebot erfolgreich persistiert wurde gibt der OffersController eine Antwort für den Client im JSON-Format zurück, die bis zu diesem weitergeleitet wird.

3.2 Identifizierte Probleme der Beispielanwendung

Die im Folgenden vorgestellten Probleme treten nicht nur in der Beispielanwendung, sondern in fast allen Rails-Anwendungen auf. Sie sind eine Folge der Verwendung des Frameworks und der strikten Einhaltung der durch das Framework vorgegebenen Strukturen.

3.2.1 Geschäftslogik in Models

Der größte Teil der Geschäftslogik in Rails-Anwendungen befindet sich in den Models. In deren Verantwortlichkeitsbereich liegt allerdings auch die Datenhaltung. Diese Tatsache ist bereits eine Verletzung des Prinzips *Separation of Concerns*, nach dem Anwendungen so aufgeteilt werden sollen, dass jeder Teil seinen eigenen Aufgabenbereich erfüllt. Beispielsweise befinden sich in dem Model `Article` folgende Zeilen:

```
class Article < ActiveRecord::Base
  ...
  scope :for_girls, -> {
    joins('LEFT JOIN article_types ON \
    articles.article_type_id = article_types.id')
    .where(*KidsSequelService.new.for_girls)
  }
  ...
end
```

Listing 3.1: Dieser *Scope* filtert alle Artikel, die nicht für Mädchen vorgesehen sind.

Durch einen sog. *Scope* kann die Ergebnismenge einer Datenbankanfrage eingeschränkt werden. Wurde ein *Scope* definiert, kann er wie eine Methode verwendet werden. In diesem Beispiel liefert der Aufruf von `Article.for_girls` alle Artikel zurück, die für Mädchen vorgesehen sind. Die Klasse `KidsSequelService` stellt einen Teil der SQL-Anweisung dafür bereit.

Zwar ist es praktisch diese Methode zur Hand zu haben, da sie auch innerhalb des Models aufgerufen werden kann, allerdings sorgt dies für eine sehr enge Kopplung der Geschäftslogik an die Datenhaltung: Der verwendete Persistenzmechanismus kann nicht mehr ohne Weiteres ausgetauscht werden und bei Änderungen am Datenbankschema müssen alle Models durchkämmt und ggf. korrigiert werden. Allein die Verwendung von Rails und dessen Modul ActiveRecord ist eine implizite Entscheidung für die Verwendung einer relationalen Datenbank, was schwer rückgängig zu machen ist, da sich Spuren des relationalen Paradigmas in der gesamten Anwendung wiederfinden.

3.2.2 Geschäftslogik in Controllern

Controller sollen hauptsächlich koordinative Aufgaben übernehmen und Nutzereingaben verarbeiten, aber keine Geschäftslogik enthalten. Dennoch findet sich in Controllern häufig Geschäftslogik, die sich nicht in einem Model unterbringen lässt, ohne dessen Verantwortungsbereich zu verwässern. Listing 4.25 zeigt einen Ausschnitt der Methode `accept` aus dem `OffersController`.

```
1  class OffersController < ApplicationController
2    def accept
3      ...
4      if offer.articles.any?(&:sold?)
5        offer.invalidate!
6      else
7        StateTransitionsService.new.accept_offer(offer)
8      end
9
10     render json: offer.message, scope: current_user, status: :ok
11     ...
12   end
13 end
```

Listing 3.2: Ausschnitt der Methode `accept` aus dem `OffersController`.

In Zeile 4 wird geprüft, ob bereits mindestens ein Artikel des betrachteten Angebots verkauft ist. Ist dies der Fall, wird das Angebot als ungültig markiert. Andernfalls wird der `StateTransitionsService` verwendet, um das Angebot zu akzeptieren. Zuletzt wird in Zeile 10 das Ergebnis der Operation im JSON-Format ausgegeben.
Diese Aktionen müssen immer geschehen, wenn ein Angebot akzeptiert wird, aber die einzige Möglichkeit sie auszuführen ist die Verwendung des Controllers. Möglicherweise beinhaltet aber ein anderer Anwendungsfall ebenfalls das Akzeptieren eines Angebots, oder der Zugriff soll beispielsweise nicht über eine HTTP-Anfrage erfolgen, sondern durch RPC[2]. Dann ist die Verwendung des Controllers nicht möglich, er ist aber die einzig vorhandene Schnittstelle.

3.2.3 Skalierbarkeit

Als Folge der beiden bereits genannten Probleme wird die Anwendung mit zunehmender Komplexität immer schlechter handhabbar, da sie eng gekoppelt ist mit dem Zugriffsmechanismus der Schnittstelle (HTTP) auf der einen Seite und dem Persistenzmechanismus auf der anderen. Rails räumt damit technischen Details eine zentrale Position in der Anwendung ein. Insbesondere bei Domänen, die naturgemäß komplex sind, sorgt die Vermischung der technischen Infrastruktur mit der Geschäftslogik der Domäne für eine unverhältnismäßig komplexe Anwendung [vgl. MT15, S. 4 f.]. Dies führt zu längeren Einarbeitungszeiten für Softwareentwickler, hohem Fehleraufkommen und somit höheren Entwicklungskosten [vgl. MT15, S. 4].

[2] *Remote Procedure Call* ist eine Methode zur Umsetzung von Interprozesskommunikation.

3.2.4 Testbarkeit

Ebenfalls eine Folge der ersten beiden genannten Probleme ist die schlechtere Testbarkeit des Codes: Durch die Vermischung der Verantwortlichkeiten lässt sich die Geschäftslogik nur schwer oder gar nicht isoliert testen. Die dann für das Testen notwendige Interaktion mit der Datenbank kostet unnötig Zeit und verlangsamt damit die gesamte Entwicklungsgeschwindigkeit.

3.2.5 Behebung der Probleme mit Domain-Driven Design

Domain-Driven Design stellt die Geschäftslogik in den Mittelpunkt der Anwendung. Sie soll sich möglichst vollständig innerhalb der Domänenschicht befinden und darf keine Abhängigkeiten von Teilen der Anwendung außerhalb der Domänenschicht aufweisen. Services sollen die Schnittstelle zur Domänenschicht darstellen und mit Infrastrukturkomponenten kommunizieren, um ihre Aufgabe zu erfüllen. Diese Schnittstelle kann dann von beliebigen Diensten genutzt und auf verschiedene Weisen bereitgestellt werden, zum Beispiel durch Controller. Um Persistenz kümmern sich ausschließlich Repositories, sodass eine Änderungen am Persistenzmechanismus lediglich Änderungen in Repositories erfordert und nicht in der gesamten Anwendung. Erreicht werden soll dadurch eine lose Kopplung zwischen den Komponenten der Anwendung und eine strikte Trennung der Verantwortlichkeiten. Daraus folgt eine verbesserte Testbarkeit des Codes. Bei einer Strukturierung der Anwendung auf diese Weise stellen die von Rails in den Mittelpunkt gerückten Teile (Zugriffs- und Persistenzmechanismus) ein austauschbares Implementierungsdetail dar.

4 Implementierung

4.1 Einführung der Domänenschicht

Da die Anwendung bisher keine dedizierte Schicht für die Geschäftslogik besitzt, muss diese noch eingeführt werden. Abschnitt 4.1.1 beschreibt die Unterteilung der Domain Objects in Entitäten und Werteobjekte. Der darauffolgende Abschnitt erläutert die Bildung der Aggregate aus den Domain Objects. Der Abschnitt 4.1.3 zeigt eine Implementierungsmöglichkeit für die modellierten Domain Objects. Da an dieser Stelle unmöglich die Implementierung der gesamten Domänenschicht im Rahmen dieser Arbeit gezeigt werden kann, wird dies am Beispiel der Annahme eines Angebots [siehe Abschnitt 3.1.1 und 3.1.3] demonstriert. Auch der letzte Abschnitt 4.1.4, der die Implementierung von Services zeigt, beschränkt sich auf dieses Beispiel.

4.1.1 Unterteilung in Entitäten und Werteobjekte

Offer

Ein Angebot kann zum Beispiel angenommen oder abgelehnt werden und hat daher einen eigenen Lebenszyklus. Die Unterscheidung zweier Angebote mit gleichen Attributen ist ebenfalls notwendig (Identität), daher ist Angebot eine Entity.

OfferItem

Ein *OfferItem* gehört immer zu einem Angebot und kann ohne dieses nicht existieren. Es wird eindeutig identifiziert durch dieses Angebot, eine Anfrage (*ArticleRequest*) und einen dazugehörigen Artikel. Ein *OfferItem* hat keinen eigenen Lebenszyklus, dieser hängt vom Angebot ab. Somit qualifiziert es sich als Value Object.

Article

Ein Artikel kann zum Verkauf stehen, verkauft oder bezahlt sein und hat damit einen Lebenszyklus. Auch die Unterscheidung zweier Artikel mit gleichen Attributen ist notwendig, somit haben sie eine Identität. Artikel sind also Entities.

ArticleBrand

Ein Hersteller gehört zwar immer zu einem Artikel und muss nicht von einem Hersteller mit gleichen Namen unterschieden werden, hat allerdings einen eigenen Lebenszyklus, da er auch jederzeit aufhören kann zu existieren. *ArticleBrand* ist somit eine Entity.

ArticleSize

Kleidungsgrößen ändern sich für gewöhnlich nicht und müssen auch bei gleichem Wert nicht voneinander unterschieden werden. Sie stehen nicht für sich, sondern gehören immer zu einem Artikel. *ArticleSize* ist ein Value Object.

ArticleRequest

Anfragen für Artikel können beantwortet, unbeantwortet oder erfüllt sein und haben somit einen Lebenszyklus. *ArticleRequest* ist eine Entity.

Transaction

Transaktionen mit gleichen Attributen müssen unbedingt unterschieden werden und haben einen durch den Zahlungsvorgang bestimmten Lebenszyklus. Eine *Transaction* ist somit eine Entity.

TransactionItem

Ein *TransactionItem* gehört immer zu einer Transaktion und hat keinen eigenen Lebenszyklus. Zwei *TransactionItem*s müssen nicht voneinander unterschieden werden. Daher sind sie Value Objects.

User

Nutzer müssen zwingend auch bei gleichem Namen voneinander unterscheidbar sein. Die Identität ist dem Nutzer inhärent. Zusätzlich haben Nutzer einen eigenen Lebenszyklus, da sie auch wieder gelöscht werden können.

4.1.2 Bildung der Aggregate

Auf *User*, *Article*, *Transaction* und *Offer* muss von außerhalb zugegriffen werden können, weshalb sie selbst nicht Teil eines Aggregats sein können, sondern Wurzeln ihres eigenen Aggregats bilden.

Das Value Object *TransactionItem* wird ausschließlich von *Transaction* referenziert und verweist selbst auch nur darauf. Daher kann es gemeinsam mit *Transaction* als Wurzel ein Aggregat bilden.

Auf das Value Object *ArticleSize* wird nur über einen Artikel zugegriffen werden und selbst hält es keine Referenzen auf andere Objekte. Selbiges gilt für *ArticleBrand*, auch wenn es sich hierbei um eine Entität handelt.
Alle Anfragen, die für einen Artikel gestellt werden, dürfen nur gültig sein solange der Artikel zum Verkauf steht. Wird der Artikel verkauft, müssen auch alle Anfragen (außer der Anfrage, die den Zuschlag erhalten hat) ungültig gemacht werden. Daher bietet es sich an auch *ArticleRequest* zu einem Teil des Aggregats zu machen. *ArticleSize*, *ArticleBrand* und eine Sammlung von *ArticleRequests* bilden also gemeinsam mit *Article* als Wurzel ein Aggregat.

OfferItem verweist zwar auf weitere Objekte (nämlich *Offer*, *Article* und *ArticleRequest*), jedoch ist dies umgekehrt nicht der Fall: Auf ein *OfferItem* wid nur über ein Angebot zugegriffen werden. *OfferItem* bildet damit ein Aggregat gemeinsam mit *Offer* als Wurzel.

4.1.3 Implementierung von Domain Objects

Im folgenden Abschnitt werden Anforderungen erläutert, die für Domain Objects generell gelten müssen. Der darauffolgende Abschnitt behandelt eine Möglichkeit diese Anforderungen zu erfüllen. Zuletzt wird die Struktur der Domänenschicht (Module und Ordner) und die konkrete Implementierung der Funktionalität zur Annahme eines Angebots erläutert.

Anforderungen

1. Die Attribute eines Domain Objects müssen leicht ersichtlich sein, sodass ein Blick in die entsprechende Klasse genügt, um zu wissen, welche Attribute bei der Instanziierung übergeben werden können.

2. Da die Attribute einzelner Domain Objects recht zahlreich sein können, sollte es möglich sein diese bei der Instanziierung als Hash zu übergeben, um die Zahl der Parameter für den Initializer niedrig zu halten und nicht jedes Attribut einzeln setzen zu müssen. Außerdem können so nicht benötigte Attribute einfach weggelassen werden.

3. Domain Objects sollten leicht serialisierbar sein (zum Beispiel im JSON-Format), um sie über das Netzwerk an den Verwender der API (zum Beispiel die iOS-App) übergeben zu können.

4. Es muss ausreichen ein Domain Object als Value Object zu kennzeichnen (zum Beispiel durch Vererbung oder Einbindung eines Moduls), um das Verhalten eines Value Objects

zu erhalten. Dazu zählt die Unveränderbarkeit der Attribute und die Vergleichbarkeit zweier Value Objects.

5. Bei der Instanziierung ganzer Aggregate sollte es möglich sein, die Attribute für die enthaltenen Domain Objects zu übergeben, sodass diese vorzugsweise ebenfalls direkt instanziiert werden können.

6. Es soll möglich sein Attribute als obligatorisch zu deklarieren, sodass bei der Instanziierung ein Wert übergeben werden muss.

7. Attribute sollen bei der Instanziierung angegeben werden, aber danach nur von innerhalb des Domain Objects verändert werden können. Dies soll verhindern, dass Domain Objects in einen ungültigen Zustand geraten.

Das Gem „Virtus“

RubyGems ist ein Paketmanager für die Sprache Ruby. Die einzelnen Pakete nennen sich *Gems*. Nach der Installation eines Gems kann auf dessen Funktionalität zurückgegriffen werden. *Virtus*[1] ist ein Gem, das einige der Anforderungen a priori erfüllt und sich somit gut für die Implementierung von Domain Objects eignet.
Listing 4.1 zeigt die Einbindung von Virtus am Beispiel der Klasse `User`.

```
1 require "virtus"
2
3 class User
4   include Virtus.model
5
6   attribute :email, String
7   attribute :age, Integer
8 end
```

Listing 4.1: Einbindung von Virtus in die Klasse `User`.

Mittels Aufruf der Methode `attribute` wird ein Attribut mit dem geg. Namen und Datentyp definiert. Die deklarierten Attribute können dann bei der Instanziierung als Hash übergeben und anschließend verwendet werden. Listing 4.2 zeigt ein Beispiel dafür.

[1] Siehe https://github.com/solnic/virtus.

```
user = User.new({:email => "user@haw-hamburg.de", :age => 34})
user.email #=> "user@haw-hamburg.de"
user.attributes #=> { :email => "user@haw-hamburg.de", :age => 34 }
```

Listing 4.2: Die Verwendung von Virtus am Beispiel der Klasse `User`.

Somit sind die ersten beiden Anforderungen erfüllt. Da der Aufruf der Methode `attributes` alle vorhandenen Attribute als Hash zurückgibt und Hashes sich unter Verwendung der Ruby-Standardbibliothek zu JSON serialisieren lassen, ist auch die dritte Anforderung erfüllt. Listing 4.3 verdeutlicht dies.

```
require "json"

user = User.new({:email => "user@haw-hamburg.de", :age => 34})
user.attributes.to_json #=> {"email":"user@haw-hamburg.de","age":34}
```

Listing 4.3: Serialisierung von Domain Objects.

Auch das Verhalten von Value Objects lässt sich mit Virtus leicht erzeugen. Dazu muss das Modul `Virtus.value_object` statt `Virtus.model` eingebunden werden. Außerdem stehen die Attribute nun in einem Block, der der Methode `values` übergeben wird, wie in Listing 4.4 gezeigt.

```
require "virtus"

class ArticleSize
  include Virtus.value_object

  values do
    attribute :label, String
    attribute :value, String
  end
end

article_size = ArticleSize.new(:label => "Groß", :value => "XL")
article_size.label #=> "Groß"
article_size.label = "Gross" #=> NoMethodError: [...]
```

Listing 4.4: Value Objects mit Virtus am Beispiel von `ArticleSize`.

Nach der Instanziierung von `ArticleSize` lassen sich so die Werte von Attributen nicht mehr ändern. Die vierte Anforderung ist also erfüllt.

Statt einem Basis-Datentyp können auch selbst definierte Klassen als Datentyp von Attributen angegeben werden. Auf diese Weise unterstützt Virtus die direkte Erzeugung ganzer Aggregate, wodurch auch die fünfte Anforderung erfüllt ist. Listing 4.5 zeigt dies am Beispiel des Aggregats bestehend aus `Article` und der o.g. Klasse `ArticleSize`.

```ruby
require "virtus"

class Article
  include Virtus.model

  attribute :text, String
  attribute :price, Integer # Preis in Cent
  attribute :size, ArticleSize
end

article = Article.new(
  :text => "Eine schöne Hose.",
  :price => 500,
  :size => { :label => "Klein", :value => "S" }
)
article.size.label #=> "Klein"
```

Listing 4.5: Erzeugung von Aggregaten am Beispiel von `Article`.

Attribute lassen sich durch die Angabe von `:required => true` bei der Deklaration als obligatorisch markieren, wie Listing 4.6 zeigt. Damit ist die sechste Anforderung erfüllt.

```ruby
require "virtus"

class User
  include Virtus.model
  attribute :email, String, :required => true
end
```

Listing 4.6: Deklaration von obligatorischen Attributen am Beispiel von `User`.

Um die letzte Anforderung zu erfüllen, muss Virtus beim Einbinden des Moduls erst konfiguriert werden, um die Manipulation von Attributen nach der Instanziierung von außerhalb zu unterbinden.

```
1 class User
2   include Virtus.model(:writer => :private)
3 end
```

Listing 4.7: Markierung der Setter-Methoden als `private`.

Auf diese Weise ist leider auch das Schreiben der Attribute bei der Instanziierung nicht möglich. Um zu bestimmen welche Attribute bei der Instanziierung gesetzt werden dürfen, ruft Virtus eine Klassenmethode namens `allowed_methods` auf, die ein Array, das die erlaubten Methodennamen enthält, zurückgibt. Diese kann nach dem Einbinden des Moduls überschrieben werden, sodass das Array zusätzlich die Setter-Methoden der Attribute beinhaltet. Listing 4.8 zeigt die Methode.

```
1 class User
2   include Virtus.model(:writer => :private)
3   def self.allowed_methods
4     setters = attribute_set.map { |attr| "#{attr.name}=" }
5     super + setters
6   end
7 end
```

Listing 4.8: Markierung der Setter-Methoden als `private`.

In Zeile 4 wird ein Array erzeugt, der die Setter-Methoden der deklarierten Attribute enthält. Anschließend wird die Implementierung der Superklasse aufgerufen und das Ergebnis mit den Setter-Methoden zusammengeführt und zurückgegeben. Somit sind alle Anforderungen erfüllt.

Struktur der Domänenschicht

Das Domänenmodell sollte sich in seinem eigenen Modul befinden, um Namenskonflikte mit anderen Teilen der Anwendung zu vermeiden. Für dieses Modul bietet sich als Bezeichnung `Domain::Model` an (das Modul `Model` ist also Teil des übergeordneten Moduls `Domain`). Da es in Ruby Konvention ist, dass für jedes Modul auch ein entsprechender Ordner existiert, werden sich die Domain Objects im Ordner `domain/model/` befinden.

Um das Wissen über die Tatsache, dass Virtus verwendet wird nicht in der gesamten Domänenschicht durch die Einbindung des jeweiligen Moduls in jedem Domain Object zu verteilen und um nicht in jedem Domain Object Virtus konfigurieren zu müssen, werden zwei Basisklassen eingeführt: `Entity` und `ValueObject`. Diese werden in einem neuen Modul mit der Bezeichnung `Domain::Support` untergebracht, das Dienste für das Domänenmodell in `Domain::Model` bereitstellen soll. Diese beiden Klassen sollen zudem Funktionalität bereitstellen, die von jedem `Entity` beziehungsweise `ValueObject` benötigt wird. Listing 4.9 zeigt die Basisklasse `Entity`. Da jede Entity auch ein identifizierendes Attribut braucht, wird das Attribut `id` hier definiert.

```ruby
require "virtus"

module Domain
  module Support
    class Entity
      include Virtus.model(:writer => :private)

      attribute :id, Integer

      def self.allowed_methods
        setters = attribute_set.map { |attr| "#{attr.name}=" }
        super + setters
      end
    end
  end
end
```

Listing 4.9: Inhalt der Datei `domain/support/entity.rb`

Dadurch, dass Domain Objects, je nachdem ob sie Entities oder Value Objects sind, nun von einer der beiden Klassen ableiten müssen, ist auch für die Entwickler leicht ersichtlich, worum es sich beim betrachteten Objekt handelt. Listing 4.10 zeigt die Basisklasse `ValueObject`. Da Value Objects ohnehin nicht veränderbar sind, müssen die Setter-Methoden hier nicht als `private` markiert werden.

```ruby
require "virtus"

module Domain
  module Support
    class ValueObject
      include Virtus.value_object
    end
  end
end
```

Listing 4.10: Inhalt der Datei `domain/support/value_object.rb`

Beispiel: Angebot annehmen

Ein Teil des Anwendungsfalls zur Annahme eines Angebots ist der Verkauf eines Artikels: Wurde ein Angebot angenommen, sind alle im Angebot enthaltenen Artikel an den Interessenten verkauft. Daher wird zunächst die Implementierung der Funktionalität zum Verkauf eines Artikels gezeigt.
Das Domain Object `Article` hat die in Listing 4.11 gezeigten Attribute.

```ruby
module Domain
  module Model
    class Article < Entity
      attribute :text, String, :required => false # Kurze Beschreibung
      attribute :price, Integer                   # Preis in Cent
      attribute :seller_id, Integer               # ID des Verkäufers
      attribute :state, String                    # Zustand, z.B. verkauft
      attribute :brand, ArticleBrand              # Hersteller
      attribute :size, ArticleSize                # Größe
      attribute :requests, Array[ArticleRequest]  # Vorliegende Anfragen
    end
  end
end
```

Listing 4.11: Das Domain Object `Article`.

Der Zustand des Artikels könnte zum Beispiel über das *State pattern* [siehe GHJV94, S. 305] verwaltet werden. In der bestehenden Anwendung existiert aber bereits eine Lösung in Form eines Gems namens *Workflow*. Dabei können die Zustände und Zustandsübergänge mit einer

DSL[2] beschrieben werden. Listing 4.12 zeigt die Verwendung am Beispiel von `Article`. Da *Workflow* den aktuellen Zustand als String in einem Attribut namens `:workflow_state` speichert, wurde das Attribut `:state` darin umbenannt. In diesem und in den folgenden Beispielen werden nicht alle Zustände gezeigt, die das Domain Object annehmen kann, sondern nur die für diesen Anwendungsfall benötigten Zustände.

```
1  require 'workflow'
2
3  module Domain
4    module Model
5      class Article < Entity
6        include Workflow
7
8        ... # Andere Attribute hier nicht gezeigt.
9        attribute :workflow_state, String, :default => "created"
10
11       workflow do
12         state :created do
13           event :sell, transitions_to: :sold
14         end
15         state :sold
16       end
17     end
18   end
19 end
```

Listing 4.12: Zustandsverwaltung von `Article` mittels des Gems *Workflow*.

`Article` kann jetzt die Zustände `:created` (Zeile 15) und `:sold` (Zeile 18) annehmen. Bei der Deklaration des Attributs wird `"created"` als Standardwert angegeben und ist daher der Initialzustand. Von ihm ausgehend kann ein Zustandsübergang zu `:sold` stattfinden, wenn das Ereignis `:sell` eintritt. Ereignisse sind lediglich Methodenaufrufe, die dem Namen des Ereignisses gefolgt von einem Ausrufezeichen entsprechen. Listing 4.13 zeigt ein Beispiel.

[2] Eine *Domain-specific language* ist eine einfache Sprache, die einen bestimmten Aspekt eines Systems beschreibt.

```
article = Domain::Model::Article.new(...)
article.workflow_state #=> "created"
article.sell!
article.workflow_state #=> "sold"
article.sell! #=> Fehler, da kein Zustandsübergang von "sold" definiert.
```

Listing 4.13: Zustandsübergang von `Article`.

Bei dieser Lösung kann aber der Zustand des Artikels von außerhalb beliebig verändert werden, vorausgesetzt die Zustandsübergänge werden eingehalten. Dies ist alles andere als wünschenswert, da das Objekt eigentlich selbst entscheiden sollte, wann es in welchen Zustand wechselt. Außerdem schränkt es die Gestaltungsmöglichkeiten der Schnittstelle des Objekts ein, da es für jeden Zustandsübergang eine öffentlich verfügbare Methode geben muss. Eric Evans spricht in seinem Buch von *Intention-Revealing Interfaces*, also Schnittstellen, die dem Verwender des Objekts verständlich machen, was das Objekt kann und wie es zu verwenden ist, ohne die Implementierung kennen zu müssen [siehe Eva03, S. 246]. Dies wird erreicht, indem sorgfältig ausgewählt wird, welche Methoden des Objekts öffentlich verfügbar sein sollen und wie diese benannt werden. Die Auslagerung des Zustands in ein eigenes Value Object, das intern verwaltet wird, ermöglicht eine freie Gestaltung der Schnittstelle. Listing 4.14 zeigt eine mögliche Lösung für die Auslagerung des Zustandes.

```
1  module Domain
2    module Model
3      class Article < Entity
4        ...
5        private class State < ValueObject
6          include Workflow
7          attribute :workflow_state, String, :default => "created"
8
9          def initialize(attrs={})
10           super(attrs)
11           self.should_be_in_valid_state
12         end
13
14         def should_be_in_valid_state
15           states = self.class.workflow_spec.state_names.map(&:to_s)
16           if !states.include?(self.workflow_state.to_s)
17             raise "Not a valid state: #{self.workflow_state}"
18           end
19         end
20
21         workflow do
22           state :created { event :sell, transitions_to: :sold }
23           state :sold
24         end
25       end
26       attribute :state, State, :reader => :private
27     end
28   end
29 end
```

Listing 4.14: Auslagerung des Zustands in das *Value Object* `State`.

Zwar handelt es sich bei dem Objekt `State` nun um ein eigentlich unveränderliches *Value Object*, aber die Umstände rechtfertigen Veränderlichkeit (siehe Abschnitt 2.3.7): Da das Objekt ohnehin nur innerhalb von `Article` verwendet wird, kann es auch nicht von außerhalb manipuliert werden und das Ersetzen des Objekts bei jedem Zustandsübergang wäre unnötig umständlich.
Der Zusatz `:reader => :private` bei der Definition des Attributs in Zeile 5 verhindert den Zugriff auf das Zustandsobjekt von außen. Die Methode `should_be_in_valid_state` prüft, ob der gesetzte Zustand in der Menge der gültigen Zustände enthalten ist. Falls nicht, wird ein Fehler geworfen.

Nun kann die eigentliche Geschäftslogik für den Verkauf eines Artikels implementiert werden. Da ein Artikel immer *an* einen Nutzer verkauft wird, bietet sich `sell_to` als Methodenname an. Der Nutzer, an den der Artikel verkauft werden soll, wird als Parameter übergeben. Listing 4.15 zeigt die Implementierung der Methode.

```
1  module Domain
2    module Model
3      class Article < Entity
4        ... # Attribute hier nicht gezeigt.
5        def sell_to(buyer)
6          requests.each do |request|
7            if request.requester_id == buyer.id
8              request.fulfill
9            else
10             request.invalidate
11           end
12         end
13
14         state.sell!
15         publish :article_sold, :article_id => self.id
16       end
17       ...
18     end
19   end
20 end
```

Listing 4.15: Implementierung der Methode `sell_to`.

In Zeile 7 wird über alle Anfragen für den Artikel iteriert. Anschließend wird in den Zeilen 8 bis 12 für jede Anfrage geprüft, ob sie vom gegebenen Benutzer stammt. Falls dies zutrifft, wird die Anfrage erfüllt (`fulfill`) und falls nicht, wird sie ungültig (`invalidate`). Anschließend findet in Zeile 15 ein Zustandsübergang zu `:sold` statt. Zuletzt wird in Zeile 16 ein Ereignis veröffentlicht, das den Verkauf des Artikels kommuniziert. Der zugrundeliegende Mechanismus für die Veröffentlichung wird im Abschnitt 4.2 behandelt.
Listing 4.16 zeigt die Attribute des Domain Objects `ArticleRequest`.

```ruby
module Domain
  module Model
    class ArticleRequest < Entity
      attribute :requester_id, String
      attribute :recipient_id, String
      attribute :state, State, :reader => :private
    end
  end
end
```

Listing 4.16: Attribute des Domain Objects `ArticleRequest`.

Da auch `ArticleRequest` mehrere Zustände durchläuft, ist dies ein guter Zeitpunkt die Funktionalität für die Zustandsverwaltung, die auch von `Article` verwendet wird, auszulagern. Dazu wird eine neue Klasse namens `StateValue` im Modul `Domain::Support` eingeführt. Die Implementierung ist dieselbe wie in Listing 4.14 gezeigt, abgesehen von der Bestimmung des Initialzustandes und der Methode `declare_states`, die die Verwendung des Gems *Workflow* verbergen soll, um es austauschbar zu halten. Für den Fall, dass kein Zustand übergeben wurde, wird nun ein Symbol als Standardwert angegeben. Bei der Instanziierung prüft Virtus, ob eine gleichnamige Methode in der Klasse existiert und übernimmt ihren Rückgabewert.

```ruby
module Domain
  module Support
    class StateValue < ValueObject
      include Workflow
      attribute :workflow_state, String, :default => :initial_state

      def initial_state
        self.class.workflow_spec.initial_state.name
      end

      def declare_states(&block)
        workflow(&block)
      end
      ... # Methoden 'initialize' & 'should_be_in_valid_state' wie bisher.
    end
  end
end
```

Listing 4.17: Die Klasse `StateValue` in der Datei `domain/support/state_value.rb`.

Nun müssen in der Klasse `ArticleRequest` nur noch die Zustände definiert werden. Listing 4.18 zeigt die Implementierung der verbleibenden Methoden und die Definition der Zustände.

```
module Domain
  module Model
    class ArticleRequest < Entity
      ... # Attribute hier nicht gezeigt

      def fulfill
        self.state.fulfill!
      end

      def invalidate
        self.state.invalidate!
      end

      class State < ::Domain::Support::StateValue
        declare_states do
          state :created do
            event :fulfill, transitions_to: :fulfilled
            event :invalidate, transitions_to: :invalidated
          end
          state :fulfilled
          state :invalidated
        end
      end
    end
  end
end
```

Listing 4.18: Implementierung von `ArticleRequest`.

Die Verwendung der Klasse `StateValue` ist für `Article` analog.
Nun muss noch das Domain Object `Offer` implementiert werden. Listing 4.19 zeigt die benötigten Attribute und Zustände.

```ruby
module Domain
  module Model
    class Offer < Entity
      attribute :buyer_id, Integer                  # ID des Käufers
      attribute :seller_id, Integer                 # ID des Verkäufers
      attribute :offer_items, Array[OfferItem]      # Artikel des Angebots

      class State < ::Domain::Support::StateValue
        declare_states do
          state :created do
            event :accept, transitions_to: :accepted
          end
          state :accepted
        end
      end
      attribute :state, State, :reader => :private # Zustand des Angebots
    end
  end
end
```

Listing 4.19: Attribute und Zustände von `Offer`.

Um ein Angebot zu akzeptieren muss sichergestellt werden, dass derjenige, der das Angebot akzeptieren möchte auch der Empfänger des Angebots ist. Wenn das der Fall ist, kann das Angebot akzeptiert werden. Listing 4.20 zeigt die Methode `accept` und die dazugehörigen Methoden zur Validierung. In Zeile 7 wird zunächst die Methode `should_be_recipient` aufgerufen, die eine Exception wirft (Zeile 20), wenn es sich beim übergebenen Nutzer nicht um den Empfänger des Angebots handelt. Anschließend findet in Zeile 9 ein Zustandsübergang zu `:accepted` statt und ein entsprechendes Ereignis wird in Zeile 10 veröffentlicht.

```ruby
module Domain
  module Model
    class Offer < Entity
      ... # Attribute hier nicht gezeigt.
      def accept(buyer)
        should_be_recipient(buyer)
        state.accept!
        publish :offer_accepted, :offer_id => self.id,
                                 :buyer_id => buyer.id
      end

      def is_recipient?(buyer)
        buyer.id == self.buyer_id
      end

      protected def should_be_recipient(buyer)
        if !is_recipient?(buyer)
          raise "The given user is not the recipient of this offer."
        end
      end
      ... # Zustände hier nicht gezeigt.
    end
  end
end
```

Listing 4.20: Implementierung von `Offer`.

4.1.4 Implementierung von Services

Die Domain Objects können jetzt von einem *Service* verwendet werden, der als Schnittstelle für die Domänenschicht dient und später von den Controllern verwendet wird. Dieser Service muss zur Annahme eines Angebots folgende Schritte durchführen:

1. Das Angebot akzeptieren.
2. Im Angebot enthaltene Artikel verkaufen.
3. Andere Angebote, die einen der verkauften Artikel enthalten, als ungültig markieren.

Da in jedem Schritt ein Aggregat geändert wird und innerhalb einer Transaktion möglichst nur ein Aggregat persistiert werden soll [siehe Ver13, S. 287], muss entschieden werden welche Schritte innerhalb einer Transaktion ausgeführt werden müssen und bei welchen *eventual*

consistency ausreichend ist. Eine Möglichkeit *eventual consistency* zu implementieren, ist die Verwendung von Domain Events. Bei einem relevanten Ereignis in der Domänenschicht (zum Beispiel „Angebot wurde akzeptiert“) wird eine Nachricht veröffentlicht, die diese Information enthält. Dies geschieht auf *asynchrone* Weise, das heißt der Veröffentlicher der Nachricht wartet nicht darauf, dass die Nachricht gesendet oder gar verarbeitet wurde, sondern fährt nach dem Senden mit dem Programmablauf fort. Ein oder mehrere Empfänger reagieren auf die Nachricht und führen die nötigen Aktionen durch, um die Konsistenz der Daten herzustellen. Dabei muss die Auslieferung der Nachrichten garantiert sein, da ansonsten dauerhaft inkonsistente Daten die Folge sein können. Mit diesem Problem beschäftigt sich Abschnitt 4.2.2.

In der Zeit zwischen dem Versand der Nachricht und der Herstellung der Konsistenz können Nutzer der Anwendung eine nicht konsistente Sicht auf die Daten der Anwendung haben: Angenommen die genannten Schritte 2 und 3 würden nicht innerhalb einer Transaktion mit dem ersten Schritt ausgeführt, sondern erst durch den Empfang einer Nachricht, die die erfolgte Ausführung des ersten Schritts kommuniziert. In dem Zeitraum zwischen Versand der Nachricht und Herstellung der Konsistenz kann es passieren, dass ein anderes Angebot angenommen wird, das ebenfalls einen Artikel des bereits akzeptierten Angebots enthält. Der Nutzer würde von der Anwendung ein positives Feedback erhalten, obwohl der Artikel bereits verkauft ist, weil die Anwendung beziehungsweise die Datenbank diesen Zustand noch nicht reflektiert. Somit müsste der Nutzer, der das Angebot zu einem späteren Zeitpunkt angenommen hat im Nachhinein benachrichtigt werden, dass seine Aktion doch nicht erfolgreich war und er die Artikel nicht erhalten kann. Dies wäre zwar sicher eine frustrierende Erfahrung, aber die Eintrittswahrscheinlichkeit für dieses Ereignis ist eher gering, wie die folgende grobe Annäherung zeigt.

Angenommen ein Artikel ist in durchschnittlich 10 Angeboten enthalten und Nutzer reagieren auf Angebote innerhalb von 12 Stunden. Zusätzlich beträgt der Zeitraum zwischen Versand der Nachricht und Herstellung der Konsistenz im Schnitt 100 Millisekunden. Das Problem lässt sich durch die Division der 12 Stunden durch die 100 Millisekunden vereinfachen. Dadurch ergeben sich $\frac{43\,200\,000\,\text{ms}}{100\,\text{ms}} = 432\,000$ mögliche Zeitpunkte zum Akzeptieren eines Angebots, da 12 Stunden = 43 200 000 ms. Natürlich können Ereignisse nicht nur zu jeder vollen Millisekunde stattfinden, sondern auch dazwischen, aber bei dieser Rechnung handelt es sich lediglich um eine Schätzung. Nun lässt sich das Problem mit dem des Geburstagsparadoxons vergleichen. Statt „Wie hoch ist die Wahrscheinlichkeit, dass in einem Raum mit n Personen mindestens 2 am selben Tag Geburstag haben?“ ist die Frage nun: „Wie hoch ist die Wahrscheinlichkeit, dass von 10 Personen mindestens 2 Personen zum selben Zeitpunkt ein Angebot

akzeptieren?“
Zunächst wird die Wahrscheinlichkeit von P' gesucht, dass jede Person einen anderen Zeitpunkt wählt, um anschließend die Wahrscheinlichkeit von $P = 1 - P'$, dass mindestens 2 Personen den gleichen Zeitpunkt wählen, zu erhalten. Für das Geburstagsparadoxon lässt sich P' wie folgt berechnen, wobei n der Anzahl der möglichen Zeitpunkte (in diesem Fall Tage im Jahr) und r der Anzahl der Personen im Raum entspricht [siehe Beh13, S. 97 f. für die Herleitung]:

$$P' = \frac{n!}{n^r * (n-r)!}$$

Also gilt für dieses Beispiel mit $n = 432\,000$ und $r = 10$:

$$P' = \frac{432\,000!}{432\,000^{10} * (432\,000 - 10)!} = 0,999895\ldots$$

$$P = 1 - P' = 0,000104\ldots \approx 0,104\,‰$$

Bei 0,104 Promille wird also im Schnitt in einem von 9 615 Fällen ein Angebot erfolgreich akzeptiert, das nicht hätte akzeptiert werden dürfen.

Würden die Schritte 1 und 2 innerhalb einer Transaktion ausgeführt und Schritt 3 asynchron über die Veröffentlichung einer Nachricht, wären die Folgen weniger gravierend: Dass ein Artikel des akzeptierten Angebots bereits verkauft wurde, wird sofort bemerkt und kann dem Nutzer unmittelbar mitgeteilt werden. Dieser Fall muss außerdem ohnehin berücksichtigt werden, da die Benutzeroberfläche der Anwendung (also der iOS-App) die Daten nicht in Echtzeit darstellt und ein Angebot seit der letzten Aktualisierung auch mit ausschließlicher Verwendung von Transaktionen durch einen anderen Nutzer bereits akzeptiert worden sein kann.
Listing 4.21 zeigt die Implementierung des `OfferService` auf diese Weise.

```
1  module Service
2    class OfferService
3      def accept_offer(offer_id, buyer_id)
4        articles = ArticleRepository.find_contained_in_offer(offer_id)
5        articles.each do |article|
6          article.sell_to(buyer_id)
7          ArticleRepository.save(article)
8        end
9
10       offer = OfferRepository.find(offer_id)
11       offer.accept(buyer_id)
12       OfferRepository.save(offer)
13     end
14   end
15 end
```

Listing 4.21: Der `OfferService`.

In Zeile 4 werden durch das `ArticleRepository` alle Artikel abgerufen, die in dem Angebot mit der ID `offer_id` enthalten sind. Anschließend wird über diese Artikel iteriert und jeder an den Nutzer mit der ID `buyer_id` verkauft. Der Artikel wird dann mithilfe des Repositories persistiert. In Zeile 10 wird das Angebot mit der ID `offer_id` abgerufen, akzeptiert und schließlich persistiert.

Das Transaktionsmanagement ist nicht Aufgabe der Domänenschicht, sondern der Anwendungsschicht [vgl. Ver13, S. 432 f.]. Es ist sinnvoll die Funktionalität hierfür den Services in einer Basisklasse zur Verfügung zu stellen. Diese Basisklasse trägt den Namen `ApplicationService` und befindet sich bei den anderen Services im Ordner `service/`. Listing 4.22 zeigt die Methode `transaction`, die einen Ruby-Block akzeptiert und dessen Inhalt in eine Transaktion einbettet. Dazu wird eine Klasse namens `UnitOfWork` verwendet, die die benötigten Anweisungen für die Durchführung einer Transaktion für die verwendete Datenbank kapselt. Die Implementierung dieser Klasse wird in Abschnitt 4.3.2 erläutert.

```
1  module Service
2    class ApplicationService
3      include Infrastructure::Persistence
4
5      def transaction
6        UnitOfWork.instance.begin
7        begin
8          yield
9          UnitOfWork.instance.commit
10       rescue Exception => e
11         UnitOfWork.instance.rollback
12         raise e
13       end
14     end
15   end
16 end
```

Listing 4.22: Die Basisklasse `ApplicationService`.

In Zeile 4 wird die Transaktion gestartet. Anschließend wird in Zeile 6 der übergebene Block ausgeführt. Wurde keine Exception geworfen, wird die Transaktion festgeschrieben (Zeile 7). Ist allerdings ein Fehler aufgetreten, wird dieser in Zeile 8 abgefangen und die Transaktion zurückgerollt. Die Exception wird anschließend erneut geworfen, um den weiteren Programmablauf nicht zu behindern.

Der `OfferService` kann nun vom `ApplicationService` erben und die `transaction`-Methode verwenden, wie Listing 4.23 zeigt.

```
1  module Service
2    class OfferService < ApplicationService
3      def accept_offer(offer_id, buyer_id)
4        transaction do
5          ... # Anweisungen, wie vorher gezeigt.
6        end
7      end
8    end
9  end
```

Listing 4.23: Der `OfferService`.

Zuletzt müssen noch andere Angebote als ungültig markiert werden (Schritt 3), was auf asynchrone Weise geschehen muss. Wie im vorigen Abschnitt beschreiben, wird in der Domänenschicht nach dem Verkauf eines Artikels ein Domain Event namens `:article_sold` veröffentlicht. Auf dieses kann nun im `OfferService` reagiert werden. Dies wird durch die Verwendung eines im Rahmen dieser Bachelorarbeit entwickelten Gems namens „Event Dispatcher“ erreicht, das eine einfache DSL für das Abonnieren und Verarbeiten von Ereignissen bereitstellt. Listing 4.24 zeigt die Verwendung des Gems. Die Funktionsweise des Gems wird im nächsten Abschnitt erläutert, der Code ist im Anhang (CD) zu finden.

```
1  module Service
2    class OfferService < ApplicationService
3      include EventDispatcher::Subscriber
4
5      subscribe :article_sold do |event|
6        OfferService.new.invalidate_offers_for_article(event.article_id)
7      end
8
9      def invalidate_offers_for_article(article_id)
10       transaction do
11         offers = OfferRepository.find_offers_for_article(article_id)
12         offers.select { |offer| offer.is_valid? }.each do |offer|
13           offer.invalidate
14           OfferRepository.save(offer)
15         end
16       end
17     end
18     ... # Andere Methoden hier nicht gezeigt.
19   end
20 end
```

Listing 4.24: Abonnieren und Verarbeiten des Ereignisses `:article_sold`.

In Zeile 3 wird das Modul `EventDispatcher::Subscriber` eingebunden, das die DSL in Form von der Methode `subscribe` bereitstellt. Diese akzeptiert ein Symbol als Ereignisnamen und einen Block, der ausgeführt wird sobald das Ereignis eingetreten ist. Dabei erhält der Block ein Objekt `event`, das alle Informationen zum Ereignis enthält, die bei der Veröffentlichung in einem Hash übergeben wurden. Im Block selbst werden zunächst beim `OfferRepository` alle Angebote erfragt, die den Artikel mit der geg. ID enthalten (Zeile 6). Anschließend wird über alle erhaltenen Angebote, die noch gültig sind (also nicht akzeptiert, abgelehnt oder ungültig)

iteriert und jedes als ungültig markiert. In Zeile 9 wird dann die Änderung persistiert.

Der Service kann nun vom `OffersController` verwendet werden. Listing 4.25 zeigt die Methode `accept` des Controllers.

```
1 class OffersController < ApplicationController
2   def accept
3     offer_service = Service::OfferService.new
4     offer_service.accept_offer(current_user.id, params[:id])
5     render json: { status: "ok", message: "Offer was accepted." }
6   end
7 end
```

Listing 4.25: Die Methode `accept` im `OffersController`.

Die Methode `current_user` ist eine Hilfsmethode, die den aktuell eingeloggten Nutzer zurückgibt. Der Hash `params` enthält alle bei der HTTP-Anfrage übergebenen Parameter. Hier wird auf den Wert des Parameters `:id` zugegriffen, der die ID des zu akzeptierenden Angebots enthält. Zuletzt wird das Ergebnis der Anfrage im JSON-Format zurückgegeben.

4.2 Implementierung von Eventual Consistency mit Domain Events

4.2.1 Veröffentlichung und Empfang von Domain Events

Aus den Aggregaten heraus sollen Domain Events durch den Aufruf der Methode `publish` veröffentlicht werden können. Diese Domain Events können für Parteien aus dem eigenen Bounded Context relevant sein, aber möglicherweise auch für Interessenten in fremden Bounded Contexts. Dies macht den Einsatz eines zuverlässigen Mechanismus für Interprozesskommunikation, wie zum Beispiel einer Message Queue, unerlässlich. Dabei soll sich kein Wissen über die dafür verwendete Infrastruktur in der Domänenschicht befinden, um eine möglichst lose Kopplung zwischen der Domänenschicht und den Komponenten der Infrastruktur zu erreichen. In seinem Buch zeigt Vaughn Vernon einen Ansatz, um dies zu erreichen [siehe Ver13, S. 296 - 300]:
Innerhalb der Domänenschicht implementiert eine Klasse namens `DomainEventPublisher` das *Observer-Pattern* [siehe GHJV94, S. 293]. Über diese können Aggregate Domain Events veröffentlichen. Services können sich dann beim `DomainEventPublisher` registrieren, um

über veröffentlichte Domain Events informiert zu werden und diese über Infrastrukturdienste im eigenen und in fremden Bounded Contexts bekannt zu machen.

Listing 4.26 zeigt eine Möglichkeit für die Implementierung der Klasse. Die Abonnenten werden dabei in einem `Set` (dem Ruby-Äquivalent zum mathematischen Konzept der Menge) gehalten, um Duplikate zu verhindern. Beim Aufruf von `subscribe` wird das übergebene Objekt in das `Set` eingetragen (Zeile 11). Bei der Veröffentlichung eines Domain Events durch den Aufruf von `publish` wird über alle eingetragenen Abonnenten iteriert und auf jedem die Methode `handle_event` aufgerufen (Zeilen 15 und 16).

```
1  require 'set'
2
3  module Domain
4    module Model
5      class DomainEventPublisher
6        def initialize
7          @listeners = Set.new
8        end
9
10       def subscribe(listener)
11         @listeners.add(listener)
12       end
13
14       def publish(event_name, event_hash)
15         @listeners.each do |listener|
16           listener.handle_event(event_name, event_hash)
17         end
18       end
19     end
20   end
21 end
```

Listing 4.26: Implementierung in der Datei `domain/model/domain_event_publisher.rb`.

Zu beachten ist, dass es für jeden Thread nur eine Instanz von `DomainEventPublisher` geben darf, um die auftretenden Domain Events von verschiedenen Anfragen nicht zu vermischen. Dies wird erreicht, indem der Konstuktor als privat markiert und die Instanziierung über eine Methode gesteuert wird, die für jeden Thread lediglich eine Instanz erzeugt. Listing 4.27 zeigt die Implementierung dieses Konzepts eines „Singletons pro Thread“.

```ruby
module Domain
  module Model
    class DomainEventPublisher
      private_class_method :new
      def self.instance
        Thread.current[:domain_event_publisher] ||= new
      end
      ... # Instanzmethoden und Initializer hier nicht gezeigt.
    end
  end
end
```

Listing 4.27: Erzeugung von einer Instanz pro Thread.

Der `DomainEventPublisher` kann nun in der Basisklasse `Entity` verwendet werden, um allen Entitäten die in den vorigen Abschnitten verwendete `publish`-Methode zur Verfügung zu stellen. Listing 4.28 zeigt die Implementierung.

```ruby
module Domain
  module Support
    class Entity < DomainObject
      def publish(event_name, event_hash)
        DomainEventPublisher.instance.publish(event_name, event_hash)
      end
      ... # Andere Methoden hier nicht gezeigt.
    end
  end
end
```

Listing 4.28: Verwendung des `DomainEventPublisher` in der Basisklasse `Entity`.

Nun können sich Services beim `DomainEventPublisher` registrieren, um auftretende Domain Events über den eigenen Prozess hinaus bekannt zu machen. Die Veröffentlichung sollte jedoch nicht sofort geschehen: Da die Änderungen an den Domain Objects erst bei Ende der Transaktion festgeschrieben werden, würden die Domain Events eine Änderung kommunizieren, die aus Sicht der anderen Teile der Anwendung noch gar nicht stattgefunden hat. Daher müssen die Domain Events zunächst gesammelt und erst *nach* Ende der Transaktion veröffentlicht werden. Diese Funktionalität muss allen Services zur Verfügung stehen und sollte daher in

der Basisklasse `ApplicationService` implementiert werden. Listing 4.29 zeigt den neuen Initializer der Klasse.

```
module Service
  class ApplicationService
    def initialize
      @occurred_events = []
      DomainEventPublisher.instance.subscribe(self)
    end
    ...
  end
end
```

Listing 4.29: Die veränderte Methode `transaction` der Basisklasse `ApplicationService`.

Auftretende Domain Events können nun mittels der Instanzvariable `@occurred_events` in einem Array vorgehalten werden. Wie in Listing 4.26 gezeigt, werden Objekten, die sich beim `DomainEventPublisher` registriert haben, durch den Aufruf der Methode `handle_event` auftretende Domain Events mitgeteilt. Listing 4.30 zeigt die Implementierung dieser Methode, die die Domain Events sammelt. Die dort verwendete Klasse `Struct` gehört zur Ruby-Standardbibliothek und ist angelehnt an das Konzept von Structs aus der Programmiersprache C. Es ist also ein reiner Datenbehälter ohne Verhalten.

```
module Service
  class ApplicationService
    ...
    Event = Struct.new(:name, :hash)

    def handle_event(event_name, event_hash)
      @occurred_events << Event.new(event_name, event_hash)
    end
    ...
  end
end
```

Listing 4.30: Die veränderte Methode `transaction` der Basisklasse `ApplicationService`.

Nach dem Ende der Transaktion müssen die gesammelten Domain Events nun noch veröffentlicht werden. Listing 4.31 zeigt die Methode `publish_events`, die ebendies bewerkstelleigt. Zur Veröffentlichung wird das Gem „Event Dispatcher“ verwendet.

```ruby
module Service
  class ApplicationService
    ...
    def publish_events
      while !@occurred_events.empty?
        event = @occurred_events.shift
        EventDispatcher.publish(event.name, event.hash)
      end
    end
    ...
  end
end
```

Listing 4.31: Die veränderte Methode `transaction` der Basisklasse `ApplicationService`.

Diese Methode muss nun noch am Ende einer erfolgreichen Transaktion aufgerufen werden, wie Listing 4.32 zeigt.

```ruby
module Service
  class ApplicationService
    ...
    def transaction
      UnitOfWork.instance.begin
      begin
        yield
        UnitOfWork.instance.commit
        publish_events
      rescue Exception => e
        UnitOfWork.instance.rollback
        raise e
      end
    end
    ...
  end
end
```

Listing 4.32: Die veränderte Methode `transaction` der Basisklasse `ApplicationService`.

Funktionsweise des Gems „Event Dispatcher“

Dieser Abschnitt erläutert die Funktionsweise des Gems „Event Dispatcher“. Dabei wird nicht die Implementierung des Gems, sondern bloß dessen Verwendung gezeigt. Der gesamte Code des Gems ist aber im Anhang (CD) zu finden.
Wie bereits gezeigt, stellt das Gem „Event Dispatcher“ eine einfache Möglichkeit zur Veröffentlichung und dem Empfang von Domain Events zur Verfügung. Dazu können verschiedene Mechanismen verwendet werden, die mittels *Adaptern* implementiert bzw. angebunden werden können. Der Event Dispatcher besitzt zwei Adapter: Einen *In-Memory*-Adapter zu Testzwecken, der Registrierungen im Hauptspeicher hält und veröffentlichte Ereignisse synchron verarbeitet, und einen Adapter für den Message Broker RabbitMQ, der Ereignisse über diesen veröffentlicht und asynchron verarbeitet. Jeder Adapter implementiert die Methoden `publish` und `subscribe` zum Veröffentlichen bzw. Abonnieren von Ereignissen. Bei der Konfiguration kann angegeben werden, welcher Adapter verwendet werden soll, wie in Listing 4.33 gezeigt.

```
1 EventDispatcher.configure do |config|
2   config.adapter = EventDispatcher::Adapters::RabbitMQ.new
3 end
```

Listing 4.33: Konfiguration von Event Dispatcher.

Dabei muss kein vordefinierter Adapter verwendet werden, auch Objekte eigens definierter Klassen können hier übergeben werden. Auf diese Weise lassen sich beliebige Message Broker anbinden.

RabbitMQ unterscheidet die Konzepte *Exchanges*, *Queues* und *Consumers*[3]. *Exchanges* nehmen zu veröffentlichende Nachrichten entgegen und leiten diese zu den *Queues* weiter, die sich zuvor bei ihnen registriert haben (im Englischen spricht man dann von *bindings*). Es existieren verschiedene Arten von *Exchanges*, die auf Basis eines mit der Nachricht assoziierten *Routing keys* entscheiden, welche der bei ihnen registrierten *Queues* die Nachricht erhalten sollen. In diesem Fall werden allerdings nur sogenannte *Fanout Exchanges* benötigt, die eingehende Nachrichten an jede bei ihnen registrierte *Queue* weiterleiten, ohne den *Routing key* zu betrachten. *Queues* sammeln alle eingehenden Nachrichten, bis diese von einem *Consumer* abgeholt werden. Eine *Queue* kann mehrere *Consumer* haben. Ist dies der Fall, werden die Nachrichten auf die vorhandenen *Consumer* aufgeteilt. *Queues* und *Exchanges* werden durch einen Namen

[3] Weitere Informationen unter: http://www.rabbitmq.com/tutorials/amqp-concepts.html

eindeutig identifiziert.

Beim Aufruf von `subscribe` des RabbitMQ-Adapters wird ein *Exchange* mit dem geg. Ereignisnamen angelegt oder abgerufen, falls er schon existiert. In analoger Weise wird für die aufrufende Klasse eine *Queue* angelegt oder abgerufen, die denselben Namen trägt wie die Klasse. Diese wird dann beim *Exchange* registriert. Schließlich wird der neben dem Ereignisnamen übergebene Block als *Consumer* für die *Queue* registriert.

Beim Aufruf von `publish` wird der neben dem Ereignisnamen übergebene Hash, der weitere Informationen zum Ereignis enthält, im JSON-Format serialisiert und an den *Exchange* für den Ereignisnamen übergeben. Beim Empfang der Nachricht werden die Informationen zum Ereignis dann wieder deserialisiert und in Form eines Objekts an den beim Aufruf von `subscribe` übergebenen Block übergeben. Listing 4.24 aus dem vorigen Abschnitt zeigt ein Beispiel für die Übergabe und Verwendung des Objekts.
Abbildung 4.1 zeigt den Lebenslauf einer Nachricht am Beispiel des Domain Events für den Verkauf eines Artikels (`"article_sold"`), das vom Domain Object `Domain::Model::Article` veröffentlicht wird, wenn ein Artikel verkauft wurde. In diesem Beispiel haben sich die Services `Service::OfferService` und `Service::ArticleService` für dieses Domain Event registriert, weshalb für jede der beiden Klassen eine separate *Queue* existiert. Die *Consumer* entsprechen dem der `subscribe`-Methode übergebenen Block und werden in der Abbildung „Handler“ genannt.

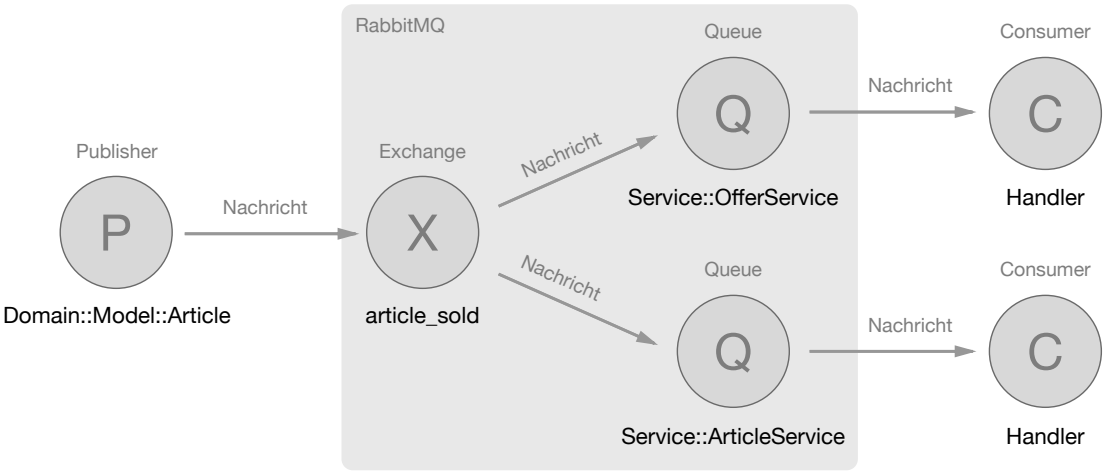

Abbildung 4.1: Lebenslauf einer Nachricht.

4.2.2 Garantierte Zustellung von Domain Events

Da Domain Events größtenteils zur Konsistenzwahrung eingesetzt werden, kann es mitunter dramatische Folgen haben, wenn veröffentlichte Domain Events verloren gehen. Dies kann durch eine Reihe von Umständen geschehen. Bei der bisher gezeigten Lösung wäre es beispielsweise denkbar, dass der RabbitMQ-Server aus nebensächlichen Gründen nicht aktiv ist und so bei dem Versuch der Veröffentlichung der Nachrichten eine Exception geworfen wird. Dieser kann zwar in der Anwendung abgefangen werden, jedoch ist zu diesem Zeitpunkt die vorhergehende Transaktion bereits festgeschrieben.
Würde die Veröffentlichung der Domain Events innerhalb der Transaktion geschehen, könnte zwar das Festschreiben der Transaktion bei Auftreten eines Fehlers verhindert werden, jedoch würden die Domain Events dann Änderungen kommunizieren, die noch gar nicht geschehen sind und evtl. auch nie geschehen werden, sollte die Transaktion aus anderen Gründen fehlschlagen.
Dies ist zwar ein klassischer Anwendungsfall für verteilte Transaktionen, spezieller *Zwei-Phasen-Commits*[4]. Allerdings bieten weder Ruby noch Rails vorhandene Lösungen für die Durchführung von verteilten Transaktionen. Eine andere Möglichkeit ist die Verwendung eines sogenannten *Event Stores*. Dabei werden die zu veröffentlichenden Domain Events innerhalb derselben Transaktion persistiert, in der die Änderungen, die die Domain Events kommunizieren persistiert werden. Nach Abschluss der Transaktion werden die Events veröffentlicht und anschließend als veröffentlicht markiert [vgl. Ver13, S. 304].
Sind in einem Fehlerfall nicht alle Domain Events veröffentlicht worden, werden die unveröffentlichten Domain Events von einem externen Dienst, der in regelmäßigen Abständen auf unveröffentlichte Domain Events prüft, veröffentlicht und entsprechend markiert. So kann es allerdings passieren, dass beim Auftreten eines Fehlers *nach* der Veröffentlichung des Domain Events, aber *bevor* das Domain Event als veröffentlicht markiert wurde, ein veröffentlichtes Domain Event nicht als solches markiert wurde. Diese Lösung bietet also *at-least-once delivery*, das heißt die Nachricht, die das Domain Event enthält, wird *mindestens einmal* zugestellt, möglicherweise aber auch mehrmals. Alle Operationen, die auf ein Domain Event folgen, müssen also *idempotent*[5] sein.

Domain Events werden mithilfe der Klasse `EventStore` persistiert, dessen Implementierung im nächsten Abschnitt gezeigt wird. Der `ApplicationService` muss angepasst werden, so-

[4] Siehe https://msdn.microsoft.com/en-us/library/aa754091(v=bts.10).aspx.
[5] Eine Operation ist *idempotent*, wenn die Ergebnisse einer einmaligen Ausführung der Operation dieselben sind, wie nach mehrmaliger Ausführung [vgl. Ver13, S. 330].

dass Domain Events nicht im Speicher vorgehalten, sondern über den `EventStore` persistiert werden. Listing 4.34 zeigt die angepassten Methoden `handle_event` und `publish_events`.

```
module Service
  class ApplicationService
    def initialize
      DomainEventPublisher.instance.subscribe(self)
    end

    def handle_event(event_name, event_hash)
      EventStore.append(event_name, event_hash)
    end

    def publish_events
      EventStore.publish_events
    end
    ...
  end
end
```

Listing 4.34: Persistierung von Events mittels der Klasse `EventStore`.

Zwar wird nun garantiert, dass Domain Events veröffentlich werden, jedoch nicht, dass auch deren Verarbeitung erfolgreich ist. Dafür bietet RabbitMQ sog. *message acknowledgements*[6] an. Dabei werden Nachrichten erst dann aus der *Queue* entfernt (und damit nicht erneut zugestellt), wenn die Nachricht explizit als erfolgreich verarbeitet markiert wurde. Das Gem „Event Dispatcher“ nutzt diesen Mechanismus: Ist eine Nachricht von einem Abonnenten vollständig und ohne Exceptions verarbeitet worden, wird die Nachricht bestätigt (engl. *acknowledged*). Wurde die Nachricht nicht bestätigt, wird sie von RabbitMQ zu einem späteren Zeitpunkt erneut zugestellt. Auch dieser Umstand erfordert idempotente Operationen für den Fall, dass eine Nachricht zwar erfolgreich verarbeitet wurde, aber eine Bestätigung aufgrund eines Fehlers nicht mehr stattgefunden hat.

4.3 Persistenz

Um Domain Objects und Aggregate zu persistieren existieren verschiedene Möglichkeiten. In diesem Abschnitt wird zunächst eine Lösung erläutert, die die bestehenden Model-Klassen

[6] Siehe https://www.rabbitmq.com/reliability.html.

und somit ActiveRecord verwendet. Diese Lösung bringt allerdings einige Probleme mit sich, weshalb die Implementierung von Repositories, der *Unit of Work* und dem *Event Store* anhand von einer alternativen Lösung gezeigt wird, die PostgreSQL und das JSON-Format verwendet.

4.3.1 Verwendung von ActiveRecord

Bei diesem Ansatz müssen die Attribute von Entitäten auf die Attribute der entsprechenden Model-Klassen abgebildet werden. Zwar wäre es technisch möglich ActiveRecord in die Domain Objects einzubinden, jedoch würde das die Datenhaltungsschicht mit der Domänenschicht vermischen und stellt somit keine Alternative dar. Die Abbildung der Attribute kann entweder manuell oder automatisiert geschehen. Listing 4.35 zeigt eine automatisierte Lösung, die funktioniert, solange die Attribute auf beiden Seiten den gleichen Bezeichner haben.

```
1  class Mapper
2    def self.map_attributes(domain_object, export_object)
3      export_object.attributes.each do |key, _|
4        if domain_object.attributes.has_key?(key)
5          export_object[key] = domain_object[key]
6        end
7      end
8
9      export_object
10   end
11 end
```

Listing 4.35: Abbildung von Attributen des Domain Objects auf Attribute der Model-Klasse.

Diese Lösung geht davon aus, dass jedes Domain Object die Methode `attributes` implementiert, die alle Attribute inklusive ihrer Werte als Hash zurückgibt (bei Verwendung von Virtus ist dies der Fall). In Zeile 3 wird über diese Attribute iteriert und in der folgenden Zeile für jedes geprüft, ob das Model (`export_object`) dasselbe Attribut besitzt. Ist dies der Fall wird in Zeile 5 der Wert des Attributs vom Domain Object in das gleichnamige Attribut des Models kopiert. In Zeile 9 wird das fertige Model zurückgegeben. Listing 4.36 zeigt ein Beispiel für die Verwendung der Klasse. In einer realen Anwendung würden Repositories die Klasse `Mapper` verwenden.

```
1 user = Domain::Model::User.new({:email => "user@example.com", :age => 34})
2 user_model = Mapper.map_attributes(user, User.new)
3 user_model.save!
```

Listing 4.36: Verwendung der Mapper-Klasse zur Persistierung von Domain Objects.

In diesem Beispiel befindet sich die Model-Klasse `User` im globalen Namespace, was in Rails-Anwendungen üblicherweise der Fall ist.

Während ein automatisierter Ansatz bei der Persistenz einzelner Entitäten noch recht simpel erscheint, ist die Persistenz ganzer Aggregate wesentlich komplizierter: Die Aggregatwurzel verweist auf weitere Domain Objects, die wiederum selbst weitere Domain Objects beinhalten können (ad infinitum), welche ebenfalls persistiert werden müssen. Eine automatisierte Lösung für die Persistenz von Aggregaten muss also die referenzierten Domain Objects auf die jeweils korrekte Assoziation in der Model-Klasse abbilden und zudem rekursiv funktionieren. Die nachfolgenden Abschnitte schildern eine Lösung für dieses Problem. Zunächst wird jedoch erläutert, was bei der Persistenz von Value Objects zu beachten ist.

Persistierung von Value Objects

Das Nichtvorhandensein von Identität ist ein entscheidendes Merkmal von Value Objects. Bei der Persistierung in relationalen Datenbanken ist es aber häufig erforderlich eine separate Tabelle für die Value Objects zu verwenden, statt sie in der Tabelle der referenzierenden Entität unterzubringen. Dies ist unter anderem der Fall, wenn eine Entität eine Sammlung von Value Objects hält (zum Beispiel die *OfferItems* aus der Beispielanwendung). In der Datenbank haben sie eine Identität, in der Domänenschicht jedoch nicht.

Diesem *impedance mismatch*[7] lässt sich mit dem Einsatz eines sog. *Layer Superytpes* [siehe Fow02, S. 475] Abhilfe schaffen. Ein *Layer Supertype* ist eine einfache Basisklasse, die für alle Objekte einer Schicht benötigte Daten und Methoden enthält und von der alle Objekte dieser Schicht ableiten. Dieser *Layer Supertype* kann die für die Persistenz benötigten Informationen, wie zum Beispiel die Datenbank-Identität eines Value Objects, halten.

Listing 4.37 zeigt eine mögliche Basisklasse für alle Domänenobjekte. In Zeile 4 werden der

[7] *Object-relational impedance mismatch* beschreibt die Unverträglichkeit von relationalem und objektorientiertem Paradigma.

Getter und der Setter für das Attribut `:storage_id` erzeugt, das die Identität des Domain Objects in der Datenbank beinhalten soll.

```
module Domain
  module Support
    class DomainObject
      attr_accessor :storage_id
    end
  end
end
```

Listing 4.37: Basisklasse `DomainObject` in der Datei `domain/support/domain_object.rb`

Nun müssen noch die existierenden Klassen `Entity` und `ValueObject` von `DomainObject` erben, wie die Listings 4.38 und 4.39 zeigen.

```
module Domain
  module Support
    class Entity < DomainObject
      ...
      def id
        @storage_id
      end
    end
  end
end
```

Listing 4.38: Neuer Inhalt von `domain/support/entity.rb`.

Standardmäßig ist nun die Datenbank-Identität gleichgesetzt mit der der Entität. Falls dies nicht der Fall sein soll, müssen die betroffenen Domain Objects die Methode `id` überschreiben.

```
module Domain
  module Support
    class ValueObject < DomainObject
      include Virtus.value_object
    end
  end
end
```

Listing 4.39: Neuer Inhalt von `domain/support/value_object.rb`.

Abbildung der Assoziationen

Im Folgenden werden die verschiedenen Arten von Assoziationen erläutert und Möglichkeiten gezeigt, wie Domain Objects auf diese abgebildet werden können. In ActiveRecord existieren die folgenden Assoziationen, die in den Model-Klassen deklariert werden können:

belongs_to
: Eine 1:1-Beziehung, bei der sich der Fremdschlüssel in der Tabelle des referenzierenden Objekts befindet (siehe folgendes Beispiel).

has_one
: Eine 1:1-Beziehung, bei der sich der Fremdschlüssel in der Tabelle des referenzierten Objekts befindet.

has_many
: Eine 1:n-Beziehung, bei der sich die Fremdschlüssel in der Tabelle der referenzierten Objekte befinden.

has_and_belongs_to_many
: Eine n:m-Beziehung, bei der sich die Fremdschlüssel in einer separaten Tabelle (*Join-Tabelle*) befinden.

Hat das Schema die in den Tabellen 4.1 und 4.2 gezeigte Form, handelt es sich aus Sicht der Model-Klasse `Article` um eine `belongs_to`-Assoziation. Dabei kann jedem Artikel genau eine der vorhandenen Größen zugeordnet werden.

id	label	value
1	Klein	S
2	Groß	L

Tabelle 4.1: Tabelle *article_sizes*.

id	text	price	article_size_id
18	Schönes T-Shirt, guter Zustand.	2500	2
32	Eine kurze Hose.	500	1

Tabelle 4.2: Tabelle *articles*.

Die Model-Klasse `Article` muss dann die Anweisung `belongs_to :article_size` enthalten, um auf die dem Artikel zugewiesene Größe zugreifen zu können. Dies wird in Listing 4.40 gezeigt.

```
1 class Article < ActiveRecord::Base
2   belongs_to :article_size
3 end
4
5 article = Article.find(18) # Gibt Artikel mit der ID 18 zurück.
6 article.article_size.value #=> "L"
```

Listing 4.40: Deklaration und Verwendung der `belongs_to`-Assoziation.

Um das Domain Object auf das entsprechende Model abbilden zu können, muss der Name des Feldes, in dem der Fremdschlüssel gespeichert wird (in obiger Tabelle `article_size_id`), bekannt sein. Diese Information kann entweder manuell gesetzt, oder besser gemäß *Konvention vor Konfiguration* aus dem Namen der Klasse abgeleitet werden. Listing 4.41 zeigt eine mögliche Lösung dafür. Wurde das Attribut `foreign_key_field` gesetzt, wird es zurückgegeben (Zeile 8). Andernfalls wird der Name des Feldes abgeleitet (selbe Zeile), indem der Name der Klasse in „snake case“ konvertiert (aus `ArticleSize` wird `article_size`) und der String `"_id"` angehängt wird. Dazu werden die Methoden `underscore` und `class_name` verwendet, die hier nicht gezeigt werden.

```ruby
module Domain
  module Support
    class DomainObject
      attr_accessor :storage_id
      attr_writer :foreign_key_field

      def foreign_key_field
        @foreign_key_field || "#{underscore(class_name)}_id"
      end
      ... # Methoden 'underscore' und 'class_name' hier nicht gezeigt.
    end
  end
end
```

Listing 4.41: Ableitung des Feldnamens für den Fremdschlüssel in `DomainObject`.

Außerdem können die Model-Klassen zu den im Aggregat enthaltenen Domain Objects nicht übergeben werden, sodass auch diese aus dem Namen der jeweiligen Klasse abgeleitet werden müssen. Listing 4.42 zeigt die Klasse `DomainObject` mit einer Methode `export_class`, die dies bewerkstelligt.

```ruby
module Domain
  module Support
    class DomainObject
      attr_accessor :storage_id
      attr_writer :export_class, :foreign_key_field

      def export_class
        @export_class || Object.const_get(class_name)
      end

      def foreign_key_field
        @foreign_key_field || "#{underscore(class_name)}_id"
      end
      ... # Methoden 'underscore' und 'class_name' hier nicht gezeigt.
    end
  end
end
```

Listing 4.42: Ableitung der passenden ActiveRecord-Klasse in `DomainObject`.

Listing 4.43 zeigt die Methode `export` für die Abbildung von Domain Objects in Attributen auf `belongs_to`-Assoziationen von Model-Klassen.

```
1  class Mapper
2    def self.export(domain_object)
3      export_object = export_object_for(domain_object)
4      map_attributes(domain_object, export_object)
5      map_belongs_to(domain_object, export_object)
6      export_object.save!
7    end
8    ... # Die Methode 'map_attributes' bleibt, wie vorher gezeigt.
9  end
```

Listing 4.43: Die Methode `export` bildet Domain Objects in Attributen auf Assoziationen ab.

Die Methode `export_object_for` in Zeile 3 prüft, ob das gegebene Domain Object eine `storage_id` besitzt, also ob es bereits persistiert ist. Falls ja, gibt es das entsprechende ActiveRecord-Objekt zurück. Falls nicht, wird ein neues Model erzeugt. Listing 4.44 zeigt die Methode.

```
1  class Mapper
2    ...
3    def self.export_object_for(domain_object)
4      if domain_object.storage_id
5        domain_object.export_class.find(domain_object.storage_id)
6      else
7        domain_object.export_class.new
8      end
9    end
10   ...
11 end
```

Listing 4.44: Die Methode `export_object_for`.

Die Methode `map_belongs_to` nimmt die eigentliche Abbildung der Domain Objects in Attributen vor. Listing 4.45 zeigt die Methode.

```
1  class Mapper
2    ...
3    def self.map_belongs_to(domain_object, export_object)
4      domain_object.attributes.each do |_, val|
5        if val.is_a?(Domain::Support::DomainObject)\
6            && export_object.has_attribute?(val.foreign_key_field)
7          Mapper.export(val)
8          export_object[val.foreign_key_field] = val.storage_id
9        end
10     end
11   end
12   ...
13 end
```

Listing 4.45: Abbildung von Domain Objects in Attributen auf Assoziationen.

In Zeile 4 wird über alle Attribute des zu exportierenden Domain Objects iteriert. In Zeile 5 wird dann für jedes Domain Object geprüft, ob es sich bei dem Attribut ebenfalls um ein Domain Object handelt und ob die Model-Klasse ein Attribut mit dem Namen des Feldes für den Fremdschlüssel des Domain Objects besitzt. Ist beides der Fall, handelt es sich um eine `belongs_to`-Beziehung. Das Domain Object und (etwaige weitere enthaltene Domain Objects) werden in Zeile 7 exportiert. Anschließend wird in Zeile 8 der korrekte Fremdschlüssel gesetzt.

Da der Aufruf der Methode `export` nun eine Rekursion zur Folge hat, reicht es nicht mehr das mit den Attributen des Domain Objects bestückte Model zurückzugeben, da sonst die enthaltenen Domain Objects nicht persistiert würden. Also muss dies am Ende jedes Aufrufs von `export` geschehen. Listing 4.46 zeigt ein Beispiel für die Verwendung der veränderten Mapper-Klasse.

```
1 article = Domain::Model::Article.new(
2   :text => "Eine schöne Hose.",
3   :price => 500,
4   :article_size => { :label => "Klein", :value => "S" }
5 )
6 Mapper.export(article)
```

Listing 4.46: Abbildung von Domain Objects in Attributen auf `belongs_to`-Assoziationen.

Zwar lässt sich diese Lösung auch für die anderen Assoziationsarten fortsetzen, jedoch haben sich bis zu diesem Punkt einige schwerwiegende Probleme gezeigt:

- Die gezeigte Lösung funktioniert nur solange die Klassennamen und Attribute der Domain Objects und der Model-Klassen sich gleichen. Ist dies nicht der Fall, müssen die Namen der Klassen und Attribute und ihre ActiveRecord-Äquivalente aufwendig konfiguriert werden.
- Die gezeigte Lösung behandelt bisher nur das Speichern von Domain Objects. Ein ähnlich großer Aufwand muss betrieben werden, um gespeicherte Domain Objects mittels ActiveRecord wieder aus der Datenbank abzurufen.
- Bei hohem Aufwand für die Übersetzung zwischen Domänen- und Datenhaltungsschicht bietet diese Lösung kaum Vorteile. Da sich keine Geschäftslogik mehr in den Model-Klassen befindet, enthalten sie bloß noch die Informationen über ihre Assoziationen, die allerdings gar nicht benötigt werden, da sie bereits aus den Domain Objects ersichtlich sind.

Aufgrund dieser Probleme wird dieser Ansatz nicht weiter verfolgt.

4.3.2 Verwendung von PostgreSQL mit JSON

Eine andere Möglichkeit ist die Verwendung eines JSON-basierten Datenspeichers. Domain Objects müssen ohnehin meist in das JSON-Format serialisierbar sein, um sie über APIs zur Verfügung stellen zu können. Diese Gegegebenheit lässt sich auch für die Persistierung von Domain Objects nutzen. In seinem Artikel „The Ideal Domain-Driven Design Aggregate Store?“ [Ver14] befürwortet Vaughn Vernon die Verwendung von PostgreSQL in Version 9.4 für die Persistierung von Domain Objects im JSON-Format, statt einer dokumentenorientierten Datenbank, wie zum Beispiel MongoDB. Als Gründe dafür nennt er unter anderem:

- PostgreSQL unterstützt seit Version 9.4 zwei Datentypen für JSON: Einen textbasierten und einen binären. Letzterer bietet eine höhere Performanz für Leseoperationen.
- Die JSON-Datentypen können mittels einer eigens für diesen Zweck eingeführten Syntax durchsucht und abgerufen werden.
- Viele dokumentenorientierte Datenbanken unterstützen keine ACID-Transaktionen[8], die allerdings benötigt werden, um Domain Events gemeinsam mit den dazugehörigen Domain Objects zuverlässig persistieren zu können.

[8] Siehe https://msdn.microsoft.com/en-us/library/aa480356.aspx.

- PostgreSQL ist ein reifes und performantes Open-Source-Produkt, das von vielen Plattformen unterstützt wird.

Zudem müssen die Attribute der Domain Objects nicht auf das Datenbankschema abgebildet werden, sondern können nach der Serialisierung einfach festgeschrieben werden. Bei diesem Ansatz existiert also kein klassisches relationales Datenbankschema, bei dem jedes Attribut einem Feld in einer Tabelle zugeordnet wird. Das Schema besteht hier lediglich aus Tabellen für die vorhandenen Aggregate mit je zwei Feldern: `id` als eindeutig identifizierendes Attribut und `data`, das die Attribute und die dazugehörigen Werte im JSON-Format enthält. Tabelle 4.3 zeigt die Tabelle `articles` mit Beispieldaten.

id	data
10	{“text“: “Schöne Hose“, “price“: 100, “size“: { ...
22	{“text“: “Ein T-Shirt“, “price“: 450, “size“: { ...

Tabelle 4.3: Tabelle *articles* (Inhalte von `data` abgekürzt).

Listing 4.47 zeigt die SQL-Anweisung zur Erstellung der Tabelle.

```
1 CREATE TABLE articles
2 (
3     id serial primary key,
4     data jsonb not null
5 );
```

Listing 4.47: SQL-Anweisung zur Erstellung der Tabelle `articles`

Connection Pool

Da ActiveRecord bei diesem Ansatz nicht verwendet wird, müssen die Datenbankverbindungen anderweitig verwaltet werden. Das Gem „pg“[9] bietet eine Ruby-Schnittstelle für PostgreSQL und wird auch von ActiveRecord zu diesem Zweck verwendet. Das Gem „connection_pool“ bietet eine generische Implementierung eines *Connection Pools*[10]. Listing 4.48 zeigt den Inhalt des Initializers `config/initializers/connection_pool.rb`, der den Verbindungsaufbau mithilfe von „pg“ übernimmt und bestehende Verbindungen bei Bedarf zur Verfügung stellt.

[9] Dokumenation unter: http://deveiate.org/code/pg/

[10] Ein Connection Pool ist eine Menge von bestehenden Verbindungen zu externen Diensten, wie z.B. Datenbanken, die bei Bedarf zur Verwendung freigegeben werden.

Dafür werden die in Rails konfigurierten Parameter aus der Datei `config/database.yml`, auf die über `Rails.configuration.database_configuration` zugegriffen werden kann, verwendet.

```
require 'pg'
require 'connection_pool'
ConnectionManager = ConnectionPool.new do
  config = Rails.configuration.database_configuration
  PG::Connection.new(
    :host     => config[Rails.env]["host"],
    :dbname   => config[Rails.env]["database"],
    :user     => config[Rails.env]["username"],
    :password => config[Rails.env]["password"]
  )
end
```

Listing 4.48: Der Initializer `config/initializers/connection_pool.rb`.

Der *Connection Pool* steht nun über die globale Variable `ConnectionManager` in der gesamten Anwendung zur Verfügung.

Unit of Work

Die Durchführung von Transaktionen wird mit einer sogenannten „Unit of Work“ realisiert. Eine „Unit of Work“ sammelt alle relevanten Änderungen, die während eines Geschäftsvorgangs[11] geschehen und die Datenbank betreffen können [vgl. Fow02, S. 184]. Am Ende des Geschäftsvorgangs werden die Änderungen in die Datenbank geschrieben. Die in Abschnitt 4.1.4 vorgestellte Basisklasse `ApplicationService` verwendet die Klasse `UnitOfWork`, um die Methode `transaction` zu implementieren. Mit dem Aufruf der Methode `begin` auf einer Instanz der Klasse wird der Beginn eines Geschäftsvorgangs gekennzeichnet. Anschließend können durch den Aufruf der Methode `write` neue Änderungen (in Form von SQL-Anweisungen) für die aktuelle Transaktion hinzugefügt werden. Diese Aufgabe übernehmen die Repositories. Beim Aufruf der Methode `commit` nach Abschluss des Geschäftsvorgangs durch die Basisklasse `ApplicationService` werden die Änderungen innerhalb einer Transaktion festgeschrieben. Jeder Thread besitzt eine eigene Instanz der Klasse, um die Überschneidung von zusammenhangslosen Änderungen zu verhindern. Abbildung 4.2 zeigt den Ablauf eines Geschäftsvorgang am Beispiel der Erstellung eines Artikels. *Connection* ist ein Objekt aus

[11] *Geschäftsvorgang* (engl. *business transaction*) meint hier jeden für die Domäne relevanten Vorgang.

dem Gem „pg“, das die Datenbankverbindung verwaltet und als Schnittstelle zur Datenbank fungiert.

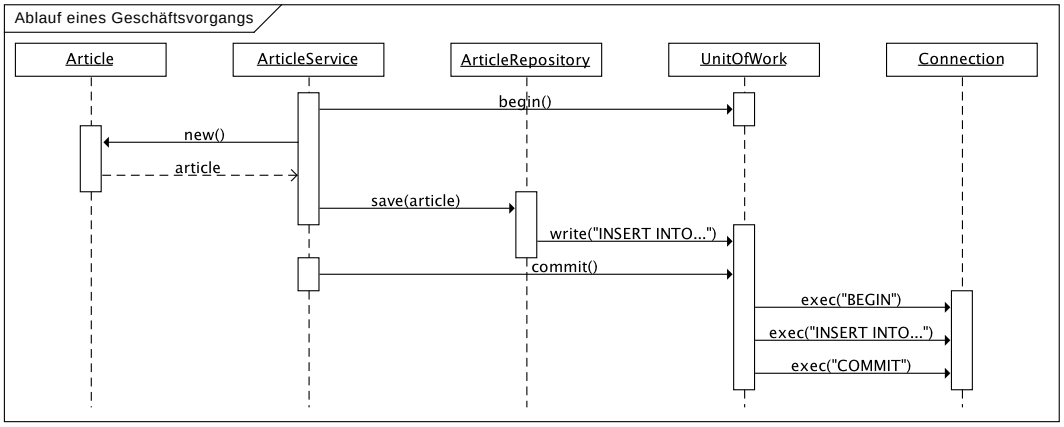

Abbildung 4.2: Sequenzdiagramm für den Ablauf eines Geschäftsvorgangs.

Lesende Zugriffe werden sofort ausgeführt und nicht gesammelt, da sie nicht innerhalb einer Transaktion passieren sollen und die Ergebnisse sofort benötigt werden. Dazu stellt die Klasse `UnitOfWork` die Methode `read` zur Verfügung, die genau wie `write` eine SQL-Anweisung akzeptiert und von Repositories genutzt wird. Listing 4.49 zeigt die Implementierung der Methode `read`, die als erstes Argument eine SQL-Anweisung akzeptiert, die zum Schutz vor maliziösen Anweisungen in übergebenen Werten (auch bekannt als *SQL Injection*) Platzhalter statt konkrete Werte enthält. Diese Platzhalter werden erst von PostgreSQL durch die konkreten Werte ersetzt, die als zweites Argument der Methode übergeben werden. Um beispielsweise die SQL-Anweisung `SELECT * FROM articles WHERE id = 4;` auszuführen, müsste die Methode `read` wie folgt aufgerufen werden:

```
UnitOfWork.instance.read("SELECT * FROM articles WHERE id = $1", [4])
```

```
1  module Infrastructure
2    module Persistence
3      class UnitOfWork
4        ...
5        def read(query, params = [])
6          ConnectionManager.with do |conn|
7            conn.exec_params(query, params)
8          end
9        end
10       ...
11     end
12   end
13 end
```

Listing 4.49: Die Methode `read` der Klasse `UnitOfWork`.

In Zeile 6 wird die Methode `ConnectionManager.with` aufgerufen, die den gegebenen Block mit einer verfügbaren Datenbankverbindung (`conn`) ausführt. In Zeile 7 findet dann die eigentliche Ausführung der Anweisung statt.

Listing 4.50 zeigt die Implementierung der Methode `write`, die wie `read` die SQL-Anweisung und die konkreten Werte als separate Argumente akzeptiert.

```
1  module Infrastructure
2    module Persistence
3      class UnitOfWork
4        ...
5        def write(query, params)
6          @write_batch << [query, params]
7        end
8        ...
9      end
10   end
11 end
```

Listing 4.50: Die Methode `write` der Klasse `UnitOfWork`.

Die auszuführenden Anweisungen werden im Array `@write_batch` gesammelt, der beim Aufruf von `begin` initialisiert wurde. Beim Aufruf der Methode `commit`, die in Listing 4.51 zu sehen ist, wird dann eine Transaktion gestartet und jede der gesammelten Anweisungen

ausgeführt. Nachdem die Transaktion ausgeführt wurde, wird das Array `@write_batch` durch den Aufruf der Methode `reset_write_batch` geleert.

```
module Infrastructure
  module Persistence
    class UnitOfWork
      ...
      def commit
        ConnectionManager.with do |conn|
          conn.transaction do |transaction|
            @write_batch.each do |statement|
              transaction.exec_params(statement[0], statement[1])
            end
          end
        end
        reset_write_batch
      end
      ...
    end
  end
end
```

Listing 4.51: Die Methode `commit` der Klasse `UnitOfWork`.

Implementierung von Repositories

Methoden zum Auffinden von Domain Objects anhand bestimmter Eigenschaften, oder zum Persistieren von geänderten Domain Objects werden von fast allen Repositories benötigt. Daher ist es sinnvoll diese Funktionalität in eine Basisklasse auszulagern. Listing 4.52 zeigt die Basisklasse `BaseRepository` und die Methode `find`, die eine ID akzeptiert und das dazugehörige Domain Object zurückliefert.

```ruby
module Infrastructure
  module Persistence
    class BaseRepository
      class RecordNotFoundError < StandardError; end
      def self.find(id)
        result = UnitOfWork.instance.read(find_sql, [id])
        if result.ntuples < 1
          raise RecordNotFoundError, "Record with id #{id} not found."
        end
        domain_object_from_result_hash(result.first)
      end
      ...
    end
  end
end
```

Listing 4.52: Die Basisklasse `BaseRepository` und die Methode `find`.

Dazu wird die Methode `find_sql` benutzt, die die SQL-Anfrage zum Auffinden des Domain Objects mit der gegebenen ID beinhaltet. Die SQL-Anfrage enthält Platzhalter, die bei der Anfrage durch PostgreSQL durch die ebenfalls übergebenen Werte ersetzt werden. Die Methode `find_sql` muss die Tabelle kennen, in der nach dem Domain Object gesucht wird, um die SQL-Anfrage korrekt zusammensetzen zu können. Da die Basisklasse nicht wissen kann, wann welche Tabelle zu verwenden ist, muss dieser Wert in den konkreten Implementierungen der Repositories gesetzt werden (zum Beispiel Tabelle *articles* für `ArticleRepository`). Listing 4.53 zeigt die Klassenmethoden `table_name` und `set_table_name`. Die Klassenmethode `set_table_name` kann dann von der konkreten Implementierung des Repositories genutzt werden, um den Namen zu verwenden Tabelle zu setzen.

```ruby
module Infrastructure
  module Persistence
    class BaseRepository
      ...
      def self.table_name
        @table_name
      end

      def self.set_table_name(table_name)
        @table_name = table_name
      end
    end
  end
end
```

Listing 4.53: Die Methoden `table_name` und `set_table_name` von `BaseRepository`.

Listing 4.54 zeigt die Verwendung der Methode im `ArticleRepository`.

```ruby
module Infrastructure
  module Persistence
    class ArticleRepository < BaseRepository
      set_table_name 'articles'
      ...
    end
  end
end
```

Listing 4.54: Setzen des Tabellennamens in der Klasse `ArticleRepository`.

Listing 4.55 zeigt die Methode `find_sql` der Basisklasse `BaseRepository`, die die Methode `table_name` verwendet, um die korrekte Tabelle für die SQL-Anfrage zu erhalten.

```ruby
module Infrastructure
  module Persistence
    class BaseRepository
      ...
      protected def self.find_sql
        "SELECT * FROM #{self.class.table_name} WHERE id = $1"
      end
    end
  end
end
```

Listing 4.55: Die Methode `find_sql`.

Auch die Methode `domain_object_from_result_hash` wird in der Methode `find` aus Listing 4.52 verwendet. Sie erstellt aus dem übergebenen Hash, der die Attribute des Domain Objects enthält, ein vollwertiges Domain Object, das dann zurückgegeben wird. Dafür muss bekannt sein, welchen Typ das Domain Object haben soll. Analaog zu den Klassenmethoden `table_name` und `set_table_name` existieren `domain_class` und `set_domain_class`, die den zurückzugebenden Typ festlegen. Listing 4.56 zeigt die Implementierung der Methode `domain_object_from_result_hash`.

```ruby
module Infrastructure
  module Persistence
    class BaseRepository
      ...
      protected def self.domain_object_from_result_hash(result)
        data = JSON.parse(result['data'])
        attributes = data.merge({:id => result['id']})
        self.class.domain_class.new(attributes)
      end
    end
  end
end
```

Listing 4.56: Die Methode `domain_object_from_result_hash`.

Um Domain Objects anhand bestimmter Eigenschaften ohne ID aufzufinden, kann die Methode `find_matching` verwendet werden. Sie akzeptiert einen Hash, der Attribute und die dazugehörigen Werte des gesuchten Domain Objects enthält. Um beispielsweise einen Artikel

mit einem Preis von 10,- € in der Größe XS aufzufinden, müsste der Methodenaufruf aussehen, wie in Listing 4.57 gezeigt.

```
Infrastructure::Persistence::ArticleRepository.find_matching({
  :price => 1000,
  :size => { :value => "XS" }
})
```

Listing 4.57: Verwendung der Methode `find_matching`.

Zurückgegeben wird ein Array, der alle Artikel enthält, auf die die angegebenen Kriterien zutreffen. Da sich mit PostgreSQL auch Felder, die Daten im JSON-Format enthalten, durchsuchen lassen, lässt sich diese Funktionalität leicht implementieren. Listing 4.58 zeigt die Methode `find_matching` der Klasse `BaseRepository`.

```
module Infrastructure
  module Persistence
    class BaseRepository
      ...
      def self.find_matching(criteria)
        result = UnitOfWork.instance.read(
          find_matching_sql,
          [criteria.to_json]
        )
        result.map { |res| domain_object_from_result_hash(res) }
      end
    end
  end
end
```

Listing 4.58: Implementierung der Methode `find_matching`.

Der übergebene Hash wird als JSON-String an die `read`-Methode gemeinsam mit der SQL-Anweisung aus der Methode `find_matching_sql` übergeben. Anschließend werden in Zeile 10 aus den erhaltenen Attributen der Domain Objects, auf die die Kriterien zutreffen wieder Domain Objects erstellt und in einem Array zurückgegeben. Listing 4.59 zeigt die Methode `find_matching_sql`.

```ruby
module Infrastructure
  module Persistence
    class BaseRepository
      ...
      protected def self.find_matching_sql
        "SELECT * FROM #{self.class.table_name} WHERE data @> $1::jsonb"
      end
    end
  end
end
```

Listing 4.59: Die Methode `find_matching_sql`.

Der Operator `@>` prüft, ob der rechte Wert im linken Wert enthalten ist. Der Ausdruck `$1::jsonb` wandelt den für den ersten Platzhalter übergebenen Wert in binäres JSON um.

Um Domain Objects zu persistieren wird die Methode `save` verwendet. Sie akzeptiert das zu persistierende Domain Object als Argument. Listing 4.60 zeigt die Methode.

```ruby
module Infrastructure
  module Persistence
    class BaseRepository
      ...
      def self.save(domain_object)
        data = domain_object.attributes
        id = data.delete(:id)
        if id
          update_record(id, data.to_json)
        else
          insert_record(data.to_json)
        end
      end
    end
  end
end
```

Listing 4.60: Die Methode `save`.

Nach dem Aufruf der Methode `attributes` auf dem Domain Object in Zeile 6 enthält die Variable `data` einen Hash mit allen Attributen des Domain Objects. Aus diesem wird an-

schließend die ID extrahiert, da sie nicht mit den anderen Attributen im Tabellenfeld `data`, sondern getrennt gespeichert wird. Ist keine ID vorhanden, nimmt die Variable `id` den Wert `nil` an. Wenn die extrahierte ID vorhanden ist (Bedingung in Zeile 8), existiert das Domain Object bereits in der Datenbank und der Datensatz muss *aktualisiert* werden (Zeile 9). Ist die extrahierte ID `nil`, existiert das Domain Object noch nicht in der Datenbank und muss neu *erstellt* werden (Zeile 11). Listing 4.61 zeigt die Methoden `update_record` und `update_sql` zur *Aktualisierung* eines Domain Objects.

```
1  module Infrastructure
2    module Persistence
3      class BaseRepository
4        ...
5        def self.update_record(id, data)
6          UnitOfWork.instance.write(update_sql, [id, data])
7        end
8
9        def self.update_sql
10         "UPDATE #{self.class.table_name} SET data = $2 WHERE id = $1"
11       end
12     end
13   end
14 end
```

Listing 4.61: Die Methoden `update_record` und `update_sql`.

Listing 4.62 zeigt die Methoden `insert_record` und `insert_sql` zur *Erstellung* eines Domain Objects.

```
module Infrastructure
  module Persistence
    class BaseRepository
      ...
      def self.insert_record(data)
        UnitOfWork.instance.write(insert_sql, [data])
      end

      def self.insert_sql
        "INSERT INTO #{self.class.table_name} (data) VALUES ($1)"
      end
    end
  end
end
```

Listing 4.62: Die Methoden `insert_record` und `insert_sql`.

Komplexe Anfragen, für die die `find_matching`-Methode nicht ausreichend ist, können in Methoden in den konkreten Repositories implementiert werden. Die in den vorigen Abschnitten verwendete Methode `find_contained_in_offer` der Klasse `ArticleRepository` soll alle Artikel zurückgeben, die in dem Angebot mit der gegebenen ID enthalten sind. Zwar könnte auch ein Service durch die Verwendung verschiedener Repositories die gewünschten Artikel zusammentragen, meist ist aber die direkte Befragung der Datenbank mit einer durchdachten Anfrage performanter. Listing 4.63 zeigt eine mögliche Implementierung der Methode.

```
module Infrastructure
  module Persistence
    class ArticleRepository
      ...
      def self.find_contained_in_offer(offer_id)
        UnitOfWork.instance.read(find_contained_in_offer_sql, [offer_id])
      end
    end
  end
end
```

Listing 4.63: Die Methode `find_contained_in_offer`.

Listing 4.64 zeigt eine SQL-Anfrage, die zum gewünschten Ergebnis führt. Die Anfrage, die sich über die Zeilen 2 und 3 erstreckt, liefert alle IDs von Artikeln zurück, die in dem Angebot mit

der gegebenen ID enthalten sind. Zu allen IDs, die diese Anfrage zurückgeliefert hat, werden anschließend in Zeile 1 die dazugehörigen Artikel gesucht.

```
1 SELECT * FROM articles WHERE id IN (
2  SELECT (jsonb_array_elements(data->'offer_items')->>'article_id')::serial
3  FROM offers WHERE id = $1
4 )
```

Listing 4.64: Die SQL-Anweisung in der Methode `find_contained_in_offer_sql`.

Optimistic Locking

Um Probleme durch nebenläufige Schreibzugriffe auf dieselben Daten zu verhindern, existiert auf Datenbankebene die Möglichkeit der Transaktionsisolation. Die strengste Isolationsebene *Serializable* garantiert, dass die Ausführung von Transaktionen in beliebiger Reihenfolge geschehen kann und jedes Mal dasselbe Ergebnis produziert wird[12]. Dies funktioniert aber nur solange Lese- und Schreibzugriffe innerhalb derselben Transaktion stattfinden. Bei Webanwendungen ist dies nur selten der Fall, da das Lesen eines Datensatzes und das Schreiben von Änderungen in der Regel zwei Anfragen und damit zwei Transaktionen erfordert:

1. Der Nutzer fragt einen Datensatz an, der auf dessen Endgerät dargestellt wird (erste Anfrage).

2. Der Nutzer verändert den Datensatz und speichert ihn anschließend (zweite Anfrage).

Die Datenbank hat somit keine Möglichkeit Lese- und Schreibzugriffe zuzuordnen und die Serialisierbarkeit der Transaktionen zu garantieren. Die gezeigte Implementierung von `UnitOfWork` reflektiert diese Tatsache und führt keine Lesezugriffe innerhalb einer Transaktion mit Schreibzugriffen aus.

Für Webanwendungen bietet sich statt Transaktionsisolation die Verwendung von *Optimistic Locking* an. Dabei erhält jeder Datensatz eine Versionsnummer. Diese wird gemeinsam mit dem Datensatz abgerufen und beim Festschreiben der Änderungen aktualisiert. Unterscheidet sich die Versionsnummer beim Festschreiben von der beim Abruf erhaltenen Versionsnummer, wurde der Datensatz zwischen Abruf und Festschreiben der Änderungen manipuliert. In diesem Fall ist es nicht sicher die eigenen Änderungen festzuschreiben und der Vorgang wird

[12] Siehe http://www.postgresql.org/docs/9.4/static/transaction-iso.html für weitere Information zu Transaktionsisolation.

abgebrochen.
Um *Optimistic Locking* zu implementieren erhält die Klasse `Entity` zunächst das Attribut `lock_version`, in dem die Versionsnummer gespeichert wird, wie Listing 4.65 zeigt.

```
1  module Domain
2    module Support
3      class Entity < DomainObject
4        ...
5        attribute :lock_version, Integer
6        ...
7      end
8    end
9  end
```

Listing 4.65: Das Attribut `lock_version` speichert die Versionsnummer des Domain Objects.

Listing 4.66 zeigt die veränderten Methoden `update_record` und `update_sql` der Klasse `BaseRepository`. Beim Festschreiben von Änderungen an bestehenden Domain Objects muss nun die Versionsnummer überprüft werden. Dies geschieht in Zeile 11.

```
1  module Infrastructure
2    module Persistence
3      class BaseRepository
4        ...
5        def self.update_record(id, data, lock_version)
6          UnitOfWork.instance.write(update_sql, [id, data, lock_version])
7        end
8
9        def self.update_sql
10         "UPDATE #{self.class.table_name} SET data = $2 \
11          WHERE id = $1 AND data->>'lock_version' = $3"
12       end
13       ...
14     end
15   end
16 end
```

Listing 4.66: Die Update-Methoden beinhalten nun die Versionsnummer.

Die Methode `update_record` erwartet nun die Versionsnummer zum Zeitpunkt des Abrufs des Domain Objects. Listing 4.67 zeigt die Methode `save` der Klasse `BaseRepository`, die

die bisherige Versionsnummer abfragt und an die Methode `update_record` übergibt (Zeilen 6 und 12). Anschließend setzt sie die neue Versionsnummer des Domain Objects auf den aktuellen Zeitstempel (Zeile 7). Der Rest der Methode bleibt unverändert.

```
1  module Infrastructure
2    module Persistence
3      class BaseRepository
4        ...
5        def self.save(domain_object)
6          lock_version = domain_object.lock_version
7          domain_object.lock_version = Time.now.to_i
8          data = domain_object.attributes
9          id = data.delete(:id)
10
11         if id
12           update_record(id, data.to_json, lock_version)
13         else
14           insert_record(data.to_json)
15         end
16       end
17       ...
18     end
19   end
20 end
```

Listing 4.67: Die Methode `save` aktualisiert die Versionsnummer.

Nun muss noch die Klasse `UnitOfWork` so modifiziert werden, dass ein Fehler geworfen wird, wenn das Festschreiben einer Änderung aufgrund einer veralteten Versionsnummer fehlschlägt. Listing 4.68 zeigt die veränderte Methode `commit`.

```
1  module Infrastructure
2    module Persistence
3      class UnitOfWork
4        ...
5        def commit
6          ConnectionManager.with do |conn|
7            conn.transaction do |transaction|
8              @write_batch.each do |statement|
9                res = transaction.exec_params(statement[0], statement[1])
10               if res.cmd_tuples == 0
11                 raise ConcurrentModificationError, "Failed to execute \
12                 '#{statement[0]}' with arguments '#{statement[1]}'."
13               end
14             end
15           end
16         end
17         reset_write_batch
18       end
19       ...
20     end
21   end
22 end
```

Listing 4.68: Die Methode `commit` wirft einen Fehler, falls eine Anweisung keinen Effekt hat.

In Zeile 10 wird nun überprüft, ob die Ausführung der zuletzt ausgeführten Anweisung einen Effekt hatte: Die Methode `cmd_tuples` gibt die Anzahl der von der letzten Anweisung betroffenen Datensätze zurück. Ist diese gleich 0, wird ein `ConcurrentModificationError` geworfen.

Implementierung des Event Stores

Der *Event Store* soll zu veröffentlichende Domain Events persistieren und verfolgen, welche erfolgreich veröffentlicht wurden und welche nicht. Die Klasse `EventStore` implementiert dazu die Methoden `append` zum Hinzufügen eines neuen Domain Events und `publish_events` zum Veröffentlichen von unveröffentlichten Domain Events. Beide lassen sich auf Methoden der Klasse `BaseRepository` zurückführen. Der Event Store ist daher lediglich ein spezielles Repository. Listing 4.69 zeigt die Klasse `EventStore`.

```
module Infrastructure
  module Persistence
    class EventStore < BaseRepository
      set_domain_class DomainEvent
      set_table_name 'domain_events'
    end
  end
end
```

Listing 4.69: Die Klasse `EventStore`.

Listing 4.70 zeigt die Klasse `DomainEvent`, die alle für Domain Events benötigten Attribute deklariert. Das Attribut `name` hält den Namen des Domain Events, während `data` die Informationen zum Ereignis speichert. Das Attribut `published` speichert die Information, ob das Domain Event bereits veröffentlicht wurde.

```
module Infrastructure
  module Persistence
    class DomainEvent
      include Virtus.model
      attribute :id, Integer
      attribute :published, Boolean, :default => false
      attribute :name, String
      attribute :data, Hash
    end
  end
end
```

Listing 4.70: Die Klasse `DomainEvent`.

Listing 4.71 zeigt die Methode `append` der Klasse `EventStore`. Diese akzeptiert den Namen des Domain Events und die dazugehörigen Informationen in Form eines Hashs. Anschließend wird das Domain Event durch den Aufruf von `save` persistiert.

```ruby
module Infrastructure
  module Persistence
    class EventStore < BaseRepository
      ...
      def self.append(event_name, data)
        domain_event = DomainEvent.new(:name => event_name, :data => data)
        save(domain_event)
      end
    end
  end
end
```

Listing 4.71: Die Methode `append` der Klasse `EventStore`.

Listing 4.72 zeigt die Methode `publish_events`, die über alle unveröffentlichten Domain Events iteriert und sie veröffentlicht.

```ruby
module Infrastructure
  module Persistence
    class EventStore < BaseRepository
      ...
      def self.publish_events
        unpublished_events = find_matching(:published => false)
        UnitOfWork.instance.begin
        unpublished_events.each do |domain_event|
          EventDispatcher.publish(domain_event.name, domain_event.data)
          domain_event.published = true
          save(event)
        end
        UnitOfWork.instance.commit
      end
    end
  end
end
```

Listing 4.72: Die Methode `publish_events` der Klasse `EventStore`.

In Zeile 6 werden alle unveröffentlichten Domain Events abgerufen. Anschließend wird eine neue Transaktion initialisiert und über die unveröffentlichten Domain Events iteriert (Zeilen 7 und 8). In Zeile 9 wird das aktuell betrachtete Domain Event veröffentlicht und in den beiden

darauffolgenden Zeilen als veröffentlicht markiert und persistiert. Zuletzt wird die Transaktion in Zeile 13 festgeschrieben.

Die Klasse kann verwendet werden, um nach einem Absturz noch unveröffentlichte Domain Events zu veröffentlichen. Dazu kann in regelmäßigen Abständen die Methode `publish_events` ausgeführt werden.

5 Bewertung

Dass Domain-Driven Design grundsätzlich mit Ruby on Rails vereinbar ist, wurde in den vorigen Abschnitten gezeigt. Der Aufwand für die konsequente Umsetzung von Domain-Driven Design ist aber teils erheblich. Daher muss in jedem Projekt abgewogen werden, welche Konzepte von Domain-Driven Design der Anwendung zuträglich sind und welche mehr Kosten als Nutzen bedeuten.

5.1 Domain-Driven Design in der Beispielanwendung

Bei dem hier gewählten Beispielprojekt konnte durch die Einführung der Konzepte von Domain-Driven Design mehr Flexibilität bei der Wahl der Infrastruktur, eine losere Kopplung und dadurch eine bessere Testbarkeit der Anwendung erreicht werden. Alle im „Analyse“-Kapitel genannten Probleme sind somit behoben worden.

Die konsequente Befolgung der Maximen von Domain-Driven Design hatte aber auch die Einführung von Lösungen zur Folge, deren Nutzen zumindest zweifelhaft ist: Die Regel nur ein Aggregat pro Transaktion zu persistieren war einer der beiden Gründe für die Einführung eines Message Brokers, was Problemen durch Nebenläufigkeit vorbeugen sollte, wie etwa Fehlermeldungen oder langen Wartezeiten für Nutzer durch Locking. Ob diese Probleme bei hohem Anfrageaufkommen aber überhaupt auftreten, sollte im Vorhinein untersucht werden.

Ebenfalls sollte der Aufwand, der betrieben werden musste, um die Domänenschicht von allen anderen Teilen der Anwendung zu isolieren, nicht unterschätzt werden. Die Isolation hat nämlich nicht nur Vorteile: Anwendungen, die kaum Geschäftslogik enthalten, sondern hauptsächlich aus CRUD-Operationen[1] bestehen, profitieren kaum von einer dedizierten Domänenschicht. Diese enthält dann Objekte, die nur als Datenbehälter fungieren und kaum Verhalten aufweisen [vgl. „Anemic Domain Model“, Fow03]. Die Trennung dieser Objekte von der Datenhaltung hat dann keine Vorteile, aber der Übersetzungsaufwand zwischen den beiden

[1] *CRUD* beschreibt die elementaren Datenbankoperationen *Create*, *Read Update* und *Delete*.

Schichten bleibt. Wann sich die Einführung einer Domänenschicht lohnt, ist eine schwierig zu beantwortende Frage. Fowler empfiehlt den Einsatz unter folgenden Umständen:
„*If you have complicated and everchanging business rules involving validation, calculations, and derivations, chances are that you'll want an object model to handle them.*“ [Fow02, S. 119]
Im Falle der Beispielanwendung trifft das nur teilweise zu: Insbesondere die in den vorigen Abschnitten gezeigte Verwaltung der Zustände erfüllt zwar diese Bedingungen, aber Anwendungsfälle, wie zum Beispiel das Favorisieren eines Artikels enthalten wenig bis keine Geschäftslogik. Für die Beispielanwendung ist dann das voraussichtliche Wachstum relevant: Steigt die Komplexität in Zukunft, ist eine Domänenschicht eher sinnvoll. Stagniert das Wachstum, ist sie es eher nicht.

Der Einsatz von Domain-Driven Design und allen Konzepten, Entwurfsmustern und Prinzipien, die dazugehören, ist also nicht immer sinnvoll, sondern lohnt sich nur bei Anwendungen mit ausreichend komplexen Domänen und den entsprechenden Problemen.
Zu einem ähnlichen Schluss kommt auch Microsoft:
„*In order to help maintain the model [gemeint ist das Domänenmodell, Anm. d. Verf.] as a pure and helpful language construct, you must typically implement a great deal of isolation and encapsulation within the domain model. Consequently, a system based on Domain Driven Design can come at a relatively high cost.*“ [Mic09, S. 25]
Und später:
„*DDD can [...] be an ideal approach if you have large and complex enterprise data scenarios that are difficult to manage using other techniques.*“ [Mic09, S. 26]

Dies bedeutet aber nicht, dass bei weniger komplexen Domänen komplett auf Domain-Driven Design verzichtet werden muss. Der folgende Abschnitt erläutert Möglichkeiten, ausgewählte Konzepte von Domain-Driven Design in Anwendungen zu verwenden, deren Domäne nicht komplex genug ist, um eine dedizierte Domänenschicht zu rechtfertigen.

5.2 Anwendungen ohne Domänenschicht

Wie bereits erwähnt, ist eine dedizierte Domänenschicht nur bei einer ausreichend komplexen Domäne sinnvoll. Anwendungen, bei denen dies nicht zutrifft können getrost auf das Entwurfmuster Active Record zurückgreifen, da es kaum Geschäftslogik gibt, die mit persistenzbezogener Logik vermischt werden könnte [vgl. Fow02, S. 117]. Im Folgenden werden zwei

Konzepte erläutert, die auch ohne eine dedizierte Domänenschicht und in Verbindung mit dem Modul ActiveRecord, eingesetzt werden können.

5.2.1 Services

Die Trennung von Applikations- und Präsentationslogik durch Services ist generell von Vorteil. Selbst wenn zu Beginn des Projekts feststeht, dass es nur eine Web-Schnittstelle zur Anwendung geben wird: Im Falle von Rails kommt es des Öfteren zu Veränderungen am Framework, zum Beispiel in der Art, wie auf HTTP-Parameter zugegriffen werden kann. Diese Änderungen haben keinen Einfluss auf die Implementierung der Anwendungsfälle. Befindet sich aber die Applikationslogik in den Controllern, muss beides angepasst werden. Eine Trennung hingegen hat keine Nachteile.
Dabei werden die Services wie in den vorigen Abschnitten gezeigt von den Controllern verwendet. Die Services greifen dann wiederum auf die ActiveRecord-Models zu, um ihre Aufgabe zu erfüllen.

5.2.2 Entities und Value Objects

Die zugrundeliegende Idee der Unterscheidung zwischen Entities und Value Objects ist die Trennung von Daten, die sich ändern von jenen, die dies nicht tun. Unveränderliche Objekte sind von Natur aus threadsicher und leichter handhabbar, da es keine Seiteneffekte bei Operationen auf diesen geben kann. Die Unterscheidung von Entities und Value Objects ist also häufig sinnvoll.

ActiveRecord unterstützt die Verwendung von Value Objects durch die Methode `composed_of`. Diese bildet ausgewählte Attribute eines Models auf eine angegebene Klasse ab[2]. Listing 5.1 zeigt ein Beispiel in Form des Models `Article`, das das Attribut `price` auf das Attribut `amount` der Klasse `Price` abbildet.

```
1 class Article < ActiveRecord::Base
2   composed_of :price, class_name: "Price", mapping: ["balance", "amount"]
3 end
```

Listing 5.1: Beispiel für die Verwendung der Methode `composed_of`.

[2] Siehe http://api.rubyonrails.org/classes/ActiveRecord/Aggregations/ClassMethods.html für die vollständige Dokumentation.

Bei der Instanziierung des Models werden dann dem Initializer der Klasse `Price` die Attribute in der Reihenfolge der Übergabe an die `composed_of`-Methode übergeben. Das resultierende Objekt kann dann durch den Aufruf von `price` auf dem Model verwendet werden.

5.3 Ruby und Rails im Vergleich zu anderen Plattformen

Im Vergleich zu Plattformen, wie .NET oder Java EE, haben Ruby und Rails einige Nachteile hinsichtlich der Unterstützung von großen und komplexen Anwendungen. Diese werden im Folgenden genannt.

- Ruby und Rails bieten wenig Unterstützung für komplexere persistenzbezogene Aufgaben. Die Durchführung von verteilten Transaktionen wird überhaupt nicht unterstützt, während für Java dazu die *Java Transaction API*[3] und für .NET die *System.Transactions*[4] API zur Verfügung stehen.

- Bei komplexen Domänenmodellen stehen Rails und ActiveRecord aufgrund des Grundsatzes *Konvention vor Konfiguration* eher im Weg, weil das Datenbankschema selten dem Domänenmodell entspricht und damit die Konvention, dass die Namen von Attributen denen der Tabellenfelder gleichen, nicht einzuhalten ist. Für Java und .NET stehen mit Hibernate bzw. NHibernate mächtige ORM-Lösungen für die Abbildung der Domain Objects auf das Datenbankschema zur Verfügung.

- Bei Anwendungen mit hohem Anfrageaufkommen ist zu hinterfragen, ob Ruby die geeignete Wahl für die Implementierung ist. In verschiedenen Benchmarks schneidet beispielsweise Java deutlich besser ab, als Ruby[5].

Dies bedeutet aber nicht, dass Java EE oder .NET grundsätzlich die besser Wahl wären. Die erwähnten Nachteile zeigen lediglich, dass beide Plattformen gegenüber Ruby und Rails a priori mehr Unterstützung bei der Entwicklung von Anwendungen bieten, die komplexe Geschäftsprozesse abbilden sollen (sogenannte *Enterprise Applications* [siehe Fow02, S. 2]).
Eine Möglichkeit diese Nachteile zu kompensieren ist die Verwendung einer speziellen Ruby-Implementierung namens *JRuby*[6]. Diese implementiert die Programmiersprache Ruby auf Basis der *Java Virtual Machine* (kurz *JVM*). Dadurch kann mittels Ruby auf Bestandteile der Java-Plattform zurückgegriffen werden, inklusive der *Java Transaction API*.

[3] Siehe http://www.oracle.com/technetwork/java/javaee/jta/index.html.
[4] Siehe https://msdn.microsoft.com/en-us/library/system.transactions(v=vs.110).aspx.
[5] Siehe http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=java&lang2=yarv.
[6] Siehe http://jruby.org/.

6 Fazit und Ausblick

Domain-Driven Design kann helfen Rails-Anwendungen skalierbarer und flexibler zu gestalten. Dabei muss aber vorsichtig abgewogen werden, welche Teile von Domain-Driven Design der Anwendung wirklich zuträglich sind, da die Einführung und vor allem die Isolation der Domänenschicht vom Rest der Anwendung ein aufwendiges Unterfangen ist.

Eine dedizierte Domänenschicht lohnt sich meist nur bei komplexen Domänen und Anwendungen, die viel Geschäftslogik enthalten. Bei Anwendungen, auf die dies nicht zutrifft, können dennoch bestimmte Konzepte von Domain-Driven Design verwendet werden, um deren Qualität zu verbessern.

Insbesondere die Einführung eines asynchronen Kommunikationsmechanismus zur Implementierung von *Eventual Consistency* und dem Informationsaustausch mit anderen Anwendungen oder weiteren Teilen der Anwendung hatte weitreichende Auswirkungen auf die Komplexität der Anwendung und sollte sorgfältig erwogen werden.

Ist diese Infrastruktur aber erst einmal vorhanden, eröffnen sich neue Möglichkeiten für die Gestaltung der Architektur des Systems. Da ein Großteil der Kommunikation über Ereignisse funktioniert und die beteiligten Komponenten somit lose miteinander gekoppelt sind, können diese auch ohne Weiteres auf unterschiedlichen Rechnern laufen.
Auf diese Weise lassen sich zum Beispiel sogenannte *Microservices*[1] realisieren. Dabei wird eine Anwendung in mehrere kleinere Services aufgeteilt, die jeweils klar definierte Verantwortungsbereiche haben und miteinander kommunizieren, um gemeinsam verschiedene Aufgaben zu erfüllen.
Auch Techniken, wie *Event Sourcing*[2], bei der der Zustand der Anwendung als eine Serie von Ereignissen persistiert wird, lassen sich so umsetzen.

[1] Siehe http://martinfowler.com/articles/microservices.html.
[2] Siehe http://martinfowler.com/eaaDev/EventSourcing.html.

Literaturverzeichnis

[Beh13] Behrends, Ehrhard: *Elementare Stochastik*. Erste Auflage. Springer Spektrum, 2013. – ISBN 978–3–834–81939–0

[Eva03] Evans, Eric: *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Erste Auflage. Addison-Wesley, 2003. – ISBN 978–0–321–12521–7

[Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Erste Auflage. Addison-Wesley, 2002. – ISBN 978–0–321–12742–6

[Fow03] Fowler, Martin: *Anemic Domain Model*. http://martinfowler.com/bliki/AnemicDomainModel.html. Version: November 2003, Abruf: 2015-06-16

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph E. ; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Erste Auflage. Addison-Wesley, 1994. – ISBN 978–0–201–63361–0

[Mar09] Martin, Robert C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Erste Auflage. Prentice Hall, 2009. – ISBN 978–0–13–235088–4

[Mic09] *Microsoft Application Architecture Guide*. Zweite Auflage. Microsoft Corporation, 2009. – ISBN 978–0–735–62710–9

[MT15] Millett, Scott ; Tune, Nick: *Patterns, Principles, and Practices of Domain-Driven Design*. Erste Auflage. Wrox, 2015. – ISBN 978–1–118–71470–6

[SH12] Sprenger, Stefan ; Hayes, Kieran: *Ruby on Rails 3.1 Expertenwissen*. Erste Auflage. dpunkt.verlag GmbH, 2012. – ISBN 978–3–898–64697–0

[Ver13] Vernon, Vaughn: *Implementing Domain-Driven Design*. Erste Auflage. Addison-Wesley, 2013. – ISBN 978–0–321–83457–7

[Ver14] Vernon, Vaughn: *The Ideal Domain-Driven Design Aggregate Store?* https://vaughnvernon.co/?p=942. Version: December 2014, Abruf: 2015-06-16

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 19. Oktober 2015 Maximilian H. Zender