UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

The role of annotation and XML in framework usage

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von
Pascal Schuler

Erstgutachter: Ralf Lämmel
Institut für Informatik

Zweitgutachter: Andrei Varanovich
Institut für Informatik

Koblenz, im August 2014

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Abstract

In the context of this thesis the system components of annotations and XML will be discussed for their benefits within certain frameworks. This includes both, a analysis of previously existing possibilities in the use of these system components, and deeper knowledge of the internal processing of each individual components. A deeper approach leads to a better understanding of the internal processes of annotations and XML and promote interesting facts in the area of use.

Zusammenfassung

Im Rahmen dieser Arbeit sollen die Systemkomponenten der Annotationen und XML auf ihren Nutzen innerhalb von Frameworks untersucht werden. Dazu gehört neben einer Analyse der bisher bestehenden Möglichkeiten in der Nutzung dieser Systemkomponenten, auch fachspezifisches Hintergrundwissen zu den einzelnen Komponenten aufzubauen. Eine tiefere Betrachtungsweise soll ein besseres Verständnis der Verarbeitung von Annotationen und XML und wissenswertes im Bereich der Nutzung fördern.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Meist wird in der Literatur zwar eine große Masse an Wissen und Informationen übermittelt, allerdings begrenzt sich dieses Wissen jedoch meist auf das oberflächliche Verständnis von Abläufen und der Ausführung von einfachen Beispielen. Diese Arbeit soll tiefer in die Marterie der Verarbeitung und Hintergrundabläufe, sowie in die Anwendung eindringen.

Motivation Annotationen sind eine Neuerung, eingeführt im Jahre 2004 mit Version 5, der Programmiersprache Java. Diese ermöglichte es dem Entwickler externe Dateien zu vermeiden und die gewünschten Daten direkt an der Stelle der Nutzung einzufügen. Doch wo genau liegen die Unterschiede in Verarbeitung von Annotationen gegenüber der Verwendung von externen XML-Dateien? Existieren vielleicht sogar Äquivalenzen in der Verarbeitung von XML und Annotationen? Dazu werden Funktionen der ausgewählten Frameworks auf ihre interne Verarbeitung überprüft und Äquivalenzen sowie Unterschiede systematisch dargestellt. Darüber hinaus sollen diese Funktionen auf ihre Mächtigkeit überprüft werden. Das bedeutet die Funktionalitäten von XML und Annotationen werden untereinander verglichen. Kann mit der Verwendung von Annotationen eine größere Menge an Funktionalität bereit gestellt werden oder ist gar die Nutzung von Annotationen eine abgespeckte Version der XML-Nutzung? Diese Fragen werden anhand einiger Funktionalitäten überprüft und versucht zu beantworten.

Neben der ausführlichen Analyse dieser Funktionalitäten, sollen auch Fragen in der Nutzung von XML und Annotationen beantwortet werden. Existieren Möglichkeiten XML und Annotationen innerhalb der Funktionalität zu vermischen? Diese Fragen sollen einem besseres Verständnis der Nutzung von XML und Annotationen innerhalb eines Projektes dienen.

Außerdem soll ein Überblick über die verschiedenen Verwendungsarten von XML und Annotationen gegeben werden. Können XML und Annotationen in anderen Bereichen des Frameworks genutzt werden, wo keine äquivalente Anwendungsart des jeweils Anderen möglich ist? Es soll dabei auch überprüft werden, ob neben XML und Annotationen noch andere Möglichkeiten existieren, die gewünschten Funktionen herzustellen.

Auswahlkriterien der Frameworks Eine wichtige Rolle in dieser Arbeit spielen Frameworks, an denen die Fragen des obigen Abschnitts überprüft werden sollen. Doch welche Gründe führten letztlich zur Auswahl der genutzten Frameworks. Ein wichtiges

Kriterium in der Auswahl war in dieser Arbeit die freie Zugänglichkeit des Frameworks. Alle hier genutzten Frameworks sind quelloffene Open-Source-Projekte. Dies ist insofern ein wichtiges Kriterium, da durch die freie Zugänglichkeit des Codes, jederzeit jeder Aspekt dieser Arbeit nachvollzogen werden kann. Darüber hinaus soll jedes Framework spezifische Eigenschaften in der Nutzung von XML und Annotationen bereitstellen. Hibernate bietet beispielsweise durch seine Erstveröffentlichung vor Einführung von Annotationen einen vollen Funktionsumfang an XML. Gerade der Unterschied zwischen einem „alten“ Framework, wie Hibernate, und einem relativ neuen Framework, wie Symfony2, kann dabei Erkenntnisse über die Entwicklung innerhalb der Nutzung beider Technologien bieten. Außerdem unterstützt sowohl Hibernate als auch Symfony2 das objektrelationale Mapping. So können auch dort Vergleiche zwischen Mapping in verschiedenen Systemen vorgenommen werden. Gerade die hohe Verbreitung solcher OR-Mapping-Tools spielte eine große Rolle in der Auswahl dieser Frameworks. Durch die Auswahl von Hibernate und Symfony2 kann außerdem eine plattformübergreifende Analyse stattfinden, da Hibernate für Java und Symfony2 für PHP das OR-Mapping bereit stellt.

Kapitel 2

Technische Grundlagen

Um den vollen Umfang der Funktionalität von Annotationen und XML innerhalb von Frameworks zu verstehen, müssen in diesem Kapitel Grundlagen gelegt werden. In diesem Kapitel handelt es sich ausschließlich um Beispiele aus der generellen Funktionsweise von Annotationen und XML in Java.

2.1 Annotationen in Java

Annotationen sind eine der größten Neuerungen, die durch die Einführung von Java 5 im September 2004 [Ull12], nutzbar gemacht wurden. Unter Annotationen versteht der Softwareentwickler Metadaten, die dem Programm zugewiesen und später an anderer Stelle im Code verwendet werden können. Sie wurden eingeführt um dem generellen Trend der Softwareentwicklung, Metadaten mit Source-Code zu kombinieren, Folge zu leisten. Zuvor wurden die Metadaten in externen XML-Dokumenten gelagert [Eck06].

Über Annotationen können zusätzliche Informationen über das Programm gespeichert werden und das in einem vom Compiler geprüften Format. Sie können beispielsweise verwendet werden um neue Klassendefinitionen zu generieren. Mit der Verwendung von Annotationen können Informationen innerhalb der Java-Dateien gehalten und damit sauberer Code geschrieben werden. Java 5 kommt mit einigen vordefinierten Annotationen, aber generell bestimmt der Softwareentwickler, wie er sie gewinnbringend einsetzt [Eck06]. Denn neben den bereits vordefinierten Annotationen der Java-Pakete, ermöglicht Java auch das Erstellen von eigenen Annotationen.
In den folgenden Abschnitten soll zunächst auf die Syntax von Annotationen und dann auf die vordefinierten Annotationen der Java-Pakete eingegangen werden.

2.1.1 Syntax von Annotationen

Die Syntax von Annotationen besteht hauptsächlich aus der Addition des Symbols '@' zu der Sprache [Ull12]. Insgesamt besteht eine Annotation aus dem Symbol @, gefolgt von einem Namen, genannt Annotationstyp. Während diese Reihenfolge fest ist, bietet beispielsweise die Reihenfolge innerhalb einer Methodensignatur mehr Spielraum.

```
Möglichkeit 1 :  @Override public String toString()
Möglichkeit 2 :  public @Override String toString()
```

Abbildung 2.1: Beispiel der möglichen Syntax[Ull12]

So können beide Schreibarten aus Abbildung 2.1 verwendet werden um die Annotation „@Override“ zu verwenden. Allerdings gilt Möglichkeit 1 als standardisierte Annotationssignatur und damit sollte die Annotation am Anfang stehen [Ull12].

Außerdem existiert die Möglichkeit Annotationen zusätzliche Informationen mitzugeben. Im Allgemeinen führt dies zu folgender Aufteilung der verschiedenen Annotationsarten:

Tabelle 2.1: Die verschiedenen Annotationsarten im Überblick

@Annotationstyp	(Marker-)Annotation
@Annotationstyp(Wert)	Annotation mit genau einem Wert
@Annotationstyp(Schlüssel1 = Wert1, Schlüssel2 = Wert2)	Annotation mit Schlüssel/Werte-Paaren

Neben der, in Tabelle 2.1 verwendeten, Schreibweise erlauben die Marker-Annotation noch das Anfügen von Klammern. Also kann sowohl „@Annotationstyp“ als auch „@Annotationstyp()“ verwendet werden, um eine Marker-Annotation zu initialisieren.

2.1.2 Vordefinierte Annotationen

In java.lang sind vier allgemein nutzbare Annotationen eingebaut:

- @Override, soll anzeigen, dass eine Methodendefinition in der Basisklasse überschrieben werden soll.
- @Deprecated, produziert eine Warnung, wenn das markierte Element genutzt wird. Besagt also, die markierte Funktion sollte nicht mehr genutzt werden.
- @SupressWarnings, stellt alle unangebrachten Compiler-Warnungen aus.

Die drei oberen Annotationen sind bereits seit der Einführung von Java 5 nutzbar. Die folgende wurde mit Einführung von Java 7 zum Softwarepaket java.lang hinzugefügt [Ull12].

- @SafeVarargs, markiert eine Funktion mit variabler Argumentzahl und generischem Argumentstyp[Ull12]

2.2 XML in Java

Die Extensible Markup Language, kurz XML, wird genutzt um Dokumente und Daten in einem standardisierten Format darzustellen. XML zählt daher zu dem Bereich der Auszeichnungssprachen [Ull12]. Auszeichnungssprachen legen die Struktur bestimmter Bausteine eines Dokumentes und deren Beziehungen zueinander fest. Dazu existiert zu jeder Sprache eine eigene Syntax. Neben XML, gehört auch die Standardized Generalized Markup Language, der Ursprung des XML, zu der Gattung der Auszeichnungssprachen [itwa].

Mitte der 80er Jahre wurde SGML als ISO-Standard[1] definiert. Allerdings musste ein SGML-Dokument einen bestimmten Aufbau gewährleisten und bat somit keine große Flexibilität, da der Aufbau eingehalten werden musste. An diesem Punkt setzte das W3C[2] an und entwickelte eine Auszeichnungssprache, die sowohl flexibel wie SGML und so einfach wie HTML sein sollte, nämlich XML. Die Einführung von XML und die damit eingeführten Vorgaben zur Wohlgeformtheit eines Dokumentes bilden die Grundlage zu einer einfacheren Verarbeitung durch den Compiler[Ull12].

2.2.1 Syntax von XML

Die Syntax eines XML-Dokumentes besteht aus einer strukturierten Schachtelung von Elementen und Texten. Programmauszug 2.1 zeigt drei verschiedene Arten von Elementen.

Programmauszug 2.1: party.xml

```
1 <?xml version="1.0"?>
2 <treffen datum="31.12.01">
3   <anwesender name="Max Mustermann">
4     <antrag>Änderung §1</antrag>
5     <antrag>Änderung §2</antrag>
6     <partei name="CDU"/>
7   </anwesender>
8 </treffen>
```

Zunächst die, in Zeile 1 definierte, Kopfdefinition des XML-Dokuments, die in minimaler Form lediglich die benutzte XML-Version angeben muss. Mit „encoding“ kann dort beispielsweise noch die Default-Zeichenkodierung des Dokumentes gesetzt werden. Die Kopfdefinition ist ein Spezialfall der allgemeinen XML-Elemente, sie besitzt kein öffnendes und schließendes Zeichen, wie in XML üblich.
Die am häufigsten verwendete Elementstruktur ist in Programmauszug 2.1 in Zeile 4/5 zu sehen. Der Inhalt des Elementes wird von jeweils einem öffnenden und einem schließenden Zeichen (Tag) umschlossen. Das Schließen eines Elementes ist dabei case-sensitiv. Innerhalb des öffnenden Tags können einem Element zusätzlich Attribute zugewiesen werden. Die Tags können in XML frei gewählt werden, da es dort, anders als bei HTML, keinen festgelegten Pool an Tags gibt.[Ull12]

[1] Ein von der „International Organization for Standardization“(ISO) eingeführtes Format zur Standardisierung von Dokumenten

[2] Hersteller von XML;voller Name: World Wide Web Consortium

Außerdem existieren noch Elemente, die keine Inhalte besitzen. Diese werden nach Angabe der Attribute mit dem Schrägstrich geschlossen und benötigen daher kein schließendes Tag.
Laut Definition muss ein XML-Dokument zwingend wohlgeformt sein. Im Gegenschluss heißt das, ist ein Dokument nicht wohlgeformt, ist es kein XML-Dokument. Unter Wohlgeformtheit wird im Bezug auf XML-Dokumente Folgendes verstanden[Ull12]:

- Alle geöffneten Tags werden wieder geschlossen (mit und ohne Inhalt)
- Geöffnete Tags müssen in umgekehrter Reihenfolge des Öffnens geschlossen werden
- Jedes XML-Dokument enthält ein Basiselement

2.2.2 API's für XML

Zur Verarbeitung von XML-basierenden Daten existieren 4 grundlegend verschiedene Verarbeitungstypen, die für verschiedene Anwendungen geeignet sind.

DOM-orientierte API's, wie JDOM („Java Document Object Model"), das speziell für Java entwickelt wurde, verarbeiten die komplette XML-Struktur im Speicher. Einsatzbereiche sind hier beispielsweise eine Sortierung von bestimmten Strukturen innerhalb eines XML-Dokumentes[Ull12].
Push-API's, wie SAX („Simple API for XML"), speichern das XML-Dokument nicht komplett im Speicher, sondern lösen für erkannte Elemente ein Ereignis zum Verarbeiten der Daten aus. So kann zwar Speicherkapazität gespart werden, allerdings nicht wahlfrei auf die XML-Elemente zugegriffen werden[Ull12].
Pull-API's, wie StAX („Streaming API for XML"), fordern aktiv den nächsten Teil eines XML-Dokumentes an und werden dann mit dem Iterator[1] oder dem Cursor[1] bearbeitet[Ull12].

Auf JAXB („Java Architecture for XML Binding") als sogenannte Mapping-API wird im kommenden Abschnitt eingegangen.
Das Festlegen auf einzelne Technologien kann durchaus nachteilig sein. Ändern sich während einem Projekt die Ansprüche, zum Beispiel von Speicherverbrauch zu Performance, kann dies zu größeren Problemen führen. Daher wurde eine API entworfen die über DOM, SAX und StAX abstrahiert, die JAXP („Java API for XML Parsing")[Ull12].

2.2.3 Annotation meets XML - JAXB

Die Java Architecture for XML Binding, kurz JAXB, ist eine API, die mithilfe von Annotationen, Objekte in XML und XML in Objekte umwandelt. Dies geschieht, im Gegensatz zu DOM und SAX nicht über die Parser sondern automatisch. Die Annotationen für JAXB liegen im Annotationspaket javax.xml.bind.annotation.

Programmauszug 2.2 zeigt die Verwendung einer dieser Annotationen des JAXB. Genauer gesagt soll diese Java-Klasse in ein XML-Dokument umgewandelt werden. Die zentrale Funktionalität liegt dabei in der Klasse JAXBContext. In Programmauszug 2.3 wird zunächst die Klasse „User" instanziiert und mit Daten befüllt.

[1] Der Iterator und der Cursor helfen beim Durchlaufen einer Menge von Elementen

Programmauszug 2.2: Die Klasse User

```
1  @XmlRootElement
2  class User {
3    private String username;
4          private Date createdAt;
5
6          public String getUsername() {
7            return this.username;
8          }
9
10         public void setUsername(String username) {
11           this.username = username;
12         }
13
14         public String getCreatedAt() {
15           return this.createdAt;
16         }
17
18         public void setCreatedAt(Date createdAt) {
19           this.createdAt = createdAt;
20         }
21 }
```

Anhand dessen wird über den JAXBContext und den Marshaller[1] das XML-Dokument ausgegeben.

Programmauszug 2.3: Nutzung von JAXB und Marshaller (main())

```
1 User admin = new User();
2 admin.setUsername = "mmustermann";
3 admin.setBirthday(new GregorianCalender(1910, Calender
     .JUNE, 22).getTime());
4
5 JAXBContext context = JAXBContext.newInstance(User.
     class);
6 Marshaller m = context.createMarshaller();
7 m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
     Boolean.TRUE);
8 m.marshal(admin, System.Out);
```

Programmauszug 2.4 zeigt die XML-Ausgabe der Ausführung des Marshaller aus Programmauszug 2.3.
Im Allgemeinen lässt sich sagen, dass JAXB dem Entwickler ermöglicht, möglichst einfach XML in Java und Java in XML umzuwandeln. Dies ist besonders wichtig, da sich

[1] Der Marshaller und der Unmarshaller sind Tools zum Schreiben(Marshaller) und zum Lesen(Unmarshaller) des XML-Inhaltes

XML als Standard zum Austausch von Daten zwischen verschiedenen Systemen durchgesetzt hat. Eine möglichst einfache Umwandlung führt also zu einer vereinfachten Kommunikation zwischen zwei verschiedenen Systemen[Ed 03].

Programmauszug 2.4: XML-Ausgabe von Listing 2.3

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"
    ?>
2 <user>
3   <createdAt>1910-06-22T00:00:00+01:00</createdAt>
4   <username>mmustermann</username>
5 </user>
```

2.3 Einführung in die Frameworks

Dieser Abschnitt soll sich mit den Grundlagen der verschiedenen Frameworks beschäftigen. Zunächst soll daher der Begriff Framework und die Vorteile in der Nutzung von Frameworks erklärt und dann spezieller auf die ausgewählten Frameworks eingegangen werden.

Unter einem Framework versteht der Software-Entwickler ein Grundgerüst, das ihm mit Hilfe seiner vordefinierten Funktionalität, eine einfachere Entwicklung von Programmen ermöglicht. Neben diesen verschiedenen Bausteinen, die die Entwicklung erleichtern sollen, geben die Frameworks auch die Verarbeitung und das Verhalten dieser untereinander vor. Durch die bereits vordefinierten Bausteine kann die Nutzung eines Frameworks den Entwickler Entwicklungszeit und Konfigurationsaufwand sparen.

2.3.1 Einführung in Hibernate

Hibernate ist ein Framework zur Nutzung von Objekt-relationalem Mapping, basierend auf der Programmiersprache Java. Es soll dem Entwickler helfen Objekte auf relationale Datenbanken mit Tabellenschema abzubilden.

Objekt-relationales Mapping

Objekt-relationales Mapping, auch ORM, ist eine Technik, die ermöglicht, Daten zwischen, eigentlich unkompatiblen, Typsystemen (Datenbanken und Objekte) zu übertragen. Die Inkompatibilität kommt durch die Darstellungsmöglichkeiten beider Typsysteme zustande, denn es müssen Objekte mit ihren Beziehungen in ein Tabellenschema der Datenbanken abgebildet werden. Dieses Problem wird auch „object-relational impedance mismatch“ oder „objektrelationale Unverträglichkeit“ genannt[Cuo08].
Durch die hohe Verbreitung von relationalen Datenbanken wird auch die Nutzung von OR-Mappern gefördert. Durch die Nutzung von OR-Mappern und der damit einhergehenden Übernahme der Datenspeicherung durch das Tool, kann der Entwickler Zeit und Arbeit sparen und sich stattdessen auf den Kern seiner Anwendung konzentrieren. Also führt die Nutzung eines OR-Mappers im Allgemeinen zu einer Verbesserung der Produktivität.
Die Automatisierung der Datenspeicherung bietet dem Entwickler noch andere Vorteile,

wie die Reduzierung von Code-Zeilen. Durch die Verwendung der vordefinierten Funktionen zur Datenspeicherung des Tools wird die Anzahl an Code-Zeilen innerhalb der Anwendung verringert. Dies führt zu einer höheren Wartbarkeit und leichteren Verständlichkeit des Systems[Cuo08].
Das Mapping ist der zentrale Bestandteil der OR-Mappern. In Java beispielsweise, wird den Mappern über JPA's („Java Persistence Annotations") Informationen mitgeteilt, die sie benötigen um die Objekte in die Tabellenstruktur der relationalen Datenbanken zu speichern. Dazu später mehr.

2.3.2 Einführung in Symfony2

Symfony2 basiert auf der Programmiersprache PHP, einer server-seitigen Skriptsprache. Der große Unterschied einer Skriptsprache zu anderen Programmiersprachen liegt in der Verarbeitung des geschriebenen Programms. Während bei anderen Programmiersprachen das Programm zur Ausführung von einem Compiler in einen Binärcode umgewandelt wird, wird das Programm in einer Skriptsprache während der Laufzeit interpretiert[itwb]. Der Bereich der Skriptsprachen lässt sich indes in weitere Unterkategorien aufteilen. Dazu zählen server-seitige Skriptsprachen, wie PHP, aber auch client-seitige Skriptsprachen, wie Javascript. Der Unterschied hier liegt am Ort der Verarbeitung. Der PHP-Quellcode läuft auf einem Web-Server, daher auch server-seitige Skriptsprache. Der Javascript-Quellcode dagegen wird nicht auf dem Server, sondern auf dem Client, in diesem Fall dem Webbrowser des Nutzers, ausgeführt. PHP ist die meist genutzte Skriptsprache zum Erstellen von Web-Services[Doy10]. Überprüft man jedoch die Syntax von Symfony2 Programmen, lässt sich schnell die Besonderheit von Symfony2 erkennen. Symfony2 basiert zwar intern auf PHP, programmiert wird allerdings objektorientiert, mit einer java-ähnlichen Syntax.

Neben XML bietet Symfony2 noch die Unterstützung einer weiteren Auszeichnungssprache. YAML („YAML Ain't Markup Language") bietet eine vereinfachte, an diversen verbreiteten Datenstrukturen angelehnte Syntax.

Das Konzept von Symfony2 bedient sich dabei dem einfachen Kommunizieren mittels HTTP oder wie Symfony2 sagt: „Back to the Basics"[Sen14]. Das führt zu folgendem Ablauf:

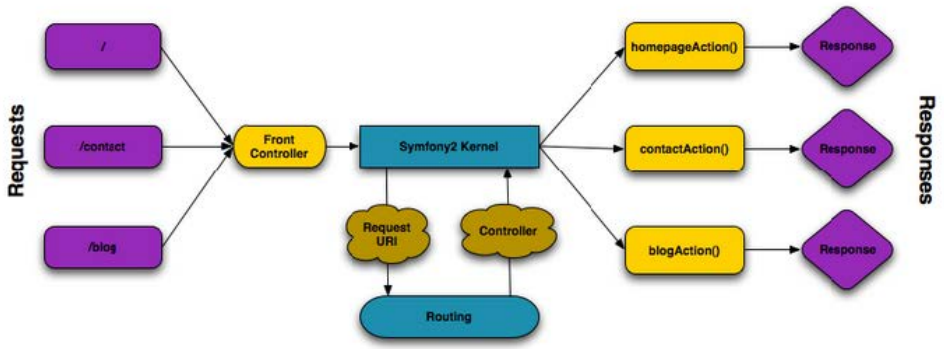

Abbildung 2.2: Aufbau Symfony2[Sen14]

Der Nutzer stellt über seinen Client/Browser eine Anfrage (Request) auf die Anzeige

eines Inhalts an den Server. Dies geschieht meist durch den Aufruf einer URL in seinem Browser. Der Server verarbeitet den Request und sendet eine Antwort (Response) an den Client, der den Inhalt dann anzeigen kann.
Zum Verarbeiten des Requests nutzt Symfony2 die sogenannte MVC-Philosophie. Das bedeutet eine Aufteilung zwischen Model, Views und Controller und damit einer Trennung der Bereiche Oberfläche, Klassen-Schema und ausführbare Logik. Auch existieren für Symfony2 diverse Erweiterungen zum Ausbauen der Funktionalität. Dazu zählt auch der OR-Mapper Doctrine, dessen Funktionalität innerhalb des nächsten Kapitels weiter untersucht werden wird. Im weiteren Verlauf wird daher vom Symfony2-Mapping gesprochen.

Kapitel 3

Nutzung von XML und Annotationen in Frameworks

3.1 Analyse der Hibernate Funktionalität

Es soll nun speziell die Funktionalität des Frameworks Hibernate untersucht werden. Genauer gesagt soll dieses Kapitel die Mächtigkeit der Hibernate-Mapping-Schemata über Annotationen und XML untersuchen und dann andere Möglichkeiten außerhalb des Mapping eruieren. Außerdem wird die Nutzung des Hibernate-Mapping innerhalb eines Projektes untersucht.

3.1.1 Das Hibernate-Mapping

Die Hauptaufgabe von Hibernate als OR-Mapper ist das Mappen von Objekten. Über Annotationen und XML werden diese Objekte in ein relationales Datenbankschema gemappt und können dann verarbeitet werden.

Mächtigkeit

Dieser Abschnitt soll sich mit der Mächtigkeit des Hibernate-Mappings beschäftigen. Generell soll untersucht werden, ob durch XML-Mapping Vorteile gegenüber Mapping mit Annotationen oder umgekehrt geschaffen werden können.

In Hibernate existieren zwei Arten von Annotationen und XML-Tags. Auf der einen Seite die Implementationen zu den Spezifikationen der JPA des Pakets javax.persistence und auf der anderen Seite Hibernate-eigene Annotationen des Pakets org.hibernate.annotations. Die Spezifikationen der JPA und der Hibernate-Annotationen finden Sie HIER und HIER. Zunächst sollen beide Varianten untereinander untersucht werden.

Dafür wird die Menge an JPA Annotationen auf ihr entsprechendes XML-Äquivalent überprüft. Als XML-Referenz galt das JPA-XML-Schema von Oracle. Eine Gegenüberstellung des Schemas und der Annotationen der JPA-Annotation ergab folgende Unterschiede:

- andere Benennung der XML-Tags

- Benutzung als Attribut eines anderen XML-Tags
- Es existieren keine Annotationen für die Mehrzahl

Ein Großteil der Annotationen existiert äquivalent innerhalb des XML Schemas. Jedoch gilt dies für manche Annotationen nicht.

Tabelle 3.1: Änderungen entsprechend Punkt 1 und 2

Annotation	XML
@Access	<entity access="">
@Cacheable	<entity cacheable="">
@MapsId	<one-to-one maps-id="">
@NamedSubgraph	<subgraph>
@QueryHint	<query>
@StoredProcedureParameter	<parameter>

Das bedeutet während die Annotation „@Access“ über das Entity-Tag gesetzt wird, wird die Annotation „@NamedSubgraph“ im XML-Schema umbenannt.

Programmauszug 3.1: Verkettung von Annotationen

```
1  @JoinColumns({
2    @JoinColumn(name="user_firstname",
       referencedColumnName="firstName"),
3    @JoinColumn(name="user_lastname",
       referencedColumnName="lastName")
4  })
```

Sollen innerhalb eines Annotation-Mappings beispielsweise mehrere „@JoinColumn“-Annotationen gebündelt werden, geschieht dies mit der Annotation „@JoinColumns“ die wiederum mehrere „@JoinColumn“-Annotationen enthält.

Programmauszug 3.2: Verkettung von XML

```
1  <join-column name="user_firstname" referenced-column
     -name="firstName"/>
2  <join-column name="user_lastname" referenced-column-
     name="lastName"/>
```

Beim XML-Mapping dagegen, können diese ungebündelt angegeben werden. Es gilt dort Punkt 3 der obigen Aufzählung, da die in Abbildung 3.3 angegebene Struktur nicht existiert.

Programmauszug 3.3: Erwartete, aber nicht existente Struktur des XML-Mappings

```
1  <join-columns>
2    <join-column/>
3    <join-column/>
4  </join-columns>
```

Generell lässt sich also für die JPA Implementation Hibernates feststellen, dass sowohl Annotationen und XML die gleiche Mächtigkeit besitzen. Nun sollen die Äquivalenzen der Hibernate-spezifischen Mapping-Funktionen überprüft werden.

Auch für die Hibernate-spezifischen Annotationen existieren alle Äquivalenzen im XML-Mapping. Jedoch existieren dort wesentlich weniger exakte Übertragungen von Annotationen auf Tags. Es werden häufiger Attribute anderer Tags zum Angeben der Informationen genutzt. Es spiegeln sich dementsprechend auch alle Punkte der obigen Aufzählung im Hibernate-Mapping wieder. Dazu kommt noch eine Aufteilung in verschiedene, ineinander verwendete Tags. Beispiel ist dafür die in Tabelle 3.2 enthaltene Annotation „@JoinFormula“. Auch hier sollen folgend ein paar Beispiele der Unterschiede genannt werden.

Tabelle 3.2: Änderungen Hibernate-Mapping

Annotation	XML
@Cascade	<any cascade="">
@Check	<column check="">
@DiscriminatorFormula	<discriminator formula="">
@GenericGenerator	<generator>
@Immutable	<set mutable="">
@JoinFormula	<join><property><formula/></property></join>

Die in Tabelle 3.2 aufgeführten Annotationen sind lediglich ein Auszug der verschiedenen Äquivalenzen. Die Annotationen „@Cascade“, „@Check“ und „@DiscriminatorFormula“ sollen die Verwendung als XML-Attribute zeigen. Dagegen zeigen die Annotationen „@GenericGenerator“ und „@Immutable“ eine Veränderung der Namen innerhalb der Umwandlung. Die Annotation „@JoinFormula“ ist eine Besonderheit in der XML-Nutzung, denn wird die Information erst in einem untergeordneten Tag gesetzt.

Auch hier lassen sich also keine Unterschiede feststellen. Es besteht dennoch die Möglichkeit einer unterschiedlichen Mächtigkeit. Dafür muss nun untersucht werden, ob innerhalb eines Mappings beide Schemata gemischt werden können. Funktioniert dies nur bei Annotationen- oder bei XML-Mapping, ist eine unterschiedliche Mächtigkeit gegeben. Dazu wird folgendes Klassendiagramm eingeführt:

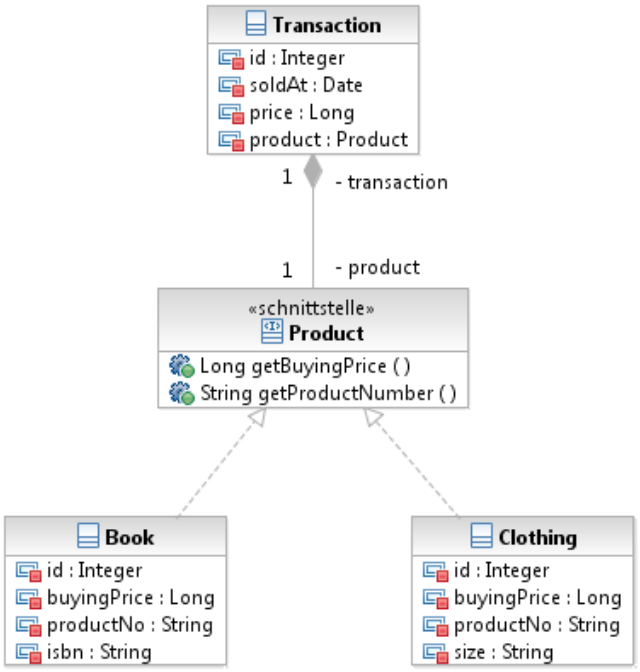

Abbildung 3.1: Klassendiagramm Transaktion

Dieses Klassendiagramm soll in die Datenbank gemappt werden. Das Problem hier liegt an der polymorphen Assoziation zwischen „Transaction“ und „Product“. Polymorph bedeutet in diesem Zusammenhang eine Verbindung der Klasse „Transaction“ zu verschiedenen Klassen, die über keinerlei Verbindungen verfügt. Dies kann jedoch über die Hibernate-spezifische Annotation „@Any“ gelöst werden. Das bedeutet, es existieren Hibernate-Funktionen, die nicht durch JPA-Funktionen abgedeckt werden können. Aus diesem Grund wird jetzt versucht dieses Klassendiagramm mit JPA und Hibernate Annotationen zu mappen.

book $\{id, buyingPrice, isbn, productNo\}$
clothing $\{id, buyingPrice, productNo, size\}$
transaction $\{id, price, productType, productId, soldAt\}$

Abbildung 3.2: Datenbankschema

Das Datenbankschema verrät was nun passiert ist. „Transaction“ bekommt über die Spalte „productType“ mitgeteilt, welcher Typ das assoziierte „Product“ besitzt, also dementsprechend „clothing“ oder „book“. Über die Spalte „productId“ wird der Identifier des jeweiligen Objektes referenziert. Das bedeutet in diesem Zusammenhang, dass sowohl Hibernate- als auch JPA-Annotationen zusammen angewendet werden können, allerdings

nur unter der Bedingung, dass nur unterschiedliche Annotationen der beiden Pakete verwendet werden. Es ist beispielsweise nicht möglich die Annotation „@OrderBy“ beider Schemata in einer Klasse zu verwenden.

Das Problem des XML-Mapping ist, es existiert kein Schema in dem sowohl XML-Mapping mit JPA-Tags und Hibernate-Mapping vermischt werden können. Daher ist es nur möglich unter verschiedenen Mapping-Dateien zu variieren. Dies lässt die Vermutung offen, die Mächtigkeit der Annotationen höher einzuschätzen, als die des XML. Denn lässt sich mit den Annotationen die Funktionalität des JPA und die Funktionalität des Hibernate-Mappings anwenden, bei XML aber nur eins der beiden. Es müssten allerdings weitere Untersuchungen angestellt werden, inwiefern sich die Mächtigkeiten von Hibernate- und JPA-Mapping via XML unterscheiden. Enthält das Hibernate-Mapping alle Funktionalitäten des JPA-Mappings via XML, würde die Diskussion teilweise ad absurdum geführt werden. Denn dann würde es generell keinen Sinn machen JPA-Annotationen mit Hibernate-Annotationen zu vermischen.

Alternative Mapping-Schemata

Vor der Einführung von Annotationen, existierte bereits eine Möglichkeit Mapping nicht, wie in XML, getrennt vom Source-Code in externen Dateien anzugeben. Die Nutzung von XDoclet zum Mapping wurde jedoch durch die Möglichkeit der Annotationen weitestgehend abgelöst. Dennoch wird dies weiterhin in Hibernate unterstützt.
Listing 3.4 zeigt einen Auszug der Anwendung eines XDoclet-Mapping.

Programmauszug 3.4: Beispiel eines XDoclet-Mapping

```
1  /**
2   * @hibernate.class
3   *  table="PERSON"
4   */
5  public class Person{
6    ...
7
8    /**
9     * @hibernate.id
10    *  column="PERSON_ID"
11    */
12   public Long getId() { ... }
13
14   /**
15    * @hibernate.many-to-one
16    *  column="ADDRESS_ID"
17    */
18   public Address getAddress() { ... }
19 }
```

Die Ergebnis einer XDoclet-Struktur unterscheidet sich im Ergebnis nicht von der durch Annotationen oder XML erstellten Datenbankstruktur. Jedoch unterscheidet sich die Ver-

arbeitung grundlegend von der der Annotationen. Denn eine XDoclet-Struktur wird nicht intern zur Laufzeit vom Compiler interpretiert, sondern vor Programmausführung in Java-Code umgewandelt[wik], indem es zunächst in ein äquivalentes XML-Mapping Schema transferiert wird.

Verwendung innerhalb eines Projektes

Nun sollen die verschiedenen Mapping-Schemata innerhalb eines Projektes vermischt werden. Es stellt sich die Frage, ob es generell funktioniert, verschiedene Mapping-Schemata in Hibernate zu vermischen.

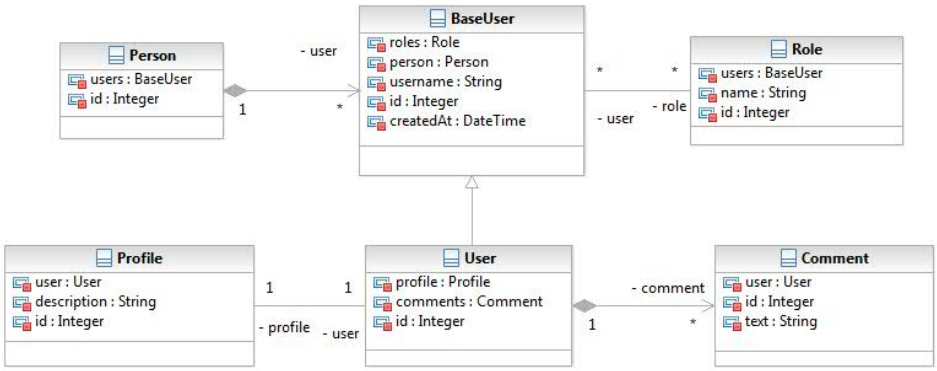

Abbildung 3.3: Klassendiagramm

Dazu soll die Klassenstruktur aus Abbildung 3.3 in beiden Mapping-Schemata implementiert werden. Zunächst soll überprüft werden, ob Annotationen und XML genutzt werden können, dazu sollen die Klassen „BaseUser“, „User“ und „Person“ als XML und die Klassen „Profile“, „Comment“ und „Role“ als Annotation gemappt werden. Die Klasse „BaseUser“ liegt als Teil der Referenz-Implementation der Arbeit im Anhang bei

Programmauszug 3.5: Auszug der Konfigurationsdatei

```
1  <?xml version="1.0" encoding="utf-8"?>
2    <!DOCTYPE ... >
3    <hibernate-configuration package="com.ba.hibernate">
4      <session-factory>
5                    ...
6        <mapping resource="BaseUser.hbm.xml"/>
7        <mapping resource="Person.hbm.xml"/>
8        <mapping class="Profile"/>
9        <mapping class="Role"/>
10       <mapping class="Comment"/>
11     </session-factory>
12 </hibernate-configuration>
```

Diese Konfiguration definiert die oben festgelegte Mapping-Struktur.

allusers {id, person_id, profile_id, createdAt, username, type}
comment {id, user_id}
person {id}
profile {id, description}
role {id, name}
role_baseuser {role_id, baseuser_id}

Abbildung 3.4: Datenbankschema

Insgesamt ergibt dies das zu erwartende Datenbankschema aus Abbildung 3.4. Das bedeutet die Schemata können innerhalb eines Hibernate-Projektes gemischt werden. Nun soll neben XML und Annotationen auch die Nutzung von XDoclet innerhalb dieses Mapping getestet werden. Dazu wird die Mappingstruktur dementsprechend verändert, dass die Klassen Comment und Person als XDoclet gemappt werden. Allerdings entspricht dies exakt dem XML-Mapping, da die XDoclet-Metadaten zunächst in ein XML-Schema umgewandelt werden, um diese zu nutzen. Auch XDoclet kann also auch ohne Probleme innerhalb eines Projektes mit Annotationen und XML verwendet werden.

3.1.2 Alternative Nutzung

Das Mapping ist im Allgemeinen das wichtigste Feature von Hibernate, jedoch existieren, neben jenem Mapping, noch andere Möglichkeiten XML und Annotationen innerhalb von Hibernate zu verwenden. Eingesetzt werden XML und Annotationen innerhalb von Konfigurationsdateien und zum Validieren von Input durch einen Validator.

Konfiguration

Zur Konfiguration innerhalb eines Hibernate-Projektes besteht die Möglichkeit eine XML-Konfigurationsdatei zu erstellen. Hibernate stellt aufgrund vieler Einstellungsmöglichkeiten eine Standard-Konfigurationsdatei „hibernate.properties“ zur Verfügung, die Standard- Belegungen zur Benutzung von Hibernate enthalten.

Programmauszug 3.6: Einbinden von hibernate.cfg.xml

```
1 SessionFactory sf = new Configuration()
2     .configure()
3     .buildSessionFactory();
```

Sollte es notwendig sein, eine eigene Konfigurationsdatei einzubinden, wird diese, anstatt der Java-Property-Datei „hibernate.properties“, als „hibernate.cfg.xml“ im Hauptpfad des Projektes hinterlegt. Für den Fall, dass sowohl die Property-Datei als auch eine XML-Konfiguration eingebunden wurde, überschreibt die XML-Datei die Standardwerte der Property-Datei.

Programmauszug 3.6 zeigt die Importierung der eigenen Konfigurationsdatei „hibernate.cfg.xml“. Über die Funktion „configure()“ kann mit angegebenem ersten Argument („.configure („/directory/other.cfg.xml“)“) eine Konfiguration mit gewünschtem Pfad eingebunden werden.

Programmauszug 3.7: XML-Konfigurationsdatei

```
1  <?xml version='1.0' encoding='utf-8'?>
2    <!DOCTYPE hibernate-configuration PUBLIC
3      "-//Hibernate/Hibernate Configuration DTD//EN"
4      "http://hibernate.sourceforge.net/hibernate-
          configuration-3.0.dtd">
5    <hibernate-configuration>
6      <session-factory name="java:hibernate/
          SessionFactory">
7        <property name="hibernate.connection.url">
8                              jdbc:mysql://localhost/myDB
9        </property>
10       <property name="hibernate.connection.username">
11         root
12       </property>
13       <property name="hibernate.connection.password">
14         password
15       </property>
16       <mapping resource="org/hibernate/auction/Item.
            hbm.xml"/>
17     </session-factory>
18 </hibernate-configuration>
```

Programmauszug 3.7 zeigt eine solche Beispiel-Konfiguration. Jedes Hibernate-Projekt benötigt innerhalb seiner Konfiguration die Definition einer „SessionFactory“, um bei erfolgreichem Zugriff eine „Session“ zu erstellen. Dieser „Session-Factory“ können über das Property-Tag Eigenschaften zur Nutzung mitgegeben werden. Beispielsweise kann über die in Programmauszug 3.7 angegebenen Property-Tags die Nutzung einer Datenbank „MyDB“ mit User „root“ angegeben werden. Über das Mapping-Tag können angelegte XML-Mapping-Dateien eingefügt werden. Neben der „Session-Factory“ kann eine Hibernate-Konfiguration noch weitere Konfigurationen enthalten, jedoch soll das Beispiel der „Session-Factory“ genügen.

3.1.3 Interne Verarbeitung

Das vorherige Kapitel handelt von der Konfiguration anhand der „Session-Factory“. Doch wie läuft die Verarbeitung des Mappings von der Konfiguration bis zum Datenbankschema ab? Dieser Frage soll in diesem Kapitel nachgegangen werden. Dazu erst mal ein vereinfachtes Schaubild des Ablaufs:

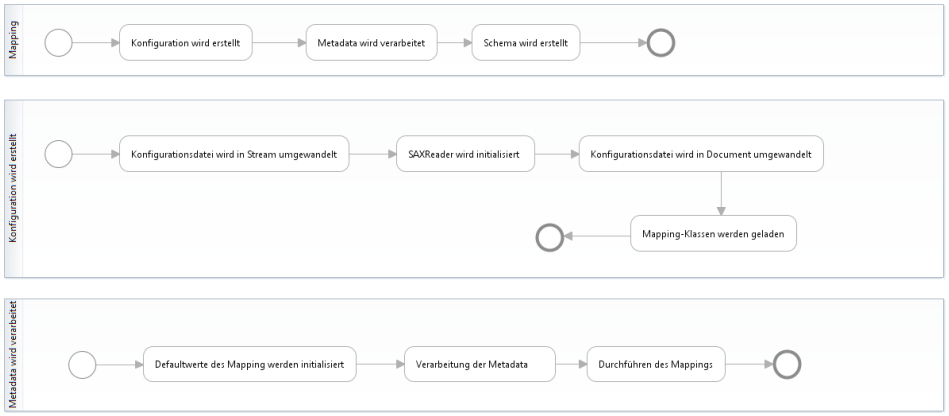

Abbildung 3.5: Ablauf Hibernate

Das Mapping lässt sich generell in 3 verschiedene Teilbereiche gliedern. Zuerst wird die Konfiguration durch einen Parser nutzbar gemacht und die gemappten Klassen in einer Collection untergebracht, dannach wird das Mapping dieser Klassen verarbeitet und zuletzt daraus das Schema in der Datenbank erstellt. Wie bereits im vorhergehenden Kapitel erwähnt, wird durch das „Session-Factory“-Attribut der Konfigurationsdatei, und der letztlich daraus resultierenden „Session“, der Datenbankzugriff des Programms gesteuert. Programmauszug 3.7 zeigt das Anlegen dieser Session-Factory und damit den Start des Mapping. Die Funktion „configure()“ des Configuration-Elementes bildet in Abbildung 3.5 den Bereich der Erstellung der Konfiguration ab, „buildSessionFactory()“ die Verarbeitung der Metadaten. Nun soll auf diese Teilbereiche gesondert eingegangen werden.

Konfiguration wird erstellt

Um die Konfiguration nutzbar zu machen, muss zunächst die XML-Konfiguration in ein intern verarbeitbares „Document“ umgewandelt werden. Dazu wird zunächst ein Input Stream[1] für die angegebene Konfigurationsdatei, deren Pfad, wie bereits erwähnt, auch explizit angegeben werden kann, angelegt. Anhand dieses Streams wird ein SAX-Reader initialisiert, der aus der XML-Konfigurationsdatei ein „Document“-Element erstellt. Für dieses Element wird nun ein Iterator angelegt und alle Property-Tags der Konfigurationsdatei als „Property“ zum Configuration-Element hinzugefügt (Configuration.addProperties()). Sind diese verarbeitet, wird ein neuer Iterator erstellt und über die restlichen Tags der Konfigurationsdatei iteriert. Findet der Iterator ein Mapping-Tag wird die Funktion „Configuration.parseMappingElement(Element, String)“ aufgerufen. Dort wird anhand des angegebenen Attributes zwischen mehreren Mappings unterschieden. Die Konfiguration aus Programmauszug 3.5 enthält beispielsweise die Attribute „resource“ und „class“. Ist das Attribut „class“ gesetzt, wird die Klasse in eine Liste („annotatedClasses[XClass]“) hinzugefügt, wird „resource“ gesetzt, müssen die Mapping-Dateien verarbeitet werden.
Dazu wird erneut ein Input Stream erstellt und über einen SAX-Reader die Mapping-Dateien, mit dem Zwischenschritt des „Document“-Elementes, in ein „XMLDocument“ umgewandelt. Das „XMLDocument“ wird untersucht, ob dort mehrere gemappte Klassen

[1] Ein Objekt zur Verarbeitung von Input als Bytes[Ull12]

genannt werden und diese dann in eine Map(„hbmMetadataByEntityNameXRef[String, XMLDocument]“) hinzugefügt.

Das bedeutet nach Ausführung der Konfiguration existieren eine Liste mit annotierten Klassen als XClass-Dateien und eine Map mit Klassen des XML Mapping aus Attribut-Paaren String und XMLDocument.

Metadata wird verarbeitet

Nachdem die Konfiguration verarbeitet wurde, müssen nun die Metadaten verarbeitet werden. Dafür wird zunächst ein Mapping Objekt erstellt und dort beispielsweise die Defaultwerte der Generatoren gesetzt werden. Dies geschieht über die Hilfsklasse „AnnotationBinder“. Sind die Defaultwerte gesetzt, werden die annotierten Klassen verarbeitet. Dazu werden alle Annotationen überprüft und sind diese als Entity deklariert, werden diese Klassen in eine Map(„annotatedClassesByEntityNameMap[String, XClass]“) hinzugefügt.

Nun startet der Prozess der Metadata-Verarbeitung. Anhand der Typen der Metadaten wird entscheiden, wie diese verarbeitet werden. Ist der Typ der Metadaten als „HBM“(XML) deklariert, wird die Funktion „processHbmXmlQueue()“ ausgeführt. Ist der Typ dagegen „Class“(Annotation), wird „processAnnotatedClassesQueue()“ ausgeführt.

Die XML-Mappings werden über die Hilfsklasse „HbmBinder“ auf ein Mapping-Element gebunden. Existieren diese Klassen auch in der Map der Annotationen, werden diese dort entfernt, genauso anders herum. So wird eine doppelte Verarbeitung ausgeschlossen. Die annotierten Klassen werden dagegen über den Annotation-Binder in ein Mapping-Element gebunden. Wie genau dies abläuft soll an dieser Stelle nicht betrachtet werden.

Nun werden die Metadaten intern zu Tabellen-Elementen umgewandelt. Auch das Erstellen der Tabellen soll an dieser Stelle nicht weiter betrachtet werden.
Diese Tabellen-Elemente dienen letztlich zur Vorlage zum Erstellen des Datenbankschemas über den Datenbankadapter der Session. Im Beispiel der Konfiguration aus Programmasuzug 3.7 wird der JDBC Adapter verwendet, um eine MySQL-Datenbank zu adressieren. Über die erstellte Session können nun alle datenbankspezifischen Anfragen abgearbeitet werden.

3.2 Analyse der Symfony2 Funktionalität

Nachdem bereits Hibernate auf seine Funktionalität im Umgang mit Annotationen und XML untersucht wurde, soll in diesem Kapitel nun eine Analyse der Symfony2-Funktionalität stattfinden. Auch hier soll die Mächtigkeit und die Nutzung innerhalb eines Projektes anhand des Mappings untersucht werden. Außerdem werden noch andere Nutzungsmöglichkeiten außerhalb des Mappings dargestellt.

3.2.1 Das Symfony2-Mapping

Symfony2 bietet neben dem XML-Mapping und dem Mapping durch Annotationen, noch die Möglichkeit über ein PHP-Mapping-Datei oder über ein Mapping mit der Auszeichnungssprache Yaml. Das PHP-Mapping soll desweiteren nicht betrachtet werden.

Mächtigkeit

Die Auswahl einer Systemkomponente kann durchaus wichtig sein. Meist spielt dabei die Mächtigkeit eines Systems eine gewichtige Rolle. Aus diesem Grund sollen die verschiedenen Frameworks nun auf ihre Mächtigkeit in verschiedenen Teilsystemen untersucht werden. Tabelle 3.3 zeigt eine Übersicht der verschiedenen XML-Tags mit ihrem äquivalenten Pendant der anderen Mapping-Tools.

In dieser Tabelle lässt sich feststellen, zu jeder Eigenschaft existiert ein äquivalentes Pendant der anderen Mapping-Tools.
Lediglich der markierten Eigenschaften bedarf es einer kurzen Erklärung. Das Field-Tag des XML und YAML-Mapping existiert nicht in der Annotation, dies wird durch die Annotation „@Column“[1] repräsentiert. Genauso existiert kein eigenes Yaml-Tag für die Angabe einer Entity und einer MappedSuperclass, dies wird über ein Tag realisiert, dem der Typ Entity oder MappedSuperclass mitgegeben wird[2]. In XML kann über das Tag der LifecycleCallbacks alle verschiedenen Callbacks über ein Attribut type gesetzt werden. In YAML und Annotationen existieren für die einzelnen Callbacks eigene Elemente[3]. Die Besonderheit der Annotationen „@Index“ und „@UniqueConstraint“ liegt am Ort der Anwendung, während diese in XML und YAML als eigenständige Tags existieren, werden die Annotationen innerhalb der Table-Annotation angegeben[4].
Dennoch scheinen sowohl XML, Annotationen und Yaml-Mapping rein subjektiv die gleiche Mächtigkeit zu besitzen. Um das genauer zu betrachten, soll im Folgenden die Klassenstruktur aus Abbildung 3.3 des vorherigen Kapitels, in die jeweiligen Möglichkeiten umgesetzt und überprüft werden, ob die Ergebnisse des Mappings äquivalent sind.

Tabelle 3.3: Vergleich der verschiedenen Mapping-Mächtigkeiten

Annotation	**XML**	**YAML**
XML	Annotation	Yaml
<field>	$@Column$[1]	fields
<entity>	@Entity	$type : entity$[2]
<id>	@Id	id
<generator>	@GeneratedValue	generator
<sequence-generator>	@SequenceGenerator	sequenceGenerator
<mapped-superclass>	@MappedSuperclass	type:$mappedSuperclass$[2]
<discriminator-column>	@DiscriminatorColumn	discriminatorColumn
<discriminator-map>	@DiscriminatorMap	discriminatorMap
<lifecycle-callbacks>	@HasLifeCycleCallbacks	lifecycleCallbacks
<lifecycle-callback>	$@PrePersist$[3]	prePersist: [methods]
<one-to-one>	@OneToOne	oneToOne
<one-to-many>	@OneToMany	oneToMany
<many-to-one>	@ManyToOne	manyToOne
<many-to-many>	@ManyToMany	manyToMany
<order-by>	@orderBy	orderBy
<indexes>	$@Index$[4]	indexes
<unique-constraints>	$@UniqueConstraint$[4]	uniqueConstraints
<change-tracking-policy>	@ChangeTrackingPolicy	changeTrackingPolicy
<join-column>	@JoinColumn	joinColumn

Tabelle 3.3 -Vergleich der verschiedenen Mapping-Mächtigkeiten

Annotation	XML	YAML
<join-table>	@JoinColumn	joinTable

Die Klasse „BaseUser“ liegt als Teil der Referenz-Implementation der Arbeit im Anhang bei. Diese Referenz-Implementation enthält, ohne Event-Annotationen des Lifecycle-Callbacks, etwa 65% der Annotationen der Doctrine Annotation Reference[Tea], dem Referenzdokument des ORM-Mappers. Die nicht verwendeten Annotationen, wie „@Index“ und „@UniqueConstraint“ besitzen allerdings, die in Tabelle 3.3 definierten Pendants. Nach Anlegen des Datenbankschemas dieser Referenz-Implementation ergab sich daraus die Datenbankstruktur aus Abbildung 3.6.
Auch die Transformation in die anderen Mapping-Schemata ergab dieses Ergebnis. Insgesamt existieren 6 verschiedene Tabellen mit den angegebenen Tabellenzeilen. Bedeutet daher, alle Mapping-Schemata des in Symfony2 verwendeten ORM besitzen die gleiche Mächtigkeit. Jedes Schema besitzt eine unterschiedliche Syntax, anhand der Mächtigkeit kann allerdings kein Unterschied erkannt werden.

allusers $\{id, person_id, profile_id, createdAt, username, type\}$
comment $\{id, user_id\}$
person $\{id\}$
profile $\{id, description\}$
role $\{id, name\}$
role_baseuser $\{role_id, baseuser_id\}$

Abbildung 3.6: Datenbankschema

Verwendung innerhalb eines Projektes

Die Mächtigkeit der Mapping-Schemata innerhalb Symfony2 unterscheidet sich also nicht. Daher soll jetzt geklärt werden, ob die Schemata innerhalb eines Projektes gemischt werden können. Dementsprechend soll nun, aus der schon bei der Mächtigkeit eingesetzten Klassenstruktur, einzelne Klassen in den Mapping-Schemata variieren. Die Klassen BaseUser und User als Annotation-Schema, Comment und Profile als XML-Schema sowie Role und Person als Yaml-Schema.
Doch ergab das Erstellen dieser Klassenstruktur mit verschiedenen Schemata, eine Mapping-Exception. Grund dafür sind die Assoziationen zwischen den Klassen aus verschiedenen Schemata. Ein Beispiel dieser nötigen Assoziationen ist BaseUser - Role. Das Mapping der Klasse BaseUser sucht das Mapping der Klasse Role. Allerdings existiert dort kein Annotation-Mapping, sondern ein XML-Mapping und dies erkennt der OR-Mapper nicht.

Dennoch beantwortet diese Erkenntnis nicht, ob generell verschiedene Mapping-Schemata verwendet werden können. Um diese Frage zu beantworten wird neben der Klassenstruktur aus Abbildung 3.7 eine zweite unabhängige Klassenstruktur eingefügt. Beide Klassenstrukturen sollen demnach in unterschiedlichen Schemata gemappt werden.

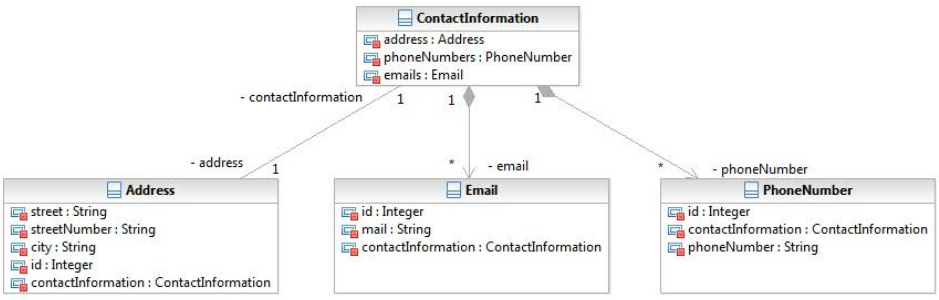

Abbildung 3.7: Klassendiagramm

Zunächst wird versucht die Mapping-Schemata innerhalb eines Bundles der Symfony2-Struktur zu vermischen. Dazu wird die Konfigurationsdatei des Mappings verändert, dass sowohl XML-Mapping und Annotation-Mapping dem gewünschten Bundle zugewiesen werden. Hier der wichtige Ausschnitt der Konfigurationsdatei am Beispiel einer YAML:

Programmauszug 3.8: Der Annotation Reader

```
1 doctrine:
2     orm:
3         auto_mapping: true
4         mappings:
5             BABABundle: { type: xml, dir: path/to/xml
                  }
6             BABABundle: { type: annotation, dir:
                 Entity/}
```

Das Überprüfen des Datenbankschemas zeigte allerdings ein Problem in der Benutzung dieser Konfiguration. Das erste Mapping wird ausgeführt, das zweite dagegen ignoriert.
Bleibt also nur noch die Möglichkeit die Mapping-Schemata in verschiedenen Bundles anzuwenden. Dazu wird ein neues Bundle erstellt, die Konfiguration angepasst und die Dateien des Mapping in das andere Bundle zu transferieren. Doch auch hier wird lediglich das Schema des ersten, in der Konfiguration angegebenen, Mappings in der Datenbank angelegt.
Die Nutzung von verschiedenen Mapping-Schemata innerhalb eines Projektes wird also in Symfony2 nicht unterstützt. Eine Rangliste der Mapping-Schemata lässt sich an den Ergebnissen daher nicht erstellen. Die Auswahl bleibt also eine subjektive Auswahl des Entwicklers, ob er das Mapping direkt in den Klassen, via Annotationen, oder in externen Dateien liegen haben möchte.

Interne Verarbeitung

Dieser Abschnitt soll sich mit der internen Verarbeitung der Metadaten innerhalb Symfony2 befassen. Es soll nachgeprüft werden, welche Schritte die verschiedenen Arten der Metadaten beim Mapping durchlaufen.

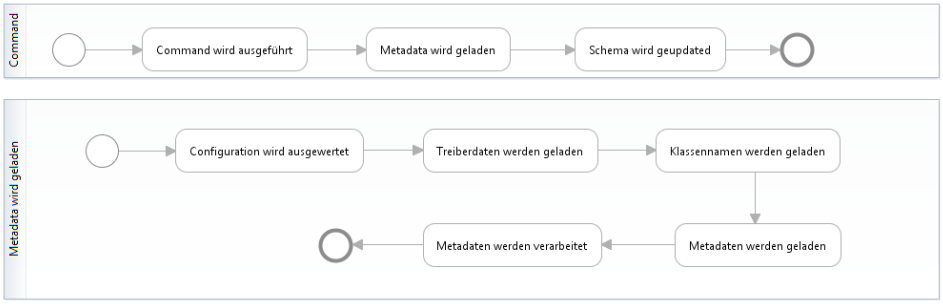

Abbildung 3.8: Prozess des Mappings

Der Ablauf des Mappings lässt sich in 3 Teilbereiche gliedern. Zunächst der Ausführungsprozess über die Kommandozeile, dann die Verarbeitung der Metadaten und zuletzt das Erstellen oder Bearbeiten des Schemas. Bereich 1 und Bereich 3 durchlaufen alle, das Verarbeiten der Metadaten sollte sich unterscheiden, daher zeigt Figure 3.4 diesen Bereich noch einmal aufgeteilt. Auch hier laufen zunächst die ersten beiden Tasks parallel ab. Sowohl bei der Nutzung von XML als auch bei allen anderen Nutzungen wird zunächst die Konfiguration geladen und anhand der Konfiguration wird ein spezifischer Treiber geladen. Der Treiber leitet die Verarbeitung der Metadaten anhand des ausgewählten Schemas. An dieser Stelle wird es interessant, da jeder Treiber für die Metadaten eine andere Verarbeitungsweise bereit stellt.

Generell lässt sich also feststellen, der spezifische Ablauf des Mapping liegt in den jeweiligen Treibern. Es lässt sich in zwei verschiedene Bereiche untergliedern. Die internen Metadaten, wie Annotationen, und die externen Metadaten, wie XML und YAML. Doch an welchen Stellen werden diese Treiber verwendet?

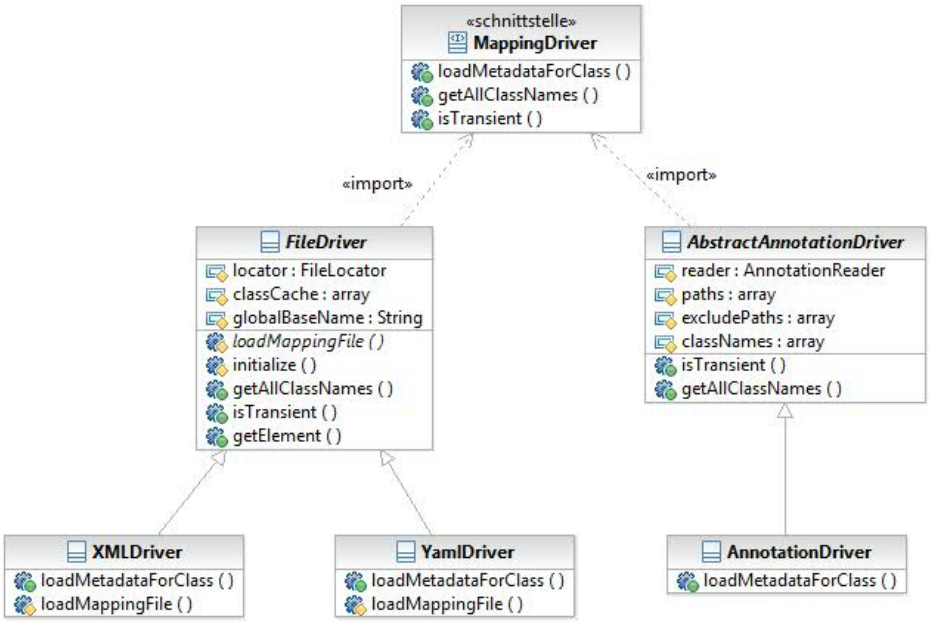

Abbildung 3.9: Treiberstruktur

Die Treiber werden an verschiedenen Stellen benötigt. Über den jeweiligen Treiber müssen die verschiedenen Klassennamen geladen werden. Bei Annotationen wird mit dem „AnnotationReader“ überprüft, ob die Pfade aus der Konfigurationsdatei wirklich annotierte Klassen enthalten. Die gefundenen Klassennamen werden nun an eine „MetadataFactory“ gegeben, die diese weiter verwendet. Anders bei den externen Mapping-Dateien. Dort müssen zunächst über einen „FileLocator“ die verschiedenen Dateien gesucht, geladen und anhand der Tags „entity“ und „mappedsuperclass“ die Klassen identifiziert werden. Auch hier werden die Klassennamen an die „MetadataFactory“ weitergeleitet. An dieser Stelle existiert keine Unterscheidung der Metadaten mehr. Nun werden für jede Klasse die Metadaten verarbeitet. Dieser Prozess beinhaltet neben der Prüfung des Namespaces auch das Laden der einzelnen Metadaten. An dieser Stelle werden treiber-spezifisch die Metadaten geladen. Hier existiert auch eine Unterscheidung von XML und YAML, da die Struktur anders interpretiert werden muss. An dieser Stelle wird die ganze Interpretation der Metadaten durchgeführt.

Abbildung 3.8 zeigt einen Ausschnitt aus der Klassenstruktur der Treiber. Dort erkennt man schnell den Unterschied zwischen den verschiedenen Mapping-Zweigen, die Methoden „getAllClassNames()“ für die Generierung der Klassennamen und das Laden der Metadaten für einzelne Klassen mit „loadMetadataForClass()“.
Nach der Interpretation der Metadaten wird das Schema aktualisiert. Ab dieser Stelle läuft das Mapping dann wieder parallel ab.

Die Vermutung liegt daher nahe, den Fehler des vorherigen Anwendungs-Abschnittes in den jeweiligen Treibern zu suchen und tatsächlich wird dort beim Laden der Metadaten der Fehler geworfen. Genauer gesagt sorgt die Funktion „loadMetadataForClass()“ für die

Mapping-Exception. Dort werden die vorher gesuchten Klassennamen geholt und für die XML-Verwendung, die dafür notwendigen Dateien, über einen FileLocator gesucht, diese findet er allerdings nicht, da die annotierten Klassen kein externes Mapping File nutzen. Da beide Metadaten verwendet wurden, wurden somit auch beide Treiber geladen und in der Ausführung des XML-Mappings entstand dieser Fehler.

3.2.2 Alternative Nutzung von Annotationen und XML

Neben dem Mapping können Annotationen und XML innerhalb Symfony2 an verschiedenen Stellen verwendet werden. An dieser Stelle soll auf diese Anwendungsmöglichkeiten eingegangen und erläutert werden.

XML in Symfony2

XML kann innerhalb Symfony2 an verschiedenen Stellen genutzt werden. Dazu zählen Konfigurationen, XML-Response und Routing.

Das Routing Das Routing ist eine zentrale Funktionalität von Symfony2. Ruft ein Nutzer eine URL in seinem Browser auf, wird über das Angeben der Route der URL-Pfad zu einem Controller gemappt. Bedeutet genauer, wird die URL angesprochen, wird eine Controller Aktion ausgelöst, die dann eine Response auslöst und die Seite anzeigt. Symfony2 bietet, mit XML, Annotationen und YAML mehrere Möglichkeiten Routen zu definieren. Hier ein Beispiel einer Route in XML:

Programmauszug 3.9: XML-Routing

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <routes xmlns="http://symfony.com/schema/routing"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
           instance"
4      xsi:schemaLocation="http://symfony.com/schema/
           routing
5          http://symfony.com/schema/routing/routing-1.0.
               xsd">
6
7      <route id="blog_show" path="/blog/{slug}">
8          <default key="_controller">AcmeBlogBundle:Blog
               :show</default>
9      </route>
10 </routes>
```

Eine neue Route wird durch das Tag „route“ neu deklariert. Es muss hier eine eindeutige Id, ein Pfad und die angesprochene Controller-Aktion angegeben werden.
In diesem Beispiel ruft der Nutzer, in seinem Browser, die Route mit dem Pfad „/blog/slug“ auf und löst damit die Aktion „showAction“ des BlogControllers im AcmeBlogBundle aus. Hierbei gilt die Besonderheit, dass mit slug dort eine Variable steht. Das bedeutet sowohl die URL „/blog/slug“ als auch „/blog/Hier-steht-ein-Slug“ werden durch die Route auf die selbe Aktion gemappt. Als Alternative zur Routendeklaration durch XML kann auch die

Auszeichnungssprache YAML verwendet werden.

Programmauszug 3.10: YAML-Routing

```
1 blog_show:
2     path:     /blog/{slug}
3     defaults: {_controller: AcmeBlogBundle:Blog:show}
```

Die Konfiguration Neben dem Routing, was streng genommen auch zu der Konfiguration zählt, bietet Symfony2 auch die Möglichkeit Konfigurationen als YAML oder als XML anzugeben. Neben externen Erweiterungen, wie beispielsweise PHPUnit, einem Testing-Framework, kann XML auch bei Symfony-internen Konfigurationen verwendet werden. Dazu zählen die Registrierung von Services, Registration von Translations (Übersetzungsdateien) und der Klassen-Validator, auf die hier nicht extra eingegangen werden sollen (Weiteres in [Sen14]). Auch die Hauptkonfiguration kann mittels XML angegeben werden.

In diesem Auszug der Hauptkonfiguration werden wichtige Daten-Imports aus anderen Konfigurationsdateien durch das „imports“ Tag gesetzt. Außerdem wird noch dem Router das Verzeichnis der Routing-Dateien zugewiesen und diverse Konfigurationen gesetzt.

Programmauszug 3.11: Auszug aus config.xml

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <container (...)>
3      <imports>
4          <import resource="parameters.yml" />
5          <import resource="security.yml" />
6      </imports>
7      <framework:config secret="%secret%">
8          <framework:router resource="%kernel.root_dir%/
               config/routing.xml" />
9      </framework:config>
10     <twig:config debug="%kernel.debug%" strict-
           variables="%kernel.debug%" />
11 </container>
```

Die Response Wird in Symfony2 vom Controller eine Response gesendet, erhält der Client ein Symfony2-spezifisches Response-Objekt. Es gibt dort mehrere Möglichkeiten eines solchen Objektes. Neben der einfachen Response existiert auch noch ein sogenanntes Redirect-Response-Objekt. Dieses Objekt sorgt für einen Aufruf einer neuen Route. Über das einfache Response-Objekt und die Templating Engine Twig, die in Symfony2 integriert ist, kann die Oberfläche geladen werden. Twig übersetzt die Templates und übergibt, durch das Response-Objekt, das gewünschte PHP-Skript an den Client als Response. Neben der Übergabe von Templates als Response kann auch hier eine XML-Response angegeben

werden. Da Symfony2 allerdings kein eigenes Objekt XMLResponse besitzt muss dies über das einfache Response-Objekt gelöst werden.

Programmauszug 3.12: XML-Response

```
1 public function showAction(){
2   $response = new Response();
3   $response->headers->set('Content-Type', 'xml');
4   return $this->render('AcmeBlogBundle:Blog:show.xml.
       php', array(), $response);
5 }
```

In diesem Beispiel wird ein neues Response-Objekt erstellt und in dessen Header festgelegt, dass dieses Objekt als XML interpretiert werden soll. Zeile 4 zeigt das Übersetzen von einem XML-Template, welches dann über die Response an den Client weitergeleitet wird.

Allerdings bietet sich an anstatt einer XML-Response eine Json-Response zu benutzen, da Symfony2 von Haus aus auch ein Json-Response-Objekt besitzt.

In Symfony2 besteht nicht nur die Möglichkeit XML-Responses zu senden, sondern auch zu verarbeiten. Dazu existiert der sogenannte Crawler, ein DOM-Parser, der einem ermöglicht bestimmte Elemente des XML's zu verarbeiten.

Programmauszug 3.13: Json-Response

```
1 public function showAction(){
2   $response = new JsonResponse();
3   return $response->setData(array('success' => false))
       ;
4 }
```

Annotation in Symfony2

In diesem Abschnitt soll auf die weitere Nutzung von Annotationen in Symfony2 eingegangen werden.

Eigene Annotationen Auch in Symfony2 können indes eigene Annotationen entworden werden. Dies ähnelt der Funktionalität in Java, auch hier muss die Annotation als Klasse erstellt werden und an der gewünschten Stelle angebracht werden. Die Funktionalität stellt hier ein Annotation Reader. Der Annotation Reader wartet auf ein Event das ausgelöst wird, wenn beispielsweise der Controller aufgerufen wird. Über das Event-Objekt lassen sich so Request-Eigenschaften vor Ausführung des Controller manipulieren. Das kann beispielsweise nützlich sein, falls bei gesetzter Annotation und Eintreten eines Falles auf einen anderen Controller geroutet werden soll. Als Beispiel kann hier die Unterscheidung von Ajax-Aufrufen und Framework-Aufrufen genannt werden.

Das Routing Auch über Annotationen lässt sich das Routing innerhalb eines Symfony2-Projektes definieren. Dafür existieren zwei Annotationen „@Route“ und „@Method“.

Programmauszug 3.14: Annotation-Routing

```
1 class BlogController extends Controller {
2   /**
3    * @Route("/blog/{slug}")
4    * @Method({"GET", "POST"})
5    */
6         public function showAction(\$slug) {}
7 }
```

Programmauszug 3.14 zeigt das Beispiel aus Programmauszug 3.10 als Controller-Annotation. Die Annotation „@Route“ definiert die Route, die auf diese Action zugreifen darf. Dazu kann über die Annotation „@Method“ definiert werden, welche Requests auf die Route zugegriffen werden darf.

Kapitel 4

Schluss

Der Hintergrund dieser Arbeit lag auf der Analyse von XML- und Annotationen-Nutzung innerhalb einer Reihe von Frameworks, doch welche Erkenntnisse konnten gezogen werden?
Die Anwendung der beiden Metadaten innerhalb von einer Datenbankerstellung der Frameworks ergab unterschiedliche Ergebnisse. Während in Hibernate es dem Nutzer ermöglicht wird beide Metadaten simultan zu verwenden, funktioniert diese Weise innerhalb von Symfony2 nicht. Demnach kann der Nutzer in Hibernate entscheiden, ob es für seinen Anwendungsfall nützlich ist beide Metadaten zu verwenden. In Symfony2 dagegen muss eine Entscheidung zwischen XML und Annotationen getroffen werden. Doch wie kann die mögliche Entscheidung begründet werden, daher werfen wir einen Rückblick auf die Analyse der Mächtigkeit.

Zunächst wurde eine Analyse der vorhandenen Annotationen und XML-Tags durchgeführt. Die Erkenntnis daraus war dennoch zu erwarten, denn es existieren syntaktisch ähnliche Anwendungsbezeichnungen in der Nutzung beider Metadaten. Es ergaben sich nur geringfügige Änderungen in der Anwendung der Funktionen. So wurden beispielsweise aus eigenen Annotationen Attribute anderer XML-Tags. Doch war nach dieser Aussage noch nicht möglich ein endgültiges Fazit über die Mächtigkeit beider Metadaten innerhalb des Mapping zu treffen. Daher wurde dann ein Anwendungsbeispiel eingeführt anhand deren Verarbeitung neu entschieden wurde. Doch auch hier ergab die Anwendung keinerlei Erkenntnis über eine unterschiedliche Mächtigkeit.
Diese Aussage gilt jedoch erst einmal nur für das Mapping von Symfony2, denn Hibernate stellt durch JPA und Hibernate-spezifisches Mapping mehrere Arten des Annotationen- und XML-Mapping bereit. Um hier doch noch eine Aussage über die Mächtigkeit treffen zu können, wurde daher versucht, ob das Mischen dieser beiden Arten für XML oder Annotationen zu Problemen führt. Dies würde bedeuten, die Mächtigkeit der Metadaten innerhalb des Mappings würden sich unterschieden. Und tatsächlich ist dieses Nutzen von JPA- und Hibernate-Mapping für Annotationen möglich. XML bietet dagegen kein allgemein nutzbares Schema zur Verarbeitung von JPA- und Hibernate-Funktionalität. Das bedeutet für die Mächtigkeit innerhalb von Hibernate, Annotationen bieten eine höhere Funktionalität und damit eine höhere Mächtigkeit als das jeweilige Äquivalent in XML. Jedoch stimmt diese Aussage nur solange die JPA-Funktionalität nicht komplett durch die Hibernate-Funktionalität abgedeckt ist.

An dieser Stelle besitzen wir nun eine mögliche Aussage über die Auswahl einer der beiden Metadaten, doch woran liegt es genau, dass die Nutzung innerhalb von Symfony2 eine vermischte Anwendung nicht unterstützt, Hibernate dagegen schon. Daher wurde ein Blick in die internen Abläufe der Metadatenverarbeitung des Mappings geworfen. Dort wurde sowohl für Symfony2 als auch für Hibernate klar, dass die Abläufe der Verarbeitung von XML- und von Annotationen im Großen und Ganzen übereinstimmen. Die Unterschiede in der Verarbeitung liegen an wenigen Stellen gekapselt.

So wird die Funktionalität in Symfony2 über spezifische Treiber gesteuert. Diese Treiber enthalten für die Metadaten vom Namen her dieselben Funktionen, die internen Verarbeitungen sind aber jeweils für den spezifischen Anwendungsfall angepasst. So werden auf der einen Seite über einen Annotation Reader die annotierten Klassen gesucht, auf der anderen Seite müssen die externen Dateien durch einen File Locator gesucht werden. Dementsprechend existiert dort auch für das Laden der Metadaten eine spezifische Funktionalität. Genau an dieser Stelle lässt sich auch das Problem in der Verwendung lokalisieren. Beim Laden der Metadaten innerhalb der Treiber werden die verschiedenen Mapping-Dateien fürs XML-Mapping gesucht, doch werden natürlich nicht alle gefunden. Diese fehlenden Mapping-Dateien sorgen für den Fehler in der Anwendung.

In Hibernate dagegen wird immer sowohl Annotation-Mapping als auch XML-Mapping ausgeführt. Es werden demnach sowohl Klassen aus XML-Mapping-Files, als auch annotierte Klassen gesucht und verwendet. Diese Handhabung macht es möglich innerhalb eines Projektes beide Metadaten zu vermischen.
Es lässt sich bei der Verarbeitung feststellen, dass für XML und Annotationen eigene Listen angelegt und auf die Elemente dieser, die spezifischen Funktionen aufgerufen werden. Beim Symfony2-Mapping existiert dagegen nur eine Liste mit allen Klassennamen. Zum Erstellen des Mappings anhand der Metadaten sind innerhalb von Hibernate die beiden Hilfsklassen HbmBinder und AnnotationBinder verantwortlich. Diese sorgen für die Erstellung des Mapping-Objektes das letztlich in die Datenbank gemappt wird.

Doch das Mapping ist nicht das einzige Anwendungsgebiet von XML und Annotation, denn es existieren noch andere Möglichkeiten diese zu nutzen. Diese Arbeit zeigt weitere Anwendungsmöglichkeiten von Annotationen und XML innerhalb der ausgewählten Frameworks. Wo verschiedene Anwendungsmöglichkeiten bestehen, findet sich auch eine größere Konkurrenz. So wurden außerdem verschiedene andere Möglichkeiten aufgezeigt, die die Frameworks bereitstellen, um dem Entwickler weitere Auswahlmöglichkeit aus dem Anwendungs-Pool mit XML und Annotationen zu bieten.

Kapitel 5

Ausblick

XML als Datenformat Die Programmierung ist stetig im Wandel und macht auch nicht vor den besten und arriviertesten Programmiertechniken halt. Mit der Einführung von XML wurde eine Standardisierung des Datentransfers festgelegt und es wurde schnell zum meistgenutzten Format zur Übertragung von Daten. Doch in den letzten Jahren zeichnete sich für XML ein negativer Trend ab, denn der eigentliche Vorteil von XML uneingeschränkt große Datenstrukturen bauen zu können, endet häufig im Verlust von Performance[NI09]. Der Trend ging in den letzten Jahren in die Richtung von performanteren und in der Nutzung leichter zu verstehenden Datenformaten, wie JSON. ProgrammableWeb veröffentlichte im Jahr 2011 eine Statistik in der sie eben jenen Trend feststellten.[Ada] Während viele Frameworks beide Datenformate zu Verfügung stellten[Sim], boten 20% der, in 2011 eingeführten, Frameworks lediglich einen JSON-Support an[Ten]. Gerade in der Nutzung innerhalb einer Umgebung mit Benutzeroberflächen bietet JSON Performance-Vorteile gegenüber XML. Denn wo Benutzeroberflächen existieren, existiert auch Javascript zum client-seitigen Verarbeiten von Benutzereingaben. Während XML innerhalb des Javascripts geparst werden muss, um Inhalte nutzbar machen zu können, kann JSON, alias JavaScript Object Notation, direkt innerhalb des Javascripts genutzt werden. Beispiele zum Performance-Vergleich zwischen XML und JSON finden Sie hier[NI09].

Durch die weitere Verbreitung von JSON zum Beispiel in den Bereich der Datenbanken wurde auch hier klar, dass es an manchen Stellen zu Problemen kommen könnte. Denn JSON fehlt beispielsweise ein Date-Objekt zur Übermittlung eines Datums. An dieser Stelle tritt das auch schon in Symfony2 genutzte YAML-Datenformat zum Vorschein.[Bin]

Der Trend geht also zu einfacheren und performanteren Datenformaten, allerdings befinden sich diese auch weiterhin im Wandel, wie einst XML. Daher ist schwer vorherzusagen, welches Datenformat in Zukunft genutzt werden wird. Letztlich bleibt die Entscheidung eine Auswahl nach persönlichen Vorlieben[Sim], oder wie David Megginson sagt[Meg]:

> „Personally, I like XML because it's familiar and has a lot of tool support, but I could easily (and happily) build an application based on any of the three — after all, once I stare long enough, they all look the same to me."

Annotationen und XML Der Trend in der Programmierung geht zur Nutzung von Annotationen gegenüber XML. XML war das Format zur Angabe von Metadaten eines

Programms bis zur Einführung von Annotationen im Jahr 2004. Während sich einige Frameworks zu einem anderen Datenformat entscheiden, anstatt XML zu nutzen, unterstützen die meisten Frameworks immer noch beide Arten der Metadatenverarbeitung. Dennoch wird das Arbeiten mit Annotationen von Entwicklern generell als verständlicher und leserlicher eingeschätzt, denn Annotationen liegen nicht in externen Dokumenten, sondern liegen direkt im Quellcode. Dennoch ist die Entscheidung XML oder Annotationen zu verwenden, eine Frage über die sich die Geister scheiden und letztlich eine Art des Geschmacks, denn es existieren für beide mögliche Anwendungsszenarien. Am besten auf den Punkt bringt es allerdings folgendes Zitat eines Users der Programmierplattform stackoverflow.com[Use]:

> „Neither one is better, and so both are supported, although annotations are more fashionable. As a result, new hair-on-fire frameworks like JPA tend to put more emphasis on them. More mature APIs like native Hibernate offer both, because it's known that neither one is enough."

Bibliography

[Ada] ProgrammableWeb Adam DuVander. *1 in 5 Api's say "Bye XML"*. URL: http://www.programmableweb.com/news/1-5-apis-say-bye-xml/2011/05/25.

[Bin] Andrew Binstock. *After XML, JSON: Then What?* URL: http://www.drdobbs.com/web-development/after-xml-json-then-what/240151851.

[Cuo08] Bc. Nguyen Viet Cuong. "Object-relational mapping Hibernate and Struts framework in Java". MA thesis. Czech Technical University Prague, 2008.

[Doy10] Matt Doyle. *Beginning PHP 5.3*. Bonn, Germany: Galileo Computing, 2010.

[Eck06] Bruce Eckel. *Thinking in Java Fourth Edition*. New Jersey, USA: Prentice Hall, 2006.

[Ed 03] Bhakti Metha Ed Ort. *Java Architecture for XML Binding (JAXB)*. Mar. 2003. URL: http://www.oracle.com/technetwork/articles/javase/index-140168.html.

[itwa] itwissen.info. *Auszeichnungssprache*. URL: http://www.itwissen.info/definition/lexikon/Auszeichnungssprache-ML-markup-language.html.

[itwb] itwissen.info. *Skriptsprache*. URL: http://www.itwissen.info/definition/lexikon/Scriptsprache-script-language.html.

[Meg] David Megginson. *All markup ends up looking like XML*. URL: http://quoderat.megginson.com/2007/01/03/all-markup-ends-up-looking-like-xml/.

[NI09] Randall Reynolds Nurzhan Nurseitov Michael Paulson and Clemente Izurieta. *Comparison of JSON and XML Data Interchange Formats: A Case Study*. Tech. rep. Department of Computer Science - Montana State University – Bozeman, Montana, 59715, USA, 2009.

[Sen14] Sensiolabs. *Symfony - The Book*. Sensiolabs, 2014.

[Sim] ProgrammableWeb Simon Hamp. *Json Developers Choice*. URL: http://www.programmableweb.com/news/json-developers-choice/2010/08/11.

[Tea] Doctrine Project Team. *Doctrine 2 ORM Documentation*.

[Ten] Roy Tennant. *JSON Replacing XML for API Responses*. URL: http://www.thedigitalshift.com/2011/05/roy-tennant-digital-libraries/json-replacing-xml-for-api-responses/.

[Ull12] Christian Ullenboom. *Java ist auch eine Insel*. Indianapolis, USA: Wiley Publishing, 2012.

[Use] stackoverflow.com User skaffman. *XML configuration versus Annotation based configuration*. URL: http://stackoverflow.com/questions/182393/xml-configuration-versus-annotation-based-configuration.

[wik] wikipedia.org. *XDoclet*. URL: http://de.wikipedia.org/wiki/XDoclet.

Anhang

Referenz-Implementationen

Hibernate

Programmauszug A.1: Annotation-Mapping Klasse BaseUser

```
1  ...
2  @Entity
3  @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
4  @Table(name="allUsers")
5  @DiscriminatorColumn(name="type", discriminatorType=
       DiscriminatorType.STRING)
6  @DiscriminatorValue("baseUser")
7  public class BaseUser {
8
9          @Id @GeneratedValue
10         @Column(name="id")
11         protected Integer id;
12
13         @Column(nullable=true)
14         protected Date createdAt;
15
16         @Column(length=50, unique=true)
17         protected String username;
18
19         @ManyToMany(targetEntity=Role.class, mappedBy=
               "users")
20         @OrderBy("name ASC")
21         protected Set<Role> roles;
22
23         @ManyToOne(targetEntity=Person.class, cascade=
               javax.persistence.CascadeType.ALL)
24         @JoinColumn(name="person_id",
               referencedColumnName="id")
25         protected Person person;
26
27         ...
28 }
```

Programmauszug A.2: XML-Mapping Klasse BaseUser

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <entity-mappings
4    xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
       "
6    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/
       persistence/orm http://xmlns.jcp.org/xml/ns/
       persistence/orm_2_1.xsd"
7    version="2.1">
8
9    <entity class="com.javacodegeeks.enterprise.
       hibernate.BaseUser" >
10     <inheritance strategy="SINGLE_TABLE"/>
11     <discriminator-column name="type" discriminator-
         type="STRING"/>
12     <discriminator-value>baseUser</discriminator-value
         >
13     <table name="allUsers"></table>
14     <attributes>
15             <id name="id"><generated-value/></id>
16       <basic name="createdAt">
17         <temporal>DATE</temporal>
18         <column nullable="true"/>
19       </basic>
20       <basic name="username">
21         <column length="50" unique="true"/>
22       </basic>
23       <many-to-one name="person" target-entity="Person
           ">
24         <join-column name="person_id" referenced-
             column-name="id"/>
25         <cascade><cascade-all/></cascade>
26       </many-to-one>
27       <many-to-many name="roles" target-entity="Role"
           mapped-by="users">
28         <order-by>ASC</order-by>
29       </many-to-many>
30     </attributes>
31   </entity>
32 </entity-mappings>
```

Symfony2

Programmauszug A.3: Annotation-Mapping Klasse BaseUser

```
1  <?php
2  ...
3  /**
4   * @ORM\Entity
5   * @ORM\InheritanceType("SINGLE_TABLE")
6   * @ORM\Table("allUsers")
7   * @ORM\DiscriminatorColumn(name="type", type="string
       ")
8   * @ORM\DiscriminatorMap({"baseuser" = "BaseUser", "
       user" = "User"})
9   * @ORM\HasLifecycleCallbacks
10  */
11 class BaseUser {
12         /**
13          * @ORM\Id()
14          * @ORM\GeneratedValue()
15          * @ORM\Column(name="id", type="integer")
16          */
17         protected $id;
18         /**
19          * @ORM\Column(type="datetime", nullable=true)
20          */
21         protected $createdAt;
22         /**
23          * @ORM\Column(type="string", length=50,
               unique=true)
24          */
25         protected $username;
26         /**
27          * @ORM\ManyToMany(targetEntity="Role",
               mappedBy="users")
28          * @ORM\JoinTable(name="userRoles")
29          * @ORM\OrderBy({"name" = "ASC"})
30          */
31         protected $roles;
32         /**
33          * @ORM\ManyToOne(targetEntity="Person",
               inversedBy="users", cascade={"all"})
34          * @ORM\JoinColumn(name="person_id",
               referencedColumnName="id")
35          */
36         protected $person;
37
38         ...
39 }
```

Programmauszug A.4: XML-Mapping Klasse BaseUser

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <doctrine-mapping>
3    <entity name="BA\BABundle\Entity\BaseUser" table="
       allUsers" inheritance-type="SINGLE_TABLE">
4      <discriminator-column name="type" type="string" />
5      <discriminator-map>
6        <discriminator-mapping value="baseUser" class="
           BaseUser" />
7        <discriminator-mapping value="user" class="User"
           />
8      </discriminator-map>
9      <id name="id" type="integer" column="id">
10       <generator strategy="AUTO"/>
11     </id>
12     <field name="createdAt" column="createdAt" type="
         datetime" nullable="true" />
13     <field name="username" column="username" type="
         string" length="50" unique="true" />
14     <many-to-many field="roles" target-entity="Role"
         mapped-by="users">
15       <order-by>
16         <order-by-field name="name" direction="ASC" />
17       </order-by>
18     </many-to-many>
19     <many-to-one field="person" target-entity="Person"
         inversed-by="users">
20       <cascade>
21         <cascade-all/>
22       </cascade>
23       <join-columns>
24         <join-column name="person_id" referenced-
             column-name="id" />
25       </join-columns>
26     </many-to-one>
27   </entity>
28 <doctrine-mapping>
```