FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Validierung von Benutzereingaben für Web 2.0 CRUD Anwendungen mittels eines modell-getriebenen Ansatzes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Manuel Blauensteiner
Matrikelnummer 0125174

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.-Prof. Dr. Uwe Zdun
Mitwirkender: Dipl.-Ing. Dr.techn. Project Assistant Ernst Oberortner

Wien, July 14, 2011 _____ (Unterschrift Verfasser/in) _____ (Unterschrift Betreueung)

Danksagung

Diese Diplomarbeit wäre ohne die Unterstützung vieler Personen nicht zustande gekommen. Deswegen möchte ich mich bei folgenden Menschen bedanken:

Zuerst möchte ich meinen Eltern, Gisela und Gerhard danken, die mich vom ersten Tag meines Studiums tatkräftig unterstützten.

Des weiteren möchte ich mich bei meinem Betreuer Ernst Oberortner für die erfolgreiche Umsetzung dieser Arbeit bedanken.

Zuletzt möchte ich noch all meinen Studien Kollegen, für ihre tatkräftige Zusammenarbeit, der sehr guten Teamarbeit und ihrer großen Hilfsbereitschaft über die ganzen Jahre danken.

Erklärung

Manuel Blauensteiner
Blumauergasse 20/8
1020 Wien

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, am 14. Juli 2011

Manuel Blauensteiner

Abstract

In the most Web 2.0 CRUD applications exists administration pages to manage the web sites' data. The problem is, that this part for every CRUD application will always developed anew, although the code is still very similar and follows the same schema. Unfortunately, the well known copy & paste behavior is used very often. All this contradicts every aspect of software reusability.

This thesis focused on automatic code generation of the administration pages. For this purpose a model driven developed approach will be used. The main task of this thesis is to define a metamodel, making it possible, to design such management pages. To make it possible to generate this pages, a code generator will be developed. A further task is the validation of the input data on this pages. These constraints also can be defined with the metamodel, which are used for the validation later. The generated web application is based on the principles of the model view controller (MVC) pattern. So the domain objects (conform to the model), all graphical views (view) and the logic (controller) will be generated automatically.

To demonstrate the feasibility of the approach of generating web applications, a prototype is introduced. This prototype uses *JavaServer Faces* (JSF) as the web framework and can be deployed on every application server. To get profound knowledge of the quality, this approach was quantitative evaluated on reusability.

Kurzfassung

In den meisten Web 2.0 CRUD Anwendungen existiert ein so genannter Administrationsbereich, in dem die Daten der Webanwendung verwaltet werden. Das Problem hierbei ist, dass dieser Bereich für jede CRUD Anwendung immer neu entwickelt wird, obwohl das Prinzip der Datenverwaltung jedes mal dem gleichen Schema folgt. Hierfür wird oft das bekannte Copy & Paste Verhalten angewendet. Dieses widerspricht jedoch jeglichen Aspekten der Software Wiederverwendung.

Diese Diplomarbeit beschäftigt sich mit der automatischen Generierung von Administrationsbereichen. Hierfür wird ein modell-getriebener Ansatz verfolgt. Insbesondere konzentriert sich diese Arbeit auf die Definition eines Metamodells, mit dem die Modellierung der Verwaltungsseiten möglich ist. Um die automatische Generierung der Verwaltungsseiten zu ermöglichen wird ein Codegenerator implementiert. Um die Funktionalität der Validierung der Eingabedaten zu ermöglich, wird das Metamodell entsprechend erweitert. Die generierte Webanwendung folgt dem *Model View Controller* (MVC) Pattern. Es werden somit alle Datenobjekte (Model), die grafischen Oberflächen der Webseiten (View) und die Navigation durch die Webseiten (Controller) automatisch erstellt.

Um die Arbeitsweise dieses Ansatzes zu demonstrieren wurde ein Prototyp entwickelt, welcher auf der *JavaServer Faces* (JSF) Technologie basiert. Die Webanwendung kann auf jedem beliebigen Applikationsserver ausgeführt werden. Um fundierte Aussagen über die Qualität des Ansatzes zu erhalten wurde eine quantitative Evaluierung bezüglich der Wiederverwendbarkeit durchgeführt.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Einleitung

Der Hauptfokus dieser Arbeit liegt in der Entwicklung eines Generators basierend auf einem modell-getriebenen Ansatz, mit dem Web 2.0 CRUD[1] Anwendungen generiert werden können. Ein weiterer wichtiger Aspekt ist die Eingabe Validierung der Daten. Zur Demonstration und Veranschaulichung der Funktionalität wurde der komplette Code-Generator entwickelt.

Die CRUD Anwendung basiert auf dem MVC Pattern [22] und verwendet als konkrete Implementierung JSF. Das MVC Pattern trennt die Darstellung, die Daten und die Definition des Page Flows. Dies hat den entscheidenden Vorteil, dass es keine direkten Abhängigkeiten der einzelnen Komponenten zu einander gibt. Durch das Modell wird die komplette Verwaltungsseite mit einer Validierung beschrieben. Es wird für jedes definierte Objekt eine Listendarstellung, eine Ansicht-Seite und eine Bearbeiten-Seite generiert. Das Modell selbst wird mit einer *XML Metadata Interchange* (XMI) Datei beschrieben, welche als als alleiniger Input für den Generator dient, der alle Artefakte generiert.

1.2 Problemstellung

Web 2.0 Anwendungen [10] [41] [61] sind komplexe Softwareapplikationen die sich mit den aktuellen Technologien beschäftigen. In vielen Webanwendungen existiert ein so genannter Administrationsbereich, in dem die Daten der Webanwendung verwaltet werden. Das Problem hierbei ist, dass dieser Administrationsbereich immer wieder neu entwickelt wird, obwohl es jedes mal dem gleichen Schema folgt. All dieses widerspricht jeglichen Aspekten der Wiederverwendung, Qualität und Sicherheit. Im Administrationsbereich wird neben den eigentlichen Seiten, auf denen die Daten bearbeitet werden, auch eine Übersichtseite erstellt, mit der eine Suche möglich ist. Da die Webanwendung die im Administrationsbereich bearbeiteten Daten verwendet und weltweit im Internet verfügbar ist, ist eine Validierung der Daten unbedingt erforderlich.

1.3 Motivation

Web Anwendungen sind in der heutigen Zeit nicht mehr wegzudenken. Sehr viele Firmen versuchen sich im Web zu etablieren. Den entscheidenden Vorteil den diese Technologie bietet ist die Erreichbarkeit aller Menschen weltweit.

Einer der Hauptgründe für die Verwendung von MDD ist, die Zeit für die Erstellung von Web Anwendungen drastisch zu reduzieren und zugleich die Fehleranfälligkeit zu verringern. Weiters wird eine wesentliche höhere Qualität der Software erzielt.

[1] Create Read Update Delete

Zuletzt wird die Wartung solcher Web Anwendungen sehr vereinfacht, da die verwendeten Modelle immer der aktuellen Softwareimplementierung entsprechen, weil eben diese automatisch aus den Modellen generiert werden.

Die eigentliche Motivation liegt in der Abschaffung dieser Neuimplementierung von Verwaltungsseiten. Alleine einfache Web Oberflächen, mit der Domänenobjekte verwaltet werden können, setzen schon eine komplexe Anwendung voraus. Dieses fällt mit dem Generator komplett weg.

Ebenso soll dieses Copy & Paste Verhalten unterbunden werden. Der übliche Ablauf der Implementierung von Verwaltungsseiten sieht folgendermaßen aus. Zuerst werden alle Ansichten für z.B. das Objekt Person erstellt. Nachdem diese Seiten implementiert wurden, wird genau dieser Code für das nächste Objekt z.B. Adresse verwendet. Wenn n Domänenobjekte erstellt werden und nachträglich eine Änderung in der Ansicht notwendig wird, müssen auch diese n Seiten angepasst werden. Mit einem Code Generator wäre diese Anpassung lediglich an einer einzigen Stelle notwendig. [8]

1.3.1 Motivations-Beispiel

Die Vorteile dieses Ansatzes sollen durch ein Motivations-Beispiel demonstriert werden. Für viele Webanwendungen ist ein Benutzerkonzept unumgänglich. Ob es sich um Mitarbeiter einer Firma, Studenten einer LVA oder einfach nur Benutzer eines Forums handelt. In solchen Webanwendungen muss man zuerst einen Benutzer besitzen, um erst die eigentlichen Funktionalitäten nutzen zu können. Es ist dabei egal, ob sich die Benutzer selbst registrieren und verwalten können. Es ist fast immer auch erforderlich eine Administrationsoberfläche zu entwerfen, auf der so eine Bearbeitung der Benutzer möglich ist. Aus diesem Grund soll auch eine Personenverwaltung als Motivations-Beispiel dienen. In Kapitel *Evaluierung* 5 wird dieses Beispiel herangezogen. Es wird dabei eine einfache Person mit mehreren möglichen Adressen abgebildet.

1.4 Erwartetes Resultat

Durch ein eigens entwickeltes Framework soll dieses Softwaremodul weitgehend automatisch generiert werden. Das Framework soll eine Übersichtseite, eine Detailseite zur Ansicht und eine Bearbeitenseite der einzelnen Objekte, welche die Daten beinhalten, automatisch erstellen können. Die Abhängigkeiten zwischen den verschiedenen Objekten sollen ebenfalls mit berücksichtigt werden. Neben den generierten Artefakten, müssen darüber hinaus diverse Konfigurationsdateien erstellt werden. Weiters ist zu berücksichtigen, dass eine korrekte Ordnerstruktur generiert wird, um die ganze Webapplikationen auch in einem Java Servlet Container laufen lassen zu können.

1.5 Methodisches Vorgehen

Diese Arbeit baut auf der Diplomarbeit „Generating web applications with abstract pageflow models“ [42] auf, welche einen modell-getriebenen Ansatz (MDA) [55] verfolgt. Da das vorhandene Domänenmodell allerdings die Objekte anhand der Webseite beschreibt, wird ein neues Metamodell entworfen. Mit diesem soll es möglich sein, neben der Abbildung der Objekte zusätzlich Validierungseigenschaften definieren zu können. Um eine exemplarische Darstellung des Prototypen zu zeigen, wird eine Beispielanwendung generiert, mit der es möglich ist Personen im Administrationsbereich zu verwalten. Für die Evaluierung wird nach absoluten Größen, relativen Größen, Komplexität, Funktionalität und Wiederverwendung mit bereits bestehenden Ansätzen verglichen [34].

Diplomarbeit Oberortner

Diese Arbeit verwendet einige Ansätze aus der Diplomarbeit *Oberortner* [42]. Die Motivation dieser Arbeit war im wesentlichen die Beliebtheit von Webapplikationen. Durch einen modell-getriebenen Ansatz sollte die Entwicklungszeit verkürzt und die Qualität erhöht werden. Die Hauptidee dieses Ansatzes war, dass die eigentlichen Webseiten modelliert werden. Um dies bewerkstelligen zu können wurde ein eigenes Metamodell entworfen. Mit diesem Metamodell können unterschiedliche Seiten mit ihren Textfeldern beschrieben werden. Dies wird ebenfalls wie in dieser Arbeit dargestellten Ansatzes in einer XML Datei definiert. Es wird in beiden Arbeiten der *openArchitectureWare* Generator verwendet.

Das Problem dieser Arbeit ist, dass die Domänen Objete implizit durch die Definition der Seiten in einem Modell festgelegt werden. Das Ziel des aktuellen Ansatzes ist eine Verwaltungsseite zu generieren, d.h. es werden letztendlich Objekte manipuliert und in einer Datenbank persistiert. Für diesen Fall ist das in dem Ansatz [42] entworfene Metamodell nicht brauchbar. Stattdessen ist es hier sinnvoller ein reines Domänenmodell zu entwerfen.

1.6 Strukturierung der Arbeit

Kapitel 2 - State of the Art Review

In Kapitel 2 *State of the Art Review* wird auf die aktuellen Technologien eingegangen, die in dieser Arbeit verwendet bzw. in Kapitel 5 verglichen werden. Da der CRUD Ansatz auf JSF aufbaut, wird auf diese Technologie umfassender eingegangen. Weiters werden die Technologien Grails und Seam beschrieben, die im Bereich Web Entwicklung MDD Ansätze verfolgen.

Kapitel 3 - Related Work

In Kapitel 3 *Related Work* wird auf die modell-getriebenen Ansätze wie dem UWE Ansatz, WebML, WebDSL oder Topcased eingegangen.

Kapitel 4 - Entwurf und Implementierung

In Kapitel 4 *Entwurf und Implementierung* wird der eigene Ansatz beschrieben. Es wird auf das eigens entworfene Metamodell und auf die Code Generierung eingegangen. Die Generierung selbst lässt sich in 4 Bereiche gliedern: Java, JSF, Persistierung und Validierung. Neben der Codegenerierung wird auch auf das speziell für diesen Ansatz entwickelte CRUD Framework eingegangen. Dieses Grundgerüst wird vom generierten Code verwendet.

Kapitel 5 - Evaluierung

In Kapitel 5 *Evaluierung* wird der eigene Ansatz mit Grails und Seam verglichen. Ein sehr starkes Augenmerk wird auf das Wachstum des Modells gelegt. Es wird genau beschrieben wie sehr das Modell in Abhängigkeit von Attributen wächst. Ebenso werden alle generierten Dateien verglichen. Genauer gesagt werden die Anzahl der generierten Zeilen gegenübergestellt.

Kapitel 6 - Ergebnis und zukünftige Arbeit

In Kapitel 6 *Ergebnis und zukünftige Arbeit* wird noch einmal kurz auf alle relevanten Aspekte und auf die mögliche Weiterentwicklung des CRUD Ansatzes eingegangen.

KAPITEL 2

State of the Art Review

2.1 CRUD Anwendungen

CRUD, *Create (Anlegen)*, *Read (Lesen)*, *Update (Aktualisieren)* und *Delete (Löschen)* beschreibt in der Informatik die 4 grundlegenden Datanbankoperationen. Eine CRUD Anwendung ist eine Software Applikation, die diese Funktionalität unterstützt. Als eine sehr einfache CRUD Anwendung könnte jeder SQL [1] Client angesehen werden. Hier besteht die Möglichkeit genau das gewünschte Verhalten direkt mit SQL Befehlen durchzuführen. Da vor allem bei CRUD Anwendungen die Benutzerfreundlichkeit im Vordergrund steht, werden unter dem Namen somit nur komplexe Software Anwendungen gesehen. Es ist dabei egal, ob es sich um Desktop oder Web Anwendungen handelt. Um ein Mindestmaß an Benutzerfreundlichkeit zu bieten werden von eigentlich allen Anwendungen ebenso Listendarstellungen wie Eingabevalidierungen unterstützt. In Kapitel *Aktuelle Technologien* 2.3 werden unter anderem die CRUD Webframeworks Seam und Grails beschrieben.

2.2 Model View Controller

MVC *(Model (Modell), View (Präsentation), Controller (Steuerung))* [29] ist ein Architekturmuster um die Struktur in Softwareanwendungen in einzelne Module zu zerlegen. Ziel ist es, durch gezielte Trennung des Codes eine bessere Codequalität, einfachere Wartbarkeit und eine erhöhte Wiederverwendbarkeit zu gewinnen. In Abbildung 2.1 ist das Konzept des MVC Musters abgebildet.

2.2.1 Model1 Architektur

Im Jahre 1997 wurde von Sun die Servlet Technologie erfunden. Mittels JSP[2] Seiten konnte HTML [3] und Javacode in einer Datei miteinander kombiniert werden. Mit Einführung von JSP konnte somit viel leichter eine dynamische Webseite in Java entwickelt werden, allerdings stieg die Komplexität sehr rasch mit der Anzahl der Codezeilen. Dies wiederum führte zu unleserlichen JSP-Seiten. Da JSP-Seiten weder HTML noch Java waren, mussten diese 2 Sprachen übersetzt werden. Es wurde somit für jede einzelne JSP-Seite eine Javaklasse, welche auch Servlet genannt wird generiert. Diese eine Java Klasse greift nun auf das verwendete Modell zu. Gleichzeitig wird die Ansicht mit so genannten *OutputStream* Methoden definiert. Somit ist der Controller, die Präsentation und das Modell in einer Klasse vereint, was eine effektive Code Entwicklung oder Wartbarkeit unmöglich macht.

[1] Structured Query Language
[2] Java Server Pages
[3] Hypertext Markup Language (HTML, dt. Hypertext-Auszeichnungssprache)

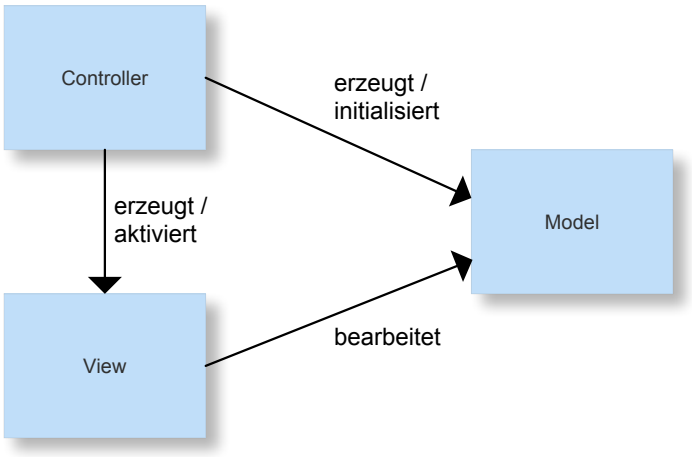

Abbildung 2.1: MVC Entwurfsmuster

2.2.2 Model2 Architektur

Da die Model1 Architektur schnell an ihre Grenzen stieß, wurde das bereits weit verbreitete Architekturmuster MVC in das Web übertragen. Die Model2 Architektur kann somit als Spezialisierung der Model-View-Controller Architektur gesehen werden.

Der Unterschied zu Model1 ist, dass kein Servlet als Container verwendet wird, stattdessen wird ein *Managed-Bean* als Controller verwendet. Ein Managed-Bean ist ein Javaklasse, die Anfragen vom Webclient entgegen nimmt und meistens auch die Entität zu Verfügung stellt (siehe Kapitel 2.3.2). In der Model2 Architektur wird lediglich genauer spezifiziert was das Modell, was die Präsentation und was die Steuerung ist (siehe Abbildung 2.2).

2.3 Aktuelle Technologien

2.3.1 Hibernate

Hibernate [24] selbst ist ein mächtiges ORM[4] Framework. Es dient dazu Java POJOs[5] in eine Relationale Datenbank abzubilden. Die POJOs müssen laut Definition lediglich einen parameterlosen Konstruktor besitzen. Wenn kein zusätzlicher Konstruktor definiert wurde, wird von Hibernate der Standard-Konstruktor verwendet. Die Attribute liest

[4]Objekt relationale Abbildung (engl. object-relational mapping, ORM)
[5]Plain Old Java Object, also ein ‘ganz normales’ Objekt in der Javawelt

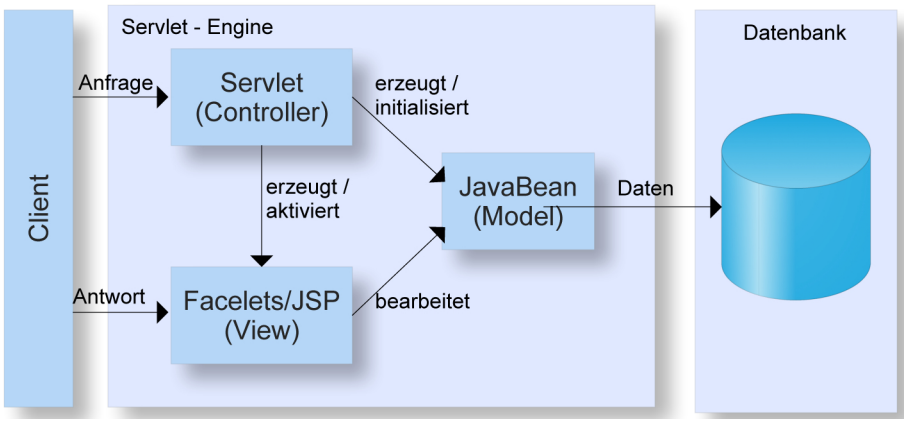

Abbildung 2.2: Model2 Entwurfsmuster

Hibernate per Reflection aus. Das Mapping selbst wird per Java Annotation oder mittels XML Dateien spezifiziert. Die erste Variante ist erst seit der Java Version 5.0 mit Einführung der Annotation möglich. Hibernate unterstützt folgende Objektreferenzen 1:1, 1:N, N:1, M:N. Weiters können Vererbungsbeziehungen zwischen mehreren Klassen abgebildet werden. Beim Laden von Objekten können die in Beziehung stehenden Objekte wahlweise sofort mit geladen werden (*eager loading*), oder erst wenn sie tatsächlich gebraucht geladen werden (*lazy loading*). In Listing 2.1 ist eine mögliche Entität, welches mit JPA[6] [5] Annotationen erweitert wurde abgebildet. Die Annotation *@Entity* bewirkt, dass dieses Java Bean in einer Datenbank abgebildet werden kann. *@Id* definiert in der relationalen Welt den eindeutigen Identifikator für dieses Objekt. Somit wird für dieses Attribut eine Primär-Spalte in der Tabelle angelegt. Standardmäßig werden die Attribute des Objektes mit dem selben Namen in der Datenbank abgebildet. Diesen Namen kann man aber mit der Annotation *@Column(name="<NAME_DER_SPALTE>")* festlegen.

```
1  @Entity
2  public class Person {
3
4    @Id
5    private Long id;
6
7    @Column(name="name")
8    private String name;
9
10   // Getter & Setter
11 }
```

Listing 2.1: POJO mit JPA Annotationen

[6] Java Persistence API

Hibernate Validator

Hibernate Validator [25] ist ein Validierungs Framework von Hibernate, welches auf den Java Bean Validation (JSR [7] 303) Standard setzt, um Objekte effektiv und einfach validieren zu können. Hierzu werden einfach jene Attribute, die bestimmte Eigenschaften besitzen müssen (z.B. maximal 30 Zeichen) mit eigenen Validierungs-Annotationen versehen. Eine Liste mit diesen Einschränkungen ist in Tabelle 2.1 zu finden. Im Listing 2.2, wird das vorige Beispiel *POJO mit JPA Annotationen 2.1* mit zusätzlichen Validierungs-Annotation wie *@NotNull @Length(min=3, max=20)*, *@Pattern(regex="[a-zA-Z]+")*, etc. erweitert. Der Name der Person muss jetzt folgende Eigenschaften erfüllen, um die Validierung zu bestehen.

1. Der Name darf nicht *null* sein.
2. Der Name muss mindestens 3 und darf nicht mehr als 20 Zeichen lang sein.
3. Der Name darf nur kleine und große Buchstaben enthalten.

```
1  @Entity
2  public class Person {
3
4    @Id private Long id;
5
6    @NotNull
7    @Length(min=3, max=20)
8    @Pattern(regex="[a-zA-Z]+")
9    private String name;
10
11   @NotNull
12   @Email
13   private String email;
14
15   @Past private Date birthday;
16 }
```

Listing 2.2: Entity Bean mit Validierungs-Annotationen

Internationalisierung

Für jede Hibernateannotation gibt es eine Standardfehlermeldung. Diese Meldungen sind internationalisiert und werden für eine handvoll Sprachen mit ausgeliefert. Die Sprachdateien befinden sich in der Datei *hibernate-validator.jar* im Verzeichnis */org/hibernate/validator/resources*. Um eine Sprache zu setzen ist dem Validator per Konstruktor ein ResourceBundle mit zu übergeben. Die deutsche Sprachdatei *DefaultValidatorMessages_de.properties* ist in Listing 2.3 abgebildet.

[7] Java Specification Request

Tabelle 2.1: Validierungs-Annotationen

Annotation	Typ	Überprüfung auf
@CreditCardNumber	String	Prüft ob der String einer gültigen Kreditkartennummer entspricht
@Email	String	Prüft ob der String einer Email laut Spezifikation entspricht
@Future	Date, Calendar	Prüft ob das Datum in der Zukunft liegt
@Length(min, max)	String	Prüft die Anzahl der Zeichen
@Max(value)	Zahl	Prüft ob der Wert <= der Zahl ist
@Min(value)	Zahl	Prüft ob der Wert >= der Zahl ist
@NotEmpty	Alle Typen	Prüft ob der Wert nicht *null* und zudem nicht leer (Leerstring) ist
@NotNull	Alle Typen	Prüft ob der Wert nicht *null* ist
@Past	Date, Calendar	Prüft ob das Datum in der Vergangenheit liegt
@Pattern(regex)	String	Prüft ob der Wert einem regulären Ausdruck entspricht
@Range(min, max)	Zahl	Prüft ob die Zahl zwischen min und max liegt
@Size(min, max)	Array, Collection, Map	Prüft ob die Anzahl der Elemente zwischen min und max liegt
@Valid	Objekt	Validiert dieses Objekt ebenfalls

```
1  validator.assertFalse=nicht garantiert
2  validator.assertTrue=garantiert
3  validator.future=muss in der Zukunft liegen
4  validator.length=muss zwischen {min} und {max} lang sein
5  validator.max=muss weniger oder gleich {value} sein
6  validator.min=muss mehr oder gleich {value} sein
7  validator.notNull=kann nicht leer sein
8  validator.past=muss in der Vergangenheit liegen
9  validator.pattern=muss Ausdruck "{regex}" entsprechen
10 validator.range=muss zwischen {min} und {max} sein
11 validator.size=muss zwischen {min} und {max} gross sein
12 validator.email=not a well-formed email address
13 validator.notEmpty=kann nicht null oder leer sein
14 validator.digits=numeric value out of bounds (<{integerDigits} digits>.<{
       fractionalDigits} digits> expected)
15 validator.ean=invalid EAN
16 validator.creditCard=Ungültige Kreditkartennummer
```

Listing 2.3: Standard Fehlermeldungen bei fehlgeschlagener Validierung

Es kann aber jede gesetzte Annotation mit einer selbst definierten Fehlermeldung versehen werden. Dafür dient das *message* Attribut. Hier ein Beispiel für das Verwenden

von eigenen Fehlermeldungen:

```
1 @Length(min=3, max=20, message="Name muss zwischen 3 und 20 Zeichen lang
      sein")
2 private String name;
```

Listing 2.4: Hibernateannotation mit Messageattribut

Um eigene internationalisierte Fehlermeldungen zu erstellen sind 2 Schritte notwendig.

1. *Messages.properties ist zu erstellen*
   Es ist eine *messages.properties* Datei im Classpath zu erstellen und der gewünschte Key zu setzen, bzw. einen eigenen zu definieren.

   ```
   nameLength=Name muss zwischen 3 und 20 Zeichen lang sein
   ```

   Es besteht auch die Möglichkeit die generierten Meldungen dynamisch zu erstellen. Alle möglichen Attribute, die für die Validierung verwendet werden (z.B. *min* und *max*) können hier verwendet werden.

2. *Message Attribut ist zu setzen*
   Es muss bei der gewünschten Validierungs-Annotation das *message* Attribut folgendermaßen gesetzt werden: $\{< KEY >\}$

   ```
   @Length(min=3, max=20, message="{nameLength}")
   private String name;
   ```

Validierung
Um annotierte Entitäten validieren zu können, muss eine Instanz der Klasse *ClassValidator* erstellt werden. Dem Konstruktor *ClassValidator(Class<T> beanClass)* muss die Klasse, die zu validieren ist übergeben werden. Optional kann mit einem *ResourceBundle* die Internationalisierung konfiguriert werden. Es gibt folgende Möglichkeiten um eine Entität zu validieren. Alle Validierungsmethoden liefern immer ein Array des Typs *InvalidValue* zurück. Bei keinem Validierungsfehler wird eine leere Liste zurückgeliefert.

1. *Validierung der gesamten Entität*
   Mit der Methode *getInvalidValues(T bean)* wird ein komplettes Bean validiert. Dh. es werden alle Attribute, die mit einer entsprechenden Validierungs-Annotation versehen sind, berücksichtigt und überprüft. Die Rückgabe beinhaltet somit alle vorhandenen Fehler.

2. *Validierung eines Attributes*
   Mit der Methode *getInvalidValues(T bean, String propertyName)* wird lediglich

ein einzelnes Attribut eines Beans validiert. Es können hier genauso mehrere Fehler zurückgeliefert werden (siehe Listing 2.5). Das Attribut *Name*, muss 2 Bedingungen erfüllen.

3. *Validierung eines Attributes mit übergebenen Wert*
   Mit der Methode *getPotentialInvalidValues(String propertyName, Object value)* kann überprüft werden, ob eine Validierung mit einem bestimmten Wert fehlschlagen würde.

```
1  public class Test {
2
3    public static void main(String[] args)
4    {
5      InvalidValue[] invalidValues = null;
6      Person person = new Person();
7      person.setName("I4");
8
9      ClassValidator<Person> validator = new ClassValidator<Person>(Person.
           class);
10     invalidValues = validator.getInvalidValues(person, "name");
11
12     System.out.println("Es wurden " + invalidValues.length + "
           Validierungs Fehler gefunden!");
13     for (InvalidValue value : invalidValues)
14       System.out.println(value);
15   }
16 }
```

Listing 2.5: Validierung der Entität Person

```
Es wurden 2 Validierungs Fehler gefunden!
name muss zwischen 3 und 30 lang sein
name muss Ausdruck "[a-zA-Z]+" entsprechen
```

2.3.2 Java Server Faces

Java Server Faces [49] [12] [15] [37] ist ein standardisiertes Web Framework, welches in der Version 1.2 [4] seit Mai 2006 auf dem Markt befindet. Aktuell existiert JSF in der Version 2.0 [6], in welcher z.B. Facelets (Sprache der View Seiten), AJAX [8] und Konfiguration mittels Annotationen hinzukam, JSF verwendet eine MVC Architektur, womit eine Trennung der unterschiedlichen Schichten realisiert wird. Es lässt sich das Ganze somit in z.B. Web-Designer, Komponentenentwickler und Applikationsentwickler gliedern. Eine JSF Anwendung kann in jedem Servlet Container wie ein Apache Tomcat [19] ausgeführt werden. Im wesentlichen dient JSF dazu um dem Entwickler Arbeit abzunehmen und lässt sich in folgende 4 Bereiche gliedern:

[8] Asynchronous JavaScript and XML

- *Komponenten*
  JSF erlaubt es die komplette Webseite als reine Komponenten (eigene Klassen) in Java aufzubauen. Darüber hinaus ist es möglich Komponenten zu erweitern bzw. neue hinzuzufügen.

- *Datentransfer*
  JSF transportiert Daten aus einem Objekt in die Benutzerschnittstelle und wieder zurück. Es existieren Standard- Konvertierungen und Validierungen, die ebenfalls erweitert und ergänzt werden können.

- *Zustandsspeicherung*
  JSF kümmert sich um die automatische Speicherung des Zustandes der Applikation. Dh. der aktuelle Zustand aller Komponenten sowie zusätzliche Applikationdaten werden automatisch über einen Request hinweg gesichert.

- *Ereignisbehandlung*
  Es besteht die Möglichkeit mittels *Eventlistener* auf diverse Ereignisse zu reagieren.

Lebenszyklus

Ein Client (Web-Browser) schickt dem Server eine Anfrage (engl. Request), welche der Server beantwortet (engl. Response). Serverseitig werden zwischen JSF-Seiten und anderen Anfragen (JSP [9], statisches HTML, etc.) unterschieden. Diese Requests und Responses können somit in 4 unterschiedlichen Kombinationen auftreten. Es kann vorkommen, dass eine reine HTML Seite (engl. Non-Faces-Request) zu einer Facesseite verweist. Dies passiert meistens bei einem initialen Aufruf der Seite. Ebenso kann beim Verlassen einer Facesanwendung auf eine Non-Faces Seite weitergeleitet werden. Der Standard Fall von JSF ist aber ein Faces Request gefolgt von einem Faces Response. Dies wird auch der Standard-Lebenszyklus eines Request genannt. Dieser JSF-Lebenszyklus ist in Abbildung 2.3 dargestellt und lässt sich in 6 Bereiche gliedern.

JSF verwendet intern im Speicher immer einen Komponentenbaum (siehe Abbildung 2.4). Dieser wird je nach Zyklus neu generiert, oder aus einem bereits vorher im Speicher existierenden Baum geladen. Dieser Baum wird entweder aus den JSF-Layoutseiten generiert, oder direkt manuell aufgebaut.

Phase 1: Komponentenbaum wiederherstellen (Restore-View)
In der Phase 1 wird unterschieden, ob es sich um einen initialen Aufruf, oder um einen wiederholten Aufruf der Seite handelt. Diese Unterscheidung ist notwendig, da beim

[9] JavaServer Pages

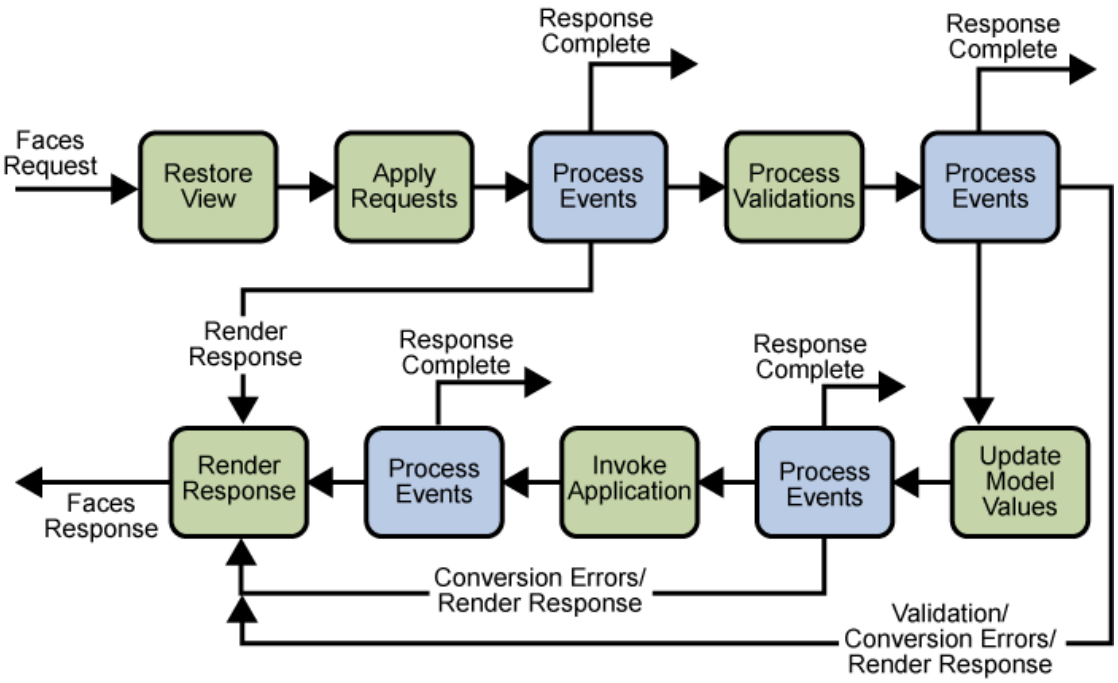

Abbildung 2.3: JSF Lebenszyklus (Quelle: Sun (Oracle))

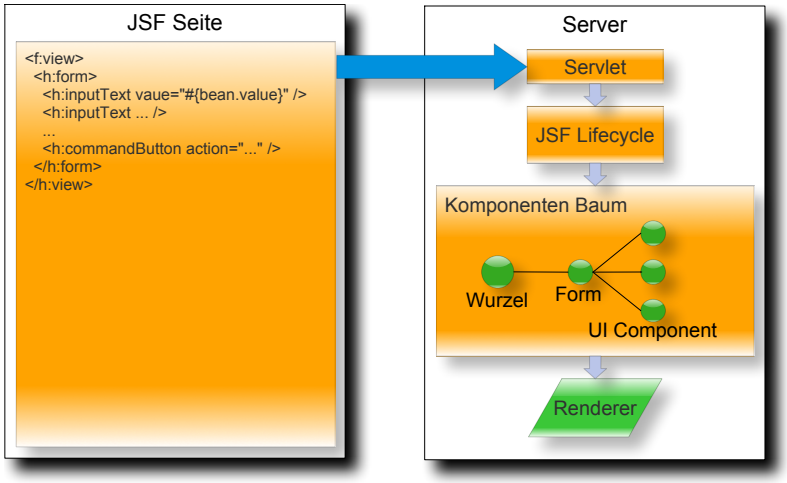

Abbildung 2.4: JSF Komponentenbaum

ersten Aufruf alle Komponenten initialisiert werden und der Komponentenbaum erstellt wird, da kein vorheriger Zustand existiert. Nach einer Standard-Initialisierung wird dann sofort zur Phase 6 weitergeleitet und die Antwort an den Browser zurückgeschickt. Bei einem wiederholten Aufruf werden alle Daten aus dem vorigen Zustand geladen und an die Phase 2 übergeben.

Phase 2: Request-Parameter auslesen (Apply Request Values)
In dieser Phase werden die vom Benutzer eingegeben Daten den einzelnen Komponenten zugewiesen. Hierzu wird im Wurzel-Knoten die Methode *processDecodes()* aufgerufen, welche dann wiederum rekursiv die gleiche Methode aller Kind-Elemente aufruft. Die Komponente selbst ist dafür verantwortlich die relevanten Daten aus dem HTTP-Header herauszulesen. Diesen Wert speichert die Komponente als „übermittelten" Wert (engl. submitted value). Dieser Prozess wird als „dekodieren" (engl. decoding) bezeichnet. Dieser neue Wert ist allerdings noch nicht der Wert, der dann im Objekt selbst gespeichert wird. Zuvor muss der „übermittelte" Wert in der Phase 3 konvertiert und validiert werden.

Phase 3: Konvertierung und Validierung durchführen (Process Validations)
In dieser Phase wird der Zeichenketten-basierte „übermittelte" Wert konvertiert und validiert. Es existieren Standard-Konverter wie z.B. Datums-Konverter, die aus dem String „09.01.2010" ein Datums-Objekt erstellen. Es besteht die Möglichkeit zusätzliche Konvertierer selbst zu schreiben, falls die Standard-Konverter nicht ausreichen. Nachdem die Konvertierung durchgeführt wurde, wird die Validierung aufgerufen. Die Validierung wird mittels eigenen JSF-Komponenten innerhalb von einem Input-Element definiert. Es existieren eine Reihe von Standard-Validatoren wie *LengthValidator* oder *LongRangeValidator*. Weiters gibt es zusätzliche Bibliotheken wie z.B. Apache MyFaces ExtVal [17], welche zusätzliche Validatoren für Email oder Kreditkarten bereitstellen. Um ein Attribut als Pflichtfeld zu markieren, wird hier im Speziellen kein eigenes Kind-Element definiert, sondern direkt in der Input-Komponente das Attribut *required* auf *true* gesetzt. Erst in Version 2.0 ist dies auch mit Kind-Elementen möglich. Nachdem die Konvertierung und Valdierung erfolgreich durchgeführt wurde, wird der typisierte Wert in dem Attribut *value* gespeichert und an die Phase 4 weitergeleitet. Zusätzlich wird noch bei einer Änderung des Wertes die Variable *localValueSet* auf *true* gesetzt. Falls die Konvertierung oder die Valdierung fehlschlägt wird eine entsprechende Fehlermeldung erstellt und direkt an die Phase 6 weitergeleitet und die Antwort an den Browser zurückgeschickt.

Phase 4: Modell aktualisieren (Update Model Values)
Erst in dieser Phase wird der vom Benutzer eingegebene Wert der eigentlichen Entität zugewiesen. Dh. der „übermittelte" Wert wurde korrekt konvertiert und validiert

und im Attribut *value* gespeichert. Dieser Wert wird jetzt mittels einem Setter im Domänenmodell gespeichert. Der aufzurufende Setter wird mittels dem *value* Attribut ($value$ = „$\#\{person.name\}$„) definiert. So weiß die Komponente, dass der Setter *setName(value)* in dem Bean *person* aufzurufen ist.

Phase 5: Applikationslogik ausführen (Invoke Application)
In dieser Phase passiert meistens eine Interaktion mit anderen Systemen wie z.B. Datenbanken. Die Objekte, die in der Benutzerschicht darzustellen sind, werden hier geladen, bzw. gespeichert. Es gibt 2 Varianten um hier Anwendungsmethoden aufzurufen. Entweder man definiert per *action* Attribut eine simple Methode, welche in dieser Phase aufgerufen wird. Mittels der Rückgabe wird festgelegt zur welcher Folgeseite delegiert werden soll. Die zweite Variante sieht die Verwendung von Action Listeners vor. Hierfür wird das Attribut *actionListener* gesetzt und ebenfalls eine serverseitige Methode referenziert. Es wird hier eine Verbindung zu einer Ereignis-Behandlungsmethode geschaffen, die kurz vor der *action* Methode aufgerufen wird. Der wesentliche Unterschied ist, das die Action-Listener als Parameter ein *ActionEvent* und keine Rückgabe besitzen. Der Vorteil eines *ActionEvent* ist, dass die Komponente, die diesen Event auslöste im Parameter mit übergeben wird. Der Nachteil ist, dass so keine Navigationsregeln aufgerufen werden können.

Phase 6: Antwort rendern (Render Response)
In dieser Phase wird der komplette Komponentenbaum gerendert und die Antwort an den Browser zurückgeschickt. Weiters speichert JSF den aktuellen Status für eine eventuell nächste Anfrage der selben Ansicht. Beim Rendern der einzelnen Komponenten kommen wieder die Konverter ins Spiel. Mittels der Methode *getAsString()* wird der Zeichenketten-basierte Wert aus dem eigentlichen Objekt erstellt, welche dann zum Client zurückgeschickt wird.

Managed Beans

Die *Managed-Beans* sind ein wesentlicher Bestandteil der Java Server Faces. Sie bilden in der Anwendung das Modell beziehungsweise die Verbindung zum Modell oder der Geschäftslogik. Die Grundvoraussetzung für Managed Beans sind simple Java Klassen, auch POJOs genannt. Für die Klassen bedeutet das, dass diese einen *public* Konstruktor ohne Parameter besitzen müssen. Da jetzt die Präsentationsschicht (JSF Seiten) auf diese Beans mit ihren Eigenschaften zugreifen um Daten lesen bzw. schreiben zu können müssen die Namen der Methoden einer bestimmen Konvention entsprechen. Für den lesenden Zugriff ist *getEigenschaftsName* und den schreibenden *setEigenschaftsName* mit dem Typ als Parameter zu implementieren. Siehe Listing 2.6.

Konfiguration
Um jetzt auf die Managed-Beans zugreifen zu können, muss neben der korrekten Implementierung dem JSF-Framework noch bekanntgegeben werden, welche dieser Beans existieren und welchen Gültigkeitsbereich (siehe nächstes Kapitel) diese besitzen. Dies ist einerseits in der Konfigurationsdatei *faces-config.xml* möglich (siehe Listing 2.7) und seit der JSF-Version 2.0 auch per Annotation (siehe Listing 2.6) möglich.

```
1  @ManagedBean
2  @RequestScoped
3  public class PersonHome {
4    private int count;
5
6    public setCount(int count) {
7      this.count = count;
8    }
9
10   public int getCount() {
11     return count;
12   }
13 }
```

Listing 2.6: Konfiguration eines Managed Bean per Annotation

```
1  <managed-bean>
2    <managed-bean-name>person</managed-bean-name>
3    <managed-bean-class>crud.bean.PersonHome</managed-bean-class>
4    <managed-bean-scope>request</managed-bean-scope>
5  </managed-bean>
```

Listing 2.7: Konfiguration eines Managed Bean per XML

Gültigkeitsbereiche
Es existieren in JSF mehrere Gültigkeitsbereiche (engl. Scopes), in denen Managed-Beans registriert werden können. Diese sind notwendig, da verschiedene Beans unterschiedliche Lebensdauern besitzen können. Folgende Gültigkeitsbereiche unterstützt JSF:

- *Non-Scope*
  Die Managed-Bean wird nie gespeichert und bei jedem Aufruf neu erstellt.

- *Request-Scope*
  Die Managed-Bean überlebt die Dauer einer HTTP Anfrage.

- *View-Scope*
  Die Managed-Bean ist an eine Ansicht geknüpft. Die Lebensdauer ist somit von der Lebensdauer der Ansicht abhängig. Dieser Scope ist allerdings erst mit Version 2.0 verfügbar.

- *Session-Scope*
  Die Managed-Bean existiert so lange, wie der Benutzer mit der Anwendung verbunden ist.

- *Application-Scope*
  Es existiert nur eine Managed-Bean Instanz für alle Benutzer, die mit der Anwendung verbunden sind.

JSF Navigation

Da jede Webanwendungen aus vielen unterschiedlichen Seiten besteht und somit die Navigation eine sehr große Rolle spielt, wurden in JSF Navigationsregeln eingeführt. Um überhaupt von der Seite A auf die Seite B navigieren zu können, ist entweder ein Link oder ein Button notwendig. Diese zwei Elemente lassen sich mit den JSF-Komponenten *h:commandButton* und *h:commandLink* generieren. Beide Komponenten besitzen das Attribut *save*, welches dazu dient serverseitig eine Methode aufzurufen. Um jetzt weder in der View noch in der aufzurufenden Servermethode die nachfolgende Seite kennen zu müssen, wird diese in der *faces-config.xml* Datei definiert. So eine Navigationsregel könnte wie in Listing 2.8 gezeigt aussehen. Hier wird eine Navigationsregel für die Seite *editPerson* beschrieben, welche zwei mögliche Folgeseiten definiert. Eine Anzeigeseite der Person und eine Listenseite aller Personen. Mit der Rückgabe der serverseitigen Aktionsmethode, lässt sich die Folgeseite bestimmen, welche in der XML-Datei definiert wurden. Es könnte z.B auf der Bearbeitenseite der Person einen Button *Speichern* und einen Button *Zeige alle Personen* geben. Der Button *Speichern* würde eine Aktionsmethode aufrufen, welche den String *ok* zurückliefert und der Button *Zeige alle Personen* würde eine Aktionsmethode aufrufen, welche den String *list* zurückliefert. Es lassen sich natürlich auch allgemeine Navigationsregeln, welche für alle Seiten gültig sind definieren.

```
1  <navigation-rule>
2    <from-view-id>/editPerson.xhtml</from-view-id>
3    <navigation-case>
4      <from-outcome>ok</from-outcome>
5      <to-view-id>/showPerson.xhtml</to-view-id>
6    </navigation-case>
7    <navigation-case>
8      <from-outcome>list</from-outcome>
9      <to-view-id>/listPerson.xhtml</to-view-id>
10   </navigation-case>
11 </navigation-rule>
```

Listing 2.8: JSF - Navigationsregel

2.3.3 Groovy

Groovy ist eine neue Open Source Programmiersprache, die als Ziel hat moderne Interpretersprachen wie Ruby oder Python nahtlos in die Java Welt zu integrieren. Groovy selbst ist keine Skriptsprache, sondern wird wie Java mittels der JVM [10] zu einem Bytecode kompiliert und ist somit ebenfalls plattformunabhängig. Ein großer Vorteil für Java-Entwickler besteht darin, dass diese nur einen geringen Lernaufwand haben, um sich mit Groovy zurechtzufinden. Es lässt sich beliebiger Groovycode jederzeit aus Java aufrufen und mittlerweile existieren schon für die meisten Entwicklungsumgebungen Plugins für Groovy.

Groovy Code

Der folgende kompakte Groovy-Quellcode

```
["Red", "Green", "Blue"].findAll{it.size() <= 4}.each{println it}
```

Listing 2.9: Groovy - Iteration

ist äquivalent zu diesem Java-Code

```
for (String it : new String [] {"Red", "Green", "Blue"})
  if (it.length() <= 4)
    System.out.println(it);
}
```

Listing 2.10: Java - Iteration

Es ist zu sehen, dass der Groovy Code sehr kompakt ist, welches die Lesbarkeit sehr stark erhöht und somit auch zu einem effizienteren programmieren führt.

2.3.4 Grails

Grails ist ein Framework zur Erstellung von Web 2.0 Anwendungen. Es basiert auf der Javaplattform und bekannte Frameworks wie Spring [51], Hibernate [24], Ant [16] [7] und Sitemesh [48]. Die verwendete Programmiersprache ist Groovy, welche im vorigen Kapitel 2.3.3 beschrieben wurde. Der Hauptfokus von Grails liegt speziell in Generierung von CRUD-Webanwendungen. Laut Entwickler, sollen sich zwar auch leicht andere Anwendungen generieren lassen, allerdings sind alle Tutorials und Beispiele immer nur für CRUD Anwendungen zu finden.

Ein grundlegendes Prinzip von Grails ist: Konvention statt Konfiguration. Dies bedeutet, dass statt Konfiguration in diversen Dateien, die Konfiguration durch bestimmte Konventionen erfolgt. Artefakte (wie z.B. Domänenentitäten) befinden sich immer im selben Verzeichnis. Dies spart Konfigurationsaufwand und erleichtert den Einblick in

[10] Java Virtual Machine

solche Projekte. Grails-Anwendungen lassen sich als Web-Archive (WAR Datei) exportieren und laufen somit auf jedem Servlet Container.

Das in Ruby on Rails [23] bekannt gewordene Scaffolding [11], welches dem Entwickler eine Generierung von CRUD Anwendungen bereitstellt, ist ebenfalls ein essenzieller Bestandteil wie die automatische Validierung von Grails.

Modell

Im Listing 2.3.4 ist eine Entität Person abgebildet. Ähnlich wie in Java wird eine Klasse mit dem Schlüsselwort *class* beschrieben. Anschließend werden die Attribute der Klasse definiert. Mittels eckigen Klammern *[addresses:Address]* wird eine 1:n Relation beschrieben.
Die Einschränkungen, die ein Objekt zu erfüllen hat, werden innerhalb von *static constraints* definiert. Es kann für jedes definierte Attribut eigene Einschränkungen wie z.B nullable, minimale oder maximale Länge festgelegt werden. Die Validierung erfolgt in der Methode *save*, welche alle Einschränkungen überprüft und eventuelle Fehler im Objekt selbst speichert. Diese Fehler können beim Aufruf einer *GSP* [12] Seite ausgegeben werden. Alle Entitäten müssen sich im Ordner *grails-app/domain* befinden.

```
1  class Person {
2    String    firstName
3    String    lastName
4    Email     email
5    static hasMany  =  [addresses:Address]
6
7    static constraints = {
8      firstName(nullable:false, minSize:3, maxSize:20)
9      lastName(size:3..20, blank:false)
10     email(nullable:true)
11   }
12 }
```

View

Die Client-seitige Darstellung erfolgt mittels GSP. Diese Technologie ähnelt sehr stark JSP und ASP, ist allerdings viel flexibler und intuitiver. Es existieren ebenfalls spezielle Grails Tags wie z.B.: *g:textField g:submitButton*, *g:hasErrors*, ... Alle GSP-Seiten existieren im Ordner *grails-app/views*. Wenn für eine Entität Scaffolding verwendet wird, werden 4 einzelne Seiten (*create.gsp*, *edit.gsp*, *list.gsp* und *show.gsp*) generiert.
Die Listenseite unterstützt eine Sortierung der einzelnen Attribute sowie eine Navigation. Für jede Adresse, wird in der ersten Spalte die Id als Link dargestellt, um auf die

[11] deutsch Gerüstbau
[12] Groovy Server Pages

Detailseiten navigieren zu können. In den weiteren Spalten, werden alle anderen definierten Attribute dargestellt.

Für die Ansichtseiten werden drei sehr ähnliche Seiten generiert. Die Unterschiede liegen im Detail. Die *create.gsp* und *edit.gsp* unterscheiden sich lediglich in diversen Labels. Zusätzlich enthält die *edit.gsp* Seite die Id der Entität als verstecktes Input Feld. In der *view.gsp* werden überhaupt keine Eingabe- sondern nur einfache Text- Felder dargestellt.

Wie bereits beschrieben unterstützt Grails eine Validierung der Eingabe. Diese Einschränkungen, welche direkt in der Entität definiert werden, können Validierungsfehler werfen. In Abbildung 2.5 ist zu sehen, dass bei der Eingabe einer negativen Zahl für die Hausnummer ein Fehler generiert wird, welcher den Benutzer informiert, dass nur positive Zahlen erlaubt sind. Die Fehlermeldung „Die Eigenschaft $[houseNumber]$ des Typs $[classAddress]$ mit dem Wert $[-2]$ ist kleiner als der Mindestwert von $[1]$“ kann natürlich auch internationalisiert werden. Dazu müssen in der *messages.properties* alle mögliche Fehlermeldungen übersetzt werden. Wenn folgende Zeile *address. houseNumber.min.error=Die Eigenschaft Hausnummer des Typs Adresse mit dem Wert* $[2]$ *ist kleiner als der Mindestwert von* $[3]$ in der Datei vorhanden wäre, würde die Fehlermeldung internationalisiert dargestellt werden. Um zu gewährleisten, dass alle Fehlermeldungen in die jeweilige Sprache übersetzt wurden, bietet Grails die Möglichkeit diese Keys mit *grails generate-i18n-messages <domainClass>* zu generieren.

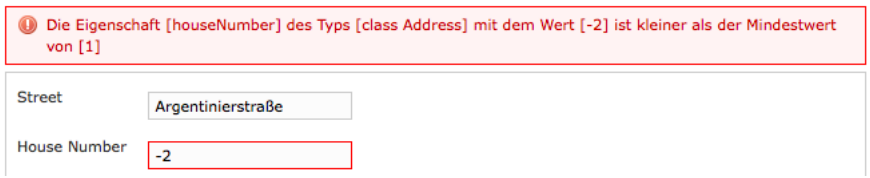

Abbildung 2.5: Grails - Validierungsfehler auf der Bearbeiten Seite der Entität Adresse

Controller

Um jetzt ein Objekt auf der Webseite darstellen zu können, wird ein Controller benötigt, der diese Logik bereitstellt. Mittels der Scaffolding Technologie lassen sich sehr schnell und einfach diese Controller implementieren. Mit dem Code *def scaffold = Address* (siehe Listing 2.3.4) werden durch Codegenerierung folgende Methoden für das Objekt Adresse unterstützt: *list*, *show*, *edit*, *delete*, *create*, *save*, *update*. Zusätzlich kann hier jederzeit weiterer Code hinzugefügt werden. Diese Klassen müssen sich im Ordner *grails-app/controllers* befinden.

```
1 class AddressController {
2     def scaffold = Address
3 }
```

2.3.5 Seam

Seam [27] ist ein sehr mächtiges Open Source Framework für Web 2.0 Anwendungen, welches von JBoss entwickelt wurde. Selber baut es auf unterschiedliche Frameworks auf. Die Kern-Frameworks sind Java EE 5, EJB 3.0[13] und JSF 1.2. JSF ist wie in Kapitel 2.3.2 bereits beschrieben ein MVC Framework für Webanwendungen. EJB 3.0 ist ein Framework für Geschäftsaufgaben und Datenbankpersistierung, welches auf normale POJOs aufbaut. Obwohl sich diese zwei Frameworks zwar gegenseitig ergänzen, wissen sie selbst nichts voneinander. Beispielsweise wird JSF mit XML-Dateien konfiguriert, während EJB 3.0 mit Annotationen konfiguriert wird. Einer der Hauptaufgaben von Seam besteht darin, diese Technologien zu verknüpfen. Seam darf jetzt aber nicht als Integrationsframework gesehen werden. Seam kann z.B. sehr gut mit ORM[14] umgehen. Es wird von Seam der Persistierungskontext über den ganzen Lebenszyklus verwaltet. Mit alten Frameworks ist es immer wieder zu *LazyInitializationException* Fehler gekommen, welches die Entwickler oft dazu bewegte Hacks wie DTO[15] zu implementieren. Seam wurde von Gaving King entwickelt, welcher auch die beliebteste ORM Lösung, Hibernate, geschaffen hat. Seam wurde von Grund auf so konzipiert, um die ORM Best Practices zu fördern.
Ein weiterer wesentlicher Bestandteil des Seam Frameworks ist die Implementierung des Design Patterns Dependency Injection (DI). Mit diesem Entwicklungsmuster, können POJO Komponenten, welche einfach mit speziellen Annotationen (*@In* und *@Out*) gekennzeichnet werden, auf z.B. einen Persistierungskontext, Logger oder eigene geschriebene Klassen jederzeit zugreifen, ohne sich darum kümmern zu müssen, woher die Instanzen kommen.
Weiters unterstützt Seam Validierungen, Geschäftsprozesse, ein Testframework und auf keinen Fall zu vernachlässigen, eine großartige Toolunterstützung.

MVC

Modell
Seam Entitäten sind normale POJOs, welche lediglich mit der zusätzlichen Annotation *@Name("person")* versehen werden. Alleine mit dieser Definition ist diese Komponente, ohne zusätzlicher Konfiguration wie z.B. in der *faces-config.xml*, überall im

[13] Enterprise JavaBeans
[14] Object Relational Mapping
[15] Data Transfer Object

Seam Kontext verwendbar. Auf den JSF Seiten kann per EL [16] (z.B.:<*h:outputText value="#{person.name}"* />) auf diese Seamkomponente zugegriffen werden. Laut Konvention muss es einen öffentlich sichtbaren Getter für das Attribut *Name* geben. Es müssen die Domänenobjekte allerdings nicht unbedingt mit der Annotation *@Name* versehen werden. Stattdessen wird wie in der MVC Architektur üblich ein Controller geschrieben, der die eigentlichen Entitäten zurückliefert.

View
Die View wird in Seam mittels JSF erstellt. Es werden aber noch eine ganze Reihe an zusätzlicher UI Komponenten wie z.B RichFaces [26], ICEFaces [28] oder Primefaces [54] unterstützt. Beide erweitern JSF mit AJAX, womit erst aktuelle Web 2.0 Anwendungen entwickelt werden können.

Controller
Der Controller wird ebenfalls mit der Annotation *@Name* definiert. Mittels seam-gen (siehe nächstes Kapitel) werden standardmäßig zwei Controller pro Objekt generiert. Ein Controller wird für die eigentliche Verwaltung und ein weiterer für die Listendarstellung dieser Entität erstellt.

seam-gen

Seam-gen ist ein Kommandozeilenprogramm, mit dem sehr schnell Seamartefakte generiert werden können. Da Seam selbst auf dem MVC Muster beruht, werden auch diese Module generiert. Es hilft dem Entwickler ungemein um einerseits sich leichter in Seam einzuarbeiten, da ein komplettes Seam Projekt quasi per Knopfdruck generiert werden kann und andererseits werden durch den Code Generator viele Code Templates zu Verfügung gestellt.

Folgende Funktionalität unterstützt seam-gen:

- *new-action*
  Generiert ein Java Interface und ein Stateless Session Bean mit einer Seam/EJB3 Annotation.

- *new-form*
  Generiert ein Java Interface und ein Stateful Session Bean mit einer Seam/EJB3 Annotation und die dazugehörige *xhtml* Datei, welche die View repräsentiert. Ebenso wird ein TestNG [1] Testfall generiert, mit dem das Verhalten des JSF-Lebenszyklus getestet werden kann.

[16] Expression Language

- *new-conversation*
  Generiert ein Java Interface und ein Stateful Session Bean mit einer Seam/EJB3 Annotation. Innerhalb dieser Klasse werden zwei Methoden mit *@Begin* und *@End* erstellt.

- *new-entity*
  Generiert eine Entität mit einer Seam/EJB3 Annotation und Beispielattributen.

- *generate-model*
  Generiert aus einem existierenden Datenbankschema die JPA-Entitäten. Es verwendet dabei das *Reverse Engineering* von Hibernate um die Klassen zu erstellen. Mittels der *seam-gen.reveng.xml* Datei im *resources* Verzeichnis, besteht die Möglichkeit zusätzlich diesen Prozess zu beeinflussen.

- *generate-ui*
  Generiert CRUD Seiten und die dazugehörigen Controller für alle existierenden Entitäten. Mehr dazu im nächsten Kapitel.

Seam-gen *generate-ui*
Wie schon vorher erwähnt lässt sich mittels *seam-gen generate-ui* eine komplette CRUD Anwendung generieren. Es verwendet entweder direkt die selbst implementierten Entitäten oder indirekt die mittels *generate-model* aus einem Datenbank Schema generierten Entitäten. Es existiert im Verzeichnis *jboss-seam-2.X.X* ein Unterverzeichnis *seam-gen*, in dem eine Vielzahl von Templates zu finden ist. Diese werden für die Generierung von Views, POJOs, Controllers, Test Fälle und diverse XML Dateien verwendet.

Seam - Generierte Listenseite
Auf der Listenseite wird zuerst eine Suchmaske generiert. Diese enthält für jedes definierte Attribut ein eigenes Eingabefeld. Es ist somit möglich nach mehreren Übereinstimmungen zu suchen.
Unterhalb der Suchmaske wird eine Liste der gefundenen Objekte dargestellt. Hier werden in der ersten Spalte die Id und in den folgenden Spalten die Attribute der Entität dargestellt. Die Listenseite unterstützt ebenso wie in Grails eine Sortierung der einzelnen Attribute sowie Navigation.

Seam - Validierungsfehler auf der Bearbeitenseite

In Abbildung 2.6 ist eine Bearbeitenseite mit Validierungsfehlern abgebildet. Diese hier zu sehenden Fehler werden mit dem Hibernate-Validator erstellt und an die View zurückgeliefert. Was anhand des Bildes alleine nicht zu sehen ist, dass diese Seite eine komplette AJAX-Unterstützung mit sich bringt. Sobald man ein Eingabefeld verlässt wird eine AJAX-Anfrage an den Server geschickt und die aktuelle Eingabe validiert.

Bei fehlerhafter Validierung wird eine entsprechende Meldung generiert und an den Browser zurückgeschickt und dargestellt.

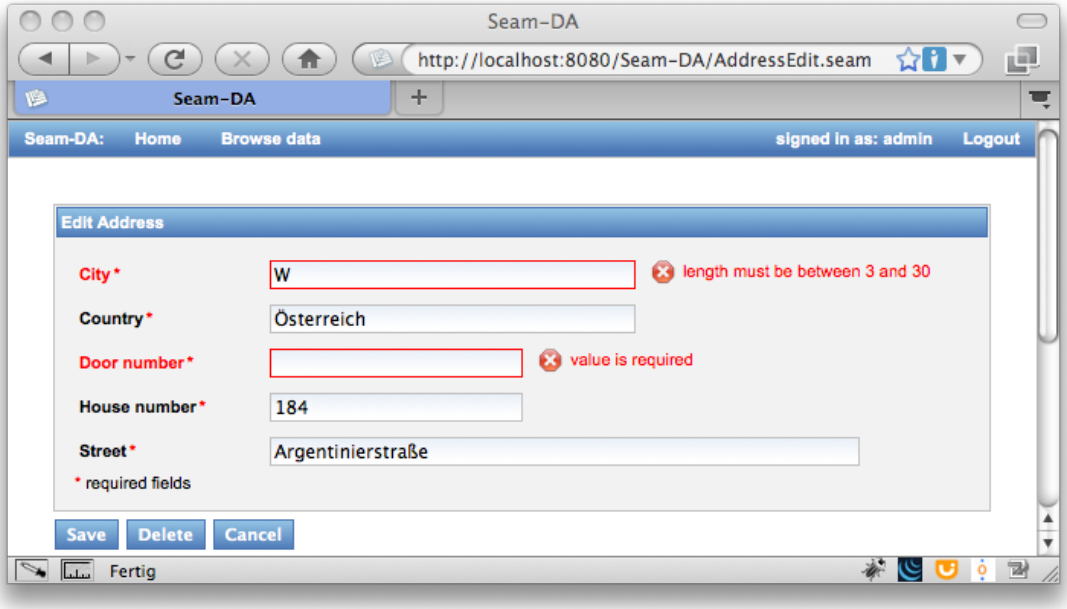

Abbildung 2.6: Seam - Validierungsfehler auf der Bearbeitenseite der Entität Adresse

2.4 Modell-getriebene Softwareentwicklung (MDD)

Dieses Kapitel soll die modell-getriebene Softwareentwicklung [14] erläutern und unter welchen Voraussetzungen eine MDD sinnvoll ist. Weiters wird auch auf die *Model Driven Architecture* (MDA) [45] [50] kurz eingegangen und die wesentlichen Unterschiede hervorgehoben.

Die modellgetriebene Softwareentwicklung oder Model Driven Software Development (MDD) befasst sich mit der Automatisierung in der Softwareentwicklung. Dies bedeutet, dass Code, einzelne Module oder ganze Anwendungen generativ aus formalen Modellen erstellt werden. Im Bereich Webgenerierung gibt es gerade einmal einige wenige MDD Ansätze. Einen interessanten Ansatz verfolgt Seam [17], der die bestehenden Java Klassen als Modell interpretiert und daraus CRUD-Webseiten generiert. Einen ähnlichen Ansatz verwendet das Web Framework Grails [18]. Dieses unterstützt das mit Ruby on Rails bekannt gewordene Scaffolding, mit dem ebenfalls die Generierung von CRUD-Seiten möglich ist. Das wohl derzeit komplexeste MDA Framework ist WebML [19] [30]. Hier können neben dem Domänenmodell auch Navigationsmodelle oder Präsentationsmodelle erstellt werden, um aus diesen Informationen die Webseiten zu generieren. All diese Ansätze verwenden einen Model-View-Controller (MVC [39]) Ansatz. Dieses Architekturmuster unterteilt die Software in drei Einheiten: Datenmodell, Präsentation und Programmsteuerung.

Modell-getriebene Softwareentwicklung definiert einen offenen Standard mit dem aus formalen Modellen lauffähige Software generiert wird. Dadurch soll eine Verbesserung der Softwareentwicklung erzielt werden. Diese ist aus folgenden Gründen wünschenswert:

- Die Komplexität der aktuellen Softwareanwendungen immer mehr zu.
- Die Anzahl der Systeme, auf der die Software laufen soll, werden immer mehr (Web Browser, Mobiltelefon, etc.).
- Die eingesetzten Technologien entwickeltn sich sehr schnell weiter.

Die wesentlichen Ziele sind zum einen eine Steigerung der Entwicklungsgeschwindigkeit und zum anderen eine bessere Handhabbarkeit aufgrund einer höheren Abstraktion. Die Steigerung wird mittels einem Generator, der diversen Code generiert erzielt. Durch diese Codegenerierung wird ebenfalls die Softwarequalität erheblich gesteigert. Weiters

[17] http://seamframework.org/
[18] http://www.grails.org/
[19] http://www.webml.org/

wird durch die Trennung von fachlichen und technischen Anteilen eine höherer Abstraktion erzielt.

Im Gegensatz verfolgt die MDA als Primärziele die Vereinheitlichung, Interoperabilität und Portabilität. Weiters setzt MDA in der Regel auf mehrere Modelle, die sich in ihrem Abstraktionsniveau unterscheiden.

2.4.1 Metamodell

Das Wort *Meta* kommt aus dem Griechischen und bedeutet *über*. Ein Metamodell ist somit ein Übermodell. Diese Übermodelle dienen dazu um Modelle definieren zu können. Dh. das Metamodell beschreibt die mögliche Struktur eines Modells. Dies beinhaltet alle Konstrukte wie Beziehungen, Einschränkungen und Modellierungsregeln.

2.4.2 Integration

Es soll jetzt keineswegs die komplette Anwendung generiert werden. Es wird immer individuelle Fälle geben, wo es einfach keinen Sinn macht dieses zu modellieren. Daraus ergibt sich nun das Problem, dass in dem generierten Code noch zusätzlicher nichtgenerierter Code zu integrieren ist. Einerseits besteht die Möglichkeit vom generierten Code zu erben und gezielte Methoden zu überschreiben. Ein weiterer Ansatz verwendet spezielle Coderegionen, die der Generator nicht überschreibt. Diese werden mit einfachen Kommentaren gekennzeichnet.

2.4.3 OpenArchitectureWare

OpenArchitectureWare [47] (oAW) ist ein Framework um modellbasierte Anwendungen zu erstellen. Es unterstützt eine Validierung des Modells, die eigentliche Generierung, sowie eine Transformation in andere Modelle. OpenArchitectureWare ist seit der Version 4.3.1 ein Bestandteil des Eclipse Modeling Framework Projekt (EMF) [21].

Check

Check ist eine an OCL[20] angelehnte Sprache, die dazu dient ein Modell auf zusätzliche Bedingungen zu überprüfen. Folgende Zusicherungen können z.B. definiert werden:

- *Invarianten*
  Invarianten müssen zu jeder gültig sein.
- *Preconditions*
  Müssen vor einer Ausführung einer Operation gelten.

[20] Object Constraint Language

- *Postconditions*
  Müssen nach einer Ausführung einer Operation gelten.

Xtend

Xtend ist eine funktionale Sprache, mit der das Metamodell mit zusätzlicher Logik versehen werden kann. Diese Erweiterungen können von Xpand und Check direkt verwendet werden.

Xpand

Xpand ist eine schablonenbasierte Sprache zur Codegenerierung. Es werden Templates geschrieben, die oAW parst und daraus einen Code bzw. ein neues Modell generiert.

2.4.4 Eclipse Modeling Framework

Eclipse **M**odeling **F**ramework Projekt [21] ist ein Open Source Projekt, welches anhand definierter Modelle Quellcode generiert. EMF verwendet als Modellierungssprache Ecore. Ecore basiert auf EMOF[21], welches eine Menge von Sprachkonzepten für eine Modellierung von objektorientierten Strukturen definiert. Ecore selbst ist die Java-basierte Implementierung von EMOF. Beide Modellierungssprachen, werden mittels XML/XMI beschrieben.

2.4.5 Graphical Modeling Framework

Das Eclipse **G**raphical **M**odeling **F**ramework ist ebenfalls ein Open Source Projekt um grafische Editoren zu entwickeln. In Abbildung 2.7 ist eine Übersicht des Ablaufes zur Erstellung eines grafischen Editors. Neben dem Metamodell, benötigt man für einen grafischen Editor weiters die Definition der Tool-Palette und der darzustellenden Objekte im Diagramm.
Unter der vereinfachten Annahme es existiert ein sehr einfaches Metamodell, welches nur Klassen und Beziehungen unterstützt, würde es für die Tool-Palette zwei Elemente geben. Ein Element *Neues Objekt* und ein weiteres Element *neue Beziehung*. Beide Elemente haben einen Text, eine Bezeichnung, ein kleines und großes Bild. Diese zwei Elemente sollen jetzt in einem grafischen Editor dargestellt werden. GMF unterstützt hier selbst Objekte und Beziehungen. Objekte können als Vektorgrafik oder als Bild definiert werden. Es existieren auch Standardobjekte wie Rechtecke, Kreise, Ellipsen, ... Für unser Beispiel würde für die Darstellung einer Klasse ein einfaches Rechteck mit Bezeichnung vollkommen genügen. Das selbe gilt für die Beziehung. Diese kann ein einfacher Strich zwischen den zwei Klassen sein, welches so von GMF ebenfalls

[21] Essential Meta Object Facility

unterstützt wird oder z.B. ein gestrichelter Pfeil. Falls die Standardformen von GMF nicht reichen, oder nicht hübsch genug sind, können und müssen diese Formen selbst entworfen werden.

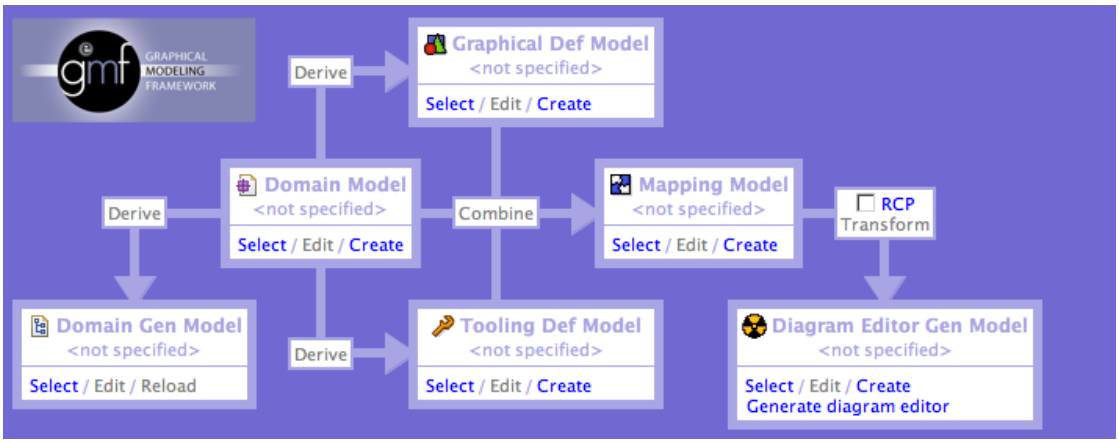

Abbildung 2.7: GMF Übersicht (Quelle: Eclipse GMF)

2.5 Arten der Eingabevalidierung

Die Eingabevalidierung wird in den aktuellen Web 2.0 Anwendungen immer populärer. Vor einigen Jahren noch wurden einfach am Anfang der Bearbeitungsmethode diverse Validierungen durchgeführt. Dies ist zwar ein sehr schlechter Ansatz, konnte aber vor AJAX noch praktiziert werden, da beim Abschicken eines Eingabeformulares immer das gesamte Formular geschickt wurde. Es konnten somit die notwendigen Validierungen durchgeführt werden und bei fehlerhaften Eingabe Daten eine entsprechende Nachricht an den Client zurückgeschickt werden. Diese Umsetzung war vor allem in den Model1 Architekturen (Siehe Kapitel 2.2.1) sehr beliebt.

Um den Server einerseits zu entlasten und andererseits eine bessere Benutzerfreundlichkeit bieten zu können, wurden diese Validierungen oft mit JavaScript in den Browser verlegt. Mit dem sehr lange erfolgreichem Web Framework Apache Struts [52] konnte JavaScript Code automatisch generiert werden, der die Eingabedaten validierte. Dies scheint zwar eine saubere Lösung zu sein, hat sich allerdings nicht etabliert, da die Validierung aus Sicherheitsgründen ebenso am Server durchgeführt werden musste und vor allem deswegen, da AJAX immer beliebter wurde.

Heutzutage unterstützen alle aktuellen Web Frameworks eine Model2 Architektur mit unterschiedlichen Lebenszyklen. In JSF, wie in Kapitel *Konvertierung und Validierung* 2.3.2 beschrieben gibt es eine eigene Phase, in der solche Validierungen durchge-

führt werden. Es ist dabei egal, ob es sich dabei um herkömmliche oder AJAX Anfragen handelt. In beiden Fällen wird in der Validierungsphase die Validierung durchgeführt und bei fehlerhaften Eingabedaten der Lebenszyklus vorzeitig abgebrochen und eine entsprechende Fehlermeldung an den Client zurückgesendet.

2.5.1 Definition der Einschränkungen

Wo werden nun die Einschränkungen definiert? Es gibt eine Vielzahl von unterschiedlichen Möglichkeiten. Einerseits kann die Definition der Validierung, wie es auch Struts macht, in XML Dateien abgelegt werden. Allerdings muss das Web Framework dieses auch unterstützen. Eine weitere Möglichkeit ist direkt in den JSP oder JSF Seiten die Einschränkung zu definieren. Hier werden für Eingabefelder Kindelemente hinzugefügt, welche die Validierungsdefinition enthalten. Beide Ansätze sind heute noch sehr oft zu finden.

Da fast alle komplexen Web 2.0 Anwendungen in der untersten Schicht auf eine Datenbank zugreifen, werden die Eingabedaten letztendlich auch dort persistiert. Da die Datenbank selbst auch Einschränkungen hat, würde es natürlich Sinn machen dieselben Einschränkungen für die Eingabe- und die Datenbankvalidierung zu verwenden. Dies lässt sich in aktuellen Web Anwendungen, in denen JPA für die Persistierung von Objekten zum Einsatz kommen, sehr leicht realisieren. Es werden hier direkt in der Klasse die notwendigen Einschränkungen an den Attributen definiert. Einerseits verwendet JPA diese Einschränkung für die Erstellung der Tabellen (wenn ein Attribut nicht mehr als 30 Zeichen lang sein darf, wird in der Datenbank eine Spalte mit der Länge von 30 angelegt) und andererseits für die Validierung gegen die Datenbank selbst.

Die selben Einschränkungen werden in aktuellen Webanwendungen ebenso für die Validierung der Eingabedaten verwendet. Es muss lediglich in der Seite selbst definiert werden, ob das Attribut validiert werden soll. Dieser Ansatz lässt sich ebenso mit AJAX erweitern.

KAPITEL 3

Related Work

3.1 Generating Web Applications with Abstract Pageflow Models

Die Arbeit von Oberortner[42] beschäftigt sich mit der automatischen Generierung von Web-Applikationen basierend auf einer MDA. Insbesondere konzentriert sich diese Arbeit auf die Definition eines Metamodells für die Modellierung von Web-Applikationen. Die Hauptaufgabe besteht in der automatischen Generierung der modellierten Web-Applikationen. Die Modelle beinhalten Informationen über die Webseiten und den Page-Flow innerhalb der zu generierenden Web-Applikation. Ein sehr interessanter Aspekt dieser Arbeit ist, dass die Objekte implizit durch die Definition der Seiten festgelegt werden. Es wird somit wirklich die echte View Darstellung modelliert und nicht wie in allen anderen Ansätzen das Modell selbst. Neben dem Metamodell wurde ein Generator entwickelt, mit dem JSF Web-Applikationen erstellt werden können. Dieser Generator erzeugt die grafische Oberfläche und für den Page-Flow die notwendigen XML Dateien.

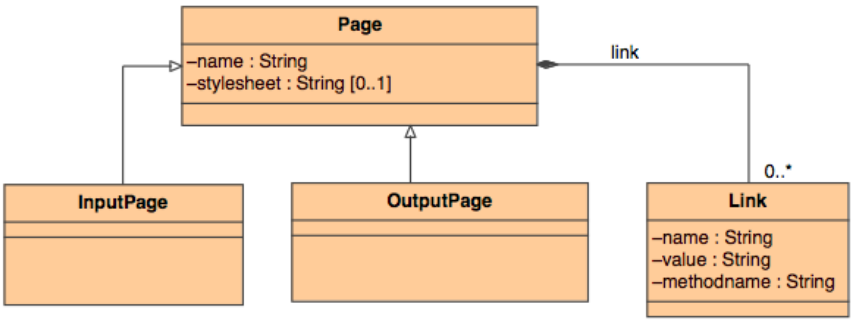

Abbildung 3.1: Seitendefinition Quelle (Ernst Oberortner[42])

In Abbildung 3.1 ist zu sehen, wie Viewseiten in Eingabe- und Ausgabe- Seiten unterteilt werden. Für Eingabgeseiten können Formelemente definiert werden, welche wiederum verschiedene Eingabe- Felder und Buttons definieren können. In den Ausgabeseiten können statische und dynamsiche Texte definiert werden.

3.2 UWE Ansatz

Der UWE [1][58] [46] Ansatz verfolgt eine modell-getriebenene Entwicklung für Webanwendungen. Dabei baut dieser Ansatz auf der standardisierten Sprache UML[2] auf. Es werden eigene UML Profile definiert, mit denen es möglich ist alle wesentlichen Aspekte einer Webanwendung, nämlich Inhalt, Navigation, Prozesse und Präsentation abzubilden.

Die Hauptziele des UWE Ansatzes sind:

- UML basierte Modellierungssprache
- Konzept: Trennung von Inhalten
- Modellgetriebener Ansatz
- Modell Konsistenzprüfung

3.2.1 MagicUWE

MagicUWE[57] [36] ist ein Plugin für MagicDraw [40]. MagicDraw selbst ist ein sehr gutes UML Visualisierungstool. Mit MagicUWE wird jetzt dieses Tool erweitert, um UWE Modelle zu unterstützen. Damit ist es möglich graphisch die unterschiedlichen Digagramme wie Inhalt, Navigation, Prozesse und Präsentation zu modellieren. Weiters werden Modell zu Modell Transformationen, Package Struktur und Icons von UWE Sterotypen unterstützt.

3.2.2 UWE4JSF

UWE4JSF[56] [13] ist ein Eclipse Plugin mit welchem eine modell-getriebene Entwicklung von JSF Webanwendungen mit UWE ermöglicht. Es wurde die Modellierungssprache von UWE überarbeitet und erweitert, damit auch zeitgemäße Benutzeroberflächen modelliert werden konnten. Es exisitiert ebenso eine Unterstützung von JSF Komponentenbibliotheken wie z.B. Apache MyFaces Tomahawk [18]. Auch der Einsatz von modernen Web-Technologien wie AJAX wird auf diese Weise möglich. Mit dem Präsentationsmodell wird beschrieben, wie und welche Elemente auf den Seiten dargestellt werden. Mit Hilfe von Aktivitätsdiagrammen lassen sich alle möglichen Aktionen auf den einzelnen Seiten beschreiben. Mehr Informationen gibt es in der Diplomarbeit *Modellbasierte Generierung von Web-Anwendungen mit UWE (UML-based Web Engineering)*[32] von Christian Kroiss.

[1] UML-based Web Engineering
[2] Unified Modeling Language

3.2.3 Vergleich

Der Unterschied zwischen diesem und dem aktuellen Ansatz besteht darin, dass bei UWE UML und beim CRUD Ansatz Ecore als Modellierungssprache verwendet wird. Allerdings der wesentliche Unterschied liegt im Umfang von UWE. Durch das Navigationsmodell lassen sich alle denkbare Seiten definieren und letztendlich auch generieren. Der Hauptfokus des CRUD Ansatzes liegt auf der Datenmanipulation, in der sich diese zwei Ansätze unterscheiden.

3.3 WebML

Web **M**odeling **L**anguage (WebML [2] [11]) ist ein modellbasiertes Framework um komplexe Webseiten zu erstellten. Es werden eine Vielzahl von Modellen unterstützt, um komplexe Webanwendungen zu beschreiben. Diese Modelle werden mit mehreren XML Dateien beschrieben. Seit einigen Jahren gibt es eine eigene Entwicklungsumgebung namens WebRatio [60], welche auf Eclipse bzw. dem grafischen Modellierungs Framework aufbaut. Dieses Entwicklungstool bietet eine graphische Oberfläche um diese Modelle zu modellieren. WebML selber ist kein eigens Webframework, sondern verwendet das bekannte Struts [52] Framework, welches von der Apache Software Foundation entwickelt wurde. Dh. es werden alle notwendigen Artefakte generiert, die in Struts benötigt werden, um eine komplette Webanwendung darzustellen.

3.3.1 Die Modelle

Das Strukturmodell

Im Strukturmodell, welches dem Domänenmodell entspricht, werden alle Entitäten, welche die eigentlichen Daten beinhalten und ihre Beziehungen beschrieben.

Das Hypertextmodell

Das Hypertextmodell gliedert sich in das Kompositions- und dem Navigations Modell.

Das Kompositionsmodell

Im Kompositionsmodell wird die Darstellung der Elemente, die im Strukturmodell definiert wurden bestimmt. Da die Entitäten in unterschiedlicher Form dargestellt werden können, wie z.B. Listen, Detailansicht oder eine Bearbeitenseite der Entität, gibt es unterschiedliche Elemente, mit denen die Ansicht der Elemente genau festgelegt werden kann.

- Data Units
  Mit diesem Element können beliebige Entitäten auf der Webseite dargestellt werden. Es besteht die Möglichkeit, alle vorhandenen Attribute oder nur eine ausgewählte Untermenge darzustellen.

- Multidata Units
  Die Multidata Unit kann als Iterator für die Data Units angesehen werden. Es können somit mehrere Entitäten des selben Typs angezeigt werden.

- Index Units
  Index Unit ähneln sehr stark den Multidata Units, nur mit dem Unterschied, dass mit diesem Element die Entitäten in einer einfachen Liste dargestellt werden. Es wird somit jede Entität in einer eigen Zeile dargestellt.

- Scroller Units
  Scroller Units stellen wie Multidata und Index Units ebenfalls mehrere Entitäten in einer Liste dar, allerdings mit der zusätzlichen Erweiterung einer Navigation. Mit den Attributen *first*, *last*, *previous* und *next*, lässt sich dieses Element sehr leicht nach den jeweiligen Bedürfnissen konfigurieren.

Es gibt noch eine Vielzahl anderer Units wie *Entry Unit*, *Filter Unit*, *Redirect Unit*, *Multichoice Unit* (siehe WebML Elemente [3]).

Das Navigationsmodell

Im Navigationsmodell werden die einzelnen Kompositionselemente miteinander verknüpft. Diese Verknüpfungen können aus kontextuellen Links bestehen, bei denen die Elemente in einem semantischen Zusammenhang stehen. In der Abbildung 3.2 ist so eine semantische Abhängigkeit zu sehen. Wenn ein Benutzer auf der Detailseite Person alle Adressen, die dieser Person zugeordnet sind, anschauen möchte, klickt er auf einen gerenderten Link (*Adressen*) um die Adressen auf einer neuen Seite anzuzeigen.
Ein nicht kontextueller Link, ist ein Verweis auf eine unabhängige Entität. Dies kann z.B. ein Link *Einstellungen*, *Logout* oder *Alle Adressen* sein. Selbst wenn das Element *Person* mehrere Adressen besitzen kann, der Link „Zeige alle Adressen“ wäre hier ein kontextunabhängiger Link, da hier alle und nicht Personen spezifische Adressen dargestellt werden.
In Abbildung 3.3 ist eine mögliche Generierung des Hypertextmodell 3.2 dargestellt. Es werden 3 Seiten erstellt. Eine Detailseite für die Person, eine Listenseite für die Adressen einer Person und zuletzt eine Detailseite einer Adresse.

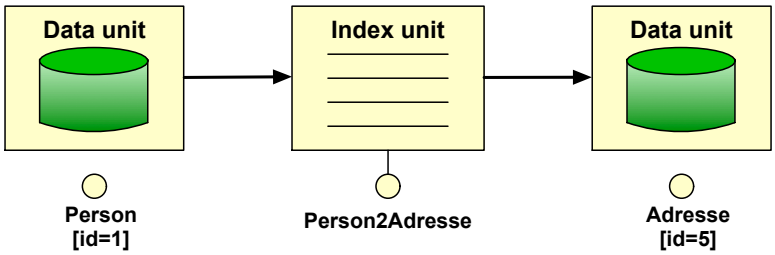

Abbildung 3.2: WebML - Hypertextmodell (Quelle WebML)

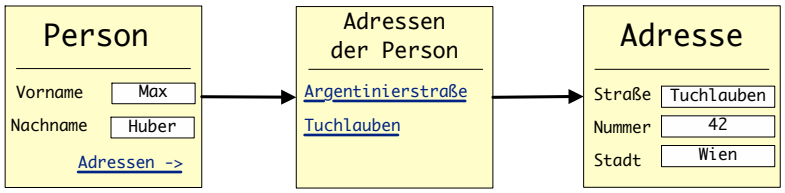

Abbildung 3.3: WebML - Hypertextmodell ins Web transformiert (Quelle WebML)

Das Präsentationsmodell

In dem Präsentationsmodell kann das Layout der Webanwendung spezifiziert werden. Dieses Layout wird mittels Stylesheet definiert. Es können einerseits allgemeine (welche unabhängig vom Inhalt sind) und andererseits spezialisierte Stylesheets für bestimmte Elemente spezifiziert werden.

Das Personalisierungsmodell

Zuletzt unterstützt WebML Benutzer- und Gruppenprofile. Somit ist es möglich für unterschiedliche Personen individuelle Ansichten zu erstellen. Damit können sehr leicht Administrationsoberflächen als Ergänzung oder in komplett neuen Seiten entworfen werden.

3.3.2 Validierung

Da die fertig generierte Webanwendung eigentlich eine normale Struts-Anwendung ist, werden somit auch alle Funktionen die dieses Framework bereitstellt unterstützt. Dies beinhaltet auch eine Validierung der Benutzereingabe. Die Konfiguration solcher Einschränkungen erfolgt in einer eigenen XML-Datei. Es können für alle Attribute der

Klassen beliebige Einschränkungen festgelegt werden. Dies sind alle herkömmlichen Validierungen wie z.B. maximale/minimale Länge, Typ-, Email-, URL Validierung und viele mehr.

3.3.3 Vergleich

WebML ist ähnlich wie UWE ein komplexer Ansatz mit dem sich weit mehr als nur CRUD Seiten erstellen lassen. Neben den zusätzlichen Modellen, welche WebML unterstützt, verwendet der generierte Code das Struts Framework. In beiden Ansätzen ist eine Validierung möglich, welche nur auf eine unterschiedliche Technologie beruht. Der Hauptfokus des CRUD Ansatzes liegt auf der Datenmanipulation, in der sich diese zwei Ansätze unterscheiden.

3.4 WebDSL

Eine domänenspezifische Sprache DSL [3] ist ein formale formale Sprache, die speziell für eine Domäne definiert wird. Diese Sprache dient dazu die Produktivität in der Softwareentwicklung zu steigern. Folgende zwei Ansätze „Tailoring a Model-Driven Quality-of-Service DSL for Various Stakeholders“ [44] und „Domain-specific Languages for Service-oriented Architectures: An Explorative Study“ [43] verwenden eine DSL, basierend auf einer MDD für Serviceorientierte Architektur (SOA).

In der WebDSL wird das Modell nicht mit einer XML-Datei spezifiziert, sondern stattdessen in wenigen Zeilen einer Entität 3.1. Der große Vorteil gegenüber Sprachen wie Java ist, dass hier redundanter Code, vorallem für Getter und Setter wegfällt. Würde man diese gezeigte Entität *Person* in Java definieren, würden mit JPA Annotation, Konstruktor, Gettern und Settern mindestens 25 Zeilen anfallen.

```
1 entity Person {
2   name :: String
3   age  :: Int
4 }
```

Listing 3.1: WebDSL Entität

Mit der WebDSL lassen sich alle Anforderungen einer Webanwendungen definieren. Dies beinhaltet neben der eigentlichen Domänendefinition folgende Artefakte:

- Präsentation
- Seitenverlauf
- Zugriffkontrolle

[3] Domain Specific Language

- Validierung
- Styling
- Ajax
- HTTPS

Eine Seitendefinition könnte wie in Listing 3.2 aussehen. Hier wird eine Seite *welcome* definiert. Weiters können in dieser Seitendefinition zusätzlich Navigationslinks, Bearbeitungsmethoden, Bilder, Listen und vieles mehr definiert werden. WebDSL wird genau in dem Artikel *WebDSL: A Case Study in Domain-Specific Language* [59] beschrieben.

```
1 define page welcome() {
2   title { "Welcome" }
3   section {
4     header{ "Hello world." }
5     "Greetings to you."
6   }
7 }
```

Listing 3.2: WebDSL Seitendefinition

KAPITEL 4

Entwurf und Implementierung

4.1 Übersicht

In diesem Kapitel wird das selbst entwickelte CRUD-Framework beschrieben. Mit diesem lässt sich aus einem wohldefinierten Modell eine komplette Webanwendung, die die CRUD-Funktionalität unterstützt generieren.

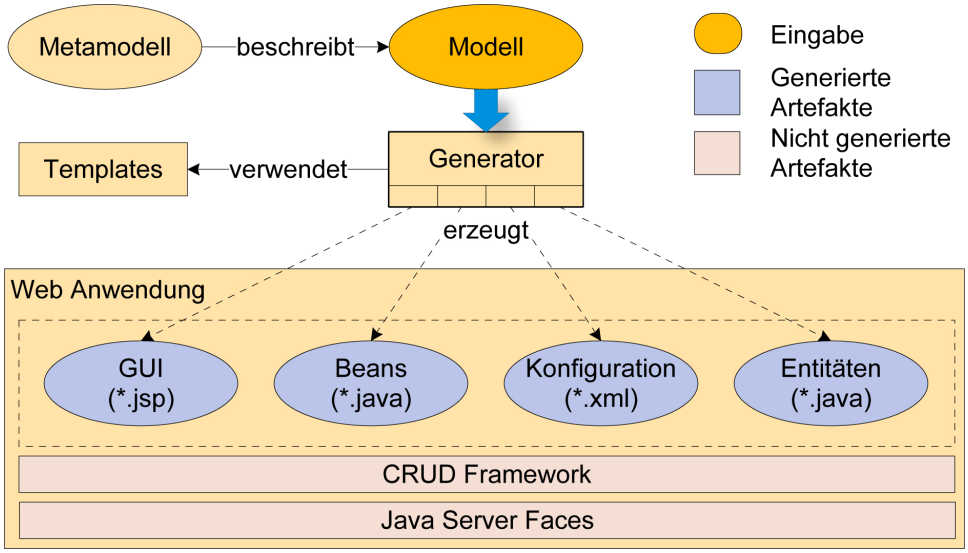

Abbildung 4.1: CRUD Framework

In Abbildung 4.1 wird der Zusammenhang der einzelnen Module grafisch dargestellt. Einzig das Modell muss definiert werden. Der Generator erzeugt dann die Artefakte GUI, Beans, Konfiguration und Entitäten. Diese bilden mit vordefinierten Templates (siehe Kapitel 4.3) die Webanwendung, welche als WAR Datei exportiert werden kann.

Die CRUD-Architektur lässt sich in 3 Module gliedern:

- *Modell*
  Es lässt sich mittels einem eigens für die CRUD-Verwaltung entworfenen Metamodell (siehe Kapitel 4.2) ein Domänenmodell mit Einschränkungen zwecks Validierung beschreiben. Dieses Modell selbst wird einer Validierung unterzogen, um eine eventuell falsch modellierte Welt noch vor der Codegenerierung feststellen zu können.

- *Source Templates*
  Es wird nie der komplette Code automatisch erstellt. Laut MDA sollen nur bis 80% des Codes generiert werden. Der andere Teil soll zwecks Flexibilität selbst

implementiert werden. Das CRUD-Framework stellt ein gewisses Grundgerüst zu Verfügung, welches zum einen die Logik für Datenbankaufrufe und andererseits spezielle Validierungsklassen bereitstellt. Diese Templates können jederzeit angepasst und erweitert werden.

- *Generierter Code*
  Der Codegenerator verwendet das definierte Modell als Input und generiert daraus Java Dateien, JSF-Seiten und diverse Konfigurationsdateien. Dieser generierte Output referenziert sehr oft auf die bereits vorhandenen Templates.

4.2 Metamodell

Abbildung 4.2 zeigt eine Übersicht des eigens entworfenen Metamodell. In diesem Modell wird beschrieben, wie Klassen miteinander in Beziehung stehen, welche Vererbungen gültig sind oder wie Attribute einer Klasse zugeordnet werden dürfen. Weiters ist in Abbildung 4.9 zu sehen, welche Validierungen verwendet werden können. Das Metamodell beschreibt lediglich die Domäne mit den Validierungen und keinesfalls eine Navigation. Die generierten JSF-Seiten ergeben sich implizit aus dem Modell. Mehr dazu in Kapitel 4.4.2.

4.2.1 Wurzelelement Model

Das oberste Element *Modell* ist eine Art Container für alle Typen, Beziehungen und Vererbungen (siehe Abbildung 4.3).
Ein Element *Modell* selbst besteht aus folgenden Attributen:

- *name*
  Der Name des Modells, welcher ein Pflichtfeld und vom Typ String ist.

- *packageName*
  Der Name des Java Package, in dem die Entitäten generiert werden. Ist ein Pflichtfeld und vom Typ String.

- *type*
  Das Modell kann mehrere *Typen* wie *Klassen* oder *Enumerationen* enthalten. Da das Element *Typ* selbst abstrakt ist, kann es somit nicht direkt als Kindelement ins Element *Modell* hinzugefügt werden. Ein Modell darf beliebig viele Typen besitzen.

- *association*
  Alle möglichen Beziehungen existieren ebenfalls direkt im Element *Modell*. Ein Modell darf beliebig viele Beziehungen besitzen.

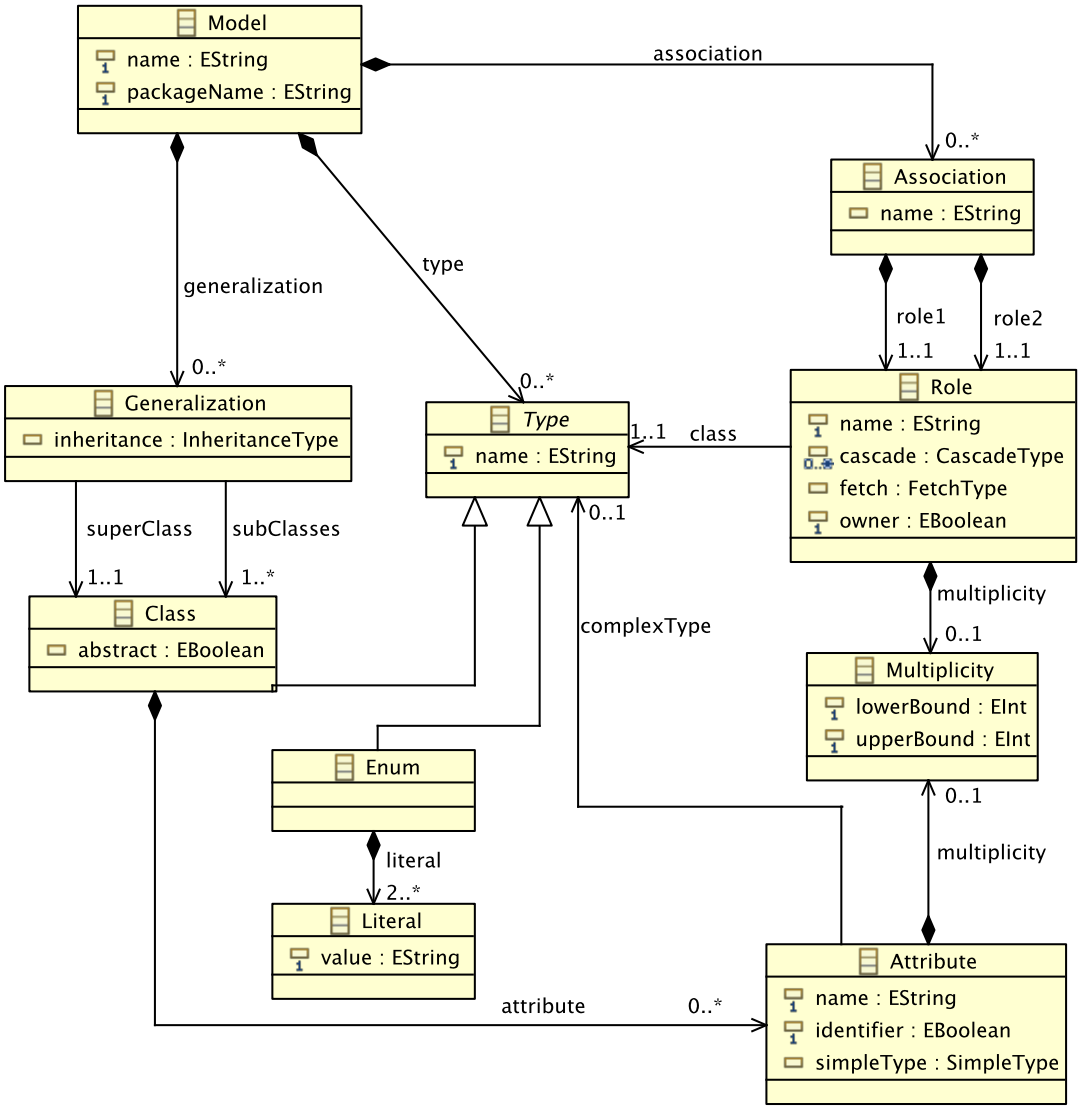

Abbildung 4.2: Metamodell Übersicht

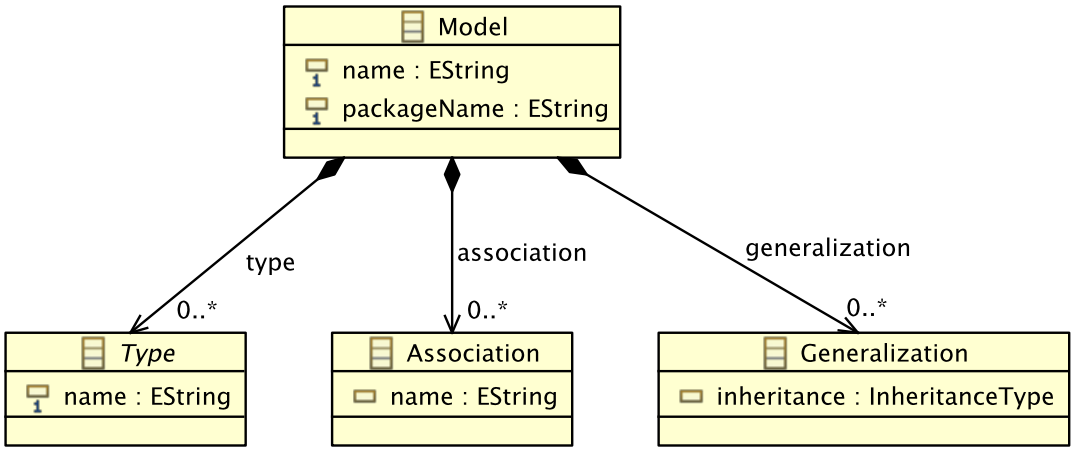

Abbildung 4.3: Wurzelelement

- *generalization*
  Das Metamodell unterstützt auch Vererbung, welches mittels dem Element *Generalization* abgebildet wird. Ein Modell darf beliebig viele Vererbungen besitzen.

4.2.2 Element Typ - Klassen und Enumerationen

Das Element *Type* ist ein abstraktes Element, welches eine *Enumeration* oder *Klasse* sein kann. Die Abbildung 4.4 beschreibt die tatsächlichen Instanzen und die möglichen Kindelemente, die ein Element *Type* besitzen kann.

Element Enumerationen

Das Element *Enum* repräsentiert eine Enumeration und besteht aus folgenden Attributen:

- *literal*
  Eine Enumeration muss 2 oder mehrere Elemente *Literal* besitzen, welche jeweils ein Attribut *value* besitzen, mit dem die Werte für die Enumeration definiert werden.

Element Klasse

Das Element *Class* repräsentiert eine Klasse und besteht aus folgenden Attributen:

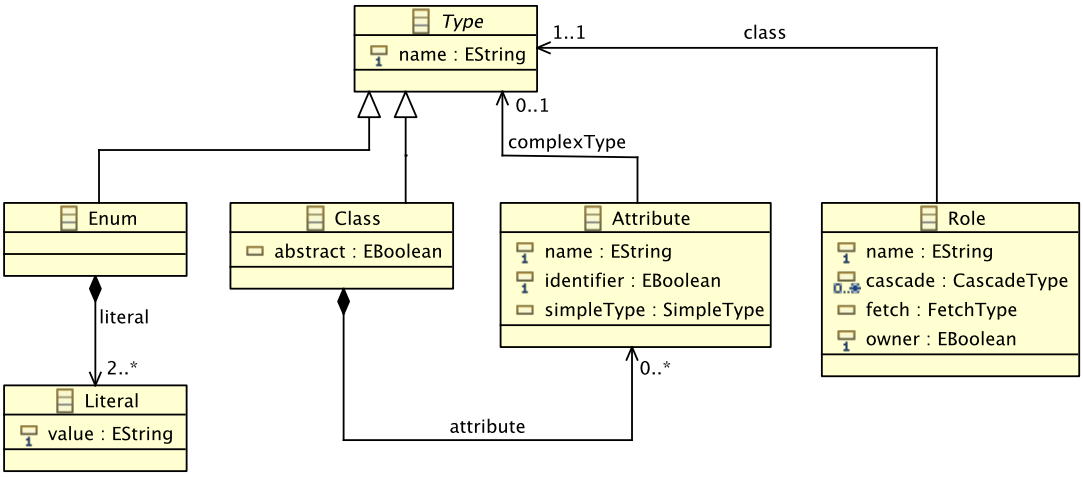

Abbildung 4.4: Klassen, Enumerationen und Attribute

- *abstract*
  Mittels diesem Attribut wird festgelegt, ob dieses Element eine abstrakte Klasse beschreibt. Mögliche Werte sind *true* oder *false*.

- *attribute*
  Eine Klasse kann 0 oder mehrere Elemente *Attribute* besitzen. Mittels dieser Attribute werden die Eigenschaften der Klasse beschrieben.

4.2.3 Element Attribut

Ein Element *Attribut* beschreibt ein Attribut einer Klasse. Es wird hier zwischen zwei Arten von Attributen unterschieden.

1. *Einfache Typen*
   Diese Typen, werden auf die Standard Java Typen (String, Integer, Long, usw.) abgebildet. In Abbildung 4.8 ist eine vollständige Liste, der unterstützten Typen.

2. *Komplexe Typen*
   Diese Typen sind Klassen oder Enumerationen, welche selbst im Modell definiert werden. Es darf nur einer der beiden Typen angegeben werden.

Das Element *Attribut* besteht aus folgenden Attributen:

- *name*
  Der Name des Attributes, welcher ein Pflichtfeld und vom Typ String ist.

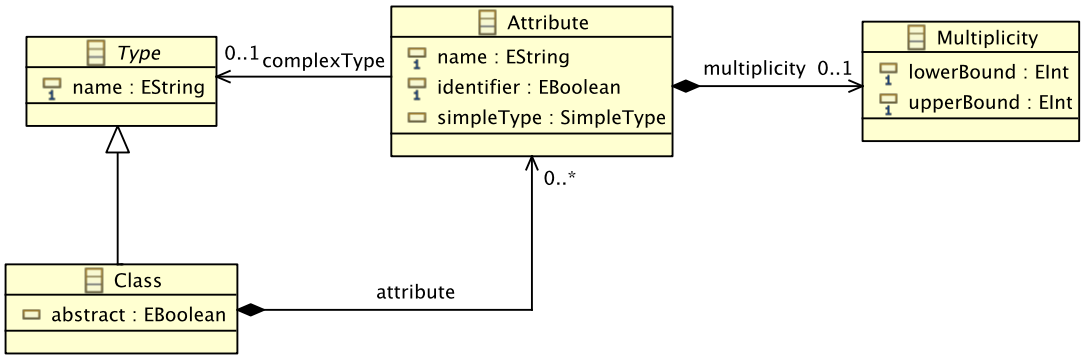

Abbildung 4.5: Einfache und komplexe Attribute

- *identifier*
  Beschreibt ob das Attribut ein Identifikator für das Objekt ist. Mögliche Werte sind *true* oder *false*.

- *simpleType*
  Der Java Typ des Attributes (String, Integer, Long, usw.).

- *complexType*
  Der Typ des Attributes. Verweist auf einen selbst definierten Typ innerhalb des Modells.

- *multipicity*
  Mittels diesem Attribut wird die Multiplizität des Attributes festgelegt. Wird keine Multiplizität angegeben, wird standardmäßig eine 1 zu 1 Beziehung angenommen.

- *constraint*
  Ist wohl das interessanteste Attribut. Es können hier beliebig viele Einschränkung (engl. constraints) auf ein Attribut definiert werden. Eine genaue Auflistung aller Einschränkungen gibt es in Kapitel 4.2.7

4.2.4 Element Beziehung

Alle Beziehungen werden direkt im Wurzelelement definiert. Dies hat folgende Gründe.

1. Da die meisten Beziehungen bidirektional sind (z.B. eine Person hat mehrere Adressen und eine Adresse gehört zu mehreren Personen) müsste man diese Beziehung zweimal definieren. Da jetzt diese Beziehungen nicht an den Objekten

hängen, sondern direkt im Wurzelelement *Modell*, muss diese Beziehung nur einmal definiert werden.

2. Es ist wesentlich leichter den Codegenerator zu implementieren.

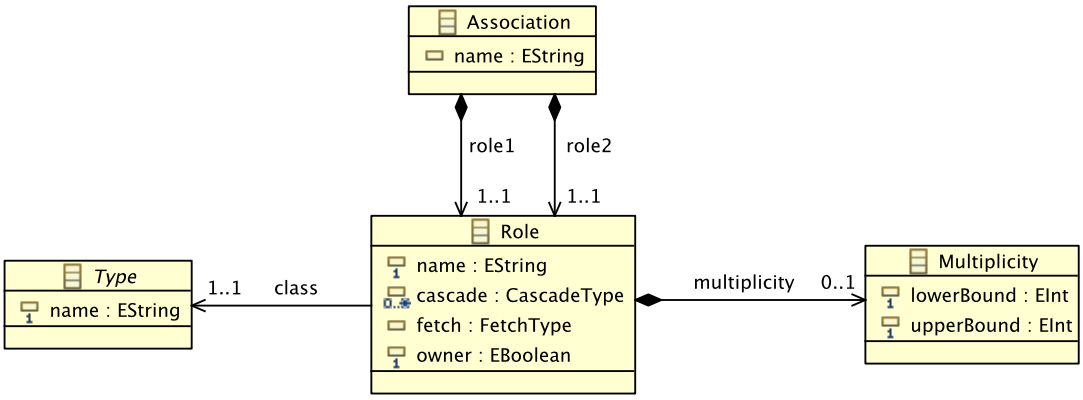

Abbildung 4.6: Beziehung und Kardinalität

Ein Element *Association*, beschreibt eine Beziehung zwischen zwei Typen und besteht aus folgenden Attributen:

- *name*
  Der Name der Beziehung, welcher ein Pflichtfeld und vom Typ String ist. Der Name kann Werte wie „wohnt“ oder „besitzt“ annehmen.
- *role1*
  Ist die Rolle 1 der bidirektionalen Beziehung. Eine Beziehung besteht immer zwischen zwei Typen. Da jetzt aber neben der Kardinalität noch zusätzliche Informationen benötigt werden, wird ein zusätzliches Element *Role* eingeführt. Dieses Attribut *Role1* ist vom Typ *Role* und ein Pflichtfeld.
- *role2*
  Die Rolle 2 der bidirektionalen Beziehung.

Element Rolle

Ein Element *Role* besteht aus folgenden Attributen:

- *name*
  Der Name der Rolle definiert den Java Variablennamen, der die zu referenzierte Klasse enthält.

- *cascade*
  Das Kaskadierungsattribut bestimmt, ob referenzierte Objekte, die in Abhängigkeit stehen aktualisiert oder gelöscht werden (siehe Abbildung 4.8).

- *fetch*
  Dieses Attribut bestimmt das Ladeverhalten der referenzierten Objekte. Hier wird zwischen sofortiges Laden (engl. eager loading) und Laden bei Bedarf (engl. lazy loading) unterschieden. „Sofortiges Laden“ bedeutet, dass alle Referenzen, die ein Objekt besitzt, egal ob es sich um eine 1:1, 1:n oder n:m Beziehung handelt, sofort beim Laden des Objektes selbst mitgeladen werden. Hingegen „Laden bei Bedarf“ bedeutet, dass die Referenzobjekte erst bei einem Zugriff per Getter „on the fly“ nachgeladen werden.

- *owner*
  Dieses Attribut definiert, welches der beiden Klassen der Besitzer dieser Beziehung ist. Darf nicht in beiden Rollen *true* sein.

- *class*
  Dieses Attribut bestimmt die eigentliche Beziehung der Klasse. Das Attribut enthält eine Referenz auf eine bereits definierte Klasse oder Enumeration im Modell.

- *multiplicity*
  Dieses Attribut bestimmt die Kardinalität der Beziehung. Die Kardinalität wird mittels einem eigenen Element *Multiplicity* bestimmt. Dieses Attribut ist optional. Wenn keine Multiplizität angegeben wird, wird eine 1:1 Beziehung angenommen.

Element Kardinalität

Ein Element *Multiplicity*, beschreibt die Kardinalität einer Beziehung und besteht aus folgenden Attributen:

- *lowerBound*
  Bestimmt den unteren Grad einer Beziehung. Das Attribut ist ein Pflichtfeld und vom Typ Integer.

- *upperBound*
  Bestimmt den oberen Grad einer Beziehung. Das Attribut ist ein Pflichtfeld und vom Typ Integer. Um keine obere Grenze festzulegen, ist der Wert *-1* einzugeben.

4.2.5 Element Vererbung

Ein sehr wesentlicher Teil der objektorientierten Welt ist die Vererbung. Vererbungen können mit dem Element *Generalization* beschrieben werden. Es wird genauso wie Klassen und Beziehungen direkt im Wurzelelement definiert.

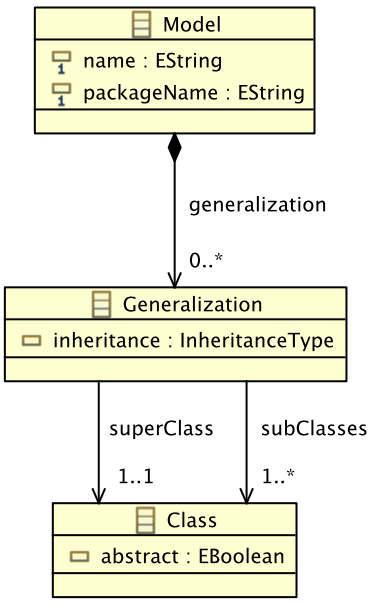

Abbildung 4.7: Vererbung

Ein Element *Generalization* besteht aus folgenden Attributen:

- *inheritance*
  Dieses Attribut bestimmt, wie Hibernate eine Klassenhierarchie in einer relationalen Datenbank abbildet. Es werden 3 Arten unterstützt: *JOINED*, *TABLE_PER_CLASS*, *SINGLE_TABLE* (siehe Abbildung 4.8).

- *superClass*
  Definiert eine Superklasse. Dieses Attribut ist optional und enthält eine Referenz auf eine bereits definierte Klasse im Modell.

- *subClasses*
  Dieses Attribut enthält alle möglichen Unterklassen.

4.2.6 Metamodell Enumeration

Das Metamodell selbst kennt 4 Enumerationen *SimpleType*, *CascadeType*, *FetchType* und *InheritanceType*. *CascadeType* und *FetchType* kommen bei Objektbeziehungen zum Einsatz. Die letzten 3 Enumerationen dienen dazu, um das Verhalten von Hibernate zu beeinflussen. In Abbildung 4.8 sind alle vorhanden Metamodell-Enumerationen zu sehen.

Enumeration SimpleType

Mit der Enumeration *SimpleType* werden die Datentypen der Attribute bestimmt. Folgende Typen werden unterstützt: *STRING*, *INTEGER*, *LONG*, *FLOAT*, *BOOLEAN*, *DATE* und *TIME*.

Enumeration CascadeType

Mit der Enumeration *CascadeType* kann die Kaskadierung der Objektbeziehungen bestimmt werden. Folgende Typen werden unterstützt: *ALL*, *PERSIST*, *MERGE*, *REMOVE* und *REFRESH*.

Enumeration FetchType

Mit der Enumeration *FetchType* kann das Datenbankladeverhalten beeinflusst werden. Folgende Typen werden unterstützt: *LAZY* und *EAGER*.

Enumeration InheritanceType

Mit der Enumeration *InheritanceType* kann die ORM-Abbildung definiert werden. Folgende Typen werden unterstützt: *JOINED*, *TABLE_PER_CLASS*, *SINGLE_TABLE*.

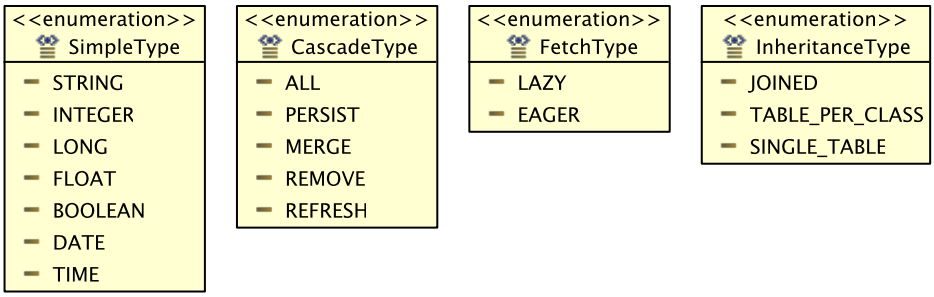

Abbildung 4.8: Metamodell-Enumerationen

4.2.7 Einschränkungen

Ein Hauptaugenmerk richtet sich auf die möglichen Einschränkungen. Jedes Attribut kann über mehrere dieser Elemente *Constraints* besitzen. Das abstrakte Element enthält ein Attribut *message*, mit dem eine eigene, auch internationalisierte Nachricht definiert werden kann. Die Einschränkungen werden in zwei große Blöcke unterteilt. Im Ersten

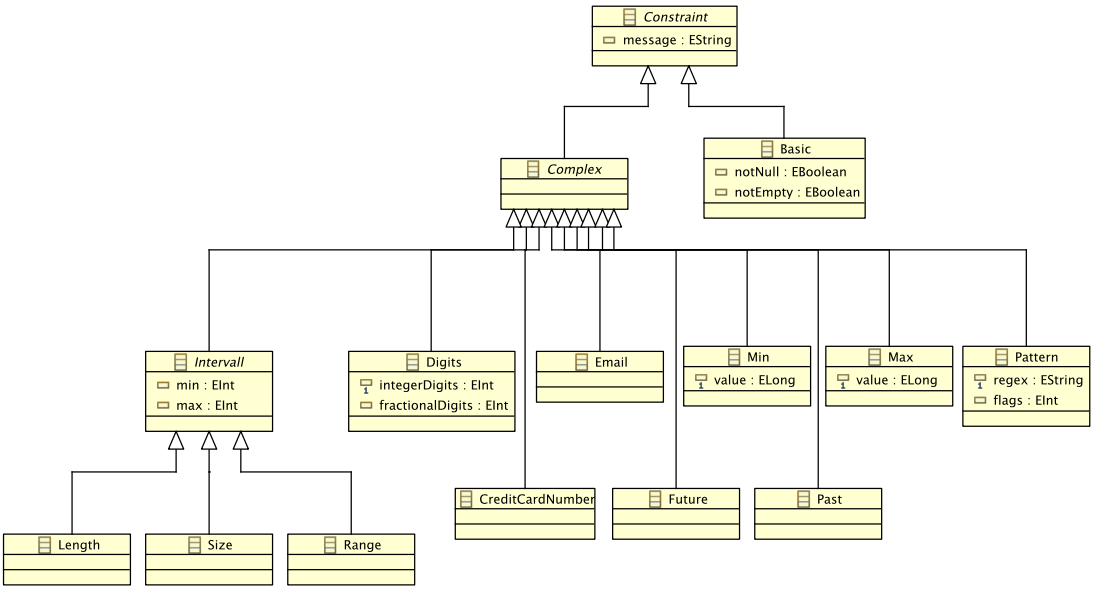

Abbildung 4.9: Einschränkungen

werden die Basiseinschränkungen realisiert (z.B. *NotNull* und *NotEmpty*). Der zweite Block besteht aus den erweiterten Einschränkungen:

- *Digits*
  Dieses Element definiert mittels dem Attribut *integerDigits* die Anzahl der Ziffern und mittels dem Attribut *fractionalDigits* die Anzahl der Ziffern nach dem Komma.

- *CreditCardNumber*
  Mittels diesem Element wird festgelegt, ob der Wert einer gültigen Kreditkartennummer entspricht.

- *Email*
  Mittels diesem Element wird festgelegt, ob der Wert einer gültigen Emailadresse entspricht.

- *Future*
  Überprüft, ob das eingegebene Datum in der Zukunft liegt.

- *Past*
  Überprüft, ob das eingegebene Datum in der Vergangenheit liegt.

- *Min*
  Überprüft die Zahl, ob diese größer als ein bestimmter Wert ist, welcher mit dem Attribut *value* definiert wird.

- *Max*
  Überprüft die Zahl, ob diese kleiner als ein bestimmter Wert ist, welcher mit dem Attribut *value* definiert wird.

- *Pattern*
  Überprüft den Wert, ob dieser einer bestimmten Regular Expression entspricht, welcher mit dem Attribut *regex* definiert wird.

- *Intervall*
  Überprüft ein Attribut, ob dieses in einem bestimmten Wertebereich, welcher mit *min* und *max* definiert wird liegt. Das Element *Intervall* selbst ist abstrakt. Mit dem Element *Length* wird diese Einschränkung auf einen String, mit dem Element *Size* auf ein Array, Collection oder Map und mit dem Element *Range* auf einen nummerischen Typen definiert.

Alle hier aufgelisteten Constraints wurden an die Hibernate-Validatoren 2.3.1 angelehnt.

4.3 Source Templates

In diesem Kapitel werden alle vorhanden Templates beschrieben, welche hauptsächlich die Logik des CRUD-Framework implementieren.

4.3.1 EntityHome

EnityHome ist eine generische abstrakte Klasse, welche eine Grundfunktionalität an Datenbankoperationen bereitstellt. In den Unterklassen wird per Generizität der Typ des jeweiligen Objektes festgelegt. Diese Klasse implementieren die *ActionListener* Methoden, welche von den JSF-Seiten aufgerufen werden können.
Folgende *ActionListener* Methoden werden unterstützt:

- *select*
  Lädt ein Objekt anhand seiner eindeutigen Id aus der Datenbank.

- *saveOrUpdate*
  Speichert ein neues Objekt, wenn keine Id angegeben wurde, bzw. aktualisiert das Objekt bei vorhandener Id in der Datenbank.

- *delete*
  Löscht ein Objekt anhand seiner eindeutigen Id aus der Datenbank.

Die Id wird jeweils per *ActionEvent* Parameter übergeben. Alle 3 Methoden öffnen bei jedem Request eine neue Transaktion, in welcher die Datenbankoperation durchgeführt wird. Bei erfolgreicher Durchführung der jeweiligen Aktion, wird eine FacesMessage erstellt, welche am Client dargestellt wird. Diese Nachricht kann mittels *messages.properties* internationalisiert werden. Falls ein unerwarteter Fehler Auftritt wird ein Rollback durchgeführt.

Folgende Methoden müssen von den Unterklassen von *EntityHome* überschrieben werden, welche mittels Codegenerator erstellt werden (siehe Listing 4.1).

```
1    protected abstract E createInstance();
2
3    protected abstract Class<E> getEntityClass();
4
5    protected abstract void setIdFromEntity(E e);
```

Listing 4.1: Abstrakte Methoden von EntityHome

4.3.2 EntityQuery

EntityQuery ist eine generische abstrakte Klasse, welche eine Methode *List<E> getResultList()* zu Verfügung stellt, die eine Liste vom generischen Typ *E* zurückliefert. Der generische Typ *E* wird durch die Unterklassen definiert und kann z.B. den Typ *Person* besitzen. Im folgenden Listing 4.2 ist fast die ganze *EntityQuery* Klasse abgebildet. Mittels der Variabel *ejbql* wird definiert, nach welcher Entität in der Datenbank gesucht werden soll. Mit der Variabel *resultCount* wird festgelegt, wieviele Resultate maximal retourniert werden. Die eigentliche Funktionalität ist dann in der Methode *List<E> getResultList()* implementiert. Es wird zuerst eine neue Transaktion erstellt, anschließend die Query gegen die Datenbank abgesetzt und zuletzt das Ergebnis retourniert.
Um diese Liste in der JSF-Seite darzustellen ist einerseits die Unterklasse zu implementieren, diese Klasse als JSF-Bean zu registrieren und die JSF-Seite zu schreiben. All dies wird mittels dem Codegenerator erstellt. Mehr dazu in Kapitel 4.4.

```
1  public abstract class EntityQuery<E>
2  {
3    private Integer              resultCount;
4    private String               ejbql;
5    private Integer              maxResults = 25;
6
7    protected static EntityManager getEntityManager() { \ldots }
8
9    protected abstract Class<E> getEntityClass();
10
11   // Getters & Setters fuer Variablen
12
13   public List<E> getResultList() {
14     List<E> resultList;
15     EntityTransaction tx;
16     tx = getEntityManager().getTransaction();
```

```
17        tx.begin();
18        resultList = getEntityManager().createQuery(ejbql).setMaxResults(
              maxResults).getResultList();
19        resultCount = resultList.size();
20        tx.commit();
21
22        if (resultList == null)
23          throw new RuntimeException("Entity not found");
24        getEntityManager().close();
25        return resultList;
26      }
```

Listing 4.2: Abstrakte EntityQuery Klasse

```
1  public class PersonList extends EntityQuery<Person> {
2      private static final String EJBQL = "from Person person";
3
4      private Person person = new Person();
5
6      public PersonList() {
7          setEjbql(EJBQL);
8          setMaxResults(25);
9      }
```

Listing 4.3: Bean PersonList

4.3.3 Konverter (EnumTypeConverter)

Wie bereits in Kapitel 2.3.2 *Phase 3: Konvertierung und Validierung durchführen (Process Validations)* beschrieben, wird in der Phase 3 des JSF-Lebenszyklus eine Konvertierung von Objekten durchgeführt. Da es laut Metamodell möglich ist Enumerationen zu erstellen, müssen diese auch auf der JSF-Seite angezeigt werden können. Das folgende Listing 4.4 zeigt eine Enumeration, welches ein Geschlecht repräsentiert. Um jetzt diese Enumeration darstellen zu können wird ein Konverter benötigt. Dieser Konverter macht nichts anderes als aus dem Typ *Gender.MALE* einen String „männlich“ zu generieren. Diese Richtung der Konvertierung erfolgt mit der Methode *public String getAsString(FacesContext context, UIComponent component, Object object) throws ConverterException*. Umgekehrt wird aus dem String „male“ die Enumeration *Gender.MALE* erstellt. Der String hier ist *male* und nicht *männlich*, da die Enumeration immer in einem *SELECT - OPTION* Tag (Markup-Element) dargestellt werden. Ein *OPTION* Tag hat jetzt ein Attribut *VALUE*, welches den Key und nicht den Wert der Enumeration enthält. Aus diesem Grund ist die Rückkonvertierung per Key und erfolgt mit der Methode *public Object getAsObject(FacesContext context, UIComponent component, String value) throws ConverterException*. Wichtig ist, dass der Konverter das Interface *Converter* implementiert und somit die vorher beschriebenen Methoden überschrieben werden. Zusätzlich implementiert der *EnumTypeConverter* ebenfalls eine Internationalisierung.

```
1  public enum Gender {
2    MALE, FEMALE
3  }
```

Listing 4.4: Enumeration Geschlecht

4.3.4 Hibernate Validator Tag

Der Hibernate Validator Tag ist ein selbst geschriebener Tag, welcher eine Validierung mittels Hibernate Validator durchführt. Die Validierung passiert in der *Phase 3: Konvertierung und Validierung durchführen (Process Validations)* siehe Kapitel 2.3.2. Falls hier Fehler auftreten, werden entsprechende FacesMessages erstellt und diese unmittelbar nach dieser Phase dem Browser zurückgeliefert und dargestellt.

Validation Tag TLD

Um einen eigenen Tag in den JSF-Seiten verwenden zu können, muss eine TLD[1] geschrieben werden. Eine TLD dient in der Java Welt als Meta-Beschreibungsdatei zur Definition eigener Tags. Im Listing 4.5 wird der *HibernateValidatorTag* beschrieben. Es werden folgende Methoden unterstützt:

- *bundle*
  Mit diesem Attribut kann ein *ResourceBundle* definiert werden um eine Internationalisierung zu ermöglichen. Dieses Feld ist optional. Falls kein Wert vorhanden, wird in der generierten Fehlermeldung der Name des Bean Attributes verwendet.

- *bean*
  Mit diesem Attribut wird die Entität definiert, in welchem unterschiedliche Attribute validiert werden. Dieses Feld ist ebenfalls optional. Falls kein Wert vorhanden, wird mittels dem Vaterelement, welches ein *UIComponentTag* sein muss das zu validierende Bean bestimmt. Der Wert wird mittels EL übergeben.

- *property*
  Mit diesem Attribut wird das eigentliche Attribut definiert, welches zu validieren ist. Wie schon beim Attribut *bean* ist dieses optional und wird ebenfalls durch das Vaterelement bestimmt, falls dieses Attribut nicht definiert wurde.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2    <taglib version="2.0">
3    <tlib-version>1.0</tlib-version>
4    <short-name>hibVal</short-name>
```

[1] Tag Library Descriptor

```
5     <uri>/WEB-INF/hibernateValidator</uri>
6     <tag>
7       <name>validate</name>
8       <tag-class>crud.validation.HibernateValidatorTag</tag-class>
9       <body-content>empty</body-content>
10      <attribute>
11        <name>bundle</name>
12      </attribute>
13      <attribute>
14        <name>bean</name>
15      </attribute>
16      <attribute>
17        <name>property</name>
18      </attribute>
19    </tag>
20  </taglib>
```

Listing 4.5: TLD Hibernate Validator Tag

Validation Tag Klasse

Die Klasse die mittels der TLD Datei referenziert wird, muss allerdings alle definierten Attribute und deren Setter implementieren. Die *HibernateValidatorTag* Klasse muss von der *javax.faces.webapp.ValidatorTag* erben und die Methode *protected Validator createValidator() throws JspException* überschreiben. In dieser Methode wird die eigentliche Validierung implementiert (siehe Zeile #14 in Listing 4.6). Anschließend werden alle gesetzten Properties auf Korrektheit überprüft.

```
1  package crud.validation;
2
3  public class HibernateValidatorTag extends ValidatorTag {
4    private String              bundle;
5    private String              bean;
6    private String              property;
7
8    // Setters
9
10   @Override
11   protected Validator createValidator() throws JspException {
12     setValidatorId(HibernateValidator.VALIDATOR_ID);
13     FacesContext facesContext = FacesContext.getCurrentInstance();
14     HibernateValidator validator = (HibernateValidator) super.
           createValidator();
15
16     if (bean != null) {
17       if (!UIComponentTag.isValueReference(bean)) {
18         throw new IllegalArgumentException("Attribute bean must be a value
               expression");
19       }
20       ValueBinding vb = facesContext.getApplication().createValueBinding(
             bean);
21       validator.setBean(vb.getValue(facesContext));
22     }
23     if (property != null) {
```

```
24          ...
25        }
26        if (bundle != null) {
27          ...
28        }
29        return validator;
30      }
31    }
```

Listing 4.6: Hibernate Validator Tag Klasse

Implementierung der Validierung

Die eigentliche Implementierung der Validierung ist in Listing 4.7 zu sehen. In der Methode *public void validate(FacesContext context, UIComponent component, Object value) throws ValidatorException* wird zuerst überprüft, ob bereits eine Instanz von *ClassValidator* erstellt wurde. Falls nicht, wird in der Methode *private void initValidator(FacesContext context, UIComponent component)* eine dementsprechende Instanz erzeugt. Hier sei zu erwähnen, dass falls per *HibernateValidatorTag* die Properties *validatedProperty* und/oder *validatedBean* nicht gesetzt wurden, werden diese Attribute mittels dem Vaterelement gesetzt. Nach der Initialisierung des Validators, wird mittels Hibernate Validator eine Validierung, wie in Kapitel 2.3.1 beschrieben durchgeführt. Ab Zeile #36 in Listing 4.7 wird bei einem aufgetretenen Fehler das Styling des *UIComponent* geändert und eine falls möglich internationalisierte Fehlermeldung erstellt.

```
1  package crud.validation;
2
3  public class HibernateValidator implements Validator
4  {
5    private String                  validatedProperty;
6    private Object                  validatedBean;
7    private ClassValidator<Object> validator;
8    private ResourceBundle          bundle;
9
10   public void validate(FacesContext context,
11                        UIComponent component,
12                        Object value) throws ValidatorException {
13     if (validator == null)
14       initValidator(context, component);
15     log.debug("validate value: " + value + ", type: " + value.getClass().
         getSimpleName());
16
17     InvalidValue[] invalids = null;
18     try {
19       // keep old value of bean
20       Object oldValue = PropertyUtils.getProperty(validatedBean,
           validatedProperty);
21
22       // set new value to bean
23       BeanUtils.setProperty(validatedBean, validatedProperty, value);
24
25       // Hibernate Validator
```

```
26          invalids = validator.getInvalidValues(validatedBean,
                validatedProperty);
27          log.debug("found " + invalids.length + " validation errors");
28
29          // restore old value
30          PropertyUtils.setProperty(validatedBean, validatedProperty, oldValue
                );
31        }
32        catch (Exception e) {
33          ...
34        }
35        StringBuffer stringBuffer = new StringBuffer();
36        for (InvalidValue invalid : invalids) {
37          String style = (String) component.getAttributes().get("styleClass");
38          ...
39          component.getAttributes().put("styleClass", style + " error_input");
40
41          String i18nProperty = bundle.getString(validatedProperty);
42          stringBuffer.append(i18nProperty + " " + invalid.getMessage()).
                append(" ");
43        }
44        if (stringBuffer.length() > 0) {
45          throw new ValidatorException(new FacesMessage(stringBuffer.toString
                ()));
46        }
47      }
48    }
```

Listing 4.7: HibernateValidator Implementierung

Verwendung der Validierung

Um jetzt die Valdierung verwenden zu können, muss zuletzt noch in der Konfigurationsdatei *faces-config.xml* der Validator definiert werden. Siehe Listing 4.8.

```
1  <validator>
2    <validator-id>HibernateValidator</validator-id>
3    <validator-class>crud.validation.HibernateValidator</validator-class>
4  </validator>
```

Listing 4.8: Hibernate Validator Tag Definition

Jetzt kann in den JSP Seiten dieser Validator Tag verwendet werden. Siehe Listing 4.9.

```
1  <%@ taglib uri="/WEB-INF/hibernateValidator" prefix="val" %>
2  ...
3  <h:inputText id="tbFirstname" value="#{person.instance.firstname}"
       required="true" styleClass="input_field" >
4    <val:validate bean="#{person.instance}" property="firstname" bundle="
         messages"/>
5  </h:inputText>
```

Listing 4.9: Verwendung von Hibernate Validator Tag

4.3.5 Pflichtfeld Validierung

Obwohl der Hibernate Validator von sich aus Validatoren wie *@NotNull* und *@NotEmpty* unterstützt, kann diese Überprüfung nicht mit dem Ansatz wie im vorigen Kapitel 4.3.4 beschrieben realisiert werden. JSF greift hier leider ganz am Anfang in der Phase Validierung ein. Die JSF Input Komponente unterstützt ein Attribut *required*, mit dem ein Pflichtfeld definiert werden kann. Dieses Attribut wird mittels dem Codegenerator generiert. Die Klasse *RequiredFieldValidatorPhaseListener* registriert sich als Listener in der Phase 3 (Siehe Listing 4.10). In dieser Klasse wird überprüft ob das aktuelle Feld ein Pflichtfeld ist. Falls dies der Fall ist und nichts vom Benutzer eingegeben wurde, erstellt diese Klasse eine *FacesMessage* und ändert das CSS[2] Styling des Eingabefeldes. Dies alles passiert ohne dem Hibernate Validator Framework.

```
1 <lifecycle>
2   <phase-listener>crud.listener.RequiredFieldValidatorPhaseListener</phase
      -listener>
3 </lifecycle>
```

Listing 4.10: RequiredFieldValidatior Listener

4.3.6 Internationalisierung

Da eine Internationalisierung vor allem auf Web Seiten sehr wichtig ist, wird dies auch vom CRUD-Framework unterstützt. Alle verwendenten Technologien wie z.B. der Hibernate Validator oder JSF bieten ebenfalls eine Internationalisierung. Im *src-gen/META-INF* Verzeichnis existiert eine *messages.properties* Datei, in der die internationalisierten Texte definiert werden. Da die Internationalisierung noch nicht vollständig implementiert wurde, enthält die aktuelle Version lediglich einen Ansatz. Diese Datei wird mit dem aktuellen Ansatz noch nicht generiert. In dieser Datei sind die Bezeichnungen der Standardaktionen wie z.B.: speichern, löschen, zurück, usw. definiert. Ebenso werden hier die Bezeichnungen, die in dem Beispielmodell definierten Objekten mit ihren Attributen definiert.
Um jetzt eine vernünftige Internationalisierung zu ermöglichen muss diese Datei ebenso generiert werden. Dh. es muss über alle Objekte und deren Attribute iteriert werden und jeweils ein Schlüssel für jede Bezeichnung generiert werden. Den eigentlichen Text, der letztendlich dann auch im Web Interface angezeigt wird, muss klarerweise vom Entwickler selbst eingegeben werden. Für die Standardbezeichnungen, wie speichern, löschen, usw. kann ein Template, welches mehrere Sprachen unterstützt geschrieben werden. Wichtig ist noch, ob mittels *Code Protection* oder anderen Mitteln sicherzustellen, dass ein bereits übersetzter Text nicht wieder mit dem Codegenerator überschrieben wird.

[2]Cascading Style Sheets

Um jetzt die Internationalisierung auf den generierten Webseiten verwenden zu können, muss auf den JSF-Seiten lediglich die Nachrichten Datei mittels *<f:loadBundle basename="messages" var="msg"/>* geladen werden (diese Datei muss sich im Java Classpath befinden). Die eigentlich Ausgabe des Textes erfolgt mit dem JSF Element *outputText*.

4.4 Codegenerierung

Die eigentliche Codegenerierung erfolgt mittels dem openArchitectureWare (oAW) Framework. Um eine hohe Wiederverwendbarkeit und Erweiterbarkeit zu erreichen wurden bei der Implementierung Cartridges (deutsch: Module) verwendet. Diese Modulfähigkeit wird direkt vom Framework zu Verfügung gestellt. Folgende 4 Cartridges wurden implementiert: Java, JSF, Persistence und Validation.

4.4.1 Java Cartridge

In dieser Cartridge-Implementierung werden einfache POJOs mit Attributen, Gettern und Settern generiert. Es werden hier auch die Assoziationen der einzelnen Beziehungen berücksichtigt. Im folgenden Listing 4.11 ist die Hauptmethode *class* zur Generierung einer Klasse dargestellt. Diese Methode verwendet weitere Methoden, wie *classImport*, *attribute*, *classSetterAndGetter*, *mappingSetterAndGetter* und *toString*, welche in den folgenden Listings noch genau erklärt werden. Die Methode *class* wird für jede definierte Klasse im Modell aufgerufen und erzeugt jeweils eine Datei *<PACKAGE_NAME>/persistence/<JAVA_KLASSE>.java*. Das Java Cartridge ist unabhängig von anderen Cartridges.

```
1  «DEFINE class FOR Class»
2    «FILE getFqPackageName() + "/persistence/" + name.toFirstUpper() + ".
       java"»
3    «("generate class " + name).info() -> null-»
4  package «getFqPackageName()».persistence;
5
6    «EXPAND classImport FOR this»
7
8    «EXPAND classAnnotation FOR this»
9  public «IF abstract»abstract«ENDIF» class «name.toFirstUpper()» «IF
       hasSuperClass()» extends «getSuperClass().name.toFirstUpper()» «
     ENDIF» implements Serializable {
10
11   «REM»Class attributes«ENDREM»
12   «EXPAND attribute FOREACH attribute-»
13
14   «REM»Mapping attributes«ENDREM»
15   «FOREACH getAssociation() AS association»
16     «LET getMappingAttribute(association) AS attribute»
17       «REM»«getMappings(attribute, getRole(association, attribute))»«
           ENDREM»
```

```
18          «("store role " + ((Role)storeGlobalVar("role", getRole(association
                , attribute))).name).info() -> null-»
19          «EXPAND attribute FOR attribute-»
20        «ENDLET»
21      «ENDFOREACH»
22
23      «REM» Class Setters & Getters «ENDREM»
24      «EXPAND setterAndGetter FOREACH attribute»
25
26      «REM» Mapping Setters & Getters «ENDREM»
27      «FOREACH getAssociation() AS association»
28        «EXPAND stterAndGetter FOR getMappingAttribute(association)-»
29      «ENDFOREACH»
30
31      «REM» toString «ENDREM»
32      «EXPAND toString FOR this»
33  }
34      «ENDFILE»
35  «ENDDEFINE»
```

Listing 4.11: Methode zur Generierung von Java Klassen

Methode „classImport“

Im Listing 4.12 ist die Methode *classImport* abgebildet, welche den Importteil einer Java Klasse generiert. Es werden hier nur die notwendigen Imports generiert. Dh. wenn die aktuelle Klasse keine 1:n Relation besitzt, wird auch nicht das Package *java.util.List* importiert.

```
1  «DEFINE classImport FOR Class»
2    «("DEFINE-JAVA classImport").info() -> null-»
3    import java.io.Serializable;
4    «IF existListAttriute()»import java.util.List;«ENDIF-»
5  «ENDDEFINE»
```

Listing 4.12: Methode zur Generierung von Imports

Methode „attribute“

Die Methode *attribute* generiert für alle definierten Attribute die Objektvariablen mit der Sichtbarkeit *protected*. Wenn das Attribut laut Modell einen komplexen Typen enthält, dh. der Wert ist nicht *null*, dann wird dieser auch verwendet und je nach Relation die Definition der Variable generiert. Dabei wird beachtet, dass die Java Typen immer einen großen und die Java Variablen immer einen kleinen Anfangsbuchstaben besitzen. Falls kein komplexer Typ angegeben wurde, wird der einfache Typ verwendet, welcher Werte wie *String, Integer, Long, ...* annehmen kann. Es ist hier jedenfalls durch eine eigene *Check* Regel sichergestellt, dass entweder der simpel oder komplexe Typ vorhanden ist.

```
1  «DEFINE attribute FOR Attribute»  «("DEFINE-JAVA attribute").info() ->
       null»
```

```
2     «("write attribute [" + getFqTypeName() + "] " + name).info() -> null
          »
3     «IF complexType != null»
4       «IF isMulti()»
5       protected List«<complexType.name»> «name.toFirstLower()»;
6       «ELSE»
7       protected «complexType.name» «name.toFirstLower()»;
8       «ENDIF»
9     «ELSE»
10      «IF isMulti()»
11      protected List«<getFqTypeName()»> «name.toFirstLower()»;
12      «ELSE»
13      protected «getFqTypeName()» «name.toFirstLower()»;
14      «ENDIF»
15    «ENDIF»
16  «ENDDEFINE»
```

Listing 4.13: Methode zur Generierung der Variablen

Methode „setterAndGetter“

Diese Methode wird einerseits für alle definierten Attribute einer Klasse aufgerufen und andererseits für alle Beziehungen, die diese Klasse zu anderen Klassen besitzt. Wie in Listing 4.14 zu sehen ist, wird für jedes übergebene Attribut zuerst ein Getter und anschließend ein Setter erstellt.

```
1  «DEFINE setterAndGetter FOR Attribute»
2    «getGetter()»
3    «getSetter()»
4  «ENDDEFINE»
```

Listing 4.14: Methode zur Generierung von Settern und Gettern

Methode „toString“

Die Methode *toString* generiert die Java Methode *public String toString()*, welche die Methode *toString* aus der Superklasse überschreibt. Laut dem Buch Effective Java [9] Item 10, soll die *toString* Methode immer überschrieben werden. In der Methode selbst wird standardmäßig allerdings wieder die *toString* Methode der Superklasse aufgerufen, was somit die Generierung dieser Methode überflüssig machen würde, wenn hier nicht ein geschützter Codeblock mitgeneriert wird. Dh. der Entwickler kann nach der ersten Generierung innerhalb des geschützten Bereiches seinen eigenen Code implementieren, welcher nicht mehr vom Codegenerator überschrieben wird. Somit lässt sich sehr einfach und komfortabel die Methode *toString* nach belieben anpassen und bleibt bei zukünftiger Generierung erhalten.

```
1  «DEFINE toString FOR Class»
2    @Override
3    public String toString()
```

```
4     {
5     «PROTECT CSTART '/*' CEND '*/' ID this.name + ".toString"»
6       return super.toString();
7     «ENDPROTECT»
8     }
9   «ENDDEFINE»
```

Listing 4.15: Methode zur Generierung von toString

4.4.2 JSF Cartridge

Das JSF-Modul generiert alle JSF relevanten Dateien. Dies sind einerseits die JSF-Beans, mit welchen die Domänenobjekte verwaltet werden, alle Viewseiten und zuletzt die Konfigurationsdatei *faces-config.xml*. Um die Übersicht zu wahren, existieren 3 Templates, welche an das MVC Muster angepasst sind. Das *JSF Beans* Template repräsentiert den Controller, das *JSF Pages* Template die View und das in bereits beschriebenen Kapitel *Java-Cartridge* 4.4.1 das Modell. Das JSF-Cartridge ist unabhängig von anderen Cartridges.

Beans

Es werden zwei unterschiedliche Beans generiert. Eines für die Anzeige der Objekte in einer Liste und ein weiteres für die CRUD-Funktionalität der Objekte. In Listing 4.16 wird so ein Listen Bean generiert, welches von *EntityQuery* erbt. Siehe Kapitel 4.3.2. Im wesentlichen wird hier ein JPQL[3] String generiert, mit dem eine Liste vom gewünschten Typ aus der Datenbank geladen wird. Der Typ des eigentlichen Objektes wird mittels Generizität in der Klassendefinition bestimmt.

```
1   «DEFINE entityList FOR Class»
2     «FILE "crud/bean/" + name.toFirstUpper() + "List.java"-»
3     «("generate list bean " + name).info() -> null-»
4   package crud.bean;
5
6   import «getFqPackageName()».persistence.«name.toFirstUpper()»;
7
8   public class «name.toFirstUpper()»List extends EntityQuery«<name.
        toFirstUpper()»> {
9     private static final String EJBQL = "from «name.toFirstUpper()» «name
          .toFirstLower()»";
10    private «name.toFirstUpper()» «name.toFirstLower()» = new «name.
          toFirstUpper()»();
11
12    public «name.toFirstUpper()»List() {
13      setEjbql(EJBQL);
14      setMaxResults(25);
15    }
16
17    public «name.toFirstUpper()» get«name.toFirstUpper()»() {
```

[3] Java Persistence Query Language

```
18        return «name.toFirstLower()»;
19      }
20
21      @Override
22      protected Class«<name.toFirstUpper()»> getEntityClass() {
23        return «name.toFirstUpper()».class;
24      }
25    }
26      «ENDFILE»
27    «ENDDEFINE»
```

Listing 4.16: Methode zur Generierung eines Listen Beans

Etwas komplexer gestaltet sich das Entity Bean. Dies lässt sich dadurch erklären, dass bei einer Relation nicht nur der aktuelle Wert dargestellt werden soll, sondern auch alle anderen möglichen Werte. Zur Veranschaulichung nehmen wir ein Objekt *Person*, welches als Geschlecht männlich oder weiblich annehmen kann. Wünschenswert wäre es auf der GUI Seite eine Dropdown Liste mit diesen zwei Werten zu haben. Und genau diese Daten müssen ebenfalls von dem Entity Bean zu Verfügung gestellt werden. In der aktuellen Version, werden nur für die Enumeration diese Methoden zur Befüllung der Dropdown Liste generiert (Siehe Listing 4.17). Um das Ganze für Objektrelationen zu erweitern müsste lediglich über die *association* Liste iteriert werden und für jede Relation ebenfalls eine Methode generiert werden, die diese Objekte in einer Liste zurückliefert. Ebenfalls müsste ein JSF-Konverter für diese Objekte geschrieben werden.

```
1   «FOREACH attribute.select(a | a.complexType.metaType == Enum) AS a-»
2   public List<SelectItem> «asGetter(a)»List() {
3     List<SelectItem> list = new ArrayList<SelectItem>(«a.asTypeName()».
          values().length + 1);
4     list.add(new SelectItem("null", ""));
5     for («a.asTypeName()» «a.asAttributeName()» : «a.asTypeName()».
          values()) {
6       list.add(new SelectItem(«a.asAttributeName()».name().toString(),
            getI18n(«a.asAttributeName()».name().toLowerCase())));
7     }
8     return list;
9   }
10  «ENDFOREACH»
```

Listing 4.17: Methode zur Befüllung von Dropdowns

Pages

Das zweite Template generiert alle Clientseiten, die im Browser nach dem JSF-Rendering dargestellt werden. Es werden 4 unterschiedliche Ansichten generiert. Die Erste dient zur Darstellung von mehreren Objekte eines bestimmten Typs in einer Liste. Zusätzlich wird in jeder Zeile ein *Ändern*, *Ansicht* und *Löschen* Button generiert. Unterhalb der Tabelle wird ein weiterer Button *Hinzufügen* und die gesamte Anzahl der gefundenen

Einträge dargestellt. In Listing 4.18 ist die Methode *genListPage* dargestellt, welche diese Übersichtseite generiert. Diese Datei, sowie alle anderen Ansichtseiten werden im Ordner *WebContent* abgelegt. In der Zeile *12* wird ein Menü inkludiert, welches selbst generiert wird. Dieses Menü enthält die Links zu den Übersichtseiten wie z.B. Personen, Rollen oder Adressen. In Zeile *14* wird das JSF-Tag <*h:dataTable*> verwendet. Dieser Tag generiert in der JSF-Rendering Phase eine Tabelle mit mehreren Spalten, welche mit <*h:column*> definiert werden. Die darzustellende Liste in der Tabelle wird mit dem Attribut *value* bestimmt. Mit dem Attribut *var* wird die Iterationsvariable festgelegt. Es wird hier einfach der erste Buchstabe der Klasse verwendet. In Zeile *15 - 20* wird dann über alle Attribute der Klasse iteriert und jeweils ein <*h:column*> Element erstellt. Als Header wird ein internationalisierter Text ausgegeben, welcher mit <*f:facet name=„header“*> zu definieren ist. Die Werte selbst, welche in der Rendering Phase ausgegeben werden, werden mit dem Element <*h:outputText*> definiert. Hier muss dem Attribut *value* die Instanzvariable und das Objektattribut angegeben werden (z.B. „#p.lastname“). Genauer gesagt, wird hier der Getter ohne „get“ angegeben, was laut Javakonvention wieder der Variabel entspricht. In den Zeilen *23*, *26*, *29* werden die Aktionbuttons (Ansicht, Bearbeiten, Löschen) erstellt. Am Ende der Liste wird noch die gefundene Anzahl der Objekte mit dem Element <*h:outputText*> ausgeben. In Zeile *35* wird noch zuletzt ein Button *Neu* erstellt um neue Instanzen der aktuellen Entität erstellen zu können.

```
1  «DEFINE genListPage FOR Class»
2  «FILE "WebContent/" + getJSPListFileName() -»
3  «EXPAND genTaglib FOR this»
4  <html>
5    <head>
6      <title>«name.toFirstUpper()» List</title>
7      <link rel="stylesheet" media="screen" href="/crud/style.css" type="
           text/css" />
8    </head>
9  <body>
10 <f:view>
11   <center>
12     <jsp:include page="menu.jsp" />
13     <h:form id="f«name.toFirstUpper()»List" styleClass="list">
14     <h:dataTable id="dt«name.toFirstUpper()»List" value="#{«
           getListBeanName()».resultList}" var="«name.toFirstLower().
           subString(0,1)»">
15      «FOREACH attribute AS attribute-»
16        <h:column>
17          <f:facet name="header"><h:outputText value="#{msg.«
                asAttributeName(attribute)»}"/></f:facet>
18          <h:outputText value="#{«name.toFirstLower().subString(0,1)».«
                asAttributeName(attribute)»}" />
19        </h:column>
20      «ENDFOREACH»
21        <h:column>
22          <f:facet name="header">...</f:facet>
23          <h:commandButton id="edit" value="#{msg.edit}" type="submit"
                action="«getEditName()»" actionListener="#{«
```

```
                getEntityBeanName() » .select}">
24              <f:param name="id" value="#{ « name.toFirstLower().subString(0,1)
                    » .id}" />
25            </h:commandButton>
26            // Buttons for View and Edit ...
27          </h:column>
28        </h:dataTable>
29        <h:outputText value="#{msg.count}" />: <h:outputText value="#{ «
              getListBeanName() » .resultCount}" />
30        <h:commandButton id="add" value="#{msg.add}" type="submit" action="«
              getEditName() »" actionListener="#{ « getEntityBeanName() » .init}"/>
31      </h:form>
32      </center>
33    </f:view>
34    </body>
35    </html>
36    « ENDFILE »
37    « ENDDEFINE »
```

Listing 4.18: Methode zur Erstellung der Übersichtseiten

In Listing 4.19 ist die Methode *genViewPage* dargestellt, welche die Detailseite für eine Entität generiert. Der wesentliche Unterschied zu den Übersicht Seiten besteht darin, dass hier kein Element *<h:dataTable>* verwendet wird, sondern stattdessen der Name und Wert des Attributes in einem Element *div* dargestellt werden. In der aktuellen Version werden nur 1:1 Relationen unterstützt. Alle anderen Kardinalitäten müssen erst implementiert werden.

```
1   « DEFINE genViewPage FOR Class »
2   « FILE "WebContent/" + getJSPViewFileName() -»
3   « EXPAND genTaglib FOR this »
4   <html>
5     <head>
6       <title>« getTitleName() »</title>
7       <link rel="stylesheet" media="screen" href="/crud/style.css" type="
              text/css" />
8     </head>
9     <body>
10      <f:view>
11      <center>
12      <h:form>
13      « FOREACH attribute AS attribute -»
14        <div class="form_row">
15          <h:outputText id="« getLabelId(attribute) »" value="#{msg.« 
                asAttributeName(attribute) »}:" styleClass="view_label"/>
16          « IF !isMulti(attribute)-»
17          <h:outputText id="« getInputTextId(attribute) »" value="#{« 
                getBeanName() » .instance.« asAttributeName(attribute) »}"
                styleClass="view_value"/>
18          « ENDIF-»
19        </div>
20      « ENDFOREACH-»
21        <h:outputLink value="« getBeanName() »_list.faces"><h:outputText
              value="#{msg.back}"/></h:outputLink>
22      </h:form>
23      </center>
```

```
24        </f:view>
25      </body>
26    </html>
27    «ENDFILE»
28    «ENDDEFINE»
```

Listing 4.19: Methode zur Erstellung der Ansichtseiten

In Listing 4.20 ist die Methode *genEditPage* dargestellt, welche eine Detailseite mit Bearbeitungsfunktionalität für eine Entität generiert. Der Aufbau der Seite ist sehr ähnlich der normalen Ansichtseite. Es wird lediglich der Wert in einem Input Element dargestellt, welches bei einem *submit* die geänderten Daten wieder zum Server schickt. Für simple Objektattribute werden *<h:inputText>* und für komplexe Objektattribute (wie z.B. Enumerationen) werden *<h:selectOneListbox>* Elemente verwendet. Es werden hier noch keine 1:n oder m:n Kardinalitäten unterstützt. Diese können aber jederzeit hinzugefügt werden. Es ist nur zu beachten, dass bei einer 1:n oder m:n Beziehung das *EntityHome* Bean ebenfalls zu erweitern ist, um eine Liste aller möglichen Einträge zu bekommen (siehe Listing 4.4.2).

```
1   «DEFINE genEditPage FOR Class»
2   «FILE "WebContent/" + getJSPEditFileName() -»
3   «EXPAND genTaglib FOR this»
4   <html>
5     <head>
6       <title>«getTitleName()»</title>
7       <link rel="stylesheet" media="screen" href="/crud/style.css" type="
            text/css" />
8     </head>
9     <body>
10      <f:view>
11      <center>
12      ...
13      <h:form>
14       «FOREACH attribute AS attribute-»
15        <div class="form_row">
16          <h:outputLabel id="«getLabelId(attribute)»" for="«
                getInputTextId(attribute)»" value="#{msg.«asAttributeName(
                attribute)»}" «IF isRequired(attribute)» styleClass="required
                " «ENDIF» />
17          «IF !isMulti(attribute) && !isEnum(attribute)-»
18          <h:inputText id="«getInputTextId(attribute)»" value="#{«
                getBeanName()».instance.«asAttributeName(attribute)»}" «IF
                isRequired(attribute)» required="true" «ENDIF» styleClass="
                input_field" «IF attribute.identifier == true» readonly="true"
                  «ENDIF»>
19            <val:validate bean="#{«getBeanName()».instance}" property="«
                  asAttributeName(attribute)»" bundle="messages"/>
20          </h:inputText>
21          «ENDIF-»
22          «IF !isMulti(attribute) && isEnum(attribute)-»
23          <h:selectOneListbox id="«getDropDownId(attribute)»" «IF
                isRequired(attribute)» required="true" «ENDIF» styleClass="
                input_field" value="#{«getBeanName()».instance.«
                asAttributeName(attribute)»}">
24            <f:selectItems value="#{«getBeanName()».genderList}" />
```

```
25              <val:validate bean="#{ «getBeanName() ».instance}" property=" «
                    asAttributeName(attribute) » " bundle="messages"/>
26            </h:selectOneListbox>
27            « ENDIF– »
28          </div>
29        « ENDFOREACH– »
30        // Buttons for Save, Delete and Cancel
31        </h:form>
32        </center>
33        </f:view>
34      </body>
35    </html>
36    « ENDFILE »
37    « ENDDEFINE »
```

Listing 4.20: Methode zur Erstellung der Bearbeitenseiten

Config

Das dritte Template generiert die Konfigurationsdatei *faces-config.xml*. In dieser Datei müssen alle Beans, welche vom CRUD-Framework generiert werden definiert werden. Hier werden auch globale und lokale Navigationsegeln definiert. In der Abbildung 4.10 ist sehr schön ersichtlich, dass von den Übersichtseiten auf eine Ansichtseite bzw. zu einer Bearbeitenseite verwiesen wird. Die Bearbeitenseiten referenzieren auf sich selbst, um mehrfach Änderungen zu unterstützen. Mit einer globalen Regel kann jederzeit von überall auf eine beliebige Übersichtsseiten navigiert werden.

Weiters wird in dieser Konfigurationsdatei der Hibernate Validator, der Enum Konverter und ein Ressource Bundle definiert.

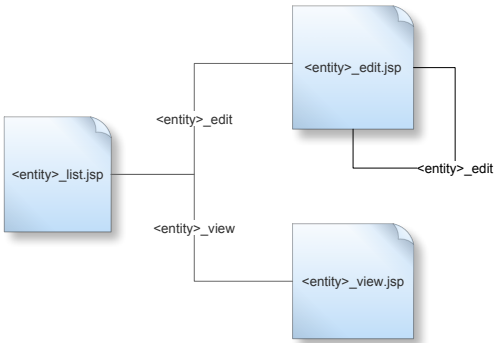

Abbildung 4.10: JSF Workflow

4.4.3 Persistence Cartridge

Die Persistence Cartridge unterteilt sich in die Generierung der Konfiguration und in die Erweiterung der Java Cartridge, welche in Kapitel 4.4.1 beschrieben wurden.

Config

Es werden zwei Konfigurationsdateien generiert. Einerseits die *hibernate.cfg.xml* Datei, in der alle Hibernate Mappings der Klassen und Enumerationen aus dem Modell definiert werden. Weiters ist ebenfalls die Datenbankkonfiguration hier definiert, welche mit der *ManiocCartridge.properties* Datei, die der Generator wie das eigentliche Modell als zusätzlichen Input verwendet, beschrieben wird. Die zweite Datei ist die *persistence.xml*, welche selbst die *hibernate.cfg.xml* Datei referenziert. Diese *persistence.xml* ist die eigentliche Konfigurationsdatei, welche von der JPA, die eine Schnittstelle für die Persistierung von Objekten in relationale Datenbanken bereitstellt, verwendet wird. In der aktuellen Version wird nur die Hibernate Persistence Implementierung unterstützt.

Beans

Das Beans Cartridge erweitert die Java Cartridge um die notwendigen Persistierungsannotationen. In der Zeile 1 des Listing 4.21 wird mittels dem *AROUND* Konstrukt die bereits im Java Cartridge implementierte Methode *attribute* (siehe Listing 4.13) überschrieben. Das *AROUND* Konstrukt benötigt das genaue Package und den Namen der Methode (*manioc::java::template:: Java::attribute*). Innerhalb der *AROUND* Methode wird dann mit Hilfe einer eigenen definierten Methode *attribute* im aktuellen Beans Cartridge der Code für die Persistierung implementiert. Zuletzt wird die überschriebene Methode mit dem Konstrukt *targetDef.proceed()* aufgerufen, womit letztendlich das Attribut mit den Annotationen wie *@OneToOne*, *@OneToMany*, *@ManyToOne* oder *@ManyToMany* generiert wird.

```
1  «AROUND manioc::java::template::Java::attribute FOR Attribute»
2    «("AROUND-PERSISTENCE attribute").info() -> null-»
3    «EXPAND attribute FOR this»
4    «targetDef.proceed()»
5  «ENDAROUND»
6
7  «DEFINE attribute FOR Attribute»
8    «IF identifier»
9    @Id
10   @GeneratedValue
11   «ENDIF-»
12   «LET (Role)getGlobalVar("role") AS role-»
13   «LET getOtherRole(role) AS roleOther-»
14
15   «IF isComplex() && role != null && roleOther != null »
16     «IF role.isSingle() && roleOther.isMulti()»
17       @OneToMany «EXPAND getFetchAndCascade FOR roleOther»
18     «ELSEIF role.isMulti() && roleOther.isMulti()»
```

```
19          @ManyToMany «EXPAND getFetchAndCascade FOR roleOther»
20          «IF !role.owner-»
21          @JoinTable(name = "«roleOther.name.toFirstLower()»_«role.name.
                toFirstLower()»")
22          «ENDIF-»
23        «ELSEIF role.isSingle() && roleOther.isSingle()»
24          @OneToOne «EXPAND getFetchAndCascade FOR roleOther»
25        «ELSEIF role.isMulti() && roleOther.isSingle()»
26          @ManyToOne «EXPAND getFetchAndCascade FOR roleOther»
27        «ENDIF»
28      «ELSE»
29        @Column(name = "«name»")
30      «ENDIF»
31
32      «ENDLET»
33      «ENDLET»
34    «ENDDEFINE»
```

Listing 4.21: Auszug aus dem Persistence Bean Template

4.4.4 Validierungs Cartridge

Das Validierungs Cartridge erweitert das im vorigen Kapitel beschriebene Persistierungscartridge um die Validierungsfunktionalität. Es generiert aus den im Modell definierten Einschränkungen die entsprechenden Hibernate Validierungsannotationen, wie z.B *@Size*, *@Range*, *@Past*, *@Future*, *@Digits*, ... Im Listing 4.22 ist wieder das *AROUND* Konstrukt zu sehen, welches diesmal die Methode *attribute* aus dem Persistierungscartridge überschreibt. Die Methode *getConstraints()* liefert dann die eigentliche Validierungsannotation (z.B. *@Length(min=3, max=30)*) zurück.

```
1   «AROUND manioc::persistence::template::Beans::attribute FOR Attribute»
2     «("AROUND-VALIDATION attribute").info() -> null-»
3     «getConstraints()»
4     «targetDef.proceed()»
5   «ENDAROUND»
```

Listing 4.22: Auszug aus dem Validierungs Template

Eine komplette Generierung des Attributes Strasse mit den Einschränkungen, dass es nicht leer, es nicht null, es mindestens 3 Zeichen enthalten muss und maximal 30 Zeichen enthalten darf, sieht wie in Listing 4.23 dargestellt aus.

```
1  @NotNull
2  @NotEmpty
3  @Length(min = 3, max = 30)
4  @Column(name = "street")
5  protected String street;
```

Listing 4.23: Generiertes Attribut mit Persistierungs- und Validierungsannotationen

KAPITEL 5

Evaluierung

5.1 Übersicht

In diesem Kapitel wird der aktuelle Ansatz mit den bereits existierenden Ansätzen Grails und Seam verglichen. Um fundierte Aussagen zwischen den einzelnen Ansätzen zu bekommen, wird dafür ein eigener Prototyp entworfen. Für jede Technologie werden die Bereiche Modell und Generierung verglichen. In der Analyse wird das Wachstum des Modells mit dem Wachstum der einzelnen Codeartefakte gegenüberstellt [33].

5.2 Prototyp

5.2.1 Beschreibung

Der Prototyp stellt ein einfaches Beispiel dar, welches in der realen Welt in ähnlicher Form sehr häufig vorkommt. Das Modell besteht aus zwei Objekten Person und Adresse, wobei eine Person mehrere Adressen enthalten kann. Eine Person besitzt einen Vor- und Nachname mit den Einschränkungen, dass diese Attribute nicht *null* sein dürfen und dass mindestens 3 und maximal 20 Zeichen erlaubt sind. Eine Adresse besteht aus einer Straße, einer Hausnummer, einer Türnummer, einer Stadt und einem Land, wobei die Straße vom Typ String ist und zwischen 3 und 50 Zeichen lang sein muss, die Haus- und Türnummer vom Typ Integer ist und zwischen 1 und 1000 sein muss, die Stadt vom Typ String ist und zwischen 3 und 50 Zeichen lang sein muss und das Land vom Typ String ist und welches ebenfalls zwischen 3 und 50 Zeichen lang sein muss. Es sollen bei beiden Objekten die *toString* Methoden angepasst werden. In Abbildung 5.1 ist das Domänenmodell des Prototypen abgebildet.

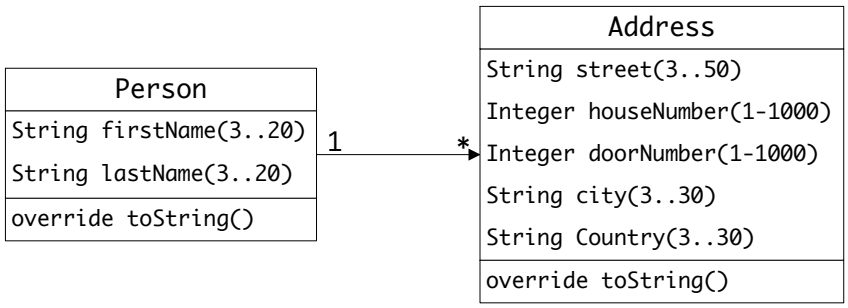

Abbildung 5.1: Domänenmodell des Prototypen

5.2.2 Modell des Prototypen

Mit dem zugrundeliegenden Metamodell, welches in Kapitel 4.2 beschrieben wurde, lässt sich der Prototyp gänzlich beschreiben. Zusätzlich zu den geforderten Attributen, wird für jedes Objekt ein Identifikator vom Typ *Long* hinzugefügt. Im Listing 5.1 ist die XML Definition des Prototypen abgebildet. In Zeile 3 wird das Objekt *Person* mit dem Element *type* definiert. Mit dem Attribut *xsi:type="manioc:Class"* wird festgelegt, dass es sich um eine Klasse handelt. Alternativ könnte hier mit *xsi:type="manioc:Enum"* ebenso eine Enumeration definiert werden. Die Attribute Id, Vorname und Nachname werden mit dem Element *attribute* definiert. Für den Vornamen und den Nachnamen werden zusätzlich die geforderten Einschränkungen definiert. Diese Einschränkungen werden mit dem Element *constraint* festgelegt (siehe Zeile 6-7 und 10-11). Neben dem Namen wird auch der Typ mit dem Attribut *simpleType="String"* bestimmt. In den Zeilen 14-36 wird das Objekt *Adresse* mit allen Attributen und Einschränkungen definiert.

Zuletzt muss noch die Beziehung zwischen den beiden Objekten festgelegt werden. Dies wird mit dem Element *association* bewerkstelligt. Hier wird mit dem Element *Role1* das erste Objekt *Person* und mit dem Element *Role2* das zweite Objekt *Adresse* der Beziehung definiert. Die Zuweisung der Objekte erfolgt mit dem Attribut *Class*. Da beide Objekte in der selben XML Datei erst definiert werden, muss hier der Verweis zu diesen Elementen angegeben werden. Diese Verweise werden mit *"//@<NAME>.<ZAHL>"*, wobei *NAME* der Name des Elementes ist (z.B. *Type*) und die *ZAHL* die absolute Position des Elements in der XML Datei ist.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <manioc:Model xmi:version="2.0" ...>
3    <type xsi:type="manioc:Class" name="Person">
4      <attribute name="id" identifier="true" simpleType="LONG"/>
5      <attribute name="firstName" simpleType="STRING">
6        <constraint xsi:type="manioc:Basic" notNull="true" notEmpty="true"/>
7        <constraint xsi:type="manioc:Length" min="3" max="20"/>
8      </attribute>
9      <attribute name="lastName" simpleType="STRING">
10       <constraint xsi:type="manioc:Basic" notNull="true" notEmpty="true"/>
11       <constraint xsi:type="manioc:Length" min="3" max="20"/>
12     </attribute>
13   </type>
14   <type xsi:type="manioc:Class" name="Address">
15     <attribute name="id" identifier="true" simpleType="LONG"/>
16     <attribute name="street" simpleType="STRING">
17       <constraint xsi:type="manioc:Basic" notNull="true" notEmpty="true"/>
18       <constraint xsi:type="manioc:Length" min="3" max="50"/>
19     </attribute>
20     <attribute name="houseNumber" simpleType="INTEGER">
21       <constraint xsi:type="manioc:Basic" notNull="true" notEmpty="true"/>
22       <constraint xsi:type="manioc:Range" min="1" max="1000"/>
23     </attribute>
24     <attribute name="doorNumber" simpleType="INTEGER">
25       <constraint xsi:type="manioc:Basic" notNull="true" notEmpty="true"/>
26       <constraint xsi:type="manioc:Range" min="1" max="1000"/>
27     </attribute>
```

```
28      <attribute name="city" simpleType="STRING">
29        <constraint xsi:type="manioc:Basic" notNull="true" notEmpty="true"/>
30        <constraint xsi:type="manioc:Length" min="3" max="50"/>
31      </attribute>
32      <attribute name="country" simpleType="STRING">
33        <constraint xsi:type="manioc:Basic" notNull="true" notEmpty="true"/>
34        <constraint xsi:type="manioc:Length" min="3" max="50"/>
35      </attribute>
36    </type>
37    <association name="has addresses">
38      <role1 name="Person" class="//@type.0" owner="true">
39        <multiplicity lowerBound="1" upperBound="1"/>
40      </role1>
41      <role2 name="Address" class="//@type.1">
42        <multiplicity lowerBound="1" upperBound="1"/>
43      </role2>
44    </association>
45  </manioc:Model>
```

Listing 5.1: XML Ansicht des Prototypen

Obwohl die Definition dieser XML Datei recht einfach ist, steigt der Aufwand mit mehreren Klassen doch erheblich. Mit Hilfe des EMF Plugins können diese XML Dateien direkt in Eclipse erstellt werden. Hier wird das Modell in einem hierarchischen Baum dargestellt. Als Wurzel dient das Element *Modell*, welches die Klassen Person und Adresse, sowie ihre Beziehung enthält. Die Attribute werden in den Klassen Definitionen als Kindelemente dargestellt. Die Einschränkungen werden wiederum als Kindelemente der Attribute dargestellt.

5.2.3 Generierter Code

Aus dem definierten Modell, lässt sich jetzt der eigentliche Prototyp generieren. Wie in Kapitel *Codegenerierung* 4.4 beschrieben werden unterschiedliche Module erstellt. Neben den POJO's, der JSF-Konfiguration wird vor allem die View erstellt. Diese unterteilt sich in 3 Ansichten.

Liste

Die erste Ansicht ist die Listendarstellung aller definierten Klassen. Der generierte Code enthält dafür ein spezielles JSF Element *h:dataTable*, mit dem alle Entitäten in einer Tabelle dargestellt werden. In Listing 5.2.3 ist zu sehen, wie das Attribut *Strasse* der Klasse *Adresse* mittels JSF definiert wird. Innerhalb des Tags *h:dataTable* können beliebig viele Tags *h:columns* definiert werden, mit denen die Attribute dargestellt werden. Mit dem Tag *f:facet* und dem Attribut *name="header"* wird ein Spaltenkopf *Strasse* definiert. Die eigentliche iterierte Ausgabe der Strasse erfolgt mit dem Element *h:outputText* und der EL *#{a.street}*.

```
1  ...
2  <h:dataTable id="dtAddressList" value="#{addressList.resultList}" var="a"
       border="0">
3    ...
4    <h:column>
5      <f:facet name="header"><h:outputText value="#{msg.street}"/></f:facet>
6      <h:outputText value="#{a.street}" />
7    </h:column>
8    ...
9  </h:dataTable>
10 ...
```

Neben dem Attribut *Strasse* werden alle anderen Attribute ebenfalls ausgegeben. Es werden für jede Zeile drei Buttons zum *Bearbeiten*, *Anzeigen* und *Löschen* erstellt. Am Ende der Tabelle wird die Anzahl der Entitäten und ein Button *Neu* angezeigt (Siehe Abbildung 5.2).

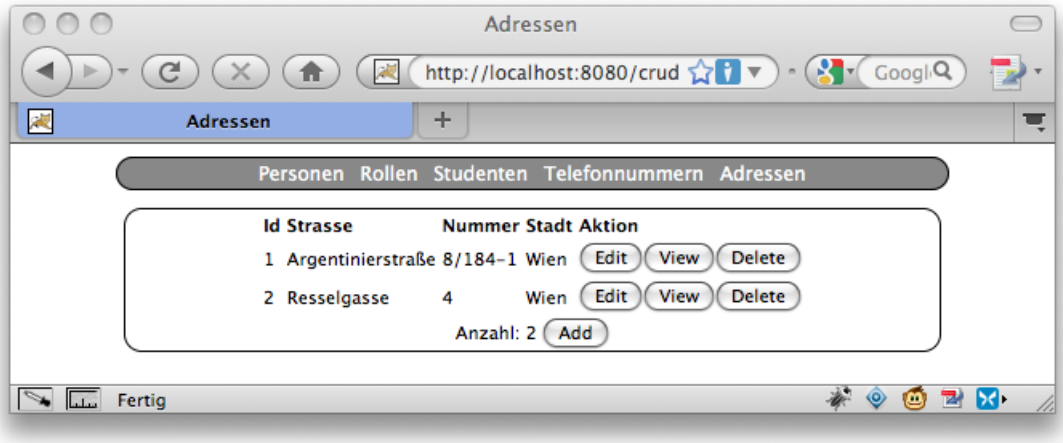

Abbildung 5.2: Übersichtseite von Adressen

Ansicht/Bearbeiten

Neben den Übersichtseiten wird jeweils eine Ansicht für den Anzeige- und eine für den Bearbeiten-Modus generiert. Diese zwei Ansichten unterscheiden sich nur durch unterschiedliche JSF-Elemente für die Klassenattribute. In der Anzeige wird für die Ausgabe des Attributs ein JSF Element *h:outputText* dargestellt. Im Bearbeitenmodus wird stattdessen das JSF Element *h:inputText* verwendet. Mit diesem Element besteht letztendlich die Möglichkeit Benutzereingaben an den Server zu schicken und somit Attribute einer Klasse zu aktualisieren. In Listing 5.2.3 wird das Attribut *Strasse* dargestellt. Es wird zuerst ein Element *h:outputLabel*, welches den Wert *Strasse* enthält ausgegeben.

Anschließend wird ein Eingabefeld dargestellt, welches ein zusätzliches Kindelement *val:validate* enthält, mit dem die Eingabevalidierung festgelegt wird.

```
1  <div class="form_row">
2    <h:outputLabel id="lStreet" for="tbStreet" value="#{msg.street}"
         styleClass="required" />
3    <h:inputText   id="tbStreet" value="#{address.instance.street}" required
         ="true" styleClass="input_field" >
4      <val:validate bean="#{address.instance}" property="street" bundle="
           messages"/>
5    </h:inputText>
6  </div>
```

In der Abbildung 5.3 ist zu sehen, wie eine fehlerhafte Eingabe der Entität Adresse aussehen könnte. Bei fehlerhafter Validierung werden alle Fehlermeldungen in einem Fehlerfeld am Anfang der Seite dargestellt. Bei der Eingabe „T“ im Feld Strasse, werden 3 Validierungsfehler geworfen. *Nummer ist ein Pflichtfeld*, *Strasse muss zwischen 3 und 30 Zeichen lang sein* und *Stadt ist ein Pflichtfeld*.

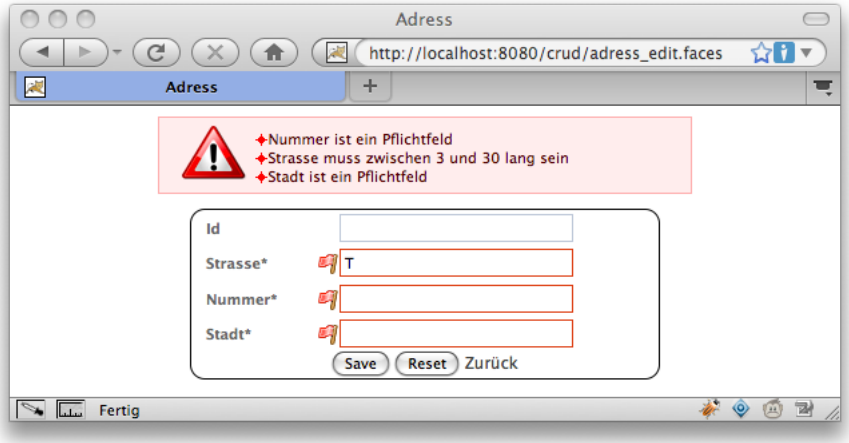

Abbildung 5.3: Web Interface: Validierung von Adressen

5.3 Quantitative Evaluierung

5.3.1 Grails

Grails unterscheidet sich zu den anderen Ansätzen in erster Linie dadurch nicht auf JSF zu setzen. Stattdessen setzt Grails auf GSP, die sehr stark JSP ähnelt und ebenfalls eine HTML basierte Webprogrammiersprache ist.

Modell

Das Modell in Grails wird direkt mit einer Groovy Klasse definiert (siehe Listing 5.3.1). Der Vorteil bei Groovy ist, dass nur die Attribute und deren Einschränkungen definiert werden müssen. Es ist nicht wie in Java notwendig, für alle Attribute auch die Getter und Setter zu implementieren.

```
1  class Person {
2    String    firstName
3    String    lastName
4    static    hasMany  =  [addresses:Address]
5
6    String toString() { "${firstName} ${lastName}" }
7
8    static constraints = {
9      firstName(nullable:false, minSize:3, maxSize:20)
10     lastName(nullable:false, minSize:3, maxSize:20)
11   }
12 }
```

Sehr positiv an diesem Ansatz ist, dass die Definition einer Person mit ihren Validierungseinschränkungen lediglich nur 12 Zeilen benötigt. Nachteil es existiert kein grafischer Editor um das Domänenmodell zu erstellen.

Generierung

Wie schon erwähnt, liefert Grails einen Kommandozeileninterpreter. Nach der Implementierung von den Klassen *Person* und *Address*, kann mittels *grails run-app* die Webanwendung auch schon gestartet werden. Grails generiert jedes mal alle Views und Controller. Es besteht auch die Möglichkeit mit *generate-controller*, *generate-i18n-messages* und *generate-views* aus den bestehenden Entitäten den Controller, die internationalisierten Texte und die Views zu generieren und diesen generierten Code im Projekt sichtbar und somit auch änderbar zu machen. Ein kompletter Start mit automatischer Generierung, benötigt auf dem Referenzsystem ca. 13 Sekunden. Hier ist allerdings der Start vom internen Applikationsserver Jetty schon mit berücksichtigt. Die Zeit, die alleine für die Generierung benötigt wird, konnte nicht exakt gemessen werden.
Es wurden insgesamt 256 Zeilen für die Viewseiten der Entität *Person* generiert. Für die Adresse wurden 322 Zeilen generiert.

Analyse

In der Tabelle 5.1 ist zu sehen, wie sich die Anzahl der Attribute in einer Entität auf die Anzahl der generierten Zeilen auswirkt. Für jedes Attribut und Einschränkung erhöht sich die Entität um genau 2 Zeilen. Die generierten GSP Seiten (*create.gsp*, *edit.gsp*, *list.gsp* und *show.gsp*) werden insgesamt um 31 Zeilen mehr. Die Anzahl 2+1 Attribute bedeutet, dass zwei normale und ein Abhängigkeitsattribut existieren.

Tabelle 5.1: Grails: Zusammenhang Anzahl der Attribute mit generierten Codezeilen

	Modell	generiert	Δ	
Anzahl Attr.	Entität	GSP	Δ Entität	Δ GSP
0	2	167	-	-
1	9	198	7	31
2	11	229	3	31
2+1	12	256	1	27
3+1	14	287	2	31
4+1	16	318	2	31
5+1	18	349	2	31

5.3.2 Seam

Einer der sofort auffallenden Unterschiede zu Grails ist, dass Seam nur 3 JSF-Seiten pro Entität generiert. Seam kombiniert die Erstellen- und Bearbeiten-Seite und erspart sich somit eine komplette Seite. Allerdings generiert Seam wiederum XML Dateien für die Navigation. Seam generiert somit pro Entität 6 anstatt wie Grails nur 4 Dateien. Die Generierung selbst benötigt ca. 3-4 Sekunden.

Modell

Das zu definierende Modell wird ebenso wie in Grails mit einer Klasse definiert. Der große Unterschied zu Grails ist, dass die Klasse ganze 80 Zeilen bei nur drei Attributen benötigt. Dies sind gegenüber Grails ca. 666% mehr Zeilen! Diese enorme Zahl kommt durch die ganzen JPA Imports, den Annotationen und den Gettern und Settern zustande. Alleine die Getter und Setter benötigen schon 40 Zeilen. Obwohl diese zwar auch z.B. mit Eclipse generiert werden können, können sich hier Fehler einschleichen und vor allem vergrößert sich das Modell damit nur unnötig.

Generierung

Wie auch Grails bietet Seam einen Kommandozeileninterpreter, mit dem diverse Codestücke generiert werden können. Mit *seam generate-ui* werden alle View- und Controller-Artefakte generiert. Seam generiert für eine Entität *Person* 3 *XHTML* Dateien für die Ansicht, 3 *XML* Dateien für die Navigation und 2 Java Dateien, welche die Logik für die Datenbankpersistierung beinhalten.

Analyse

In der Tabelle 5.2 ist zu sehen, wie sich die Anzahl der Attribute einer Entität auf die Anzahl der generierten Zeilen in View und Logik auswirkt. Obwohl die Entität selbst

nicht generiert wird ist sie der Vollständigkeit halber ebenfalls in der Tabelle. Dadurch sieht man auch sehr schön, dass die Entität von 0 auf 1 Attribut um 17 Zeilen wächst und anschließend nur mehr um 13 Zeilen pro Attribut. Die Begründung lässt sich schlicht und einfach mit der Generierung der Imports erklären. Beim ersten Attribut müssen neben Gettern, Settern und Annotation ebenso die benötigten Imports mitgeneriert werden. Es kann ebenso ein weiterer Anstieg bei einem folgenden Attribut erfolgen, wenn für dieses eine Einschränkung definiert wurde, welche eine neue Annotation benötigt und somit auch einen neuen Import.
In den JSF-Seiten werden insgesamt pro Attribut zusätzlich 32 Zeilen generiert mit der Ausnahme wenn das zu generierende Attribut eine Beziehung zu einer anderen Entität darstellt. In diesem Fall wachsen die JSF-Seiten um 121 Zeilen.
Die Logik selbst ändert sich nur minimal. Es wird lediglich ein JPA Attribut erweitert, welches die Einschränkungen der Suche steuert.

Tabelle 5.2: Seam: Zusammenhang Anzahl der Attribute mit generierten Codezeilen

	Modell	generiert			Δ			
Attr.	Entität	JSF	XML	Logik	Δ Entität	Δ JSF	Δ XML	Δ Logik
0	38	224	60	68	-	-	-	-
1	55	256	61	68	17	32	1	0
2	68	288	62	70	13	32	1	2
2+1	81	409	62	77	13	121	0	7
3+1	94	441	63	78	13	32	1	1
4+1	107	473	64	79	13	32	1	1
5+1	120	505	65	80	13	32	1	1

5.3.3 CRUD

Der große Unterschied zwischen der Generierung mit GRAILS und SEAM liegt in erster Linie am verwendeten Modell. In Seam und Grails werden jeweils die Entitäten selbst als Modell verwendet. Im eigenen Ansatz wird das Modell durch eine XML Datei beschrieben, welches wiederum durch ein Metamodell beschrieben wird.

Modell

Wie bereits beschrieben, wird eine XML Datei für die Beschreibung verwendet.

Generierung

Für die Generierung vom Code wird oAW verwendet. Hierzu muss auf die *Generator.oaw* Datei mit der rechten Maustaste geklickt werden und *Run As–>oAW Workflow*

ausgewählt werden. Die Ausführungszeit liegt in etwa bei 3-4 Sekunden. In der aktuellen Version wird die Generierung nur innerhalb von Eclipse unterstützt. Es kann aber jederzeit ein ANT Task geschrieben werden, da oAW ANT unterstützt.

Analyse

In Tabelle 5.3 ist die Anzahl der generierten Zeilen der Entität, der View und der Logik in Abhängigkeit zu den Attributen zu sehen. Auffällig ist, dass sich die Controllerklassen (einzelne Entität und Listen) als einzige nicht von den Attributen abhängig sind. Im Unterschied zu Seam wird in der aktuellen CRUD-Version keine Suche nach Attributen unterstützt. Aus diesem Grund wird hier auch kein Code mitgeneriert.

Es kann aber ein anderer Code durch das CRUD-Framework generiert werden. Und zwar dann, wenn ein Enumerationattribut in der Entität existiert. Für diesen Fall wird eine zusätzliche Methode in der Controllerklasse *<ENTITY>Home.java* generiert, welche eine Liste aller möglichen Enumerationen zurückliefert.

Tabelle 5.3: CRUD: Zusammenhang Anzahl der Attribute mit generierten Codezeilen

	Modell	generiert			Δ			
Attr.	XML	Entität	JSF	Logik	Δ XML	Δ Entität	Δ JSF	Δ Logik
0	6	37	129	81	-	-	-	-
1	10	54	143	81	4	17	14	0
2	14	67	157	81	4	13	14	0
2+1	25	79	157	81	11	12	0	0
3+1	29	92	171	81	4	13	14	0
4+1	33	105	185	81	4	13	14	0
5+1	37	118	199	81	4	13	14	0

5.3.4 Ergebnis - Interpretation

Vergleich View

In Abbildung 5.4 ist die Abhängigkeit der Attribute zu den generierten Zeilen in der View der einzelnen Technologien zu sehen. Zwei wesentliche Merkmale lassen sich aus diesem Diagramm herauslesen.

Erstens ist die Anzahl der Viewzeilen des CRUD-Ansatzes doch deutlich geringer als in Grails und Seam. Dies liegt darin, dass einerseits der CRUD-Ansatz keine AJAX Funktionalität unterstützt und andererseits Seam für jedes JSF-Element die Attribute immer in einer neuen Zeile generiert. Würde Grails nicht zwei unterschiedliche Seiten für die Erstellung und Bearbeitung generieren, hätte es wahrscheinlich den besten Faktor.

Zweitens ist in Seam ein sehr hoher Anstieg von 2 normalen Attributen auf 2 normale und 1 Beziehungsattribut zu sehen. Dies lässt sich dadurch erklären, da Seam bei einer 1 zu n Relation, die referenzierte Entität in einer Liste mit ihren Attributen darstellt, welches natürlich wiederum mit JSF-Code beschrieben werden muss. Grails verwendet stattdessen nur die *toString* Methode, wodurch hier auch kein Anstieg zu sehen ist.

Im CRUD-Ansatz ist hier kein Anstieg zu sehen, da die Codegenerierung der View noch keine Beziehungen zu anderen Attributen unterstützt.

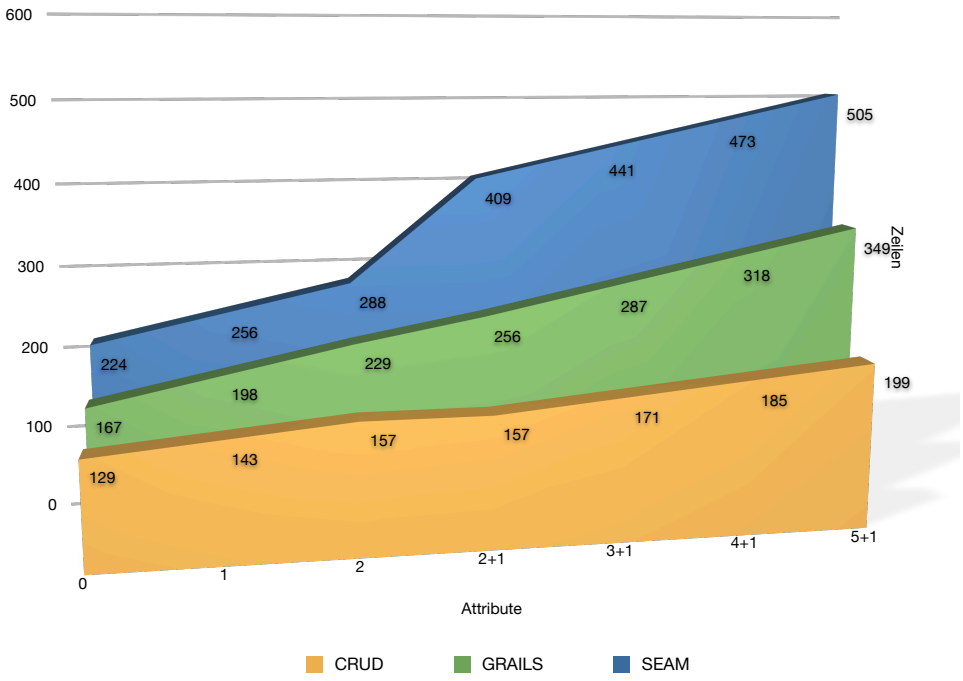

Abbildung 5.4: Anzahl der generierten Viewzeilen

Vergleich Controller

In Abbildung 5.5 ist die Abhängigkeit der Attributen zu den generierten Zeilen der Logik der einzelnen Technologien zu sehen. In diesem Diagramm gibt es insgesamt 4 Kurven, da in Grails die Möglichkeit der Verwendung des *Scaffolding* besteht. Der Idealfall sieht die Verwendung des automatischen Scaffolding vor, dh. es werden nur die Grundfunktionalitäten wie Anlegen, Ändern, Lesen oder Löschen benötigt. In diesem Fall muss kein zusätzlicher Code in Abhängigkeit zu den Attributen erstellt werden. Es

würden lediglich 4 Zeilen per Entität hinzukommen. Ein Controllerobjekt pro Entität mit der Information, dass Scaffolding verwendet wird. Falls dies nicht ausreicht, bietet Grails die Möglichkeit diesen Code im Controller generieren zu lassen. Und genau dadurch kommt die erheblich größere Zeilenanzahl im Vergleich zu den anderen Technologien zustande.
Die generierten Zeilen zwischen dem CRUD-Ansatz und Seam ist fast identisch. Dies ist auch nicht verwunderlich, da der CRUD-Ansatz dem Seam Ansatz sehr ähnlich ist. Es wird pro Entität eine generische Klasse generiert, welche von einer Superklasse erbt, in der die notwendigen Funktionen wie Anlegen, Lesen, Bearbeiten und Löschen implementiert sind.

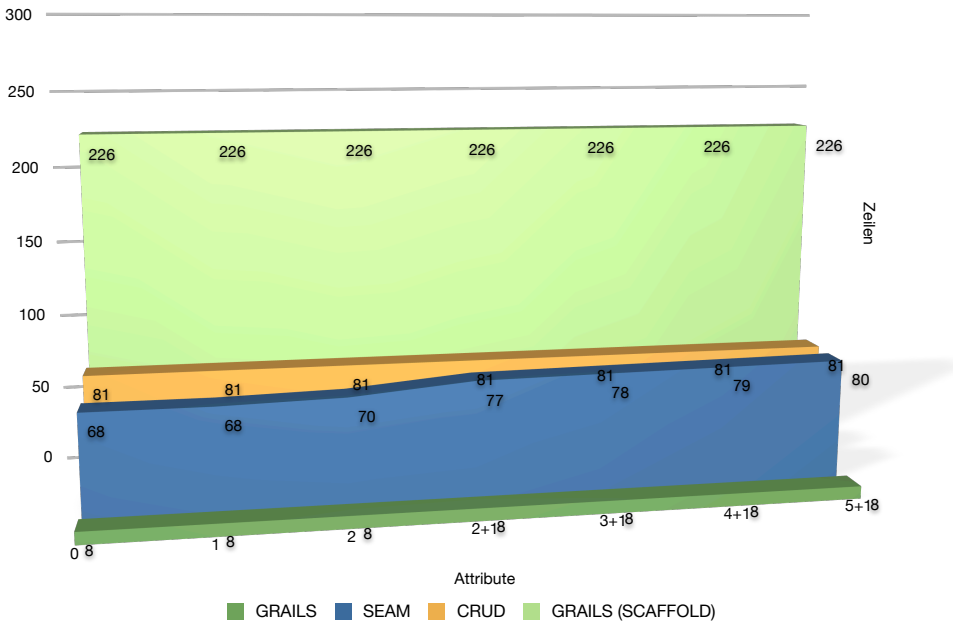

Abbildung 5.5: Anzahl der generierten Controllerzeilen

Vergleich Entitäten

In Tabelle 5.4 ist sehr schön zu sehen, dass vor allem Grails sehr kompakte Entitäten besitzt. Da Seam und CRUD die gleiche Technologie nämlich Java mit JPA und Hibernate Validierungsannotationen verwenden, ist es auch nicht verwunderlich, dass die

Anzahl der Zeilen fast genau übereinstimmen. Allerdings bei genauerer Betrachtung ist doch ein kleiner Unterschied festzustellen. Mit Seam hat man keine Möglichkeit, die *toString* Methode zu überschreiben. Warum sind jetzt aber die Klassen fast gleich groß? Der Grund liegt darin, dass Seam ein Versionierungsattribut mitgeneriert, welches für optimistisches Locking benötigt wird. Dies ist in CRUD nicht vorgesehen. Beim optimistischen Locking wird der Datensatz während der Bearbeitung für andere Zugriffe nicht blockiert, aber beim späteren Schreibversuch wird überprüft, ob der Datensatz noch aktuell ist und somit geändert werden darf.

Tabelle 5.4: Übersicht Entitäten

		Technologie		
		Grails	Seam	CRUD
Modell Person	Source	Groovy Klasse	Java Klasse	XML(Ecore)
	Zeilen	12	80	79
	Zeichen	287	148	158
Modell Adresse	Source	Groovy Klasse	Java Klasse	XML(Ecore)
	Zeilen	16	118	119
	Zeichen	357	390	250

Beziehungen

Eine wichtige Betrachtung bei den Entitäten sind ihre Beziehungen zu anderen Entitäten. Bei normalen Attributen muss z.B. nur ein einfaches *INPUT* Feld für eine Texteingabe generiert werden. Bei Beziehungen zu anderen Entitäten treten bei der Generierung viel komplexere Darstellungen auf. Der einfachste Fall ist noch eine 1:1 Beziehung. Eine Person hat genau eine Adresse. Dh. in der View wird dafür nur eine einfache *Dropdown Box* generiert, mit der eine Adresse ausgewählt wird. Probleme können aber auch schon hier auftreten. z.B. welchen Text möchte man in der *Dropdown Box* darstellen. Die Strasse, die Strasse mit Hausnummer oder die Strasse mit einer Id? Es gibt viele Möglichkeiten. Die wahrscheinlich beste Variante ist wie auch im Vergleichsbeispiel gefordert, die *toString* Methode zu überschreiben. Es muss allerdings sichergestellt werden, dass bei einer neuerlichen Generierung diese Methode nicht wieder überschrieben wird. Für die 1:n und m:n Relationen wird das ganze sogar noch viel komplexer. Aus diesem Grund unterstützen die Frameworks nie alle möglichen Relationen (siehe Tabelle 5.5).

Wachstum Modell

In Tabelle 5.6 ist leicht zu erkennen, dass Grails das geringste Wachstum in Abhängigkeit zu den Attributen aufweist. Es reichen für ein Attribut bereits 2 Zeilen, eine

Tabelle 5.5: Übersicht Relationen

	Relation			
	1:1	1:n	n:1	m:n
Grails	✔	✔	✔	✘
Seam	✔	✔	✔	✘
CRUD	✔	✘	✘	✘

für den Typ und Name und eine für die Einschränkung. In der Mitte befindet sich der aktuelle CRUD-Ansatz. Hierfür werden schon 4 Zeilen pro Attribut benötigt. Gegenüber Grails wird das Modell im CRUD-Ansatz mittels XML definiert, wodurch mit den schließenden Elementen </... > gleich eine Zeile mehr generiert wird. Weiters wird jede Einschränkung pro Attribut in eine neue Zeile geschrieben, im Unterschied dazu generiert Grails dafür ebenfalls nur eine Zeile. Am schlechtesten schneidet hier Seam mit 13 Zeilen pro Attribut ab. Der Grund dafür ist, dass normale Java POJO Objekte mit Annotationen als Modell verwendet werden. Laut Konvention existiert somit für jedes Attribut ein Getter und Setter, womit die Zeilen leider durch redundanten Code erheblich erhöht werden.

Tabelle 5.6: Wachstum Modell

	Grails	Seam	CRUD
Zeilen pro Attribute	2	13	4

Laufzeitverhalten

Die Performance ist bei allen 3 Technologien fast identisch, bzw. schwer zu vergleichen. In Grails wird die gesamte Generierung über alle Entitäten nur mit dem Task *run-app*, welcher gleich den Webserver mitstartet ausgeführt. Dieser benötigt 13 Sekunden und ist mit den anderen Zeiten nicht direkt vergleichbar. In Tabelle 5.7 sind die Ausführungszeiten der Generierung der einzelnen Technologien aufgelistet.

Tabelle 5.7: Laufzeitverhalten Generierung

	Grails	Seam	CRUD
Ausführungszeiten Generierung (Sekunden)	13 (mit Server Start)	3-4	3-4

Gesamtübersicht

In Tabelle 5.8 ist nochmals eine Gesamtübersicht der Technologien aufgelistet.

[1] mit Webserver Start

Tabelle 5.8: Vergleich Technologien

	Grails	Seam	CRUD
Programmiersprache	Grails & Java	Java	Java
Modell	Groovy	Java	Ecore
AJAX	✔	✔	✘
Generierung CLI	✔	✔	✘
Generierung Eclipse	✘	✔	✔
Kompilierung/Deployment CLI(ANT)	✔	✔	✔
Kompilierung/Deployment Eclipse	✘	✔	✘
Internationalisierung	✔	✔	✔
Ausführungszeiten Generierung	13 Sekunden[1]	3-4 Sekunden	3-4 Sekunden

Vor/Nachteile

Zuerst sei einmal zu erwähnen, dass Grails und Seam zwei komplexe Webframeworks sind, die zusätzlich MDD Ansätze verfolgen. Im Gegensatz dazu ist das CRUD-Framework als reiner MDD Ansatz zu sehen, das Technologien wie JSF und Hibernate verwendet. Es geht bei diesem Ansatz nicht darum, Webseiten recht einfach und schnell zu entwickeln, sondern ein Modell zu definieren und aus diesem die Webseiten zu generieren. Aus diesem Grund existiert im aktuellen CRUD-Ansatz ein eigenes Metamodell um Modelle nach bestimmten Vorgaben zu modellieren. Der große Vorteil an diesem Ansatz liegt darin, dass die bereits vorhandene EMF Technologie verwendet wird. Dh. es besteht die Möglichkeit das Modell, welches als XML definiert wird durch GMF zu erstellen. Somit kann dieser Ansatz sehr leicht durch einen grafischen Editor erweitert werden.

Ein weiterer Vorteil am aktuellen CRUD-Ansatz liegt darin, dass für die Codegenerierung ebenfalls die standardisierte Technologie oAW verwendet wird. Bei der Entwicklung des Codegenerators wurde sehr stark darauf geachtet eine möglichst hohe Wiederverwendbarkeit zu erzielen, weshalb mehrere Cartriges entworfen wurden. Es ist somit ein leichtes diese Cartriges nach belieben um weitere Funktionalität zu erweitern. Seam und Grails setzen hier auf keine standardisierte Technologien, sondern verwenden einfache Templates zur Generierung von View und Controllern.

KAPITEL 6

Zusammenfassung

6.1 Ergebnis

Das Problem war, dass es in vielen Webseiten eine sogenannte Datenverwaltungsseite gibt, die immer wieder neu entwickelt wird, obwohl es jedesmal dem gleichen Schema folgt. All dieses widerspricht jeglichen Aspekten der Wiederverwendung, Qualität und Sicherheit.

Aus diesem Grund wurde ein Code-Generator geschrieben, der diese Verwaltungsseite automatisch erstellt. Dazu wurde ein Metamodell entworfen, mit welchem die Domäne selbst und eine zusätzliche Validierung, welche vor allem wegen der Sicherheit und der Datenqualität unerläßlich war, abgebildet werden können. Der generierte Code komplementiert mit einem fertigen Grundgerüst, welches wiederum auf JSF und Hibernate aufbaut eine fertige Web Anwendung.

Ein großer Vorteil am aktuellen CRUD Ansatz liegt sicherlich in der Verwendung des Open Source Generators *openArchitectureWare*. Es ist jederzeit möglich View-Seiten oder Java Klassen mittels *Code Protection* um diverse Funktionalität zu erweitern. Dies ist ein großer Vorteil gegenüber Ansätze, die sich speziell auf die Erstellung von solcher Datenverwaltungsseite spezialisiert haben.

6.2 Zukünftige Arbeit

Der aktuelle Ansatz zeigt, welches Potential in modell-getriebenen Software Anwendungen steckt. Der einfache Prototyp zeigt schon sehr deutlich wie viel Code erzeugt werden muss um eine fertige Web Anwendung zu erstellen. Der aktuelle Ansatz, mit dem jede mögliche Art von Beziehungen, die auch in Java möglich sind abgebildet werden kann zeigt schon die Mächtigkeit des bisherigen Metamodells. Ebenso zeigt der generierte Code ebenfalls sehr deutlich den aktuellen Stand dieses Ansatzes.

6.2.1 Metamodell

Zukünftig sollte das Metamodell noch bezüglich Interfaces erweitert werden. Es könnte auch ein Rollenkonzept hinzugefügt werden. Statt nur einen einzigen Admin Benutzer zu unterstützen macht es durch aus Sinn hier mehrere Rollen einzuführen.

Es könnte zusätzlich auch eine Art Navigationsmodell entworfen werden, mit dem es möglich ist gezielte Viewseiten zu definieren. Entweder um die statische Navigation von CRUD-Webanwendungen zu erweitern oder um ganz in die Richtung von dynamischer Webentwicklung zu gehen. [35]

6.2.2 Generator

Im Generator findet sich ein viel höheres Potential für eine Weiterentwicklung. Der aktuelle Stand des Generators beinhaltet gerade einmal die Grundfunktionalität um CRUD-Webanwendungen zu erstellen. Es wird zum Beispiel nur eine 1:1 Relation unterstützt. Die anderen Assoziationen wie 1:n oder m:n müssen erst implementiert werden. Weiters könnte man die derzeit aktuellen und weitverbreiteten JavaScript Bibliotheken wie zum Beispiel jQuery [31], Dojo [20] oder Prototype [53] unterstützen. Ebenso könnte man die JSF Version 2.0 unterstützen, mit der dann auch AJAX möglich wäre. Oder man entwirft einen Generator für ganz anderen Technologie wie zum Beispiel Microsoft .Net [38] oder PHP. Wobei allerdings für eine neue Technologie der Generator komplett neu geschrieben werden müsste. Dies würde sicherlich zu einem deutlichen Mehraufwand gegenüber einer manuell geschrieben Anwendung führen, hätte aber alle Vorteile dieses Ansatzes.

6.2.3 Graphischer Editor

Zuletzt wäre es wünschenswert das Modell nicht im EMF Editor erstellen zu müssen, sondern stattdessen einen graphischen Editor zu unterstützen. Mittels GMF wäre dies jedenfalls zu realisieren, hätte aber den vorgesehenen Aufwand dieser Arbeit überstiegen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingsverzeichnis

Literaturverzeichnis

[1] Testng. http://testng.org/.

[2] Webml. http://www.webml.org/.

[3] Webml elements. http://www.webml.org/webml/upload/ent17/1/webml_elements.pdf/.

[4] Jsr 252: Javaserver faces 1.2. http://jcp.org/en/jsr/detail?id=303/, 2006.

[5] Jsr 303: Bean validation. http://jcp.org/en/jsr/detail?id=303/, 2009.

[6] Jsr 314: Javaserver faces 2.0. http://jcp.org/en/jsr/detail?id=314/, 2009.

[7] Joey Gibson Andy Wu Alan Williamson, Kirk Pepperdine. *Ant - Developer's Handbook*. Sams, 2002.

[8] Allan J. Albrecht. *Measuring application development productivity. In Tutorial - Programming Productivity: Issues for the Eighties*. Sams, 1986.

[9] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, 2 edition, 2008.

[10] Gerti Kappel und Gustavo Rossi Boualem Benatallah, Fabio Casati. *Web Engineering, 10th International Conference, ICWE 2010, Vienna Austria*. Springer, 1 edition, 2010.

[11] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Www9 / computer networks; web modeling language (webml): a modeling language for designing web sites. 33(1-6):137–157, 2000.

[12] Ed Burns Chris Schalk. *JavaServer Faces: The Complete Reference*. Mcgraw-Hill Professional, 1 edition, 2006.

[13] Alexander Knapp Christian Kroiss, Nora Koch. Uwe4jsf: A model-driven generation approach for web applications. In Oscar Díaz Martin Gaedke, Michael Grossniklaus, editor, *Web Engineering, 9th International Conference, ICWE 2009*, volume 5648 of *Lecture Notes in Computer Science*, pages 493–496. Springer Berlin / Heidelberg, 2009.

[14] Damiano Distante, Damiano Distante, Paola Pedone, Gustavo Rossi, and Gerardo Canfora. Model-driven development of web applications with uwa, mvc and javaserver faces.

[15] Neil Griffin Ed Burns. *JavaServer Faces 2.0. The Complete Reference*. Mcgraw-Hill Professional, 1 edition, 2010.

[16] The Apache Software Foundation. Ant. `http://ant.apache.org/`.

[17] The Apache Software Foundation. Apache myfaces apache myfaces extensions validator. `http://myfaces.apache.org/extensions/validator/index.html/`.

[18] The Apache Software Foundation. Apache myfaces tomahawk. `http://myfaces.apache.org/tomahawk/index.html/`.

[19] The Apache Software Foundation. The jakarta site - apache tomcat. `http://tomcat.apache.org/`.

[20] The Dojo Foundation. Dojo. `http://http://dojotoolkit.org/`.

[21] The Eclipse Foundation. Eclipse modeling framework. `http://www.eclipse.org/modeling/emf/`.

[22] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[23] David Heinemeier Hansson. Ruby on rails. `http://rubyonrails.org/`.

[24] JBoss (Red Hat). Hibernate. `http://www.jboss.com/products/hibernate`.

[25] JBoss (Red Hat). Hibernate validator. `http://docs.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/4.3/pdf/Hibernate_Validator_Reference_Guide/JBoss_Enterprise_Application_Platform-4.3-Hibernate_Validator_Reference_Guide-en-US.pdf/`.

[26] JBoss (Red Hat). Richfaces. `http://www.jboss.org/richfaces/`.

[27] JBoss (Red Hat). Seam. http://www.jboss.com/products/seam.

[28] ICESOFT Technologies Inc. Icefaces. http://www.icefaces.org/.

[29] Sun Microsystems Inc. Java blueprints, model-view-controller. http://java.sun.com/blueprints/patterns/MVC-detailed.html/.

[30] Ila Neustadt Jim Arlow. *Designing Data-Intensive Web Applications*. Publisher, 2002.

[31] The jQuery Project. jquery. http://jquery.com/.

[32] Christian Kroiss. Modellbasierte generierung von web-anwendungen mit uwe (uml-based web engineering). Master's thesis, Ludwig-Maximilians Universität München, 2008.

[33] Christian F. J. Lange. Model size matters. In *MoDELS Workshops*, pages 211–216, 2006.

[34] Christian F.J. Lange. Model size matters. Technical report, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, The Netherlands, 2006.

[35] Nora Koch Maria José Escalona. Metamodeling the requirements of web systems. In Vitor Pedrosa Joaquim Filipe, José Cordeiro, editor, *Web Information Systems and Technologies*, volume 1 of *Lecture Notes in Business Information Processing*, pages 267–280. Springer Berlin / Heidelberg, 2007.

[36] Nora Koch Marianne Bush. Magicuwe - a case tool plugin for modeling web applications. In Oscar Díaz Martin Gaedke, Michael Grossniklaus, editor, *Web Engineering, 9th International Conference, ICWE 2009*, volume 5648 of *Lecture Notes in Computer Science*, pages 505–508. Springer Berlin / Heidelberg, 2009.

[37] Gerald Müllan Martin Marinschek, Michael Kurz. *JavaServer Faces 2.0: Grundlagen und erweiterte Konzepte*. dpunkt Verlag, 2 edition, 2009.

[38] Microsoft. Microsoft .net. http://www.microsoft.com/net/.

[39] Hanspeter Mössenböck. *Objektorientierte Programmierung*. Springer-Verlag, 1993.

[40] Inc No Magic. Magicdraw. http://www.magicdraw.com/.

[41] Gefei Zhang Tatiana Morozova Nora Koch, Matthias Pigerl. Patterns for the model-based development of rias. In Oscar Díaz Martin Gaedke, Michael Grossniklaus, editor, *Web Engineering, 9th International Conference, ICWE 2009*, volume 5648 of *Lecture Notes in Computer Science*, pages 283–291. Springer Berlin / Heidelberg, 2009.

[42] Ernst Oberortner. Generating web applications with abstract pageflow models. Master's thesis, Vienna University of Technology, 2007.

[43] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. Domain-specific languages for service-oriented architectures: An explorative study. In Petri Mähönen, Klaus Pohl, and Thierry Priol, editors, *Towards a Service-Based Internet*, volume 5377 of *Lecture Notes in Computer Science*, pages 159–170. Springer Berlin / Heidelberg, 2008.

[44] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. Tailoring a model-driven quality-of-service dsl for various stakeholders. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, MISE '09, pages 20–25, Washington, DC, USA, 2009. IEEE Computer Society.

[45] Object Management Group (OMG). Model driven architecture. `http://www.omg.org/mda/`.

[46] Object Management Group (OMG). Unified modeling language, uml 2.0 superstructure specification. `http://www.omg.org/uml/`.

[47] openArchitectureWare. openarchitectureware. `http://www.openarchitectureware.org/`.

[48] OpenSymphony. Sitemesh. `http://www.opensymphony.com/sitemesh/`.

[49] Oracle. Javaserver faces technology. `http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html/`.

[50] C. Cachero S. Meliá. An mda approach for the development of web applications. In *Proceedings of of 4th ICWE 04*, LNCS 3140, pages 300–305, 2004.

[51] SpringSource. Spring. `http://www.springsource.org/`.

[52] Struts. The apache software foundation. `http://struts.apache.org`.

[53] Prototype Core Team. Prototype. `http://www.prototypejs.org/`.

[54] Prime Technology. Primefaces. `http://www.primefaces.org/`.

[55] Sven Efftinge Arno Haase Tomas Stahl, Markus Völter. *Modellgetriebene Softwareentwicklung, 2. Auflage, Techniken, Engineering, Management*. dPunkt, 2007.

[56] UWE. Jsf4uwe. `http://uwe.pst.ifi.lmu.de/toolUWE4JSF.html`.

[57] UWE. Magicuwe. `http://http://uwe.pst.ifi.lmu.de/toolMagicUWE.html`.

[58] UWE. Uml-based web engineering. `http://uwe.pst.ifi.lmu.de/aboutUwe.html`.

[59] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008.

[60] WebRatio. Web models. `http://www.webratio.com`.

[61] Bebo White. The implications of web 2.0 on web information systems. In Vitor Pedrosa Joaquim Filipe, José Cordeiro, editor, *Web Information Systems and Technologies*, volume 1 of *Lecture Notes in Business Information Processing*, pages 3–7. Springer Berlin / Heidelberg, 2007.

Stichwortverzeichnis