Bachelor/Master-Projekt SS 2012

Erweiterung eines Web–basierten Umfragesystems um eine XML–Schnittstelle und um XSL Transformationen

Bachelorarbeit

Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion

Institut für Informatik

Christian-Albrechts-Universität zu Kiel

vorgelegt von

Christian Wischmann

Erstgutachter: Prof. Dr. Michael Hanus

Zweitgutachter: Björn Peemöller

Kiel, den 26. September 2012

Inhaltsverzeichnis

Bildverzeichnis

Programmausdrucke

1 Einleitung

Im Rahmen des Projektmoduls Programmiersprachen und Programmiersysteme im Sommersemester 2012 sollte ein komplexes Softwaresystem mit Hilfe fortgeschrittener programmiersprachlicher Methoden erstellt werden. Ziel war die Entwicklung einer Umfragesoftware zur Erstellung, Durchführung und Auswertung von Online–Umfragen. Als Name wurde SuSy! gewählt, kurz für „*su*rvey *sy*stem“, wobei das „Ausrufezeichen“ dem Web–Framework Play! entliehen wurde.

Die Umfragesoftware sollte mit der Programmiersprache Scala [u1] sowie dem Web–Framework Play! [u2] entwickelt werden.

Im folgenden Abschnitt werden einige Kernelemente der Umfragesoftware aufgezählt und kurz erläutert.

Nachdem man sich bei SuSy! als Benutzer registriert hat, kann man durch einen Administrator das Recht bekommen, Umfragen zu erstellen und auszuwerten. Dazu kann man einen öffentlichen Fragebogen benutzen oder einen eigenen mit Hilfe eines Web–Interfaces erstellen. Ihn kann man lokal in textform speichern, bearbeiten und verschicken. Es gibt eine Importfunktion, die einen solchen Fragebogen in das System einliest und so zum Erstellen von Umfragen zur Verfügung steht. Durch eine große Anzahl an Fragetypen wird eine Vielzahl von Fragebögen unterstützt.

Umfragen startet man als SuSy!–Benutzer mit einem Web–Interface, indem man einen Fragebogen auswählte und weitere umfragespezifische Informationen hinterlegt. Die Umfrage kann für alle zugänglich sein oder durch TAN gesteuert werden.

Wer eine Umfrage erstellt hat, kann jederzeit eine aktuelle Auswertung abrufen. Die Rohdaten der Antworten kann man lokal als CSV– oder XML–Datei speichern, um diese mit anderer Software unabhängig von SuSy! weiter zu verarbeiten. Das Auswerten von Umfragen ist dem erlaubt, der die Umfrage erstellt hat. Er kann das Recht auf alle erweitern.

Der gesamte Kurs war an der Planung und Erstellung der Anwendung beteiligt. Die Erweiterung der Anwendung war das Thema meiner Bachelorarbeit. Da ich schon viel mit Java programmiert habe, interessierten mich weitere Möglichkeiten von Scala und Play! XML–Technologie zu verwenden. Eine XML–Schnittstelle für SuSy! und Transformationen für diese Schnittstelle zu programmieren war meine Zusatzaufgabe. Beim Erledigen dieser Aufgabe wird nicht nur XML mit Scala erzeugt, sondern es wird auch großen Wert auf das Transformieren und Prüfen von XML gelegt. Mit einer XSLT–Entwicklungsumgebung habe ich zwei XSLT–Dokumente geschrieben und zum Transformieren verwendet.

Einen XSLT–Prozessor bei Scala oder Play! einzubinden stellte sich als Schwierigkeit heraus, eine Scala–Alternative zu der aus Java bekannten Schnittstelle JAXP [3] gibt es derzeit nicht. Als eine Notlösung dieses Problems wurde das Transformieren in den Web–Browser ausgelagert.

Als Kommunikationsplattform und Projektleitungswerkzeug wurde während der Entwicklung Redmine[u3] verwendet. Die Projekthomepage beinhaltet alle wichtigen Informationen, die beim Entwickeln und Entwerfen der Umfragesoftware wichtig waren, wie z.B. das Lastenheft, die Entwicklungsdokumentation und die Kernelemente eines Umfragesystems. Die Projekthomepage liegt auf dem Webserver der Arbeitsgemeinschaft und ist unter der URL `http://www-ps.informatik.uni-kiel.de/redmine/projects/projekt-12s` zu erreichen.

Unser Projekt SuSy! ist zusammen mit Play! eine Web–basierte Umfragesoftware und im Internet auf `http://www-ps.informatik.uni-kiel.de/susy/` zu erreichen.

2 Aufgabe und Ziel des Projektes

Aufgabe und Ziel des Kurses war es, eine Web–Anwendung zur Eingabe, Ausführung und Auswertung von Umfragen anhand eines Lastenheftes zu entwickeln. Das Lastenheft ist auf der Projekthomepage zu sehen. Das fertige, lauffähige Programm soll zur allgemeinen Verwendung im Netz verfügbar werden. Die Teilnahme an Umfragen soll auf bestimmte Besuchergruppen der Web–Seite eingeschränkt werden. Besucher gehören einer der Gruppen an, wobei die Teilnahme an der Umfrage anonym oder personenbezogen ist, d.h. Typen der Teilnehmer von Umfragen sind:

- jeder Besucher
- registrierter Besucher
- TAN zur Teilnahme

Aus Effizienzgründen wurden drei Arbeitsgruppen gebildet, die je einen Teilbereich verstärkt zu bearbeiten hatten. Zwischen den Gruppen erfolgte ein reger Gedankenaustausch und über Redmine konnten sich alle über den Fortgang informieren und selbst eigene Ergebnisse einstellen. Auch privat wurde sich noch weiter ausgetauscht.

Die drei Teilbereiche sind:

- Fragebögen erstellen, verändern und speichern
- Umfragen erstellen, auswerten und durchführen
- Antworten sammeln, auswerten und freigeben

Der Kurs wurde, wie in Bild 2.1 zu sehen ist, aufgeteilt.

Gruppe 1 Fragebogen	Gruppe 2 Umfragen	Gruppe 3 Auswertung
Jan Bracker	Thomas Kiupel	Matthias Böhm
Mirko Heinold	Florian Micheler	Lukasz Rybinski
Lennart Spitzner	Marcus Schmöhl	Christian Wischmann
Jan Meyer		

Bild 2.1: Gruppen

2.1 Erstellen von Fragebögen

Aufgabe der ersten Gruppe war es, Fragebögen so zu entwickeln, dass sie verschiedene Kriterien erfüllten. Hier eine kurze Zusammenfassung der Überlegungen und Ergebnisse: Zunächst wurde eine Spezifikationssprache für Fragebögen erstellt. Ein kurzes Beispiel ist im Programmausdruck 1 zu finden. Eine Umfrage beginnt mit den Metainformationen,

die mit dem Schlüsselwort `Head` beginnen. Jede Frage beginnt mit einem Schlüsselwort entsprechend des Fragetyps, gefolgt von einer Identifikation und dem Fragetext. Je nach Fragetyp folgen bestimmte Parameter. Im Beispiel ist es der Parameter `required` und `length 256`. Eine Antwort wird so erzwungen und der Antworttext darf nicht aus mehr als 256 Zeichen bestehen. In Bild 2.2

```
Head evalForm "FormName" #Die Identifikation evalForm
  description "Fragebogen zum Testen der Evaluation"
  language "de" #Standardsprache ist deutsch (de)
  author "cwi" #Der Author des Fragebogens
  version 1 #Version 1.0

FreetextQuestion q1 "FreetextQuestion: Welche Tiere magst du?"
  required # Antworttext darf nicht leer sein.
  length 256 # maximal 256 Zeichen

#auf eine Seitestruktur wird verzichtet
#auf Übersetzungen wird verzichtet
```

Programmausdruck 1: Beispiel für einen Fragebogen in Textform

Zudem sollte die Möglichkeit geschaffen werden mit Hilfe einer webbasierten Schnittstelle Fragebögen zu erstellen und zu verändern.

Jeder Fragebogen hat einen Namen und eine Reihenfolge in der die Fragen angezeigt werden. Die Fragenbögen sollten Mehrsprachigkeit unterstützen. Alle Fragen sollten eine richtige Antwort haben, die eine festgelegte Punktezahl bringen. Als mögliche Fragearten waren Freitext–, Auswahl–, Matrixauswahl–, Rangordnung–, Zahlen–, Datum–, Textfragen gefordert.

Aus den Forderungen ergaben sich drei Gruppen von Fragen:

Eingabefrage: Die Eingabefrage erwartet als Antwort eine beliebig Zeichenkette. Diese Zeichenketten kann man durch reguläre Ausdrücke einschränken.

Auswahlfrage: Die Auswahlfrage erwartet als Antwort eine Teilmenge aus den vorgegebenen Antworten, d.h. der Fragesteller definiert eine Menge an Antworten, die der Antwortgeber als Teilmenge auswählen kann. Dabei wird die minimale und maximale Größe der Antwortmenge festgelegt.

Rangordnungsfrage: Die Rangordnungsfrage erwartet als Antwort eine Permutation von einer gegebenen Menge.

Der Fragebogen sollte die Möglichkeit bieten, Fragenblöcke in Seiten einzuteilen und durch logische Sprungregeln das Lenken von Seite zu Seite je nach gegebener Antwort bereitstellen.

Diese Funktionalität für Fragebögen wurde mit Scala programmiert. Dafür wurde mit Hilfe von definierten Schlüsselwörtern ein Format für eine Textdatei entworfen. Als Ergebnis wurde ein umfangreicher Fragekatalog erstellt (siehe Bild 2.2).

Eingabefrage	Auswahlfrage	Rangordnungsfrage
TextQuestion	SingleChoiceQuestion	RankingQuestion
IntegerQuestion	MultipleChoiceQuestion	
Floatquestion	SingleChoiceMatrixQuestion	
DateQuestion	MultipleChoiceMatrixQuestion	
UrlQuestion		
EmailQuestion		
TelephoneQuestion		
Freetextquestion		

Bild 2.2: Schlüsselwörter für Fragetypen

Um benutzerfreundlich mit diesem Format umgehen zu können, musste eine Benutzerführung in Play! programmiert werden. Das Ergebnis ist als Screenshot in Bild 2.3 zu sehen.

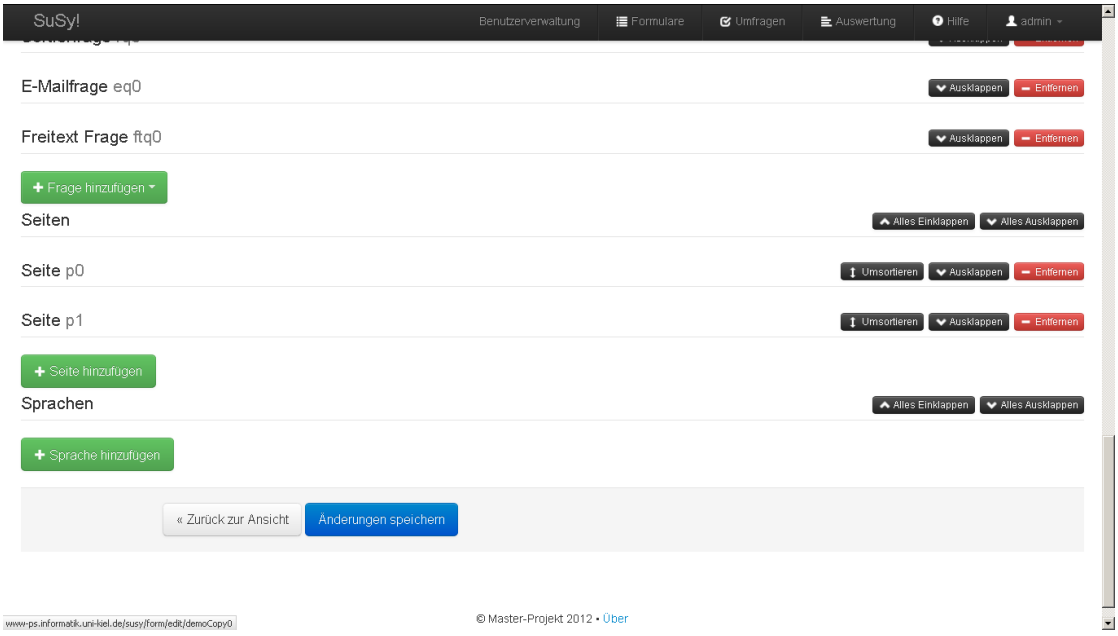

Bild 2.3: Benutzerschnittstelle für Fragebögen

Das Use–Case–Diagramm in Bild 2.4 zeigt, wie wir uns das Arbeiten mit dem Fragebögen vorgestellt haben.

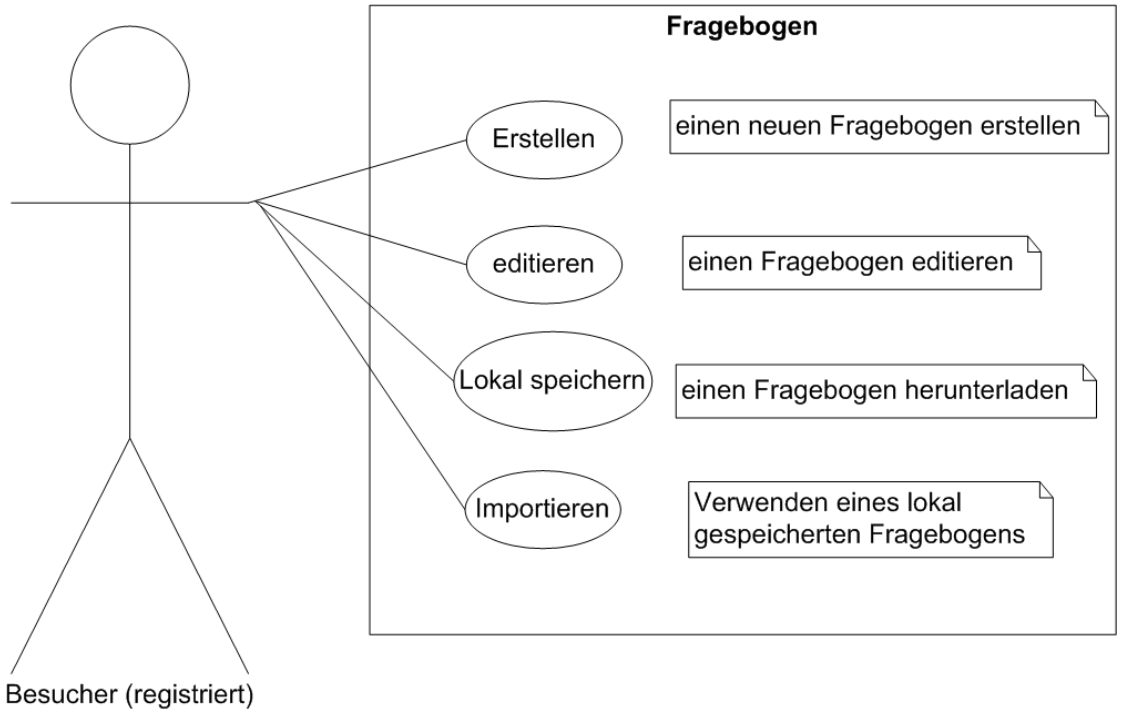

Bild 2.4: Verwendung von Fragebögen

Um aufzuzeigen, was in dieser Gruppe programmiertechnisch mit Scala geleistet wurde, folgt ein kleiner Ausschnitt aus deren Implementation:
`Form` ist eine Scala–Klasse, die in der Datei `Form.scala` definiert wird und im Verzeichnis app/model/formdsl zu finden ist. Diese Klasse wurde als Schnittstelle für die anderen Gruppen verwendet und wurde laufend angepasst und verbessert.
In der Klasse `Form` gibt es Objekte, die eine Frage repräsentieren. Die Fragen haben eine Identifikation und sind in Seiten zusammengefasst, die in Page.scala definiert sind. Die Fragen sind in der Klasse `Question.scala` definiert. Dies ist eine „case—Klasse“ und bietet dadurch gewisse Designmöglichkeiten beim Verwenden von Frageobjekten. Die unterschiedliche Funktionalität der Fragetypen wird mit Hilfe von Traits implementiert, wie im Bild 2.5 dargestellt. Jeder Fragetyp wird durch einen bestimmten Satz von Traits erweitert.

Die Kernaufgabe war es, einen Parser zu entwickeln, der entsprechende Textdateien einlesen und verarbeiten kann. Aus den geparsten „Token“ wird dann ein Fragebogen erstellt, der gewisse symantische und syntaktische Regeln befolgen muss.

Trait	Funktion	Werte
HasPoints	gewichtet die richtige Antwort	points
HasRequired	es muss eine Antwort gegeben werden	required
HasDefault	vorgegebene Antwort	default
HasChoices	hat vordefinierte Antworten	choices
HasLineAnswers	Antwort besteht aus mehr als einer Zeichenkette	minLines, maxLines
HasRangeAnswer	Antwort kann durch Minimum / Maximum eingeschränkt werden	min, max
HasFreetextAnswers	Auswahl an vordefinierten Antworten kann durch Eingabe erweitert werden	freetext
HasCorrect	Alle richtigen Antworten	correct
HasLength	Einschränkung der Länge der Antwort–Zeichenkette	length

Bild 2.5: Question Traits und die Funktion

2.2 Umfragen erstellen und ausfüllen

Die todo–Liste für die zweite Gruppe zur Nutzung des Fragebogens und zur Planung und Durchführung von Umfragen beinhaltete u.a. folgende Punkte:

Erstellen Beim Erstellen der Umfrage soll ein Fragebogen ausgewählt werden.

Ausfüllen Beim Ausfüllen von Fragebögen sollen die Antworten in einer Datenbank gespeichert werden.

Benutzerführung Eine Benuzerführung durch den Fragebogen muss implementiert werden.

Verwaltung Umfragen müssen verwaltet werden können (siehe 2.6).

Zusätzlich musste beachtet werden, dass pro Fragebogen flexibel mehrere Umfragen durchgeführt werden können. Nach einer Unterbrechnung beim Beantworten eine Fortsetzung möglich sein soll. Am Ende einer Umfrage sollte eine Zusammenfassung angezeigt werden, die ggf. die richtigen Antworten beinhaltet.

Im folgenden beschränke ich mich auf einige wichtige Teilaufgaben. Um mit den Umfragedaten zu arbeiten, musste erst ein geeignetes **Datenbankmodell** erstellt werden. Die Datenbank muss für das Speichern eines Fragebogens, der Antworten und später für das

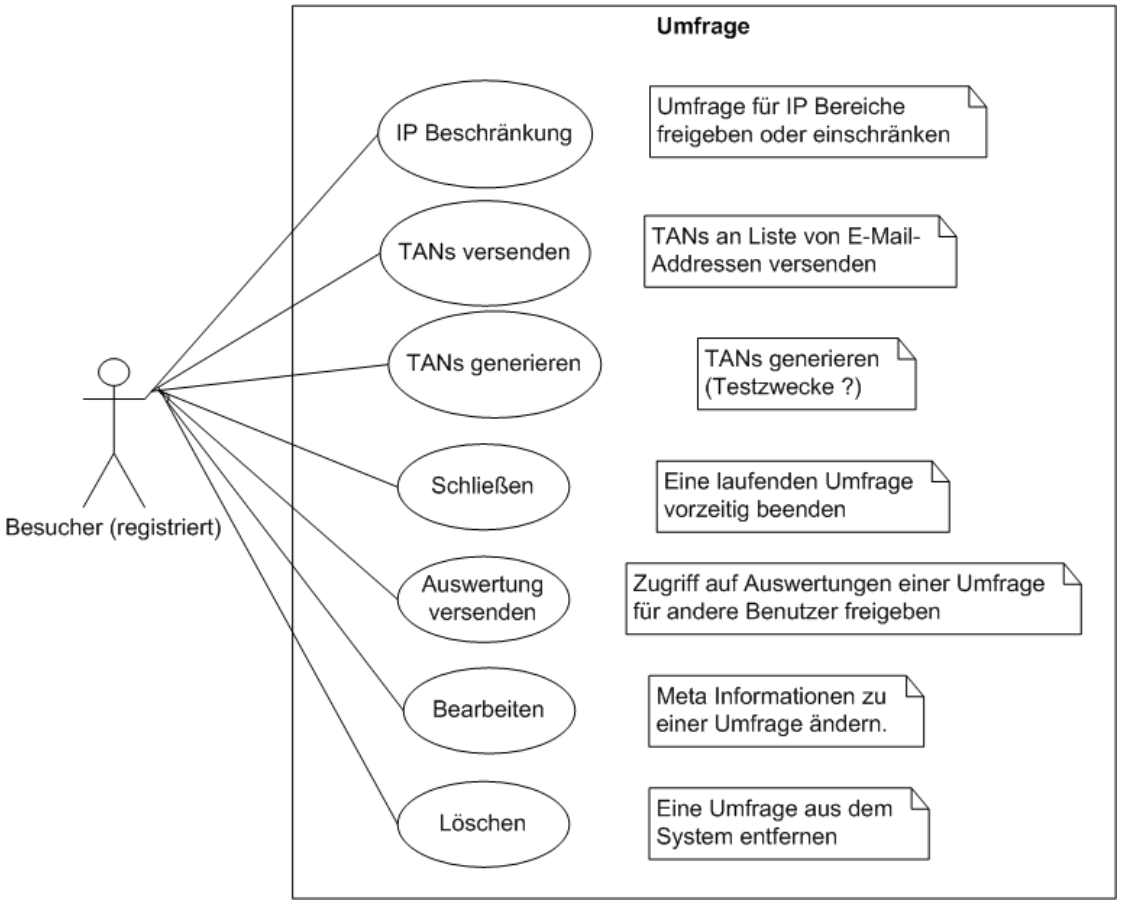

Bild 2.6: Umfrageverwaltung

Auswerten der Antworten mit Hilfe von SQL[1] geeignet sein. Alle Kursteilnehmer haben sich beim Entwerfen dieser gruppenübergreifenden Aufgabe eingebracht.

Die Struktur eines Fragebogens kann also als Identifikation in die jeweiligen Antworttabellen dienen. Antworttabellen kann man nach Antwortdatentypen definieren, damit das Auswerten der Antworten mit den Funktionen, die SQL anbietet, erledigt werden können. Das Paar Fragebogenschlüssel und Frageidentifikation kann als Identifikation in jeder Antworttabelle dienen. Eine Tabelle speichert die Fragebögen, beinhaltet also alle Fragebogenidentifikationen. Damit ergibt sich dann das folgende Datenbankschema, das im Bild 2.7 zu sehen ist.

Das Datenbankschema erfüllt alle Ansprüche. Bei der Betrachtung des Schemas fällt die hohe Anzahl an Referenzen auf. Einige davon wären nicht notwendig, wenn man gewisse Regeln beim Entwerfen befolgt hätte. Man sollte in jeder Tabelle eine Spalte für die Primärschlüssel verwenden, die durch die Datenbank generiert wird.

[1]Structured Query Language

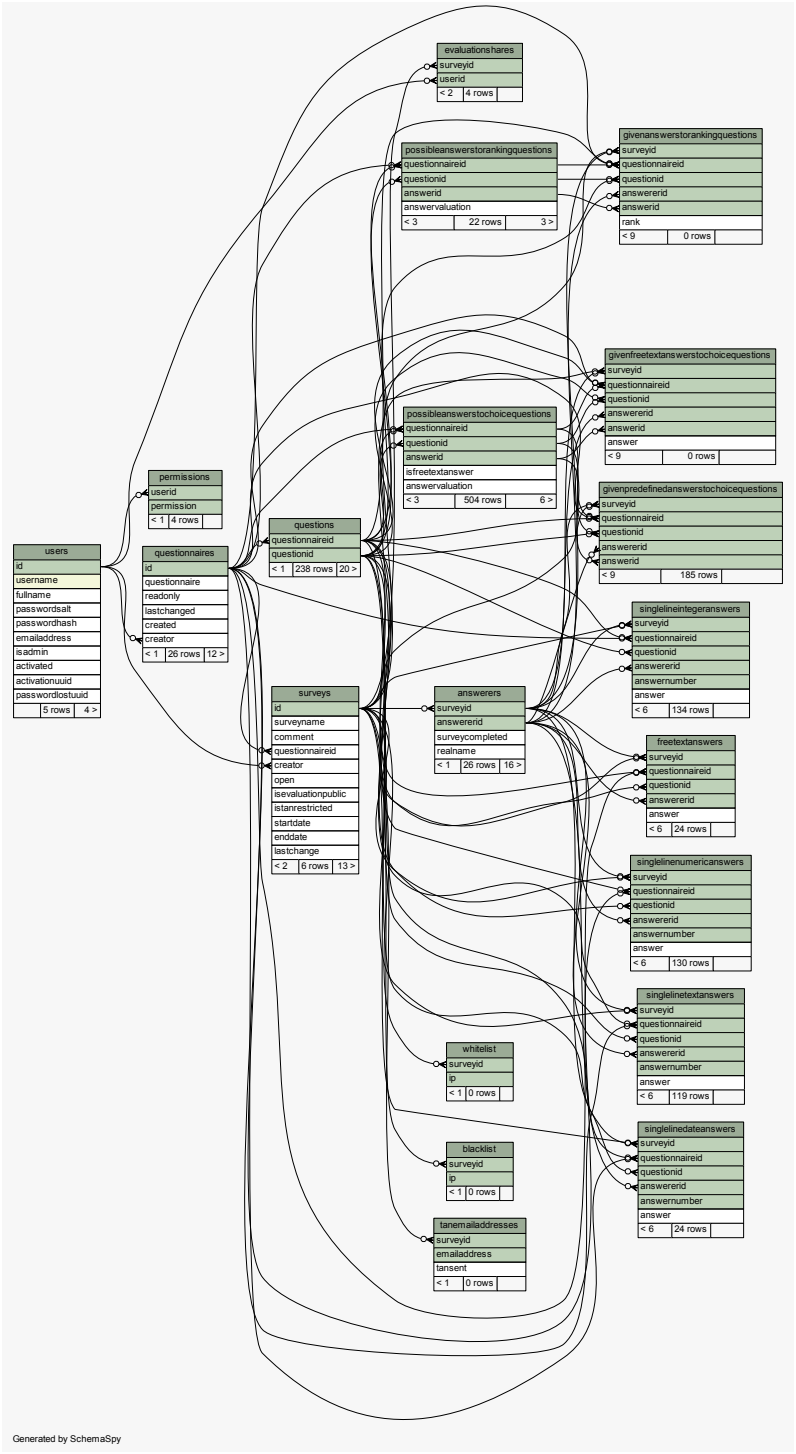

Bild 2.7: Datenbankschema mit vielen redundanten Referenzen

Leider kam mir der Erkenntnis zu spät, um das noch mit vertretbaren Einsatz einzubauen. Das resultierende Schema war schnell erstellt, eine graphische Darstellung im Bild 2.8 zeigt die vorteilhafte Klarheit.

Die Anbindung an die mit SQL erzeugte Datenbank wurde mit Scala programmiert. Mit Hilfe von Play! wurde eine Benutzerführung erzeugt, die den Teilnehmer durch den Fragebogen führt. Ein Beispiel einer Antwortmaske zeigt Bild 2.9.

2.3 Auswerten von Umfragen

Die dritte Projektgruppe befasste sich mit dem **Auswerten von Umfragen**. Aus der Aufgabenstellung ist zu ersehen, dass sie auf die Ergebnisse der ersten und zweiten Gruppe angewiesen war. Das Anwendungsfall–Diagramm 2.10 zeigt das Lastenheft dieser Gruppe. Weitere Punkte waren u.a. das Entwerfen einer Ansicht auf eine Umfrage, die z.B. für Auswahlfragen und Rankingfragen eine prozentuale Verteilung der Antworten anzeigt. Die Ansicht sollte auch Gruppen von Umfragen, basierend auf einem Fragebogen, ermöglichen. Die Rechte zum Auswerten sollten personengebunden vergeben werden. Als weitere Aufgabe sollte eine Exportfunktion angeboten werden, die weiteres Auswerten ermöglicht.

Ein kurzer Überblick über die geleistete Arbeit soll im folgenden gegeben werden:
Um die Antworten der Fragebögen auswerten zu können, muss auf die Datenbank mit SQL zugegriffen werden, um die Daten als HTML–Dokument zu erzeugen. Dafür werden die Play–Templates benutzt.

Um die Schnittstelle zu der Datenbank zu abstrahieren, wurde eine Klasse namens EvaluationDbLogic entworfen. Das programmierte Scala–Interface hat die Funktion die Umfrageergebnisse auszuwerten. Zusätzlich lieferte das Interface eine Korrelation zwischen Umfragegruppen. Die Daten aus dem Interface werden im Controller EvalDataVieController mit Hilfe von `Tupeln` mit den Daten aus dem Fragebogen zusammen geführt. In der View `evalMain.scala.html` werden die Daten mit Hilfe von CSS, unter anderem mit `display:Table` formatiert. Die Daten wurden aus dem Interface `EvalDBLogic` im Controller `EvalDataViewConnector` gespeichert, der das `Question`–Objekt und das `Answer` Objekt als `Tupel` zusammenfasst. Das `Answer`–Objekt ist dabei eine Liste von Antworten. In einem „match—statement“ werden die unterschiedlichen Fragetypen auf entsprechende Play!–Templates abgebildet, die ein HTML–Abschnitt zurück geben. Zuständig ist dabei die Logik in dem Template `evalMain.scala.html`. Dort werden mit dem Befehl `match` auf die Fragetypen die Antworten in einem entsprechendem Template verarbeitet. Die entsprechenden Templates heißen `evalQuestion.scala.html` erweitert um den jeweiligen Typ von Frage, und sind im Unterordner `app/view/eval` zu finden. Die akkumulierten Daten und die Antworten werden durch Play! als HTML verschickt und im Browser dargestellt.i Ein Beispiel zeigt das Bild 2.11.

Das Interface `EvaluationDbLogic` benutzt anorm und liefert zusammen mit den beiden Interfaces `SharingDbLogic` und `SurveysDbLogic` alle Funktionen, die für das Auswerten

von Umfragen gebraucht werden. `SharingDbLogic` bietet die Funktion, Zugriffsrechte auf Umfragen zu prüfen. `SurveysDbLogic` liefert die Daten aus der Datenbank, die zum Navigieren von Besuchern gebraucht werden.

Der sich an die Auswertung anschließende **Export** wurde von mir bearbeitet und wird im folgenden kurz dargestellt.

Absicht war es die Rohdaten aus der Datenbank mit dem Interface `EvalDbLogic.scala` zu extrahiert. Leider war das Interface nicht flexibel genug entworfen, es war zu speziell für das Web–Interface programmiert. Ich musste also mit SQL arbeiten, um an die Rohdaten zu gelangen. Es werden alle Rohdaten geliefert, dazu gehören auch die TANs. Damit die TANs geheim bleiben, müssen diese noch durch neutrale Werte ausgetauscht werden.

Als Exportmöglichkeit bietet sich an, die Antworten in das Dateiformat **CSV**[2] zu speichern. Die erste Phase war das Finden eines Schemas zum Erfassen möglichst vieler Fragetypen, außerdem sollten auch mehrere Umfragen zu einem Fragebogen unterstützt werden. Hierzu mussten zunächst die Antworten zu den Fragetypen analysiert werden. Jeder Fragetyp hat eine Umfrage–, eine Fragebogen–, eine Frage– und eine Antwortgeberidentifikation. Die Fragebogenidentifikation ist dabei bei allen Antworten gleich und kann ausgelassen werden. Es bleiben also die drei Identifikationen Umfrage, Frage und Antwortgeber übrig, um eine Struktur zu erzeugen, die mehrere Umfragen auf einmal unterstützt. Der Fragetyp `RankingQuestion` besitzt als Antwort eine Auswahlidentifikation und den Rang. Die Antwort für diesen Fragetyp ist also eine `ChoiceId` und der Rang. Der Fragetyp `ChoiceQuestion`, Einzel– und Mehrfachauswahlfragen hat vordefinierte Antworten, eine Antwortidentifikation und optional einen benutzerdefinierten Text. Es ergeben sich also vier unterschiedliche Tabellen, die jeweils in eine eigenen Tabelle gespeichert werden müssen. Die Struktur der Antworten kann man im Bild 2.12 sehen.

Damit die Identifikationen der Auswahl, der Frage und der Umfrage wieder aufzulösen ist, werden noch Zusatzinformationen gebraucht, die Meta–Daten genannt werden. Diese sind aus der `Form` zu erhalten und werden als weitere Tabelle gespeichert.

Der Scala–Quelltext für den Export ist im Unterordner `app/evaluation/export/CsvExport.scala` zu finden und wird im folgenden kurz erläutert. Mit Hilfe von SQL–Anfragen werden die Antworten aus der Datenbank in `ResultSets` gespeichert und mit Hilfe von Scala in CSV–Strings geparst. Bevor der Inhalt geschrieben wird, müssen die TANs zwecks Daten–Anonymisierung durch ein Objekt der Klasse `FakeIdHandler` maskiert werden. Die Textantworten werden durch die Funktion `escape` in der Klasse `CsvExport` sicher für eine CSV–Datei gemacht, indem alle Steuerzeichen maskiert werden. Bei CSV gibt es nur ein Steuerzeichen, der Feldtrenner, der maskiert werden muss. In der Datei `CsvExport` werden die Daten in einer `scala.Map` gesammelt. Die `Map` hat als Schlüssel den Dateinamen und als Wert den Inhalt der Datei. Um den Inhalt der Datei zu generieren wird mit Hilfe von SQL eine Anfragen generiert und mit einem Objekt

[2] „comma seperated values“ englisch für kommagetrennte Werte

von Typ `CsvExportInterface` in CSV umgewandelt. Mit dem Objekt `ZipWriter` wird die `Map` mit der Funktion `writeToFile` in einen beliebigen `OutputStream` geschrieben, der danach eine Zip–Datei enthält mit folgendem Inhalt:

- Hilfedatei
- Fragebogen (DSL)
- Metainformationen (CSV)
- Antworten (CSV)

Mit Hilfe von Play! wird die Zeit für das Erstellen des Exports in die Log–Datei `logs/application.log` geschrieben. Play! kann `InputStreams` mit Hilfe von `Enumerator` verschicken. In der Klasse `EvalViewController` wird die entsprechende Funktionalität bereitgestellt, wie man im Programmausdruck 2 sehen kann.

```
1  private def showCsv(surveyIds: List[Long])
2      (implicit conn: java.sql.Connection, form: model.formdsl.Form)
3      = {
4    val(len, fileContent) = CsvExport.getContent(form, surveyIds)
5    val filename = makeFilename(surveyIds)
6    SimpleResult(
7      header= ResponseHeader(200,
8        Map(CONTENT_LENGTH -> len,
9            CONTENT_TYPE -> "application/zip",
10           CONTENT_DISPOSITION -> ("attachment; filename=\"" +
11             filename + "\"")
12         )
13     ),
14     body = fileContent
15   )
16 }
```

Programmausdruck 2: Senden eines OutputStream mit Play!

2.4 Weitere Aufgaben

Weitere Aufgaben wie Benutzerverwaltung, Mehrsprachigkeit und das einheitliche Gestalten der Webseiten mit Hilfe von CSS [u4] und dem Play! typischen Templates haben wir nicht aufgeteilt, sondern wurden nach Auslastung der Kursteilnehmer bearbeitet.

Benutzerverwaltung Zunächst wurden drei Rechte für registrierte Benutzer geplant.

1. Mit dem Recht „Evaluator“ kann der Benutzer für ihn frei gegebenen Umfragen auswerten.

2. Mit dem Recht „Creator“ kann der Benutzer Fragebögen erstellen und Umfragen starten.

3. Mit dem umfassend Recht „Administrator“ alle Funktionen von SuSy! uneingeschränkt nutzen.

Mehrsprachigkeit Bei der http–Anfrage eines Besuchers von SuSy! sendet der Web–Browser einen Parameter seiner Sprache mit. Diesen kann Play! auslesen und den Inhalt des Rückantwort dynamisch anpassen. Dafür musste man beim Programmieren für jeden Text ein eindeutiges Schlüsselwort setzen. Eine Funktion ruft mit diesem Schlüssel und der Sprachparameter den Text ab und verwendet diesen übersetzten Text. Dieses Modul für Play! heißt „i18n“.

Einheitliches Aussehen Das Aussehen einer Internetseite wird durch CSS bestimmt. Dynamische Inhalte werden durch JavaScript erreicht. Um ein einheitliches Aussehen zu erreichen werden in einem Template alle wichtigen Merkmale gespeichert. Nun müssen alle anderen Templates dieses Template importieren. In unserem Projekt wurden diverse CSS– und JavaScript–Dateien verwendet. Bootstrap [u5] wurde als Grundlage für alle Seiten genutzt.

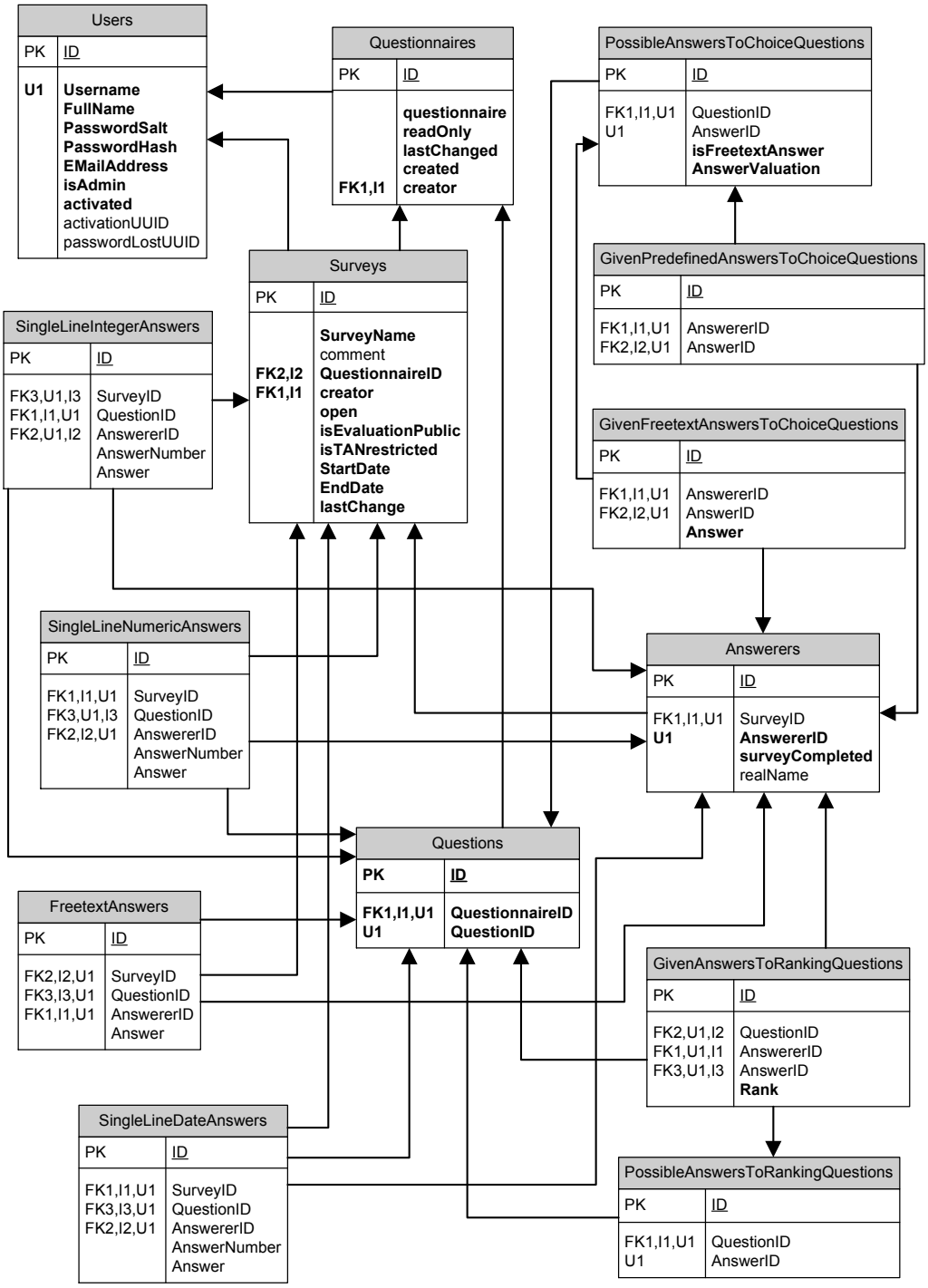

Bild 2.8: Das Datenbankschema mit zusätzlichen Feldern für die Primärschlüssel und reduzierten Referenzen

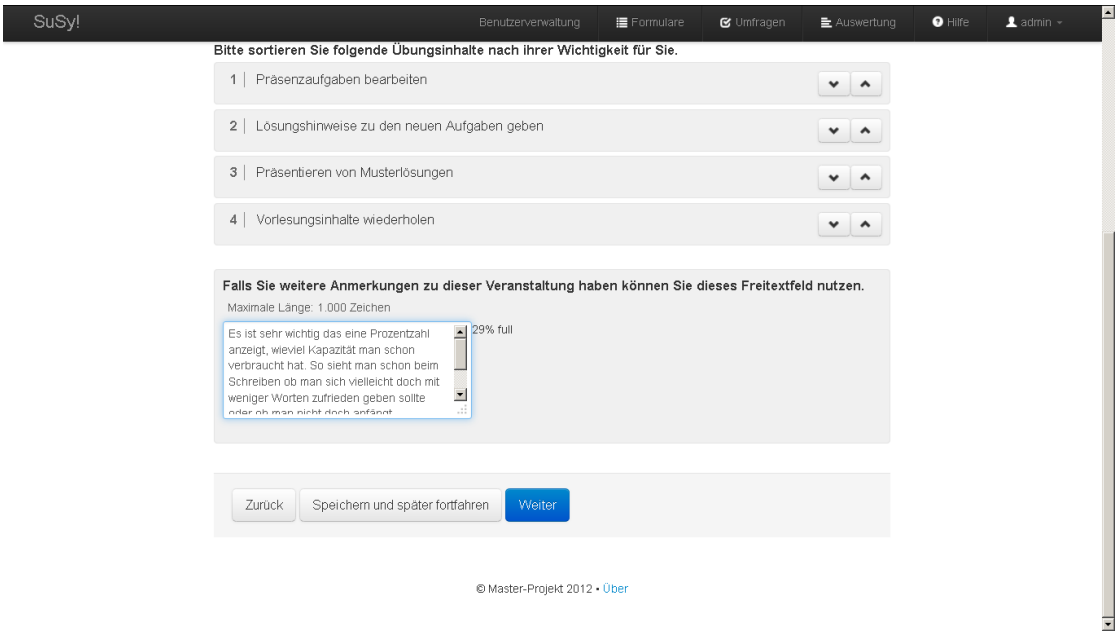

Bild 2.9: Beispiel einer Antwortmaske

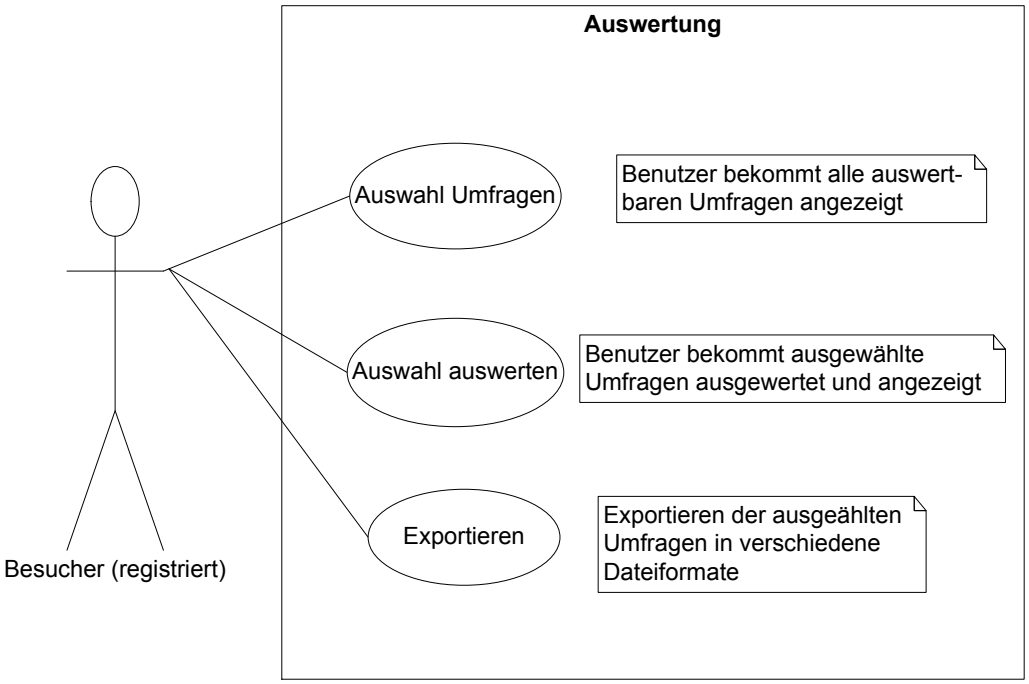

Bild 2.10: Anwendungsfall–Diagramm – Auswertung von Umfragen

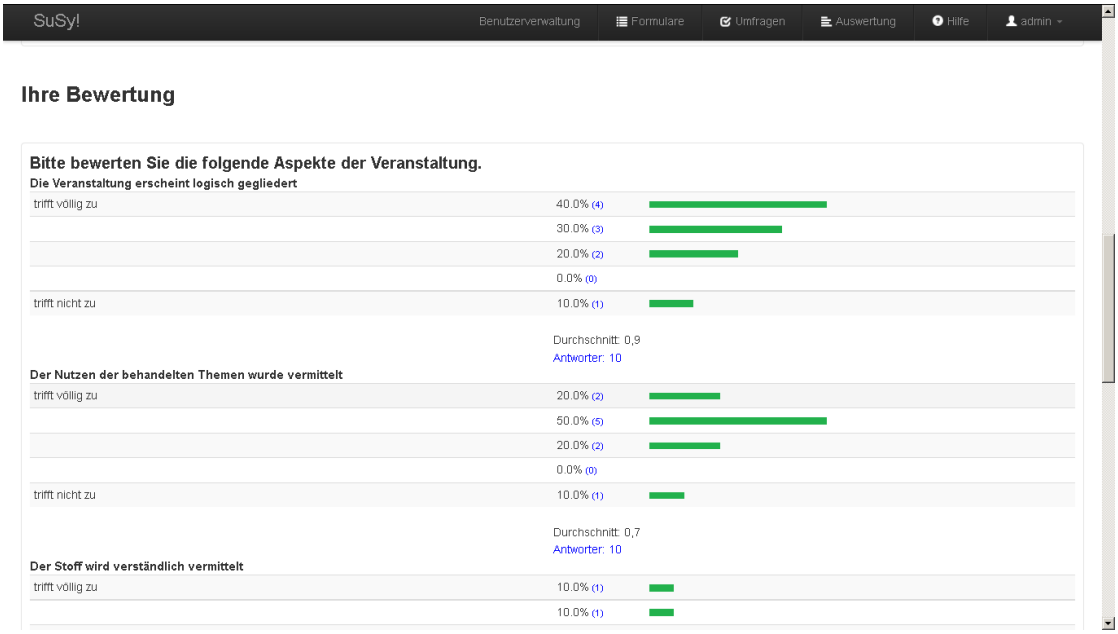

Bild 2.11: Beispiel einer Auswertungsansicht

Fragetyp	Antwort
Zeilenfrage	Text
Auswahlfrage	Auswahl mit optionalen Text
Rangordnungsfragen	Auswahl mit Rang

Bild 2.12: Antwortstruktur je Fragetyp

3 Verwendete Software

Der kurzer Exkurs soll die verwendete Systemsoftware vorstellen. Es werden nur wenige ausgewählte Besonderheiten aufgezeigt.

Als Entwicklungsumgebung wurde Eclipse[u6] zusammen mit dem Scala–IDE–Plugin[u7] empfohlen. Eine andere Möglichkeit ist die Verwendung der Play!–Console. Für Scala gibt es außerdem eine Erweiterungen für den Texteditor vim, den ich bevorzugt habe.

Einen kurzen Überblick verschafft das Bild 3.1.

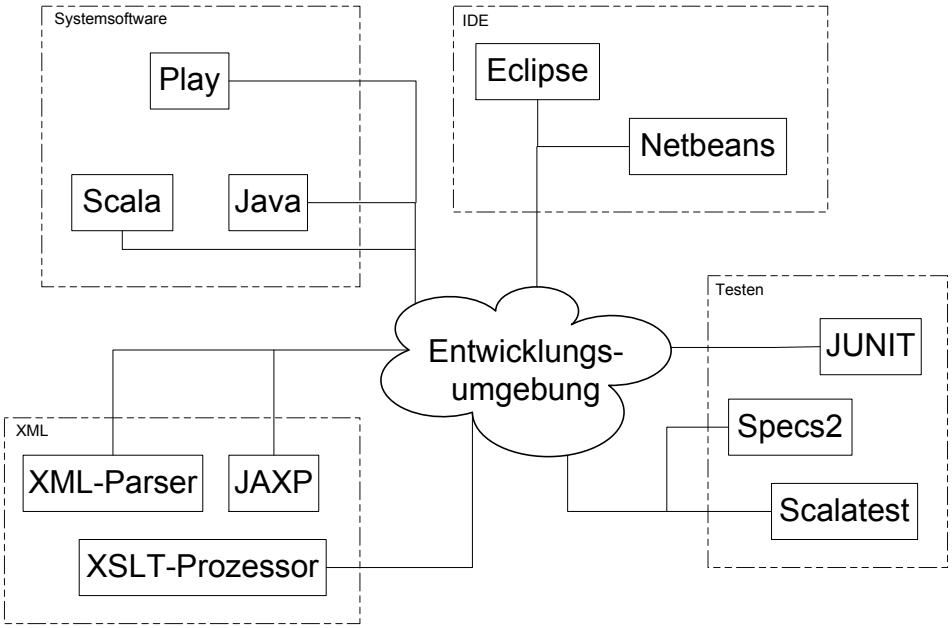

Bild 3.1: Entwicklungsumgebung

Git [u8] wurde als Versionskontrollsystem eingesetzt, um eine kontrollierbare und nachvollziehbare Teamarbeit zu ermöglichen.

Um Daten serverseitig speichern zu können, wird eine Datenbank benötigt. Als Datenbank wurde während der Entwicklung aus praltischen Gründen die in Play! integrierte H2–Datenbank [u9] verwendet, für das fertige Produkt der leistungsstarke Datenbankserver Postgres [u10].

3.1 Scala

Scala[3] ist eine funktionale, imperative, erweiterbare und objektorientierte Programmiersprache, die dem Projektkurs vorgegeben wurde. Im ersten Teil des Kurses und privat u.a. mit dem Lehrbuch [11] wurde sie erarbeitet. Viele typische Eigenschaften dieser Programmierparadigmen haben großen Einfluss auf den Entwurf von Scala genommen. Durch die gelungene Kombination dieser Paradigmen sind Möglichkeiten für den Programmierer entstanden, die zu leicht lesbarem und vor allem kurzem Quellcode führt. Durch das statische Typsystem können Typ–Fehler beim Kompilieren gefunden werden. Das Typsystem ist dabei flexible. Durch „type targeting“, ein Mechanismus zum Ermitteln der Typen im Quelltext, kann die Anzahl von Typenbezeichnung reduziert werden, wie man im Programmausdruck 3 sehen kann. Jede Zeile hat dabei denselben Effekt.

```
1 val typeInt :Int = 5 :Int
2 val typeInt: Int = 5
3 val typeInt = 5
```

Programmausdruck 3: Typensystem bei Scala

In Scalas eingebaute Typen sind identisch mit denen von Java. Dadurch kann Scala–Code in Java–Bytecode kompiliert werden und so in jeder Standard–Java–Plattform laufen und umgekehrt.

Im folgenden werden einige Konzepte von Scala vorgestellt, die beim Erstellen der Anwendung verwendet wurden.

Funktionale Konzepte

Alle Objekte und Symbole können unveränderbar sein. Mit dem Schlüsselwort `val` definiert man unveränderbare Symbole, vergleiche mit dem Programmausdruck 4. Ein erneutes Setzen einer solchen Variablen führt zu einem Kompilierungsfehler. In Java wird das Verhalten durch verwenden von `final` erhalten.

```
1 val immutableVariable : Int = 10
```

Programmausdruck 4: unveränderbares Symbol

Scala bietet ganze Pakete von Klassen an, deren Objekte unveränderbar sind. Ein typischer Vertreter sind die Listen bei Scala. Bemerkenswert ist der konstante Speicherverbrauch bei der Verwendung bestimmten Listenfunktionen, der durch „structural sharing“ erreicht wird. Ein Hinzufügen eines Elementes speichert nur das neue Element und verwendet das Objekt der Liste im Speicher weiter. Werden zwei Listen verbunden wird kein weiterer Speicher verbraucht [11, Kapitel 18].

[3] Scala wird skah—lah ausgesprochen und ist die Kurzschreibweise für das englische „*sca*lable *la*nguage“, was mit frei wachsende Sprache übersetzt werden kann.

Methoden, die unveränderbare Objekte als Parameter haben, können ohne Seiteneffekte operieren. Es gilt, Felder sind immer veränderbar, Listen dagegen nicht. Mengen und Abbildungen sind je nach Erstellung entweder das eine oder das andere. Dies wird durch die Verwendung von Traits erreicht, die zu den objektorientierten Aspekten von Scala gehören. Das Bild 3.2 zeigt die Verwendung am Beispiel von einer `HashMap`.

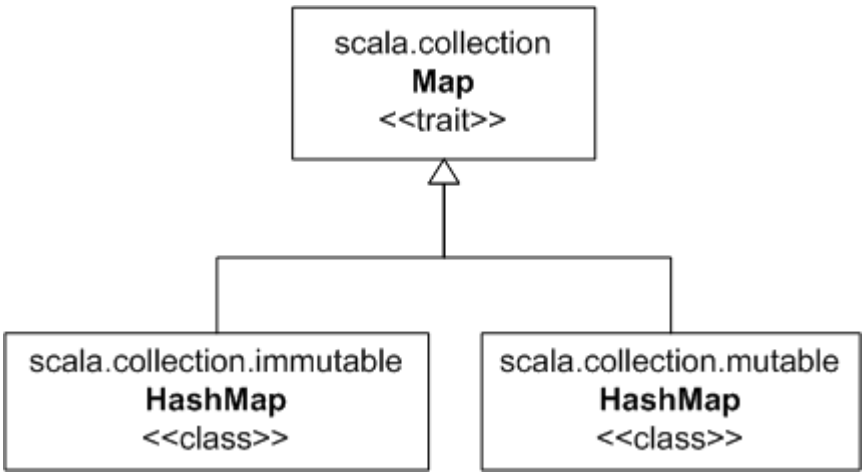

Bild 3.2: Paketstruktur bei Scalas Map–Datentyp

Ein weiterer Aspekt der funktionalen Programmierung ist das Behandeln einer Funktion als gleichwertig zu Symbolen. So können Funktionen wie Symbole definiert, in Funktionen selbst definiert werden und anonym sein. Bei Funktionen als Symbol und als anonyme Funktionen werden die Parameterlisten mit Klammern gekennzeichnet und mit dem Zeichen `=>` wird der Funktionsrumpf angezeigt. Ein Beispiel findet man im Programmausdruck 5.

```
1 // Funktionsdefinition
2 def successor: Int (x: Int) = x + 1
3 //Funktion als Symbol
4 val successor = (x: Int) => x + 1
```

Programmausdruck 5: Funktion

Mit dem Befehl `successor(0)` liefert Scala den Nachfolger von 0, also die 1 zurück.

Parameter einer Funktion können auch Funktionen sein. Indem man Parameterlisten von Funktionen klammert definiert man sie als partielle Funktion. Im Programmausdruck 6 kann man sehen, wie man aus einer Funktion `mult`, die zwei Zahlen multipliziert, eine partielle Funktion `mult2` definieren kann, die jeden Wert verdoppelt.

```
1 def mult(a: Int)(b: Int) = a * b
2 val mult2 = mult(2) // mulmult2(a: Int) = 2 * a
```

```
3 val acht = mult2(4) // acht = 8
```

Programmausdruck 6: Partielle Funktion

Man kann im Scala–Code Symbole und Objekte veränderbar machen und damit können Methoden Seiteneffekte haben.

Bei Scala sind imperativer Code und Kontrollstrukturen möglich.

Konzepte von OOP

Klassen werden mit dem Schlüsselwort `class` definiert. Eine Liste an Parametern mit Typen hat gleich mehrere Funktionen. Die Klasse definiert Felder und es wird ein Constructor sowie Methoden zum Lesen und Schreiben der Felder generiert. Der Programmausdruck 7 zeigt eine Klassendefinition.

```
1 class MyClass(val name: String)
```

Programmausdruck 7: Klassen

Klassen, die mit dem Schlüsselwort `case` definiert werden, bieten eine ganze Reihe weiterer Funktionen an. So bezeichnete Klassen verwendet man zum Erstellen rekursiver Datenstrukturen. Auf die kann man die Kontrollstruktur `match` anwenden. Ein Beispiel der Klasse `MayBeString` findet man im Programmausdruck 8.

```
1  //Fallklasse
2  import java.lang.String
3  sealed abstract class MayBeString
4  case class IsString(val string:String) extends MayBeString
5  case class NoString extends MayBeString
6
7  //Strukturprüfung der Fallklassen
8  def print(printMe: MayBeString) = printMe match{
9    case NoString() => println("Nothing to print.")
10   case IsString(string) => println(string)
11 }
12
13 val nothing = NoString()
14 val hello = IsString("Hello World!")
15
16 print(nothing) // Ausgabe: Nothing to print.
17 print(hello) // Ausgabe: Hello World!
```

Programmausdruck 8: Klassen mit `case`

Alle Befehle bei Scala sind Funktionsaufrufe. Hinter der Schreibweise `1+2` verbergen sich Objekte vom Typ `Int` und der Methodenaufruf `+`, Namen für Funktionen können aus Sonderzeichen bestehen.

Es gibt bei Scala abstrakte Klassen und Traits, die ähnlich wie Klassen definiert werden. Im Programmausdruck 9 ist eine Definition eines Traits zu sehen, die eine abstrakte Klasse erweitert. Die Funktion `calcSomething` wird dabei überschrieben. Traits haben den Verwendungsumfang der Java–Interfaces, können aber mehr.

```
1  abstract class Superclass{
2    def calcSomething(x: Int) = {
3      val result = 2 * x
4      println(x + " * 2 = " + result)
5      result
6    }
7  }
8
9  trait Double extends Superclass{
10   override def calcSomething(x: Int) = {
11     val result = x * x
12     println(x + "^2 = " + result)
13     super.calcSomething(result)
14   }
15 }
16
17 trait Successor extends Superclass{
18   override def calcSomething(x: Int) = {
19     val result = x + 1
20     println(x + " + 1 = " + result)
21     super.calcSomething(result)
22   }
23 }
```

Programmausdruck 9: Traits und abstrakte Klassen

Traits können dazu benutzt werden abstrakte oder definierte Methoden und Felddefinitionen zu vererben oder zu überschreiben. Dabei sind Traits vergleichbar mit abstrakten Klassen von C++, aber ohne das undefinierte Verhalten bei Mehrfachvererbungen. Es ist klar definiert, was bei Mehrfachvererbung in der Ergebnisklasse mit der Verwendung des Schlüsselworts `super` im Objekt passiert. Als Beispiel wird eine einfache Rechnung durch das Hinzufügen von Traits durchgeführt. Das Ergebnis der Rechnung ist dabei klar durch das Konzept der Linearität von Klassenhierarchien und Traits definiert [11, Kapitel 12.6]. Im Programmausdruck 10 erkennt man, wie Scala mit Mehrfachvererbung umgeht. Genaue Informationen gibt es in der Scala Spezifikation.

```
1 /*a calculator who use double trait than successor*/
```

```
2  class Calculator23 extends Superclass with Double with Successor
3
4  val calc23 = new Calculator23
5
6   /* first surccesor than double */
7  class Calculator32 extends Superclass with Successor with Double
8
9  val calc32 = new Calculator32
10
11 println(calc23.calcSomething(7))
12 /*
13  * 7 + 1 = 8
14  * 8 ^ 2 = 64
15  * 64 * 2 = 128
16  * 128
17  */
18
19 println(calc32.calcSomething(7))
20 /*
21  * 7 ^ 2 = 49
22  * 49 + 1 = 50
23  * 50 * 2 = 100
24  * 100
25  */
```

Programmausdruck 10: Mehrfachvererbung bei Traits

Scala bietet an, viele programmiersprachliche Methoden zu unterstützen. Mit Scala wurde das Web–Framework Play! entwickelt. Die Besonderheiten von Play! werden im nächten Abschnitt zusammengefasst.

3.2 Play! 2.0

Play! ist ein Framework[4] zum Erstellen von Web–Anwendungen. Es wurde zunächst von der Firma Zenexity [u11] entwickelt und anschließend von einer Gemeinschaft als ein Open–Source–Projekt weiter entwickelt. Play! übernimmt ein breites Spektrum an Aufgaben in der Web–Entwicklung [4]. Die Entwickler von Play! haben großen Wert darauf gelegt, die neuesten Techniken des Webs zu unterstützen. Die Play!–API [5] ist momentan lediglich eine Darstellung aller Klassen und Funktionen mit wenig Erklärung und Bespielen. Das Verständnis muss man sich mühselig aus der umfangreichen Dokumentation und aus den Beispielprogrammen erarbeiten. Bei unserem Projekt musste oft Einblick in den Play!–Quellcode genommen werden, um ein unerwartetes Verhalten des Programmes zu

[4]englisch für Rahmenstruktur

verstehen. Als Nebeneffekt erhielt man so einen tieferen Einblick in die Hochsprache Scala.

Der Aufbau einer Play!–Anwendung wird im Web gut beschrieben [1] und auch in unserem Projekt wurde diese Struktur eingehalten. Ein Teil der Struktur basiert auf das M–V–C–Designpattern (siehe 3.3).

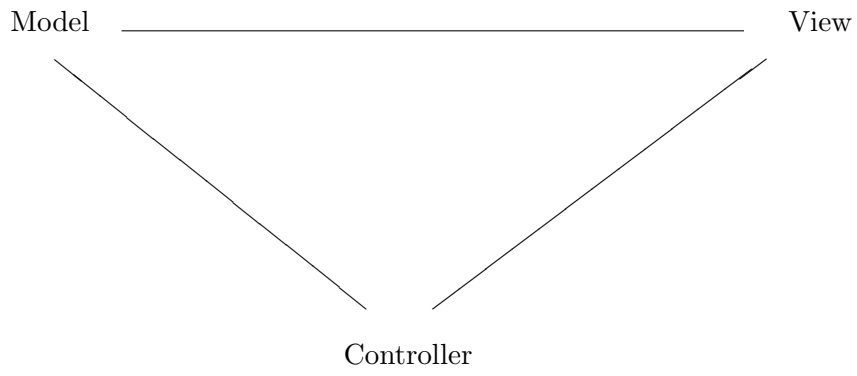

Bild 3.3: Das MVC–Designpattern

Der Browser übernimmt die Aufgabe der View. Der Inhalt wird mit Hilfe von Templates erstellt. Die HTTP–Anfragen werden mit Hilfe einer Datei namens `routes` zum Controller gelenkt, die dann den entsprechenden Programmcode ausführen.

Bei Play! wird das sogenannte „3–Tier–Schichtenmodell“ befolgt [12, Kapitel 1]. Der Web–Browser wird als getrennte Schicht betrachtet, die man mit Hilfe von HTML, CSS und JavaScript programmieren kann. Die zweite Schicht ist die Anwendungsschicht, die bei Play! wahlweise mit Scala oder Java programmiert werden kann und übernimmt alle Aufgaben die serverseitig anliegen. Die dritte Schicht ist die Datenhaltungsschicht. Wie bereits oben erwähnt bietet Play! H2 als Entwicklungs–Datenbank an. Mit Hilfe von „evaluations“ kann ein Datenbankschema im laufenden Betrieb entwickelt und geändert werden und Änderungen am Datenbankschema können direkt eingespielt und getestet werden.

Als Erstellungssystem wird bei Play! „sbt“[u12] verwendet.

Zum Testen werden JUnit [u13], scalatest [u14] und specs2 [u15] angeboten. Im Kurs wurde scalatest und specs2 vorgestellt und intensiv verwendet.

3.3 XML und XSLT

Die für meine Arbeit wichtigen Aspekte der Technologien rund um XML sollen kurz vorgestellt werden.

Mit XML werden Daten getrennt von jeglicher View gespeichert und strukturiert. In XML gibt es Tags und Attribute. Jede Bezeichnung (Tag) hat einen öffnenden und einen schließenden Teil, gemeinsam als Paar sind sie ein Element, das mit anderen ineinander verschachtelt werden kann. Dabei ist zu beachten, dass bei einem schließenden Tag von außen kein öffnendes Element im Inneren sein darf. Im Programmausdruck 11 sieht man zwei falsch verschachtelte Elemente.

```
1 <a><b></a></b> <!-- FEHLER -->
```

Programmausdruck 11: Fehlerhaftes XML–Element

Die Elemente im Inneren werden Kindelemente genannt. In einem XML–Dokument gibt es genau ein Wurzelelement, alle anderen Elemente werden von diesem umschlossen. Ein Element kann eine beliebige Anzahl von Attributen enthalten. Den Attributen wird ein Wert mit dem Zeichen = zugeordnet und dieser wird in Anführungszeichen geschrieben. Man kann jedes Attribut eines Elements zu Kindelemente umwandeln, was den Vorteil hat, dass ganz auf Attribute verzichten werden kann. Zur Erläuterung dient der Programmausdruck 12.

```
1 <tag attrib1="1" attrib2="2"></tag>
2 <tag><attrib1>1</attrib1><attrib2>2<attrib2></tag>
```

Programmausdruck 12: Entfernen von Attributen

XML [2], XSD, HTML und XSLT [7] sind so genannte Auszeichnungssprachen und bilden den Kern der XML–Technologie.

In einem XML Schema werden Typen definiert, aus denen die Elemente des XML Dokumentes bestehen. Diese Typendefinitionen können atomare Typen (siehe Bild 3.3) mit Restriktionen erweitern, um die mögliche Eingabe zu begrenzen. Weiter können sie

XSD–Datentyp	Scala Datentyp
xs:string	scala.Predef.String
xs:float	scala.Float
xs:integer	scala.Int
xs:Boolean	scala.Boolean
xs:date	java.util.Date

Bild 3.4: Kleine Auswahl einfacher Daten–Typen

komplexe Datentypen erstellen, die selbst Kindelemente enthalten.

XSLT ist eine funktionale, beschreibende, regelbasierte Sprache. Es werden Template–Regeln durch Mustervergleiche mit dem Quelldokument gefeuert. Eine Wertzuweisung im Sinne von Seiteneffekten gibt es nicht (siehe [8, Seite 173-174]).

XSLT besteht aus drei Technologien und wurde von W3C spezifiziert [7].

- XSL Transformation
- XSL Formatierungsobjekt
- XPath

XSLT wurde entwickelt, um XML–Dateien in andere XML–Dateien zu transformieren. Eine XML–XSLT basierte Web–Anwendung kann theoretisch ein breites Spektrum an Klienten beliefern, wobei das M–V–C–Designpattern beachtet wird.

4 Erweiterung von SuSy! mit XML–Technologie

4.1 Web–Anwendung mit XML und XSLT

Wie schon in der Einleitung erwähnt, wollte ich (auch aus persönlichem Interesse) versuchen, SuSy! mit XML–Technologie zu erweitern.

Wie schon dargestellt werden bei einer Web–Anwendung Daten seperat von der Darstellung verarbeitet. Nochmals der Vorteil, man kann dieselben Daten für unterschiedliche Darstellungsanforderungen verwenden.

Die Überlegungen zur Verknüpfung von XSLT mit SuSy! ist im Bild 4.1 veranschaulicht dargestellt:

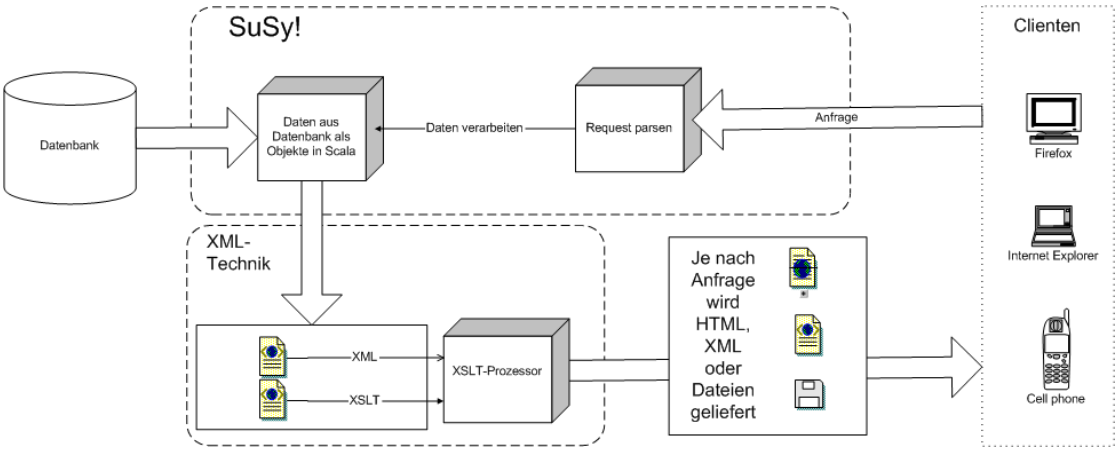

Bild 4.1: SuSy! mit XML–XSLT Technologie

Nach dem Eingang einer Anfrage werden Daten als XML–Objekt aus der Datenbank geholt. Ein Controller entscheidet dann, welches XSLT–Objekt zum Transformieren verwendet werden soll. Falls der Client über einen XSLT–Prozessor verfügt, kann das Transformieren ausgelagert werden. Wenn man eine XML–Schnittstelle für SuSy! entwickelt, muss man eine Auszeichnungssprache definieren. Dazu ist eine Analyse der verwendeten Daten des Projekts als Vorarbeit zwingend notwendig. Während bei der Analyse des CSV–Exports die Antworten im Fordergrund stehen (siehe 2.3), wird beim XML–Export der Fragebogen zusätzlich genauer analysiert.

Einen Fragebogen kann man in XML von außen nach innen definieren, wie im Programmausdruck 13 zu sehen ist:

```
1 <form>
2   <id>Eine eindeutige Identifikation</id>
3   <name>Fragebogenname<name>
```

```
4   <description>Dieser Text beschreibt den Fragebogen</description>
5   <language>Die Sprache des Fragebogens</language>
6   <author>Der Autor des Fragebogens</author>
7   <version>2.0</version>
8   <pages><!-- Alle Seiten als XML --></pages>
9 </form>
```

Programmausdruck 13: XML–Darstellung eines Form–Objektes

Der Fragebogen besteht aus Seiten. Jede Seite besitzt eine Kontrollstruktur, mit der die nachfolgenden Seiten bestimmt werden. Auf jeder Seite sind beliebig viele Fragen vorhanden. Diesen Abschnitt kann man als XML, wie im Programmausschnitt 14 gezeigt, definieren:

```
1 <page>
2   <id>Eine eindeutige Identifikation</id>
3   <name>Seitenname<name>
4   <questions><!--Alle Fragen als XML--></questions>
5   <gotos><!--Folgende Seiten--></gotos>
6 </page>
```

Programmausdruck 14: XML–Darstellung eines Page–Objektes

Um einen Fragebogen als XML darzustellen, müssen die Fragen mit den Eigenschaften abgebildet und ihre Reihenfolge gespeichert werden.

Beim Analysieren der Datenstruktur der Fragen, also der Klasse `Question.scala`, fallen die vielen Constructoren mit ihren vielen Parametern auf, was aber nicht von Nachteil ist. Die Fragetypen sind in einer „case–Klasse“ zusammengestellt. Mit Traits werden zum abstrakten Typ Question noch zusätzliche Funktionen eingemischt. Viele davon sind in `List` oder `Option` eingebunden. Weshalb man die Werte wegen „type erasure“ nicht zum Überladen von Funktionen verwenden kann. Aus dem Grund gibt es in der Datei `XmlHandler.scala` Funktionen wie `correct1 correct2 correct3`, die explizit je nach Datentyp aufgerufen werden.

Alle Fragetypen haben als gemeinsame Werte:

- `Id`
- `text`
- `typeOf`

Der Wert`Id` ist die eindeutige Identifikation zu anderen Fragen im selben Fragebogen, der Wert `typeOf` ist eine Aufzählung, bekannt als „Enumeration“ und dient als zweite Möglichkeit, neben der `case--class`–Eigenschaft, die Fragetypen in einem `switch--case` abzufragen. Beim Entwerfen des XML Schemas kann auf unterschiedliche Bezeichnungen von Fragen verzichtet und nur das `typeOf` wird gespeichert, d.h. alle Fragen sind

`question`–Elemente im XML–Dokument. Das Kindelement `type` dient im XML als ausreichendes Unterscheidungskriterium. Die `case--class` bietet neben dem switch–case auch das so genannte Patternmatching an. In XPath ist ein Patternmatching auch über Attribute und Kindelemente möglich und es werden somit genügend Möglichkeiten geboten.

In der Klasse `Form` sind alle Fragen in einer `Map` gespeichert. In einer Page werden die `QuestionId` verwendet, um die Fragen aufzulisten. Jede Seite hat eine `PageId`. Diese wird dafür verwendet, die Reihenfolge und die Verzweigungen der Seiten zu beschreiben. Dies passiert in der Klasse `Goto`.

Dies alles lässt sich bei XML ohne Identifikationen als Baumstruktur darstellen:
Jede Frage ist genau auf einer Seite vorhanden, jedes Element einer Seite bekommt alle Fragen als Kindelement. Der Graph, der durch die Verknüpfung der Seiten aufgespannt wird, ist frei von Kreisen.

Alle Werte der `Form` sind nicht durch `private` versteckt. Deshalb kann die Adapterklasse `XmlHandler` alle Werte auslesen und die XML–Elemente mit entsprechenden Kindelemente füllen. Eine Änderung im Scala–Code der `Form` ist an keiner Stelle notwendig.

Die Antworten werden mit Hilfe von `anorm` und einem Parser in XML umgewandelt. Dabei werden die SQL–Anfragen des CSV–Exports in leicht veränderter Form wiederverwendet. Zur Ausgabe werden beide Elemente durch eine Transformation mit Hilfe von `scala.xml.transform` zu einem zusammen geführt. Dabei werden die Antworten nach der FrageIdentifikation gefiltert. Wegen des besonderen Designs der Matrixfragen musste der Scala–Code nachträglich erweitert werden, damit die Filterfunktion für diesen Fragetyp korrekt arbeiten konnte.

Um die Struktur der Auszeichnungssprache zu definieren, gibt es zwei Ansätze: Zum einem das DTD und zum anderen das XML Schema. Um die Korrektheit eines XML–Dokuments sicher zu stellen, wurde eine XSD [5] definiert. Dies ist notwendig für den Import von XML. Man analysiert die XML–Datei und erzeugt entsprechend der Datenstruktur ein Schema.

Die einzelnden Teilergebnisse sind im folgenden zusammengestellt:

1. Die XSD–Datei findet man in unserem Projektordner „public/xsd/survey.xsd“
2. Ein XML–Dokument ist im Programmausdruck 15 zu sehen.
3. Eine Auflistung aller Datenobjekte, die um XML erweitert wurden, stellt Bild 4.2 dar.
4. Das Objekt `XmlHandler.scala` mit den Funktionen (siehe Bild 4.3).

[5] XML Schema Definition

```xml
<?xml version="1.0" encoding="UTF-8"?>
<surveys>
<survey>
<name>Umfragebeispiel</name>
<form>
<pages>
<page>
<questions>
  <question>
    <id>scq1</id>
    <type>SingleChoice</type>
    <choices>
      <choice><id>c1</id></choice>
      <choice><id>c2</id></choice>
    </choices>
    <answers>
      <answer>
        <question><id>scq1</id></question>
        <answerer><id>1</id></answerer>
        <choice><id>c2</id></choice>
      </answer>
      <answer>
        <question><id>scq1</id></question>
        <answerer><id>2</id></answerer>
        <choice><id>c2</id></choice>
       </answer>
    </answers>
  </question>
  </questions>
  </page>
  </pages>
  </form>
  <answerers>
    <answerer><id>1</id><complete/></answerer>
    <answerer><id>2</id><complete/></answerer>
  </answerers>
</survey>
</surveys>
```

Programmausdruck 15: SuSy–XML Beispiel von einer Umfragen

Die Funktionen, wie sie Bild 4.3 zeigen, können vielseitig verwendet werden. Es können sowohl ein Fragebogen, die Antworten oder Antwortgeber zu einer Umfrage als auch die

Objekte	Vorgehen
Form	Adapterklasse zum Generieren von XML
Answer	Alle Antworten aus der Datenbank

Bild 4.2: Mögliche Daten für die XML–Representation

Funktion	Ausgabe (als scala.xml.Elem)
`surveys(ids: List[Long])`	alle Umfragen aus der Liste `ids` aus der Datenbank
`survey(id: Long)`	die Umfrage mit der id aus der Datenbank
`answers(id: Long)`	alle Antworten zu einer Umfrage `id`
`answerers(id:Long)`	die Antwortgeber einer Umfrage
`form2xml(form: Form)`	Ein Form–Objekt als XML

Bild 4.3: XmlHanlder — Adapterklasse für XML

gesamte Umfrage abgerufen werden. Eine Umfrage besteht dabei aus einem Fragebogen und den Antworten, zu denen eine Liste aller Teilnehmer gehört. Eine weitere `RewriteRule` anonymisiert mit einem Objekt vom Typ `FakeIdHandler` die TANs.

4.2 XSL Transformation

Das Entwickeln der XSLT wurde mit Hilfe von Treebeard [u16] gemacht, weil Scala keine Schnittstelle zum Transformieren von XML anbietet, um XSLT Dokumente zu parsen und als Scala–Objekt zu verwenden. Eine Verwendung von JAXP in Scala wie im Bild 4.4 zu sehen ist wenig praktikabel.

Nach Rücksprache wurde die gängige Praxis angewendet, alle Transformationen mit XSLT auf der Seite des Clients durchzuführen. Eine kleine Einschränkung ist die Tatsache, dass die gängigen Browser XSLT–Prozessoren verwenden, die nur XSLT 1.0 verstehen, wobei XSLT 2.0 die fortschrittlichere Methoden anbieten würde, die man — wenn überhaupt — nur auf umständliche Art und Weise in XSLT 1.0 erzeugen kann (viele dieser vortteilhaften Methoden von XSLT 2.0 findet man im Buch [8]).

4.2.1 Transformation von XML nach HTML

Damit der Browser das XML–Dokument in HTML umwandeln kann, bedarf es folgender Zeile im XML–Dokument:

```
<?xml-stylesheet type="text/xsl"
href="*/assets/xslt/susyxml2html.xsl"?>
```

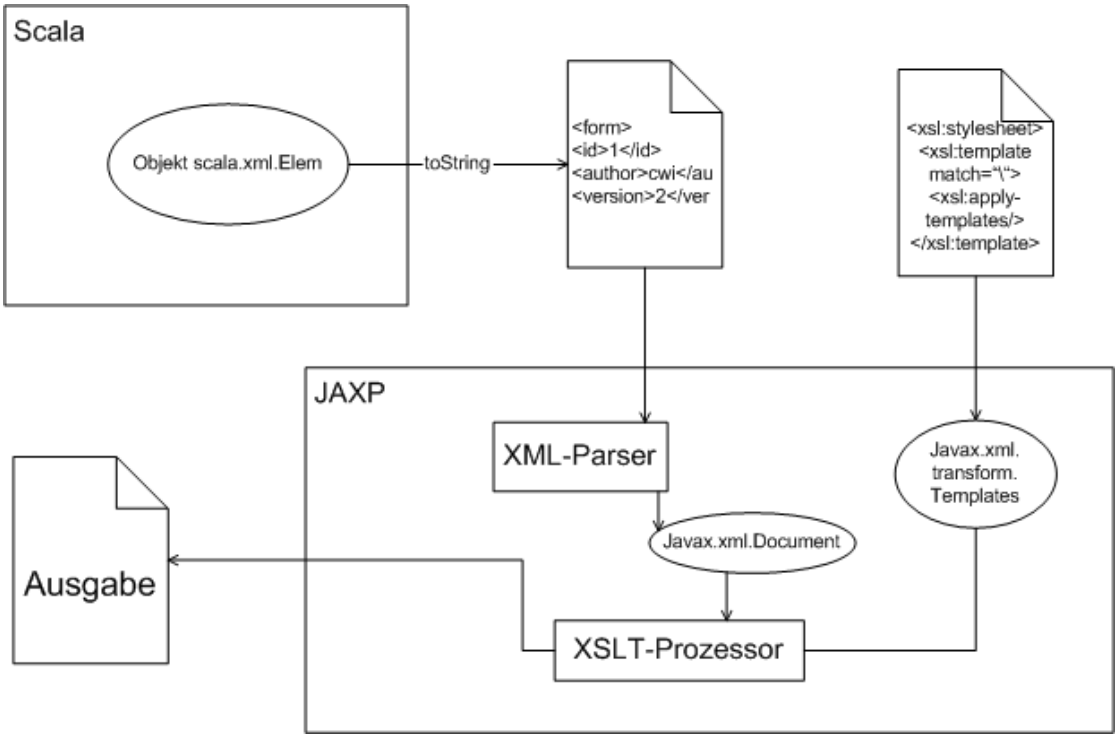

Bild 4.4: Verwendung von JAXP in Scala

Der Stern steht dabei als Platzhalter für die entsprechende Domain, mit der die Web–Anwendung SuSy! zu erreichen ist. Um die Ausgabe des XSLT–Prozessors zu steuern, wird der Programmausdruck im XSLT–Dokument gebraucht [8, Kapitel 1.17]:

```
<xsl:output method="html"/>
```

Die erzeugte HTML–Seite zeigt den Überblick einer Umfrage im Browser. (Das Stylesheet zum Erstellen der HTML–Seite findet man im Projektordner unter `public/xslt/susyxml2html.xsl`)

Das Erstellen von Play!–Templates ähnelt sehr dem Erstellen von XSLT–Dokumenten. Das erstellen von XSLT–Templates ähnlt dem Erstellen von Play!–Templates. Eine Aufgabe beider ist das Anzeigen von Daten in einem Browser. Dabei kann das Play!–Template nur serverseitig ausgeführt werden, während XSLT–Templates auch Clientseitig verwendet werden können. Dies macht XSLT zu einen starken Werkzeug in der Entwicklung von Web–Anwendungen.

Ein XSL—Template mit dem Schlüsselwort `xsl:template` beschreibt mit Regeln, wie das Ergebnis aussehen soll. Dabei werden nur Regeln gefeuert, wenn im XML–Baum ein entsprechender Knoten gefunden wird. Beim Verarbeiten der Fragen ist deshalb keine Verzweigung mit Hilfe des Fragetyps notwendig, trotzdem werden z.B. Auswahlfragen

anders verarbeitet als Matrix–Fragen. Der Programmausdruck 16 zeigt genau, wie elegant XSLT dieses Problem gelöst hat:

```
<xsl:template match="questions">
 <xsl:for-each select="question">
  <h3><xsl:value-of select="text"/></h3>
  <xsl:apply-templates select="choices"/>
  <xsl:apply-templates select="subquestions/sub"/>
  <xsl:apply-templates select="answers"/>
 </xsl:for-each>
</xsl:template>
```

Programmausdruck 16: XSL–Template für Question–Nodes

Beim Verarbeiten der Antworten wurde durch die Kontrollstruktur `xsl:if` im Template answer alle Antwortstrukturen unterschiedlich beschrieben.

Weitere Möglichkeiten mit XSLT, wie z.B. Session–Tracking ohne Cookies, die Browsererkennung und die Internationalisierung werden im Buch [9, Kapitel 8] genau vorgestellt.

4.2.2 XML nach CSV

Die Daten in einem XML–Dokument für genau eine Umfrage können in eine CSV–Repräsentation umgewandelt werden. Dabei gibt es einen Unterschied zu dem CSV–Export aus dem Projekt. Jeder Teilnehmer an der Umfrage ist eine Zeile in dem CSV–Dokument, und es werden nur Fragen ausgewertet, die zum Berechnen für Statistiken interessant sind. Jede Spalte gehört dann zu einer Antwort einer bestimmten Frage. Die Reihenfolge der Fragen ist abhängig von der Reihenfolge im XML–Dokument wie man am Beispiel Bild 4.5 sehen kann. Somit ist der XML–basierte Export nach CSV eine zusätzliche Funktion, die sich vom ursprünglichen CSV–Export unterscheidet.

Antworten zu Auswahl–Fragen	Antworten zu Ranking–Fragen	Antworten zu Line–Fragen
Frage–Identifikation + Auswahlidentifikation verbunden mit einem Punkt „.“	Frage–Identifikation + Auswahlidentifikation verbunden mit einem Punkt „.“	Frage–Identifiaktion + Fragenummer verbunden mit einem Punkt „.“

Bild 4.5: Aufbau der Identifikation in der CSV–Datei

Um die CSV–Darstellung zu erzeugen, muss als erstes der Kopf erstellt werden. Die XSL–Templates bekommen alle das `mode`–Attribut mit dem Wert `csvhead`. Die Reihenfolge von Frageknoten im XML–Dokument dient dabei als Reihenfolge für die Antworten im

Antwortgeber Identifikation	Antworten zu Auswahl–Fragen	Antworten zu Ranking–Fragen	Antworten zu Line–Fragen	Umfrage beendet
Eine Nummer	0 - nicht Ausgewählt 1 - Ausgewählt	der Rang	die Eingabe	true false

Bild 4.6: Aufbau der CSV–Datei

CSV–Dokument. In der ersten Zeile der CSV–Darstellung stehen die Frage– und Antwortidentifikationen. Nur der Befehl `xsl:sort` könnte die Reihenfolge für einen Prozessor ändern, ohne wird die Reihenfolge allein durch die feste Struktur des Input–Dokuments festgelegt.

Die Reihenfolge der Antwortgeber wird durch die Reihenfolge des Knoten `answerers` bestimmt. Es werden nur Elemente vom Typ `answerer` verarbeitet, wenn dieses Element einen Kindknoten mit dem Namen `complete` besitzt. Um unnötiges Verwenden von `xsl:if` zu vermeiden, werden die Fragen mit Hilfe von `xsl:choose` getrennt verarbeitet, dabei können bestimmte Fragetypen aussortiert werden.

Um alle Antworten eines bestimmten Antwortgebers zu filtern, wird XPath verwendet. Im Programmausdruck 17 kann man sehen, wie aus allen Antworten zu einer Frage nur die ausgewählt werden, die ein bestimmter Antwortgeber gegeben hat.

```
1 <xsl:with-param name="answer" select=
2 "answers/answer[answerer/id=$answerer/id]" />
```

Programmausdruck 17: XPath zum Filtern der Antworten

Eine Beispieltransformation zum Programmausdruck 15 ergibt den CSV–Inhalt der im Bild 4.7 zu sehen ist, im Kopf der CSV–Datei stehen die Frage–Identifikationen:

```
Answerer scq1.c1 scq1.c2 complete
1 0 1 true
2 0 1 true
```

Bild 4.7: Ergebnis einer Transformation nach CSV

4.2.3 Verwendung von XSLT bei Play!

Um die XML–Schnittstelle in Play! zu integrieren, muss das listSurveys.scala.html–Template erweitert werden mit:

```
<input type="submit" class="btn btn-success" name="xml2html"
  value="@Messages("listSurveys.export.xml2html.button.text")">
```

Die entsprechende Übersetzung muss in die Datei `messages` eingefügt werden. Die Dateien findet man im Ordner `conf`. Der Klick auf einen Button dieses Templates wird im Controller `EvalViewConnector` verarbeitet. Dort müssen die zusätzlichen Ereignisse eingetragen werden. Es ist notwendig, die `routes` zu ändern. Da die Funktionalität des Exports und der Auswertung bis auf die Generierung der http-Antwort identisch sind, kann man den vorhanden Controller aus der Projektarbeit weiter verwenden. Er wird mittels `match--case` und dem String `view` erweitert. In Bild 4.8 ist ein Screeshot mit der zusätzlichen Funktionalität zu sehen.

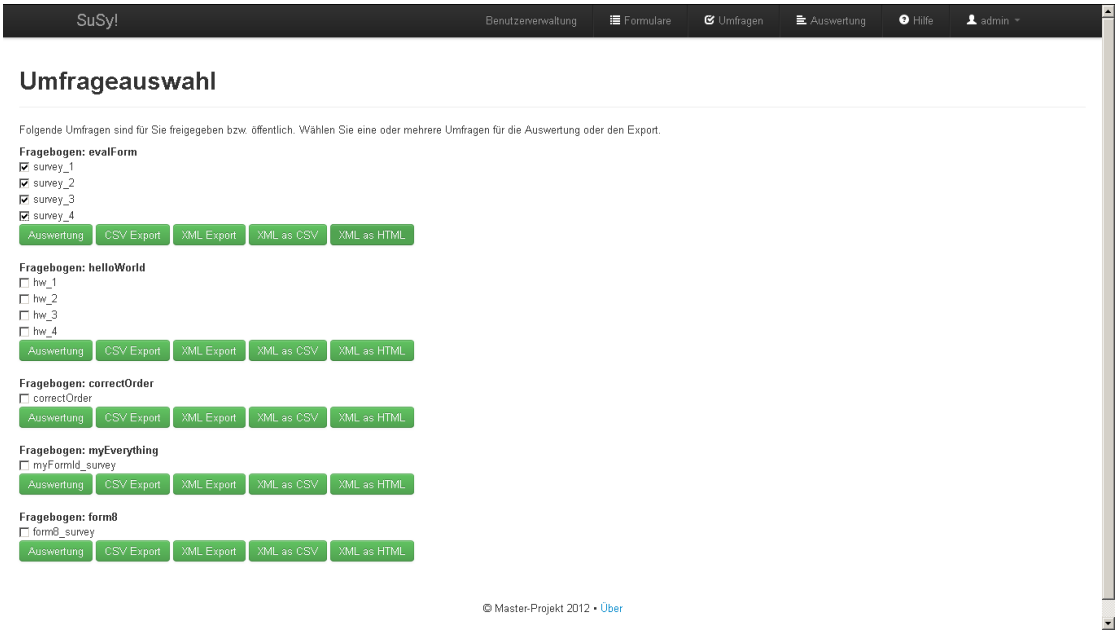

Bild 4.8: SuSy! erweitert mit XML und XSLT

5 Rückblick und Ausblick

Wir alle haben aus dem Projekt viel mitgenommen und sind auch ein wenig stolz auf unsere SuSy!. Hilfreich war das ausführliche Lastenheft, das uns die Aufgabenteilung und das gezielt Arbeiten erleichterte. Mit Redmine und git konnte man jederzeit den Fortschritt des Projektes betrachten und lenken.

Die Aufgaben aus dem Lastenheft wurden gemeinsam besprochen, und es wurden mögliche Lösungsansätze besprochen. Nachdem die Gruppen eingeteilt waren, wurden die Aufgaben gruppenintern weiter besprochen. Jede Gruppe hatte eigenverantwortlich eine Lösung erarbeitet, die beim Treffen dem Kurs dargestellt wurde.

Mit Play! eine Web–Anwendung zu programmieren hat sich als eine gute Entscheidung herausgestellt.

Mein spezieller Arbeitsauftrag ist zum Teil projektgebunden und zum anderen eine Ausweitung, in die ich meine persönliche Erfahrung einbringen konnte. Nicht zuletzt deshalb, weil ich am Institut für Tierzucht und Tierhaltung an der CAU an einem Projekt zur Simulation von Seuchen mit gearbeitet habe. Dort habe ich das System mit Hilfe von XML erweitert. Ein weiterer Grund ist das Buch [10] aus meiner Schulzeit, das damals mein Intersse an XML geweckt hat, und ich seitdem die Entwicklung von XML verfolge.

XML ist seit der Einführung eine sehr flexible Schnittstelle geworden, die den Einzug in viele Hochsprachen gefunden hat. Es hat sich gezeigt, dass Scala für die Verwendung von XML sehr gut geeignet ist, bietet es mit den Klassen im Paket `scala.xml.transform`[6] eine überzeugende Lösung zum Transformieren von XML an. Leider hilft Scala nicht weiter, wenn man schon XSLT–Dateien hat. Hier fehlt ein Werkzeug, das die fehlende Verbindung zwischen XSLT–Datei und Scala anbietet. Dieser „Schönheitsfehler“ wird z.T. mit Play! wieder behoben, wenn XML zusammen mit XSLT versendet wird.

Wie könnte man SuSy! mit Hilfe der implementierten XML–Schnittstelle erweitern? Mit Hilfe von XML Schema kann man das Importieren von Fragebögen und Umfragen anbieten. Eine Datensicherung, und das Austauschen von Umfragen zwischen verschiedenen SuSy!–Anwendungen wäre damit möglich.

Mit XSLT kann man SuSy!–XML beliebig transformieren und so für andere Programme aufarbeiten. Dafür muss man nicht im Scala–Code von SuSy! arbeiten, sondern kann unabhängig entwickeln. Eine weitere interessante Transformation wäre, einen Fragebogen in druckbarer Form (z.B. pdf) zu erstellen.

6 Web–Seiten

$u1$. http://www.scala-lang.org/

$u2$. http://www.playframework.org/

$u3$. http://www.redmine.org/

$u4$. http://www.w3.org/Style/CSS/

$u5$. http://twitter.github.com/bootstrap/

$u6$. www.eclipse.org

$u7$. http://www.scala-ide.org

$u8$. http://git-scm.com/

$u9$. http://www.h2database.com/html/main.html

$u10$. http://www.postgresql.org/

$u11$. http://www.zenexity.com

$u12$. http://www.scala-sbt.org/

$u13$. http://www.junit.org

$u14$. http://scalatest.org/

$u15$. http://etorreborre.github.com/specs2/

$u16$. http://treebeard.sourceforge.net/

7 Literatur

[1] Anatomie einer Play–Anwendung. http://www.playframework.org/documentation/2.0/Anatomy.

[2] Extensible Markup Language (XML). http://www.w3.org/XML/.

[3] Java^{TM} API for XML processing (JAXP). http://docs.oracle.com/javase/7/docs/technotes/guides/xml/jaxp/.

[4] Philosophie von Play. http://www.playframework.org/documentation/2.0.3/Philosophy.

[5] Play 2.0 API für Scala. http://www.playframework.org/documentation/api/2.0/scala/index.html.

[6] Scala Application Programming Interface. www.scala-lang.org/api/current/index.html.

[7] XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/.

[8] Frank Bongers. *XSLT 2.0.* Galileo Computing, 2004.

[9] Eric M. Burke. *Java und XSLT.* O'Reilly Verlag GmbH & Co. KG, 2002.

[10] O. Kürten F. Harms, D. Koch. *HTML & XML.* Data Becker, 2000.

[11] B. Venners M. Odersky, L. Spoon. *Programming in Scala.* artima, 2008.

[12] Prof. Dr. Thomas Walter. *Kompendium der Web-Programmierung.* Springer–Verlag Berlin Heidelberg, 2008. www.webkompendium.de.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin versichere ich, dass diese Arbeit noch nicht als Abschlussarbeit an anderer Stelle vorgelegen hat.

Kiel, den 26. September 2012

Unterschrift von Christian Wischmann