Studienarbeit

Entwicklung einer webbasierten Anwendung zur Planung des Masterstudiums mit Ruby on Rails

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion

angefertigt von: Thomas Kiupel
betreut von: Prof. Dr. Michael Hanus
M.Sc. Björn Peemöller

Kiel, 17. Juli 2013

Inhaltsverzeichnis

1 Aufgabenstellung

1.1 Anforderungen

Ziel der Arbeit ist die Planung und Umsetzung einer webbasierten Anwendung, die es Masterstudenten des Instituts für Informatik an der Christian-Albrechts-Universität zu Kiel erlaubt, ihr Studium online zu planen (*Studienplaner*). Die Entwicklung soll auf Basis des Web-Frameworks *Ruby on Rails* erfolgen.

Im Verlauf des Masterstudiums müssen in verschiedenen Schwerpunktbereichen (Kategorien) jeweils eine vorgegebene Anzahl an Leistungspunkten erreicht werden. Masterprogramme geben einen Themenschwerpunkt des Studiums durch Empfehlungen in jeder dieser Kategorien vor. Der Studienplaner erlaubt den Studierenden ein Masterprogramm zu wählen und auf dessen Basis festzulegen, welche Veranstaltungen in welchen Semestern zu besuchen sind, um das Studium in dem angestrebten Zeitraum von in der Regel 4 Semestern abschließen zu können. Das letzte Semester ist dabei dem Erstellen der Masterarbeit vorbehalten.

Motiviert ist der Studienplaner durch die Tatsache, dass die Verwaltung der Masterstudenten bisher nicht an zentraler Stelle erfolgt. Damit haben Dozenten keinen Überblick darüber, wie viele Teilnehmer in den von ihnen angebotenen Veranstaltungen zu erwarten sind. Dies macht insbesondere Streichungen und Verschiebungen von Angeboten schwierig, da diese Auswirkungen auf den Studienverlauf von Studierenden haben können, welche die Teilnahme an einer Veranstaltung in ihrem Studium vorgesehen haben. Die Anwendung soll es erlauben, solche Studierenden zu identifizieren, um durch Rücksprachen Veränderungen möglichst konfliktfrei durchführen zu können. Weiterhin liegen die Studienpläne leicht zugreifbar in einem einheitlichen Format vor, wodurch die Beratung von Studierenden erleichtert wird.

Eine Voraussetzung für eine solche Planung ist, dass bekannt ist, welche Module in welchen Semestern angeboten werden. Diese Informationen werden mindestens für die drei folgenden Semester durch die Moduldatenbank der Informatik[1] (im Folgenden nur “Moduldatenbank”) bereitgestellt. Folglich ist eine Kernforderung an den Studienplaner die Aufbereitung dieser Daten, so dass eine einfache und übersichtliche Planung des Studiums möglich ist.

Zusätzlich zu den Modulangeboten verwaltet die Moduldatenbank auch die angebotenen Masterprogramme. Entscheiden sich Studierende einem solchen Masterprogramm zu folgen, sollte der Studienplaner entsprechende Unterstützung bieten.

Da die Masterprüfungsordnung der Informatik zum einen viele Freiheiten im Studium bietet und zum anderen gelegentlichen Änderungen unterworfen ist, sollte die Anwendung möglichst flexibel sein, um alle studierbaren Kombinationen zu erlauben. Die Überprüfung der Konformität zur jeweils aktuellen Studienordnung fällt dem zuständigen Studienberater zu.

[1] Moduldatenbank der Informatik: http://www-ps.informatik.uni-kiel.de/~mh/studiengaenge/show.cgi.

1.2 Moduldatenbank der Informatik

Wesentlich für die Applikation ist die Schnittstelle zur Moduldatenbank der Informatik. Sie stellt Informationen zu allen angebotenen Modulen im Fach Informatik bereit. Die Daten gehen dabei über die im Vorlesungsverzeichnis verfügbaren Informationen hinaus. Konkret sind zu jedem Modul unter anderen die folgenden Einträge vorhanden:

- Eindeutiger Modulcode
- Deutscher und englischer Titel
- ECTS-Punkte
- Zugeordnete Studiengänge
- Zugeordnete Kategorien
- Semester, in denen dieses Modul angeboten wird

Eine weitere Funktion der Moduldatenbank ist die Verwaltung der Masterprogramme. Dozenten haben die Möglichkeit, angebotene und geplante Masterprogramme einzupflegen und Studierenden somit eine zentrale Übersicht der Möglichkeiten in ihrem Masterstudium anzubieten. Die wesentlichen Informationen zu jedem Programm sind:

- Eindeutige Id
- Titel
- Startsemester
- Betreuer (*research advisor*)
- Masterstudiengang
- eine Menge von Pflichtmodulen
- eine Menge von empfohlenen Modulen

Die dem Masterprogramm zugeordneten Module beziehen sich dabei auf angebotene Veranstaltungen in einem konkreten Semester. Dies stellt sicher, dass die Studierenden die Möglichkeit haben, das Studium in der Regelstudienzeit abzuschließen. Weiterhin beziehen sich diese Empfehlungen auf eine Kategorie, in der die erworbenen Leistungspunkte einzubringen sind. Dies ist notwendig, da ein Modul auch mehreren Kategorien zugeordnet sein kann.

Die obigen Daten werden über eine XML-Webschnittstelle bereitgestellt. Diese wird genutzt, um die Datenbank des Studienplaners zu füllen und aktuell zu halten (s. Synchronisation).

1.3 Übersicht der Entwicklung

Im Laufe der Planung zeigte sich schnell, dass die Daten der Moduldatenbank in einer internen Datenbank dupliziert werden müssen. So werden ungültige Änderungen an den Daten und daraus folgende Inkonsistenzen vermieden. Folglich musste ein Synchronisationsmechanismus entwickelt werden, der die Daten regelmäßig einliest, in das lokale Datenbankschema übersetzt und einpflegt. Für diesen Prozess ist es notwendig, die von der Moduldatenbank bereitgestellten XML-Daten zu parsen und zu validieren. Dabei ist eine gewisse Flexibilität gefordert, um die Auswirkungen auftretender Fehler zu begrenzen. Besonders fehlende oder fehlerhafte Daten in Feldern, die nicht zwingend notwendig sind, sollten nicht den gesamten Vorgang abbrechen. Zusätzlich ist ein Mechanismus notwendig, welcher die Synchronisation periodisch in einem Hintergrundprozess durchführt.

Es ist damit ein Datenbankschema erforderlich, welches die verwendeten Informationen aus der Moduldatenbank möglichst exakt abbildet und dabei Verbindungen zwischen den Datensätzen erhält. Dazu sind Klassen und zugehörige Datenbanktabellen zu Veranstaltungen (`Lecture`), Veranstaltungsangeboten (`LectureOffer`), Masterprogrammen (`StudyProgram`) und Kategorien (`LectureCategory`) sowie die zugehörigen Verknüpfungen dieser Datensätze untereinander notwendig. Besonders die Unterscheidung zwischen einer Veranstaltung und dem Angebot dieser in einem gegebenen Semester bestimmt dabei die Struktur des Datenmodells. Da Semester ein häufig vorkommendes Datum sind, musste für diese eine Datenstruktur zur Repräsentation entworfen werden.

Zusätzlich ergibt sich aus den Anforderungen direkt der Bedarf einer umfangreichen Benutzerverwaltung, da die Studienpläne Studierenden zugeordnet werden müssen. Damit sind mindestens zwei verschiedene Benutzerrollen im System erforderlich: Die Studierenden, die ihr Studium planen und sogenannte Berater, welche die Studienpläne zugeordneter Studierender einsehen und bearbeiten können.

Berater können somit die Studierenden direkt bei der Planung ihres Studiums unterstützen. Weiterhin erlaubt diese Rolle den Verantwortlichen, sich eine Übersicht über die Teilnehmer in den von ihnen angebotenen Masterprogrammen und Modulen zu verschaffen. Zusätzlich liefert die Trennung von Beratern und Studierenden eine Möglichkeit, die Sichtbarkeit der Studienpläne gezielt einzuschränken. Überdies sind Administratoren notwendig, welchen neben Verwaltungsaufgaben insbesondere die Zuordnung von Beraterrollen und der damit einhergehenden Rechte zukommt.

Es werden außerdem ergänzende Funktionen wie das Aktivieren von Benutzerkonten zur Validierung der E-Mail-Adresse sowie Unterstützung zum Zurücksetzen des Passwortes benötigt. Somit ist auch eine Funktionalität zum Generieren und Versenden der entsprechenden E-Mails vorzusehen.

Das Erstellen und Bearbeiten von Studienplänen selbst ist der Kern des Studienplaners. Hierbei sind eine Reihe von Kategorien und zugehörigen Veranstaltungsangeboten übersichtlich darzustellen und Änderungen zu ermöglichen. Eine klassische Implementierung nur über HTTP-Anfragen würde dabei eine unnötig zähe Bedienung zur Folge haben. Entsprechend wurde früh in der Entwicklung eine Umsetzung dieser Funktionalität in *JavaScript* (bzw. *CoffeeScript*) vorgesehen. Das Layout wurde in der durch *Ruby on Rails* bereitgestellten *CSS*-Erweiterung *SASS* realisiert.

Weiterhin wurde eine Schnittstelle implementiert, die es erlaubt, die Anzahl der Teilnehmer von Modulen sowie die Anzahl Studierender in einem Masterprogramm abzufragen. Aus Datenschutzgründen werden diese Informationen zwar ohne Zugangsbeschränkung, aber nur anonymisiert zur Verfügung gestellt.

2 Verwendete Technologien

2.1 Ruby on Rails

Ruby on Rails ist ein Webframework basierend auf der namensgebenden Skriptsprache *Ruby*. Um eine schnelle und homogene Entwicklung neuer Applikationen zu ermöglichen, verfolgen die Entwickler das Motto *Konvention vor Konfiguration*. Zusammen mit der strengen Nutzung des *MVC*-Patterns und der Flexibilität von *Ruby* erlaubt dieses, viele Anwendungsfälle in sehr kompaktem Code umzusetzen.

Die Basis einer *Ruby on Rails* Anwendung bilden typischerweise die *Model*-Klassen. Sie repräsentieren die in der Anwendung verwendeten Datensätze. Solche Klassen werden in *Ruby on Rails* durch das *ActiveRecord*-Pattern unterstützt. Dieses stellt eine objektrelationale Abbildung zur Verfügung und verknüpft Datenbanktabellen direkt mit *Ruby*-Klassen. Dadurch stehen komfortable Funktionen zum Auslesen, Erstellen und Ändern von Einträgen bereit. Dabei werden eine Reihe von gängigen Datenbanksystemen unterstützt.

Controller einer *Ruby on Rails* Anwendung interpretieren die HTTP-Anfrage und setzen *Model* und *View* Klassen in Beziehung zueinander. Zusätzlich nehmen sie Benutzereingaben entgegen und übergeben diese an die zuständigen *Model*-Instanzen. Sie sind in der Regel schlank und folgen einem festen Schema. Dies folgt vor allem aus der Tatsache, dass die Validierung der meisten Benutzereingaben durch die *Model*-Klassen vorgenommen wird.

Die Zuordnung von Anfragen zu den entsprechenden *Controllern* geschieht durch die Routing-Tabelle. Wegen der starken Fixierung auf Datensätze wird dabei zwischen den Methoden `SHOW`, `EDIT`, `UPDATE`, `NEW`, `DESTROY` und `INDEX` unterschieden. Das Framework stellt Methoden bereit, um diese durch entsprechende `GET`- bzw. `POST`-Anfragen zu emulieren.

Zur einfachen Generierung von Dokumenten stellt *Ruby on Rails* den Mechanismus des *Embedded Ruby* (*erb*) bereit. Dies bildet den Kern der *Views* und erlaubt es (ähnlich zu *PHP*), Text und *Ruby*-Code in einer Datei zu kombinieren. In der Regel wird es dazu verwendet, HTML-Dokumente zu erzeugen; aber auch *JavaScript*-Code, XML oder reiner Text lassen sich mit diesem Mechanismus erstellen.

Die Applikation wurde in *Ruby on Rails* Version 3.2 entwickelt. Als Datenbank wurde *SQLite* in der Version 3 verwendet. Da für fast alle Anfragen an die Datenbank die von *Ruby on Rails* bereitgestellte Schnittstelle verwendet wurde, sollte eine Umstellung auf einen anderen Datenbanktreiber unproblematisch möglich sein. Für die Entwicklung wurde *Ruby* 1.8.7 statt den neueren Versionen 1.9.x bzw. 2.0 verwendet, da zum Zeitpunkt der Entwicklung kein Webserver mit höherer Version zur Verfügung stand.

2.2 SASS

Ruby on Rails unterstützt *SASS*[1] (*Syntactically Awesome Stylesheets*), eine *CSS*-Metasprache. *SASS*-Code wird in *CSS* übersetzt und erlaubt es, einige Strukturierungsmechanismen anzuwenden, die reines *CSS* nicht bietet. Dabei ist *SASS* eine echte Obermenge von *CSS*. Die folgenden Erweiterungen werden bereitgestellt

- **Variablen** erlauben die Verwendung von einmalig definierten Farb- und Zahlendefinitionen an verschiedenen Stellen. Somit ist es möglich, viele auftretende Änderungen lokal zu begrenzen.

```
$gray: #AAAAAA;

body {
  background-color: $gray;
}
```

- **Geschachtelte Regeln** dienen insbesondere der Übersicht, da viele potenziell komplexe Selektoren vermieden werden bzw. nur noch an wenigen Stellen auftreten. Viele Änderungen an Klassennamen oder der Dokumentstruktur können so einfacher in die Stylesheets übernommen werden.

```
ul.navigation {
  li {
    color: #0000FF;
  }
}
```

- **Mixins** vermeiden - ähnlich wie die Variablen - die Wiederholung gleichartiger Ausdrücke, allerdings auf Regelebene. Durch die Angabe von Parametern können Eigenschaften gleichartiger Elemente an zentraler Stelle definiert werden.

```
@mixin border($color) {
  border: 1px solid $color;
}

div.message {
  @include border(#0000FF);
}
```

- **Vererbung** liefert ein ähnliches Konzept wie *Mixins*. Sie erlaubt es, Elemente zu spezialisieren, ohne dass diese eine gemeinsame Klasse mit dem beerbten Objekt teilen müssen. Dies verschiebt diese Informationen aus dem Dokument in die *CSS*-Definitionen.

```
.button {
  font-weight: bold;
  border: 1px solid #000000;
}

.button-ok {
```

[1] Offizielle *SASS*-Seite: http://sass-lang.com.

```
  @extend .button
  background-color: #00FF00;
}
```

- **Funktionen** erlauben es, Werte miteinander zu kombinieren. *SASS* stellt sowohl einfache Arithmetik (`+`, `-`, `*`, `/`, `min`, `max`, ...) auf Zahlenwerten, wie auch Funktionen zum Manipulieren von Farbwerten und Zeichenketten zur Verfügung. Es ist ebenfalls möglich, eigene *Ruby*-Funktionen einzubinden.

  ```
  $padding: 10px;

  @mixin button($color) {
    color: $color;
    background-color: lighten($color,50%);
    padding: $padding / 2;
  }
  ```

  Die Möglichkeiten der Kombination von Zahlenwerten sind allerdings dadurch beschränkt, dass sie in statisches *CSS* übersetzt wird. Daher ist die Kombination von Werten mit verschiedenen Einheiten nicht möglich (z. B. `10px + 2em`). Sie ersezten daher nicht vollständig die in *CSS3* definierte Funktion `calc()`[2].

Im Gegensatz zu *CSS* bietet *SASS* die Möglichkeit, den Code auf mehrere Dateien zu verteilen. Sie werden im Übersetzungsvorgang zu einer Datei zusammengefasst. Dies erlaubt es, die Anzahl notwendiger HTTP-Anfragen beim Aufruf einer Seite zu reduzieren. Allerdings bietet die Asset-Pipeline von *Ruby on Rails* einen ähnlichen Mechanismus.

Da es einen Übersetzungsvorgang gibt, werden zusätzlich einige syntaktische Fehler in den Stylesheet-Dateien frühzeitig erkannt.

2.3 CoffeeScript

CoffeeScript[3] ist eine von Jeremy Ashkenas entwickelte Skriptsprache, die in *JavaScript* übersetzt wird. Sie besitzt eine Syntax, die von modernen Sprachen wie *Ruby* und *Haskell* beeinflusst wurde.

Die Unterschiede von *CoffeeScript* gegenüber *JavaScript* sind dabei hauptsächlich syntaktischer Natur. Es ist eine imperative, objektorientierte und dynamisch getypte Sprache. Wie *JavaScript* bietet es Funktionen höherer Ordnung.

Funktionen werden durch den `->` Operator erstellt. Analog zu *Ruby* liefert das Ergebnis der letzten ausgeführten Anweisung in einer Funktion implizit den Rückgabewert.

```
odd = (n) ->
  if n % 2 == 1
    true
  else
    false
```

[2] Candidate Recommendation "CSS Values and Units Module Level 3": http://www.w3.org/TR/css3-values/#calc.
[3] Offizielle CoffeeScript-Seite: http://coffeescript.org.

Zuweisungen, Listen und Objekte haben eine *JavaScript*-ähnliche Syntax. Dabei werden Blöcke durch entsprechende Einrückungen zusammengefasst und das Trennen von Anweisungen durch ein Semikolon ist optional, falls sie durch einen Zeilenumbruch getrennt sind.

```
x = 42
list = [1,2,3,4]

object =
  value1: 42
  value2: [1,2,3,4]
```

Das Objektmodell bleibt prototyp-basiert. Allerdings bietet *CoffeeScript* Schlüsselwörter, die den Aufbau einer Objekthierarchie vereinfachen.

```
class Greeter
  constructor: (@subject) ->

  greet: ->
    alert "Hello #{@subject}!"

class WorldGreeter extends Greeter
  constructor: ->
    super "World"
```

3 Übersicht über das Benutzerinterface

3.1 Erstellen und Bearbeiten eines Studienplans

3.1.1 Masterprogramm

Bevor Veranstaltungen hinzugefügt werden können, muss ein Masterprogramm gewählt werden. Die verfügbaren Programme sind in einem Auswahlmenü aufgeführt. Die Wahl muss mit einem Klick auf *Studienprogramm speichern* bestätigt werden.

Jedes Programm startet in einem vorgegebenen Semester. Da sich die zu belegenden Veranstaltungen mit dem Startsemester ändern können, sollte dieses mit dem Beginn des Masterstudiums übereinstimmen. Wird das gewünschte Programm in diesem Semester nicht angeboten, ist eine Rücksprache mit dem zuständigen *research advisor* notwendig.

Das Masterprogramm lässt sich jederzeit ändern. Gewählte Veranstaltungen werden - soweit möglich - übernommen (s. Studienpläne).

3.1.2 Berater

Die Wahl der Berater erfolgt über das zugeordnete Feld. Die so festgelegten Berater können den Studienplan der Studierenden einsehen und bearbeiten. In den meisten Fällen ist es sinnvoll, zumindest dem gewählten *research advisor* Zugriff zu gewähren, um zukünftige Absprachen zu erleichtern.

Eine Mehrfachauswahl ist in den meisten Browsern durch Halten der Taste `Strg` möglich.

3.1.3 Veranstaltungen

Nach Bestätigung der Wahl des Masterprogrammes werden die zugeordneten Veranstaltungskategorien angezeigt. Diese können je nach gewähltem Masterprogramm variieren. Sie lassen sich mit einem Klick auf den Titel ein- bzw. ausklappen.

Die rechte Seite enthält die in dieser Kategorie verfügbaren Veranstaltungsangebote nach Semester gruppiert. Mit einem Klick auf ein weiß hinterlegtes Semester werden diese angezeigt. Ein Klick auf die Schaltfläche links neben dem Veranstaltungsnamen fügt diese zu dem aktuellen Studienplan in dieser Kategorie hinzu.

Alle bisher in dieser Kategorie hinzugefügten Veranstaltungen werden auf der linken Seite angezeigt. Durch einen Klick auf die zugehörige Schaltfläche können sie aus dem Studienplan entfernt werden.

Jede Veranstaltung darf im gesamten Studienplan nur einmal vorhanden sein. Ist ein Veranstaltungsname ausgegraut, so ist diese Veranstaltung in einer anderen Kategorie und/oder einem anderen Semester bereits ausgewählt.

Abbildung 3.1: Ansicht zur Planung des Studienplans in einer Kategorie

Dozenten können bei der Erstellung des Masterprogramms Veranstaltungen sowohl empfehlen als auch als verpflichtend festlegen. Diese sind durch blaue bzw. grüne Textfarbe markiert. Solche Markierungen beziehen sich immer auf ein Angebot in einem konkreten Semester und auf eine Kategorie. Abweichungen von den Pflichtveranstaltungen werden durch den Studienplaner erlaubt, sollten allerdings nicht ohne Rücksprache mit dem zuständigen *research advisor* in den Studienplan aufgenommen werden.

Alle Änderungen werden erst durch einen Klick auf *Studienplan speichern* übernommen. Auf der Übersichtsseite werden die Veranstaltungen und die Summe der Leistungspunkte sowohl nach Semester als auch nach Kategorie sortiert angezeigt. Die insgesamt notwendigen ECTS-Punkte werden durch die jeweils aktuelle Studienordnung festgelegt.

3.2 Beraterschnittstelle

Die Startseite der Berater enthält eine Liste der von ihnen betreuten Studierenden. Neben dem Namen und der E-Mail-Adresse wird das belegte Masterprogramm angezeigt.

Die entsprechende Schaltfläche rechts in der Spalte führt auf den Studienplan des jeweiligen Studierenden. Die Ansicht ist dabei mit der des Besitzers identisch. Die Studienpläne können ebenfalls bearbeitet werden. Allerdings ist eine Änderung der zugeordneten Berater nicht möglich.

Berater haben zudem Zugriff auf die Übersichtsseiten zu den Veranstaltungen und Masterprogrammen. Auf diesen werden die in der Datenbank abgelegten Informationen (insbesondere die Anzahl der teilnehmenden Studierenden) dargestellt.

3.3 Benutzerverwaltung

Der erste Schritt für neue Benutzer ist die Registrierung. Dazu ist sowohl die Angabe des Namens als auch einer E-Mail-Adresse sowie eines Passwortes notwendig.

Nach der Registrierung ist vorerst keine Anmeldung möglich. Um die Gültigkeit der E-Mail Adresse festzustellen wird nach dem Bestätigen der Registrierungsdaten automatisch eine Nachricht an die angegebene E-Mail Adresse gesendet. Diese enthält einen Link, dessen Aufruf das Benutzerkonto freischaltet und somit das Anmelden ermöglicht.

Die Seite *Mein Konto* ermöglicht es, das Benutzerpasswort zu ändern oder das Konto vollständig zu löschen.

Unter der Anmeldemaske wird eine Schaltfläche *Passwort vergessen* angezeigt. Diese führt zu einem Formular, welches nach Eingabe einer E-Mail-Adresse einen Link an diese versendet, dessen Aufruf das Setzen eines neuen Passwortes für den zugeordneten Benutzer ermöglicht.

Alle Benutzer werden nach der Anmeldung als Studierende geführt. Die Startseite ist somit der (zu diesem Zeitpunkt leere) Studienplan.

Die Beraterrolle muss durch einen Administrator explizit zugewiesen werden.

3.4 Webschnittstelle

Eine Anforderung an den Studienplaner ist, Daten zu den Teilnehmerzahlen in Veranstaltungen und Masterprogrammen über eine Schnittstelle verfügbar zu machen. Dies ist über die Basis-URL `/student_count` möglich.

Veranstaltungsangebote können über den Parameter `offers` abgefragt werden. Da sich der zugeordnete Code auf ein Modul und nicht auf ein Semester bezieht, muss dieses zusätzlich angegeben werden. Semester und Modulcode werden durch einen Doppelpunkt getrennt. Somit könnte z. B. durch Aufruf `/student_count?offers=ss13:MOD01` die Anzahl von Studierenden im Modul mit dem Code `MOD01` im Sommersemester 2013 abgefragt werden.

Mehrere Anfragen können durch `+` zusammengefasst werden. Die Übergabe der Parameter `offers=ss13:MOD01+ss14:MOD02` würde also `MOD01` im Sommersemester 2013 und `MOD02` im Sommersemester 2014 abfragen.

Die Antwort auf diese Anfrage ist ein Text-Dokument, welches die Zahlen durch Kommata getrennt in der Reihenfolge enthält, in der die Code/Semester Kombinationen in der URL auftreten.

Dabei ist es auch möglich mehrere Semester mit mehreren Modulen zu kombinieren. Dies geschieht durch Zusammenfassen mehrerer Semester bzw. Module mit Kommata. Zurückgegeben werden dann die Zahlen zu allen Kombinationen aus Semester und Modul.

Bei Kombinationen von mehreren Modulcodes mit mehreren Semestern werden die Anfragen zuerst nach Semester gruppiert. Dies bedeutet, dass `offers=ss12,ss13:MOD01,MOD02` äquivalent ist zu `offers=ss12:MOD01,MOD02+ss13:MOD01,MOD02`.

Ist eine Modul/Semester-Kombination nicht vorhanden, so wird für den entsprechenden Eintrag '`-1`' ausgegeben. Hat die Anfrage ein ungültiges Format, so wird eine leere Antwort zurückgegeben. Der HTTP Antwort-Code ist in diesem Fall 400.

Die Semester können wie im Beispiel im Format `ssyy` bzw. `wsyy` angegeben werden. Die Schreibweise `wsyy/yy` und vierstellige Jahreszahlen sind ebenfalls möglich. Leerzeichen sind allerdings unzulässig. Da die Methode `Semester.parse` verwendet wird, sind auch alternative Formate erlaubt. Diese werden im Kapitel Repräsentation von Semestern beschrieben.

Die Abfrage der Anzahl Studierender in einem Masterprogramm ist über den Parameter `programs` möglich. Da der Code des Masterprogramms in diesem Fall das Startsemester bestimmt, ist das Format deutlich einfacher; es wird einfach eine Liste von Codes getrennt durch `+` oder `,` übergeben. Zum Beispiel also `/student_count?programs=MasterProgram1,MasterProgram2`

Antwortformat und Fehlerbehandlung sind vollständig analog zu denen der Module.

4 Implementierung

4.1 Datenmodell

Durch die enge Verknüpfung mit der Datenbank wird der Großteil der Anwendungslogik durch die *Model*-Klassen realisiert. Charakteristisch sind insbesondere die Zusammenhänge zwischen den Daten. *Ruby on Rails* bietet eine Reihe von Makros wie `has_many` und `belongs_to`, um diese durch objektrelationale Abbildungen direkt in die Datenstrukturen zu übernehmen.

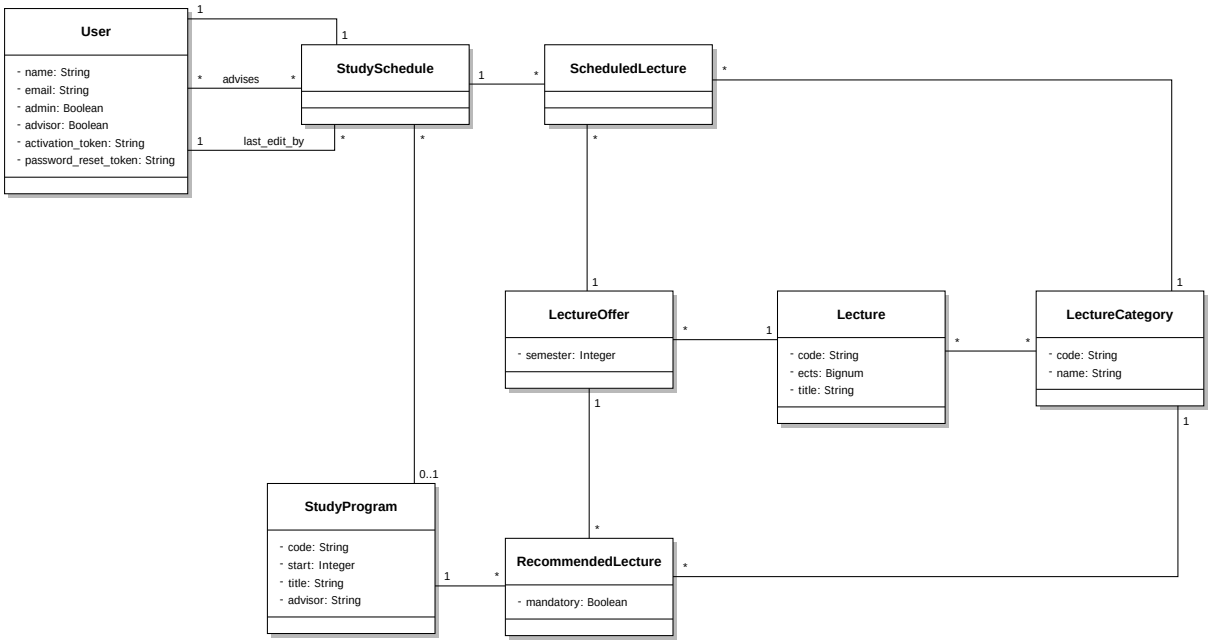

Abbildung 4.1: Datenmodell. Jede abgebildete Klasse entspricht sowohl einer Datenbanktabelle als auch einer *ActiveRecord*-Klasse in *Ruby on Rails*. `many-to-many` Beziehungen zwischen den Klassen benötigen jeweils eine eigene Relation. Diese sind nicht explizit aufgeführt.

Neben den Verknüpfungen der Daten ist die Validierung ein Kernaspekt der *Model*-Klassen. Alle Felder einer solchen Klasse können mit einem oder mehreren *Constraints* versehen werden. Dies ist zum einen notwendig, da *Ruby* eine schwach typisierte Sprache ist und somit Werte in Feldern ausgeschlossen werden können, die nicht sinnvoll in den Typ des darunterliegenden Datenbankfeldes konvertierbar sind. Allerdings ist es ebenfalls möglich, beliebige Prädikate zur Validierung zu verwenden. Diese können Fehlerbeschreibungen liefern, welche über die zu validierende Instanz verfügbar sind. Damit ist es möglich, Formulareingaben direkt an eine *Model*-Instanz zu übergeben und präzise Fehlermeldungen auszugeben, falls die Daten unzulässig sind.

Leider sind die so formulierten *Constraints* unabhängig von denen der darunterliegenden Daten-

bank. Die Beschränkungen auf Datenbankebene bilden in der Regel nur einen Teil der in den Klassen formulierten Prädikate ab. *Ruby on Rails* direkt bietet nur sehr wenige Optionen zur Validierung auf dieser Ebene, da jede Funktionalität durch alle unterstützten Datenbanksysteme zur Verfügung gestellt werden muss.

Dies ist von Bedeutung, da webbasierte Applikationen inhärent parallel sind und insbesondere die Gültigkeit von Referenzen letztlich nur durch das Datenbanksystem sichergestellt werden kann. Relevant wird dies in dieser Applikation allerdings nur bei der Synchronisation mit der Moduldatenbank, da durch die Benutzer nur Einträge vom Typ `ScheduledLecture` gelöscht werden können. Einträge dieses Typs werden im Schema nicht referenziert, da alle entsprechenden Verknüpfungen (zu `StudySchedule`, `StudyCategry` und `LectureOffer`) durch Fremdschlüssel in der Tabelle `scheduled_lectures` realisiert wurden.

4.1.1 Repräsentation von Semestern

Ein in der gesamten Applikation oft auftretendes Datum sind Semester. Eine Möglichkeit der Repräsentation wären Zeichenketten der Form '`WS yy/yy`' bzw. '`SS yy`'. Diese wären zwar gut lesbar und könnten ohne weitere Formatierungen ausgegeben werden, allerdings wäre die Implementierung einer Ordnung aufwändig und fehleranfällig. Und selbst bei einer vorliegenden Implementierung, könnte man diese auf Datenbankebene nicht nutzen.

Daher liegt es nahe, jedes Semester durch eine Zahl zu repräsentieren. Eine Möglichkeit wäre die Aufzählung bezogen auf einen festlegten Zeitpunkt. Definiert man beispielsweise das Sommersemester 2000 als `0`, so würde das Wintersemester 2000/2001 durch `1`, das Sommersemester 2001 durch `2` usw. repräsentiert. Durch diese Methode wäre es durch Addition und Subtraktion einfach möglich, das vorhergehende oder nachfolgende Semester zu bestimmen. Auch die Ordnung wäre durch den Vergleich von ganzen Zahlen direkt festgelegt und sogar in der Datenbank verfügbar.

Im Studienplaner werden die Semester allerdings durch eine in der Regel fünfstellige Zahl der Form `yyyys` repräsentiert. Dabei entsprechen die ersten vier Ziffern dem Jahr und die letzte Ziffer dem Semester; `0` für das Sommersemester dieses Jahres und `1` für das Wintersemester, welches in diesem Jahr beginnt. Somit wird beispielsweise das Sommersemester 2000 durch die Zahl `20000` und das Wintersemester 2000/01 durch `20001` beschrieben. Dieses Format hat den Vorteil, dass es intuitiv lesbar ist. Als Ordnungsfunktion kann ebenfalls der Vergleich auf ganzen Zahlen verwendet werden. Dadurch, dass der `Integer`-Wertebereich nicht vollständig genutzt wird, ist zur Bestimmung des nachfolgenden bzw. vorhergehenden Semesters allerdings eine einfache Fallunterscheidung notwendig.

Durch dieses Format wäre es relativ einfach möglich den Wertebereich zu erweitern, um beispielsweise Frühjahrs- und Herbstsemester zu unterstützen.

Die Verwendung dieser Repräsentation erfolgt nicht über eine eigene Klasse, sondern durch das *Ruby*-Modul `Semester` in `lib/`, welches eine Reihe von Hilfsfunktionen bereitstellt.

Das nachfolgende Semester wird beispielsweise durch die Funktion `Semester.following` bereitgestellt

```
def following(s)
  if s % 2 == 0
    s + 1
  else
    s + 9
```

```
  end
end
```

Eine in diesem Zusammenhang wichtige Funktionalität ist das Parsen von Zeichenketten in einen Semester-Code. Das Modul `Semester` unterstützt durch die Funktion `parse` mehrere Formate, die durch reguläre Ausdrücke gelesen werden. Dabei wird neben einem validen Zahlencode als Zeichenkette auch eine Kurzform `yys` unterstützt, bei der implizit angenommen wird, dass der Jahreszahl `20` vorzustellen ist. Weiterhin erlaubt diese Funktion, Zeichenketten der Form '`SS yy`', '`WS yy/yy`' und einige Varianten davon (z. B. 4-stellige Jahreszahlen und fehlendes Folgejahr bei Wintersemestern) einzulesen.

Die Ausgabe als lesbare Zeichenkette ist durch die Funktion `string` möglich. Diese ist ebenfalls durch eine einfache Fallunterscheidung umgesetzt.

```
def string(code)
  year = code / 10
  sem  = code % 2

  if sem == 0
    "SS #{year}"
  else
    next_year = (year+1)%100;
    "WS #{year}/#{next_year.to_s.rjust(2,"0")}"
  end
end
```

4.1.2 Verwaltung von Studienmodulen (`Lecture`/`LectureOffer`)

Die Klassen `Lecture` bzw. `LectureOffer` repräsentieren angebotene Module. Die Daten werden über eine XML-Schnittstelle aus der Moduldatenbank übernommen (s. Synchronisation). Die Klasse `Lecture` enthält dabei alle wesentlichen Informationen zu einem gegebenen Modul. Dies sind der Titel (`title`), die zugeordneten ECTS-Punkte (`ects`) und der Modulcode (`code`).

Da die ECTS-Punkte nicht notwendigerweise ganzzahlig sind, werden sie intern durch die *Ruby*-Klasse `BigDecimal` repräsentiert und mit einer Genauigkeit von zwei (Dezimal-)Stellen gespeichert. Dies wird von *Ruby on Rails* durch die Angabe des Typs `decimal` im Datenbankschema unterstützt.

Der Modulcode ist von Typ `String` und wird gespeichert, um die Beziehung der Daten zu der Moduldatenbank zu erhalten. Intern wird er als (Sekundär-)Schlüssel geführt. Als Primärschlüssel werden trotzdem die von *Ruby on Rails* automatisch erzeugten `Integer`-Schlüssel verwendet.

Zusätzlich zu den oben aufgeführten Attributen ist jedes Modul einer Menge von Kategorien zugeordnet. Diese Informationen werden ebenfalls durch die Moduldatenbank bereitgestellt.

Die Klasse `LectureOffer` repräsentiert ein Angebot eines durch `Lecture` spezifizierten Moduls in einem konkreten Semester. Sie enthält neben einer Referenz auf eine `Lecture` daher einen Semestercode (`semester`) im oben beschriebenen Format.

Die zugehörigen *Controller* (`LecturesController` und `LectureOffersController`) stellen nur Methoden zur Einsicht in die gespeicherten Daten bereit, da die Daten aus der Moduldatenbank übernommen werden und Änderungen durch Benutzer des Studienplaners nicht sinnvoll sind. Eine

der Methoden ist `index` im `LecturesController`, wodurch eine Auflistung aller Veranstaltungen einschließlich der zugeordneten Angebote generiert wird. Von dort aus sind Detailinformationen abrufbar, die durch die Methode `show` im `LectureOffersController` generiert werden.

4.1.3 Verwaltung von Masterprogrammen (`StudyProgram`)

Ein Eintrag in der Tabelle `StudyProgram` repräsentiert ein angebotenes Masterprogramm. Eine Eingabemaske zum Erstellen eines solchen Programms wird Dozenten durch die Moduldatenbank bereitgestellt. Eine Liste der Programme ist ebenfalls über die XML-Schnittstelle verfügbar.

Analog zu `Lecture` enthält auch `StudyProgram` ein Datum `code`, welches dieses außerhalb der Applikation eindeutig identifiziert. Das Feld `start` gibt an, in welchem Semester dieses Programm beginnt. Der Inhalt ist folglich eine ganze Zahl in dem oben beschriebenen Format.

Jedes `StudyProgram` besitzt eine Reihe von `RecommendedLecture`s. Diese Klasse hat keine eigenen Felder, sondern setzt nur jeweils ein `StudyProgram` mit einer `LectureCategory` und einem `LectureOffer` in Beziehung. Eine explizite Definition als *Model*-Klasse ist notwendig, da *Ruby on Rails* keine Assoziationen zwischen mehr als zwei Modellen unterstützt.

Die `RecommendedLecture`s definieren implizit durch die referenzierten Instanzen von `LectureCategory`, in welche Kategorien im zugehörigen Programm Veranstaltungen belegt werden können. Alternativ könnten alle vorhandenen Kategorien für jedes Masterprogramm verwendet werden. Allerdings sind in der Moduldatenbank eine Vielzahl von für die meisten Programme irrelevanten Kategorien vorhanden. Dies resultiert vor allem daraus, dass auch solche aus dem Bachelorstudiengang aufgeführt sind. Weiterhin ist die Kategorisierung von Veranstaltungen vielen Änderungen unterworfen. Dadurch existieren eine Menge von Kategorien, die für neuere Programme nicht von Belang sind.

Eine von den Empfehlungen unabhängige Liste von Kategorien für jedes Programm wäre im Datenmodell zwar problemlos zu ergänzen, allerdings wird ein solcher Zusammenhang durch die Moduldatenbank nicht bereitgestellt.

Der zugeordnete *Controller* (`StudyProgramsController`) implementiert die Methoden `index` und `show`. Diese erlauben die in der Datenbank gespeicherten Informationen einzusehen. Analog zum `LecturesController` ist auch hier die Bearbeitung durch Benutzer nicht sinnvoll.

4.1.4 Verwaltung von Studienplänen (`StudySchedule`)

Die Klasse `StudySchedule` bildet den Kern der Applikation. Sie fasst für einen Benutzer eine Menge von Veranstaltungen (`LectureOffer`) in Kategorien (`LectureCategory`) und ein Masterprogramm (`StudyProgram`) zusammen. Damit repräsentiert sie den geplanten Verlauf des Masterstudiums.

Jeder Benutzer ist mit einer Instanz von `StudySchedule` verknüpft. Die zugehörigen Seiten werden durch den `StudySchedulesController` in den Methoden `view`, `edit` und `update` bereitgestellt. Sie bilden für Studierende die primären Interaktionsmöglichkeiten mit dem System.

Berater (s. Benutzerrollen) haben im System die Möglichkeit, die Studienpläne der von ihnen betreuten Studierenden einzusehen und auch zu bearbeiten. Um Änderungen nachvollziehen zu können besitzt jeder `StudySchedule` ein Feld `last_edit_by`, welches auf einen Benutzer verweist und zusammen mit dem durch das *Ruby on Rails* Makro `timestamps` erzeugten Feld

`updated_at` verwendet wird, um Informationen über die letzte Änderung an einem Studienplan anzuzeigen.

Analog zu den `RecommendedLectures` ist eine in einem Studienplan enthaltene Veranstaltung eine Relation zwischen `StudySchedule`, `LectureCategory` und `LectureOffer`. Daher ist dazu die Klasse `ScheduledLecture` notwendig, die eine Beziehung zwischen drei solchen Objekten herstellt.

Validierung

Eine wesentliche Einschränkung der Studienpläne ist, dass jede Veranstaltung (`Lecture`) nur einmal eingebracht werden kann. Ansonsten könnten die ECTS-Punkte durch Aufnahme in mehrere Kategorien mehrfach gezählt werden. Da jedoch jede Veranstaltung in mehreren Semestern angeboten werden könnte (repräsentiert durch `LectureOffer`), ist ein einfaches Schlüssel-*Constraint* auf `lecture_offer_id` und `study_schedule_id` in `ScheduledLecture` unzureichend, da nur `lecture_id` statt `lecture_offer_id` die Einschränkung repräsentiert. Um auf diese zuzugreifen, muss dem Verweis auf `Lecture` in `LectureOffer` gefolgt werden.

Die naheliegende Lösung wäre, ein einfaches *Constraint* in `ScheduledLecture` einzufügen, welches sicherstellt, dass keine Instanz erstellt werden kann, die dieses Kriterium verletzt. Leider ist auch diese Lösung (neben den potenziellen Laufzeitproblemen) ungeeignet, da Wechselwirkungen mit dem `StudySchedule` vorhanden sind. Dies ist der Fall, da durch das Editieren eines `StudySchedule` regelmäßig eine Menge von `ScheduledLectures` erzeugt werden. Dies geschieht vorerst über die Methode `scheduled_lecture.build` und nicht `scheduled_lecture.create`. Der Unterschied zwischen den Aufrufen ist, dass eine durch `build` erzeugte Instanz erst an die Datenbank übergeben wird, wenn das Elternobjekt (in diesem Fall `StudySchedule`) durch einen Aufruf von `save` gespeichert wird. Dies ist geboten, da durch den in *Ruby on Rails* typischen Kontrollfluss neue Instanzen von `ScheduledLecture` erzeugt würden bevor sichergestellt ist, dass die Aktion gültig war.

```
1  validate :lecture_uniqueness
2
3  def lecture_uniqueness
4    current_lecture_ids = scheduled_lectures  \
5                          .select { |sl| not sl.marked_for_destruction? } \
6                          .map { |sl| sl.lecture_offer.lecture_id }
7    if current_lecture_ids.size != current_lecture_ids.uniq.size
8      errors.add(:scheduled_lectures,:lecture_scheduled_multiple_times)
9    end
10 end
```

Da durch `build` erzeugte Instanzen nur in genau der Objektinstanz sichtbar sind, durch die sie erstellt wurden, muss die Prüfung der Eindeutigkeit der Veranstaltungen in `StudySchedule` erfolgen. Dies hat allerdings den Nachteil, dass die Gültigkeit jedes `StudySchedule` außerhalb der zugeordneten Felder beeinflusst werden kann, indem weitere Instanzen von `ScheduledLecture` erzeugt werden. Dies könnte zu einer Situation führen, in der eine direkt aus der Datenbank geladene Instanz nicht gültig ist - und somit nicht wieder gespeichert werden kann.

Die Methode `mark_for_destruction` ist das Analogon zu `build` für das Löschen von Einträgen. Entsprechend markierte Objekte werden gelöscht, sobald das Elternobjekt gespeichert wird. Diese Funktionalität hat gegenüber `build` noch den weiteren Nachteil, dass alle zur Löschung

bestimmten Objekte weiterhin durch die Hilfsfunktionen der entsprechenden Relation (in diesem Fall `scheduled_lectures`) zurückgegeben werden. Somit muss potenziell bei jeder Verwendung zwischen markierten und unmarkierten Objekten unterschieden werden. Dies ist durch die boolesche Eigenschaft `marked_for_destruction?` möglich, die in `lecture_uniqueness` in Zeile 5 verwendet wird.

Aktualisierung

In diesem Kontext ist das genaue Vorgehen beim Erzeugen der `ScheduledLectures` zu beschreiben. Die Eingabemaske liefert beim Speichern durch den Benutzer eine Liste von Ids der gewählten `LectureOffers`. Diese werden durch den `StudySchedulesController` an die Methode `update_offers` des zugeordneten `StudySchedule` übergeben.

Die Eingabe ist ein *Ruby* `Hash`, welches zu jeder Kategorie eine Liste von zugeordneten Veranstaltungsangeboten (in Form der entsprechenden `LectureOffer.id`) enthält. Der Kern der Funktion ist das Herausfiltern von Angeboten, für die bereits eine Relation existiert (Zeile 8). Dies ist notwendig, damit für die seit dem letzten Speichern enthaltene Veranstaltungen nicht erneut eine Instanz von `ScheduledLecture` erzeugt wird.

Wie im Zusammenhang mit dem *Constraint* `lecture_uniqueness` bereits diskutiert, wird zum Löschen der zu verwerfenden `ScheduledLectures` die Methode `mark_for_destruction` (Zeile 5) verwendet. Sollte der Speichervorgang nicht erfolgreich sein, wird somit die letzte gültige Version des Studienplanes erhalten.

```
1  def update_offers(schedule)
2    scheduled_lectures \
3      .select { |sl| schedule[sl.lecture_category_id].nil? or
4                     not schedule[sl.lecture_category_id].include? sl.lecture_offer_id } \
5      .each { |sl| sl.mark_for_destruction }
6    lecture_categories.each do |c|
7      schedule[c.id].each do |oid|
8        if not scheduled_lectures.any? { |sl| sl.lecture_category_id == c.id and
9                                              sl.lecture_offer_id == oid }
10         begin
11           o = LectureOffer.find(oid)
12           scheduled_lectures.build(:lecture_category => c, :lecture_offer => o)
13         rescue ActiveRecord::RecordNotFound
14         end
15       end
16     end unless schedule[c.id].nil?
17   end
18 end
```

Ein weiterer Aspekt ist das Ignorieren von ungültigen Ids für Veranstaltungsangebote. Zeile 11 löst eine *Exception* aus, falls unter der angegebenen `oid` kein Eintrag existiert. Diese wird durch Zeile 13 direkt verworfen. Es kann in diesem Fall nicht ohne weiteres eine `ScheduledLecture` erzeugt werden, da kein gültiger Eintrag für die Referenz auf `LectureOffer` existiert. Es könnte zwar durch Anpassung der entsprechenden *Constraints* `nil` verwendet werden, dazu wäre aber bei jedem Zugriff auf das entsprechende Feld eine Fallunterscheidung notwendig. Besonders

in den *Views* müsste dann auch eine Darstellung einer ungültigen Veranstaltung eingebunden werden.

Da eine ungültige Anfrage dieser Art bei vorgesehener Verwendung des Systems nicht auftreten sollte, stände die notwendige Komplexität in keinem Verhältnis zum Nutzen. Aus dem gleichen Grund werden durch Zeile 6 alle hinzuzufügenden Veranstaltungen verworfen, die einer für das aktuelle Studienprogramm irrelevanten Kategorie zugeordnet sind.

Ein ähnliches Problem entsteht beim Ändern des Studienprogramms, falls Veranstaltungen in einer Kategorie vorhanden sind, die vom neuen Programm nicht verwendet wird. Daher wurde die Methode `study_program=` in `StudySchedule` wie folgt überschrieben

```
def study_program=(p)
  scheduled_lectures.each do |sl|
    if not p.lecture_categories.any? { |c| c.id == sl.lecture_category_id }
      sl.mark_for_destruction
    end
  end
  write_attribute :study_program_id, p.id
end
```

Es werden also alle nicht mehr relevanten `ScheduledLectures` entfernt. Ein Nachteil dieser Methode ist, dass bei mehrfachem Wechsel des Studienprogramms evtl. Veranstaltungen verloren gehen, die sowohl im ersten als auch letzten Programm problemlos einzubringen wären.

Da allerdings Leistungspunkte von Veranstaltungen in nicht aufgeführten Kategorien nicht in die Gesamtpunktzahl eingehen dürfen, müssten sie bei jeder anderen Lösung gesondert geführt werden. Dies hätte potenziell Auswirkungen auf jede Stelle, an der `ScheduledLecture` verwendet wird. Zusätzlich müssten solche `LectureOffer` und die zugehörigen Instanzen von `LectureCategory` gesondert dargestellt werden. Besonders letzteres würde die Bedienung weniger intuitiv gestalten, da diese Problematik von den meisten Benutzern nicht erkannt werden würde und auch nicht auf einfache Weise zu vermitteln ist.

HTML/JavaScript

Die Erstellung der *edit*-Ansicht des Studienplans stellt eine besondere Herausforderung dar, da nicht einzelne Attribute, sondern die zugehörigen `ScheduledLectures` verändert werden. Aus diesem Grund nutzt die Ansicht *Java-* bzw. *CoffeeScript* um das Hinzufügen bzw. Entfernen von Veranstaltungen aus einzelnen Kategorien zu ermöglichen. Das zugehörige Skript befindet sich in `app/assets/javascripts/study_schedules.js.coffee`.

Für jede geplante Veranstaltung wird ein verstecktes Feld erzeugt, welches die Id des zugehörigen `LectureOffer` enthält. *Ruby on Rails* unterstützt dabei eine Listen-Notation. Der folgende Code bezieht sich auf ein Veranstaltungsangebot mit der Id `42` in der Kategorie mit Id `1`. Typischerweise enthält jedes Dokument mehrere Kategorien mit unterschiedlicher Id.

```
<input name="categories[1][]" type="hidden" value="42">
```

Alle Werte in Feldern mit dem Namen `categories[i][]` werden durch *Ruby on Rails* in einer Liste zusammengefasst. Diese sind in *Controllern* über das Hash `params` verfügbar. Das zugehörige Semester muss nicht extra codiert werden, da es implizit über das referenzierte `LectureOffer`

festgelegt wird. Die Listen werden durch die `update`-Methode des `StudySchedulesController` entgegengenommen und an `update_offers` übergeben.

Diu verfügbaren Veranstaltungen sind im Dokument in einer Liste aufgeführt. Dabei werden in diesem Fall keine `input`-Felder verwendet, da die Daten beim Absenden des Formulars nicht mit übertragen werden dürfen.

```
<ul class="offers">
    <li data-offer-id="588"
        data-lecture-id="202"
        data-ects="8.0"
        class="offer-scheduled lecture-scheduled">
      <a class="add-offer" href="#"></a>
      <span class="ects">8.0 ECTS</span>
      <span class="title">Collaboration and Integration of Information Systems</span>
    </li>
    [..]
</ul>
```

Diese Liste bildet gleichzeitig das Datenmodell für das zugehörige Skript. Zu jedem Eintrag kann damit der Titel der Veranstaltung, das Semester und die ECTS durch die entsprechenden Elternobjekte im Baum bestimmt werden. Dies ist durch die Verwendung der in *Ruby on Rails* automatisch eingebundenen *JavaScript*-Bibliothek *jQuery*[1] einfach möglich.

```
@semester_code: (e) ->
  r = e.attr('data-semester-code')
  r = e.parent().closest('li').attr('data-semester-code') if r == undefined
  parseInt r

@lecture_title: (e) ->
  e.find('.title').text()

@offer_ects: (e) ->
  parseFloat e.attr('data-ects')
```

Daraus kann dann ein entsprechender Knoten an der zugehörigen Stelle im Dokument erzeugt werden. Die Id der zugehörigen `Lecture` ist ebenfalls codiert, damit alle weiteren Angebote entsprechend markiert werden können.

Im Detail treten noch einige Schwierigkeiten auf - insbesondere dadurch, dass auf Seite der geplanten Veranstaltungen evtl. Listeneinträge für Semester generiert oder entfernt werden müssen. Auch sind alle Angebote, die sich auf dieselbe Veranstaltung beziehen mit den Klassen `lecture-scheduled` bzw. `offer-scheduled` zu markieren.

Ein Nachteil dieses Vorgehens ist, dass der zugehörige HTML-Code sowohl auf Server- als auch Clientseite erzeugt werden muss. Zusätzlich orientiert sich das Datenformat zwangsweise an der Dokument- und damit an der Darstellungsstruktur. Für jede Änderungen sind daher sowohl `app/views/study_schedules/edit.html.erb` als auch `app/assets/javascripts/study_schedules.js.coffee` anzupassen. Dafür ist die Bedienung sehr flüssig, da nicht jede Änderung eine neue Anfrage an den Server auslöst.

[1] Offizielle *jQuery*-Seite: http://jquery.com/.

4.1.5 Benutzerverwaltung

Die technische Grundlage der Benutzerverwaltung ist [1] entnommen. Wenig überraschend bildet die Klasse `User` den Kern. Neben den oben aufgeführten Relationen zu anderen Klassen, enthält sie als Informationen zu den Nutzern die Felder `name` und `email`. Zusätzlich sind die booleschen Felder `admin` und `advisor` enthalten, die für die Rechtevergabe genutzt werden.

Zum Speichern der Passwörter wird das Plugin `bcrypt-ruby` verwendet. Dieses stellt das Makro `has_secure_password` zur Verfügung, welches beim Setzen oder Ändern des Benutzerpasswortes im Feld `password_digest` automatisch einen sicheren Hash-Wert zum Speichern in der Datenbank ablegt. Weiterhin wird die Methode `authenticate` zur Klasse hinzugefügt, welche überprüft ob ein gegebenes Klartextpasswort auf diesen Hash abgebildet wird.

Die eigentliche An- bzw. Abmeldung wird durch den `SessionController` realisiert. Er stellt die Methoden `new`, `create` und `destroy` zur Verfügung, welche der Login-Maske bzw. dem An- und Abmeldevorgang entsprechen.

```
def destroy
  sign_out
  flash[:success] = t 'interface.messages.signout_successful'
  redirect_to signin_path
end

def create
  user = User.find_by_email(params[:session][:email].downcase)
  if user && user.activated? && user.authenticate(params[:session][:password])
    sign_in user
    flash[:success] = t 'interface.messages.signin_successful', :name => user.name
    if params[:redirect]
      redirect_to params[:redirect]
    else
      redirect_to home_path
    end
  else
    flash.now[:error] = t 'interface.messages.signin_failed'
    @redirect = params[:redirect]
    render 'new'
  end
end
```

Beim Anmelden wird nach erfolgreicher Validierung des Passwortes ein signiertes Cookie (*session*) erzeugt, welches die Benutzer-Id enthält. Die Definition von `sessions_helper` stellt die entsprechende Methode `sign_in` bereit.

```
def sign_in(user)
  session[:user] = user.id
  self.current_user = user
end

def sign_out
  self.current_user = nil
```

```
  session.delete :user
end
```

Weiterhin stellt `sessions_helper` (durch Einbinden von `SessionsHelper` in `ApplicationController`) in jedem *Controller* die Methode `current_user` zur Verfügung, welche versucht, dieses Cookie auszulesen und im Erfolgsfall die zugeordnete `User`-Instanz zurückgibt.

Die Erstellung neuer Benutzerkonten wird über die Methoden `new` bzw. `create` des `UsersController` realisiert. Die Validierung der Benutzerdaten wird - wie in *Ruby on Rails* typisch - über *Constraints* des `User`-Objektes vorgenommen. Insbesondere werden die Eindeutigkeit und das Format der E-Mail Adresse, die Länge des Passwortes (mindestens 6 Zeichen) sowie eine minimale und maximale Länge des Benutzernamens geprüft.

```
1  has_secure_password
2  has_one :study_schedule, :dependent => :destroy
3  has_and_belongs_to_many :advises, :class_name => 'StudySchedule'
4
5  before_create :create_activation_token
6  before_save { |user| user.email = email.downcase }
7
8  VALID_EMAIL_REGEX = /\\A[\\w+\\-.]+@[a-z\\d\\-.]+\.[a-z]+\\z/i
9
10 validates :name, :presence => true,
11                 :length => { :minimum => 2, :maximum => 50 }
12
13 validates :email, :presence => true,
14                  :format => { :with => VALID_EMAIL_REGEX },
15                  :uniqueness => { :case_sensitive => false }
16
17 validates :admin, :advisor, :inclusion => {:in => [true,false]}
18
19 after_validation :delete_password_digest_errors
20
21 validates :password, :presence => true,
22                      :length => { :minimum => 6 },
23                      :on => :create
24
25 validates :password, :length => { :minimum => 6 },
26                      :unless => 'password.nil?',
27                      :on => :update
```

Das Feld `password` ist durch das Makro `has_secure_password` virtuell - es besitzt also keine zugeordnete Spalte in der Datenbank. Insbesondere enthält es nur beim Erzeugen einer neuen Instanz oder nach dem expliziten Setzen Daten, da das Passwort nicht im Klartext gespeichert wird. Fehler im Feld `password_digest` - welches den Passwort-Hash enthält - sind somit auf Fehler im Feld `password` zurückzuführen. Daher werden in Zeile 19 Fehler, welche sich auf dieses Feld beziehen, gelöscht, damit dem Benutzer nicht mehrere Hinweise für denselben Fehler angezeigt werden.

Ein weiteres Problem ergibt sich durch die Unterscheidung von Erstellen und Ändern der Benutzer. Wird ein neuer Benutzer erzeugt, so ist es notwendig, dass ein Passwort angegeben

wird. Würde dieselbe Einschränkung auch beim Editieren gelten, so müsste bei jeder Änderung das Passwort neu eingegeben werden. Dies ist besonders problematisch, da Administratoren zwar Benutzerdaten ändern können sollten, aber in der Regel die (Klartext-)Passwörter nicht kennen. Daher wird durch Zeile 25 die Anforderung an das Passwort-Feld abgeschwächt.

Um die Gültigkeit der E-Mail-Adresse sicherzustellen, sind Benutzerkonten nach der Anmeldung deaktiviert. Dazu wird im Feld `activation_token` beim Erstellen eine zufällige Zeichenkette erzeugt. Ist eine solche Zeichenkette vorhanden, verbietet der `SessionsController` den Login.

Die Methode `activation_url` der Klasse `User` erzeugt aus dem `activation_token` eine URL, deren Aufruf über die Methode `activate` des `UsersController` diese Zeichenkette löscht und somit den Benutzer freischaltet. Nach dem Erstellen des Kontos versendet der `UsersController` in der Methode `create` automatisch eine E-Mail, die diesen Link enthält.

Ähnlich ist auch die Funktionalität zum Zurücksetzen des Passwortes implementiert. Der `UsersController` stellt dazu die Methoden `reset_password_form`, `reset_password_request` und `reset_password` zur Verfügung. `reset_password_form` generiert eine Eingabemaske für eine E-Mail-Adresse. Die Eingabe wird an `reset_password_request` übergeben. Ist der angegebenen Adresse ein Benutzer zugeordnet, so wird auf dem zugehörigen `User`-Objekt `reset_password!` aufgerufen und über den `UserMailer` eine E-Mail mit Anweisungen zum Zurücksetzen des Passwortes geschickt. Die *Controller*-Methode `reset_password` meldet den Benutzer an, falls das übergebene Token gültig ist.

Diese Funktionalität ist durch das Feld `reset_password_token` umgesetzt. Dieses wird durch einen Aufruf der `User`-Methode `reset_password!` mit einer zufälligen Zeichenkette gefüllt. Durch `reset_password_url` lässt sich eine zugehörige URL konstruieren, die durch die Methode `reset_password` in `UsersController` ein einmaliges Anmelden erlaubt. Gleichzeitig löscht `reset_password` auch ein evtl. vorhandenes `activation_token`. Damit erlaubt diese Funktionalität auch ein Aktivieren des Kontos, falls die ursprüngliche E-Mail nicht mehr vorhanden ist.

Anders als `activation_token` sperrt ein vorhandenes `password_reset_token` den Account nicht. Da beim Anfordern der entsprechenden E-Mail keine weiteren Daten (insbesondere auch kein Passwort) abgefragt werden, könnte sonst über die zugehörige Seite jeder Benutzer nur durch Kenntnis der E-Mail Adresse beliebige Konten sperren.

4.1.6 Benutzerrollen

Die Eigenschaften `advisor` und `admin` legen die Rollen von Benutzern im System fest. Die Rechte können durch Administratoren in der Benutzerverwaltung vergeben werden.

Administratoren haben Zugriff auf alle Benutzerprofile und Studienpläne sowie auf die Synchronisation (s. Abschnitt Synchronisation). Berater wie auch Administratoren haben Zugriff auf die Übersichten zu Masterprogrammen und Veranstaltungen. Dabei ist den Administratoren die Information vorbehalten, welche Studierenden ein gegebenes Masterprogramm gewählt haben und welche eine bestimmte Veranstaltung besuchen.

Die Berater sind zusätzlich über eine *many-to-many* Beziehung mit den Studienplänen verknüpft. Sie können in der entsprechenden Ansicht (`StudySchedule/edit`) durch den Studenten hinzugefügt werden. Dabei werden nur solche Benutzer zur Auswahl angeboten, denen explizit die Beraterrolle zugewiesen wurde.

Die Notwendigkeit der Auswahl ist evtl. für die Studierenden nicht offensichtlich, da jedes Masterprogramm einen *research advisor* besitzt, was erwarten lassen könnte, dass der Berater durch die Wahl des Programms festgelegt wird. Dieser ist zwar in den Daten der Moduldatenbank vorhanden, allerdings fehlt der Bezug zu den Benutzern des Studienplaners. Die zuerst gewählte Alternative war, jedem Benutzer in der Rolle eines Beraters Zugriff auf alle Studienpläne zu gewähren. Durch die Vielzahl an Personen, die für die Beraterrolle in Frage kommen, wäre diese Lösung zwar komfortabel, aber auch sehr grob.

Die explizite Wahl ihrer Berater durch die Studierenden vermeidet die Notwendigkeit, nach Erstellen eines Masterprogramms die entsprechende Verbindung im Studienplaner erneut herstellen zu müssen. Dies wäre insbesondere durch die Verzögerung bis zur nächsten Synchronisation mit der Moduldatenbank (s. Synchronisation) unkomfortabel. Zusätzlich erlaubt es Flexibilität in der Form, dass Studierenden sich von Beratern betreuen lassen können, die nicht explizit als Verantwortliche für das gewählte Masterprogramm eingetragen wurden.

Alle Berater haben ebenfalls eine gegenüber den Administratoren eingeschränkte Übersicht der Benutzer. Diese zeigt nur das gewählte Masterprogramm für alle von ihnen betreuten Studierenden an und erlaubt das Editieren der zugehörigen Studienpläne. Somit können die Daten eines Benutzers nur von solchen Beratern eingesehen werden, denen explizit der Zugriff erlaubt wurde.

4.2 Synchronisation

Die Veranstaltungsdaten und Informationen zu den Masterprogrammen stammen aus der Moduldatenbank. Diese Daten werden ausgelesen und in der internen relationalen Datenbank dupliziert.

Diese Speicherung ist notwendig, da viele Anfragen eine Reihe von Informationen benötigen und die Schnittstelle der Moduldatenbank nicht auf eine direkte online-Verwendung ausgelegt ist. Zusätzlich hätte die Applikation keinen Einfluss auf die Löschung von Daten. Dies wäre problematisch, da Studienpläne so nicht mehr vorhandene Datensätze zu Modulen und Masterprogrammen referenzieren könnten.

Die Daten werden durch die Moduldatenbank im XML-Format bereitgestellt. Dabei sind diese in zwei Arten von Datensätzen unterteilt. Zum einen Informationen zu Veranstaltungsangeboten und zum anderen eine Liste von Masterprogrammen, die auf diese Angebote verweisen.

Moduldaten bestehen aus einem Index, welcher zu allen Modulen jeweils einen eindeutigen Modulcode und eine URL zu einem XML-Dokument mit detaillierten Informationen enthält.

```
<index>
  <modul>
    <code>A3.1</code>
    <url>
      http://www-ps.informatik.uni-kiel.de/~mh/studiengaenge/show.cgi?xml=A3%2E1
    </url>
  </modul>
  <modul>
    <code>A3.2</code>
    <url>
      http://www-ps.informatik.uni-kiel.de/~mh/studiengaenge/show.cgi?xml=A3%2E2
```

```
    </url>
  </modul>
  [..]
</index>
```

Die einzelnen Dokumente enthalten eine Reihe von Daten, wobei nicht alle für den Studienplaner relevant sind. Sie könnten zwar genutzt werden, um das Interface sinnvoll zu erweitern - allerdings hätte dies zur Folge, dass im Wesentlichen nur die Funktionalität der Moduldatenbank dupliziert würde.

```
<modul>
  <modulcode>MS0303</modulcode>
  <modulname>
    <deutsch>Deklarative Programmiersprachen</deutsch>
    <englisch>Declarative Programming Languages</englisch>
  </modulname>
  <verantwortlich>Prof. Dr. Michael Hanus</verantwortlich>
  <ectspunkte>8</ectspunkte>
  <workload>240 Std.</workload>
  <lehrsprache>Deutsch</lehrsprache>
  <kurzfassung>[..]</kurzfassung>
  <lernziele>[..]</lernziele>
  <lehrinhalte>[..]</lehrinhalte>
  <voraussetzungen/>
  <pruefungsleistung>[..]</pruefungsleistung>
  <lehrmethoden/>
  <verwendbarkeit/>
  <literatur>[..]</literatur>
  <verweise/>
  <kommentar/>
  <studiengaenge>
    <studiengang key="MSc">Masterstudiengang Informatik</studiengang>
  </studiengaenge>
  <kategorien>
    <kategorie key="MSc_IG">Vertiefende Informatik-Grundlagen (MSc Inf.)</kategorie>
    <kategorie key="MSc_MV">Mastervertiefungsbereich (MSc Inf.)</kategorie>
  </kategorien>
  <durchfuehrung>
    <praesenz>4V 2Ü 0P 0S</praesenz>
    <dauer>1</dauer>
    <turnus>unregelmäßig</turnus>
    <veranstaltung>
      <semester>SS07</semester>
      <dozent>Prof. Dr. Michael Hanus</dozent>
    </veranstaltung>
    <veranstaltung>
      <semester>SS09</semester>
      <dozent>Prof. Dr. Michael Hanus</dozent>
    </veranstaltung>
    [..]
```

```
  </durchfuehrung>
</modul>
```

Der erste Schritt ist das Parsen der Dokumente. Leider wird durch die Moduldatenbank kein Schema bereitgestellt. Die XML-Struktur ist zwar selbsterklärend, allerdings gibt es keine Aussagen über die Existenz oder das Format von Attributen. Weiterhin ist es zum Auslesen der Daten zu einzelnen Modulen notwendig, ein weiteres Dokument zu laden. Auch dies ist eine potenzielle Fehlerquelle. Daher ist ein flexibler Mechanismus notwendig.

Eine weitere Herausforderung bildet das Löschen von Modulen. Dies verbietet das schrittweise Einlesen der einzelnen Veranstaltungsdaten, da für jedes in der Datenbank vorhandene Datum geprüft werden muss, ob dieses noch in der Moduldatenbank vorhanden ist. Entsprechend besteht die Datenaktualisierung aus zwei Schritten: Dem Parsen der Daten in eine Datenstruktur und dem eigentlichen Abgleich mit der lokalen Datenbank. Dies fügt außerdem noch eine weitere Abstraktionsebene hinzu, die insbesondere den Umgang mit Änderungen des Formats erleichtert und es relativ einfach erlauben würde, Moduldaten aus gänzlich anders strukturierten Quellen zu übernehmen.

4.2.1 Einlesen der Daten aus der Moduldatenbank

Das Lesen der Daten ist in der Klasse `ModuleXml` in `lib/` realisiert. Die statischen Funktionen `get_modules` und `get_programs` parsen die Modul- bzw. Programmdaten. Die Quellen sind in den Feldern `config.modules_xml_path` und `config.programs_xml_path` in `config/application.rb` festgelegt.

Einlesen der Module

Das eigentliche Parsen der XML-Dokumente wird durch das *Ruby*-Modul `rexml` übernommen. Dies übersetzt die Dokumente in eine DOM-Struktur und erlaubt das Traversieren des resultierenden Baums.

Alle Daten werden in geschachtelte *Ruby* Hashes übersetzt. Die Funktionalität dazu stellt das Modul `RemoteXml` in `lib/` bereit. Dies erlaubt es auch die bis dahin noch nicht durchgeführte Validierung der Daten vorzunehmen.

Das Einlesen der Daten ist durch `RemoteXml` relativ übersichtlich. Der folgende Code liest einen XML-Knoten ein und gibt einen `Hash` mit Einträgen für `:code` und `:name` zurück.

```
def self.parse_category(e)
  {
    :code =>
     (verify parse_attr(e,'key'),
             on_empty => 'empty key-attribute in category' ).upcase,
    :name =>
     (verify e.text,
             on_empty => 'category without name' )
  }
end
```

Der folgende Aufruf kann beispielsweise dazu verwendet werden, um eine Liste solcher Hashes zu erstellen. Der im zweiten Argument angegebene *XPath* bestimmt, für welche Knoten ein Eintrag erzeugt wird.

```
parse_elems(r,'kategorien/kategorie') { |e| parse_category e }
```

Das Parsen der Module funktioniert nach einem ähnlichen Prinzip. `RemoteXml` stellt Methoden zur Verfügung, die es erlauben, das Laden und Parsen eines Dokumentes zusammenzufassen. Die Funktion `nil_on_error` liefert bei einem Fehler `nil` zurück. Dies ist insbesondere im Zusammenhang mit der Funktion `verify` wichtig, welche eine Exception auslöst, falls die Daten nicht dem erwarteten Format entsprechen.

Generiert wird ein `Hash` welcher alle Module mit dem jeweiligen `code` als Schlüssel enthält. Der `code` erlaubt es, die zugehörige Instanz von `Lecture` in der lokalen Datenbank zu identifizieren.

```
index = get_and_parse(connection,url) { |doc| parse_module_index doc }

modules = Hash.new
index.each do |m|
  modules[m[:code]] = nil_on_error {
                        get_and_parse(connection,m[:url]) { |doc| parse_module doc } }
end
```

Die Funktion `parse_module` ist analog zu `parse_category` und liest die für die Anwendung relevanten Felder `code`, `title` und `ects` ein. Zusätzlich werden - wie oben beschrieben - Listen von Kategorien und Semestern, in denen dieses Modul angeboten wird, unter den Schlüsseln `:categories` bzw. `:offers` abgelegt.

```
def self.parse_module(doc)
  r = doc.root
  {
    :code =>
     (verify parse_elem(r,'modulcode') { |c| c.upcase },
             on_empty => 'empty module-code'),
    :title =>
     (verify parse_elem(r,'modulname/deutsch') {
                |t| t.empty? ? parse_elem(r,'modulname/englisch') : t },
             on_empty => 'module title missing'),
    :ects =>
      parse_elem(r,'ectspunkte') { |e| BigDecimal(e.sub(',','.')) },
    :offers =>
      parse_elems(r,'durchfuehrung/veranstaltung/semester') do |e|
        verify Semester.parse(e.text),
               on_nil => 'malformed semester in offer: #v'
      end,
    :categories =>
      parse_elems(r,'kategorien/kategorie') { |e| parse_category e }
  }
end
```

Wichtig ist in diesem Zusammenhang, dass für jede Kategorie sowohl der Eintrag `code` (generiert aus dem XML-Attribut `key`) als auch `name` eingelesen werden. Zur Identifizierung würde der

(eindeutige) `code` genügen. Allerdings wird durch die Moduldatenbank keine Liste aller Kategorien bereitgestellt, so dass der Name nur an dieser Stelle vorhanden ist.

Einlesen der Masterprogramme

Die Liste der Masterprogramme ist ein XML-Dokument, welches in den Elementen `<studyprogram>` alle Informationen zu jeweils einem Programm enthält. Das Lesen ist etwas einfacher als bei den Moduldaten, da die Informationen nicht über verschiedene Dokumente verteilt sind. Das Fehlschlagen einer weiterführenden HTTP-Anfrage scheidet somit als Fehlerquelle aus.

```
<studyprogram ID="MasterProgram3">
  <title>Internet-Programmierung</title>
  <advisor>Prof. Dr. Michael Hanus</advisor>
  <start>WS07/08</start>
  <url>
    http://www-ps.informatik.uni-kiel.de/~mh/studiengaenge/
      show.cgi?listMasterProgram/MasterProgram3
  </url>
  <description>[..]</description>
  <prerequisites/>
  <comments/>
  <degreeprogram key="MSc">Masterstudiengang Informatik</degreeprogram>
  <lecture>
    <mandatory>yes</mandatory>
    <category>MSc_IG</category>
    <code>MS0306</code>
    <semester>WS07/08</semester>
  </lecture>
  <lecture>
    <mandatory>no</mandatory>
    <category>MSc_IG</category>
    <code>MS0502</code>
    <semester>SS08</semester>
  </lecture>
  [..]
</studyprogram>
```

Die relevanten Daten sind das Attribut `ID` - welches analog zu `code` in den Modulen - die eindeutige Identifizierung des Programms erlaubt und der Titel (`title`) sowie das Startsemester (`start`). Aus Konsistenzgründen wird das Feld `ID` intern als `code` geführt.

Die XML-Knoten `<lecture>` beschreiben jeweils eine Veranstaltungsempfehlung. Sie bestehen aus einer Referenz auf eine Kategorie `<category>`, ein Modul `<code>` und ein Semester `<semester>`. Zusätzlich gibt das boolesche Feld `<mandatory>` an, ob es sich um eine Pflichtveranstaltung handelt. Diese Daten entsprechen genau denen der Model-Klasse `RecommendedLecture`.

```
:lectures =>
  parse_elems(e,'lecture') do |l|
    {
```

```
      :mandatory =>
        parse_elem(l,'mandatory') { |m| m.downcase == 'yes' },
      :category =>
       (verify parse_elem(l,'category') { |m| m.upcase },
               on_empty => 'empty category for recommended lecture' ),
      :code =>
       (verify parse_elem(l,'code') { |c| c.upcase },
               on_empty => 'empty code for recommended lecture' ),
      :semester =>
       (verify parse_elem(l,'semester') { |s| Semester.parse s },
               on_nil => 'malformed semester code in recommended lecture: #v' )
    }
  end
```

Alle Programmdaten werden (analog zu `get_modules`) durch `get_programs` in einen `Hash` mit der `ID` bzw. `code` als Schlüssel gelesen.

4.2.2 Abgleich mit der lokalen Datenbank

Nach dem Einlesen können die lokalen Daten aktualisiert werden. Die Funktionalität wird durch die *Model*-Klassen `Lecture`, `LectureCategory` und `StudyProgram` jeweils durch die statische Methode `update_all` bereitgestellt.

`Lecture#update_all` extrahiert in einem ersten Schritt alle Kategorien aus den vorhandenen Moduldaten und übergibt diese an `LectureCategory#update_all`. Dies ist notwendig, da die Daten zu Kategorien nur implizit durch die Modulschnittstelle der Moduldatenbank bereitgestellt werden.

```
categories = Hash.new
lectures.each do |code,data|
  if data
    data[:categories].each do |c|
      categories[c[:code]] = c
    end
  end
end
es.concat LectureCategory.update_all(categories)
```

Das Vorgehen der drei `update_all`-Funktionen ist gleichartig. In einem ersten Schritt werden alle in der Datenbank vorhandenen Einträge mit den entsprechenden Werten der Eingabe abgeglichen. Sollten keine Daten vorhanden sein, so wird versucht die zugehörige Instanz zu löschen. Dies kann fehlschlagen, falls diese noch durch einen Studienplan referenziert wird. In diesem Fall wird ein Fehler (in `es`) zurückgegeben.

Im Beispiel von `Lectures#update_all` sieht dies wie folgt aus

```
all.each do |l|
  if lectures.has_key? l.code
    data = lectures.delete(l.code)
    es.concat l.update_data(data)
  elsif not l.destroy
```

```
    es << "Unable to delete lecture #{l.code}: #{l.errors.full_messages.join(", ")}"
  end
end
```

Durch dieses Vorgehen ist es möglich, dass der Studienplaner Module oder Masterprogramme enthält, die in der Moduldatenbank nicht mehr aufgeführt sind. Dieses Verhalten ist erwünscht, damit Studienpläne - und somit das zugehörige Studium - nicht automatisch verändert werden. Die in diesem Fall auftretende Fehlermeldung ist eine Rückmeldung an die Verantwortlichen, dass eine durchgeführte Änderung nicht ohne Rücksprache mit den betroffenen Studierenden möglich ist.

Die Verwendung von `has_key?` zur Existenzprüfung von Einträgen verhindert, dass Datensätze, deren `code` `nil` als Wert liefert, gelöscht werden. Dies ist der Fall, da `ModuleXml` mit `nil` Einträge markiert, die zwar in der Moduldatenbank vorhanden sind, deren Daten aber nicht vollständig geparst werden konnten. Solche Datensätze werden unverändert weitergeführt.

Das Löschen von Einträgen aus den Eingabedaten führt dazu, dass nach dieser Iteration nur noch solche Datensätze vorhanden sind, die kein Äquivalent in der lokalen Datenbank besitzen. Für diese wird dann in einem weiteren Schritt ein neuer Eintrag erzeugt.

```
lectures.each do |code,data|
  l = Lecture.new :code => code
  es.concat l.update_data(data)
end
```

Es ist zu beachten, dass diese Methode die eingegebene Datenstruktur (das Hash `lectures` im Beispiel) verändert. Dies hat den Grund, dass eine Kopie an dieser Stelle unnötig ist, da die Daten durch den gegebenen Kontrollfluss nur einmalig gelesen werden. Eine (tiefe) Kopie würde somit nur unnötigen Aufwand bedeuten.

Die Methode `update_data` auf einer Instanz von `Lecture` muss dabei neben den Attributen `title` und `ects` auch noch die Verweise auf `LectureCategory` anpassen und ggf. Instanzen von `LectureOffer` erzeugen und löschen. Die notwendigen Daten sind dabei in `data` vorhanden.

```
def update_data(data)
  return ["no data for lecture #{code}"] if data.nil?
  es = []
  write_attribute :title, data[:title]
  write_attribute :ects, data[:ects]
  es.concat update_categories(data[:categories])
  es.concat update_offers(data[:offers])
  if save
    es
  else
    es << "Unable to update lecture #{code}: #{errors.full_messages.join(", ")}"
  end
end
```

Die Methoden `update_categories` und `update_offers` müssen dabei jeweils mit der Schwierigkeit umgehen, dass bestehende Referenzen nicht unnötigerweise verändert werden dürfen. Somit können nicht einfach alle Instanzen gelöscht und neue anhand der Eingabedaten erzeugt werden. In beiden Fällen werden auch wieder nach außen sichtbare Änderungen am Eingabeparameter vorgenommen.

```
def update_categories(data)
  return ["no category data for lecture #{code}"] if data.nil?
  to_delete = []
  lecture_categories.each do |c|
    ci = data.find_index { |d| d[:code] == c.code }
    if ci
      data.delete_at(ci)
    else
      to_delete << c
    end
  end
  to_delete.each { |c| lecture_categories.delete c }
  return [] if data.empty?
  categories = LectureCategory.where(:code => data.map { |d| d[:code] })
  data.each do |d|
    ci = categories.find_index { |c| c.code == d[:code] }
    lecture_categories << categories[ci] if ci
  end
  []
end

def update_offers(semesters)
  semesters.uniq!
  es = []
  to_delete = Array.new
  lecture_offers.each do |o|
    to_delete << o if not semesters.delete(o.semester)
  end
  to_delete.each do |o|
    if o.destroy
      lecture_offers.delete o
    else
      es << "unable to delete lecture_offer #{o.lecture.title}
             in #{Semester.string(o.semester)}:
             #{o.errors.full_messages.join(", ")}"
    end
  end
  semesters.each do |s|
    lecture_offers.build :semester => s if not lecture_offers.any? { |o| o.semester == s }
  end
  es
end
```

Die Variable `es` sammelt alle im Verlauf der Synchronisation auftretenden Fehler. Dabei wird versucht, möglichst viele Daten der Moduldatenbank zu übernehmen und die Auswirkungen eines Fehlers möglichst lokal zu begrenzen. Eine mögliche Fehlerquelle ist eine bisher unbekannte Inkompatibilität zwischen den Daten der Moduldatenbank mit dem lokalen Datenbankschema. Dies ist nicht auszuschließen, da - wie bereits erwähnt - das Format der XML-Dateien nicht formal spezifiziert ist. Fehler dieser Art würden auftreten, falls die Eingabedaten die *Constraints*

einer *ActiveRecord*-Klasse nicht erfüllen. Ein damit verwandter Fehler sind ungültige Verweise innerhalb der Daten. Diese sind z. B. möglich, falls ein Masterprogramm auf ein Modulangebot in einem Semester verweist, in dem die betreffende Veranstaltung nicht angeboten wird. Eine letzte Möglichkeit ist die oben beschriebene Löschung eines Datensatzes, der im Studienplaner noch benötigt wird. Alle diese Fehler erlauben es, eine entsprechende Meldung zu generieren und die Synchronisation fortzusetzen.

Die Methoden `update_all` in `LectureCategory` und `StudyProgram` folgen einem analogen Muster. Wichtig ist dabei, dass die Aktualisierung auf `StudyProgram` nach der auf `Lecture` vorgenommen wird, damit Referenzen auf neue Module gültig sind.

Ein Problem bei diesem Vorgehen ist, dass große Teile der Datenbank (insbesondere die komplette Tabelle `Lectures`) in den Hauptspeicher geladen werden müssen. Bei der Beschränkung auf einen Studiengang sollte dies problemlos möglich sein. Wird die Anwendung auf mehrere Studiengänge erweitert, so ist zu überlegen ob diese unabhängig voneinander behandelt werden können. Verweise zwischen verschiedenen Studiengängen, die z. B. die Einführung eines Vertiefungsfaches notwendig machen kann, müssen in diesem Fall ggf. durch einen nachfolgenden Prozess gesondert behandelt werden.

4.2.3 Durchführung

Um die interne Datenbank aktuell zu halten, muss die Synchronisation regelmäßig durchgeführt werden. Zusätzlich stellt die Anwendung Administratoren eine Möglichkeit bereit, diese manuell über ein Webinterface zu starten. Da die Durchführung durch die Menge von HTTP-Anfragen einige Zeit in Anspruch nehmen kann, läuft bei letzterem ein synchroner Aufruf Gefahr, einen Timeout zu verursachen. Daher wurde das Plugin *delayed_job*[2] verwendet. Dieses ermöglicht es, anfallende Aufgaben in Hintergrundprozessen asynchron abzuarbeiten.

Die eigentliche Synchronisation wird in der Methode `perform` der Klasse `SynchronizationJob` in `lib/` durchgeführt. Im Falle eines manuellen Aufrufs wird sie durch den `SynchronizationController` in der Methode `start` durch den Aufruf `Delayed::Job.enqueue SynchronizationJob.new` gestartet.

Um den Synchronisationsvorgang möglichst transparent zu halten, wird eine Log-Datei geführt. Diese liegt in `log/synchronization.log` und wird durch die Klasse `SynchronizationLogger` verwaltet. Für die auftretenden Fehler wird jeweils ein Eintrag generiert. Zusätzlich wird der Start- und Endzeitpunkt einer Synchronisation festgehalten.

delayed_job bietet zusätzlich die Möglichkeit, eine durch eine Exception abgebrochene Aufgabe erneut zu starten. Die Anzahl der Versuche ist in `config/initializers/delayed_job.rb` festgelegt. Ein solcher Abbruch wird ebenfalls in der Log-Datei vermerkt.

Der `SynchronizationController` erlaubt Administratoren - durch die Methode `overview` - die Log-Datei einzusehen und den Synchronisationsprozess zu starten.

Die Methode `running` in `SynchronizationJob` liefert eine Liste der gerade aktiven Aufgaben. Dies ist möglich, da *delayed_job* eine Datenbanktabelle (`delayed_jobs`) verwendet, um diese zu verwalten.

```
def self.running
  Delayed::Job.where "handler LIKE '%SynchronizationJob%' AND locked_at IS NOT NULL"
end
```

[2] Delayed Job: https://github.com/collectiveidea/delayed_job.

Um Inkonsistenzen zu vermeiden, wird während einer aktiven Synchronisation der Zugriff auf die gesamte Applikation gesperrt. Dies ist über einen *Hook* in `ApplicationController` realisiert, welcher die in `SynchronizationHelper` definierte Methode `synchronization_in_progress?` verwendet.

Da der Mechanismus des asynchronen Synchronisierens damit vorhanden ist, bietet es sich an, die periodische Ausführung ebenfalls auf diese Weise durchzuführen. Dies verringert externe Abhängigkeiten und erlaubt es, die vorhandenen Funktionen zum Sperren der Applikation und zum Loggen zu verwenden. Leider bietet *delayed_job* keine Unterstützung für solche regelmäßigen Aufgaben. Allerdings ist es ohne weiteres möglich, dass eine abgeschlossene Aufgabe sich selbst neu startet. Zusammen mit der vorhandenen Funktionalität zur verzögerten Ausführung zu einem angegebenen Zeitpunkt erlaubt dies die Implementierung von periodischen Aufgaben.

Hierzu wird ein boolesches Klassenattribut `auto_restart` in `SynchronizationJob` eingeführt. Durch Überschreiben der Methoden `success` und `failure` kann in Abhängigkeit dieses Feldes nach Abschluss des Prozesses eine neue Aufgabe erstellt werden. Die Methode `after` eignet sich dazu nicht, da sie auch bei einem Fehlschlag aufgerufen wird, obwohl evtl. automatisch ein erneuter Versuch gestartet wird.

Die Zeitpunkte der Synchronisation werden in `config/application.rb` durch das Feld `config.synchronize_at` festgelegt. Die statische Methode `enqueue_next_regular_synchronization` in `SynchronizationJob` trägt dann eine Synchronisation zum nächsten Zeitpunkt ein.

Beim Start der Applikation werden durch `config/initializers/synchronization_job.rb` alle bisher eingetragenen Aufgaben gelöscht und `enqueue_next_requlal_synchronization` aufgerufen.

5 Erweiterbarkeit

Die naheliegendste Erweiterung des Studienplaners ist die Aufnahme weiterer Studiengänge. Dies würde eine weitere *Model*-Klasse notwendig machen, die eine Repräsentation für die unterstützten Studiengänge liefert. Der Studienplan sowie die Kategorien und Masterprogramme müssten dann um eine entsprechende Referenz erweitert werden. Die Auswahl des Studienfaches könnte problemlos ähnlich der des Masterprogrammes gestaltet werden. Die zur Auswahl stehenden Veranstaltungen sind dann durch die bestehenden Relationen einfach einzuschränken.

Bei der Unterstützung vieler Studiengänge ist zu überlegen, ob die Rechte der Benutzer weiter verfeinert werden sollten. So ist zu überlegen, ob sich die Beraterfunktion nur noch auf einen bestimmten Studiengang bezieht. Dies würde die Einführung einer weiteren *one-to-many* oder sogar *many-to-many* Beziehung zwischen Benutzern und Studiengängen erfordern. Auch wäre es sicherlich sinnvoll, einen Verwalter benennen zu können, der administrative Aufgaben nur im Kontext eines bestimmten Studienganges wahrnehmen kann. Dies würde eine weitere Benutzerrolle erfordern. Die Zuordnung kann dann analog zu der des erweiterten Beraters erfolgen.

Die obigen Erweiterungen fügen zwar einiges an Komplexität zum Klassenschema hinzu, sollten aber problemlos integriert werden können. Die Synchronisation der Datenbank erfordert unter Umständen allerdings einige Erweiterungen. Wie bereits im Abschnitt Synchronisation diskutiert, sollte diese nach Möglichkeit für jeden Studiengang unabhängig vorgenommen werden. Die Methoden `update_all` in `Lecture`, `LectureCategory` und `StudyProgram` sind somit entsprechend auf ein gegebenes Studienfach einzuschränken.

Die Anpassung des XML-Parsers kann einfach vorgenommen werden - zumindest falls die Daten auch weiterhin über die Moduldatenbank bereitgestellt werden. Die zugehörigen Daten finden sich in den Feldern `<studiengaenge>` der Modulinformationen und `<degreeprogram>` der Masterprogramm-Daten. Die Datensätze können somit einfach gruppiert werden.

Sollten Daten auch aus anderen Quellen übernommen werden, so müssen Vorschriften für die jeweiligen Schnittstellen implementiert werden. Für Daten, die im XML-Format vorliegen, könnte dies durch die im Modul `RemoteXML` bereitgestellten Methoden geschehen. Andere Formate benötigen eine eigene Parserimplementierung. Die eigentliche Herausforderung wird in der Regel allerdings nicht das Datenformat selbst, sondern die Art und Struktur der bereitgestellten Daten bilden. Um diese sinnvoll einbinden zu können, müssen sie semantisch äquivalent auf das Format des Studienplaners abgebildet werden. Problematisch werden dadurch besonders fehlende Felder. Über mögliche Lösungen könnte an dieser Stelle allerdings nur spekuliert werden, da keine konkreten Schnittstellen bekannt sind.

Mit Zunahme der Datenmenge wird auch die Synchronisation entsprechend mehr Zeit in Anspruch nehmen. Daher ist in diesem Fall zu überlegen, ob die Sperrung der gesamten Anwendung während dieses Prozesses noch zu rechtfertigen ist. Ggf. sollte der Zugriff auf nicht betroffene Bereiche und Studiengänge erlaubt werden.

Eine weitere sinnvolle Erweiterung wäre es, weitere von der Moduldatenbank bereitgestellte Informationen zu nutzen, um insbesondere die *edit*-Ansicht der Studienpläne zu erweitern. Dabei wäre zu überlegen ob z. B. Modulbeschreibungen, Voraussetzungen und Dozenten zu

jeder Veranstaltung angezeigt werden können. Allerdings ist zu beachten, dass jede angezeigte Information Einfluss auf die Übersichtlichkeit und damit Zugänglichkeit der Anwendung hat. Ein Hinzufügen von Verweisen zu Seiten mit diesen Informationen ist problematisch, da durch die starke Nutzung von *JavaScript* der Zustand der Seite und somit vorgenommene Änderungen verloren gehen, sobald einem Link gefolgt wird. Hinzu kommt, dass durch die Moduldatenbank bereits eine strukturierte Darstellung aller Moduldaten verfügbar ist.

Alle im Studienplaner verwendeten Texte wurden über das von *Ruby on Rails* bereitgestellte Schema zur Mehrsprachigkeit eingebunden. Obwohl derzeit nur ein deutsches Interface zur Verfügung gestellt wird, ist damit die Übersetzung in andere Sprachen durch Erstellen einer neuen Sprachdatei in `config/locales` sehr einfach möglich.

6 Zusammenfassung

Grundsätzlich wurden durch die Notwendigkeit von dreistelligen Relationen an einigen Stellen Grenzen des von *Ruby on Rails* implementierten *ActiveRecord*-Patterns erreicht. Dies betrifft vor allem das Erzeugen und Löschen von `ScheduledLecture`-Instanzen im Kontext von `StudySchedule`.

Daraus folgt auch, dass einige Klassen verhältnismäßig eng gekoppelt sind. Dies betrifft ebenfalls besonders den Zusammenhang zwischen `StudySchedule` und `ScheduledLecture`. Folglich ist die Wartung und Erweiterungen an dieser Stelle fehleranfällig. Allerdings ist durch die von *Ruby on Rails* vorgegebene Struktur keine losere Kopplung möglich.

Die Trennung der Verantwortungen nach der *MVC*-Struktur war in den meisten Fällen problemlos möglich. Insbesondere der Mechanismus des *Embedded Ruby* in den *View*-Klassen erlaubte sehr übersichtlichen und damit leicht erweiterbaren Code. An dieser Stelle fällt dadurch die etwas unhandliche Übergabe von Parametern an die *Views* durch Klassenvariablen um so mehr auf. Insbesondere in den sog. *Partials* sind nicht alle benötigten Daten ohne weiteres zu identifizieren. Trotzdem sind dadurch einfache Änderungen der Darstellung problemlos möglich.

Das umgesetzte Datenmodell erlaubt eine verlustfreie Abbildung der aus Moduldatenbank extrahierten Daten. Der implementierte Synchronisationsmechanismus ist robust und kann in gewissen Grenzen mit fehlerhaften oder inkonsistenten Daten umgehen. Die Unterstützung von *delayed_job* verringert externe Abhängigkeiten und erlaubt sogar eine direkte Rückmeldung durch die Applikation.

Der Studienplaner emöglicht die Planung des Masterstudiengangs für Studierende durch ein übersichtliches und reaktives Interface. Die Rolle von Beratern erleichtert dabei eine Rückmeldung durch die verantwortlichen Betreuer. Gleichzeitig ist es den Dozenten möglich, schnell eine Übersicht über die zu erwartenden Teilnehmer in den von ihnen angebotenen Veranstaltungen zu erhalten. Durch die Web-Schnittstelle kann diese sogar automatisiert erfolgen.

Die Umsetzung von Teilen des Interfaces in *JavaScript* erlaubt dabei eine intuitive und von den Antwortzeiten des Servers weitgehend unabhängige Bedienung. Diese sollte die Nachteile in Form von Inkompatibilitäten mit älteren Browsern und enge Verknüpfung der *Views* mit den Client-Seitigen Skripten aufwiegen.

7 Literaturverzeichnis

[1]Hartl, M.: *Ruby on Rails 3 Tutorial: Learn Rails by Example* : Pearson Education, 2010 — ISBN 9780132564199