UNIVERSITÄT OSNABRÜCK

Institut für Informatik

Bachelorarbeit

Agile Entwicklung einer Webapplikation zur Verwaltung und Auswertung von Klausur- und Übungsaufgaben in der Hochschullehre

Nils Haldenwang

15.12.2011

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachterin: Jun.-Prof. Dr. Elke Pulvermüller

Danksagungen

Hiermit möchte ich allen Personen danken, die mich bei der Erstellung der Arbeit unterstützt haben:

- Herrn Prof. Dr. Oliver Vornberger für die Tätigkeit als Erstgutachter und für die Bereitstellung der interessanten Thematik.
- Frau Jun.-Prof. Dr. Elke Pulvermüller, die sich als Zweitgutachterin zur Verfügung gestellt hat.
- Frau Maren Mikulla, Frau Jana Lehnfeld, Herrn Nicolas Neubauer, Frau Krista Haldenwang, Herrn Benjamin Molitor, Herrn Philipp Middendorf und Herrn Julian Kniephoff für das wertvolle Feedback während der Verfassung der Arbeit.

Zusammenfassung

Die vorliegende Arbeit stellt zunächst Agile Methoden und Technologien zur Entwicklung von Webapplikationen mit qualitativ hochwertigem Softwaredesign heraus. Außerdem wird eine Vorgehensweise zur Gewährleistung der hinreichenden Erfüllung sämtlicher Benutzeranforderungen durch die umgesetzte Funktionalität erarbeitet. Schließlich wird eine Applikation zur Verwaltung und Auswertung von Übungs- und Klausuraufgaben in der Hochschullehre unter Verwendung der vorgestellten Konzepte und Werkzeuge umgesetzt.

Abstract

This thesis exposes agile methodologies and technologies for high quality web application designs. Further, it identifies an approach to guarantee certain levels of end user satisfaction with the implemented functionality. Finally, a system to manage and evaluate assignments for assignment sheets and written examinations in academic contexts is implemented using the acquired concepts and tools.

Inhaltsverzeichnis

1 Einleitung

Vorlesungsbegleitende Übungsaufgaben sind ein wichtiger Bestandteil der Hochschullehre. Die Ergänzung des theoretisch vermittelten Stoffes durch praktische Übungen kann erheblich zum tiefgreifenden Verständnis der Inhalte beitragen. Geeignete technische Hilfsmittel im Prozess der Erstellung, Archivierung und Reflexion von Aufgaben können die entsprechenden Abläufe erheblich vereinfachen, sind in der Praxis aber nicht sonderlich präsent. In diesem Kapitel wird zunächst die Erstellung einer technischen Lösung etwas umfangreicher motiviert, als nächstes werden die genauen Ziele festgelegt und schließlich ein kurzer Überblick über den Aufbau der Arbeit gegeben.

1.1 Motivation

Am Institut für Informatik in Osnabrück werden fast alle Lehrveranstaltungen durch vorlesungsbegleitende Praxisaufgaben ergänzt. Der Ablauf der Erstellung, Archivierung und Auswertung unterscheidet sich innerhalb des Instituts und der einzelnen Arbeitsgruppen enorm. Meist besitzt jeder Übungsleiter ein eigenes System zur Aufgabenverwaltung. Einige heften ausgedruckte Exemplare der Übungsblätter und Klausuren zur Archivierung in Ordner, andere legen die Quelldateien in diversen Verzeichnisstrukturen ab.

Gerade bei Lehrpersonal in der Hochschullehre herrscht zudem große Fluktuation. Die Leitung der Übungsgruppen und die Verwaltung der Aufgaben wird oft von Promovierenden erledigt. Da diese nur für einen sehr begrenzten Zeitraum an der Universität tätig sind, muss sich nach deren Fortgang ein Nachfolger im uneinheitlichen Archivierungssystem seines Vorgängers zurechtfinden. Oftmals wird dabei neben der bloßen Aufgabenstellung und einer Musterlösung nicht viel Material festgehalten. Metainformationen wie Statistiken über den Ausfall der Aufgaben, Notizen aus Tutorenbesprechungen oder Material zur Durchführung der Übungen werden in vielen Fällen nicht archiviert.

Die Thematiken einer bestimmten Veranstaltung hingegen unterliegen eher geringen Fluktuationen, eine Wiederverwendung von angepassten Aufgaben kann daher sinnvoll sein. Zunächst steht ein Übungsleiter also vor dem Problem, den vorhandenen Datenbestand durchsuchen zu müssen. Ohne technische Unterstützung muss dies in den meisten Fällen von Hand geschehen. Finden sich zu einer Thematik mehrere Aufgabenstellungen, so kann ohne Metainformationen kaum eine plausible Aussage über den Erfolg der Durchführung getroffen werden. Die Auswahl einer geeigneten Aufgabe fällt also schwer.

Ein System zur effizienten Erstellung und Verwaltung von Aufgaben kann die Arbeitsabläufe vereinheitlichen und deutlich vereinfachen. Um Plattformunabhängigkeit und eine

zentrale Verwaltung zu ermöglichen, bietet es sich an, solch ein System als Webapplikation umzusetzen. Ein Browser als Client steht in jedem gängigen Betriebssystem zur Verfügung und die Administration des Applikationsservers kann durch den Systemadministrator des Instituts geschehen. Für die einzelnen Übungsleiter entsteht dadurch kein Zusatzaufwand im Vergleich zum jetzigen Vorgehen.

Ein System zur Aufgabenverwaltung soll längerfristig eingesetzt werden. Diese Eigenschaft bringt einige Randbedingungen mit sich. Die Software muss im Laufe der Verwendung möglicherweise gewartet oder an neue Anforderungen angepasst werden. Im Rahmen von Abschlussarbeiten durch Studenten entwickelte Software ist durch Dritte allerdings oft schwierig zu durchschauen. Der ursprüngliche Entwickler steht ebenfalls nur für einen begrenzten Zeitraum zur Verfügung. Das Design sollte also qualitativ hochwertig, leicht durchschaubar, mit wenig Aufwand anzupassen und hinreichend dokumentiert sein. Obgleich diese Eigenschaften generell auf jede entwickelte Software zutreffen sollten, werden geeignete Methoden und Technologien zur sicheren Erfüllung dieser Kriterien in Abschlussarbeiten selten eingesetzt.

1.2 Zielsetzung und Aufbau der Arbeit

Die Zielsetzung dieser Arbeit gliedert sich in zwei Teilbereiche. Zunächst sollen in allgemeiner Form geeignete Methoden, Konzepte und Technologien zur Entwicklung von Webanwendungen mit oben genannten Eigenschaften erarbeitet werden. Viele Softwareentwicklungs-Methodiken sind auf Gruppen aus mehreren Entwicklern zugeschnitten. Es ist daher geeignet zu berücksichtigen, dass in Abschlussarbeiten entwickelte Systeme meistens von Einzelpersonen umgesetzt werden. Im zweiten Teil wird mittels der erarbeiteten Methoden und Technologien eine Webapplikation zur Unterstützung der Arbeitsabläufe in der Aufgabenverwaltung und Auswertung umgesetzt.

Im folgenden Kapitel 2 wird zunächst die Agile Softwareentwicklung als mögliche Vorgehensweise diskutiert und dann werden im Detail ein spezielles Vorgehensmodell und dessen wesentliche Bestandteile beleuchtet. Außerdem wird ein geeigneter Architekturstil für Webapplikationen präsentiert, der sich zusätzlich positiv auf das Softwaredesign auswirken kann. Schließlich erfolgt die Vorstellung einer Menge von Werkzeugen zur effizienten Umsetzung der erarbeiteten Konzepte und Methoden.

In Kapitel 3 werden die wesentlichen Schritte von Konzeption und Umsetzung der Applikation dokumentiert. Insbesondere wird dabei auf Aspekte eingegangen, die sich noch nicht aus der im vorangehenden Kapitel dargestellten Architektur und Vorgehensweise ergeben, um die noch ausstehenden Designentscheidungen hinreichend zu beleuchten.

Abschließend erfolgen in Kapitel 4 eine Zusammenfassung der Resultate, eine eine Diskussion der Übertragbarkeit der Ergebnisse auf andere Bereiche und ein Ausblick auf weitere Arbeiten.

2 Methoden und Technologien

Dieses Kapitel führt in allgemeiner Form die verwendeten Methoden und Technologien ein. Zunächst wird eine Definition von **Agiler Softwareentwicklung** erarbeitet und daraufhin wird die spezielle Methode **Behaviour-Driven Development** mit ihren Bestandteilen erläutert. Im nächsten Abschnitt wird mit der **Resource-Oriented Architecture** ein architektureller Stil für Webapplikationen beleuchtet. Schließlich wird im dritten Teil die Umsetzung der vorher beschriebenen Methoden und Konzepte unter Verwendung des Frameworks **Ruby on Rails** und der Tools **RSpec** und **Cucumber** dargestellt.

2.1 Agile Softwareentwicklung

Agile Softwareentwicklung bezeichnet eine Menge von Methoden, Herangehensweisen und Prinzipien, die im Wesentlichen auf iterativer und inkrementeller Entwicklung beruhen. Der Begriff wurde erstmalig im Februar 2001 von der **Agile Alliance**[1] im Rahmen des **Manifesto for Agile Software Development** (Agiles Manifest, [Alli 01]) geprägt. Der Originaltext [Alli 01] definiert die Kernpunkte in prägnanter Weise:

> „We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> **Individuals and Interactions** over processes and tools
>
> **Working software** over comprehensive documentation
>
> **Customer collaboration** over contract negotiation
>
> **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.“

Es sei darauf hingewiesen, dass leichtgewichtige Ansätze mit direkterer Relevanz für die Endanwender zwar bevorzugt, dabei die traditionellen Techniken aber keineswegs gänzlich verworfen werden (vgl. [Chel 10, S. 115]).

Das Agile Manifest beinhaltet noch zwölf weitere, von den Kernpunkten inspirierte Charakteristiken, welche zur Differenzierung von Agilen Methoden gegenüber schwergewichtigen, traditionellen Prozessen herangezogen werden können. Viele Punkte des Dokuments beziehen sich auf die Organisation und die Interaktion von Entwicklerteams und werden hier nicht weiter herausgestellt. Einige Prinzipien lassen sich aber gut auf

[1] `http://www.agilealliance.org/`

die Arbeit als Einzelentwickler übertragen und werden auf Grundlage der Ausführungen von Robert C. Martin (vgl. [Mart 03, S. 3ff]) im Folgenden dargestellt.

Customer collaboration over contract negotiation. Es ist kaum möglich die Anforderungen einer Software derart zu beschreiben, dass ein Team von Entwicklern diese ohne weitere Rückfragen und Validierungen seitens des Kunden zufriedenstellend implementieren kann. Oft sind zu Beginn eines Projektes nicht einmal dem Kunden selbst alle Details und Anforderungen in voller Gänze bewusst. Diese Tatsache macht eine enge Zusammenarbeit zwischen Kunde und Entwickler in der Entwicklungsphase unabdinglich und eine zu detaillierte Vorausplanung möglicherweise obsolet.

Our highest priority is to satisfy the customer through early and continous delivery of valuable software. Eine empirische Untersuchung konnte eine Korrelation zwischen regelmäßig und frühzeitig durchgeführten Auslieferungen des Systems an den Kunden und der Qualität der finalen Softwareversion bestätigen [MacC 01]. Auslieferung bezeichnet in diesem Zusammenhang nicht die finale Übergabe der Anwendung, sondern die regelmäßige Übergabe lauffähiger Versionen während der Entwicklungsphase. Der Kunde kann diese bereits mit den bis dato existierenden Funktionen verwenden oder aber auch nur Rückmeldung dazu geben. Agile Vorgehensweisen präsentieren dem Anwender mit möglichst hoher Frequenz funktionierende Zwischenergebnisse und akquirieren dabei wertvolles Feedback.

Welcome changing requirements, even late in development. Konsequente Anwender Agiler Methoden schrecken nicht vor Änderung der Anforderungen zurück. Die Umsetzung angepasster oder neuer Anforderungen wird als positiv empfunden, da sie letztendlich zu einem besseren Endprodukt führt. Voraussetzung für diese Herangehensweise ist eine flexible und qualitativ hochwertige Softwarestruktur, so dass Änderungen lediglich einen kleinen Teil des Systems beeinflussen.

Continuous attention to technical excellence and good design enhances agility. Um schnell voran zu kommen ist es essentiell, das System stets so strukturiert und robust wie möglich zu halten. Um dieser Anforderung gerecht zu werden, ist ein gewisses Maß an Disziplin nötig. Unsauberer Quellcode wird zeitnah und vor allem vor Beendigung der aktuellen Tätigkeit bereinigt.

Simplicity – the art of maximizing the amount of work not done – is essential. Agile Entwickler streben stets den einfachsten möglichen Pfad an, der noch konsistent mit den Anforderungen und Zielen ist. Zur Zeit nicht benötigte Dinge, die zu einem späteren Zeitpunkt nützlich sein könnten, werden nicht implementiert. Sollte die Änderung irgendwann tatsächlich nötig werden, so lässt sie sich leicht durchführen.

Eine treffende und prägnante Zusammenfassung dieser Prinzipien liefern Subramanian und Hunt [Subr 06]:

> „Agile development uses feedback to make constant adjustments in a highly collaborative environment.“

Bekannte Beispiele für Agile Softwareentwicklung sind **SCRUM**[Schw 02], **Extreme Programming** [Beck 01b] und **Behaviour-Driven Development** (BDD)[Chel 10]. In den folgenden Abschnitten werden einige auch für Einzelentwickler anwendbare Teilaspekte dieser Herangehensweisen detailliert aufgegriffen.

2.1.1 Refactoring

Die dem **Refactoring** zugrunde liegenden Ideen und Konzepte tauchten bereits in den 1980er Jahren in der Smalltalk-Community auf. Der Begriff Refactoring wurde allerdings erst durch die Arbeiten von Johnson und Opdyke [Opdy 90, Opdy 92] zu Beginn der 1990er Jahre geprägt (vgl. [Fowl 99]). Für den weiteren Verlauf wird die Definition von Fowler zugrunde gelegt, welche auf den oben genannten Arbeiten basiert (vgl. [Fowl 99]):

> „Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs.“

Fowler unterscheidet dabei je nach Kontext zwischen der Verwendung des Begriffes als Nomen oder als Verb [Fowl 99, S. 54f]:

> **Refactoring**(noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.
>
> **Refactor**(verb): to restructure software by applying a series of refactorings without changing its observable behaviour.

Ein einfaches Beispiel für Refactoring ist **Extract Method** (vgl. [Fowl 99, S. 110f]). Listing 1 zeigt eine simple Java-Methode, auf die Extract Method angewendet wird.

```
1 void printOwing(double amount){
2   printBanner();
3
4   //print details
5   System.out.println("name: " + _name);
6   System.out.println("amount" + amount);
7 }
```

Listing 1: Java-Methode vor der Anwendung von Extract Method

Die grundlegende Idee ist es, Fragmente, die zusammen gehören und einen Zweck erfüllen, in eine Methode auszulagern, deren Name diesen Zweck hinreichend beschreibt und Kommentare stellenweise überflüssig macht. Das Ergebnis der Anwendung ist in Listing 2 dargestellt.

```
1 void printOwing(double amount){
2   printBanner();
3   printDetails(amount);
4 }
5
6 void printDetails(double amount){
7   System.out.println("name: " + _name);
8   System.out.println("amount" + amount);
9 }
```

Listing 2: Java-Methode nach der Anwendung von Extract Method

Fowler betitelt Extract Method als das am häufigsten eingesetzte Refactoring. Vorteile sieht er vor allem in der feineren Granularität der Struktur, welche die Wahrscheinlichkeit für Wiederverwendung erhöht und beispielsweise das Überschreiben von Methoden vereinfachen kann. Außerdem verbessert sich die Lesbarkeit von Methoden höherer Abstraktionsniveaus, wie es bei `printOwing(double amount)` im obigem Beispiel der Fall ist.

Mittlerweile existiert eine große Menge identifizierter Refactorings, von denen viele in dieser Arbeit eingesetzt worden sind. Da eine vollständige Auflistung den Rahmen sprengen würde, sei für weitere Beispiele auf die einschlägige Literatur verwiesen ([Fowl 99, Mart 09]).

Insgesamt kommt Fowler zu dem Schluss, dass regelmäßiges Refactoring zu besseren, leichter verständlichen Designs führt und in gewissen Situationen sogar die Produktivität steigern kann. Ist erst einmal eine qualitativ hochwertige Basis vorhanden, so ist der Aufwand für das Hinzufügen von Funktionalität oder das Einarbeiten von Änderungswünschen deutlich einfacher. Dadurch können auf lange Sicht Zeit und somit auch Kosten eingespart werden. Natürlich gelten diese Aussagen nicht für alle Situationen. Steht ein Projekt kurz vor dem Fertigstellungstermin, so reichen die positiven Effekte möglicherweise nicht mehr aus, um den Aufwand zu rechtfertigen. Ein weiterer Nachteil kann im Bezug auf Performanz angeführt werden. In der Softwareentwicklung muss meist ein guter Mittelweg zwischen Performanz und Abstraktion gefunden werden, da sich diese beiden Faktoren in der Regel antiproportional zueinander verhalten. Fowler führt hier als Gegenargument an, dass ein wohlstrukturiertes, modularisiertes und verständliches System viel einfacher an den wirklich kritischen Stellen auf Performanz zu optimieren sei. Seine auf Erfahrung aus realen Projekten basierende Empfehlung lautet, zuerst die Struktur des Systems in den Griff zu bekommen und sich anschließend Gedanken über Optimierungen an geeigneter Stelle zu machen.

Der richtige Einsatz von Refactoring führt im Allgemeinen zu qualitativ hochwertigerem, flexiblerem Code und erweist sich damit als ein wertvolles Werkzeug, um die geforderte Flexibilität der Agilen Softwareentwicklung zu erreichen.

2.1.2 Test-Driven Development

Test-Driven Development (TDD) ist eine Entwicklungsmethode, in der die Tests vor dem zu testenden Quellcode geschrieben werden [Beck 03]. Die Tests werden im Gegensatz zum traditionellen **Wasserfallmodell** [Royc 70, Boeh 84] bereits während der Entwicklungsphase hinzugefügt. Nachdem genug Code geschrieben wurde, um ein erfolgreiches Durchlaufen eines momentan bearbeiteten Tests zu erreichen, wird die interne Codestruktur durch Refactoring verbessert ehe zum nächsten Test übergegangen wird. Abbildung 2.1 stellt den Ablauf schematisch dar. Dieser Zyklus wird wiederholt, bis die gewünschte Funktionalität vollständig implementiert ist. In regelmäßigen Abständen müssen zusätzlich die gesamten Tests des Systems ausgeführt werden, um eine ungewollte Beeinflussung anderer Systemkomponenten auszuschließen beziehungsweise zu erkennen. Dies sollte spätestens vor dem Einchecken in die Versionskontrolle geschehen [Bhat 06].

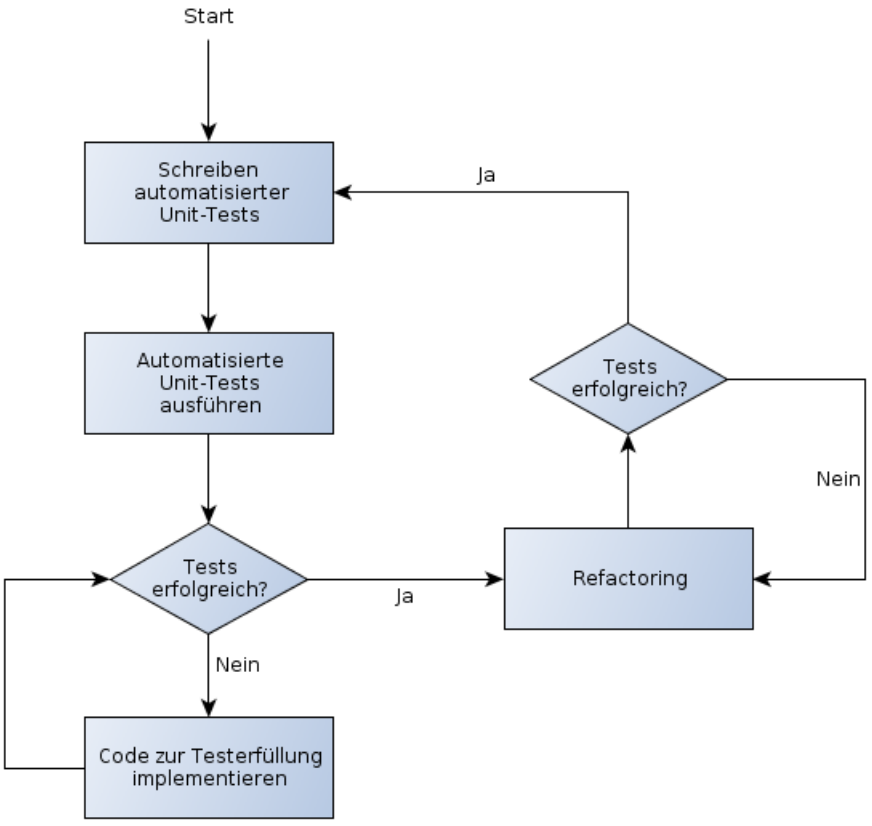


Abb. 2.1: TDD-Zyklus als Flowchart

Test-Driven Development erlangte große Popularität als wesentlicher Bestandteil des Extreme Programming. Tatsächlich kamen erste ähnliche Ideen aber bereits in den 1960er Jahren im NASA Project Mercury auf (vgl. [Sini 07, Beck 01b, Larm 03]).

In der Theorie proklamieren die Verfechter des TDD viele positive Effekte. Code wird nur geschrieben, um bereits vorhandene Tests zu erfüllen. Bei konsequenter Einhaltung dieses Prinzips führt TDD zu einer verbesserten Testabdeckung [Aste 03]. Weiterhin

kann das Design durch sehr lose **Kopplung** (Abhängigkeit der Module untereinander) und starke **Kohäsion** (Abbildung genau einer Verantwortlichkeit auf ein Modul) innerhalb des Systems erheblich vereinfacht werden [Beck 01a]. Wenn der Test zuerst geschrieben wird, ist der Entwickler gezwungen, sich zunächst Gedanken über die Schnittstelle des zu schreibenden Codes zu machen [Subr 06]. Dabei kann sowohl zu enge Kopplung als auch zu starke Kohäsion frühzeitig erkannt werden, da in diesen Fällen das Testen kompliziert oder aufwändig ist. Das daraus resultierende, qualitativ hochwertige Design wirkt sich im weiteren Verlauf durch seine Flexibilität und Robustheit positiv auf die Produktivität bei Erweiterungen und Änderungen aus.

Siniaalto und Abrahamsson [Sini 07] geben einen Überblick über vorhandene empirische Daten bezüglich der Effekte des Einsatzes von TDD auf Designqualität, Produktivität, Fehlerraten und Testabdeckung im Vergleich zum Wasserfallmodell und der **iterative test-after**-Methode. Die Studien werden zunächst in die Kategorien **Industry** (industrieller Kontext, reale Industrieprojekte), **Semi-industry** (semiindustrieller Kontext, Industrie-Entwickler gemeinsam mit studentischen Entwicklern, experimentelle Aufgabenstellungen) und **Academic** (Akademischer Kontext, studentische Entwickler, experimentelle Aufgabenstellungen) eingeteilt. Je nach Kontext kommen die Studien zu abweichenden Ergebnissen. Die Studien im industriellen Kontext [Bhat 06, Lui 04, Maxi 03, Damm 05] liefern als Ergebnisse einen um 15-35% erhöhten Zeitaufwand, eine stark reduzierte Fehlerrate von bis zu 50%, erhebliche Qualitätssteigerungen und eine generell verbesserte Testabdeckung. Im semiindustriellen Kontext [Mull 06, Canf 06, Geor 04, Gera 04, Abra 05] fällt der Effekt der Qualitätssteigerung weniger stark aus. Die Ergebnisse bezüglich der verbesserten Testabdeckung stimmen mit denen des industriellen Kontextes überein. Als mögliches Problem konnte im semiindustriellen Kontext ein falsches Gefühl von Sicherheit durch die Unit-Tests aufgedeckt werden, welches zu einer größeren Fehlerzahl in den **Acceptance Tests** (automatisierte Akzeptanztests aus Anwendersicht, siehe Abschnitt 2.1.3) führen kann. Die Studien im akademischen Kontext [Janz 06, Kauf 03, Mull 02, Panc 03, Erdo 05, Stei 01, Edwa 04] fördern ebenfalls Verbesserungen bezüglich der Qualität zu Tage, allerdings in geringerem Ausmaß als in den anderen Kontexten. Obwohl im akademischen Kontext eine größere Skepsis gegenüber TDD vorherrschte, wurden durchweg positive Auswirkungen auf die Produktivität festgestellt.

Schließlich führten Siniaalto und Abrahamsson eine Studie im semiindustriellen Kontext durch, die den Fokus auf die Beeinflussung des Designs legt. Auch hier kamen sie zu dem Schluss, dass die Testabdeckung sich stark verbessert. Weiterhin wird festgestellt, dass der Einfluss von TDD auf das Design, bezogen auf Kohäsion und Modulkopplung sehr stark von der Vorerfahrung der Entwickler abhängig ist. Bei unerfahrenen Programmierern sind nur schwache Effekte zu verzeichnen.

Es scheint, als würden die beobachteten positiven Effekte des TDD mit zunehmender Entfernung von der Praxis immer geringer ausfallen. Zum einen könnte dies in der Art der untersuchten Projekte begründet liegen, da die Effekte sich bei künstlich erzeugten Experiment-Problemen scheinbar nicht so stark äußern, wie bei realen Softwareprojekten. Zum anderen liegen bei den studentischen Versuchspersonen vermutlich deutlich weniger Erfahrungen im Bezug auf Softwareentwicklung vor. Die Entwickler benötigen also eine gewisse Vorerfahrung oder müssen sich intensiv in die notwendigen Thema-

tiken, wie zum Beispiel Softwaredesign und Refactoring, einarbeiten. Gleichzeitig darf sich der Entwickler nicht zu sehr auf die entstandenen Unit-Tests verlassen, sondern sollte stets die Acceptance Tests im Hinterkopf behalten, damit die Software letztendlich allen Anforderungen gerecht wird. Zusammenfassend lässt sich also sagen, dass TDD zahlreiche positive Auswirkungen haben kann und damit ein wichtiges Werkzeug für die Agile Softwareentwicklung darstellt.

2.1.3 Behaviour-Driven Development

Behaviour-Driven Development (BDD) begann ursprünglich als Neuausrichtung des Test-Driven Developments, um Anfängern die Verwendung von TDD als Designtool zu erleichtern. Mittlerweile hat sich BDD aber zu einer vollwertigen, agilen Entwicklungsmethode entwickelt. Dabei werden mehrere Methoden aus anderen agilen Vorgehensweisen entliehen.

Chelimsky et al. [Chel 10] liefern eine Kurzdefinition und die drei zentralen Prinzipien des BDD:

> „*Behaviour-Driven Development is about implementing an application by describing its behaviour from the perspective of its stakeholders.*
>
> **Enough is enough** Up-front planning, analysis, and design all have a diminishing return. We shouldn't do less than we need to get started, but any more than that is wasted effort. This also applies to process automation. Have an automated build and deployment, but avoid trying to automate everything.
>
> **Deliver stakeholder value** If you are doing something that isn't either delivering value or increasing your ability to deliver value, stop doing it, and do something else instead.
>
> **It's all behavior** Whether at the code level, the application level, or beyond, we can use the same thinking and the same linguistic constructs to describe behavior at any level of granularity.“

Die wesentlichen eingesetzten Methoden zur Berücksichtigung dieser Prinzipien sind Test-Driven Development (siehe Abschnitt 2.1.2), **Domain-Driven Design** (DDD) [Evan 04] und **Acceptance Test-Driven Planning** (ATDP). Die Umsetzung erfolgt in Form von **Iterationen**. Iteration bezeichnet einen im Vorfeld festgelegten Zeitabschnitt, in dessen Rahmen eine gewisse Menge von Funktionalitäten implementiert wird und an dessen Ende stets eine lauffähige Version der Software zur Verfügung steht (vgl. [Chel 10]). Aus dem Domain-Driven Design wird hauptsächlich die **Ubiquitous Language** entliehen. Die Idee ist dabei die Festlegung eines gemeinsamen Vokabulars, auf dessen Grundlage Anwender und Entwickler bezüglich der zu erstellenden Applikation kommunizieren, um Missverständnisse und Unklarheiten bereits im Vorfeld auszuräumen. Acceptance Test-Driven Planning basiert auf **Acceptance Test-Driven Development** (ATDD) aus dem Extreme Programming. Dieses spezifiziert lediglich, dass Acceptance Tests vor der Implementation in Zusammenarbeit mit dem Anwender entstehen, der genaue Zeitpunkt wird dabei nicht festgelegt. ATDP hingegen legt zusätzlich

fest, dass direkt vor Beginn einer Iteration die Akzeptanzkriterien für die in der Iteration umzusetzenden Funktionalitäten spezifiziert werden. Das Ziel ist es, relevante Details zur besseren Präsenz der Anforderungen und effizienteren Planung direkt vor der betreffenden Iteration zu aggregieren.

Abbildung 2.2 stellt den Ablauf eines mittels BDD realisierten Projektes schematisch dar.

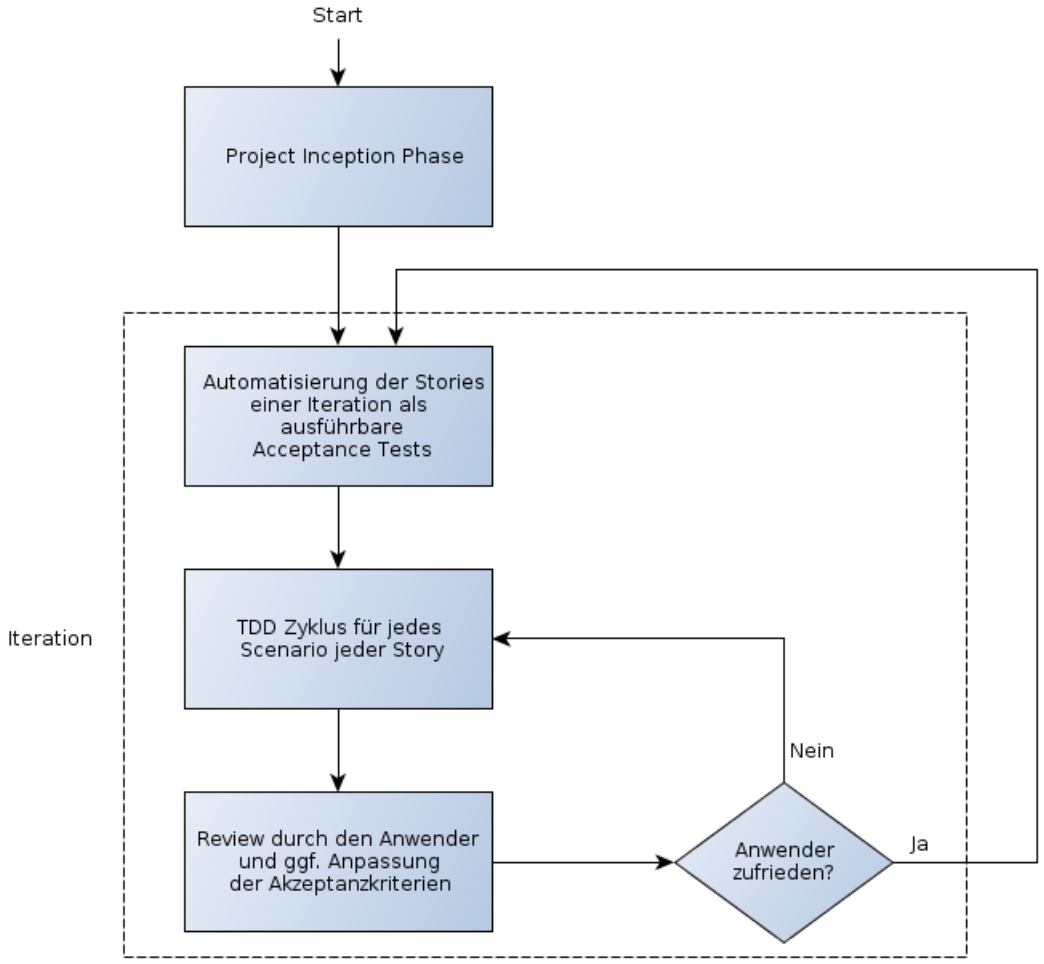


Abb. 2.2: BDD-Vorgehen als Flowchart. Für Details zum TDD-Zyklus siehe Abb. 2.1

Zu Beginn des Projektes findet die **Project Inception Phase** statt, in der die Anwender dem Entwicklerteam einen abstrakten Eindruck bezüglich ihrer Anforderungen in Form von **Feature Sets** vermitteln. Ein Beispiel für ein Feature Set könnte die Anforderung *Creating an assignment sheet* sein. In Zusammenarbeit mit dem Endanwender werden die Feature Sets dann in feingranulare **Stories** zerlegt. Eine Story enthält einen Titel, eine Erzählung und eine Menge von Akzeptanzkriterien. Der Titel dient als Referenz für die weitere Kommunikation. In der Erzählung wird geschildert, was aus welchem Grund geschehen soll. Häufig wird hier das **Connextra Format** der Form *As a [stakeholder] I want [feature] so that [benefit]* eingesetzt. Akzeptanzkriterien werden in Form von **Scenarios** (Szenarien) formuliert, die ihrerseits wiederum aus mehreren **Steps** bestehen.

Eine Story innerhalb des obigen Feature Set Beispiels könnte so aussehen:

> **Story: Searching assignments**
> As a teaching assistant
> I want to search for assignments with a search form
> So that I can see easily which assignments have been used in the preceding years

Ein mögliches Scenario innerhalb dieser Story könnte folgende Gestalt haben:

> **Scenario: Performing a search**
> **Given** I am logged in as teaching assistant
> **When** I go to the assignments page
> **And** submit a search phrase into the search form
> **Then** I want to see the assignments matching the search phrase

In der Umsetzungsphase werden dann nach weiterer Rücksprache mit dem Anwender vor der jeweiligen Iteration automatisierte Akzeptanztests für die in der Iteration zu bearbeitenden Stories implementiert. Für jeden Step jedes Szenarios wird der TDD-Zyklus (vgl. Abschnitt 2.1.2) durchgeführt, bis alle Akzeptanztests der Story fehlerfrei durchlaufen. Sind alle Stories der Iteration umgesetzt, so wird das Ergebnis dem Endnutzer präsentiert und ihm wird die Möglichkeit gegeben, Änderungswünsche anzubringen. Gegebenenfalls werden entsprechende Korrekturen an den Stories vorgenommen. Sollten sich Änderungswünsche ergeben haben, so werden neben weiteren Stories auch die Änderungen im Rahmen der folgenden Iteration duchgeführt.

Mit diesem Vorgehen lässt sich die möglicherweise unbewusste Vernachlässigung der Acceptance Tests (vgl. Abschnitt 2.1.2) im Rahmen von reinem TDD umgehen. Der TDD-Zyklus findet überhaupt nur für vorher als Acceptance Test formulierte Funktionalitäten statt und zwar genau so lange, bis die Anforderungen des Anwenders erfüllt sind. Dies gewährleistet gleichzeitig, dass keine überflüssigen Funktionalitäten implementiert werden, sondern nur Funktionsumfang mit direktem Wert für den Endnutzer umgesetzt wird. Die aufgezeigten Vorteile des Test-Driven Development haben natürlich weiterhin Gültigkeit. Außerdem bekommen die Entwickler aufgrund der zu erfüllenden Acceptance Tests ein viel besseres Gefühl dafür, was die Applikation als Ganzes überhaupt leisten soll und fokussieren sich so auf die Implementation der eigentlichen Anforderungen. Zusätzlich liefern die Endanwender im Verlauf der Entwicklung direktes Feedback zu den kürzlich implementierten Funktionen. Dies bietet den Vorteil, dass den Entwicklern die Funktionen samt ihrer Umsetzung noch präsent sind und sich Änderungen so viel leichter durchführen lassen als erst am Ende des gesamtem Projektes.

Abschließend lässt sich sagen, dass Behaviour-Driven Development allen für Einzelentwickler relevanten Aspekten der Agilen Softwareentwicklung (vgl. Abschnitt 2.1) genügt und bei korrekter Anwendung in einem flexiblen, einfach erweiterbaren und qualitativ hochwertigem Design resultiert, welches die Anforderungen des Endanwenders in hinreichender Weise umsetzt.

2.2 Resource-Oriented Architecture

Der Begriff **Resource-Oriented Architecture** (ROA) wurde im Jahr 2007 von Richardson und Ruby geprägt [Rich 07]. Er bezeichnet eine Menge spezifischer Richtlinien zur Implementierung von auf **Representational State Transfer** (REST, [Fiel 00]) basierenden Architekturen für Webservices und Webapplikationen. Die Grundidee besteht aus der Rückbesinnung auf die Konzepte und Ideen, die maßgeblich am Erfolg des Webs beteiligt waren. Im Folgenden werden die für diese Arbeit relevanten Eigenschaften zusammengefasst dargestellt.

Resources (Ressourcen) sind das zentrale Konzept der ROA. Alles, was wichtig genug ist, um als eigenständige Entität referenziert zu werden, nimmt in der ROA den Platz einer Ressource ein. Sämtliche Dinge, für die eine Manipulation beliebiger Art durch die Nutzer der Applikation vorgesehen ist, werden auf Ressourcen abgebildet. Im Kontext der in dieser Arbeit zu entwickelnden Applikation wären mögliche Beispiele *AssignmentSheet*, *Assignment* oder *User*.

Um Ressourcen gezielt beeinflussen zu können, ist ein Konzept zur **Adressabilty** (Adressierbarkeit) nötig. Jeder Anfrage wird ein eindeutiger **Universal Resource Identifier** (URI) zugewiesen, um sie referenzieren zu können. Häufig wird in der Praxis dabei eine artifizielle ID zur Gewährleistung der Eindeutigkeit eingeführt. Ein URI für ein Aufgabenblatt mit der ID 3 könnte `http://www.example.com/assignment_sheets/3` sein. Jede Ressource benötigt mindestens einen URI, kann aber auch mehrfach adressierbar sein. Um einen klaren Zusammenhang zwischen URI und der referenzierten Ressource herzustellen, sollte die URI so deskriptiv wie möglich sein. Weiterhin ist es sinnvoll, URIs für unterschiedliche Ressourcen nach demselben Schema zu konstruieren, um individuelle Festlegungen für jeden Ressourcen-Typ zu vermeiden.

Ein weiteres wichtiges Konzept ist **Statelessness** (Zustandslosigkeit). Jede Anfrage an den Server muss in Isolation ausführbar sein und alle zur Verarbeitung nötigen Daten mitliefern. Wenn irgendeine Art von Serverstatus durch den Client manipulierbar sein soll, dann muss dieser in Form einer Ressource zugänglich gemacht werden. Der Begriff **Zustand** ist in diesem Kontext nicht ganz eindeutig. Es muss unterschieden werden zwischen **Ressourcenzustand** und **Applikationszustand**. Ressourcenzustand bezeichnet den globalen, für alle Nutzer sichtbaren, Zustand einer Ressource. Applikationszustand hingegen bezeichnet den spezifischen Clientzustand eines einzelnen Nutzers, ein Beispiel ist die Historie besuchter Seiten im Browser. Die meisten Webapplikationen müssen Zustände von Ressourcen speichern, um ihren Daseinszweck zu erfüllen. Es wäre zum Beispiel absurd, wenn ein Client einer Applikation zur Foto-Verwaltung in jeder Anfrage alle seine Fotos mitschicken müsste. In der ROA muss also die serverseitige Speicherung des Applikationszustands eines Nutzers vermieden werden, das Speichern von Ressourcen-Zuständen ist legitim. Würde ein Server in Abhängigkeit von Applikationszustand unerschiedliche Aktionen durchführen, dann könnten die Anfragen nicht mehr in Isolation verarbeitet werden. Im Webkontext führt die inkonsequente Umsetzung dieser Anforderung häufig dazu, dass der "Zurück-Button" des Browsers nicht mehr wie erwartet funktioniert. Weitere Nachteile wären starke Einschränkungen im Bezug auf **Caching** und **Load Balancing**. Caching lässt sich viel effizienter durchführen, wenn lediglich die Anfrage betrachtet werden muss und keine weiteren Informationen

des Servers nötig sind. Beim Load Balancing lässt sich durch Zustandslosigkeit vermeiden, dass mehrere Server untereinander den Applikationsstatus replizieren müssen. Das **Hypertext Transfer Protocol** (HTTP) ist von sich aus bereits zustandslos. Häufig wird diese Zustandslosigkeit aber umgangen, da sich die Anfragen damit oft vereinfachen lassen. In gewissen Fällen kann es dennoch sinnvoll sein, die Nachteile für eine Vereinfachung in Kauf zu nehmen. Generell sollte aber eine maximale Zustandslosigkeit angestrebt werden, um das Protokoll einfach und konsistent zu halten.

Daten zur Manipulation von Ressourcen werden in Form von **Representations** (Repräsentationen) übermittelt. Ressourcen sind die Quellen für Repräsentationen und die Repräsentationen enthalten eine Menge von Daten, die eine Teilmenge des Ressourcenstatus beschreiben. Eine mögliche Repräsentation einer Ressource könnte eine Datei im XML- oder JSON-Format sein, welche die serialisierten Attribute in entsprechender Syntax enthält. Die Listings 3 und 4 zeigen Beispielrepräsentationen eines Nutzers.

```xml
<user>
  <name>ssorg</name>
  <first-name>Susi</first-name>
  <last-name>Sorglos</last-name>
  <birthday type="date">
    1994-10-30
  </birthday>
  <matr-nr type="integer">
    123456
  </matr-nr>
</user>
```

Listing 3: XML-Repräsentation

```json
{
 "birthday" : "1994-10-30",
 "first_name" : "Susi",
 "last_name" : "Sorglos",
 "matr_nr" : 123456,
 "name" : "ssorg"
}
```

Listing 4: JSON-Repräsentation

Der Browser erhält als Repräsentation meist eine HTML-Seite. Ein vom Nutzer ausgefülltes Formular sendet ebenfalls die Repräsentation einer Ressource als eine Menge von Anfrageparametern an den Server. Client und Server müssen jeweils festlegen, welche Art von Repräsentation gewünscht, beziehungsweise gefordert, ist. Auf Clientseite kann das gewünschte Format am einfachsten durch Anhängen an die URI übermittelt werden: `http://www.example.com/users/3.xml`.

Das **Uniform Interface** (Einheitliche Schnittstelle) legt schlussendlich fest, wie dem Server übermittelt wird, was genau mit der über ihren URI identifizierten Ressource geschehen soll. Zu diesem Zweck wird den HTTP-Verben GET, PUT, POST und DELETE jeweils eine Semantik zugeordnet. Mit der URI wird die Ressource identifiziert, auf der operiert werden soll, das HTTP-Verb gibt die Aktion an. Diese Festlegung ist ziemlich intuitiv, da sie analog zu Nomen und Verben im natürlichen Sprachgebrauch ist. Werden die Ressourcen einer Webapplikation in einer Datenbank vorgehalten, so könnte die in Tabelle 2.1 dargestellte Abbildung der HTTP-Verben auf die grundlegenden Datenbankoperationen eine durchaus sinnvolle Semantik darstellen.

Allgemein liefert eine GET-Anfrage auf eine URI eine Repräsentation der gewünschten Ressource zurück. Würde man eine DELETE-Anfrage auf die gleiche URI absetzen,

POST	→	CREATE
GET	→	READ
PUT	→	UPDATE
DELETE	→	DELETE

Tabelle 2.1: Semantik der `HTTP`-Verben

hätte dies das Löschen der Ressource zur Folge. Der Server antwortet auf ein DELETE meistens nur mit einem HTTP-Status-Code, um Erfolg oder Misserfolg anzuzeigen. Anfragen mit der Methode PUT können unterschiedliche Bedeutungen haben. Eine PUT-Anfrage auf die URI einer existierenden Ressource führt zur Ersetzung dieser Ressource durch die mit der Anfrgae mitgelieferte Repräsentation. Verwendet die PUT-Anfrage eine URI, die noch keine Ressource referenziert, so wird die mitgelieferte Repräsentation unter dieser URI abgelegt. POST-Anfragen dienen nach der HTTP/1.1 Spezifikation [Fiel 99] der Annotation existierender Ressourcen, der Übergabe von Formulardaten an den datenverarbeitenden Prozess und der Erweiterung einer Datenbank durch eine Anhängeoperation. Im Kontext der ROA wird POST meistens verwendet, um untergeordnete Ressourcen zu erzeugen. Die Ressource `http://www.example.com/users` könnte die Gesamtheit aller Nutzer repräsentieren. Eine POST-Anfrage auf diese Ressource würde die übermittelte Repräsentation als neue Nutzerressource interpretieren, die an die vorhandene Liste angehängt werden soll. Dabei wird die URI der neuen Ressource an den Client zurückgeliefert, beispielsweise `http://www.example.com/users/3`. Tabelle 2.2 zeigt eine ROA-konforme Umsetzung einer einheitlichen HTTP-Schnittstelle für Ressourcen vom Typ *User*. Daraus wird deutlich, dass keineswegs jede Ressource auf sämtliche HTTP-Verben reagieren muss.

URI	GET	PUT	POST	DELETE
www.example.com/users	Liste aller Nutzer	-	Nutzer erzeugen	-
www.example.com/users/3	Repräsentation von User 3	Update User 3	-	User 3 Löschen

Tabelle 2.2: Beispiel für die Umsetzung der einheitlichen Schnittstelle

Durch konsequente Verwendung der einheitlichen Schnittstelle erhält man zusätzlich und ohne weiteren Aufwand die Eigenschaften **Safety** (Sicherheit) und **Idempotence** (Idempotenz) für einige der HTTP-Verben. Anfragen vom Typ GET sind sicher. Sie dienen ausschließlich zum Lesen von Daten und dürfen niemals den Status einer Ressource verändern. PUT und DELETE sind idempotente Operationen. Sie können beliebig oft hintereinander angewendet werden und haben immer denselben Effekt. Führt man eine DELETE-Anfrage auf eine Ressource durch, so ist sie nach Ende der Anfrageverarbeitung gelöscht. Führt man die Anfrage nochmals durch, bleibt die Ressource weiterhin gelöscht. Gleiches gilt für die Manipulation einer Ressource mit PUT. Die Durchführung einer PUT-Anfrage zur Aktualisierung einer Ressource ersetzt diese durch die übermittelte Repräsentation. Führt man die Anfrage nochmals durch, so hat dies keinen Effekt mehr, da sie bereits die Daten aus der übersendeten Repräsentation inne hat. Hier wird auch klar, warum es notwendig ist, stets Repräsentationen auszutauschen. Per PUT

versendete Anweisungen der Form *inkrementiere die Matrikelnummer um eins* wären nicht mehr idempotent. POST-Anfragen können aufgrund ihrer Natur weder sicher noch idempotent sein. Sie werden verwendet, um neue Ressourcen zu erzeugen, ohne deren URI im Voraus zu kennen. Mehrere POST-Anfragen mit denselben Daten würden also auch mehrere Ressourcen anlegen. Mögliche Fehler in diesem Zusammenhang abzufangen, liegt in der Verantwortung der Applikation. Sicherheit und Idempotenz liefern in gewissem Rahmen die Möglichkeit, verlässliche Anfragen über ein unverlässliches Netzwerk durchzuführen. Erhält ein Browser nach einer bestimmten Zeit keine Antwort auf eine getätigte GET-Anfrage, so wird er einfach eine neue Anfrage starten und nicht weiter auf die Antwort der ersten Anfrage warten. Dennoch kann die erste Anfrage ihr Ziel erreicht haben. Sicherheit und Idempotenz schließen in solchen Fällen unerwünschte Effekte aus.

Insgesamt können die Konzepte der Resource-Oriented Architecture bei konsequenter Anwendung mehrere nützliche Nebeneffekte haben. Die Abbildung der Geschäftslogik auf die ROA-konforme Manipulation von Ressourcen zwingt zum Design lose gekoppelter Module, die zuverlässig und möglichst unabhängig voneinander operieren. Weiterhin lässt sich aufgrund der einheitlichen Schnittstelle die Verarbeitung sehr gut generalisieren. Als festzulegende Parameter verbleiben die URI-Konstruktionsschemata, die Menge der Ressourcen und deren Repräsentationen. Im späteren Verlauf lassen sich ohne großen Aufwand weitere Repräsentationsformate hinzufügen. Entwickelt man zum Beispiel eine Webapplikation, die zunächst nur über den Browser genutzt werden soll als ROA, so lässt sich eine XML- oder JSON-API mühelos durch Ergänzung weiterer Repräsentationsformate ohne große Änderungen an der sonstigen Applikationsstruktur integrieren. Wegen der Zustandslosigkeit lässt sich das System bei Bedarf durch Zuschaltung weiterer Server und geeignetes Load Balancing skalieren, da die Server im Idealfall nicht untereinander kommunizieren müssen.

2.3 Ruby on Rails

Ruby on Rails ist ein Open Source Framework zur Entwicklung von Webapplikationen in der Programmiersprache Ruby. Version 1.0 wurde im Jahr 2005 von David Heinemeier Hansson veröffentlicht. Wichtige Elemente der Frameworkarchitektur sind das Designpattern **Model-View-Controller** (MVC), der Grundsatz **Don't Repeat Yourself** (DRY) und das Paradigma **Convention over Configuration** (COC). In dieser Arbeit wird die im August 2011 erschienene Version 3.1 des Frameworks eingesetzt. Zum besseren Verständnis späterer Codebeispiele erfolgt zunächst eine kurze Einführung in die Sprache Ruby. Die weiteren Abschnitte beleuchten die wesentlichen Teile des Frameworks, die zur Umsetzung der in den Abschnitten 2.1 und 2.2 dargelegten Konzepte verwendet worden sind. Eine vollständige Darstellung würde den Rahmen dieser Arbeit sprengen, daher sei dafür auf die diesem Abschnitt zugrunde liegende Fachliteratur verwiesen [Ruby 11, Fern 10, Perr 10, Pyte 10, Thom 09].

2.3.1 Ruby

Ruby ist eine dynamische, objektorientierte, general-purpose Programmiersprache. Die Sprache entstand Mitte der 1990er Jahre in Japan und wurde entwickelt von Yukihiro Matsumoto. Neben der Objektorientierung werden noch die Programmiersprachenparadigmen **Funktionale Programmierung** und **Reflection** unterstützt. Weiterhin bringt die Sprache einen Garbage-Collector mit, der Entwickler muss sich also nicht um die Speicherverwaltung kümmern. Die in dieser Arbeit verwendete Ruby-Version ist 1.9.2-p290. Neben der genutzten Referenzimplementierung MRI/YARV[2] existieren noch einige weitere Implementationen. Die Java-Implementierung JRuby[3] bietet nahtlose Integration in nahezu beliebige Java-Umgebungen. Mit MacRuby [4] bietet sich die Möglichkeit, sämtliche Bibliotheken der Appleplattformen verwenden zu können.

Klassen, Methoden und Objekte

In Ruby existieren keine Primitiven Datentypen wie `int` oder `char` in Java. Auch die Klassen der Objekte selbst sind wieder Objekte, die zur Laufzeit manipuliert werden können. Listing 5 zeigt dazu einige Beispiele. Das Zeichen `#` markiert einen Kommentar, der die Rückgabe der dargestellten Methodenaufrufe beschreibt.

```
1 1.class             # liefert: Fixnum Klassenobjekt
2 1.0.class           # liefert: Float Klassenobjekt
3 "Ein String".class  # liefert: String Klassenobjekt
4 1.class.class       # liefert: Class Klassenobjekt
```

Listing 5: Beispiel zu Datentypen

Rubycode innerhalb einer Datei wird beim Laden direkt interpretiert, der Code in Listing 5 stellt bereits ein lauffähiges Rubyskript dar. Listing 6 zeigt eine einfache Klassendefinition. Sie definiert die Klasse `Student`, die von der Basisklasse `Person` erbt. In der Wurzel der Vererbungshierarchie befindet sich die Klasse `Object`. Wird keine Basisklasse angegeben, erben definierte Klassen implizit von `Object`. Die Methode `initialize` in Zeile 3 wird nach dem Allokieren des Speichers für das Objekt aufgerufen und dient zur Herstellung eines sinnvollen Startzustandes. Sie verhält sich analog zu einem Konstruktor in Java. Im Beispiel kann ihr ein Name übergeben werden, den sie der Instanzvariablen `@name` zuweist. Die Methode `say_name` in Zeile 9 gibt diesen Namen bei Aufruf auf der Kommandozeile aus.

Auch Klassendefinitionen sind nur Skripte, die beim Laden ausgeführt werden und erst zur Laufzeit die Klasse definieren. Dies ermöglicht insbesondere auch die Wiederöffnung bereits vorhandener Klassen und damit das Hinzufügen weiterer Methoden, wie in Listing 7 dargestellt. Diese Erweiterung wird dem System durch Laden der beinhaltenden Datei hinzugefügt.

[2] `http://www.ruby-lang.org/`
[3] `http://www.jruby.org/`
[4] `http://www.macruby.org/`

```ruby
class Student < Person
  # Konstruktor
  def initialize(name)
    # Setzen einer Instanzvariable
    @name = name
  end

  # Methodendefinition einer Instanzmethode
  def say_name
    puts @name
  end
end

willi = Student.new("Willi Wacker")
willi.say_name # gibt "Willi Wacker" auf der Kommandozeile aus
```

Listing 6: Klassendefinition

```ruby
class String
  def hellotize
    "Hello #{self}!"
  end
end

puts "Test".hellotize
# Gibt "Hello Test!" auf der Kommandozeile aus.
```

Listing 7: Erweiterung der Stringklasse

Das Schlüsselwort `self` liefert eine Referenz auf die aktuelle Instanz zurück, ähnlich wie `this` in Java. Der `#{}`-Operator dient zur Interpolation eines Strings in einen anderen. Innerhalb der geschweiften Klammern dürfen beliebige Rubyausdrücke stehen. Am Ergebnis dieses Ausdrucks wird vor der Interpolation die Methode `to_s` aufgerufen, welche eine String-Repräsentation des jeweiligen Objektes zurückliefert. Weiterhin ist an diesem Beispiel erkennbar, dass in Ruby implizit der Wert der letzten Anweisung in einer Methode als Rückgabewert verwendet wird. Es ist zwar möglich, explizit das Schlüsselwort `return` zu verwenden, in den meisten Fällen besteht dafür allerdings keine Notwendigkeit.

Datenstrukturen

Die wichtigsten Datenstrukturen in Ruby sind das **Array** und der **Hash**. Ein Array verhält sich ähnlich wie eine `List` in Java, bietet jedoch eine größere Dynamik, da beliebige Datentypen eingefügt werden können. Listing 8 zeigt einige Beispiele zur Erzeugung von Arrays. Die Datenstruktur Hash ist vergleichbar mit `Map` aus Java. Sie dient der

Speicherung von Schlüssel-Wert Paaren beliebiger Typen. Meist werden **Symbole** als Schlüssel verwendet, um nicht immer wieder neue String-Objekte erzeugen zu müssen. Symbole sind konstante Strings, die nur einmalig im System vorhanden sind, sie werden durch einen führenden Doppelpunkt gekennzeichnet. In Listing 9 sind einige Beispiele für die Verwendung von Hashes dargestellt.

```ruby
# Array-Literale
a = []
a = [1, 2, 3]

# Zugriff über den index
puts a[0] # Element 0 ausgeben

# Größe vorinitialisieren
a = Array.new(10)

# Beliebige Typen
a = [1, 2, "Ein String", 1.0]
```

Listing 8: Array Beispiele

```ruby
# Hash-Literale
h = {}
h = {a: 1, b: 2}

# Zugriff
puts h[:a] # gibt 1 aus

# Beliebige Typen
h = {"String Schlüssel" => 1,
     symbol_schluessel: [1,2],
     1 => "String Wert"}
```

Listing 9: Hash Beispiele

Es sei hier besonders auf die teilweise abweichende Syntax in den Literalen hingewiesen. In Ruby 1.8 war die einzig mögliche Syntax `schlüssel => wert`. Die häufige Verwendung von Symbolen als Schlüssel hatte die Einführung der Syntax `symbol: wert` mit der Ruby-Version 1.9 zur Folge. Diese stellt lediglich eine Abkürzung für `:symbol => wert` dar. Für Schlüssel, die keine Symbole sind, ist weiterhin die alte Syntax zu verwenden.

Hashes werden in Rails vorwiegend als optionale Parameter für Methoden verwendet. Ein Beispiel solch einer Anwendung findet sich in Listing 10. In der Parameterliste hat der Ausdruck `opts = {}` die Zuweisung eines leeren Hashes zu `opts` zur Folge, sofern kein Hash als Parameter übergeben wird.

```ruby
def some_method(opts = {})
  puts opts # Optionen ausgeben
end

some_method                # Ausgabe: {}
some_method({a: 1, b: 2}) # Ausgabe: {:a => 1, :b => 2}

# Auf Klammern kann bei Eindeutigkeit verzichtet werden
some_method a: 1, b: 2     # Ausgabe: {:a => 1, :b => 2}
```

Listing 10: Beispielanwendung eines Hashes als optionaler Methodenparameter

Dieses Vorgehen bietet den Vorteil, dass die Schnittstelle bei Änderung der Parameter nicht angepasst werden muss. Weiterhin spielt die Reihenfolge der Argumente keine Rolle mehr, da die Werte über ihren Schlüssel identifiziert werden.

Blöcke

Kaum ein in Ruby geschriebenes Programm kommt ohne die funktionalen Elemente der Sprache aus. Der **Block** ist das meist genutzte dieser Elemente. Blöcke bieten eine Möglichkeit, Codefragmente an Methoden zu übergeben. Eine wichtige Anwendung ist die Iteration über enumerierbare Collections. Listing 11 zeigt als Beispiel die Iteration über ein Array mit der Methode `each`.

```
1 [1, 2, 3].each do |element| # Ausgabe: 1
2   puts element              #          2
3 end                         #          3
```

Listing 11: Beispielanwendung eines Blocks zur Iteration über ein Array

Die Methode `each` ist eine Instanzmethode der Klasse `Array`. Der Beginn des an die Methode übergebenen Blocks wird durch das Schlüsselwort `do` markiert, das Ende durch `end`. Ein Block verhält sich wie eine anonyme Methode, er kann also auch Parameter entgegen nehmen. Die Parameterliste wird nach dem `do`-Schlüsselwort von `|`-Symbolen umrahmt angegeben. Der Block im Beispiel erwartet also einen Parameter und stellt diesen unter dem Identifier `element` in seinem Rumpf zur Verfügung.

Eine (mögliche) Implementation der Methode `Array#each` ist in Listing 12 dargestellt.

```
1 class Array
2   def each
3     # (0..n) ist ein Range-Objekt, es beinhaltet
4     # die ganzen Zahlen von 0 bis n.
5     for i in ( 0 .. self.size - 1 )
6       yield(self[i])
7     end
8   end
9 end
```

Listing 12: Mögliche Implementation der Methode each in der Klasse Array

Die Methode iteriert mittels einer For-Schleife und einer Laufvariablen `i` über alle Indizes des Arrays. Das Schlüsselwort `yield` in Zeile 6 ruft den übergebenen Block auf und übergibt ihm das Arrayelement an Stelle i.

Auch Blöcke sind in Ruby Objekte und können von einer Methode zur späteren Verarbeitung zwischengespeichert werden, sie müssen keineswegs wie im obigen Beispiel umgehend ausgeführt werden.

Class Macros

Class Macros werden insbesondere im Rails-Kontext exzessiv eingesetzt. Das bekannteste Class Macro ist `attr_accessor`, welches Getter- und Settermethoden für die über-

gebenen Attribute bereitstellt. Listing 13 zeigt die Verwendung.

```ruby
class Person
  attr_accessor :name
end

# folgende Funktionalität wird bereitgestellt
susi = Person.new
susi.name = "Susi Sorglos"
puts susi.name # gibt "Susi Sorglos" auf der Kommandozeile aus.
```

Listing 13: Verwendung des attr_accessor Class Macros

Es hat den Anschein, als sei `attr_accessor` ein Schlüsselwort der Sprache, tatsächlich ist es aber eine Klassenmethode von `Module`, der Superklasse von `Class` (für weitere Details zum Object-Model in Ruby sei auf die Literatur verwiesen [Perr 10, Thom 09]). Als Parameter erwartet die Methode ein Objekt der Klasse `Symbol`. Beim Laden der Datei wird der Code innerhalb der Klassendefinition ausgeführt und somit `attr_accessor` mit dem Parameter `:name` aufgerufen. Die Methode definiert nun die Methoden `name` und `name=(param)` als Instanzmethoden für Objekte der Klasse `Person`. Auf die Implementierung wird an dieser Stelle nicht weiter eingegangen, da dafür tiefgreifendere Kenntnisse bezüglich Metaprogramming in Ruby nötig wären.

Allgemein führt die Nutzung von Class Macros zu deutlich lesbarerem Code, da sich wiederholende Codefragmente einfach ausgelagert werden können. Außerdem bieten sie die Möglichkeit, beim Laden der Klasse Code in Abhängigkeit von Konfigurationen zu erzeugen.

Weitere Spracheigenschaften

Die Sprache Ruby bietet noch zahlreiche weitere Eigenschaften wie dynamische Methodenaufrufe, auf Reflection und Introspection basierendes Metaprogramming, programmatische Methodendefinitionen zur Laufzeit, ein integriertes Templatesystem und vieles mehr. Die Darstellung der weiteren Eigenschaften würde aber nicht mehr nennenswert zum Verständnis der Codebeispiele beitragen, daher wird für weiterführende Informationen an dieser Stelle auf entsprechende Literatur verwiesen [Thom 09, Perr 10].

2.3.2 Model-View-Controller

Model-View-Controller bezeichnet ein bekanntes Pattern für Softwarearchitekturen. Die Grundidee besteht in der Isolation von Geschäftslogik und Userinterface. Die Modelschicht ist zuständig für die Geschäftslogik und die Interaktion mit der Datenbasis. Der Aufgabenbereich der Viewschicht beinhaltet die Bereitstellung von Elementen zur Nutzerinteraktion und die Darstellung von Daten. Der Controller stellt das Bindeglied zwischen den Schichten dar und vermittelt zwischen ihnen. Das Ziel ist eine lose Kopplung der Teilkomponenten, um diese möglichst unabhängig voneinander entwickeln, testen

und warten zu können. Damit sollen neben eine Reduktion der Komplexität verbesserte Wiederverwendbarkeit, Wartbarkeit und Flexibilität erreicht werden.

In eventgetriebenen Softwaresystemen, wie beispielsweise Desktopanwendungen, kommt zusätzlich oft das Pattern **Observer** [Vlis 95] zum Einsatz. Dabei beobachten sich die Komponenten gegenseitig und reagieren auf Änderungen oder Ereignisse in anderen Komponenten. Eine View zur Darstellung eines Models könnte zum Beispiel das Model beobachten und bei Änderungen die Darstellung aktualisieren. Im Kontext von Webapplikationen lässt sich durch den linearen Ablauf des Anfrage-Antwort-Zyklus' eine besonders lose Kopplung erreichen.

Abbildung 2.3 stellt den Anfrage-Antwort-Zyklus in Rails schematisch dar.

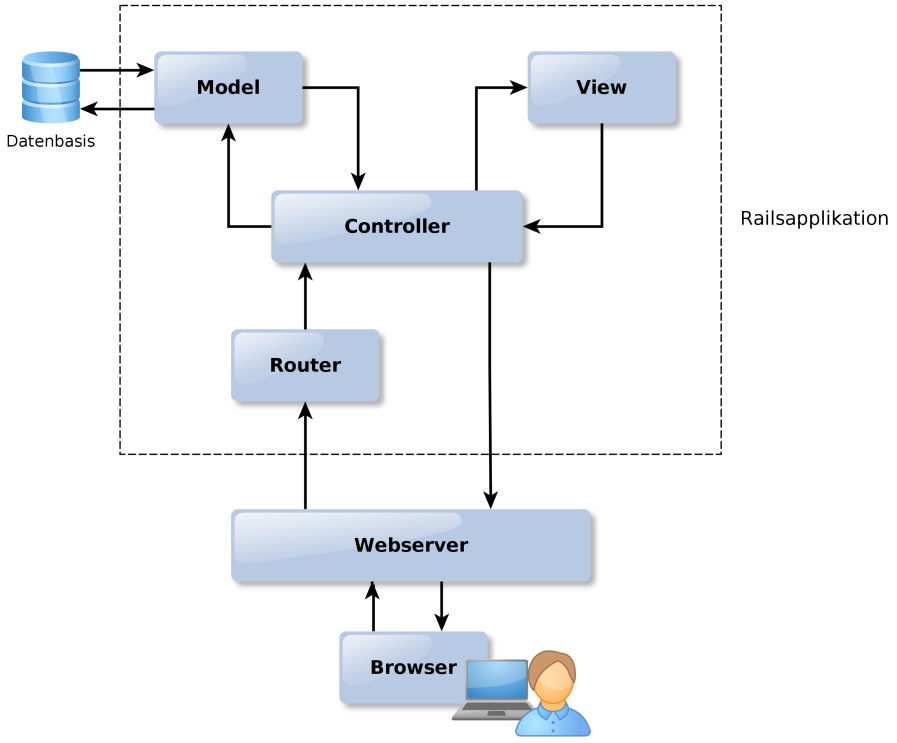


Abb. 2.3: Verarbeitung einer Anfrage durch die MVC-Struktur von Rails. Pfeile zwischen den Komponenten markieren den Datenfluss.

Ausgangspunkt ist eine Anfrage des Browsers an den Webserver. Dieser leitet die Anfrage an die Railsapplikation weiter. Die Klasse **Router** nimmt zunächst die Anfrage entgegen und entscheidet anhand der `URI`, welcher Controller für die Verarbeitung zuständig ist. Je nach Aktion werden entsprechende Models vom Controller angewiesen, die nötigen Operationen auf der Datenbasis durchzuführen oder angeforderte Daten bereitzustellen. Nach Beendigung aller Modeloperationen stellt der Controller der View die darzustellenden Daten zur Verfügung und fordert die in der Anfrage gewünschte Repräsentation (zum Beispiel HTML, XML oder JSON) an. Die View erzeugt die geforderte Darstellung und liefert diese an den Controller zurück. Der Controller generiert schließlich die entsprechende Antwort und übergibt sie dem Webserver zur Weiterleitung an den Browser. Die folgenden Unterabschnitte beschäftigen sich mit den Grundlagen zur Umsetzung der einzelnen Komponenten im Rails-Framework.

Model

Modelklassen werden in der Regel von der Klasse `ActiveRecord::Base` abgeleitet. Das ActiveRecord-Framework in Rails ist eine Implementation des erstmals 2003 von Martin Fowler erwähnten ActiveRecord-Patterns [Fowl 03]. Ein ActiveRecord-Model beinhaltet sowohl die Geschäftslogik als auch die nötigen Kenntnisse sich ohne eine weitere Persistenzschicht in der Datenbasis zu speichern. Der Hauptzweck ist die Abstraktion einer relationalen Datenbank durch eine objektorientierte Schnittstelle (Objektrelationales Mapping, ORM). Eine Modelklasse repräsentiert eine Tabelle oder eine Sicht der Datenbank, einzelne Instanzen der Klasse repräsentieren einzelne Datensätze und die Attribute der Instanzen die Elemente in den Tabellenspalten. `ActiveRecord::Base` stellt Methoden zum Einfügen, Lesen, Aktualisieren und Löschen von Datensätzen (CRUD-Operationen) ebenso wie Zugriffsmethoden für die Attribute bereit. Das ActiveRecord-Framework in Rails kann auf unterschiedlichen Datenbanken operieren. In dieser Arbeit wird für die Test- und Entwicklungsumgebung SQLite3 verwendet. Für die spätere Produktivumgebung ist eine MySQL-Datenbank vorgesehen.

Die dynamische Natur der Sprache Ruby macht die Verwendung extrem einfach. In Listing 14 sind ein voll funktionstüchtiges Model und einige der bereitgestellten Funktionen dargestelt.

```
1  class User < ActiveRecord::Base
2  end
3
4  u = User.new name: "Susi"
5  puts u.persisted?                  # Ausgabe: false
6  u.save
7  puts u.name                        # Ausgabe: Susi
8  puts u.persisted?                  # Ausgabe: true
9  u.update_attribute :name, "Willi"
10 puts u.name                        # Ausgabe: Willi
11 u.name = "Hans"
12 u.save
13 puts u.name                        # Ausgabe: Hans
```

Listing 14: Beispiel für eine voll funktionstüchtige ActiveRecord-Klasse

Woher aber weiß die Modelklasse, welche Tabelle sie ansprechen muss und welche Attribute zur Verfügung stehen? Hier zeigt sich ein effizienter Einsatz der Konzepte Don't Repeat Yourself und Convention over Configuration. Der Tabellenname wird aus dem Klassennamen des Models abgeleitet. ActiveRecord erwartet, dass zu einem Model namens `User` eine korrespondierende Tabelle `users` existiert (für weitere Details zu COC siehe 2.3.3). Beim Laden der Applikation fragt das Model die zur Verfügung stehenden Attribute von der Datenbank ab und definiert sich dynamisch entsprechende Methoden. Dies bietet den Vorteil, dass diese Daten nur an einem einzigen Ort abgelegt sind, nämlich in der Datenbank. Änderungen müssen nur am Datenbankschema durchgeführt

werden, die Modelklassen passen sich automatisch an. Zwar könnte man das Datenbankschema auch im Model ablegen, dies würde jedoch die Möglichkeit der iterativen Datenbankentwicklung mittels Migrations zunichte machen. Für weitere Details sei hier auf die Literatur verwiesen [Ruby 11, Fern 10].

View

Die View-Komponenten sind für die Darstellung und Erfassung von Daten zuständig. Der Entwickler muss in Rails lediglich View-Templates definieren und an geeigneter Stelle ablegen (siehe 2.3.3). Das Rendering und die Bereitstellung des korrekten Kontextes übernimmt die Bibliothek `ActionPack`, auf diesbezügliche Details wird hier nicht weiter eingegangen. Als Standard wird in Rails die in der Standardbibliothek enthaltene Templateengine **Embedded Ruby** (ERB) verwendet. Wie der Name schon sagt, bietet sie die Möglichkeit Rubycode in Plaintext einzubetten. Im Falle einer Webapplikation ist dies meist `HTML`. Listing 15 zeigt ein einfaches Beispiel der Darstellung eines Nutzerobjektes und dessen Posts.

```
1  <html>
2    <body>
3      <div id="profile">
4        <h2><%= @user.name  %></h2>
5        <% @user.posts.each do |p| %>
6          <div class="post_link">
7            <%= link_to p.title, post_path(p) %>
8          </div>
9        <% end %>
10     </div>
11   </body>
12 </html>
```

Listing 15: ERB-Beispiel

Im Wesentlichen sind zwei Tags nötig, zum einen `<%= ... %>`, um das Ergebnis des umschlossenen Ausdrucks in den Plaintext zu interpolieren, und zum anderen `<% ... %>`, um Rubycode auszuführen, aber nicht hineinzuinterpolieren. Im Beispiel wird der zweite Tag verwendet, um über die Collection `posts` zu iterieren.

Im Mai 2010 wurde die **HTML Abstraction Markup Language** (HAML)[5] veröffentlicht. Sie stellt eine Alternative zu ERB dar. Listing 16 zeigt den zu Listing 15 äquivalenten HAML-Code.

Schon in diesem kleinen Beispiel fällt die deutlich erhöhte Lesbarkeit von HAML gegenüber ERB auf. Die Angabe schließender Tags ist in HAML nicht notwendig, die Verschachtelung wird durch korrekte Einrückung umgesetzt. Zum Erzeugen eines HTML-Knotens beginnt man eine Zeile mit dem `%`-Zeichen, gefolgt vom Namen des Knotens. Der Inhalt aller folgenden, um zwei Zeichen eingerückten Zeilen wird dann in diesen

[5] `http://www.haml-lang.com/`

```
%html
  %body
    #profile
      %h2= @user.name
      - @user.posts.each do |p|
        .post_link
          = link_to p.title, post_path(p)
```

Listing 16: HAML Beispiel

Knoten verschachtelt. Für `DIV`-Elemente existieren noch zwei besondere Abkürzungen, da sie so häufig auftauchen. Mit einem `#`-Zeichen beginnende Zeilen erzeugen ein DIV-Element mit der auf das `#` folgenden ID. Zeilen, die mit einem Punkte beginnen, funktionieren analog, sie setzen aber statt der ID das CLASS-Attribut des DIV-Elements. Wie bei ERB wird mit dem Gleichheitszeichen das Ergebnis eines Rubyausdrucks in den Text interpoliert. Mit einem Bindestrich beginnende Zeilen enthalten Rubycode, der nur ausgeführt, aber nicht interpoliert wird.

Durch die notwendige Einrückung werden lesbare Template Files quasi erzwungen. HAML erzeugt eine Art semantisches Markup, dessen Struktur und Intention viel besser und mit deutlich weniger Code kommuniziert wird. Aufgrund dieser Vorteile wird im weiteren Verlauf dieser Arbeit HAML für die View-Templates eingesetzt.

Controller

Alle Controller in Rails erben von der Klasse `ActionController::Base`. Meist wird aber noch eine Zwischenklasse names `ApplicationController` eingefügt, die selbst von `ActionController::Base` erbt und von der dann die Controller der Applikation abgeleitet werden können. Dieses Vorgehen eröffnet die Möglichkeit, für alle Controller nötige Funktionalitäten gemäß dem DRY-Prinzip an einer zentralen Stelle zu verwalten. Ein Beispiel für solch eine Funktionalität wäre die Behandlung von Fehlern, um zu verhindern, dass der Nutzer jemals eine Fehlermeldung direkt zu Gesicht bekommt. Listing 17 zeigt einen Controller mit einer einzigen **Action**.

```
class UsersController < ApplicationController
  def show
    @user = User.find(params[:id])
  end
end
```

Listing 17: Controller-Beispiel

Actions bezeichnen Instanzmethoden von Controllern und sind die eigentlichen Endpunkte der Anfrageverarbeitung. Der im Beispielcode definierte Controller ist voll einsatzfähig und sorgt für die Darstellung eines Nutzerobjektes. Die Übergabe der Daten an das View-Template erfolgt automatisch, Templates werden im Kontext des aktiven

Controller ausgewertet und erhalten direkten Zugriff auf gesetzte Instanzvariablen. Die ID des anzuzeigenden Nutzers muss als Anfrageparameter übergeben werden. Innerhalb von Controller-Actions sind alle Anfrageparameter im Hash `params` verfügbar. Welches View-Template zu verwenden ist, wird automatisch anhand der aktiven Action ermittelt (siehe Abschnitt 2.3.3). Durch explizite Verwendung der Methode `render` kann dieses Verhalten bei Bedarf angepasst werden.

2.3.3 Convention over Configuration

Hauptintention des Prinzips Convention over Configuration ist die Vermeidung von unnötigem Aufwand im Bereich Konfigurationsmanagement. Natürlich kann ein Framework nicht das gesamte Konfigurationsmanagement ersetzen, aber gewisse Konfigurationselemente bedürfen dank geschickter Konventionen und der dynamischen Natur von Ruby keiner weiteren Beachtung.

Repräsentative Beispiele sind die Namenskonventionen und die Verzeichnisstruktur, die hier exemplarisch beleuchtet werden. Abbildung 2.4 zeigt einen Ausschnitt der automatisch erzeugten Verzeichnisstruktur einer Railsapplikation.

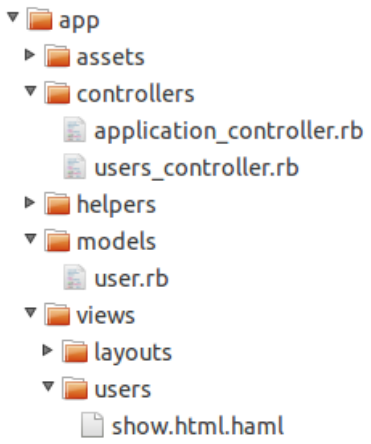


Abb. 2.4: Konventionskonforme Verzeichnisstruktur

In dieser Applikation wurden bereits ein Controller, ein Model und eine View zur Darstellung von Usern angelegt. Der Ordner `app` beinhaltet sämtliche Applikationsdateien. Neben den MVC-Komponenten und den korrespondierend benannten Ordnern finden sich im Ordner `assets` die Stylesheets und JavaScript-Dateien. Der Ordner `helpers` beinhaltet Helferklassen zur Auslagerung von Logik aus den View-Templates.

In Rails ist es Konvention, Klassen im Camelcase zu benennen. Weiterhin werden sie in englischer Sprache benannt, da die automatisierte Pluralisierung sonst zu sehr eigenartigen Konstrukten führen kann. Die Dateien, in denen die Klassendefinitionen liegen, werden im Snakecase benannt. Die Modelklasse für die Nutzer hieße also `User` und würde im Ordner `app/models` in der Datei `user.rb` abgelegt. Der Controller zur Verwaltung einer Modelklasse trägt den pluralisierten Namen des Models. Die Klasse zur Verwaltung des User-Models hieße also `UsersController` und würde in `app/controllers/users_controller.rb` abgelegt. Die zugehörigen View-Templates befinden sich

in einem nach dem pluralisierten Model benannten Ordner, hier in `app/views/users`. Der Name eines View-Templates korrespondiert zur Controller-Action in der es gerendert wird. Das Template `app/views/users/show.html.haml` wird in der Action `show` im `UsersController` gerendert. Aus dem Namen des aktiven Controllers und der aufgerufenen Action kann direkt der Pfad des zu rendernden Templates abgeleitet werden. Die dem Model zugrunde liegende Datenbanktabelle würde ebenfalls im pluralisierten Snakecase des Modelnamens benannt, also schlicht `users`.

Natürlich ist es durch Konfiguration auch möglich, von diesen Konventionen abzuweichen. Abgesehen von der Arbeit mit Legacy-Systemen oder möglicherweise abweichenden persönlichen Präferenzen gibt es aber objektiv betrachtet keinen Grund dazu. Eine Verzeichnisorganisationsstruktur und ein Namensschema ist in den meisten Fällen nicht besser oder schlechter als ein anderes. Die Einhaltung der Konventionen bringt im Fall von Rails eleganten Code und wenig Konfigurationsaufwand mit sich. Außerdem findet sich jeder Entwickler mit Erfahrung bezüglich des Frameworks sofort in der Applikation zurecht und kann sich so deutlich schneller einarbeiten. Im Weiteren werden die Konventionen daher strikt eingehalten.

2.3.4 Behaviour-Driven Development

Testen spielt in der Ruby-Community eine wichtige Rolle. Bereits die Standardbibliothek beinhaltet mit `Test::Unit` ein rudimentäres Framework um Test-Driven Development zu praktizieren. Besonders im Kontext auf Basis von Rails entwickelter Webapplikationen wird hinreichendes Testen als notwendig betrachtet. Rails war das erste Webapplikations-Framework mit einem *"integrated full-stack testing framework"* [Chel 10, S. 277]. Dieses Framework basiert auf `Test::Unit` und liefert **Unit Tests** für Models, **Functional Tests** für Controller und Views und **Integration Tests** zum Testen von Workflows über mehrere Anfragen. Listing 18 zeigt ein Beispiel für einen Unit-Test mit `Test::Unit`.

```
1  class UserTest < Test::Unit::TestCase
2    def setup # wird vor jedem Test ausgeführt
3      @user = User.new
4    end
5
6    # Tests müssen mit test_ beginnen, um automatisch
7    # als Test erkannt zu werden.
8    def test_name_setter
9      assert_nil @user.name, "The user name did not init with nil"
10     @user.name = "Willi"
11     assert_equal @user.name, "Chuck", "@user.name has not been set"
12   end
13 end
```

Listing 18: Test::Unit-Beispiel

Der letzte String-Parameter der `assert_*`-Methoden repräsentiert dabei die bei Fehlschlagen des Tests angezeigte Fehlermeldung.

Die Autoren des Standardwerkes zu BDD im Rails-Kontext [Chel 10] empfehlen allerdings die Verwendung der nachfolgend beschriebenen optimierten Tools **RSpec**[6] und **Cucumber**[7].

RSpec

Die Bibliothek RSpec stellt eine **Domain-Specific Language** (DSL) zur Spezifikation des Verhaltens von Objekten bereit. Sie wurde im Jahr 2005 von Steven Baker entwickelt. RSpec wird im Rahmen dieser Arbeit in Version 2.6.4 zur Spezifikation von Models und Controllern verwendet.

Der entscheidende Vorteil gegenüber `Test::Unit` besteht in der Syntax. Listing 19 zeigt die Umsetzung des Unit-Tests aus Listing 18 mittels RSpec. Der Test als Ganzes wird in RSpec als **Spec** (Abkürzung von Specification) bezeichnet.

```
1  describe User do
2    before(:each) do
3      @user = User.new
4    end
5
6    it "should assign the user name when the setter is called" do
7      @user.name.should be_nil
8      @user.name = "Willi"
9      @user.name.should == "Willi"
10   end
11 end
```

Listing 19: RSpec-Beispiel

Die Spec kommuniziert gegenüber dem Test::Unit-Test deutlich klarer die eigentliche Intention, in RSpec werden Testmethoden daher als **Example** bezeichnet. Dies soll zum Ausdruck bringen, dass der Existenzzweck eines Examples nicht nur die reine Testfunktionalität ist, sondern ebenfalls als Anwendungsbeispiel und damit Dokumentation des zu testenden Codes zu verstehen ist. Ein lautes Vorlesen liefert ohne weiteres Nachdenken eine natürlichsprachliche Beschreibung des gewünschten Verhaltens. Ein Example wird durch die Methode `it` definiert, welche sich semantisch auf das Objekt des umgebenden `describe`-Blocks bezieht. Sie erwartet als Parameter einen String mit einer natürlichsprachlichen Beschreibung des Tests und einen Block mit dem Testcode. Der beschreibende String dient nicht nur der Erläuterung des Codes, sondern kann verwendet werden, um eine natürlichsprachliche Form der Spec zu erzeugen.

[6] `https://www.relishapp.com/rspec`
[7] `http://cukes.info/`

Um zusammenhängende Examples auch zusammenhängend darzustellen, werden diese zu **Example Groups** aggregiert. In Listing 20 ist eine Ergänzung der Spec aus Listing 19 um ein weiteres Example dargestellt.

```
1  describe User do
2    context "when new created" do
3      before(:each) do
4        @user = User.new
5      end
6
7      it "should assign the user name when the setter is called" do
8        @user.name.should be_nil
9        @user.name = "Willi"
10       @user.name.should == "Willi"
11     end
12
13     it "should not save without a name" do
14       @user.save.should_not be_true
15     end
16   end
17 end
```

Listing 20: Beispiel für eine Example Group

Beide Examples spezifizieren Funktionalität eines neu erstellten Objektes, sie können also mittels der Methode `context` zu einer Example Group zusammengefasst werden. Der `before`-Block wird vor der Ausführung jeden Examples im `context`-Block erneut ausgeführt werden, so dass stets ein unangetastetes `User`-Objekt zur Verfügung steht. Listing 21 zeigt die Ausgabe der erfolgreichen Ausführung der Spec aus Listing 20.

```
User
  when new created
    should assign the user name when the setter is called
    should not save without a name

Finished in 0.00085 seconds
2 examples, 0 failures
```

Listing 21: Ausgabe beim Ausführen der Spec aus Listing 20

RSpec bietet noch weitere Möglichkeiten zur Ausgabe der Testberichte. Erfolgreiche und fehlschlagende Examples können farbig hervorgehoben oder der ganze Ablauf der Spec als HTML-Bericht exportiert werden.

Die Verwendung von RSpec liefert automatisch eine vollständige Spezifikation sämtlicher Module mit, da durch testgetriebenes Vorgehen nahezu kein ungetesteter Code

entsteht. Die Specs zeigen in verständlicher Art und Weise, wie die Systemkomponenten zu verwenden sind. Neue Entwickler können sich anhand der Specs leicht in das System einarbeiten. Gleichzeitig haben sie bei der Durchführung von Änderungen oder Erweiterungen durch Ausführung der Tests die Sicherheit, keine anderen Komponenten beeinträchtigt zu haben. Es ist also auch ohne vollständige Kenntnis des Systems leicht möglich, Änderungen durchzuführen.

Mit RSpec wird also in effizienter Weise die Funktionalität der einzelnen Module in Isolation spezifiziert und getestet. Damit ist aber keineswegs die Gesamtfunktionalität und das korrekte Zusammenspiel der einzelnen Komponenten gewährleistet. Für diesen Zweck kommt die Bibliothek Cucumber zum Einsatz.

Abschließend sei noch erwähnt, dass die Einarbeitung in RSpec nicht nur für den Rails-Kontext nützlich ist. Dank JRuby und MacRuby lassen sich mit RSpec auch beliebige Java-Systeme oder ObjectiveC-Systeme auf der Mac-Plattform spezifizieren und testen.

Erzeugung von Testdaten

Zum Testen von komplexem Verhalten müssen oft auch recht umfangreiche Szenarien hergestellt werden, für die eine gewisse Menge an Testdaten notwendig ist. Mit den sogenannten **Fixtures** liefert Rails bereits eine Möglichkeit zur Testdatenerzeugung. Beispieldaten für Nutzerobjekte würden in der Datei `test/fixtures/users.yml` im **YAML**-Format abgelegt werden. Listing 22 zeigt ein Beispiel.

```
1 willi:
2   name: Willi Wacker
3   admin: false
4
5 susi:
6   name: Susi Sorglos
7   admin: true
```

Listing 22: Fixtures für Nutzerobjete

Innerhalb des Tests wird mit einer zum Namen des Models und der YAML-Datei analog benannten Methode auf die Fixtures zugegriffen. Der Aufruf `users(:willi)` würde aus den Daten mit dem Schlüssel `willi` in der YAML-Datei `users.yml` ein User-Objekt mit entsprechenden Attributen erzeugen. Auf den ersten Blick mag dies ein sinnvolles Vorgehen sein, im Detail betrachtet bringt es allerdings einige Nachteile mit sich. Als Entwickler muss man stets die Fixture-Daten im Kopf haben, oder aber regelmäßig in den Dateien nachsehen. Je mehr Assoziationen zwischen den Models bestehen, desto unübersichtlicher werden die Fixtures. Besteht zum Beispiel eine 1:n-Beziehung zwischen Nutzer und Veranstaltung, so würde bei der Veranstaltung die ID des Nutzers als Fremdschlüssel hinterlegt sein. Um diese Testdaten zu überblicken, müsste man also bereits in zwei Fixture-Dateien nachsehen. Bei mehrschichtigen Assoziationen kann schnell die Übersicht verloren gehen. Außerdem ist unklar, welche Fixtures für welchen

Test gedacht sind. Eine kleine Änderung kann zum Fehlschlagen einer großen Menge von Tests führen.

Um die Nachteile von Fixtures auszugleichen, wurde von der Firma Thoughtbot Inc.[8] die Bibliothek **FactoryGirl**[9] entwickelt und frei zur Verfügung gestellt. FactoryGirl basiert auf dem Designpattern **Factory** (für Details siehe [Vlis 95]). Die Bibliothek erlaubt die Definition dynamischer Factories zur Erzeugung von Testdaten. In Listing 23 ist eine Factory-Definition dargestellt, welche die wichtigsten Konzepte verdeutlicht.

```
1  FactoryGirl.define do
2    factory :user  do
3      name  { |n| "FirstName#{n} LastName#{n}" }
4      admin false
5
6      factory :admin do
7        admin true
8      end
9    end
10 end
```

Listing 23: Beispiel für eine Factory-Definition

Die Methode `factory` erwartet zunächst den Namen der Modelklasse, für die die Factory Instanzen erstellen soll. Innerhalb des darauf folgenden Blocks steht für jedes Attribut der Modelklasse eine Methode zur Verfügung. Den Attribut-Methoden kann ein konstanter Wert oder aber wiederrum ein Block übergeben werden. Wird ein konstanter Wert übergeben, so erhalten alle mittels der Factory erzeugten Instanzen diesen als Wert für das entsprechende Attribut. Der Block würde für jede erzeugte Instanz neu evaluiert werden und bekommt als Parameter eine für jede Instanz verschiedene, fortlaufende Nummer. Damit besteht die Möglichkeit, Testdaten auch für Attribute mit Uniqueness-Constraints automatisiert erzeugen zu können. Weiterhin ist es möglich, verschachtelte Factories zu definieren, ein Beispiel ist die Factory `:admin` in Listing 23. In verschachtelten Factories werden alle Definition der umgebenden Factory übernommen. Es können sowohl weitere Attributwerte hinzugefügt als auch existierende überschrieben werden. Auch assoziierte Objekte können so automatisch durch entsprechende Definition erzeugt werden.

Die Verwendung der Factories ist in Listing 24 dargestellt. Das mittels der User-Factory erzeugte Objekt bekommt die generierten Werte aus der Factory-Definition. Diese Werte können durch zusätzliche Übergabe eines optionalen Hashes (siehe 2.3.1) überschrieben werden, wie bei dem durch die Admin-Factory erzeugten Objekt im Beispiel.

Die definierten Factories können nun direkt in den Examples verwendet werden. Für die meisten Fälle genügen die Default-Werte, so dass der Code im Example selbst nicht

[8] `http://thoughtbot.com/`
[9] `https://github.com/thoughtbot/factory_girl`

```ruby
user = Factory(:user)
puts user.class     # Ausgabe: User
puts user.name      # Ausgabe: FirstName1 LastName1
puts user.admin     # Ausgabe: false

other = Factory(:admin, name: "Willi Wacker")
puts other.class    # Ausgabe: User
puts other.admin    # Ausgabe: true
puts other.name     # Ausgabe: Willi Wacker
```

Listing 24: Anwendungsbeispiel der in Listing 23 definierten Factory

viel länger als bei der Verwendung von Fixtures ausfällt. Damit befinden sich alle Informationen bezüglich des zu testenden Szenarios an einem Ort und ermöglichen eine bessere Übersicht und absolute Isolation der Testdaten eines jeden Examples.

Cucumber

Aslak Hellesøy extrahierte Cucumber im April 2008 aus dem RSpec Story Runner, woraufhin die RSpec-Entwickler dessen Weiterführung zu Gunsten von Cucumber aufgaben. Die Bibliothek wird im BDD mit Rails zur Umsetzung von User-Stories (siehe Abschnitt 2.1.3) zu automatisierten Akzeptanztests verwendet. Neben der Ruby-Version existieren Implementationen für Java, .NET und Flex. Die Stories werden in Cucumber als **Features** bezeichnet, ein Beispiel mit zwei Szenarien ist in Listing 25 dargestellt.

```
Feature: Logging in and out
  In order to protect my session on my computer
  As an existing user
  I want to be able to log in and out

  Scenario: Logging in
    Given I am an existing user
    When I go to the login page
    And submit the form with my data
    Then I should be on the home page
    And should see "Welcome to the application!"

  Scenario: Logging out
    Given I am logged in as user
    When I click the logout button
    Then I should be on the login page
    And should see "Logout successfull."
```

Listing 25: Cucumber-Feature mit zwei Szenarien

Ein Feature beginnt mit dessen Titel, der in prägnanter Weise die wesentliche Funktion beschreiben sollte. Als nächstes folgt die eigentliche User-Story im Connextra-Format (siehe Abschnitt 2.1.3). Für die automatisierte Ausführung hat die Story keine Bedeutung, sie dient lediglich der Dokumentation und Erläuterung. Im Weiteren folgen die eigentlichen Akzeptanzkriterien in Form der Szenarien, welche wiederrum aus einem Titel und einer Menge von **Steps** bestehen. Mit dem Schlüsselwort `Given` beginnende Steps beschreiben die Voraussetzungen des Szenarios. Das Schlüsselwort `When` wird für Steps verwendet, die eine Nutzeraktion beschreiben. Schließlich wird mit dem Schlüsselwort `Then` das erwartete Verhalten der Applikation spezifiziert. Für die bessere Lesbarkeit wurde zusätzlich noch `And` eingeführt, welches je nach Kontext gleichbedeutend mit dem zuletzt verwendeten Schlüsselwort ist.

Im Rahmen von Webapplikationen findet die Interaktion des Nutzers mit dem System durch den Browser statt. Cucumber selbst beinhaltet keine direkte Möglichkeit zur Browsersimulation. Für die Interaktion mit der Railsapplikationen steht die hier verwendete Bibliothek **Capybara**[10] zur Verfügung. Capybara emuliert Browserverhalten durch geschicktes Parsen und Manipulieren der von der Applikation generierten HTML-Seiten. Ist für ein Szenario die Ausführung von JavaScript nötig, so kann durch Setzen eines Tags am entsprechenden Szenario auf **Selenium**[11] als Driver umgeschaltet werden. Selenium startet im Hintergrund einen Browser ohne GUI und steuert diesen automatisiert.

Die Implementation der Steps erfolgt durch sogenannte **Step Definitions**. Listing 26 zeigt einige Beispiele.

```
1  Given /^(?:|I )am an existing user$/ do
2    @user = Factory(:user)
3  end
4
5  When /^(?:|I )submit the form with my data$/ do
6    fill_in "Username", with: @user.name
7    fill_in "Password", with: @user.password
8    click_button "Login"
9  end
10
11 Then /^(?:|I )should see "([^"]*)"$/ do |content|
12   page.should have_content(content)
13 end
```

Listing 26: Beispiele für Step Definitions

Eine Step Definition beginnt mit dem gleichen Schlüsselwort wie der zu implementierende Step. Der erste Parameter ist ein regulärer Ausdruck, anhand dessen zur Laufzeit entschieden wird, welche Step Definition für die Ausführung eines Steps zuständig ist. Die regulären Ausdrücke beginnen häufig mit dem Ausdruck `(?:|I )`. Dieser dient lediglich dazu, das *I* zu Beginn optional zu machen, um den Lesefluss bei `And`-Steps zu

[10] `https://github.com/jnicklas/capybara`
[11] `http://seleniumhq.org/`

verbessern. Konflikte können auftreten, wenn mehrere Step Definitions einen Step matchen. Cucumber erkennt dies und weist mit einer Fehlermeldung darauf hin. Es ist sogar möglich, den Konflikt von Cucumber automatisch auflösen zu lassen, dies funktioniert jedoch nicht in allen Fällen. Step Definitions müssen also möglichst eindeutig gestaltet werden. Als zweiter Parameter wird ein Block erwartet. Dieser Block kann seinerseits Parameter beinhalten, die zur Laufzeit mit den Matches aus dem regulären Ausdruck befüllt werden.

Step Definitions vom Typ `Given` dienen zur Herstellung der nötigen Voraussetzungen. Im Beispiel legt der `Given`-Step unter Verwendung von FactoryGirl (siehe Abschnitt 2.3.4) ein Nutzerobjekt an und weist dieses einer Instanzvariablen zu. Alle Steps innerhalb eines Szenarios werden im Instanzkontext desselben Objektes ausgewertet. Die zugewiesene Instanzvariable `@user` im Beispiel steht nun also den nachfolgenden Step Definitions zur Verfügung.

Innerhalb der `When`-Step Definitions findet die simulierte Nutzerinteraktion statt. Die Step Definition im Beispiel füllt ein Login-Formular mit den Werten des vorher im `Given`-Step angelegten Nutzers aus. Die Methode `fill_in` wird durch den Browsersimulator zur Verfügung gestellt. Der Aufruf in Zeile 6 würde im HTML-Dokument nach einem Formularfeld mit Label oder ID *Username* suchen und dies mit dem Namen des Nutzers ausfüllen. Die Methode `click_button` in Zeile 8 sucht nach einem Button mit Aufschrift oder ID *Login* und clickt diesen. Damit wird das entsprechende Formular abgesendet.

Schließlich wird die erwartete Reaktion der Applikation in einer Step Definition für einen `Then`-Step umgesetzt. Die dargestellte Step Definition ist etwas flexibler als die anderen beiden. Sie enthält den Ausdruck `"([^"]*)"`, welcher jeden String matched, der in doppelten Anführungszeichen an entsprechender Stelle steht. Auf diese Weise lassen sich Parameter übergeben und damit wiederverwendbare Step Definitions erstellen, um den Aufwand zur Implementation späterer Features zu reduzieren. Dieser String wird an den Block übergeben und ist innerhalb des Blocks über die Variable `content` erreichbar. Die vom Browsersimulator bereitgestellte Methode `page` liefert ein Objekt zurück, welches die HTML-Seite repräsentiert. Der Matcher `have_content` prüft, ob der übergebene String innerhalb der Seite als Textinhalt eines Knotens auftaucht.

Die Cucumber-Features erfüllen mehrere Aufgaben. Zum einen dienen sie als Maß für den Fortschritt. Wenn alle User-Stories in Features umgesetzt worden sind, dann ist die geforderte Funktionalität genau dann vollständig implementiert, wenn alle Cucumber-Features erfolgreich durchlaufen. So wird gleichzeitig sichergestellt, dass keine Funktionalitäten vergessen werden können. Zum anderen fungieren die Features als Integration-Tests. Die Applikation wird durch den Browsersimulator weitgehend so verwendet, wie auch die Nutzer sie später verwenden würden. Zum erfolgreichen Durchlaufen der Features ist es also notwendig, dass das Zusammenspiel der mit RSpec in Isolation getesteten Einzelkomponenten korrekt funktioniert. Schließlich decken die Features auch weite Teile der View ab und machen separate View-Specs häufig überflüssig.

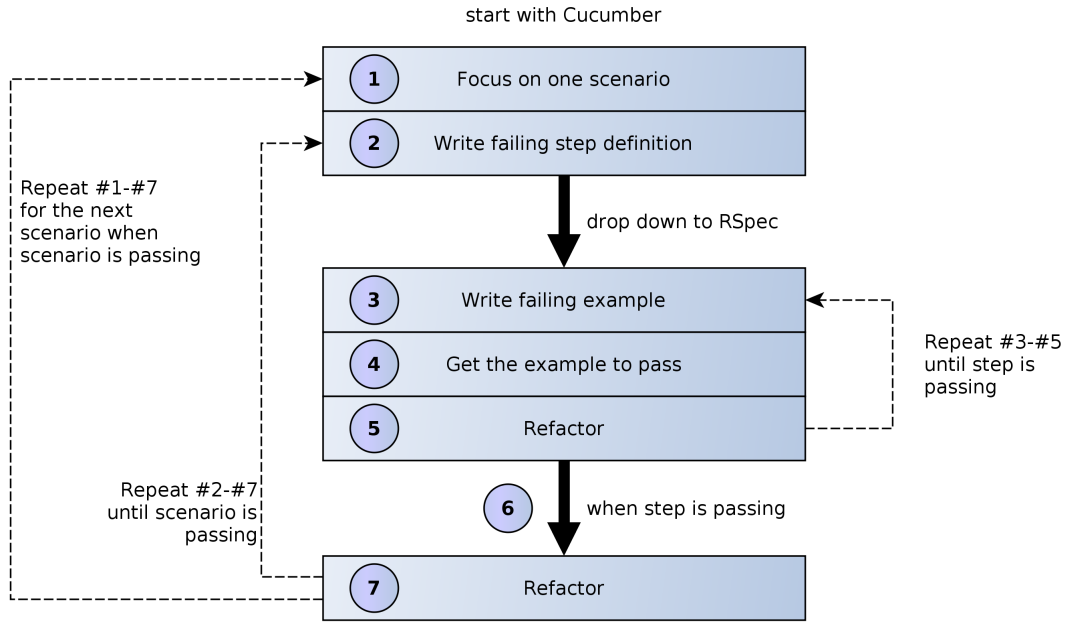


Abb. 2.5: BDD-Zyklus mit Cucumber und RSpec nach [Chel 10, S. 10]

Outside-In Entwicklung mit Cucumber und RSpec

Outside-In Entwicklung bezeichnet das schrittweise Aufspüren notwendiger Objekte und Schnittstellen aus Anwendersicht [Chel 10]. Ausgehend von einem Akzeptanzkriterium wird mit der testgetriebenen Entwicklung der entsprechenden View begonnen. Bei der Umsetzung der View werden die nötigen Daten und Mechanismen zur Anzeige aufgedeckt. Damit sind die vom Controller bereitzustellenden Objekte identifiziert. Im Laufe der Controller-Umsetzung wird klar, welche Modelklassen nötig sind und welche Attribute und Methoden diese bereitstellen müssen. Schnittstellen und Objekte können so bereits sehr früh im Ablauf erkannt und entsprechende Designentscheidungen direkt auf Basis der vorliegenden Notwendigkeiten getroffen werden. Auf diese Weise werden tatsächlich nur die absolut notwendigen Features implementiert. Damit lässt sich die Umsetzung von unnötigen Funktionalitäten vermeiden, denn in vielen Fällen kommen solche niemals zum Einsatz und stellen somit eine Verschwendung von Zeit und eine unnötige Vergrößerung der Codebasis dar.

In Abbildung 2.5 ist schematisch der Ablauf der Outside-In Entwicklung mit Cucumber und RSpec illustriert. In Schritt `#1` wird das zu implementierende Szenario ausgewählt. Das Implementieren der Step Definition für den ersten Step des Szenarios ist Schritt `#2`. Das Feature wird zunächst zur Sicherstellung des Fehlschlagens des Steps ausgeführt. Dieser Schritt ist wichtig, um zu gewährleisten, dass der Step ohne die zu implementierende Funktionalität fehlschlägt und nicht versehentlich erfolgreich ist, obwohl die Funktion nicht umgesetzt ist. Der Step impliziert in der Regel direkt die Notwendigkeit bestimmter Komponenten. Im Schritt `#3` wird nun für die erste identifizierte Komponente ein erstes Example in RSpec formuliert. Die Implementierung von ausreichendem Code, um das Example erfolgreich durchlaufen zu lassen, ist Schritt `#4`. Schließlich wird in Schritt `#5` der geschriebene Code auf Möglichkeiten zur Durchführung von Refacto-

ring untersucht. Läuft das Example nach dem Refactoring immer noch durch, so kann der Entwickler sicher sein, durch seine Änderungen keine Funktionalität beeinträchtigt zu haben. Die Schritte `#3-#5` werden für alle nötigen Komponenten wiederholt bis der Step erfolgreich durchläuft. Nun wird in Schritt `#7` nochmals auf Möglichkeiten zum Refactoring geprüft. Es sei hier erwähnt, dass nicht nur der Applikationscode, sondern auch die Tests selbst refaktorisiert werden. Nach erfolgreichem Refactoring beginnt der Zyklus erneut mit Schritt `#1` und dem nächsten Step des Szenarios. Der Zyklus wird iteriert, bis alle Steps umgesetzt sind und das gesamte Szenario fehlerfrei durchläuft.

2.3.5 Resource-Oriented Architecture

Im professionellen Umfeld ist die Resource-Oriented Architecture aufgrund ihrer positiven Eigenschaften (siehe Abschnitt 2.2) der bevorzugte Architekturstil für Railsapplikationen [Pyte 10, Fern 10, Rich 07]. Dieser Umstand hat eine sehr gute Unterstützung bei der Umsetzung von ROAs mit Rails zur Folge.

In den meisten Fällen werden Ressourcen in Rails auf Modelklassen abgebildet, deren Instanzen über einen korrespondierenden Controller verwaltet werden. Der Controller beinhaltet in der Regel sieben Actions: `index, show, new, create, edit, update, destroy`. Zur Umsetzung der einheitlichen Schnittstelle stellt der Router eine Hilfsmethode bereit und legt damit das Schema für die Konstruktion der URI einer Ressource und das Mapping der HTTP-Verben auf entsprechende Controller-Actions fest.

Im Folgenden wird dieses Prinzip anhand des Beispiels der Ressource `User` erläutert. Mit dem Ausdruck `resource :users` in der Routing-Konfiguration würden die folgenden Routen angelegt (Ausgabe des Kommandos `rake routes`):

```
   users GET    /users(.:format)         {:action=>"index",
                                        :controller=>"users"}
         POST   /users(.:format)         {:action=>"create",
                                        :controller=>"users"}
 new_user GET    /users/new(.:format)     {:action=>"new",
                                        :controller=>"users"}
edit_user GET    /users/:id/edit(.:format) {:action=>"edit",
                                        :controller=>"users"}
    user GET    /users/:id(.:format)     {:action=>"show",
                                        :controller=>"users"}
         PUT    /users/:id(.:format)     {:action=>"update",
                                        :controller=>"users"}
         DELETE /users/:id(.:format)     {:action=>"destroy",
```

Listing 27: Standard REST-Routen in Rails

Mit einem Doppelpunkt beginnende Ausdrücke in einem URI-Schema signalisieren, dass der entsprechende Wert an dieser Stelle eingesetzt werden muss. Diese Werte werden

mit entsprechendem Schlüssel als zusätzliche Anfrageparameter an den Controller übergeben. Zum Beispiel enden alle Routen mit dem Ausdruck `(.:format)`. Es kann also an die URI optional noch ein String angehängt werden, um das geforderte Format zu übermitteln. Durch Anhängen von `.xml` an eine URI würde `params[:format]` den Wert `xml` erhalten. Wird kein Format angegeben, so wird `html` verwendet. In den folgenden Erläuterungen wird der Format-Ausdruck der Übersichtlichkeit halber weggelassen.

Eine Anfrage der Art `GET /users` wird auf die Action `index` des `UsersController` geleitet. Diese URI stellt eine sogenannte **Collection Resource** dar. Für gewöhnlich wird hier eine Liste aller User-Ressourcen zurückgeliefert. Sie ist nicht zwingend Teil der einheitlichen Schnittstelle für die Nutzer-Ressource, häufig ist es aber für die Geschäftslogik sinnvoll, eine Liste aller Datensätze erhalten zu können.

Eine POST-Anfrage auf die Collection Ressource (`POST /users`) hat die Semantik des Anhängens einer neuen Instanz an die Liste und ist damit Bestandteil der einheitlichen Schnittstelle. Der Endpunkt einer solchen Anfrage ist die Action `create` des `UsersController`, welche aus der übermittelten Repräsentation eine neue Nutzer-Ressource erzeugt und zu deren URI weiterleitet. Die Erzeugung von Ressourcen unter Angabe der URI mittels PUT ist in Rails nicht vorgesehen.

Die URI für eine konkrete Nutzer-Ressource hat die Form `/users/:id`. Requests mit den HTTP-Verben GET, PUT oder DELETE werden auf die Actions `show`, `update` und `destroy` geleitet, welche genau die Aktion durchführen, die von der Semantik des jeweiligen HTTP-Verbs impliziert wird. Mit diesen Routes ist die einheitliche Schnittstelle vollständig.

Die verbleibenden beiden URI-Schemata `/users/new` und `/users/:id/edit` dienen lediglich der Anzeige entsprechender Formulare und erfüllen keinen direkten Zweck in der Umsetzung der einheitliche Schnittstelle.

In der ersten Spalte von Listing 27 finden sich noch die Namen der entsprechenden **Path Helper**. Rails generiert für alle URIs Helfermethoden zur Verwendung in Controllern und Views. Angenommen, die Instanzvariable `@user` hätte den Wert eines User-Objektes mit ID=3, dann würde der Aufruf `edit_user_path(@user)` den Wert `/users/3/edit` zurückliefern.

Entsprechender Code zur Implementation des Controllers gestaltet sich ähnlich simpel. Listing 28 zeigt einen Ausschnitt aus einem Controller für das obige Beispiel. Jede Action findet beziehungsweise erzeugt das entsprechende `User`-Objekt und übergibt es mittels der Methode `respond_with` an den **Responder** des Controllers. Ein Responder ist eine Hilfsklasse zur Anfrageverarbeitung. Je nach Status des Objektes und aufgerufener Action ist der Responder in der Lage zu entscheiden, was weiterhin zu tun ist. In der Action `create` würde er bei erfolgreichem Speichern des Objektes unter Verwendung des HTTP-Verbs GET zu der URI des neuen Nutzers weiterleiten, um das neu generierte Objekt anzuzeigen. Bei Fehlschlagen des Speichervorgangs würde erneut die Action `new` gerendert werden, um den Anwender über das Problem zu informieren und ihm die Möglichkeit zur Anpassung der Eingaben zu geben. Auch das zu rendernde Template wird vom Responder anhand des aktiven Controllers und der Action inferiert. Die Action `update` zeigt analoges Verhalten, mit dem einzigen Unterschied, dass statt der Action

`new` im Fehlerfall die Action `edit` gerendert würde. Im Fall der Action `destroy` würde nach dem Löschen des Objektes zur Collection Resource weitergeleitet werden.

```
1  class UsersController < ApplicationController
2    def show
3      @user = User.find(params[:id])
4      respond_with(@user)
5    end
6
7    def create
8      @user = User.new(params[:user])
9      @user.save
10     respond_with(@user)
11   end
12
13   def destroy
14     @user = User.find(params[:id])
15     @user.destroy
16     respond_with(@user)
17   end
18 end
```

Listing 28: Ausschnitt aus einem Controller

Abschließend sei gesagt, dass die Hilfsmittel zur Umsetzung von ROAs ebenfalls wieder ein gutes Beispiel des Prinzips Convention over Configuration sind. Hält man sich strikt an die Konzepte der ROA, so ist der resultierende Code schlank und elegant. Will man von den Konventionen abweichen, so ist dies ohne Probleme möglich, erfordert aber aber einen gewissen Mehraufwand.

2.4 Zusammenfassung

Ein konsequent mit den in den vorangegangenen Abschnitten dieses Kapitels beschriebenen Konzepten und Technologien entwickeltes System lässt sich wie folgt hinreichend beschreiben:

> Das System wurde als **Resource-Oriented Architecture** auf Basis von **Ruby on Rails** mittels **Behaviour-Driven Development** mit **Cucumber** und **RSpec** umgesetzt.

Damit ist die gesamte Grundstruktur der Applikation charakterisiert. Die Einhaltung der Rails-Konventionen eliminiert die Freiheitsgrade der ROA bezüglich der einheitlichen Schnittstelle durch genaue Festlegung der URI-Schemata. Ein Benennungsschema samt Verzeichnisstruktur für die einzelnen Module des Systems ist ebenfalls festgelegt. Durch die verhaltensgetriebene Entwicklung sind sowohl die Einzelkomponenten als auch das Zusammenspiel als Gesamtsystem durch Specs und Cucumber-Features

dokumentiert und getestet. Die verhaltensgetriebene Entwicklung garantiert bei korrekter Anwendung hohe Anwenderzufriedenheit und Codequalität. Jeder Entwickler mit Kenntnis der verwendeten Methoden und Technologien findet sich aufgrund obiger Beschreibung sofort und ohne lange Einarbeitung in der Applikation zurecht und kann diese anpassen und erweitern.

Bei der Umsetzung eines konkreten Projektes sind im Wesentlichen noch zwei Probleme zu lösen. Zum einen müssen die Anforderungen geeignet in eine Menge von untereinander verlinkten Ressourcen umgesetzt werden und zum anderen ist bei einer über den Browser genutzten Webapplikation die ansprechende und funktionale Gestaltung der Benutzeroberfläche zu erledigen. Alle weiteren Schritte ergeben sich aus den im Vorfeld erläuterten Konzepten und Technologien.

3 Umsetzung der Applikation

Im vorangegangenen Kapitel wurden in allgemeiner Form die verwendeten Methoden, der gewählte Architekturstil und der Einsatz geeigneter Werkzeuge beschrieben. Das vorliegende Kapitel dokumentiert die wesentlichen, noch offenen Designentscheidungen und gibt einen Überblick über Umsetzung und Aufbau der Applikation. Die Darstellung erfolgt nicht in chronologischer Reihenfolge der BDD-Iterationen, sondern gruppiert nach inhaltlichem Zusammenhang. Um den Rahmen der Arbeit nicht zu sprengen, wird die Darstellung auf die Dokumentation der entscheidenden Schritte konzentriert und vernachlässigt dabei die detaillierte Erläuterung der zugehörigen Tests.

3.1 Project Inception

Im Rahmen der Project Inception Phase (siehe Abschnitt 2.1.3) werden zunächst durch Gespräche mit den späteren Anwendern deren Nutzungsperspektiven und Anforderungen analysiert, um eine erste Idee der umzusetzenden Funktionalität zu erhalten. Schließlich werden die Anforderungen in recht abstrakter Form zu User-Stories konsolidiert. Für die zu entwerfende Applikation existieren Anwender in zwei möglichen Rollen. Am Institut für Informatik in Osnabrück ist es zumeist üblich, dass die Professoren die Vorlesungen halten und die wissenschaftlichen Mitarbeiter für den Übungsbetrieb zuständig sind. Im Folgenden wird die Rolle der Professoren als **Dozent** und die der wissenschaftlichen Mitarbeiter als **Übungsleiter** bezeichnet. In mehrfachen Gesprächen mit Dozenten und Übungsleitern konnte erfolgreich die nun dargestellte Menge von User-Stories erfasst werden. Die Beschreibung erfolgt aus der Perspektive des jeweiligen Anwenders in Anlehnung an das Connextra-Format (siehe Abschnitt 2.1.3).

Erstellen von Übungsblättern und Klausuren Als Übungsleiter möchte ich die Möglichkeit haben, Übungsblätter und Klausuren bequem über eine Weboberfläche erstellen zu können, um meinen Arbeitsablauf zu erleichtern. Die Aufgaben sollen nach Anlegen eines Blattes oder einer Klausur mit direkter Zuordnung neu erstellbar sein oder von alten Blättern oder Klausuren kopiert werden können. Eine Aufgabe darf nur einmal verwendet werden, zur erneuten Verwendung muss sie zunächst kopiert werden. Es soll weiterhin die Möglichkeit geben, pro Blatt/Klausur ein eigenes Template festzulegen, um eine gewisse Flexibilität bei der Gestaltung zu haben. Pro Veranstaltung soll es ein Standard-Template geben, das verwendet wird, sofern kein Template angegeben worden ist. Schließlich möchte ich Übungsblätter und Klausuren als PDF exportieren können. Das System soll um weitere Exportformate erweiterbar sein.

Erstellen von Aufgaben Als Übungsleiter möchte ich Aufgaben über ein Formular erstellen. Jede Aufgabe besteht aus einem optionalen Titel, einem Aufgabentext, einem Lösungstext, optionalen Dateianhängen und einer optionalen Menge von

Schlagworten zur stichpunktartigen Beschreibung der inhaltlichen Thematik. Aufgabenstellung und Lösung sollen in Latex angegeben werden. Weiterhin muss es möglich sein, den Inhalt angehängter Dateien im Latex-Code einzufügen oder die Dateien geeignet zu referenzieren. Neben dem Neuerstellen sollen vorhandene Aufgaben zur erneuten Verwendung samt ihrer Anhänge kopierbar sein.

Suchen nach Aufgaben Als Übungsleiter möchte ich die Möglichkeit haben nach Aufgaben zu suchen, um leicht festellen zu können, welche Aufgaben in den letzten Jahren zu einer bestimmten Thematik verwendet worden sind. Dazu soll es ein Suchfeld geben, in das entsprechende Schlüsselwörter eingegeben werden können. Zur Einschränkung auf bestimmte Veranstaltungen soll das Kürzel der Veranstaltung im Suchfeld anzugeben sein. Die eingegebenen Schlüsselworte sollen vorrangig mit den Schlagworten der Aufgaben abgeglichen werden, aber auch Titel und Inhalt müssen mit verringerter Gewichtung Berücksichtigung finden.

Ergebnisimport Als Übungsleiter möchte ich die Möglichkeit haben, die mit einem externen Tool erfassten Punkte der Studenten in Form einer CSV-Datei zur späteren Auswertung zu importieren. Noch nicht im System existente Studenten sollen dabei aus den Daten in der CSV-Datei automatisch erstellt und zur entsprechenden Veranstaltung hinzugefügt werden.

Auswertung von Aufgaben Als Übungsleiter oder Dozent möchte ich anhand der importierten Punkte den Erfolg einer Aufgabe auswerten können, um diese für die Zukunft eventuell verbessern zu können. Dazu sollen die Zahl der Bearbeitungen, die durchschnittliche Punktzahl und die Punkteverteilung als Histogramm in der Übersicht der Aufgabe angezeigt werden.

Studenteninformationen Als Dozent oder Übungsleiter möchte ich mir alle vorhandenen Informationen zu einem Studenten anzeigen lassen, um mir ein Bild von seinen Leistungen in meinen Veranstaltungen machen zu können. Dies soll das Erstellen von Gutachten für Stipendien erleichtern oder zur Entscheidungsfindung bei Preisvergaben beitragen. Dazu möchte ich sehen können, welche Veranstaltungen der Student bei mir gehört hat und in welchem Studiengang er studiert. Zur Veranstaltung soll sowohl das Prüfungsergebnis als auch die Durchschnitts- und Gesamtpunktzahl in den Übungsblättern angezeigt werden. Weiterhin soll angezeigt werden, wie der Student prozentual gesehen im Vergleich zu anderen steht.

Suchen von Studenten Als Dozent oder Übungsleiter möchte ich über ein Suchfeld nach Studenten suchen können, um schnellen Zugriff auf deren Daten zu erhalten. Studenten sollen durch ihren Vor- und Nachnamen, ihren Rechenzentrum-Logins und ihre Matrikelnummer auffindbar sein.

Zugriffsrechte Als Dozent möchte ich nicht, dass unbefugte Personen die Ergebnisse meiner Studenten oder die Aufgabenblätter und Klausuren meiner Veranstaltungen sehen können. Andere Dozenten dürfen nur ihre eigenen Studenten, Blätter und Klausuren sehen. Meine Übungsleiter hingegen sollen Zugriff auf alle meine Aufgabenblätter, Klausuren und Studentenleistungen haben.

Mit diesen User-Stories ist ein Überblick über die gewünschten Funktionalitäten gegeben. Die konkreten Details und Akzeptanzkriterien werden erst im Rahmen der jeweiligen Iteration im erneuten Gespräch mit dem entsprechenden Anwender genauer spezifiziert.

3.2 Benutzeroberfläche

Im Rahmen des Behaviour-Driven Development wird den späteren Nutzern der Applikation nach jeder Iteration eine funktionierende Version mit den hinzugekommenen Funktionalitäten präsentiert (siehe Abschnitt 2.1.3). Um dem Anwender eine ausdrucksstärkere Vision bezüglich des Endergebnisses zu liefern, ist es sinnvoll, spätestens im Rahmen der ersten Iteration, ein grundlegendes Konzept für die optische Gestaltung und deren technischer Umsetzung zu entwickeln. Die Benutzeroberfläche ist ein bislang durch die Wahl der verwendeten Methoden und Technologien fast uneingeschränkter Bereich. Dieser Abschnitt stellt daher zunächst die zur Gestaltung verwendeten Werkzeuge und Bibliotheken vor und zeigt schließlich einige beispielhafte Anwendungen aus der Applikation.

3.2.1 Semantically Awesome Stylesheets

Semantically Awesome Stylesheets (SASS)[1] ist eine Erweiterung von Revision 3 der **Cascading Stylesheets** (CSS3). Cascading Stylesheets werden verwendet, um Dokumentinhalt und Dokumentpräsentation voneinander zu trennen. Der Dokumentinhalt wird in Weboberflächen in der Regel in der Markup-Sprache **HTML** formuliert. Die Art der Darstellung, zum Beispiel das Layout oder die Schriftart, werden unabhängig davon in CSS definiert. SASS bietet gegenüber CSS ähnliche Vorteile wie HAML gegenüber ERB. Neben verbesserter Lesbarkeit lassen sich durch die Einführung von **Variables**, **Mixins** und **Nesting** zahlreiche, in CSS notwendige, Redundanzen vermeiden. Dies führt zu einfacherer Wartbarkeit, da Änderungen nur noch an einer Stelle durchgeführt werden müssen. SASS wird vor der Auslieferung an den Client zu standardkonformem CSS kompiliert.

Listing 29 zeigt ein ein SASS-Beispiel. In Listing 30 ist der äquivalente CSS-Code dargestellt. Mittels der Variable `$blue` werden redundante Farbangaben eliminiert. Der Mixin `blue-border` spezifiziert Dicke, Farbe und Beschaffenheit einer Umrandung. Mit dem Ausdruck `+blue-border` lässt sich dieser Style an entsprechender Stelle einfügen. Im CSS-Code sind die Style-Definitionen der Umrandung dreimal vorhanden, in SASS genügt die Festlegung an einem zentralen Ort. Durch das Nesting der Elemente `ul` und `li` in Zeile 11ff des SASS-Codes wird ausgedrückt, dass die entsprechenden Style-Definitionen nur innerhalb der Klasse `.main-navigation` Gültigkeit haben. Alle Definitionen für Unterelemente befinden sich in SASS eingerückt unterhalb des Elternknotens und lassen sich so leicht identifizieren und warten. In CSS muss eine separate Style-Definition der Form `.main-navigation ul li` angelegt werden. Eigentlich zusammengehörige Definitionen werden häufig weit in der CSS-Datei verstreut und sind

[1] `http://sass-lang.com/`

```
$blue: #3bbfce

@mixin blue-border
  border-color: $blue
  border: 1px solid

.main-navigation
  +blue-border
  color: darken($blue, 3%)
  margin: 10px
  ul
    li
      +blue-border
      color: black

.sub-navigation
  +blue-border
  color: white
  margin: 3px
```

Listing 29: SASS Beispiel

```
.main-navigation {
  border-color: #3bbfce;
  border: 1px solid;
  color: #32b8c8;
  margin: 10px;
}

.main-navigation ul li {
  border-color: #3bbfce;
  border: 1px solid;
  color: black;
}

.sub-navigation {
  border-color: #3bbfce;
  border: 1px solid;
  color: white;
  margin: 3px;
}
```

Listing 30: CSS Beispiel

deswegen schwierig wartbar. Schon dieses kleine Beispiel macht die wesentlichen Vorteile von SASS gegenüber CSS deutlich. Bei Stylesheets mit größerem Umfang fallen diese natürlich noch deutlich stärker ins Gewicht. Zur Gestaltung der Benutzerobfläche wird daher im Folgenden ausschließlich SASS eingesetzt.

Eine besondere Schwierigkeit bei der Erstellung von Webseiten mit CSS ist die **Cross-Browser Kompatibilität**. Unterschiedliche Browser interpretieren die Style-Definitionen stellenweise verschieden. Dies führt häufig zu unerwarteten Ergebnissen in der Darstellung. Es muss mühsam sichergestellt werden, dass die verwendeten Ausdrücke in allen Browsern korrekt funktionieren. Zur Vermeidung derartiger Probleme wird das SASS-Framework **Compass**[2] eingesetzt. In Listing 31 ist ein Compass-Mixin zur Erzeugung eines Schattens dargestellt, Listing 32 zeigt das daraus von erzeugte CSS.

```
+box-shadow(0 1px 0 #111)
```

Listing 31: Compass-Mixin Beispiel

```
-moz-box-shadow: 0 1px 0 #111;
-webkit-box-shadow: 0 1px 0 #111;
-o-box-shadow: 0 1px 0 #111;
box-shadow: 0 1px 0 #111;
```

Listing 32: Beispiel für von Compass erzeugtes, Cross-Browser kompatibles CSS

Compass stellt unter Verwendung des CSS-Frameworks **Blueprint**[3] eine Menge von Mixins zur Verfügung, welche automatisch die nötigen Anpassungen vornehmen. Um

[2] http://compass-style.org/
[3] http://blueprintcss.org/

maximale Cross-Browser Kompatibilität zu gewährleisten, werden zur Gestaltung – sofern möglich – bevorzugt Compass-Mixins eingesetzt.

3.2.2 CoffeeScript

CoffeeScript[4] ist eine Abstraktionsschicht für die Multiparadigmen-Programmiersprache **JavaScript**. Die Version 1.0 wurde im Dezember 2010 veröffentlicht. JavaScript unterstützt die Paradigmen Objektorientierung, Imperative und Funktionale Programmierung. Die ursprüngliche Anwendung von JavaScript liegt in der Erweiterung statischer HTML-Seiten um dynamische Funktionen, beispielsweise asynchrone Serverkommunikation zur Manipulation serverseitiger Daten oder zum Nachladen neuer Inhalte. JavaScript kann ebenfalls zur Verbesserung der Benutzeroberfläche durch Animationen und andere Funktionen eingesetzt werden. Die Erfinder von CoffeeScript beschreiben die Intention ihrer Entwicklung wie folgt:

> „CoffeeScript is a little language that compiles into JavaScript. Underneath all those awkward braces and semicolons, JavaScript has always had a gorgeous object model at its heart. CoffeeScript is an attempt to expose the good parts of JavaScript in a simple way."

CoffeeScript ist inspiriert von den Sprachen Ruby, Python und Haskell. Neben erhöhter Lesbarkeit und kürzerem Code im Vergleich zu JavaScript werden zusätzliche Funktionalitäten wie **Array Comprehensions** (syntaktischer Zucker zur Erzeugung von Listen aus anderen Listen) hinzugefügt. CoffeeScript wird in vorhersehbarer Weise zu JavaScript **transkompiliert** (Übersetzung einer Programmiersprache in eine andere). Zur Laufzeit existiert kein zusätzlicher Overhead gegenüber einer direkten Implementation in JavaScript. In einigen Fällen kann CoffeeScript sogar effizienteren Code erzeugen, als die meisten Entwickler ihn in JavaScript schreiben würden.

```
1  info = (student, matrnr) ->          # Der Code erzeugt ein
2    "#{student} (#{matrnr}) \n"        # Dialogfenster mit
3                                       # folgendem Textinhalt:
4  students =                           #   Willi (123456)
5   Willi: 123456,                      #   Susi (654321)
6   Susi: 654321
7
8  all = for student, matrnr of students
9        info student, matrnr
10
11 alert all.join(" ")
```

Listing 33: CoffeeScript-Beispiel

In Listing 33 ist ein Beispiel dargestellt. Listing 34 zeigt den resultierenden JavaScript-Code. Unnötige Syntaxelemente wie Klammern oder Semikola können in CoffeeScript

[4] http://coffeescript.org/

bei Eindeutigkeit weggelassen werden. Mittels des Operators -> lassen sich anonyme Funktionen in übersichtlicher Weise definieren. In den Zeilen 1 und zwei von Listing 33 wird eine solche anonyme Funktion mit den beiden Parametern `student` und `matrnr` definiert und der lokalen Variable `info` zugewiesen. Die Funktion interpoliert die übergebenen Werte mit einer zu Ruby ähnlichen Syntax in einen String.

```
1  var all, info, matrnr, student, students;
2
3  info = function(student, matrnr) {
4    return "" + student + " (" + matrnr + ") \n";
5  };
6
7  students = {
8    Willi: 123456,
9    Susi: 654321
10 };
11
12 all = (function() {
13   var _results;
14   _results = [];
15   for (student in students) {
16     matrnr = students[student];
17     _results.push(info(student, matrnr));
18   }
19   return _results;
20 })();
21
22 alert(all.join(" "));
```

Listing 34: Erzeugtes JavaScript zu Listing 33

Die Zeilen 8 und 9 von Listing 33 zeigen die Verwendung einer Array Comprehension. CoffeeScript erkennt, dass es sich bei der Variable `students` um eine Menge von Schlüssel-Wert Paaren handelt und übergibt entsprechend den Schlüssel als `student` und den Wert als `matrnr`. Für alle Paare wird die vorher definierte anonyme Funktion `info` mit diesen Parametern aufgerufen. Die Ergebnisse werden in einem Array gesammelt und der Variable `all` zugewiesen.

Zusammenfassend lässt sich sagen, dass CoffeeScript-Code im Vergleich zu äquivalentem JavaScript-Code in der Regel kürzer und auch lesbarer ist. Dies erleichtert die Übersicht und auch die Verständlichkeit. Die Verwendung von CoffeeScript ist damit voll und ganz im Sinne der Agilen Softwareentwicklung. Da CoffeeScript direkt zu JavaScript transkompiliert wird, lassen sich nahtlos beliebige Bibliotheken einbinden. Im Rahmen dieser Arbeit wurden die Bibliotheken **jQuery**[5] und **jQuery UI**[6] verwendet.

[5] `http://jquery.com/`
[6] `http://jqueryui.com/`

3.2.3 Umsetzungsbeispiele aus der Applikation

Zur Vermittelung eines grundsätzlichen Eindrucks bezüglich der Oberflächengestaltung werden in diesem Abschnitt einige Beispiele gezeigt und die zugrunde liegenden Intentionen erläutert.

Basiskonzept

In Abbildung 3.1 ist die Ansicht nach der Authentifizierung dargestellt.

Abb. 3.1: Basislayout der Applikation

Der Nutzer wird nach dem Einloggen umgehend zur Übersichtsseite seiner Veranstaltungen weitergeleitet. Das Layout besteht im Wesentlichen aus zwei Spalten. In der linken Spalte ist die Navigationsleiste angesiedelt, die rechte Spalte beherbergt den eigentlichen Inhalt der jeweiligen Seite. Auf gesonderte Kopf- oder Fußzeilen wurde aus Gründen der Übersichtlichkeit verzichtet, da dafür keine Notwendigkeit besteht. Ältere Monitore oder Beamer verwenden häufig Auflösungen mit einer Breite von 1024 Pixeln. Damit die Applikation auch auf solchen Ausgabegeräten ohne Umbrüche dargestellt werden kann, nehmen Navigationsleiste und Inhaltsbereich gemeinsam stets 960 Pixel ein.

Die Navigationsleiste beinhaltet Links zu den wichtigsten Funktionalitäten. Seitenspezifische Navigationselemente werden in einem neuen Block dargestellt, im Beispiel wären dies „Chronologisch“ und „Gruppiert“, welche unterschiedliche gestaffelte Ansichten der Veranstaltungsliste ermöglichen. Der Link zu dem momentan aktiven Bereich wird zur visuellen Unterstützung der Orientierung hervorgehoben. Im Beispiel ist die Schaltfläche „Veranstaltungen“ besonders markiert, da sich der Nutzer auf der Übersichtsseite der Veranstaltungen befindet. Der Mauszeiger wurde über die Schaltfläche „Aufgaben“ bewegt, woraufhin diese ebenfalls hervorgehoben wird. Damit erhält der Anwender während der Navigation ein verbessertes visuelles Feedback und kann dadurch leichter erkennen, auf welcher Schaltfläche sich der Mauszeiger momentan befindet.

Elemente im Inhaltsbereich füllen den zur Verfügung stehenden Platz stets vollständig aus. Befinden sich mehrere Elemente im Inhaltsbereich, so werden diese durch einen global festgelegten, vertikalen Abstand voneinander getrennt. Statusmeldungen der Applikation, wie „Sie haben sich erfolgreich eingeloggt.“, werden zur besseren Wahrnehmung farblich hervorgehoben. Erfolgsmeldungen erhalten einen grünen Hintergrund, Fehlermeldungen einen roten und Hinweise einen gelben. Ein Mausklick auf die Statusmeldung hat deren animierte Ausblendung zur Folge. Damit lässt sich der Platz nach dem Lesen wieder vollständig ausnutzen.

Tabellendarstellung

Tabellen bieten eine übersichtliche Darstellungsmöglichkeit für eine Vielzahl von relevanten Daten der Applikation. Abbildung 3.2 zeigt einen Ausschnitt aus der Darstellung eines Übungsblattes, dessen Aufgaben in tabellarischer Form aufgeführt sind. Tabellenzeilen werden zur besseren Unterscheidung in alternierenden Farben dargestellt. Nach dem Laden der Seite wird dafür mittels CoffeeScript zu ungeraden Tabellenzeilen eine zusätzliche Klasse hinzugefügt, welche eine – gegenüber den anderen Zeilen – etwas abgedunkelte Hintergrundfarbe aufweist.

Abb. 3.2: Ausschnitt aus der Darstellung eines Aufgabenblattes

Die dargestellte Beispieltabelle beinhaltet neben purem Text auch Links, zum Beispiel findet sich in der Spalte „Titel“ ein Link zur entsprechende Aufgabe. Will man nun abhängig vom Inhalt einer anderen Spalte zur entsprechenden Veranstaltung navigieren, so kann die Hervorhebung der ganzen Tabellenzeile sinnvoll sein. Dies geschieht durch Hinzufügen eines auf Maus-Ereignisse reagierenden Eventlisteners mittels CoffeeScript. Wenn die Maus über das Element bewegt wird, wird eine zusätzliche Klasse mit einer aufgehellten Hintergrundfarbe hinzugefügt. Wird die Maus wieder weg bewegt, so wird auch die Klasse wieder von dem Element entfernt.

Ist es dem aktuell eingeloggten Nutzer möglich mit den in den Tabellenzeilen dargestellten Datensätzen zu interagieren, so beinhaltet die letzte Spalte eine Menge von Links zur Durchführung entsprechender Aktion. Um Platz zu sparen, werden zur Darstellung der Links möglichst deskriptive Icons verwendet. Beim Bewegen der Maus über ein Icon erscheint zusätzlich ein Hinweis, welche Aktion bei Klick des Links ausgeführt wird.

Ein- und Ausblenden von Inhalten

Um unnötiges Scrollen zu vermeiden, ist es möglich, Teile des Inhaltes ein- oder auszublenden. Die Elemente mit der Aufschrift „Aufgaben“ und „LaTeX Template“ in Abbildung 3.2 agieren als Steuerelemente. Ein Klick auf die jeweilige Aufschrift führt zur Einbeziehungsweise Ausblendung des untergeordneten Inhalts. Im Falle des Steuerelements „Aufgaben“ würde die Tabelle mit den Aufgaben ausgeblendet. Um einfacher erkennen zu können, welche Inhalte aktiv sind, wird der vertikale Abstand zu den Nachbarelementen leicht vergrößert. Kann der Nutzer Aktionen bezüglich des untergeordneten Inhalts durchführen, so befinden sich entsprechende Icons am rechten Rand des Steuerelements. Der Inhalt des Steuerelements „LaTeX Template“ ist momentan ausgeblendet. Ein Klick auf den Link würde das Template einblenden. Gleichzeitig würde automatisch nach unten gescrollt werden, um den eingeblendeten Inhalt optimal im Bildbereich zu positionieren.

3.3 Modelklassen, Ressourcen und Datenbankschema

In diesem Abschnitt werden die aufgrund der Anforderungen identifizierten Entitäten, deren Beziehungen und die Abbildung auf eine Menge von Modelklassen und Ressourcen dargestellt. Damit steht implizit durch die Konventionen von Rails (siehe Abschnitt 2.3.3) auch das Datenbankschema fest. Obwohl die hier dargestellten Zusammenhänge inkrementell entstanden sind (siehe Abschnitt 2.1.3), werden sie zur besseren Übersicht ganzheitlich präsentiert.

In Abbildung 3.3 ist ein vereinfachtes ER-Diagramm der Applikation dargestellt. Die Benennung der Entitäten und Beziehungen ist bereits konform zu den Konventionen von Rails (siehe Abschnitt 2.3.3). Die meisten Attribute sind aus Gründen der Übersicht hier nicht aufgeführt. Eine vollständige Darstellung als ER-Diagramm oder UML-Klassendiagramm lässt sich bei Bedarf innerhalb der Applikation mit Hilfe der Bibliotheken Railroady[7] und Rails-ERD[8] direkt aus den Modelklassen erzeugen.

Ein **User** (Nutzer) kann mehrere **StudyCourses** (Studiengänge) studieren. Ist jemand in einem Master-Studiengang immatrikuliert, so hat derjenige vermutlich vorher einen Bachelor-Studiengang absolviert. Außerdem wird der Beginn des jeweiligen Studiengangs festgehalten. Des Weiteren ist diese Beziehung optional, da nicht immer zu allen Personen entsprechende Daten vorhanden sind.

[7] `https://github.com/preston/railroady`
[8] `http://rails-erd.rubyforge.org/`

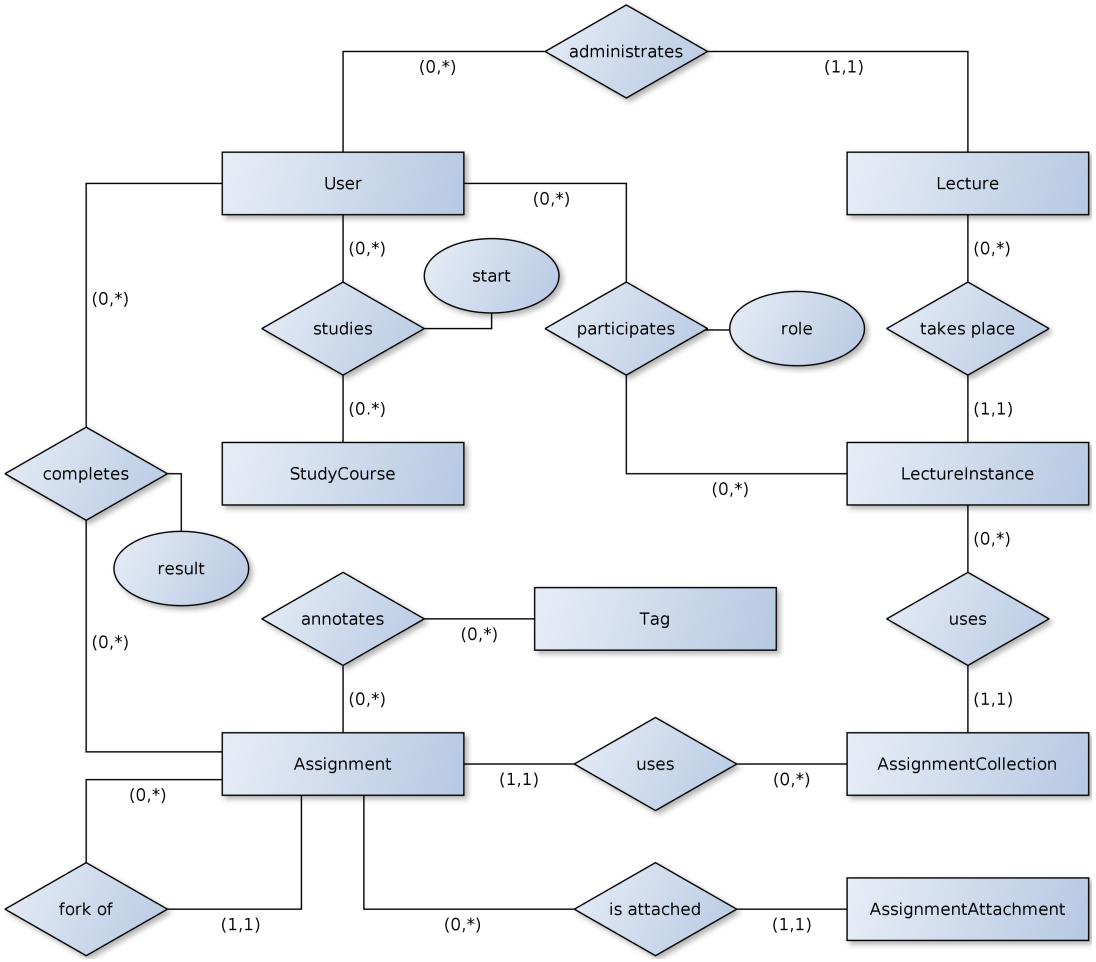


Abb. 3.3: Vereinfachtes ER-Diagramm der Applikation mit Kardinalitäten in min-max-Notation

Die Nutzer nehmen mit unterschiedlichen Rollen an **LectureInstances** (Veranstaltungsausprägungen) teil. Mögliche Rollen sind *Dozent, Übungsleiter, Tutor* und *Student*. Ein Beispiel für eine Veranstaltungsausprägung wäre *Algorithmen und Datenstrukturen WS 2011/2012*. Zur verbesserten Übersicht und erleichterten Administration der Veranstaltungsausprägungen sind diese einer übergeordneten **Lecture** (Veranstaltung) zugeordnet. Die Veranstaltung zu dem vorherigen Beispiel wäre *Algorithmen und Datenstrukturen*. Eine Ausprägung einer Veranstaltung entsteht also, wenn diese in einem konkreten Semester durchgeführt wird. Jede Veranstaltung hat einen Nutzer als Administrator, wobei ein Nutzer mehrere Veranstaltungen verwalten darf. Der Administrator legt neue Veranstaltungsausprägungen an und fügt die initialen Nutzer (Dozent, Übungsleiter) hinzu.

Innerhalb einer Veranstaltungsausprägung werden mehrere Aufgabenblätter und Klausuren verwendet. Diese unterscheiden sich in der Umsetzung lediglich durch ihre Klassifikation und werden daher zur Entität **AssignmentCollection** (Aufgabensammlung) zusammengenfasst. Eine Aufgabensammlung kann nur in genau einer Veranstaltungsausprägung verwendet werden. Eine Aufgabensammlung beinhaltet ihrerseits mehrere **Assignments** (Aufgaben). Eine Aufgabe ist immer genau einer Aufgabensammlung

zugeordnet. Aufgaben können mehrere **AssignmentAttachments** (Dateianhänge) zugeordnet sein, zum Beispiel Quelltexte oder Bilder, welche im Dateisystem abgelegt werden. In der Datenbank werden lediglich die Pfade zu den Dateien gespeichert. Eine Aufgabe kann ein **Fork** (Kopie) einer anderen Aufgabe sein und auch beliebig viele Kopien haben. Aufgaben können mit **Tags** (Schlagworten) annotiert werden, welche die Thematik näher beschreiben. Eine Aufgabe kann mehrere Schlagworte haben und ein Schlagwort kann für viele Aufgaben verwendet werden. Schließlich wird durch die Bearbeitung von Aufgaben durch die Nutzer ein Ergebnis erzielt.

Die Entitäten mit ihren Attributen können direkt auf Modelklassen und entsprechende Datenbanktabellen abgebildet werden. Fast alle dargestellten Entitäten werden mit Hilfe entsprechender Routen und Controller als Ressourcen zur Verfügung gestellt (siehe Abschnitt 2.3.5). Die einzige Ausnahme bilden die Schlagworte. Diese sollen automatisch und intern von der Applikation verwaltet und nicht erst über die Schnittstelle als Ressource angelegt werden müssen.

Zur Umsetzung der **n:m-Beziehungen** (many-to-many) ist stets eine zusätzliche Tabelle nötig, welche die Beziehung mittels Fremdschlüsseln zu den jeweiligen Entitäten herstellt. Im allgemeinen Fall enthält diese Tabelle neben den Fremdschlüsseln keine weiteren Attribute. Eine entsprechende Ressource zur Verwaltung der Beziehung bereitzustellen, gestaltet sich damit allerdings schwierig. Im professionellen Rails-Umfeld wird in solch einer Situation die Einführung eines **Join Models** als beste Vorgehensweise betrachtet (vergleiche [Pyte 10, Fern 10]), insbesondere dann, wenn die Beziehung zusätzliche Attribute benötigt. Wie jedes andere Model erhält das Join Model einen artifiziellen Primärschlüssel namens `id`. Damit ist auch die Beziehung leicht als Ressource adressierbar und kann mit entsprechendem Routing und Controller umgesetzt werden. Für die n:m-Beziehungen `completes, administrates` und `participates` werden entsprechend die Join Models `AssignmentResult, Study` und `LectureInstanceParticipation` eingeführt. Die Beziehung `annotates` ist nicht attributiert und soll auch nicht als Ressource bereitgestellt werden, sie wird daher nicht als Join Model umgesetzt.

Als Datenbank kommt **MySQL**[9] mit der Storage-Engine **InnoDB**[10] zum Einsatz. Die MySQL-Standard-Storage-Engine **MyISAM**[11] unterstützt die von Rails benötigten und im Folgenden erläuterten **ACID**-Eigenschaften für Transaktionen nicht. Innerhalb einer Transaktion durchgeführte Änderungen müssen entweder ganzheitlich oder gar nicht durchgeführt werden (**A**tomicity). War die Datenbasis vor einer Transaktion in einem konsistenten Zustand, so muss sie dies auch danach wieder sein (**C**onsistency). Während der Transaktion darf temporär auch ein inkosistenter Zustand vorherrschen. Außerdem dürfen sich Transaktionen nicht gegenseitig beeinflussen, sie müssen ausgeführt werden, als seien sie die einzige Transaktion im System (**I**solation). Schließlich müssen alle Änderungen einer erfolgreichen Transaktion auch garantiert persistiert werden (**D**urability). Ein Nachteil von InnoDB gegenüber MyISAM besteht in der fehlenden Möglichkeit zur Volltextindizierung. Zur effizienten Volltextsuche muss also auf andere Hilfsmittel zurückgegriffen werden. Da Rails aber ACID-konforme Transaktionen zur

[9] `http://www.mysql.de/`
[10] `http://dev.mysql.com/doc/refman/5.1/de/innodb.html`
[11] `http://dev.mysql.com/doc/refman/5.1/de/myisam-storage-engine.html`

Gewährleistung der dynamischen Integrität im ActiveRecord-Lifecycle benötigt, fiel die Wahl letztlich auf InnoDB.

3.4 Zugriffsbeschränkung

Hinreichende Restriktionen und Sicherheitsvorkehrungen spielen für die Applikation eine wichtige Rolle. Unbefugte dürfen keinesfalls Zugriff auf die sensiblen Daten erhalten. Außerdem soll das System von mehreren Arbeitsgruppen am Institut für Informatik verwendet werden. Es muss also für eine eingeschränkte Sichtbarkeit der Informationen gesorgt werden, da jeder Nutzer nur Zugriff auf seine eigenen Lehrveranstaltungen, Aufgaben und Studenten erhalten darf.

Die wesentlichen Hilfsmittel zur Beschränkung des Zugriffs auf die Applikation und ihre Funktionen sind **Authentifizierung** und **Autorisierung**. Authentifizierung bezeichnet die Verifikation einer Nutzeridentität. Ein Anwender authentifiziert sich mit seinem Loginnamen und Passwort. Versucht ein authentifizierter Nutzer eine Aktion durchzuführen, so muss stets die entsprechende Berechtigung geprüft werden. Dieser Prüfvorgang wird Autorisierung genannt. Zunächst werden die grundlegenden Berechtigungen erläutert, ehe die Umsetzung im Detail beleuchtet wird.

Im System wird während der Installation ein globaler Administrator angelegt. Dieser ist in der Lage, weitere Nutzer hinzuzufügen. Außerdem erstellt der globale Administrator die Veranstaltungen und weist ihnen einen Veranstaltungsadministrator zu, welcher dadurch die Berechtigung zur Verwaltung entsprechender Veranstaltungsausprägungen erhält.

Der Administrator einer Veranstaltungsausprägung kann dieser nun weitere Teilnehmer mit den Rollen *lecturer* (Dozent), *teaching_ assistant* (Übungsleiter), *tutor* (Tutor) und *student* (Student) hinzufügen, wobei Studenten automatisch importiert werden können, die anderen Teilnehmer jedoch von Hand angelegt werden müssen. Dozenten und Übungsleiter verfügen faktisch über identische Berechtigungen in der Veranstaltungsausprägung, sie werden hauptsächlich zu Darstellungszwecken in ihrer Rolle unterschieden. Sie können teilnehmende Studenten und Tutoren verwalten, Aufgabenblätter und Klausuren mit entsprechenden Aufgaben anlegen und zugehörige Punktedaten importieren. Für Studenten und Tutoren besteht vorerst keine Möglichkeit zum Einloggen in die Applikation, entsprechende Nutzeraccounts sind nicht aktiviert und haben somit auch keine Berechtigungen.

3.4.1 Authentifizierung

Zur Authentifizierung wird die Bibliothek **Devise**[12] verwendet. Sie wurde von der Firma *Plataforma tecnologia*[13] entwickelt und unter freier Lizenz zur Verfügung gestellt. Devise ist eine der meist verwendeten Bibliotheken zur Authentifizierung und damit gut erprobt und getestet [Fern 10]. Die Funktionalitäten sind modular aufgebaut und lassen sich je

[12] `https://github.com/plataformatec/devise`
[13] `http://www.plataformatec.com.br/en/`

nach Bedarf einzeln zur Applikation hinzufügen. Entsprechende Controller und Views werden direkt mitgeliefert und können bei Bedarf angepasst werden.

Außerdem lassen sich problemlos verschiedene Backends durch eine einfache Änderung der Konfiguration verwenden. Der Standard `database_authenticable` (Datenbankauthentifizierung) verwendet die Datenbank der Applikation zur Datenspeicherung und Authentifizierung. Passwörter werden dabei hinreichend verschlüsselt abgelegt. Bei Bedarf können aber auch ein CAS-Server, Facebook oder ein beliebiger OpenID-Dienst verwendet werden. Da in der Applikation lediglich `database_authenticable` zur Anwendung kommt, wird hier nicht weiter auf die alternativen Backends eingegangen und es sei für weitere Informationen auf die Devise-Dokumentation verwiesen.

Zur Nutzung der Datenbankauthentifizierung sind lediglich der Aufruf eines Class-Macros' (siehe Abschnitt 2.3.1) und das Hinzufügen der benötigten Spalten zur entsprechenden Tabelle erforderlich. In Listing 35 ist die Klasse `User` mit der notwendigen Konfiguration dargestellt.

```
1 class User < ActiveRecord::Base
2   devise :database_authenticatable, :rememberable
3 end
```

Listing 35: Devise Konfiguration in der Modelklasse

Der Code zum Anpassen der Tabelle lässt sich daraufhin automatisch generieren. Das Modul `rememberable` trägt Sorge dafür, dass man sich als Nutzer nicht bei jeder neuen Verwendung der Applikation neu einloggen muss, sondern die Session aktiv bleibt. Im ersten Moment scheint dies nicht konform zur ROA zu sein, tatsächlich stellt Devise für das Session-Management aber eine eigene Ressource bereit und ist damit in der Tat ROA-kompatibel. Schließlich muss noch festgelegt werden, auf welche Seite ein Nutzer nach erfolgreicher Authentifizierung weitergeleitet werden soll, dafür wurde die Veranstaltungsübersicht ausgewählt. Weitere Konfigurationen sind nicht notwendig.

Die Applikation besitzt keine öffentlichen Bereiche. Der Zugriff lässt sich damit leicht an zentraler Stelle regeln. Da sämtliche Interaktion über die Controller stattfindet, bietet sich deren gemeinsame Superklasse `ApplicationController` für diesen Zweck an. In Listing 36 ist der entsprechende Ausschnitt dargestellt.

```
1 class ApplicationController < ActionController::Base
2   protect_from_forgery
3   before_filter :authenticate_user!
4 end
```

Listing 36: Für die Authentifizierung relevanter Ausschnitt aus dem ApplicationController

Devise stellt innerhalb von Controllern die Methode `authenticate_user!` bereit. Die Methode prüft, ob derzeit ein authentifizierter Nutzer vorhanden ist. Ist dies der Fall, so wird die Anfrage verarbeitet. Kann keine valide Authentifizierung festgestellt werden, so wird der Client zum Authentifizierungsformular weitergeleitet. Der Zeitpunkt der

Überprüfung wird mit Deklaration der Methode als `before_filter` festgelegt. Derartige Filter werden vom Controller vor der Verarbeitung der Anfrage aufgerufen. Damit ist unbefugter Applikationszugriff global ausgeschlossen.

Die Methode `protect_from_forgery` wird von Rails bereitgestellt. Ihr Aufruf aktiviert einen Schutzmechanismus zur Abwehr von als **Cross-site request forgery** (CSRF) bezeichneten Angriffen. Ist einem Angreifer bekannt, dass ein Anwender in einem bestimmten System authentifiziert ist, so kann er sich dies zur unerlaubten Manipulation von Daten zu Nutze machen. Da eine Anfrage vom Typ GET in der ROA keine Seiteneffekte hat (siehe Abschnitt 2.2), ist entsprechender Schutz nur bei Verwendung der HTTP-Verben POST, PUT oder DELETE nötig. Als Gegenmaßnahme fügt Rails beim Rendern der Views automatisch ein speziell generiertes **Authenticity Token** als verstecktes Feld zu Formularen hinzu. Dieses wird bei der Verarbeitung entsprechender Anfragen geprüft und im Fehlerfall die Verarbeitung umgehend abgebrochen. Für weitere Details zu CSRF und dem Abwehrmechanismus in Rails sei auf entsprechende Literatur verwiesen [Rist 05, Powe 09].

Damit ist sichergestellt, dass nur eingeloggte Nutzer Zugriff auf die Applikation erhalten und deren Sessions nicht ohne Weiteres manipuliert werden können.

3.4.2 Autorisierung

Um dafür zu sorgen, dass authentifizierte Nutzer tatsächlich nur Aktionen durchführen können, zu denen sie auch berechtigt sind, wird die Bibliothek **CanCan**[14] verwendet. Dazu werden in Abhängigkeit des eingeloggten Nutzers sogenannte **Abilities** (Berechtigungen) definiert. Dies geschieht innerhalb des Konstruktors einer Klasse namens `Ability`, welcher den aktuellen Nutzer übergeben bekommt. Dazu muss im System die Methode `current_user` vorhanden sein, welche von Devise bereitgestellt wird. Die Überprüfung geschieht dabei nach dem **Whitelist**-Prinzip. Das bedeutet, dass zunächst alle Aktionen verboten werden. Der Nutzer kann also nur Aktionen durchführen, für die explizit eine entsprechende Berechtigung definiert wurde. Auf diese Weise ist sichergestellt, dass keine Restriktionen vergessen werden können. Ein Auszug der Klasse aus der Applikation ist in Listing 37 dargestellt.

Eine Berechtigung wird über ein Schlüsselwort in Form eines Symbols und über die Modelklasse der zu schützenden Ressource identifiziert. Zur konkreten Definition existieren drei Möglichkeiten, die im Folgenden anhand von Beispielen erläutert werden.

In Zeile 6 wird durch Aufruf der Methode `can` die Berechtigung `read` für Ressourcen vom Typ `Lecture` festgelegt. Der dritte Parameter ist hier ein optionaler Hash, der Einschränkungen bezüglich der Attribute macht, anhand derer die betroffenen Veranstaltungsinstanzen ermittelt werden können. In diesem Fall ist lesender Zugriff für Veranstaltungen erlaubt, die vom aktiven Nutzer administriert werden.

Die in Zeile 10 beginnende Definition ist davon abhängig, ob der Nutzer als Dozent oder Übungsleiter an einer Veranstaltung teilnimmt. Ist dies der Fall, so erhält er Zugriff auf die Suchfunktion für Nutzer. Welche Nutzer genau für die Suche zur Verfügung stehen,

[14] `https://github.com/ryanb/cancan`

```
1  class Ability
2    # Berechtigungsfunktionalität importieren
3    include CanCan::Ability
4
5    def initialize(user)
6      # Definition durch Attributeinschränkung
7      can :read, Lecture, admin_id: user.id
8
9      # Bedingte Definition
10     if user.lecturer_or_ta?
11       can :search, User
12     end
13
14     # Definition für konkrete Instanzen
15     can :show, User do |user_to_show|
16       user_to_show.student_of?(user)
17     end
18   end
19 end
```

Listing 37: Auszug aus der `Ability`-Klasse

lässt sich nicht ohne Weiteres durch eine Menge von Attributeinschränkungen ermitteln. Die Berechtigung regelt lediglich den Zugriff auf die Suchfunktion. Für Details bezüglich der Sichtbarkeitseinschränkung der Suche siehe die Abschnitte 3.4 und 3.6.

Schließlich lässt sich die Berechtigung auch direkt in Abhängigkeit einer konkreten Instanz definieren. Die in Zeile 14 beginnende Definition ist solch ein Beispiel. Der Methode `can` wird zusätzlich ein Block übergeben, der als Parameter die Instanz der zu schützenden Ressource erhält. In diesem Fall handelt es sich um die Berechtigung, die Daten eines konkreten Nutzers einzusehen. Dies soll genau dann möglich sein, wenn der zu zeigende Nutzer (`user_to_show`) Student des aktuellen Nutzers (`user`) ist. Eine Methode zur Feststellung dieser Tatsache ist in der Modelklasse implementiert und wird hier nicht näher erläutert.

In Listing 38 ist die Anwendung der vorher definierten Berechtigung zur Nutzeransicht dargestellt. Nach dem Laden des entsprechenden Nutzerobjektes wird dieses – zusammen mit dem Bezeichner der Berechtigung – an die Methode `authorize!` übergeben. Die Modelklasse wird anhand der Klasse des übergebenen Objektes ermittelt und die Instanz an den entsprechenden Block übergeben. Das Ausrufungszeichen am Ende eines Methodennamens ist in Ruby ein Hinweis darauf, dass die Methode im Fehlerfall eine Exception wirft. Hat der aktuelle Nutzer also keine Berechtigung, die Daten des angeforderten Nutzers (`@user`) zu betrachten, so wird eine Exception geworfen. Besteht die Autorisierung zur Ansicht, wird die Verarbeitung fortgesetzt.

Das Werfen einer Exception im Controller hätte den sofortigen Abbruch der Anfragenverarbeitung zur Folge und würde dem Anwender eine Fehlermeldung präsentieren.

```
1 class UsersController < ApplicationController
2   def show
3     @user = User.find(params[:id])
4     authorize! :show, @user
5     respond_with(@user)
6   end
7 end
```

Listing 38: Autorisierung in der Action `show` des `UsersController`

Endnutzer einer Applikation sollten allerdings niemals eine Fehlermeldung zu sehen bekommen. Es besteht daher die Notwendigkeit, die Fehlermeldung an geeigneter Stelle zu behandeln. Listing 39 zeigt die Behandlung des Fehlers im `ApplicationController`.

```
1 class ApplicationController < ActionController::Base
2   rescue_from CanCan::AccessDenied do |exception|
3     flash[:error] = "Dafür haben Sie nicht die Berechtigung."
4     redirect_to root_path
5   end
6 end
```

Listing 39: Reaktion auf eine Autorisierungsverletzung im `ApplicationController`

Die Methode `rescue_from` erwartet als Parameter die Klasse der zu behandelnden Exception und einen Block mit entsprechendem Code. Der Block bekommt die Exception als Parameter übergeben, wodurch alle innerhalb einer Action geworfenen Exceptions entsprechenden Typs gefangen werden. Als Rückmeldung für den Anwender wird eine Fehlermeldung zur späteren Anzeige (ähnlich wie in Abbildung 3.1, nur mit rotem Hintergrund) festgelegt. Schließlich erfolgt eine Weiterleitung auf die Startseite. Auf diese Weise wird für sämtliche möglichen Aktionen sichergestellt, dass diese nur von dafür autorisierten Nutzern durchgeführt werden können.

3.4.3 Sichtbarkeiten

Ein repräsentatives Beispiel für die Einschränkungen von Sichtbarkeiten ist die Studentenübersicht eines Dozenten. Jeder Dozent kann auf die Collection-Ressource (siehe Abschnitt 2.3.5) der Nutzer zugreifen, also eine Liste aller Nutzer anfordern. Dabei muss sichergestellt werden, dass nur die Daten von Nutzern auftauchen, die Studenten des Dozenten sind, also an mindestens einer entsprechenden Veranstaltungsausprägung teilgenommen haben.

Zur Umsetzung dieser Anforderung werden sogenannte **Scopes** verwendet. Scopes werden in Rails eingesetzt, um die Menge der gewünschten Instanzen komfortable einschränken zu können. Ein einfaches Beispiel ist der Scope `ordered_by_last_name` in Zeile 2 von Listing 40. Dieser liefert alle geforderten Nutzerinstanzen aufsteigend sortiert nach ihrem Nachnamen und lässt sich so verwenden: `User.ordered_by_last_name`. Scopes

lassen sich beliebig verketten, zum Beispiel liefert der Ausdruck `User.ordered_by_last_name.where("first_name = 'Willi'")` alle Nutzer mit dem Vornamen *Willi* in entsprechender Sortierung. Der Scope zur Einschränkung der Nutzersichtbarkeit ist ebenfalls in Listing 40, beginnend in Zeile 4, dargestellt.

```
1  class User < ActiveRecord::Base
2    scope :ordered_by_last_name, order("last_name ASC")
3
4    scope :visible_for, lambda { |other|
5      # Ausschließen von SQL-Injection über other.id
6      unless other.is_a?(User)
7        raise ArgumentError, "other has to be of class User"
8      end
9
10     select("DISTINCT users.*").
11       joins("JOIN lecture_instance_participations vf_u_lip
12              ON vf_u_lip.user_id = users.id
13              AND vf_u_lip.role = 'student'").
14       joins("JOIN lecture_instances vf_u_li
15              ON vf_u_li.id = vf_u_lip.lecture_instance_id").
16       joins("JOIN lectures vf_lectures
17              ON vf_lectures.id = vf_u_li.lecture_id").
18       joins("JOIN lecture_instance_participations vf_o_lip
19              ON vf_o_lip.user_id = #{other.id}
20              AND (vf_o_lip.role = 'lecturer'
21                   OR vf_o_lip.role = 'teaching_assistant')").
22       joins("JOIN lecture_instances vf_o_li
23              ON vf_o_li.lecture_id = vf_lectures.id
24              AND vf_o_lip.lecture_instance_id = vf_o_li.id")
25   }
26 end
```

Listing 40: Sichtbarkeitseinschräkung in der Klasse `User`

Da der Scope vom aktuell authentifizierten Nutzer abhängt, muss es eine Möglichkeit geben, diesen geeignet zu übergeben. Dies geschieht durch Verwendung eines parametrisierten **Lambda**. Lambdas verhalten sich wie an Methoden übergebene Blöcke (siehe Abschnitt 2.3.1), können aber an beliebiger Stelle definiert werden. Für weitere Details sei hier auf die Fachliteratur [Thom 09, Perr 10, Fern 10] verwiesen.

Zur Umsetzung der Sichtbarkeitseinschränkung sind mehrere komplexe Joins nötig, die hier nicht im Detail erläutert werden. Diese Anforderung übersteigt leider die Fähigkeiten von ActiveRecord, so dass man schließlich doch selbst entsprechendes SQL formulieren muss. Gleichzeitig muss aber ein Scope-Objekt zurückgeliefert werden, damit noch weitere Verkettungen mit dem Ergebnis durchgeführt werden können. Es sind also dennoch die Methoden aus der Query-API zum Zusammensetzen der Anfrage zu verwenden. Weiterhin müssen die verwendeten Tabellen eindeutig benannt werden, da-

mit bei weiteren Verkettungen keine Kollisionen bezüglich der Namen stattfinden. Sie erhalten das Präfix `vf_`, in Anlehnung an den Namen des Scopes (`visible_for`).

Listing 41 zeigt schließlich die Verwendung der beiden Scopes im `UsersController`. Nach der Autorisierung werden die für den aktuell authentifizierten Anwender sichtbaren Nutzer mittels des erstellten Scopes in gewünschter Sortierung angefordert und entsprechend weitergegeben.

```
1 class UsersController < ApplicationController
2   def index
3     authorize! :index, User
4     @users = User.visible_for(current_user).ordered_by_last_name
5     respond_with(@users)
6   end
7 end
```

Listing 41: Ausschnitt aus dem `UsersController` zur Nutzung der Sichtbarkeitseinschränkung

Für alle weiteren, in der Sichtbarkeit beschränkten Ressourcen wurde analog vorgegangen.

3.5 Aufgabenblätter und Klausuren

Das Erstellen von Übungsblättern und Klausuren ist der Kern der Applikation. Die wichtigsten Schritte der Umsetzung werden in den folgenden Abschnitt ausführlich erläutert.

3.5.1 Erstellen von Aufgabensammlungen

Zur Erstellung von Aufgabenblättern und Klausuren muss zunächst zur Seite der entsprechenden Veranstaltungsausprägung navigiert werden. Von dort aus gelangt der Anwender zu einem einfachen Formular, welches die nötigen Daten erfasst. Der Nutzer muss festlegen, ob es sich bei der anzulegenden Aufgabensammlung um ein Übungsblatt oder eine Klausur handelt. Weiterhin müssen Beginn und Ende der Bearbeitungszeit, sowie der Veröffentlichungszeitpunkt von Aufgabenstellung und Musterlösung angegeben werden. Das Eintragen eines Titels ist optional. Die folgenden Abschnitte beschreiben die wesentlichen Punkte zur Umsetzung der zugehörigen Models und Controller. Auf die View zur Erstellung wird nicht näher eingegangen, da diese lediglich ein Formular erzeugt. Die Übersichtsseite einer Aufgabensammlung ist in Abbildung 3.2 auf Seite 46 zu sehen und vermittelt einen Eindruck der Darstellung.

Models

Klausuren und Übungsblätter werden durch die Entität Aufgabensammlung mit entsprechend gesetztem Attribut `collection_type` repräsentiert. Der Ausschnitt eines Klassendiagramms in Abbildung 3.4 verdeutlicht die Beziehung zwischen Aufgabensammlungen und Veranstaltungsausprägungen, sowie deren – für die Erstellung relevanten – Methoden und Attribute.

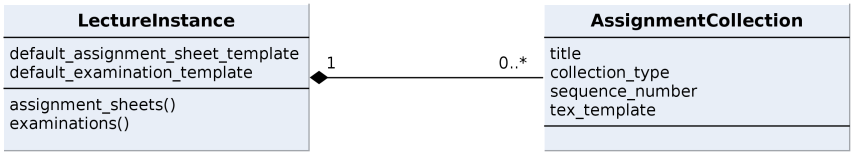


Abb. 3.4: Klassendiagramm in UML. Details ohne Relevanz für die Erläuterungen sind weggelassen worden.

Eine Aufgabensammlung ist immer genau einer Veranstaltungsausprägung zugeordnet und kann nicht ohne diese existieren. Zu einer Veranstaltungsausprägung können mehrere Klausuren und Übungsblätter vorhanden sein. Das Attribut `sequence_number` repräsentiert die fortlaufende Nummer der Aufgabensammlung. Die Nummerierung wird für Aufgabenblätter und Klausuren separat und automatisiert durchgeführt. Außerdem darf pro Aufgabensammlung eine Kombination aus `collection_type` und `sequence_number` nur einmalig existieren, das heißt, es darf jeweils nur eine Klausur bzw. ein Aufgabenblatt mit gleicher Nummer vorhanden sein. Bei Löschung einer Aufgabensammlung müssen die Nummern der anderen, zur selben Veranstaltungsausprägung gehörigen Aufgabensammlungen, entsprechend angepasst werden, um Lücken in der Nummerierung zu verhindern. Würden zum Beispiel drei Aufgabensammlungen mit den Nummern 1,2 und 3 existieren und die Aufgabensammlung mit Nummer 2 würde gelöscht, dann müsste die 3 zu einer 2 korrigiert werden. Schließlich wird bei der Erstellung noch das entsprechende Standard-Template aus der Veranstaltungsausprägung in das Attribut `tex_template` kopiert, um dieses pro Aufgabensammlung individuell anpassen zu können. Listing 42 zeigt den ersten Teil der Umsetzung in der Veranstaltungsausprägung.

```
1  class LectureInstance < ActiveRecord::Base
2    has_many :assignment_collections, dependent: destroy
3
4    def assignment_sheets
5      assignment_collections.assignment_sheets
6    end
7
8    def examinations
9      assignment_collections.examinations
10   end
11 end
```

Listing 42: Ausschnitt aus der Modelklasse `LectureInstance`

Das Class-Macro (siehe Abschnitt 2.3.1) `has_many` stellt die Methode `assignment_collections` bereit, welche automatisch anhand der Fremdschlüssel entsprechende Joins durchführt und die gewünschten Objekte zurückliefert. Die Angabe der Option `dependent: destroy` hat die Zerstörung sämtlicher assoziierter Aufgabensammlungen bei Löschung einer Veranstaltungsausprägung zur Folge. Die Methoden `assignment_sheets` und `examinations` dienen der einfacheren Zugänglichkeit entsprechender Aufgabensammlungen. Sie verwenden hier nicht dargestellte Scopes der Klasse `AssignmentCollection`.

In Listing 43 ist ein repräsentativer Ausschnitt aus der Klasse `AssignmentCollection` aufgeführt.

```
1  class AssignmentCollection < ActiveRecord::Base
2    belongs_to :lecture_instance
3
4    validates :sequence_number,
5      uniqueness: {scope: [ :lecture_instance_id,
6                            :collection_type ]}
7
8    before_validation :set_sequence_number, on: :create
9
10   private
11   def set_sequence_number
12     self.sequence_number = lecture_instance.
13       send("#{self.collection_type}s").count + 1
14   end
15 end
```

Listing 43: Ausschnitt aus der Modelklasse `AssignmentCollection`

Mittels des Class-Macro `belongs_to` wird die Beziehung zur Klasse `LectureInstance` umgesetzt. Sie stellt die Methode `lecture_instance` bereit, welche die zugehörige Veranstaltungsausprägung liefert.

Der in Zeile 4 beginnende Ausdruck definiert eine Validierung für das Attribut `sequence_number`. Die fortlaufende Nummer muss einzigartig (`uniqueness`) für jede Kombination aus dem Fremdschlüssel `lecture_instance_id` und dem `collection_type` der Aufgabensammlung sein. Schlägt diese Validierung fehl, so wird das Objekt nicht gespeichert und eine entsprechende Hinweismeldung zur Anzeige für den Anwender hinterlegt. Validierungen werden sowohl bei Erzeugung als auch bei Änderung eines Datensatzes durchgeführt.

In Zeile 8 wird ein sogenannter **Callback** definiert, dessen Aufruf bei Erzeugung eines neuen Objektes vor der Validierung geschieht. Der Name der aufzurufenden Methode wird als Parameter übergeben. Die Methode `set_sequence_number` zählt die Aufgabensammlungen der Veranstaltungsausprägung mit entsprechendem Typ und setzt die eigene Nummer auf den um eins erhöhten Wert. Dies geschieht durch Verwendung der Methode `send`, welche den dynamischen Aufruf einer Methode ermöglicht. Als Parameter wird der Name der Methode in Form eines Symbols oder eines Strings übergeben.

Wäre zum Beispiel der Typ der Aufgabensammlung `examination`, so würde die Methode `examinations` aufgerufen werden. Durch die Interpolation des eigenen `collection_type` in den Parametern werden stets die korrekten Instanzen gezählt, gleichzeitig muss aber mit Validierungen sichergestellt werden, dass `collection_type` keine unzulässigen Werte annehmen kann. Schließlich wird die ermittelte Zahl inkrementiert und über die entsprechende Setter-Methode als Instanzvariable gesetzt. Dies muss vor der Validierung geschehen, damit der errechnete Wert ebenfalls überprüft wird.

Das Kopieren des Standard-Templates aus der Veranstaltungsausprägung funktioniert analog zum Setzen der fortlaufenden Nummer. Zum Anpassen der Nummerierung bei Löschung einer Aufgabensammlung kommt ebenfalls ein geeigneter Callback ähnlicher Bauart zum Einsatz und wird daher auch nicht näher beleuchtet. Validierungen für die restlichen Attribute sind vorhanden, aber aus Gründen der besseren Übersicht im Beispiel nicht mit aufgeführt.

Es sei noch angemerkt, dass zwar explizit keine Transaktionen verwendet wurden, implizit aber dennoch der ganze Vorgang transaktional ausgeführt wird. Dies ist im Beispiel auch notwendig. Um einen Wert zu inkrementieren, muss dieser zuerst gelesen, um eins erhöht und wieder geschrieben werden. Geschieht dies nicht transaktional, so kann zwischen dem Lesen und Schreiben bereits eine Änderung durchgeführt worden sein, die dann einfach überschrieben wird. Der gesamte Prozess der Erstellung oder Aktualisierung eines Datensatzes wird von ActiveRecord als Transaktion durchgeführt, inklusive sämtlicher Callbacks und Validierungen. Es ist in den meisten Fällen nicht erforderlich, selbst Transaktionen innerhalb von Callback-Methoden zu starten.

Verschachtelte Ressource

Wenn Ressourcen in einer Eins-zu-Viele Beziehung zueinander stehen und die Vielen ausschließlich im Kontext des Einen manipuliert werden müssen, dann ist es im Rails-Umfeld gängige Praxis, die Viele-Ressource unter der Einen-Ressource zu verschachteln [Fern 10]. Im konkreten Fall werden den Veranstaltungsausprägungen die Aufgabensammlungen untergeordnet. Ziel dieser Vorgehensweise ist die vereinfachte Verwaltung der Beziehung. Würden beide Ressourcen alleinstehend umgesetzt werden, so müssten die Fremdschlüssel in View und Controller manuell abgegriffen und verarbeitet werden. Durch Verschachtelung der Ressourcen lässt sich dieser Aufwand vermeiden und führt zu übersichtlicherem Code. Zunächst muss das Routing angepasst werden. Listing 44 zeigt die entsprechende Routen-Konfiguration.

```
1 AssignmentManager::Application.routes.draw do
2   resources :lecture_instances do
3     resources :assignment_collections, except: [:index]
4   end
5   resource :assignment_collections, only: [:index]
6 end
```

Listing 44: Ausschnitt aus der Routen-Konfiguration

Verschachtelte Routen werden erzeugt, indem innerhalb eines an die Methode `resources` übergebenen Blockes weitere Routen definiert werden. Die Collection-Ressource (abgebildet auf die `index`-Action) der Aufgabensammlungen wird nicht verschachtelt und erhält eine eigene Routen-Definition, damit der Anwender eine Möglichkeit hat, eine Liste aller Datensätze zu erhalten. Würde die Collection-Ressource ebenfalls verschachtelt werden, so würde die Liste nur die Aufgabensammlungen zu einer konkreten Veranstaltungsausprägung beinhalten. Alle anderen Aktionen werden verschachtelt durchgeführt. Für die Ressource stehen die normalen Routen (siehe Abschnitt 27) zur Verfügung, sie erhalten nun aber zusätzlich die URI der übergeordneten Ressource als Präfix. Die Route zum Lesen der Ressource

```
GET assignment_collections/:id
```

wird durch Verschachtelung zu

```
GET lecture_instances/:lecture_instance_id/assignment_collections/:id
```

umgesetzt. Die restlichen Routen werden analog transformiert. Damit steht im Controller während der Verarbeitung zusätzlich der Parameter `:lecture_instance_id` zur Verfügung und muss nicht mehr gesondert übergeben werden.

Der Controller für die verschachtelte Ressource ist in Listing 45 dargestellt. Die Action `index` ist nicht mit aufgeführt, da sie analog zum `UsersController` mit entsprechender Sichtbarkeitseinschränkung umgesetzt ist (siehe Listing 41). Im Folgenden wird die Action `create` als exemplarisches Beispiel für die restlichen Actions erläutert.

Da die `lecture_instance_id` als Parameter übergeben wird, lässt sich die übergeordnete Veranstaltungsausprägung im Vorfeld der Action laden. Die entsprechende Methode (Zeile 14) fordert das Objekt von der Datenbank an und weist es einer Instanzvariablen zu. Über einen `before_filter` (Zeile 2) wird festgelegt, dass diese Methode vor der Ausführung jeder Action (abgesehen von `index`) aufgerufen wird. Da die Berechtigung zur Manipulation einer Aufgabensammlung direkt an die übergeordnete Veranstaltungsausprägung und deren Teilnehmer gebunden ist, lässt sich auch die Autorisierung mittels eines `before_filter` und der vorher geladenen Ressource ausführen (Zeile 4 und 23f). Schließlich wird mit der in den Zeilen 19ff definierten Methode noch ein einfacher Zugriff auf den Scope aller Aufgabensammlungen der Veranstaltungsausprägung ermöglicht.

Die eigentliche Implementierung der Action `create` (Zeile 7ff) gestaltet sich mit diesen Hilfsmitteln sehr übersichtlich. Der Scope `assignment_collections` hat die hilfreiche Eigenschaft, dass auf ihm direkt neue, assoziierte Objekte erzeugt werden können, Fremdschlüssel werden dabei automatisch verwaltet. Das Model trägt mit seinen Callbacks Sorge für das automatisierte Setzen der nicht vom Anwender eingegebenen Attribute (siehe Abschnitt 3.5.1). Der resultierende Code im Controller ist sehr übersichtlich, leicht verständlich und wartbar.

```ruby
class AssignmentCollectionsController < ApplicationController
  before_filter :grab_lecture_instance_from_lecture_instance_id,
     except: :index
  before_filter :authorize_nested_access,
     except: :index

  def create
    @assignment_collection = assignment_collections.
                           new(params[:assignment_collection])
    respond_with(@assignment_collection)
  end

  private
    def grab_lecture_instance_from_lecture_instance_id
      @lecture_instance = LectureInstance.
             find(params[:lecture_instance_id])
    end

    def assignment_collections
      @lecture_instance.assignment_collections
    end

    def authorize_nested_access
      authorize! :manage_assignment_collections_of,
           @lecture_instance
    end
end
```

Listing 45: Ausschnitt aus dem `AssignmentCollectionsController`

3.5.2 Hinzufügen von Aufgaben

Um Aufgaben zu einer Aufgabensammlung hinzuzufügen, gibt es zwei unterschiedliche Möglichkeiten. Zum einen kann eine Aufgabe neu erstellt, zum anderen kann eine Aufgabe durch Kopieren einer vorhandenen Aufgabe erzeugt werden. Aufgaben sind als verschachtelte Ressource einer Aufgabensammlung untergeordnet, außerdem erhalten sie eine fortlaufende Nummer. Die Umsetzung erfolgt analog zum in Abschnitt 3.5.1 beschriebenen Vorgehen. Abweichende Besonderheiten und Zusätze werden im Folgenden erläutert.

Neuerstellung einer Aufgabe

Zur Neuerstellung einer Aufgabe navigiert der Endnutzer zur Übersichtsseite der Aufgabensammlung, der die Aufgabe untergeordnet werden soll. In der Navigationsleiste befindet sich ein Menüpunkt zum Hinzufügen einer Aufgabe. Im entsprechenden Formular muss eine Punktzahl und eine Aufgabenstellung eingegeben werden. Optional

können ein Titel, eine Menge von Schlagworten, eine Musterlösung, Notizen und ein Erwartungshorizont angegeben werden.

Die einzige Besonderheit stellt hier die Verarbeitung der Schlagworte dar. Aus Gründen der Datenbanknormalisierung wurden Schlagworte als eigene Entität ohne zugehörige Ressource modelliert. Der Nutzer gibt im Formular eine durch Kommata separierte Liste an, welche zu einer Menge von Beziehungen zu entsprechenden Schlagworten umgesetzt werden muss. Existiert ein angegebenes Schlagwort noch nicht, so muss es zunächst angelegt werden.

Die Verarbeitung erfolgt durch ein **virtuelles Attribut** im Model der Aufgabe. Ein virtuelles Attribut zeichnet sich dadurch aus, dass keine korrespondierende Spalte in der Datenbanktabelle existiert. Dem Formular wird für die Schlagwortliste ein Feld namens `tag_list` hinzugefügt. Dieses wird – wie alle anderen Formularfelder – als Anfrage-Parameter an den Controller übergeben. Im Laufe der Verarbeitung wird für jeden übergebenen Anfrage-Parameter dynamisch die entsprechende Setter-Methode des Models aufgerufen. Durch geeignetes Überschreiben dieser Methode lässt sich die korrekte Verarbeitung der Schlagwortliste gewährleisten ohne Änderungen im Controller durchführen zu müssen. Listing 46 zeigt die Implementation des virtuellen Attributes.

```
1  class Assignment < ActiveRecord::Base
2    def tag_list=(new_tag_list)
3      transaction do
4        new_tag_list = new_tag_list.split(",")
5                       .map {|t| t.downcase.strip}
6        old_tag_list = self.tags.map(&:name)
7
8        to_add = new_tag_list - old_tag_list # A ohne B
9        to_add.each do |tag|
10         tags << Tag.find_or_create(tag)
11       end
12
13       to_remove = old_tag_list - new_tag_list # B ohne A
14       to_remove.each do |tag|
15         tags.delete Tag.find_by_name(tag)
16       end
17     end
18   end
19 end
```

Listing 46: Ausschnitt zur Schlagwortverarbeitung aus dem Aufgaben-Model

Die Methode muss allerdings nicht nur bei der Erstellung der Aufgabe, sondern auch bei Aktualisierungen funktionieren. Das Vorgehen lässt sich gut mit Hilfe von Mengen veranschaulichen. Sei A die Menge der neuen und sei B die Menge der alten Schlagworte. Für die Menge $A \cap B$ ist keine Aktion erforderlich, die zugehörigen Schlagworte waren vor der Aktualisierung vorhanden und müssen es danach immer noch sein.

Die Menge $A \backslash B$ beinhaltet die neu hinzuzufügenden Schlagworte. Falls diese existieren, muss lediglich die Beziehung zur Aufgabe erzeugt werden, im anderen Fall muss das Schlagwort vorher angelegt werden. Dies geschieht in den Zeilen 8-11 mittels der in `Tag` implementierten Methode `find_or_create`.

In der Menge $B \backslash A$ liegen jene Schlagworte, die vor dem Update der Aufgabe zugeordnet waren, es danach aber nicht mehr sein sollen. Die Beziehung zur Aufgabe muss also entfernt werden (siehe Zeilen 13-16).

Im Spezialfall der Neuerstellung einer Aufgabe gilt $B = \emptyset$ und damit auch $B \backslash A = \emptyset$ und $A \backslash B = A$, es werden also nur die neuen Schlagworte hinzugefügt und ansonsten keine weiteren Aktionen durchgeführt.

An dieser Stelle ist es notwendig, den Code in eine Transaktion einzubetten. Im Fall einer Aktualisierung oder Neuerstellung wäre dies zwar nicht zwingend erforderlich (siehe 3.5.1), stellt aber sicher, dass die Methode auch in anderen Kontexten korrekt funktioniert.

Dateianhänge

Dateianhänge sind ihrer Aufgabe als verschachtelte Ressource untergeordnet. Anhänge können erst nach Erstellung der Aufgabe hinzugefügt werden. Der Anwender navigiert über die Ansichtsseite der Aufgabe zu einem Formular, um einen Dateianhang anzulegen. Im Formular muss eine Datei und die Zugehörigkeit zur Aufgabenstellung oder Musterlösung angegeben werden.

In der Datenbank werden neben dem Dateinamen und der Zugehörigkeit lediglich einige Metadaten gespeichert, wie zum Beispiel der MIME-Type oder die Größe der Datei. Zur vereinfachten Verwaltung wird die freie Bibliothek **Paperclip**[15] verwendet. Paperclip lagert die Logik zur Dateiverwaltung mit einem dem virtuellen Attribut ähnlichen Mechanismus in die Modelklasse aus, damit Dateianhänge im Controller als gewöhnliches Attribut betrachtet werden können.

Der eigentliche Anhang wird in einer Ordnerstruktur im Dateisystem abgelegt. Diese Ordnerstruktur konstruiert sich nach einem festzulegenden Schema, so dass das Model aus seinen Daten den Pfad zur Datei rekonstruieren kann und diesen nicht vollständig speichern muss. Das Schema lautet wie folgt:

```
:rails_root/app/assets/assignment_attachments/
        :rails_env/:assignment_id/:basename.:extension
```

Die mit einem Doppelpunkt beginnenden Ausdrücke werden für jeden Anhang hineininterpoliert. Der Ausdruck `:rails_root` bezeichnet den Basispfad der Applikation. Mittels `:rails_env` werden zusätzliche Ordner für die Entwicklungs-, Produktiv- und Testumgebung angelegt. Für jede Aufgabe wird durch die Interpolation der ID der Aufgabe

[15] `https://github.com/thoughtbot/paperclip`

(`:assignment_id`) ein eigener Ordner angelegt, in welchem schließlich die Dateien unter ihrem Namen mit entsprechender Dateiendung abgelegt werden. Wird eine Aufgabe gelöscht, so werden auch ihre Dateianhänge automatisch entfernt.

Kopieren von Aufgaben

Zur erneuten Verwendung alter Aufgaben müssen diese kopiert werden. Im Idealfall wird die Aufgabe dabei verbessert oder für das aktuelle Semester angepasst. Die Kopie wird daher im Folgenden als **Fork** bezeichnet, da sie eine mögliche Verzweigung der Weiterentwicklung repräsentiert.

Um einen Fork zu erstellen, muss der Anwender zunächst zur Übersicht der Aufgabensammlung navigieren, der die kopierte Aufgabe untergeordnet werden soll. In der Navigationsleiste befindet sich eine Schaltfläche zur Aktivierung des **Fork-Modus**. Ein Klick leitet auf die Action `create` einer **Singleton-Ressource** namens `AssignmentForkSession` um, der kein Model zugrunde liegt. Als Singleton-Ressourcen werden Ressourcen bezeichnet, die pro übergeordneter Entität nur einmal existieren können. Ein anderes Beispiel für eine Singleton-Ressource ist die Login-Session, welche pro Nutzer einmalig existent ist. Die Action `create` im `AssignmentForkSessionsController` hält serverseitig die Aktivierung des Fork-Modus fest. Dieses Verhalten wirkt zunächst wie ein Verstoß gegen die Prinzipien der ROA, da serverseitig Applikationsstatus gespeichert zu werden scheint. Tatsächlich handelt es sich aber um den Status einer Ressource und stellt somit keine Komplikation dar (siehe dazu auch 2.2).

Nach der Statusänderung kann der Anwender vorhandene Aufgaben durchblättern. Befindet er sich im Fork-Modus, wird im Menü der Aufgabenübersicht eine zusätzliche Schaltfläche eingeblendet. Ein Klick leitet schließlich zur Action `create` des `AssignmentForksController` weiter. Diesem Controller liegt ebenfalls kein Model zugrunde. Er erhält ohne Zutun des Anwenders über den Link die IDs der Aufgabensammlung und der zu kopierenden Aufgabe, prüft entsprechende Berechtigungen und erzeugt den Fork. Der Fork erhält Kopien der primitiven Attribute, der Schlagworte und aller Dateianhänge. Schließlich erfolgt eine Weiterleitung zur Ansichtsseite der neuen Aufgabe.

Auf Auszüge aus dem Code wird an dieser Stelle verzichtet, da zur vollständigen Darstellung eine beträchtliche Menge von Beispielen nötig wäre.

Umsortieren von Aufgaben

Im Verlauf der Erstellung einer Aufgabensammlung kann es notwendig werden, die Reihenfolge der Aufgaben anzupassen. Zur Umsortierung ist eine Änderung der fortlaufenden Nummer nötig. Die manuelle Anpassung der Zahlen wäre ziemlich unkomfortabel, da die Uniqueness-Constraints geeignet vom Nutzer beachtet werden müssten. Als Lösung wurde die Umsortierung mittels **Drag&Drop** implementiert. Abbildung 3.5 zeigt die Aufgabenliste eines Aufgabenblattes zum Thema *Sortieren*.

Das orangefarbene Icon in der Spalte *Aktion* der Tabelle ist die Schaltfläche zur Umsortierung. Die Darstellung soll an einen Griff erinnern, der *„angefasst“* werden kann.

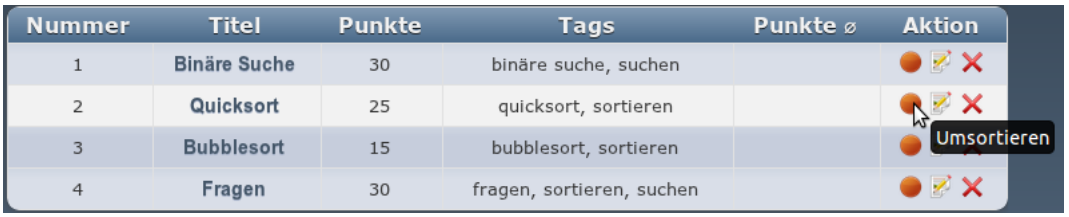


Abb. 3.5: Vor dem Umsortieren

Der Anwender muss zunächst den Mauszeiger auf die Schaltfläche bewegen und dann die Maustaste gedrückt halten (Drag). Die gewählte Tabellenzeile kann nun, wie in Abbildung 3.6 dargestellt, verschoben werden.

Abb. 3.6: Während des Umsortierens

Um die Umsortierung anzustoßen, muss die Tabellenzeile auf einer anderen Zeile durch Loslassen der Maustaste wieder abgelegt werden (Drop). Die Aufgabe der bewegten Zeile wird hinter der Aufgabe einsortiert, auf deren Tabellenzeile sie abgelegt worden ist. Das Resultat ist in Abbildung 3.7 gezeigt.

Abb. 3.7: Nach dem Umsortieren

Zur serverseitigen Umsetzung wird der Ressource `AssignmentCollection` eine verschachtelte Singleton-Ressource untergeordnet, welche lediglich die Schnittstelle `PUT /assignment_collections/:id/sort` unterstützt.

Die Ressource kann also vom Benutzer weder angefordert noch erzeugt oder gelöscht werden, eine Aktualisierung ist die einzig mögliche Interaktion. Zur Verarbeitung ist ein Anfrage-Parameter nötig, welcher ein Array mit den IDs der Aufgaben enthält, zum Beispiel `[42, 23, 73]`. Die Aufgaben erhalten als fortlaufende Nummer den Index ihrer Position im Array, erhöht um eins. Aufgabe `42` erhielte also die Nummer `1`, Aufgabe `23` die Nummer `2` und Aufgabe `73` die Nummer `3`.

Die eigentliche Drag&Drop-Funktionalität wird auf der Seite des Clients mit CoffeeScript und jQuery umgesetzt, Listing 47 zeigt den vollständigen Code.

Die Tabelle mit den Aufgaben erhält in der View die ID `#assignment_show_table`, damit sie von jQuery selektiert und ihr `tbody` als `sortable` markiert werden kann. Die

```
jQuery ->
  $("#assignment_show_table tbody").sortable(
    axis: "y"
    handle: '.handle'
    update: ->
      ary = $(this).sortable('toArray')
      $.put(
        $(this).data('update_url')
        $(this).sortable('serialize')
        (data, textStatus, jqXHR) ->
          for value, index in ary
            $("#seq_num_#{value}").text(index + 1)
      )
  )
```

Listing 47: CoffeeScript zur Umsetzung der Drag&Drop-Funktionalität im Client

Festlegung auf den `tbody` ist notwendig, damit der Tabellenkopf nicht mit Drag&Drop-Funktionalität versehen wird. Damit sind sämtliche Elemente innerhalb der Tabelle sortierbar. Mit dem Parameter `axis` wird festgelegt, in welche Richtung die Elemente bewegt werden können. Durch Übergabe einer Klasse als `handle` lässt sich das *„anfassbare“* Element in der Tabellenzeile angeben, im Beispiel das orangefarbene Icon. Schließlich wird im Parameter `update` eine anonyme Funktion übergeben, die beim Ablegen einer Zeile aufgerufen wird. Die anonyme Funktion erzeugt zunächst das Array mit den IDs der Aufgaben und weist es der Variablen `ary` zu. Damit jQuery dies automatisch tun kann, müssen die Tabellenzeilen IDs der Form `assignment_:id` erhalten. Als nächstes wird die PUT-Anfrage abgesetzt, welche wiederum drei Parameter erwartet. Der erste Parameter ist die Ziel-URI, die hier aus einem Datenattribut der Tabelle entnommen wird. Die Hilfsmethode `serialize` von `sortable` erzeugt das Array mit den IDs und wandelt es zur Übergabe zu einem Anfrage-Parameter um. Schließlich wird eine anonyme Funktion übergeben, die bei erfolgreicher Anfrage ausgeführt wird. Diese verwendet das erstellte Array und passt die Tabellenspalte *Nummer* entsprechend der neuen Sortierung an, um dem Nutzer ein Feedback bezüglich seiner Aktion zu liefern.

3.5.3 PDF Darstellung

Als Format zur Definition von Aufgaben wurde vom Endanwender Latex gewählt, da es zum einen sehr große Flexibilität bietet und zum anderen sehr einfach in qualitativ hochwertige PDFs umwandelbar ist. Das Grundgerüst für eine Aufgabensammlung wird in Form eines ERB-Templates definiert. Die Verwendung von HAML ist in diesem Zusammenhang wenig sinnvoll, da kein strukturiertes Markup erzeugt werden soll, sondern lediglich einige Daten in einen Latex-Quelltext eingebettet werden müssen (siehe dazu auch Abschnitt 2.3.2). Listing 48 zeigt das Grundgerüst eines Aufgabenblattes aus der Veranstaltungsausprägung *Algorithmen und Datenstrukturen WS 11/12*. Der Autor hat maximale Gestaltungsfreiheit bezüglich des Dokuments, es gibt keine Vorgaben.

```
\documentclass[11pt]{article}
\begin{document}
\noindent
Institut für Informatik
  \hfill Universität Osnabrück, <%= start_date %>\\
Prof.\ Dr.\ Oliver Vornberger
  \hfill {\tt http://www-lehre.inf.uos.de/\symbol{126}ainf}\\
Nicolas Neubauer, M.Sc.
  \hfill Testat bis <%= due_date %>\\
Sebastian Büscher, M.Sc.

\begin{center}
  {\Large \bf Übungen zu Algorithmen} \\[3mm]
  {\it Wintersemester 2011/2012} \\[3mm]
  {\bf <%= assignment_collection_title %>}
\end{center}

<% assignments.each do |assignment| %>
  {\bf <%= assignment_display_title(assignment) %>\\[3mm]}
  <%= assignment.to_tex(solution: rendering_solution?) %>
<% end %>
\end{document}
```

Listing 48: Grundgerüst eines Aufgabenblattes aus der AG Medieninformatik. Die Präambel ist der Übersicht halber gekürzt dargestellt.

Zur Unterstützung sind innerhalb des Templates verschiedene Hilfsmethoden verfügbar. Die Methoden `due_date` und `start_date` liefern zum Beispiel entsprechende Attribute der Aufgabensammlung in geeigneter Formatierung zurück. Weitere Hilfsmethoden existieren zur Darstellung des Titels und zum Einfügen der Aufgaben. Das Template wird im Kontext eines Objektes der Klasse `AssignmentCollectionLatexTransformer` ausgewertet, welche die Hilfsmethoden bereitstellt. Ein repräsentativer Ausschnitt aus der Klasse ist in Listing 49 dargestellt.

Die Superklasse `BasicAssignmentCollectionTransformer` stellt einen Konstruktor bereit, mit dessen Hilfe eine Aufgabensammlung und mögliche Optionen übergeben werden können. Für die Aufgabensammlung ist die Getter-Methode `assignment_collection` definiert. Die Methode `rendering_solution?` stammt ebenfalls aus der Superklasse und zeigt an, ob die Aufgabenstellung oder die Musterlösung erzeugt werden soll. Außerdem stellt die Superklasse die Methode `german_date_time` bereit, die ein übergebenes Datum in ein deutsches Datumsformat umwandelt.

Soll eine Methode im Template verfügbar sein, so wird diese als Instanzmethode definiert, wie `assignments` oder `due_date` im Beispiel. Die Auswertung des Templates geschieht in der Methode `transform`. Zunächst wird aus dem Template ein ERB-Objekt erzeugt, anschließend erfolgt die Auswertung über den Aufruf der Methode `result` am ERB-Objekt. Dieser erhält als Parameter das aktuelle **Binding**, also den Sichtbarkeits-

```ruby
class AssignmentCollectionLatexTransformer <
              BasicAssignmentCollectionTransformer
  def assignments
    assignment_collection.assignments.in_order
  end

  def due_date
    german_date_time(assignment_collection.due_date)
  end

  def transform
    template = ERB.new(assignment_collection.tex_template)
    template.result(get_binding)
  end
end
```

Listing 49: Ausschnitt aus Klasse `AssignmentCollectionLatexTransformer`

bereich, der dem Template zur Verfügung stehen soll (für weitere Ausführungen siehe [Perr 10, Thom 09]). Im Beispiel ist dies der Instanzkontext des Transformers. Damit ist gleichzeitig auch sichergestellt, dass der Anwender nicht über das Template mit dem Rest der Applikation interagieren kann. Für die einzelnen Aufgaben existiert eine ähnliche Transformer-Klasse, welche sich hauptsächlich durch die bereitgestellten Methoden unterscheidet. Im Beispieltemplate der Aufgabensammlung (Listing 48) wird über alle Aufgaben iteriert und mittels der Methode `to_tex`, welche den entsprechenden Transformer verwendet, der Latex-Code aller Aufgaben an geeigneter Stelle eingefügt.

Die Methode `transform` liefert nun also syntaktisch korrekten Latex-Code zur Darstellung der Aufgabensammlung zurück. Die Klasse **Renderer** aus dem Modul **LaTeX** (`LaTeX::Renderer`) übernimmt die Überführung in das PDF-Format. Sie erhält im Konstruktor den zu rendernden Latex-String, erzeugt einen temporären Ordner für die Verarbeitung, schreibt den String innerhalb dieses Ordners in eine Datei und startet einen `pdflatex`-Prozess zum Rendern. Bei Erfolg wird der Pfad zum erzeugten PDF zurückgeliefert, bei Misserfolg die Fehlermeldung und der Name des Logfiles. Auf Implementierungsdetails soll an dieser Stelle nicht weiter eingangen werden.

Um ein PDF zu erzeugen, muss der Anwender in der Applikation zur Übersichtsseite der Aufgabensammlung navigieren. In der Menüleiste gibt es für Aufgabenstellung und Musterlösung jeweils eine Schaltfläche. Durch einen Klick wird der Nutzer zur Action `show` des `AssignmentCollectionsController` weitergeleitet. Dort ist als mögliches Antwortformat zusätzlich zu HTML noch PDF aktiviert, so dass der Controller entsprechend das PDF erzeugt und an den Nutzer sendet. Im Fehlerfall wird dem Anwender die von `pdflatex` ausgegebene Fehlermeldung dargestellt. Abbildung 3.8 zeigt einen Ausschnitt einer PDF-Datei, die aus dem im Umsortierungsbeispiel verwendeten Aufgabenblatt (siehe 3.5.2) mit dem in Listing 48 gezeigten Template erzeugt wurde.

Beim Rendern der Musterlösung würde diese automatisch unterhalb der jeweiligen Aufgabenstellung eingefügt werden.

Institut für Informatik
Prof. Dr. Oliver Vornberger
Nicolas Neubauer, M.Sc.
Sebastian Büscher, M.Sc.

Universität Osnabrück, 23.11.2011
http://www-lehre.inf.uos.de/~ainf
Testat bis 30.11.2011, 10:51 Uhr

Übungen zu Algorithmen

Wintersemester 2011/2012

Blatt 3: Sortieren

Aufgabe 3.1: Binäre Suche (30 Punkte)

Schreiben Sie ein Programm `BinSearch.java`, welches die Binäre Suche in der Methode `public static int binSearch(int[] data)` implementiert. Die Methode soll den Index des Elements zürückgeben, wenn es gefunden wurde, ansonsten -1.

Schreiben Sie zusätzlich eine geeignete `main`-Methode zum Testen des Programms. Diese sollte vom Benutzer eine Menge von Zahlen und die zu suchene Zahl einlesen, die Suche durchführen und das Ergebnis geeignet ausgeben.

Aufgabe 3.2: Bubblesort (15 Punkte)

Abb. 3.8: Ausschnitt aus einem gerenderten PDF

3.6 Suchsystem

Zur verbesserten Durchsicht von Aufgaben aus der Vergangenheit sollten diese einfach durchsuchbar sein. Dabei werden vorrangig die vergebenen Schlagworte berücksichtigt, aber auch der Titel der Aufgabe und der Aufgabentext müssen in die Wertung einfließen.

Die einfachste Möglichkeit zum Durchsuchen der Daten wäre eine SQL-Anfrage der Art:

```
SELECT * FROM tabelle WHERE attribut LIKE '%suchbegriff%'
```

In der Applikation werden allerdings InnoDB-Tabellen verwendet (siehe Abschnitt 3.3), die keine Möglichkeit zur Textindizierung bieten. Anfragen mit LIKE wären also bei zunehmender Datenmenge nicht besonders effizient, da sie keinen Index zur optimierten Suche verwenden können. Eine mögliche Lösung des Problems wäre die Verwaltung zusätzlicher MyISAM-Tabellen. Damit wäre allerdings erheblicher Mehraufwand nötig, da die Daten stets synchron gehalten werden müssten. Außerdem müsste die Gewichtung der Attribute manuell in den Queries geschehen, was voraussetzt, dass jede mögliche Suchanfrage im Vorfeld formuliert sein muss.

Zur Lösung dieser Probleme werden das auf Volltextsuche spezialisierte Werkzeug **Apache Solr**[16] und die Bibliothek **Sunspot**[17] zur Rails-Integration von Solr verwendet. Die Installation von Sunspot liefert automatisch einen vorkonfigurierten Solr-Server mit, es ist aber durch Anpassung einer Konfigurationsdatei leicht möglich, einen beliebigen Server einzusetzen oder den mitgelieferten Server nach Wunsch zu konfigurieren.

[16] `http://lucene.apache.org/solr/`
[17] `http://sunspot.github.com/`

Mit Sunspot lässt sich direkt im Model festlegen, welche Attribute auf welche Weise zu indizieren sind. Listing 50 zeigt einen repräsentativen Ausschnitt aus der Indizierungsdefinition der Aufgaben.

```
1  class Assignment < ActiveRecord::Base
2    searchable do
3      text :text_tag_list, boost: 3 do
4        tags.map {|t| t.name}
5      end
6      text :title, boost: 2
7      text :body, :solution, boost: 1
8      integer :assignment_id, using: :id
9    end
10 end
```

Listing 50: Ausschnitt aus der Indizierungsdefinition des Models `Assignment`

Der Methodenaufruf `searchable` fügt die Suchfunktionalität zum Model hinzu. Eine Instanz wird dadurch nach der Neuerstellung und Aktualisierung automatisch neu indiziert. Innerhalb des übergebenen Blockes wird die Art der Indizierung spezifiziert. Dazu gibt es zwei wesentliche Möglichkeiten: Zur Indizierung der Schlagworte wird der Feldname `text_tag_list` festgelegt und in einem übergebenen Block der zu indizierende Wert erzeugt. Bei Attributen des Models, wie `title` und `body` im Beispiel, ist dies nicht notwendig, da sie direkt indiziert werden können. Mit dem Parameter `boost` wird jeweils die Gewichtung für die spätere Suche festgelegt. Schließlich muss zur Einhaltung der Sichtbarkeitseinschränkungen die ID der Aufgabe indiziert werden.

Zur Durchführung der Suche ist in der Klasse `Assignment` eine Klassenmethode definiert, Listing 51 zeigt die Implementierung.

```
1  class Assignment < ActiveRecord::Base
2    def self.scoped_search(query_text, user)
3      visible_assignment_ids = Assignment.visible_for(user).
4                                          map(&:id)
5      unless visible_assignment_ids.empty?
6        Assignment.search do
7          keywords query_text
8          with(:assignment_id).any_of(visible_assignment_ids)
9        end.results
10     else
11       []
12     end
13   end
14 end
```

Listing 51: Ausschnitt aus der Methode zur Durchführung der Suche von Aufgaben

Die Schwierigkeit ist hier die Einhaltung der in Abschnitt 3.4.3 erläuterten Sichtbarkeitseinschränkungen. Die Methode bekommt neben der Suchanfrage den Nutzer übergeben, der sie gestellt hat. Nur Ergebnisse, die für diesen Nutzer sichtbar sind, dürfen angezeigt werden. Zunächst werden die IDs aller für den Nutzer sichtbaren Aufgaben ermittelt. Sollten keine entsprechenden Aufgaben existieren, so wird ein leeres Array zurückgeliefert, andernfalls wird die Suche durchgeführt. Als Randbedingung wird festgelegt, dass der indizierte Wert `assignment_id` mit mindestens einer der übergebenen IDs übereinstimmt. Damit ist die Sichtbarkeit hinreichend berücksichtigt.

Der Nutzer interagiert mit dem Suchsystem über die Collection-Ressource der Aufgaben. Wird kein Suchbegriff übergeben, so werden alle sichtbaren Aufgaben angezeigt. Durch Angabe eines Suchbegriffes als Anfrage-Parameter wird die Menge der Aufgaben, wie oben erläutert, eingeschränkt. Die Eingabe der Anfrage geschieht über ein in Abbildung 3.9 dargestelltes Formular oberhalb der Aufgabenliste.

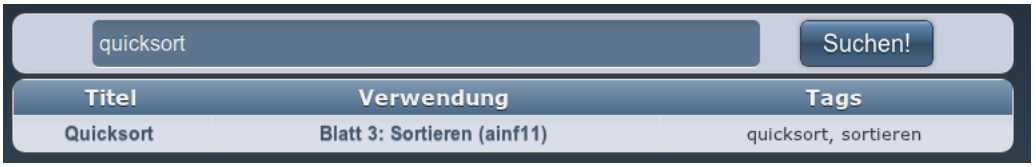


Abb. 3.9: Screenshot der Sucheingabe

Die Suche nach Studenten ist analog umgesetzt.

3.7 Datenimport und Auswertung

Zur Auswertung von Studentenleistungen und dem Erfolg der Aufgabenbearbeitung müssen zunächst die erzielten Punkte importiert werden. Diese werden in der AG Medieninformatik mit einem externen Tool namens **OTTER** erfasst, welches die Daten im in Listing 52 gezeigten CSV-Format exportieren kann. Andere Arbeitsgruppen müssten zum automatisierten Import ihre Daten gegebenenfalls zunächst in dieses Format überführen.

```
Nachname;Vorname;Mail;MatNr;aktueller Tutor;MatNrTutor;Studiengaenge;m-n;
```

Listing 52: Exportformat

Die ersten sechs Werte enthalten die relevanten Studentendaten. Der Wert `m-n` beinhaltet das Ergebnis des Studenten bei der Bearbeitung von Aufgabe `n` aus Aufgabensammlung `m`. Je nach Anzahl der Aufgabensammlungen und Aufgaben kann die Menge der Daten also variieren. Die Verarbeitung erfolgt durch die Klasse `OtterImporter`. Aufgrund der umfangreichen Implementierung wird auf Auszüge aus dem Code verzichtet und nur das generelle Vorgehen beschrieben.

Der `OtterImporter` erhält im Konstruktor die CSV-Datei, die Veranstaltungsausprägung, zu der die Daten gehören, und einen Parameter zur Festlegung, ob Klausur- oder

Übungsdaten importiert werden sollen. Zunächst wird geprüft, ob im System überhaupt alle als `m-n` angegebenen Aufgaben und Aufgabensammlungen existieren. Sollte bei der Prüfung ein Fehler auftreten, so wird die Verarbeitung abgebrochen.

Nun wird Zeile für Zeile jeder Datensatz innerhalb einer Transaktion verarbeitet. Sollte der entsprechende Student noch nicht existieren, so wird er mit den übergebenen Daten erzeugt. Ist einer seiner Studiengänge ebenfalls nicht existent, so wird auch dieser erzeugt. Schließlich wird der Student als Teilnehmer zur Veranstaltungsausprägung hinzugefügt und die entsprechenden Punkte werden als `AssignmentResults` eingetragen.

Damit ist gleichzeitig auch eine Möglichkeit gegeben, die Teilnehmer einer Veranstaltung automatisiert hinzuzufügen. Im Falle des Imports von Klausurdaten existieren die Teilnehmer in der Regel bereits. Ist ein Student bereits im System existent, so genügt die Angabe der Matrikelnummer in der CSV-Datei, auf die Angabe der restlichen Daten kann verzichtet werden.

Zum Importieren navigiert der Nutzer zur Übersichtsseite der Veranstaltungsausprägung. Im Menu steht die Schaltfläche *Datenimport* zur Verfügung. Diese leitet zu einem Formular weiter, in dem die Auswahl zwischen Klausur- oder Übungsdaten getroffen und die CSV-Datei hochgeladen werden müssen. Die Import-Ressource ist der Veranstaltungsausprägung als verschachtelte Singleton-Ressource untergeordnet und der entsprechende Controller heißt `DataImportsController`. Er besitzt die Action `new` zur Anzeige des Formulars und die Action `create`, welche die Daten mittels einer Instanz des `OtterImporters` verarbeitet.

Sind die Punktedaten erst einmal im System, lassen sich zahlreiche Informationen daraus ableiten. In Abbildung 3.10 ist ein Ausschnitt aus der Übersicht einer Aufgabe dargestellt.

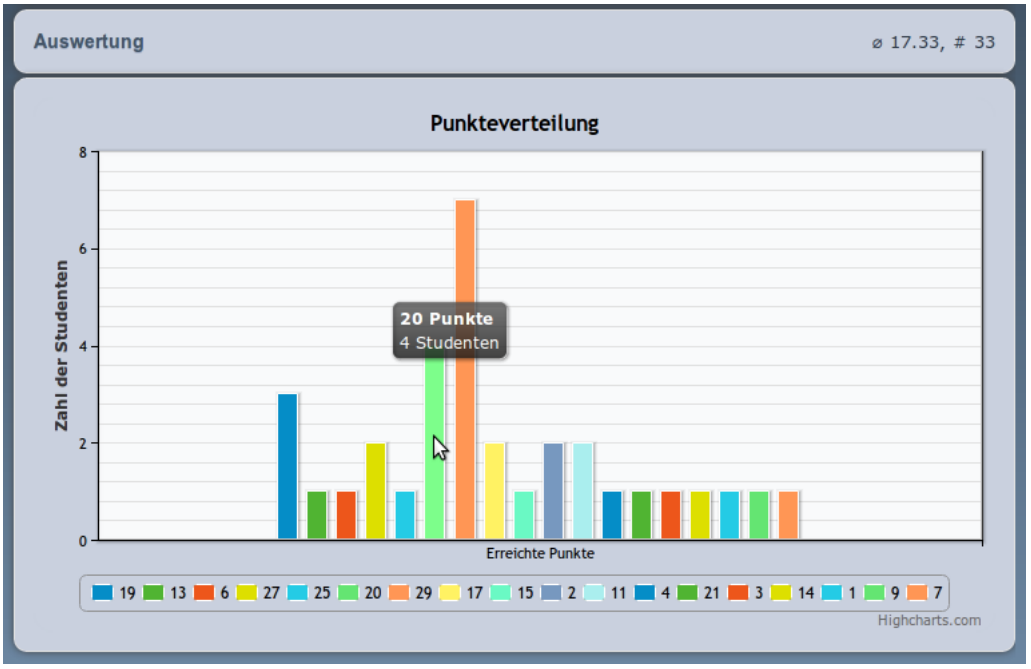
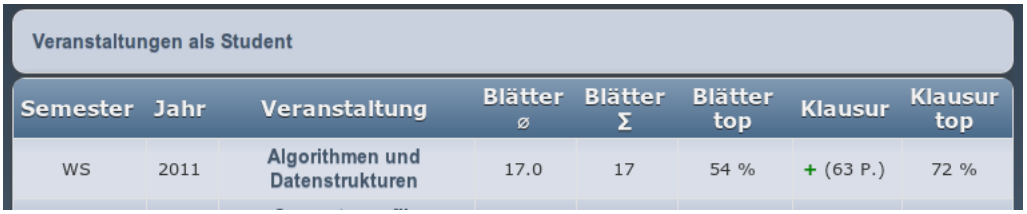


Abb. 3.10: Ausschnitt aus der Übersichtsseite einer Aufgabe

Um nach abgeschlossener Bearbeitung einen Eindruck bezüglich des Lernerfolges zu erhalten, ist zunächst die erreichte Durchschnittspunktzahl interessant. Diese wird in der oberen rechten Ecke dargestellt. Direkt daneben findet sich die Anzahl der Studenten, welche die Aufgabe bearbeitet haben. Schließlich ist noch die Verteilung der Punkte als Histogramm dargestellt. Jeder Balken repräsentiert eine erreichte Punktzahl. Die Höhe des Balkens gibt an, wie viele Studenten diese Punktzahl erreicht haben. Durch Bewegen der Maus über einen der Balken werden die Daten gebündelt dargestellt. Damit lässt sich zum Beispiel bei schlechter durchschnittlicher Punktzahl einer Aufgabe ein Eindruck darüber gewinnen, ob die Aufgabe generell schlecht gelöst wurde, oder ob einige Studenten auch gut mit der Bearbeitung zurecht gekommen sind.

Die Daten werden in der View in eine Tabelle eingetragen, auf Clientseite mittels der JavaScript Bibliothek **Highcharts**[18] extrahiert und in die gezeigte Grafik umgesetzt. Sollte in einem Client-Browser JavaScript nicht aktiviert sein, so wird statt der Grafik die Tabelle dargestellt.

Für einen Dozenten ist häufig die Leistung eines Studenten im Vergleich mit anderen interessant, zum Beispiel, wenn ein Gutachten für eine Bewerbung auf ein Stipendium geschrieben werden soll. In der Übersichtsseite eines Studenten werden daher alle für den aktuell authentifizierten Nutzer sichtbaren Veranstaltungen mit den erbrachten Leistungen angezeigt. Abbildung 3.11 zeigt einen Screenshot.



Veranstaltungen als Student

Semester	Jahr	Veranstaltung	Blätter ⌀	Blätter Σ	Blätter top	Klausur	Klausur top
WS	2011	Algorithmen und Datenstrukturen	17.0	17	54 %	+ (63 P.)	72 %

Abb. 3.11: Ausschnitt aus der Übersichtsseite eines Studenten

In der Spalte *Klausur* findet sich nebst erreichter Punktzahl ein Symbol zur Indikation, ob der Student bestanden hat. Die Spalte *Klausur top* gibt an, wieviel Prozent der anderen Teilnehmer besser gewesen sind als der betrachtete Student. Im Beispiel kann festgestellt werden, dass sich der Student unter den besten 72 % der Teilnehmer befindet.

Die Spalte *Blätter* $\emptyset$ zeigt die durchschnittlich erreichte Punktzahl aller Blätter an, in der Spalte *Blätter* $\sum$ ist die Summe aller erreichten Punkte dargestellt. Schließlich zeigt die Spalte *Blätter top* ähnlich wie bei der Klausur an, wie gut der Student im Vergleich zu seinen Kommilitonen beim Bearbeiten der Übungsaufgaben abgeschnitten hat.

Die Implementierung vollständig darzustellen, würde den Rahmen dieses Kapitels sprengen, es sei aber gesagt, dass die Funktionalitäten von ActiveRecord bei der komplexen Aggregation von Daten schnell an ihre Grenzen stießen. Zur Bereitstellung der gewünschten Daten mussten abermals von Hand SQL-Anfragen formuliert werden, und in diesem Fall sogar direkt an die Datenbank gesendet werden, ohne Verwendung der Modelklasse. Listing 53 zeigt einen Ausschnitt aus der Implementierung, welcher der Berechnung der Anzahl besserer Kommilitonen eines Studenten dient.

[18] `http://www.highcharts.com`

```
query = "SELECT COUNT(*) as better FROM
          ( SELECT SUM(ar.points) as summed_points,
                   ar.user_id as user_id
              FROM assignment_results ar
              INNER JOIN assignments a
                ON a.id = ar.assignment_id
              INNER JOIN assignment_collections ac
                ON ac.id = a.assignment_collection_id
              WHERE ac.lecture_instance_id = #{lecture_instance.id}
                AND ac.collection_type = 'assignment_sheet'
              GROUP BY ar.user_id ) others
         WHERE user_id != #{self.id}
           AND summed_points > #{sum}"
```

Listing 53: Ausschnitt zur Berechnung der Anzahl besserer Kommilitonen

Die Auswertung der Daten erleichtert entsprechende Arbeitsabläufe erheblich und kann durch Unterstützung der Ursachenforschung bei fehlgeschlagenen Aufgaben möglicherweise zur Verbesserung der Aufgabenstellung im nächsten Durchlauf der Veranstaltung führen.

4 Reflexion

In diesem Kapitel werden zunächst die Ergebnisse der Arbeit zusammengefasst und deren Übertragbarkeit auf andere Kontexte diskutiert. Abschließend folgt ein Fazit und ein Ausblick auf weitere mögliche Arbeiten.

4.1 Zusammenfassung und Übertragbarkeit der Ergebnisse

Die entstandene Applikation kann die gestellten inhaltlichen Anforderungen in vollem Umfang erfüllen. Sämtliche von den Endanwendern gewünschten Funktionalitäten wurden umgesetzt und am Ende jeder Iterationen von den Nutzern verifiziert. Aufgabenblätter und Klausuren können komfortabel erstellt, archiviert, durchsucht und ausgewertet werden. Zusätzlich besteht für die Dozenten aufgrund der vorliegenden Daten die Möglichkeit, die Leistungen der Studenten effizient zu vergleichen, um beispielsweise Gutachten zu erstellen.

Die Funktionalitäten sind zwar an die konkreten Anforderungen in der Arbeitsgruppe Medieninformatik angepasst, dennoch lässt sich die Applikation ohne große Änderungen auch in anderen Arbeitsgruppen verwenden. Die Übertragbarkeit auf andere Fachbereiche ist nur in Teilen möglich. In der Informatik werden Vorlesungen stets von den gleichen Dozenten gehalten, in anderen Fachbereichen muss dies nicht der Fall sein. Wird eine Vorlesung von unterschiedlichen Dozenten gehalten, so muss gegebenenfalls die Einschränkung der Sichtbarkeiten angepasst werden. Dank der erweiterungsfreundlichen Umsetzung stellt dies aber kein großes Problem dar.

Das Ziel der Identifikation eines geeigneten Vorgehensmodells zur Entwicklung von qualitativ hochwertigen Webapplikationen konnte ebenfalls erreicht werden. Behaviour-Driven Development bietet zahlreiche Vorteile und kann auch von Einzelentwicklern erfolgreich durchgeführt werden. Die Umsetzung als Resource-Oriented Architecture mit Ruby on Rails nimmt dem Entwickler eine Vielzahl zu treffender Entscheidungen ab und führt zu einem konsistenten, für Dritte leicht verständlichen, Design. Die proklamierten positiven Effekte des Vorgehens können aufgrund während der Applikationsentwicklung gemachter Erfahrungen bestätigt werden. Die entstandenen Tests dokumentieren sowohl die Funktionalität der Applikation als auch die Verwendung der einzelnen Systemkomponenten sowie deren Interaktion. Außerdem ermöglichen sie effizientes Refactoring, da dabei sehr aggressiv vorgegangen werden kann, weil die Tests die Sicherheit bieten, dass das Gesamtsystem immer noch funktionstüchtig ist. Mittels des Acceptance Test-Driven Planning kann während der Entwicklung in übersichtlicher Art und Weise der Fortschritt gemessen werden. Gleichzeitig wird sichergestellt, dass alle Anforderungen hinreichend umgesetzt und keine unnötigen Funktionen implementiert werden. Der Mehraufwand für testgetriebenes Vorgehen kann wegen der überwiegend positiven Effekte billigend in Kauf genommen werden.

Einem neuen Entwickler steht eine als Test ausführbare Dokumentation zur Einarbeitung zur Verfügung und vereinfacht diese damit erheblich. Voraussetzung ist dabei natürlich, dass der Entwickler sich intensiv in die entsprechenden Thematiken und Werkzeuge eingearbeitet hat. Selbst wenn dies erst im Rahmen der Einarbeitung in das System geschieht, so können die erworbenen Kenntnisse in Teilen auch in anderen Kontexten nützlich sein.

Die agilen Prinzipien lassen sich auf eine sehr große Menge von Softwareprojekten übertragen. Behaviour-Driven Development ist ebenfalls in anderen Kontexten einsetzbar. Einschränkungen bestehen allerdings bei Softwareprojekten, bei denen sich die Anwenderinteraktion mit dem System nur schwierig automatisiert testen lässt. Auch in solchen Systemen kann aber zur Entwicklung der Einzelkomponenten Test-Driven Development eingesetzt und dessen positive Effekte genutzt werden. Die Werkzeuge RSpec und Cucumber sind keinesfalls nur an Ruby gebunden, sondern lassen sich auch in anderen Kontexten verwenden. Webapplikationen jeglicher Art können von einer Umsetzung als Resource-Oriented Architecture profitieren, unabhängig davon, mit welchem Framework sie realisiert werden.

4.2 Fazit und Ausblick

Im zeitlichen Rahmen dieser Arbeit konnten keine tiefgreifenden Vergleiche mit anderen Vorgehensmodellen, Architekturen und Werkzeugen durchgeführt werden. Empirische Daten zur Untermauerung der Effizienz agiler Vorgehensweisen existieren zwar, es wäre aber sicherlich interessant, mehrere konkrete Verfahren und Technologien zu vergleichen. Eine Applikation mit einer festgelegten Funktionalität könnte parallel mit unterschiedlichen Vorgehensmodellen, Werkzeugen und Architekturen umgesetzt und die Resultate verglichen werden. Aus den Ergebnissen ließen sich möglicherweise Schlüsse auf die tatsächliche Effizienz der Verfahrensweisen im Vergleich ziehen.

Auch im Bezug auf die entwickelte Applikation sind noch weitere Arbeiten möglich. Es könnte zum Beispiel sinnvoll sein, vor der Inbetriebnahme verschiedene Laufzeitumgebungen zu evaluieren und miteinander zu vergleichen. Als Erweiterung der Funktionalität wäre es vorstellbar, den Studenten begrenzten Zugriff auf die Applikation zu geben, so dass zum Beispiel Feedback in Form von Multiple-Choice-Fragebögen oder Freitexten zu den Aufgaben eingeholt werden kann, um besser festzuhalten, wie Schwierigkeit und Umfang der Aufgaben empfunden worden sind oder welche Probleme es bei der Bearbeitung gab. Interessant wäre sicherlich auch der Versuch, Mails von der veranstaltungsbegleitenden Mailingliste basierend auf deren Textinhalt automatisiert zu Übungsaufgaben zuzuordnen, um so bei erneuter Verwendung im Vorfeld einen Überblick über mögliche Unklarheiten zu erhalten und diese gegebenenfalls zu beseitigen. Natürlich könnte dies auch per Hand geschehen, das wäre allerdings deutlich umständlicher und würde vermutlich auch nicht immer von allen Aufgabenerstellern konsequent umgesetzt.

Zusammenfassend lässt sich sagen, dass es auch als Einzelentwickler möglich ist, durch korrekte Anwendung geeigneter Methoden qualitativ hochwertige Software zu entwickeln, welche den Anforderungen der Endnutzer gerecht wird und auch durch Dritte

leicht gewartet und erweitert werden kann. Die erarbeiteten Methoden und Werkzeuge lassen sich sogar in zahlreichen anderen Kontexten einsetzen. Die entwickelte Applikation kann die Arbeitsabläufe im Übungsbetrieb erheblich vereinfachen und stellt damit eine wertvolle Bereicherung der Lehre dar, da die Übungsleiter die gewonnene Zeit nun für die intensive Gestaltung und Reflexion der Aufgaben verwenden können.

Literaturverzeichnis

[Abra 05] P. Abrahamsson, A. Hanhineva, and J. Jäälinoja. "Improving business agility through technical solutions: A case study on test-driven development in mobile software development". *Business Agility and Information Technology Diffusion*, pp. 227–243, 2005.

[Alli 01] A. Alliance. "Manifesto of Agile Software Development". `http://agilemanifesto.org`, 2001.

[Aste 03] D. Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, 2003.

[Beck 01a] K. Beck. "Aim, Fire [test-first coding]". *Software, IEEE*, Vol. 18, No. 5, pp. 87–89, 2001.

[Beck 01b] K. Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesly, 2001.

[Beck 03] K. Beck. *Test-Driven Development By Example*. Addison-Wesley, 2003.

[Bhat 06] T. Bhat and N. Nagappan. "Evaluating the efficacy of test-driven development: industrial case studies". In: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pp. 356–363, ACM, 2006.

[Boeh 84] B. Boehm. "Software engineering economics". *Software Engineering, IEEE Transactions on*, No. 1, pp. 4–21, 1984.

[Canf 06] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. Visaggio. "Evaluating advantages of test driven development: a controlled experiment with professionals". In: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pp. 364–371, ACM, 2006.

[Chel 10] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North. *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. The Pragmatic Programmers, LLC., 2010.

[Damm 05] L. O. Damm, L. Lundberg, and D. Olsson. "Introducing Test Automation and Test-Driven Development: An Experience Report". *Electronic Notes in Theoretical Computer Science*, Vol. 116, No. SPEC.ISS., pp. 3–15, 2005.

[Edwa 04] S. Edwards. "Using software testing to move students from trial-and-error to reflection-in-action". *ACM SIGCSE Bulletin*, Vol. 36, No. 1, pp. 26–30, 2004.

[Erdo 05] H. Erdogmus *et al.* "On the effectiveness of test-first approach to programming". In: *Proceedings of the IEEE Transactions on Software Engineering*, pp. 226–237, 2005.

[Evan 04] E. Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[Fern 10] O. Fernandez. *The Rails 3 Way.* Addison-Wesley Professional, 2010.

[Fiel 00] R. Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, 2000.

[Fiel 99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. "RFC 2616: Hypertext transfer protocol–HTTP/1.1, June 1999". *Status: Standards Track*, 1999.

[Fowl 03] M. Fowler. *Patterns of enterprise application architecture.* Addison-Wesley Professional, 2003.

[Fowl 99] M. Fowler and K. Beck. *Refactoring: Improving the design of existing code.* Addison-Wesley Professional, 1999.

[Geor 04] B. George and L. Williams. "A structured experiment of test-driven development". *Information and Software Technology*, Vol. 46, No. 5, pp. 337–342, 2004.

[Gera 04] A. Geras, M. Smith, and J. Miller. "A prototype empirical evaluation of test driven development". In: *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pp. 405–416, IEEE, 2004.

[Janz 06] D. Janzen and H. Saiedian. "On the influence of test-driven development on software design". In: *Software Engineering Education and Training, 2006. Proceedings. 19th Conference on*, pp. 141–148, IEEE, 2006.

[Kauf 03] R. Kaufmann and D. Janzen. "Implications of test-driven development: a pilot study". In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 298–299, ACM, 2003.

[Larm 03] C. Larman and V. Basili. "Iterative and incremental developments. a brief history". *Computer*, Vol. 36, No. 6, pp. 47–56, 2003.

[Lui 04] K. Lui and K. Chan. "Test driven development and software process improvement in china". *Extreme Programming and Agile Processes in Software Engineering*, pp. 219–222, 2004.

[MacC 01] A. MacCormack. "How internet companies build software". *MIT Sloan Management Review*, Vol. 42, No. 2, pp. 75–84, 2001.

[Mart 03] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices.* Pearson Education Inc., 2003.

[Mart 09] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education, 2009.

[Maxi 03] E. Maximilien and L. Williams. "Assessing test-driven development at IBM". In: *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 564–569, IEEE, 2003.

[Mull 02] M. Müller and O. Hagner. "Experiment about test-first programming". In: *Software, IEE Proceedings-*, pp. 131–136, IET, 2002.

[Mull 06] M. Müller. "The effect of test-driven development on program code". *Extreme Programming and Agile Processes in Software Engineering*, pp. 94–103, 2006.

[Opdy 90] W. F. Opdyke and R. E. Johnson. "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems". In: *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications*, ACM, September 1990.

[Opdy 92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Panc 03] M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar. *Towards empirical evaluation of test-driven development in a university environment.* Vol. 2, IEEE, 2003.

[Perr 10] P. Perrotta. *Metaprogramming Ruby.* Pragmatic Bookshelf, 2010.

[Powe 09] B. Poweski and D. Raphael. *Security on Rails.* Pragmatic Bookshelf, 2009.

[Pyte 10] C. Pytel and T. Saleh. *Rails Antipatterns: Best Practice Ruby on Rails Refactoring.* Addison-Wesley Professional, 2010.

[Rich 07] L. Richardson and S. Ruby. *RESTful web services.* O'Reilly Media, 2007.

[Rist 05] I. Ristic and E. Corporation. *Apache security.* O'Reilly Media, 2005.

[Royc 70] W. Royce. "Managing the development of large software systems". In: *proceedings of IEEE WESCON*, Los Angeles, 1970.

[Ruby 11] S. Ruby, D. Thomas, D. Hansson, *et al. Agile web development with rails.* Pragmatic Bookshelf, 2011.

[Schw 02] K. Schwaber and M. Beedle. *Agile software development with Scrum.* Prentice Hall, 2002.

[Sini 07] M. Siniaalto and P. Abrahamsson. "A comparative case study on the impact of test-driven development on program design and test coverage". In: *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pp. 275–284, IEEE, 2007.

[Stei 01] D. Steinberg. "The effect of unit tests on entry points, coupling and cohesion in an introductory Java programming course". In: *XP Universe*, Citeseer, 2001.

[Subr 06] V. Subramanian and A. Hunt. *Practices of an Agile Developer.* The Pragmatic Programmers, LLC., 2006.

[Thom 09] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby*. Pragmatic Bookshelf, 2009.

[Vlis 95] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. "Design patterns: Elements of reusable object-oriented software". *Reading: Addison-Wesley*, 1995.

Erklärung

Ich versichere, dass ich die eingereichte Bachelor-Arbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, den 15.12.2011

(Nils Haldenwang)