AUTOMATISIERTE BEWERTUNG
IN DER PROGRAMMIERAUSBILDUNG

Oliver J. Bott
Peter Fricke
Uta Priss
Michael Striewe
(Hrsg.)

ELAN e.V.
WAXMANN

DIGITALE MEDIEN
IN DER HOCHSCHULLEHRE
Eine Publikationsreihe des ELAN e.V.

herausgegeben vom

ELAN e.V.

Band 6

Der gemeinnützige Verein E-Learning Academic Network e.V. (ELAN e.V.) wirkt als Impulsgeber zur stetigen Qualitätsverbesserung der medienbasierten Lehre an niedersächsischen Hochschulen und befördert durch seine Unterstützungsmaßnahmen die Kooperation der Mitgliedshochschulen und weiterer Mitglieder im Bereich standortübergreifender und E-Learning gestützter Lehre.

Oliver J. Bott, Peter Fricke,
Uta Priss, Michael Striewe (Hrsg.)

Automatisierte Bewertung in der Programmierausbildung

Waxmann 2017
Münster • New York

Bibliografische Informationen der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Digitale Medien in der Hochschullehre, Band 6

ISSN 2199-7667
Print-ISBN 978-3-8309-3606-0

Steinfurter Straße 555, 48159 Münster

www.waxmann.com
info@waxmann.com

Umschlaggestaltung: Steffen Ottow, Clausthal-Zellerfeld
Titelbild: © goodluz – fotolia.com
Druck: Hubert & Co., Göttingen
Gedruckt auf alterungsbeständigem Papier, säurefrei gemäß ISO 9706

Printed in Germany

Inhalt

1 Einleitung **11**

Sven Strickroth und Niels Pinkwart
2 Automatisierte Bewertung in der Programmierausbildung – Eine Übersicht **17**
2.1 Einleitung 17
2.2 Übersicht über verschiedene Reviews 19
2.3 Zusammenfassung und Diskussion 32

I Didaktische Einsatzszenarien **39**

Michael Striewe und Robert Garmann
3 Automatisierte Bewertung in der objektorientierten Programmierausbildung am Beispiel von Java **41**
3.1 Einleitung 41
3.2 Aufgabenformen 42
3.3 Einsatzszenario 48
3.4 Erfahrungen 52
3.5 Fazit und Ausblick 55

Oliver J. Bott, Felix Heine und Carsten Kleiner
4 Automatisierte Bewertung in der Ausbildung relationaler Datenbankabfragesprachen am Beispiel SQL **57**
4.1 Einleitung 57
4.2 Aufgabenstellung und Bewertung 58
4.3 Unterstützbare Lernziele und Abgrenzung 60
4.4 Einsatzszenarien und Erfahrungen 62
4.5 Fazit und Ausblick 70

Tobias Thelen, Helmar Gust und Elmar Ludwig
5 Automatisierte Programmbewertung in der deduktiven/logischen Programmierung am Beispiel der Prolog-Ausbildung . 73
5.1 Einleitung 73
5.2 Einsatzszenario: Introduction to Artificial Intelligence and Logic Programming 78
5.3 Prolog im Übungsbetrieb 79
5.4 Prolog im Klausurbetrieb 84
5.5 Erfahrungen, Fazit und Ausblick 86

Marianus Ifland und Frank Puppe
6 Automatisierte Bewertung in der UML-Modellierung 89
6.1 Einleitung 89
6.2 Einsatzszenario „Automatische Bewertung von Klassendiagrammen“ 90
6.3 Das Einsatzszenario „Automatische Bewertung von Aktivitätsdiagrammen“ 98
6.4 Fazit und Ausblick 105

Britta Herres, Rainer Oechsle und David Schuster
7 Automatisierte Bewertung im Kontext der App-Entwicklung am Beispiel Android 109
7.1 Einleitung 109
7.2 Einsatzszenarien 110
7.3 Das Modul *Entwicklung mobiler Anwendungen* 114
7.4 Benutzung des ASB-Systems aus Studierendensicht 115
7.5 Benutzung des ASB-Systems aus Dozentensicht 118
7.6 Vorgaben für ASB-Aufgaben 119
7.7 Erfahrungen 121
7.8 Fazit und Ausblick 124

Uta Priss und Peter Riegler
8 Automatisierte Programmbewertung in der Mathematikausbildung 125
8.1 Einleitung 125
8.2 APOS-Theorie 127
8.3 Besonders geeignete Aufgabentypen 129
8.4 Die Erstellung von Tests 133
8.5 Einsatzszenarien und Erfahrungen 134
8.6 Fazit und Ausblick 137

II Systeme zur automatisierten Programmbewertung . . . 141

Michael Striewe
9 Der Grader JACK . . . 143
9.1 Einleitung . . . 143
9.2 Aufgabendesign . . . 144
9.3 Feedbackmöglichkeiten . . . 147
9.4 Technischer Hintergrund . . . 154
9.5 Einsatzerfahrung . . . 155
9.6 Zusammenfassung . . . 156

Joachim Breitner, Martin Hecker und Gregor Snelting
10 Der Grader Praktomat . . . 159
10.1 Geschichte und Überblick . . . 159
10.2 Abläufe und Besonderheiten . . . 160
10.3 Sicherheitskonzept . . . 167
10.4 Einsatz . . . 169
10.5 Erfahrungen . . . 169

Robert Garmann
11 Der Grader Graja . . . 173
11.1 Was ist Graja? . . . 173
11.2 Nutzerperspektive . . . 173
11.3 Technische Funktion . . . 175
11.4 Technische Architektur . . . 179
11.5 Beispielaufgabe und -einreichung . . . 180
11.6 Aufgabenstruktur . . . 181
11.7 Bisherige Einsätze . . . 186
11.8 Zusammenfassung . . . 188

Felix Heine und Carsten Kleiner
12 Der Grader aSQLg . . . 191
12.1 Einleitung und Motivation . . . 191
12.2 Verwandte Arbeiten . . . 192
12.3 Konzept zur automatisierten Bewertung von SQL-Aufgaben . . . 194
12.4 Beispiele . . . 197
12.5 Dozentenunterstützung und LMS-Einbettung . . . 199
12.6 Ausblick . . . 201
12.7 Zusammenfassung . . . 202

Oliver Müller und Sven Strickroth
13 Der Grader GATE 207
13.1 Einleitung 207
13.2 Funktionen des GATE-Systems 208
13.3 Architektur und eingesetzte Technologien 216
13.4 Nutzerstatistiken und Evaluationsergebnisse 219
13.5 Zusammenfassung und Ausblick 221

Helmar Gust
14 Der Grader VEA 225
14.1 Einleitung 225
14.2 Der Vips-Evaluations-Assistent VEA 227
14.3 Ein Beispiel 235
14.4 Zusammenfassung und Ausblick 239

Lukas Iffländer und Alexander Dallmann
15 Der Grader PABS 241
15.1 Einleitung 241
15.2 Technologie und Architektur 241
15.3 Aufgabenstruktur 246
15.4 Grading-Methoden und Feedbackmöglichkeiten 249
15.5 Bisheriger Einsatz 250
15.6 Verfügbarkeit und Ausblick 252
15.7 Abschluss 253

Britta Herres, Rainer Oechsle und David Schuster
16 Der Grader ASB 255
16.1 Einleitung 255
16.2 Anforderungen 257
16.3 Grundsätzliche Entwurfsentscheidungen 258
16.4 Architektur des ASB-Servers 260
16.5 Testen von Android-Apps 267
16.6 Einsatz des Systems 269
16.7 Verfügbarkeit des Systems 270

17 Steckbriefe und Featurematrix der Grader 273

III Interoperabilität von Gradern und Lernmanagement-systemen 279

Peter Fricke und Michael Striewe
18 Integration automatisierter Programmbewertung in Moodle . 281
18.1 Einleitung 281
18.2 Integration durch IMS-LTI 282
18.3 Integration durch Plugins 285

Frauke Sprengel und Oliver Rod
19 Integration automatisierter Programmbewertung in LON-CAPA 295
19.1 LON-CAPA 295
19.2 External Response 298
19.3 Beispiel: JFLAP für die theoretische Informatik 300
19.4 Anbindung von JFLAP an LON-CAPA 303
19.5 Fazit 308

Elmar Ludwig
20 Integration automatisierter Programmbewertung in Stud.IP . 311
20.1 Stud.IP 311
20.2 Das Vips-Modul in Stud.IP 312
20.3 Programmieraufgaben in Vips 314
20.4 Im- und Export 320

Thomas Richter
21 Integration automatisierter Programmbewertung in ILIAS . . 325
21.1 Einführung und Hintergrund 325
21.2 Historische Entwicklung 327
21.3 Die Benutzeroberfläche 328
21.4 Automatische Korrektur studentischer Lösungen 330
21.5 Softwarearchitektur 331
21.6 Aufbau von ViPLab-Aufgaben 332
21.7 Automatische und halbautomatische Korrektur von Aufgaben . . . 333
21.8 Erfahrungen und Evaluation von ViPLab 335
21.9 Ausblick: Elektronische Klausuren 336

Oliver Rod
22 Integration mithilfe der Middleware ProFormA-Server 339
22.1 Einleitung 339
22.2 System und Ablauf 341
22.3 Schnittstellen 342

22.4 Aufbau der Middleware . . . 345
22.5 Ablauf . . . 346
22.6 Eigene Erfahrungen und Ausblick . . . 348

Peter Fricke
23 Integration mithilfe der Middleware Grappa . . . 351
23.1 Motivation und Anforderungen . . . 351
23.2 Fachdatenmodell . . . 353
23.3 Komponenten . . . 359
23.4 Öffentliche Schnittstelle . . . 360
23.5 Grader-Schnittstelle – BackendPlugin . . . 363
23.6 Ablauf . . . 364
23.7 Zusammenfassung und Ausblick . . . 366

Sven Strickroth, Oliver Müller und Uta Priss
24 Ein XML-Austauschformat für Programmieraufgaben . . . 369
24.1 Einleitung . . . 369
24.2 ProFormA: Ein XML-Austauschformat für Programmieraufgaben 370
24.3 Erfahrungen mit dem Austauschformat und Diskussion . . . 381
24.4 Zusammenfassung und Ausblick . . . 387

IV Abschluss und Ausblick . . . 391

Nils Jensen
25 Automatisierte Bewertung in der Programmierausbildung – Ausblick . . . 393
25.1 Zielsetzung der hochschulspezifischen Programmierausbildung . . 393
25.2 Didaktik . . . 394
25.3 Operationalisierung und Rollen . . . 395
25.4 Tools . . . 396
25.5 Community . . . 397
25.6 Aufgabenpools und Qualitätssicherung . . . 397
25.7 E-Assessments und Feedback . . . 399
25.8 Plagiarismus . . . 402
25.9 Fazit . . . 404

V Verzeichnis der Autorinnen und Autoren . . . 407

1 Einleitung

Die Programmierausbildung als Bestandteil der Lehre in der Informatik und in verwandten Fächern lebt ganz wesentlich auch von praktischen Übungen anhand von Programmieraufgaben. Es ist allgemein anerkannt, dass solche Aufgaben ein wichtiger Baustein sind, um aus reinem Faktenwissen über eine Programmiersprache operationales Wissen zu machen und somit höhere Stufen in den gängigen Lernziel- oder Kompetenztaxonomien zu erreichen. Die manuelle Korrektur der von Studierenden abgegebenen Lösungen zu Programmieraufgaben ist allerdings aufwändig und erfolgt meist mit einem deutlichen zeitlichen Abstand zum Abgabezeitpunkt.

Hier liegt die Überlegung nahe, analog zu rechnergestützten Bewertungssystemen für Lösungen zu zum Beispiel Multiple-Choice- oder Mathematikaufgaben auch Lösungen zu Programmieraufgaben automatisiert bewerten zu lassen. Dies nicht nur, um insbesondere im Rahmen von E-Klausuren (respektive summativen Assessments) Zeit beziehungsweise Ressourcen für die manuelle Durchsicht zu sparen, sondern auch um den Lernenden lernunterstützend möglichst umgehend nach Abgabe der Lösung ein individualisiertes Feedback zur Korrektheit und Qualität ihrer Programme und gegebenenfalls zu Verbesserungsmöglichkeiten bereitzustellen (formatives Assessment). Der Einsatz automatisierter Bewertung für das diagnostische Assessment zum Beispiel zur Eignungsüberprüfung oder zur Einstufung in Programmierkurse ist ein weiteres mögliches Einsatzszenario. Idealerweise ermöglichen Systeme für automatisiert beurteilte Programmieraufgaben eine Darstellung des Lernfortschritts, die sowohl für Lernende als auch Lehrende hilfreich ist. Auch können Lernende durch regelmäßige und zeitnah bewertete Übungsaufgaben besser zur kontinuierlichen Mitarbeit angehalten werden. Lehrende hingegen erhalten einen Überblick der Lernfortschritte und Lernhürden einzelner Studierender und der gesamten Gruppe und können ihre Lehre daran ausrichten.

Vor diesem Hintergrund wird seit mehr als 30 Jahren intensiv an der automatisierten Bewertung von Lösungen zu Programmieraufgaben geforscht, wobei die ältesten derartigen Systeme bereits in den 1960er Jahren entwickelt und publiziert wurden. Daher verwundert es nicht, dass mittlerweile eine Vielzahl von Systemen (im Buch wird die im Englischen übliche Bezeichnung „Grader“ verwendet) mit unterschiedlichem Leistungsumfang und für verschiedene Programmiersprachen existiert. Häufig verfügen diese Grader neben der Bewertungsfunktionalität

über eine einfache Benutzer- oder Kursverwaltung. Sie bieten jedoch meist weniger Funktionalität als umfassendere Lernmanagementsysteme (LMS), welche die verschiedensten E-Learning-Bedarfe der Lehre unterstützen und mittlerweile immer häufiger eingesetzt werden. Daher ist es sinnvoll, Grader in LMS zu integrieren, wobei das LMS die Lerngruppen- und Aufgabenverwaltung und der Grader die Bewertung der von den Studierenden im LMS eingereichten Aufgabenlösungen übernimmt. Diese Integration ist allerdings mangels hierauf abgestimmter Schnittstellen zur Integration von Gradern in LMS in der Regel technisch aufwendig sowie oft auf einzelne LMS ausgerichtet und damit nicht ohne Weiteres auf andere LMS übertragbar.

Aufgrund des partiell sehr hohen Aufwands zur Entwicklung von Aufgaben für die automatisierte Programmbewertung ist die Austauschbarkeit von Aufgaben zwischen unterschiedlichen Gradern für eine Programmiersprache sowie der Aufbau von Aufgabenbibliotheken beziehungsweise Aufgaben-Repositorys ein wesentlicher Erfolgsfaktor für die automatisierte Programmbewertung. Allerdings mangelt es aktuell an international anerkannten Standards für die Beschreibung und den rechnergestützten Austausch von Programmieraufgaben, die auch für den Aufbau von LMS- beziehungsweise Grader-unabhängigen Repositorys erforderlich wären. Von Bedeutung sind zudem Algorithmen für einen effizienten Einsatz automatisierter Programmbewertung sowie die Etablierung von Konzepten zur Qualitätssicherung. Eng hiermit verbunden ist die Frage nach geeigneten didaktischen Konzepten zur Integration automatisierter Programmbewertung in die Programmierausbildung an Hochschulen und Universitäten, aber auch an Schulen und anderen Ausbildungseinrichtungen.

Zwar gibt es bereits eine Vielzahl von wissenschaftlichen Veröffentlichungen zum Thema der automatisierten Programmbewertung, unseres Wissens nach sind diese aber über verschiedene Fachcommunitys, Tagungs- und Zeitschriftenreihen verstreut. Das vorliegende Buch unternimmt den Versuch, den Stand der Technik verfügbarer Grader und deren Integration in LMS sowie von Konzepten für den didaktisch sinnvollen Einsatz automatisierter Programmbewertung nach aktuellem Kenntnisstand zusammenzustellen. Dabei fließen in dieses Buch sowohl die Erfahrungen der ersten Projektphase des eCULT-Projekts[1] als auch der ABP-Workshopreihe [ABP13; ABP15] ein. eCULT ist ein vom Bundesministerium für Bildung und Forschung (BMBF) gefördertes Verbundprojekt von 13 niedersächsischen Hochschulen und zwei Vereinen, das seit 2011 die Unterstützung der Präsenzlehre durch digitale Lehr- und Lerntechnologien als Mittel zur Verbesserung der Qualität der Lehre fördert. Das eCULT-Teilprojekt ProFormA[2]

1 http://www.ecult-niedersachsen.de/
2 ProFormA: Programming for Formative Assessment

fokussiert dabei auf Methoden zur Verbesserung der Qualität in der Programmierausbildung durch geeignete Verfahren zur automatisierten Programmbewertung. Drei der Herausgeber dieses Buches und einige der Autoren sind derzeit oder waren in vergangenen Jahren im eCULT-Projekt aktiv involviert. Insbesondere war die eCULT-ProFormA-Gruppe maßgeblich an der Entwicklung eines XML-Austauschformats für Programmieraufgaben (Kapitel 24) beteiligt. Auch die Workshops „Automatische Bewertung von Programmieraufgaben“ (ABP), die 2013 in Hannover und 2015 in Wolfenbüttel stattfanden und ein Forum für deutschsprachige Forscher zu diesem Thema boten, wurden von der eCULT-ProFormA-Gruppe mitinitiiert.

Die Ziele dieses Buches sind zum einen, Lehrenden die Anwendungsreife der automatisierten Programmbewertung durch eine geschlossene Darstellung deutlich zu machen und damit für mehr Verbreitung des Einsatzes automatischer Programmbewertung zu werben. Zum anderen soll dem Thema der automatisierten Programmbewertung ein größerer Bekanntheitsgrad in der Fachöffentlichkeit verschafft werden und auch Arbeitsergebnisse aus dem eCULT-Projekt und den ABP-Workshops einem größeren Anwenderkreis präsentiert werden. Zielgruppen des Buches sind Lehrende an Hochschulen und Schulen, Hochschuldidaktiker/innen respektive E-Learning-Serviceeinrichtungen und Informatiker/innen, die mit der Entwicklung von Gradern und Integrationslösungen befasst sind.

Das erste Kapitel des Buches bietet eine Übersicht verfügbarer Systeme zur automatisierten Programmierbewertung anhand einer Zusammenfassung von Review- und Survey-Artikeln zu diesem Thema (Kapitel 2). Das Buch schließt mit einem Ausblickkapitel, in welchem noch geplante Entwicklungen des eCULT-Projekts und offene Fragestellungen der automatisierten Programmbewertung dargestellt werden (Kapitel 25). Dazwischen ist das Buch in drei Teile gegliedert, welche im Folgenden kurz beschrieben werden.

Teil I fokussiert auf didaktische Einsatzszenarien rechnergestützter Programmbewertung ausgehend von derzeit relevanten Programmierparadigmen und Programmiersprachen, welche in je einem Unterkapitel beschrieben sind. Im Vordergrund steht dabei nicht die Beschreibung des verwendeten Graders, sondern die Beschreibung der Einsatzszenarien mit Angaben zu Zielgruppen (z. B. Studierende eines Informatikstudiengangs im ersten Studienabschnitt), Veranstaltungen (z. B. „Einführung in die objektorientierte Programmierung“) und didaktischen Konzepten der Veranstaltungen, falls relevant ergänzt um Angaben zu den allgemein eingesetzten digitalen Medien und E-Learning-Konzepten (z. B. Blended Learning unter Einsatz eines LMS). Primäre Zielgruppe dieses Teils sind Lehrende an Hochschulen, Schulen und in der beruflichen Ausbildung, die automatisierte Programmbewertung in ihre Lehre integrieren möchten. Weiterhin sind die Unter-

kapitel interessant für Mitarbeiter von E-Learning-Serviceeinrichtungen, die ein Beratungsangebot für Werkzeuge zur automatisierten Programmbewertung aufbauen wollen.

Teil II des Buches nimmt eine technisch orientierte Sichtweise ein und betrachtet verschiedene Systeme zur Bewertung von Aufgabenlösungen. Diese Grader haben zum Teil sehr unterschiedliche Anwendungsbereiche, Schwerpunkte und Entwicklungsgeschichten. Ihre Auswahl ist exemplarisch und erhebt keinen Anspruch auf Vollständigkeit, da eine umfassende Darstellung aller aktuell und früher verfügbaren Systeme den Rahmen dieses Buches mehr als sprengen würde. Ein Kriterium für die Auswahl der betrachteten Grader ist allerdings, dass die Grader zum Zeitpunkt der Buchveröffentlichung aktiv in der Lehre eingesetzt werden und auch für Interessierte potenziell verfügbar sind. Die ausgewählten Grader werden in den jeweiligen Kapiteln detailliert vorgestellt und am Ende dieses Teils des Buches mit kurzen Tool-Steckbriefen und einer Feature-Matrix übersichtlich einander gegenübergestellt. Primäre Zielgruppen dieses Teils sind die wissenschaftlich und technisch interessierte Fachcommunity (insbesondere Entwickler von Gradern und Gradingverfahren) sowie Mitarbeiter von E-Learning-Servicestellen, die geeignete Grader für die Lehrenden ihrer Einrichtung auswählen und bereitstellen wollen. Weiterhin interessant sind die Kapitel für an spezifischen Gradern interessierte Lehrende.

In Teil III des Buches wird die Interoperabilität von Gradern untereinander und mit LMS thematisiert. In den ersten vier Unterkapiteln steht die Perspektive der LMS im Vordergrund, es werden verschiedene Ansätze und die damit verbundenen Integrationsaufwände und -probleme der verschiedenen LMS vorgestellt. Hierbei werden im deutschsprachigen Raum bekannte LMS, wie Moodle (Kapitel 18), LON-CAPA (Kapitel 19), Stud.IP (Kapitel 20) und ILIAS (Kapitel 21) anhand von standortbezogenen Erfahrungen vorgestellt. In weiteren zwei Unterkapiteln werden Middleware-Konzepte diskutiert, welche die Arbeiten und Aufwände an den Schnittstellen der LMS und Grader vereinfachen sollen. Um Programmieraufgaben zwischen LMS, aber auch Gradern austauschbar zu machen wird im letzten Kapitel das Austauschformat der ProFormA-Gruppe erläutert. Primäre Zielgruppen dieses Teils sind die wissenschaftlich und technisch an automatisierter Programmbewertung interessierte Fachcommunity sowie Mitarbeiter von E-Learning-Servicestellen, die Grader und Aufgabenbibliotheken für die Lehrenden ihrer Einrichtung integriert in das ebenda verwendete LMS bereitstellen wollen.

Einige Themen bleiben in diesem Buch bewusst ausgeklammert, ohne ihnen damit Relevanz im Rahmen der automatischen Bewertung von Programmieraufgaben absprechen zu wollen. Gleich aus zwei Gründen gibt es kein eigenes Kapi-

tel über Plagiatserkennung: Einerseits sind Plagiate kein spezielles Phänomen der Programmierausbildung, sondern sind in nahezu jedem Fach anzutreffen. Andererseits gibt es in vielen Gradern sehr gute individuelle Lösungen, die im Detail in einigen Kapiteln von Teil II beschrieben werden. Das Thema ist aber auch in der Softwareentwicklung generell relevant im Hinblick auf Codeanalysen und Patentverletzungen. Eine umfassende Darstellung dieses Themas müsste also weit über den Rahmen dieses Buches hinausgehen. Gleiches gilt für den Datenschutz mit Blick auf die sensiblen, personenbezogenen Daten der Studierenden, die insbesondere im Rahmen von Prüfungen anfallen. Auch hier wird bei der Vorstellung einzelner Grader auf besonders bemerkenswerte Lösungen eingegangen, während eine umfassende Darstellung weit über die Programmierausbildung hinaus reichen müsste. Auch andere für das E-Learning relevante Themen, wie zum Beispiel Open Educational Resources (OER), Lizenzierung von Software und Aufgaben bezüglich des Austauschs und IT-sicherheitsrelevante Fragestellungen können im Rahmen dieses Buches nur randständig angesprochen werden.

Trotz dieser Auslassungen hoffen wir, Ihnen mit diesem Buch einen Überblick über den Stand der Technik sowie die Potenziale und aktuellen Grenzen des Einsatzes automatisierter Programmbewertung in der Programmierausbildung verschaffen zu können. Noch mehr hoffen wir, dass dieses Buch Lehrende ermutigt, automatisierte Programmbewertung in der Lehre einzusetzen, sowie E-Learning-Servicestellen unterstützt, geeignete Grader idealerweise integriert in das hauseigene LMS anzubieten. Und vielleicht trägt das Buch ja sogar dazu bei, dass sich weitere Informatikerinnen und Informatiker in die aktive Weiterentwicklung der automatisierten Programmbewertung einbringen.

Wir danken den zahlreichen Autoren für deren Beiträge zu diesem Buch, dem ELAN e.V. für die Aufnahme dieses Buches in dessen Reihe „Digitale Medien in der Hochschullehre“ und die damit verbundene finanzielle Förderung dieses Buchprojekts sowie dem BMBF für die Unterstützung der partiell dieser Veröffentlichung zugrunde liegenden Projekte. Weiterhin danken wir der Hochschule Hannover, der Ostfalia Hochschule für angewandte Wissenschaften, der Technischen Universität Clausthal sowie der Universität Osnabrück für die Förderung des Buchprojekts. Schlussendlich danken wir dem Waxmann Verlag für die hervorragende Betreuung des Veröffentlichungsprozesses.

Oliver J. Bott, Peter Fricke, Uta Priss und Michael Striewe
Hannover, der 17.03.2017

Literatur für dieses Kapitel

[ABP13] *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. (http://ceur-ws.org/Vol-1067/). 2013.

[ABP15] *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2015)*. Bd. 1496. CEUR Workshop Proceedings. (http://ceur-ws.org/Vol-1496/). 2015.

2 Automatisierte Bewertung in der Programmierausbildung – Eine Übersicht

Sven Strickroth und Niels Pinkwart

Zusammenfassung
Dieses Kapitel gibt eine Übersicht über verschiedene Sichtweisen und Klassifikationsansätze für Programmierlernsysteme beziehungsweise automatische Bewertungssysteme. Dabei werden besondere Ansätze innerhalb dieser Kategorisierungen mit Beispielen hervorgehoben sowie Trends identifiziert und diskutiert.

2.1 Einleitung

An vielen Schulen, Hochschulen und in der Weiterbildung nimmt die Relevanz des Themas Informatik gegenwärtig stark zu – so werden zum Beispiel Rufe nach einem Pflichtfach Informatik in Schulen lauter, zahlreiche Universitäten führen Einführungskurse zu IT-Themen in die Curricula von auch nicht informatischen Studiengängen ein, und Selbstlernangebote wie MOOCs werden insbesondere zu IT-Themen stark frequentiert. In diesen Lern- und Ausbildungskontexten spielt Programmierung typischerweise eine zentrale Rolle. Nicht allen fällt es jedoch leicht, Programmierfähigkeiten zu erwerben: Oft stehen Lernende beim Erstellen, Testen und Debuggen von Programmen vor schier unüberwindlichen Hindernissen. Lehrpersonen können hier natürlich unterstützen, haben aber nicht immer die Zeit, auf die Bedürfnisse aller Kursteilnehmer ausführlich einzugehen.

Computer sind in der Lage, Lernerlösungen zu Programmieraufgaben automatisiert zu analysieren. Die Ergebnisse der automatischen Bewertung können zum Beispiel als Feedback an Lerner zurückgespiegelt werden, um diese bei der individuellen Lösungsfindung zu unterstützen. In Lernszenarien, an denen viele Lernende teilnehmen, aber nur begrenzte Lehrpersonenzeit zur Verfügung steht, wie etwa in E-Learning-Kursen oder großen universitären Veranstaltungen, ist diese

partielle Ersetzung von Lehrpersonal bei der Aufgabe der Prüfung von Lösungsversuchen praktisch hochrelevant [KJH16] und wird von Lernenden auch nicht grundsätzlich abgelehnt [LMM98; Mat94]. Nach Auffassung von Kay und Kollegen [Kay+94] sei auch ganz grundsätzlich die manuelle Bewertung von Lernerlösungen hinsichtlich Konsistenz und Genauigkeit als problematisch anzusehen. Eine andere Nutzungsvariante der automatischen Programmanalyse besteht darin, Lehrpersonen bei der Bewertung und Feedbackgabe zu unterstützen [SG14] – so kann eine semiautomatische Bewertung insbesondere Vorteile hinsichtlich einer ganzheitlichen Bewertung bieten, da so auch Punkte bei kleineren Fehlern vergeben werden können (vgl. [AM05]).

Unabhängig von der Zielperson, welcher die Ergebnisse der automatischen Programmanalyse präsentiert werden (Dozent oder Lerner), müssen Analyseverfahren für Lernerlösungen im Bereich der Programmierung in der Lage sein, auch mit fehlerhaften und unvollständigen Lösungen umzugehen, die zusätzlich noch bei komplexeren Aufgabenstellungen auf verschiedenen und eventuell fehlerbehafteten Lösungsstrategien beruhen können [LLP13]. Diese Aspekte sind eine Herausforderung für automatisierte Analyseverfahren wie auch für den Entwurf von Feedbackmechanismen auf Basis der Analyseergebnisse. Professionelle Programmierentwicklungsumgebungen (IDEs – Integrated-Development-Environments) sind diesen Herausforderungen mit ihren Analyseverfahren nicht gewachsen: Sie haben Funktionen, die für Novizen zu schwierig sein können. Außerdem sind die in IDEs eingebauten Typen von Codeanalyse und Rückmeldungen nicht auf Lerner zugeschnitten und können diese daher oft eher verwirren als ihnen helfen [Pea+07].

Aufgrund der praktischen Relevanz und der weder eindeutigen noch einfachen Konstruierbarkeit von Systemen zur automatisierten Bewertung von Lernerlösungen zu Programmieraufgaben ist es wenig überraschend, dass Forschung und Entwicklung zu Unterstützungssystemen für die Programmierausbildung mittlerweile eine beachtliche Tradition in der Informatik haben. In der Forschung reicht diese mit dem LISP-Tutor mindestens bis in die 1980er Jahre zurück [AFS84]. Forschungsergebnisse zu mehreren hundert weiteren Systemen wurden in der Zwischenzeit publiziert – so beinhalteten 99 von den insgesamt 444 zwischen 1984 und 2003 auf dem SIGCSE Technical Symposium veröffentlichten Forschungsartikeln ein Softwareunterstützungssystem [Val04]. Die Zahl nicht publizierter Systeme, die sich in der universitären Praxis in der Anwendung befinden, ist sicherlich noch deutlich größer, da wahrscheinlich jedes Informatikinstitut in irgendeiner Weise ein eigenes System zur Unterstützung des Lehrbetriebs implementiert hat. Viele Systeme weisen sehr ähnliche Features auf und wären grundsätzlich auch für ein Assessment in anderen Szenarien einsetzbar [Iha+10], was jedoch

nicht praktiziert wird. Die folgenden Gründe für die Vielfalt an Systemen wurden von [Pea+07] und [Iha+10] identifiziert:

- Ein System wurde für einen speziellen Kurs entwickelt. Dabei wird ein Tool beziehungsweise Analyseverfahren genau für diesen Kurs angepasst und ist dadurch in der Regel nicht ohne weiteres in anderen Kontexten erneut einsetzbar. Die Adaptierung an andere Kontexte wäre oft zwar möglich – aber es gibt niemanden, der dies tut.

- Ein System wurde im Rahmen einer Abschlussarbeit oder eines Forschungsprojekts entwickelt. Dabei steht die Erstellung und Evaluation eines neuen Ansatzes im Mittelpunkt, weniger die Praxistauglichkeit. Bei so erstellten Systemen handelt es sich meist um einen Prototypen, der nach dem Abschluss des Projekts beziehungsweise der Abschlussarbeit keinen Support mehr erhält.

Diese Gründe, wie auch die teils nicht erfolgte Publikation von Systemen, trugen sicherlich dazu bei, dass immer wieder ähnliche Systeme entwickelt und lokal eingesetzt wurden, dass sich aber bislang kaum Systeme breit durchgesetzt haben [Iha+10]. Neben diesen eher praktisch-organisatorischen Argumenten ist es jedoch natürlich auch so, dass auf der Ebene der Forschung zu Programmierlernsystemen in den letzten 4 Jahrzehnten etliche Weiterentwicklungen stattgefunden haben. Folglich gibt es auch eine Reihe von Review- und Survey-Artikeln [DE88; DM98; LMM98; KP05a; Guz04; GA05; DLO05; AM05; Pea+07; Iha+10; CR13; SG14; Le16; KJH16], von denen im folgenden Abschnitt die acht aktuellsten genauer betrachtet und vorgestellt werden. Diese zeigen auf, welche verschiedenen Sichtweisen und Klassifikationsansätze für die Menge an Programmierlernsystemen in der jüngeren Vergangenheit gewählt worden sind.

2.2 Übersicht über verschiedene Reviews

In diesem Abschnitt werden verschiedene Sichtweisen und Klassifikationsansätze für Programmierlernsysteme und automatische Bewertungssysteme vorgestellt. Tabelle 2.1 gibt dabei einen Überblick über die Fokusse der einzelnen betrachteten Reviews. Aufgrund des Themas dieses Buches wurden Reviews zu Visualisierungsansätzen (z. B. [SKM13]), Programmierumgebungen (z. B. [Guz04; KP05a]) und intelligenten Tutoring-Systemen (ITS, z. B. [Pil03]) nicht genauer betrachtet.

Review	Fokus/Klassifizierung
Douce, Livingstone und Orwell [DLO05]	historische Übersicht; identifiziert auch allgemeine Schwierigkeiten und Probleme von automatischem Assessment
Ala-Mutka [AM05]	automatische Bewertungsmethoden; diskutiert auch semiautomatisches vs. automatisches sowie formatives vs. summatives Assessment
Pears u. a. [Pea+07]	Lehrmethoden; geht auch auf Curricula, Pädagogik und Sprachauswahl ein
Ihantola u. a. [Iha+10]	Systemfeatures; geht dabei auch auf Aspekte wie Sicherheit, Lizenzierung und Verfügbarkeit ein
Caiza und Ramiro [CR13]	Bewertungsmetriken; grobe Klassifikation in „mature“ und „recently developed tools“
Striewe und Goedicke [SG14]	statische Analysetechniken; stellt auch Vor- und Nachteile dieser Analysetechniken vor
Le [Le16]	adaptive Feedbackansätze; stellt auch aktuelle Forschungslücken dar
Keuning, Jeuring und Heeren [KJH16]	formative Feedbackansätze; analysiert zudem Anpassbarkeit und Qualität von Evaluationen

Tabelle 2.1: Schwerpunkte der betrachteten Review- und Survey-Artikel

2.2.1 Historische Übersicht und Klassifizierung nach Generationen

Das Review von [DLO05] betrachtet die Geschichte von automatischen testbasierten Bewertungstools ausgehend von ersten Publikationen in den 1960er Jahren bis 2005. [DLO05] klassifiziert die Tools anhand des Alters in drei Generationen:

Erste Generation – frühe Bewertungssysteme Die erste Generation beinhaltet erste Ansätze von Entwicklungen automatisierter Bewertungssysteme für Programmiersprachen auf digitalen Systemen im weitesten Sinne.

Zweite Generation – Toolorientierte Systeme Die zweite Generation beschreibt verschiedene Ansätze und Tools, die von Dozenten genutzt und entwickelt wurden, um die automatische Bewertung für sie zu vereinfachen (z. B. durch skriptbasierte Ansätze zum Testen, oftmals kommandozeilenbasiert), teilweise schon mehrere Kriterien auf einmal zu überprüfen und daraus Reports zu generieren. Fortgeschrittenere Systeme dieser Genera-

tion konnten auch von Lernenden genutzt werden, um Vorlesungsnotizen sowie Aufgaben abzurufen (z. B. Ceilidh, [Hig+03]).

Dritte Generation – Weborientierte Systeme Einige Tools der zweiten Generation wurden um Webtechnologien erweitert und richten sich sowohl an Lernende als auch an Lehrende, um sie bei Schwierigkeiten zu unterstützen und einen Überblick über den Fortschritt aller Lernenden bereitzustellen (z. B. CourseMarker, [Hig+03], oder BOSS2, [JGB05]). Zentrale Erweiterungen bestehen in der Integration von Plagiatserkennungen und Eingriffsmöglichkeiten von Lehrenden in die automatische Bewertung.

Darüber hinaus wurden drei Schwierigkeiten als allgemeine Probleme von automatischen Bewertungssystemen identifiziert: 1) Die Spezifikation von Aufgaben muss sehr sorgfältig durchgeführt werden, um Interpretationsschwierigkeiten zu vermeiden. 2) Das Erstellen von grafischen Programmen ist für Lernende attraktiver, jedoch sind diese sehr schwer automatisiert zu überprüfen. Zum Beispiel sind sehr konkrete Events oder Methoden vorzugeben oder es ergeben sich allgemeine Probleme: „Zeichen Sie die britische Flagge in einem Applet“. 3) Eine Lösung kann nach Meinung von Douce zwar funktional korrekt, aber pathologisch konstruiert sein – daher könne nur durch die Verbindung verschiedener Kriterien eine automatische Bewertung erfolgen.

2.2.2 Übersicht und Kategorisierung automatischer Bewertungsmethoden

[AM05] fokussiert in ihrem Review auf Merkmale, die automatisch bewertet werden können. Sie unterscheidet grundsätzlich zwischen dynamischen Bewertungen, die eine Ausführung der Lernerlösung erfordern, und statischen Bewertungen, die allein anhand des Quellcodes durchgeführt werden können:

2.2.2.1 Dynamisches Assessment

Charakteristisch für dynamisches Assessment ist das Ausführen der Lernerlösungen. [AM05] identifiziert dabei die folgenden Aspekte, die betrachtet werden können. Grundsätzlich wird für die Ausführung eine besonders gesicherte Umgebung benötigt.

Funktionalität Die Funktionalität wird durch Überprüfung von Ausgaben gegen vorher definierte Testeingaben geprüft. Dies kann über festgelegte er-

wartete Ausgaben des gesamten Programms (Blackbox-Test, exakt oder zum Beispiel mit regulären Ausdrücken) oder auf der Basis von einzelnen Methoden (Whitebox-Test) erfolgen, wobei die Ausgaben auch dynamisch mit denen einer Musterlösung verglichen werden können. Im Fall von Whitebox-Tests können Lernenden dabei konkrete zu implementierende Schnittstellen vorgegeben werden (die Mehrzahl der Systeme geht so vor) oder Lernende lösen sogenannte Fill-in-the-Gap-Aufgaben, in denen der studentische Quellcode in eine Vorlage eingefügt wird (z. B. ELP, [Tru07]).

Effizienz Effizienz kann zum Beispiel durch das Messen der Ausführungszeit mit verschiedenen Eingabedaten bewertet und mit einer Musterlösung verglichen werden. Ceilidth und Assyst verwenden zum Beispiel dynamisches Profiling für die Messung der Effizienz, wobei gezählt wird, wie oft bestimmte Blöcke ausgeführt werden [AM05].

Testfähigkeiten Lernende sollen lernen, Tests für ihre Programme zu entwickeln und selbstständig vor der Abgabe zu überprüfen. Dabei gibt es verschiedene Ansätze, wobei Lernende sowohl ihre Lösung als auch ihre Tests einreichen: [Che04] überprüft die Tests der Lernenden durch fehlerhafte „Musterlösungen“. Als ein weiteres Beispiel wird Web-CAT [Edw04] genannt; ein System, das ausschließlich eine Aufgabenstellung sowie eine durch den Dozenten erstellte Musterlösung benötigt. Zur Bestimmung einer Bewertung werden schließlich die von den Lernenden erstellten Tests gegen die Musterlösung sowie deren eigene Lösung ausgeführt und mit der Testabdeckung verrechnet.

Spezielle Features [Rin99] überprüft spezifische Aspekte der Programmiersprachen, indem das normale Memorymanagement mit einem eigenen überschrieben wird, um Missmanagement, wie zum Beispiel nicht freigegebenen Speicher, zu entdecken.

2.2.2.2 Statisches Assessment

Statisches Assessment erfolgt ausschließlich auf der Quellcodeebene. Gemeinsame Voraussetzung für die durch [AM05] identifizierten Ansätze sind syntaktisch korrekte Lösungen.

Coding-Style Neben der Überprüfung, ob ein Programm syntaktisch korrekt ist, können Abgaben zum Beispiel auf ungenutzte Variablen (wofür viele

Interpreter und Kompiler Funktionalität mitbringen) oder Einhaltung von Style-Regeln geprüft werden (z. B. Checkstyle).

Programmierfehler Unabhängig von dynamischen Funktionstests, die von den meisten Tools im Review genutzt werden, können statische Analysen zum Einsatz kommen, um häufige Fehler zu erkennen (z. B. Zählvariable in einer Schleife wird nicht aktualisiert, redundante Zuweisungen oder idempotente Operationen).

Softwaremetriken Softwaremetriken können genutzt werden, um allgemeine Indikatoren einer Software zu bestimmen (z. B. Anzahl der Attribute, Anzahl von Operationen und Operanden, Codezeilen oder die zyklomatische Komplexität).

Design Zum Design zählen Überprüfungen, ob eine Lösung eine vorgegebene Schnittstelle einhält oder wie sich die Struktur einer Lösung zu Musterlösungen verhält. Daneben gibt es Ansätze, die aus den Lösungen UML-Diagramme generieren, Software-Patterns erkennen und dies zur Bewertung nutzen.

Spezielle Features Hierzu zählen zum einen Überprüfungen, ob eine Lösung bestimmte Strukturen der Sprache oder Bibliotheksfunktionen nutzt, die zum Beispiel für eine Aufgabe ausgeschlossen sind. Zum anderen spielen hier auch Plagiatserkennungsmethoden eine Rolle.

Neben der Kategorisierung diskutiert [AM05] auch allgemein die Nutzung von automatischen Bewertungsmethoden in der Ausbildung: Es darf keine Uneindeutigkeiten in der Spezifikation geben und selbst kleinste Fehler in den Tests können zu großen Problemen führen, wobei die Auswahl der eingesetzten Bewertungsmethoden didaktisch begründet werden sollte. Zudem werden Systeme oftmals als reine Bewertungssysteme angesehen. Sie können darüber hinaus auch zur „automatisierten Verwaltung“ von zum Beispiel Peer-Reviews oder komplexeren Settings, wie im Quiver-System [EFK04] genutzt werden: Dort erstellen fortgeschrittene Lernende Schnittstellen und Tests, die danach von Anfängern in Einführungsveranstaltungen implementiert werden. Weiterhin werden semiautomatische und vollautomatische Bewertung sowie formatives und summatives Feedback diskutiert. [AM05] weist darauf hin, dass nicht alle Kriterien „guten Programmierens“ automatisch bewertet werden können: Menschen bewerten Lösungen ganzheitlich und vergeben eventuell auch bei kleineren Fehlern (z. B. Syntaxfehlern) die volle Punktzahl. Dies kann durch semiautomatische Bewertung ebenfalls erreicht werden, wobei Tests vollautomatisch durchgeführt und die Ergebnisse den Bewertern

präsentiert werden. Einige Systeme sind auf ausschließlich summative Bewertungen ausgelegt, andere geben Feedback und erlauben erneute Einreichungen, wobei sehr viele Ansätze, obwohl sie iterative Entwicklungen hervorheben, immer eine vollständige Version der Lösung erwarten. [AM05] identifiziert innerhalb der formativen Ansätze Tutoring-Systeme als weitere Unterkategorie, die hauptsächlich zur Unterstützung von Lernenden entwickelt wurden und dadurch im Vergleich mehr (adaptives) Feedback bereitstellen als reine Assessment-Systeme.

2.2.3 Übersicht über Lehrmethoden zur Einführung in die Programmierung

Das Literatur-Survey von [Pea+07] beschäftigt sich mit der Beantwortung der Frage: „Welche Forschungsliteratur kann Lehrende unterstützen neue Einführungsprogrammierkurse zu entwerfen?“. Dazu beschäftigt sich das Survey sowohl mit Curriculadesign, pädagogischen Aspekten, Auswahl einer passenden Programmiersprache und schließlich auch mit Tools, die sich speziell an Programmieranfänger richten und in diesem Kontext eingesetzt werden können. [Pea+07] klassifizieren Systeme in die folgenden vier Kategorien. Insgesamt spielen Systeme zur automatischen Programmbewertung dabei nur eine untergeordnete Rolle.

Visualisierungstools Da Menschen gut beim Verarbeiten von grafischen Informationen sind, liegt es nahe, eher abstrakte Algorithmen und Konzepte zu visualisieren. Es kann grundsätzlich zwischen Tools unterschieden werden, die Strukturen und die Ausführung von Code veranschaulichen (z. B. visuelle Debugger, Animationen oder Simulationen).

Automatische Bewertungstools Tools zur automatischen Programmbewertung dienen meist sowohl der Erleichterung von Lehrenden als auch der Bereitstellung von formativem Feedback für Lernende in angemessener Zeit. Viele Tools bieten dabei Unterstützung, um die Korrektheit von Abgaben zu prüfen (meist Blackbox-Testing). Es gibt aber auch Ansätze interne Datenstrukturen oder Programme detaillierter zu überprüfen. Die Bewertung erfolgt in den Tools oftmals semiautomatisch, aber zum Teil auch vollautomatisch. Dies kann zum Beispiel zu einer Verringerung der Subjektivität bei Bewertungen genutzt werden. Grundsätzlich sind diese Tools nicht auf Programmcode beschränkt, sondern es gibt auch Systeme, die Ablaufdiagramme (CourseMarker, [Hig+05]) oder Algorithmen in Simulationsumgebungen (TRAKALA2, [Nik+04]) bewerten.

Programmierumgebungen Umgebungen stellen Programmierern aller Erfahrungsstufen Möglichkeiten und Unterstützungen bereit, um Programme zu erstellen und auszuführen. Eine Unterteilung erfolgt in „Programming support tools“ (z. B. reduzierte IDEs wie BlueJ, [Kö+03]) und „Microworlds“ (Umgebungen basierend auf physischen Metaphern, z. B. Karel oder Robocode). Eine weitere Übersicht über Programmierumgebungen findet sich bei [KP05a].

Andere Tools In diese Kategorie fallen Tools zur Plagiatserkennung oder intelligente Tutoring-Systeme (z. B. LISP Tutor, [AS86]).

2.2.4 Features automatischer Bewertungstools

Ziel des Reviews von [Iha+10] ist die Darstellung der Features von automatischen Bewertungstools für Programmierübungen aus den Jahren 2006–2010 basierend auf einem systematischen Literaturreview. Zentrale Ergebnisse des Reviews sind:

Unterstützte Programmiersprachen Die Mehrheit der Systeme zielen auf Java oder unterstützen diese Sprache. Weiterhin werden C/C++, Python und Pascal, aber auch Assembler, Shell Scripts oder VHDL von einigen Systemen angeboten. Es gibt aber auch Systeme, die unabhängig der Sprache zum Beispiel für Blackbox-Tests (darunter auch zum Testen von Webapplikationen, z. B. AWAT, [SQF08]) eingesetzt werden können.

Integration in Learning-Management-Systeme Es zeigt sich ein Trend zur Integration von Bewertungssystemen in LMS, da dadurch zum Beispiel keine Kursverwaltungsfunktionen erneut implementiert werden müssen.

Definition der Tests Tests werden entweder mit Tools aus der Industrie (XUnit, Akzeptanztest-Frameworks sowie Web-Testing-Frameworks wie Watir) oder spezialisierten Lösungen (Blackbox-Testing über Ausgabenvergleiche, Scripting sowie neuen Ansätzen aus der Forschung) durchgeführt.

Erneute Einreichung von Lösungen Erneute Einreichungen werden partiell sehr unterschiedlich gehandhabt: Bei den meisten Systemen kann die Anzahl der Abgaben limitiert werden. Andere Ansätze bestehen im Einschränken des Feedbacks, Zeitstrafen für fehlgeschlagene Tests, Nutzung parametrisierter Aufgaben oder Durchführung von sogenannten Programmierkontesten (Lösung von Aufgaben gegen die Zeit und andere Teilnehmer) und Kombinationen daraus.

Möglichkeiten einer manuellen Bewertung Im Review wurden drei Level für den Umfang manueller Bewertungen identifiziert: keine Eingriffsmöglichkeiten, reine manuelle Bewertung sowie Kombinationen aus manuellen und automatischen Bewertungen, bei denen Lehrende das automatische Feedback durch weitere Anmerkungen anreichern. Für kein System des Reviews ist es explizit möglich, die automatische Bewertung zu überschreiben.

Sandboxing (Sicherheit) Die Bewertung von eingereichten Lösungen erfordert es oftmals, dass diese ausgeführt werden sollen. Lösungsansätze beinhalten die Ausführung in einer geschützten Umgebung, Entfernen von möglichem schädlichen Code mittels statischer Analyse und Durchführung der Bewertung direkt auf dem Client. Teilweise werden Lösungen auch auf gesonderten Servern ausgeführt.

Verbreitung und Verfügbarkeit Im Review zeigte sich, dass nur sehr wenige Systeme unter einer Open-Source-Lizenz veröffentlich wurden beziehungsweise frei verfügbar sind. Viele der publizierten Prototypen konnten nicht im Internet gefunden werden.

Spezielle/einmalige Funktionen Als Besonderheiten stellten sich Systeme für die automatische Bewertung von grafischen Benutzeroberflächen, SQL-Tutoring, nebenläufige Programmierung, Webprogrammierung und Ansätzen heraus, in denen Lernende gegenseitig die Qualitätssicherung vornehmen.

2.2.5 Kategorisierung in ausgewachsene und neue Tools sowie Übersicht über Bewertungsmetriken

[CR13] unterteilen die Tools in ihrem Review in „mature tools“ und „recently developed tools“ und beschreiben unter anderem „key features“, unterstützte Programmiersprachen und Bewertungsmetriken. Auf welcher Grundlage die Systeme in diese beiden Kategorien eingeordnet werden, wird nicht genauer erläutert, außer allgemein Feature-Reichtum und Verbreitung zu nennen. Die meisten der „mature tools“ sind unter einer Open-Source-Lizenz veröffentlicht und unterstützen Java, wobei insbesondere bei den neuen Tools ein Trend zu LMS-Extensions (Moodle-Plugins) erkennbar ist.

Ein weiterer Beitrag des Reviews besteht in der Diskussion von Bewertungsmetriken. Es wird hervorgehoben, dass jede Institution beziehungsweise jeder Dozent eigene Metriken verwendet, wobei diese nach Meinung der Autoren jedoch möglicherweise nicht vollständig automatisch zur Bestimmung eines Wer-

tes angewendet werden können. Jedes Tool des Reviews enthält eigene Bewertungsmethoden und -metriken, um eine Note für studentische Abgaben zu bestimmen. Teilweise sind diese im System fest einprogrammiert, können aber zum Teil auch durch Plug-Ins erweitert werden. Insgesamt wurden oftmals für gleiche beziehungsweise sehr ähnliche Metriken verschiedene Namen gewählt. Tabelle 2.2 zeigt die identifizierte Klassifikation von Metriken. Letztlich wird die Lücke eines fehlenden konfigurierbaren Bewertungsschemas identifiziert, um beliebige Metriken abzubilden.

Kategorisierung		**Metrik**
Ausführung		Kompilierung (z. B. Compiler)
		Ausführung (z. B. Interpreter)
Funktionstests		Funktionalität (System- oder Methodenlevel, z. B. Unit-Tests)
Nicht-funktionale Tests	Spezielle Anforderungen	Spezielle Anforderungen an eine Aufgabe
	Wartbarkeit	Design
		Stil (z. B. Style-Checker)
		Komplexität
	Effizienz	Nutzung physischer Ressourcen
		Ausführungszeiten
		Anzahl von Prozessen
		Datei- oder Quellcodegröße

Tabelle 2.2: Klassifikation von Bewertungsmetriken nach [CR13]

2.2.6 Statische Analysetechniken für die automatische Programmbewertung

Das Review von [SG14] fokussiert auf didaktische Vor- und Nachteile von statischen Analysetechniken, die zur automatischen Programmbewertung eingesetzt werden können. Das Review beschränkt sich dabei auf Ansätze für die objektorientierte Programmierung in Java sowie Technologien, die in serverbasierten Systemen eingesetzt werden können. Folglich werden Analyse- und Feedbackmechanismen aus IDEs ausgeklammert. Eine weitere Annahme besteht darin, dass Lernende in der Lage sind, lokal auf ihren Computern (z. B. mithilfe von IDEs) syntaktisch korrekte Programme zu erzeugen.

Die folgenden zentralen Anforderungen für statische Analysen in diesem Kontext werden genannt:

- Statische Analysen sind in der Lage Misskonzeptionen aufzudecken, die auch in syntaktisch korrekten Lösungen auftreten können.
- Feedback muss auch für Teile der studentischen Abgabe gegeben werden, die keinen relevanten Beitrag zur Lösung liefern.
- Statische Analysen können den Quellcode nach „verbotenen“ oder „erforderlichen“ Konstrukten durchsuchen (z. B. soll Rekursion und keine Schleifen benutzt werden).
- In Tutoring-Szenarien sollten Lernende nicht nur über Fehler, sondern auch über mögliche Fehlerbehebungen oder nächste Schritte informiert werden.
- Überprüfungen auf Plagiate.

Neben den Anforderungen und technischen Lösungen werden auch die Integration von Tools in existierende Systeme sowie das Erstellen von Tests und Feedbackregeln diskutiert: Die Tools des Reviews unterstützen hauptsächlich die Anwendung von regelbasierten Überprüfungen auf Quell- oder Bytecodeebene. Zum Einsatz kommen vor allem Analysetools aus der professionellen Softwareentwicklung, wie zum Beispiel Checkstyle, PMD und FindBugs, oder Analysen von abstrakten Syntaxbäumen beziehungsweise Graphen. Eine wichtige Voraussetzung zum Einsatz in der Lehre sind konfigurierbare Tests, die auch von Lehrpersonen angepasst werden können. Zum Teil sind diese fest einprogrammiert (z. B. FindBugs), erfordern das Erstellen von Regeln in vorgegebenen Anfragesprachen (z. B. XPath oder GReQL, [BE08]) oder die Nutzung von komplexen Analyseprogrammen, die in das Analyseframework integriert werden müssen. Um zu einer Bewertung zu gelangen, kann bei existierenden professionellen Tools zur Gewichtung auf vorgegebene Schweregrade zurückgegriffen werden (z. B. bei PMD, Checkstyle oder FindBugs).

2.2.7 Klassifikation von adaptiven Feedbackansätzen

Das Review von [Le16] fokussiert auf eine Klassifikation von adaptiven Feedbackansätzen. Dabei werden die Feedbackansätze wie folgt unterteilt:

Ja/Nein-Feedback Bei dem Ja/Nein-Feedback handelt es sich um die kürzeste Variante, die lediglich angibt, ob eine Lösung korrekt ist oder nicht. Dazu zählen zum Beispiel auch „Es liegen Syntaxfehler vor“, „Es gibt keine Syntaxfehler, aber die Berechnung ist falsch“ oder „Lösung ist korrekt“.

Handelt es sich um abgefragte Berechnungen zum Beispiel von Variablenbelegungen, wird den Lernenden teilweise auch eine richtige Lösung präsentiert. In der Regel werden Lösungen mit vorgegebenen Werten verglichen.

Syntaxfeedback Dieses Feedback basiert auf den Ausgaben eines Compilers, die von den meisten Systemen unverändert an die Lernenden durchgereicht werden. Es gibt aber auch Ansätze (z. B. bei VC Prolog, [Pey+00]), in denen eine detaillierte syntaktische Fehlerbeschreibung generiert wird.

Semantisches Feedback Semantisches Feedback bezieht sich auf Hinweise auf Fehler in der Erfüllung der Anforderungen. Dabei kann das Feedback einerseits zweistufig („intention- and code-based“) oder andererseits ausschließlich „code-based“ generiert werden. Im ersten Fall wird versucht die Intention beziehungsweise Lösungsstrategie der Lernenden zu erkennen und basierend darauf Hinweise zu geben. Ansätze finden sich hier vor allem für Prolog und funktionale Sprachen (z. B. Hong's Prolog, [Hon04], und Ask-Elle, [Ger+16]), aber zum Beispiel mit Java-Bugs [SS08], einer Bibliothek von häufigen Programmierfehlern beziehungsweise Misskonzepten, auch für objektorientierte Sprachen. Im zweiten Fall beschränkt sich die Analyse auf das Finden von fehlerhaftem Code. Zum Beispiel untersucht JACK (vgl. Kapitel 9) Programmtraces von Lernenden und zeigt Ausschnitte, in denen von der korrekten Lösung abgewichen wird. Einen datengetriebenen Ansatz verwenden zum Beispiel FIT Java Tutor [GP15] und ITAP [RK15], die eine Lösung mit existierenden Lösungen aus einem Lösungsraum vergleichen und diese zur Hilfe (eventuell in Auszügen) bereitstellen oder auf konkrete Unterschiede hinweisen.

Layout-Feedback Dieses Feedback soll Lernenden helfen, Programme zu erstellen, die gewissen Coding-Conventions entsprechen. Ziel dabei ist, dass die Lernenden den Quellcode leichter verstehen und so zum Beispiel auch Fehler einfacher finden können. Dies kann u. a. durch Style-Checker, Codemetriken oder auch durch Layoutvergleiche mit einer Musterlösung (VC Prolog, [Pey+00]) erfolgen, wird aber insgesamt nur von wenigen Systemen angeboten.

Qualitätsfeedback Zur Messung der Qualität werden Metriken des Software-Engineering eingesetzt. Gemessen wird dabei nicht, ob ein Algorithmus korrekt ist, sondern wie effizient er hinsichtlich Zeit beziehungsweise Speicher implementiert wurde. Dazu zählt ebenfalls ELP [Tru07], wo Hinweise für schlechte Programmierpraktiken (wie „if (a==true)“ anstatt von „if (a)“)

gegeben werden, oder JACK (vgl. Kapitel 9), wo Java-Programmtraces mit Musterlösungen verglichen werden, um unnötig komplexe Programme zu identifizieren.

2.2.8 Betrachtung von formativen Feedbackansätzen

Das Review von [KJH16] fokussiert auf die systematische Betrachtung von formativen Feedbackansätzen und klassifiziert Learning- als auch Assessment-Tools hinsichtlich unterstützter Feedbacktypen. Die Feedbacktypen orientieren sich an von [Nar08] aufgestellten Kategorien.

- Kenntnis des Erfolgs für eine Menge von Aufgaben (z. B. summatives Feedback: 85 % korrekt)
- Kenntnis des Ergebnisses bzw. der Antworten (z. B. erfüllt alle Testfälle/Constraints, Ausgabe entspricht der Musterlösung)
- Kenntnis eines korrekten Ergebnisses (z. B. Darstellung einer Musterlösung)
- Kenntnis über Aufgabenbeschränkungen (z. B. Hinweise zu Anforderungen: for-Schleife muss genutzt werden)
- Kenntnis über Konzepte (z. B. weitergehende Erklärungen oder Beispiele zur Thematik)
- Kenntnis über Fehler (z. B. Testfehlschläge, Syntaxfehler, Laufzeit- bzw. logische Fehler, Styleverletzungen, Performanzprobleme). Oftmals werden hierfür professionelle Testtools eingesetzt. Feedback kann sehr einfach (z. B. Anzahl von Fehlern, Note, Prozentsatz, einfacher Hinweis) oder ausführlicher mit Beschreibungen der Fehler erfolgen.
- Kenntnis darüber, wie fortzufahren ist (z. B. Hinweise, wie ein Fehler behoben werden kann oder wie die aktuelle Lösung ergänzt werden sollte). Feedback kann in Form von Hinweisen (z. B. Vorschlag, Fragen, Beispiel), einer Lösung, die direkt anzeigt, was noch zu erledigen ist, oder als Kombinationen davon erfolgen.
- Kenntnis über Metakognition (z. B. Überprüfung, ob ein Lernender weiß, warum eine Lösung korrekt ist)

In etwas mehr als der Hälfte (60 %) der Systeme wird nur ein einzelner der oben genannten Feedbacktypen angeboten. [KJH16] stellen fest, dass das bereitgestellte Feedback im Allgemeinen nicht sehr divers ist, sondern hauptsächlich auf die Identifikation von Fehlern abzielt. Feedback des Typs „Kenntnis über Fehler“ findet sich in allen bis auf einem System innerhalb des Reviews: Hinweise auf Testfehlschläge bieten 75 %, Laufzeit- beziehungsweise logische Fehler 42 %, Syntaxprüfungen 36 %, Style- 17 % und Performanzanalysen 9 % der Systeme des Reviews. Feedback hinsichtlich „Kenntnis darüber, wie fortzufahren ist“ beinhalten 32 % der Tools, wobei 26 % Hinweise zur Fehlerbehebung und 14 % zum weiteren Vorgehen zur Lösung der Aufgabe geben. Feedback der Typen „Kenntnis über Aufgabenbeschränkungen“ wird durch 16 %, „Kenntnis über Konzepte“ durch 12 % und „Kenntnis über Meta-Kognition“ durch lediglich 1 % der Systeme ausgegeben. Als mögliche Erklärung geben [KJH16] an, dass viele Systeme als reine Bewertungssysteme für eine große Anzahl von Lernenden konzipiert und entwickelt wurden und somit der Fokus auf der automatischen Bewertung und nicht auf detailliertem Feedback liege. Trotzdem haben viele Autoren angegeben, dass ein Ziel der Systeme die Verbesserung des Lernens sei, wozu wiederum einfache Listen mit Fehlern nach Meinung der Autoren des Reviews nicht ausreichend seien.

Bei den unterstützten Programmiersprachen ist ein starker Fokus auf imperative beziehungsweise objektorientierte Sprachen festzustellen, wobei neuere Tools vor allem Support für Java, C und C++ bereitstellen.

Neben diesen Feedbackkategorien werden die Systeme hinsichtlich ihrer „ill-definedness“ der höchsten möglichen Aufgabentypen gruppiert. Dabei wurde auf die Klassifikation von [LP14] zurückgegriffen. Aufgaben der Klasse 1 haben eine einzige korrekte Lösung und sind oft Multiple-Choice Fragen oder Programmvorlagen, in die Codeteile eingefügt werden müssen. Aufgaben der Klasse 2 können über verschiedene Implementierungen gelöst werden – wobei oftmals eine grobe Vorlage (z. B. Schnittstelle) oder eine Beschreibung, die auf die Lösungsstrategie hindeutet, gegeben wird. Die dritte Klasse ist die höchste von [KJH16] betrachtete und beinhaltet Aufgaben, die sowohl durch unterschiedliche Strategien als auch durch Algorithmen gelöst werden können. 20 % der Systeme unterstützten maximal Aufgaben der Klasse 2 und die restlichen 80 % maximal Aufgaben der Klasse 3.

Daneben werden sowohl Techniken für die Generierung von Feedback, Möglichkeiten für Lehrende die Tools für ihre Bedürfnisse anzupassen (auch das Einstellen von Übungsaufgaben zählt dazu) als auch die Qualität und Effektivität der Tools dargestellt. Bemerkenswert ist dabei, dass durch viele Tools dynamische und statische Analysetechniken eingesetzt werden. Teilweise werden auch kom-

plexere Techniken eingesetzt, wie zum Beispiel Model-Tracing, constraint-based modelling, dynamisches Testen mit Reflections, statische Codeanalysen oder Intentionserkennung. Jedoch erschweren diese komplexeren Techniken das Hinzufügen neuer Aufgaben und das Vornehmen von Anpassungen am System – publizierte Angaben für die Dauer zur Erstellung einer Aufgabe variieren zwischen einigen Stunden und einer Personenwoche.

2.3 Zusammenfassung und Diskussion

In diesem Kapitel wurden acht verschiedene Sichtweisen und Klassifikationsansätze für Programmierlernsysteme beziehungsweise automatische Bewertungssysteme vorgestellt. Dabei wurden besondere Ansätze innerhalb dieser Kategorisierungen mit Beispielen hervorgehoben. Insgesamt ist auffällig, dass viele verschiedene Klassifikationsansätze mit teilweise sehr ähnlichen Kategorien vorgenommen wurden, es jedoch (noch) keinen Konsens für die Benennungen gibt (vgl. [KJH16]). Zudem scheint kein Review oder Survey eine umfassende Darstellung der existierenden Systeme und Tools vorzunehmen. Dies äußert sich vor allem darin, dass oftmals die Kriterien für die Aufnahme eines Systems oder Tools in das Review nicht eindeutig nachvollziehbar sind (vgl. [KJH16]) – eine besondere Schwierigkeit besteht sicherlich auch darin, dass einige Systeme nicht auf Englisch publiziert wurden, sondern „lediglich“ auf lokalen Workshops oder Konferenzen, wie zum Beispiel im deutschsprachigen Raum der DeLFI-Tagung.

Generell wird bei den vorgestellten Reviews ausführlich auf grundsätzliche Analyseansätze eingegangen. Speziell widmen sich die neueren Reviews sehr detailliert verschiedenen Feedbackarten. Dennoch gibt es keine umfassenden Reviews oder Übersichten, die sich genauer mit den Features der Systeme beschäftigen. Insbesondere könnten solche eine gute Grundlage für die Auswahl eines passenden existierenden Systems für Praktiker bieten, die ein Unterstützungssystem neu einführen möchten. Dabei sollte darüber hinaus auch die Verfügbarkeit und Lizenzierung der Systeme betrachtet werden. Zudem gibt es keine umfassenden Reviews mit Fokus auf technische Aspekte, wie zum Beispiel Architekturen und konkrete technische Lösungsansätze, die für Entwickler Aufschluss über bewährte Ansätze beziehungsweise Möglichkeiten zur Wiederverwendung existierender Systeme geben könnten. Ein Grund dafür ist wahrscheinlich, dass in vielen Publikationen technische Details oftmals nur relativ knapp am Rande oder ausschließlich innovative Features der Systeme erwähnt werden (vgl. [Iha+10]). Dennoch scheint sich ein Trend zu webbasierten Systemen zu entwickeln, die mindestens eine Anbindung an ein LMS bieten. Aber auch bei Lösungen zur sicheren Ausfüh-

rung von Lernerlösungen finden sich nur selten genauere Angaben. Insbesondere bei älteren Systemen oder Prototypen, die später in den Produktivbetrieb gewechselt sind, wurde oftmals auf Sicherheit kein großer Wert gelegt (vgl. [Iha+10]). [For06] gibt einen guten Einstieg in grundsätzliche Probleme und Lösungsmöglichkeiten.

Die unterstützten Programmiersprachen sind sehr divers, wobei von den meisten Systemen imperative oder objektorientierte Sprachen wie C/C++ oder Java angeboten werden. Hierbei ist besonders erwähnenswert, dass es hinsichtlich der Feedbacktypen und Analysetechniken eine starke Abhängigkeit von speziellen Sprachen zu geben scheint (vgl. [Le16]). Beispielsweise findet sich sehr detailliertes Feedback mit Intentionserkennung hauptsächlich für logische oder funktionale Programmiersprachen.

Die Mehrzahl der Systeme scheint sich auf das Bewerten von klassischen Programmieraufgaben zu fokussieren. Das automatische Bewerten von Aufgaben mit grafischen Oberflächen, „webbasierten Sprachen“ oder Android Apps [Hei+15] sind Ausnahmen. Speziell für die Bewertung von Webapplikationen reicht es in der Regel nicht, einen solchen Interpreter lokal auszuführen, sondern mit einer auf einem Webserver bereitgestellten Version zu interagieren. Ebenso sind Systeme sehr selten, die Kollaboration über Peer-Reviews hinaus unterstützen oder fördern. Eine nennenswerte Ausnahme ist das Quiver-System [EFK04], in dem verschiedene Kurse auf unterschiedlichen Erfahrungsstufen zusammenarbeiten.

Hinsichtlich automatisierter Bewertung wird von [AM05] und [Iha+10] hervorgehoben, dass sich nicht alle Programmieraufgaben vollständig automatisch bewerten lassen: [AM05] spricht dabei vom Vorteil einer ganzheitlichen Bewertung von Aufgaben durch Lehrende – so dass zum Beispiel teilweise Punkte vergeben werden können, obwohl einzelne Tests eventuell aufgrund von Kleinigkeiten fehlgeschlagen sind (vgl. [SOP11]). In diesem Zusammenhang beklagen [Iha+10], dass nur selten berichtet wird, wie mit solchen Aufgaben umgegangen wird beziehungsweise werden sollte.

Bereits bei [AM05] wurde festgestellt, dass viele Systeme unabhängig voneinander entwickelt wurden und es keine gemeinsamen Standards oder Schnittstellen gibt. Ein Grund dafür ist sicherlich, dass es sich um ein aktuelles Forschungsfeld handelt und immer wieder innovative, neue Ansätze entwickelt werden. Grundsätzlich gibt es daran auch nichts auszusetzen, dass verschiedene Systeme mit unterschiedlichen Schwerpunkten und Ansätzen existieren – einige Systeme haben auch eine größere Verbreitung gefunden. Aber unter Berücksichtigung der oftmals zeitaufwändigen Entwicklung guter Aufgaben und Analysetechniken wird hier auch viel Potenzial verschenkt: Gute, bewährte Aufgaben, welche nicht nur eine Beschreibung beinhalten, sondern speziell auch anspruchsvolle Feedback-

verfahren, können nicht über Systemgrenzen hinweg ausgetauscht oder eingesetzt werden. Auf der einen Seite gibt es hierfür technische Gründe: unterschiedliche Funktionalitäten von Systemen, die schwer zu überbrücken sind. Auf der anderen Seite gibt es aber auch kein etabliertes gemeinsames Austauschformat für Programmieraufgaben – in Kapitel 24 wird eine Spezifikation vorgeschlagen, die versucht, dieses Problem zu lösen.

Literatur für dieses Kapitel

[AFS84] John R. Anderson, Robert Farrell und Ron Sauers. „Learning to program in LISP“. In: *Cognitive Science* 8.2 (1984), S. 87–129.

[AM05] Kirsti M. Ala-Mutka. „A Survey of Automated Assessment Approaches for Programming Assignments“. In: *Computer Science Education* 15.2 (2005), S. 83–102. DOI: 10.1080/08993400500150747.

[AS86] J. R. Anderson und E. Skwarecki. „The Automated Tutoring of Introductory Computer Programming“. In: *Commun. ACM* 29.9 (Sep. 1986), S. 842–849. DOI: 10.1145/6592.6593.

[BE08] Daniel Bildhauer und Jürgen Ebert. „Querying software abstraction graphs“. In: *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2005), collocated with ICPC 2008* (2008).

[Che04] P. M. Chen. „An automated feedback system for computer organization projects“. In: *IEEE Transactions on Education* 47.2 (Mai 2004), S. 232–240. DOI: 10.1109/TE.2004.825220.

[CR13] Julio C. Caiza und José María del Álamo Ramiro. „Programming assignments automatic grading: review of tools and implementations“. In: *7th International Technology, Education and Development Conference (INTED2013)*. 2013, S. 5691–5700.

[DE88] M. Ducassé und A.-M. Emde. „A Review of Automated Debugging Systems: Knowledge, Strategies and Techniques“. In: *Proceedings of the 10th International Conference on Software Engineering*. ICSE '88. IEEE Computer Society Press, 1988, S. 162–171.

[DLO05] Christopher Douce, David Livingstone und James Orwell. „Automatic Test-based Assessment of Programming: A Review“. In: *ACM Journal of Educational Resources in Computing* 5.3 (2005).

[DM98] Fadi P. Deek und James A. McHugh. „A Survey and Critical Analysis of Tools for Learning Programming“. In: *Computer Science Education* 8.2 (1998), S. 130–178. DOI: 10.1076/csed.8.2.130.3820.

[Edw04] Stephen H. Edwards. „Using Software Testing to Move Students from Trial-and-error to Reflection-in-action“. In: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '04. ACM, 2004, S. 26–30. DOI: 10.1145/971300.971312.

[EFK04] Christopher C. Ellsworth, James B. Fenwick Jr. und Barry L. Kurtz. „The Quiver system“. In: *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. SIGCSE '04. ACM, 2004, S. 205–209. DOI: 10.1145/971300.971374.

[For06] Michal Forišek. „Security of programming contest systems“. In: *Informatics in Secondary Schools, Evolution and Perspectives* (2006).

[GA05] Mercedes Gómez-Albarrán. „The Teaching and Learning of Programming: A Survey of Supporting Software Tools“. In: *The Computer Journal* 48.2 (2005), S. 130–144. DOI: 10.1093/comjnl/bxh080.

[Ger+16] Alex Gerdes u. a. „Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback“. In: *International Journal of Artificial Intelligence in Education* (2016), S. 1–36. DOI: 10.1007/s40593-015-0080-x.

[GP15] Sebastian Gross und Niels Pinkwart. „Towards an Integrative Learning Environment for Java Programming“. In: *IEEE 15th International Conference on Advanced Learning Technologies (ICALT), 2015*. Los Alamitos, CA: IEEE Computer Society Press, Juli 2015, S. 24–28. DOI: 10.1109/ICALT.2015.75.

[Guz04] Mark Guzdial. „Programming environments for novices“. In: *Computer science education research* 2004 (2004), S. 127–154.

[Hei+15] Mathis Heimann u. a. „Automatische Bewertung von Android-Apps“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2015)*. Bd. 1496. CEUR Workshop Proceedings. 2015.

[Hig+03] Colin Higgins u. a. „The CourseMarker CBA System: Improvements over Ceilidh“. In: *Education and Information Technologies* 8 (3, 2003), S. 287–304. DOI: 10.1023/A:1026364126982.

[Hig+05] Colin A. Higgins u. a. „Automated Assessment and Experiences of Teaching Programming“. In: *J. Educ. Resour. Comput.* 5.3 (Sep. 2005). DOI: 10.1145/1163405.1163410.

[Hon04] Jun Hong. „Guided programming and automated error analysis in an intelligent Prolog tutor“. In: *International Journal of Human-Computer Studies* 61.4 (2004), S. 505–534.

[Iha+10] Petri Ihantola u. a. „Review of Recent Systems for Automatic Assessment of Programming Assignments“. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling '10. ACM, 2010, S. 86–93. DOI: 10.1145/1930464.1930480.

[JGB05] Mike Joy, Nathan Griffiths und Russell Boyatt. „The boss online submission and assessment system“. In: *J. Educ. Resour. Comput.* 5.3 (Sep. 2005). DOI: 10.1145/1163405.1163407.

[Kay+94] David G. Kay u. a. „Automated grading assistance for student programs“. In: *Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education, 1994, Phoenix, Arizona, USA, March 10-12, 1994*. 1994, S. 381–382. DOI: 10.1145/191029.191184.

[KJH16] Hieke Keuning, Johan Jeuring und Bastiaan Heeren. *Towards a Systematic Review of Automated Feedback Generation for Programming Exercises – Extended Version*. Techn. Ber. UU-CS-2016-001. Department of Information und Computing Sciences, Utrecht University, März 2016. URL: http://www.cs.uu.nl/research/techreps/repo/CS-2016/2016-001.pdf.

[KP05a] Caitlin Kelleher und Randy Pausch. „Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers“. In: *ACM Computing Surveys (CSUR)* 37.2 (2005), S. 83–137.

[Kö+03] Michael Kölling u. a. „The BlueJ System and its Pedagogy“. In: *Computer Science Education* 13.4 (2003), S. 249–268. DOI: 10.1076/csed.13.4.249.17496.

[Le16] Nguyen-Thinh Le. „A Classification of Adaptive Feedback in Educational Systems for Programming“. In: *Systems* 4.2 (2016). DOI: 10.3390/systems4020022.

[LLP13] Nguyen-Tinh Le, Frank Loll und Niels Pinkwart. „Operationalizing the Continuum between Well-Defined and Ill-Defined Problems for Educational Technology“. In: *IEEE Transactions on Learning Technologies* 6.3 (2013), S. 258–270. DOI: 10.1109/TLT.2013.16.

[LMM98] José Paulo Leal, Nelma Moreira und Leal Nelma Moreira. *Automatic Grading of Programming Exercises*. Techn. Ber. Kurnia, A.; Cheang, B., Lim, A „Online Judge“, Computers und Education, 1998.

[LP14] Nguyen-Thinh Le und Niels Pinkwart. „Towards a classification for programming exercises“. In: *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science*. 2014.

[Mat94] Urs von Matt. „Kassandra: The Automatic Grading System“. In: *SIGCUE Outlook* 22.1 (Jan. 1994), S. 26–40. DOI: 10.1145/182107.182101.

[Nar08] Susanne Narciss. „Feedback strategies for interactive learning tasks“. In: *Handbook of research on educational communications and technology*. Hrsg. von David Jonassen u. a. Bd. 3. Routledge, 2008, S. 125–144.

[Nik+04] Jussi Nikander u. a. „Visual algorithm simulation exercise system with automatic assessment: TRAKLA2“. In: *Informatics in Education – An International Journal* Vol. 3_2 (2004), S. 267–288.

[Pea+07] Arnold Pears u. a. „A Survey of Literature on the Teaching of Introductory Programming“. In: *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '07. ACM, 2007, S. 204–223. DOI: 10.1145/1345443.1345441.

[Pey+00] Christoph Peylo u. a. „A Web-based intelligent educational system for PROLOG“. In: *Proceedings of the International Workshop on Adaptive and Intelligent Web-Based Education Systems held in conjunction with ITS 2000 Montreal, Canada.* 2000, S. 85–96.

[Pil03] Nelishia Pillay. „Developing Intelligent Programming Tutors for Novice Programmers“. In: *ACM SIGCSE Bulletin* 35.2 (Juni 2003), S. 78–82. DOI: 10.1145/782941.782986.

[Rin99] M. Rintala. *Tutnew memory management library*. Englisch. 1999. URL: https://www.cs.tut.fi/~bitti/tutnew-www/english/.

[RK15] Kelly Rivers und Kenneth R. Koedinger. „Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor“. In: *International Journal of Artificial Intelligence in Education* (2015), S. 1–28. DOI: 10.1007/s40593-015-0070-z.

[SG14] Michael Striewe und Michael Goedicke. „A Review of Static Analysis Approaches for Programming Exercises“. In: *Computer Assisted Assessment. Research into EAssessment: International Conference, CAA 2014, Zeist, The Netherlands, June 30 – July 1, 2014. Proceedings*. Springer International Publishing, 2014, S. 100–113. DOI: 10.1007/978-3-319-08657-6_10.

[SKM13] Juha Sorva, Ville Karavirta und Lauri Malmi. „A Review of Generic Program Visualization Systems for Introductory Programming Education“. In: *Trans. Comput. Educ.* 13.4 (Nov. 2013), 15:1–15:64. DOI: 10.1145/2490822.

[SOP11] Sven Strickroth, Hannes Olivier und Niels Pinkwart. „Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben?“ In: *DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik*. Bd. 188. LNI. GI, 2011, S. 115–126.

[SQF08] Mate' Sztipanovits, Kai Qian und Xiang Fu. „The Automated Web Application Testing (AWAT) System“. In: *Proceedings of the 46th Annual Southeast Regional Conference on XX*. ACM-SE 46. ACM, 2008, S. 88–93. DOI: 10.1145/1593105.1593128.

[SS08] Merlin Suarez und Raymund Sison. „Automatic Construction of a Bug Library for Object-Oriented Novice Java Programmer Errors“. In: *Intelligent Tutoring Systems: 9th International Conference, ITS 2008, Montreal, Canada, June 23-27, 2008 Proceedings*. Hrsg. von Beverley P. Woolf u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 184–193. DOI: 10.1007/978-3-540-69132-7_23.

[Tru07] Nghi Truong. *A Web-Based Programming Environment for Novice Programmers*. 2007. URL: http://eprints.qut.edu.au/16471/1/Nghi_Truong_Thesis.pdf.

[Val04] David W. Valentine. „CS educational research: a meta-analysis of SIGCSE technical symposium proceedings“. In: *ACM SIGCSE Bulletin* 36.1 (2004), S. 255–259.

Teil I

Didaktische Einsatzszenarien

3 Automatisierte Bewertung in der objektorientierten Programmierausbildung am Beispiel von Java

Michael Striewe und Robert Garmann

Zusammenfassung
Das folgende Kapitel stellt verschiedene Aufgabentypen sowie ein Veranstaltungskonzept für die Einführung in die objektorientierte Programmierung in Java vor. Es basiert auf Erfahrungen an der Universität Duisburg-Essen sowie der Hochschule Hannover, an denen die Einführung in die objektorientierte Programmierung seit mehreren Jahren durch automatische Bewertungssysteme unterstützt wird.

3.1 Einleitung

Das Erlernen der objektorientierten Programmierung stellt Lernende vor mehrere Herausforderungen: Zum einen müssen die verschiedenen Aspekte der Objektorientierung verstanden und verinnerlicht werden und zum anderen muss eine konkrete Programmiersprache erlernt werden, in der diese Konzepte auf eine bestimmte Weise umgesetzt werden. Programmieranfänger, die zuvor keine andere Sprache erlernt haben, müssen zudem grundlegende Programmierfähigkeiten unabhängig von der Objektorientierung erlernen.

Aufgrund dieser Herausforderungen ist es sinnvoll, in der Lehre zur objektorientierten Programmierung im großen Umfang Übungsaufgaben einzusetzen, in denen verschiedene Fähigkeiten in unterschiedlichen Situationen angewandt werden müssen. Die Aufgaben sollten detailliertes Feedback vorsehen, um Fehler auf den verschiedenen Ebenen der Umsetzung zielgerichtet kommentieren zu können: Eine funktional korrekte Lösung kann im Sinne der Aufgabenstellung falsch sein, wenn sie beispielsweise nicht die gewünschte Vererbungsstruktur umsetzt.

Umgekehrt ist eine strukturell korrekte Lösung nicht zwangsläufig auch fehlerfrei lauffähig.

Daraus ergeben sich sowohl Chancen als auch Herausforderungen, denn große Mengen an Lösungen lassen sich in der Regel nur automatisiert zeitnah bewerten und mit Feedback versehen [Iha+10]. Der didaktische Wert hängt jedoch stark von der Qualität des Feedbacks ab. Im folgenden wird auf Basis des Einsatzes von automatischen Systemen zur Programmbewertung an der Universität Duisburg-Essen sowie der Hochschule Hannover ein umfangreiches Beispielszenario beschrieben. Es basiert auf dem Einsatz der Systeme JACK an der Universität Duisburg-Essen sowie GRAJA an der Hochschule Hannover; jeweils in einer Einführungsvorlesung zur objektorientierten Programmierung mit Java.

Java wird ungeachtet anhaltender Diskussionen über dessen didaktische Eignung (s. bspw. [Bös98], [Pea+07]) in vielen Hochschulen als erste Programmiersprache gelehrt. Java vereint Elemente verschiedener Programmierparadigmen, u. a. die der strukturierten, prozeduralen und objektorientierten Programmierung. Das in diesem Kapitel beschriebene didaktische Szenario geht davon aus, dass Studierende Konzepte der strukturierten und der prozeduralen vor denen der objektorientierten Programmierung erlernen (s. bspw. [Reg06]), setzt diese Reihenfolge jedoch nicht voraus. Die von uns vorgestellten Aufgabenformen und Einsatzszenarien sind ebenso bei einer Einführung nach einem Objects-first-Ansatz oder gar einem Components-first-Ansatz anwendbar (s. bspw. [Gri08], [HTW04]).

Obwohl sich selbstverständlich in allen Schritten des Szenarios eine Lösung manuell prüfen und einzeln mit verschiedenen Testfällen ausführen lässt, werden primär spezifische Vorteile der automatischen Bewertung genannt. Mit „Bewertung“ ist dabei jede Art von Analyse und Generierung von Feedback gemeint, die durch ein System durchgeführt wird, unabhängig davon, ob dies im Kontext formativen oder summativen Assessments erfolgt.

3.2 Aufgabenformen

Dieser Abschnitt stellt verschiedene Aufgabenformen vor, die im Kontext der objektorientierten Programmierung mit Java anwendbar sind. Es werden die jeweiligen Vorteile der automatischen Bewertung genannt, ohne auf die Einbettung der Aufgaben in den organisatorischen Kontext einer Veranstaltung einzugehen. Diese erfolgt im darauf folgenden Abschnitt.

Grundsätzlich gilt für alle Aufgabenformen, dass ein automatisches Bewertungssystem einige grundlegende Eigenschaften der Lösung wie Klassenbezeichner oder Methodensignaturen kennen muss, um ein Programm automatisch bewer-

ten zu können. Aufgabenautoren müssen daher bei der Aufgabenerstellung einige Entwurfsentscheidungen vorweg nehmen, um Funktionstests an internen Schnittstellen des zu erstellenden Programms zu ermöglichen. Studierende müssen sich daher bei der Lösung üblicherweise an einige Vorgaben halten. Diese verringern die Freiheitsgrade bei der Programmierung, was eher untypisch für die professionelle Programmierung ist. Hier müssen also inhaltliche Lernziele gegen die Vorteile einer automatischen Bewertung abgewogen werden. Bei Aufgaben, die auf einzelne, isoliert zu lernende Programmierkonzepte bzw. Algorithmen fokussieren, nimmt man Einschränkungen in der Entwurfsfreiheit in der Regel zugunsten umfangreicherer Bewertungsmöglichkeiten in Kauf.

3.2.1 Einführende Übungsaufgaben

Ein klassischer Einstieg in die objektorientierte Programmierung mit Java ist es, schrittweise nur ausgewählte Teile des Sprachumfangs detailliert zu betrachten. So kann beispielsweise der Fokus einer ersten Lektion auf primitiven Datentypen und einfachen Operationen liegen, während das zwingend notwendige Erstellen einer `main`-Methode ohne weitere Erläuterung als gegeben hingenommen wird und möglicherweise auftretende Exceptions erst einmal ignoriert werden.

Für diesen Einstieg kann die automatische Bewertung verschiedene didaktische Mehrwerte erzeugen: Erstens können Lösungen durch ein Bewertungssystem ausgeführt werden, auch wenn sie keinen vollständigen Programmcode darstellen, da das System die notwendigen fehlenden Teile automatisch ergänzen kann. Erste Programmieraufgaben können somit als Lückentexte mit Lücken beliebiger Größe gestellt werden, in denen wahlweise einzelne Ausdrücke und Statements oder ganze Methodenkörper ergänzt werden müssen. Die Lernenden können dazu detailliertes Feedback erhalten, wie sich ihr Code für verschiedene Eingaben verhält, ohne jedoch tatsächlich vollständigen Code schreiben zu müssen. Zweitens können auftretende Probleme wie etwa Exceptions automatisch behandelt und in Rückmeldungen umgewandelt werden, die dem Niveau der Lernenden entsprechen. Gleiches gilt insbesondere auch für Meldungen des Compilers, die in ihrer Rohform nicht immer für Anfänger geeignet sind. Mit etwas mehr Aufwand können Aufgabenautoren zudem vorgeben, wie exakt die Ausgabe einer Lösung mit der Vorgabe übereinstimmen muss. So kann zum Beispiel zwischen grundsätzlich fehlerhaften Ausgaben und einem vergessenen Leerzeichen unterschieden und die Rückmeldung an das Niveau bzw. die Ansprüche der Lernenden angepasst werden. Abbildung 3.1 zeigt eine Aufgabe mit vorgegebenem Coderahmen. Die ebenfalls dargestellte Musterlösung wird selbstverständlich nicht an Studierende

ausgegeben. Die von der Musterlösung abweichende Lösung zeigt, dass exakte Division und Ausgabeformatierung korrekt eingesetzt werden, auch wenn die ausgegebenen Zeichenketten aufgrund der Abkürzung von „Stunden“ zu „Std.“ voneinander abweichen. Mit einem geeigneten Bewertungssystem kann auch die abweichende Lösung als korrekt erkannt werden.

Alternativ oder konsekutiv folgend zum klassischen Einstieg muss die Lehre der objektorientierten Programmierung ferner die Konzeption von Klassen und Objekten vermitteln. Hierbei steht weniger die funktionale Korrektheit eines Programms im Vordergrund, sondern dessen struktureller Aufbau.

Trotz dieser scheinbar kleinen Änderung des Fokus eröffnen automatische Bewertungssysteme hier eine sehr große Bandbreite an didaktischen Möglichkeiten: Die reine Testautomatisierung kann alleine bereits große und komplexe Objektstrukturen mit einer geforderten Lösung vergleichen sowie systematisch Getter und Setter von Objekten auf funktionale Korrektheit überprüfen. Mit geeigneten Systemen können eventuelle Fehler aber nicht nur textuell, sondern auch grafisch ausgegeben werden, indem den Lernenden jeweils individuell die von ihrer Lösung erzeugte Objektstruktur angezeigt wird. Auf diese Weise wird vom Bewertungssystem eine Brücke geschlagen zurück zur objektorientierten Modellierung, die häufig parallel zur Programmierung gelehrt wird. Die Lernenden erhalten so die Möglichkeit zum unmittelbaren Vergleich zwischen dem geplanten Systemdesign und ihrer tatsächlichen Umsetzung. Ein Beispiel für eine solche Visualisierung wird in Kapitel 9 diskutiert.

Ferner können statische Prüfverfahren angewandt werden, mit denen unabhängig von der Ausführung der Lösungen geprüft werden kann, ob beispielsweise Vorgaben bezüglich der Vererbungsstruktur eingehalten wurden. So kann insbesondere auch die strukturelle Korrektheit einer Lösung mit (positivem) Feedback

```
public class Beispiel {
  public static void main(String[] args) {
    int sekunden= Integer.parseInt(args[0]);
    // Aufgabe: rechnen Sie die Sekunden in Stunden um
    // und geben Sie diese auf der Console aus.
    // Etwa so: "2,5 Stunden"
    // ...
  }
}
```

Musterlösung:

```
System.out.format("%.1f Stunden%n", sekunden/3600.0);
```

Abweichende, dennoch korrekte Lösung:

```
System.out.format("%.1f Std.%n", (double)sekunden/3600);
```

Abbildung 3.1: Beispiel einer einführenden Übungsaufgabe

versehen werden, selbst wenn die Lösung insgesamt funktional fehlerhaft oder sogar aufgrund von syntaktischen Fehlern gar nicht lauffähig ist. In diesem Fall ist es mit geeigneten Systemen zudem möglich, für die Durchführung von Tests Teile der Lösungen der Lernenden – beispielsweise eine fehlerhaft oder unvollständig implementierte Klasse – automatisiert durch eine gültige Musterlösung zu ersetzen, um so den Rest der Lösung bewerten zu können.

Über das Feedback zu funktionalen und strukturellen Eigenschaften der Lösung hinaus ist es möglich, Feedback zur Programm-*Qualität* zu geben. Während Qualitätseigenschaften wie Benutzbarkeit oder Effizienz in der Regel erst in fortgeschrittenen Übungsaufgaben oder Projektaufgaben (s. u.) relevant werden, sollten bereits Programmieranfänger auf gut lesbaren und konventionsgemäß erstellten Programmcode achten. Statische Prüfverfahren können Qualitätsmängel bzgl. der Wartbarkeit der Lösung feststellen und beispielsweise fehlende Kommentare oder Verstöße gegen Bezeichnerkonventionen monieren (s. Beispiel in Abbildung 3.2). Zeitnahes automatisches Feedback kann dafür sorgen, dass sich nachteilige Codierpraktiken gar nicht erst einschleifen. Weiche Kriterien wie beispielsweise die *sprechende* Benennung von Bezeichnern oder die *sinnvolle* Kommentierung von Methoden können mit heutigen Werkzeugen jedoch nicht automatisch geprüft werden. Solche von automatischen Bewertungssystemen offen gelassene Lücken im Feedback können und müssen derzeit noch von Menschen geschlossen werden.

Lösung:
```
double Stunden= sekunden/3600.0;
System.out.format("%.1f Stunden%n", Stunden);
```

Feedback:
Variables should start with a lowercase character, 'Stunden' starts with uppercase character.
```
double Stunden= sekunden/3600.0;
```

Abbildung 3.2: Lösung mit Qualitätsmängeln

3.2.2 Fortgeschrittene Übungsaufgaben

Wenn die Grundkonzepte der Objektorientierung und die Grundlagen der Programmierung bekannt sind, spielen in der Lehre oft algorithmische Details des korrekten Umgangs mit objektorientierten Datenstrukturen eine wichtige Rolle. Typische Aufgaben sind dazu beispielsweise das korrekte Iterieren über alle Elemente einer Liste oder ein rekursiver Durchlauf durch eine Baumstruktur.

Auch hier können Bewertungssysteme durch die Kombination statischer und dynamischer Prüfungen helfen. Zunächst einmal kann durch statische Verfahren geprüft werden, ob Vorgaben bezüglich des Kontrollflusses tatsächlich erfüllt sind. So kann beispielsweise darauf hingewiesen werden, dass eine iterative Lösung Schleifen enthalten muss, während eine rekursive Lösung genau dies nicht sollte. Es können aber auch detaillierte Hinweise gegeben werden, wenn die falsche Methode rekursiv gestaltet wird oder eine verschachtelte Schleife zum Einsatz kommt, obwohl auch eine einfache Schleife reicht. Durch eine reine Testautomatisierung, die lediglich die Ergebnisse einer Lösung vergleicht, werden solche Fehler nicht erkannt.

Trotzdem hat eine dynamische Prüfung weitere Vorteile. Neben Aussagen zur funktionalen Korrektheit können durch die Ausführung einer Lösung mit verschiedenen Eingabewerten insbesondere auch Aussagen zur Performanz gewonnen werden, so dass auch dieser Qualitätsaspekt berücksichtigt werden kann. Zudem können bei der Ausführung der Lösung Traces des Programmablaufs aufgezeichnet werden, mit denen der Kontrollfluss visualisiert werden kann. Wie im vorherigen Szenario bietet dies den Lernenden die Möglichkeit, das tatsächliche Programmverhalten unmittelbar mit der Planung zu vergleichen. Bewertungssysteme können so die Rolle eines Debuggers übernehmen, ohne dass sich die Lernenden tatsächlich im Detail in die Bedienung eines solchen Werkzeugs einarbeiten müssen. Ein Beispiel für einen aufgezeichneten Trace wird in Kapitel 9.3.2 gezeigt.

Nicht unüblich ist es, in einführenden Übungsaufgaben einige Testfälle im Aufgabentext zu verraten, während die Studierenden mit weiteren Testfällen nur durch das daraus resultierende Feedback konfrontiert werden. Je fortgeschrittener die Übungsaufgaben werden, desto mehr sollten die Studierenden gefordert sein, sich selbst möglichst umfassende Testfälle auszudenken. Fehlschlagende Testfälle der automatischen Bewertung geben dabei Hinweise, welche Sonderfälle die Lösung nicht abdeckt. Manche Bewertungssysteme erlauben es sogar, von Lernenden vorbereitete Testfälle als Teil der dynamischen Prüfung auszuführen und dabei den erreichten Abdeckungsgrad zu bewerten [Edw03].

3.2.3 Projektaufgaben

In den bisherigen Übungsaufgaben spielten die Interaktion eines Programms mit dem Benutzer sowie allgemein die Frage von Ein- und Ausgabe keine Rolle. Üblicherweise werden solche Aufgaben daher über Methodenrückgaben oder Konsolenausgaben abgewickelt. In der fortgeschrittenen Lehre ist es aber relevant und

für die Lernenden oft motivierend, kleine Anwendungen zu entwickeln, die über eine grafische Oberfläche (GUI) verfügen und größere Datenmengen verarbeiten können.

Dies stellt Bewertungssysteme vor die Herausforderung, Nutzerinteraktion zu simulieren, um eine Lösung zu prüfen. Als mögliche Fehlerquelle neben strukturellen oder funktionalen Fehlern kommen insbesondere Fehler in der Ausgabe hinzu. Diese können eine Testdurchführung verhindern, wenn beispielsweise ein Button nicht so platziert oder beschriftet ist, wie ihn die Testautomatisierung erwartet. Obwohl dies zu unerwartet schlechten Testergebnissen führen kann, kann es auch genutzt werden um zu motivieren, warum eine exakte Erfüllung der Spezifikation einer (Benutzer-)Schnittstelle notwendig ist.

Lösungen, die größere Datenmengen verarbeiten sollen, produzieren oder konsumieren diese üblicherweise über Dateischnittstellen. Dies stellt das Bewertungssystem vor die Aufgabe, technische Randbedingungen wie Zeichenkodierungen und begrenzten Speicherplatz in didaktisch geeigneter Form an die Lernenden heran zu tragen. Zeichenkodierungsfehler oder Überschreitungen des zur Verfügung stehenden Platzes auf einem Speichermedium führen zu unverständlichen Fehlermeldungen, die das Bewertungssystem verständlich aufbereiten kann.

Gleichzeitig bieten Bewertungssysteme durch die Kombination vieler verschiedener Verfahren die Möglichkeit, eine Lösung auf allen Ebenen zu testen, so dass über die Gestaltung und Implementierung des GUI bzw. der Datenverarbeitung die zuvor erlernten Aspekte der funktionalen und strukturellen Korrektheit einer Lösung nicht vergessen werden. Dabei können auch wieder Qualitätsaspekte aufgegriffen werden. Neben Codequalität und Performanz sind beispielsweise die gute Analysierbarkeit des Codes oder die sinnvolle Aufteilung der Funktionalität in mehrere Methoden relevante Facetten dieses Themas. Bewertungssysteme können durch die Anwendung klassischer und objektorientierter Codemetriken (s. etwa [SSB10]) beispielsweise unnötig aufgeblähten und damit schlecht analysierbaren Code identifizieren oder einen aus Wartungssicht nachteiligen prozeduralen bzw. objektorientierten Entwurf diagnostizieren. Weiterhin können sie Programmierpraktiken aufdecken, die zu mangelnder Robustheit führen können, z. B. einen leeren `catch`-Block oder ein `return`-Statement in einem `finally`-Block. Anstatt Programmieraufgaben ausschließlich hierzu zu stellen, können Qualitätsaspekte durch eine schrittweise strengere Prüfung im Verlaufe allgemeiner Programmieraufgaben eingeführt werden, wozu sich Projektaufgaben insbesondere eignen.

3.3 Einsatzszenario

Dieser Abschnitt betrachtet nun ein konkretes Einsatzszenario, in dem die bisher vorgestellten Aufgabentypen und ihre automatische Bewertung in den Kontext einer Lehrveranstaltung eingebettet werden. Diese Veranstaltung ist als Einführung in die objektorientierte Programmierung im Kontext einer Hochschule ausgelegt. Der Vorlesungsumfang beträgt 2 oder 4 SWS[1] Vorlesungszeit und 2 SWS Übung, die im Semesterverlauf durch freie Tutorien und verpflichtende Testate ergänzt werden.

3.3.1 Vorlesung

Die Vorlesung ist im Wesentlichen durch den Einsatz automatischer Bewertungssysteme für Programmieraufgaben nicht betroffen, sondern wird im klassischen Format mit der Präsentation des Stoffes durch den Dozenten durchgeführt. Zur Auflockerung der Präsentation des Stoffes können einführende Übungsaufgaben eingestreut werden, die von den Studierenden live in der Vorlesung gelöst und sofort besprochen werden können. Dazu ist es hilfreich, dass ein automatisches Prüfungssystem Zusammenfassungen der häufigsten Fehler erstellen kann, so dass auf diese sofort eingegangen werden kann. Zudem kann das Bewertungssystem dem Dozenten Informationen zur Verfügung stellen, die im Feedback für die Studierenden nicht enthalten sind, da sie über den aktuellen Stoff der Vorlesung hinausgehen oder einer Erläuterung bedürfen. Beispielsweise könnte das Bewertungssystem den Dozenten auf eine unnötig komplexe oder aber eine besonders elegante Lösung aufmerksam machen, um diese sofort in der Vorlesung zu zeigen und zu diskutieren. Die rohen Metrikwerte, auf denen eine solche Auswahl basieren kann, wären dagegen für den Autor der Lösung von weit geringerem Wert oder sogar völlig unverständlich.

3.3.2 Übung und Tutorium

Zur selbstständigen Vertiefung des Stoffs werden wöchentlich mehrere zunächst einführende und später fortgeschrittene Übungsaufgaben zur Verfügung gestellt, die von den Studierenden wahlfrei bearbeitet werden können. Das Bewertungssys-

1 Das hier beschriebene Einsatzszenario ist hypothetisch in dem Sinne, dass es Aspekte mehrerer in Essen und Hannover tatsächlich durchgeführter Lehrveranstaltungen kombiniert.

tem übernimmt dabei zunächst die Rolle eines Tutors, der zu jeder eingereichten Lösung zeitnah Feedback geben kann und den Studierenden somit selbstgesteuertes Arbeiten ermöglicht. Die Präsenzübung und das Tutorium dienen somit lediglich als Ergänzung für diejenigen Studierenden, die eine intensivere, individuelle Betreuung benötigen beziehungsweise Beispiele detaillierter erläutert bekommen wollen, als dies im Rahmen der Vorlesung möglich wäre. Die Bearbeitung der Übungsaufgaben ist daher auch freiwillig und ohne Einfluss auf die spätere Note. Zur Vorbereitung auf die regelmäßigen Testate (s. u.) wird zudem im selben Rhythmus eine Projektaufgabe ausgegeben, die verpflichtend zu bearbeiten ist.

Der Fokus der automatischen Bewertung liegt auf der Erzeugung von möglichst umfangreichem Feedback. Eine detaillierte Gewichtung der verschiedenen Fehler für eine differenzierte Punktevergabe ist daher nicht notwendig, solange die Punktzahlen in etwa den Fortschritt bei der Bearbeitung einer Aufgabe widerspiegeln und somit eine motivierende Rolle einnehmen. Allerdings kann auch ein Übermaß an Feedback nachteilige Effekte haben, wenn es zu umfangreich ist, um von Lernenden noch als hilfreich wahrgenommen zu werden. Insbesondere könnte es Lernenden selbst bei Einsatz einer unterschiedlichen Gewichtung verschiedener Bewertungsaspekte schwer fallen, ihre zentralen Fehler zu erkennen, um diese vorrangig zu beheben. Zudem kann eine große Menge negativen Feedbacks einen demotivierenden Effekt haben. Daher kann es sinnvoll sein, wenn ein Bewertungssystem eine solche Priorisierung intern vornimmt und nur die wichtigsten Rückmeldungen ausgibt. Die weiteren Meldungen können dann beispielsweise nur für Tutoren sichtbar sein, die diese im Rahmen ihrer individuellen Betreuung für zusätzliches Feedback nutzen können. Eine vom Bewertungssystem vorgenommene Gruppierung der Meldungen nach zu lernenden Kompetenzen bzw. nach Bewertungsaspekten ist hierbei häufig hilfreicher, als eine sequenzielle Ordnung nach Programmzeilen.

Auch wenn der Einsatz von Bewertungssystemen eine Einsparung beim Korrekturaufwand bedeutet, der die Korrektur massenhafter Übungsaufgaben für große Studierendengruppen überhaupt erst ermöglicht, darf der Aufwand für die Erstellung guter Aufgaben nicht unterschätzt werden. Insbesondere für detailliertes, aufgabenspezifisches Feedback müssen entsprechende Prüfregeln, Testfälle usw. individuell erstellt werden. Daher muss mit einem Zeitaufwand von 1-3 Stunden für eine einfache Übungsaufgabe und noch einmal erheblich mehr Zeit für eine Projektaufgabe mit umfangreichem Feedback gerechnet werden. Ein Erstellungsaufwand von 16 Stunden ist für eine komplexe Projektaufgabe nicht ungewöhnlich. Der Aufwand lohnt sich, da einmal erstellte Aufgaben mehrere Jahre oder in verschiedenen Studiengängen wiederverwendet werden können. Nötige An-

passungen können dabei in der Regel leicht durchgeführt werden, beispielsweise durch das Ändern der Punktegewichtung oder den Austausch von Textbausteinen.

Will man die Anzahl der verpflichtend zu bearbeitenden Übungsaufgaben erhöhen, kann man von Studierenden als Teil der Prüfungsleistung fordern, dass alle oder ein großer Anteil aller Übungsaufgaben semesterbegleitend bearbeitet und automatisiert bewertet werden müssen. Der menschliche Überprüfungsaufwand steigt in diesem Fall, denn in der Regel müssen Tutoren die Einreichungen nach Abgabefrist zumindest auf grobe Fehler des Bewertungssystems hin prüfen, ggf. die erreichte Punktzahl anpassen und Kommentare ergänzen sowie eine geeignete Überprüfung auf Plagiatsfälle und deren Behandlung sicherstellen.

3.3.3 Testat

Zur regelmäßigen, verpflichtenden Kontrolle des Lernfortschritts werden alle zwei Wochen Testate durchgeführt, in denen eine fortgeschrittene Übungsaufgabe unter Prüfungsbedingungen innerhalb von 45 Minuten gelöst werden muss. Die Studierenden bekommen dazu Arbeitsplätze im Rechnerpool zugewiesen, die über eine Entwicklungsumgebung verfügen, über die die Aufgabe vom Bewertungssystem herunter- und hochgeladen werden kann. Weiterer Netzwerkzugriff ist jedoch nicht zugelassen, so dass die Studierenden insbesondere keine Ressourcen aus dem Lernmanagementsystem der Vorlesung oder dem Internet nutzen dürfen und auch während der Bearbeitung keinen Zugriff auf die Ergebnisse der automatischen Bewertung haben.

Die Testate müssen von den Studierenden nicht alle bestanden werden, aber es muss eine Mindestpunktzahl erreicht werden, um die Zulassung zur abschließenden Klausur zu erhalten. Das Bewertungssystem muss daher bei den Testaten sehr viel differenzierter bei der Vergabe von Punktzahlen vorgehen. Eine der ersten Entscheidungen ist dabei, ob die Lauffähigkeit einer Lösung als K.O.-Kriterium verwendet wird. Wie in Abschnitt 3.2 bereits erwähnt, sind Bewertungssysteme grundsätzlich in der Lage, verschiedene Aspekte unabhängig voneinander zu bewerten. Es besteht also keine technische Notwendigkeit, nicht lauffähige Lösungen mit 0 Punkten zu bewerten. Da es üblich ist, in Klausuren Teilpunkte zu vergeben und die Testate der Klausurzulassung dienen, sollte daher nach Möglichkeit ein Bewertungssystem verwendet werden, mit dem Teilpunkte vergeben werden können.

Eine zweite Entscheidung ist, ob nur zwischen verschiedenen Kategorien von Fehlern gewichtet wird, oder ob auch innerhalb der Kategorien Abstufungen vorgenommen werden können. Da ein einzelner tatsächlicher Fehler (z. B. eine falsch

gesetzte Klammer) zu mehreren Fehlermeldungen des Compilers führen kann, kann es sinnvoll sein, das Vorhandensein von Compilerfehlern pauschal zu bewerten. Gleichzeitig kann es sinnvoll sein, verschiedenen Testfällen ein unterschiedliches Gewicht zu geben, um zentrale Fälle stärker in die Bewertung einfließen zu lassen als Randfälle. Auch die Art des Fehlers kann mit in die Bewertung aufgenommen werden, um zwischen einer fehlenden oder völlig falschen Ausgabe und einem leichten Rundungsfehler zu unterscheiden. Auf diese Weise wird sichergestellt, dass eine im Kern richtige Lösung eine hinreichend hohe Punktzahl erhält, auch wenn es zahlreiche Sonderfälle gibt, die nicht korrekt beachtet wurden.

Zu beachten ist außerdem, dass nicht alle Funktionen eines Bewertungssystems für den Einsatz bei der Testat- oder Klausurbewertung geeignet sind. So kann beispielsweise randomisiertes Testen bei Übungsaufgaben sehr nützlich sein, weil Lösungen so bei jedem Versuch mit neuen Eingaben konfrontiert werden. Dies spornt dazu an, eine allgemeingültige Lösung zu gestalten, die alle Randfälle abdeckt, ohne dass der Aufgabenautor für jeden dieser Fälle manuell einen Testfall erzeugen muss. Das gleiche Verfahren kann jedoch in einem Testat oder einer Klausur rechtlich problematisch sein, wenn zwei identische Lösungen unterschiedlich bewertet werden, weil durch die Randomisierung der Testfälle ein Fehler nur in einem der beiden Fälle entdeckt wird.

Gegenüber den Übungsaufgaben steigt bei Testataufgaben noch einmal der Aufwand für die Erstellung, da insbesondere mehr Sorgfalt bei der Punktevergabe nötig ist. Idealerweise werden dazu nicht nur eine, sondern mehrere Musterlösungen oder auch prototypische fehlerhafte Lösungen angefertigt, um später das Systemverhalten für diese Fälle überprüfen zu können. Anschließend müssen wie bei Übungsaufgaben für alle gewählten Prüfverfahren die nötigen Testfälle, Prüfregeln, usw. sowie ihre Gewichtung gemäß des gewählten Punkteschemas und die zugehörigen Rückmeldungen erstellt werden. Erfahrungsgemäß gelingt eine für alle denkbaren Lösungen zutreffende Aufteilung von Teilpunkten auf Bewertungsaspekte erst, nachdem viele Lösungen bewertet wurden und die Ergebnisse nachjustiert werden konnten. Dies macht einen großen Teil des Zeitaufwandes für die Vorbereitung aus. Insgesamt kann daher im Schnitt ein Aufwand von etwa 8 Stunden für die Einrichtung einer Testataufgabe angenommen werden.

Wie bereits weiter oben erwähnt wurde, ist es erforderlich und sinnvoll, die automatisch erzeugten Bewertungen durch Menschen zu überprüfen, sobald sie direkt oder indirekt in die Prüfungsnote eingehen. Die entstehenden Überprüfungsaufwände liegen allerdings erheblich unter den Aufwänden einer vollständig manuellen Korrektur, da korrigierende Eingriffe in das Bewertungsresultat bei sorgfältigem Aufgabendesign erfahrungsgemäß eher selten sind. Der Erstellungsaufwand einer Testataufgabe kann gegen den zu erwartenden menschlichen

Überprüfungsaufwand abgewogen werden. Je häufiger eine Testataufgabe genutzt wird, desto eher lohnt sich die Entwicklung eines alle denkbaren Lösungen abdeckenden, ausgefeilten Punkteschemas.

3.3.4 Klausur

Da durch die regelmäßigen Testate bzw. durch regelmäßige Übungseinreichungen bereits im Verlauf des Semesters sichergestellt ist, dass die Studierenden ausreichende Programmierfähigkeiten nachweisen, um überhaupt zur Teilnahme an der Klausur zugelassen zu werden, kann der Schwerpunkt der Klausur auch auf konzeptionelle Fragen der Programmierausbildung gelegt werden. Die Klausur wird daher idealerweise als Hybridklausur durchgeführt, in der Programmieraufgaben am Rechner gelöst werden, während weitere Aufgaben schriftlich auf Papier oder in einem allgemeinen elektronischen Prüfungssystem bearbeitet werden. Fehlt die entsprechende Infrastruktur für eine E-Prüfung, gehen wir in diesem Szenario von vollständig handschriftlich angefertigten Klausuren aus, die sich einer automatisierten Bewertung entziehen.

3.4 Erfahrungen

3.4.1 Erfahrungen der Lehrenden

Die Entwicklung und der Einsatz des oben beschriebenen Konzepts mit wöchentlichen Übungs- und Testataufgaben war in einer großen Einführungsveranstaltung überhaupt erst möglich, weil automatische Bewertungssysteme zu Hilfe genommen werden konnten. Die einzige Alternative ist der keineswegs kostengünstige Einsatz von Tutoren als Bewertungspersonal. Zwar können Tutoren prinzipiell ein differenziertes Feedback geben, zu bedenken ist dabei allerdings, dass die Qualität der von Tutoren abgegebenen Bewertungen erheblich schwanken kann und die Wartezeiten für die Studierenden bis zum Erhalt von Feedback auf jeden Fall höher liegen als bei einer automatisierten Lösung.

Gleichwohl sollen automatische Bewertungssysteme die menschliche Betreuung nicht gänzlich abschaffen, sondern ihren Einsatz gewinnbringender gestalten. In Tutorien können Lehrende ihre Zeit erfahrungsgemäß insbesondere für sinnvollere Aufgaben als die Fehlersuche in studentischen Programmen investieren. Individuelle Beratung von schwächeren Studierenden, die Wiederholung von in der Vorlesung vermittelten Konzepten an weiteren Beispielen, aber auch die ein-

gehende Beantwortung und manchmal Recherche zu über den Prüfungsstoff hinausgehenden Fragen von ambitionierten Studierenden wird so ermöglicht. Die Arbeitsweise in Tutorien und Übungsgruppen hat sich unserer Erfahrung nach entsprechend geändert und führt insbesondere auch auf der Seite der Tutoren und Lehrenden zu höherer Motivation.

Die Erfahrung zeigt allerdings auch, dass man gute, automatisiert bewertbare Java-Programmieraufgaben erst nach vielen Monaten Erfahrung zügig erstellen kann. Wie jede Spezifikation oder Software können auch Testtreiber oder Prüfregeln Lücken oder Fehler enthalten, die erst dann offenbar werden, wenn die Aufgabe zum ersten Mal vielen studentischen Lösungen ausgesetzt war. Um Programmieraufgaben hoher Qualität nachhaltig zu entwickeln, müssen diese wie ein langfristig eingesetztes Softwareprodukt über die üblichen Entwicklungsstadien von der Beta-Fassung über ein erstes Release und dann folgende Hotfixes und Patches gepflegt werden.

Die Aufgaben, die im hier beschriebenen Einsatzszenario verwendet wurden, sind zunächst vollständig an der Universität Duisburg-Essen bzw. der Hochschule Hannover neu entwickelt worden. Die Aufgaben wurden dann in mehreren aufeinander folgenden Semestern wiederverwendet, teilweise nach Korrekturen oder einfachen Anpassungen. Ursprünglich für Studierende der Informatik bzw. der Angewandten Informatik konzipiert, wurden einige Aufgaben auch in anderen, informatiknahen Studiengängen (z. B. Medizinisches Informationsmanagement) genutzt. Unsere Erfahrungen mit der Wiederverwendung von Aufgaben sind gut, weil sie das aus einer größeren Zahl von Lernenden stammende Feedback und damit die Robustheit der Aufgaben erhöht. Ferner wurden in Essen entwickelte Aufgaben auch an Lehrende anderer Hochschulen weitergegeben, die ebenfalls das Bewertungssystem JACK einsetzen und so mit weniger Vorbereitungsaufwand in den Produktivbetrieb starten konnten.

Eine weitere interessante Erfahrung ist, dass sich mit dem Einsatz automatischer Bewertungssysteme auch die grundsätzliche Art der Bewertung anpassen muss. Beispielsweise kann es sinnvoll sein, Studierende zu belohnen, die sich eher durch Nachdenken an die richtige Lösung herantasten, als durch inflationär viele unzureichende Einreichungsversuche [Iha+10]. Automatisch vergebene Strafpunkte ab einer gewissen Zahl von Versuchen sind eine mögliche Maßnahme. Eine andere Maßnahme sind dynamische Prüfungen mit zwei Testfallmengen, von denen eine vor Abgabefrist aktiv ist, während die zweite erst nach Abgabefrist verwendet wird. Als spielerisches Element könnte man Studierenden sogar erlauben, einzelne Testfälle der zweiten Menge gegen Punktabzug „hinzuzukaufen“. Derartige Verfahren werden allerdings nicht von allen Bewertungssystemen

unterstützt. Ferner sind sie nicht spezifisch für die objektorientierte Programmierung in Java und sollen daher hier nicht weiter vertieft werden.

3.4.2 Erfahrungen der Studierenden

Evaluationen aus studentischer Sicht zum Einsatz automatischer Bewertungssysteme in der Java-Lehre sind z. B. in [Stö+13; SG09; SG11b] verfügbar. Für Evaluationen zu den von uns eingesetzten Systemen JACK und GRAJA wird auf Kapitel 9 und 11 verwiesen. In diesem Abschnitt sollen beispielhaft Details aus weiteren Studien erörtert werden, die in direkterem Bezug zu den oben beschriebenen Aufgabentypen und Einsatzszenarien stehen.

Im Wintersemester 2012/13 wurde dazu eine Umfrage unter 56 Studierenden eines 1. Semesters im Studiengang „Angewandte Informatik“ der Hochschule Hannover durchgeführt. Im Einsatzszenario „Übung und Tutorium“ (siehe Abschnitt 3.3.2) mussten die Studierenden 60% aller in Übungsaufgaben erreichbaren Punkte als verpflichtenden Teil der Prüfungsleistung erlangen. Die Aufgabenform entsprach weitestgehend einfachen und fortgeschrittenen Übungsaufgaben (siehe Abschnitt 3.2.1 und 3.2.2). Wenige Aufgaben hatten den Charakter von Projektaufgaben (siehe Abschnitt 3.2.3). Zusammenfassend beurteilten 78% der Studierenden das Bewertungssystem GRAJA als hilfreich. Verglichen mit menschlichen Tutoren wurde GRAJA als schneller, vergleichbar gut beim Entdecken von Fehlern, jedoch nicht als gerechter eingeschätzt. Dem Einsatz in einer Klausur (siehe Abschnitt 3.3.4) ohne flankierende, durch menschliche Bewerter durchgeführte Überprüfungen, standen die Befragten dieser Studie mit 73% mehrheitlich ablehnend gegenüber.

Positiver bzgl. des letztgenannten Aspekts fallen im Vergleich dazu die Ergebnisse einer ähnlichen Befragung unter den Studierenden der Studiengänge „Angewandte Informatik“ und „Wirtschaftsinformatik“ an der Universität Duisburg-Essen im Wintersemester 2011/12 aus. Von 63 Teilnehmern gaben hier ca. 55% an, ihre Klausur gerne durch JACK bewerten zu lassen. In der gleichen Befragung gaben zudem ca. 50% der Teilnehmer an, dass ihnen das Konzept mit freien Übungsaufgaben und automatischer Bewertung ein selbstständiges und unabhängiges Arbeiten ermöglicht und dass JACK im Rahmen dieses Konzeptes ein sinnvolles E-Learning-Werkzeug darstellt. Folgerichtig würden 40% der Befragten das System auch in anderen Lehrveranstaltungen nutzen wollen.

Unterstützt werden diese einmaligen Befragungsergebnisse durch Beobachtungen des Nutzungsverhaltens in den folgenden Semestern. So wurden in Essen im Wintersemester 2015/16 von knapp 700 Studierenden in der Vorlesung nicht nur

knapp 11.000 Lösungen zu Testaten und etwa 12.000 Lösungen zu verpflichtenden Übungs- und Projektaufgaben eingereicht, sondern darüber hinaus auch gut 9.000 Lösungen zu weiteren freiwilligen Übungsaufgaben. Nicht zuletzt konnte durch das gesamte Konzept auch der Anteil derjenigen Studierenden, die die Lehrveranstaltung erfolgreich abschließen, erheblich gesteigert werden.

3.5 Fazit und Ausblick

Perspektiven für den weiteren zukünftigen Einsatz automatischer Bewertungssysteme ergeben sich vor allem daraus, dass sich noch einige Analyseaufgaben automatisieren lassen sollten. Insbesondere ist dabei zu berücksichtigen, dass in der industriellen Softwareentwicklung zum Teil schon Verfahren existieren, die in E-Learning-Systemen noch nicht zum Einsatz kommen, aber interessante weitere Möglichkeiten für Feedback und Bewertung eröffnen können.

Grenzen hat der Einsatz automatischer Bewertungssysteme immer dort, wo die kreative Leistung eines Studierenden wichtiger ist als die Umsetzung formaler Regeln. So kann eine automatisierte Bewertung zwar sowohl auf syntaktische als auch inhaltliche Fehler hinweisen und dabei durch den Vergleich mit Musterlösungen möglicherweise sehr präzise benennen, an welcher Stelle ein Fehler liegt, aber dies bedeutet nicht zwangsläufig, dass der Fehler nur durch eine Korrektur an dieser Stelle behoben werden kann. Stattdessen ermöglichen Programmieraufgaben im Allgemeinen beliebig viele korrekte Lösungen und damit auch beliebig viele Möglichkeiten, einen gegebenen Fehler zu beheben.

Literatur für dieses Kapitel

[Bös98] László Böszörményi. „Why Java is not my favorite first-course language“. In: *Software-Concepts & Tools* 19.3 (1998), S. 141–145.

[Edw03] Stephen H. Edwards. „Improving Student Performance by Evaluating How Well Students Test Their Own Programs“. In: *J. Educ. Resour. Comput.* 3.3 (Sep. 2003). DOI: 10.1145/1029994.1029995.

[Gri08] David Gries. „A principled approach to teaching OO first“. In: *ACM SIGCSE Bulletin* 40.1 (2008), S. 31–35.

[HTW04] Emily Howe, Matthew Thornton und Bruce W. Weide. „Components-first approaches to CS1/CS2: principles and practice“. In: *ACM SIGCSE Bulletin* 36.1 (2004), S. 291–295.

[Iha+10] Petri Ihantola u. a. „Review of Recent Systems for Automatic Assessment of Programming Assignments“. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling '10. ACM, 2010, S. 86–93. DOI: 10.1145/1930464.1930480.

[Pea+07] Arnold Pears u. a. „A Survey of Literature on the Teaching of Introductory Programming“. In: *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '07. ACM, 2007, S. 204–223. DOI: 10.1145/1345443.1345441.

[Reg06] Stuart Reges. „Back to basics in CS1 and CS2“. In: *ACM SIGCSE Bulletin* 38.1 (2006), S. 293–297.

[SG09] Michael Striewe und Michael Goedicke. „Effekte automatischer Bewertungen für Programmieraufgaben in Übungs- und Prüfungssituationen“. In: *DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik*. Bd. 153. LNI. GI, 2009, S. 223–234.

[SG11b] Michael Striewe und Michael Goedicke. „Studentische Interaktion mit automatischen Prüfungssystemen“. In: *DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik*. Bd. 188. LNI. GI, 2011, S. 209–220.

[SSB10] Harry M. Sneed, Richard Seidl und Manfred Baumgartner. *Software in Zahlen*. Carl Hanser Verlag GmbH & Co. KG, 2010.

[Stö+13] Andreas Stöcker u. a. „Evaluation automatisierter Programmbewertung bei der Vermittlung der Sprachen Java und SQL mit den Gradern *aSQLg* und *Graja* aus studentischer Perspektive.“ In: *DeLFI 2013 – Die 11. E-Learning Fachtagung Informatik*. Bd. 218. LNI. GI, 2013, S. 233–238.

4 Automatisierte Bewertung in der Ausbildung relationaler Datenbankabfragesprachen am Beispiel SQL

Oliver J. Bott, Felix Heine und Carsten Kleiner

Zusammenfassung
Der Beitrag führt ein in die Nutzung automatisierter Programmbewertung durch SQL-Grader in der SQL-Ausbildung und stellt die typischen Abläufe bei Aufgabenstellung und Bewertung vor. Weiterhin wird diskutiert, welche Lernziele mit dem Einsatz von SQL-Gradern erreicht werden können, und wie der Einsatz von SQL-Gradern gegen alternative Ansätze des rechnergestützten Assessments abzugrenzen ist. Der Hauptteil des Beitrags stellt potenzielle didaktische Einsatzszenarien anhand konkreter Beispiele im Rahmen eines Informatikstudiengangs und eines informatiknahen Studiengangs vor. Das Kapitel schließt mit einer Zusammenfassung didaktischer Einsatzszenarien und einem Ausblick auf offene Forschungs- und Entwicklungsfragestellungen.

4.1 Einleitung

Datenbanksysteme sind Basis zahlreicher Anwendungssysteme und werden in den unterschiedlichsten Anwendungsfeldern eingesetzt. Relationale und objektrelationale Datenbankmanagementsysteme (DBMS) dominieren seit Jahrzehnten

Das diesem Bericht zugrundeliegende Vorhaben eCULT (Teilvorhaben e-Assessment) wurde im Rahmen des Programms „Qualitätspakt Lehre“ vom Bundesministerium für Bildung und Forschung gefördert (Förderkennzeichen 01PL16066D).

den Markt. Die im Kontext dieser Systeme vorherrschende Datenbankabfragesprache Structured Query Language (SQL) [DD97] ist Gegenstand der Vermittlung in einer Vielzahl von Studienfächern und Lehrveranstaltungen.

Hierbei ist das Spektrum der Studienfächer nicht nur auf die Kerninformatik und die anwendungsbezogenen Informatikstudiengänge begrenzt, vielmehr spielt das Wissen um die Nutzung in relationalen Datenbanken abgelegter Daten in allen Disziplinen eine Rolle, in denen die Analyse und Weiterverarbeitung systematisch erhobener Daten relevant ist.

Die Vermittlung der SQL kombiniert üblicherweise die theoretische Vermittlung relationaler Datenbank- und Datenbankabfragekonzepte mit praktischen Anteilen in Form von Präsenzübungen und Übungen, die als Hausaufgaben bearbeitet werden. Zur Unterstützung der praktischen Übungen, aber auch zur Leistungsüberprüfung im Rahmen von formativen oder summativen Assessments oder auch zum Self-Assessment bietet sich die automatisierte Programmbewertung als ergänzendes didaktisches Instrument an. Damit einher geht die Zielsetzung, Studierende effizienter betreuen zu können und ihnen zudem zu mehr Selbstbestimmtheit beim Lernen zu verhelfen. Eine Vielzahl von Veröffentlichungen zu entsprechenden SQL-Trainingssystemen bzw. SQL-Gradern (s. auch Kapitel 12) konnte inzwischen den Nutzen derartiger Systeme für die Lehre demonstrieren, so zum Beispiel SQLT-Web [Mit03], SQLator [Sad+04], SQL-ACME [Sol+06], LEARN-SQL [Abe+08], IDLE-SQL [PK09], eledSQL [UGB13], aSQLg [Stö+13], XDaTA [Bha+15] oder ÜPS [Ifl+14b].

Für den Lehrenden stellt sich die Frage, wie diese Werkzeuge in der SQL-Ausbildung eingesetzt werden können. Dieser Beitrag stellt zur Beantwortung dieser Frage zunächst generelle Konzepte der Aufgabenstellung und Bewertung von SQL-Gradern vor, diskutiert mit diesen Systemen potenziell erreichbare Lernziele und grenzt den Einsatz automatisierter Bewertungssysteme für SQL gegen alternative e-Assessment-Ansätze ab. Anhand von Berichten aus der Praxis werden dann didaktische Konzepte der SQL-Ausbildung unter Einsatz automatisierter Programmbewertung vorgestellt. Das Kapitel schließt mit einem Fazit und einem Ausblick. Vorausgesetzt werden mindestens grundlegende Kenntnisse der SQL.

4.2 Aufgabenstellung und Bewertung

Eine Besonderheit in Bezug auf die Ausgestaltung von Aufgaben für die automatisierte Programmbewertung von SQL-Anweisungen gegenüber entsprechenden Konzepten für zum Beispiel objektorientierte Programmiersprachen ist, dass in der Regel

1. den Aufgaben ein vorgegebenes Datenbankschema zugrunde liegt, das
2. mit exemplarischen Daten gefüllt ist,
3. einzelne SQL-Anweisungen Gegenstand der Aufgaben sind und
4. eine Musterlösung in Form eines entsprechenden SQL-Statements vorliegt.

Die Bewertung der Aufgaben durch die Trainingssysteme erfolgt dann anhand einer Auswahl aus üblicherweise folgenden Bewertungskriterien:

1. Syntaktische Korrektheit der SQL-Anweisungen.
2. Einhaltung stilistischer Regeln (z. B. Großschreibung von Schlüsselworten) und/oder einschränkender Regeln zur Auswahl und Kombination von Schlüsselworten (z. B. zwingende Verwendung von INNER JOIN).
3. Vergleich der Ergebnisse der Ausführung der abgegebenen Lösung mit dem Ergebnis der Musterlösung in Bezug auf beispielsweise Spaltenanzahl und -benennungen sowie Anzahl, Sortierung und inhaltliche Übereinstimmung der enthaltenen Tupel.
4. Dauer bzw. allgemein Ressourcenbedarf der Bearbeitung der Anweisung.

Die Angabe einschränkender Regeln zur Auswahl und Kombination von Schlüsselworten ist dann erforderlich, wenn (wie häufig) verschiedene SQL-Anweisungen zum gleichen Resultat führen, aber nur eine vorgegebene Anweisung oder ein eingeschränktes Spektrum von Anweisungen für die Antwort zulässig sein soll. Hier sind Black-Lists verbotener oder White-Lists erlaubter Schlüsselworte gängige Konzepte, Trainingssysteme wie ÜPS [Ifl+14b] vergleichen zu diesem Zweck dagegen den Parsebaum des Statements mit Parsebäumen der Musterlösung(en) und bewerten Abweichungen als Fehler.

Wesentlicher einschränkender Faktor der SQL-Grader ist das Spektrum des unterstützten Sprachumfangs. Die meisten Systeme unterstützen ausschließlich auf SELECT basierende Datenbankabfragen als Teil der Data Manipulation Language (DML), andere erlauben auch DML-Anweisungen, die den Datenbankinhalt verändern, wenige auch Teile der Data Definition Language (DDL). Die Unterstützung von DDL- und auch die Datenbank verändernden DML-Anweisungen wird seltener von SQL-Gradern unterstützt, da die Bewertung deutlich komplexer ausfällt: Für jede Bewertung muss in der Regel eine Kopie der Datenbank bereitgestellt werden. SQL-Grader, die auf Basis eines konkreten DBMS arbeiten schränken den Sprachumfang in der Regel zusätzlich auf denjenigen des eingesetzten DBMS ein.

Die Rückmeldung der Trainingssysteme orientiert sich an den zur Verfügung stehenden Bewertungskriterien und liefert je nach Grader mehr oder weniger aussagekräftige Hinweise insbesondere im Falle negativer Bewertungen zu den genannten Kriterien. Zudem können die Bewertungskriterien mit einer Bewertungsvorschrift verknüpft sein, die im einfachsten Falle eine Punktzahl berechnet, die dann häufig zu einer Summenpunktzahl zusammengeführt wird. Auch werden von einigen Trainingssystemen die Musterlösung und/oder die Ergebnistabellen im Rahmen der Rückmeldung präsentiert.

Ein weiteres häufiges und didaktisch relevantes Merkmal der Trainingssysteme ist die Wiederholbarkeit einer Abgabe. Je nach Vorgabe des Lehrenden kann dem Lernenden in einem Testsystem ermöglicht werden, die Abgabe einer Lösung mehrfach oder beliebig oft zu wiederholen und so auf Basis der rückgemeldeten Hinweise eine fehlerhafte Lösung zu verbessern. Diese Funktionalität ermöglicht formative, das Lernen begleitende e-Assessment-Szenarien, in denen die Rückmeldungen des Systems der Lernunterstützung dienen. Für diagnostische e-Assessments zur Unterstützung der Selbsteinschätzung oder für summative e-Assessments zur abschließenden Prüfung und Bewertung der vom Lernenden erworbenen Kompetenzen des Lernenden spielt diese Funktionalität keine wesentliche Rolle.

Ein Autorensystem eines SQL-Trainingssystems bietet auf Basis dieser Grundkonzeption bei der Gestaltung von Aufgaben mehr oder weniger komfortabel folgende Optionen:

1. Anlage des zugrundeliegenden Datenbankschemas inklusive Datenbasis.
2. Textuelle Beschreibung der Aufgabenstellung.
3. Vorgabe einer Musterlösung.
4. Festlegung der Bewertung ggf. auf Basis der Parametrierung jedes einzelnen Bewertungskriteriums.
5. Vorgabe der Wiederholbarkeit der Abgabe einer Lösung gegebenenfalls unter Angabe zunehmend spezifischerer Lösungshinweise.

4.3 Unterstützbare Lernziele und Abgrenzung

Betrachtetet man die auf dieser Grundlage prinzipiell erreichbaren Lernziele, so sind durch den Einsatz von SQL-Trainingssystemen nach obigem Muster direkt

folgende Kompetenzen im Rahmen eines formativen oder summativen e-Assessments überprüfbar bzw. deren Vermittlung unterstützbar (unter Bezugnahme auf die Bloom'sche Taxonomie, vgl. [AKB01]):

1. *Kennen*: Lernende *kennen* die Syntax relevanter SQL-Kommandos und die typischen Regeln der stilistischen Gestaltung von SQL-Statements. Beispielfrage: „Formulieren Sie eine syntaktisch und stilistisch korrekte SQL-Anfrage zur Projektion der Tabelle Personen auf die Spalten Name und Vornamen."

2. *Verstehen*: Lernende *verstehen* die Semantik relevanter SQL-Kommandos. Beispielfrage: „Formulieren Sie eine SELECT-Anweisung zur Auswahl aller Patienten und der mit Ihnen in Behandlungsbeziehung stehenden ärztlichen Mitarbeiter. Verwenden Sie hierfür ‚INNER JOIN'."

3. *Anwenden*: Lernende können die SQL *anwenden*, um klassische Anwendungsprobleme zu lösen. Beispielfrage: „Formulieren Sie eine SQL-Anfrage zur Auswahl aller Patienten, die *nicht* in Behandlungsbeziehung zu irgendeinem ärztlichen Mitarbeiter stehen."

4. *Analysieren/Bewerten*: Lernende können verschiedene Lösungsvarianten im Hinblick auf deren Performance *analysieren* und die bestmögliche auswählen. Beispielfrage: „Formulieren Sie das folgende SQL-Statement mithilfe von JOINS und ohne Unterabfragen neu und überlegen Sie, welche Variante potenziell schneller ausgeführt werden kann." (das gegebene Statement enthält dabei eine Unterabfrage, die mit IN oder EXISTS eingebunden ist).

Die Lernzielklassen *Zusammenführen* und *Beurteilen* werden durch die automatisierte Programmbewertung in SQL höchstens marginal erreicht, da hier beispielsweise bei detaillierter Analyse der Ursachen für ein ineffizientes SQL-Statement weitere Werkzeuge zum Einsatz kommen sollten.

Insbesondere im Hinblick auf die Lernzielklassen 1 und 2 lassen sich die o. g. Lernziele auch mit alternativen Methoden des e-Assessments wie insbesondere die Verwendung von Lückentext-, Multiple-Choice- oder Zuordnungsaufgaben überprüfen bzw. unterstützen. Allerdings haben diese Optionen Nachteile gegenüber der automatischen Programmbewertung insbesondere im Hinblick auf die Anforderung an den Lernenden, das Antwort-Statement aktiv zu konstruieren. Lückentexte erzwingen zwar ein aktives Erarbeiten des gefragten SQL-Statements, lassen allerdings in der Regel nur ein eng umschriebenes Spektrum erlaubter Antworten zu und ermöglichen lediglich allgemeinere Hinweise zur Bewertung der Qualität der Lösung. Ob das Ergebnis auch mit einer eventuell von

der Musterlösung abweichenden, aber dennoch in Bezug auf das Ergebnis korrekten Lösung erreicht wird, kann bei einem Lückentext nur dann berücksichtigt werden, wenn alle möglichen Lösungen antizipiert und explizit beschrieben werden.

Bei der Verwendung von Multiple-Choice- oder Zuordnungsaufgaben besteht gegenüber der automatischen Bewertung der Nachteil, dass Lernende hierbei nicht gefordert sind, die Lösung mit ihren Bestandteilen aktiv zu erarbeiten, sondern *lediglich* eine vermutlich funktionierende Lösung zu erkennen. Ein weiterer Nachteil von Multiple-Choice- und Zuordnungsaufgaben gegenüber Aufgaben eines Testsystems eröffnet sich vor dem Hintergrund der Wiederholbarkeit einer Abgabe (s. o.): Während in formativen e-Assessment-Szenarien dem Lernenden in einem Testsystem nahezu beliebig oft ermöglicht werden kann, eine Lösung abzugeben und auf Basis der rückgemeldeten Hinweise eine fehlerhafte Lösung zu verbessern, erschöpft sich diese Möglichkeit bei Multiple-Choice und Zuordnungsaufgaben naturgemäß sehr rasch. Nichts desto trotz können insbesondere MC-Fragen zur korrekten Syntax oder Bedeutung von SQL-Anweisungen zum Beispiel im Rahmen von mit Audience-Response-Systemen (ARS; auch Klicker-Systeme) an das Plenum einer Vorlesung gestellten Fragen hilfreich sein, um Kennen und Verstehen von SQL-Anweisungen zu vertiefen oder den bisher erreichten Lehrerfolg zu überprüfen.

4.4 Einsatzszenarien und Erfahrungen

Im folgenden Kapitel werden anhand konkreter Einsatzszenarien und didaktischer Konzepte Optionen des Einsatzes automatisierter Programmbewertung in der SQL-Ausbildung diskutiert.

4.4.1 SQL im Studiengang Medizinisches Informationsmanagement

Im Bachelorstudiengang Medizinisches Informationsmanagement (BMI) an der Hochschule Hannover erfolgt im ersten Semester eine Einführung in die relationale Datenbanktheorie inklusive einer ersten Einführung in SQL. Im zweiten Semester werden in einer hierauf aufbauenden Veranstaltung die SQL-Kenntnisse vertieft. In beiden Veranstaltungen wird automatisierte Programmbewertung in unterschiedlichen didaktischen Szenarien eingesetzt.

4.4.1.1 Grundlagen relationaler Datenbanken (Datenbanken I)

Die Veranstaltung Grundlagen relationaler Datenbanken (Datenbanken I) führt im ersten Semester des BMI-Studiengangs in grundlegende Konzepte der Entwicklung relationaler Datenbanken ein:

- Grundbegriffe der Datenbanktechnologie (DBMS etc.),
- Datenbankentwicklungsprozess,
- Datenmodellierung auf Basis der ER-Modellierung,
- Grundlagen des relationalen Modells (insb. Einführung in die relationale Algebra) und Abbildung von ER-Modellen in Relationenschemata,
- Einführung in die SQL (DDL und DML).

Die Veranstaltung ist organisiert in eine zweistündige Vorlesungseinheit im Plenum und eine vierstündige praktische Übung in Kleingruppen. In der Übung wird anhand eines konkreten DBMS die Datenbankentwicklung eingeübt. In die Vorlesung integriert sind Übungseinheiten, die partiell in Form begleitender e-Assessment-Aufgaben unter Einsatz eines ARS im Präsenzunterricht durchgeführt werden, sowie Aufgaben, die freiwillig als Hausaufgabe bearbeitet werden können. Für die Lehrveranstaltung wird ein begleitender Kurs im Lernmanagementsystem (LMS) Moodle angeboten, über den Materialien wie Folien, Übungsaufgaben und weitere Materialien zugänglich gemacht werden.

Automatisierte Programmbewertung wird auf Grundlage des Graders aSQLg (s. Kapitel 12) integriert in das LMS Moodle im letzten Viertel der Veranstaltung durch Angebot von zwei Übungsblättern mit jeweils 10 Aufgaben (s. Abb. 4.1) eingesetzt. Bewertet wurden Syntax, Kosten (Zeit) und Korrektheit des Ergebnisses (s. Abb. 4.2). Auf Grundlage eines durchgängig in der Lehrveranstaltung verwendeten Datenmodells einer Hochschule werden SQL-Aufgaben in Bezug auf SELECT-Aufgaben gestellt. Dabei steigert sich der Schwierigkeitsgrad der Aufgaben im ersten Aufgabenblatt von einfachen Projektionen und Umbenennungen über die Erzeugung sortierter Ergebnisrelationen und die Ausblendung doppelter Ergebnisse hin zu Selektionen mit einfachen und komplexeren Selektionsprädikaten. Das zweite Aufgabenblatt fokussierte dann auf Kreuzprodukte und Kreuzprodukte mit Selektionsprädikaten im Hinblick auf die Nutzung von Beziehungen zwischen Entitäten, Natural Joins und Theta-Joins, Mengenoperatoren, Aggregationsoperatoren und Gruppierungen ohne und mit Filterung über

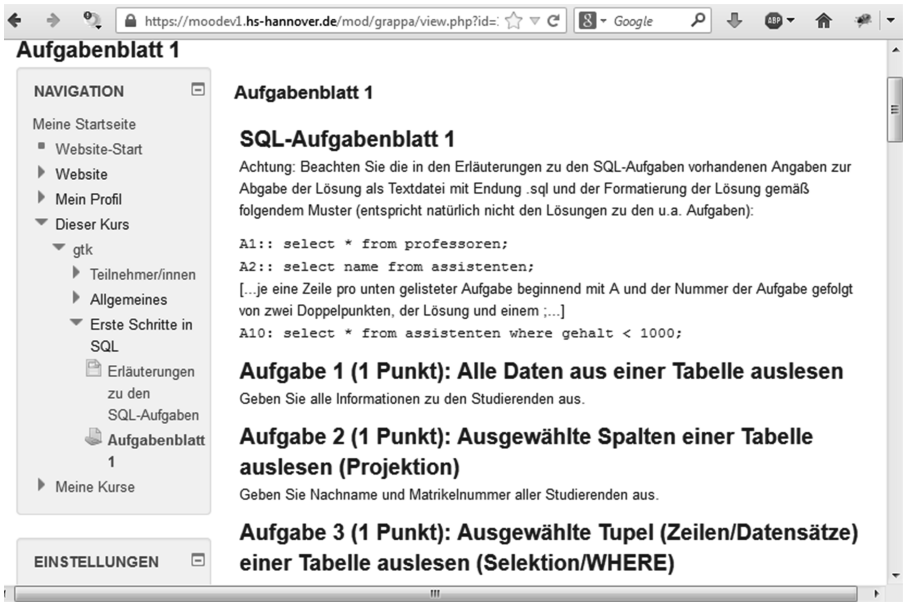


Abbildung 4.1: Beispielhafte Aufgabenbeschreibung eines SQL-Übungsblatts

HAVING sowie äußere Joins zur Auswertung von (Nicht-)Beziehungen. Dabei wurden so oft wie möglich Aufgaben gestellt, die bereits in zwei analogen Aufgabenblättern zur relationalen Algebra gestellt wurden – in diesem Fall allerdings ohne Option der automatisierten Programmbewertung. Bei Evaluationen dieses Einsatzszenarios im Wintersemester 14/15 (n=44 Studierende) und im Wintersemester 15/16 (n=43) haben 77 % (n=43) bzw. 64 % (n=32) der Studierenden das freiwillig nutzbare Angebot in Anspruch genommen. Die Übersichtlichkeit der Oberfläche sowie die Qualität des Feedbacks insgesamt wurde mit gut bewertet (jeweils Schulnote 2,1 (n=35/34) bei Noten von 1-5 bzw. Note 2,0 und 2,2 (n=30)) und die Geschwindigkeit mit gut bis sehr gut (1,6 (n=34) bzw. 1,9 (n=30)). 12 % (n=33) bzw. 3 % (n=29) der Studierenden waren eher nicht der Meinung, dass der Einsatz des Programms geholfen hätte, Fehler im Quellcode der SQL-Lösung zu finden, 70 % bzw. 60 % bewerteten diese Frage mit „eher Ja“ bzw. „Ja“, der Rest mit „teils-teils“. 87,5 % (n=32) bzw. 79,3 % (n=29) der Studierenden können sich vorstellen, automatisierte Programmbewertung auch in anderen Fächern einzusetzen. Die Didaktik der Lehrveranstaltung insgesamt wurde mit 1,5 (n=24) und 1,9 (n=10) bewertet, wobei die letzte Bewertung aufgrund der niedrigen Teilnehmerzahl nur bedingt aussagekräftig ist. Ergänzend angemerkt sei, dass der Einsatz des ARS, eingeführt im Wintersemester 13/14, im Wintersemester 14/15 mit der Note 1,2 (n=47) als sehr gut benotet wurde. Das ARS wurde allerdings nicht im Kon-

text SQL, sondern im Zusammenhang insbesondere mit der ER-Modellierung, der Überführung von ER-Modellen in Relationenschemata und der relationalen Algebra eingesetzt, würde sich aber natürlich analog für die SQL-Vorlesungsanteile eignen.

4.4.1.2 Vertiefungsveranstaltung Datenbanken II

Die Veranstaltung Datenbanken II des BMI-Studiengangs vertieft die im ersten Semester vermittelten SQL-Grundkenntnisse. Vermittelt werden alle relevanten Sprachbestandteile der DDL, DML und der DQL. Weiterführende Themen sind Transaktionssicherheit, Stored Procedures/Trigger und datenbankbasierte Anwendungsentwicklung.

Die Veranstaltung ist organisiert in eine zweistündige Vorlesungseinheit im Plenum und eine zweistündige begleitende Übung in Kleingruppen. Für die Lehrveranstaltung wird analog zu Datenbanken I ein begleitender Kurs im Lernmanagementsystem (LMS) Moodle angeboten. In der Übung wird parallel zu den Vorlesungsinhalten der jeweils behandelte Sprachumfang eingeübt. Da im Rahmen der Übungen zur Vorlesung bereits Aufgaben zu SQL gestellt und die Lösungen diskutiert werden, beschränkt sich die Nutzung der automatisierten Programm-

A10 2.0/2.0

Prüfungstyp	Punkte	Kommentare
Syntaxprüfung	0.67/0.67	Die Syntax ist korrekt.
Kostenprüfung	0.67/0.67	Kosten Ihres Statements: 4.0. Absolute Kostenobergrenze ist: 100000. Kosten der Musterlösung: 4.0. Die Kosten der Abfrage sollte kleiner als 8.0 sein. Die Kosten der Abfrage sind ok.
Ergebnisprüfung	0.67/0.67	Die Spaltenanzahl Ihres Ergebnisses ist richtig. Die Datentypen der Spalten Ihres Ergebnisses sind richtig. Die Namen der Spalten Ihres Ergebnisses sind richtig. Ihr Ergebnis hat die richtige Zeilenanzahl. Die Werte Ihrer Ergebniszeilen sind richtig. Die Sortierung der Ergebnisse ist richtig. Das Ergebnis ist vollständig richtig.

Abbildung 4.2: Beispielhaftes Feedback des aSQLg-Graders

bewertung – erneut wird aSQLg integriert in Moodle eingesetzt – auf den Zeitraum der Klausurvorbereitung. Bereitgestellt wird eine Probeklausur mit 20 Aufgaben zu SQL, davon sieben Aufgaben zu Datenbankabfragen mit dem SELECT-Statement. Das Spektrum der Aufgaben umfasste komplexere Abfragen über eine und mehrere Tabellen inklusive Sortierung und Selektion, Aggregationen und Gruppierungen und Unterabfragen. Nur diese Aufgabenkategorien konnten mit aSQLg automatisiert bewertet werden, die weiteren aufgrund des diesbezüglich eingeschränkten Sprachumfangs des verwendeten Graders noch nicht.
Insgesamt nutzten im Sommersemester 15 (n=39) 31 % der Studierenden (n=32) das freiwillige Angebot. Die Übersichtlichkeit wurde ebenso wie die Geschwindigkeit der Rückmeldung mit Note 2,4 (n=20, n=12) bewertet, die Qualität des Feedbacks im Hinblick auf dessen Unterstützungspotenzial die richtige Lösung zu finden mit 2,5 (n=10). 27,3 % (n=11) empfanden das Feedback geeignet, um Fehler in der eigenen Lösung zu finden, 27,3 % waren diesbezüglich unentschieden, 45,5 % schätzten das Feedback als diesbezüglich wenig hilfreich ein. Bei vergleichsweise geringer Antwortrate ist dieses im Verhältnis zu den Evaluationsergebnissen des Wintersemesters 14/15 unerwartete Ergebnis der Bewertung der Nützlichkeit des Feedbacks mit Vorsicht zu interpretieren. In den Freitextantworten nannten die Studierenden technische Probleme des Systems während der Vorbereitungszeit, was sich negativ auf die Bewertung des Feedbacks ausgewirkt haben könnte. Die Studierenden, die das System nutzen konnten, gaben in den Freitextrückmeldungen zum Teil positive Rückmeldungen, zum Teil wünschten sie sich detailliertere Informationen zu den eigenen Fehlern. Dies untermauert die Vermutung, dass gerade bei komplexeren Aufgaben die auf Syntax, Ausführungszeit und Ergebniskorrektheit fokussierten Rückmeldungen des Systems nur noch sehr bedingt geeignet sind, die Gründe der Fehler in der eigenen Lösung zu finden. 83,3 % (n=18) der Studierenden können sich vorstellen, automatisierte Programmbewertung auch in anderen Fächern einzusetzen.

4.4.2 SQL im Bachelorstudiengang Angewandte Informatik

Der Bachelorstudiengang Angewandte Informatik (BIN) an der Hochschule Hannover ist ein sechssemestriger Informatikstudiengang. Im Bereich der Datenbanksysteme sieht das Curriculum drei Module im Pflichtbereich vor:

- eine Grundlagenveranstaltung (Datenbanken), die sich mit konzeptioneller Datenmodellierung, Grundlagen relationaler Datenbanken sowie einer Einführung in SQL befasst,

- eine Vertiefungsveranstaltung (Informationssysteme I), die Interna relationaler Datenbanken, Anfrageoptimierung, Transaktionen und eine Vertiefung in SQL zum Inhalt hat sowie

- eine weitere Vertiefungsveranstaltung (Informationssysteme II), die sich mit der Anwendungsprogrammierung von relationalen Datenbanken sowie weiteren Datenbankkonzepten beschäftigt.

Für die automatisierte Bewertung von SQL sind somit insbesondere die ersten beiden Veranstaltungen relevant.

4.4.2.1 Datenbanken

Insbesondere in der Basisveranstaltung, in der die Studierenden erstmals mit SQL in Kontakt kommen, ist eine automatisierte Bewertung von SQL sinnvoll, da hier einerseits die größten Teilnehmerzahlen zu erwarten sind. Andererseits sind die zu erwartenden Fehler der Studierenden leichter automatisiert zu erkennen, da es sich eher um grundlegende Probleme handelt. Daher kann hier die Lernunterstützung besonders effizient eingesetzt werden. Allerdings spielen in dieser Veranstaltung neben einfachen Anfragen besonders auch Anweisungen der DDL und DML eine große Rolle. Diese werden bisher jedoch noch nicht vom eingesetzten Grader aSQLg unterstützt (s. Kap. 12.7). Aus diesem und anderen organisatorischen Gründen konnte an der Hochschule Hannover bisher jedoch noch keine systematische Auswertung des Einsatzes von aSQLg in der Veranstaltung Datenbanken erfolgen.

4.4.2.2 Informationssysteme I

In dieser Veranstaltung stehen im SQL-Teil insbesondere komplexere SELECT-Abfragen im Vordergrund, die sich ebenfalls sehr gut für eine automatisierte Bewertung eignen. Daher wurde in dieser Veranstaltung bereits mehrfach das entsprechende Werkzeug aSQLg eingesetzt und dieser Einsatz empirisch untersucht. Die Ergebnisse basieren auf Befragungen von Studierenden im Sommersemester 2012 und 2013. Zu dieser Zeit wurde das System nicht im Kontext eines vollwertigen LMS eingesetzt, sondern als eigenständiges Werkzeug angeboten. Insgesamt beteiligten sich 39 Studierende an der Umfrage im Sommersemester 2012, die im folgenden genauer erläutert wird; dies entspricht in etwa 80 % der Teilnehmenden. Die Ergebnisse (vgl. auch Abb. 4.3) zeigen, dass über 50 % sich mindestens

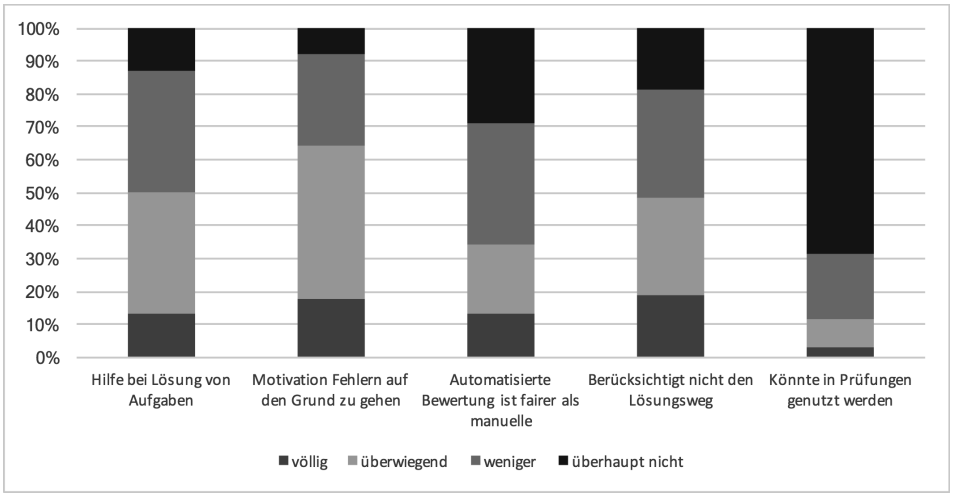


Abbildung 4.3: Einschätzung der Studierenden zur automatisierten SQL-Bewertung in der Veranstaltung Informationssysteme I des Studiengangs BIN (n=39)

überwiegend bei der Lösung der Aufgaben unterstützt fühlten. Dazu passt die Angabe, dass etwa 40 % das unmittelbare Feedback von aSQLg nach Einreichen einer Lösung als die wertvollste Funktion bewerteten.

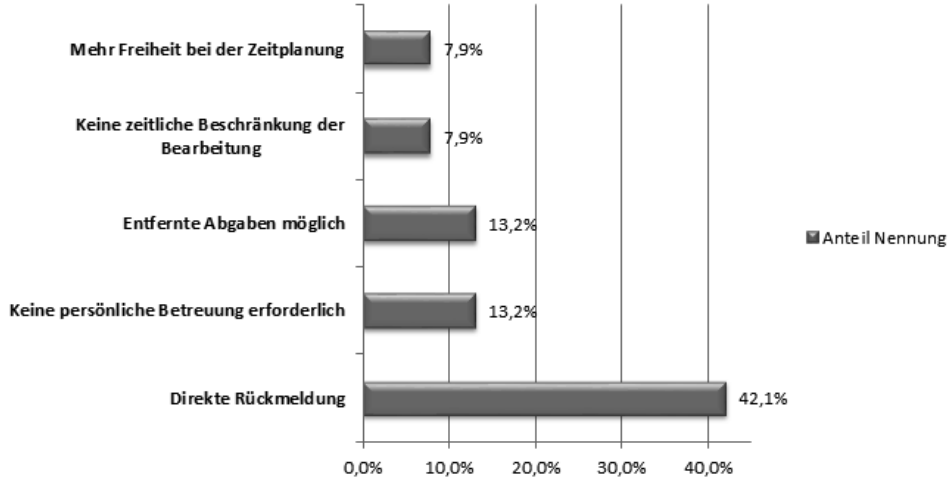


Abbildung 4.4: Zustimmung zu positiven Aspekten in der Veranstaltung Informationssysteme I des Studiengangs BIN (n=39)

Die Steigerung der Motivation bei der Bearbeitung der Aufgaben sowie die tiefer gehende Fehlersuche bei initialen Problemen ist insbesondere im Bereich der SQL-Ausbildung problematisch. Insofern ist es wichtig, dass etwa 70 % der Studierenden sich durch das Werkzeug motiviert fühlten, verbliebene Probleme in ihren Abgaben zu beheben. Möglicherweise hat damit das Werkzeug auch zur insgesamt verbesserten Bewertung der Lehrveranstaltung beigetragen.

Im Bereich der kritischen Anmerkungen ist insbesondere die Befürchtung zu nennen, dass manuelle Bewertungen fairer sind (70 %) und dass bei automatisierter Bewertung sehr ergebnisorientiert und weniger lösungswegorientiert (50 %) bewertet wird. Diese sind allerdings nur für Szenarien relevant, in denen die Bewertungen vom Lehrenden weiterverwendet werden. Für die reine Unterstützung der Studierenden bei der Lösungsfindung ist dies hingegen nicht von Bedeutung.

Die wichtigsten genannten Vorteile des Systems (vgl. Abb. 4.4) neben der unmittelbaren Rückmeldung auf die Lösung waren die Möglichkeit selbstbestimmter zu lernen (in Bezug auf Raum und Zeit) sowie die Möglichkeit einer entfernten Abgabe von Aufgaben. Insofern treten auch hier sowohl didaktische wie auch organisatorische Vorteile auf. Das Werkzeug selbst zeigte nur sehr geringe Fehlerraten und ca. 60 % würden es auch in Datenbankveranstaltungen zu fortgeschrittenen Themen einsetzen wollen. 40 % der Studierenden können sich einen Einsatz in anderen Informatikveranstaltungen vorstellen.

Bei den negativen Aspekten der automatisierten SQL-Bewertung wurden neben didaktischen (bspw. Nichtberücksichtigung der Lösungswege) vor allem organi-

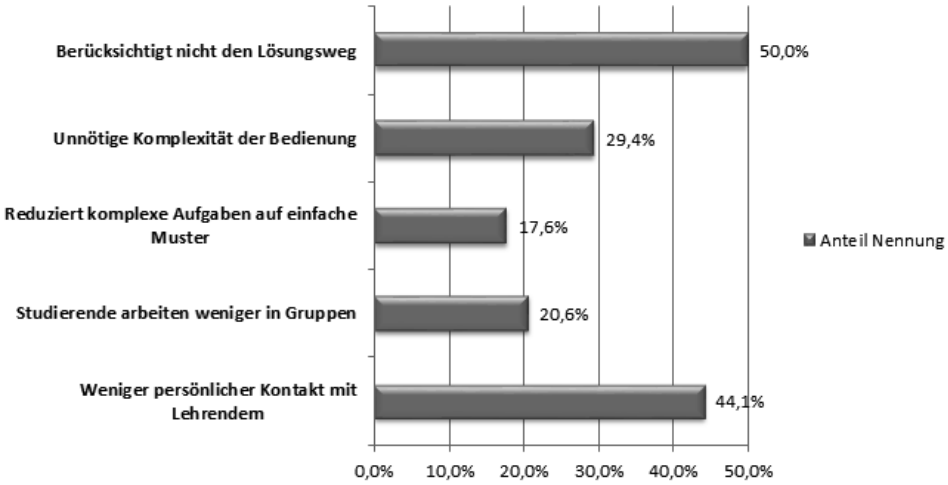


Abbildung 4.5: Zustimmung zu negativen Aspekten in der Veranstaltung Informationssysteme I des Studiengangs BIN (n=39)

satorische Themen (bspw. Befürchtungen zur zunehmenden sozialen Isolation der Lernenden) genannt (vgl. Abb. 4.5). Diese Aspekte hängen nicht unmittelbar mit dem Werkzeug selbst zusammen, sondern eher mit der Betreuungsform im Allgemeinen. Der Einsatz sollte also daher ergänzend zu Präsenzveranstaltungen denn ersetzend erfolgen. Dies wird auch durch die Zurückhaltung bei der Einschätzung der Eignung für Prüfungen deutlich.

4.5 Fazit und Ausblick

Als primäres didaktisches Einsatzszenario automatisierter Programmbewertung bei der Vermittlung der SQL ist das formative Assessment im Rahmen begleitender Übungen zur Vermittlung von Anwendungskompetenz zu nennen. Abhängig von der Feedbackqualität bietet sich der Einsatz von SQL-Gradern vor allem als ergänzendes Angebot zu betreuten Übungen an. Aber auch summatives Assessment (E-Klausuren), die Vorbereitung hierauf sowie Selbsteinschätzung der Studierenden sind sinnvoll umsetzbare Szenarien.

Wird automatisierte Bewertung von SQL-Anfragen in E-Klausuren eingesetzt, sollte bei der Aufgabenstellung sowie der Auswahl des Graders berücksichtigt werden, dass viele Grader auf eine Syntaxprüfung und einen Vergleich der Abfrageresultate fokussieren, und daher ggf. Lösungen akzeptiert werden, die zwar das gewünschte Resultat produzieren, von der Aufgabenstellung gegebenenfalls so aber nicht intendiert waren. Produziert ein Grader unter Umständen Fehlbewertungen, sollten Lösungen, welche die volle Punktzahl nicht erreicht haben, ggf. noch einmal manuell geprüft werden.

Wichtige Erfolgsfaktoren des Einsatzes von SQL-Gradern sind zudem eine gute technische Unterstützung bei Fehlern und Bedienungsproblemen, eine möglichst nahtlose Integration in die gewohnte Arbeitsumgebung bzw. das an der jeweiligen Hochschule verwendete Lernmanagementsystem, eine möglichst abteilungs- bzw. hochschulweit einheitliche Präsentation über mehrere Veranstaltungen bzw. Programmiersprachen hinweg sowie der Einsatz von automatisierter Bewertung in mehreren Veranstaltungen (z. B. Programmieren, Theoretische Informatik, Modellierung).

Um die Effektivität des Einsatzes von SQL-Gradern in den oben genannten didaktischen Szenarien zu steigern, ist weitere Forschungs- und Entwicklungsarbeit notwendig, vor allem auf den Gebieten Feedbackqualität und semantische Analyse der Lösungen, Aufgaben-Repositories und Austauschstandards für SQL-Aufgaben, unterstützter Sprachumfang und LMS-Integration. Studierende würden im Rahmen des formativen Assessments von einem Feedback profitieren,

das nicht nur etwaige syntaktische Fehler auflistet, sondern bei fehlerhaften Lösungsvorschlägen die Fehler erklärt und hieraus abgeleitete Verbesserungsvorschläge gibt. Hierfür ist eine semantische Analyse der Lösung im Vergleich zur Musterlösung erforderlich, die zudem erlauben würde, von einer „ganz oder gar nicht"-Bewertung eines Lösungsvorschlages wegzukommen. Lehrende würden von online verfügbaren Aufgaben-Repositories profitieren, die im Falle von SQL-Aufgaben nicht nur die Aufgabenstellung sowie die Musterlösung beinhalten, sondern auch die zugrundeliegende Beispieldatenbank, Bewertungsvorschläge und Feedbackvorgaben. Idealerweise lassen sich die Aufgaben des Repositories für beliebige SQL-Grader einsetzen, was wiederum ein standardisiertes Aufgabenaustauschformat erfordert (siehe hierzu Kapitel 24). Weiterhin wäre die vollständige Unterstützung auch von DDL, DML und DCL (Data Control Language) erforderlich, um das gesamte Spektrum von SQL-Statements abfragen zu können. Für Aufgaben im Rahmen des summativen Assessments wären parametrierbare, d. h. für jeden Studierenden individualisierte Aufgaben hilfreich, um Plagiate zu verhindern. Für Studierende und Lehrende gleichermaßen hilfreich wäre die nahtlose Integration der SQL-Grader in das gewohnte Lernmanagementsystem.

Literatur für dieses Kapitel

[Abe+08] Alberto Abelló u. a. „LEARN-SQL: Automatic Assessment of SQL Based on IMS QTI Specification". In: *ICALT*. IEEE, 2008, S. 592–593.

[AKB01] Lorin W. Anderson, David R. Krathwohl und Benjamin Samuel Bloom. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. Allyn & Bacon, 2001.

[Bha+15] Amol Bhangdiya u. a. „The XDa-TA system for automated grading of SQL query assignments". In: *2015 IEEE 31st International Conference on Data Engineering*. Apr. 2015, S. 1468–1471. DOI: 10.1109/ICDE.2015.7113403.

[DD97] Chris J. Date und Hugh Darwen. *A Guide To Sql Standard*. Bd. 3. Addison-Wesley Reading, 1997.

[Ifl+14b] Marianus Ifland u. a. „ÜPS – Ein autorenfreundliches Trainingssystem für SQL-Anfragen". In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014, S. 259–264. ISBN: 978-3-88579-627-5.

[Mit03] Antonija Mitrovic. „An Intelligent SQL Tutor on the Web“. In: *I. J. Artificial Intelligence in Education* 13.2-4 (2003), S. 173–197.

[PK09] Claus Pahl und Claire Kenny. „Interactive Correction and Recommendation for Computer Language Learning and Training“. In: *IEEE Transactions on Knowledge and Data Engineering* 21.6 (Juni 2009), S. 854–866. DOI: 10.1109/TKDE.2008.144.

[Sad+04] Shazia Sadiq u. a. „SQLator: an online SQL learning workbench“. In: *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*. ITiCSE '04. ACM, 2004, S. 223–227. DOI: 10.1145/1007996.1008055.

[Sol+06] Josep Soler u. a. „A Web-based tool for teaching and learning SQL“. In: *International Conference on Information Technology Based Higher Education and Training, ITHET*. 2006.

[Stö+13] Andreas Stöcker u. a. „Evaluation automatisierter Programmbewertung bei der Vermittlung der Sprachen Java und SQL mit den Gradern *aSQLg* und *Graja* aus studentischer Perspektive.“ In: *DeLFI 2013 – Die 11. E-Learning Fachtagung Informatik*. Bd. 218. LNI. GI, 2013, S. 233–238.

[UGB13] Anja Ufert, Andreas Grillenberger und Torsten Brinda. „eledSQL – Entwicklung und Erprobung einer webbasierten Lernumgebung für Datenbanken und SQL“. In: *DeLFI 2013 – Die 11. E-Learning Fachtagung Informatik*. Bd. 218. LNI. GI, 2013, S. 167–178.

5 Automatisierte Programmbewertung in der deduktiven/logischen Programmierung am Beispiel der Prolog-Ausbildung

Tobias Thelen, Helmar Gust und Elmar Ludwig

Zusammenfassung
In der Prolog-Ausbildung als Teil des Studienmoduls „Künstliche Intelligenz“ wird an der Unversität Osnabrück automatisierte Programmbewertung sowohl für den Übungs- als auch den Klausurenbetrieb genutzt. Dieses Kapitel skizziert zunächst die Besonderheiten der Programmiersprache Prolog, legt anschließend die didaktischen Prämissen des Grader-Einsatzes dar und diskutiert schließlich anhand von Beispielaufgaben und Einsatzerfahrungen die Möglichkeiten der autmatisierten Programmbewertung für Prolog.

5.1 Einleitung

5.1.1 Die Programmiersprache Prolog

Die Programmiersprache Prolog wurde zu Beginn der 1970er Jahre in Marseille, Frankreich entwickelt, um natürlichsprachliche Dialogsysteme zu entwerfen (vgl. [CR96]). Prolog ist eine dynamisch typisierte interpretierte Sprache, die im Kern einen Theorembeweiser für eine Teilmenge der Prädikatenlogik enthält. Die Prädikatenlogik wird in vielen Wissenschaftsbereichen verwendet, um Argumente zu formalisieren und daraus folgende Schlussfolgerungen auf ihre Gültigkeit zu überprüfen. Auf diese Weise weist zumindest der Sprachkern von Prolog wohldefinierte semantische Eigenschaften auf. In der Künstlichen Intelligenz wird Prädikatenlogik als ein wichtiger Formalismus zur Wissensrepräsentation verwendet und

Prolog dementsprechend als Programmiersprache, die „intelligentes“ Schlussfolgern modelliert und ermöglicht (vgl. z. B. [Bra01]).

Im Gegensatz zu den meisten verbreiteten Programmiersprachen sind Prolog-Programme deklarativ zu verstehen: Anstatt Schritt für Schritt den Lösungsweg niederzulegen, beschreibt ein Prolog-Programm Ausgangsdaten und Eigenschaften von Lösungen, die dann vom Prolog-Interpreter anhand eines Beweisverfahrens gesucht werden. In wesentlichen Konzepten unterscheidet sich Prolog radikal von anderen Programmiersprachen, so zum Beispiel beim Variablenkonzept. Variablen in Prolog sind logische Variablen, die entweder frei (unbelegt) oder gebunden (belegt) sind und dann ihre Bindung nicht mehr ändern können. Prolog ermöglicht nichtdeterministische Programme, d. h. Programme, die mehr als eine Lösung haben. So ist es typisch, dass ein Prolog-Programm eine Lösung generiert und dann aufgefordert wird, weitere Lösungen zu generieren.

Prolog bietet mit komplexen Termen und Listen zwei eingebaute flexible Datenstrukturen. Auch hier treten wieder Besonderheiten von Prolog zutage: Für den Vergleich und die Zuweisung von Datenstrukturen und Variablen kommt ein Unifikationsverfahren zum Einsatz, das nach Möglichkeiten sucht, Strukturen durch die Belegung darin enthaltener Variablen einander anzugleichen. Dieses Prinzip spielt eine zentrale Rolle für den Beweisprozess und stellt einen wichtigen Schlüssel für das Verständnis von Prolog dar. Die Verarbeitung von Listen kann nur rekursiv erfolgen, so dass Rekursion in Prolog-Programmen der Regelfall und nicht die Ausnahme ist.

Die deklarative Herangehensweise, die Verwendung von Rekursion, eingebaute komplexe Datenstrukturen und der Unifikationsmechanismus sorgen dafür, dass Prolog-Programme in aller Regel sehr kompakt sind. Mit nur wenigen Codezeilen lassen sich komplexe Programme realisieren. Zwei Beispiele sollen einige besonders typische Eigenschaften von Prolog verdeutlichen.

```
sehenswuerdigkeit(osnabrueck, schloss).
sehenswuerdigkeit(osnabrueck, museum).
sehenswuerdigkeit(fuerstenau, schloss).
sehenswuerdigkeit(hannover, schloss).
sehenswuerdigkeit(hannover, see).
sehenswuerdigkeit(bramsche, museum).
besuchstipp(X) :-
    sehenswuerdigkeit(X, schloss).
besuchstipp(X) :-
    sehenswuerdigkeit(X, S1),
    sehenswuerdigkeit(X, S2),
    S1 \= S2.
```

Dieses Prolog-Programm ist ein typisches Beispiel für eine deduktive Datenbank. Das Prädikat sehenswürdig enthält (analog zu einer Datenbanktabelle) Aussagen über Orte und ihre Sehenswürdigkeiten in Form von Prolog-Fakten. An diese Datenbank können zum Beispiel im Prolog-Interpreter interaktiv Anfragen gestellt werden (% leitet Zeilenkommentare ein):

```
% Gibt es in Osnabrück ein Museum?
[1] ?- sehenswuerdigkeit(osnabrueck, museum).
true
% In welcher Stadt gibt es einen See?
[2] ?- sehenswuerdigkeit(X, see).
X=hannover
% In welcher Stadt gibt es Schloss und Museum?
[3] ?- sehenswuerdigkeit(X, schloss),
       sehenswurdigkeit(X, museum).
X=osnabrueck
% In welcher Stadt gibt es ein Museum?
[4] ?- sehenswuerdigkeit(X, museum).
X=osnabrueck ;
X=bramsche ;
false
```

Anfrage [4] zeigt den nichtdeterministischen Charakter von Prolog-Programmen: Anfragen können mehrere gültige Antworten haben, die Prolog sukzessive berechnen kann, bis keine weiteren korrekten Antworten mehr herleitbar sind.

```
[5] ?- besuchstipp(osnabrueck).
true
[6] ?- besuchstipp(X).
X=osnabrueck ; X=hannover ;
X=fuerstenau ; X=osnabrueck ;
X=hannover ;
false
```

Die Regel Besuchstipp besteht aus zwei Teilregeln (Klauseln), die als Oder-verknüpfte Implikationen („:-“ steht für den umgedrehten Imkplikationspfeil) zu lesen sind. Eine Stadt ist also dann ein Besuchstipp, wenn sie über ein Schloss verfügt oder mindestens zwei Sehenswürdigkeiten aufweist. Das Oder ist nichtexklusiv, deshalb erscheinen Osnabrück und Hannover, die beide Kriterien erfüllen, in der Ergebnisliste doppelt.

Das zweite Beispiel ist ein rekursiv definiertes Prädikat mit drei Argumenten, das wahr ist, wenn das dritte Argument eine Liste enthält, die als Verkettung der Listen in den ersten beiden Argumenten zu sehen ist.

```
verkette([],L,L).
verkette([H|R],L2,L3) :-
    verkette(R,L2,[H|L3]).
```

Dieses sehr einfach erscheinende Prädikat bedient eine Fülle von Anwendungsfällen, was Prolog-Anfänger häufig vor große Schwierigkeiten stellt. Als Standardlesart wird häufig die Verketten-Operation angenommen, also eine prozedurale Lesart, in der zwei gegebene Listen zu einer dritten, noch unbekannten verkettet werden. Da aber jeder Parameter auch mit einer freien Variablen belegt werden kann, ist diese Möglichkeit nur eine von vielen.

```
[1] ?- verkette([1,2],[3,4],[1,2,3,4]).
true
[2] ?- verkette([1,2,3],[4,5,6],X).
X = [1,2,3,4,5,6]
[3] ?- verkette(X,[4,5,6],[1,2,3,4,5,6]).
X = [1,2,3]
[4] ?- verkette([1,2,3],X,[1,2,3,4,5,6]).
X = [4,5,6]
```

Anfrage [1] prüft, ob die drei gegebenen Listen die Verkettungsrelation erfüllen, Anfrage [2] belegt X mit der Verkettung der beiden gegebenen Listen. Das Prädikat kann aber auch verwendet werden, um bekannte Präfixe (Anfrage [4]) oder Suffixe (Anfrage [3]) abzuspalten. All diese Anfragen haben nur eine Lösung.

Wird nur eine oder gar keine Variable belegt, ergeben sich weitere Möglichkeiten, die mehrere Lösungen haben und dem nichtdeterministischen Programmieren zuzurechen sind.

```
[5] ?- verkette(L1,L2,[1,2,3,4]).
[6] ?- verkette(L, L, L2).
```

Anfrage [5] erzeugt alle möglichen Zerteilungen der als drittes Argument gegebenen Liste. In der interaktiven Verwendung des Prolog-Interpreters müssen all diese Lösungen nacheinander explizit angefordert werden. Bei der Ausführung eines Prolog-Programms (d. h. der Suche nach einem Beweis samt Variablenbelegungen für die gestellte Anfrage) passiert die Generierung einer weiteren Lösung immer dann, wenn sich ein möglicher Lösungsweg als Sackgasse erwiesen hat

und der Prolog-Interpreter weitere Lösungenmöglichkeiten durchprobiert. Denkbar wäre hier zum Beispiel die Verwendung von Anfrage [5] in einem Programm, das prüfen soll, ob es eine Zerlegung einer Liste von Zahlen gibt, so dass beide Teile die gleiche Summe aufweisen. In diesem Programm würde die Zeile `verkette(L1,L2,Liste)` dafür sorgen, dass bei der Suche nach einer Lösung wie im Beispiel alle möglichen Zerlegungen von einer gegebenen `Liste` in zwei Teile generiert werden.

Anfrage [6] verwendet nur freie Variablen, fordert aber, dass das erste und zweite Argument identisch sind. Die Anfrage generiert beliebig viele Lösungen (bis zum Erreichen von Speicherlimits), die alle ausschließlich freie Variablen enthalten. Die Anfrage generiert somit alle Muster für Listen, die als Verdopplung einer Liste betrachtet werden können. Solch eine Form der Mustergenerierung wird häufig für Generate-And-Testansätze verwendet: Ein Generator-Ziel wie in Anfrage [6] generiert sukzessive alle möglichen Muster für erlaubte Lösungen, die anschließend getestet werden. Diese Form der Programmierung ist in der Regel nicht besonders effizient, aber besonders einfach zu implementieren.

Die beiden in diesem Abschnitt präsentierten Beispiele sollten einen kleinen Einblick in die Programmierung mit Prolog bieten und dabei vor allem aufzeigen, dass sehr kompakte Programme bereits komplexe Verhaltensweisen hervorrufen können. Die Gründe dafür liegen in der automatischen Beweisstrategie, der Verwendung logischer Variablen mit Unifikationsmechanismus und der rekursiven Verarbeitung komplexer Datenstrukturen.

5.1.2 Der Studiengang Cognitive Science

Die noch relativ junge Disziplin Kognitionswissenschaft befasst sich in interdisziplinärer Weise mit Fragestellungen rund um das Phänomen (menschlicher) Kognition, d. h. des menschlichen Geistes. An der Universität Osnabrück werden seit 1998 internationale kognitionswissenschaftliche Studiengänge (Bachelor, Master, Promotionsprogramm) eingerichtet, die in großer Breite kognitionswissenschaftliche Fragestellungen abbilden. Die Studiengänge sind mittlerweise sehr gut etabliert und ziehen Studierende mit sehr unterschiedlichen Hintergründen und Interessenspektren in großer Zahl an.

Insbesondere der Bachelorstudiengang zählt mit über 120 Studienanfängern pro Jahr zu den größeren Studiengängen an der Universität Osnabrück und steht an der Schwelle zum Massenfach. Das Bachelorprogramm besteht aus 8 Teilmodulen, die jeweils eine kognitionswissenschaftliche Teildisziplin abdecken. Im Einzelnen sind dies: Computerlingustik, Informatik, Kognitive (Neuro-)Psychologie,

Künstliche Intelligenz, Mathematik, Neuroinformatik, Neurowissenschaft (Neurobiologie, Neurophysiologie) und Philosophie des Geistes. Studierende müssen in jedem dieser Module ein bis zwei Einführungs- bzw. Grundlagenveranstaltungen belegen, und in fünf der acht Module weiterführende Wahlpflichtveranstaltungen belegen. Darüber hinaus gibt es einen individuell zu füllenden Profilbildungsbereich mit 22-33 Leistungspunkten und ein verpflichtendes Auslandssemester.

5.2 Einsatzszenario: Introduction to Artificial Intelligence and Logic Programming

Die Vorlesung/Übung „Introduction to Artificial Intelligence and Logic Programming“ (Intro AI) ist eine Pflichtverstaltung im Modul „Artificial Intelligence“. Sie wird also von allen Studierenden des Bachelorprogramms besucht und ist laut Studienplan für das zweite Fachsemester empfohlen. Zu den Voraussetzungen zählen im Wesentlichen eine Einführung in die Logik und die Informatikeinführung „Informatik A – Algorithmen und Datenstrukturen“, die im ersten Fachsemester besucht werden. Parallel zu „Intro AI“ werden überlicherweise die ebenfalls sehr arbeitsintensiven Vorlesungen und Übungen zur Einführung in die Computerlinguistik und die Philosphie des Geistes belegt.

Die Veranstaltung besteht aus Vorlesung, Übung und Tutorien. Im Vorlesungsteil werden Konzepte eingeführt, die im Übungsteil mit praktischen Programmieranteilen in Prolog verknüpft werden. Studentische Tutoren führen Tutorien für Gruppen von ca. 30 Studierenden durch, die der Wiederholung und Festigung des Wissens dienen. Die von den Studierenden zu erbringenden Leistungen bestehen aus vier Teilen:

- zwei Blöcke von Übungsaufgaben, die in Teams von bis zu vier Studierenden als wöchentliche Hausarbeiten zu erledigen sind,
- zwei Klausuren, eine in der Mitte und eine am Ende des Semesters.

Die Veranstaltung baut inhaltlich vor allem auf der Einführung in die Logik auf und verdeutlicht das Potenzial der Prädikatenlogik erster Stufe als Wissensrepräsentationsformalismus. Von dieser Basis aus wird Prolog als automatisiertes Beweisverfahren motiviert, das (künstliche) Intelligenz als Schlussfolgern aus bekannten Fakten und Regeln darstellt. In den ersten Wochen spielt daher die „Übersetzung“ natürlichsprachlicher Aussagen in Prädikatenlogik und von dort in Prolog eine große Rolle. Neben einer Einführung in die (sehr einfache) Prolog-Syntax steht dabei auch das ungewohnte Variablenkonzept samt Unifikationsverfahren im

Vordergrund. Die erste Hälfte der Veranstaltung wird mit der prozeduralen Semantik (wie arbeitet der Beweiser?), der in Prolog sehr einfachen Implementation kontextfreier Grammatiken und der Verarbeitung komplexer Datenstrukturen, d. h. vor allem der Verwendung von Rekursion zur Verarbeitung von Listen, abgerundet.

In der zweiten Hälfte der Veranstaltungen stehen zunächst extra-logische Eigenschaften von Prolog im Vordergrund, vor allem die Kontrolle des Beweisverfahrens und dynamische Wissensbasen. Anschließend werden einige ausgewählte Probleme aus der Künstlichen Intelligenz (KI) in den Block genommen und mit verschiedenen Verfahren implementiert. Leitendes Grundprinzip ist dabei die Vorstellung von „Intelligenz als Suche“, d. h. Probleme werden so formuliert, dass ihre Lösungen als Suchraum aufgefasst werden können. Wird ein solcher Suchraum als Baum verstanden, lassen sich Baumtraversierungsalgorithmen als Suchstrategien für KI-Probleme anwenden. Auf diese Weise werden uninformierte und informierte Suchverfahren in Prolog implementiert und auf klassische Probleme wie Schiebepuzzle, Wegfindung und Zahlenrätsel angewandt. Den Abschluss bilden Alternativen und Erweiterungen dieser Herangehensweise wie Constraint-Logisches Programmieren, Meta-Programmierung, und Prolog-Varianten wie Datalog und F-Logic.

5.3 Prolog im Übungsbetrieb

Die wöchentlichen Übungsaufgaben der Vorlesung/Übung bestehen überwiegend aus Prolog-Aufgaben. Häufig verwendete Aufgabentypen sind die Formulierung von Prolog-Anfragen, die Vervollständigung von Prolog-Programmen und die vollständige Eigenimplementation von Prolog-Programmen.

Als Grader-Umgebung wird das LMS Stud.IP mit dem Vips-Plugin (s. Kapitel 14 und 20) verwendet und in zwei verschiedenen Arbeitsweisen eingesetzt: Als Auswertungshilfe für die Aufgabenkorrektur und als Arbeitsunterstützung für Studierende. Die Kombination dieser beiden Modi ist ein typisches Merkmal von intelligenten tutoriellen Systemen (ITS), die zusätzlich eine Benutzermodellierungs- und Ablaufsteuerungskomponente haben. Der Prolog-Grader für Vips basiert auf Vorarbeiten, die so ein ITS für Prolog umsetzen ([Bön+99], [Pey+00], [Pey02]), allerdings auch als Unterstützung für den Übungsbetrieb gedacht sind. Auch wenn Rolf Schulmeister den ITS für die Prolog-Programmierung vorwirft, rein akademischen Charakter zu haben ([Sch97, S. 198]), haben sich die hier verfolgten Ansätze in der Praxis vor allem als Auswertungsunterstützung bewährt,

wie auch andere Arbeiten für die Programmbewertung demonstrieren ([Bra+06], [BKW05]).

In letzterem Fall wird nicht die Bewertungsfunktion der Grader-Umgebung verwendet, sondern die Möglichkeit, Prolog-Programme aus der Stud.IP-/Vips-Eingabe heraus auszuführen und die Berechnungsergebnisse angezeigt zu bekommen. Das Lernmanagementsystem wird auf diese Weise zu einer einfachen Entwicklungsumgebung, in der Programme formuliert, getestet und gespeichert werden können. In der Praxis wird diese Möglichkeit relativ selten genutzt, da die Implementation dieser Umgebung vergleichsweise rudimentär ist. Die Darstellung der Ergebnisse erfolgt nicht über AJAX-Requests, d. h. es baut sich jedesmal die komplette Seite neu auf, was bei kleinen Anfragen und schrittweisem Vorgehen zu merklichen Verzögerungen führt. Zum anderen fehlt eine History-Funktion, um ältere Anfragen zu wiederholen. Es gibt kein Syntax-Highlighting und keine Möglichkeit, Programm-, Anfrage- und Ausgabebereiche frei anzuordnen. Daher verwenden die Studierenden entweder eine lokale SWI-Prolog-Installation auf ihren Arbeitsrechnern oder nutzen die komfortablere Online-Entwicklungsumgebung SWISH[1].

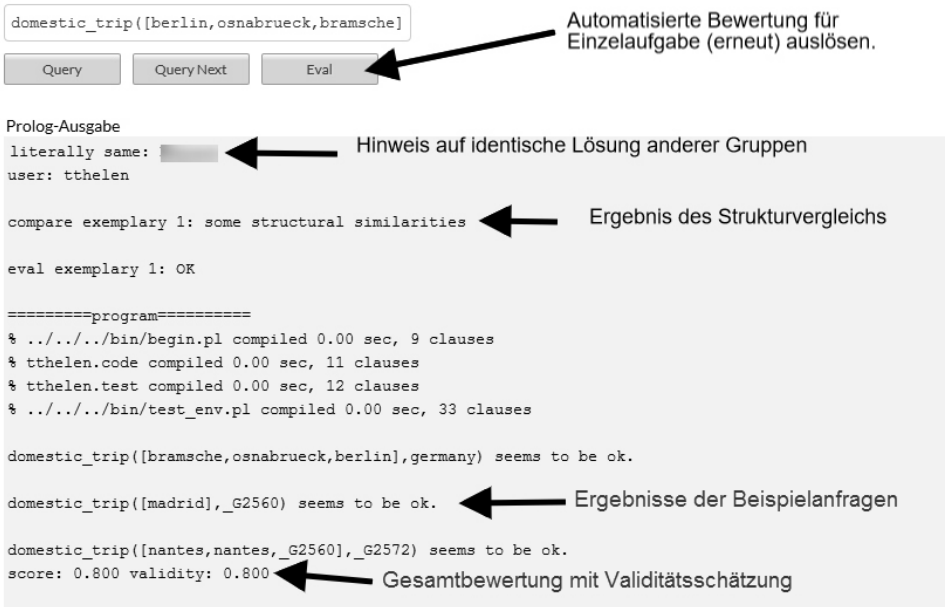


Abbildung 5.1: Bewertung einer Übungsaufgabe als Auswertungshilfe für die Korrektur

1 http://swish.swi-prolog.org/

Die Studierenden werden allerdings aufgefordert, die Funktionsfähigkeit ihrer abgegebenen Lösung in der Stud.IP-/Vips-Umgebung zu testen, da vor allem die Korrektur auf die interaktiven Möglichkeiten zurückgreift. Grundsätzlich gibt es bei der automatischen Korrektur zwei zweifelsfreie Fälle: Entweder wurde eine leere Lösung abgegeben (bzw. am vorgegebenen Programm nichts verändert), oder eine Lösung, die einer Musterlösung entspricht.

Als Auswertungshilfe für die Korrektur wendet der Grader dann zwei verschiedene Strategien an (s. Kapitel 14), die auch im Beispieloutput (s. Abbildung 5.1) zu erkennen sind:

1. Vergleich der resultierenden Variablenbelegungen: Getestet wird immer mit einer Menge vorgegebener Testanfragen, die als Prologanfragen ausgewertet werden und als Resultat eine Menge von Variablenbelegungen liefern. Diese Variablenbelegungen werden zwischen einer oder mehrerer gegebenen Musterlösungen und der abgegebenen Lösung verglichen, und zwar nicht nur eine/die erste mögliche Belegung, sondern deren Gesamtmenge. Da auch unendlich große Lösungsmengen möglich sind, ist die Maximalzahl der Vergleiche zu beschränken. Im Beispiel liefern alle Beispielanfragen korrekte Ergebnisse.

2. Vergleich der Programmstruktur: Da Prolog-Programme wie oben demonstriert sehr kompakt sind, ist es vergleichsweise gut möglich, die Struktur von Lösungen zu bewerten. Dazu werden stufenweise strukturelle Vereinfachungen vorgenommen (Entfernung von Kommentaren, Entfernung von Variablennamen etc.) und Vergleiche mit mehreren Musterlösungen, die als unterschiedlich gut gekennzeichnet werden können, vorgenommen. Im Beispiel werden nur „some similarities“ konstatiert. Die zugrunde liegende Lösung verwendet ein falsches Kriterium für den Rekursionsabbruch, was durch den Strukturvergleich trotz fehlender Testanfrage für diesen Fall erkannt wird.

Die vorgenommene Bewertung wird aus beiden Aspekten berechnet und den Korrekteuren zugänglich gemacht. Dabei wird auch graphisch unterschieden, ob eine Korrektur im oben skizzierten Sinne sicher ist (also nicht manuell kontrolliert werden muss), oder nur als Bewertungsvorschlag zu verstehen ist. In letzterem Fall nutzen die korrigierenden Tutoren typischerweise die Möglichkeit, Testanfragen direkt über Stud.IP/Vips an die abgegebene Lösung zu stellen und die Ausgaben mit der vorgeschlagenen Bewertung abzugleichen. Der Beispielscreenshot oben schlägt 4/5 Punkte als Bewertung vor (5 Punkte maximal * Bewertung 0.8 = 4).

5.3.1 Übungstyp: Formulierung von Prolog-Anfragen

Dieser Übungstyp ist vor allem in den ersten Wochen relevant, wenn Grundkonzepte von Prolog erarbeitet werden. Typischerweise wird ein Prolog-Programm und eine natürlichsprachliche Beschreibung des Anfrageziels vorgegeben, die Studierenden sollen dann eine geeignete Anfrage formulieren und sowohl Anfrage als auch Ergebnis dokumentieren.

Typisches Beispiel ist hier die Modellierung von Familienstammbäumen als deduktive Datenbank (zum Beispiel aus der griechischen Mythologie entnommen), an die dann Anfragen zu stellen sind, wie: „Wie heißen die Kinder von Zeus?“, „Welche Individuen haben gemeinsame Kinder?“.

5.3.2 Übungstyp: Vervollständigung von Prolog-Programmen

Bei diesem Aufgabentyp soll ein vorgegebenes Programm ergänzt, verändert oder korrigiert werden. Ein grundsätzlicher Vorteil dieses Aufgabentyps ist es, dass auch unsichere Studierende in der Regel eine gute Vorstellung davon haben, was zu tun ist, da keine völlig freien Entscheidungen über Aufbau und Strukturierung des Programms zu treffen sind. Ein typisches Beispiel ist die Fomulierung eines einzelnen Prädikates zu einem gegebenen Programm.

```
% city(City, Country)
city(osnabrueck, germany).
city(bramsche, germany).
city(berlin, germany).
city(paris, france).
city(nantes, france).
city(marseille, france).

% domestic_trip(Tour, Country)
%  The predicate is provable if all cities
%  are located in the same country.
domestic_trip(Tour, Country) :-
   ...
```

Ergebnis ist ein typisches rekursives Prädikat mit zwei Klauseln, von denen die erste die Rekursionsbasis darstellt und die zweite den rekursiven Fall abbildet. Hier kann es zum Beispiel zwei Sturkturvarianten geben:

```
% Variante 1: Rekursionsabbruch bei
% einelementiger Liste
domestic_trip([City],Country) :-
    city(City, Country).
domestic_trip([City|Rest], Country) :-
```

```
    city(City, Country),
    domestic_trip(Rest, Country).

% Variante 2: Rekursionsabbruch bei leerer Liste
domestic_trip([],_).
domestic_trip([City|Rest], Country) :-
    city(City, Country),
    domestic_trip(Rest, Country).
```

Die zweite Variante scheint zunächst besser zu sein, da sie auf Codeduplizierung (city-Subgoal in beiden Klauseln) verzichtet. Allerdings führt sie dazu, dass auch eine leere Route als gültiger Trip bewertet wird, was der Aufgabenstellung widerspricht. Die beiden Varianten werden daher gewichtet: Variante 1 führt zu 100% der Punkte, Variante 2 nur zu 75%. Wichtig ist auch, bei der Formulierung von Testanfragen alle intendierten Verwendungsweisen mit zu berücksichtigen.

5.3.3 Übungstyp: Vollständige Eigenimplementation von Prolog-Programmen

In einem dritten Aufgabentyp sind die Studierenden angehalten, aufgrund einer Problemstellung völlig frei zu entscheiden, wie die Implementation ausgestaltet werden soll. Hierbei sind zwei wichtige Einschränkungen zu beachten:

1. Alle Programmeingaben erfolgen in einem einzigen Textfeld. Damit ist es nicht möglich, Programme zu implementieren und einzureichen, die aus mehr als einer Datei bestehen. Angesichts der typischen Kompaktheit von Prolog-Programmen ist das für die vorliegende Veranstaltung kein grundsätzliches Problem, da sich alle behandelten Problemstellungen mit maximal 200-300 Zeilen Code erledigen lassen. Sollen allerdings größere Datenbasen verwendet werden, müssen diese auch Teil der Programme sein, was im Einzelfall etwas unübersichtlich werden kann.

2. Das Programm kann nur getestet werden, wenn eine bekannte Testanfrage verwendet werden kann, d. h. zumindest das Top-Level-Prädikat einen erwarteten Namen und eine erwartete Stelligkeit aufweist.

Mit komplexer werdenden Problemstellungen und insbesondere flexibleren Möglichkeiten der Aufteilung in Teilprobleme (und Teilprädikate) besteht kaum noch die Möglichkeit, sinnvolle Musterlösungen für den Strukturvergleich anzugeben. In diesen Fällen dient die Musterlösung im Wesentlichen nur noch dazu, die Variablenbelegungen für Testfälle zu prüfen.

5.4 Prolog im Klausurbetrieb

Die an den Prolog-Grader angeschlossene Online-Übungsumgebung wird neben dem Übungsbetrieb auch für die Durchführung von Online-Klausuren zur Vorlesung/Übung „Introduction to Artificial Intelligence and Logic Programming“ im PC-Pool verwendet. In den Klausuren spielt Prolog eine wichtige Rolle. Der überwiegende Teil der Klausur besteht allerdings aus geschlossenen Aufgabenformaten, insbesondere Einfach- oder Mehrfachauswahlfragen.

Für Prolog-Aufgaben werden grundsätzlich die gleichen Übungstypen wie im Übungsbetrieb verwendet, allerdings in veränderter Gewichtung. Umfangreiche Programmieraufgaben spielen nur eine geringe Rolle, sehr viel häufiger sind Ergänzungs- oder Korrekturaufgaben, bei denen wenige Freiheitsgrade in der Implementation bestehen bzw. keine umfangreichen Entwurfstätigkeiten notwendig sind.

Die Grader-Umgebung wird auch hier wieder in zweierlei Hinsicht verwendet:

1. Als Korrekturunterstützung. Wie im Übungsbetrieb werden als sicher falsch oder sicher richtig erkannte Lösung vollautomatisch korrigiert und für alle anderen ein Ergebnis für die manuelle Nachkorrektur vorgeschlagen. Die Korrekteure verwenden die Möglichkeit, online Prolog-Testanfragen an die abgegebenen Lösungen zu stellen, um die Vorschläge zu überprüfen und mit effizienten Arbeitsabläufen zu einer endgültigen Bewertung zu gelangen.

2. Als Arbeitsunterstützung während der Klausur. Studierende können bei allen Prolog-Aufgaben ihre Ergebnisse testen, indem sie Testanfragen stellen bzw. vorformulierte Anfragen abschicken. Das Grader-Ergebnis im Sinne einer Bewertung des Vergleichs mit Musterlösungen wird hier nicht verwendet, sondern nur die oben skizzierte Funktionalität einer eingeschränkten Online-Entwicklungsumgebung.

Im einfachsten Fall bestehen Korrekturaufgaben darin, syntaktische und andere einfache Fehler in vorgegebenen Programmen zu finden und zu korrigieren. Im folgenden Beispiel fehlt ein Komma nach einem Subgoal in der zweiten Klausel (Syntaxfehler), und es wird ein falscher Listenoperator in der letzten Zeile verwendet (, statt |, semantischer Fehler). Der Syntaxfehler ist durch korrekte Interpretation der Prolog-Fehlermeldung beim Ausprobieren des Programms zu finden, der semantische Fehler am ehesten durch Nachvollziehen des Programms oder Ausprobieren, da das Muster `[X|F]` an dieser Stelle deutlich typischer ist als `[X,F]`.

```
% Korrigieren Sie das folgende Programm.
% (Hinweis: Es sind 2 Fehler enthalten)
rotate(_,0,L,L).
rotate(left,N,[X|R],L2) :-
    N > 0,
    append(R,[X],L1)
    N1 is N - 1,
    rotate(left,N1,L1,L2).
rotate(right,N,L,L2) :-
    N > 0,
    append(F,[X],L),
    N1 is N - 1,
    rotate(right,N1,[X,F],L2).
```

Dieser Aufgabentyp wäre natürlich auch durch einen reinen Textvergleich automatisch zu korrigieren. Der Grader-Einsatz bietet aber zum einen den Nebeneffekt der Online-Entwicklungsumgebung und ist zum anderen in der Lage, auch Teillösungen oder zu komplizierte Lösungen zu bewerten. Insbesondere letztere treten häufig auf: Variablen werden umbenannt, Klauselreihenfolgen geändert, Leerzeichen verschoben etc. und diese Lösungsversuche werden nicht vollständig rückgängig gemacht.

Ein anderer häufiger Ergänzungs- bzw. Veränderungstyp ist die Transformation von Programmen, die Negationen enthalten in solche, die stattdessen Kontrollanweisungen für den Beweisprozess verwenden („Cuts“) und umgekehrt. Hier sind nur sehr geringe Eingriffe in den Programmcode an richtiger Stelle nötig. Dabei ergeben sich die gleichen Vorteile wie bei den Korrekturaufgaben.

Freie Programmieraufgaben werden in den Klausuren wie erwähnt in nur geringem Maße genutzt, sind aber vorhanden. Im Gegensatz zum Übungsbetrieb ist hier sehr viel häufiger mit unvollständigen Lösungen zu rechnen, da die Studierenden unter sehr viel größerem Zeitdruck stehen und hoffen, auch für Lösungsansätze Punkte zu bekommen. Eine typische Aufgabenstellung sieht wie folgt aus:

```
/*
Define a predicate 'powerset' which computes
the powerset of a set.
Assume that sets are represented as sorted list.
Make sure, that the resulting powerset is sorted,
too (according to the termorder in prolog, you
may use 'sort'))
Example:
    ?- powerset([a,b,c],P)
     P = [[], [a], [a, b], [a, b, c], [a, c],
         [b], [b, c], [c]]
*/
```

Die Musterlösung zu dieser Aufgabe umfasst zwei Prädikate und insgesamt neun Zeilen. Eine solche Lösung ist noch einfach genug, dass eine hinreichende Zahl

von korrekten Lösungen sich auch strukturell so weit ähneln, dass die automatische Korrektur sie erkennen kann.

Wenn Prolog-Aufgaben in einer Klausur verwendet werden, muss allerdings ein weiteres Problem bedacht werden: In das Anfrageinterface können beliebige Anfragen eingespeist werden, insbesondere auch solche aus anderen Aufgaben, die gegebenenfalls so konzipiert waren, dass sie ohne Zugriff auf einen Prolog-Interpreter zu lösen waren. Beispiele für solche geschlossenen Aufgaben sind: Ist folgender Ausdruck ein korrekter Prolog-Term? Welche der folgenden Aussagen treffen auf das gegebene Programm zu? Welches Ergebnis liefert folgende Anfrage?

In dem vorliegenden Kurs gab es auch solche Aufgaben und es konnte nicht ausgeschlossen werden, dass zur Beantwortung ein Prolog-Interpreter verwendet wurde. Der hier verfolgte Ansatz zur Verhinderung solcher Tricks war ein zeitlicher: Die Anzahl der Aufgaben ist so hoch gewählt, dass sie sämtlich unter Zeitdruck zu bearbeiten sind und das Kopieren eines Ausdrucks, der Aufruf einer Prolog-Aufgabe, das Einfügen und Bewerten des Ausdrucks und die Rückkehr zur ursprünglichen Aufgabe zu viel Zeit in Anspruch nehmen.

5.5 Erfahrungen, Fazit und Ausblick

Die hier skizzierte Verwendungsweise eines Prolog-Graders, der auch als einfache webbasierte Entwicklungsumgebung benutzt werden kann, hat sich sowohl für den Übungsbetrieb als auch die Verwendung in Klausuren in den letzten Jahren bewährt. Der Hauptnutzen des Grader-Einsatzes ist die gesteigerte Effizienz bei der Korrektur, die es Tutoren und Dozenten ermöglicht, sich auf unklare Fälle zu konzentrieren und dort verbesserte Rückmeldungen zu geben.

Die Teilnehmerinnen und Teilnehmer der Veranstaltung „Introduction to Artificial Intelligence and Logic Programming“ profitieren in mehrfacher Hinsicht vom Einsatz der Grader-Technologie. Zuächst stellt die Veranstaltung sie grundsätzlich vor hohe Herausforderungen: Viele der Studierenden haben kaum programmiertechnische Vorkenntnisse und interessieren sich gegebenenfalls eher für die neurowissenschaftlichen und psychologischen Aspekte des Cognitive-Science-Studiums. Die Veranstaltung findet im zweiten Fachsemester neben weiteren arbeitsintensiven Einführungsveranstaltungen statt und verlangt hohen Arbeitseinsatz. In diesem Umfeld ist das Ziel, auch praktische Kenntnisse und Fertigkeiten zu vermitteln, schwierig zu erreichen. Die konsequente Einbeziehung programmierpraktischer Aufgaben in Übungen und Klausuren verstärkt den Praxisbezug aller-

dings grundsätzlich und wird auf Auswertungsseite nur durch die automatisierte Bewertung handhabbar.

Literatur für dieses Kapitel

[BKW05] Christoph Beierle, Marija Kulaš und Manfred Widera. „A pragmatic approach to pre-testing Prolog programs“. In: *Applications of Declarative Programming and Knowledge Management*. Springer, 2005, S. 294–308.

[Bra+06] Erik Braun u. a. „Interactive Problem Solving in Prolog“. In: *arXiv preprint cs/0611014* (2006).

[Bra01] Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 2001.

[Bön+99] Carsten Bönnen u. a. „PLOT: Prolog-Online-Tutor“. In: *Osnabrück: Institut für Semantische Informationsverarbeitung* (1999).

[CR96] Alain Colmerauer und Philippe Roussel. „The birth of Prolog“. In: *History of programming languages—II*. ACM. 1996, S. 331–367.

[Pey+00] Christoph Peylo u. a. „A Web-based intelligent educational system for PROLOG“. In: *Proceedings of the International Workshop on Adaptive and Intelligent Web-Based Education Systems held in conjunction with ITS 2000 Montreal, Canada.* 2000, S. 85–96.

[Pey02] Christoph Peylo. *Wissen und Wissensvermittlung im Kontext von internetbasierten intelligenten Lehr- und Lernumgebungen*. Akademische Verlagsgesellschaft, 2002.

[Sch97] Rolf Schulmeister. *Grundlagen hypermedialer Lernsysteme*. Oldenbourg München, 1997.

6 Automatisierte Bewertung in der UML-Modellierung

Marianus Ifland und Frank Puppe

Zusammenfassung
Die Modellierung von UML-Klassen- und Aktivitätsdiagrammen ist ein wichtiges Thema im Informatikstudium und verwandten Studiengängen. Da deren Bewertung zeitaufwändig ist, andererseits das Stellen möglichst vieler Übungsaufgaben den Lernerfolg steigert, ist eine automatisierte Bewertung erstrebenswert. Während bei Klassendiagrammen ein textueller Vergleich mit einer Musterlösung im Vordergrund steht, kann man Aktivitätsdiagramme darüber hinaus in Programmcode übersetzen und ausführen. Wir präsentieren Übungssysteme zur Modellierung von Klassen- und Aktivitätsdiagrammen für Studierende mit generiertem Feedback auf der Basis von Musterlösungen sowie deren Evaluation im Rahmen von Übungen zur Vorlesung Softwaretechnik der Universität Würzburg.

6.1 Einleitung

Die Programmierausbildung ist ein zentrales Element im Informatikstudium und informatiknaher Studiengänge. Dabei soll auch ein Denken über Programme auf höheren Abstraktionsebenen vermittelt werden. Dafür eignet sich die UML-Notation ([RJB10], [Bal05]). Mit Klassendiagrammen wird die Datenstruktur übersichtlich dargestellt, mit Aktivitätsdiagrammen kann die Programmlogik sowohl auf abstrakter als auch auf konkreter, ausführbarer Ebene modelliert werden. Die visuelle Darstellung erleichtert dabei das Verständnis und abstrahiert von syntaktischen Details der Programmiersprache. Der erste Schritt bei der Modellierung ist meist die Definition der Datenstrukturen mit Klassendiagrammen, die auch einfach auf Entity-Relationship-Diagramme für den Datenbankentwurf abgebildet werden können. Dabei werden neben Klassen und deren Attributen und Relationen (Assoziationen und Vererbungsbeziehungen) auch die wichtigsten Operationen eingetragen, die die Grundlage für die Definition der Aktivitätsdiagramme

darstellen. Letztere sind ein gutes didaktisches Mittel, um das Denken über große, aber auch kleine Programme, wie sie in Anfängerlehrveranstaltungen genutzt werden, zu fördern.

Wir präsentieren Übungsprogramme für die Bewertung von Klassen- und Aktivitätsdiagrammen und deren Evaluation in Kap. 6.2 und 6.3 (eine ausführliche Darstellung findet sich in [Ifl14]). Eine besondere Schwierigkeit ist die Bewertung von Folgefehlern. Daher basiert die Bewertung von Aktivitätsdiagrammen auf vordefinierten Klassendiagrammen und noch nicht auf einer Kopplung beider Bewertungen, was auch unserer Praxis beim Stellen von Übungsaufgaben entspricht.

6.2 Das Einsatzszenario „Automatische Bewertung von Klassendiagrammen“

6.2.1 Stand der Forschung

Einen Ansatz zur Bewertung von Klassendiagrammen, der auf dem Vergleich zu einer Musterlösung basiert, stellen [ASI07] mit dem UML Class Diagram Assessor (UCDA) vor. Dabei werden die Klassendiagramme von Dozenten und Studierenden mit dem kommerziellen UML-Werkzeug (Rational Rose) erstellt. Die von UCDA zur Überprüfung der Lernerlösungen anhand der Musterlösung verwendete Softwarekomponente besteht aus drei Modulen, die sequenziell ausgeführt werden, wobei jedes Modul den Studierenden im Falle von Fehlern Feedback gibt. Dabei wird das zweite (bzw. dritte) Modul nur dann ausgeführt, wenn das erste (bzw. zweite) keine Fehler liefert. Studierende können dabei ihr Diagramm iterativ verbessern. Zuerst überprüft das Class Structure Analysis Module die Lernerlösung beispielsweise auf die korrekte Anzahl von Klassen, die korrekte Anzahl von Attributen innerhalb von Klassen oder die Korrektheit von Parametertypen. Das Verification Process Module generiert Fehlermeldungen bezüglich der Relationen zwischen den Klassen, also ob die Anzahl der Relationen insgesamt korrekt ist, ob die Anzahl der Relationen bestimmten Typs (Assoziation, Generalisierung, Aggregation, Komposition) korrekt ist und ob konkrete Relationen zwischen Klassen vorhanden sind. Das Language Checking Module überprüft, ob bei der Benennung von Klassen und Attributen Substantive und bei der Benennung von Operationen Verben verwendet wurden. Eine Möglichkeit, wie mehrere Musterlösungen verwendet werden können, wird nicht gezeigt. Von einer Evaluation wird nicht berichtet.

[Sol+10] stellen einen Ansatz zur Klassendiagrammbewertung vor, der in das webbasierte ACME-DB Framework integriert ist. Dabei handelt es sich um eine E-Learning-Plattform der Universität Girona (Spanien), welche in Kursen zu Datenbanken eingesetzt wird. Es können mehrere Musterlösungen pro Aufgabe hinterlegt werden, wobei diese direkt in einer XML-ähnlichen Struktur eingegeben werden. In einem webbasierten Editor können Lernende dann zu den gestellten Aufgaben Klassendiagramme eingeben und prüfen lassen. Nach einer Prüfung durch das Korrekturmodul werden den Lernenden entsprechende Fehler in ihrem Klassendiagramm in Textform mitgeteilt. Fehler sind beispielsweise eine unkorrekte Anzahl an Klassen, falsch benannte Klassen oder falsche Werte bei Kardinalitäten. Nach einer Prüfung kann die eigene Lösung weiter verbessert werden, so dass sich Studierende in einem iterativen Prozess der Musterlösung annähern. Es bleibt unklar, wie das Korrekturmodul den Vergleich von Lernerlösung zu Musterlösungen durchführt, also beispielsweise wie erkannt wird, ob eine bestimmte Klasse aus der Musterlösung auch in der Lernerlösung vorkommt. Ebenfalls unklar bleibt, wie entschieden wird, welche Musterlösung bei einer Prüfung als Referenzlösung verwendet wird. Das System wurde in einem Parallelgruppenvergleich mit 48 Studierenden evaluiert. Eine Gruppe bereitete sich mit 4 Aufgaben aus der ACME-Lernplattform auf eine Prüfung vor, während die andere Gruppe ermutigt wurde, die Aufgaben per Hand zu lösen und bei Fragen das Büro des Dozenten aufzusuchen. Die Prüfung beinhaltete dabei eine Aufgabe, die ähnlich zu den zuvor gestellten Aufgaben war. Die Gruppe, welche die Lernplattform benutzen sollte, erzielte dabei im Durchschnitt leicht bessere Ergebnisse als die zweite Gruppe, doch waren die Unterschiede nicht signifikant. Die Eindrücke der Studierenden wurden insgesamt als positiv beschrieben.

Einen Ansatz ohne explizite Musterlösungen verfolgen [SG11a]. Klassendiagramme werden hier als formale Graphen interpretiert, welche mit einer geeigneten Anfragesprache, einer sogenannten Graph Query Language, untersucht werden können. Als Anfragesprache wird GReQL verwendet, mit welcher Anfragen gestellt werden können, die über die Existenz oder Nichtexistenz von Elementen anhand deren Typs oder deren Eigenschaften Auskunft geben. Um eine Aufgabe zu stellen, muss nun seitens des Aufgabenstellers neben der Domänenbeschreibung eine Reihe von Regeln mittels solcher Abfragen definiert werden. Dabei muss bei jeder Abfrage markiert werden, ob es sich um ein gewünschtes oder nicht gewünschtes Element handelt, damit bei der automatischen Korrektur entsprechende Fehlermeldungen gegeben werden können. Die Regeln können dabei logisch kombiniert werden, so dass beispielsweise die Akzeptanz alternativer Schreibweisen abgebildet werden kann. Es ist möglich, mit diesen Regeln auch stilistische Eigenschaften der Lernerlösung zu prüfen, beispielsweise ob die

Namen aller vorkommenden Klassen mit Großbuchstaben beginnen. Da Dozenten, die Aufgaben zu UML-Klassendiagrammen stellen, GReQL im Allgemeinen wohl nicht beherrschen, müssen diese Regeln also von entsprechenden Experten eingegeben werden. Es wird hier nur die Gleichheit der entsprechenden Namen geprüft. Alternative Schreibweisen, die beispielsweise durch Flexion oder Tippfehler entstehen, müssten in den Regeln explizit angegeben werden. Für eine Beispielaufgabe mussten 17 solche Regeln definiert werden. 5 davon sind allerdings allgemeine Regeln, die auch in anderen Aufgaben weiter verwendet werden können, so dass speziell für diese Aufgabe 12 Regeln definiert werden mussten. Für die Auswertung, also die Berechnung der Punktzahl für eine Lernerlösung, gibt es nun zwei Strategien. Entweder wird von 0 Punkten ausgegangen und Studierende erhalten Punkte für jede Regel, die nach einem gewünschten Element sucht und dieses auch findet (pessimistischer Ansatz). Oder es wird von der maximalen Punktzahl ausgegangen und den Studierenden werden für jede Regel Punkte abgezogen, die nach einem gewünschten Element sucht und dieses nicht findet (optimistischer Ansatz). Der Punktwert muss bei der Definition der entsprechenden Regel hinterlegt werden. In einem Experiment, bei welchem der pessimistische Ansatz verwendet wurde, wurden sechs Gruppen von Studierenden eine Aufgabe vorgelegt, die diese in Teamarbeit lösten. Die automatischen Bewertungen der Lernerlösungen reichten von 67 bis 91 von 100 Punkten. Diese Bewertungen lagen im gleichen Bereich der Bewertungen, die ein menschlicher Korrektor vorgenommen hatte, dem die Regeln zur automatischen Bewertung nicht bekannt waren. Über die Korrelation der Bewertungen der beiden Methoden wird allerdings keine Aussage gemacht. Es zeigte sich außerdem, dass die definierten Regeln nicht alle Aspekte abdecken konnten, die bei einer manuellen Bewertung betrachtet würden. Dies wird nicht als Problem der Technik beschrieben, sondern als Problem der Tatsache, dass die Regeln geschrieben werden, ohne die Lernerlösungen zu kennen. Es ist also nötig, die Regeln in einem iterativen Prozess nach der Bewertung einiger Lernerlösungen anzupassen. Unter der Voraussetzung, dass eine praxiserprobte Menge an allgemeinen Regeln besteht und dass Dozenten die Fähigkeit besitzen, in akzeptabler Zeit gute Regeln zu erstellen, verspricht dieser Ansatz gute Ergebnisse. Positiv hervorzuheben ist, dass anhand der Anfrageergebnisse ein eindeutiges Feedback, beispielsweise in natürlichsprachiger Form, generiert werden kann, in welchem mitgeteilt wird, welche Regeln verletzt wurden. Das Fehlen einer Musterlösung im Feedback ist allerdings durchaus als Nachteil zu sehen. Ebenfalls bleibt die Problematik verschiedener Schreibweisen von Elementen ohne deren explizite Angabe ungelöst, wobei die Autoren andeuten, dass es eine Möglichkeit gibt, GReQL um entsprechende Funktionen zu erweitern. Als problematisch anzusehen ist die komplex erscheinende Eingabe der Regeln, die

eine Hürde für Dozenten darstellen kann, ein entsprechendes Trainingssystem einzusetzen.

6.2.2 Eigener Ansatz

Wir verwenden ein Überdeckungsmaß zum Vergleich von Lerner- und Musterlösung. Dabei ist ein Klassendiagramm D ein 6-Tupel $(K_D, V_D, A_D, M_D, T_D, O_D)$ mit

- K_D: Menge aller in D enthaltenen Klassen,
- V_D: Menge aller in D enthaltenen Vererbungsbeziehungen,
- A_D: Menge aller in D enthaltenen Assoziationen,
- M_D: Menge aller in D enthaltenen Multiplizitäten ($|M_D| = |A_D|$),
- T_D: Menge aller in D enthaltenen Attribute,
- O_D: Menge aller in D enthaltenen Operationen.

Beim Vergleich eines vom Lerner erstellten Klassendiagramms mit einer Musterlösung wird zunächst ein Zuordnungsproblem gelöst, dass möglichst jedem Element des Lernerklassendiagramms einem Element der Musterlösung zuordnet. Anschließend werden die Zuordnungen mit einem Bewertungsmaß B für jedes Tupelelement mit einer Zahl zwischen 0 und 1 bewertet:

- $B_K : K \times K \mapsto [0..1]$,
- $B_V : V \times V \mapsto [0..1]$,
- $B_A : A \times A \mapsto [0..1]$,
- $B_M : M \times M \mapsto [0..1]$,
- $B_T : T \times T \mapsto [0..1]$,
- $B_O : O \times O \mapsto [0..1]$.

Die einzelnen Bewertungstypen werden für die Tupelelemente gewichtet. Der Gesamtscore kann als Prozentsatz erzielter Punkte zu maximal möglichen Punkten berechnet werden. Alternativ kann er auch als Abzugsmodell („optimistischer Ansatz“) von der Maximalpunktzahl umgesetzt werden, dass für jeden Fehler Punkte

abgezogen werden. Ein Fehler bedeutet hier, dass für ein Element aus der Musterlösung kein passendes Element der Lernerlösung gefunden wurde. Die Bewertungsfunktionen für Assoziationen und Vererbungsbeziehungen zwischen Klassen und für die Multiplizitäten sind leicht zu berechnen, falls die Klassen richtig erkannt wurden. Dagegen ist die Bewertung von Klassen-, Attribut- und Operatornamen schwieriger, da Studierende bei der Namensgebung nicht eingeschränkt sind. Um das Problem zu umgehen, werden die Aufgaben so gestellt, dass die jeweiligen Namen im Aufgabentext explizit genannt werden und die Studierenden darauf hingewiesen werden, wenn möglich Namen aus der Aufgabenstellung zu übernehmen. Trotzdem gibt es in Lösungen von Übungsaufgaben viele abweichende Schreibweisen. Um diese zuordnen zu können, wird die Ähnlichkeit von Zeichenketten zu den Texten der Musterlösung heuristisch berechnet, zu der auch Synonyme hinterlegt sein können, indem Standardtechniken wie Ignorieren von Groß-/Kleinschreibung, Stemming und String-Ähnlichkeit auf Basis der Levenshtein-Distanz benutzt werden. Falls Datentypen angegeben sind, werden Grundtypen wie Boolean, String, Zahl, Enumeration, usw. betrachtet. Die Ähnlichkeit von vordefinierten Varianten innerhalb dieser Grundtypen (z. B. für Zahlen: integer, int, double, long, BigDecimal, ...) wird wesentlich höher bewertet als zwischen verschiedenen Grundtypen. Zusätzlich wird bei der Bewertung der Ähnlichkeit von Attribut- und Operatornamen beachtet, ob diese in der gleichen Klasse vorkommen. Abb. 6.1 zeigt zwei Klassendiagramme und Abb. 6.2 die be-

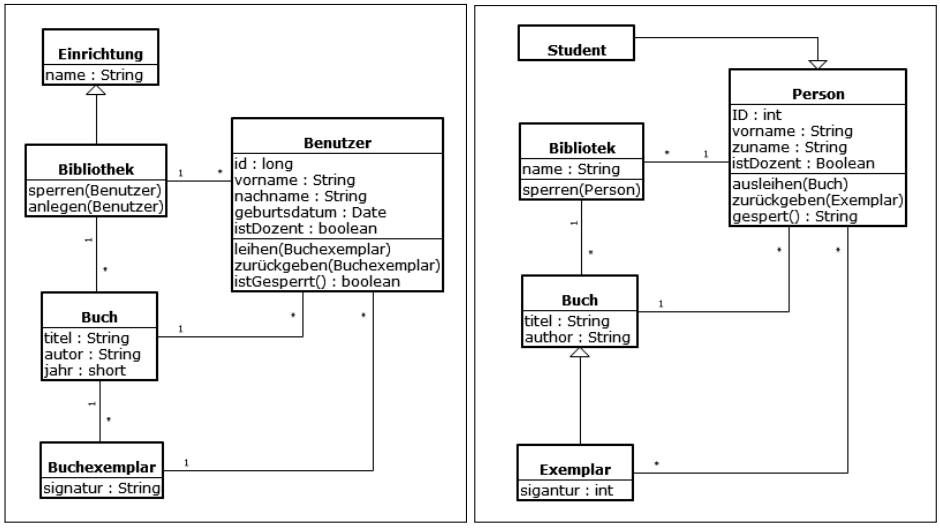


Abbildung 6.1: UML-Klassendiagramm: Beispiel einer Musterlösung (links) und Lernerlösung (rechts)

	A	B	C	D	E	F
1		**Musterlösung**	**Ihre Lösung**	**max. Punkte**	**Bewertung**	**Ihre Punkte**
2	**Klassen**			**10**	**80%**	**8,0**
3	Einrichtung		-	2	0%	0
4	Bibliothek		Bibliotek	2	100%	2
5	Benutzer		Person	2	100%	2
6	Buch		Buch	2	100%	2
7	Buchexemplar		Exemplar	2	100%	2
8						
9	**Generalisierungen:**			**1**	**0%**	**0,0**
10	Bibliothek -> Einrichtung		-	1	0%	0
11						
12	**Assoziationen:**			**10**	**80%**	**8,0**
13	Bibliothek <-> Benutzer		Bibliotek <-> Person	2	100%	2
14	Bibliothek <-> Buch		Bibliotek <-> Buch	2	100%	2
15	Buch <-> Benutzer		Buch <-> Person	2	100%	2
16	Buch <-> Buchexemplar		-	2	0%	0
17	Buchexemplar <-> Benutzer		Exemplar <-> Person	2	100%	2
18						
19	**Kardinalitäten:**			**5**	**50%**	**2,5**
20	Bibliothek <-> Benutzer	1:*	*:1	1	0%	0
21	Bibliothek <-> Buch	1:*	1:*	1	100%	1
22	Buch <-> Benutzer	1:*	1:*	1	100%	1
23	Buch <-> Buchexemplar	1:*	-	1	0%	0
24	Buchexemplar <-> Benutzer	1:*	*:*	1	50%	0,5
25						
26	**Attribute:**			**10**	**64%**	**6,4**
27	Buch	titel : String	titel : String	1	100%	1,0
28	Buch	autor : String	author : String	1	100%	1,0
29	Buch	jahr : short	-	1	0%	0,0
30	Einrichtung	name : String	-	1	0%	0,0
31	Buchexemplar	signatur : String	sigantur : int	1	50%	0,5
32	Benutzer	id : long	ID : int	1	88%	0,9
33	Benutzer	vorname : String	vorname : String	1	100%	1,0
34	Benutzer	nachname : String	zuname : String	1	100%	1,0
35	Benutzer	geburtsdatum : Date	-	1	0%	0,0
36	Benutzer	istDozent : boolean	istDozent : Boolean	1	100%	1,0
37						
38	**Methoden:**			**5**	**70%**	**3,5**
39	Bibliothek	sperren(Benutzer)	sperren(Person)	1	100%	1,0
40	Bibliothek	anlegen(Benutzer)	-	1	0%	0,0
41	Benutzer	leihen(Buchexemplar)	ausleihen(Buch)	1	75%	0,8
42	Benutzer	zurückgeben(Buchexemplar)	zurückgeben(Exemplar)	1	100%	1,0
43	Benutzer	istGesperrt() : boolean	gespert() : String	1	75%	0,8
44						
45						
46	**Gesamt:**			**41**	**69%**	**28,4**

Abbildung 6.2: Beispiel eines Feedbackdokuments für Muster und Lernerlösung aus Abbildung 6.1

rechneten Ähnlichkeiten. Als Autorenwerkzeug wurde ein spezieller Editor innerhalb des an der Universität eingesetzten fallbasierten Trainingssystems CaseTrain [Hö+09] entwickelt (s. Abb. 6.3).

6.2.3 Evaluation

Das System wurde beim Einsatz in der Vorlesung Softwaretechnik der Universität Würzburg im Sommersemester 2013 und 2014 evaluiert. Dabei zeigen wir zunächst den Effekt des komplexen Zuordnungsmaßes im Vergleich zu einem ein-

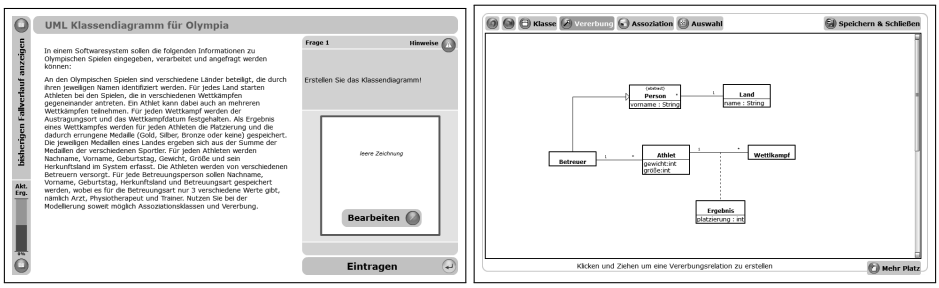


Abbildung 6.3: Editor zum freien Zeichen von UML-Klassendiagrammen in CaseTrain (links: Aufgabenstellung, rechts: Editor, der nach Klicken auf den Button „Bearbeiten“ im linken Fenster erscheint).

	Anz.	RP	FP	FN	Precision	Recall	F_1-Maß	Fehlerr.
SoSe13 trivial	2849	1699	0	291	100 %	85,4 %	92,6 %	10,2 %
SoSe13 komplex	2849	1969	16	21	99,2 %	98,9 %	99,1 %	1,3 %
SoSe14 trivial	1976	1282	0	278	100 %	82,2 %	90,2 %	14,1 %
SoSe14 komplex	1976	1540	24	20	98,5 %	98,7 %	98,6 %	2,2 %
gesamt trivial	4825	2981	0	569	100 %	84,0 %	**91,3 %**	**11,8 %**
gesamt komplex	4825	3509	40	41	98,9 %	98,8 %	**98,9 %**	**1,7 %**

Tabelle 6.1: Ergebnisse der manuellen Evaluation der Zuordnungen von 2849 Klassen im Sommersemester 2013 und 1976 Klassen im Sommersemester 2014 (RP=Richtig Positiv; FP=Falsch Positiv; FN=Falsch Negativ).

fachen (trivialen) Zuordnungsmaß (Tab. 6.1), das eine exakte Übereinstimmung der Namen von Klassen, Attributen und Methoden verlangt, wie sie in der Aufgabenstellung explizit gefordert wurden. Dabei sind Assoziationen, Vererbungen und Kardinalitäten wie oben erwähnt einfach zu bewerten und werden hier nicht betrachtet. Als Goldstandard diente eine manuelle Zuordnung von Dozenten. Das triviale Maß hat eine Precision von 100% (wenn zwei Namen identisch sind, ist die Zuordnung korrekt), aber nur einen Recall von 84%, d. h. es werden viele Zuordnungen nicht erkannt. Mit dem komplexen Maß, das Namenszuordnungen aufgrund der oben erwähnten Heuristiken berechnet, steigt der Recall um 14,4 Prozentpunkte auf 98,8% an, wobei die Precision nur um 1.1 Prozentpunkte auf 98,9% fällt.

Obwohl das Programm nicht perfekt arbeitet, wurde es von den Studierenden akzeptiert. In einer Umfrage im Sommersemester 2013, die zwei Wochen nach

Aspekt	Ø Schulnote (1-6)
Konzept des Tools, unabhängig von der Implementierung	2,25
Umsetzung des Tools, Gesamtnote	2,61
Aspekt Bedienbarkeit (Einarbeitung, intuitive Bedienbarkeit)	2,47
Aspekt Technik (Verfügbarkeit, Robustheit, Effizienz)	3,57
Aspekt Inhalt der Übungsaufgaben	2,18
Aspekt Fairness der Bewertung der Übungsaufgaben	2,59
Aspekt hilfreiche Erklärungen als Text oder nützliches Feedback	2,95

Tabelle 6.2: Umfrageergebnisse in Schulnoten zu verschiedenen Aspekten des freien Zeichnens von Klassendiagrammen bei Studierenden im Sommersemester 2013 (n=240)

Bearbeitung der Aufgabe durchgeführt wurde, wurde das Programm mit Schulnoten zwischen gut und befriedigend hinsichtlich der verschiedenen Aspekte bewertet; lediglich die Technik wurde um eine Notenstufe schlechter bewertet, da das Programm noch nicht robust genug war (Tab. 6.2). Die Umfrage 2014 war breiter angelegt und daher nicht direkt mit der von 2013 vergleichbar, hatte aber ähnliche Bewertungen (bis auf die Technik, die um eine Notenstufe besser ausfiel). 90% befürworteten einen weiteren Einsatz des Trainingssystems für UML-Klassendiagramme.

6.2.4 Diskussion und Verbesserungen

Die Evaluationen haben gezeigt, dass es noch viel Potenzial für Verbesserungen gibt. Dazu gehört der regelbasierte Ansatz von [SG11a], der spezifische Schwächen des Überdeckungsansatzes ausgleichen kann:

- Regeln, die Konventionen in UML bewerten, zum Beispiel dass Klassennamen Substantive im Singular sein sollen.

- Regeln, die zu viele Angaben wie zum Beispiel mehr Attribute oder Methoden, als in der Musterlösung angegeben, und die im Allgemeinen toleriert werden, in speziellen Fällen negativ bewerten (z. B. überflüssige Assoziationen zwischen Klassen).

- Regeln, die für typische Fehler in der Aufgabe ein angemessenes Feedback geben, zum Beispiel wenn in Lernerlösungen Vererbungsbeziehungen fehlen (die Überdeckungsmetrik liefert da oft zu schlechte Bewertungen).

- Spezielle Regeln für Besonderheiten der Aufgabenstellung, dass man zum Beispiel zwar Buchexemplare, aber keine Bücher ausleihen kann, wenn beide Klassen im Klassendiagramm vorkommen.

Weitere Verbesserungen beinhalten verfeinerte Verfahren zur Berechnung der Levenshtein-Distanz zwischen Namen, das Erkennen von Teilwörtern in Komposita, das zum Beispiel auch für die Angabe von Synonymen für Teilwörtern genutzt werden kann, oder die variable Zugehörigkeit von Operationen zu verschiedenen Klassen, um die aufwändige Angabe verschiedener Musterlösungen zu verringern.

6.3 Das Einsatzszenario „Automatische Bewertung von Aktivitätsdiagrammen“

Das Ziel der Modellierung von Aktivitätsdiagrammen ist, die wesentliche Funktionalität des Programms übersichtlich darzustellen. Bei großen Programmen ist eine starke Abstraktion erforderlich. Kleine Programme können dagegen auch in vollem Umfang exakt abgebildet werden. Beides erlaubt die Syntax der Aktivitätsdiagramme in UML. Typische Lernziele beim Einüben der Erstellung von Aktivitätsdiagrammen sind:

- Erstellen von syntaktisch korrekten Aktivitätsdiagrammen,
- Verwendung von Bedingungen und entsprechenden Verzweigungen,
- Verwendung von Schleifen,
- Übergabe und Rückgabe von Parametern,
- Kapselung häufig verwendeter Subroutinen,
- Zugriff auf Objektattribute,
- Verständnis des Zusammenhangs von Diagrammen und Programmcode,
- algorithmisches Problemlösen.

Durch die Verwendung von Subroutinen, deren konkretes Verhalten vordefiniert ist, lassen sich Aktivitätsdiagramme bei geeigneter Syntax direkt in lauffähigen Programmcode übersetzen.

6.3.1 Stand der Forschung

Für die einzelnen Aspekte gibt es viel Literatur. So sind in der Programmierausbildung Systeme, die eingegebenen Programmcode überprüfen und entsprechendes Feedback generieren, bereits etabliert. Ähnliches gilt für die automatische Codegenerierung aus UML. [UN09] stellen ein Werkzeug namens „UJECTOR“ vor, das UML-Klassen-, Sequenz-, und Aktivitätsdiagramme in Java-Code übersetzen kann. Das Werkzeug wird mit bestehenden kommerziellen und quelloffenen Werkzeugen verglichen. Es wird belegt, dass der generierte Programmcode funktionsfähig und verständlich ist. Ein weiteres Werkzeug, welches UML-Diagramme in Java-Code übersetzt, ist „Fujaba“ [NNZ00]. Der Name ist ein Akronym für „From UML to Java and back again“. Auch eine Rückübersetzung des modifizierten Codes in Diagramme ist damit möglich. [GR11] stellen ebenfalls einen Ansatz zur Generierung von Programmcode aus Aktivitätsdiagrammen vor. Programmieren mit Aktivitätsdiagrammen kann als visuelle Programmierung aufgefasst werden. Dabei werden Programme in zwei- oder mehrdimensionaler Weise spezifiziert [Mye90]. Bekannte Systeme zur Programmierausbildung mit visueller Programmierung und Programmvisualisierung sind „Alice“, „Greenfoot“ und „Scratch“. Alice ist eine um das Jahr 2000 erschienene Programmiersprache mit zugehöriger Entwicklungsumgebung, mit der das Erscheinungsbild und das Verhalten von Personen und Objekten in einer virtuellen dreidimensionalen Welt modifiziert werden kann [CDP00]. Ein Drag-and-Drop-Konzept verhindert hier die Erstellung syntaktisch unkorrekter Programme. Alice wird an „hunderten Hoch- und Sekundarschulen“ eingesetzt [Fin+10]. Einige Jahre später wurde Greenfoot [Kö10] veröffentlicht, eine Entwicklungsumgebung mit didaktischem Fokus, die darauf spezialisiert ist, interaktive, grafische Anwendungen zu erstellen. Die Programme werden hier direkt mit Java-Code entwickelt, die Ausführung wird visualisiert, wobei parallel die Vererbungsstruktur der beteiligten Klassen angezeigt wird. Im Jahr 2007 erschien Scratch [Mal+10], eine visuelle Programmierumgebung für den schulischen Einsatz, deren Zielgruppe bei Kindern und Jugendlichen von 8 bis 16 Jahren liegt. Der Fokus liegt dabei auf dem Umgang mit Medien, also mit Bildern, Geräuschen, Musik und Videos. Scratch ist mittlerweile weit verbreitet; laut Angaben auf der offiziellen Webseite waren im August 2016 fast 13 Millionen Nutzer registriert[1]. Auch für die Programmierung von Anwendungen für mobile Endgeräte („Apps“) wurde mit dem Google „App Inventor“ bereits eine visuelle Programmierumgebung in der Lehre eingesetzt [WS11].

1 http://scratch.mit.edu/statistics/ (August 2016)

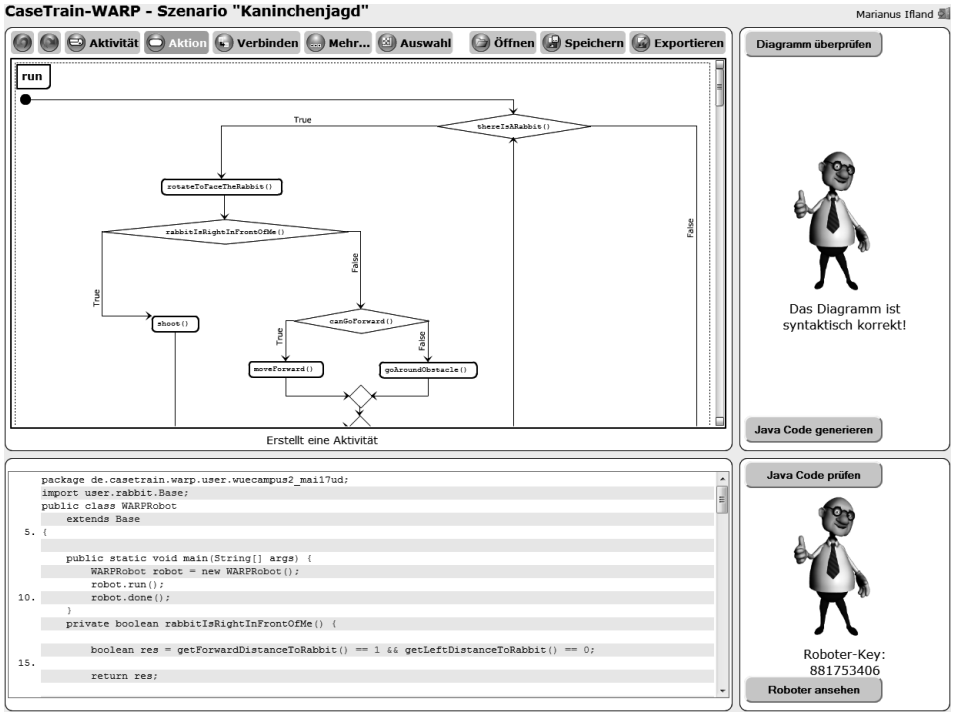
6.3.2 Eigener Ansatz

Im hier vorgestellten Trainingssystem „WARP“ (Würzburger Aktivitätsdiagramm Roboter Programmierung) [Ifl+14a] werden Aktivitätsdiagramme zur Definition des Verhaltens eines virtuellen Roboters erstellt und direkt in Java-Code übersetzt. Dieser Java-Code wird in einer vordefinierten Simulationsumgebung ausgeführt, wobei der Roboter eine bestimmte Aufgabe erfüllen soll. Dabei wird den Lernenden Feedback in vier Ebenen angeboten:

- Erste Feedbackebene: Falls das Diagramm syntaktisch nicht valide ist, werden Fehler direkt im Diagramm mit einem textuellen Kommentar angezeigt (Abb. 6.4, oben).
- Zweite Feedbackebene: Falls der aus dem Diagramm automatisch erzeugte Java-Code syntaktische Fehler hat, werden entsprechende Hinweise generiert (Abb. 6.4, unten).
- Dritte Feedbackebene: Der Java-Code wird in der Robotersimulationsumgebung RoSE [Her13] ausgeführt und eine Animation erzeugt, welche die Studierenden schrittweise abspielen können (Abb. 6.5).
- Vierte Feedbackebene: Die Studierenden erhalten eine Rückmeldung, ob die Aufgabe korrekt gelöst wurde. Zudem werden Metriken über die primitiven Roboteraktionen angezeigt, zum Beispiel wie viele Schritte der Roboter gebraucht hat.

WARP wurde als Webanwendung implementiert, die sich leicht an Lernmanagementsysteme wie Moodle koppeln lässt. Der Editor besteht aus vier Teilen. Zunächst gibt der Benutzer in einer grafischen Oberfläche ähnlich zu dem UML-Klassendiagrammeditor (s. Abb. 6.3) sein Aktivitätsdiagramm ein. Dabei werden folgende Knoten unterstützt:

- Startknoten und Endknoten,
- Aktionsknoten (mit konkretem Java-Code),
- Verzweigungsknoten und Verbindungsknoten,
- strukturierte If-Then-Knoten,
- strukturierte Schleifenknoten: For-Do-Knoten, Do-While-Knoten, While-Do-Knoten.




```
package de.casetrain.warp.user.wuecampus2_mai17ud;
import user.rabbit.Base;
public class WARPRobot
    extends Base
{

    public static void main(String[] args) {
        WARPRobot robot = new WARPRobot();
        robot.run();
        robot.done();
    }
    private boolean rabbitIsRightInFrontOfMe() {

        boolean res = getForwardDistanceToRabbit() == 1 && getLeftDistanceToRabbit() == 0;

        return res;
```

Abbildung 6.4: CaseTrain-WARP Editor: Vier Bereiche des zentralen Dialogs, die zur Erstellung der Aktivitätsdiagramme dienen. Links oben: grafischer Editor, rechts oben: Feedback zur Syntax des Diagramms, links unten: automatisch generierter Java-Code, rechts unten: Feedback zur Syntax des Java-Codes. Eventuelle Fehler im grafischen Editor oder im generierten Code würden links rot markiert und rechts kommentiert. Wenn der Java-Code korrekt ist, wird ein entsprechendes Roboterprogramm für eine „Arena“ erstellt, die sich der Benutzer anschauen kann (s. Abb. 6.5 mit komplexerer Aufgabe der Kaninchenjagd).

Das Programm überprüft in der ersten Feedbackebene nur die syntaktische Korrektheit des gezeichneten Diagramms, zum Beispiel ob alle Knoten die passende Anzahl von eingehenden und ausgehenden Kanten haben. Wenn das Programm syntaktisch korrekt ist, kann es der Benutzer in Java-Code übersetzen lassen. Dabei wird der erzeugte Java-Code auf syntaktische Korrektheit überprüft und gegebenenfalls Fehler markiert und kommentiert. Wenn der Java-Code korrekt ist, kann sich der Benutzer einen Roboter (d. h. ein ausführbares Programm) erzeugen, den er in einer Arena (Abb. 6.5) ausführen kann. Die Ausführung kann schrittwei-

se gesteuert werden, damit der Benutzer sich das Verhalten an kritischen Stellen genauer anschauen kann, um eventuelle Laufzeitfehler zu finden. Schließlich werden als vierte Feedbackebene noch Metriken erzeugt, die Aussagen über die Programmqualität machen (z. B. wie viele Schritte zur Erledigung der Aufgabe benötigt wurden). Als eine spezielle Art von Metrik gibt es auch Wettbewerbsspiele, bei denen zwei Roboter von zwei Benutzern im Wettbewerb gegeneinander antreten (in Abb. 6.5 sollen sie ein sich zufällig bewegendes Kaninchen jagen).

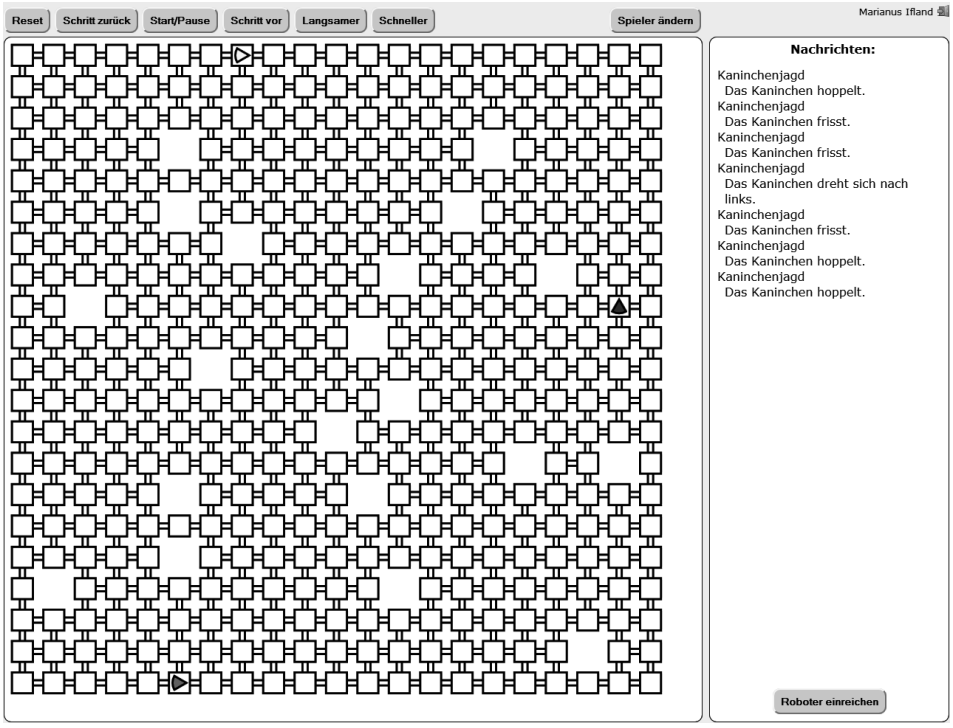


Abbildung 6.5: CaseTrain-WARP Arena: Die Abbildung zeigt einen Zustand der Umgebung eines Multiplayer-Szenarios, in welchem die Aufgabe für die beiden Spieler (in der Arena mit unterem rotem und oberen gelben Pfeil markiert) darin besteht, sich zu einem sich zufällig bewegenden Kaninchen (blauer Pfeil, Mitte rechts) zu bewegen und es dann zu erlegen. Auf der rechten Seite werden Statusmeldungen angezeigt.

6.3.3 Evaluation

Im Sommersemester 2013 wurden in der Vorlesung Softwaretechnik an der Universität Würzburg Aufgaben zu 3 Szenarien angeboten: Zeichnen eines Rechtecks als Einstiegsaufgabe, Markieren jedes begehbaren Feldes einer Arena mit nichtbegehbaren Feldern („Staubsauger“), Finden eines Weges zu einem vorgegebenen Ziel („Labyrinth“). Im Sommersemester 2014 kamen noch zwei anspruchsvollere Aufgaben dazu: die Staubsauger-Aufgabe wurde um stochastische Elemente erweitert. Die Labyrinth-Aufgabe wurde so erweitert, dass ein bewegliches Ziel („ein Kaninchen“) gejagt werden muss, dessen Position immer abgefragt werden kann. Letztere Aufgabe wurde auch im Wettbewerb zweiter Roboter angeboten.

Die Studierenden konnten Aufgaben sooft abgeben, wie sie wollten. 2013 wurden für ca. 250 verschiedene Studierende für die drei Aufgaben mehr als 12.000 Aktivitätsdiagramme geprüft. Insgesamt konnten die einfachen Aufgaben (Rechteck, Staubsauger, Labyrinth) von ca. 90% der Studierenden 2013 und 98% 2014 erfolgreich gelöst werden. Die schwereren Aufgaben in 2014 (Staubsauger mit Zufallselementen, Kaninchenjagd) wurden von weniger Studierenden bearbeitet, aber auch noch mit knapp 90% erfolgreich gelöst (s. Tab. 6.3).
Die subjektiven Bewertungen der Studierenden aus Umfrageergebnissen zeigen ein gemischtes Bild (s. Tab. 6.4). Während 2013 die Bewertungen ähnlich wie bei den UML-Klassendiagrammen zwischen den Schulnoten gut und befriedi-

2013	**RE**	**ST**	**LAB**	**gesamt**
Anzahl Einreichungen	244	228	206	678
Anteil bestandener Einreichungen	91,8 %	91,7 %	87,3 %	90,4 %
Ø Anzahl Aktivitäten	2,9	3,0	2,4	2,8
Ø Anzahl Knoten	12,8	20,3	13,1	15,4
Ø Anzahl Kanten	11,6	16,3	11,0	13,0

2014	**RE**	**ST**	**STZ**	**LAB**	**KAN**	**gesamt**
Anzahl Einreichungen	256	243	209	221	144	1073
Anteil bestandener Einreichungen	98,0 %	98,8 %	89,5 %	98,2 %	88,2 %	95,3 %
Ø Anzahl Aktivitäten	2,5	3,0	3,8	2,3	3,4	2,9
Ø Anzahl Knoten	17,2	21,5	39,3	12,9	34,5	23,2
Ø Anzahl Kanten	15,0	17,9	30,9	9,9	18,5	18,2

Tabelle 6.3: Erfolgsquoten der Bearbeitungen der drei Aktivitätsdiagrammaufgaben im Sommersemester 2013 (oben) und 2014 (unten) mit Anzahl der Aktivitäten (die Studierenden sollten große Aktivitätsdiagramme in mehrere kleine aufteilen) sowie der Knoten und Kanten pro Aufgabe. (RE = Rechteck, ST = Staubsauger, STZ = Staubsauger mit Zufallselementen, LAB = Labyrinth, KAN = Kaninchen)

Aspekt	2013	2014
Konzept des Tools, unabhängig von der Implementierung	2,28	1,91
Umsetzung des Tools, Gesamtnote	2,79	3,24
Bedienbarkeit (Einarbeitung, intuitive Bedienbarkeit)	2,76	3,59
Technik (Verfügbarkeit, Robustheit, Effizienz)	3,81	3,87
Inhalt der Übungsaufgaben	2,19	2,66

Tabelle 6.4: Ergebnisse der Benutzerumfrage; Mittelwerte von Schulnoten (1-6)

gend liegen (mit Ausnahme wiederum der Technik mit der Note 3,81, die auf mangelnde Robustheit des Programms zurückzuführen ist), haben sich 2014 die Noten insgesamt verschlechtert. Ein möglicher Grund sind die anspruchsvolleren Aufgaben, die auch zu mehr Problemen bei der Technik geführt haben. Das wurde insofern anerkannt, dass das Konzept im SoSe 2014 mit einer besseren Note (1,91) bewertet wurde. Ähnlich wie bei dem UML-Tool hat sich die große Mehrheit für die weitere Nutzung elektronischer Korrekturtools ausgesprochen (2014: 78%). Allerdings wurden auch zahlreiche Verbesserungen angesprochen.

6.3.4 Diskussion und Verbesserungen

Da die Syntax von Aktivitätsdiagrammen weitgehend vorgegeben ist, beziehen sich die Verbesserungen aufgrund der bisherigen Erfahrungen mit dem Einsatz nicht auf die Vereinfachung der Sprache, sondern hauptsächlich auf pragmatische Aspekte des Editors:

- Variables Platzangebot im Editor: Es sollte für den Benutzer möglich sein, die Zeichenfläche nicht nur in der Tiefe, sondern auch in der Breite zu vergrößern. Wie beim Programmieren wachsen auch Aktivitätsdiagramme oft inkrementell in alle Richtungen.
- Verschieben von Elementen in Knoten-Container: Beim herkömmlichen Programmieren kann man mit Copy-and-Paste Code für ein Teilproblem an eine andere Stelle beliebig verschieben. Derzeit wird beim grafischen Programmieren das Verschieben von grafischen Strukturen direkt auf der Zeichenfläche unterstützt, aber nicht in andere Container und insbesondere in strukturierte Knoten hinein, deren Begrenzungen sich dann vergrößern müssten.
- Alternative Eingabe von Name und Ein-/Rückgabe-Parametern einer Aktivität: Momentan werden Name und Ein-/Rückgabe-Parametern einer Akti-

vität mittels der Eingabe einer Methodensignatur in Java-Syntax definiert. Da das Lernziel hauptsächlich in der Erstellung von Aktivitätsdiagrammen und nicht in der Erstellung von Programmcode liegt, sollte die Eingabe der Signaturen vereinfacht werden.

- Einbinden von vorgegebenen Befehlen: Die Eingabe vorgegebener aktorischer und sensorischer Befehle (z. B. Abfrage der Farbe eines Feldes) wird aktuell nur durch die Funktion des Auto-Vervollständigens unterstützt. Die Analyse der aufgetretenen syntaktischen Fehler im generierten Quellcode ergab, dass sich ein großer Teil (etwa 50 %) der meist durch Schreib- oder Tippfehler verursachten Fehler des Typs „Methode existiert nicht“ auf die vorgegebenen Befehle bezog. Beispielsweise wurde statt rotateRight() fälschlicherweise rotateright() oder turnRight() eingegeben. Diesen Fehlern kann vorgebeugt werden, indem für den Editor eine Funktion implementiert wird, mit der Anwender vorgegebene Befehle beispielsweise wie in Alice per Drag-and-Drop [CDP00] eingeben können.

6.4 Fazit und Ausblick

Die praktischen Erfahrungen im Einsatz zeigen, dass die Idee, den Studierenden Werkzeuge bereitzustellen, mit denen sie zu von ihnen erstellten UML-Klassen- und Aktivitätsdiagrammen automatisch generiertes Feedback bekommen, gut ankommt und die weitaus meisten eine Fortsetzung des Angebotes wünschen. Das galt trotz Problemen bei der Robustheit der Programme, was sich in den im Vergleich zu den übrigen Bewertungen um ca. eine Schulnote schlechteren Bewertungen zur Technik niederschlug. Sehr viele Verbesserungsvorschläge beziehen sich auf die Gestaltung des Editors. Bezüglich des automatischen Feedbacks ist die Übersetzung von Aktivitätsdiagrammen in ausführbaren Code, dessen Anwendung in einer Arena visualisiert wird, wesentlich attraktiver als der Vergleich von Klassendiagrammen mit Musterlösungen gemäß einer Überdeckungsmetrik. Diese sollte durch regelbasierte Bewertungen ergänzt werden, die eine höhere Flexibilität ermöglichen. Für Dozenten wäre eine Unterstützung bei der Korrektur von Prüfungsaufgaben sehr attraktiv, was allerdings erst möglich ist, wenn die Prüfungsaufgaben direkt am Computer geschrieben werden.

Literatur für dieses Kapitel

[ASI07] Noraida Haji Ali, Zarina Shukur und Sufian Idris. „A design of an assessment system for UML class diagram“. In: *Proceedings – The 2007 International Conference on Computational Science and its Applications, ICCSA 2007*. 2007, S. 539–544.

[Bal05] Heide Balzert. *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. Heidelberg: Spektrum, 2005.

[CDP00] Stephen Cooper, Wanda Dann und Randy Pausch. „Alice: a 3-D tool for introductory programming concepts“. In: *Journal of Computing Sciences in Colleges* 15 (2000), S. 107–116. DOI: 10.1145/1953163.1953243.

[Fin+10] Sally Fincher u. a. „Comparing Alice, Greenfoot & Scratch“. In: *41st ACM technical symposium on Computer science education*. 2010, S. 192–193. DOI: 10.1145/1734263.1734327.

[GR11] Dominik Gessenharter und Martin Rauscher. „Code Generation for UML 2 Activity Diagrams Towards a Comprehensive Model-Driven Development Approach“. In: *ECMFA'11 Proceedings of the 7th European conference on Modelling foundations and applications*. 2011, S. 205–220.

[Her13] Felix Hermann. *Erweiterung der Robotersimulationsumgebung RoSE auf Multiagentensysteme mit didaktischem Fokus*. Bachelorarbeit. 2013.

[Hö+09] Alexander Hörnlein u. a. „Konzeption und Evaluation eines fallbasierten Trainingssystems im universitätsweiten Einsatz (CaseTrain)“. In: *GMS Medizinische Informatik, Biometrie und Epidemiologie* 5.1 (2009). DOI: 10.3205/mibe000086.

[Ifl+14a] Marianus Ifland u. a. „WARP – ein Trainingssystem für UML-Aktivitätsdiagramme mit mehrschichtigem Feedback“. In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014.

[Ifl14] Marianus Ifland. *Feedback-Generierung für offene, strukturierte Aufgaben in E-Learning-Systemen*. Dissertation. 2014.

[Kö10] Michael Kölling. „The Greenfoot Programming Environment“. In: *ACM Transactions on Computing Education (TOCE)* 10.4 (2010).

[Mal+10] John Maloney u. a. „The Scratch Programming Language and Environment“. In: *ACM Transactions on Computing Education (TOCE)* 10 (2010), 16:1–16:15. DOI: 10.1145/1868358.1868363.

[Mye90] Brad A. Myers. *Taxonomies of visual programming and program visualization*. 1990. DOI: 10.1016/S1045-926X(05)80036-9.

[NNZ00] Ulrich Nickel, Jörg Niere und Albert Zündorf. „The FUJABA environment“. In: *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium* (2000). DOI: 10.1109/ICSE.2000.870485.

[RJB10] James Rumbaugh, Ivar Jacobson und Grady Booch. *The Unified Modeling Language Reference Manual (Paperback)*. The Addison-Wesley object technology series. ADDISON WESLEY Publishing Company Incorporated, 2010.

[SG11a] Michael Striewe und Michael Goedicke. „Automated checks on UML diagrams“. In: *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education – ITiCSE '11*. New York, USA: ACM Press, 2011, S. 38. DOI: 10.1145/1999747.1999761.

[Sol+10] Josep Soler u. a. „A web-based e-learning tool for UML class diagrams“. In: *IEEE EDUCON 2010 Conference*. Ieee, 2010, S. 973–979. DOI: 10.1109/EDUCON.2010.5492473.

[UN09] Muhammad Usman und Aamer Nadeem. „Automatic Generation of Java Code from UML Diagrams using UJECTOR“. In: *International Journal of Software Engineering and Its Applications* 3.2 (2009), S. 21–38.

[WS11] David Wolber und Fulton Street. „App Inventor and Real-World Motivation“. In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011, S. 601–606. DOI: 10.1145/1953163.1953329.

7 Automatisierte Bewertung im Kontext der App-Entwicklung am Beispiel Android

Britta Herres, Rainer Oechsle und David Schuster

Zusammenfassung
In diesem Kapitel wird das ASB-System (Automatische Softwarebewertung) der Hochschule Trier beschrieben, das seit mehreren Jahren für unterschiedliche Programmierlehrveranstaltungen eingesetzt wird. In den letzten Jahren wurde das System erweitert, um auch Android-Apps automatisch bewerten zu können. Aus Sicht der Studierenden ist die Einreichung einer Android-App und das Betrachten der Ergebnisse der Bewertungsmaßnahmen nicht wesentlich anders als für andere Programme. Wir beschreiben in diesem Kapitel, wie das ASB-System im Rahmen der unterschiedlichen Lehrveranstaltungen eingesetzt wird, wie die Studierenden und die Dozenten das System benutzen, welche Art von Vorgaben in den Programmieraufgaben vorkommen und welche Erfahrungen wir im Umgang mit dem System gemacht haben.

7.1 Einleitung

Im Fachbereich Informatik der Hochschule Trier wird seit 2006 das webbasierte System ASB (Automatische Softwarebewertung) zur Bewertung der studentischen Lösungen von Programmieraufgaben entwickelt und eingesetzt. Das ASB-System ermöglicht Dozenten, Programmieraufgaben zu ihren Vorlesungen zu erstellen, zu denen Studierende Lösungen hochladen können. Die von den Studierenden programmierten Lösungen werden durch mehrere Bewertungsmaßnahmen automatisch überprüft.

Neben einer Überprüfung der Einhaltung von Programmierkonventionen (z. B. Einrückungen) werden die studentischen Programme ausgeführt. Dabei wird getestet, ob die Programme sich so wie vorgegeben verhalten. Die Ergebnisse der

Bewertungen werden den Studierenden angezeigt. Studierende können ihre Programme daraufhin ändern und erneut zur Überprüfung einreichen, solange die Abgabefrist noch nicht abgelaufen ist. Das System wird seit mehreren Jahren für die Übungen mehrerer Module sowohl im Präsenz- als auch im Fernstudium eingesetzt. Eine frühe Version des ASB-Systems wurde in [Mor+07] beschrieben. Momentan arbeiten wir mit der Version 4, auf der dieser Beitrag basiert.

Relativ neu ist die Nutzung der automatischen Programmbewertung für das Bachelormodul *Entwicklung mobiler Anwendungen*, das im Sommersemester 2013 erstmalig angeboten wurde. Als Programmierplattform wird in diesem Modul Android verwendet. Um auch für dieses Modul die Lösungen von Programmieraufgaben automatisch bewerten zu können, wurde das ASB-System so erweitert, dass nun auch Studierende ihre in den Übungen entwickelten Android-Apps auf das ASB-System hochladen können und entsprechendes Feedback dazu bekommen. In diesem Beitrag wird der Einsatz des ASB-Systems im Allgemeinen und speziell für das Modul *Entwicklung mobiler Anwendungen* beschrieben.

7.2 Einsatzszenarien

Das ASB-System wird im Fachbereich Informatik der Hochschule Trier sowohl im Präsenz- als auch im Fernstudium genutzt. Im Präsenzstudium wird es momentan in vier und im Fernstudium in zwei Modulen eingesetzt. Im Folgenden wird die Einbettung des ASB-Systems in die Lehre etwas allgemeiner und unabhängig von den konkreten Modulen beschrieben, wobei jedoch zwischen dem Einsatz im Präsenz- und Fernstudium unterschieden wird.

Die Module des Präsenzstudiums, in denen ASB genutzt wird, bestehen wie die meisten Module der Informatik-Bachelorstudiengänge aus je einer wöchentlich abgehaltenen 90-minütigen Vorlesung (= 2 SWS [Semesterwochenstunden]) und einer wöchentlich durchgeführten 90-minütigen Übungsveranstaltung (= 2 SWS). Die Übungen werden in Gruppen mit jeweils ca. 20 – 30 Studierenden durchgeführt, wobei sich die Anzahl der Gruppen nach der voraussichtlichen Gesamtzahl der teilnehmenden Studierenden richtet. Jedes dieser Module wird mit 5 ECTS-Punkten kreditiert. Als Lernplattform für das Präsenzstudium wird an der Hochschule Trier *Stud.IP*[1] eingesetzt. Über *Stud.IP* werden den Studierenden Lehrmaterialien wie Vorlesungsunterlagen, Beispielprogramme und Übungsaufgaben von den Dozenten bereitgestellt. Außerdem wird das Forum von *Stud.IP* genutzt. Die Forumnutzung ist mal mehr, mal weniger intensiv.

1 http://www.studip.de

In der Regel wird jede Woche nach der Vorlesung ein Übungsblatt auf *Stud.IP* veröffentlicht, das die Studierenden bis zur Übungsstunde der folgenden Woche bearbeiten müssen. In den Übungsstunden werden die Aufgaben von den Studierenden dem betreuenden Dozenten sowie den anderen anwesenden Studierenden der Gruppe vorgestellt und diskutiert, wobei die Studierenden sich teils dazu freiwillig melden können, teils werden sie auch zufällig aufgerufen. Studierende, die zu häufig die Aufforderung, ihre Lösung vorzustellen, verweigern, können die Prüfungsvorleistung, die zur Teilnahme an der schriftlichen 90-minütigen Prüfung berechtigt, im laufenden Semester nicht mehr erwerben. Zu Beginn des Semesters wird bekannt gegeben, wie häufig eine solche Aufforderung ohne Konsequenzen abgelehnt werden darf.

Einige der Aufgaben der veröffentlichten Übungsblätter sind als sogenannte *ASB-Aufgaben* gekennzeichnet. Das heißt, hierzu muss bis zu einer angegebenen Frist eine Lösung auf den ASB-Server hochgeladen werden. Die Studierenden können relativ schnell die automatischen Bewertungsergebnisse zu ihren hochgeladenen Lösungen einsehen. Sollte der ASB-Server anzeigen, dass Mängel festgestellt wurden, können die Studierenden ihre Software verändern und die so veränderte Software erneut auf den ASB-Server laden. Es gibt keine Versuchszählung, sodass dies beliebig oft möglich ist, bis die Abgabefrist erreicht ist. Die Abgabefrist liegt in der Regel jede Woche immer einige Stunden vor Beginn der Übungsstunde der ersten Gruppe, sodass die Übungsbetreuenden noch die Möglichkeit haben, die finalen Bewertungsergebnisse der eingereichten Programme vor Beginn der Übungsstunden einzusehen. Auch für die ASB-Aufgaben gibt es bestimmte Mindestanforderungen, die zu Beginn des Semesters mitgeteilt werden und die die Studierenden erfüllen müssen, um die Prüfungsvorleistung zu erwerben. Die Anforderungen sind nicht derart, dass alle Einreichungen der Studierenden komplett mängelfrei sein müssen. Es wird stattdessen gefordert, dass für einen Großteil der gestellten ASB-Aufgaben ein ernsthaftes Bemühen zur Lösung der Aufgabe erkennbar ist. Ob es ein solches „ernsthaftes Bemühen“ gibt oder nicht, kann nicht vollautomatisch vom ASB-System festgestellt werden. Hierzu begutachten die Übungsbetreuenden die Lösungen, wobei hier insbesondere die Lösungen in Augenschein genommen werden, die nach Ablauf der Abgabefrist noch Mängel aufweisen. Nach einer kurzen Anlaufzeit wird von den Studierenden erwartet, dass die formalen Anforderungen, die an ihre Programme gestellt werden (z. B. die richtige Struktur der Ordner und Dateien in der hochgeladenen ZIP-Datei oder die korrekte Formatierung der Quellcodedateien), eingehalten werden. Sollten diese eher einfach zu erfüllenden Anforderungen wiederholt nicht erfüllt werden, führt dies zum Verlust der Prüfungsvorleistung.

Der Anteil derjenigen Aufgaben, die als *ASB-Aufgaben* gekennzeichnet sind, ist im Präsenzstudium je nach Modul unterschiedlich. Dies hängt im Wesentlichen davon ab, ob für die unterschiedlichen Aufgabenstellungen überhaupt Tests entwickelt werden können, wie aufwändig dies gegebenenfalls ist und wie schnell die Tests entwickelt werden können. Im Moment wird ASB nur in Modulen eingesetzt, in denen Java als Programmiersprache verwendet wird. Obwohl geplant war, den Einsatz auch auf Module, die Python oder C++ nutzen, auszudehnen, ist dies bis jetzt nicht geschehen. Es folgt eine Übersicht über die einzelnen Module des Präsenzstudiums, in denen ASB eingesetzt wird:

- Modul *Grafische Benutzeroberflächen*: Der Anteil an ASB-Aufgaben für dieses Modul war vergleichsweise hoch. Mit der Umstellung von Swing auf JavaFX müssen allerdings alle Tests neu entwickelt werden, wobei zunächst geklärt werden musste, wie JavaFX-Programme grundsätzlich getestet werden können. Hierbei war die besondere Herausforderung, herauszufinden, ob und wie es möglich ist, JavaFX-Programme im sogenannten Headless-Modus (d. h. ohne einen an den Rechner angeschlossenen Bildschirm) zu testen. Dies ist eine wichtige Voraussetzung, da der ASB-Server auf einer virtuellen Maschine ohne Bildschirm betrieben wird. Nachdem diese Schwierigkeit überwunden wurde, beginnt nun die Entwicklung neuer Testfälle für die ASB-Aufgaben. In den Testfällen wird die Benutzerinteraktion (Eingaben über Tastatur und Maus) mit dem zu testenden Programm über eine Programmierschnittstelle simuliert. Anschließend wird überprüft, ob Elemente der Benutzeroberfläche bestimmte Eigenschaften aufweisen (z. B. ob in einem Anzeigenfeld ein bestimmter Text steht oder ob eine Schaltfläche eine bestimmte Farbe hat). Um einen größeren Nachdruck auf die Bearbeitung der Übungsaufgaben zu legen, müssen die Studierenden ihre Lösungen auch zu manchen Aufgaben auf den ASB-Server laden, obwohl es dafür (noch) keine Tests gibt. Hier werden dann lediglich formale Überprüfungen durchgeführt. Die Übungsbetreuer haben somit aber die Möglichkeit, stichprobenhaft die eingereichten Lösungen durchzusehen.

- Modul *Parallele Programmierung*: Das Testen paralleler Programme ist eine besondere Herausforderung, da sich der genaue Ablauf nicht steuern lässt und sich somit Synchronisationsfehler nicht bei jeder Ausführung des Programms manifestieren. Entsprechend ist hier die Abdeckung mit ASB-Aufgaben eher gering. Wie für das Modul *Grafische Benutzeroberflächen* beschrieben, gibt es auch hier ASB-Aufgaben ohne Tests. Für einige wenige Aufgaben werden statische Codeüberprüfungen vorgenommen.

- Modul *Entwicklung verteilter Anwendungen*: In diesem Modul geht es um die Entwicklung von Client-Server-Anwendungen über Sockets und RMI. Ferner wird die Entwicklung webbasierter Anwendungen, die über Servlets und JSF (Java Server Faces) realisiert werden, behandelt. Für die Sockets- und RMI-Aufgaben wird der Client-Code der Studierenden getestet, indem der Testcode einen Server simuliert, der Server-Code der Studierenden wird getestet, indem die Testfälle als Client agieren. Für die webbasierten Anwendungen läuft der Code der Studierenden in einer Umgebung, die einen Webserver nachahmt. Über eine Programmierschnittstelle kann die Testsoftware mit dem ausgeführten Code der Studierenden interagieren. Das Testen ist in diesem Modul für eine Reihe von Fällen gut möglich, sodass wir hier auf eine mittlere Abdeckung mit ASB-Aufgaben kommen.

- Modul *Entwicklung mobiler Anwendungen*: Wenn auch eine Android-App in Java programmiert ist, so kann sie dennoch nicht in einer normalen Java-Laufzeitumgebung ausgeführt werden, sondern nur auf realen Android-Geräten oder innerhalb von Android-Emulatoren. Es musste also zunächst die Möglichkeit geschaffen werden, Android-Apps auf dem ASB-Server mithilfe von Emulatoren ausführen und testen zu können, was eine gewisse Herausforderung darstellte. Nachdem dies nun grundsätzlich möglich ist (vgl. Kapitel 16), liegen die ersten Testfälle vor und können genutzt werden. Im Prinzip funktionieren die Testfälle wie für das Modul *Grafische Benutzeroberflächen*, indem Benutzereingaben simuliert werden und das Vorhandensein bestimmter Eigenschaften in den Elementen der Oberfläche überprüft wird. Die Abdeckung der Aufgaben mit Testfällen ist momentan noch relativ gering, soll aber in Zukunft erhöht werden.

Nachdem damit die Einbettung der automatischen Programmbewertung in den Lehrbetrieb für das Präsenzstudium beschrieben wurde, soll nun der Einsatz im Fernstudium erläutert werden. Das Informatikfernstudium der Hochschule Trier ist ein weiterbildendes Masterstudium, das sich an Quereinsteiger und Quereinsteigerinnen in die Informatik wendet. Damit ist gemeint, dass als Voraussetzung zur Zulassung in der Regel ein Bachelorabschluss gefordert wird in einem Fach, das sich genügend deutlich von der Informatik unterscheidet wie zum Beispiel Maschinenbau oder Elektrotechnik. Durch das gebührenpflichtige und in der Regel berufsbegleitend durchgeführte Fernstudium erhalten die Studierenden eine Zusatzqualifikation. Die Bezeichnung dieses Studiengangs führt aus diesem Grund den Klammerzusatz *Aufbaustudium*.

Im Fernstudium wird das ASB-System in zwei Modulen eingesetzt, die wie fast alle anderen Module des Fernstudiums mit 10 ECTS-Punkten kreditiert werden. Die Vorlesung wird im Fernstudium generell durch ein Selbststudium ersetzt,

wobei hierfür je nach Modul unterschiedliche Arten von Lehrmaterialien zur Verfügung gestellt werden: speziell für das Fernstudium entwickelte Lehrmaterialien (in Papierform und als PDF-Datei bereitgestellt), Bücher, webbasierte Lehrmaterialien sowie Vorlesungsaufzeichnungen. Dazu gibt es im Verlauf des Semesters pro Modul mehrmals abendliche Tutorien, die als Videokonferenz durchgeführt werden. Im Verlauf dieser Tutorien werden exemplarisch Lehrinhalte vertieft, wobei die teilnehmenden Studierenden hierzu direkt Fragen stellen können. Eine Beratung per E-Mail ist ebenfalls jederzeit möglich. Die Übungen bestehen aus Einsendeaufgaben und einem einwöchigen Präsenzpraktikum. Wenn von den Einsendeaufgaben mindestens 50 % erfolgreich bearbeitet wurden, erfolgt eine Zulassung zum einwöchigen Präsenzpraktikum, das mit einer schriftlichen oder mündlichen Prüfung am letzten Tag des Praktikums abgeschlossen wird. In den Modulen *Einführung in die objektorientierte Programmierung* und *Fortgeschrittene Programmiertechniken (FOPT)* wird zur Kontrolle der Einsendeaufgaben das ASB-System eingesetzt, wobei es sich im Modul *FOPT* bei allen Einsendeaufgaben um ASB-Aufgaben handelt. In beiden Modulen ist die verwendete Programmiersprache auch wieder Java. Abschließend sei noch erwähnt, dass das ASB-System auch in einem anderen Fachbereich der Hochschule Trier eingesetzt wird, nämlich im Fachbereich Wirtschaft für die Java-Programmierausbildung des Bachelorstudiengangs Wirtschaftsinformatik.

7.3 Das Modul *Entwicklung mobiler Anwendungen*

Im Folgenden soll näher auf das Modul *Entwicklung mobiler Anwendungen (EMA)* eingegangen werden, da die Nutzung des ASB-Systems in diesem Modul eine besondere Herausforderung darstellt. Bei diesem Modul handelt es sich um ein Pflichtmodul des Bachelorstudiengangs „Informatik – Sichere und mobile Systeme“, der vom Fachbereich Informatik der Hochschule Trier seit 2008 (damals noch unter der Studiengangsbezeichnung „Informatik – Internetbasierte Systeme“) angeboten wird. Der Studienplan sieht vor, dass dieses Modul im 4. Semester zu absolvieren ist. Das Modul *EMA* kann von den Studierenden der drei anderen Informatik-Bachelorstudiengänge („Informatik“, „Informatik – Digitale Medien und Spiele“ und „Medizininformatik“) als Wahlpflichtmodul belegt werden. Das Modul erfreut sich großer Beliebtheit. Es wird jedes Jahr in der Regel von ca. 80 Studierenden belegt, sodass hierfür drei Übungsgruppen benötigt werden. Mehrere Personen treten als Lehrende dieses Moduls auf: die Vorlesung teilen sich 3 – 4 Dozenten im Verlauf der Vorlesungszeit eines Semesters auf, auch die Übungsgruppen werden von mehreren Personen betreut.

Die Studierenden lernen in diesem Modul unter anderem, welche Interaktionselemente Android zur Gestaltung von Oberflächen anbietet (Buttons, TextViews, EditTexts, Menüs, Dialoge usw.), wie die Bildschirminhalte sogenannter Activities über XML-Layout-Dateien spezifiziert werden, wie Reaktionen auf Benutzeraktionen programmiert werden, wie Activities der eigenen App und unterschiedlicher Apps über sogenannte Intents gekoppelt werden, wie die Problematik einer länger dauernden Ereignisbehandlung (z. B. das Herunterladen größerer Dateien) aufgrund einer Benutzerinteraktion (z. B. eines Button-Klicks) gelöst werden kann und wie unterschiedliche Arten von Services (gestartete Services und gebundene Services) zur Realisierung von Aktionen „im Hintergrund“ (d. h. ohne Benutzeroberfläche) eingesetzt werden können. Als wesentliches Lernziel wird im Modulhandbuch beschrieben, dass die Studierenden die Fähigkeit erlernen, selbstständig kleinere bis mittelgroße Android-Anwendungen zu spezifizieren und mithilfe der Entwicklungsumgebung Android Studio umzusetzen.

7.4 Benutzung des ASB-Systems aus Studierendensicht

Das ASB-System ist nach erfolgreicher Anmeldung mit der Hochschulkennung für jeden Studierenden der Hochschule Trier über einen Browser erreichbar. Die Nutzerin in der Rolle Student erhält auf der Startseite eine Übersicht über die Lehrveranstaltungen, zu denen sie Lösungen einreichen kann, sowie eine Auflistung offener Aufgaben (d. h. Aufgaben, zu denen aktuell Lösungen eingereicht werden können). Abbildung 7.1 zeigt die Startseite des ASB-Systems nach erfolgreicher Anmeldung. Wie in Abschnitt 7.1 erwähnt wurde, besitzt jede Auf-

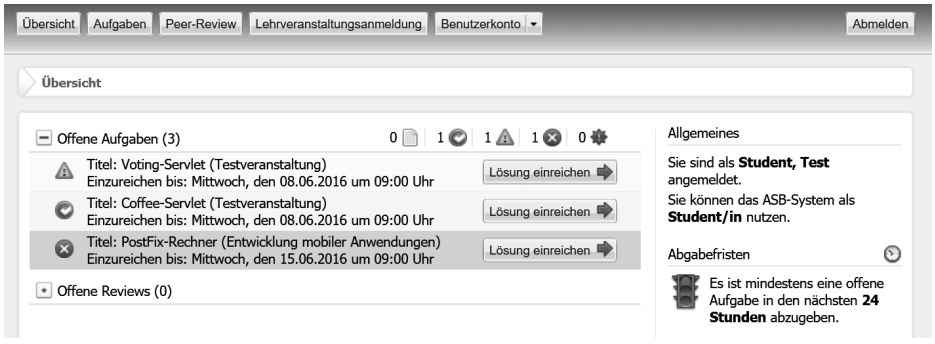


Abbildung 7.1: Studentische Startseite des ASB-Systems

gabe einen Zeitraum, in dem Studierende Lösungen zu dieser Aufgabe hochladen können. Wird diese Abgabefrist überschritten, erscheint die Aufgabe auf der Übersichtsseite nicht mehr. In der Regel wird jede eingereichte studentische Lösung mehreren Bewertungsmaßnahmen unterworfen. Jede einzelne Bewertungsmaßnahme hat pro Aufgabe und pro registriertem Studierenden einen von fünf möglichen Zuständen:

- Noch keine Lösung eingereicht,
- Bewertungsmaßnahme fehlerfrei abgeschlossen (d. h. die Bewertungsmaßnahme konnte erfolgreich durchgeführt werden, dabei gab es keine Beanstandungen),
- Bewertungsmaßnahme mit Warnung abgeschlossen (d. h. die Bewertungsmaßnahme konnte erfolgreich durchgeführt werden, hat aber Punkte gefunden, die noch verbessert werden können wie zum Beispiel die Deklaration einer lokalen Variablen, die im Code gar nicht verwendet wird und somit entbehrlich ist),
- Bewertungsmaßnahme mit Fehlern abgeschlossen (d. h. die Bewertungsmaßnahme konnte erfolgreich durchgeführt werden, hat aber Fehler im zu bewertenden Programm entdeckt),
- Bewertungsmaßnahme fehlgeschlagen (d. h. die Bewertungsmaßnahme konnte nicht durchgeführt werden).

Die Ergebnisse der einzelnen Bewertungsmaßnahmen für eine eingereichte Lösung werden zu einem Gesamtergebnis zusammengefasst. Dabei ist das Gesamtergebnis das Ergebnis einer Einzelbewertungsmaßnahme, das in der obigen Liste am weitesten unten steht. Wenn also beispielsweise ein eingereichtes studentisches Programm durch drei Maßnahmen bewertet wird, wobei eine fehlerfrei, eine mit Warnungen und eine mit Fehlern abgeschlossen wird, so ist das Gesamtergebnis: mit Fehlern abgeschlossen. Als Ergebnis einer Bewertung werden also keine Punkte wie beispielsweise bei einer Klausur vergeben.

Ungeachtet des Ergebnisses der Bewertungsmaßnahmen können die Studierenden bis zum Erreichen der Abgabefrist ihre Lösungen beliebig oft hochladen und bewerten lassen. Allerdings wird für den Zustand der Aufgabe lediglich die letzte Bewertung betrachtet. Auf der Webseite zum Hochladen einer Lösung wird den Studierenden auch eine Übersicht über die Bewertung der zuletzt hochgeladenen Lösung angezeigt, falls zu dieser Aufgabe überhaupt schon etwas eingereicht wurde. Diese Übersicht zeigt an, welche Bewertungsmaßnahmen für die Aufgabe

durchgeführt wurden und zu welchem Ergebnis dies jeweils geführt hat. Abbildung 7.2 zeigt die Bewertungsübersicht nach dem Einreichen einer Lösung.

Die Studierenden sollen bei dieser Aufgabe einen Postfix-Taschenrechner als Android-App implementieren. Die Lösungen zu dieser Aufgabe durchlaufen drei Bewertungsmaßnahmen: *Android*, *ECJ* und *Checkstyle*. Die *Android*-Bewertungsmaßnahme prüft, ob sich die hochgeladene App so verhält, wie es von der Aufgabenstellung gefordert wird. Dazu wird die Anwendung ausgeführt und getestet. Die detaillierte Bewertungsübersicht zeigt in einer Baumstruktur, welche Tests mit Erfolg beziehungsweise nicht erfolgreich abgeschlossen wurden. In der Abbildung ist zu sehen, dass die Funktionalitätstests zum Teil fehlgeschlagen sind. Damit ist das Gesamtergebnis dieser Bewertungsmaßnahme, dass Fehler gefunden wurden. Durch die Auswahl eines fehlgeschlagenen Tests erhalten die Studierenden alle verfügbaren Informationen über den durchgeführten Test, über das

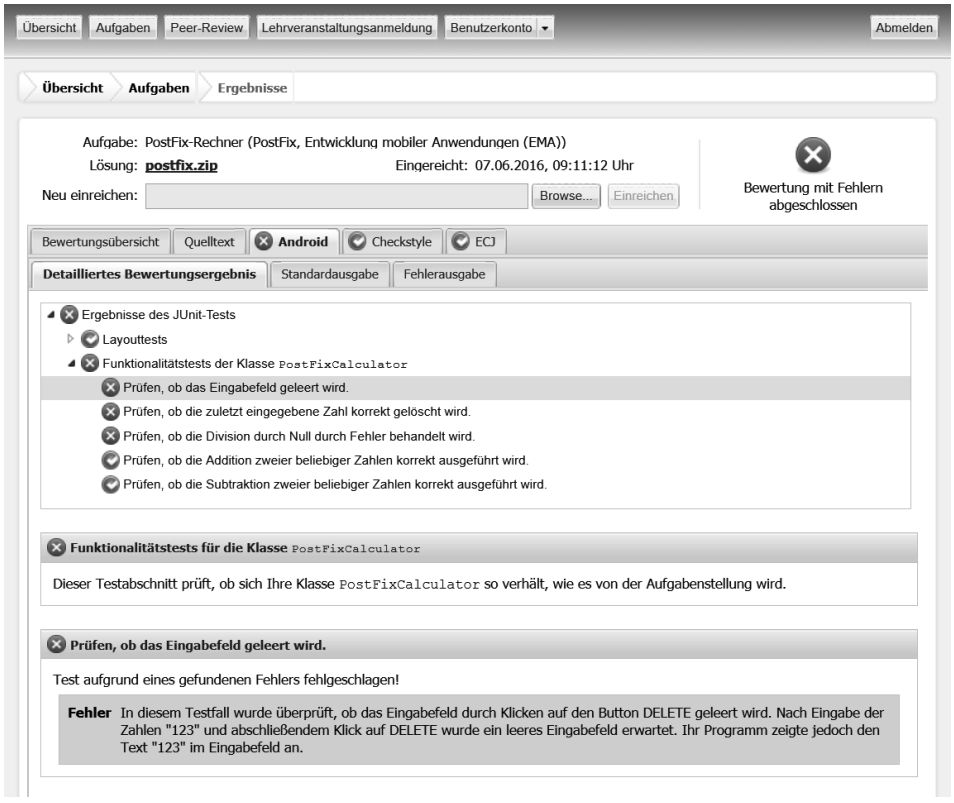


Abbildung 7.2: Studentische Bewertungsübersicht einer eingereichten Lösung am ASB-System

Ist- und das Soll-Verhalten der Anwendung sowie über alle erzeugten Warnungs- und Fehlermeldungen. In diesem konkreten Fall hat der Studierende unter anderem die Funktion des DELETE-Buttons nicht korrekt implementiert, sodass das Eingabefeld nicht geleert wird.

Bei der zweiten Bewertungsmaßnahme *ECJ* wurde der Code mit dem Eclipse-Compiler übersetzt. Abbildung 7.2 zeigt, dass diese Bewertungsmaßnahme erfolgreich war. Gleiches gilt auch für die dritte Bewertungsmaßnahme *Checkstyle*[2]. Diese prüft den eingereichten Quellcode auf zuvor bekannt gegebene Programmierkonventionen. Falls es in dieser Maßnahme zu einem Fehler kommt, erhält die Nutzerin eine genaue Fehlermeldung, wie und wo die Konvention verletzt wurde.

Da es in diesem Beispiel zwei Maßnahmen gab, die erfolgreich durchgeführt wurden, und eine Maßnahme, die Fehler entdeckt hat, lautet das Gesamtergebnis, dass Fehler entdeckt wurden. Das Gesamtergebnis wird in der oberen rechten Ecke angezeigt.

7.5 Benutzung des ASB-Systems aus Dozentensicht

Wie Studiernde können sich Dozenten mittels ihrer Kennung der Hochschule Trier am ASB-System anmelden. Nach erfolgreicher Anmeldung sind diese in der Lage, ihre im ASB hinterlegten Lehrveranstaltungen zu administrieren. Dazu gehört das Anlegen von Aufgaben innerhalb ihrer Lehrveranstaltungen. Diese bestehen neben der Beschreibung und Abgabefrist aus einer oder mehreren Bewertungsmaßnahmen. Auf den Aufbau von Bewertungsmaßnahmen wird in Kapitel 16 eingegangen.

Wichtigster Bestandteil der Nutzung des ASB-Systems aus Dozentensicht ist die Bewertungsübersicht einer Aufgabe. Hier liegt dem Dozenten eine vollständige Übersicht aller Bewertungen der Studierenden vor, die Lösungen zu einer bestimmten Aufgabe eingereicht haben. Durch Auswählen eines einzelnen Studierenden wird eine wie schon in Abschnitt 7.4 beschriebene detaillierte Bewertungsübersicht angezeigt. Zusätzlich zu den Bewertungen kann der Dozent den hochgeladenen Quelltext im Browser betrachten und auch herunterladen. Dies unterstützt die Lehrenden in der manuellen Überprüfung studentischer Lösungen. Abbildung 7.3 zeigt, dass zunächst die Postfix-Aufgabe ausgewählt wurde. Links unten sieht man eine Übersicht über alle Studierende, die zu der dazugehörigen

2 http://checkstyle.sourceforge.net

Lehrveranstaltung registriert sind. Man kann dadurch auch leicht erkennen, wer nichts eingereicht hat. Wird eine Person aus der Liste ausgewählt, so werden die dazugehörigen Bewertungsergebnisse angezeigt. Neben den für Studierende einsehbaren Bewertungsmaßnahmen kann das ASB-System auch versteckte Maßnahmen durchführen, deren Resultate nur die Dozenten sehen können. Hierzu zählt beispielsweise die selbst entwickelte Plagiatserkennung *JMaat*. Diese wird nach Ablauf der Abgabefrist angestoßen und zeigt dem Dozenten eine Übersicht möglicher Plagiate.

7.6 Vorgaben für ASB-Aufgaben

Zur Bewertung von Lösungen müssen bestimmte Vorgaben seitens der Studierenden eingehalten werden. Diese werden zu den jeweiligen ASB-Aufgaben auf den Übungsblättern erläutert. So werden in vielen Aufgaben Package-, Klassen- und Methodennamen vorgegeben. Darüberhinaus ist es notwendig, dass für manche Methoden die Parametertypen und der Rückgabetyp genau festgelegt werden. Dies ist notwendig, damit die durchzuführenden Unit-Tests ausgeführt werden können. Häufig werden hierfür vordefinierte Schnittstellen in Form von Dateien zur Verfügung gestellt, die die Studierenden implementieren müssen. Den Unit-

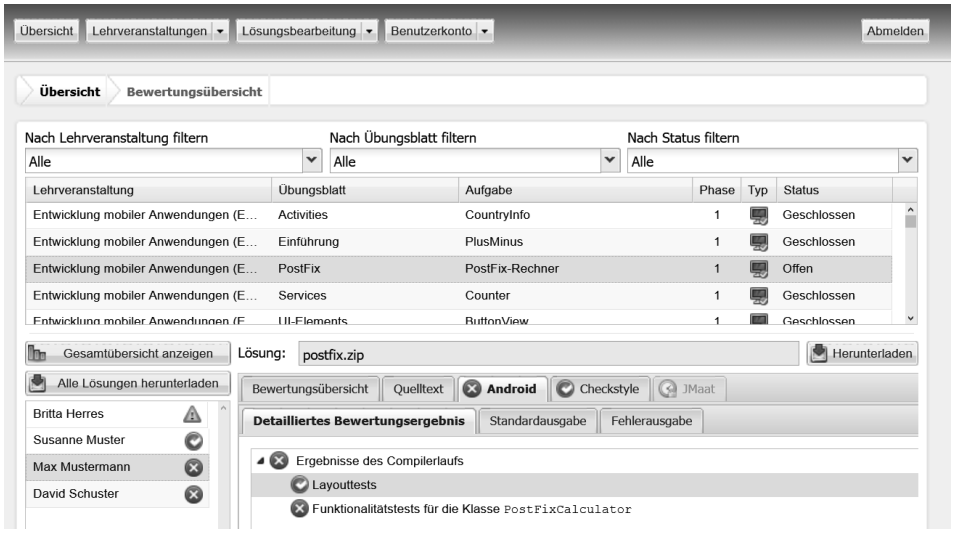


Abbildung 7.3: Bewertungsübersicht eines Dozenten zu einer Aufgabe am ASB-System

Tests sind größtenteils Signaturprüfungen vorgeschaltet, die überprüfen, ob die Lösung diese Vorgaben erfüllt. Ist dies nicht der Fall, werden keine Funktionalitätstest angestoßen, sondern es erfolgt eine Fehlermeldung, dass die eingereichte Klasse zum Beispiel eine bestimmte Methode nicht besitzt.

Zum Simulieren von Nutzertests im Kontext der Programmierung von Android-Apps und JavaFX-Programmen müssen die Interaktionselemente, die in den Programmen der Studierenden erzeugt werden, mit vorgegebenen Identifikatoren ausgestattet werden. Dies gestattet eine automatisierte Prüfung der erstellten Benutzeroberflächen, indem die studentischen Programme gestartet, mittels der Identifikatoren die Interaktionselemente identifiziert, Nutzerinteraktionen wie Eingaben in Textfelder oder Klicks auf Buttons simuliert und daraus resultierende Anzeigen des Programms auf Richtigkeit geprüft werden.

Durch Betrachtung von Abbildung 7.4 erhält man eine gewisse Vorstellung, wie solche Tests implementiert werden. Es geht dabei um das Testen der Implementierung eines Postfix-Rechners als Android-App. Der erste Teil des dargestellten Codes besteht aus der Simulation einer Benutzereingabe. In diesem Fall drückt man auf den Button mit der Ziffer 4 und anschließend auf den Push-Button (damit ist die Eingabe einer Zahl beendet, die Zahl 4 wird somit auf den Keller gelegt). Danach wird der Button mit der Ziffer 3 gedrückt und wieder Push (damit wird nun auch noch die Zahl 3 auf den Keller gelegt). Schließlich wird der Plus-Button gedrückt. Damit ist die Simulation der Benutzereingabe beendet. Wichtig ist hierbei, dass die Studierenden die in der Aufgabenstellung vorgeschriebenen Identifikatoren für ihre Buttons wie buttonDigit4, buttonPush und buttonOperationPlus genauso einhalten, denn ansonsten kann der Test nicht durchgeführt werden. Wenn die App nach dieser simulierten Eingabe richtig implementiert wurde, müsste sie

```
/** Test Plus */
@TestDestriction(R.string.FuncTestAdd)
public void testPlus() {

  TouchUtils.clickView(this, this.buttonDigit4);
  TouchUtils.clickView(this, this.buttonPush);

  TouchUtils.clickView(this, this.buttonDigit3);
  TouchUtils.clickView(this, this.buttonPush);

  TouchUtils.clickView(this, this.buttonOperationPlus);

  final float result = this.readTextViewValue();

  Assert.assertEquals(this.res.getString(R.string.FuncTestAddMsg1, result), 7F, result, 0.001F);
}
```

Abbildung 7.4: Programmcode eines Tests für eine Android-App

jetzt die beiden obersten Kellerelemente (3 und 4) vom Keller genommen, addiert und das Ergebnis als oberstes Element auf den Keller gelegt und angezeigt haben. In der vorletzen Zeile wird der angezeigte Wert des obersten Kellerelements von der Oberfläche ausgelesen. Dazu wird eine Hilfsmethode verwendet. In der letzten Zeile schließlich wird der ausgelesene Wert mit 7 (dem erwarteten Resultat der Addition von 4 und 3) verglichen. Sind beide Werte (bis auf eine angegebene Toleranz von 0.001) gleich, endet der Test erfolgreich, andernfalls mit einem Fehler.

Das Einreichen von Lösungen wird den Studierenden zu Beginn der Lehrveranstaltungen erläutert. Das ASB-System erwartet alle zu testenden und darüber hinaus genutzten Dateien (**.java, *.fxml, *.xml, ...*) innerhalb der vorgegebenen Paketstruktur als ZIP-Archiv. Dieses wird auf dem ASB-System entpackt und den Bewertungsmaßnahmen zur Bewertung übergeben.

Die Tests zu den Lehrveranstaltungen werden von Mitarbeiterinnen und Mitarbeitern des Fachbereiches in Git-Repositories gepflegt und erweitert. Sowohl das ASB-System selbst als auch die Tests sind Eigenentwicklungen der Hochschule Trier. Die Testkonzepte zum Testen von Android-Applikationen sind im Rahmen von studentischen Projekten entstanden.

7.7 Erfahrungen

Die Erfahrung zeigt, dass das ASB-System im Allgemeinen sehr gut angenommen und von vielen Studierenden sehr positiv hervorgehoben wird. In den hochschulweit standardisierten Fragebögen zur Lehrveranstaltungsbewertung gibt es (selbstverständlich) keine Frage zum ASB-System. In den Freitextantworten wird die Nutzung des ASB-Systems immer wieder positiv hervorgehoben, insbesondere auch von den Fernstudierenden. Es gibt jedoch auch Situationen, bei denen die Studierenden sich über das ASB-System beklagen. Eine typische Aussage lautet dann: *„Ich versuche jetzt seit Stunden, die vom ASB-Sever gemeldeten Fehler zu beheben. Bei mir lokal auf dem Rechner funktioniert alles einwandfrei, aber der ASB-Server meldet ständig Fehler. Ich drehe gleich durch“*. Die Ursache dafür liegt nach unserer Erfahrung nur in einigen solcher Fälle beim ASB-System. So kann es beispielsweise vorkommen, dass die Aufgabenspezifikation und die Tests nicht konsistent zueinander sind, insbesondere wenn die Aufgabenbeschreibung angepasst und dabei vergessen wird, diese Änderungen auch in den automatischen Tests nachzuvollziehen. Dann steigt die Anzahl der Forumsbeiträge auf der Lernplattform *Stud.IP* stark an. In der Regel werden die Tests oder die Aufgabenstellung dann umgehend angepasst und die Änderung im Forum mitgeteilt. In

manchen Fällen finden die Studierenden durch Ausprobieren heraus, was getan werden muss, damit die Lösung vom ASB-Server nicht mehr beanstandet wird, und teilen dies den anderen Studierenden im Forum mit. In seltenen Fällen treten auch noch Fehler im ASB-System selbst auf, was zu einer gewissen Frustration führt. Oft liegt die Ursache des Problems allerdings nicht auf Seiten des ASB-Servers, sondern auf Seiten der Studierenden selbst. Die Studierenden probieren ihre Software aus, kommen dabei jedoch nicht in solche Situationen, die in den Tests auf dem ASB-System überprüft werden. Wenn dann die Studierenden die Fehlermeldungen des ASB-Systems nicht verstehen oder nicht richtig interpretieren, was manchmal an zu knappen oder nicht treffend formulierten Meldungen des ASB-Systems liegen kann, dann beklagen sich die Studierenden über das Forum oder per Mail. Die Lehrenden geben daraufhin dezente Hinweise, in welche Richtung die eingereichte Lösung geändert werden sollte.

Wie soeben angesprochen können schwer verständliche Fehlermeldungen ein Problem sein. Sind hingegen die Fehlermeldungen sehr klar und ausführlich formuliert, so kann dies wiederum auch zu Problemen führen. So haben wir zum Beispiel schon beobachtet, dass die Studierenden ihre Programme genau auf die getesteten Spezialfälle hin ausrichten, wenn der ASB-Server sehr genau meldet, was das Programm ausgibt und was erwartet wird. Um zu verstehem was damit gemeint ist, stelle man sich als Beispiel ein Programm vor, das zwei eingegebene Zahlen addieren und das Ergebnis ausgeben soll. In einem Fehlerfall würde das ASB-System beispielsweise melden: *„Es wurden die Zahlen 3 und 4 eingegeben. Ihr Programm gibt 9 aus, erwartet wird aber 7“*. Wir haben schon erlebt, dass dann Programme so geändert werden, dass die Eingaben ignoriert werden und genau das vom ASB-System erforderte Resultat produziert wird. Würde bezogen auf das oben angegebene Beispiel immer 7 als Ergebnis der Addition ausgegeben, wäre der Test zwar bestanden, die programmierte Lösung wäre aber eigentlich falsch. Als Gegenmaßnahmen führen wir nach Möglichkeit mehrere unterschiedliche Tests aus. Wenn es die Aufgabenstellung erlaubt, variieren die Tests die Eingaben auch zufällig, sodass bei jedem erneuten Hochladen einer Lösung ein anderer Testfall geprüft wird. Grundsätzlich geben wir den Testcode den Studierenden nicht bekannt.

Das ASB-System wird von den Lehrenden ausschließlich als Assistenzsystem genutzt. Es geht dabei um die Vergabe des Nachweises einer unbenoteten Prüfungsvorleistung. Zur Unterstützung von Prüfungen wird das System nicht eingesetzt (wie zuvor beschrieben wurde, gibt es auch kein Punktesystem). Und auch die Bestätigung der erbrachten Prüfungsvorleistung erfolgt nicht vollautomatisch. Zum Erwerb der Prüfungsvorleistung wird nicht gefordert, dass ein bestimmter Prozentsatz der von den Studierenden eingereichten Lösungen fehlerfrei ist; es

wird lediglich gefordert, dass es das „ernsthafte Bemühen“ gibt, einen bestimmten Teil der Aufgaben zu lösen. Das ASB-System unterstützt die Lehrenden dabei herauszufinden, welche Lösungen sie genauer betrachten sollten, um dann zu entscheiden, ob ein „ernsthaftes Bemühen“ vorliegt. Eine eingereichte Lösung wird auf jeden Fall betrachtet, wenn dazu eine Frage oder Beschwerde zum Verhalten des ASB-Systems eintrifft. Aber auch ohne eine solche Rückmeldung ist ein Blick in fehlerhaft gekennzeichnete Lösungen interessant. Man kann dann nicht nur das „ernsthafte Bemühen“ überprüfen, sondern in den Übungsstunden ganz speziell darauf hinweisen, wie etwas nicht gelöst werden sollte. Stichprobenhaft werden auch als korrekt markierte Lösungen aufgrund der oben beschriebenen Tricksereien angeschaut. Dies ist insbesondere dann nötig, wenn es zu einer ASB-Aufgabe keine Tests gibt und nur formale Überprüfungen (korrekte Syntax, Einhaltung von Codierungskonventionen) durchgeführt werden. Es ist sogar schon vorgekommen, dass Programme, die mit der gestellten Aufgabe nichts zu tun haben, eingereicht wurden (z. B. Lösungen zu anderen Aufgaben). In diesem Fall werden die Studierenden ermahnt, dass dies als Täuschungsversuch interpretiert und im wiederholten Fall die Prüfungsvorleistung nicht vergeben wird. Ein weiteres Problem ist die einfache Kopierbarkeit von Dateien. Musterlösungen zu den Aufgaben werden seit langer Zeit nicht mehr herausgegeben, weil dies in der Vergangenheit zu dem Problem geführt hat, dass von einigen Studierenden die Musterlösung des Vorjahres inklusive aller Kommentare eingereicht wird. Auch Plagiate sind ein Problem. Die Studierenden werden intensiv darauf hingewiesen, dass sie alle ihre Lösungen selbst zu programmieren haben. „Gruppenarbeit“ ist grundsätzlich nicht erlaubt, da die Lehrveranstaltungen auf den Erwerb individueller Programmierkompetenz ausgerichtet sind, was nicht immer von allen Studierenden klaglos akzeptiert wird. Ob Lösungen als Plagiate eingestuft werden, wird ebenfalls nicht vollautomatisch allein aufgrund der ASB-Meldungen entschieden. Die im ASB-System genutzte Plagiatserkennung gibt lediglich Hinweise, welche Lösungen sehr ähnlich sind. Diese verdächtigen Lösungen werden dann von den Lehrenden näher in Augenschein genommen. Dies führt bei starkem Verdacht zu Verwarnungen, in besonders dreisten Fällen direkt zum Verlust der Prüfungsvorleistung. Aus unserer Erfahrung scheinen solche abschreckenden Vorgehensweisen notwendig zu sein.

Studierenden, die zwar eine korrekte Lösung eingereicht haben, die sich jedoch darüber hinaus fragen, ob ihre Lösung „Schönheitsmängel“ besitzt und trotz erfolgreicher ASB-Prüfung verbessert werden kann, bieten wir seit Kurzem ein sogenanntes Programmierberatungsbüro an, in dem Assistenten und Assistentinnen den Studierenden Verbesserungshinweise zu ihrer Software geben.

7.8 Fazit und Ausblick

Insbesondere in Zeiten mit hohen Studierendenzahlen bietet das ASB-System eine wertvolle und komfortable Unterstützung bei der Durchführung des Übungsbetriebs programmierlastiger Lehrveranstaltungen im Fachbereich Informatik der Hochschule Trier. Bei der Nutzung des ASB-Systems fallen immer wieder Punkte auf, wie das ASB-System, insbesondere dessen Bedienbarkeit, verbessert werden könnte. Je nach Aufwand werden dann entsprechende Änderungen am ASB-System vorgenommen oder auch nicht.

Eine signifikante Erhöhung der Automatisierung bei der Bewertung studentischer Lösungen ist dagegen im Augenblick schwer vorstellbar. Die Lehrenden wissen, dass beim Programmieren aufgrund eines kleinen Fehlers sehr viel schief gehen kann. Zum Beispiel kann ein vergessenes Semikolon einen Syntaxfehler erzeugen, sodass die studentischen Programme nicht übersetzt, damit nicht ausgeführt und nicht getestet werden können, oder eine fehlende Initialisierung kann in einer NullPointerException resultieren, die alle Testläufe scheitern lässt. Würde man in solchen Fällen vollautomatisch zu dem Ergebnis kommen, dass nichts an dem eingereichten Programm richtig ist, würde man die studentischen Programme sicher zu negativ bewerten. Auch die Überprüfung von Strukturvorgaben (z. B. ob ein Programm nach dem Model-View-Presenter-Prinzip gestaltet wurde) ist schwer automatisch überprüfbar. Umgekehrt werden auch die Studierenden immer wieder darauf hingewiesen, dass sie auf keinen Fall davon ausgehen dürfen, dass die Anzeige der Fehlerfreiheit durch ASB bedeutet, dass ihr Programm korrekt ist. Aus diesen Gründen wird das ASB-System nicht zur vollautomatischen Bewertung genutzt, sondern lediglich als Assistenzsystem sowohl für Lehrende als auch für Studierende.

Literatur für dieses Kapitel

[Mor+07] Thiemo Morth u. a. „Automatische Bewertung studentischer Software“. In: *Workshop „Rechnerunterstütztes Selbststudium in der Informatik“*. Universität Siegen, 2007.

8 Automatisierte Programmbewertung in der Mathematikausbildung

Uta Priss und Peter Riegler

Zusammenfassung
Insbesondere in Mathematikvorlesungen der ersten Semester mit hohen Teilnehmerzahlen ermöglichen automatisiert bewertete Programmieraufgaben regelmäßiges und zeitnahes Feedback. Zusammen mit einer geeigneten didaktischen Lehrmethode (in diesem Kapitel beruhend auf der APOS-Theorie) können Programmieraufgaben Verstehensprozesse in der Mathematiklehre unterstützen. Neben der APOS-Theorie werden Aufgabentypen vorgestellt, die sich hierfür besonders eignen, und es werden die benötigten Tests und erste Erfahrungen kurz erörtert.

8.1 Einleitung

Der Einsatz elektronischer Werkzeuge in der Mathematikausbildung ist nicht ungewöhnlich und aufgrund der Nähe von Mathematik und Informatik nahe liegend. Der Einsatz von Software als Berechnungswerkzeug ist mit Taschenrechner und zum Teil auch Computeralgebrasystemen (CAS) in der Mathematiklehre, wenn auch immer wieder kontrovers diskutiert [Hol95], weit verbreitet. CAS werden auch als eine Art Laborumgebung eingesetzt, in der Studierende Mathematik explorieren können. Hinzu kommt der Einsatz von Software meist in Form von *Applets*, um mathematische Zusammenhänge zu visualisieren.

Teile dieses Kapitels entstanden im Rahmen des Projekts eCULT, Teilvorhaben eAssessment, gefördert durch das Bundesministerium für Bildung und Forschung unter dem Förderkennzeichen 01PL16066H. Die Verantwortung für den Inhalt dieses Kapitels liegt bei der Autorin und dem Autor.

Im Vergleich zu den genannten Werkzeugen und Zielen scheint Programmieren als Tätigkeit, die das Erlernen von Mathematik begünstigt, bisher eher selten Teil der Mathematiklehre zu sein. Für einen solchen Einsatz sprechen aus didaktischer Sicht gute Gründe, die durchaus die Nachteile, die mit einem solchen Einsatz einhergehen (z. B. das Hinzufügen der Schwierigkeiten, die mit Programmieren verbunden sind, zu der ohnehin schon als herausfordernd empfundenen Lehre von Mathematik), kompensieren können. Mathematik wie auch andere MINT-Disziplinen erfordert erstens die koordinierte Nutzung verschiedener Repräsentationsformen (symbolisch, graphisch etc.) [WM06]. Programme stellen eine weitere Möglichkeit dar, Mathematik zu repräsentieren. Die Bedeutung dieser Repräsentationsmöglichkeit hat in akademischen Berufsfeldern auch außerhalb der Mathematik in den letzten Jahrzehnten deutlich zugenommen.

Zweitens erfordert Mathematik und das Erlernen von Mathematik wie jede wissenschaftliche Disziplin den Austausch mit anderen. Aus einer sozial-konstruktivistischen Perspektive betrachtet entstehen Fachkonzepte aus einer Konsensbildung in einer *Community*. Für das Erlernen dieser Konzepte ist Dialog hilfreich, wenn nicht sogar notwendig. Programmieren kann als Dialog aufgefasst werden, als Dialog mit einem gewissermaßen harten Gesprächspartner, der streng auf die Einhaltung vereinbarter (Syntax-)Regeln beharrt. Programmieren in der Mathematikausbildung bietet also den räumlich und zeitlich praktisch unbegrenzt verfügbaren Dialogpartner, der zudem sehr penetrant auf Strenge in der Konversation achtet.

Drittens kann Programmieren ein wertvoller Mediator der kognitiven Prozesse sein, die für das Verinnerlichen und Weiterentwickeln mathematischer Konzepte notwendig sind. In der Tat sprechen kognitionstheoretische Gründe für einen bestimmten Einsatz von Programmieraufgaben in der Mathematiklehre. Abschnitt 8.2 erläutert dies im Detail. Abschnitt 8.3 stellt Aufgabentypen vor, welche besonders für die Programmierung geeignet sind. Abschnitt 8.4 diskutiert technische Aspekte der automatischen Bewertung von Programmieraufgaben. Zum Schluss berichtet Abschnitt 8.5 von unseren Erfahrungen mit dem Einsatz von Programmieraufgaben in einer Lehrveranstaltung zur Diskreten Mathematik. Es sollte noch erwähnt werden, dass aus softwaretechnischer Sicht Programmieraufgaben in der Mathematik nicht anders zu implementieren sind als zum Beispiel Java-Programmieraufgaben. Wir selbst verwenden den Praktomat Grader (Kapitel 10), welchen wir um einen SetlX-Checker erweitert haben, in einer Kombination mit LON-CAPA wie in Kapitel 19 beschrieben.

8.2 APOS-Theorie

Die APOS-Theorie [Arn+13] ist eine Kognitionstheorie, die beschreibt und erklärt, wie mathematische Konzepte gelernt werden. Das Akronym APOS steht für *Action*, *Process*, *Object*, *Schema* und benennt charakteristische, in der Regel sukzessive Stadien im Verständnis eines mathematischen Konzepts. Konzeptverständnis unterliegt also einer Entwicklung. Studierende haben ein mathematisches Konzept nicht einfach verstanden oder eben nicht, sondern Verständnis hat charakteristische, aufeinander folgende Ausprägungen zunehmender Abstraktion.

Diese Ausprägungen sollen im Folgenden sowohl allgemein als auch am Beispiel des Konzepts Funktion an einer konkreten Fragestellung und charakteristischen studentischen Antworten erläutert werden. Die Aufgabenstellung besteht in der Frage, ob es eine Funktion gibt, die aus jeder Buchstabenfolge, die eine Zahl bezeichnet (z. B. einhundertdreiundzwanzig), die entsprechende Ziffernfolge (z. B. 123) gewinnen kann. Typische studentische Antworten auf diese Fragestellung sind von der Art [MR11]:

> Student A1: „Ich würde die Buchstabenfolge zuerst in Zahlworte wie Eins, Zwei, Drei, Hundert, Tausend usw. zerlegen und die Zahlworte dann mithilfe einer Tabelle in Ziffern übersetzen."
>
> Student A2: „Mir fehlt der Eingabewert, der in die Funktion gesteckt wird. Eine Funktion braucht eine Eingabe, den ersten Schritt. Der fehlt mir gerade."
>
> Student P: „Da man die Buchstabenfolgen 1:1 auf Ziffernfolgen abbilden kann, sehe ich da keine Schwierigkeiten."

Alle drei Studierenden bejahen letztendlich die Existenz einer solchen Funktion. Die Art ihrer Antworten weist jedoch auf charakteristische Unterschiede hinsichtlich ihres Verständnisses des Konzepts Funktion hin. Für die Studenten A1 und A2 ist es für die Antwortfindung wichtig, ob es einen konkreten Algorithmus gibt, während Student P alleine mit der Eindeutigkeit als charakteristische Eigenschaft einer Funktion argumentiert.

Im *Action*-Stadium verstehen Studierende ein Konzept als eine Aktion. Aktionen sind wiederholbare Abfolgen von Objektmanipulationen. Manipulierbar sind dabei in der Regel nur konkret erfahrbare Objekte, wie physikalische Objekte oder auf Papier geschriebene mathematische Symbole. Studierende benötigen ein explizites, schrittweises Rezept (wie Studenten A1 und A2) oder eine Formel, die solche Aktionen beschreibt, um ein Konzept anwenden zu können. Einzelne

Schritte können dabei nicht alleine in Gedanken ausgeführt werden oder mental übersprungen werden.

Im *Process*-Stadium dagegen können Studierende die Transformationen ausführen ohne jeden dazu notwendigen Schritt explizit ausführen zu müssen (vgl. Student P). Dafür charakteristisch ist die Fähigkeit die Transformation allein im Geiste ausführen zu können. Dabei müssen keine konkreten Objekte manipuliert werden. Der Mechanismus, der vom *Action*- zum *Process*-Stadium führt, wird in der APOS-Theorie als Internalisierung bezeichnet. Das Verständnis eines mathematischen Konzepts wandelt sich dabei von einer Abfolge extern durchzuführender Aktionen hin zu einem geistigen, internen Prozess. Im Fall des Konzepts Funktion werden Funktionen im *Process*-Stadium als eindeutige Abbildungen von Objekten aus einer Klasse auf Objekte einer möglicherweise anderen Klasse verstanden, ohne dass dazu die Abbildungsvorschrift explizit bekannt sein muss.

Charakteristisch für ein *Object*-Verständnis ist, dass Prozesse zu Objekten werden, die selbst wieder manipuliert werden können. Beispielsweise im Fall des Funktionenkonzepts erfordert das Konzept der Ableitung ein Objektverständnis von Funktionen, denn die Ableitung transformiert eine Funktion in eine neue Funktion. Das Konzept der Ableitung setzt also mit dem *Object*-Verständnis ein ziemlich elaboriertes Verständnis des Konzepts Funktion voraus. Der Mechanismus, der vom *Process*- zum *Object*-Stadium führt, wird als Kapselung bezeichnet.

Das *Schema*-Stadium schließlich ist dadurch charakterisiert, dass ein Konzept mit weiteren mathematischen Konzepten verbunden wird und daraus ein neues Konzept entsteht, das jeweils wieder als *Action*, *Process* und *Objekt* konzeptualisiert werden kann.

Die didaktische Konsequenz der APOS-Theorie besteht darin, dass Studierenden Gelegenheit gegeben werden muss ihr Verständnis mathematischer Konzepte entlang der namensgebenden Phasen weiterzuentwickeln. Studierenden muss also die Gelegenheit zu Internalisierung bzw. Kapselung gegeben werden. Internalisieren von Aktionen zu mentalen Prozessen kann dadurch begünstigt werden, dass Studierende Aktionen wiederholt ausführen und ihr Vorgehen reflektieren. Dies kann dadurch geschehen, dass Studierende Code, der spezifische Berechnungen ausführt, durch Code ersetzen, der die Berechnung für unspezifizierte Werte ausführt. Um beispielsweise die Internalisierung des Konzepts Assoziativität für eine bestimmte Operation $\otimes$ zu unterstützen, kann eine Programmiertätigkeit hilfreich sein, in der Studierende für die Operation $\otimes$ ausgehend von Befehlen wie `(1⊗2)⊗3==1⊗(2⊗3)` und `(0⊗2)⊗42==0⊗(2⊗42)` Programmcode schreiben, der die Assoziativitätseigenschaft nicht nur anhand konkreter Fälle, sondern für alle möglichen Belegungen aus einer Grundmenge überprüft (vgl. auch Abschnitt 8.3.2).

Aus Sicht der APOS-Theorie hat Programmieren in der Mathematik jenseits der in Abschnitt 8.1 genannten Funktionen die wesentliche Funktion die Entstehung mentaler Strukturen bei Studierenden zu unterstützen. Dies erfordert zunächst keine automatisierte Bewertung der von Studierenden erstellten Programme. Wenn Programmieren allerdings zu einem wesentlichen Bestandteil der Mathematiklehre wird und Studierenden durch Programmieraktivitäten die Gelegenheit gegeben werden soll, ihr Verständnis mathematischer Konzepte zu entwickeln und weiterzuentwickeln, muss dies unter anderem an prominenter Stelle, das heißt während der Kontaktzeit, geschehen. Die für die Hochschullehre typischen hohen Teilnehmerzahlen erlauben es jedoch nicht solche kritischen und denkintensiven Aktivitäten von Studierenden in angemessener Intensität zu betreuen. Dabei kann (Teil-) Automatisiertes Feedback wertvolle Unterstützung leisten. Zudem ermöglicht eine automatisierte Programmbewertung Lehrszenarien, die auf regelmäßigem und zeitnahem Feedback basieren [BT11].

8.3 Besonders geeignete Aufgabentypen

In diesem Abschnitt werden Aufgabentypen vorgestellt, welche sich besonders für eine auf APOS-Theorie beruhende Lehre eignen. Normale Rechenaufgaben lassen sich mit gängigen Lernmanagementsystemen einsetzen, indem die Studierenden eine Zahl oder Formel eingeben, welche dann direkt mit dem Ergebnis verglichen wird oder über ein CAS ausgewertet wird. Aus Sicht der APOS-Theorie sind aber Aufgaben interessanter, in denen Studierende mathematisch modellieren und selbst konstruieren, wie zum Beispiel Aufgaben, in denen Studierende eine Datenstruktur (wie beispielsweise eine Menge) definieren. Diese lassen sich zum Teil auch noch mit einem CAS überprüfen. Noch interessanter sind Aufgaben, bei denen ein mathematischer Sachverhalt als Funktion[1] modelliert wird, da damit mathematische Eigenschaften definiert und angewendet werden können. Funktionen können im Allgemeinen nicht mit einem CAS überprüft werden, da es viele verschiedene Lösungsmöglichkeiten geben kann, welche nicht durch einen einfachen Vergleich oder über eine Formel auf Richtigkeit geprüft werden können. Funktionen können aber mit Unit-Tests evaluiert werden auf ähnliche Art wie diese Technik im Software-Engineering eingesetzt wird. Man testet also, ob für eine Auswahl von Eingabewerten die richtigen Ausgabewerte produziert werden. Die Tests müssen gut gewählt sein, damit mögliche Fehler auch tatsächlich gefunden

1 Wir bezeichnen in diesem Kapitel alles als Funktion, was Eingabe- und Ausgabewerte hat, und unterscheiden nicht zwischen Methoden, Subroutinen, Prozeduren, Funktionen und Abbildungen.

werden. Außerdem benötigt man für viele Aufgaben noch weitere Tests, die ausschließen, dass die Studierenden „schummeln“ und zum Beispiel in der Programmiersprache schon vordefinierte Funktionen verwenden oder fundamental andere Lösungswege wählen, als sie es bei einer bestimmten Aufgabe tun sollen. Gegebenenfalls bietet sich auch eine Plagiatsüberprüfung an. Diese stößt aber insbesondere bei kurzen mathematischen Aufgaben dann an ihre Grenzen, wenn es nur eine begrenzte Anzahl verschiedener Lösungsansätze gibt und die Lösungen sich somit zwangsläufig ähnlich sehen.

Verschiedene Programmiersprachen kommen für Mathematikaufgaben in Frage. Ein Vorteil von imperativen Sprachen ist, dass Studierende oft schon damit vertraut sind oder diese parallel lernen. Aus technischer Sicht ist es relativ belanglos, welche Sprache verwendet wird, solange es möglich ist, Tests zu spezifizieren, welche in Batchverarbeitung auf die studentische Einreichung angewendet werden können. Aus pädagogischer Sicht ist es natürlich von Vorteil, wenn die Notation der Programmiersprache möglichst mathematiknah ist. Daher verwenden wir in diesem Kapitel die Sprache SetlX[2], die eine Weiterentwicklung von Jack Schwartz' SETL (Set Language) ist, welche auch sonst in auf APOS-Theorie beruhenden Textbüchern eingesetzt wird [LD95]. Im Prinzip könnte man auch Python verwenden, da sich SetlX und Python recht ähnlich sind. Ein Vorteil von Python ist, dass es sich um eine bekanntere Sprache handelt, für die es auch viele andere Anwendungsmöglichkeiten gibt. Allerdings dürfen Mengen in Python keine „mutablen Objekte“ (wie zum Beispiel Mengen oder Listen) enthalten. Daher kann man in Python Mengen von Mengen nur dann darstellen, wenn die innere Menge als unveränderlich deklariert wird. Außerdem sind die Darstellungsformen von SetlX noch näher an der mathematischen Notation als die von Python. Für eine einführende Mathematikvorlesung ist SetlX daher die geeignetere Sprache.

Im Folgenden stellen wir verschiedene Aufgabentypen vor, die insbesondere in Vorlesungen zur Einführung in die Mathematik und zu Diskreten Strukturen verwendet werden können. Unsere Liste beruht auf eigenen Erfahrungen, auf der Liste von Farahani & Uhlig [FU09] und auf Aufgaben aus Textbüchern, welche die APOS-Theorie einsetzen [LD95].

8.3.1 Mengendefinition durch Set-Comprehension

Mengen können bekanntermaßen entweder durch Aufzählung der Elemente oder durch die sogenannte Set-Builder- (oder Set-Comprehension-) Notation beschrie-

2 http://www.randoom.org/Software/SetlX

ben werden. Zum Beispiel kann die Menge der geraden Zahlen mathematisch durch $\{n \mid n \in \mathbb{Z} \wedge n \bmod 2 = 0\}$ und in der Programmiersprache SetlX durch `{n : n in [1..100]| n % 2 == 0};` beschrieben werden. Ein Unterschied zwischen der mathematischen und der programmiersprachlichen Notation ist, dass in der Programmiersprache nur endliche Mengen verwendet werden können. Ansonsten sind sich die beiden Notationen aber recht ähnlich. Ein wesentliches Einsatzgebiet für Aufgaben, in denen die Studierenden Set-Comprehension Notation verwenden sollen, ist das Üben von logischen Ausdrücken einschließlich von Quantoren (forall und exists). Um die Mengen korrekt zu bilden, müssen die Studierenden logische Ausdrücke richtig anwenden. Tupel (oder Listen) lassen sich ebenso definieren, da sie sich in SetlX von Mengen nur durch die Art der Klammern unterscheiden. Da Listen und Mengen auch summiert werden können, lassen sich auch Folgen und Reihen auf diese Art darstellen.

8.3.2 Mathematische Eigenschaften verstehen

Ein Vorteil des Einsatzes von Programmiersprachen ist, dass sich viele mathematische Eigenschaften (oder Axiome) direkt darstellen und überprüfen lassen. Das Assoziativgesetz kann beispielsweise wie folgt implementiert werden:

```
istAssoziativ := procedure(set,operat){
   return forall(a in set, b in set, c in set|
   operat(a,operat(b,c)) == operat(operat(a,b),c));
};
```

Der Anhang dieses Kapitels enthält für dieses Beispiel eine mögliche Aufgabenstellung und Unit-Tests. Eine komplexere Struktur „istGruppe“ kann dann durch Zusammenfügen der Axiome (Assoziativität usw.) definiert werden. Die Eingaben für die Funktion „istAssoziativ“ sind eine Menge und eine Operation (zum Beispiel `addition := procedure(a,b){return a+b;};`). Die Studierenden können zum einen dadurch, dass sie die Eigenschaften selbst implementieren, sehen, wie diese genau funktionieren. Zum anderen können sie mit Beispielen experimentieren, auf die die Eigenschaften zutreffen oder nicht zutreffen, und somit durch Programmieren ein Gefühl dafür entwickeln, wie sich die Strukturen verhalten. Das Übergeben der Operationen als Parameter fördert außerdem ein abstraktes Verständnis des Operationsbegriffs.

8.3.3 Ein abstrakter Funktionsbegriff

Operationen, die wie im vorherigen Beispiel als Parameter übergeben werden, sind aus Programmiersprachensicht als Funktionen implementiert. Diese Darstellung kann im Unterricht thematisiert werden. Neben mehrstelligen, partiellen, rekursiven und verketteten Funktionen können Funktionen auch anonym definiert, verschachtelt und als Ausgabewert zurückgegeben werden. Damit sind Lambda-Ausdrücke darstellbar und insgesamt ein sehr abstrakter Umgang mit Funktionen möglich. Für mathematische Grundlagenveranstaltungen sind davon vermutlich hauptsächlich die rekursiven Funktionsdefinitionen von Interesse.

Außerdem kann es zumindest für Informatikstudierende hilfreich sein, ein Funktionsverständnis zu entwickeln, welches sowohl mathematische Funktionen umfasst als auch die vielfältigen Anwendungen von Funktionen in Programmiersprachen, die sie schon aus anderen Vorlesungen kennen. Damit kann auch ein Beitrag geschaffen werden, Studierenden zumindest die Möglichkeit zu geben Studieninhalte miteinander zu vernetzen. Die wichtige Aufgabe „Wissensinseln“ zu einem kohärenten Ganzen zu verbinden wird vielleicht selten in Lehrveranstaltungen aktiv unterstützt. Es ist zumindest fraglich, ob Lehre davon ausgehen kann, dass ein Großteil der Studierenden diese Integration ohne Hilfe leisten kann oder ohne Anlass leistet.

8.3.4 Algorithmisches Denken in der Mathematik

Algorithmisches Denken hat in der Informatik vermutlich einen ähnlichen Stellenwert wie logisches Denken in mathematischen Beweisen. Algorithmen lassen sich gut mit automatisch bewerteten Programmieraufgaben üben – für Beweisverfahren, die nicht nur aus Umformen bestehen, ist das schwieriger oder unmöglich. In einigen mathematischen Gebieten, wie zum Beispiel in der Graphentheorie, sind aber Algorithmen auch von Bedeutung. Studierenden können durch das Programmieren von Algorithmen kreativer tätig werden, als das bei den anderen schon vorgestellten Aufgabentypen der Fall ist. Zumindest Informatikstudierende kann man eventuell für Algorithmen eher begeistern als für Beweisverfahren. Außerdem lassen sich durch Algorithmen auch gut Grenzen erfahren. Zum Beispiel ist es bei Primzahlen schon so, dass nur relativ effiziente Algorithmen noch 4-stellige Primzahlen bestimmen können.

8.3.5 Weitere Aufgabentypen

Viele weitere Aufgabentypen sind denkbar. Farahani & Uhlig [FU09] erwähnen Aufgaben zu grundlegenden Abzählverfahren der Kombinatorik. Zum Beispiel kann das Auswählen von drei Elementen aus einer Menge durch drei geschachtelte Schleifen dargestellt werden. Dass beim Auswählen ohne Zurücklegen die Elemente verschieden sein müssen, kann dann über eine if-Abfrage hinzugefügt werden. Ebenso kann über eine Änderung des Datentyps (Liste oder Menge) oder durch Sortierung der Elemente gesteuert werden, ob die Reihenfolge der Elemente beachtet werden soll oder nicht.

In der Programmiersprache Python gibt es durch die Verbindung zum CAS SageMath eine immense Auswahl von vorprogrammierten Funktionen für alle Gebiete der Mathematik. Studierende können das Anwenden solcher Funktionen im Zusammenhang komplexer Problemstellungen üben. In SetlX gibt es nur eine geringere Auswahl vorprogrammierter Funktionen. Daher eignet sich SetlX eher für das Üben mathematischer Grundbegriffe, während Python auch für weiterführende Anwendungen verwendbar ist. Es ist aber nicht schwierig für Lehrende, auch in SetlX selbst Funktionen zu definieren, welche von den Studierenden verwendet werden können, zum Beispiel um graphische Darstellungen zu ermöglichen.

8.4 Die Erstellung von Tests

Wie bereits erwähnt, reicht bei einigen Aufgabentypen ein CAS zur Bewertung aus – bei Funktionsaufrufen benötigt man aber Unit-Tests. Für den Einsatz von Unit-Tests muss den Studierenden der Name der zu testenden Funktion und die Liste der Eingabeparameter vorgegeben werden und vereinbart werden, welche Datentypen die Funktion zurückgeben soll (siehe Anhang). Der Test führt dann die eingereichte Funktion mit den Eingabewerten aus und vergleicht die Ausgabewerte mit den Ausgabewerten einer Musterlösung. Einige Aspekte des Modellierens mathematischer Probleme werden den Studierenden durch die benötigten detaillierten Aufgabenstellungen daher vorenthalten.

Für Lehrende stellt sich die Herausforderung, die Tests so vorzubereiten, dass es tatsächlich für jeden möglichen Fehler eine Überprüfung durch einen Test gibt. Bei einigen Aufgaben ist das leicht, weil man die Eingabewerte so typisieren kann, dass alle Fälle getestet werden können (wobei Extremfälle wie zum Beispiel die leere Menge besonders beachtet werden sollten, da die Studierenden diese oft übersehen). Bei einigen Aufgaben ist das Erstellen guter Tests aber auch schwie-

riger. Wenn zum Beispiel die Gruppenaxiome implementiert werden sollen, benötigt man als Testfälle jeweils Strukturen, die alle Axiome bis auf eins nicht erfüllen, damit man sieht, ob jedes Axiom korrekt implementiert ist. Das Axiom bezüglich inverser Elemente bedingt aber, dass ein neutrales Element existiert. Es gibt also keine Struktur, welche inverse Elemente ohne ein neutrales Element enthält. Ein Beispiel einer algebraische Struktur, die alle Gruppenaxiome außer der Assoziativität erfüllt, sind die Oktonionen. Insbesondere, wenn die Tests durch studentische Hilfskräfte geschrieben werden, kann man aber davon ausgehen, dass diese die Oktonionen nicht kennen und vermutlich den Test für das Assoziativgesetz weglassen werden.

Tests können also aus verschiedenen Gründen unvollständig oder auch fehlerhaft sein. Eine gute Möglichkeit fehlende oder falsche Tests zu entdecken besteht darin, dass man die studentischen Einreichungen zumindest beim ersten Einsatz der Aufgaben auch manuell kontrolliert und aufpasst, ob es Einreichungen gab, die als korrekt bewertet wurden, obwohl sie fehlerhaft waren. Gleichzeitig kann man damit auch feststellen, ob es vielleicht Fehler aufgrund einer unklaren Aufgabenstellung gab, welche verbessert werden sollte. Es empfiehlt sich also, die Aufgaben und Tests einem Entwicklungszyklus zu unterstellen.

8.5 Einsatzszenarien und Erfahrungen

8.5.1 Erfahrungen mit Programmieraufgaben

Wir haben Programmieraufgaben bereits mehrfach in Mathematikveranstaltungen eingesetzt. Die Aufgaben wurden von den Studierenden gut angenommen und in studentischen Evaluationen zumindest nicht als schlechter als traditionelle Übungsaufgaben bewertet. Ein Test auf Plagiate ergab, dass im Durchschnitt 50% der Einreichungen plagiiert sein könnten. Den Studierenden wurde aber nicht explizit verboten, die Aufgaben vor der Einreichung in Lerngruppen zu diskutieren. Außerdem können insbesondere bei einfachen Aufgaben Übereinstimmungen auch zufällig sein. Bei schwierigeren Aufgaben scheint auf jeden Fall mehr plagiiert zu werden als bei einfacheren Aufgaben, da bei solchen Aufgaben die Anzahl der verschiedenen Einreichungen abnimmt, obwohl zufällige Ähnlichkeiten weniger wahrscheinlich sind. Laut einer anonymen Umfrage in einer Veranstaltung (mit 70 Studierenden) besprachen 65% der Studierenden die Programmieraufgaben miteinander, aber kaum jemand gab an, nur abzuschreiben. Wir wissen nicht, inwieweit sich dieses Verhalten vom Verhalten bei der Bearbeitung traditioneller Hausaufgaben unterscheidet, bei denen vermutlich auch abgeschrieben oder in

Lerngruppen gearbeitet wird. In der gleichen Veranstaltung haben wir auch die Klausur, in der die Studierenden auch Programmcode schreiben mussten, analysiert. Die Auswertung ergab, dass 10% der Studierenden, die die Hausaufgaben regelmäßig eingereicht hatten, überhaupt keinen Programmcode schreiben konnten. Es könnte also sein, dass 10% die tatsächliche Quote der Studierenden darstellt, welche die Programmieraufgaben als Lernmethode nicht annehmen. Wir wissen nicht, ob andere Methoden bei diesen Studierenden erfolgreicher wären oder nicht.

Im Allgemeinen zeigte die Klausur eine Korrelation zwischen Klausurnote und Programmierfähigkeit. Im Sinne der APOS-Theorie enthielt die Klausur eine geringe Anzahl von Aufgabenteilen, welche rezeptartig (im *Action*-Stadium) lösbar waren. Da es sich um eine Erstsemesterveranstaltung handelte, gab es auch nur eine geringe Anzahl von Aufgaben, welche ein *Object*-Stadium voraussetzten, und gar keine zum *Schema*-Stadium. Die meisten Aufgabenteile überprüften also das Verstehen mathematischer Konzepte im *Process*-Stadium. Interessanterweise gab es einen Aufgabenteil zum *Object*-Verständnis von Gruppen, welcher mit dem Erreichen der Note 3 korrelierte, und zwar nicht aufgrund der Bepunktung, da diese nicht höher war als für andere Aufgabenteile. Fast alle Studierenden mit Note 3 und fast keine Studierenden mit einer schlechteren Note hatten diesen Aufgabenteil richtig gelöst. Im Großen und Ganzen bestätigte die Klausur damit unsere Erwartungen zur APOS-Theorie und zum Einsatz von Programmieraufgaben.

Für die meisten Studierenden scheint der Einsatz von Programmieraufgaben somit hilfreich oder zumindest nicht nachteilig zu sein. Wir haben aber in der Klausur beobachtet, dass es einerseits einige wenige Studierende zu geben scheint, die mathematische Notation wesentlich besser lesen und schreiben können als Programmcode, aber andererseits auch Studierende, die Programmieren können, aber keine mathematische Notation lesen und schreiben. Es wäre interessant, genauer zu untersuchen, warum das der Fall ist. Möglicherweise gibt es einerseits selbst bei Aufgaben, die auf *Process*-Verständnis zielen, noch Studierende, die diese Aufgaben ohne Verständnis anhand von Regeln und Strategien lösen. Und zwar gibt es bei einigen Themen nur eine begrenzte Art von möglichen Aufgabenstellungen, so dass Klausuraufgaben eventuell vorher behandelten Aufgaben ähneln und trainiert werden können. Andererseits ist vielleicht die imperative Struktur von Programmcode leichter verständlich für einige Studierende als die mathematische Notation. Es gibt also noch viele offene Fragen bezüglich des Einsatzes von Programmieraufgaben mit (oder auch ohne) Berücksichtigung der APOS-Theorie in der Mathematiklehre.

8.5.2 Studentisches Verständnis mathematischer Ausdrücke

Wie in der Einleitung geschildert verstehen wir Programmieren auch als Dialog. Aus dieser Perspektive betrachtet schaffen Programmierübungen für Lehrende die Gelegenheit Studierenden bei solchen Dialogen „zuzuhören“ und dabei herauszufinden, welche Aspekte des Stoffes diesen schwerfallen und womit diese Schwierigkeiten haben. Die Präzision, die eine Programmierspache hinsichtlich des Formulierens mathematischer Sachverhalte fordert, erlaubt es mitunter studentische Schwierigkeiten ebenso präzise wahrzunehmen.

In einer unserer Lehrveranstaltungen ist dieses Diagnosepotenzial beispielsweise bei der folgenden Aufgabenstellung zutage getreten. Studierende sollten mittels SetlX überprüfen, ob eine gegebene Funktion f mit Definitionsmenge $\mathbb{D}$ injektiv ist, ob also für alle $x_1, x_2 \in \mathbb{D}$ gilt, dass aus $x_1 \neq x_2$ auch $f(x_1) \neq f(x_2)$ folgt. Ein merklicher Teil der Studierenden hat als Lösung einen Ausdruck der Art

```
forall (x1 in domain, x2 in domain | f(x1) != f(x2) );
```

formuliert (wobei `domain` hier für $\mathbb{D}$ steht). In diesem Ausdruck ist die erforderliche Bedingung $x_1 \neq x_2$ nicht genannt. Diese Studierenden gehen also potenziell davon aus, dass $x_1 \neq x_2$ automatisch erfüllt ist. Gespräche mit den Studierenden haben diese Vermutung bestätigt wobei sich herausgestellt hat, dass für sie $x_1 \neq x_2$ eine Folge von $x_1, x_2 \in \mathbb{D}$ ist, zum Beispiel weil sie sich vorstellen, dass ein Element aus $\mathbb{D}$ ohne Zurücklegen entnommen wird und x_1 als Wert zugewiesen wird, und daher für x_2 nicht mehr als Wert zur Verfügung stehen kann.

8.5.3 Erfahrungen der Lehrenden

Es ist sicherlich keine neue Erkenntnis, dass die Sprache der Mathematik nicht immer alles im Detail expliziert. Beispielsweise ist eine gängige Formulierung der Assoziativität einer Operation $\otimes$:

$$a \otimes (b \otimes c) = (a \otimes b) \otimes c.$$

Dabei wird nicht explizit genannt, dass dies für alle Werte von a, b und c aus einer ebenfalls nicht benannten Menge gelten muss. Experten können damit umgehen, dass nicht alles explizit genannt ist. Bei Studierenden ist es allerdings zweifelhaft, ob diese den Aussagegehalt verstehen können, ohne dass Lehrende sie darauf aufmerksam machen, was implizit ist und wie man das Implizite herausarbeiten kann.

Uns als Lehrende hat die Formulierung von Programmieraufgaben immer wieder vor Augen geführt, wie viel Implizites in mathematischen Aussagen enthalten

ist. Unabhängig vom tatsächlichen Einsatz der Programmieraufgaben hatten wir so häufig Gelegenheit, potenzielle Schwierigkeiten der Studierenden als Experten selbst wieder wahrzunehmen. Gleichzeitig hat uns dies angeregt, derartige Verständnisschwierigkeiten in der Lehrveranstaltung zu thematisieren.

8.6 Fazit und Ausblick

In diesem Beitrag haben wir verschiedene Aspekte des Einsatzes von automatisch bewertbaren Programmieraktivitäten in der Mathematikausbildung beleuchtet. Wir haben einerseits aus der Perspektive der APOS-Theorie heraus Argumente genannt, die für einen solchen Einsatz sprechen, und andererseits potenzielle Folgen benannt und von ersten Erfahrungen berichtet.

Aus technologischer Sicht besteht eine besondere Herausforderung im Erstellungsprozess der Aufgaben. Das Formulieren der Unit-Tests kann sich auch für elementare Aufgabenstellungen als äußerst herausfordernd erweisen, wenn potenzielle Fehler trennscharf identifiziert werden sollen.

Aus didaktischer Sicht erweisen sich Programmieraufgaben für uns in zweifacher Hinsicht als wertvoll. Zum einen erlauben sie Einblicke in konzeptuelle Verständnisschwierigkeiten der Studierenden. Wenn Lehre auch als Identifikation charakteristischer studentischer Schwierigkeiten gepaart mit Anstrengungen, dabei zu helfen, diese zu überwinden, verstanden wird [Rie14], braucht es Gelegenheiten, stoffbezogene Verständnisschwierigkeiten zu identifizieren. Programmierübungen können dies leisten. Zum anderen hat das Formulieren von Programmierübungen uns sehr deutlich vor Augen geführt, wie sehr mathematische Aussagen in gängigen Lehrbüchern nicht explizit genannte Dinge voraussetzen. Insofern hat uns die Beschäftigung mit Programmieraufgaben unabhängig von deren tatsächlichem Einsatz wertvolle Einblicke in die Herausforderungen gegeben, die mit der Lehre von Mathematik verbunden sind.

Der Aspekt der Wirksamkeit steht für uns dabei nicht, beziehungsweise zu diesem Zeitpunkt noch nicht, im Vordergrund. Aus einer Sicht des *Constructive Alignment* [BT11], also der Wechselwirkung von Lernzielen, Lehrmethode und Prüfung, hat der Einsatz von Programmieraufgaben in Bezug auf die Didaktik für uns einen Reflexionsprozess angestoßen: Die Veränderung in der Lehrmethode hat zu wertvollen Erkenntnissen unter anderem hinsichtlich der Schwierigkeit des Lernstoffs und charakteristischer Probleme von Studierenden mit diesem Stoff geführt. Diese Erkenntnisse fordert uns auf, über unsere Lernziele nachzudenken und zu entscheiden, welches Gewicht diese Aspekte in unseren Lehrveranstaltungen haben sollen.

Anhang: Beispiel einer Programmieraufgabe

Der Aufgabentext muss genau spezifizieren, was implementiert werden soll, einschließlich der Namen von Variablen oder Funktionen, die in den Tests aufgerufen werden. Wie viel sonstige Hilfestellung (mathematische Definitionen, Tipps zur Implementierung) gegeben wird, hängt davon ab, was die Studierenden schon vorher im Unterricht gelernt haben und welche anderen Aufgaben sie schon programmiert haben.

Beispiel einer Aufgabenstellung:

> Eine binäre Verknüpfung $* : A \times A \to A$ auf einer Menge A heißt *assoziativ*, wenn für alle $a, b, c \in A$ gilt: $a * (b * c) = (a * b) * c$.
> Schreiben Sie in SetlX eine Funktion **istAssoziativ**, die (in dieser Reihenfolge) eine Menge und eine Verknüpfung entgegennimmt und entscheidet, ob die Verknüpfung bezüglich der Menge assoziativ ist.
> Tipps: Die Verknüpfung können Sie auch als Funktion definieren, z. B. `add := procedure(a,b) {return a+b;};`. Ein Beispiel eines Funktionsaufrufs, der True zurückgibt ist dann: `istAssoziativ({1,2,3},add)`. Überlegen Sie sich auch weitere Beispiele für Funktionsaufrufe, die jeweils True oder False zurückgeben.

Musterlösung (zum Testen der Tests):

```
istAssoziativ := procedure(set,operat){
 return forall(a in set, b in set, c in set|
 operat(a,operat(b,c)) == operat(operat(a,b),c));
};
```

Für die Tests müssen Beispielmengen und Verknüpfungen definiert werden. Jedes print-Statement enthält zwischen den $-Zeichen einen Ausdruck, der True zurückgibt, wenn der Test bestanden ist, und sonst False. Es ist eine Herausforderung an Autoren, sich genau zu überlegen, welche Arten von Tests wirklich alle möglichen Fehlerquellen abdecken. Die ersten beiden Testfälle enthalten jeweils ein positives und negatives Beispiel der Assoziativität. Im dritten Test wird Assoziativität für einen String-Datentyp getestet, um sicherzustellen, dass die Funktion abstrakt genug definiert ist. Im letzten Testfall wird die leere Menge getestet, da nach unserer

Erfahrung Studierende manchmal Lösungen mit anderen Programmierkonstrukten implementieren, bei denen die Grenzfälle nicht funktionieren.

Testskript:

```
set1 := {1,2,3,4};
set2 := {};
set3 := {"a","b","c"};
add := procedure(a,b) {return a+b;};
minus := procedure(a,b) {return a-b;};
concat := procedure(a,b) {return a+" "+b;};
print("Test Addition:$istAssoziativ(set1,add) == true$");
print("Test Subtraktion:$istAssoziativ(set1,minus) == false$");
print("Test anderer Datentyp:$istAssoziativ(set3,concat) == true$");
print("Test Leere Menge:$istAssoziativ(set2,minus) == true$");
```

Literatur für dieses Kapitel

[Arn+13] Ilana Arnon u. a. *APOS theory: A framework for research and curriculum development in mathematics education*. Springer Science & Business Media, 2013.

[BT11] John Biggs und Catherine Tang. *Teaching for quality learning at university: What the student does*. McGraw-Hill Education (UK), 2011.

[FU09] Ali Farahani und Ronald P. Uhlig. „Use of Python in Teaching Discrete Mathematics“. In: *American Society for Engineering Education Annual Conference and Exposition*. American Society for Engineering Education. 2009.

[Hol95] Judy Holdener. „Calculus&Mathematica: Instructors' Perspectives on Continuing Controversies“. In: *Mathematica in Education and Research* 6 (1995), S. 6–10.

[LD95] Uri Leron und Ed Dubinsky. „An abstract algebra story“. In: *The American Mathematical Monthly* 102.3 (1995), S. 227–242.

[MR11] Philipp Marwan und Peter Riegler. „Entwicklung des Funktionenkonzepts bei Studierenden der Informatik“. In: *Wismarer Frege-Reihe* 2 (2011).

[Rie14] Peter Riegler. „Schwellenkonzepte, Konzeptwandel und die Krise der Mathematikausbildung“. In: *Zeitschrift für Hochschulentwicklung* 9.4 (2014), S. 241–257.

[WM06] Anne Watson und John Mason. *Mathematics as a constructive activity: Learners generating examples*. Routledge, 2006.

Teil II

Systeme zur automatisierten Programmbewertung

9 Der Grader JACK

Michael Striewe

Zusammenfassung
JACK ist ein webbasiertes Bewertungssystem für Übungs- und Prüfungsaufgaben, das seit 10 Jahren für die Bewertung von Programmieraufgaben eingesetzt wird. In diesem Kapitel werden Aufgabendesign, Feedbackmöglichkeiten und Einsatzerfahrungen vorgestellt.

9.1 Einleitung

Zur Unterstützung der Einführung in die Programmierung für Erstsemester wurde an der Universität Duisburg-Essen im Jahr 2006 ein System zur automatischen Bewertung von Programmieraufgaben entwickelt und aufgrund seiner Aufgabe als „Java Checker“ auf den Namen JACK getauft [SGB08]. Im Laufe mehrerer Jahre wurde JACK im Rahmen von Forschungsaktivitäten, Förderprojekten des BMBF und studentischen Projekten immer weiter erweitert, so dass das System als Framework für ganz verschiedene Arten von Aufgabenstellungen mit automatischem Feedback genutzt werden kann [SZG15]. Im Folgenden wird jedoch weitgehend nur das ursprüngliche Kerngeschäft von JACK, d. h. die automatische Bewertung von Programmieraufgaben in der Programmiersprache Java betrachtet.

JACK ist ein webbasiertes System, das grundsätzlich unabhängig von anderen Systemen wie Lernmanagementsystemen betrieben werden kann und das sich sowohl für den Übungs- als auch für den Prüfungsbetrieb eignet. An der Universität Duisburg-Essen wird eine Installation mit Anbindung an den LDAP-Server der Universität betrieben, so dass sich die Studierenden im Übungsbetrieb mit ihren üblichen Login-Daten anmelden können. Im Prüfungsbetrieb wird das System in einem anderen Login-Modus betrieben, in dem individuelle Einmalpassworte für die Prüfungen zum Einsatz kommen.

JACK ist nicht open-source, steht jedoch deutschen Hochschulen frei zur Verfügung, da die Entwicklung in den letzten Jahren primär über ein BMBF-Projekt

finanziert wurde. Weitere Informationen und Kontaktmöglichkeiten können der Projektwebseite[1] entnommen werden.

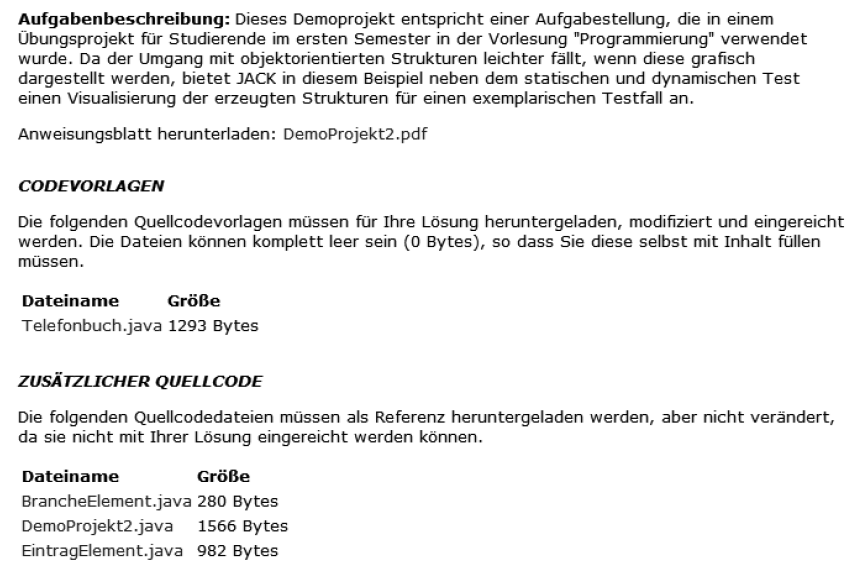
9.2 Aufgabendesign

Programmieraufgaben in JACK bestehen grundsätzlich aus einer oder mehreren Dateien Quellcode, die den Studierenden als Vorlage zur Verfügung gestellt werden und die nach der Bearbeitung auf den Server hochgeladen werden müssen. Dies erfolgt im Übungsbetrieb über den Browser, während für Prüfungen ein spezielles Plugin für die Entwicklungsumgebung Eclipse zur Verfügung steht, das die Vollständigkeit der Dateien und die richtige Einordnung in ein Projekt in der Entwicklungsumgebung sicherstellt. Auf diesem Wege können über JACK auch vollständige Eclipse-Projekte verteilt und eingesammelt werden.

Die Aufgaben können so gestaltet werden, dass nicht alle zur Verfügung gestellten Dateien bearbeitet werden müssen oder dürfen. Es kann auch nur eine Teilmenge der Dateien wieder eingesammelt werden. Vorlagendateien dürfen ferner leer sein, wenn diese von den Studierenden komplett selbst gefüllt werden sollen. Es ist jedoch nicht möglich, den Studierenden völlige Freiheit bei der Zahl und Benennung ihrer Dateien zu lassen, sofern nicht im Prüfungsmodus ganze Projekte eingesammelt werden. Ebenso wenig können Programmieraufgaben als Lückentexte im Browser angezeigt werden, in denen die Studierenden nur einzelne Zeilen ergänzen müssen. Derartige Aufgaben können in JACK über andere Aufgabentypen gestellt werden. Dann steht allerdings nicht der volle Umfang der automatischen Programmbewertung zur Verfügung.

Zu jeder Programmieraufgabe können vom Autor beliebig viele weitere versteckte Dateien definiert werden, die der Durchführung der Bewertung und der Erzeugung von Feedback dienen. Dazu werden so genannte Checker-Module für die Aufgabe konfiguriert, die jeweils einen Teil der gewünschten Funktionalität umsetzen. Für jede Aufgabe können prinzipiell beliebig viele Checker-Module konfiguriert werden, die für die Punktevergabe pro Aufgabe beliebig gewichtet werden können. Es ist auch möglich, Module ohne Gewichtung zu verwenden, so dass deren Feedback angezeigt wird, ohne Einfluss auf die Punktevergabe zu nehmen. Ebenso ist es möglich, Module mehrfach einzubinden, wenn ein Teil der von ihnen erzeugten Rückmeldung nicht in die Bewertung einfließen oder nur für Lehrende sichtbar sein soll. Lehrende haben unabhängig von der Konfiguration von Checker-Modulen immer die Möglichkeit, manuell Lösungen zu kommentieren und Punkte zu vergeben sowie automatisch erzeugte Meldungen zu unterdrücken.

1 http://www.s3.uni-duisburg-essen.de/jack/



Aufgabenbeschreibung: Dieses Demoprojekt entspricht einer Aufgabestellung, die in einem Übungsprojekt für Studierende im ersten Semester in der Vorlesung "Programmierung" verwendet wurde. Da der Umgang mit objektorientierten Strukturen leichter fällt, wenn diese grafisch dargestellt werden, bietet JACK in diesem Beispiel neben dem statischen und dynamischen Test einen Visualisierung der erzeugten Strukturen für einen exemplarischen Testfall an.

Anweisungsblatt herunterladen: DemoProjekt2.pdf

CODEVORLAGEN

Die folgenden Quellcodevorlagen müssen für Ihre Lösung heruntergeladen, modifiziert und eingereicht werden. Die Dateien können komplett leer sein (0 Bytes), so dass Sie diese selbst mit Inhalt füllen müssen.

Dateiname	Größe
Telefonbuch.java	1293 Bytes

ZUSÄTZLICHER QUELLCODE

Die folgenden Quellcodedateien müssen als Referenz heruntergeladen werden, aber nicht verändert, da sie nicht mit Ihrer Lösung eingereicht werden können.

Dateiname	Größe
BrancheElement.java	280 Bytes
DemoProjekt2.java	1566 Bytes
EintragElement.java	982 Bytes

Abbildung 9.1: Beispielaufgabe aus Sicht der Studierenden. Die Aufgabe ist auf dem JACK-Demo-Server unter https://jack-demo.s3.uni-due.de/ frei verfügbar.

Als durchgehendes Beispiel zur Erläuterung der verschiedenen Möglichkeiten von JACK soll folgende Übungsaufgabe dienen: Inhalt der Aufgabenstellung ist das Erstellen einer Telefonbuchverwaltung, mit der Personen- und Brancheneinträge erstellt, geändert und gesucht werden können. Abbildung 9.1 zeigt die Aufgabenstellung mit Downloadmöglichkeiten aus Sicht der Studierenden. Sichtbar ist eine kurze Aufgabenbeschreibung sowie Links zum Herunterladen des Aufgabenblattes, der Codevorlagen und weiterer Codedateien, die für die Lösung nicht verändert werden sollen. Es ist den Studierenden überlassen, wie sie die Dateien auf ihrem Rechner ablegen und mit welchen Werkzeugen sie sie bearbeiten. Die Einreichung der Lösung erfolgt über eine weitere Seite (siehe Abbildung 9.2), über

Abbildung 9.2: Ansicht zum Hochladen einer Lösung für die Beispielaufgabe aus Abbildung 9.1.

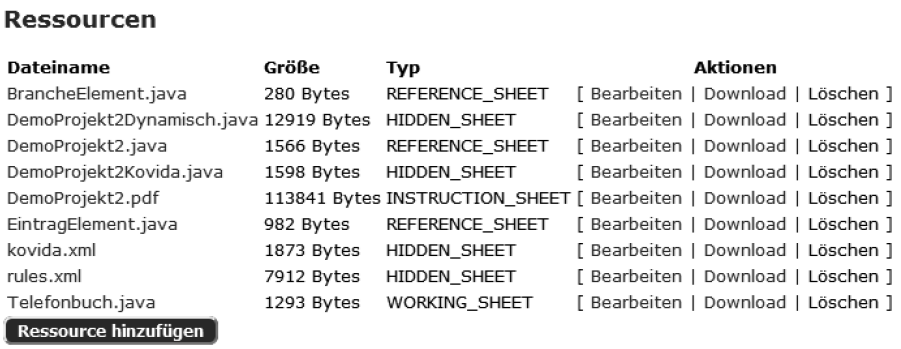


Abbildung 9.3: Auflistung aller zur Beispielaufgabe aus Abbildung 9.1 gehörenden Dateien aus Sicht des Aufgabenautors.

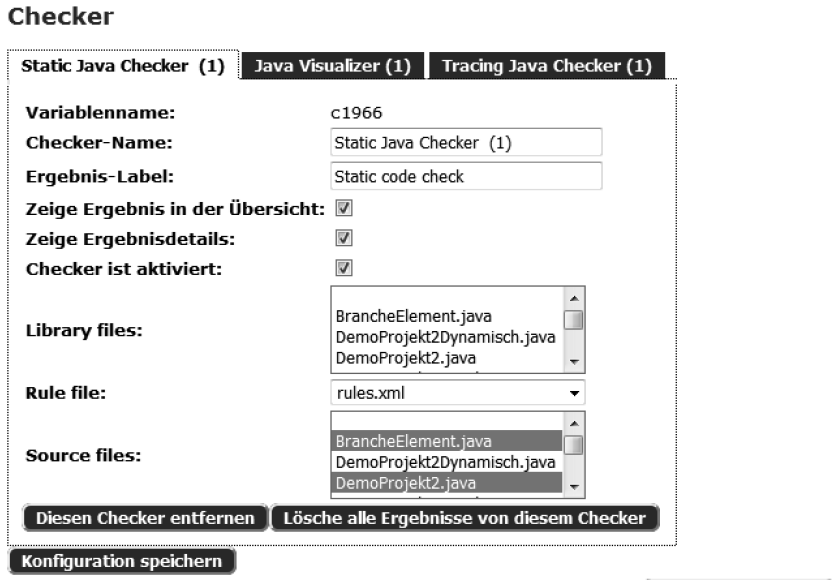


Abbildung 9.4: Konfiguration eines Checker-Moduls für die Beispielaufgabe aus Abbildung 9.1.

die nur die zur Bearbeitung vorgesehenen Codevorlagen wieder hochgeladen werden können. Aus Perspektive des Aufgabenautors besteht die Aufgabe neben der Grundkonfiguration (ohne Abbildung) insbesondere aus einer Liste von Dateien (siehe Abbildung 9.3) und der Konfiguration mehrerer Checker-Module (siehe Abbildung 9.4). Die Zuordnung von Dateien zu Checker-Modulen ist dabei beliebig und unabhängig von der Sichtbarkeit der Dateien für die Studierenden. Insbe-

sondere wird in dieser Aufgabe zwischen zu bearbeitendem Quellcode („working sheet“) und weiterem Quellcode („reference sheet“) unterschieden, aber alle diese Dateien werden allen Checker-Modulen zur Verfügung gestellt, da der Quellcode nur im Zusammenspiel aller Dateien kompilieren kann. Das Aufgabenblatt („instruction sheet“) ist dagegen für keines der Checker-Module relevant, während weitere Dateien für die Studierenden nicht sichtbar sind („hidden sheet“) und nur jeweils einem Modul zugewiesen werden, wie in den Abbildungen exemplarisch anhand der „rules.xml“ gezeigt.

9.3 Feedbackmöglichkeiten

Durch die Verwendung unabhängiger Checker-Module ist JACK sehr flexibel bezüglich der möglichen Rückmeldung zu den Lösungen einer Programmieraufgabe. Grundsätzlich müssen alle Checker-Module eine Punktzahl im Intervall von 0 bis 100 zurückgeben, wobei ein höherer Wert für eine bessere Lösung steht. Die Ergebnisse der einzelnen Checker-Module können gemäß einer Gewichtung oder eines komplexeren Ausdrucks miteinander verrechnet werden, um die Gesamtpunktzahl für die Lösung zu erhalten. Ferner können Checker-Module einen globalen Feedbacktext, eine Liste einzelner Fehlermeldungen sowie eine beliebige Menge an zusätzlichen Dateien zurückgeben, die während des Prüfvorgangs erstellt wurden. Im Folgenden werden die drei wichtigsten Checker-Module für Programmieraufgaben in Java vorgestellt.

9.3.1 Regelbasierte statische Prüfung

Die regelbasierte statische Prüfung von Programmcode zielt darauf ab, syntaktische und strukturelle Fehler und Schwächen einer Lösung zu entdecken und zu kommentieren. Sie besteht aus zwei Schritten: Im ersten Schritt wird die Einreichung kompiliert und gegebenenfalls werden die Fehlermeldungen des Compilers ohne weitere Verarbeitung an die Studierenden weitergereicht. Im zweiten Schritt erfolgt die Erzeugung eines Syntaxgraphen, auf den vom Aufgabenautor definierte Regeln angewandt werden können.

Jede dieser Regeln kann eine erwünschte oder unerwünschte Struktur definieren, die für den Fall ihres Auftretens bzw. Fehlens mit einem Feedback verknüpft wird. Zur Formulierung der Regeln kommt die Graphabfragesprache GReQL zum Einsatz, mit der beliebige Abfragen über Graphstrukturen definiert werden können. Die dazu verwendete Notation ähnelt Datenbankabfragen in SQL. Ein Editor

zur Unterstützung bei der Erstellung der Regeln ist über die o. g. Projektwebseite von JACK verfügbar.

Ein Beispiel für eine einfache Regel ist in Abbildung 9.5 angegeben. In dieser Regel wird in der `<query>` nach Methodendeklarationen gesucht, bei denen der Name mit einem Großbuchstaben beginnt, ohne dass es sich um die Deklaration eines Konstruktors handelt. Die Regel ist vom Typ `absence`, d. h. die beschriebene Struktur soll in einer korrekten Lösung nicht vorkommen. Kommt sie trotzdem vor, wird das in der Regel definierte Feedback ausgegeben. Eine solche Regel ist unabhängig von der konkreten Aufgabenstellung und kann daher für viele Aufgaben wiederverwendet werden. Regeln dieser Art können recht einfach für viele stilistische und strukturelle Aspekte eines Programms entwickelt werden. Ein weiteres Beispiel mit einer komplexeren Regel wird in Abbildung 9.6 gezeigt. In dieser Regel wird eine umfangreichere Struktur angegeben, nach der eine Methode namens `neu` den Konstruktor der Klasse `BrancheElement` aufrufen und dabei ihren eigenen Parameter an diesen Aufruf übergeben soll. Diese Struktur ist offensichtlich aufgabenspezifisch zu verwenden und müsste für andere Aufgaben angepasst werden. Die Regel ist vom Typ `presence`, so dass das Feedback ausgegeben wird, wenn die Lösung den beschriebenen Aufruf nicht enthält.

Das Erstellen derartiger Regeln erfordert vom Aufgabenautor eine gewisse Einarbeitung sowohl in die Notation der Abfragesprache als auch in den genauen Aufbau des Syntaxbaums von Java. Nach dieser Einarbeitung ermöglichen derartige Regeln aber eine hohe Flexibilität bei der Erstellung des Feedbacks. Für komplexe Fälle können dazu auch mehrere Abfragen in einer Regel kombiniert werden. Außerdem ist es möglich, Feedback auch für den positiven Fall zu geben, wenn die Studierenden nicht nur über einen Fehler, sondern auch über die Abwesenheit von Fehlern explizit informiert werden sollen. Neben der textuellen Rückmeldung

```
<rule type="absence" points="0">
    <query>from x : V{MethodDeclaration} with
        x.name=capitalizeFirst(x.name) and
        x.constructor="false"
        report 0 end
    </query>
    <feedback prefix="Hinweis (ohne Punktabzug)">
        Du verwendest Methodennamen, die mit einem Großbuchstaben
        beginnen. Das ist möglich, aber es entspricht nicht dem
        üblichen Programmierstil für Java.
    </feedback>
</rule>
```

Abbildung 9.5: Diese Regel zur statischen Codeprüfung sucht nach groß geschriebenen Methodennamen und gibt ein entsprechendes Feedback.

```
<rule type="presence" points="5">
    <query>from m : V{MethodDeclaration},
        c : V{ClassInstanceCreation},
        v : V{SingleVariableDeclaration},
        t : V{SimpleType}
        with
        m.name="neu" and
        m -->{MethodDeclarationParameters} v and
        m -->{Child}* c --> t and
        c -->&amp;{SimpleName} -->{Access} v and
        t.name="BrancheElement"
        report 0 end
    </query>
    <feedback>In der Methode "neu(String branche)" erfolgt kein
        Aufruf des Konstruktors von "BrancheElement", dem der
        übergebene String für den Namen weitergegeben wird.
    </feedback>
</rule>
```

Abbildung 9.6: Diese Regel zur statischen Codeprüfung überprüft, ob in einer bestimmten Methode ein vorgegebener Konstruktor aufgerufen wird.

kann jede Regel mit einem unterschiedlichen Punktgewicht versehen werden, um ihren Einfluss auf die Gesamtbewertung zu steuern.

9.3.2 Testfallbasierte dynamische Prüfung

Die dynamische Prüfung zielt darauf ab, die funktionale Korrektheit einer Lösung festzustellen und führt dazu Testfälle aus. Jeder Testfall muss vom Aufgabenautor durch eine Methode in einem Testtreiber definiert werden, wie dies auch bei klassischen Unit-Tests mit Werkzeugen wie JUnit der Fall ist. JACK bietet im Vergleich zu derartigen Werkzeugen jedoch einige abweichende Funktionen.

Erstens können in JACK Testfälle nicht nur bei einem Fehlschlag oder einem Erfolg eine definierte Fehlermeldung zurückgeben, sondern im Verlauf ihrer Ausführung nach Bedarf beliebig viele Meldungen erzeugen. Ebenso können Testfälle je nach Verlauf oder Ergebnis unterschiedliche Auswirkungen auf die Punktzahl haben und es können beliebige Abhängigkeiten zwischen den Testfällen erzeugt werden. Die Entscheidung, welche dieser Möglichkeiten in welchem Umfang genutzt werden, ist Sache der Aufgabenautoren und bedarf einer sorgfältigen Planung, da die Testergebnisse sonst schlecht nachvollziehbar werden können. Zweitens kann JACK während der Ausführung der Testfälle alle Programmschritte der geprüften Lösung aufzeichnen und dabei vom Testtreiber gesteuert werden. So ist

es möglich, dass in einem Testfall die Initialisierung eines Objektes geprüft wird und dabei alle Schritte aufgezeichnet werden, während in einem zweiten Testfall mehrere dieser Objekte ohne Aufzeichnung der Schritte erzeugt werden und erst anschließend für Operationen auf diesen Objekten die Aufzeichnung beginnt. Die Aufzeichnung gibt immer die jeweilige Codezeile und die Variablenwerte vor Ausführung derselben an. Ein kurzes Beispiel für eine solche Aufzeichnung ist in Abbildung 9.7 zu sehen. Die erste und letzte Spalte liefern Informationen zur Codezeile, während die übrigen Spalten Variablenwerte enthalten. Im Beispiel in der Abbildung gibt es ein Objekt mit der internen ID 160, dessen Feld `Ort` im Laufe des Aufrufs einen neuen Wert zugewiesen bekommt. Bei vielen Aufgaben können diese Aufzeichnungen Studierenden und Tutoren helfen, die Ursache einer fehlerhaften Programmausgabe im Detail nachzuvollziehen. Auch eine automatisierte Analyse der Aufzeichnungen, durch die weiteres Feedback generiert werden kann, ist bereits möglich, wird allerdings derzeit nur experimentell eingesetzt.

Unabhängig von den vom Aufgabenautor gewählten Einstellungen ist JACK immer in der Lage, Exceptions abzufangen und so zu behandeln, dass erstens ein erklärender Kommentar zu den gängigsten Exceptions erzeugt wird und zweitens die Ausführung weiterer Testfälle nicht gestört wird. Dasselbe gilt für das Auftreten von Endlosschleifen, die JACK nach einem Timeout pro Testfall beendet und automatisch mit dem nächsten Testfall fortfährt.

Ein Beispiel für einen Testtreiber ist in Abbildung 9.8 angegeben. Die Abbildung zeigt einen Ausschnitt von zwei Testmethoden aus einem größeren Testtreiber. Der erste Test führt einen Konstruktoraufruf auf der vom Lernenden zu implementierenden Klasse `Telefonbuch` durch und überprüft das Ergebnis durch zwei Vergleiche. In beiden Fällen wird bei einem Fehler eine Meldung ausgegeben und es werden keine Punkte gutgeschrieben. Ansonsten werden vier Punkte vergeben. Im zweiten Testfall wird zunächst überprüft, ob der erste Testfall erfolgreich war. Ist dies nicht der Fall, wird der zweite Testfall übersprungen und eine entsprechende Warnung ausgegeben. Anderenfalls wird als erstes die Aufzeichnung der Programmschritte ausgeschaltet, um denselben Aufruf wie im ersten Test durchzuführen. Anschließend wird die Aufzeichnung wieder eingeschaltet und es wird

Aufruf des Konstruktors der Klasse Telefonbuch durch JACK

Klasse und Zeile	Variablenwerte				Nächste ausgeführte Codezeile
	160.KopfBranche	160.KopfEintrag	160.Ort	Ort	
Telefonbuch:6	**null**	**null**	**null**	"Syke"	public Telefonbuch(String Ort){
Telefonbuch:7	**null**	**null**	**null**	"Syke"	this.Ort = Ort;
Telefonbuch:8	**null**	**null**	**"Syke"**	"Syke"	}

Abbildung 9.7: Beispielhafte Ausgabe der aufgezeichneten Programmschritte bei einem einfachen Konstruktoraufruf.

```
public class DemoProjekt2Dynamisch {

  private int[] punkte = new int[18];

  @Test(name = "Test 1")
  public void test1() {
    Telefonbuch t1 = new Telefonbuch("Syke");

    if (t1.ort() == null) {
      TracingFramework.printError(/* . . . */);
    } else if (!(t1.ort().equals("Syke"))) {
      TracingFramework.printError(/* . . . */);
    } else punkte[0] = 4;
  }

  @Test(name = "Test 2")
  public void test2() {
   if (punkte[0] == 0)
      TracingFramework.printWarning(/* . . . */);
   else {
      TracingFramework.switchOffTracing();
      Telefonbuch t1 = new Telefonbuch("Syke");
      TracingFramework.switchOnTracing();

      t1.neu("Mueller",   7978);

      if (t1.ersterEintrag() == null){
        TracingFramework.printError(/* . . . */);
      } else if (!t1.ersterEintrag.name().equals("Mueller")){
        TracingFramework.printError(/* . . . */);
        punkte[1] = 1;
      } else if (t1.ersterEintrag.nummer() != 7978){
        TracingFramework.printError(/* . . . */);
        punkte[1] = 1;
      } else punkte[1] = 3;
    }
  }

  // . . . (weitere Testmethoden)

  public int getResult() {
    // . . . (Punkte summieren und zurückgeben)
  }
}
```

Abbildung 9.8: Ausschnitt aus einem Testtreiber für den dynamischen Test einer Klasse. Die Inhalte der über `TracingFramework.printError()` ausgegebenen Fehlermeldungen sind zur besseren Lesbarkeit ausgeblendet.

eine Methoden aufgerufen. Auch hier erfolgen zur Kontrolle mehrere Vergleiche, wobei diesmal in einigen Fällen Teilpunkte vergeben werden. Alle Punkte wer-

den in ein Array geschrieben und am Ende über die Methode `getResult()` aufsummiert und zurückgegeben.

9.3.3 Testfallbasierte Programmvisualisierung

Gerade in der objektorientierten Programmierung ist ein rein textuelles Feedback in Form von Fehlermeldungen und aufgezeichneten Programmschritten schwierig zu verstehen, wenn Lösungen und die von ihnen erzeugten Datenstrukturen größer werden. JACK bietet daher zusätzlich die Möglichkeit an, grafisches Feedback in Form von Objektdiagrammen zu erzeugen. Dazu muss vom Aufgabenautor ähnlich zu den dynamischen Tests ein Testfall definiert werden, für den die Visualisierung erzeugt werden soll. In einer Konfigurationsdatei können dann Zeilennummern aus diesem Testtreiber angegeben werden, in die ein Breakpoint gesetzt wird, um einen Snapshot der Datenstruktur zu erzeugen. Es ist zwar technisch genauso möglich, die Breakpoints direkt in der eingereichten Lösung zu setzen, aber da deren Struktur zum Zeitpunkt der Konfiguration der Aufgabe unbekannt ist, sind feste Zeilennummern in diesem Zusammenhang nicht sinnvoll. Bei der Erzeugung des Snapshots berücksichtigt JACK auch Objekte, die im Programmverlauf vor dem Breakpoint erzeugt und später bereits wieder gelöscht wurden. Diese Objekte werden vom Java Garbage Collector aus dem Speicher entfernt, aber von JACK im Objektdiagramm noch mit einem roten Rand angezeigt. Im Beispiel in Abbildung 9.9 sind mehrere solcher Objekte enthalten. In Verbindung mit dem zur Visualisierung gehörenden Testfall (hier nicht gezeigt) lässt sich erkennen, dass offenbar eine Löschoperation fehlerhaft implementiert wurde, da es mehrere „hängende“ Objekte gab, die vom Garbage Collector entfernt wurden, während die verbleibenden Einträge zyklisch aufeinander verweisen.

9.3.4 Weitere Feedbackmöglichkeiten

Basierend auf der Aufzeichnung aller Programmschritte während der Ausführung der Testfälle ist JACK in der Lage, die Abdeckung des studentischen Codes durch diese Testfälle zu berechnen bzw. im Code grafisch darzustellen. Auf diese Weise können Programmteile hervorgehoben werden, die in keinem der Testfälle erreicht werden. Die Interpretation dieser Information erfordert allerdings ein fortgeschrittenes Verständnis von Testverfahren von den Studierenden, da nicht erreichte Programmzeilen sowohl auf fehlerhafte Programmierung („dead code“) als auch auf unzureichende Testfälle zurückzuführen sein können. Zudem kann

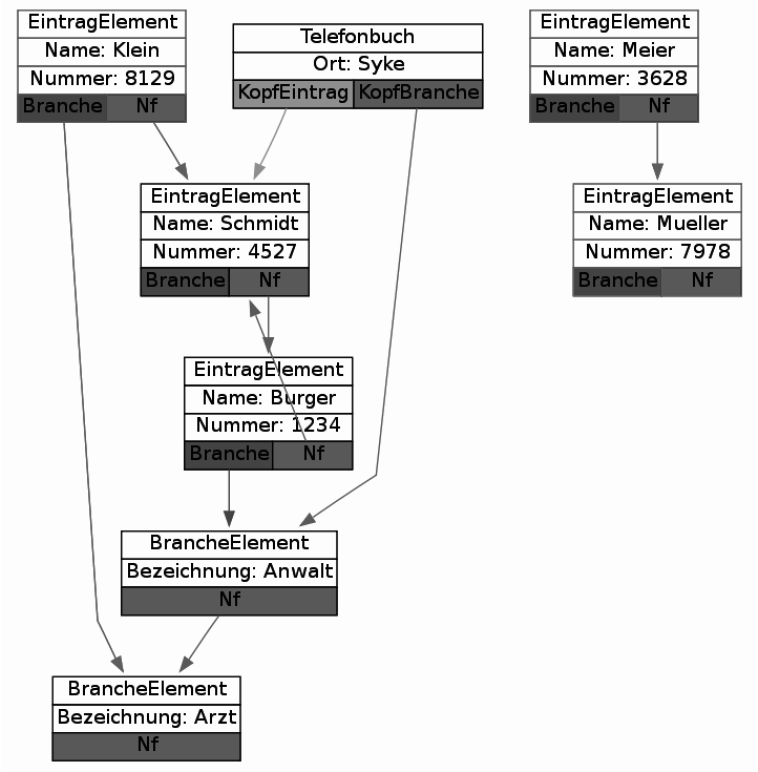


Abbildung 9.9: Beispielhafte Ausgabe eines Objektdiagramms. Objekte, die zum Zeitpunkt der Erzeugung des Snapshots bereits vom Garbage Collector entfernt worden sind, werden mit einem roten Rand dargestellt.

unerreichter Code grundsätzlich überflüssig sein, oder aufgrund eines bestimmten Fehlers nicht erreicht werden und damit ursächlich für einen fehlgeschlagenen Testfall sein. Der Grad der Testabdeckung wird daher auch nicht zur Berechnung der Punktzahl herangezogen, sondern als reine Zusatzinformation verwendet.

Als weitere Feedbackmöglichkeiten, die nur den Aufgabenautoren angezeigt werden, kann JACK auch Softwaremetriken berechnen und Aussagen zur Laufzeit einer Lösung machen. Grundsätzlich könnten diese Informationen zwar auch den Lernenden zur Verfügung gestellt werden, doch bietet JACK derzeit keine Möglichkeit, diese Informationen sinnvoll in einen Kontext einzubetten und zu interpretieren. Metriken wie die zyklomatische Komplexität einer Lösung sind daher momentan eher für Lehrende interessant, um einzelne auffällige Lösungen schneller identifizieren zu können, oder generelle Aussagen zum Vergleich von

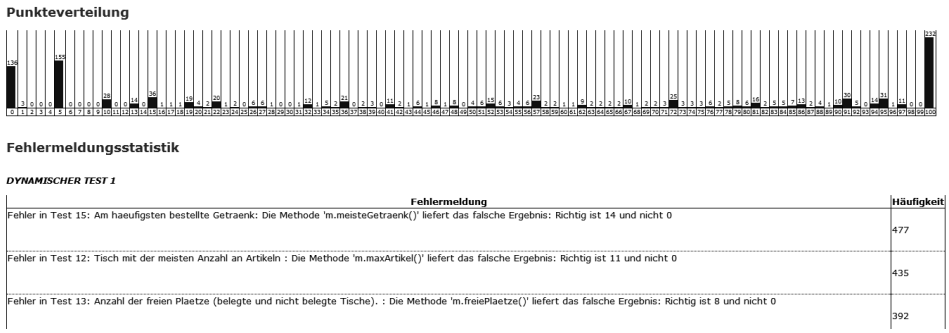


Abbildung 9.10: Auszug aus der Fehlerstatistik einer Aufgabe. Es ist zu erkennen, dass nahezu alle möglichen Punktzahlen von 0 bis 100 Punkten erreicht werden und das Bewertungsschema dieser Aufgabe somit offenbar sehr feingranular ist. Ferner werden unter dem Diagramm die am häufigsten fehlgeschlagenen Testfälle aufgeführt.

Aufgaben untereinander zu gewinnen [SG13]. Als zusammenfassendes Feedback für Lehrende bietet JACK zudem einige Statistiken an, die Auskunft über die erreichten Punktzahlen sowie die häufigsten Fehler geben. Auf diese Weise kann etwa die Granularität des Bewertungsschemas überprüft werden, um die Effektivität gegebenenfalls steigern zu können [Fal+14].

9.4 Technischer Hintergrund

JACK basiert auf einer Master-Worker-Architektur, in der der Master die Entgegennahme, Bereitstellung und Verwaltung von Aufgaben und Lösungen übernimmt, während die Worker die Bewertung und Erzeugung von Feedback durchführen. Der Master ist als EJB-Anwendung auf einem JBOSS Application Server realisiert und verwendet eine PostGreSQL-Datenbank zur Datenhaltung. Die Worker sind eine OSGi-Anwendung, in die die einzelnen Checker-Module als Plugins eingeführt werden können. Es ist insbesondere nicht notwendig, dass alle Worker-Instanzen mit denselben Plugins ausgestattet werden, so dass stattdessen dedizierte Worker für bestimmte Aufgaben konfiguriert werden können, die auf einer entsprechend angepassten Hardware laufen.

Master und Worker kommunizieren miteinander über Webservices und eine vom Master verwaltete Queue für Jobs. Zu jeder eingereichten Lösung wird für jedes in der Aufgabe konfigurierte Checker-Modul ein Job erzeugt, der von einem Worker abgeholt und bearbeitet werden kann. Die Kommunikation wird dabei

ausschließlich von den Workern initialisiert, die Jobs aus der Queue entnehmen bzw. Ergebnisse zurückmelden. Die Worker können daher aus Sicherheitsgründen in Netzwerksegmente abgelegt werden, die von außen nicht erreichbar sind.

Die Architektur hat sich in den vergangenen Jahren als sehr leistungsfähig erwiesen und ist insbesondere gut skalierbar, um beispielsweise durch das Hinzufügen weiterer Backend-Instanzen auf hohe Last zu reagieren. Das Konzept ist dabei nicht auf Programmieraufgaben beschränkt, sondern für alle Aufgabentypen in JACK nützlich [Str16].

Auch wenn JACK vollständig in Java entwickelt ist, sind Programmieraufgaben nicht grundsätzlich auf diese Sprache beschränkt. Über die Einbindung geeigneter externer Werkzeuge in den Checker-Modulen können auch weitere Programmiersprachen geprüft werden. Als prototypische Module, die im Rahmen studentischer Arbeiten entstanden sind, existieren derzeit Erweiterungen für Python [Loh15] und die .NET-Sprachfamilie [D'A15]. Eine Erweiterung für C++ wurde an der Universität Heidelberg entwickelt [HWP13].

9.5 Einsatzerfahrung

An der Universität Duisburg-Essen kommt JACK am Campus Essen seit dem Wintersemester 2006/2007 zur Bewertung von Programmieraufgaben in Java zum Einsatz. Bis einschließlich Wintersemester 2015/16 wurden von dem System gut 150.000 Lösungen zu Übungs- und Testataufgaben bewertet. Es existiert ein Aufgabenpool von etwa 70 Aufgaben, die zum Teil als Prüfungsaufgaben in mehreren Varianten vorliegen.

Der Einsatz wurde in mehreren Semestern durch eine umfangreiche Evaluation begleitet, um Einschätzungen der Studierenden über den Nutzen des Systems zu erhalten. Die wichtigste Frage dabei ist, ob JACK von den Studierenden als hilfreich für Übungen oder Testate betrachtet wird. Die Ergebnisse der Befragung sind in Tabelle 9.1 zusammengefasst. Es ist ersichtlich, dass JACK von den Studierenden mit 75% bzw. 68% vollständiger oder überwiegender Zustimmung als nützlich im Übungs- und Testatbetrieb betrachtet wird. Diese grundsätzliche Einstellung spiegelt sich auch tatsächlich im Nutzerverhalten wieder, indem im Übungsbetrieb mehrere Einreichungen durchgeführt werden, um schrittweise die durch das Feedback benannten Fehler auszubessern [SG11b].

Spezielle statistische Auswertungen für die verschiedenen Feedbackmöglichkeiten wurden in einzelnen Semestern durchgeführt. So waren im Wintersemester 2012/13 55% aller Feedbacknachrichten auf fehlgeschlagene Testfälle zurückzuführen, 23% entfielen auf Compilerfehler, 13% auf Exceptions, 6% auf stilistische

		++	+	o	-	- -	#Antworten
Einstellung bzgl. der Aussage „JACK ist im *Übungsmodus* insgesamt betrachtet nützlich.“	2008/2009	34%	44%	15%	7%	0%	61
	2009/2010	53%	29%	9%	5%	4%	77
	2010/2011	26%	43%	25%	5%	2%	61
	2011/2012	35%	27%	27%	10%	2%	63
	2012/2013	Keine Daten verfügbar					
	2013/2014	49%	33%	13%	5%	0%	61
	Schnitt	**40%**	**35%**	**17%**	**6%**	**2%**	
Einstellung bzgl. der Aussage „JACK ist im *Testatbetrieb* insgesamt betrachtet nützlich.“	2008/2009	30%	36%	23%	8%	3%	61
	2009/2010	48%	16%	12%	13%	11%	77
	2010/2011	34%	38%	16%	10%	2%	61
	2011/2012	30%	35%	19%	10%	6%	63
	2012/2013	Keine Daten verfügbar					
	2013/2014	36%	38%	13%	11%	4%	56
	Schnitt	**36%**	**32%**	**16%**	**11%**	**6%**	

Tabelle 9.1: Ergebnisse mehrerer Befragungen der Studierenden zu ihrer grundsätzlichen Einstellung zu JACK in den beiden hauptsächlichen Einsatzszenarien. Es steht ++ für die volle Zustimmung zur Aussage und - - für die vollständige Ablehnung.

Fehler, 2% auf strukturelle Fehler und 1% auf Timeouts für mutmaßliche Endlosschleifen. Die Kombination verschiedener Feedbackmöglichkeiten ist also auch tatsächlich effektiv bei der Erstellung vielfältiger Rückmeldungen.

9.6 Zusammenfassung

Nach inzwischen 10 Jahren Einsatzzeit zeigt sich JACK als stabiles System mit vielfältigen Feedbackmöglichkeiten, das von den Studierenden überwiegend als hilfreich erachtet wird. Der Aufwand für die Lehrenden bei der Erstellung und Konfiguration neuer Aufgaben liegt im Durchschnitt höher als bei anderen Systemen, aber dafür sind auch vielfältigere und detailliertere Rückmeldungen möglich. Obwohl ursprünglich nur für Programmieraufgaben in Java entwickelt, ist die Architektur von JACK so generisch, dass Checker-Module für weitere Programmiersprachen einfach ergänzt werden können.

Literatur für dieses Kapitel

[D'A15] Mario D'Amico. „Konzeption universeller .NET-Prüfkomponenten für das E-Assessment-System JACK“. In: *Workshop „Automatische*

Bewertung von Programmieraufgaben“ (ABP 2015). Bd. 1496. CEUR Workshop Proceedings. 2015.

[Fal+14] Nickolas Falkner u. a. „Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units“. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE ’14. ACM, 2014, S. 9–14. DOI: 10.1145/2538862.2538896.

[HWP13] Tom-Michael Hesse, Axel Wagner und Barbara Paech. „Automated assessment of C++ programming exercises with unit tests“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. 2013.

[Loh15] Enno Lohmann. „Erweiterung eines E-Assessment-Systems um eine Prüfkomponente für die Programmiersprache Python“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2015)*. Bd. 1496. CEUR Workshop Proceedings. 2015.

[SG11b] Michael Striewe und Michael Goedicke. „Studentische Interaktion mit automatischen Prüfungssystemen“. In: *DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik*. Bd. 188. LNI. GI, 2011, S. 209–220.

[SG13] Michael Striewe und Michael Goedicke. „Analyse von Programmieraufgaben durch Softwareproduktmetriken“. In: *SEUH*. 2013, S. 59–68.

[SGB08] Michael Striewe, Michael Goedicke und Moritz Balz. *Computer Aided Assessments and Programming Exercises with JACK*. Techn. Ber. 28. ICB, University of Duisburg-Essen, 2008.

[Str16] Michael Striewe. „An architecture for modular grading and feedback generation for complex exercises“. In: *Science of Computer Programming* 129 (2016), S. 35–47. DOI: 10.1016/j.scico.2016.02.009.

[SZG15] Michael Striewe, Björn Zurmaar und Michael Goedicke. „Evolution of the E-Assessment Framework JACK“. In: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. März 2015.* Bd. 1337. CEUR Workshop Proceedings. 2015, S. 118–120.

10 Der Grader Praktomat

Joachim Breitner, Martin Hecker und Gregor Snelting

Zusammenfassung
Der Praktomat *[Gif+16] ist ein webbasierter Dienst, auf dem Studierende Programmieraufgaben abrufen und ihre Lösungen einreichen können. Diese werden vollautomatisch durch eine große Zahl von Funktionstests, sowie weitere Tester wie Stilchecker geprüft. Die so erhaltene Rohbewertung der Einreichungen wird durch wissenschaftliche Mitarbeiter finalisiert, wobei zusätzlich softwaretechnische Kriterien herangezogen werden. Der Praktomat wird seit vielen Jahren an diversen Universitäten in sehr großen Programmierkursen eingesetzt. Er wird zusammen mit dem Plagiatserkenner JPlag betrieben und sorgt nachweislich für eine deutlich verbesserte Programmierkompetenz der Teilnehmer.*

10.1 Geschichte und Überblick

Seine Ursprünge hat der Praktomat an der Universität Passau, als im Jahre 1999 die Abgabe der Programmieraufgaben in der Erstsemesterveranstaltung von Prof. Snelting automatisiert wurde. Die erste Version, von Sneltings erstem Doktoranden Andreas Zeller entwickelt, war noch ein relativ loser Verband von Shell- und Perl-Skripten. Trotz der anfänglichen technologischen Schwächen sorgte das System für eine signifikante Verbesserung des Übungsbetriebs und führte zu einer nachweislich stark verbesserten Programmierkompetenz der Teilnehmer. Praktomat wurde im Laufe der Jahre von Andreas Zeller, Jens Krinke und anderen weiter ausgebaut [KSZ02].

Im Jahre 2009 stellten Dennis Giffhorn und Daniel Kleinert den Praktomaten für die Erstsemestervorlesung „Programmieren“ von Prof. Snelting, inzwischen am KIT, auf die heutige technische Basis: Der Praktomat ist in Python implementiert und basiert auf dem Web-Framework Django. Seit dem wurde der Praktomat von weiteren Mitarbeitern von Prof. Snelting weiterentwickelt, insbesondere Martin Hecker und Joachim Breitner.

Der Praktomat wird heute in diversen Veranstaltungen verschiedener Universitäten verwendet. Insbesondere wird am KIT die Veranstaltung „Programmieren“ für Informatikerstsemester mit jährlich ca. 800 – 900 Teilnehmern mit dem Praktomaten abgewickelt.

Die Programmierkompetenz der Teilnehmer verbessert sich nachweislich vor allem durch folgende Faktoren:

- Die Aufgabenstellungen müssen – da die eingereichten Programme vollautomatisch getestet werden – sehr genau spezifiziert sein, bis in kleinste Details der Ein-/Ausgabe. Dies führt zu einer intensiven Beschäftigung mit der Aufgabe und zu einer großen Sorgfalt der Teilnehmer.

- Das automatische Testen anhand ca. 50 – 200 geheimer Testfälle pro Aufgabe, in Kombination mit automatischem Check des Java Style Guides (`checkstyle`, [Che]), führt zu einer belastbaren, *objektiven*, präzisen, und anonymisierten Bewertung der Programmierkompetenz.

- Die abschließende Bewertung softwaretechnischer Kriterien durch wissenschaftliche Mitarbeiter erzwingt schon im ersten Semester eine gute (objektorientierte) Programmarchitektur.

- Praktomat wird immer in Kombination mit dem am KIT entwickelten Plagiatsprüfer JPlag eingesetzt (der laut eines Vergleichstests der international beste Java-Plagiatserkenner ist, siehe [WW12]); so werden Teilnehmer gezwungen, wirklich selbst zu programmieren.

Diese Vorteile erlaubten es, auf eine Abschlussklausur in der Vorlesung „Programmieren“ zu verzichten und stattdessen zwei anspruchsvolle Abschlussaufgaben (mit je ca. 500 – 1500 LOC) als Prüfungsleistungen festzulegen; dies ist didaktisch als wesentlich effektiver anzusehen („*learning by doing*“) und verstärkt den kompetenzfördernden Effekt von Praktomat.

10.2 Abläufe und Besonderheiten

Mit dem Praktomaten interagiert man meist in einer von vier Rollen:

- als Administrator des Systems, auf dem der Praktomat läuft;

- als Übungsleiter, der die Aufgaben erstellt und die letzendliche Verantwortung für die Benotung der Studenten trägt;

- natürlich als Student, der seine Lösung einreicht;

- als (meist studentischer) Tutor, der die Lösungen sichtet, kommentiert und bewertet.

Bei den großen Programmiervorlesungen sind diese Rollen disjunkt, während bei kleineren Vorlesungen der Übungsleiter oft auch die Rolle des Tutors übernimmt.

Im Folgenden beschreiben wir die Abläufe und Besonderheiten des Praktomaten aus der jeweiligen Rollensicht.

10.2.1 ... aus Sicht des Administrators

Der Praktomat ist, technisch gesehen, eine weitgehend herkömmliche Django-Anwendung und läuft auf einem üblichen Linux-basierten System. In Karlsruhe wird Debian `stable` eingesetzt. Der Quellcode, der unter den Bedingungen der GPL-Lizenz Version 2 verteilt wird, ist über Github[1] zu beziehen; die enthaltene README-Datei beschreibt den Installationsprozess.

Es wird eine Datenbank vorausgesetzt. Prinzipiell unterstützt der Praktomat – dank Django – dabei eine Reihe von verschiedenen Systemen. In der Praxis getestet ist dabei allerdings nur Postgresql und SQLite, wobei letzteres aus Performance-Gründen nur für Test- und Entwicklungsinstanzen geeignet ist. Weiter wird als Webserver Apache eingesetzt.

Für die Verwaltung der Benutzer (Übungsleiter, Tutoren, Studenten) kann die Django-eigene Benutzerverwaltung verwendet werden, mit der üblichen Registrierung per Bestätigungs-E-Mail und Authentifizierung per Benutzername und Passwort. Komfortabler ist allerdings die Integration in ein universtätsweites Single-Sign-On-System. Hier unterstützt der Praktomat Shibboleth, mit Hilfe des Apache-Moduls `mod-shib2`.

Aus Sicht des Praktomaten gibt es nur eine Vorlesung; er ist nicht mandatenfähig. Um trotzdem auf einem Server die Praktomat-Instanzen für verschiedene Vorlesungen laufen zu lassen, können mehrere Praktomat-Instanzen, jeweils mit eigener Datenbank, in verschiedenen Verzeichnissen installiert werden und so völlig unabhängig voneinander betrieben werden.

Für das Testen des nicht vertrauenswürdigen studentischen Codes legt der Administrator entweder einen dedizierten Benutzer oder – besser – einen dedizierten Container in der Containerlösung `docker` an. Mehr zur Sicherheitsarchitektur in Abschnitt 10.3.

Der Administrator legt fest, wie viele Prozesse pro Praktomat-Instanz die Benutzeranfragen bearbeiten. Da Einreichungen synchron getestet werden, also die HTTP-Abfrage entsprechend lange läuft, sorgen zu wenige Prozesse dafür, dass

1 https://github.com/KITPraktomatTeam/Praktomat

der Praktomat nicht erreichbar ist, wenn alle Prozesse ausgelastet sind; daher sollte diese Zahl nicht zu klein gewählt werden.

Eine Skalierung in die Breite, also über mehrere Server, wird nicht unterstützt. Da sowohl das Webinterface als auch die Prüfung der Abgaben auf dem gleichen System läuft, sollte dieses adäquat ausgestattet sein.

In Karlsruhe kommt bei 10 Prozessen pro Instanz ein virtuelles System mit 4GB RAM zum Einsatz.

10.2.2 ... aus Sicht des Übungsleiters

Der Übungsleiter, auch *trainer* im Praktomat-Jargon, ist der für den Übungsbetrieb verantwortliche Dozent oder Mitarbeiter. Er ist der einzige, der mit dem Praktomaten auch über die sogenannte „Admin-Oberfläche“ interagiert.

Dort legt er Tutorien (Übungsgruppen) an, ernennt Benutzer zu Tutoren, und weist den Studenten diese Gruppen zu.

Weiter legt er hier die Aufgaben (*tasks* im Praktomat-Jargon) an. Diese bestehen aus einem Titel, einer Aufgabenbeschreibung aus formatierbarem Text, eventuellen Anhängen und einer Reihe von *Checkern*. Diese Checker sind das Herz des Praktomaten, und stellen abstrakt einen Schritt in der automatischen Prüfung der studentischen Abgabe dar. Manche Checker stellen Funktionalität bezüglich einer bestimmten Programmiersprache bereit (siehe Tabelle 10.1), andere sind generisch. Die wichtigsten Checker sind:

- Ein Java-Checker, der Java-Code in der studentischen Abgabe sucht und kompiliert. Analoge Checker existieren für C, C++, Fortran, Haskell und Isabelle.

- Stil-Checker, die mit Hilfe von `checkstyle` gewisse, vom Übungsleiter vorgegebene Stilvorgaben prüfen.

Sprache	**Kompilation**	**Unit-Tests**	**Stilprüfung**
Java	✓	✓	✓
C/C++/Fortran	✓		
Haskell	✓	✓	
R	✓		
Isabelle	✓		

Tabelle 10.1: Programmiersprachenspezifische Features

- Unit-Test-Checker, die vom Übungsleiter vorgegebene JUnit- oder DejaGnu-Tests gegen den studentischen Code laufen lassen. Haskell-Abgaben werden durch HUnit- und QuickCheck-Tests überprüft.

- Für die Sprache R existiert ein spezieller Checker, der ein bei der Ausführung des R-Codes erzeugtes PDF-Dokument abspeichert, damit es später bei der Begutachtung der Einreichungen betrachtet werden kann.

- Ein generischer Skript-Checker, der es dem Übungsleiter erlaubt, beliebigen Code (etwa Python- oder Bash-Skripte) im Kontext der studentischen Abgabe laufen zu lassen. Dieser Checker bietet dem Übungsleiter maximale Freiheiten, und einige Übungsleiter nutzen diesen, um mit selbst entwickelten Frameworks aufwendige Akzeptanztests gegen das eingereichte Programm laufen zu lassen.

Jeder Checker produziert einen Ausgabetext und einen Status, der den Erfolg des Checkers angibt.

Der Ausgabetext kann dabei entweder reiner Text sein (etwa die Statusmeldungen des Compilers) oder HTML-Fragmente. Letzteres erlaubt es insbesondere dem Skript eines Skript-Checkers, ansprechende und interaktive dynamische Ausgaben zu erstellen.

Der Übungsleiter markiert jeden Checker als zwingend nötig oder optional sowie als privat oder öffentlich. Meldet ein zwingend nötiger Checker einen Fehlschlag, so werden keine weiteren Checker ausgeführt und die Einreichung wird gar nicht erst angenommen. So kann erreicht werden, das nicht kompilierender Code nicht bewertet wird. Die Ergebnisse von privaten Checkern werden dem Studenten nicht angezeigt, sondern sind nur dem Übungsleiter und dem Tutor sichtbar.

Über die Funktion, Testabgaben hochzuladen, kann der Übungsleiter die Konfiguration der Aufgabe überprüfen.

Die Bewertungsskala für die Aufgabe (z. B. bestanden/nicht bestanden, Schulnoten, Zahlenbereiche etc.) ist auch konfigurierbar. Auch mehrere Skalen (z. B. Funktionalität, Stil) können angelegt werden.

Zuletzt legt der Übungsleiter fest, ab wann die Aufgabe für die Studenten sichtbar ist und bis wann Einreichungen angenommen werden sollen.

Nach dem Ende der Einreichungsfrist stößt der Übungsleiter einmal die erneute Prüfung aller finalen Abgaben an, bevor die Tutoren die Einreichungen zu sehen bekommen. So wird sichergestellt, dass auch bei späteren Verbesserungen der Checker alle Bewertungen mit denselben, finalen Versionen der Checker bewertet werden.

Auch kann der Übungsleiter nun mit einem Klick alle Abgaben von der Plagiatserkennungssoftware `jPlag` [Jpl] prüfen lassen. Der dabei produzierte Bericht listet auf, welche Abgaben sich verdächtig stark ähneln. Dabei lässt sich `jPlag` bei unterstützten Programmiersprachen (Java, C#, C, C++, Scheme) nicht von naiven Verschleierungstechniken wie dem Umbenennen von Variablen täuschen.
Nachdem die Tutoren die Aufgaben bewertet haben, und sofern die Konfiguration vorgibt, dass die Tutoren das nicht selber machen können, gibt der Übungsleiter die Bewertungen frei. Die Studenten erhalten eine Nachricht per E-Mail und können nun die Bewertung sehen.

Zum Abschluss des Semesters kann der Übungsleiter aus den Bewertungen der einzelnen Übungsblätter die Gesamtnote berechnen lassen. Das Schema, nachdem die Einzelbewertungen verrechnet werden, gibt er dabei als JavaScript-Funktion an; so sind hier große Freiheiten möglich.

10.2.3 ... aus Sicht des Studenten

Für die Studenten beginnt die Interaktion mit dem Praktomaten dann, wenn die vom Übungsleiter erstellten Aufgaben veröffentlicht werden. Von nun an können sie ihre Lösungen hochladen. Bei Abgaben, die aus mehreren Dateien bestehen, können diese entweder einzeln hochgeladen werden, oder als Archivdatei, die vom Praktomaten dann automatisch entpackt wird.

Die vom Übungsleiter wie oben beschriebenen Checker werden unmittelbar auf die Lösung des Studenten angewandt. Dabei werden dem Studenten die Ausgaben der als öffentlich markierten Checker angezeigt, so dass er seine Abgabe entsprechend verbessern und erneut einreichen kann. Schlägt dabei ein zwingend nötiger Checker fehl, so wird die Einreichung abgelehnt, ansonsten angenommen. Der Student hat eine Übersicht über seine bisherigen Einreichungen und sieht, welche angenommen wurden und welche derzeit die *finale* Abgabe ist. In der Regel ist das die jüngste, bei der keine zwingenden Tests fehlgeschlagen sind, aber er kann auch manuell eine ältere als final markieren.

Bis zur Deadline kann der Student beliebig oft eine neue Lösung eintragen.

Sobald sein Tutor die Abgabe wie im nächsten Abschnitt beschrieben bewertet hat, bekommt der Student per E-Mail eine Benachrichtigung und kann die Bewertung lesen.

Abbildung 10.1: Studentische Abgabe mit Testresultaten

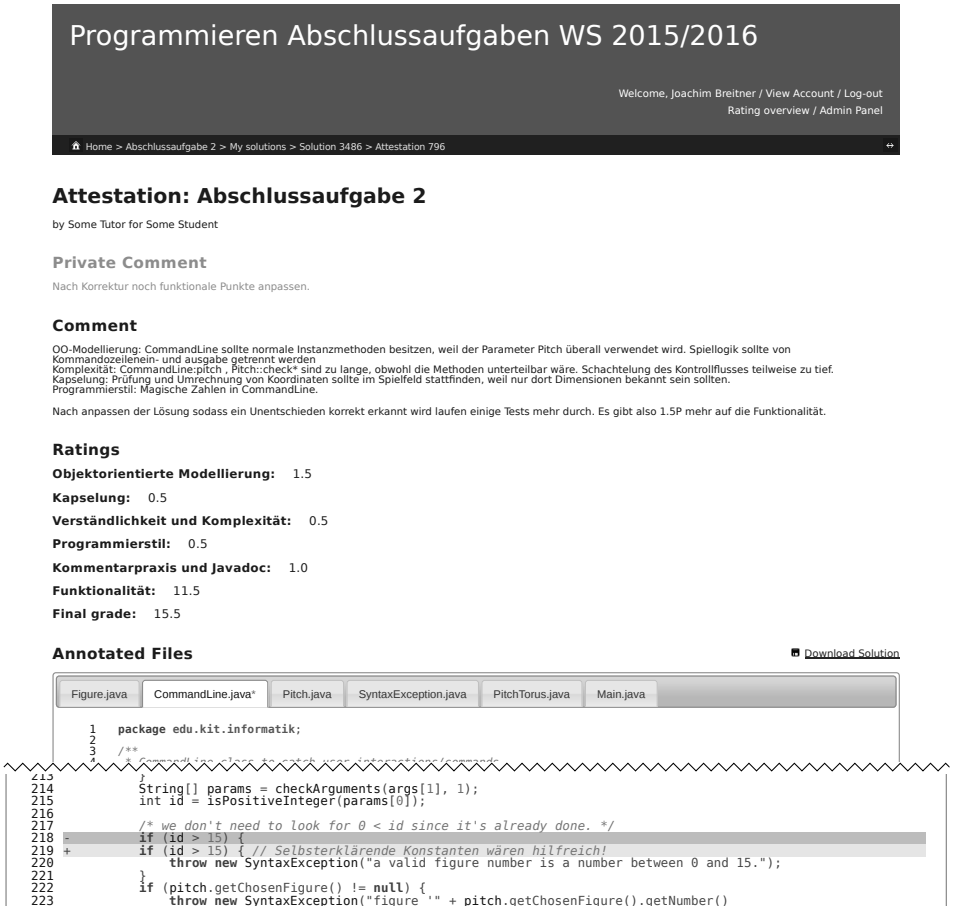


Abbildung 10.2: Begutachtung mit Kommentaren im Quelltext

10.2.4 . . . aus Sicht des Tutors

Die Aufgabe eines Tutors ist es, die Abgaben der Studenten in seiner Übungsgruppe zu begutachten. Dabei können einer Übungsgruppe auch mehrere Tutoren zugewiesen werden, die sich die Arbeit teilen. Der Tutor sieht die Ausgabe aller Checker, auch der als privat markierten, sowie den Quellcode des Studenten, mit ensprechendem Syntax-Highlighting (Abbildung 10.1). Die Quelldateien darf er bearbeiten, etwa um bestimme Codezeilen zu kommentieren oder Verbesserungsvorschläge zu machen. Dem Studenten werden diese Änderung mit entsprechend deutlicher farblicher Hervorhebung als *diff* angezeigt (Abbildung 10.2).

Zusätzlich hat der Tutor die Möglichkeit, die Begutachtung (*attestation* im Praktomat-Jargon) mit einem für den Studenten sichtbaren Kommentar sowie mit einer privaten, nur für den Übungsleiter sichtbaren Bemerkung versehen. Letztere kann genutzt werden, um diesen auf mögliche Betrugsfälle hinzuweisen. Die Bewertung oder – bei mehreren Skalen – Bewertungen legt der Tutor ebenfalls fest. Sobald der Tutor zufrieden ist, markiert er die Bewertung als final; finale Bewertungen werden je nach Konfiguration entweder vom Übungsleiter oder vom Tutor selbst veröffentlicht. Nach der Veröffentlichung kann der Student den Kommentar, die Bewertung und die Änderungen am Quellcode einsehen. Der Tutor kann nun keine Änderungen mehr vornehmen. Eine bereits dem Studenten veröffentlichte Begutachtung kann nur vom Übungsleiter zurückgezogen werden.

Selbstverständlich sehen Studierende nur ihre eigenen Abgaben und Bewertungen, und Tutoren nur die aus der ihnen zugewiesenen Übungsgruppe.

10.3 Sicherheitskonzept

Der Praktomat führt notwendigerweise von den Studenten eingereichten und damit nicht vertrauenswürdigen Programmcode aus. Es muss also dafür gesorgt sein, dass auch ein böswilliger Student nicht Daten lesen kann, die er nicht lesen darf, und erst recht nicht die Kontrolle über den Server übernehmen kann.

Der Haupteinsatz von Praktomat ist in Vorlesungen, in denen Java gelehrt wird. Hier wird der Java Security Manager so konfiguriert, dass nur Dateien im aktuellen Verzeichnis gelesen und geschrieben werden dürfen, und kein Netzwerkzugriff möglich ist.

Inzwischen wird der Praktomat allerdings auch mit anderen Programmiersprachen (z. B. R, Isabelle) verwendet, die keine solchen eingebauten Sicherheitsmechanismen mitbringen. Hier wird zusätzlich die Ausführung jedes potenziell gefährlichen Befehls, was neben der Ausführung auch die Kompilation des Benutzercodes einschließt, in einen für diesen Zweck gestarteten Docker-Container verlegt. Dieses „virtuelle“ System hat sein eigenes, nur schreibbares Dateisystem, sein eigenes temporäres Verzeichnis und keinen Netzwerkzugriff. Lediglich das Verzeichnis mit den eingereichten Dateien wird per *bind-mount* in den Container hineingereicht.

Container bieten hier die nötige Kapselung und zusätzlich eine komfortable Beschränkung des dem Programm zur Verfügung stehenden Hauptspeichers und der Laufzeit. Diese Funktionalität ist als `safe-docker` [Bre16] auch unabhängig vom Praktomaten verfügbar.

Semester	Studenten	Tutoren	Tasks	Abgaben	$\sum$ LOC	$\oslash$ LOC
Algorithmen 1 (Java)						
SS 2014	267	26	5	391	101 417	259
Computational Risk and Asset Management (R, Python)						
WS 2014	34	3	12	256	55 427	216
WS 2015	21	2	13	173	34 949	202
Programmieren (Java)						
SS 2011	90	4	11	380	108 784	286
WS 2011	848	29	10	3 455	5 101 898	1476
SS 2012	203	8	12	807	235 509	291
WS 2012	864	37	11	4 959	1 781 767	359
SS 2013	267	7	7	378	98 041	259
WS 2013	724	27	9	3 623	1 198 904	330
SS 2014	238	5	9	798	204 685	256
WS 2014	798	30	20	7 899	1 377 868	174
SS 2015	212	10	12	1 212	236 248	194
WS 2015	933	29	22	12 210	1 495 461	122
SS 2016	192	8	9	803	200 314	249
Programmieren: Abschlussaufgaben (Java)						
SS 2011	105	6	2	84	152 027	1809
WS 2011	855	30	2	972	1 538 585	1582
SS 2012	147	4	2	176	188 556	1071
WS 2012	590	23	2	922	1 076 255	1167
SS 2013	187	7	2	154	96 171	624
WS 2013	391	20	2	596	673 590	1130
SS 2014	126	11	2	187	167 701	896
WS 2014	397	24	2	703	949 137	1350
SS 2015	133	8	2	222	189 844	855
WS 2015	453	17	2	797	1 053 362	1321
Theorembeweiser: Anwendungen in der Sprachtechnologie (Isabelle)						
SS 2013	29	1	7	51	20 788	407
SS 2016	18	2	6	53	15 472	291

LOC: lines of code der Abgaben, $\oslash$ pro Task, beinhaltet Leer-, Kommentarzeilen.
Stand der Zahlen für SS 2016: 10.06.2016.

Tabelle 10.2: Einsatz von Praktomat am KIT seit 2011

10.4 Einsatz

Tabelle 10.2 zeigt den Umfang des Praktomat-Einsatzes am KIT in den letzten 5 Jahren. Neben „Programmieren“ wurde Praktomat an der Informatikfakultät eingesetzt in „Algorithmen 1“ (Java) sowie im Praktikum „Theorembeweiser: Anwendung in der Sprachtechnologie“. Im letzteren werden keine Programme eingereicht, sondern mit dem Theorembeweiser Isabelle erstellte formale Beweise, deren Korrektheit dann serverseitig geprüft wurde. Die formellastigen Beweise reizten die Unicode- und Syntax-Highlighting-Fähigkeiten des Praktomaten aus. Auch jenseits der Informatik kommt der Praktomat zum Einsatz: Die Fakultät für Wirtschaftswissenschaften des KIT setzt den Praktomaten in einer Vorlesung zum Risikomanagement ein.

Außerhalb von Karlsruhe sind uns Praktomat-Instanzen an der HS Emden, der HFT Leipzig, dem University College London, der HS Regensburg und der TU Dresden bekannt.

An der HS Ostfalia enstand im Rahmen des eCult-Projektes eine Integration des Praktomaten in das LMS LON-CAPA (siehe Kapitel 19), bei der der Praktomat im Hintergrund über eine REST-Schnittstelle angesprochen wird, während die Studierenden lediglich mit LON-CAPA interagieren [KJ13; MR15].

10.5 Erfahrungen

Allein am KIT haben seit 2008 viele tausend Informatikstudierende ihre Programmieraufgaben über den Praktomaten eingereicht. Zum Schluss dieser Beschreibung von Praktomat möchten wir deshalb einige strategische Auswirkungen des Praktomat-Einsatzes aufzeigen, die nachgerade „politischer“ Natur sind.

Praktomat und PSE. Die Veranstaltung „Praxis der Softwareentwicklung“ (PSE) wurde 2009 an der Informatikfakultät des KIT eingeführt mit dem Ziel, Drittsemestern Kompetenz in professioneller Softwareentwicklung zu vermitteln. PSE ist charakterisiert durch: Arbeit in Teams von 5 Teilnehmern; Entwicklung eines mittelgroßen Projektes durch alle Phasen vom Pflichtenheft bis zur Abschlusspräsentation; objektorientierter Entwurf mit UML; integrierte Qualitätssicherung. Alle Lehrstühle der Fakultät stellen Aufgaben, dabei ist das Prozessmodell vorgegeben, inhaltlich sind die Lehrstühle aber frei. Die meisten Lehrstühle wählen Themen aus ihrem Forschungsbereich (Stichwort „forschungsorientierte Lehre“).

Die Vorgabe für die Systemgröße ist ca. 100 Klassen, ca. 10000 LOC (meist Java, aber auch C#; oft Android-Apps). PSE hat 8 LP, also nominell insgesamt

$5 \times 8 \times 30 = 1200$ Arbeitsstunden – was, verglichen mit typischen Industriekennzahlen, sehr wenig ist für ein Produkt von 10000 LOC. Alle Teams erzielen zumeist glänzende Ergebnisse mit Industriequalität (auch in der GUI), und preisen ihre Projekte in ihrem CV, im Internet und anderswo an.

Möglich ist das nur durch die mittels Praktomat erworbenen Vorkenntnisse in Programmieren. Frühere PSE-ähnliche Veranstaltungen des Lehrstuhl Snelting hatten klassische „manuelle“ Programmierkurse als Vorstufe, so dass bei weitem nicht die Größe und Professionalität von PSE-Projekten erreicht wurde. Entscheidend ist die stringente Qualitätssicherung durch das automatische Testen. Der Lehrstuhl Snelting erhielt für die Einführung von PSE den Fakultätslehrpreis.

Systemakkreditierung. Ein interessanter, „politischer“ Nebeneffekt des Einsatzes von Praktomat ergab sich 2014 im Zuge der Systemakkreditierung des KIT-Informatikstudiengangs. Die Frauenbeauftragte hatte festgestellt, dass sich die durchschnittlichen Bachelorabschlussnoten von Männern und Frauen um ca. 0,25 Notenpunkte unterscheiden, und dafür eine (bewusste oder unbewusste) Diskriminierung von Studentinnen durch die Dozenten verantwortlich gemacht. Da „Programmieren“ die größte Veranstaltung der KIT Informatik ist, wurden Daten aus Praktomat zur Überprüfung dieser Behauptung analysiert. In der Tat schnitten in den Jahre 2008 – 2010 Frauen durchschnittlich ca. 0,3 Notenpunkte schlechter in „Programmieren“ ab. In diesen Jahren wurden Aufgaben strikt anonymisiert bewertet. Es war den Bewertern definitiv unmöglich, das Geschlecht (oder andere Eigenschaften) der Teilnehmer zu erkennen: Praktomat zeigte keinerlei Informationen zu Einreichern an, und Einreichungen, die Hinweise auf den Autor im Quelltext enthielten, wurden zurückgewiesen.

Ab 2011 konnten die Bewerter (auf Wunsch des damaligen Dozenten) die Namen der Einreicher sehen. Daraufhin glichen sich die Durchschnittsnoten von Männern und Frauen in „Programmieren“ an. Da der Schwierigkeitsgrad der Aufgaben und die Bewertungskriterien gleich blieben (insbesondere wird weiterhin vollautomatisch getestet) war gezeigt, dass es entweder ab 2011 eine signifikante Veränderung in der Erstsemesterpopulation gab, oder dass die Betreuer Frauen (bewusst oder unbewusst) *besser* bewerten als Männer. Die Frauenbeauftragte zog daraufhin ihre Behauptung zurück.

Plagiatsversuche. Ein anderes, offenes Problem sind Betrugsversuche. Zwar erkennt JPlag zuverlässig Plagiate, insbesondere Kopien und Varianten von eingereichten Programmen (weil z. B. ein Teilnehmer das Programm abgeschrieben hat); auch geänderte Variablennamen oder THEN-ELSE Vertauschungen nützen nichts. Wenn ein Teilnehmer sich jedoch das Programm extern erstellen lässt, und

es nur einmal eingereicht wird, kann JPlag das nicht erkennen. Nachdem mehrere derartige Fälle aufgeflogen sind – weil externe „Nachhilfelehrer“ Praktomat-spezifische Programmierdienste gegen Bezahlung im Internet anboten – sollen nun die Abschlussaufgaben in „Programmieren“ durch eine Klausur ergänzt werden. Da aber weiterhin zwei Abschlussaufgaben bearbeitet werden müssen, wird dies den kompetenzsteigernden Effekt des Praktomaten nicht beeinträchtigen.

Abschließend lässt sich feststellen: Die Entwicklung und der Einsatz von Praktomat waren und sind ein überwältigender Erfolg. Die Programmierkompetenz wird durch Praktomat stark verbessert – wir arbeiten z.Zt. an einer statistisch belastbaren Studie, die auf Praktomat-Rohdaten beruht, wobei PSE- und Praktomat-Noten korreliert werden. Auch ohne belastbare Statistik zeigen weiterführende Softwareprojekte seit Jahren glänzende Ergebnisse, die nach Überzeugung aller Beteiligten ohne Praktomat nicht möglich wären.

Literatur für dieses Kapitel

[Bre16] Joachim Breitner. *safe-docker*. 2016. URL: https://github.com/nomeata/safe-docker (besucht am 09. 06. 2016).

[Che] *Checkstyle*. 2016. URL: http://checkstyle.sourceforge.net/ (besucht am 09. 06. 2016).

[Gif+16] Dennis Giffhorn u. a. *Praktomat*. 2016. URL: http://pp.info.uni-karlsruhe.de/projects/praktomat/praktomat.php (besucht am 09. 06. 2016).

[Jpl] *jPlag*. 2016. URL: https://jplag.ipd.kit.edu/ (besucht am 09. 06. 2016).

[KJ13] Marcel Kruse und Nils Jensen. „Automatische Bewertung von Datenbankaufgaben unter Verwendung von LON-CAPA und Praktomat“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. CEUR-WS.org, 2013.

[KSZ02] Jens Krinke, Maximilian Störzer und Andreas Zeller. „Web-basierte Programmierpraktika mit Praktomat“. In: *Softwaretechnik-Trends* 22.3 (2002), S. 51–53.

[MR15] Sebastian Maier und Oliver Rod. „Umsetzung eines Laborversuches auf LON-CAPA mit automatischer Bewertung von ABEL-Programmieraufgaben“. In: *Workshop „Automatische Bewertung von Program-*

mieraufgaben“ (ABP 2015). Bd. 1496. CEUR Workshop Proceedings. CEUR-WS.org, 2015.

[WW12] Debora Weber-Wulff. *Collusion Detection System Test Report 2012*. 2012. URL: http://plagiat.htw-berlin.de/collusion-test-2012/ (besucht am 09. 06. 2016).

11 Der Grader Graja

Robert Garmann

Zusammenfassung
Graja ist ein Autobewerter für Java-Programmieraufgaben. Dieses Kapitel beschreibt die in Graja verfügbaren Funktionen, die technische Architektur, die Aufgabenstruktur sowie Einsatzerfahrungen mit Graja.

11.1 Was ist Graja?

Graja – *Gra*der for *ja*va programs – ist ein automatischer Bewerter für Java-Programme. Die Ausführungen dieses Kapitels basieren auf der Graja-Version 1.5 (März 2016). Graja ist derzeit auf Anfrage für interessierte Lehrende verfügbar, die über die Graja-Webseite[1] Kontakt mit dem Graja-Projekt aufnehmen können.

11.2 Nutzerperspektive

In diesem Abschnitt werden die den Nutzern angebotenen Bewertungs- und Feedbackfunktionen sowie die verschiedenen Nutzerrollen dargestellt.

11.2.1 Bewertungsmethoden

Graja stützt sich zurzeit im Wesentlichen auf den Java Compiler, auf den dynamischen Softwaretest durch das Werkzeug JUnit4[2] und auf die statische Analyse des Quellcodes durch das Werkzeug PMD[3]. Dadurch besitzt Graja in erster Linie Stärken bei der Prüfung der funktionalen Korrektheit einer Einreichung sowie bei der Prüfung von Wartbarkeitseigenschaften des studentischen Programms. Jedoch

1 http://graja.hs-hannover.de
2 http://junit.org
3 http://pmd.github.io

sind die eingesetzten Werkzeuge nicht auf diese Anwendungsbereiche beschränkt. Auch Aspekte der Effizienz, Sicherheit, Zuverlässigkeit und weiterer Qualitätseigenschaften können berücksichtigt werden. Nicht zuletzt kann Graja zu verschiedenen Bewertungsaspekten ein später durchzuführendes menschliches Feedback in das Gesamtfeedback integrieren.

Graja fokussiert ausschließlich auf den Bewertungsprozess und bietet keine Funktionen für üblicherweise in einem Lernmanagementsystem (LMS) implementierte Funktionen wie Kurs- oder Benutzerverwaltung. Auch Plagiatsprüfungen sind nicht in Graja integriert.

11.2.2 Rollen

Wir betrachten drei Nutzerrollen (vgl. Tabelle 11.1): Aufgabenautoren erschaffen eine Aufgabe und müssen sich dazu mit den in Graja eingesetzten Werkzeugen gut auskennen. Die Erstellung einer Aufgabe ist ein kleines Softwareprojekt mit definierten Anforderungen und deren Umsetzung. Lehrpersonen nutzen Aufgaben

Rolle	**Verhalten**
Aufgaben-autor	Entwickelt einen Aufgabentext, Bewertungskriterien, JUnit-Testmethoden und PMD-Regeln; entwickelt Musterlösungen (richtige und falsche)
Lehrperson	Setzt eine Aufgabe in einer Lehrveranstaltung ein; adaptiert ggf. einige Parameter wie Aufgabentext oder Bewertungsschema; führt evtl. nachträglich manuelle Bewertung zu Teilaspekten durch
Student	Verwendet eine Aufgabe als Lernobjekt

Tabelle 11.1: Rollen im Prozess der Aufgabenentwicklung und -nutzung

in ihrem individuellen Lehrkontext. Dazu passen sie Aufgaben an die Lehrsituation an, indem sie von der Aufgabe angebotene Stellschrauben geeignet justieren. Graja bietet standardmäßig Stellschrauben für zu vergebende Punkte und einige Ausgabetexte an, wodurch bereits eine gute Wiederverwendbarkeit einer Aufgabe erreicht wird. Studierende schließlich blicken auf eine Aufgabe als einen Dienst, der ihnen Feedback zu einer eingereichten Lösung liefern kann.

11.2.3 Feedbackmöglichkeiten

Das von Graja erzeugte Feedback besteht aus einer erreichten Punktzahl zusammen mit einem Kommentar. Mit Kommentar bezeichnet Graja eine detaillierte Aufschlüsselung und Erläuterung von Teilergebnissen in strukturierter Textform. Überschriften, Tabellen, Auflistungen, Codeausschnitte und Grafiken können, wenn dies für die betrachtete Programmieraufgabe sinnvoll ist, integriert werden. Sämtliche Bewertungsaspekte können gewichtet werden [GFB16].

Die Compiler-, Test- und Analyseergebnisse werden von Graja aufbereitet, um von Programmieranfängern besser verstanden zu werden. Es wird sowohl eine überblicksartige Zusammenfassung der Ergebnisse in tabellarischer Form generiert als auch Detailfeedback zu jedem bewerteten Aspekt. Der Client – in der Regel ein LMS – erhält das Feedback wahlweise im HTML-Format oder als einfachen Text[4]. Das LMS kann die gewünschte Detailtiefe des Kommentars in mehreren Stufen vorgeben.

Graja bereitet Kommentare für zwei Zielgruppen auf: Kommentare für einreichende Studierende, kurz S(tudent)-Feedback, und Kommentare für Lehrpersonen, kurz T(eacher)-Feedback. Die beiden Kommentare weichen je nach Aufgabe erheblich voneinander ab. Graja nimmt an, dass ein LMS der Lehrperson Einblick in beide Feedbacks gewährt, während es dem Studenten den Einblick in das T-Feedback verwehrt. Das S-Feedback fokussiert auf die Erläuterung der entdeckten Mängel und vermeidet lange Stacktraces oder Fehlerprotokolle, die teilweise sicherheitssensitive Informationen enthalten könnten. Im T-Feedback hingegen können je nach Programmieraufgabe Musterlösungen enthalten sein, Hinweise für Tutoren zur manuellen Nachbearbeitung, Ablaufprotokolle der eingesetzten Werkzeuge, detaillierte Stacktraces zur Fehleranalyse, etc.

11.3 Technische Funktion

Graja erwartet grundsätzlich Quellcode als Einreichung. Eventuell zusätzlich eingereichter Bytecode wird ignoriert und stattdessen selbst von Graja durch Aufruf des Compilers generiert.

4 Derzeit können Bilder und Tabellen nicht in einfachen Text integriert werden. In HTML-Feedback einzufügende, dynamisch vom Grader erzeugte oder statisch vorliegende Bilder werden als Data-URL realisiert.

11.3.1 Besondere Funktionen des dynamischen Softwaretests

Das Spektrum der von Graja bewertbaren Programme reicht von kleinen Anfängerprogrammen des „Hello world“-Typs bis hin zu komplexen nebenläufigen Anwendungen. Studentische Programme können an den folgenden Schnittstellen kontrolliert und beobachtet werden: Console, Datei, Umgebungsinformation der Laufzeitumgebung, Java-Methode und -Attribut, grafische Ein-/Ausgabe (z. B. Java 2D). Die vorgenannten Schnittstellen wurden im praktischen Lehreinsatz genutzt. Noch nicht genutzt, aber prinzipiell möglich ist die Kontrolle und Beobachtung eines studentischen Programms an einer ereignisgesteuerten graphischen Benutzerschnittstelle (z. B. Swing). Hierzu wäre der Einsatz von Drittbibliotheken wie UISpec4J[5] erforderlich.

In der Graja-Bibliothek angebotene Funktionen erleichtern es einem Aufgabenautor, ein studentisches Programm durch Aufruf von dessen Methoden bzw. durch Setzen und Auslesen von dessen Attributen zu steuern und zu beobachten. Da sich Graja auf Java Reflection stützt, kann ein Aufgabenautor Bewertungsroutinen spezifizieren, ohne eine Musterlösung erstellen zu müssen. Selbstverständlich ist es auch möglich, Prüfungen durch Vergleiche von Ausgaben der studentischen Einreichung mit Ausgaben einer vom Aufgabenautor erstellten Musterlösung durchzuführen. Graja überlässt hierbei dem Aufgabenautor die didaktische Entscheidung zwischen der Forderung nach exakter Übereinstimmung und der Tolerierung von bestimmten Abweichungen. Für Ungefähr-Vergleiche stellt Graja verschiedene Routinen[6] zur Verfügung, die erwartete und beobachtete Textausgaben vor dem Vergleich beispielsweise bezüglich Leerraumzeichen und Interpunktion normieren.

Eine studentische Einreichung kann im einfachsten Fall als ganzes Programm inklusive `main`-Methode übersetzt, zur Ausführung gebracht und dabei in seinem Verhalten beobachtet werden. Graja unterstützt den Aufgabenautor durch vorgefertigte Lösungen für den `main`-Methodenaufruf, für die Simulation von Benutzereingaben und für das Abfangen und Prüfen der Consolenausgabe. Weiterhin gibt es praktische Lehreinsätze von Graja, in denen ein eingereichtes Codefragment, z. B. ein Ausdruck mit bestimmten gewünschten funktionalen Eigenschaften, automatisch in einen vom Aufgabenautor erstellten Coderahmen eingesetzt

5 https://github.com/UISpec4J

6 Diese und einige weitere Routinen hat Graja von Web-CAT (http://web-cat.org) geerbt, dem System das an der Hochschule Hannover erstmals für die Autobewertung von Java-Programmen erprobt wurde. Die Erfahrungen mit Web-CAT waren hinsichtlich der Bewertungsergebnisse gut, jedoch wurde Web-CAT als ungeeignet eingeschätzt, um an hiesige Lernmanagementsysteme angebunden zu werden. Dies führte letztlich zur Graja-Neuentwicklung.

und zur Ausführung gebracht wird. Darüber hinaus können durch Verwendung von Mocking-Bibliotheken Teile der studentischen Einreichung durch Musterlösungen ersetzt werden, um den verbleibenden Teil der Einreichung isoliert zu prüfen und auf diese Weise eine irreführende Kette von Folgefehlermeldungen zu vermeiden.

11.3.2 Besondere Funktionen der statischen Analyse

Graja kann eine beliebige Menge der verfügbaren PMD-Regeln unverändert oder in mit PMD-Bordmitteln parametrierter Form einsetzen, um eine Einreichung zu analysieren. Jede PMD-Regel kann separat gewichtet werden. Auch selbst implementierte PMD-Regeln sind denkbar, wurden bisher jedoch nicht eingesetzt. Die Darstellung aller verfügbarer Regeln und deren Einsetzbarkeit würde den Rahmen der hier beabsichtigten Darstellung sprengen. Beispielhaft seien einige Regeln in Tabelle 11.2 zusammen mit dem prüfbaren Bewertungsaspekt genannt. Für detaillierte Beschreibungen der Regeln verweisen wir auf die PMD-Projektwebseite[7].

Noch nicht zum Funktionsumfang von Graja gehört die Möglichkeit, bestimmte Teile des studentischen Codes von der statischen Analyse bzw. von einzelnen Regelprüfungen auszunehmen. Das könnte sinnvoll sein, wenn die studentische Einreichung eine Weiterentwicklung eines von der Lehrperson zur Verfügung gestellten Rumpfprogramms ist.

11.3.3 Weitere technische Funktionen

Graja erlaubt einem Aufgabenautor die Steuerung der Default-Lokalisierung und der Default-Zeichenkodierung der Laufzeitumgebung, die die studentische Einreichung ausführt. So können Aufgaben gestellt und bewertet werden, bei denen die bewusste Berücksichtigung von Lokalisierung und Zeichenkodierung ein Bewertungskriterium darstellt.

Im Normalfall bewertet Graja eine Einreichung zu genau einer Programmieraufgabe. Im Graja-Jargon heißt eine solche Bewertungsanforderung *single request*. Graja kann andererseits eine Einreichung bewerten, die Lösungen zu mehreren Übungsaufgaben enthält. Im Übungsbetrieb einer Präsenzlehrveranstaltung ist es üblich, wöchentlich Aufgabenblätter auszugeben, wobei jedes Blatt mehrere kleine Programmieraufgaben stellt. Graja ermöglicht die Bewertung einer mehre-

7 https://pmd.github.io/pmd-5.4.1/pmd-java/rules/index.html

Aspekt	**Einige PMD-Regeln**
Unnötig aufgeblähter Code	CollapsibleIfStatements, SimplifyBooleanReturns, LogicInversion, AvoidDeeplyNestedIfStmts
Code conventions	MethodNamingConventions, FieldDeclarationsShouldBeAtStartOfClass
Lesbarer Code	ShortClassName, CommentRequired
Wartbarkeitsmetriken	CouplingBetweenObjects, NPathComplexity
Einhaltung von Schnittstellenverträgen	OverrideBothEqualsAndHashcode
Hinweise auf Funktionsfehler oder mangelnde Robustheit	ReturnEmptyArrayRatherThanNull, EmptyCatchBlock, UseEqualsToCompareStrings, SwitchStmtsShouldHaveDefault, ConstructorCallsOverridableMethod
Effizienzfehler	CloseResource

Tabelle 11.2: Beispiele der durch PMD-Regeln prüfbaren Aspekte.

re Übungsaufgaben abdeckenden Einreichung in einem Durchgang. Ein diesbezüglicher Bewertungsauftrag an Graja heißt *multi request*.

Als studentische Einreichung wird sowohl im Falle eines single request als auch im Falle eines multi request ein Zip-Archiv erwartet. Die interne Struktur des Archivs ist vorgegeben (Details: [Gar16]). Studentische Klassen können je nach Aufgabenstellung im unbenannten default Package oder in benannten Packages eingereicht werden.

Von Studierenden eingereichte Quelldateien müssen nicht in einer ganz bestimmten Zeichenkodierung vorliegen. Durch Einsatz der Bibliothek icu4j[8] detektiert Graja die Zeichenkodierung mit hoher Zuverlässigkeit. Auf der sicheren Seite sind Studierende mit Einreichungen in der Zeichenkodierung UTF-8.

Ein Aufgabenautor kann Einreichungen auf bestimmte Packages beschränken. Nicht konforme Einreichungen weist Graja mit einer Erläuterung ab. Zudem kann Graja gezielt einige eingereichte Klassen ignorieren. Dies wird z. B. genutzt, um an Studierende ausgegebenen Code nicht in der ausgegebenen Form, sondern während des Bewertungsvorgangs in einer speziell für die Bewertung abgewandelten Version einzusetzen.

8 http://site.icu-project.org

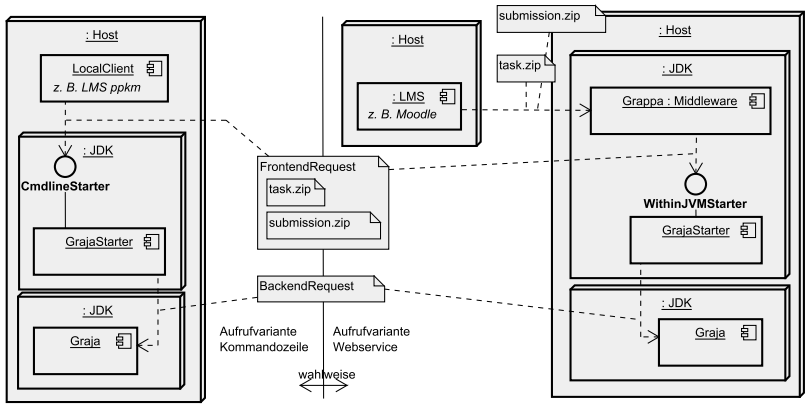


Abbildung 11.1: Ein LMS kann Graja wahlweise als LocalClient über die Kommandozeile (links) oder über einen von der Middleware Grappa angebotenen Webservice starten (rechts).

11.4 Technische Architektur

Graja ist in Java implementiert und wurde bisher auf verschiedenen Windows- und Linuxsystemen genutzt. Graja ist als eigenständiges Programm konzipiert, das über Schnittstellen an verschiedene LMS angebunden werden kann. Zur Ausführung wird ein Java Development Kit (JDK; Java SE 7 oder höher) benötigt (vgl. Abbildung 11.1). Zunächst kann Graja einfach auf der Kommandozeile mit Dateieingaben gestartet werden (Schnittstelle *CmdlineStarter*). Eine Dateieingabe besteht aus einer sogenannten *FrontendRequest*-Datei, welche die Aufgabendatei[9] *task.zip* und die Einreichung *submission.zip* enthält. Die Bewertungsergebnisse stellt Graja nach Abschluss der Bewertung als Ausgabedateien bereit. Auf der Basis der Schnittstelle *CmdlineStarter* wurde Graja in die LMS-Eigenentwicklung *ppkm* der Hochschule Hannover eingebunden. Außerdem besitzt Graja eine Java-API *WithinJVMStarter*, die zur Einbindung in die Middleware Grappa (vgl. Kapitel 23 und [GHW15]) genutzt wurde, welche wiederum als Webservice beispielhaft in das LMS Moodle integriert wurde (vgl. Kapitel 18 und [Stö+14]). Graja ist interoperabel durch die zur Wahl stehenden Schnittstellenformate XML, JSON oder Java-Objekte. XML bzw. JSON werden an der Schnittstelle *CmdlineStarter* erwartet, Java-Objekte an der Schnittstelle *WithinJVMStarter*.

9 Im Falle eines multi request sind mehrere Aufgabendateien in einem *FrontendRequest* enthalten. Die Darstellung hier beschränkt sich auf single requests.

Als Bewertungsergebnis steht ein Aufgabenergebnis zur Verfügung, welches u. a. eine erreichte Punktzahl und ein oder mehrere Base64-kodierte Kommentare im HTML-Format oder in einem einfachen Textformat enthält.

Graja ist intern in zwei Module aufgeteilt. Ein Startermodul sorgt dafür, dass alle benötigten Dateien ausgepackt vorliegen, und startet dann das Hauptmodul. Das Hauptmodul wird in einem separaten Prozess unter den vom Aufgabenautor vorgegebenen Ressourcen- und Klassenpfadbedingungen ausgeführt. Beim Aufruf des Hauptmoduls trifft Graja besondere Vorkehrungen für die Sicherheit des ausführenden Systems und unterbindet unerlaubte Aktionen des studentischen Programmcodes. Dabei stützt sich Graja vornehmlich auf die Java Sicherheitsarchitektur [Gar13].

Graja begrenzt die vom studentischen Programm genutzten Speicherresourcen (flüchtiger und nichtflüchtiger Speicher). Nichtflüchtiger Speicher im Dateisystem wird nur dann begrenzt, wenn das Betriebssystem das Mounten von loop devices[10] unterstützt. Der Bewertungsprozess kann nach einer vom Aufgabenautor oder der Lehrperson vorzugebenden maximalen Zahl von Systemzeitsekunden abgebrochen werden, um etwaige Verklemmungen und Endlosschleifen im studentischen Programm aufzulösen.

11.5 Beispielaufgabe und -einreichung

Um die Bewertung einer Aufgabe illustrieren zu können, betrachten wir eine kleine Beispielaufgabe mit einigen Bewertungskriterien. Die Aufgabe ist ganz bewusst sehr einfach gehalten, bietet aber dennoch Einblick in einige der Bewertungsmöglichkeiten von Graja:

> ***Description**: Write a class `Student` in the package `de.hsh` that stores a name and a matriculation number. Provide a constructor and a `toString` method. Your class should reject illegal, i. e. negative matriculation numbers. Test your class using the following client code:*

```
Student s1= new Student("Smith", 68930);
System.out.println(s1); // prints Smith (68930)
Student s2= new Student("Smith", -1); // throws
                      // IllegalArgumentException
```

10 http://man7.org/linux/man-pages/man8/mount.8.html#THE_LOOP%20DEVICE

> ***Grading criteria**: The program is working functionally correct in the normal case (10 p), it is maintainable (15 p; aspects are encapsulation, code conventions, comments, readability) and it proves to be robust against illegal parameters (5 p).*

Ein Student reiche die folgende Datei als Teil einer Zip-Datei ein:

```
package de.hsh;
/** This class can store a student's data. */
public class Student {
  String name;
  int matrn;
  /** Creates a student.
   * @param name name of the student
   * @param matrn matriculation no. (must not be negative)
   * @exception Exception on invalid parameter */
  public Student(String Name, int Matrn) throws Exception {
    if (Matrn < 0) throw new Exception();
    name= Name;
    matrn= Matrn;
  }
  /** @return a string representation */
  @Override public String toString() {
    return name+" "+matrn;
  }
}
```

Abbildung 11.2 zeigt S(tudent)-Feedback in geringer Detailtiefe zu dieser Einreichung. S(tudent)-Feedback hoher Detailtiefe zeigt Abbildung 11.3. Die erstgenannte Darstellung dient den einreichenden Studierenden vor allem zur ersten Orientierung, bevor sie sich die Detail-Kommentare ansehen. Es ist zu erkennen, dass JUnit, PMD und Mensch als Quelle von Bewertungen auftauchen.

11.6 Aufgabenstruktur

Eine Aufgabe besteht aus Sicht der einzelnen Rollen aus verschiedenen Elementen (vgl. Tabelle 11.3). In diesem Abschnitt betrachten wir die Sichtweisen des Aufgabenautors und der Lehrperson. Die Sicht der Studierenden ist hauptsächlich durch das LMS vorgegeben, dessen Gestaltung nicht im Einflussbereich von Graja liegt.

Category	Aspect	Source	Result	Achieved	Max.
Functional correctness	Should have a functionally correct constructor and toString method	JUnit	**wrong**	**0.00**	**10.00**
				0.00	*10.00*
Maintainability	Attributes in class Student should be private	JUnit	**wrong**	**0.00**	**4.00**
	Variable naming conventions	PMD	**wrong**	**0.00**	**2.00**
	Fields (attributes) should be at the start of the class	PMD		2.00	2.00
	Comments needed in front of methods and classes	PMD		2.00	2.00
	Code should be readable	Non-automated activity	delayed	0.00	5.00
				4.00	*15.00*
Reliability	Should reject illegal matriculation number	JUnit	**wrong**	**0.00**	**5.00**
				0.00	*5.00*
Total scores				**4.00**	**30.00**

Abbildung 11.2: Beispielfeedback von geringer Detailtiefe für Studierende

Functional correctness. Score: 0.00/10.00

- *Wrong. Should have a functionally correct constructor and toString method. Score: 0.00/10.00*

  Output of new Student("John", 79205).toString();

  Expected and observed behaviours differ.

	Expected		Observed	
1	John (79205)	<D>	John 79205	1

  Legend: <D>=difference

Maintainability. Score: 4.00/15.00

- *Wrong. Attributes in class Student should be private. Score: 0.00/4.00*

  There are at least 2 non-private attributes in de.hsh.Student.

- *Wrong. Variable naming conventions. Score: 0.00/2.00*

  Variables should be named conventionally.

  - Variables should start with a lowercase character, 'Matrn' starts with uppercase character.

    ```
    Student.java
    10 : public Student(String Name, int Matrn) throws Exception {
    ```

  - Variables should start with a lowercase character, 'Name' starts with uppercase character.

    ```
    Student.java
    10 : public Student(String Name, int Matrn) throws Exception {
    ```

- *Delayed - Non-automated activity. Code should be readable. Score: 0.00/5.00*

  A grading assistant will manually assign scores for readability of your code. - The evaluation and grading of this aspect is a human activity.

Reliability. Score: 0.00/5.00

- *Wrong. Should reject illegal matriculation number. Score: 0.00/5.00*

  Your constructor should reject an illegal matriculation number with an IllegalArgumentException (observed: Exception).

Abbildung 11.3: Beispielfeedback von hoher Detailtiefe für Studierende

11.6.1 Aufgabenstruktur aus Sicht des Aufgabenautors

Graja bietet einem Aufgabenautor ein vorkonfiguriertes Eclipse[11]-Projekt an, in dem dieser in der Regel mehrere JUnit-Testmethoden programmiert bzw. PMD-Regeln konfiguriert. Die oben beschriebene Beispielaufgabe enthält eine JUnit-Testklasse *Grader.java*, eine PMD-Regeldatei *ruleset1.xml* sowie optional diverse falsche und korrekte Musterlösungen. Eine Klammer um all diese Artefakte bildet eine Datei *descriptor.xml*. Graja bietet dem Aufgabenautor die Möglichkeit, automatisch mit einem gegebenen Buildscript[12] eine verteilbare Komponente im ProFormA-Aufgabenformat (vgl. Kapitel 24 und [Str+15]) zu generieren, die anschließend z. B. über die Weboberfläche eines LMS hochgeladen und dann direkt genutzt werden kann. Graja bietet dem Aufgabenautor alle Freiheiten von JUnit und PMD an, so dass dieser den Prozess der Feedbackerzeugung auf fast unbegrenzte Weise auf die jeweilige Lernsituation einstellen kann. Im Rahmen der bisherigen Einsätze (vgl. Abschnitt 11.7) wurden sowohl kleine Aufgaben mit einer kleinen einstelligen Anzahl von Bewertungsaspekten realisiert als auch Auf-

Rolle	Charakterisierung
Aufgabenautor	**Sicht auf eine Aufgabe:** JUnit-Testmethoden, PMD-Regelauswahl, *descriptor.xml* (vgl. Abschnitt 11.6.1)
	Werkzeugeinsatz: Vorkonfiguriertes eclipse-Projekt oder andere Entwicklungsumgebung
Lehrperson	**Sicht auf eine Aufgabe:** Verteilbare Komponente im ProFormA-Aufgabenformat [Str+15]
	Werkzeugeinsatz: Unzipper, Editor für XML-Dateien, Weboberfläche des LMS zur Konfiguration einer Aufgabe
Student	**Sicht auf eine Aufgabe:** Vom LMS ausgegebener oder auf anderem Wege kommunizierter Aufgabentext; ggf. zugehörige Artefakte wie ein vorgegebener Programmrumpf, sonstige Dateien oder Bibliotheken
	Werkzeugeinsatz: Entwicklungsumgebung zur Erstellung einer Lösung, Weboberfläche des LMS zur Einreichung einer Aufgabenlösung

Tabelle 11.3: Aufgabenstruktur und Werkzeugeinsatz aus Sicht der verschiedenen Rollen

11 https://eclipse.org
12 Derzeit wird ant (http://ant.apache.org) genutzt; in zukünftigen Graja-Versionen wird gradle (http://gradle.org) als Buildwerkzeug genutzt werden.

gaben mit über 20 verschiedenen, in Form von JUnit-Testmethoden umgesetzten Bewertungsaspekten. Der Aufgabenautor muss JUnit-Testmethoden in der Regel aufgabenspezifisch implementieren. Die statische Codeanalyse mit PMD-Regeln hingegen kann in der Regel aufgabenübergreifend in wiederverwendbarer Form spezifiziert werden, indem aus der fast unüberschaubaren Menge existierender Regeln die geeignetsten ausgewählt werden.

In der Datei *descriptor.xml* spezifiziert der Aufgabenautor diverse Einstellungen: Erlaubnisanforderungen an den Securitymanager der die Bewertung durchführenden Laufzeitumgebung, Dateianhänge wie genutzte Drittbibliotheken oder Eingabedaten, Beschränkungen der Ausführungszeit und des Ressourcenbedarfs, und schließlich das Bewertungsschema. Zu jedem technisch umzusetzenden Bewertungsaspekt gibt der Aufgabenautor ein Bewertungsgewicht und einen Titel an und gruppiert diese nach frei einzustellenden Kategorien. Außerdem kann man Platzhalter für nachträglich von Menschen durchzuführende Bewertungen in der Datei *descriptor.xml* vorsehen. Mit vom LMS angebotenen Mitteln kann eine Lehrperson später im Feedback enthaltene Platzhalter durch aktuelle Kommentare ersetzen.

Die Dateien *Grader.java* und *ruleset1.xml* enthalten die technische Umsetzung der Bewertungsaspekte. Abbildung 11.4 definiert drei mit *@Test* annotierte Testmethoden. Alle Methoden setzen Routinen der Graja-Bibliothek ein. Die Methode *setupClass* lädt durch Aufruf von *getPublicClassForName* die studentische Klasse. Sollte diese nicht existieren, weil sie vielleicht vom Studenten falsch benannt wurde, erzeugt *getPublicClassForName* den notwendigen JUnit-Abbruch mit einem Feedback für den Studenten. Die Testmethode *shouldReturnString* demonstriert die Instanziierung der studentischen Klasse, den Aufruf einer Methode sowie den Vergleich von erwartetem und beobachtetem Wert. Es wird ein ungefährer Vergleich spezifiziert, der Unterschiede in Leerraumzeichen ignoriert.

Schließlich werfen wir noch einen Blick auf die Datei *ruleset1.xml* (vgl. gekürzte Darstellung in Abbildung 11.5). Das Format dieser Datei ist von PMD vorbestimmt. In diesem Fall definiert und parametriert der Aufgabenautor drei Regeln. Der Aufgabenautor kann wie hier genau die genutzten Regeln oder auch eine Obermenge der Regeln definieren, die er in einer konkreten Aufgabe benutzen will. Im letzteren Fall muss die Regeldatei nicht für jede Aufgabe separat erstellt und gepflegt werden.

```
package org.myins....studentv01.grader;  ...

@RestrictSubmissionToPackages("de.hsh")
public class Grader extends AssignmentGrader {

  private static final String VALID_NAME= "John";
  private static final int VALID_MATRN= 79205;
  private static Class<?> submission;

  @BeforeClass public static void setupClass() {
    submission= getPublicClassForName("de.hsh.Student");
  }
  @Test public void attributesShouldBePrivate() {
    Support.assertAllAttributesArePrivateOrClassConstants(submission);
  }
  private String toString(Object student) {
    return ReflectionSupport.invoke(student, String.class, "toString");
  }
  @Test public void shouldRejectIllegalMatrn() {
    final int illegalMatrn= -55;
    try {
      ReflectionSupport.createEx(submission, VALID_NAME, illegalMatrn);
      org.junit.Assert.fail("Your constructor should reject "+illegalMatrn+
                            " as matriculation number");
    } catch (IllegalArgumentException ex) {
    } catch (Exception ex) {
      org.junit.Assert.fail("Your constructor should reject an illegal "+
        "matriculation number with an IllegalArgumentException (observed: "+
        ex.getClass().getSimpleName()+")");
    }
  }
  @Test public void shouldReturnString() {
    Object student= ReflectionSupport.create(submission, VALID_NAME, VALID_MATRN);
    DiffHelper
      .compareStrings(VALID_NAME+" ("+VALID_MATRN+")", toString(student))
      .normalizeOutputExcludedFromDiffSynopsis(
        new StringNormalizer(StandardRule.OPT_IGNORE_ALL_WHITESPACE))
      .includeExplanationInDiffSynopsis(new ParagraphComment(Level.INFO,
        Audience.STUDENT,
        "Output of new Student(\""+VALID_NAME+"\", "+VALID_MATRN+").toString();"))
      .start();
  }
}
```

Abbildung 11.4: Gekürzter Inhalt von *Grader.java*

```
<?xml version="1.0"?>
<ruleset name="Simple rules" ...>
  <description>A few simple rules</description>
  <rule ref="rulesets/java/naming.xml/VariableNamingConventions">
    <description>Variables should be named conventionally.</description>
  </rule>
  <rule ref="rulesets/java/design.xml/FieldDeclarationsShouldBeAtStartOfClass"/>
  <rule ref="rulesets/java/comments.xml/CommentRequired">
    <properties>
      <property name="fieldCommentRequirement" value="Ignored"/>
      <property name="protectedMethodCommentRequirement" value="Required"/>
      <property name="publicMethodCommentRequirement" value="Required"/>
      <property name="headerCommentRequirement" value="Required"/>
    </properties>
    <description>Leading comments are required before a class and ...</description>
  </rule>
</ruleset>
```

Abbildung 11.5: Gekürzter Inhalt von *ruleset1.xml*

11.6.2 Aufgabenstruktur aus Sicht der Lehrperson

Eine Aufgabe wie die oben beschriebene wird als Zip-Archivdatei im ProFormA-Aufgabenformat verteilt. Eine im Archiv enthaltene Datei *task.xml* (ohne Abb.) enthält die zentralen Einstellungen für die Aufgabe. Im Prinzip sind in *task.xml* die gleichen Informationen enthalten wie in der Datei *descriptor.xml*. Die Struktur der Daten ist lediglich an das Standard-ProFormA-Format angepasst worden. Da es sich um eine XML-Datei handelt, kann eine Lehrperson den Inhalt der Datei mit Standardwerkzeugen einsehen und verändern. So lassen sich z. B. Punktgewichte und Ausgabetexte problemlos anpassen. Neben der Datei *task.xml* enthält das Zip-Archiv weitere Dateien, u. a. eine Datei *Grader.jar*, welche aus dem Quelltext *Grader.java* entstanden ist, sowie die Datei *ruleset1.xml*.

Die Sicht auf eine Aufgabe ist für eine Lehrperson stark von dem Funktionsangebot des LMS abhängig. Erlaubt das LMS, mehrere Aufgaben eines Übungsblattes gebündelt an einen angeschlossenen Autobewerter zu versenden? Welche diesbezüglichen Konfigurationsmöglichkeiten stehen der Lehrperson zur Verfügung? Kann die Lehrperson eine Aufgabe direkt im LMS bearbeiten oder editiert sie die *task.xml* mit separaten Werkzeugen? Auch der Blick auf die Bewertungsergebnisse ist stark vom LMS abhängig und soll aus diesem Grunde hier nicht weiter vertieft werden.

11.7 Bisherige Einsätze

Graja wird seit 2012 fakultätsübergreifend an der Hochschule Hannover in Programmierlehrveranstaltungen in Informatikstudiengängen eingesetzt. Der Einsatz wurde aus didaktischer und teilweise aus technischer Sicht evaluiert [Stö+13].

Graja wurde bisher überwiegend im Studiengang Angewandte Informatik der Hochschule Hannover im 1. und 2. Semester eingesetzt, und zwar nahezu durchgehend seit September 2012 bis heute (März 2016). In jedem Semester nahmen zwischen 80 und 120 Studierende teil. Darüber hinaus wurde Graja im Studiengang Medizinisches Informationsmanagement der Hochschule Hannover im 1. Semester durchgehend jährlich seit September 2013 bis heute eingesetzt. In jedem Semester nahmen hier ca. 60 bis 70 Studierende teil.

In den bisherigen Einsätzen wurden ca. 50 verschiedene Programmieraufgaben genutzt. Um einen Eindruck von der durch die Aufgaben abgedeckten thematischen Breite zu vermitteln, nennen wir einige Aufgabeninhalte: einfache Consolenaus- und -eingabe, Methodenparameter und -rückgabewerte, Kontrollstrukturen, boolesche Logik, mathematische Problemstellungen, Arrays, Collections,

Datei-Ein-/Ausgabe, Rekursion, Klassen, Vererbung, Interfaces, abstrakte Klassen, nebenläufige Programme. Insgesamt wurde Graja im Zeitraum September 2012 bis März 2016 von ca. 700 an der jeweiligen Lehrveranstaltung teilnehmenden Studierenden mit der Bewertung von ungefähr 18.000 Einreichungen beauftragt.

Eine im Wintersemester 2012/13 unter 56 Erstsemestern durchgeführte Umfrage [Stö+13] belegt durch die in Tabelle 11.4 dargestellten Antworten, dass Studierende das Bewertungssystem als hilfreich erachten.

Fragestellung	**++/+**
Die Meldungen von Graja mussten mir durch andere Personen erklärt werden	79
Graja hat Fehler angezeigt, die keine waren	75
Graja hat Lösungen akzeptiert, die falsch waren	86
Die Überprüfung mittels Graja hat mir beim Verständnis der Aufgaben geholfen	48
Die Überprüfung mittels Graja hat mir bei der Lösung der Aufgaben geholfen	57
Die Überprüfung mittels Graja hat mich dazu animiert, meine Fehler genauer zu analysieren	79
Graja ist beim Auffinden von Fehlern mindestens genauso gut wie ein Mensch	43
Die schnelle Prüfung von Programmieraufgaben mit Graja ist ein großer Vorteil gegenüber menschlich überprüften Lösungen	79
Die automatische Überprüfung von Graja ist gerechter als menschliche Überprüfungen	25
Die Unterstützung durch Graja bei Programmierübungen ist insgesamt hilfreich	79
Ich kann mir vorstellen, auch eine Klausur mit Hilfe von Graja bewerten zu lassen	27

Tabelle 11.4: Evaluierung der Leistungsfähigkeit des Autobewerters Graja (Prozent aller Befragten). Fragen 1 bis 3: fünfstufige Antwortskala immer, oft, manchmal, selten (+), gar nicht (++). Fragen 4 ff.: vierstufige Skala trifft völlig (++) / überwiegend (+) / weniger / überhaupt nicht zu.

Tabelle 11.5 zeigt Ergebnisse von Kurzevaluationen im Einsatzszenario „Übung und Tutorium“. Wenn auch auf dieser Datenbasis nicht gesichert abzuleiten, deu-

tet sich an, dass Studierende von „Bindestrich-Studiengängen“ etwas weniger stark vom Feedback des Autobewerters profitieren können.

Fragestellung	Ergebnisse		
	L1	L2	L3
Wie hilfreich war für Sie das automatisierte Feedback zu Ihren Aufgabenlösungen (1=sehr / 5=nicht hilfreich)?	2,0	2,1	2,3
Führte der Graja-Einsatz dazu, dass Sie Fehler im Code Ihrer Lösung leichter erkennen konnten? (1=ja, 5=nein)?	1,8	2,0	2,7

Tabelle 11.5: Kurzevaluierung des Autobewerters Graja seit 2014 in drei Erstsemesterlehrveranstaltungen der Hochschule Hannover. L1: Angewandte Informatik (WS 2014/15, N=72), L2: Angewandte Informatik (WS 2015/16, N=63), L3: Med. Informationsmanagement (WS 2015/16, N=56). Ergebnisse sind Mittelwerte der von Studierenden gegebenen Antworten.

11.8 Zusammenfassung

Verglichen mit anderen Autobewertern für Java-Programme ist Graja noch relativ jung, bietet jedoch innovative Ansätze zur Feedbackgestaltung und ist durch seine Schnittstellen sehr gut für die Einbindung in fremde Kursverwaltungswerkzeuge vorbereitet. Da Graja weit verbreitete Softwareentwicklungswerkzeuge wie JUnit und PMD einsetzt, ist der Einarbeitungsaufwand für mit diesen Werkzeugen vertraute Aufgabenautoren relativ klein.

Die automatisierte Bewertung von Java-Programmen mit Graja ist Gegenstand intensiver Forschungsbemühungen. In Zukunft ist geplant, die Bewertungsmöglichkeiten von Graja auszudehnen. So sollen Bewertungen ohne Internetzugang, internationalisierte Aufgaben und parametrierte Aufgaben als Grundlage einer zufallsbasierten Erzeugung gleichwertiger aber im Detail unterschiedlicher Aufgaben ermöglicht werden. Zudem ist geplant, die didaktischen Handlungsspielräume durch Maßnahmen wie Strafpunkte für mehrfache Einreichungen, Hinzubuchung zusätzlicher Tests gegen Punktabzug, etc. zu erweitern.

Literatur für dieses Kapitel

[Gar13] Robert Garmann. „Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito.“ In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. 2013.

[Gar16] Robert Garmann. *Graja – Autobewerter für Java-Programme*. Bericht (SerWisS) 941. Hochschule Hannover, 2016. URL: http://serwiss.bib.hs-hannover.de/frontdoor/index/index/docId/941.

[GFB16] Robert Garmann, Peter Fricke und Oliver Bott. „Bewertungsaspekte und Tests in Java-Programmieraufgaben für Graja im ProFormA-Aufgabenformat“. In: *DeLFI 2016 – Die 14. E-Learning Fachtagung Informatik*. Bd. 262. LNI. GI, 2016, S. 215–220.

[GHW15] Robert Garmann, Felix Heine und Peter Werner. „Grappa – die Spinne im Netz der Autobewerter und Lernmanagementsysteme“. In: *DeLFI 2015 – Die 13. E-Learning Fachtagung Informatik*. Bd. 247. LNI. GI, 2015, S. 169–181.

[Str+15] Sven Strickroth u. a. „ProFormA: An XML-based exchange format for programming tasks“. In: *eleed* 11.1 (2015). URL: https://eleed.campussource.de/archive/11/4138.

[Stö+13] Andreas Stöcker u. a. „Evaluation automatisierter Programmbewertung bei der Vermittlung der Sprachen Java und SQL mit den Gradern *aSQLg* und *Graja* aus studentischer Perspektive.“ In: *DeLFI 2013 – Die 11. E-Learning Fachtagung Informatik*. Bd. 218. LNI. GI, 2013, S. 233–238.

[Stö+14] Andreas Stöcker u. a. „Die Evaluation generischer Einbettung automatisierter Programmbewertung am Beispiel von Moodle und aSQLg.“ In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014, S. 301–304.

12 Der Grader aSQLg

Felix Heine und Carsten Kleiner

Zusammenfassung
In diesem Kapitel beschreiben wir das Konzept sowie Implementierungsdetails des Graders aSQLg, der zur Bewertung von SQL-Aufgaben verwendet wird. Zielgruppe der Software sind Studierende in typischen Datenbankvorlesungenen, die im Rahmen der Veranstaltung SQL lernen. Das aSQLg-System ermöglicht es zum einen dem Dozenten bzw. der Dozentin, Aufwand bei der Korrektur von Übungsarbeiten einzusparen, zum anderen unterstützt das System die Studierenden beim Erlernen von SQL, da es möglich ist, die gleiche Aufgabe mehrfach abzugeben und dabei mit Hilfe der Rückmeldungen des Systems die Antwort kontinuierlich zu verbessern.

12.1 Einleitung und Motivation

Das Beherrschen der Sprache SQL ist, neben der Modellierung, ein wichtiges Ziel von Einführungsvorlesungen im Bereich Datenbanken. Die automatische Bewertung von Modellierungsaufgaben erscheint schwierig, wohingegen dies für SQL-Aufgaben gut von Gradern übernommen werden kann. In Einführungsveranstaltungen ist dies besonders wünschenswert, da hier normalerweise große Gruppen unterrichtet werden und die Heterogenität besonders ausgeprägt ist.

Der Grader aSQLg verfolgt daher zwei Hauptziele: zum einen soll er die Studierenden im Lernprozess unterstützen, indem eine Aufgabe ggf. mehrfach und mit Hilfe von Hinweisen des Systems bearbeitet werden kann, und zum anderen soll das Lehrpersonal bei der Korrektur großer Mengen von Aufgabenzetteln entlastet werden. Zusätzlich können die Studierenden zeit- und raumunabhängig die Aufgaben bearbeiten.

Der Einsatz von aSQLg beschränkt sich aber nicht auf Einführungsvorlesungen. Auch in späteren Veranstaltungen kann das System gut verwendet werden, wenn bestimmte SQL-Features unterrichtet werden oder zur Wiederholung in Veranstaltungen, die SQL-Kenntnisse bei den Studierenden voraussetzen.

Da häufig in Kursen bereits ein LMS wie z. B. Moodle im Einsatz ist, beinhaltet aSQLg kein eigenes Interface. Der Grader ist als Bibliothek konzipiert, die in unterschiedlichen Systemen zum Einsatz kommen kann. Es existiert eine Schnittstelle zu Grappa (siehe Kapitel 23), über die aSQLg z. B. in Moodle verwendet wird. Dadurch müssen sich die Studierenden nicht mit einem weiteren System auseinandersetzen, sondern können aSQLg im gewohnten Kontext verwenden.

aSQLg wurde bereits in zahlreichen Kursen unterschiedlicher Niveaus eingesetzt. Evaluationen zeugen von einer generellen Zufriedenheit bei den Studierenden (vgl. [KTH13]). Aufgezeigte Schwächen wurden im Rahmen des Entwicklungsprozesses aufgegriffen und führten u. a. zur Detaillierung der Rückmeldungen.

Nach einem Überblick über verwandte Arbeiten beschreiben wir das Konzept und die Bewertungsschritte des Graders. Darauf folgt eine exemplarische Interaktion eines Studierenden mit dem Grader. Vor der Zusammenfassung beschreiben wir noch aktuelle Entwicklungen und Planungen für den Grader.

12.2 Verwandte Arbeiten

Andere Arbeiten mit Bezug zu aSQLg können grob in zwei Bereiche eingeteilt werden. Zum einen Systeme, die das Lernen von SQL unterstützen und zum anderen solche, die automatisch Lösungen prüfen und bewerten.

Wir betrachten zunächst Publikationen, die sich mit der Unterstützung im Lernprozess beschäftigen. In [Mit03] wird ein solches System beschrieben. Die Studierenden bekommen sofortiges Feedback zu den eingeschickten SQL-Statements. Während der Bearbeitung werden die Studierenden unterstützt, bis sie eine korrekte Lösung gefunden haben. Ähnlich arbeitet das in [KP05b] beschriebene System. In diesem System werden auch über SQL-Abfragen hinausgehende Aspekte berücksichtigt. Beide Systeme haben ein eigenes webbasiertes Frontend. Im Gegensatz zu diesen Systemen setzt aSQLg auf eine Integration in die bekannte LMS-Umgebung.

In den Veröffentlichungen [Pri03; PL04] beschreiben die Autoren ein Bewertungsschema für SQL-Abfragen. Ziel ist eine Unterstützung des Lernprozesses unter der Vermutung, dass die Art der Bewertung das Lernverhalten beeinflusst. Wie auch bei aSQLg steht die interaktive Verbesserung der Antwort im Zentrum, wobei unser Ansatz detaillierter ist. In [DRL07] wird das Werkzeug SQLify beschrieben, welches eine teilweise automatisierte Bewertung von SQL-Abfragen unterstützt. Hier ist allerdings der Schwerpunkt auf Peer-Reviews und Interaktion zur Unterstützung des Lernprozesses, so dass keine vollständige Automatisierung

angestrebt wird. In [Abe+08] wird eine Architektur für SQL-Lernunterstützung und Bewertung beschrieben. Die Bewertungskomponente ist allerdings nicht detailliert erläutert.

In [CP09] ist eine Anwendung basierend auf dem GNU SQL Tutor (siehe [Gnu]) beschrieben. Die Anwendung beinhaltet sowohl einen Teil zum Bewerten als auch zur Lernunterstützung. Der Bewertungsteil ist allerdings nicht detailliert beschrieben. Die Bewertung für einzelne Aufgaben scheint binär zu sein.

Die Generierung von unterschiedlichen Aufgaben aus Vorlagen wird in SQL-KnoT [Bru+10] vorgestellt. Wir halten dies für ein sehr sinnvolles Feature und arbeiten an der Integration einer vergleichbaren Funktionalität.

Im Werkzeug ÜPS, das unter anderem in [Ifl+14b] präsentiert wird, liegt der Fokus auf der Benutzerfreundlichkeit des Systems für den Lehrenden. Es werden auch hier syntaktische Prüfungen sowie Korrektheitsprüfungen des Ergebnisses durchgeführt. Ferner wird durch eine Strukturanalyse versucht, dem Lernenden Unterstützung bei der Verbesserung seiner Lösung zu geben. Das System ist eher als eine eigenständige Anwendung konzipiert und nicht primär für die Integration in ein LMS gedacht. Ferner ist das System eher auf kleine Übungsdatenbanken ausgelegt, da der Lehrende ein Skript zum Anlegen der Testdatenbank mit übergeben muss, das für jeden Lernenden ein eigenes Datenbankschema erzeugt. Dieses Verfahren wäre für sehr große Datenbestände aus Performance-Gründen nicht geeignet. Große Datenbestände ermöglichen es, Aufgaben zu in der Praxis typischen Aspekten von Datenbanksystemen wie bspw. Laufzeitverhalten von Anfragen bei großen Datenbeständen zu stellen. Das System bietet einen Übungs- und einen Abgabebereich, eignet sich also sowohl für den Übungsbetrieb wie auch für Prüfungen/Tests.

Die meisten Tools setzen wie aSQLg u. a. auf einen Ergebnisvergleich um die Korrektheit einer Lösung zu prüfen. Eine alternative Vorgehensweise wird im System SQLator vorgeschlagen [Sad+04]. Hier werden Heuristiken eingesetzt, um die Äquivalenz zwischen einer studentischen Lösung und der Musterlösung festzustellen. Der Nachteil ist, dass möglicherweise korrekte Lösungen nicht akzeptiert werden. Eine Mittelweg wird in [Dol10] vorgestellt. Hier dient der Ergebnisvergleich zur Korrektheitsbestimmung. Ein semantischer Vergleich wird genutzt, um Vorschläge zur Verbesserung der Lösung zu erzeugen.

12.3 Konzept zur automatisierten Bewertung von SQL-Aufgaben

In diesem Abschnitt beschreiben wir unser Konzept zur Bewertung von SQL-Aufgaben. Das Kapitel konzentriert sich dabei auf die Bewertung von Abfragen (Select-Statements). Auf andere Typen von SQL-Kommandos gehen wir im Ausblick ein. Das Konzept ist in Abbildung 12.1 als Flussdiagramm dargestellt. Die Schritte zur Bewertung eines SQL-Statements sind im Einzelnen:

1. Prüfung des Statements auf untersagte oder geforderte Elemente (optional),
2. Prüfung auf syntaktische Korrektheit,
3. Prüfung auf Kosten für die Ausführung,
4. Prüfung der Korrektheit des Ergebnisses,
5. Prüfung des Stils,
6. Ermittlung der Punkte und Erstellung des Berichts.

Als Eingabe für die Prüfung dient zum einen die studentische Abgabe und zum anderen eine vorab vom Dozenten erstellte Musterlösung in Form eines SQL-Statements, welches das in der Aufgabe erwünschte Ergebnis liefert.

Im ersten Schritt wird das Statement auf untersagte oder geforderte Elemente überprüft. Hiermit kann eine bestimmte Lösung für eine Aufgabe erzwungen werden, falls die Aufgabe z. B. ein bestimmtes Konstrukt üben soll. Z. B. kann ein Aufgabentext den Zusatz „Lösen Sie die Aufgabe ohne Verwendung von EXISTS“ enthalten, wenn der Dozent eine Lösung mit Hilfe eines Verbunds wünscht. Die Aufgabe kann dann so konfiguriert werden, dass „EXISTS“ als nicht zulässiges Schlüsselwort aufgeführt wird. Ein Statement mit „EXISTS“ würde dann in diesem Schritt ohne weitere Prüfung abgewiesen. Zur Durchführung dieser Prüfung ist eine syntaktische Analyse des SQL-Statements nötig. Hierzu greift aSQLg auf den JSqlParser [Jsq] zurück. Der Parser erzeugt einen Parse-Baum, welcher von aSQLg durchlaufen wird, um erlaubte oder verbotene Elemente zu identifizieren.

Im nächsten Schritt wird die syntaktische Korrektheit des Statements überprüft. Dies geschieht mit Hilfe der Zieldatenbank, da es ggf. Unterschiede im SQL-Dialekt zwischen JSqlParser und der Zieldatenbank (z. B. Oracle) gibt. Das Statement muss in jedem Fall eine passende Syntax für die Zieldatenbank aufweisen; die Kompatibilität mit dem JSqlParser-Dialekt ist nur erforderlich, falls

der Dozent die optionalen Prüfschritte verwendet. Je nach Konfiguration muss der Dozent die Studierenden über den geforderten Dialekt informieren.

Der genaue technische Ablauf der Syntaxprüfung ist abhängig von der Zieldatenbank. Im Falle von Oracle wird die Syntaxprüfung und die Kostenprüfung in einem Schritt durchgeführt, indem der Ausführungsplan angefordert wird. Wenn dieser nicht erstellt werden kann, wird das Statement als syntaktisch fehlerhaft zurückgewiesen. Für andere DBMS-Typen kann diese Prüfung angepasst werden.

Die Kostenprüfung dient im Wesentlichen dazu, ggf. fehlerhafte Statements abzuweisen, die beim Prüfen zu viel Last auf der Datenbank erzeugen würden. Dazu kann in der Konfiguration eine absolute oder eine relative Kostenobergrenze hinterlegt werden. Die relative Obergrenze bezieht sich auf die Kosten der Musterlösung. Wenn die Obergrenze überschritten wird, wird die Korrektheitsprüfung übersprungen, um das Prüfsystem nicht zu überlasten. Für die Kosten können auch Punkte vergeben werden, wenn z. B. mit Hilfe von Optimizer-Hints effizientere Lösungen möglich sind und dies Ziel der Aufgabe ist.

Im nächsten Schritt wird die Korrektheit des Statements ermittelt. Kern dieser Prüfung ist ein Ergebnisvergleich der studentischen Lösung mit der Musterlösung,

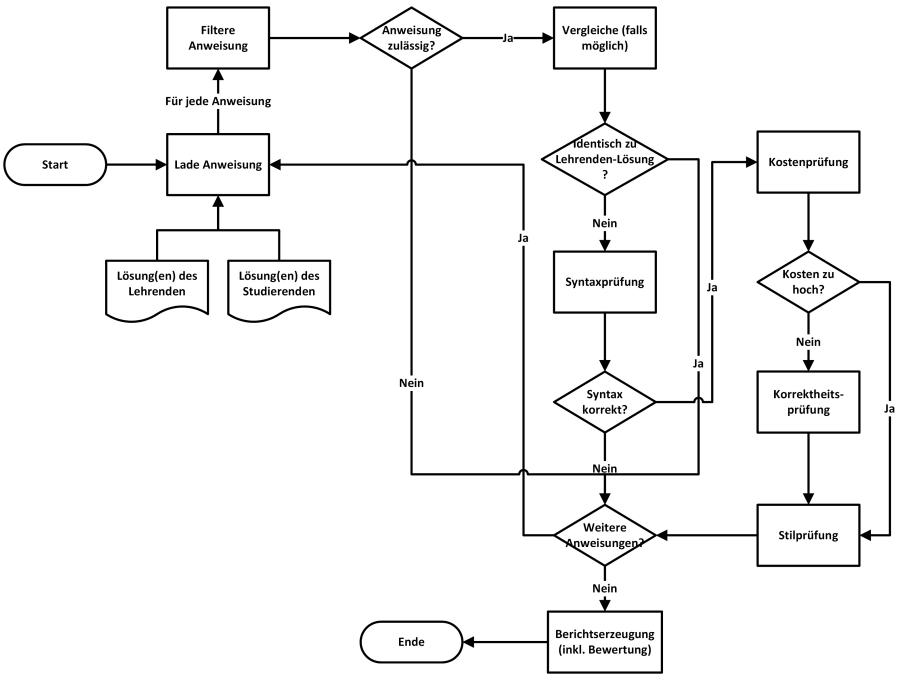


Abbildung 12.1: Flussdiagramm des Grading-Ablaufs

wie in Abbildung 12.2 gezeigt. Um ein genaueres Feedback sowie eine abgestufte Punktevergabe zu ermöglichen, werden allerdings noch weitere Prüfungen durchgeführt.

Zunächst wird geprüft, ob die Ergebnisspalten der beiden Statements in Anzahl, Benennung und Datentyp übereinstimmen. Entsprechende Hinweise können bei der Fehlersuche hilfreich sein. Wenn z. B. eine Spalte mit einem Datum in der Musterlösung den Datentyp „String“ und in der studentischen Lösung den Typ „Date“ hat, kann ggf. geschlossen werden, dass eine passende Formatierung des Datums Teil der Aufgabe ist. Anschließend wird die Zeilenanzahl zwischen den beiden Lösungen verglichen, um Hinweise auf Abweichungen zu geben. So kann z. B. eine zu hohe Zeilenanzahl in der studentischen Lösung ein Hinweis auf einen vergessenen Teil in der „WHERE“-Klausel sein. Zum Abschluss der Korrektheits-

```
( (<student-statement>)
    UNION
  (<reference-statement>))
MINUS
( (<student-statement>)
    INTERSECT
  (<reference-statement>));
```

Abbildung 12.2: Statement zur Korrektheitsprüfung

prüfung wird noch die Reihenfolge der Zeilen geprüft, falls die Musterlösung eine „ORDER-BY“-Klausel enthält.

Der letzte Schritt, in dem Punkte vergeben werden, ist die Stilprüfung. Da es für SQL keine allgemein anerkannten Stilregeln gibt, kann der gewünschte Stil bei aSQLg individuell angepasst werden. Dazu kann der Dozent Stilregeln definieren, deren Einhaltung den Studierenden Punkte einbringt. Es kann z. B. vor bestimmten Teilen ein Zeilenumbruch gefordert werden (z. B. vor der „WHERE“-Klausel), oder es kann gefordert werden, dass Schlüsselwörter in Großschrift geschrieben werden.

Am Ende der Prüfung wird ein Bericht erstellt, der alle Punkte und Feedback sowohl für den Studenten als auch für den Dozenten enthält. Dieser Bericht wird intern als XML-Dokument vorbereitet und über ein Style-Sheet in ein Ausgabeformat wie Text oder HTML umgewandelt. Diese Style-Sheets können angepasst werden, um eine passende Ausgabe zu erreichen. Die fertig formatierten Berichte werden dann z. B. über Grappa an Moodle weitergegeben und dort angezeigt.

Die relative Gewichtung der Punkte aus den einzelnen Schritten lässt sich ebenfalls konfigurieren. So können z. B. bei den ersten Aufgaben bereits 25 % der

Punkte für syntaktische Korrektheit vergeben werden, was ggf. bei fortgeschrittenen Studierenden nicht mehr sinnvoll ist.

Da wir Code der Studierenden auf der Datenbank ausführen, könnten Sicherheitslücken entstehen. Bösartige Studierende könnten versuchen, über diesen Weg die Datenbank anzugreifen. Allerdings ist unser Konzept durch zwei Dinge gut geschützt: Zum einen wird der Code der Studierenden nicht als eigenes Statement ausgeführt, sondern immer nur in Unterabfragen wie in Abbildung 12.2 gezeigt. Zum anderen wird für das Prüfen ein Datenbankzugang verwendet, der nur sehr eingeschränkte Leserechte benötigt. Wenn dieser passend konfiguriert ist, ist ein entsprechender Angriff unmöglich. Bei der geplanten Erweiterung auf DML- und DDL-Kommandos (vgl. Abschnitt 12.6) müsste auch das Sicherheitskonzept angepasst werden, da weder eine Einbettung noch ausschließliche Leserechte möglich sind.

Studierende könnten weiterhin versuchen das System auszunutzen, indem sie z. B. ein Statement „SELECT 5 FROM DUAL“ eingeben, um zumindest Punkte für syntaktische Korrektheit, Kosten und Stil zu bekommen. Als Gegenmaßnahme könnte im Filterschritt z. B. die Verwendung der „DUAL“-Tabelle verboten werden.

Generell ist allerdings anzumerken, dass der Korrektheitstest auf Identität des Ergebnisses beruht, und kein semantischer Vergleich der Lösung mit der Musterlösung stattfindet. Dies bedeutet natürlich, dass Lösungen, die mit völlig anderen Mitteln die korrekte Ergebnisrelation erstellen, als korrekt bewertet werden. Ebenso ist es aktuell nicht vorgesehen, besonders elegante oder ggfs. sogar effizientere Lösungen als die Musterlösung mit besonders positivem Feedback zu versehen. Eine Plagiatserkennung ist derzeit auch nicht vorgesehen. Abhilfe zu diesem Punkt diskutieren wir im Abschnitt 12.6.

12.4 Beispiele

In diesem Abschnitt zeigen wir ein Beispiel aus einer einführenden Datenbankvorlesung. Zu der Veranstaltung gehört eine SQL-Einführung, in deren Rahmen die Studierenden ca. 40 SQL-Aufgaben lösen müssen. Das Beispiel stammt aus dem Beginn der SQL-Einführung. Es wurde ein bekanntes Oracle-Beispielschema verwendet („employees“). Die Aufgabe war es, alle Angestellten mit dem kompletten Namen als ein String sowie dem Einstellungsdatum zu selektieren. Das Ergebnis musste nach Einstellungsjahr und Nachname sortiert werden. Die Musterlösung ist folgende:

```
SELECT first_name || ' ' || last_name name,
TO_NUMBER(TO_CHAR(hire_date, 'YYYY')) hired
FROM hr.employees
ORDER BY hired, last_name;
```

Wir beschreiben jetzt ein vereinfachtes Protokoll eines Studenten, der die Aufgabe lösen wollte. Der erste Versuch war folgende Antwort:

```
Select First_Name||' '||Last_Name,
to_char(Hire_Date, YEAR)
from hr.employees
order by Hire_Date, Last_Name;
```

Die Syntaxprüfung des Graders schlug fehl. Als Teil des Ergebnisberichts wurde auch die Datenbankfehlermeldung von Oracle ausgegeben:

```
\texttt{ORA-00904: "YEAR": invalid identifier.}}
```

Aufgrund der fehlerhaften Syntax wurden alle weiteren Prüfungen übersprungen. Im nächsten Schritt hat der Student den Syntaxfehler korrigiert:

```
... to_char(Hire_Date, 'YYYY') ...
```

Da der Syntaxcheck jetzt erfolgreich war, konnte aSQLg die nächsten Schritte durchführen. Die Kostenprüfung war in diesem Fall unkritisch. Als erster Teil der Korrektheitsprüfung wurden die Spalten vergleichen. Die Spaltenanzahl war zwar wie erwartet zwei, aber die Typprüfung schlug für die Datumsspalte mit folgender Meldung fehl:

```
Datatype of column 2 is wrong.
Expected: NUMBER, your solution: VARCHAR2.
```

Dies veranlasste den Studenten zu folgender Modifikation:

```
... to_number(to_char(Hire_Date, 'YYYY')) ...
```

Die Änderung war erfolgreich, und somit konnte aSQLg die Spaltenprüfung erfolgreich beenden. Zeilenanzahl und Dateninhalte waren ebenfalls korrekt. Allerdings waren die Zeilen noch nicht wie erwünscht sortiert, da die Musterlösung nach Jahr und nicht nach dem gesamten Datum sortiert war. In der letzten Einsendung reagierte der Student auf den entsprechenden Hinweis:

```
Select First_Name||' '||Last_Name,
to_number(to_char(hire_date,'YYYY')) AS datum
from hr.employees
order by datum,Last_Name;
```

Damit konnte die Lösung mit voller Punktzahl akzeptiert werden. Es blieben nur Warnungen bzgl. der Spaltenbenennung, welche aber nicht zu Punktabzug führen. Eine Stilprüfung war in diesem Beispiel nicht konfiguriert.

Insgesamt kann man in diesem Beispiel sehen, wie das detaillierte Feedback Studierende Schritt für Schritt zu richtigen Lösung führen kann.

12.5 Dozentenunterstützung und LMS-Einbettung

Um einen möglichst breiten Einsatz eines Werkzeugs zu erreichen, kommt der Benutzungsfreundlichkeit sowohl für die Lehrenden wie auch für die Lernenden eine große Bedeutung zu. Das Tool aSQLg war ursprünglich als Plugin für das Werkzeug WebCAT ([AEPQ06]) realisiert worden, da dieses automatisierte Programmbewertung für verschiedene Programmiersprachen anbietet. Ein wichtiger Aspekt der Benutzerfreundlichkeit war damit erreicht: ein möglichst breiter Einsatz eines Werkzeugs über mehrere Veranstaltungen. Leider stellte sich heraus, dass die bereitgestellte Oberfläche von WebCAT zum einen weder für Lehrende noch für Lernende besonders benutzerfreundlich war und außerdem auf einer technisch veralteten Basis realisiert worden war. Da WebCAT zudem nicht als vollwertiges LMS konzipiert war, zeigte sich, dass eine Integration von aSQLg in LMS die benutzerfreundlichste Variante für die Lernenden sein würde. Um sich nicht auf ein LMS festlegen zu müssen, wurde das Werkzeug (über eine Zwischenstufe als Standalone-Werkzeug) auf ein Plugin für das Framework Grappa (vgl. Kapitel 23) umgebaut. Damit ist dann eine beliebige LMS-Einbettung möglich, die für die Lernenden maximalen Komfort bei geringem Einarbeitungsaufwand bietet. Die Eingabeschnittstelle für die Lernenden sind Dateien, in denen sie ihre SQL-Kommandos (versehen mit bestimmten minimalen Metainformationen wie Aufgaben-ID) über das LMS bereitstellen. Nach der Bewertung wird das Ergebnis von aSQLg gemäß dem Grappa-Format bereitgestellt und dann im LMS für die Lernenden aufbereitet.

Für die Lehrenden ist der Zugang zum System über ein LMS für den Prozess des Erstellens von Aufgaben zu komplex. Daher wurde basierend auf der o. g. Standalone-Variante für aSQLg eine einfache Java-GUI erstellt, mit der Lehrende Aufgaben und auch Konfigurationsinformationen einfach anlegen und testen können. Die Bedienung von aSQLg über diese Anwendung ist speziell auf die

Bedürfnisse der Lehrenden zugeschnitten und ermöglicht schnelles Erstellen und Testen von Aufgaben. Es können auch Ergebnisse für mögliche studentische Abgaben berechnet und so die Qualität der Unterstützung der Studierenden geprüft werden.

Das Werkzeug aSQLg verwendet intern unterschiedliche Arten von Konfigurationsinformationen, um einen möglichst flexiblen Einsatz zu ermöglichen. Diese Informationen lassen sich grob in die folgenden Bereiche einteilen: Datenbankkonfiguration, Bewertungskonfiguration, Stilprüfungskonfiguration, Sicherheitskonfiguration. Um diese Konfigurationsdateien nicht alle für jede Aufgabe neu anlegen zu müssen (zumal es viele Parameter gibt, die sich nur selten ändern), sieht das Konzept von aSQLg inzwischen vor, dass alle Konfigurationsinformationen selbst in einer Datenbank abgelegt werden und die Inhalte dieser Datenbank dann nur referenziert werden. Die tatsächlich zu übergebende Konfiguration ist damit minimal und kann so im LMS leicht angelegt werden. Bei der Pflege der Detailkonfigurationen in der Datenbank werden Lehrende von der Standalone-GUI unterstützt, so dass auch hier nur minimaler Aufwand anfällt. Abbildung 12.3 illustriert die Lehrendenschnittstelle. Eine Abbildung der aSQLg-Konfigurationen auf

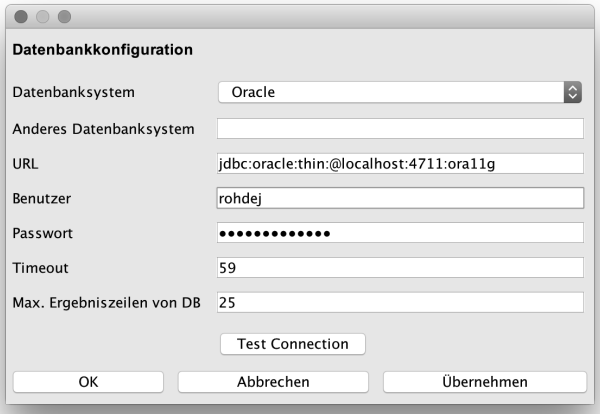


Abbildung 12.3: Screenshot der aSQLg-GUI für Lehrende

das Format zum Austausch von automatisierten Programmieraufgaben (vgl. Kapitel 24) ist ebenfalls erfolgt. Die Standalone-GUI kann die Aufgabenkonfiguration in diesem Format lesen und speichern, um einen Austausch mit anderen Systeme zu erleichtern. Allerdings ist dies bisher nur für die Konfigurationsinformationen und Aufgabentexte realisiert, die Bereitstellung des erforderlichen

Datenbankschemas, auf denen der SQL-Code auszuführen ist, ist bisher nicht implementiert, da das Konzept von aSQLg eine Pflege dieser Daten direkt in der entsprechenden Zieldatenbank vorsieht.

Insgesamt umfasst der Bearbeitungsprozess aus Sicht der Lehrenden die folgenden Aufgaben:

- Anlegen des für die Aufgaben zu verwendenden Datenbankschemas und Import der Daten direkt auf der Ausführungsdatenbank (meist nur einmal pro Veranstaltung).
- Anlegen der häufig zu verwendenden Konfigurationsinformationen in der aSQLg-GUI und Speicherung dieser Informationen in der Datenbank (Grundkonfiguration in wenigen Fällen pro Veranstaltung).
- Anlegen von Aufgabentexten, Musterlösungen und spezifischen Konfigurationen für jede Aufgabe in der aSQLg-GUI; dieser Prozess kann iterativ mit Test der Ausführung aus der GUI heraus erfolgen. Am Ende wird die erstellte Aufgabe gespeichert.
- Anlegen einer Aufgabe im LMS und Hochladen der zuvor gespeicherten Aufgaben- und Konfigurationsdatei (für jede Aufgabe).
- Optional: Nutzung der Auswertungsfunktionalitäten des LMS, um die Ergebnisse der Lernenden zu einer Aufgabe bewerten und berücksichtigen zu können.

Für die Lernenden stellt sich eine aSQLg-Aufgabe im LMS analog zu anderen Aufgaben im LMS dar, so dass eine einfache Bedienung für die Studierenden in der ohnehin gewohnten Umgebung möglich ist.

12.6 Ausblick

In diesem Abschnitt beschreiben wir geplante Erweiterungen des Systems. Dazu gehört das Thema Plagiatsprüfung, die Verfeinerung der Bewertung und des Feedbacks, sowie die Unterstützung von weiteren Typen von SQL-Befehlen.

Wie oben beschrieben, gibt es derzeit in aSQLg keine Plagiatsprüfung. Da das System jedes Statement individuell prüft, ist dies auch im derzeitigen Rahmen nicht lösbar, bzw. würde die persistente Speicherung der bisherigen Lösungen erfordern. Eine weitere Schwierigkeit ergibt sich insbesondere bei einfachen Aufgaben. Diese haben häufig nur eine bzw. wenige sinnvolle Lösungen, so dass eine Plagiatsprüfung zu häufig anschlagen würde.

Wir bevorzugen daher eine Lösung auf Basis der Individualisierung von Aufgaben. Die Grundidee ist dabei, dass Aufgaben mit Platzhaltern variabel gestaltet werden. Ein neues Modul kann dann auf Basis einer solchen Vorlage eine konkrete Aufgabe generieren, wobei z. B. Konstanten verändert werden, Abfragen verändert werden oder sogar andere Spalten genutzt werden. Der Aufgabentext wird entsprechend angepasst. Der eigentliche Bewertungsprozess läuft dann identisch ab, nur kann aufgrund der Veränderungen keine Lösung mehr kopiert werden. Neben der Verhinderung von Plagiaten hat diese Lösung den weiteren Vorteil, dass Studierende zum Üben weitere Variationen einer Aufgabe anfordern können.

Auch wenn der Bewertungsprozess schon verschiedene Teilprüfungen durchläuft, ist eine weitere Verfeinerung denkbar. Zum einen könnten über einen semantischen Vergleich weitere Hinweise zur Verbesserung erzeugt werden. So könnte z. B. erkannt werden, dass das Statement im Prinzip korrekt ist, allerdings in einem Vergleich fälschlicherweise „<“ statt „<=“ verwendet wurde. Diese Analysen könnten neben dem verbesserten Feedback an die Studierenden auch zur weiteren Verbesserung der Abstufung der Bewertungen genutzt werden, mit dem Ziel, dass jede schrittweise Verbesserung auch mehr Punkte erzielt und damit zur Motivation der Studierenden beiträgt.

Aktuell unterstützt aSQLg nur SQL-Abfragen. Wir haben bereits ein Konzept zur Erweiterung auf DML- und DDL-Kommandos erarbeitet. Dies würde es ermöglichen, z. B. „CREATE TABLE“- oder „INSERT“-Anweisungen zu prüfen. Dies erfordert jeweils ein gleiches initiales Schema vor der Bewertung, welches im Anschluss wieder bereinigt werden muss. Ansätze sind bereits prototypisch realisiert worden, diese müssen jedoch noch erweitert und vervollständigt werden. Insbesondere ist das bisher eingesetzte Sicherheitskonzept entsprechend zu erweitern bzw. verbessern.

12.7 Zusammenfassung

In diesem Kapitel haben wir das Konzept von aSQLg vorgestellt. Kern von aSQLg ist ein Bewertungsmodul, welches studentische Lösungen für SQL-Abfragen bewertet und ggf. Hinweise zur Verbesserung der Lösung erstellt. Die Bewertung wird dabei anhand der folgenden Kriterien vorgenommen: syntaktische Korrektheit, Effizienz, Korrektheit des Ergebnisses und Stil des SQL-Statements. Der genaue Ablauf der Prüfung und die Punktevergabe sind konfigurierbar. Optional können über Filter bestimmte Lösungen verboten werden. Die Prüfung läuft so ab, dass auch böswillige Studierende das Prüfsystem nicht kompromittieren können.

Das Ausführen von Statements, welche zu große Last auf das Prüfsystem bringen würden, wird ebenfalls verhindert.

Das System befindet sich seit einigen Jahren im Einsatz. Das Feedback des Systems wurde anhand der im Einsatz gewonnenen Erfahrung (durch Gespräche mit Studierenden und systematische Evaluationen) kontinuierlich verbessert. Ziel ist es, das System so zu gestalten, dass Studierende bestmöglich beim Lernen von SQL unterstützt werden.

Das System kann sowohl zur Lernunterstützung als auch zur Bewertung verwendet werden. Zur reinen Bewertung wird die Wiederholung unterbunden, so dass eine eingeschickte Lösung ohne weitere Verbessungsmöglichkeiten bewertet wird. Das System kann in unterschiedliche LMS eingebunden werden und ermöglicht den Studierenden dadurch, ihre gewohnte Umgebung weiterhin zu nutzen. Das Erlernen der Bedienung einer speziellen Oberfläche ist somit überflüssig.

Durch die automatische Bewertung sind große Effizienzgewinne möglich. Auf der einen Seite wird der Aufwand der Dozenten zur Korrektur minimiert. Auf der anderen Seite müssen Studierende nicht mehr Tage auf die Korrektur ihrer Lösung warten, sondern bekommen sofortige Rückmeldungen, auf die sie direkt mit einer verbesserten Lösung reagieren können. Die schrittweise Erhöhung der Punktzahl wirkt bei manchen Studierenden auch motivierend mit dem Ziel, die volle Punktzahl zu erreichen.

Bereits heute ist das System eine wertvolle Komponente in der SQL-Ausbildung. Durch im Abschnitt 12.6 beschriebene Verbesserungen wird aSQLg in Zukunft noch flexibler einsetzbar sein.

Literatur für dieses Kapitel

[Abe+08] Alberto Abelló u. a. „LEARN-SQL: Automatic Assessment of SQL Based on IMS QTI Specification“. In: *ICALT*. IEEE, 2008, S. 592–593.

[AEPQ06] Rahul Agarwal, Stephen H. Edwards und Manuel A. Pérez-Quiñones. „Designing an adaptive learning module to teach software testing“. In: *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. SIGCSE ’06. ACM, 2006, S. 259–263. DOI: 10.1145/1121341.1121420.

[Bru+10] Peter Brusilovsky u. a. „Learning SQL Programming with Interactive Tools: From Integration to Personalization“. In: *Trans. Comput. Educ.* 9.4 (Jan. 2010), 19:1–19:15. DOI: 10.1145.1656255.1656257.

[CP09] Aleš Cepek und Jan Pytel. „SQLtutor“. In: *Professional Education 2009 – FIG International Workshop Vienna*. FIG Fédération Internationale de Géomètres, 2009.

[Dol10] Robert Dollinger. „SQL Lightweight Tutoring Module – Semantic Analysis of SQL Queries based on XML Representation and LINQ“. In: *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2010*. Toronto, Canada: AACE, Juni 2010, S. 3323–3328.

[DRL07] Stijn Dekeyser, Michael de Raadt und Tien Yu Lee. „Computer Assisted Assessment of SQL Query Skills“. In: *ADC*. Hrsg. von James Bailey und Alan Fekete. Bd. 63. CRPIT. Australian Computer Society, 2007, S. 53–62.

[Gnu] *GNU SQL tutor – website*. URL: http://www.gnu.org/software/sqltutor/.

[Ifl+14b] Marianus Ifland u. a. „ÜPS – Ein autorenfreundliches Trainingssystem für SQL-Anfragen“. In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014, S. 259–264. ISBN: 978-3-88579-627-5.

[Jsq] *JSqlParser – website*. (besucht am 23.07.2011). URL: http://jsqlparser.sourceforge.net/.

[KP05b] Claire Kenny und Claus Pahl. „Automated tutoring for a database skills training environment“. In: *Proceedings of the 36th SIGCSE technical symposium on Computer science education*. SIGCSE '05. ACM, 2005, S. 58–62. DOI: 10.1145/1047344.1047377.

[KTH13] Carsten Kleiner, Christopher Tebbe und Felix Heine. „Automated Grading and Tutoring of SQL Statements to Improve Student Learning“. In: *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. Koli Calling '13. ACM, 2013, S. 161–168. DOI: 10.1145/2526968.2526986.

[Mit03] Antonija Mitrovic. „An Intelligent SQL Tutor on the Web“. In: *I. J. Artificial Intelligence in Education* 13.2-4 (2003), S. 173–197.

[PL04] Julia Coleman Prior und Raymond Lister. „The backwash effect on SQL skills grading“. In: *ITiCSE*. Hrsg. von Roger D. Boyle, Martyn Clark und Amruth N. Kumar. ACM, 2004, S. 32–36.

[Pri03] Julia Coleman Prior. „Online Assessment of SQL Query Formulation Skills“. In: *ACE*. Hrsg. von Tony Greening und Raymond Lister. Bd. 20. CRPIT. Australian Computer Society, 2003, S. 247–256.

[Sad+04] Shazia Sadiq u. a. „SQLator: an online SQL learning workbench“. In: *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*. ITiCSE ’04. ACM, 2004, S. 223–227. DOI: 10.1145/1007996.1008055.

13 Der Grader GATE

Oliver Müller und Sven Strickroth

Zusammenfassung
Das an der TU Clausthal entstandene System GATE *(Generic Assessment & Test Environment) wurde insbesondere zur Verbesserung der Ausbildung im Bereich der Java-Programmierung und zur Unterstützung von Tutorinnen und Tutoren in Java-Programmierübungen mit mehreren hundert Studierenden entwickelt.* GATE *ist webbasiert, plattformunabhängig und in Java geschrieben. Dieses Kapitel geht zum einen auf die Funktionalität des* GATE*-Systems ein und stellt zum anderen dessen Architektur näher vor. Zum Abschluss folgt eine Zusammenfassung wesentlicher Evaluationsergebnisse zum* GATE*-System sowie eine Übersicht über dessen Nutzerzahlen.*

13.1 Einleitung

GATE ist ein Online-Abgabesystem mit integrierter LMS (Learning Management System)-Funktionalität, das verschiedene Funktionen zur Organisation und Verwaltung von Praktika oder vorlesungsbegleitenden Übungen bereitstellt.

Das System wurde v. a. für den Einsatz im Übungsbetrieb von Einführungsveranstaltungen zur Java-Programmierung an Universitäten entwickelt, an denen häufig eine größere Anzahl an Studierenden teilnimmt. Konkret wird das System zurzeit regelmäßig im Rahmen des Übungsbetriebs zweier Veranstaltungen dieser Art verwendet. Bei diesen handelt es sich um einen einführenden Java-Programmierkurs für Studierende der Betriebswirtschaftslehre (ca. 200–300 Studierende) und um einen Java-Programmierkurs für Studierende der (Wirtschafts-) Informatik im zweiten Semester (ca. 80–100 Studierende).

Teile dieses Kapitels entstanden im Rahmen des Projekts eCult, Teilvorhaben eAssessment, gefördert durch das Bundesministerium für Bildung und Forschung unter dem Förderkennzeichen 01PL16066L. Die Verantwortung für den Inhalt dieses Kapitels liegt bei den Autoren.

Um den tendenziell recht hohen Bewertungs- und Korrekturaufwand für Tutorinnen und Tutoren in für diese Veranstaltungen durchgeführten Programmierübungen zu reduzieren, bietet das System Funktionen an, die Tutorinnen und Tutoren insbesondere bei der Korrektur und Bewertung von Aufgaben aus den Bereichen der Java-Programmierung und Modellierung von UML-Klassen- und Aktivitätsdiagrammen unterstützen. Zu Zwecken des Self-Assessment können einige dieser Funktionen auch Studierenden zur Verfügung gestellt werden, mit dem Ziel, die Qualität studentischer Abgaben zu erhöhen.

13.2 Funktionen des GATE-Systems

In den folgenden Abschnitten werden die wichtigsten Funktionen des GATE-Systems erläutert. Abschnitt 13.2.1 geht insbesondere auf den in Abbildung 13.1 gezeigten Anwendungsfall *Aufgaben verwalten* ein. In Abschnitt 13.2.2 wird auf die Möglichkeit eingegangen, in GATE Aufgaben mit randomisierten Werten (dynamische Aufgaben) anzulegen. Abschnitt 13.2.3 beschäftigt sich mit den im GATE-System zur Verfügung stehenden Funktionstests (*Funktionstests definieren*) sowie mit der Frage, wie in GATE Lösungen zu Aufgaben bewertet werden (u. a. *Punkte vergeben*, *Abgaben überprüfen*). Abschnitt 13.2.4 befasst sich mit den Möglichkeiten, Studierenden sowohl textuelles Feedback (*Textuelles Feedback geben*) als auch automatisiert generiertes Feedback über GATE zur Verfügung zu stellen, während Abschnitt 13.2.5 auf die im System zur Verfügung stehende Funktion zur Plagiatserkennung (*Plagiatserkennung konfigurieren*) eingeht.

13.2.1 Funktionen zur Übungsorganisation und -verwaltung

Die wesentlichen Funktionen zur Organisation und Verwaltung von Übungen können wie in Abbildung 13.1 gezeigt ausschließlich von Betreuern einer Veranstaltung genutzt werden. Diese können also u. a. für Teilnehmer einer Veranstaltung Tutorenrechte vergeben (*Tutorenrechte vergeben*) sowie Übungsgruppen anlegen und bearbeiten (*Gruppen verwalten*). Eine weitere zentrale Funktion stellt in diesem Zusammenhang das Anlegen und Bearbeiten von Aufgaben dar (*Aufgaben verwalten*), für die Studierende Lösungen im GATE-System einreichen sollen (*Lösungen einreichen*).

Beim Anlegen einer Aufgabe in GATE lassen sich in einem *ersten Schritt* die im Folgenden aufgeführten Aspekte konfigurieren bzw. angeben:

Allgemeine Informationen zur Aufgabe Für jede Aufgabe lässt sich ein Aufgabentitel und die Aufgabenstellung angeben bzw. formulieren. Jede Aufgabe kann einer von einem Betreuer zuvor angelegten Aufgabengruppe zugeordnet werden, die üblicherweise einem Aufgabenzettel entspricht. Soll eine Aufgabe mit randomisierten Werten angelegt werden, so kann für diese ein in GATE zur Verfügung stehender Aufgabentyp ausgewählt werden.

Form der Abgabe und Bearbeitungszeitraum Das Einreichen einer Lösung zu einer Aufgabe kann in GATE entweder durch Eingabe dieser in ein durch das System bereitgestelltes Textfeld oder durch einen Dateiupload erfolgen. Die erstgenannte Möglichkeit ist für Theorie/Freitext- sowie dynamische Aufgaben vorgesehen. In das bereitgestellte Textfeld können z. B. nicht nur reine Ergebnisse, sondern auch ausführlichere Erläuterung/Begründungen zu Ergebnissen eingetragen werden. Soll zur Abgabe einer Lösung ein Dateiupload vorgenommen werden, so ist es erforderlich, einen regulären Ausdruck anzugeben, der die möglichen Dateinamen für die hochzuladenden Dateien festlegt. Diese Vorgabe ist z. B. für das Einreichen von Java Dateien sinnvoll, um zu verhindern, dass Studierende Java-Programme lediglich als Bytecode einreichen oder diese auf Grund

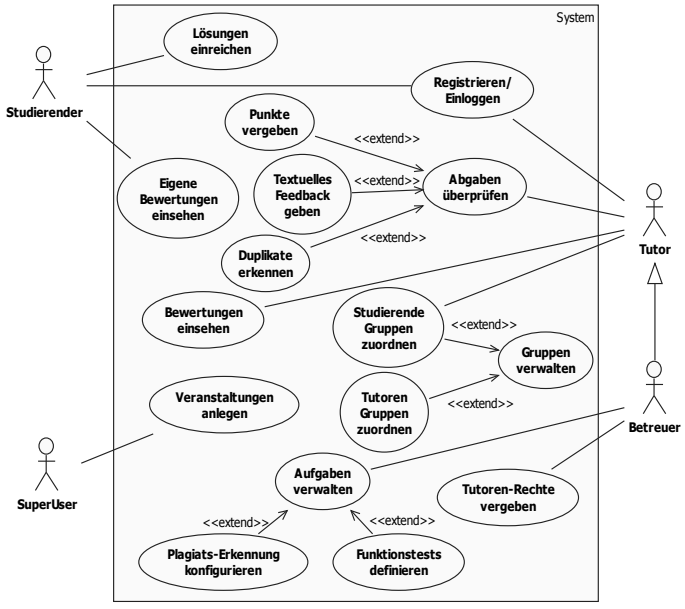


Abbildung 13.1: GATE: Funktionen des Systems und zur Ausführung berechtigte Personengruppen nach [Str09]

eines falsch gewählten Dateinamens nicht ausführbar sind. Ebenso ist es möglich Archive, bestehend aus mehreren Dateien, hochzuladen, die bei entsprechender Konfiguration automatisch entpackt werden. Zudem lässt sich angeben, welche Dateien für einen Tutor automatisch in der Ansicht für Bewertung und Korrektur einer Abgabe hervorgehoben werden sollen.

Das GATE-System lässt neben Einzelabgaben auch Abgaben in Gruppen zu. Die maximale Größe einer Abgabegruppe kann dabei vom Betreuer einer Veranstaltung für jede Aufgabe konfiguriert werden.

Weiterhin kann für eine Aufgabe definiert werden, in welchem Zeitraum eine Abgabe einer Lösung in GATE möglich ist. Innerhalb des festgelegten Abgabezeitraums lassen sich Abgaben stets erweitern bzw. eingereichte Dateien durch überarbeitete/korrigierte Versionen beliebig oft ersetzen. Dies ist u. a. notwendig, um Studierenden eine sinnvolle Nutzung der bereits erwähnten Self-Assessment Funktionalität zu ermöglichen.

Betreuer einer Veranstaltung können optional für jede Aufgabe erlauben, dass von Studierenden für die Lösung einer Aufgabe erstellte Dateien auch von Tutorinnen und Tutoren in das GATE-System hochgeladen werden können. Dies kann z. B. dann hilfreich sein, falls Studierende, die das System zum ersten Mal verwenden, auf Probleme beim Dateiupload stoßen oder aus Kulanz eine Abgabe in Einzelfällen auch nach der Abgabefrist erlaubt wird.

Spezifikation der Punktevergabe Für jede Aufgabe lässt sich die maximal zu erreichende Punktzahl definieren sowie festlegen, ob nur volle Punkte oder auch halbe Punkte, viertel Punkte etc. vergeben werden können. Zudem kann bestimmt werden, wann Studierende für ihre Abgaben zu einer Aufgabe die jeweils vergebenen Punkte im GATE-System angezeigt bekommen. Die Übersicht über die erreichten Punkte kann entweder manuell oder automatisch zu einem vorgegebenen Zeitpunkt für die Studierenden sichtbar geschaltet werden. Die Wahl der erstgenannten Option bietet sich an, wenn eine erfolgreiche persönliche Abnahme einer Lösung durch Tutorinnen/Tutoren erforderlich ist, damit den jeweiligen Studierenden die für ihre Lösungen bereits im System eingetragenen Punkte zugesprochen werden. Bei letztgenannter Option werden die Punkte für alle Studierenden gleichzeitig sichtbar.

In einem *zweiten Schritt* können für eine anzulegende Aufgabe beliebig viele Bewertungskategorien/-kriterien hinzugefügt werden (inkl. Kategorien für Bonuspunkte), für die sich jeweils die maximal zu erreichenden Punkte festlegen lassen.

Außerdem besteht die Möglichkeit, Musterlösungsdateien, die nur für Tutorinnen und Tutoren bzw. Betreuer der jeweiligen Veranstaltung einsehbar sind sowie Dateien, die Studierenden zur Bearbeitung einer Aufgabe bereitgestellt werden sollen, für eine Aufgabe im GATE-System hochzuladen.

Zu guter Letzt lassen sich Funktionstests (siehe Abschnitt 13.2.3) und Plagiatstests (siehe Abschnitt 13.2.5) konfigurieren. Die Tests sollen einerseits Tutorinnen und Tutoren bei der Bewertung von Lösungen unterstützen. Andererseits können Studierende innerhalb der Abgabefrist, sofern vorgesehen, Feedback vom GATE-System zu ihren eingereichten Lösungen über die zur Verfügung gestellten Funktionstest erhalten (für Details siehe Abschnitt 13.2.4).

13.2.2 Aufgaben mit randomisierten Werten

GATE unterstützt prototypisch das Anlegen von Aufgaben mit randomisierten Werten [MS13]. Die Funktion steht zurzeit nur für einige spezielle, vorgegebene Aufgabentypen zur Verfügung. Bei diesen handelt es sich im Wesentlichen um theoretische Aufgaben z. B. zur Berechnung von Speicherplatz bzw. Dateigrößen oder um Aufgaben zum Überführen einer Zahl in ein anderes Zahlensystem.

Bei Aufgaben mit randomisierten Werten wird Studierenden die gleiche Aufgabe mit jeweils individuellen Wertebelegungen für bestimmte Variablen präsentiert. Dies dient u. a. der Vermeidung möglicher Plagiate, da sich Studierende so über Lösungswege austauschen können, aber alle Studierenden erforderliche Berechnungen selbstständig durchführen müssen.

Die Abgabe einer Lösung zu einer Aufgabe mit randomisierten Werten erfolgt über separate Textboxen in GATE, in die der Rechenweg bzw. Zwischenlösungen sowie das von GATE automatisiert überprüfbare Endergebnis eingetragen werden können. Tutorinnen und Tutoren bekommen zu Korrekturzwecken nicht nur die Eingaben der Studierenden, sondern auch die jeweiligen individuellen Wertebelegungen und die für diese Werte vom System automatisch berechneten korrekten Zwischen- und Endergebnisse angezeigt. Somit können Tutorinnen und Tutoren Berechnungen der Studierenden auch im Fehlerfall nachvollziehen, ohne diese mit den individuellen Werten für alle Studierenden manuell durchführen zu müssen.

13.2.3 Grading-Methoden

Hinsichtlich der Korrektur und Bewertung von Lösungen zu in GATE erstellten Aufgaben verfolgt das System einen semiautomatischen Ansatz. Der manuelle Korrektur-/Bewertungsaufwand wird hierbei durch die Bereitstellung automatisch ausführbarer Tests, mit denen überprüft werden kann, ob eine Lösung bestimmte Kriterien (z. B. Kompilierbarkeit eines Java-Programms) erfüllt, verringert. Da die Tutorinnen und Tutoren die abschließende Bewertung bzw. Punktevergabe manu-

ell vornehmen müssen, ist trotzdem eine ganzheitliche Bewertung einer in das GATE-System eingereichten Lösung möglich (vgl. [AM05]).

Ergebnisse der in GATE zur Verfügung stehenden Tests können den für die Bewertung und Korrektur zuständigen Personen nach Ablauf der Frist für die Abgabe einer Lösung zu einer Aufgabe automatisch zur Verfügung gestellt werden. Damit können Tutorinnen und Tutoren schneller sehen, ob eine Lösung korrekt ist oder nicht und somit einfacher zielgerichtetes Feedback für die Lösung vergeben.

Das GATE-System bietet die im Folgenden erläuterten Testmethoden für Java-Programme bzw. UML-Aktivitäts- und Klassendiagramme an.

Compile-/Syntaxtest für Java-Programme Zur Überprüfung eines eingereichten Java-Programms auf syntaktische Korrektheit werden von GATE ein Kompiliervorgang für das Programm angestoßen und die Ausgaben aufgezeichnet.

UML-Vergleichstest Mit Hilfe dieser Art von Test können UML-Klassen- und Aktivitätsdiagramme überprüft werden, die von Studierenden mit einer in GATE integrierten, modifizierten und über Java Webstart ausführbaren Version des Tools Argo-UML[1] erstellt wurden [Sch+12]. Die Prüfmethode basiert dabei auf einem Vergleich der erstellten Diagramme mit einer Musterlösung, die beim Anlegen eines UML-Vergleichstests in GATE hochgeladen werden muss.

Tests auf funktionale Korrektheit von Java-Programmen Da bei dieser Art von Tests eingereichte Java-Programme innerhalb des GATE-Systems ausgeführt werden, kann für diese ein Timeout festgelegt werden, der dafür sorgt, dass die Ausführung nach einer vorgegebenen Zeit abgebrochen wird.

JUnit-Test: Die funktionale Korrektheit eines Java-Programms kann in GATE durch selbst definierte JUnit-Tests überprüft werden. Zum Anlegen eines JUnit-Tests in GATE wird eine .jar Datei, die den kompilierten Test enthält, in das GATE-System hochgeladen. In diesem Zusammenhang ist zudem die Angabe der Main-Testklasse erforderlich. Wird z. B. eine modifizierte Java-JOptionPane Klasse bereitgestellt, so unterstützt GATE auch das automatisierte Testen von Java-Programmen, die GUI (Graphical User Interface) Funktionen nutzen.

Regexp-Test: Bei dieser Art von Test wird ein eingereichtes Java-Programm, optional mit Kommandozeilenparametern, in GATE ausgeführt. Die Ausgabe des Programms wird dabei mit einem regulären Ausdruck verglichen,

1 http://argouml.tigris.org/

der bei der Konfiguration des Tests neben den Kommandozeilenparametern und dem vorzugebenden Namen der Main-Klasse des Programms angegeben wird.

13.2.4 Feedback für Studierende

Feedback für Studierende kann in GATE sowohl manuell durch Tutorinnen und Tutoren bzw. Betreuer vergeben, als auch über die in Abschnitt 13.2.3 aufgeführten Testmethoden automatisiert generiert und bereitgestellt werden. GATE bietet dabei verschiedene Möglichkeiten das Feedback einzuschränken. Zum einen kann die Anzahl der Feedbackanforderungen limitiert werden, um z. B. Gaming-the-System-Ansätze von Studierenden zu unterbinden (vgl. [Bak+05; Bak+08]), zum anderen ist es auch möglich, die Rückmeldung für die Studierenden auf eine einfache „bestanden/nicht bestanden“ Wertung statt detaillierter Ausgaben einzuschränken (vgl. [SOP11]).

Manuell vergebenes Feedback Für Feedback, das manuell vergeben werden soll, steht in GATE eine Freitextkommentarfunktion zur Verfügung. Das Feedback kann dabei an Studierende zur genaueren Erläuterung der Korrektur und Bewertung ihrer Abgaben gerichtet sein. Zudem existiert eine Möglichkeit der abgabenbezogenen internen Kommunikation zwischen den Tutorinnen und Tutoren und Betreuern einer Veranstaltung, die z. B. dazu genutzt werden kann, sich über vermutete Plagiate oder bezüglich Unklarheiten bei der Korrektur und Bewertung einer konkreten Abgabe auszutauschen.

Feedback für UML-Klassen- und Aktivitätsdiagramme Wie bereits in Abschnitt 13.2.3 erwähnt, kann direkt aus GATE eine erweiterte Version des Tools Argo-UML aufgerufen werden. Mit dieser lassen sich Lösungen zu einer in GATE angelegten UML-Modellierungsaufgabe erstellen und im XMI Format nach GATE exportieren. Möchte ein Studierender eine bereits exportierte Abgabe bearbeiten, so wird diese automatisch von der modifizierten Argo-UML-Version abgerufen und angezeigt. Das modifizierte Tool bietet zudem einen Feedback-Button (s. Abbildung 13.2), über den Studierende spezielles Feedback zu ihren nach GATE exportierten UML-Klassen- bzw. Aktivitätsdiagrammen anfordern können [Sch+12].

Das Feedback wird über den in GATE implementierten UML-Vergleichstest automatisiert generiert und nach Anforderung in Argo-UML in einem separaten Fenster in textueller Form ausgegeben (s. Abbildung 13.2). Konkret wird dabei

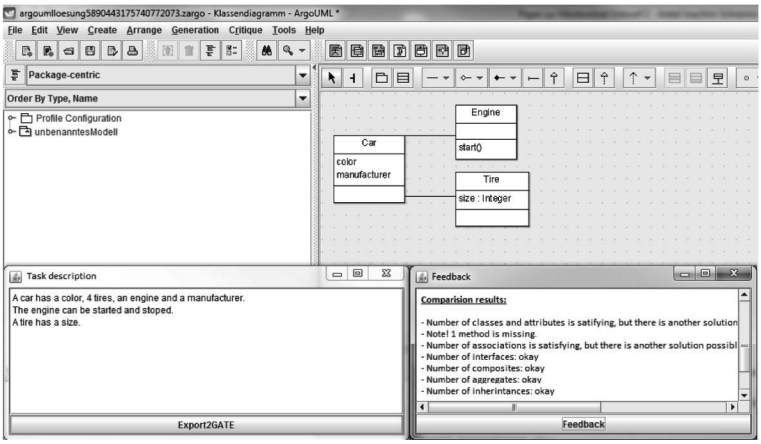


Abbildung 13.2: Feedback mit Argo-UML [Sch+12]

z. B. für jeden durch den UML-Vergleichstest berücksichtigten Elementtyp eines UML-Klassen- bzw. Aktivitätsdiagramms (vgl. [Sch+12], S. 3) angezeigt, ob im Vergleich zur Musterlösung zu wenige, zu viele oder eine richtige Anzahl von Elementen des jeweiligen Typs in der zu prüfenden Modellinstanz enthalten sind.

Automatisiert generiertes Feedback für Java-Programme Diese Art Feedback kann in GATE durch den in Abschnitt 13.2.3 erläuterten Compile-/Syntaxtest bzw. durch die im selben Abschnitt beschriebenen JUnit- und Regexp-Tests bereitgestellt werden. Als Feedback wird dabei angezeigt, ob ein Test erfolgreich war oder nicht. Zusätzlich können weitere Hinweise (wie z. B. Meldungen des Compilers oder der Tests) ausgegeben werden. Als Ergebnis eines nicht erfolgreichen Regexp-Tests wird lediglich die Ausgabe des getesteten Java-Programms sowie das Testergebnis angezeigt.

13.2.5 Plagiatserkennung

Als generelles Problem bei der Bearbeitung von Übungsaufgaben haben sich Plagiate gezeigt. Daher wurde eine Unterstützung für das übungsgruppenübergreifende Auffinden von Plagiaten in GATE integriert. Insgesamt stehen hierfür aus Gründen der Flexibilität drei verschiedene Algorithmen zur Verfügung (vgl. [Str09]), die sich jeweils für spezielle Einsatzzwecke eignen und auch gleichzeitig ausgeführt werden können. Bei der Ausführung eines Plagiatstests werden alle Abgaben zu einer Aufgabe nach einer Normalisierung paarweise miteinander vergli-

chen. Dabei kann, abhängig vom verwendeten Algorithmus, aus einer vorgegebenen Menge von möglichen Normalisierungen gewählt werden (z. B. Entfernen von Leerzeichen oder Kommentaren, Konvertierung zu Kleinbuchstaben, etc.). Als Ergebnis eines Plagiatstests wird für jede Abgabe in GATE für die Tutorinnen und Tutoren die Information hinzugefügt, zu welchen anderen Abgaben eine Ähnlichkeit besteht. Der Grad der Ähnlichkeit wird dabei in Prozent angegeben. Für einen Plagiatstest kann die minimale Ähnlichkeit konfiguriert werden, die jeweils zwei Abgaben zueinander aufweisen müssen, damit sie als hinreichend ähnlich gelten und somit im GATE-System aufgeführt werden. Ebenso können Dateien bestimmt werden, die durch den jeweiligen Plagiatstest nicht berücksichtigt werden sollen (z. B. vorgegebene Vorlagen oder automatisch generierte Dateien der verwendeten Entwicklungsumgebung).

Insbesondere für die Bewertung von möglichen Plagiaten gibt es in GATE eine Funktion, um eingesandte Quelltexte ohne Kommentare anzuzeigen. Tutorinnen und Tutoren können somit z. B. verhindern, dass Studierende, deren Abgaben unter Plagiatsverdacht stehen, bei der Präsentation ihrer Lösungen auf Kommentare zurückgreifen können, die nicht von ihnen selbst verfasst wurden.

Plaggie-Test Für diesen Test wurde die Standalone-Plagiatserkennungsengine *Plaggie* [ASR06], bei der es sich um eine frei verfügbare Variante von JPlag [PMP00] handelt, in GATE integriert. Über Plaggie lässt sich der Source-Code von Java-Programmen miteinander vergleichen, um automatisiert Plagiate in einer Menge von Java-Programmen entdecken zu können. Der Test ist besonders dafür geeignet, Ähnlichkeiten zwischen komplexeren Java-Programmen zu erkennen [SOP11; LC04].

Levenshtein-Test Neben der auf Java spezialisierten Plagiatserkennungsengine Plaggie wird eine Ähnlichkeitsbestimmung zweier textbasierter Abgaben mit Hilfe der Berechnung der *Levenshtein-Distanz* [Lev66] durchgeführt. In diesem Zusammenhang kann zudem in einem gewissen Rahmen festgelegt werden, welche Teile einer Lösung getestet werden sollen (nur Kommentare, nur Quellcode oder beides?). Da insbesondere durch die Berechnung der Levenshtein-Distanz keine Methodenpermutationen erkannt werden können, eignet sich der Einsatz des Levenshtein-Tests vor allem für kleinere Aufgaben (wie z. B. Aufzählen der Zahlen von 1 bis 10 in einer Schleife) bei denen Plaggie größere Übereinstimmungen identifizieren würde. Auch wenn diese Prüfmethode recht elementar erscheint, hat sie sich in der Praxis als sehr nützlich erwiesen.

Normalized-Compression-Distance-Test Als dritte Möglichkeit zur Bestimmung von Ähnlichkeiten von Abgaben wurde eine *Normalized Compression Distance* implementiert, die auf der theoretischen Kolmogorow-Komplexität bzw. der universellen Normalized Information Distance beruht, welche alle anderen berechenbaren Metriken minorisiert, und sich generell für jegliche Art von Daten eignet (auch UML-Diagramme oder sonstige Binärdaten, vgl. [Li+04]). Grundlage dieser Metrik ist die Abschätzung der gemeinsamen Information zweier Strings, für die eine gute Kompressionsfunktion (wie z. B. der Lempel-Ziv-Markow-Algorithmus) benötigt wird. Für Programmcode bzw. Texte sind die gleichen Normalisierungen bzw. Konfigurationen wie beim Levenshtein-Test möglich.

13.3 Architektur und eingesetzte Technologien

Die folgenden Unterabschnitte beschreiben den technischen Aufbau des GATE-Systems. Aus Platzgründen wird in Abschnitt 13.3.1 nur ein Teil seines Datenmodells genauer erläutert (Paket **userdata**, s. Abb. 13.3) und in Abschnitt 13.3.2 ein allgemeiner Überblick über die Systemarchitektur gegeben, ohne auf konkrete Implementierungsdetails einzugehen. Nähere Details zum Datenmodell und zur Architektur des GATE-Systems können in [Str09] nachgelesen werden.

13.3.1 Datenmodell

Das in Abbildung 13.3 dargestellte Datenmodell des GATE-Systems teilt sich in vier wesentliche, als Pakete dargestellte Bereiche auf. Der Bereich **taskdata** umfasst alle Daten zu den in GATE anzulegenden Aufgaben (siehe hierzu auch Abschnitt 13.2.1). Im Bereich **submissiondata** sind alle Daten definiert, die sich auf die Abgaben von Lösungen zu Übungsaufgaben durch Studierende beziehen. Der Bereich **lecturedata** kapselt alle veranstaltungs- oder vorlesungsbezogenen Daten bzw. Klassen. Der Bereich **userdata** umfasst alle Daten, die in GATE zu einem angemeldeten Nutzer angegeben werden können.

Ein Nutzer (User) kann Teilnehmer (Participation) einer in GATE angelegten Veranstaltung (Lecture) sein. Innerhalb der Veranstaltung hat jeder Teilnehmer eine bestimmte Rolle (ParticipationRole). Diese Rolle bestimmt, welche Funktionen ein Teilnehmer in der jeweiligen Veranstaltung ausführen kann. Die bei der Anmeldung in einer Veranstaltung automatisch vergebene Rolle *NormalParticipation* wird für Studierende, die Rolle *Tutor* für die Tutorinnen und Tutoren einer Veranstaltung und die Rolle *Advisor* typischerweise an Betreuer einer Veranstal-

tung vergeben. Unabhängig davon kann ein User in GATE zusätzlich den Status eines Super Users mit administrativen Rechten erhalten, der global für das gesamte GATE-System gilt. Einem Super User ist es z. B. als einzigem erlaubt, neue Veranstaltungen im GATE-System anzulegen.

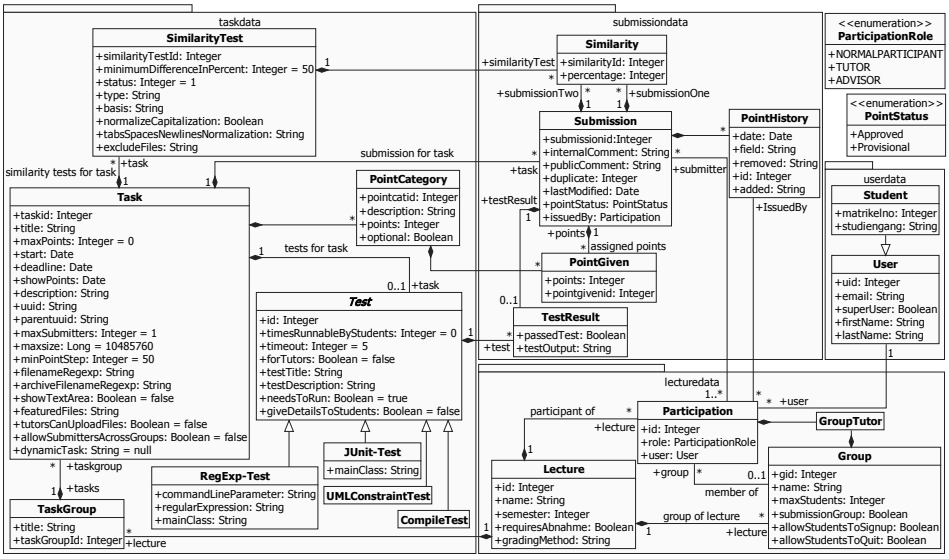


Abbildung 13.3: Das Datenmodell des GATE-Systems nach [Str09]

13.3.2 Systemarchitektur

Bei der Architektur des GATE-Systems handelt es sich um eine Vier-Schichten-Architektur, die sich in eine Benutzerschnittstelle (Graphical User Interface, GUI), die Applikationsschicht, die Persistenzschicht und die Datenbankschicht aufteilt. Grundsätzlich wurden dabei alle Schichten mit Hinblick auf Modularität und einfache Erweiterbarkeit entworfen.

Die **GUI**, die clientseitig im Webbrowser eines Nutzers läuft, nimmt Nutzereingaben entgegen und kommuniziert mittels HTTPs mit der Applikationsschicht. Die Darstellung der Views des GATE-Systems erfolgt über HTML und CSS.

Die GATE-Datenbankschicht basiert auf einem relationalen Datenbankmanagementsystem – MySQL wird dabei zurzeit als **Datenbank** eingesetzt.

Die **Persistenzschicht** enthält das in Abschnitt 13.3.1 beschriebene Datenmodell, verbirgt die Details der konkreten Datenbank und ermöglicht der Applikati-

onsschicht Zugriff auf deren Daten. Die Kommunikation mit der Datenbank wird über Java Database Connectivity (JDBC) in Verbindung mit Hibernate[2] realisiert.

Die **Applikationsschicht** kapselt die gesamte Anwendungslogik. Sie nimmt Anfragen der GUI entgegen, ändert Daten der Persistenzschicht und bereitet Daten für die GUI auf.

Um den speziellen Anforderungen einer webbasierten Anwendung gerecht zu werden, wurde für die Applikationsschicht MVC Model 2 (MVC-2), eine Variante des Model-View-Controller-Musters ([LR06], Kapitel 8.2) angewendet. Die Webserver Virtual Machine (VM) tritt dabei als Front-Controller auf, nimmt die Anfragen der Nutzer entgegen und leitet diese an spezielle Controller-Servlets weiter, welche die weitere Bearbeitung der Anfragen durchführen und schließlich an einen View bzw. weiteren Controller verweisen. Zwecks Kapselung existiert für jede Aufgabe (z. B. Abgabe einer Lösung bzw. Bewertung oder Auflistung aller Aufgaben) ein eigener Controller. Diese verarbeiten die Eingaben der Nutzer, greifen über Data-Access-Object-Klassen (DAO) der Persistenzschicht auf die Datenbank zu, führen die zugehörige Logik aus und leiten an einen entsprechenden View weiter, der schließlich HTML generiert. Den Controllern vorgelagert ist ein Servlet-Filter, der sicherstellt, dass nur authentifizierte Anfragen an die Servlets geleitet werden. Somit können weitere Funktionen hinzugefügt werden, ohne andere Controller verändern zu müssen.

Die Plagiat- und Funktionstests (Studenten- und Tutorentests) wurden innerhalb des Systems als erweiterbares Framework entworfen und implementiert, so dass relativ einfach weitere Tests bzw. Codenormalisierungen eingebunden werden können. Grundsätzlich kann so die Unterstützung durch Tests für weitere Programmiersprachen hinzugefügt werden. Einzige Voraussetzung ist, dass für das Betriebssystem des ausführenden Systems Interpreter bzw. Compiler zur Verfügung stehen und eine sichere Ausführungsumgebung eingerichtet werden kann.

Die in Abschnitt 13.2.3 beschriebenen Tests für Java-Programme werden von dem hierfür zuständigen Controller-Servlet mit Hilfe des Test-Frameworks asynchron im Hintergrund ausgeführt, so dass es zu keinen HTTP-Timeouts kommen kann. Die Ausführung kann zum einen direkt lokal in der VM des Webservers (dabei auch parallel, um alle Cores des Servers zu nutzen) oder aber z. B. durch Nutzung von Remote Method Invocation (RMI) verteilt auf mehreren Computern erfolgen. Die Plagiatstests werden nach Ablauf der Abgabefrist nicht in der Webserver VM, sondern asynchron durch den Einsatz eines Taskplaners (z. B. cron) automatisch in einer separaten VM ausgeführt, so dass Anfragen weder verlangsamt noch blockiert werden.

2 http://www.hibernate.org

Da die Funktionstests für Java-Programme automatisch die eingereichten Programme der Studierenden mit den Rechten des Webservers ausführen, wurden Sicherheitsmaßnahmen eingerichtet, um so gut wie möglich zu verhindern, dass schadhafter Code ausgeführt wird, der z. B. unberechtigterweise Dateien des Servers löscht oder Daten ändert. In diesem Zusammenhang wird die Möglichkeit der Programmiersprache Java genutzt, Rechte (sogenannte *Permissions*) für Java-Applikationen festzulegen. Über den Security-Manager kann dann verhindert werden, dass bei fehlenden Rechten bestimmte mögliche gefährliche Operationen ausgeführt werden. In GATE wird über diesen Mechanismus grundsätzlich die Ausführung aller gefährlichen Operationen (z. B. Löschen oder Schreiben von Dateien, Aufbau von Netzwerkverbindungen ins Internet) durch eine vorgegebene Policy unterbunden. Das Löschen und Schreiben von Dateien wird in einem temporären Verzeichnis für Tests erlaubt, um auch Aufgaben testen und ausführen zu können, bei denen Studierende Dateien anlegen bzw. modifizieren sollen.

Aufgrund der gewählten Systemarchitektur wird zur Ausführung von GATE eine Webserver-Software benötigt, die mit der Java-Servlet-3.0-Spezifikation kompatibel ist und einen Servlet- bzw. Webcontainer bereitstellt (z. B. Tomcat 7).

13.4 Nutzerstatistiken und Evaluationsergebnisse

GATE wird seit dem Wintersemester (WS) 2009/2010 regelmäßig im Rahmen des Übungsbetriebs mehrerer Veranstaltungen der TU Clausthal eingesetzt. Das System kommt dabei nicht nur in den in Abschnitt 13.1 genannten Einführungsveranstaltungen zur Java-Programmierung zum Einsatz, sondern auch in erster Linie zur Organisation und Verwaltung von Übungen sowie zur Unterstützung bei der Aufdeckung von Plagiaten in Einführungsveranstaltungen der Wirtschaftsinformatik und Informatik, in denen keine Java-Programmierung gelehrt wird. Tabelle 13.1 gibt eine Übersicht über die Anzahl der Nutzer, die im GATE-System der TU Clausthal seit dem WS 2009/2010 angemeldet waren.

Eine Evaluierung des GATE-Systems fand zum ersten Mal im Rahmen der Grundlagenveranstaltung zur Java-Programmierung für Studierende der Betriebswirtschaftslehre an der TU Clausthal im Jahre 2009 statt [SOP11]. Die Evaluierung ergab einen statistisch signifikanten Unterschied bezüglich der Qualität der in GATE eingereichten Lösungen der Studierenden, welche die in GATE zur Verfügung gestellten Compile-/Syntaxtests einsetzten (im Schnitt 90% der Lösungen syntaktisch korrekt) und den Lösungen der Studierenden, die dies nicht taten (im Schnitt 65% der Lösungen syntaktisch korrekt). Ebenso verhielt es sich mit den bereitgestellten Tests auf logische Korrektheit von Java-Programmen, bei denen

Lösungen mit durchgeführtem Test im Schnitt zu 52% korrekt waren und Lösungen ohne durchgeführtem Test im Schnitt zu 18%.

Bezüglich der Plagiatstests, die bei Abgaben zu Java-Programmieraufgaben durchgeführt wurden, wurde von den befragten Tutorinnen und Tutoren der Plaggie-Test als am nützlichsten empfunden (Bewertung im Schnitt 4,3 auf einer Skala von 1 bis 5, wobei 5 besser ist), der Levenshtein-Test und der Normalized-Compression-Distance-Test dagegen als weniger nützlich (durchschnittliche Bewertung jeweils 2,8 auf einer Skala von 1 bis 5).

Die befragten Tutorinnen und Tutoren bescheinigten, dass durch die zur Verfügung gestellten Funktionen des GATE-Systems der manuelle Korrektur- und Bewertungsaufwand verringert werden kann.

Einige der eingereichten Lösungen (201 Lösungen von 1031 Lösungen) wiesen nur kleinere Fehler auf (z. B. Tippfehler, Packages falsch). Die auf diese Lösungen angewendeten Syntax-/Compile-Tests bzw. Tests auf die logische Korrektheit von Java-Programmen schlugen folgerichtig fehl, da die abgegebenen Programme aufgrund der vorhandenen Fehler nicht kompilierbar waren. Die Abgaben wurden nach manueller Kontrolle der Tutorinnen und Tutoren trotzdem mit der vollen

Semester	Veranstaltungen	Betreuer	Tutoren	Studierende
SS 2016	2	4	5	164
WS 2015/2016	3	7	14	421
SS 2015	2	6	5	218
WS 2014/2015	2	3	13	399
SS 2014	2	6	7	185
WS 2013/2014	2	3	15	342
SS 2013	3	6	6	107
WS 2012/2013	2	4	18	376
SS 2012	3	7	1	113
WS 2011/2012	2	5	18	521
SS 2011	2	4	3	87
WS 2010/2011	1	2	12	341
SS 2010	1	2	2	45
WS 2009/2010	1	2	12	317
Gesamt	28	61	131	3636
Durchschnitt	2	4	9	259

Tabelle 13.1: GATE-System der TU Clausthal: Anzahl angemeldeter Nutzer pro Semester

Punktzahl bewertet. Die Tutorinnen und Tutoren haben also, wie für das System vorgesehen, eine ganzheitliche Bewertung vorgenommen, die nicht nur auf den Ergebnissen automatisiert ausführbarer Tests basiert.

Die Plagiatstests wurden insbesondere als Indikator benutzt, um mögliche Plagiate leichter erkennen zu können. Abgaben, die in GATE als mögliche Plagiate angezeigt wurden, wurden dennoch normal bewertet, wenn eine persönliche Abnahme der jeweiligen Lösung eines Studierenden durch Tutorinnen und Tutoren erfolgreich war. Das Ziel war es also herauszufinden, ob Studierende ihre eingereichte Lösung verstanden haben und erklären konnten.

Die Wirksamkeit des in GATE integrierten UML-Vergleichstests wurde in einer Studie mit 30 Teilnehmern mit geringen Programmier- bzw. Modellierungskenntnissen evaluiert [Sch+12]. Es konnte dabei festgestellt werden, dass die mit Hilfe der zur Verfügung gestellten UML-Vergleichstests erstellten Lösungen im Schnitt qualitativ besser waren (im Schnitt 94,4 von 104 Punkten), als die Lösungen, die mit dem Tool Argo-UML ohne weitere Hilfestellung generiert wurden (im Schnitt 82,2 von 104 Punkten). Die Teilnehmer, die auf Feedback eines Tutors zurückgreifen konnten, schnitten bei dem Test am besten ab (im Schnitt 100,9 von 104 Punkten). Die Qualitätsunterschiede zwischen den Lösungen der verschiedenen Gruppen waren hierbei jedoch nicht statistisch signifikant. Insbesondere konnte somit die Hypothese nicht bestätigt werden, dass durch den Einsatz des erweiterten GATE-Systems Modellierungsfähigkeiten statistisch signifikant besser erlernt werden können als durch die Verwendung eines UML-Tools ohne Feedbackfunktion. Dennoch bestätigten die Studienteilnehmer, dass durch die vom GATE-System bereitgestellten UML-Vergleichstests ein guter Lerneffekt erzielt werden kann (Bewertung auf einer Skala von 0 bis 5, wobei 5 besser ist, im Schnitt 4,4). Das durch die Tests automatisiert generierte Feedback wurde ebenfalls als sehr nützlich empfunden (Bewertung auf einer Skala von 0 bis 5 im Schnitt 4,3).

13.5 Zusammenfassung und Ausblick

Dieses Kapitel stellt das webbasierte, plattformunabhängige System GATE vor, das seit dem WS 2009/2010 regelmäßig in mehreren Veranstaltungen der TU Clausthal zum Einsatz kommt. Bei GATE handelt es sich um ein Online-Abgabesystem mit integrierter LMS-Funktionalität, das verschiedene Funktionen zur Organisation und Verwaltung von Praktika bzw. vorlesungsbegleitenden Übungen bereitstellt. Der Funktionsumfang des Systems umfasst des Weiteren eine konfigurierbare automatische Plagiatserkennung sowie verschiedene Methoden zur automatisierten Überprüfung von Java-Programmen und UML-Klassen- bzw. Akti-

vitätsdiagrammen, die Studierenden sowie Tutorinnen und Tutoren automatisiert generiertes Feedback für eingereichte Java-Programme bzw. UML-Klassen- und Aktivitätsdiagramme zur Verfügung stellen können. Zudem wird prototypisch das Anlegen bestimmter Aufgaben mit randomisierten Werten unterstützt.

Die modulare Architektur des Systems erlaubt es grundsätzlich, dieses z. B. um neue Tests für weitere Programmier- oder Modellierungssprachen zu erweitern und das bereitgestellte Feedback zu optimieren bzw. zu flexibilisieren. Ebenso können zur Optimierung der Plagiatsprüfungsfunktion weitere Algorithmen bzw. Frameworks zur Erkennung von Plagiaten integriert werden.

Das in der Programmiersprache Java geschriebene GATE-System ist eine GPLv3 lizenzierte Open-Source Software. Der aktuelle Quellcode des Systems inklusive kurzer Installationsanleitung wird auf GitHub zur Verfügung gestellt[3].

Literatur für dieses Kapitel

[AM05] Kirsti M. Ala-Mutka. „A Survey of Automated Assessment Approaches for Programming Assignments“. In: *Computer Science Education* 15.2 (2005), S. 83–102. DOI: 10.1080/08993400500150747.

[ASR06] Aleksi Ahtiainen, Sami Surakka und Mikko Rahikainen. „Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises“. In: *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*. Baltic Sea '06. ACM, 2006, S. 141–142. DOI: 10.1145/1315803.1315831.

[Bak+05] Ryan Shaun Baker u. a. „Do Performance Goals Lead Students to Game the System?“ In: *Artificial Intelligence in Education: Supporting Learning Through Intelligent and Socially Informed Technology*. Frontiers in artificial intelligence and applications 125. IOS Press, 2005, S. 57–64.

[Bak+08] Ryan Baker u. a. „Why students engage in “gaming the system” behavior in interactive learning environments“. In: *Journal of Interactive Learning Research (JILR)* 19.2 (2008), S. 185–224.

[LC04] Thomas Lancaster und Fintan Culwin. „A comparison of source code plagiarism detection engines“. In: *Computer Science Education* 14.2 (2004), S. 101–112.

3 https://github.com/csware/si, https://repo.cses.informatik.hu-berlin.de/gitlab/gate/

[Lev66] Vladimir I. Levenshtein. „Binary Codes Capable of Correcting Deletions, Insertions and Reversals“. In: *Soviet Physics Doklady* 10 (1966), S. 707.

[Li+04] Ming Li u. a. „The similarity metric“. In: *Information Theory, IEEE Transactions on* 50.12 (2004), S. 3250–3264.

[LR06] Bernhard Lahres und Gregor Raýman. *Praxisbuch Objektorientierung – Professionelle Entwurfsverfahren*. Galileo Computing, 2006. ISBN: 978-3-88579-282-6. URL: http://openbook.galileodesign.de/oo/.

[MS13] Oliver Müller und Sven Strickroth. „GATE – Ein System zur Verbesserung der Programmierausbildung und zur Unterstützung von Tutoren“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. 2013.

[PMP00] Lutz Prechelt, Guido Malpohl und Michael Philippsen. „Finding Plagiarisms among a Set of Programs with JPlag“. In: *JOURNAL OF UNIVERSAL COMPUTER SCIENCE* 8 (2000), S. 1016–1038.

[Sch+12] Joachim Schramm u. a. „Teaching UML Skills to Novice Programmers Using a Sample Solution Based Intelligent Tutoring System“. In: *Proceedings of the 25th International Conference of the Florida Artificial Intelligence Research Society (FLAIRS)*. Marco Island, FL, USA: AAAI, 2012, S. 472 –477.

[SOP11] Sven Strickroth, Hannes Olivier und Niels Pinkwart. „Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben?“ In: *DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik*. Bd. 188. LNI. GI, 2011, S. 115–126.

[Str09] Sven Strickroth. *Unterstützungsverfahren für Tutoren bei der Online-Abgabe von Übungsaufgaben*. Bachelorarbeit. 2009.

14 Der Grader VEA

Helmar Gust

Zusammenfassung
Vips ist ein virtuelles Prüfungssystem und dient zur Verwaltung, Durchführung und Auswertung von Online-Übungen und Online-Klausuren. Der Schwerpunkt der Vips-Entwicklung liegt auf der automatischen Bewertung von Aufgaben, um demjenigen, der die Aufgaben stellt und beurteilen muss, möglichst viel Arbeit abzunehmen. Hierfür wurde das Gradermodul VEA entwickelt, das Lösungen von Programmieraufgaben in Prolog, Lisp und ähnlichen Programmiersprachen bewerten kann [GW13]. Vips ermöglicht es, neben dem Prüfungsmodus Übungen und Klausuren für die Studierenden in einem Selbsttestmodus anzubieten. Das System unterstützt verschiedene Möglichkeiten zur Entwicklung, Pflege und Auswertung virtueller Aufgabenblätter und Prüfungen sowie die Verwaltung des gesamten Übungsbetriebs eines Kurses.

14.1 Einleitung

Vips wurde entwickelt, um einfach online Übungen und Klausuren durchführen zu können und die Lehrenden bei einer Reihe organisatorischer Aufgaben zu entlasten. Neben der Erstellung und Verwaltung von Übungsblättern und Klausuren umfasst Vips eine Arbeitsgruppenverwaltung, Punkte- und Notenübersichten für einzelne Teilnehmer und Arbeitsgruppen, sowie eine flexible Notenberechnung. Die Korrektur und Bewertung von Übungsaufgaben im Rahmen von Lehrveranstaltungen erfordert von den Lehrenden einen nicht unerheblichen Aufwand an Zeit. Daher haben sich in vielen Bereichen stark schematisierte Aufgabentypen durchgesetzt. Typische Beispiele dafür sind Multiple- und Single-Choice-Aufgaben, Ja/Nein-Fragen sowie Zuordnungsaufgaben. Auch für solche Aufgaben ist die händische Korrektur zeitaufwändig und lästig. Allerdings lässt sich die Auswertung dieser Aufgaben sehr leicht automatisieren. Vips führt für diese Aufgaben standardmäßig eine komplett automatische Bewertung durch. Allerdings

kann das Ergebnis auch bei diesen Aufgaben von Lehrenden nachträglich überschrieben werden.

Immer dann, wenn freie Texteingaben erwartet werden (und dies gilt bereits bei Lückentextaufgaben), ist für die Beurteilung aber ein gewisses Verständnis der Aufgabe notwendig, zumindest dann, wenn von der vorgegebenen Musterlösung abgewichen wird. Aber auch bei diesen Aufgaben lässt sich die Korrektur und Bewertung von automatischen Systemen zumindest unterstützen. So hilft bereits das Herausfiltern klarer Standardfälle, einen erheblichen Teil des Zeitaufwandes einzusparen, etwa wenn auf der einen Seite Musterlösungen erkannt und auf der anderen Seite eindeutige Fälle der Nichtbeantwortung herausgefiltert werden können. Programmieraufgaben nehmen eine Sonderstellung ein. Zum einen handelt es sich um Aufgaben, die sich weitgehend wie Freitextaufgaben verhalten. Zum andern gibt es aber starke formale Restriktionen an die eingegebenen Texte: Sie müssen als Programmcode fehlerfrei kompilierbar sein und sie müssen das geforderte Programmverhalten produzieren. Die Korrektur und Bewertung von Programmieraufgaben ist normalerweise noch erheblich aufwändiger als bei Freitextaufgaben, insbesondere dann, wenn sie Fehler enthalten. Eine Unterstützung bei diesen Aufgaben ist also sehr hilfreich, aber wegen der Komplexität solcher Aufgaben auch entsprechend schwierig. Genau hier liegt ein Schwerpunkt der Vips-Entwicklung: die automatische Auswertung von Programmieraufgaben, mit dem Ziel, demjenigen, der die Aufgaben stellt und beurteilen muss, möglichst viel Arbeit abzunehmen. Darüber hinaus ermöglicht Vips Übungen und Klausuren für die Studierenden in einem Selbsttestmodus anzubieten.

Vips stellt für Programmieraufgaben eine Runtime-Umgebung mit einer einfachen GUI zur Verfügung. Dies ermöglicht den Kursteilnehmern, die Aufgaben ohne lokale Installationen der Programmiersprachen zu lösen. Allerdings stehen interaktive Debug-Möglichkeiten in einer solchen Umgebung nicht zur Verfügung. Es kann daher auf eine lokale Installation der Programmiersprachen vor allem bei komplexen Aufgaben nicht immer verzichtet werden. Um lokale Installationen zum Aufgabenlösen benutzen zu können, gibt es Up- und Download-Möglichkeiten für die Aufgaben. Die Runtime-Umgebung steht auch für die Aufgabenkorrektur zur Verfügung. Zusammen mit einem vorgegebenen Default-Aufruf der Hauptfunktion eines Programms ermöglicht dies sowohl dem Kursteilnehmer als auch dem Korrekteur sich mit einem Mausklick einen ersten Überblick über die Lauffähigkeit der Lösung zu verschaffen.

Zur Unterstützung der Bewertung dieser Programmieraufgaben gibt es ein eigenes Servermodul VEA (Vips-Evaluation-Assistent), das auf einem vom StudIP/Vips-System getrennten Rechner läuft. Die Kommunikation zwischen StudIP/Vips und dem Servermodul VEA geschieht über HTTP. Mit einem Browser kann diese

Schnittstelle auch direkt benutzt werden[1], um VEA zu testen und zu konfigurieren. Vips entstand aus Ansätzen, die im Vorfeld und im Rahmen eines Projektes am Institut für Kognitionswissenschaft entwickelt wurden [Pey+00]. Vips ist als Plugin für Stud.IP[2] konzipiert und daher zurzeit nur in Kombination mit Stud.IP verfügbar. Einen Überblick über die Architektur gibt Abbildung 14.1.

14.2 Der Vips-Evaluations-Assistent VEA

Zur Unterstützung der Auswertung von Programmieraufgaben gibt es ein Bewertungsmodul (Grader Module) VEA (Vips-Evaluation-Assistent), das als Evaluationsserver auf einem vom StudIP-System getrennten Rechner läuft. Sinnvoll ist hier ein Rechner, der nur für diese Aufgabe zur Verfügung steht, da die Ausführung von fremdem Programmcode immer auch mit Sicherheitsrisiken verbunden ist. VEA stellt Runtime-Umgebungen für Programmieraufgaben in ausgewählten Programmiersprachen zur Verfügung. Diese Umgebungen können zur Entwicklung der Lösungen und zur Bewertung dieser Lösungen benutzt werden. Vips/VEA wurde ursprünglich für Scriptsprachen ausgelegt, bei denen nach dem Laden von Runtime-Umgebung und Programm eine (interaktive) Oberfläche zur Verfügung steht, in der einzelne Codesegmente oder Prozeduren und Funktionen

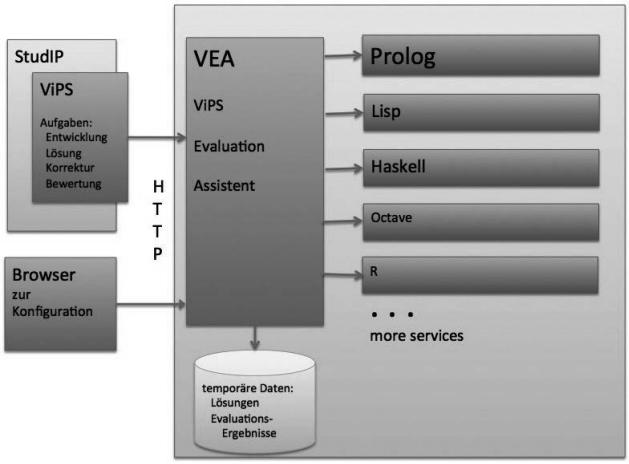


Abbildung 14.1: ViPS/VEA-Architektur

1 https://mvc.ikw.uni-osnabrueck.de/vips/vips.php
2 www.studip.de

ausgeführt werden können, ohne dass ein expliziter Compilationsschritt des gesamten Programms notwenig ist. Es läßt sich aber auch auf Compilersprachen erweitern. Nicht vorgesehen sind allerdings komplexe Programmstrukturen aus vielen interagierenden Modulen. Im Gegensatz zu klassischen Unit-Tests basieren die Tests auf Musterlösungen, die die korrekte Input/Output-Relation für die zu testende Funktion spezifizieren:

- Ein Generator erzeugt Input-Parameter für die zu testende Funktion.
- Die Musterlösung berechnet die zugehörigen Ausgabewerte.
- Diese Ausgabewerte werden mit den Ausgabewerten der abgegebenen Lösung verglichen.

Da die Kommunikation zwischen StudIP/Vips und VEA über HTTP geschieht, kann diese Schnittstelle auch mit einem normalen Browser direkt benutzt werden. Einen Eindruck von der Browseroberfläche zum Testen und Konfigurieren von VEA zeigt Abbildung 14.2.

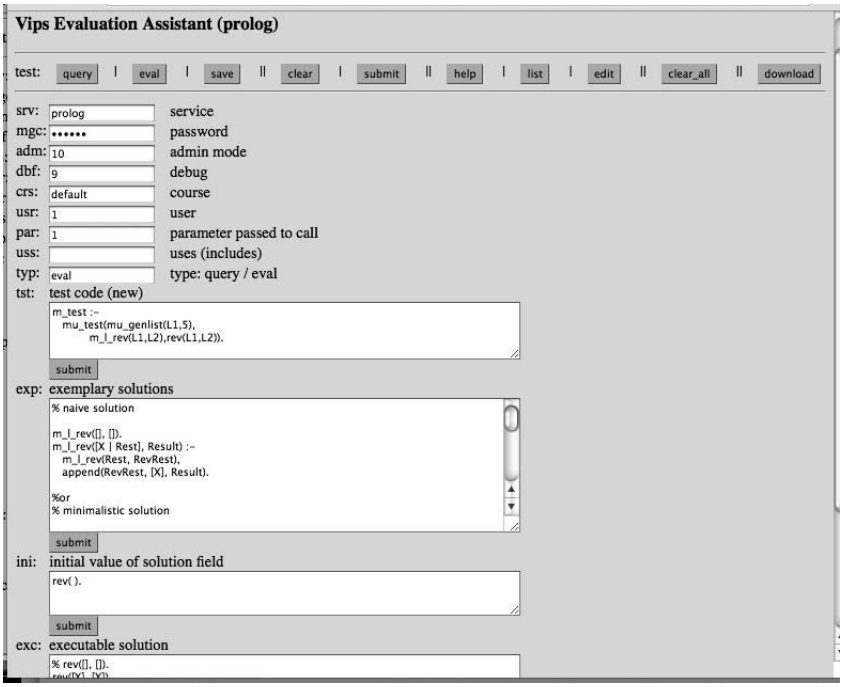


Abbildung 14.2: Browseroberfläche VEA

Dieses Interface gestattet neben der Konfiguration auch den Test aller Funktionen, sowie die Erweiterung des Systems um weitere Programmiersprachen.

14.2.1 Kommunikation mit Vips

Vips stellt für eine Programmierübungsaufgabe neben der Aufgabenstellung eine Reihe von Informationsfeldern zur Verfügung: ein Textfeld für den zu entwickelnden Code, Musterlösungen und den Aufruf der Hauptfunktion. Diese Felder werden zusammen mit einem Auswertungs-Modus an den VEA-Server übertragen.

Zurzeit gibt es zwei Modi für die Kommunikation zwischen Vips und VEA:

1. Alle Datenfelder werden als eigene POST-Felder übertragen (wird zurzeit von Vips benutzt).
2. Die Aufgabenspezifikation wird als XML-Strutur übertragen und nur die abgegebene Lösung wird als eigenes Feld übertragen. Die XML-Struktur entspricht dem Aufgabenaustauschformat, das im Rahmen von eCULT[3] entwickelt wurde [Str+15].

Die wichtigsten Felder, die VEA erwartet, sind in Tabelle 14.1 aufgeführt. Im Fall, dass eine XML-Struktur übertragen wird, werden diese Felder teilweise aus der XML-Struktur gefüllt.

Feldname	Beschreibung	Werte
adm	Administrationsebenen	0|1|2|3|4|5|6
typ	Auswertungsmodus	query | eval
mgc	Password	
srv	Service	Prolog | Lisp ....
qry	Query / Call	
exp	Musterlösungen	
ini	Lösungs-Template	
exc	ausführbare Lösung	
xml	XML-Spezifikation einer Aufgabe	

Tabelle 14.1: Liste der wichtigsten Felder

14.2.2 Implementierte Funktionen

Die Browserschnittstelle erlaubt neben der Konfiguration und der Ausführung von Programmen eine Reihe von Funktionen: *list*, *hash*, *query* und *eval*.

3 http://www.ecult-niedersachsen.de/

list Es wird eine Liste der verfügbaren Services zurückgegeben. Zurzeit ist nur der Service Prolog und Lisp komplett realisiert. Haskel, Oktave und R sind nur rudimentär vorhanden.

hash Es werden vier MD5-Hash-Werte zurückgegeben, die den vier Codenormalisierungen (s. u.) entsprechen. Diese Werte können in Vips benutzt werden, um Ähnlichkeiten zwischen den Lösungen unterschiedlicher Teilnehmer zu erkennen. So kann z. B. festgestellt werden, ob eine Lösung schon als Lösung eines anderen Teilnehmers korrigiert wurde.

query Der Auswertungsmodus $query$ wertet das qry-Feld relativ zum Inhalt des Lösungsfeldes aus. In Prolog enthält qry ein $Ziel$ ($Goal$), d. h. einen logischen Ausdruck, der bewiesen werden soll. Bei funktionalen Sprachen wie Lisp enthält qry einen Funktionsaufruf. Der Ergebnistext (z. B. das Compiler-Protokoll und das Ergebnis der Ausführung, falls das Programm syntaktisch fehlerfrei war) wird an Vips übertragen und ausgegeben. Im Falle einer Fehlermeldung oder einer Warnung wird zusätzlich der mit Zeilennummern versehene Code ausgegeben, um die Fehlersuche zu erleichtern. Damit wird eine rudimentäre Entwicklungsumgebung realisiert, in der Programmentwürfe und Lösungsansätze getestet werden können.

eval Der Auswertungsmodus $eval$ testet die abgegebene Lösung und versucht, eine automatische Bewertung durchzuführen. In diesem Modus liefert VEA im Wesentlichen zwei Zahlen s (score) und v (validity) zwischen 0 und 1 zurück. $v = 1$ bedeutet, dass eine valide Bewertung vorgenommen werden konnte. $s = 0$ bedeutet in diesem Fall, dass die Lösung vollkommen falsch ist und $s = 1$ entsprechend, dass sie vollkommen richtig ist (z. B. wenn sie mit einer Musterlösung übereinstimmt). Eine Überprüfung durch einen Korrekteur sollte in diesem Fall nicht notwendig sein.

Werte von v kleiner 1 deuten darauf hin, dass keine sichere Bewertung vorgenommen werden konnte, der Score s also nur ein Anhaltspunkt für den Korrekteur sein sollte. Es stehen mehrere Testverfahren zur Verfügung. VEA iteriert die beiden folgenden Verfahren jeweils über alle Musterlösungen m.

- Vergleich der Ergebnisse der eingereichten Lösung und der Musterlösung bei verschiedenen Eingaben („black box“-Test). Der Auswertungsmodus $eval$ integriert den Lösungsfeldinhalt und die Musterlösung zusammen mit einem Stückchen Testcode in ein Programm, das die Ergebnisse des Lösungsvorschlags und der Musterlösungen vergleicht. Die Werte für s und v für diesen Testteil werden folgendermaßen berechnet:

$$s_{comp} = \frac{1}{1 + \#compiler\_fehler} \tag{14.1}$$

$$s^m_{eval} = \frac{0.5 * \#pos}{\#pos + \#neg} + 0.5 * s_{comp} \quad (14.2)$$

$$v^m_{eval} = s^m_{eval} \quad (14.3)$$

$\#pos$ ist dabei die Anzahl der Testfälle, für die korrekte Ergebnisse geliefert wurden, und $\#neg$ entsprechend die Fälle, für die falsche Ergebnisse geliefert wurden. Die letzte Gleichung ist in sofern plausibel, als dass ein niedriger Score-Wert nicht bedeuten muss, dass das Programm komplett falsch ist: Ein kleiner Fehler kann etwa eine Reihe von Compiler-Fehlermeldungen evozieren. In diesem Fall sollte also auch ein entsprechend niedriger Validitätswert angesetzt werden. Auf der anderen Seite bedeutet ein hoher Score-Wert, dass das Programmverhalten korrekt ist, was ein relativ sicherer Hinweis darauf sein sollte, dass auch das Programm korrekt ist, solange genügend viele Beispiele getestet wurden.

- Textueller Vergleich des Lösungsvorschlags mit der Musterlösung und der Vorbelegung des Lösungsfeldes. Auf der Basis eines Ähnlichkeitsmaßes[4] sim werden ein Score s_{sim} und ein Validitätswert v_{sim} nach folgender Formel berechnet

$$s' = sim(EXC, EXP_m) * \frac{1 - sim(INI, EXC)}{1 - sim(INI, EXP_m)} \quad (14.4)$$

$$s^m_{sim} = s'^2 * s^m_{max} \quad (14.5)$$

$$v^m_{sim} = max(1 - s', s^m_{sim})^2 \quad (14.6)$$

EXC und INI referieren auf die entsprechenden Feldinhalte (Lösungsvorschlag und Vorbelegung) und EXP_m auf die m-te Musterlösung. s' liefert also 0, falls die Vorbelegung nicht geändert wurde, und 1, falls eine der Musterlösungen eingegeben wurde. Formel 14.5 realisiert eine pessimistische Sichtweise für den Score-Wert und Formel 14.6 entsprechend für den Validitätswert. s^m_{max} ist der vorgegebene maximale Score für die m-te Musterlösung[5]. Kritisch ist natürlich die Wahl der Funktion sim. Das mögliche Spektrum reicht von spezifischen Funktionen für die einzelnen Programmiersprachen, die die syntaktische Struktur berücksichtigen können, bis zu robusten Textvergleichsmethoden unabhängig von der konkreten Program-

4 Für ein Ähnlichkeitsmaß sim muss gelten $0 \leq sim(x, y) \leq sim(x, x) = 1$.

5 Musterlösungen können unterschiedliche Güte haben. Für die erste Musterlösung muss immer gelten $s^m_{max} = 1$.

miersprache. In der gegenwärtigen VEA-Version wird die zweite Möglichkeit benutzt:

$$sim(t_1, t_2) = \frac{2 * similar\_text(t_1, t_2)}{|t_1| + |t_2|} \tag{14.7}$$

Dabei ist $similar\_text$ eine PHP-Funktion, die die übereinstimmenden Zeichen zählt.

- Textueller Vergleich des Lösungsvorschlags und der Musterlösungen auf der Basis verschieden starker Normalisierungsstufen. Diese Stufen sollen an folgendem Beispiel veranschaulicht werden:

```
% Aufgabe: invertieren einer Liste
% naive reverse
rev([], []). % Rekursionsabbruch
rev([X | Rest], Result) :-
       % rekursiver Aufruf für den Rest
       rev(Rest, RevRest),
       % erstes Element hinten anhängen
       append(RevRest, [X], Result).
% fertig
```

  Die wesentlichen Stufen sind folgende:

  - In der Standardstufe (0) (wird grundsätzlich angewendet) werden nur Kommentare am Anfang und Ende[6] der Lösung sowie doppelte Zeilenumbrüche und Blanks am Zeilenende entfernt.[7] Beispiel:

```
rev([], []). % Rekursionsabbruch
rev([X | Rest], Result) :-
       % rekursiver Aufruf für den Rest
       rev(Rest, RevRest),
       % erstes Element hinten anhängen
       append(RevRest, [X], Result).
```

  - In der schwachen Stufe (1) werden zusätzlich Blanks vor und nach Operatoren und Trennzeichen sowie mehrfache Blanks und Kommentare entfernt. Beispiel:

```
rev([],[]).
rev([X|Rest],Result) :-
       rev(Rest,RevRest),
       append(RevRest,[X],Result).
```

6 Meist bezieht sich ein Kommentar vor dem Code nicht auf die Programmstruktur, sondern beschreibt das Problem. Ob ähnliche Überlegungen für Endkommentare gelten, ist unklar.

7 Wie Kommentare zu erkennen sind und welche gelöscht werden, kann konfiguriert werden.

– In der mittleren Stufe (2) werden zusätzlich alle Blanks und Zeilenumbrüche entfernt. Beispiel:

```
rev([],[]).rev([X|Rest],Result):-rev(Rest,RevRest),
                                 append(RevRest,[X],Result).
```

– In der starken Stufe (3) werden zusätzlich alle kleingeschriebenen Symbole auf „a“ und alle großgeschriebenen Symbole auf „V“ reduziert und alle Symbole durch Blanks getrennt, so dass nur die Programmstruktur selbst erhalten bleibt. [8] Beispiel:

```
a ( [ ] , [ ] ) . a ( [ V | V ] , V ) : - a ( V , V ) ,
                                         a ( V , [ V ] , V ) .
```

Für die ersten drei Stufen wird auf Identität der normalisierten Texte getestet. Für die stärkste Stufe wird ein toleranter Textmatch, der maximale gemeinsame Teilstrings berücksichtigt, verwendet. Diese Vergleiche werden u. a. dazu benutzt, um ein textuelles Ergebnis ausgeben zu können:

literally same (Gleichheit auf Stufe 0) Die eingereichte Lösung entspricht wörtlich einer Musterlösung inklusive der Kommentare.

same with similar layout (Gleichheit auf Stufe 1) Die eingereichte Lösung entspricht wörtlich einer Musterlösung inklusive des Layouts. Kommentare unterscheiden sich.

same up to layout (Gleichheit auf Stufe 2) Die eingereichte Lösung entspricht wörtlich einer Musterlösung. Layout und Kommentare unterscheiden sich.

stucturally similar (ähnlich auf Stufe 3) Die eingereichte Lösung entspricht strukturell einer Musterlösung. Funktionsnamen und Variablennamen können sich unterscheiden. Die Abfolge von Definitionen kann sich unterscheiden.

Für diese Fälle können auch feste vorgegebene Werte für den Score s und den Validitätswert v vergeben werden. In diesem Fall überschreiben diese die Werte aus den vorigen Verfahren.

Aus den Werten für die verschiedenen Musterlösungen berechnen sich die endgültigen Werte (v_{sim}, s_{sim}) als

$$(v_{sim}, s_{sim}) \quad = \quad max_m((v^m_{sim}, s^m_{sim})) \tag{14.8}$$

8 Insbesondere die stärkste Normalisierungsstufe sollte abhängig von der Programmiersprache sein. Es ist daran gedacht, für alle Stufen in der Konfiguration parametrisierbare Pattern zu erlauben.

$$(v_{eval}, s_{eval}) \quad = \quad max_m((v^m_{eval}, s^m_{eval})) \tag{14.9}$$

bezüglich der lexikalischen Ordnung.

Für jede Aufgabe kann festgelegt werden, ob nur der Lösungsmengenvergleich, nur der Textvergleich oder eine Kombination von beiden (Default) in die endgültigen Werte von s und v eingehen, die zur Berechnung eines Bewertungsvorschlages benutzt werden. Diese relativ groben heuristischen Verfahren zur Berechnung der Werte s und v haben sich insbesondere für Prolog als durchaus brauchbar erwiesen.

Es ergibt sich folgende grobe Interpretation der Werte s und v:

$v = 1$ Die Lösung konnte sicher bewertet werden. s kann als Faktor zur Berechnung der Punkte für die eingereichte Lösung verwendet werden. Es könnte sich aber um ein Plagiat halten, falls z. B. eine der Musterlösungen eingereicht wurde.

$v \geq 0.7, s \geq 0.5$ Die Lösung scheint partiell richtig zu sein.

$v = 0.5, s = 0.5$ Die Lösung konnte nicht bewertet werden. Vermutlich ist sie falsch.

Dieses Schema ist natürlich sehr grob und muss noch verfeinert werden.

14.2.3 Konfiguration

Zur Konfiguration eines Services sind zwei Dateien notwendig:

config enthält eine Reihe von Attribut-Wert-Paaren zur Konfiguration des Systems. Hier werden u. a. die Auswertungsparameter eingestellt, die Kommentarspezifikation der Programmiersprache festgelegt sowie die erlaubten Funktionen und Symbole aufgelistet.

Eine typische Konfiguration für die Programmiersprache Prolog sieht folgendermaßen aus:

```
# Debug-Flag
dbf=0

# Lösungen, die gleich sind bezüglich dieser
# Normalisierungsebene werden nicht neu evaluiert
hash_result=1

# Comments (line comments, multi line comments
line_comment=%
multi_line_comment=\/\*.*?\*\/
```

```
# Eine einfache Methode Missbrauch zu erschweren
# Alle entsprechenden Symbole werden mit einem Präfix
# versehen
save_pattern=\b[a-z]\w*\b
save_prefix=s__

# check for non-meaningful input
empty_pattern=[^()[\].:]*

# Nur diese Symbole können in Lösungen benutzt werden.
# Für diese Symbole wird der Präfix unterdrückt
allow="""
   member,xfx,
   fx,yfx,xfy,fy,xf,yf,op,current_op, display,
   not,true,fail, trace,
   findall, forall,
   length, nth0, random, mod, max, number, is_list,
   time, use_module,
   dynamic, assert, asserta, assertz,
   retract, abolish, sort, maplist, reverse, permutation
"""
```

exec ist ein Shell-Script und implementiert für die Funktionen $query$ und $eval$ den Compiler-Aufruf. Hier ist darauf zu achten, dass der Ressourcenverbrauch des Prozesses limitiert und die Priorität niedrig eingestellt wird, damit der Prozess die Maschine nicht blockiert. Zudem sind Sicherheitsmaßnahmen gegen den Missbrauch dieser Schnittstelle zu bedenken.

Darüber hinaus muss für die Funktion $eval$ ein Programm zur Verfügung gestellt werden, das die IO-Relation des Lösungsvorschlags überprüft. In den implementierten Services wird dazu die erste Musterlösung benutzt und für eine Reihe von Testfällen die Ergebnisse beider Programme verglichen.

14.3 Ein Beispiel

Als typisches kleines Beispiel für eine Prologaufgabe wählen wir wieder das Problem der Invertierung der Reihenfolge der Listenelemente.

Beispiel: Invertieren einer Liste

Die Aufgabenstellung

> Definiere ein Prädikat $rev(L1, L2)$ das wahr ist falls $L2$ die Elemente von $L1$ in inverser Reihenfolge enthält.

Beispielaufruf

```
rev([1,2,3,4,5,6,7,8,9,0],Result)
```

liefert

```
Result=[0,9,8,7,6,5,4,3,2,1]
```

Drei Musterlösungen

- Naive Variante (benutzt *append*)

```
% naive reverse
m_l_rev([], []).
m_l_rev([X | Rest], Result) :-
   m_l_rev(Rest, RevRest),
   append(RevRest, [X], Result).
```

- Minimalistische Variante (keine Hilfsprädikate oder zusätzliche Argumente)

```
% minimalistic solution
m_l_rev([], []).
m_l_rev([X], [X]).
m_l_rev([F | R1], [L | R4]) :-
   m_l_rev(R1, [L | R2]),
   m_l_rev(R2, R3),
   m_l_rev([F | R3], R4).
```

- Optimale Variante (benutzt ein Akkumulator-Argument)

```
% optimal solution
m_l_rev(L1, L2) :-
   m_l_rev(L1, [], L2).
m_l_rev([], AC, AC).
m_l_rev([X |R], AC, Result) :-
   m_l_rev(R, [X | AC], Result).
```

Testprogramm

```
m_test :-
   mu_test(mu_genlist(L1,5),
           m_l_rev(L1,L2),rev(L1,L2)).
```

mu_test erwartet im ersten Argument einen Testdatengenerator (hier werden durch $mu\_genlist$ Listen generiert) und im zweiten und dritten Argument die zu vergleichenden Lösungen.

Lösungsbeispiele und ihre Bewertung

Werden Musterlösungen eingegeben, werden diese erkannt und entsprechend bewertet. Daher werden im Folgenden nur Fälle diskutiert, die sich von allen Musterlösungen unterscheiden.

- Korrekt, aber geänderte Klauselreihenfolge

```
rev([X], [X]).
rev([F | R1], [L | R4]) :-
   rev(R1, [L | R2]), rev(R2, R3),
   rev([F | R3], R4).
rev([], []).
```

  Bewertung:

  - Strukturell ähnlich zu Musterlösung 2
  - korrekte Ein/Ausgabe-Relation
  - validity=1, score=1

- Korrekte Programmstruktur, Variablennamen falsch

```
rev([X], [X]).
rev([F | R1], [L | R4]) :-
   rev(R1, [L | R3]), rev(R4, R3),
   rev([F | R3], R4).
rev([], []).
```

  Bewertung:

  - Strukturell ähnlich zu Musterlösung 2
  - Gegenbeispiel:

    `rev([5,4],[4,5])`

    sollte wahr sein, ist aber falsch
  - validity=0.7, score=0.7

- Inkorrekt: falsche Definition

```
rev(X, [X|_]).
rev(X, [Y|R]) :-
   rev(X, R).
```

  Bewertung:

  - Keine Ähnlichkeit zu einer Musterlösung
  - Gegenbeispiel:

    ```
    rev([],[[]|B])
    ```

    ist wahr, sollte aber falsch sein.
  - validity=0.5, score=0.5

- Korrekt: andere Reihenfolge der Prädikatsdefinitionen, andere Prädikatsnamen, andere Variablennamen und andere Verwendung der Argumente.

```
rev_a([], Accu, Accu).
rev_a([X | Rest], Result, Accu) :-
   rev_a(Rest, Result, [X | Accu]).

rev(Liste1, Liste2) :- rev(Liste1, Liste2, []).
```

  Bewertung:

  - Strukturell ähnlich zu Musterlösung 3
  - korrekte Ein/Ausgabe-Relation
  - validity=1, score=1

- Lösungstemplate

```
rev( ).
```

  Bewertung:

  - leere Eingabe
  - validity=1, score=0

- Unsinnseingabe

```
dazu habe ich keine Lust!!!!!!!
```

  Bewertung:

  - leere Eingabe
  - validity=1, score=0

14.4 Zusammenfassung und Ausblick

Das Modul Vips (VIrtuelles PrüfungsSystem) übernimmt in StudIP die komplette Verwaltung des Übungsbetriebs einer Veranstaltung. Vips wurde um eine Komponente VEA (Vips-Evaluation-Assistent) zur automatischen Bewertung von Programmieraufgaben erweitert. VEA stellt sowohl semantische (Programmverhalten) als auch syntaktische (Beurteilung des Programmcodes) Verfahren zur Bewertung von Programmieraufgaben zur Verfügung. VEA ist immer noch in einem experimentellen Stadium. Sowohl die Funktionalität, als auch die Kommunikation zwischen Vips und VEA sollen weiterentwickelt werden.

Literatur für dieses Kapitel

[GW13] Helmar Gust und Nadine Werner. „Automatische Bewertung von Übungsaufgaben in VIPS.“ In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. 2013.

[Pey+00] Christoph Peylo u. a. „A Web-based intelligent educational system for PROLOG“. In: *Proceedings of the International Workshop on Adaptive and Intelligent Web-Based Education Systems held in conjunction with ITS 2000 Montreal, Canada.* 2000, S. 85–96.

[Str+15] Sven Strickroth u. a. „ProFormA: An XML-based exchange format for programming tasks“. In: *eleed* 11.1 (2015). URL: https://eleed.campussource.de/archive/11/4138.

15 Der Grader PABS

Lukas Iffländer und Alexander Dallmann

Zusammenfassung
Der modulare Grader PABS unterstützt die Programmiersprache Java sowie andere, auf der Java Virtual Machine (JVM) basierende, Sprachen und wird an der Universität Würzburg entwickelt und eingesetzt. Er dient zur Unterstützung von Vorlesungen und zur Durchführungen von Prüfungen in Praktika im Rahmen der Ausbildung der Würzburger Informatikstudenten und in angelehnten Fächern.

15.1 Einleitung

Der Grader PABS (ProgrammierAufgabenBewertungsSystem) ist eine Eigenentwicklung des Instituts für Informatik der Universität Würzburg. Die Entwicklung begann im Jahr 2010, um das damals im Einsatz befindliche Praktomat zu ersetzen, dessen damalige Version einen umständlichen Upload der Lösungsdateien erforderlich machte und bei hoher Belastung Performance-Probleme generierte, so dass das Bestehen von Tests die in der Ausführungszeit limitiert waren (z. B. Aufgaben bestimmte Algorithmen effizient zu implementieren) zur Glückssache wurde.

Aus diesen Erfahrungen und dem Einsatzweck für das Java-Programmierpraktikum ergaben sich als Kriterien: Skalierbarkeit, eine (der Zielgruppe entsprechend) elegante Möglichkeit für den Upload der Lösungen und die Unterstützung von Sprachen, die auf der Java Virtual Machine [MD97] (JVM) basieren.

Dieses Kapitel basiert in weiten Teilen auf [Iff+15].

15.2 Technologie und Architektur

PABS besteht aus mehreren, miteinander interagierenden Modulen: Einer Webanwendung, dem zugehörigen SVN Repository und mehreren Aktoren, welche in einem Master-Worker-Konzept angeordnet sind.

Die Webanwendung dient dazu, mit den Studierenden zu interagieren. Sie gewährt Zugang zu der Aufgabenstellung (siehe Abbildung 15.1) und zeigt den Studierenden eine Liste der hochgeladenen Lösungen mit der jeweiligen Einstufung der Abgabe (nicht bewertet, abgelehnt, akzeptiert, als zu bewertende Lösung markiert) an. Ist noch keine Bewertung erfolgt (automatisches Bewerten beim Hochladen kann vom Kursadministrator selektiv pro Aufgabe aktiviert werden, siehe Abbildung 15.2), können die Studierenden über die Oberfläche diese auslösen. Bei Aufgaben, deren Bewertung durchgeführt wurde, können die Studierenden das generierte Feedback einsehen, wie in Abbildung 15.3 gezeigt. Abgaben, die als akzeptiert markiert sind, können als zu bewertende Lösung markiert werden. So bleibt es möglich, nach Einreichen einer gültigen Lösung noch Nachbesserungen (z. B. Beseitigung von Debug-Code oder Ergänzung weiterer Kommentare) durchzuführen. Abhängig von der Entscheidung des Kursadministrators kann, wenn keine Lösung markiert wurde, zum Zeitpunkt des Abgabetermins die zuletzt hochgeladene akzeptierte Abgabe gewählt werden. Diese Möglichkeit wurde integriert, da sich gezeigt hat, dass viele Studierende vergessen eine endgültige Lösung zu markieren.

Kursadministratoren können neue Aufgaben erstellen (siehe Abbildung 15.2), für diese Aufgaben sowohl einen Bereitstellungs- als auch einen Abgabezeitpunkt festlegen und entscheiden, ob bestimmte Bewertungen optional oder geheim (Ergebnisse nur für Lehrende sichtbar) sind. Kursadministratoren und Assistenten haben die Möglichkeit, die Bewertungen der Kursteilnehmer zu betrachten, z. B.

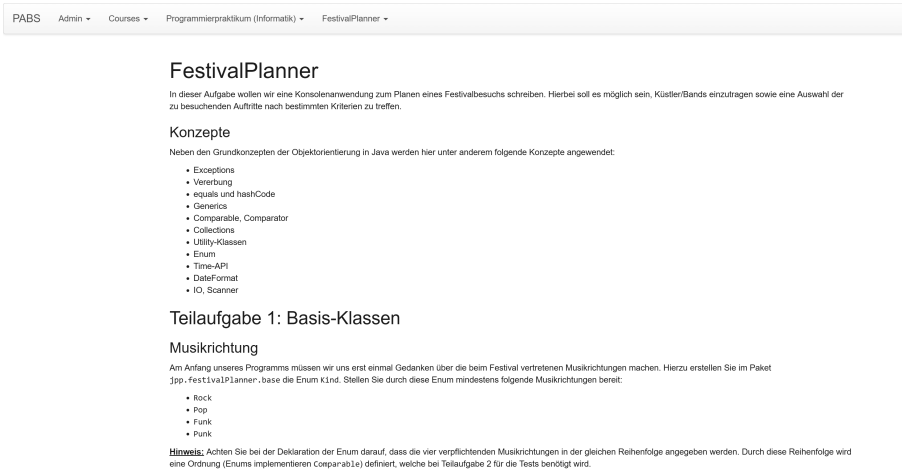


Abbildung 15.1: Die als XWIKI-Code bereitgestellte Aufgabenstellung wird den Studierenden im Webinterface angezeigt.

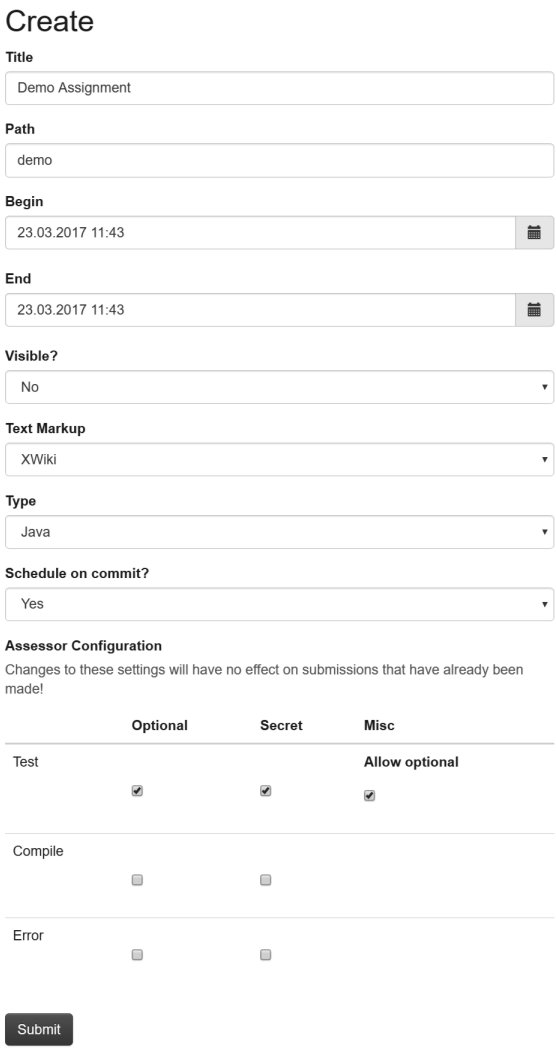


Abbildung 15.2: Kursadministratoren können beim Erstellen einer Aufgabe deren Titel [Title], Pfad im SVN [Path], Start [Begin] und Ende [end] angeben. Die Aufgabe kann zu Beginn auf unsichtbar [Visible?] gestellt werden. Es kann die Programmier- [Type] und Markupsprache [Text Markup] ausgewählt werden. Ob Aufgaben beim Commit bereits geprüft werden lässt sich konfigurieren [Schedule on commit?]. Unter Assessor Configuration lässt sich für Tests, Kompilierschritte und Fehlermeldungen wählen, ob es optionale und/oder versteckte Kriterien gibt.

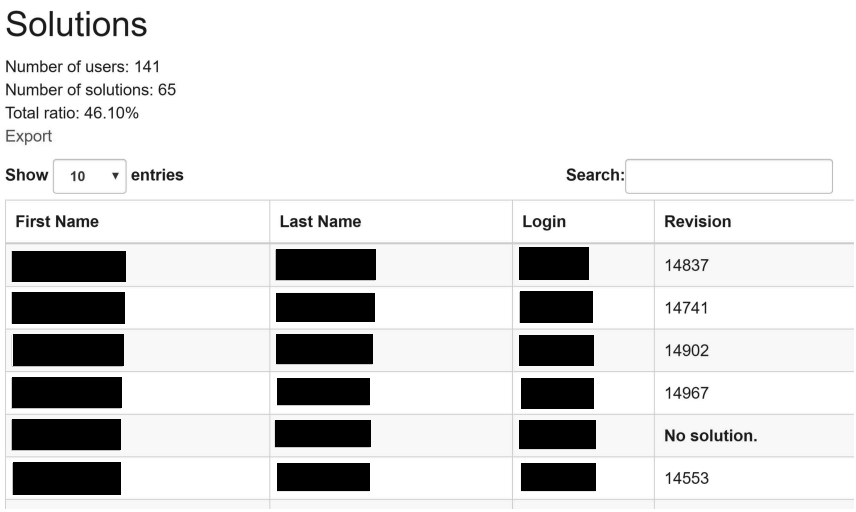
zur manuellen Nachkorrektur oder um bei Problembeschreibungen in den Foren oder per E-Mail die Umstände besser nachvollziehen zu können. Auch ist es möglich, sich ausgeben zu lassen, welche Studierenden bereits eine Lösung markiert haben, wie in Abbildung 15.4 gezeigt.

Zur Einreichung der zu bewertenden Abgaben, aber auch zur Verwaltung der Aufgabenstellung, der Projektkonfiguration und der Bewertungsfälle wird Subversion[1] (SVN) genutzt. Für diese Zwecke wird pro Kurs ein SVN Repository erstellt. Im Repository werden die Konfiguration und die Abgaben in unterschiedliche Pfade getrennt. Studierende erhalten lediglich Zugriff auf ihren eigenen Ordner mit ihren Abgaben, während Tests, Konfiguration, etc. in anderen Ordnern liegen. Dadurch wird verhindert, dass Studierende den Quellcode von Blackboxtests einsehen oder gar manipulieren können.

Das SVN Repository ermöglicht Studierenden und Leherenden gleichermaßen einfachen Zugang zur Arbeitshistorie. Das ermöglicht Studierenden falsch eingeschlagene Lösungswege durch das Zurückspringen auf alte Versionen zu korrigieren. Lehrende auf der anderen Seite können so einfacher Feedback geben und werden bei neu aufgetretenen Problemen durch in SVN integrierte Tools (z. B. diff) unterstützt. Gleichzeitig ist es einfacher möglich Plagiate zu erkennen, da vor allem die Unterscheidung zwischen Plagiiertem und Plagiator erleichtert wird (letzterer hat in der Regel deutlich weniger Commits). Gegenüber der Speicherung der vollständigen Abgaben (z. B. in Datenbanken) hat die Speicherung in

Abbildung 15.3: Studierende können bestandene und nicht bestandene Tests einsehen. Zu letzteren wird das Feedback der TestSuite (hier JUnit) angezeigt.

1 https://subversion.apache.org



Solutions

Number of users: 141
Number of solutions: 65
Total ratio: 46.10%
Export

Show 10 entries Search:

First Name	Last Name	Login	Revision
[redacted]	[redacted]	[redacted]	14837
[redacted]	[redacted]	[redacted]	14741
[redacted]	[redacted]	[redacted]	14902
[redacted]	[redacted]	[redacted]	14967
[redacted]	[redacted]	[redacted]	No solution.
[redacted]	[redacted]	[redacted]	14553

Abbildung 15.4: Betreuer können einsehen, welche und wie viele Studierende bereits Lösungen zur Abgabe markiert haben

einem SVN Repository technische Vorteile. Ein SVN Repository ist äußerst kompakt. Selbst Kurse mit vielen Studierenden und tausenden Commits sind selten größer als 100 MB. Weiterhin kann das Repository unabhängig von der Webanwendung genutzt werden, was ein Weiterarbeiten bei Ausfällen dieser ermöglicht. Auch ist den Studierenden und Betreuern so die Wahl der IDE oder gar der Verzicht auf eine solche freigestellt. Eine Diskussion, ob nun Eclipse oder IntelliJ eingesetzt werden soll, wird so elegant umgangen. Gleichzeitig verbleibt den Kursadministratoren trotzdem die Möglichkeit, als Gerüst einen vorbereiten Workspace einer Entwicklungsumgebung anzubieten. SVN fordert zwar eine gewisse Einarbeitung, unter Betrachtung des Einsatzszenarios für Studierende der Informatik und verwandte Fachrichtungen ist der Erwartungshorizont angemessen, dass Studierende sich selbstständig in Tools wie SVN hineinarbeiten können. Die Lernkurve wird dadurch etwas vereinfacht, dass es einen Schalter gibt, um bei Konflikten einfach das Repository zurück zu setzen, so dass Studierende, die nur an einem Gerät arbeiten, theoretisch alleine mit der Nutzung des Commit-Befehls auskommen können.

Die Generierung des Feedbacks kann abhängig von der Anzahl der Tests und der Codequalität eine zeit- und ressourcenaufwändige Aufgabe sein. Insbesondere komplexere Tests (z. B. unter Nutzung von Java-Reflection) können mehrere Sekunden beanspruchen. Um den Prozess der Feedbackgenerierung bei einer wachsenden Nutzerzahl zu unterstützen, nutzt PABS sog. Agenten, die mit der Web-

anwendung über das Netzwerk kommunizieren. Agenten können auf separaten Hosts ausgeführt werden und unabhängig voneinander aktiviert und deaktiviert werden. Während des Betriebs befindet sich ein Agent permanent in Kommunikation mit einer Warteschlange, die von der Webanwendung mit den zu bewerteten Abgaben befüllt wird. Dies ist in Abbildung 15.5 dargestellt. Eine in die Warteschlange eingestellte Abgabe wird an den ersten freien Agent weitergegeben, der diese in der Folge auswertet. Sobald das Feedback generiert wurde, wird dies an die Webanwendung zurück gesendet und der Warteschlange wird gemeldet, dass der Agent seine Arbeit abgeschlossen hat und wieder verfügbar ist.

Die Kommunikation zwischen Webanwendung und Agenten ist mittels Akka[2] realisiert. Zuerst wird ein Actor gestartet, der die Warteschlange verwaltet. Anschließend wird ein weiterer Actor gestartet, der Abgaben an die Warteschlange senden kann. Weiterhin wird eine beliebige Anzahl von Agent-Actors gestartert und beim Warteschlangen-Actor registriert. Sobald diese Pipline initialisiert wurde, kann damit begonnnen werden, Abgaben zu verarbeiten. Es ist sinnvoll anzumerken, dass das Feedback direkt an den Actor, der die Aufgabe in die Warteschlange eingestellt hat, weitergegeben und die Warteschlange so umgangen wird.

Der Agent nutzt Gradle[3] [MB11], um die Lösung der Studierenden zu kompilieren, sowie Tests und andere Analysetools auszuführen. Währenddessen sammelt ein spezialisiertes Plugin Informationen über den Build und stellt die Informationen dem Agent zur Verfügung. Der Agent beobachtet so den Build-Prozess und terminiert diesen wenn nötig (z. B. Endlosschleifen im abgegebenen Code). Nach einer erfolgreichen Ausführung des Builds erstellt der Agent die Feedbackdaten aus den Build-Informationen und sendet diese an die Webanwendung zurück.

15.3 Aufgabenstruktur

PABS baut auf dem Konzept eines Kurses auf. Ein Kurs besteht aus einer beliebigen Anzahl von Aufgaben und hat eine Anzahl an Studierenden, die in den Kurs eingeschrieben sind. Jede Aufgabe hat ein Zeitfenster, bis wann sie gelöst sein muss, um die eingereichte Lösung als gültig markieren zu können. Nachdem diese Zeit abgelaufen ist, kann die Aufgabe zwar weiter bearbeitet werden und Abgaben erhalten weiterhin Feeback, aber es ist nicht mehr möglich, eine endgültige Lösung abzugeben. Für jede Aufgabe kann vom Lehrenden gewählt werden, was nötig ist, damit die Abgabe vom System akzeptiert wird. So kann das Beste-

2 http://www.akka.io
3 http://www.gradle.org

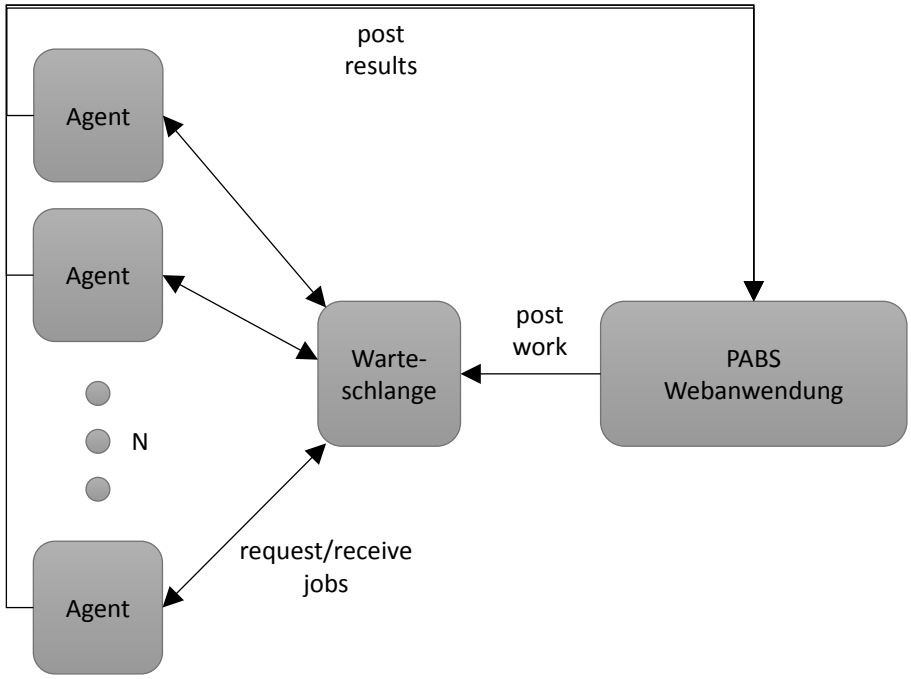


Abbildung 15.5: Zeigt die asynchrone Kommunikation zwischen der PABS Webanwendung und ihren Agenten durch eine Warteschlange. Zuerst fügt die Webanwendung eine Abgabe in die Warteschlange ein. Ein freier Worker-Agent fragt anschließend nach Arbeit. Ist der Agent mit seiner Arbeit fertig, sendet er die Ergebnisse an die Webanwendung zurück und informiert das Warteschlangensystem, dass die Abgabe verarbeitet wurde.

hen aller Tests verlangt werden, aber auch können bestimmte Tests als optional markiert werden oder sogar versteckt werden, so dass zwar die Lehrenden die Ergebnisse sehen, die Studierenden aber gezwungen werden selbst zu testen. Ggf. kann PABS auch so konfiguriert werden, dass es lediglich prüft, ob der eingereichte Quellcode überhaupt kompiliert und so als Filter eingesetzt werden kann, um Aufgaben, die syntaktisch falsch sind, auszusortieren.

Im Folgenden werden Teile des Aufbaus des SVN Repositories beschrieben. Das Verzeichnis ist zur Veranschaulichung in Abbildung 15.6 als Baumstruktur dargestellt.

Die Aufgaben befinden sich in je einem Unterverzeichnis des Verzeichnisses „assignments“, welches auf oberster Ebene des SVN Repositories zu finden ist.

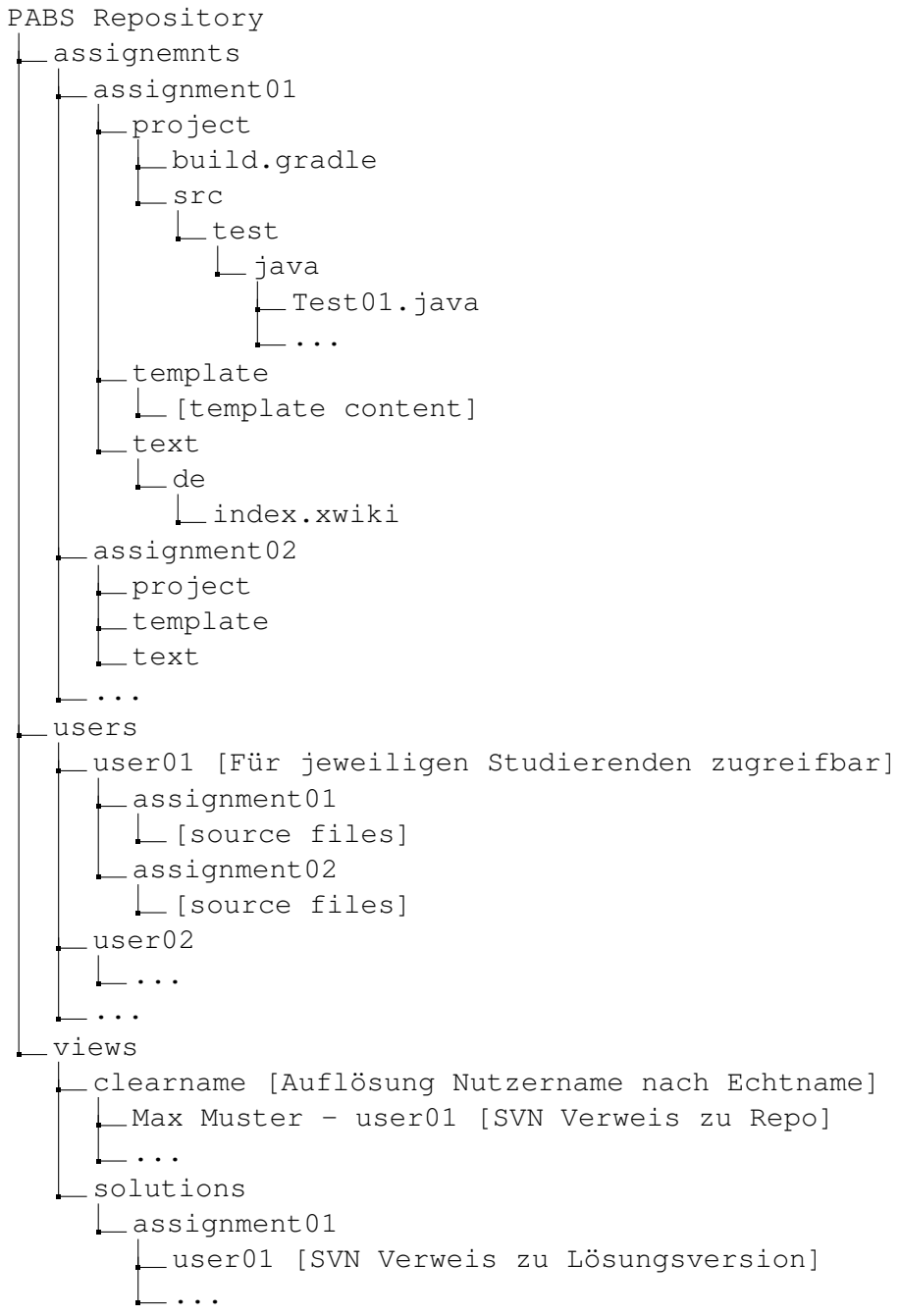


Abbildung 15.6: Verzeichnisstruktur des PABS SVN Repositories

Im Ordner jeder Aufgabe finden sich die Ordner „text“, „template“ und „project“. Im Ordner „text“ findet sich eine XWiki-Datei[4], welche die Aufgabenstellung enthält. Im Ordner „template“ kann ein Musterverzeichnis angelegt werden, das in den Aufgabenverzeichnissen der Studierenden, die sich einschreiben, automatisch erstellt wird. So ist es möglich, Bibliotheken mit auszuliefern, die für ein lokales Testen benötigt werden. Einzelne Anfängerkurse liefern sogar vollständige Workspaces inklusive Konfiguration aus, um so die Einstiegsschwelle in die Programmierung zu reduzieren. Die für das Feedback relevanten Daten finden sich im Ordner „project“. Dort findet sich zuerst eine Konfigurationsdatei für Gradle, in der man die zu kompilierenden Pfade setzen kann. Hier lassen sich auch weitere Bibliotheken oder Plugins ergänzen. Die Tests finden sich anschließend in einem, in der Konfigurationsdatei spezifizierten, Unterordner. Die Tests lassen sich einfach in die Gruppen `required`, `optional` und `hidden` kategorisieren. Finden sich im Pfadnamen die Zeichenfolgen „optional“ oder „hidden“, werden die Tests entsprechend klassifiziert. Alle anderen Tests sind immer `required`. Ausnahmen bestehen, wenn versteckte und/oder optionale Tests in der Weboberfläche für die Aufgabe deaktiviert wurden, dann werden auch diese als `required` gesetzt. Diese einfache Struktur ermöglicht es, zusammen mit der Konfigurationsdatei auch Anforderungen umzusetzen, die über das einfache Abarbeiten von Unit-Tests hinausgehen (z. B. Stilprüfung, Ressourcenverbrauchsmessungen).

15.4 Grading-Methoden und Feedbackmöglichkeiten

Gegenwärtig verfügt PABS lediglich über die Möglichkeit, Bewertungsfälle als „bestanden“ oder „nicht bestanden“ zu markieren. Eine Bepunktung muss gegenwärtig noch manuell durchgeführt werden. Zur Bewertung können beliebige Verfahren herangezogen werden, solange sie auf der JVM laufen und in Gradle konfiguriert werden können.

Das Feedback findet in textueller Form statt. Dazu werden den Studierenden die bestandenen und nicht bestandenen Bewertungsfälle aufgelistet, sowie bei letzteren noch die Auswertung, warum der Fall nicht bestanden wurde. Bestandene und nicht bestandene Bewertungsfälle sind farblich entsprechend hervorgehoben.

Eine Plagiatserkennung ist derzeit als externes Modul an PABS angebunden, das manuell ausgelöst werden muss. Das Tool ist mit der Dateistruktur der PABS-Abgaben vertraut und kann so eine einfache Zuordnung der Plagiate ermöglichen.

4 http://www.xwiki.org

Nach dem Abschluss der Plagiatsprüfung wird eine Weboberfläche geboten, in der Paarungen ähnlicher Klassen aufgelistet werden. Die Similaritäten werden wie in Abbildung 15.7 graphisch aufbereitet dargestellt, so dass einfach zu erkennen ist, welche Teile übernommen wurden. Für die Erkennung werden unterschiedliche Methoden angewandt. Unter anderem werden durch Tokenizing [BTZ07] und die Anwendung von Metriken [Wha90] die beliebten Methoden der Umbenennung von Variablen oder der Umstellung der Methodenreihenfolge erkannt. Gegenwärtig wird das Modul um eine Erkennung von Plagiaten in den in Java 8 neu eingeführten Lambdas erweitert. Das Tool kann auch unabhängig von PABS genutzt werden, verliert dann aber einige Komfortfunktionen.

Bisher findet die Feedbackerstellung und die Bewertung nur im PABS-System direkt statt. Eine Integration für E-Learning-Plattformen wird angestrebt.

15.5 Bisheriger Einsatz

PABS wurde ursprünglich primär für die Unterstützung des Java-Programmierpraktikums für Studierende der Informatik (Bachelor Informatik, Bachelor Luft- und Raumfahrtinformatik und Lehramt) an der Universität Würzburg eingeführt. Die aktuelle Major Revision PABS 3 wurde zu diesem Zweck inzwischen sechsmal erfolgreich angewendet. Während zu Beginn etwa 60 Kursteilnehmer zu betreuen waren, ist die Zahl der Teilnehmer pro Semester inzwischen auf etwa 150 gewachsen.

Nach ersten erfolgreichen Tests wurde das Programm auch für das etwas einfacher ausgerichtete Praktikum für die Bachelorstudiengänge Wirtschaftsinformatik, Wirtschaftsmathematik, Computational Mathematics und Mensch-Computer-Systeme eingeführt. Hier finden sich pro Semester inzwischen etwa 200 Studierende.

Trotz seiner ursprünglichen Orientierung an den Praktika wird PABS in immer mehr Vorlesungen unterstützend zum Übungsbetrieb eingesetzt. Besonders sind hier die Vorlesungen Algorithmen und Datenstrukturen (ca. 400 Teilnehmer), Grundlagen der Programmierung (ca. 170 Teilnehmer) und Informatik für Hörer anderer Fakultäten (ca. 90 Teilnehmer) zu nennen. Zuletzt wurde PABS auch in dem Vorkurs für neue Studierende (ca. 130 Teilnehmer) eingesetzt. In diesem wird inzwischen auch der Umgang mit dem SVN erläutert. Für weitere Veranstaltungen wird der Einsatz geplant oder evaluiert.

Betrachtet man die Kennzahl Studierende mal belegter Kurse, so wird derzeit im Sommersemester ein Wert von etwa 570 und im Wintersemester von etwa 1.140 erreicht. Die Schwankung ist dadurch erklärbar, dass die oben genannten

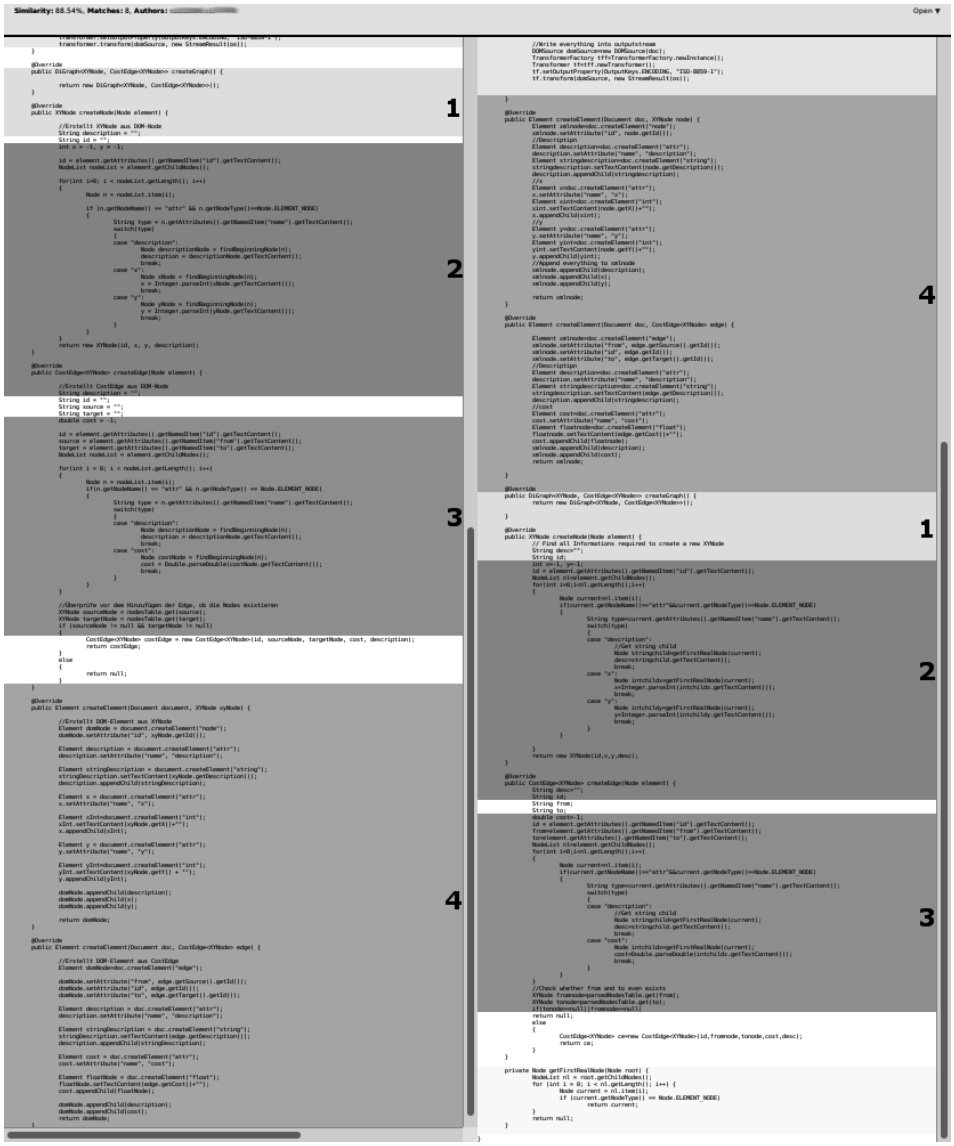


Abbildung 15.7: Bei der Plagiatsprüfung werden als ähnlich erkannte Bereiche in einer Splitansicht mit gleichen Farben/Zahlen hervorgehoben, so dass Plagiate einfach und schnell erkannt werden können.
Im Beispiel hat das Programm zwei Abgaben zu 88% als übereinstimmend erkannt. Es ist zu erkennen, dass zur Verschleierung des Plagiates die Reihenfolge der Blöcke geändert wurde (der letzte Block auf der linken Seite wurde nach vorne gezogen), sowie teilweise Leerzeilen entfernt wurden.

großen Vorlesungen nur im Winter angeboten werden. Derzeit sind in PABS 1.738 Nutzer eingeschrieben, die zum großen Teil mehrere Kurse belegen oder belegt haben.

15.6 Verfügbarkeit und Ausblick

> Wer die geschützten Testfälle knacken kann, dem gebührt es auch, das Praktikum zu bestehen.
>
> – Prof. Dr. Jürgen Wolff von Gudenberg

PABS ist derzeit noch als Closed Source in Entwicklung, da noch einige Sicherheitsaspekte geklärt bzw. verbessert werden müssen. Unter den wissenschaftlichen Hilfskräften, die die Praktika betreuen, hat sich ein kleiner Wettbewerb entwickelt, um durch Exploits mit Studentenrechten an die Quellen der Testfälle zu kommen. Zwar ist dem obigen Zitat des Dozenten, der lange Jahre das Programmierpraktikum organisiert und verantwortet hat, zuzustimmen. Dies gilt allerdings nur für den ersten Studierenden, der diese Schranke umgeht, während dann die weiteren die automatischen Tests durch Code umgehen können, der exakt die Lösungen der getesteten Fälle zurückliefert. Sobald die bekannten Sicherheitslücken beseitigt sind, ist geplant, das Programm unter einer OpenSource-Lizenz über eine öffentlich zugängliche Platform zur Verfügung zu stellen. An der Entwicklung interessierte Personen können aber bereits jetzt einen Zugang zu den Repositories erhalten.

In Zukunft sind viele Weiterentwicklungen für PABS geplant, die sich großteils in den Bereichen der Unterstützung weiterer Funktionen, die Vereinfachung der Nutzung und der Erhöhung der Sicherheit ansiedeln.

Während bisher primär Java implementiert wurde, soll die Unterstützung von anderen Sprachen auf Basis der JVM erweitert werden. Während für Scala und Groovy bereits Musterbeispiele vorliegen, soll demnächst der JVM Python-Interpreter Jython[5] integriert werden, da geplant ist, die Sprache Python aufgrund ihrer einfachen Syntax bei Vorlesungen für Hörer außerhalb der Informatik einzusetzen. Auch die Unterstützung von C und C++ ist angedacht, um in Vorlesungen, die hardwarenahes Programmieren lehren, eine Feedbackmöglichkeit zu schaffen. Besonders bei diesen beiden Sprachen müssen allerdings zusätzliche Sicherheitsmaßnahmen ergriffen werden, da sie aufgrund ihrer Hardwarenähe zusätzliche Angriffsvektoren bieten.

5 http://www.python.org

Aus den genannten Sicherheitsgründen, aber auch zur Verbesserung der Elastizität [HKR13] sollen die die Bewertung ausführenden Agenten auf eine Containerarchitektur umgestellt werden. Dadurch wird eine Kapselung erreicht, so dass Rechte-Exploits des ausgeführten Codes keinen Schaden außerhalb anrichten können. Weiterhin soll so innerhalb der Cloud Environments der Universität ermöglicht werden, auf Lastspitzen (z. B. während der Praktikumstutorien) flexibel durch das Ausbringen zusätzlicher Agenten zu reagieren.

Die Verwendung von PABS soll vor allem für die Lehrenden vereinfacht werden. Dazu soll unter anderem für die Bearbeitung der Aufgabenstellungen zusätzlich zum Upload des XWIKI-Quellcodes über das SVN direkt über eine Weboberfläche ermöglicht werden, die einen WYSIWYG-Editor (What You See Is What You Get) bereitstellt. Auch sollen Lehrende nicht mehr manuell die Plagiaterkennung anstoßen müssen, sondern nach Abgabenende automatisch über solche informiert werden. Zur Reduzierung von Redundanzen soll der Bewertungsexport in Moodle realisiert werden.

Auch das Bewertungskonzept in bestanden und nicht bestanden soll um die Möglichkeit der Punktevergabe erweitert werden. Dazu ist eine Annotierung der Testfälle angestrebt, so dass jedem Testfall eine Punktezahl zugeordnet werden kann.

15.7 Abschluss

Auf den vergangenen Seiten wurde der Grader PABS vorgestellt. Der Grader ist auf Skalierbarkeit ausgelegt und hat eine stetig wachsende Nutzerbasis. Die Funktionen befinden sich weiter in der Entwicklung und werden regelmäßig erweitert.

Literatur für dieses Kapitel

[BTZ07] Steven Burrows, Seyed M.M. Tahaghoghi und Justin Zobel. „Efficient plagiarism detection for large code repositories“. In: *Software-Practice and Experience* 37.2 (2007), S. 151–176.

[HKR13] Nikolas Roman Herbst, Samuel Kounev und Ralf Reussner. „Elasticity in cloud computing: What it is, and what it is not“. In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. 2013, S. 23–27.

[Iff+15] Lukas Iffländer u. a. „PABS – a Programming Assignment Feedback System“. In: *Workshop „Automatische Bewertung von Programmierauf-*

gaben“ (ABP 2015). Bd. 1496. CEUR Workshop Proceedings. Nov. 2015.

[MB11] Matthew McCullough und Tim Berglund. *Building and Testing with Gradle*. O’Reilly Media, Inc., 2011.

[MD97] Jon Meyer und Troy Downing. *Java virtual machine*. O’Reilly & Associates, Inc., 1997.

[Wha90] Geoff Whale. „Software metrics and plagiarism detection“. In: *Journal of Systems and Software* 13.2 (1990), S. 131–138.

16 Der Grader ASB

Britta Herres, Rainer Oechsle und David Schuster

Zusammenfassung
Im Fachbereich Informatik der Hochschule Trier wird seit 2006 das ASB-System (Automatische Softwarebewertung) entwickelt und eingesetzt. Die webbasierte Client-Server-Anwendung ermöglicht die automatische Bewertung von studentischen Lösungen zu Programmieraufgaben innerhalb eines festgelegten Zeitraums. In diesem Beitrag wird das ASB-System aus technischer Sicht beschrieben. Das ASB-System ist ein Komponentensystem, dessen Infrastruktur von einem verteilten Ereignisbus gebildet wird, über den die Komponenten innerhalb eines Rechners, aber auch über Rechnergrenzen hinweg, kommunizieren können. Die einzelnen Bestandteile bilden ein Schichtensystem bestehend aus Ausführungsumgebungen, Bewertungsmaßnahmen (Plugins) und Konfigurationseinstellungen für die Bewertungsmaßnahmen. Es werden statische und dynamische Bewertungsmaßnahmen unterstützt. Eine besondere Herausforderung stellt die dynamische Bewertung von Android-Apps dar. Auf die Integration der Bewertungsmaßnahme für Android-Apps in das ASB-System wird in diesem Beitrag speziell eingegangen.

16.1 Einleitung

Das System zur automatischen Bewertung von Software (ASB) ist eine webbasierte Client-Server-Anwendung, die seit 2006 im Fachbereich Informatik der Hochschule Trier entwickelt und eingesetzt wird. Ziel dieses Systems ist die automatisierte Bewertung studentischer Lösungen von Programmieraufgaben. Durch die Vielzahl von Studierenden in programmierlastigen Lehrveranstaltungen soll das ASB-System den Aufwand des händischen Korrigierens von Lösungen zu Programmieraufgaben deutlich reduzieren und Studierenden sofortiges Feedback über die Korrektheit ihrer erstellten Lösungen geben.

Im ASB-System werden Lehrveranstaltungen und die dazugehörigen Programmieraufgaben abgebildet. Zu Aufgaben können Lösungen eingereicht werden. Eine Lösung besteht in der Regel aus einer oder mehreren Quellcodedateien, die als ZIP-Datei gepackt und hochgeladen werden. Für Java-Programme muss eine ZIP-Datei so strukturiert sein, dass sich nach dem Entpacken eine Verzeichnisstruktur ergibt, die den Packages entspricht, in denen sich die Klassen aus den Quellcodedateien befinden. Eine eingereichte Lösung wird in der Regel mehreren Bewertungsmaßnahmen unterzogen, wobei man zwischen statischen und dynamischen Bewertungsmaßnahmen unterscheiden kann. Bei statischen Bewertungsmaßnahmen wird der Quellcode der eingereichten Programme analysiert. Bei dynamischen Bewertungsmaßnahmen wird das zu bewertende Programm ausgeführt und sein Verhalten geprüft.

Es lassen sich drei Nutzergruppen des ASB-Systems unterscheiden:

- eine eher technisch orientierte Nutzergruppe: Personen dieser Gruppe legen die Bewertungsmaßnahmen auf dem ASB-System an, konfigurieren, ändern und löschen diese gegebenenfalls auch wieder, erzeugen Lehrveranstaltungen und Aufgaben, ordnen existierende Bewertungsmaßnahmen den Lehrveranstaltungen und Aufgaben zu und legen für die Aufgaben Zeiträume fest, in denen Lösungen eingereicht werden können;
- die Gruppe der Lehrenden: die Lehrenden sehen sich die von den Studierenden eingereichten Lösungen und deren Bewertungen an;
- die Gruppe der Studierenden: die Studierenden laden ihre Lösungen auf den ASB-Server, betrachten ihre Bewertungsergebnisse und laden im Fehlerfall ihre veränderten Lösungen erneut hoch.

Das ASB-System wurde im Laufe der Jahre ständig weiterentwickelt, zum einen, weil neue Anforderungen dazu gekommen sind, zum anderen, weil die Architektur des Systems und die technologische Basis geändert wurden. Die Software wurde von Assistenten, Assistentinnen und Studierenden entwickelt. Eine frühe Version des ASB-Systems wurde in [Mor+07] beschrieben. Momentan arbeiten wir mit der Version 4, auf der dieser Beitrag basiert.

In diesem Kapitel wird das ASB-System aus technischer Sicht dargestellt, wobei wir insbesondere auf die Implementierung zur Bewertung von Android-Apps eingehen werden. Der Einsatz des ASB-Systems im Hochschulkontext ist in Kapitel 7 beschrieben. Hier haben wir auch erwähnt, dass wir ASB nicht zur vollautomatischen Bewertung, sondern lediglich als Assistenzsystem nutzen, das den Dozenten anzeigt, ob die Bewertungsmaßnahmen alle erfolgreich waren bzw. welche Probleme festgestellt wurden. Wie mit diesen Bewertungsergebnissen umgegangen wird, kann jeder Dozent für sich entscheiden.

16.2 Anforderungen

Basierend auf den Erfahrungen, die wir mit früheren Versionen des ASB-Systems gesammelt haben, und den im Laufe der Jahre aufgekommenen Wünschen der Nutzenden wurden die nachfolgend aufgeführten Anforderungen an die aktuelle Version gestellt. Diese sollten insbesondere die Modularität und Flexibilität des Systems sicherstellen:

- A1: Das System soll Programme bewerten, die in unterschiedlichen Programmiersprachen geschrieben sind.

- A2: Auch die auf die eingereichten Programme angewendeten Bewertungsmaßnahmen sollen in unterschiedlichen Programmiersprachen implementiert werden können.

- A3: Das System soll sowohl statische als auch dynamische Bewertungsmaßnahmen unterstützen.

- A4: Die im Kontext von dynamischen Bewertungsmaßnahmen ausgeführten Programme, die zur Bewertung eingereicht wurden, sollen beschränkte Zugriffsrechte besitzen. Welche konkreten Rechte solche Programme haben, soll von der jeweiligen Bewertungsmaßnahme abhängig sein. So soll es beispielsweise den eingereichten Programmen in den allermeisten Fällen nicht möglich sein, Netzverbindungen aufzubauen und darüber Daten zu senden. Für Bewertungsmaßnahmen von Programmen, die einen Client einer Client-Server-Anwendung realisieren, soll dies aber sehr wohl möglich sein. Allerdings sollen diese Programme dann nur eine Verbindung zum Testprogramm aufbauen können, welches die Serverseite nachahmt und dabei überprüft, ob sich das zu bewertende Client-Programm so verhält, wie es soll.

- A5: Bereits vorhandene Werkzeuge wie zum Beipsiel Checkstyle und Findbugs sollen als Bewertungsmaßnahmen in das System integriert werden können.

- A6: Die Bewertungsmaßnahmen sollen individuell je nach Anwendungsfall konfigurierbar sein. Damit ist zum Beispiel gemeint, dass für unterschiedliche Lehrveranstaltungen unterschiedliche Kodierungskonventionen von der Bewertungsmaßnahme Checkstyle überprüft werden.

- A7: Die Installation neuer Bewertungsmaßnahmen soll zur Laufzeit erfolgen. Ferner sollen bestehenden Bewertungsmaßnahmen zur Laufzeit verändert und gelöscht werden können.

16.3 Grundsätzliche Entwurfsentscheidungen

Wie die Vorgängerversionen ist auch das aktuelle ASB-System eine webbasierte Anwendung. Das heißt, die Nutzenden des Systems können über einen Browser auf den ASB-Server zugreifen. Die Nutzung des ASB-Systems ist von überall aus möglich. Allerdings können sich nur registrierte Personen durch Angabe einer Kennung und eines Passworts am System anmelden, wobei die Nutzenden hierfür in der Regel ihre Kennung und ihr Passwort der Hochschule Trier verwenden, deren Korrektheit bei jedem Login mit Hilfe der Server des Rechenzentrums der Hochschule Trier überprüft wird.

Um das ASB-System mit einer möglichst großen Modularität und Flexibilität auszustatten, wurde es als Komponenten-Framework mit austauschbaren Komponenten realisiert [Oec13]. Insbesondere Anforderung A7 legt eine solche Entwurfsentscheidung nahe. Das Komponenten-Framework, das wir als ASB-Kern bezeichnen, besteht im Wesentlichen aus einem selbst implementierten Ereignisbus, an den die Komponenten angeschlossen werden und über den die Komponenten miteinander kommunizieren können. Dieser Ereignisbus wurde nicht nur auf der Serverseite realisiert, sondern auf die Clients ausgedehnt. Das heißt, dass die Ereignisbusse der Clients mit dem Ereignisbus des Servers gekoppelt sind. Eine direkte Kommunikation zwischen den Ereignisbussen der Clients ist nicht möglich. Da es sich um eine webbasierte Anwendung handelt, bedeutet dies zum einen, dass die Browser nicht nur reine HTML-Seiten darstellen, sondern dass auch in den Browsern Programmcode ausgeführt werden muss. Zum anderen bedeutet eine webbasierte Anwendung, dass zur Kommunikation zwischen Client und Server das HTTP-Protokoll benutzt wird, welches ursprünglich ein reines Request-Response-Protokoll war. Damit ist gemeint, dass die Initiative für eine Kommunikation immer vom Client ausgeht und der Server nur auf Anfragen eines Clients antwortet, nicht aber von sich aus Nachrichten zu einem Client senden kann. Dies hat gewisse Nachteile, die im Kontext ASB anhand eines Beispiels erläutert werden sollen: Nach dem Hochladen eines Programms werden die Bewertungsmaßahmen angestoßen, deren Ausführung auch länger dauern kann. Da der Server das Ergebnis nach Beendigung von einer oder aller Bewertungsmaßnahmen nicht von sich aus an den Client mitteilen kann, muss der Client erst wieder eine Anfrage an den Server senden, um die Ergebnisse anzuzeigen. Dies kann

im einfachsten Fall manuell durch die Nutzerin erfolgen, indem sie immer wieder klickt um die Seite neu zu laden und nachzusehen, ob die Ergebnisse schon verfügbar sind. Es gibt auch unterschiedliche Techniken, die ohne das Zutun der Nutzenden funktionieren. Letztlich handelt es sich dabei aber immer um eine Form des Pollings, was bedeutet, dass der Client immer wieder Anfragen sendet. Durch das Konzept von sogenannten WebSockets, das es seit einigen Jahren gibt, ist es nun inzwischen möglich geworden, dass zwischen Client und Server eine länger dauernde Verbindung eingerichtet wird, über die sowohl der Server als auch der Client zu jeder Zeit von sich aus etwas senden kann. Diese WebSockets werden im ASB-System verwendet, um den Ereignisbus des Servers mit den Ereignisbussen der Clients zu koppeln. Die Grundstruktur unseres ASB-Systems ist somit ein verteiltes Ereignisbussystem, das in Abbildung 16.1 schematisch dargestellt ist. Das komplette System ist ausschließlich in Java programmiert. Clientseitig wird das Webanwendungsframework *Google Web Toolkit (GWT)*[1] verwendet. Dies ermöglicht, dass die grafische Benutzeroberfläche, die vom Browser dargestellt wird, in Java entwickelt werden kann. Ein von GWT mitgelieferter Compiler kompiliert den Java-Quellcode nach JavaScript, der von allen aktuellen Browsern interpretiert werden kann. Für die WebSocket-Kommunikation zwischen den Ereignisbussen werden *JSON*[2]-formatierte Nachrichten benutzt. Damit können Java-Objekte als Zeichenketten serialisiert werden. Auf der Empfängerseite kann die Zeichenkette wieder deserialisiert werden, was bedeutet, dass daraus wieder ein Objekt erzeugt wird, das die gleichen Werte wie das auf der Senderseite serialisierte Ob-

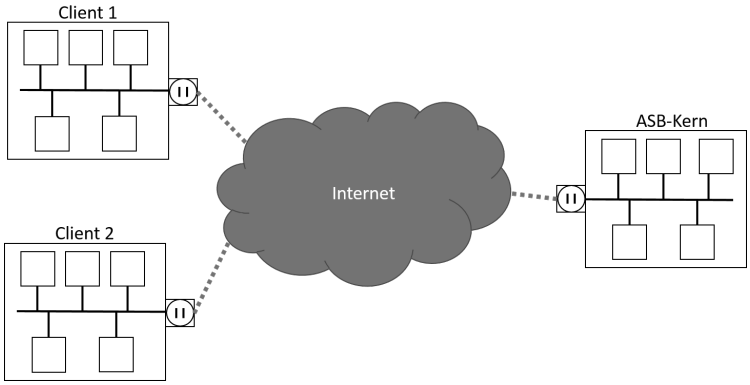


Abbildung 16.1: Client-Server-Kommunikation im ASB-System

1 http://www.gwtproject.org/?hl=de
2 http://www.json.org/json-de.html

jekt besitzt. Umgangssprachlich kann man dies so ausdrücken, dass damit Objekte über das Netz gesendet werden können.

Die Verwendung von GWT, den Ereignisbussen und den WebSockets ermöglicht sowohl die strikte Trennung zwischen Client und Server als auch die vollständige Entwicklung des Systems in der Programmiersprache Java. Somit ist es möglich, client- und serverseitig mit den gleichen Java-Objekten zu arbeiten und diese als JSON-formatierte Zeichenketten zu übertragen.

Im Folgenden gehen wir auf die Architektur des ASB-Servers ein.

16.4 Architektur des ASB-Servers

Wie zuvor erläutert wurde, handelt es sich beim ASB-System um ein Komponenten-Framework mit austauschbaren Komponenten. Das Framework wird ASB-Kern genannt. Die restlichen Teile bilden ein dreischichtiges System, das sich aus den Anforderungen ergibt:

- Schicht 1: Ausführungsumgebungen: Aufgrund der Anforderungen A1 und A2, dass sowohl die zu bewertenden Programme als auch die Bewertungsmaßnahmen in unterschiedlichen Programmiersprachen geschrieben sein können, stellt das ASB-System Ausführungsumgebungen zur Verfügung, die die Ausführung von Programmen unterschiedlicher Programmiersprachen erlauben. Auch die Beschränkung der Rechte, die die eingereichten Programme während der Ausführung haben, wird durch die Ausführungsumgebungen realisiert (Anforderung A4).

- Schicht 2: Plugins bzw. Bewertungsmaßnahmen: Alle Bewertungsmaßnahmen werden in einheitlicher Weise im System als Plugins verwaltet. Dies gilt sowohl für statische als auch für dynamische Bewertungsmaßnahmen (Anforderung A3). Das ASB-System unterscheidet nicht zwischen statischen und dynamischen Bewertungsmaßnahmen. Durch Programmierung eines kleinen Adapters können bereits existierende Werkzeuge gekapselt und als Bewertungsmaßnahmen in das ASB-System integriert werden (Anforderung A5). Bewertungsmaßnahmen können auch im laufenden Betrieb hinzugefügt, geändert und gelöscht werden (Anforderung A7).

- Schicht 3: Konfigurationen: Ein Plugin allein definiert eine Bewertungsmaßnahme in der Regel nicht vollständig. So müssen die Parameter von Kodierungskonventionen, die von der statischen Bewertungsmaßnahme Checkstyle geprüft werden, in einer Konfigurationsdatei angegeben werden. Zum

Beispiel überprüft Checkstyle, ob nach einer öffnenden Klammer korrekt eingerückt wird. Ob man mit Tabs oder Leerzeichen einrücken soll und gegebenenfalls mit vielen Leerzeichen, wird in einer Konfigurationsdatei festgelegt. Unterschiedliche Dozenten bevorzugen unterschiedliche Konventionen. In einem solchen Fall muss Checkstyle als Plugin nur ein Mal im ASB-System installiert werden. Es kann dann aber mit unterschiedlichen Konfigurationsdateien verwendet werden (Anforderung A6). Da statische und dynamische Bewertungsmaßnahmen einheitlich im System behandelt werden, wurde das Prinzip der Konfiguration auf die dynamischen Bewertungsmaßnahmen übertragen. Für das Testen von Java-Programmen ist die Bewertungsmaßnahme (Schicht 2) ein kleines Rahmenprogramm namens Unit-Test-Runner, das JUnit-Tests im Rahmen des ASB-Systems ausführt. Der konkrete Testcode wird hier als Konfiguration bezeichnet und vom Unit-Test-Runner ausgeführt.

Abbildung 16.2 zeigt den strukturellen Aufbau des ASB-Servers als Schichtensystem. Im Folgenden gehen wir näher auf den ASB-Kern (Schicht 0), die Ausfüh-

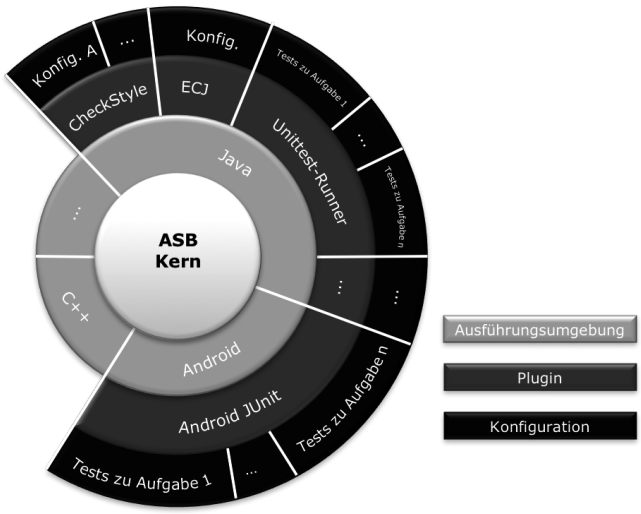


Abbildung 16.2: Struktureller Aufbau des ASB-Systems

rungsumgebungen (Schicht 1), die Plugins (Schicht 2) und die Konfigurationen (Schicht 3) ein.

Der ASB-Kern

Der *ASB-Kern* stellt die Infrastruktur des Serversystems dar. Sie wird durch einen *Ereignisbus* realisiert. Der Ereignisbus stellt Methoden bereit, um *Event*-Objekte (Ereignismeldungen) über den Bus zu senden. Diese Methoden können von den Komponenten des Serversystems benutzt werden, wobei es Methoden gibt zum Senden einer Nachricht nur auf den lokalen Bus des Servers, aber auch zum Senden auf den Serverbus mit Weitergabe an die Busse der Clients. Trifft über eine WebSocket-Verbindung eine Anfrage eines Clients in Form eines Event-Objekts auf dem Server ein, wird dieses ebenfalls über den Serverbus verteilt. Um auf den Bus gesendete Nachrichten empfangen zu können, müssen sich sogenannte *Ereignisbehandler* am Bus registrieren. Um zu entscheiden, welcher Behandler welche Nachrichtenarten verarbeitet, muss dieser eine entsprechend annotierte Methode bereitstellen, die das empfangene Ereignis als Parameter enthält. Wie schon zuvor beschrieben wurde, haben WebSockets den Vorteil, dass sie über einen sogenannten *Push*-Mechanismus verfügen. Das heißt, sobald sich eine Information serverseitig geändert hat, beispielsweise das Anlegen einer neuen Aufgabe in einer Lehrveranstaltung, ergreift der Server die Initiative und übermittelt den aktualisierten Datensatz an alle Clients, die von dem entsprechenden Ereignis betroffen sind. In diesem Beispiel beträfe das alle Studierenden, die sich für die betreffende Lehrveranstaltung im ASB-System angemeldet haben und gerade eingeloggt sind.

Der ASB-Kern enthält bereits Ereignisbehandler, um lesende und schreibende Zugriffe auf eine Datenbank durchzuführen. Die Datenbank und ihre Schnittstelle über das Bussystem bilden einen Teil des ASB-Kerns. Darüber hinaus verwaltet der ASB-Kern die im weiteren Verlauf dieses Kapitels vorgestellten Komponenten. Der Kern wurde als Komponenten-Framework implementiert, das ermöglicht, einzelne Komponenten zur Laufzeit zu installieren, zu aktualisieren und zu entfernen. Dadurch kann das ASB-System modular erweitert und angepasst werden.

Die Ausführungsumgebungen

Ausführungsumgebungen (AU) dienen dazu, Programme einer beliebigen Programmiersprache zur Ausführung zu bringen. Im Kontext des ASB-Systems bedeutet dies, dass durch eine Ausführungsumgebung ein Plugin mit der dazu gehörigen Konfiguration gestartet werden kann, um studentische Lösungen sowohl statisch als auch dynamisch bewerten zu können. Die Programme werden in einem eigenen Prozess oder auf einem anderen Rechner (siehe Kapitel 16.5) ausgeführt. Dabei haben sie minimale Ausführungsrechte. Unter anderem wird ihnen

kein Zugriff auf das Dateisystem gewährt. Über sogenannte *Umgebungsparameter* können jedoch Zugriffsrechte definiert werden. Ein Beispiel dafür sind die *Android-Permissions* für die Android-AU. Zu einer Ausführungsumgebung kann es beliebig viele Konfigurationen geben, bei denen die Werte der Umgebungsparameter je nach Kontext variieren. Programme für verteilte Anwendungen dürfen zum Beispiel Socket-Verbindungen aufbauen, wobei die erlaubten Portnummern auf einen bestimmten einstellbaren Bereich beschränkt sind. Programmen zu einer Lehrveranstaltung über grafische Benutzeroberflächen ist im Gegensatz dazu jegliche Netzkommunikation verboten. Weiterhin kann eine Ausführungsumgebung *Ausführungsparameter* verlangen. Innerhalb der Java-AU ist es beispielsweise nötig, die Einstiegsklasse des zu startenden Programms anzugeben. Die Android-AU hingegen verlangt keine Ausführungsparameter. Darüberhinaus definiert eine Ausführungsumgebung das Dateiformat, in dem zu analysierende studentische Lösungen eingereicht werden müssen.

Eine Ausführungsumgebung besitzt eine Programmierschnittstelle mit der zentralen Methode `execute()`, der als Parameter ein *File*-Objekt übergeben wird. Dieses repräsentiert einen Ordner, der alle Dateien enthält, die zur Ausführung eines Programms benötigt werden. Dazu gehören die eingereichten Dateien der studentischen Lösung, das anzuwendende Plugin und deren Konfigurationsdateien sowie zusätzlich benötigte Bibliotheken. In diesen Ordner werden nach durchgeführter Bewertung auch die Bewertungsergebnisse als XML-Datei abgelegt, die den Studierenden in aufbereiteter Form im Browser angezeigt werden. Die Ausführungsumgebung bereitet alle vorliegenden Daten auf, setzt die Parameter und bringt das Programm zur Ausführung. Jede Konfiguration eines Plugins enthält eine Maximalzeit, die die Ausführung des Plugins nicht überschreiten darf. Wird diese Zeit erreicht, wird die Ausführung abgebrochen, indem die von jeder Ausführungsumgebung bereitgestellte Methode `cancel()` automatisch aufgerufen wird. Diese Methode kümmert sich um die korrekte Terminierung der Bewertung und generiert eine entsprechende Fehlermeldung, die dem Studierenden, dessen Programm gerade bewertet wurde, als Feedback angezeigt wird.

Bisher wurden Ausführungsumgebungen für Java, Python, C++ und Android implementiert.

Die Plugins

Programme, die Bewertungsmaßnahmen auf studentischen Lösungen durchführen, werden *Plugins* genannt. Diese werden für Ausführungsumgebungen definiert und durch Konfigurationsdateien konkretisiert. Innerhalb eines Plugins wird

der Ablauf der Bewertungsmaßnahme festgelegt und benötigte Funktionalität wie beispielsweise das Schreiben der Bewertungsergebnisse gekapselt. Durch den modularen Aufbau des ASB-Systems können Plugins über die Benutzeroberfläche sogar zur Laufzeit angelegt, konfiguriert, erweitert und gelöscht werden.

Die wichtigsten Plugins der Java- und Android-Ausführungsumgebung sind:

- *Checkstyle*[3]: Das Plugin Checkstyle untersucht den an das System übertragenen Java-Quellcode auf das Einhalten von Programmierkonventionen. Die Konventionen werden mit Hilfe der dazugehörigen Konfigurationsdateien eingestellt. Dadurch können verschiedenste Prüfungen, wie beispielsweise das korrekte Klammern von Codeblöcken oder das Vorhandensein von doppeltem Code, vorgenommen werden.

- *Eclipse Compiler for Java (ECJ)*[4]: Dieses Plugin ist für die Kompilierung der Java-Dateien verantwortlich. Ein Vorteil des ECJ gegenüber des herkömmlichen Java-Compilers[5] ist die Konfigurierbarkeit. Mit Hilfe einer Konfigurationsdatei lassen sich verschiedene Fehler- bzw. Warnmeldungen an- und ausschalten. Falls beispielsweise ein Parameter einer Methode ungenutzt ist, informiert der ECJ in der aktuellen Konfiguration den Nutzer darüber. Im Gegensatz dazu würde der Java-Compiler von Oracle diese Tatsache ignorieren.

- *JMaat*: Das eigens entwickelte Plugin dient der Plagiatserkennung der eingereichten studentischen Lösungen [Mü07]. Bewertungsmaßnahmen können auf eine einzige Lösung oder auf alle Lösungen zu einer Aufgabe angewendet werden. Die Plagiatserkennung ist eine Maßnahme, die offensichtlich auf alle Lösungen angewendet werden muss. In einem solchen Fall wird die Bewertungsmaßnahme erst angestoßen, nachdem die Einreichungsfrist abgelaufen ist und alle eingereichten Lösungen in ihrer finalen Version vorliegen. Zur Plagiatserkennung wird der eingereichte Quellcode mittels eines Parsers in eine Folge von Symbolen umgewandelt. Diese Folge von Symbolen stellt eine abstrakte Programmstruktur dar. Diese Programmstrukturen werden durch Plagiaterkennungsalgorithmen bezüglich ihrer Ähnlichkeit untersucht.

- *Unit-Test-Runner*: Der Unit-Test-Runner bringt kompilierte Java-Programme zur Ausführung. Jede Konfiguration für dieses Plugin definiert für jede

3 http://checkstyle.sourceforge.net/
4 http://mvnrepository.com/artifact/org.eclipse.jdt.core.compiler/ecj/4.3.1
5 http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

einzelne Aufgabe eine Menge von Testfällen, die nach erfolgreicher Kompilierung durchgeführt und ausgewertet werden. Dazu wird unter anderem das Testframework *JUnit*[6] verwendet, welches von uns um mehrere Annotationsmöglichkeiten erweitert wurde. Mit Hilfe dieser Annotationen lässt sich der Inhalt eines automatisch generierten Testberichts steuern. Der Testbericht beschreibt u. a., welche Tests durchgeführt wurden. Mit Hilfe einer Annotation lässt sich jeder Testmethode eine Beschreibung des durchgeführten Tests zuordnen. Je nach Ausgang der Testmethode wird der Testfall dann als erfolgreich oder fehlgeschlagen im Testbericht gekennzeichnet. Für den Fall des Scheiterns eines Tests können längere Fehlermeldungen und Hinweise zur Behebung des Problems auch in Annotationen angegeben werden. Schließlich können der besseren Übersicht wegen Testfälle für den Testbericht gruppiert und ihre Reihenfolge im Testbericht festgelegt werden.

- *Android-JUnit*: Das Plugin Android-JUnit dient der Ausführung von Testfällen für Android-Applikationen. Durch dieses Plugin lassen sich Tests für die Oberflächen von Android-Apps durchführen. Android-JUnit ist momentan das einzige Plugin der Android-AU. Kapitel 16.5 geht noch ausführlicher auf das Testen von Android-Apps ein.

Die möglichen Ergebnisse jeder einzelnen Bewertungsmaßnahme sind (aufsteigend geordnet nach dem Grad der Problematik): erfolgreich, mit Warnung abgeschlossen, Fehler entdeckt oder Ausführung der Bewertungsmaßnahme gescheitert. Das Gesamtbewertungsergebnis zu einer eingereichten Lösung ist das problematischste Ergebnis aller Einzelbewertungen. Es gibt weder eine Gewichtung der Bewertungsmaßnahmen noch werden den Ergebnissen Punktezahlen zugeordnet.

Wie im Abschnitt über Ausführungsumgebungen schon erwähnt wurde, erzeugt jedes Plugin außerdem eine XML-Ergebnisdatei, in der Details zur Ausführung der Bewertungsmaßnahme zu finden sind. Wenn in ASB ein anderes Programm wie z. B. Checkstyle oder JUnit zur Bewertung benutzt wird, so ist immer ein entsprechendes Plugin nötig, das das benutzte Programm kapselt. Das Plugin führt sowohl vorbereitende Schritte durch, um die Ausführung des genutzten Programms zu ermöglichen, als auch nachbereitende Schritte, um das Ergebnis des Programms in ein ASB-eigenes XML-Format zu transformieren. Diese Transformation ist natürlich stark davon abhängig, in welcher Form das benutzte Programm seine Ergebnisse hinterlässt. Das eigentliche Checkstyle z. B. erzeugt bereits eine XML-Datei, die nach Ausführung vom Checkstyle-Plugin geparst und in das ASB-eigene XML-Format transformiert wird. Beim Unit-Test-Runner wird

6 http://junit.org/junit4/

die XML-Ergebisdatei aus den Ergebnissen der JUnit-Ausführung und den Texten der oben beschriebenen Annotationen erzeugt. In allen Fällen wird die Ergebnisdatei später aus dem ASB-eigenen XML-Format zur Anzeige im Browser in HTML gewandelt.

Die Plugin-Konfigurationen

Zu jedem Plugin kann es verschiedene *Konfigurationen* geben. Art und Umfang der Konfigurationsdateien unterscheiden sich je nach Plugin. Das Checkstyle-Plugin verlangt eine XML-Datei, in der die Codierungskonventionen definiert werden. Der *Unit-Test-Runner* und *Android-JUnit* hingegen benötigen die durchzuführenden Tests zu einer gestellten Aufgabe. Tests für Aufgaben zur Lehrveranstaltung „Grafische Benutzeroberflächen“ funktionieren zum Beispiel so: Ein studentisches Programm wird gestartet, auf der Oberfläche werden bestimmte Interaktionselemente identifiziert, die von den Studierenden mit vorgegebenen Identifikatoren gekennzeichnet sein müssen, auf diesen Elementen werden bestimmte Aktionen wie Texteingaben oder Mausklicks über eine Programmierschnittstelle ausgelöst und schließlich wird überprüft, ob bestimmte Interaktionselemente die erwarteten Beschriftungen aufweisen. Die Tests für Android-Apps arbeiten nach demselben Prinzip. Beim Testen paralleler Programme werden die studentischen Programme instrumentiert, um bestimmte zeitliche Abläufe zu erzwingen und zu prüfen, ob sich die studentischen Programme dabei so verhalten wie erwartet. Ein spezielles Prüfverfahren, um die Ausgaben parallel laufender Threads zu überprüfen, wurde in [OB07] präsentiert.

Es ist möglich, Plugin-Konfigurationen zur Vor- oder Nachbereitung für Plugins oder deren Konfigurationen anzulegen. So wird beispielsweise eine Konfiguration des Java-Compilers als Vorbereitung des Unit-Test-Runners genutzt, da der von den Studierenden eingereichte Quellcode erst übersetzt werden muss, bevor er ausgeführt werden kann. Weiterhin werden mit den Konfigurationen die maximale Laufzeit der Bewertung und die von der Ausführungsumgebung geforderten Ausführungsparameter festgelegt.

Plugin-Konfigurationen können wie Plugins ebenfalls zur Laufzeit im System angelegt werden. Eine Plugin-Konfiguration bezieht sich immer auf genau ein Plugin. Das heißt, durch die Angabe einer Konfiguration wird eine ganz konkrete Bewertungsmaßnahme definiert. Plugin-Konfigurationen können global definiert werden. Damit sind sie für jede Lehrveranstaltung nutzbar. Konfigurationen können aber auch nur für einzelne Lehrveranstaltungen angelegt werden. Damit können sie nur für die betreffende Lehrveranstaltung verwendet werden.

Lehrveranstaltungen und Aufgaben

Das ASB-System unterscheidet zwischen der Konfiguration von Bewertungsmaßnahmen und dem Administrieren von Lehrveranstaltungen. Lehrveranstaltungen und ihre Aufgaben, zu denen Lösungen hochgeladen werden können, sind der zentrale Kontext, in dem sowohl Dozenten als auch Studierende das ASB-System innerhalb eines Semesters nutzen.

Die Dozenten bzw. die sie unterstützenden Assistenten können Lehrveranstaltungen sowie dazugehörige Aufgaben anlegen und diesen die durchzuführenden Bewertungsmaßnahmen in Form von Plugin-Konfigurationen zuordnen. Bewertungsmaßnahmen können der gesamten Lehrveranstaltung oder aber nur einzelnen Aufgaben zugeordnet werden. Der Lehrveranstaltung zugeordnete Bewertungsmaßnahmen gelten automatisch für jede Aufgabe dieser Lehrveranstaltung. Dies bietet sich für Checkstyle und das Kompilieren an, denn in der Regel werden in einer Lehrveranstaltung über das gesamte Semester dieselbe Programmiersprache und dieselben Programmierkonventionen verwendet. Will man eine Plagiatserkennung immer für alle Aufgaben durchführen, so würde man die entsprechende Konfiguration auch der Lehrveranstaltung zuordnen. Plugin-Konfigurationen, die Testfälle zu einer konkreten Aufgabe beinhalten, werden direkt einer Aufgabe zugeordnet. Einmal konfigurierte Bewertungsmaßnahmen müssen nur im Falle einer Änderung angepasst werden. Gibt es keine Änderungen, so können einmal angelegte Aufgaben über mehrere Semester verwendet werden. Beim Anlegen von Aufgaben müssen die Dozenten einen Zeitraum angeben, in dem Lösungen zu dieser Aufgabe hochgeladen werden können. Lediglich dieser Zeitraum muss in jedem Semester angepasst werden.

Studierende melden sich zu den Lehrveranstaltungen im System an und haben somit Zugriff auf alle offenen Aufgaben. Zu erstellten Aufgaben können die Studierenden innerhalb des festgelegten Zeitraumes Lösungen einreichen. Beim Einreichen einer Lösung werden die Konfigurationen der anzuwendenden Bewertungsmaßnahmen ausgelesen, die oben vorgestellte Ordnerstruktur aufgebaut und der Ausführungsumgebung übergeben. Nach Beenden einer Bewertung wird die Bewertungsdatei ausgelesen, aufbereitet und dem Client zur Darstellung übermittelt.

16.5 Testen von Android-Apps

Das Testen von Android-Applikationen stellt für das ASB-System eine besondere Herausforderung dar. Da Android-Apps in Java implementiert werden, kön-

nen diese auch durch die bereits vorhandenen statischen Codeanalysen für Java-Programme (zum Beispiel Checkstyle und JMaat) evaluiert werden. Jedoch ist es nicht möglich, den Unit-Test-Runner für Android-Applikationen zu nutzen, da diese innerhalb eines Android-Betriebssystems ausgeführt werden müssen. Im Folgenden wird der Ablauf einer solchen Bewertung durch das Android-JUnit-Plugin innerhalb der Android-AU nach [Hei+15] erläutert.

Aufgrund benötigter Speicher- und Rechenressourcen werden Android-Programme nicht auf dem eigentlichen ASB-Server, sondern auf einem eigens dafür bereitgestellten Server ausgeführt. Das ASB-System fungiert hierbei als Client, der die Anfragen weiterleitet. Abbildung 16.3 veranschaulicht den Bewertungsablauf einer von einer Studierenden eingereichten Lösung.

Die Bewertung von Android-Apps verläuft wie folgt:

1. Eine Studierende lädt ihr zur Aufgabe erstelltes Android-Projekt (Quelldateien, XML-Dateien, Manifest-Datei, ...) als ZIP-Archiv auf den ASB-Server.

2. Der ASB-Server nimmt das Archiv entgegen. Die zur Bewertung der Aufgabe benötigten Testklassen werden zusammen mit dem studentischen Programmcode erneut gepackt.

3. Im nächsten Schritt schickt der ASB-Server diese Dateien mittels SSH an den Android-Server.

4. Zum Ausführen einer Applikation auf realen Android-Geräten oder -Emulatoren wird diese in einer APK-Datei verpackt. In diese Form wird nun die studentische Lösung vom Android-Server gebracht. Zum Testen von Android-Apps wird ebenfalls eine eigene Test-App in Form einer APK-Datei benötigt. Diese bleibt für alle Bewertungen dieser Aufgabe gleich. Da

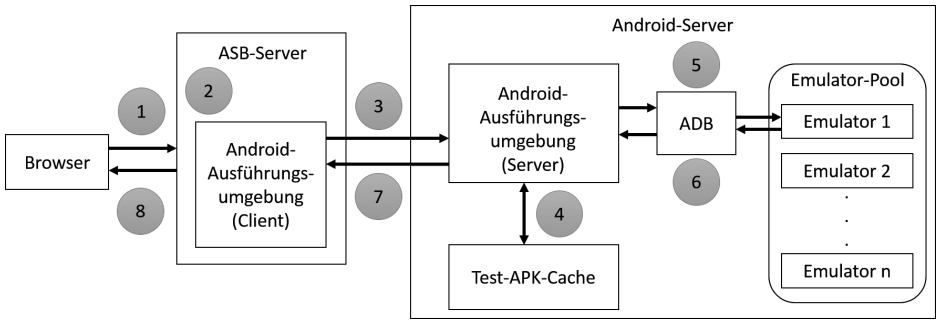


Abbildung 16.3: Ablauf

die Erstellung einer APK-Datei ressourcenintensiv ist, werden Test-APK-Dateien in einem Cache vorgehalten. Falls noch keine solche APK vorhanden ist, wird diese aus den übergebenen Testklassen erzeugt und im Cache gespeichert.

5. Android-Apps können nicht direkt auf dem Server ausgeführt werden, sondern benötigen Android-Emulatoren. Diese werden in einem Pool vorgehalten. Der Server wählt einen freien Emulator aus und übergibt diesem sowohl die studentische APK als auch die Test-APK über die *Android Debug Bridge (ADB)*. Über die ADB wird das Installieren, Starten und Testen der Applikationen veranlasst. Sollte kein Emulator verfügbar sein, wird der Auftrag in einer Auftragswarteschlange gestaut.

6. Nach Ausführung der Tests wird die erstellte Ergebnisdatei über die ADB vom Emulator geladen. Der Emulator wird heruntergefahren und ein neuer Emulator im Pool bereitgestellt. Dies garantiert gleiche Bedingungen für jeden Bewertungsvorgang und verhindert mögliche Seiteneffekte vorheriger Bewertungen.

7. Der ASB-Server empfängt nun die Ergebnisdatei vom Android-Server via SSH.

8. Dieser bereitet die Ergebnisdatei des Testlaufes so auf, dass sie der Studierenden im Browser angezeigt werden kann.

16.6 Einsatz des Systems

Das ASB-System befindet sich derzeit bei vier Modulen des Präsenz- und zwei Modulen des Fernstudiums produktiv im Einsatz. In den diesen Modulen entsprechenden Lehrveranstaltungen auf dem ASB-Server werden pro Jahr ca. 400-500 Studierende registriert, wobei bei dieser Zahl manche Studierende auch mehrfach gezählt werden können, nämlich dann, wenn sie im Laufe eines Jahres mehrere Module belegen, in denen ASB eingesetzt wird. Die Ausführungsumgebungen für C++ und Python wurden realisiert, weil entsprechender Bedarf angemeldet wurde. Tatsächlich wurden sie aber bislang nicht genutzt. Auch die Bewertungsmaßnahmen haben sich im Laufe der Zeit geändert und wurden weiterentwickelt. So hatten wir eine Zeit lang das Tool Findbugs sowie ein Tool zur Bestimmung von Softwaremetriken im Einsatz. Da die Bewertungsergebnisse aber doch nicht so sehr aussagekräftig waren, wurden diese Werkzeuge in letzter Zeit nicht mehr eingesetzt.

Auch müssen Tests für neue Aufgaben zusätzlich entwickelt und bei einer Änderung von Aufgaben angepasst werden. Als das Modul „Grafische Benutzeroberflächen“ beispielsweise von Swing auf JavaFX umgestellt wurde, waren alle bisher vorhandenen Tests nicht mehr nutzbar. Sie müssen nun für die neue Plattform JavaFX neu entwickelt werden.

16.7 Verfügbarkeit des Systems

Das ASB-System ist unter der asb.hochschule-trier.de erreichbar. Neben der Hochschulkennung ist es nach Absprache möglich, eine *ASB-Kennung* anzulegen. Dadurch ist das System auch für Nutzer außerhalb der Hochschule Trier zum Testen nutzbar. Der Quellcode des Systems kann auf Anfrage bereitgestellt werden. Zur Nutzung des Systems an einer anderen Hochschule muss die Instanz dort verwaltet werden. Ein Hosting auf einem Server der Hochschule Trier ist nicht möglich.

Danksagung

Mehrere Personen haben im Laufe der Jahre zur Entwicklung des ASB-Systems beigetragen. Bei allen möchten wir uns an dieser Stelle für ihre Arbeit bedanken, insbesondere bei Patrick Fries, Mathis Heimann, Thiemo Morth, Klaus Müller und Christian Schmal. Ohne sie wäre dieser Beitrag nicht möglich gewesen.

Literatur für dieses Kapitel

[Hei+15] Mathis Heimann u. a. „Automatische Bewertung von Android-Apps“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2015)*. Bd. 1496. CEUR Workshop Proceedings. 2015.

[Mor+07] Thiemo Morth u. a. „Automatische Bewertung studentischer Software“. In: *Workshop „Rechnerunterstütztes Selbststudium in der Informatik“*. Universität Siegen, 2007.

[Mü07] Klaus Müller. *Vergleich und Implementierung von Verfahren zur Plagiaterkennung von Software*. Abschlussarbeit. 2007.

[OB07] Rainer Oechsle und Kay Barzen. „Checking Automatically the Output of Concurrent Threads“. In: *11th Annual SIGCSE Conference on In-*

novation and Technology in Computer Science Education. Dundee, Scotland, 2007.

[Oec13] Rainer Oechsle. *Java-Komponenten*. München: Carl Hanser Verlag GmbH & Co. KG, 2013.

17 Steckbriefe und Featurematrix der Grader

JACK Grader-Steckbrief (siehe Kapitel 9)	
Entwicklung seit:	2006
Aktueller Hauptentwickler:	Universität Duisburg-Essen Arbeitsgruppe Prof. Dr. Michael Goedicke
Kontaktmöglichkeit:	jack@paluno.uni-due.de
Online-Demo:	https://jack-demo.s3.uni-due.de/
Zielgruppe:	Veranstalter und Teilnehmer einführender und fortgeschrittener Lehrveranstaltungen zur objektorientierten Programmierung

Praktomat Grader-Steckbrief (siehe Kapitel 10)	
Entwicklung seit:	1999
Aktueller Hauptentwickler:	KIT Lehrstuhl Prof. Dr. Gregor Snelting
Kontaktmöglichkeit:	https://www.lists.kit.edu/sympa/info/praktomat-users
OpenSource:	https://github.com/KITPraktomatTeam/Praktomat
Zielgruppe:	Vor allem Programmiereinsteiger, aber auch fortgeschrittene Vorlesungen (Isabelle-Praktikum, Financial Economics)

Graja Grader-Steckbrief (siehe Kapitel 11)	
Entwicklung seit:	2012
Aktueller Hauptentwickler:	Hochschule Hannover Prof. Dr. Robert Garmann
Kontaktmöglichkeit:	robert.garmann@hs-hannover.de
Zielgruppe:	Einführende und fortgeschrittene Programmierung mit Lerninhalten von prozeduraler Programmierung über objektorientierte Konzepte bis hin zu Threads und Grafik

aSQLg Grader-Steckbrief (siehe Kapitel 12)	
Entwicklung seit:	2010
Aktueller Hauptentwickler:	Hochschule Hannover Prof. Dr. Carsten Kleiner, Prof. Dr. Felix Heine
Zielgruppe:	Studierende in Datenbankvorlesungen mit SQL-Anteil, sowohl Anfänger als auch Fortgeschrittene

GATE Grader-Steckbrief (siehe Kapitel 13)	
Entwicklung seit:	2009
Aktueller Hauptentwickler:	HU Berlin (Sven Strickroth) TU Clausthal (Oliver Müller)
Kontaktmöglichkeit:	sven@strickroth.net, oliver.mueller@tu-clausthal.de
OpenSource:	https://github.com/csware/si
Zielgruppe:	Java-Programmierübungen, Tutorinnen und Tutoren in Java-Programmierübungen mit mehreren hundert Studierenden, Studierende insbesondere in Einführungsveranstaltungen zur Java-Programmierung

VEA Grader-Steckbrief (siehe Kapitel 14)	
Entwicklung seit:	2000
Aktueller Hauptentwickler:	Universität Osnabrück, Zentrum für Informationsmanagement und virtuelle Lehre
Kontaktmöglichkeit:	Tobias Thelen
Online-Demo:	https://mvc.ikw.uni-osnabrueck.de/vips/vips.php?adm=1&mgc=!ViPS!&srv=prolog
OpenSource:	ja / GPL
Zielgruppe:	Einführungskurse Prolog / Lisp

PABS Grader-Steckbrief (siehe Kapitel 15)	
Entwicklung seit:	2010
Aktueller Hauptentwickler:	Universität Würzburg Alexander Dallmann, Lukas Iffländer
Kontaktmöglichkeit:	lukas.ifflaender@uni-wuerzburg.de, alexander.dallmann@uni-wuerzburg.de
Zielgruppe:	Programmierpraktika und praktische Aufgaben zur Vorlesungsbegleitung

ASB Grader-Steckbrief (siehe Kapitel 16)	
Entwicklung seit:	2006
Aktueller Hauptentwickler:	Hochschule Trier, Fachbereich Informatik, ASB-Projektteam (Britta Herres, David Schuster)
Kontaktmöglichkeit:	Rainer Oechsle (oechsle@hochschule-trier.de)
Zielgruppe:	Programmierpraktika und praktische Aufgaben zur Vorlesungsbegleitung

Feature	**JACK**	**Praktomat**	**Graja**	**aSQLg**	**GATE**	**VEA**	**PABS**	**ASB**
Programmiersprachen								
Java	✓	✓	✓		✓		✓	✓
C/C++	✓	✓						✓
Fortran		✓						
.NET	✓							
Haskell		✓						
Python	✓	✓						✓
R	✓	✓						
Prolog						✓		
Lisp						✓		
Andere Sprachen								
Isabelle		✓						
UML	✓				✓			
SQL				✓				
Statische Prüfverfahren								
Syntax	✓	✓	✓	✓	✓			✓
Stil	✓	✓		✓				✓
FindBugs/PMD			✓					✓
Metriken	✓							✓
andere	✓			✓	✓	✓		
Dynamische Prüfverfahren								
Ein-/Ausgabetests (z. B. Unit-Tests)	✓	✓	✓	✓	✓	✓	✓	✓
GUI-Tests								✓
Remote-Tests								✓
Weitere								✓
Prüfverfahren erweiterbar	✓	✓			✓			✓

Tabelle 17.1: Featurematrix (Teil 1) der Grader aus Teil II dieses Buches.

Feature	JACK	Praktomat	Graja	aSQLg	GATE	VEA	PABS	ASB
Feedbackmöglichkeiten								
Punkte/Noten	✓	✓	✓	✓	✓	✓		
Textuell	✓	✓	✓	✓	✓	✓	✓	✓
Codeannotation	✓	✓						
Grafisch	✓		✓					
Manuelle Nachbewertung	✓	✓	✓		✓			
Plagiatserkennung	✓	✓			✓	✓	✓	✓
Integration in LMS								
Für Studierende	✓	✓	✓	✓		✓		
Für Lehrende		✓	✓	✓		✓		
Unterstützte Standards								
IMS-LTI	✓							
ProFormA XML	✓	✓	✓	✓	✓			
Grappa Middleware			✓	✓				
Didaktische Einsatzszenarien								
Objektorientierte Programmierung	✓	✓	✓		✓		✓	✓
Deduktive/logische oder funktionale Programmierung		✓				✓		
Relationale Datenbankabfragen				✓				
UML-Modellierung	✓				✓			
App-Entwicklung								✓
Mathematikausbildung	✓	✓						

Tabelle 17.2: Featurematrix (Teil 2) der Grader aus Teil II dieses Buches.

Teil III

Interoperabilität von Gradern und Lernmanagementsystemen

18 Integration automatisierter Programmbewertung in Moodle

Peter Fricke und Michael Striewe

Zusammenfassung
Das OpenSource-Lernmanagementsystem MOODLE *wurde 1999 von Martin Dougiamas entwickelt [DT03]. Es wurde seither stetig erweitert und ist an vielen Hochschulen im Einsatz. Die Standardinstallation von Moodle bietet jedoch keine Möglichkeit der automatischen Programmbewertung. Dieses Kapitel gibt einen groben Überblick über die Möglichkeiten, externe Grader mit Moodle zu verbinden.*

18.1 Einleitung

Das Lernmanagementsystem Moodle ist weit verbreitet und bietet eigene Features zur Durchführung einfacher Aufgabenformate wie Multiple Choice und Fill-In an, jedoch keine Funktionen zur automatischen Bewertung von Programmieraufgaben. Sollen Lernende oder auch Lehrende in einer entsprechenden Veranstaltung, in der Moodle als Lernmanagementsystem zum Einsatz kommt, nicht dazu gezwungen werden, sich in die Bedienung eines separaten Werkzeugs für die automatische Bewertung von Programmieraufgaben einzuarbeiten, ist eine Integration solcher Werkzeuge in Moodle notwendig. Neben der Frage der Bedienung sprechen auch die Bequemlichkeit des Logins (Single Sign-On in Moodle statt separates Login im externen Werkzeug) sowie die übersichtliche Zusammenführung von Ergebnissen (alle Leistung in Moodle statt verteilt über mehrere Systeme) für eine Integration.

Moodle bietet verschiedene Schnittstellen an, über die zusätzliche Features in eine bestehende Installation integriert werden können. Abschnitt 18.2 diskutiert die Nutzung der LTI-Schnittstelle, über die externe Werkzeuge über ein standardisiertes Verfahren aufgerufen werden können. Abschnitt 18.3 befasst sich mit der Nutzung von Plugins zur Bereitstellung zusätzlicher Features.

18.2 Integration durch IMS-LTI

Der von IMS-Global veröffentlichte LTI-Standard (Learning Tools Integration) beschreibt einen generellen Mechanismus, mit dessen Hilfe eine allgemeine Lernplattform und ein spezialisiertes Lernwerkzeug auf Basis von HTTP-Anfragen Daten austauschen können [Ims]. Moodle stellt eine Implementierung der Version 1.1.1 des LTI-Standards bereit, die von Autoren genutzt werden kann, indem in den Kursraum eine Aktivität des Typs „Externes Tool“ eingebunden wird. Innerhalb dieser Aktivität muss dann eine URL eingetragen werden, unter der das einzubindende Werkzeug angesprochen werden kann. Die tatsächliche Interaktion der Lernenden geschieht dann ausschließlich mit dem externen Werkzeug. Dies bedeutet insbesondere, dass weder die Aufgaben durch Lehrende direkt in Moodle eingetragen werden, noch dass die Einreichung der Lösungen der Lernenden direkt in Moodle erfolgt.

18.2.1 Technik

Moodle implementiert durch den Aufruf der URL das minimale Verfahren, das im Standard als „Basic Launch“ bezeichnet wird. Die Rolle der Lernplattform ist darin die des sogenannten „Tool Consumer“ (TC), der die Dienste eines spezialisierten Werkzeuges nutzen möchte, indem er Aufrufe an dieses Werkzeug delegiert. Dazu werden dem aufgerufenen Werkzeug mehrere Parameter übertragen, die ihm Auskunft über den eingeloggten Benutzer, den aufrufenden Kursraum, den vom Benutzer angeklickten Link usw. geben. Es können auch zusätzliche Parameter definiert werden, die von den Autoren individuell in der Aktivität belegt werden können.

Das aufgerufene externe Werkzeug wird im Standard als Tool Provider (TP) bezeichnet und stellt dem Tool Consumer seine Dienste zur Verfügung. Es ist dabei Sache des Tool Consumers zu entscheiden, ob das Werkzeug nach einem Klick auf den Link in einem neuen Browserfenster oder eingebettet in die Lernplattform angezeigt wird. Der Tool Provider stellt in jedem Fall seine eigene Weboberfläche bereit und arbeitet mit den Daten, die ihm im Rahmen des Aufrufs übertragen wurden. Im Gegenzug kann der Tool Provider den sogenannten „Basic Outcome Service“ nutzen, der vom Tool Consumer zur Verfügung gestellt wird, um in einer einzelnen HTTP-Anfrage eine Punktzahl zurückzuliefern, die im Tool Consumer als Bewertung für die Durchführung der Aktivität verbucht wird.

Die Kommunikation zwischen Tool Provider und Tool Consumer kann über SSL-verschlüsselte Verbindungen (HTTPS) durchgeführt werden und wird in je-

dem Fall über OAuth signiert, um die nötige Sicherheit zu gewährleisten. Dabei wird im Standard davon ausgegangen, dass ein Vertrauensverhältnis zwischen Tool Consumer und Tool Provider etabliert wird. Der Tool Provider muss dem Tool Consumer dazu einen Anwendungsschlüssel und ein zugehöriges Passwort zuweisen, die der Tool Consumer nutzt, um sich beim Aufruf des Links gegenüber dem Tool Provider zu authentifizieren. Der Tool Provider nutzt dieselben Daten, um seine eigenen Anfragen zum Setzen von Bewertungen zu signieren. Dieses Vertrauensverhältnis bedeutet insbesondere, dass zwischen dem Tool Consumer und dem Tool Provider keine Passwörter der Benutzer ausgetauscht werden. Der Tool Consumer kann jedoch bei Ausführung des Links Daten wie den Benutzernamen oder eine Benutzer-ID übertragen, so dass der Tool Provider diese nutzen kann, wenn sie für die Durchführung des angebotenen Dienstes notwendig sind.

Eine exemplarische Konfiguration für die Nutzung des Graders JACK in Moodle ist in Abbildung 18.1 dargestellt. Alle o. g. Einstellungen wie die URL des Tool Providers oder die Art der Anzeige im Kurs (Einstellung „Startcontainer“) können hier vorgenommen werden.

Neben dem LTI-Standard für die Integration von Werkzeugen und Plattformen gibt es von IMS Global auch den LIS-Standard (Learning Information Services), der sich um eine systematische Beschreibung der ausgetauschten Informationen bemüht und der im Rahmen des LTI-Standards genau an dieser Stelle zum Einsatz kommt. Dazu gehört auch, dass der Tool Consumer in jedem Fall eine eindeutige ID des Aufrufs generiert, auf die sich der Tool Provider bei der Rückmeldung der Bewertung beziehen muss. Die tatsächliche Zuweisung der gesendeten Punktzahl zum Benutzer ist damit wiederum Sache des Tool Consumers und es wird somit verhindert, dass der Tool Provider beliebige Daten im Datenbestand des Tool Consumers verändern kann.

18.2.2 Vorteile

Mit dem LTI-Standard steht eine einfache Möglichkeit bereit, durch die ein externes Werkzeug wie beispielsweise ein Grader zur automatischen Bewertung von Programmieraufgaben schnell und mit sehr geringem Aufwand in Moodle integriert werden kann. Einzige Voraussetzung dazu ist, dass dieses Werkzeug die passende Schnittstelle implementiert. Insbesondere müssen keine zusätzlichen Installationen durchgeführt, sondern lediglich einige Einträge in der Konfiguration vorgenommen werden. Ferner sind keine Kompromisse oder Anpassungen bei bestehenden Aufgabensammlungen für das externe Werkzeug notwendig, da die

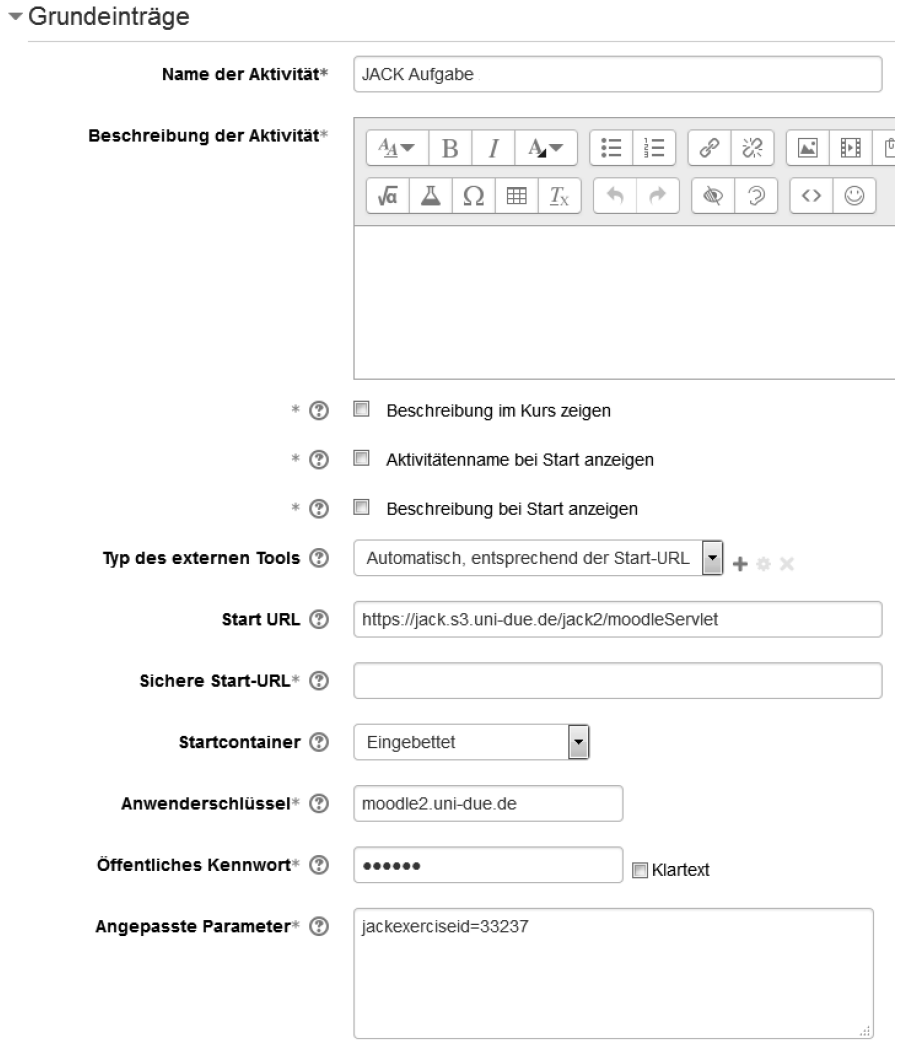


Abbildung 18.1: Beispielhafte Konfiguration einer Aufgabe vom Typ „Externes Tool“. Relevant sind insbesondere die Start-URL des externen Werkzeugs sowie der Anwenderschlüssel und das Kennwort, mit dem sich der Tool Consumer bei diesem Werkzeug anmeldet. Als angepasste Parameter können spezifische Informationen übertragen werden. In diesem Beispiel ist es die ID der Aufgabe, die innerhalb des externen Werkzeugs angezeigt werden soll.

Inhalte nicht nach Moodle importiert, sondern ausschließlich im externen Werkzeug verwaltet werden.

Auch den Entwicklern von Gradern bietet diese einfache Schnittstelle Vorteile, denn sie müssen nicht Plugins (siehe Abschnitt 18.3) für verschiedene LMS in den jeweiligen Programmiersprachen dieser Systeme entwickeln, sondern sie können mit der einmaligen Implementierung der LTI-Schnittstelle Interoperabilität mit verschiedenen LMS herstellen.

18.2.3 Nachteile

Die Möglichkeiten des „Basic Outcome Service“ zur Rückmeldung eines Ergebnisses in Moodle beschränken sich auf eine Punktzahl im Interval von 0.0 bis 1.0. Dies ist zwar ausreichend, um den erfolgreichen Abschluss einer Aktivität zu signalisieren und beispielsweise in Moodle-Lernpfaden entsprechend auf das Ergebnis zu reagieren, aber es bietet keinen Platz für weitere Informationen. Jegliches weiteres Feedback muss daher durch das externe Werkzeug und dessen Oberfläche erfolgen. Dies ist insbesondere bei Programmieraufgaben nachteilig, da die Auswertung solcher Aufgaben eine gewisse Zeitspanne in Anspruch nehmen kann. Das externe Werkzeug muss also ggf. sicherstellen, dass bei einem späteren erneuten Aufruf des entsprechenden Links in Moodle auch eine Einsicht in frühere Lösungen möglich ist. Ferner muss sichergestellt sein, dass Moodle eine späte Rückmeldung überhaupt noch entgegen nimmt, da jeder Aufruf einer externen Aktivität mit einem individuellen Token von begrenzter Gültigkeitsdauer versehen ist. Ist diese verstrichen, ist keine Rückmeldung mehr zu diesem Aufruf möglich.

Für Lehrende muss zudem eine Möglichkeit bestehen, sich am externen Werkzeug anzumelden, sofern sie dort eigene Inhalte bereitstellen oder Bewertungen einsehen wollen. Das „Basic Launch“-Verfahren von LTI ist ausschließlich für die Sicht der Lernenden gedacht und bietet somit keinerlei Unterstützung für Lehrende.

18.3 Integration durch Plugins

Moodle (ursprünglich Modular Object-Oriented Dynamic Learning Environment) ist, wie der Name schon sagt, modular aufgebaut und bietet mehrere Schnittstellen zur Integration verschiedener Arten von Plugins. Dabei ist Moodle in der Programmiersprache PHP entwickelt und bedient sich verschiedener Webtechnolo-

gien wie zum Beispiel JavaScript und Ajax und ist unter der GNU GPLv3[1] lizenziert. Somit ist eine Erweiterung und auch die Veränderung des bestehenden Codes von Moodle selbst und den angebotenen Plugins möglich.

Die Plugins können genutzt werden um die Seiten- und Kursansicht z. B. durch „Blöcke“ zu verändern, Funktionen über „Aktivitäten“ hinzuzufügen, oder die Administration durch „Admin-Tools“ zu verbessern. Die Community hat sich in den letzten Jahren stark vergrößert, so dass es eine große Auswahl dieser Plugins gibt. Für die Entwicklung eines Plugins stehen ebenfalls eine Reihe von APIs online zur Verfügung.

Möchte ein Entwickler sein Plugin der Community zur Verfügung stellen, muss er sich zudem an eine Vielzahl von Programmierrichtlinien halten, damit sein Plugin der automatischen und manuellen Validierung genügt. Erst nach erfolgreichem Bestehen wird dieses Plugin in das Moodle-Plugin-Directory aufgenommen.

Sollte eine lokale Anpassung ausreichend sein, können die Programmierrichtlinien ignoriert werden. Eine Aufnahme in das Moodle-Plugin-Directory ist somit ausgeschlossen, jedoch hat man somit alle Freiheiten ggf. sogar Änderungen am Moodle-Core vorzunehmen um seine Ziele zu erreichen.

Durch die stetige Weiterentwicklung von Moodle ist es allerdings auch nötig, dass selbst entwickelte Plugins angepasst und auf dem neuesten Stand gehalten werden, damit es keine Kompatibilitätsprobleme gibt.

18.3.1 Integration durch Anbindung von Grappa

Für die in Kapitel 23 vorgestellte Middleware Grappa wurde ein Moodle-Plugin entwickelt. Es basiert auf der Hauptaktivität „Aufgabe“[2] und wurde für die Anbindung an Grappa angepasst. Somit sind die Grundfunktionalitäten und auch die Ansicht innerhalb von Moodle den Lehrenden und Studierenden bekannt. Das Moodle-Plugin nutzt die in Kapitel 23.4.2 vorgestellte Clientbibliothek um mit Grappa zu kommunizieren.

Der Erweiterungsmechanismus wurde auch innerhalb der *Programmieraufgaben*-Plugins beibehalten, so dass je nach Unterstützung des Graders weitere Mini-Plugins für Abgaben[3] und Feedback[4] möglich sind. Derzeit wird die normale Dateiabgabe und Textfeldabgabe unterstützt. Das Feedback kann ebenfalls über die Rückmeldung des Graders hinaus erweitert werden, indem ein Kommentar-

1 https://www.gnu.org/licenses/gpl-3.0.en.html
2 https://docs.moodle.org/dev/Assignment
3 https://docs.moodle.org/dev/Assign_submission_plugins
4 https://docs.moodle.org/dev/Assign_feedback_plugins

feld oder eine Datei genutzt werden. Der Moodle-Administrator kann mehrere Grappa-Grader-Instanzen (siehe auch Abbildung 18.2) hinzufügen. Der Lehrende braucht grundlegende Kenntnisse, welche Konfigurationen und Dateien der Grader benötigt. Ebenso müssen Studierende instruiert werden, wie eine Lösung eingereicht werden muss.

18.3.1.1 Installation

Für die Installation des Moodle-Plugins sollte zunächst eine Grappa-Instanz mit mindestens einem Grader eingerichtet werden. Für die aktuellen Moodle-Versionen wurde eine neue Plugin-Struktur gewählt. Vorher bestand die Anbindung an Grappa aus zwei *Activity*-Plugins – eines für *Problems* und eines für *GrdCfgs*. Dies hatte den Nachteil, dass für eine Aufgabe immer mindestens zwei Aktivitäten angelegt werden mussten. Da Aktivitäten innerhalb des Kurses erscheinen, wurde die Übersichtlichkeit des Kurses für den Lehrenden gestört. Seit Moodle 3.0 besteht die Möglichkeit der Manipulation der Navigationsleiste, so dass die Aktivität für *GrdCfgs* mit dem Haupt-Plugin *mod_grappa* zur Verwaltung von Programmieraufgaben (*Problems*) verschmolzen wurde. Es gibt nun eine extra Übersicht für die angelegten Konfigurationsdateien (siehe Abbildung 18.3) und lediglich die Programmieraufgabe selbst ist im Kurs zu sehen.

Nach der Installation des Moodle-Plugins müssen zunächst für jeden Grader die dazugehörigen Grappa-Serverinstanzen angelegt werden. Abbildung 18.2 zeigt eine beispielhafte Konfiguration mit den Gradern *aSQLg* (Kapitel 12) und *Graja* (Kapitel 11). Es können beliebig viele Instanzen angelegt werden. Dabei müssen die unterstützten Rollen der Konfigurationsdateien (vergleiche *roles* in Kapitel 23.2.1) des Graders bzw. BackendPlugins hinterlegt werden.

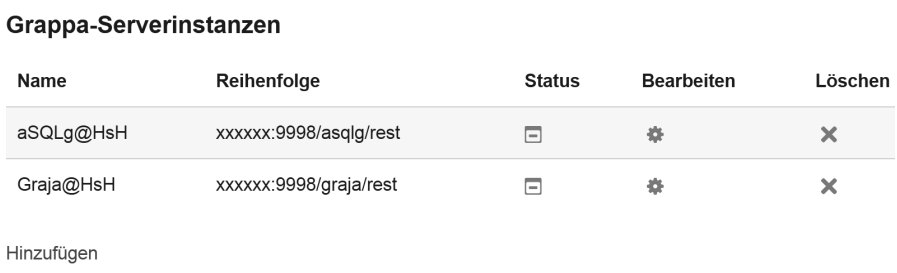


Abbildung 18.2: Übersicht der Konfigurationsseite für Grappa-Instanzen

18.3.1.2 Konfiguration

Nach erfolgreicher Installation des Moodle-Plugins und dem Anlegen von Grappa-Serverinstanzen kann der Lehrende innerhalb seiner Kurse *GrdCfgs* und *Programmieraufgaben* anlegen. Durch die Abhängigkeit von *GrdCfgs* innerhalb der *Problem*-Struktur muss der Lehrende zwingend folgende Reihenfolge beachten:

1. Lokales Erstellen/Konfigurieren von *GrdCfgs* (bspw. task.zip-Dateien, properties-Dateien, Musterlösungen).
2. Hochladen dieser *GrdCfgs* unter Angabe der Rolle und eines Namens in der *GrdCfg-Verwaltung*.
3. Erstellen der Aktivität *Programmieraufgabe* innerhalb des Kurses.

In Abbildung 18.3 ist eine beispielhafte Übersicht von *GrdCfgs*. Dabei muss zu jeder *GrdCfg* eine Grappa-Server-Instanz, ein beliebiger Name, eine Rolle und eine Datei angegeben werden. Der Grappa-Server generiert eine *GrdCfg-ID*, welche vom Moodle-Plugin hinterlegt wird. *GrdCfgs* können auf dieser Seite angelegt, geändert oder gelöscht werden und werden in Zukunft ebenfalls für Quizfragen innerhalb von Moodle genutzt. Im einfachsten Fall, wie bspw. in Graja (Kapitel 11), reicht eine im Austauschformat (siehe Abschnitt 24) vorliegende *task.zip*-Datei.

Sind alle benötigten *GrdCfgs* angelegt, kann innerhalb des Kurses die Aktivität „Programmieraufgabe“ anlegt werden. Diese gleicht in den Grundeinstellungen der normalen Moodle-„Aufgabe“. Lediglich der Unterabschnitt *Grader Ein-*

Abbildung 18.3: Übersicht der Konfigurationsseite für GrdCfgs

stellungen und *Teilaufgaben o. -aspekte* sind zusätzlich zu sehen. Lehrende können wie gewohnt Aufgabenname und Beschreibung angeben und grundlegende Aufgabenkonfiguration wie bspw. Aufgabeneröffnung, Abgabezeitpunkt, Voraussetzungen und Bewertungsrichtlinien konfigurieren. Detaillierte Informationen zu diesen Attributen einer Aufgabe sind nicht Teil dieses Artikels.

Innerhalb der zusätzlichen Abschnitte (Abbildung 18.4) muss der Lehrende zunächst die *Aufgabenwurzel* konfigurieren, indem die Informationen wie *Grader*, *Knotenschlüssel*, *GrdCfgs* und maximale Punktzahl angegeben werden müssen. Andere Einstellmöglichkeiten werden ebenfalls unterstützt und in Kapitel 23.2.2 erläutert.

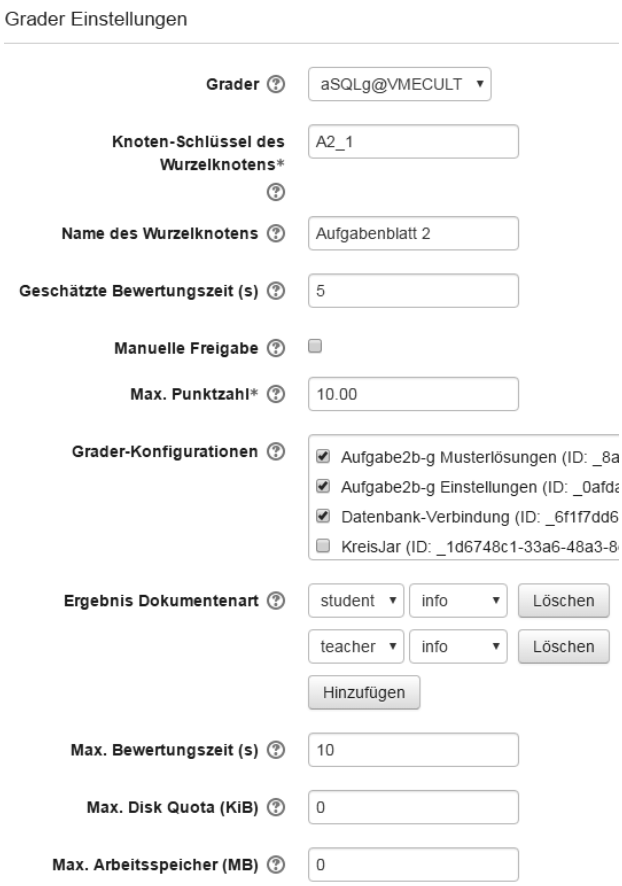


Abbildung 18.4: Ansicht der Lehrenden zum Anlegen einer Programmieraufgabe

Zusätzlich können der *Aufgabenwurzel* Teilaufgaben oder Teilaspekte in einem weiteren Abschnitt angehängt werden. Diese Einstellungen sind identisch und spiegeln die Baumstruktur eines Grappa *Problems* wider.

Erstellte Aufgaben können beliebig kopiert und in andere Kurse innerhalb von Moodle übernommen werden. Ein Im- und Export in das Austauschformat (Kapitel 24) ist in Planung.

18.3.1.3 Bewertung studentischer Abgaben

Die studentische Einreichung kann ebenfalls bei der Programmieraufgabenerstellung konfiguriert werden. Eine Auswahl zwischen Datei und Freitext ist derzeit möglich. Sobald ein Studierender seine Abgabe bestätigt hat, wird diese als Datei und unter Angabe der zu der Programmieraufgabe zugeordneten *problem-id* an Grappa übermittelt. Dabei bekommt das Moodle-Plugin eine erste Abschätzung der Dauer der Bewertung und zeigt diese Zeit als Countdown-Timer auf einem Button an. In Abbildung 18.5 ist der Abgabebildschirm eines Studierenden zu sehen. Sobald der Countdown-Timer abgelaufen ist, kann der Studierende auf „Status aktualisieren“ klicken. Im Hintergrund wurde bereits das Ergebnis von Grappa abgefragt und zur Verfügung gestellt.

Die Darstellung des Grader-Feedbacks innerhalb von Moodle wurde von der Hochschule Hannover mehrfach evaluiert [Stö+13; Stö+14; Fri+15]. Die Studierenden legten dabei großen Wert auf die Übersicht der Darstellung und die inhaltliche Verständlichkeit. Das Moodle-Plugin selbst kann auf den Inhalt des Feedbacks keinen Einfluss nehmen, dies obliegt der Rückmeldung des Graders und ggf. des *BackendPlugins*. Die Darstellung innerhalb von Moodle wurde jedoch stetig verbessert. Die mögliche Baumstruktur der von Grappa gelieferten Ergebnisdokumente (vergleiche Kapitel 23.2.3) wurde als auf- und zuklappbare DIV-Container, wie in Abbildung 18.6 umgesetzt. Dabei ist das Moodle-Plugin in der Lage die Rolle der angemeldeten Person zu unterscheiden und die dementsprechenden Informationen zur Verfügung zu stellen. Abbildung 18.6 zeigt die Sicht eines Lehrenden. Der Grader „aSQLg“ lieferte in diesem Beispiel die Musterlösung als zusätzliche Information zurück. Andere Hinweise die vom Grader selbst oder aus der Kommandozeile stammen, sind ebenso denkbar.

Der Lehrende hat zusätzlich die Möglichkeit, in die Bewertung einzugreifen oder sie manuell nach Bestätigung freizuschalten. Ein Freitextfeedback steht ebenso zur Verfügung.

18.3.1.4 Fazit

Durch die Anbindung von Grappa innerhalb von Moodle ist die Möglichkeit gegeben, eine große Menge von Gradern anzusprechen. Die Unterstützung komplexer Aufgaben- und Feedbackstrukturen ist dabei ebenfalls gegeben. Der Lehrende muss jedoch das Wissen über die Funktionsweise der einzelnen Grader besitzen und die Aufgaben dahingehend konfigurieren können.

Eine Unterstützung des Austauschformates befindet sich derzeit in Entwicklung und soll diese Hürde beseitigen, indem eine fertige Aufgabe im *task.xml*-Format einfach in den Kurs geschoben werden kann. Ebenso ist eine grafische Benutzeroberfläche für das in der ProFormA-Gruppe in eCULT+ geplante Repository denkbar. Der Austausch von Aufgaben durch Im- und Export soll Lehrende zusätzlich entlasten.

Durch die baldige Möglichkeit der Erstellung von *Fragetypen* für Grappa sind zusätzlich Prüfungen und Tests im Rahmen des summativen Assessments denkbar.

Abbildung 18.5: Ansicht der Dateiabgabe eines Studierenden

Abbildung 18.6: Darstellung des Grader-Feedbacks in Moodle im Grappa-Format aus Sicht des Lehrenden

18.3.2 Weitere Plugins im Moodle-Plugin-Directory

Das Moodle-Plugin-Directory bietet derzeit nur eine Aktivität zur automatischen Programmbewertung an. Das „Virtual Programming Lab“ (VPL) wurde 2009 von der University of Las Palmas de Gran Canaria (ULPGC) entwickelt und seither stetig vorangetrieben. Mittlerweile liegt VPL in Version 3 vor und zeichnet sich durch die Unterstützung vieler Programmiersprachen mit mehr oder weniger Funktionen aus. [Thi15] zeigt eine ausführliche Beschreibung der Funktionsweise und einer Evaluation dieser Aktivität.

VPL besteht aus zwei Teilen. Zum einen muss das Moodle-Plugin installiert werden und zum anderen der *jail-server*. Die gewünschten unterstützen Program-

miersprachen müssen auf diesem *jail-server* installiert und ggf. konfiguriert werden. Unterstützt werden laut Dokumentation bekannte Sprache wie beispielsweise C, C++, C#, Fortran, Haskell, Java, Pascal, Perl, Php, Prolog, Python und Ruby.

Auf dem *jail-server* werden die Skripte der studentischen Abgaben ausgeführt, was einen enormen Sicherheitsaspekt birgt. Schädlicher oder schlicht falscher Code von Abgaben können dem Moodle-Server keinen Schaden zufügen. Sollte durch Fehlverhalten der studentischen Lösungen der *jail-server* abstürzen, bleibt die Moodle-Instanz davon unberührt und ein Weiterarbeiten in anderen Kursen ist gewährleistet.

Die Kommunikation zwischen dem Moodle-Plugin und dem *jail-server* geschieht über XMLRPC, während Moodle-Server und Client (Browser des Studierenden oder Lehrenden) über Ajax kommunizieren. Beim Aufruf einer Konsole wird ebenfalls ein Websocket (WS/WSS) zwischen Client und *jail-server* verbunden.

18.3.2.1 Fazit

VPL ist ein mächtiges Werkzeug, welches viele verschiedene Programmiersprachen unterstützt. Die Funktionen innerhalb von Moodle und die gesicherte Anbindung an einen Jail-Server bieten einen optimalen Komfort und Sicherheit, da die Studierenden und Lehrenden mit dem Browser arbeiten können (Syntax-Highlighting) und diese Programme nicht auf dem Moodle-Server ausgeführt werden. Der Lehrende kann verschiedene Tests vorbereiten um den studentischen Code auf Input und Output zu analysieren und auch VPL-eigene Skripte erstellen um ggf. externe Tools aufzurufen. Jedoch sieht VPL den Lehrenden ebenfalls als Aufgabenersteller an. Für kleinere Programme oder Skripte kann man davon ausgehen, dass diese Anforderung vom Lehrenden umgesetzt werden kann. Größere Grading-Systeme wie bspw. Graja und JACK (siehe Kapitel 11 und 9) bieten jedoch detaillierte Einstellungsmöglichkeiten und auch dementsprechendes Feedback an. Lehrende sollten komplexere Programmieraufgaben von Programmieraufgabenautoren zur Verfügung gestellt bekommen und lediglich Gewichtungen und Punkte für Bewertungsaspekte einstellen können. So ist ein minimaler Aufwand für den Lehrenden gesichert. [Gar16] stellte diese These auf und beschrieb die Parallelen zur Softwareentwicklung. Mit dem VPL könnte man zwar die Aktivitäten innerhalb von Moodle kopieren und ggf. anderen Lehrenden zur Verfügung stellen, jedoch bedarf es für Anpassungen immer der Kenntnis des Aufgabenaufbaus, der Programmiersprache und der VPL-Syntax.

Literatur für dieses Kapitel

[DT03] Martin Dougiamas und Peter Taylor. „Moodle: Using Learning Communities to Create an Open Source Course Management System“. In: *Proceedings of EdMedia: World Conference on Educational Media and Technology 2003*. Hrsg. von David Lassner und Carmel McNaught. Honolulu, Hawaii, USA: Association for the Advancement of Computing in Education (AACE), 2003, S. 171–178. URL: https://www.learntechlib.org/p/13739.

[Fri+15] Peter Fricke u. a. „Grading mit Grappa – Ein Werkstattbericht.“ In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2015)*. Bd. 1496. CEUR Workshop Proceedings. 2015.

[Gar16] Robert Garmann. *Graja – Autobewerter für Java-Programme*. Bericht (SerWisS) 941. Hochschule Hannover, 2016. URL: http://serwiss.bib.hs-hannover.de/frontdoor/index/index/docId/941.

[Ims] *IMS Learning Tools Integration*. IMS Global Learning Consortium, 2012. URL: https://www.imsglobal.org/specs/ltiv1p1p1.

[Stö+13] Andreas Stöcker u. a. „Evaluation automatisierter Programmbewertung bei der Vermittlung der Sprachen Java und SQL mit den Gradern *aSQLg* und *Graja* aus studentischer Perspektive.“ In: *DeLFI 2013 – Die 11. E-Learning Fachtagung Informatik*. Bd. 218. LNI. GI, 2013, S. 233–238.

[Stö+14] Andreas Stöcker u. a. „Die Evaluation generischer Einbettung automatisierter Programmbewertung am Beispiel von Moodle und aSQLg.“ In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014, S. 301–304.

[Thi15] Dominique Thiébaut. „Automatic Evaluation of Computer Programs Using Moodle's Virtual Programming Lab (VPL) Plug-in“. In: *J. Comput. Sci. Coll.* 30.6 (Juni 2015), S. 145–151. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=2753024.2753053.

19 Integration automatisierter Programmbewertung in LON-CAPA

Frauke Sprengel und Oliver Rod

Zusammenfassung
Das Lernmanagementsystem LON-CAPA stellt schon von sich aus eine Vielzahl möglicher Aufgabentypen, sowie vielfältige Möglichkeiten zur Kombinierbarkeit und Programmierbarkeit von Aufgaben bereit. Daneben gibt es eine einfache Schnittstelle zur externen Bewertung, die natürlich auch zur Programmbewertung benutzt werden kann und wird (z. B. an der Ostfalia und der Hochschule Hannover). Diese soll im Folgenden vorgestellt werden.

19.1 LON-CAPA

Das Lernmanagementsystem LON-CAPA hat seinen Ursprung an der Michigan State University in CAPA (a Computer-Assisted Personalized Approach). Dort wurde es eingesetzt, um automatisch Hausaufgaben zu randomisieren und anschließend zu bewerten. Der Fokus lag damals auf computergenerierten Aufgabenzetteln für Studierende. Im Jahre 2000 haben sich zwei weitere Gruppen der Michigan State University, welche sich ebenso mit dem elektronischen Bereitstellen von Unterlagen mit CAPA beschäftigt haben, zusammengetan und LON-CAPA gegründet. LON-CAPA ist quelloffen und nach der Gnu GPL lizenziert. Bereits 2007 hatten 120 Institutionen Inhalte in LON-CAPA bereitgestellt (vgl. [Kor+08]). Der Austausch wurde besonders für Einführungsveranstaltungen genutzt, da die Erstellung neuer Inhalte sehr zeitaufwendig war bzw. ist.

Teile dieses Kapitels entstanden im Rahmen des Projekts eCULT, Teilvorhaben eAssessment, gefördert durch das Bundesministerium für Bildung und Forschung unter dem Förderkennzeichen 01PL16066H. Die Verantwortung für den Inhalt dieses Kapitels liegt bei der Autorin und dem Autor.

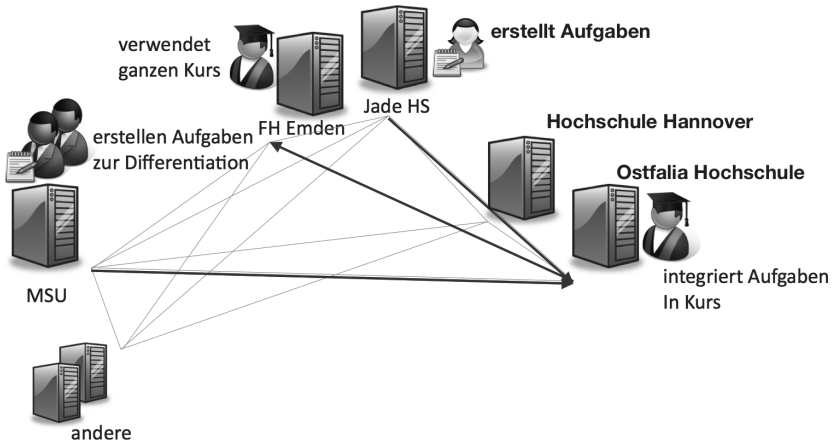


Abbildung 19.1: Veranschaulichung des LON-CAPA-Netzwerkes und der Austausch von Aufgaben.

19.1.1 Aufgabenaustausch

Derzeit (Stand 2016) haben sich über 60 Institutionen[1] weltweit dem LON-CAPA-Netzwerk angeschlossen. In diesem Netzwerk sind insgesamt über 200 000 Aufgaben verfügbar – in LON-CAPA Probleme genannt. Die meisten dieser Aufgaben sind in den technischen Fakultäten angesiedelt und im Rahmen von Forschungsprojekten entstanden. Die Aufgaben sind allen zugänglich. Jede Einrichtung innerhalb des LON-CAPA-Netzwerkes kann sie nutzen oder eigene erstellen. LON-CAPA bietet somit ein Repository mit Versionierung für die Aufgaben an und ermöglicht darüber hinaus deren Austausch. Es ist aber auch möglich, ganze Kurse innerhalb des Netzwerkes zu klonen.
Die Abbildung 19.1[2] soll vereinfacht die Vernetzung der einzelnen Hochschulen im LON-CAPA-Netzwerk verdeutlichen. Es wird dabei sichtbar, dass die verwendeten Aufgaben nicht nur aus der eigenen Hochschule (Domäne) kommen müssen. Dies ermöglicht Lehrenden bereits erstellte Aufgaben anderer Mitglieder im Netzwerk zu nutzen und die gewonnene Zeit in ihre Lehre fließen zu lassen.

1 http://www.lon-capa.org/institutions.html
2 http://www.lon-capa.org/presentations/futureparc.pdf

19.1.2 Aufgabentypen

Bis heute ist LON-CAPA eines der meist eingesetzten Lernmanagementsysteme im Bereich der Hochschullehre. Neben dem Aufgabenaustausch liegt der Schwerpunkt des Systems im Erstellen und Bereitstellen verschiedener Inhalte und Aufgabentypen. So unterstützt LON-CAPA:

- Algebraische Aufgaben (Formel mit Computer-Algebra-System),
- Aufgaben mit freier Gestaltung (Funktionsplot-Antwort mit Hintergrund-Plot),
- Auswahlaufgaben (1-aus-n, Rangordnung, Zuordnung mit Optionen),
- Chemische Aufgaben (Chemische Reaktion, Organisches Material),
- Eingabeabhängige Aufgaben,
- Manuell bewertete Aufgaben (Essay, Dropbox),
- Numerische Aufgaben,
- Verschiedenes (Klick-ins-Bild),
- Externe Bewertung (externalresponse).

Die verschiedenen Aufgabentypen in LON-CAPA werden „problems“ genannt und sind in einer XML-ähnlichen Struktur aufgebaut. Alle problem-Dateien können auch Gebrauch von der Skriptsprache Perl machen, in welcher LON-CAPA zu großen Teilen geschrieben ist. Darüber hinaus verwenden einzelne Aufgabentypen das Computer-Algebra-System Maxima oder die Statistiksprache R. Für Aufgaben im Gebiet der Chemie gibt es zudem die Möglichkeit, den JSME Molecular Editor zu nutzen. Für die Eingabe oder Darstellung von Formeln kann LaTeX verwendet werden (vgl. [Gro15]). Prinzipiell sind in LON-CAPA Gruppenabgaben und Datei-Uploads als Abgabe möglich, allerdings beides nur für den Aufgabentyp *essay*.

Außerdem bietet LON-CAPA eine Schnittstelle für Aufgaben, die außerhalb des LMS bewertet werden sollen. Diese Aufgaben nutzen den Aufgabentyp *externalresponse*. Hier können Texteingaben von einzelnen Studierenden externen Programmen zur Bewertung übergeben werden.

Der Fokus in diesem Kapitel liegt in der Erstellung und Nutzung von External-Response-Aufgaben. Im Folgenden werden zunächst der Aufbau, die Parameter und anschließend die Anzeige vorgestellt. Danach wird am Beispiel von JFLAP eine Einbindung veranschaulicht.

19.2 External Response

Am Beispiel des in Abbildung 19.2 dargestellten XML-Codes werden die wichtigsten Bestandteile einer External-Response-Aufgabe erläutert.

19.2.1 Aufbau einer External-Response-Aufgabe

Die vereinfacht dargestellte Aufgabe soll den prinzipiellen Aufbau einer External-Response-Aufgabe erläutern[3]. Die Zahlen am linken Rand der Abbildung korrespondieren mit Erläuterungen der nachfolgenden Liste.

1. Die komplette Aufgabe befindet sich im Problem-Element.
2. Das Import-Statement erlaubt externe Bibliotheken mit dieser Problem-Datei zu verbinden. Hier bietet es sich an, eine Bibliothek zu laden. Es können somit komplexe Skripte vor dem Kurskoordinator verborgen oder die

```
1> <problem>
2>   <import id="11">/Pfad/zur/header-library</import>
3>   <script type="loncapa/perl">
       # URL des externen Service festlegen
       $externalurl = '$url'
       # Formular-Daten des HTTP-POST-Request festlegen
       %args = 'key => value'</script>
4>   <startouttext/> Aufgabentext <endouttext/>
5>   <instructorcomment>
         $error
     </instructorcomment>
6>   <startouttext/> $ausgabe
       <div id="codemirror-textfield">
     <endouttext/>
7>   <externalresponse answerdisplay="" answer=""
       url="$externalurl" form="%args" id="1">
       <textfield> </textfield>
     </externalresponse>
6>   <startouttext/>
       </div>
     <endouttext/>
8>   <import id="92">/Pfad/zur/footer-library</import>
   </problem>
```

Abbildung 19.2: Minimalbeispiel einer External-Response-Aufgabe (https://media.elan-ev.de/proforma/ProFormA/LON-CAPA/JFLAP-Minimal.problem).

3 Ein ausführlicheres „Hello World“-Beispiel findet sich im LON-CAPA-Aufgabenkatalog: /res/fhwf/ecult/Java/Hello_World.problem

Darstellung des Textfeldes optimiert werden. Die Verwendung von CodeMirror[4] ermöglicht Syntax-Highlighting, automatisches Einrücken und andere Features, welches aus dem Textfeld einen Editor zum Programmieren machen. Es wird aber immer eine Footer-Library benötigt, da erst diese auf das komplette Document Object Model zugreifen kann.

3. In diesem Abschnitt werden die Formulardaten und die Zieladresse des externen Dienstes für die Bewertung festgelegt.

4. Aufgabentext für die jeweilige Aufgabe.

5. Im Instructor-Comment kann man Ausgaben, wie zum Beispiel Fehlerausgaben, vor den Nutzern verbergen bzw. Hinweise nur dem Kurskoordinator anzeigen.

6. Die Variable `$ausgabe` ist ein Platzhalter für aktuelle bzw. alte Ausgaben. Das DIV „codemirror-textfield“ soll die eindeutige Zuordnung für das Eingabetextfeld ermöglichen.

7. In diesem Tag befinden sich alle wichtigen Parameter einer External-Response-Aufgabe. Die drei Hauptbestandteile[5] sind die externe *URL*, die Musterlösung *answer* und Formulardaten *form*. Die *URL* zeigt auf eine Ressource im Internet, welche dieses Problem bewerten soll.

8. In der Footer-Library sollten Skripte geladen werden, die das Textfeld oder die Rückmeldung des externen Dienstes manipulieren.

In den Formulardaten des POST-Request werden von LON-CAPA, ohne dass der Autor weitere Key-Value-Paare hinzufügt, die folgenden drei Felder übermittelt:

1. LONCAPA_correct_answer => answer LON-CAPA nimmt an, dass immer eine Musterlösung für die Bewertung der studentischen Einreichung notwendig ist. Dies kann aber auch vernachlässigt werden.

2. LONCAPA_student_response => textfield Dies entspricht dem kompletten studentischen Text, welcher im Textfeld eingegeben wurde.

3. LONCAPA_language => Dieses Feld wird von LON-CAPA automatisch mit der spezifizierten Sprache der Ressource gefüllt.

4 CodeMirror ist ein in JavaScript programmierter vielseitiger Text-Editor (https://codemirror.net).

5 https://s3.lite.msu.edu/adm/help/Authoring_ExternalResponse.hlp

19.2.2 Antwortformat

Der externe Service hinter der *URL* aus Abschnitt 19.2.1 würde die Einreichung automatisiert testen und das Ergebnis an LON-CAPA zurücksenden. Die Antwort muss folgender XML-Struktur folgen:

```
<loncapagrade>
        <awarddetail></awarddetail>
        <message></message>
        <awarded></awarded>
</loncapagrade>
```

Alle möglichen Antworten finden sich in der LON-CAPA-Hilfe[6]. Es soll hier nur auf Beispiele eingegangen werden, die häufig in der Praxis verwendet werden:

EXACT_ANS, APPROX_ANS: Die Einreichung ist korrekt.

INCORRECT: Die Einreichung ist falsch.

ASSIGNED_SCORE: Die Einreichung ist nur teilweise richtig. Hier wird aber noch ein Gleitkommazahl zwischen 0 und 1 im XML-Element *awarded* erwartet.

ERROR: Bei der Einreichung ist ein Fehler aufgetreten. Der Versuchszähler des Studenten wird nicht erhöht.

In der zurückzugebenden Nachricht (Message) können weitere XML-Elemente enthalten sein. Falls keine zusätzliche Formatierung durch den Aufgabenautoren (oder Benutzer einer Bibliothek) erfolgt, wird der enthaltene Text unter der studentischen Eingabe einfach als gelb unterlegter Text angezeigt.

19.3 Beispiel: JFLAP für die theoretische Informatik

Das Programm JFLAP ist ein Werkzeug, das z. B. im Grundstudium an der Hochschule Hannover in der theoretischen Informatik eingesetzt wird. Es hat eine graphische Oberfläche und kann sowohl Bilder als auch ein internes XML-Format exportieren. Das Programm wurde entwickelt unter der Leitung von Susan Rodgers an der Duke University. Genauere Informationen liefern die JFLAP-Webseite [Rod] und das Buch [RF06].

6 http://source.lon-capa.org/cgi-bin/cvsweb.cgi/doc/homework/datastorage?rev=1.26

Die Studierenden können in der graphischen Oberfläche Automaten und Maschinen konstruieren und testen (z. B. endliche Automaten und Maschinen, Kellerautomaten, Turing-Maschinen). Automaten können in passende Grammatiken (oder reguläre Ausdrücke) umgewandelt werden und umgekehrt. Automaten und Grammatiken können auf ihren Typ getestet werden, sowie auf die Akzeptanz bzw. die Erzeugung von einzugebenden Wörtern. Nicht-Determinismus in den Automaten kann man sich hervorheben lassen. Daneben existieren Veranschaulichungen von beliebten Algorithmen: NFA[7] zu DFA[8], Minimierung von DFA, vollständige Automaten und einige mehr.
Die Benutzung ist einfach und mehr oder minder selbsterklärend. Ein Tutorial findet sich auf der JFLAP-Webseite [Rod], wo man sich auch das Programm selbst herunterladen kann.

Die aktuell stabile und für Lehrveranstaltungen empfohlene Version ist JFLAP7. An der Hochschule Hannover ist JFLAP7 unter der Leitung von Carsten Kleiner internationalisiert worden. Diese Version kann auf Nachfrage bereitgestellt werden.

Es existiert zudem eine Weiterentwicklung (JFLAP8-Beta), die einige Ungenauigkeiten und Fehler in JFLAP7 behebt. Leider sind die Formate nicht in allen Fällen abwärtskompatibel. Auch stehen nicht alle Funktionen von JFLAP7 zur Verfügung, dafür einige andere. Die oben genannten Funktionen gibt es aber in beiden Versionen.

An der Hochschule Hannover ist unter der Leitung von Frauke Sprengel ein Wrapper für JFLAP7 (mittlerweile auch JFLAP8) entwickelt worden ([Spr16], [Tos13], [Hel16]), der die Fähigkeiten des Programms für eine Grading Engine

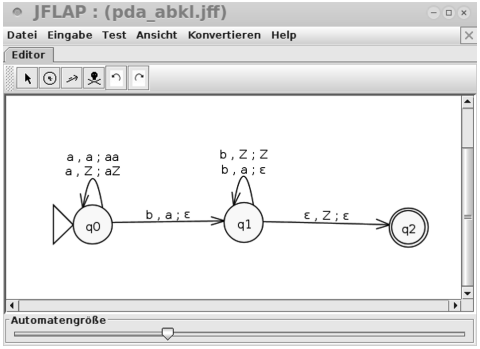


Abbildung 19.3: JFLAP7 mit deutscher Oberfläche (Kellerautomat).

7 NFA: Nondeterministic Finite Automaton, (nichtdeterministischer endlicher Automat)
8 DFA: Deterministic Finite Automaton, (deterministischer endlicher Automat)

benutzt, die innerhalb eines LMS wie z. B. LON-CAPA verwendet werden kann. Die Studierenden geben dazu einen String ein oder kopieren eine aus JFLAP exportierte XML-Datei in ein Textfeld. Die Bewertung von Automaten gleicht hierbei im Wesentlichen einer Programmbewertung, da sowohl „Syntax“ als auch Funktionalität des Automaten überprüft werden müssen.

Aktuell können z. B. folgende Arten von Fragen gestellt werden:

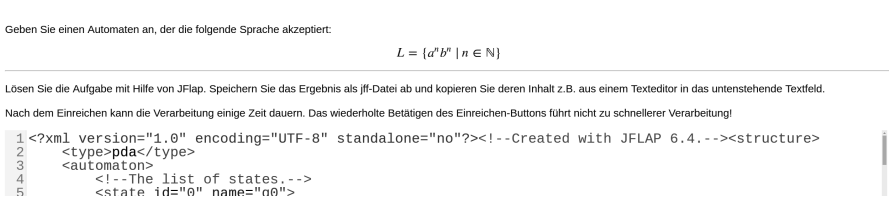
- Zu einer gegebenen Sprache (definiert z. B. durch eine Grammatik, einen regulären Ausdruck, eine Menge, eine Beschreibung als Text) soll ein akzeptierender Automat konstruiert werden (endlicher Keller-/Turing-Automat).

- Zu einer gegebenen Sprache (definiert z. B. durch einen Automaten, einen regulären Ausdruck, eine Menge, eine Beschreibung als Text, . . .) soll eine erzeugende Grammatik (eines bestimmten Typs) angegeben werden.

- Zu einer gegebenen Sprache (definiert z. B. durch eine Grammatik, einen Automaten, einen regulären Ausdruck, eine Menge, eine Beschreibung als Text, . . .) soll eine Anzahl von Wörtern angegeben werden.

- Zu einem vorgegebenen Typ von Automat oder Grammatik soll ein Beispiel gegeben werden.

Die im Fragentext gegebene Sprache wird an das Programm durch Angabe eines regulären Ausdrucks, einer Grammatik oder zwei Mengen von Wörtern (Wörter aus der Sprache bzw. nicht aus der Sprache) übergeben.


Geben Sie einen Automaten an, der die folgende Sprache akzeptiert:

$$L = \{a^n b^n \mid n \in \mathbb{N}\}$$

Lösen Sie die Aufgabe mit Hilfe von JFlap. Speichern Sie das Ergebnis als jff-Datei ab und kopieren Sie deren Inhalt z.B. aus einem Texteditor in das untenstehende Textfeld.

Nach dem Einreichen kann die Verarbeitung einige Zeit dauern. Das wiederholte Betätigen des Einreichen-Buttons führt nicht zu schnellerer Verarbeitung!

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><!--Created with JFLAP 6.4.--><structure>
    <type>pda</type>
    <automaton>
        <!--The list of states.-->
        <state id="0" name="q0">
```

Abbildung 19.4: LON-CAPA-Aufgabe (Ausschnitt): Hier ist der Kellerautomat aus Abbildung 19.3 gesucht.

19.4 Anbindung von JFLAP an LON-CAPA

19.4.1 Eine Beispielaufgabe

Die Sprache aus dem Beispiel der Abbildungen 19.3 und 19.4 wird erzeugt durch die kontextfreie Grammatik $S \to aSb \mid ab$. In einer LON-CAPA-Aufgabe, die den Automaten dazu verlangt, könnte man das Alphabet noch randomisieren und als Autor eine Aufgabe[9] verfassen wie in Abbildung 19.5.

In der ersten eingebundenen Bibliothek verbergen sich Funktionen für die Zusammenstellung aller nötigen Informationen für den JFLAP-Aufruf (`jflapUrlInput`), die Darstellung der studentischen Eingaben (`listOfSubmissions`) und der Rückgabe von JFLAP. Die dazu benötigten Angaben werden in den Abschnitten 19.4.4 und 19.4.5 beschrieben. In den beiden anderen Bibliotheken werden nur das Syntax-Highlighting im Eingabefeld und einige Ausgaben (wiederkehrender Text, Fehlerausgaben) erledigt.
Der externe Aufruf als solcher verwendet hier nur die Felder `answer` und `url`. Auf die zusätzliche Angabe von Key-Value-Paaren wurde hier verzichtet.

Als Ergebnis kann man dann in LON-CAPA die Ausgabe wie in Abbildung 19.6 erhalten.

19.4.2 JFLAP als externer Service

Die Anbindung von JFLAP erfolgte in starker Anlehnung an das Skript zur Beispielaufgabe für externe Aufrufe, die auf jedem LON-CAPA-Server[10] liegt.

Es wird ein Perl-Skript auf einem Webserver aufgerufen, das die nötigen Angaben aus dem <externalresponse>-Tag verarbeitet und eine XML-Struktur wie in Abschnitt 19.2.2 beschrieben zurückliefert:

```
 ... Vorbereitung
POST Variablen werden in %FORM geladen
...
my $call_jflap= "java -jar JFLAP.jar"
   . " \'$FORM{'LONCAPA_correct_answer'}\' "
   . " \'$FORM{'LONCAPA_student_response'}\'";
... Fehler und Timeout-Behandlung ...
```

9 In LON-CAPA zu finden unter: /res/fh-hannover/sprengel/Informatik/TheoretischeInformatik/PushDownAutomata/pda_anbn_jflap.problem.

10 z. B. https://loncapa.hs-hannover.de/res/adm/includes/templates/sampleexternal.pl

```
$result = `$call_jflap`;
print (<<ENDOUT);
$result
ENDOUT
exit;
```

```
<problem>
  <import Vorbereitung />
  <script type="loncapa/perl">
    @letter = ('a' .. 'g');
    $choice = random(0,$#letter-2,2);
    $a = $letter[$choice];
    $b = $letter[$choice+1];
    $given = 'S->'.$a.'S'.$b.'|'.$a.$b;
    $mode = 'agtw';
    $type = "pda";
    $numberOfWords = 20;
    $maxlength = 20;
    $tasktitle = "Automat für $a^n $b^n";
    ($externalurl,$input,$error) =
     jflapUrlInput($mode,$given,$type,
     $tasktitle,$numberOfWords,$maxlength);
    $listSubmission =  listOfSubmissions(JFC,JFlapCall);
  </script>
  <startouttext />
    Geben Sie einen Automaten an, der die folgende
    Sprache akzeptiert:
    <m eval="on">
     \[ L = \{ $a ^n $b ^n  \,|\, n \in
      \mathbb{N} \}
    </m>  <p />
  <endouttext />
  <import AutomatonHead>
  <part id="JFC">
     <externalresponse answer="$input" id="JFlapCall"
           url="$externalurl">
         <textfield spellcheck="none" />
     </externalresponse>
  </part>
  <import AutomatonFoot>
</problem>
```

Abbildung 19.5: LON-CAPA-Aufgabe: Codebeispiel für JFLAP-Aufgabe.

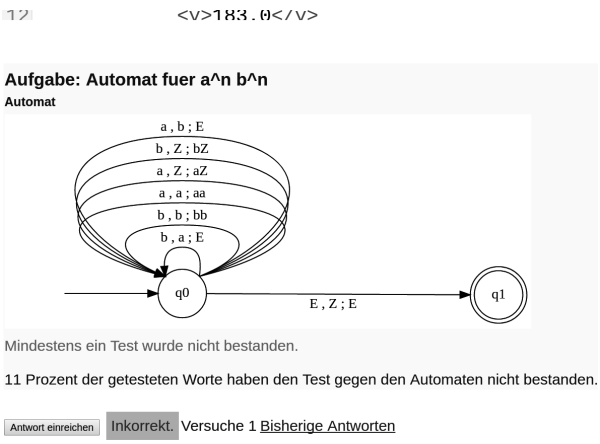


Abbildung 19.6: LON-CAPA-Ergebnis: Hier ist der Kellerautomat aus Abbildung 19.3 und 19.4 nicht sofort richtig eingegeben worden.

Wenn noch weitere Variablen in den POST-Variablen übermittelt werden, können diese auch verwendet werden. Die XML-Struktur wird hier direkt von JFLAP erzeugt. Für dieses Beispiel müssen auf dem Webserver neben Perl nur noch ein Java Runtume Environment (JRE) und die Graphhviz-Bibliothek zur Darstellung der Automaten als SVG installiert sein.

19.4.3 Einzubindende Bibliotheken

Alle im Folgenden beschriebenen Bibliotheken stehen als Sourcecode[11] zur Verfügung. Benutzt werden muss auf jeden Fall

JFlap_call_preparation.library: Hier erfolgt die gesamte Vorbereitung. Der Name des Servers ist dort im Code hinterlegt und muss geändert werden für einen eigenen Server.

Die restlichen Bibliotheken müssen nicht benutzt werden, beinhalten aber häufig verwendeten Code und sorgen für das Syntaxhighligthing und die Darstellung der Ergebnisse:

JFlap_call_automaton_head.library, JFlap_call_automaton_foot.library: Header und Footer für den Aufgabenteil mit dem externen Aufruf von JFLAP.

11 In LON-CAPA zu finden unter: /res/fh-hannover/sprengel/Informatik/TheoretischeInformatik/Libraries, dort findet man auch Beispielaufgaben.

Sorgt für die Ausgabe wiederkehrende Formulierungen und Syntax-Highlighting der JFLAP-Datei und die Darstellung der Ergebnisse.

jff_automaton_parts_head.library, jff_automaton_parts_foot.library: Header und Footer für einen Aufgabenteil, der die Eingabe der JFLAP-Datei liest und die Komponenten des Automaten abfragt (für endliche und Kellerautomaten).

JFlap_call_grammar.library: Aufgabenteil (sorgt für die Ausgabe wiederkehrender Formulierungen und den Aufruf von JFLAP für Grammatiken).

JFlap_call_words_head.library, JFlap_call_words_foot.library: Header und Footer für einen Aufgabenteil, in dem nach Beispielwörtern einer gegebene Sprache gefragt wird.

19.4.4 Funktion für die URL und die Parameter des JFLAP-Aufrufs

Im Perl-Skript der LON-CAPA-Aufgabe sollte sich solch ein Funktionsaufruf befinden:

```
($externalurl,$input,$error) =
     jflapUrlInput($mode,$given,$type,$tasktitle,
                   $numberOfWords,$maxlength);
```

Dabei ist es wichtig, die Ausgabevariablen `($externalurl, $input, $error)` genauso zu benennen, wie im obigen Aufruf in der External-Response-Aufgabe aus dem letzten Abschnitt. Sie enthalten dann

- `$externalurl`: Link zum aufzurufenden Perl-Skript,
- `$input`: im Wesentlichen eine Konkatenation der nötigen Parameter, einem Eingabeparameter für JFLAP,
- `$error`: falls der Aufruf fehlschlägt, kann dieser String in einem Instructor-Comment für den Aufgabenautoren oder Kurskoordinator angezeigt werden.

Für den Aufruf der Funktion `jflapUrlInput` sind die folgenden Parameter nötig:

- `$mode` (String): Einer der Modi wie im nächsten Abschnitt 19.4.5 beschrieben.

- `$given` (String): Regulärer Ausdruck oder Grammatik zur Beschreibung der Sprache aus dem Aufgabentext (falls nur ein Beispiel gefragt wird: leerer String).

Dazu kommen noch optionale Parameter, die von hinten nach vorn weggelassen werden können

- `$type` (String): Typ der Grammatik oder des Automaten (Abschnitt 19.4.5), Default: non.
- `$tasktitle` (String): Titel der Aufgabe, Default: JFlap-Aufgabe oder JFlap task, in Abhängigkeit von der Sprache
- `$numberOfWords` (Integer): Anzahl der zu generierenden Wörter, mit denen getestet werden soll, Default: 10.
- `$maxlength` (Integer): Maximale Länge der zu generierenden Wörter, mit denen getestet werden soll, Default: 10.

19.4.5 Modi und Typen

Die JFLAP Grading Engine kann bisher mit folgenden Modi benutzt werden:

- `ar[t][w]` (Automaton, Regex[, Type][, Words]): Gesucht ist ein endlicher Automat, interne Beschreibung der Sprache als regulärer Ausdruck.
- `ag[t][w]` (Automaton, Grammar[, Type][, Words]): Gesucht ist ein Automat, interne Beschreibung der Sprache als Grammatik.
- `gg[t][w]` (Grammar, Grammar[, Type][, Words]): Gesucht ist eine Grammatik, interne Beschreibung der Sprache als Grammatik.
- `egt` (Example, Grammar, Type): Gesucht ist ein Beispiel für eine Grammatik von bestimmtem Typ.
- `eat` (Example, Automaton, Type): Gesucht ist ein Beispiel für einen Automaten von bestimmtem Typ.

Bei der Angabe von `t` sollte ein bestimmter Typ verlangt werden. Bei der Angabe von `w` werden die Wörter auf LON-CAPA-Seite erzeugt.

Automatentypen sind (entweder mit `d`=deterministisch oder `n`=nichtdeterministisch oder ohne=egal) `[d/n]fa` für endliche Automaten, `[d/n]pda` für Kellerautomaten, `[d/n]ta` für Turing-Automaten oder `non` nicht angegeben.

Grammatiktypen sind wie erwartet `rl` rechtslinear, `rlcfg` rechtslinear oder kontextfrei, `cfg` kontextfrei, aber nicht rechtslinear, `ncfg` nicht kontextfrei oder `non` nicht angegeben.

19.4.6 Reguläre Ausdrücke und Grammatiken

Die regulären Ausdrücke werden entweder in Java (ohne `w`) oder in Perl (mit `w`) verarbeitet. Somit hat man viele Formulierungsmöglichkeiten. Als Terminale sind alle Kleinbuchstaben (`a` ... `z`) sowie `0,1` erlaubt.

Eine Grammatik wird für die Verarbeitung in JFLAP wie folgt als String geschrieben:

```
S -> aB, B -> bS | E | a, S -> aSa
```

Als Konvention wird hiernach verwendet: Kommata zur Trennung der Regeln, ein Pfeil (`->`) zwischen linker und rechter Seite der Regel, der senkrechte Strich | zur Trennung mehrerer rechter Seiten, ein großes `E` (statt ε) für das leere Wort, ein großes `S` als Startsymbol, alle anderen Großbuchstaben als Nichtterminale, alle Kleinbuchstaben (`a` ... `z`) und `0,1` als Terminale. Dazwischen sind beliebig viele Leerzeichen erlaubt.

Innerhalb der LON-CAPA-Aufgabe kann man wie gewohnt die Perl-Funktionen zur Randomisierung z. B. der Buchstaben des Alphabets der Grammatik oder des regulären Ausdrucks benutzen.

19.5 Fazit

In LON-CAPA ist die Anbindung externer Grader für Programmbewertung oder ähnliches durch die Problemart External-Response sehr einfach möglich. Dem aufgerufenen Grader können neben der studentischen Eingabe weitere Parameter als POST-Request übergeben werden. Studentische Programmeingaben sind nur einzeln und nur über das Textfeld und nicht als hochzuladende Dateien möglich. Rückgabeinformationen sind strukturiert in einer XML-Nachricht möglich und durch JavaScript formatierbar.

Literatur für dieses Kapitel

[Gro15] LON-CAPA Group. *Learning Online Network with CAPA – Author's Tutorial And Manual*. Michigan State University, Mai 2015. URL: https://loncapa.msu.edu/adm/help/author.manual.pdf.

[Hel16] Benjamin Held. *Erweiterung einer Bewertungssoftware für LON-CAPA unter Verwendung von JFLAP*. Masterarbeit. Hannover, 2016.

[Kor+08] Gerd Kortemeyer u. a. „Experiences using the open-source learning content management and assessment system LON-CAPA in introductory physics courses". en. In: *American Journal of Physics* 76.4 (2008).

[RF06] Susan H. Rodgers und Thomas W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Sudbury, MA: Jones & Bartlett Publishers, 2006.

[Rod] Susan H. Rodgers. *JFLAP web page*. URL: www.jflap.org.

[Spr16] Frauke Sprengel. *Electronic Exercises in Theoretical Computer Science*. (verfügbar in LON-CAPA /res/fh-hannover/sprengel). Hochschule Hannover, 2016.

[Tos13] Ufuk Tosun. *Automatische Bewertung von Studierendenabgaben in der theoretischen Informatik auf Basis von JFLAP*. Bachelorarbeit. Hannover, 2013.

20 Integration automatisierter Programmbewertung in Stud.IP

Elmar Ludwig

Zusammenfassung
Das Lernmanagementsystem Stud.IP stellt – was ungewöhnlich ist – selbst noch keine Funktionen zur Erstellung von Aufgaben bereit, es gibt allerdings verschiedene Erweiterungsmodule, die in das System installiert werden können, um Aufgaben- und Bewertungsfunktionen nachzurüsten. Eine dieser Komponenten ist das „Vips“- Modul, das bereits eine Schnittstelle zu dem VEA Grader für Prolog (und einige andere Programmiersprachen) mitbringt. Die Einbindung dieser Schnittstelle in Vips als Stud.IP-Modul soll im Folgenden vorgestellt werden.

20.1 Stud.IP

Das Lernmanagementsystemen Stud.IP entstand 1999 an der Universität Göttingen als internetbasiertes System zur Unterstützung von Lehrveranstaltungen. Anfangs noch auf wenige Funktionen rund um die Themen Kommunikation (Forum), Datenaustausch (Dateiordner) und Organisation (Stundenplan) fokussiert, entstand im Laufe der Zeit ein umfangreiches Lernmanagementsystem, das mittlerweile von einer größeren Zahl von Hochschulen und Universitäten, aber auch außeruniversitäten Bildungseinrichtungen zur Organisation und Unterstützung des Lehrangebots eingesetzt wird.

Neben den klassischen Funktionen eines Lernmanagementsystems hinsichtlich der onlinebasierten Durchführung einer Lehrveranstaltung gibt es auch einzelne Funktionen aus dem Bereich des Campusmanagement – z. B. eine umfangreiche Raum- und Ressourcenverwaltung sowie Funktionen zur Personaldatenverwaltung, die Stud.IP ein breites Einsatzspektrum ermöglichen. Infolgedessen ist der Grad der Nutzung der einzelnen Komponenten an den verschiedenen Standorten zum Teil sehr unterschiedlich.

Zur tiefergehenden Unterstützung von elektronischen Lehrmaterialien gibt es noch eine Schnittstelle zum Lernmanagementsystem ILIAS (siehe Kapitel 21). Eine vergleichbare Schnittstelle zu Moodle ist gerade in der Entwicklung.

20.2 Das Vips-Modul in Stud.IP

Das onlinebasierte System Vips entstand aus einem Vorläufer, dem „Virtual Campus PROLOG Tutor“ ([Pey+00]), der ab 1997 an der Universität Osnabrück entwickelt und für die Ausbildung in der Programmiersprache Prolog eingesetzt wurde. Anfangs als reine Prolog-Umgebung konzipiert, wurde das System im Laufe der Zeit um die Funktionen eines vollwertigen E-Assessment-Systems erweitert, so dass heute eine breite Auswahl an unterschiedlichen Aufgabentypen und Einsatzszenarien unterstützt wird.

Seit dem Sommersemester 2004 ist es unter dem Namen „Vips“ (Abkürzung für „Virtuelles Prüfungssystem“) als onlinebasiertes Übungs- und Prüfungssystem in der Lehr- und Lernplattform Stud.IP verfügbar (siehe [Hüg+05]), in den ersten Versionen nur an der Universität Osnabrück, später dann auch an anderen Hochschulen. Als modulare Erweiterung kann es als Stud.IP-Plugin installiert und dann in einer Lehrveranstaltung von den Lehrenden als Inhaltselement aktiviert werden.

Das Modul bietet den Dozenten verschiedene Möglichkeiten zur Entwicklung und Verwaltung von Aufgabenblättern und Klausuren, inklusive der Unterstützung der eigenständigen Lernfortschrittskontrolle durch die Studierenden anhand von Selbsttests. Einige Aufgabentypen können vollautomatisch ausgewertet werden, bei anderen gibt es zumindest eine technische Unterstützung für den Auswertungsprozess, z. B. in Form von Bewertungsvorschlägen anhand der Ähnlichkeit einer abgegebenen Lösung zu einer erwarteten korrekten Lösung. Integriert ist auch eine Schnittstelle zum Import und Export von Aufgaben und Aufgabensammlungen in verschiedenen Formaten.

Studierende können Vips nutzen, um von Dozenten bereitgestellte Aufgabenblätter zu bearbeiten, entweder als Selbsttest mit unmittelbarem Feedback, in Form klassischer Hausaufgaben mit nachgelagerter (teils manueller) Bewertung und Freigabe durch Lehrende oder Tutoren oder auch in Form einer echten Online-Klausur – dann in der Regel unter Aufsicht in einem speziellen PC-Raum.

Vips bietet beim Erstellen von Tests eine Auswahl aus verschiedenen Aufgabentypen an:

Single Choice Eine Frage mit Auswahl aus einer Liste von vorgegebenen Antworten, nur eine Antwort kann gewählt werden.

Multiple Choice Eine Frage mit Auswahl aus einer Liste von vorgegebenen Antworten, mehrere Antworten können gewählt werden.

Ja/Nein Frage Eine spezielle Variante der Single Choice Aufgabe mit den Antwortmöglichkeiten „Ja“ und „Nein“.

Freitext Eine Frage mit freier Anwort (einzeilig oder mehrzeilig), die nur eingeschränkt automatisch ausgewertet werden kann.

Lückentext Ein Fragetext, in dem Lücken durch die Teilnehmer auszufüllen sind. Hier ist eine automatische Auswertung in den meisten Fällen möglich.

Mathematischer Term Eine Frage mit freier Anwort, die einen algebraischen Term erwartet, z. B. eine Formel. Die Antwort kann in den meisten Fällen automatisch ausgewertet werden.

Zuordnung Eine Aufgabe, bei der eine Liste von Begriffen, Texten oder Bilder einer Menge von Kategorien zugeordnet oder in eine bestimmte Reihenfolge gebracht werden soll.

Programmieraufgabe Aufgabe, die zur Lösung die Erstellung eines Programms erwartet. Aktuell werden von Vips nur Prolog Aufgaben unterstützt, weitere Grader zur Auswertung anderer Programmiersprachen sollen in Zukunft über die ProFormA-Middleware angebunden werden.

Das System bietet dabei umfangreiche Möglichkeiten zur Entwicklung, Pflege und Auswertung elektronischer Aufgaben sowie die Verwaltung des gesamten Übungsbetriebs eines Kurses. Dabei gibt es grundsätzlich Übungsblätter und Klausuren als Aufgabensammlungen innerhalb von Vips. Beide unterscheiden sich nicht in der Struktur der Aufgaben, sondern nur im Modus des Ablaufes und der Zuordnung zu einzelnen Teilnehmern (bei Klausuren) bzw. zu Arbeitsgruppen (bei Übungsblättern). Zudem gibt es für Übungsblätter einen Selbsttestmodus, bei dem einem Kursteilnehmer nach dem Absenden einer Lösung sofort das Ergebnis der automatischen Auswertung präsentiert wird. Der Klausurmodus erlaubt eine Online-Überprüfung der Zugriffsparameter sowie eine Überwachung des Arbeitsfortschritts der Teilnehmer z. B. durch die Aufsicht.

Vips umfasst ebenfalls eine Arbeitsgruppenverwaltung (Aufgaben werden dann von mehreren Teilnehmern gemeinsam gelöst), Punkte- und Notenübersichten für einzelne Teilnehmer und Arbeitsgruppen im Kurs, sowie eine flexible Notenberechnung. Darüber hinaus gibt es auch statistische Übersichten über die Anzahl

der korrekten bzw. falschen Lösungen der Teilnehmer bei den einzelnen Aufgaben bzw. Aufgabenteilen. Aufgabenblätter und Klausuren können auch als Text- bzw. XML-Dateien importiert und exportiert werden. Darüber hinaus gibt es eine Druckansicht zum Ausdrucken der leeren Aufgabenformulare, der eigenen Lösung (für die Teilnehmer selbst) oder auch aller Lösungen im Kurs (für die Dozenten). Bild 20.1 zeigt die Ansicht zum Anlegen und Verwalten von Aufgabenblättern in Vips.

Im Folgenden wird insbesondere auf Programmieraufgaben und die automatische Unterstützung bei deren Auswertung eingegangen.

20.3 Programmieraufgaben in Vips

Vips stellt für Programmieraufgaben eine Runtime-Umgebung mit einer einfachen, webbasierten Oberfläche zur Verfügung. Diese GUI ist für die Teilnehmer direkt über einen Reiter einer Lehrveranstaltung erreichbar. Sie ermöglicht für Programmieraufgaben den Programmcode einzugeben und zu editieren sowie die Programme in einer definierten Umgebung auszuführen. So können Kursteilnehmer die Aufgaben lösen, ohne die Programmiersprachen lokal auf ihrem Rechner installieren zu müssen. Auch muss bei Online-Tests in PC-Räumen der Hochschule keine spezielle Umgebung zur Ausführung der Programme bereitgestellt werden. Allerdings stehen interaktive Debug-Möglichkeiten in einer solchen Umgebung nicht zur Verfügung – es werden nur die Meldungen des Compilers bzw. Interpreters bei der Ausführung zurückgemeldet. Um für komplexe Aufgabenstellungen auch lokale Installationen zum Aufgabenlösen benutzen zu können, gibt

Abbildung 20.1: Anlegen und Verwalten von Aufgabenblättern

es Down- und Upload-Möglichkeiten für die Aufgaben. In diesem Fall passiert dann oft nur noch die finale Abgabe der Lösung in Vips. Hierbei sollte aber durch die Lehrenden sichergestellt werden, dass die Teilnehmer auch lokal mit einer kompatiblen Version des Interpreters arbeiten, da die spätere Auswertung in Vips auf dem zentralen Auswertungssystem passiert und sonst ggf. andere Ergebnisse liefern könnte.

Die gleiche Runtime-Umgebung wie den Teilnehmern steht auch den Dozenten und Tutoren für die Vorbereitung der Aufgaben sowie bei der Aufgabenkorrektur zur Verfügung. Zusammen mit einem vorgegebenen Default-Aufruf der Hauptfunktion des Programms ermöglicht dies sowohl dem Kursteilnehmer als auch dem Korrektor, sich mit einem Mausklick einen ersten Überblick über die Lauffähigkeit der Lösung zu verschaffen. Eventuelle Fehler bei der Übersetzung oder Auswertung werden dem Teilnehmer angezeigt.

20.3.1 Anlegen eines Übungsblattes mit Programmieraufgaben

Um ein Übungsblatt mit Programmieraufgaben in Vips anzulegen, gibt es unter dem Reiter „Vips“ einer Lehrveranstaltung mehrere Aktionen. Der Dozent kann ein neues Übungsblatt anlegen, ein Übungsblatt aus einer Text- oder XML-Datei importieren oder die Daten aus bereits vorhandenen Übungsblättern übernehmen. Ist das Übungsblatt in der Oberfläche mit Titel und Beschreibung sowie Start- und Endzeitpunkt versehen, können Übungsaufgaben neu hinzugefügt werden. Alternativ hat der Dozent hier auch die Möglichkeit, Aufgaben aus bereits im Stud.IP vorhandenen Übungsblättern dieses oder anderer Kurse zu übernehmen. Bild 20.2 zeigt ein bereits angelegtes Übungsblatt mit mehreren Aufgaben. Vips stellt für eine Programmierübungsaufgabe, die einem Aufgabenblatt hinzugefügt wird, neben der Aufgabenstellung eine Reihe von Informationsfeldern bereit (siehe Abbildung 20.3):

- Ein Lösungsfeld, in das der Teilnehmer den zu entwickelnden Code eingeben kann. Dieses Feld kann vorbelegt werden, so dass dem Kursteilnehmer bereits ein Lösungsgerüst präsentiert werden kann, das z. B. vom Bearbeiter ergänzt bzw. geändert oder korrigiert werden muss.

- Ein Feld für Musterlösungen. Dieses Feld kann mehrere Musterlösungen enthalten. Zudem enthält dieses Feld noch zusätzliche Informationen für die automatische Auswertung durch den Prolog-Grader. Das kann z. B. ein Testprogramm sein oder spezifische Angaben dazu, wie die Auswertung er-

folgen soll – beispielsweise nur durch einen strukturellen Vergleich mit den Musterlösungen. In Zukunft werden diese Informationen zur Auswertung in Vips in einem eigenen Textfeld verwaltet.

- Ein Feld mit dem Aufruf der Hauptfunktion des Programms. In Prolog ist dies eine Folge von Subgoals durch Kommas getrennt. In anderen Programmiersprachen kann dies ein Prozedur- bzw. Funktionsaufruf oder ein arithmetischer Ausdruck sein.

Da das Erscheinungsbild einer Aufgabenspezifikation aus der Sicht der Kursteilnehmer nicht immer einfach vorhergesehen werden kann, gibt es eine direkte Möglichkeit zwischen der Darstellung für die Aufgabenentwicklung und der Darstellung für den Kursteilnehmer umzuschalten. In der Darstellung aus Teilnehmersicht kann der Aufgabenersteller – wie die Teilnehmer später auch – die integrierte Runtime-Umgebung zum Testen der Aufgabe verwenden.

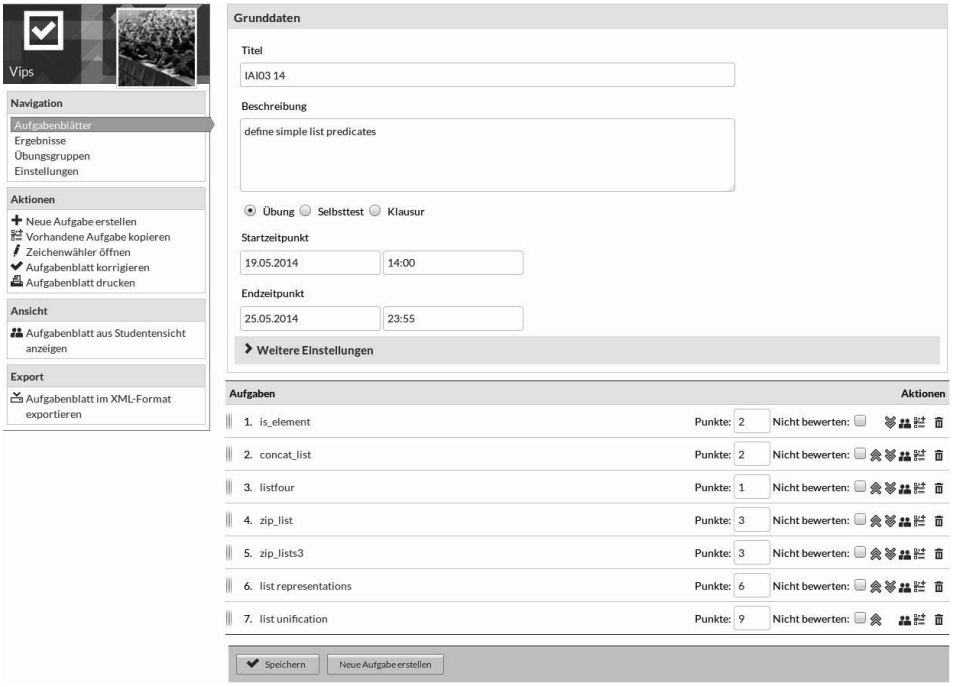


Abbildung 20.2: Übungsblatt mit mehreren Aufgaben

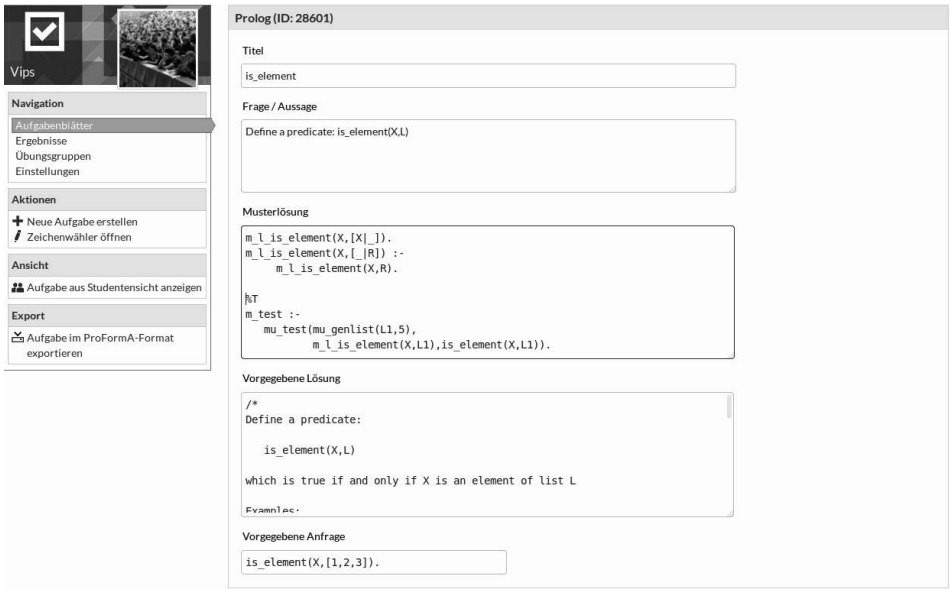


Abbildung 20.3: Erstellung einer Programmierübungsaufgabe

20.3.2 Bearbeiten eines Übungsblattes mit Programmieraufgaben

Generell können Programmieraufgaben, wie alle anderen Aufgabentypen auch, innerhalb des vom Dozenten beim Erstellen des Aufgabenblattes festgelegten Zeitraums bearbeitet werden. Aus Kursteilnehmersicht bietet die GUI (siehe Abbildung 20.4) dabei folgende Möglichkeiten:

- Der Kursteilnehmer kann seine Lösung in ein Textfeld eintragen. Dieser Bereich ist eventuell bereits mit einem Lösungsgerüst vorbelegt, das vom Teilnehmer erweitert oder korrigiert werden soll.

- Basierend auf der aktuellen Lösung im Textfeld kann eine Testanfrage gestellt werden. Diese wird auf dem Auswertungsserver ausgeführt und das Ergebnis des Programmlaufs angezeigt. Gab es Fehler bei der Ausführung, werden diese ebenfalls hier angezeigt.

- Der Inhalt des Lösungsbereichs kann in eine lokale Datei heruntergeladen werden, so dass eine lokale Installation der Programmiersprache und/oder

ein umfangreicherer Editor oder eine IDE zur Programmentwicklung benutzt werden kann.

- Anschießend kann die fertige Lösung wieder in das Bearbeitungsfeld hochgeladen werden.

- Über den Button *Abschicken* wird die eigene Lösung abgegeben. Übungsaufgaben können im Rahmen der Bearbeitungszeit mehrfach abgegeben werden, die als letztes abgegebene Lösung zählt.

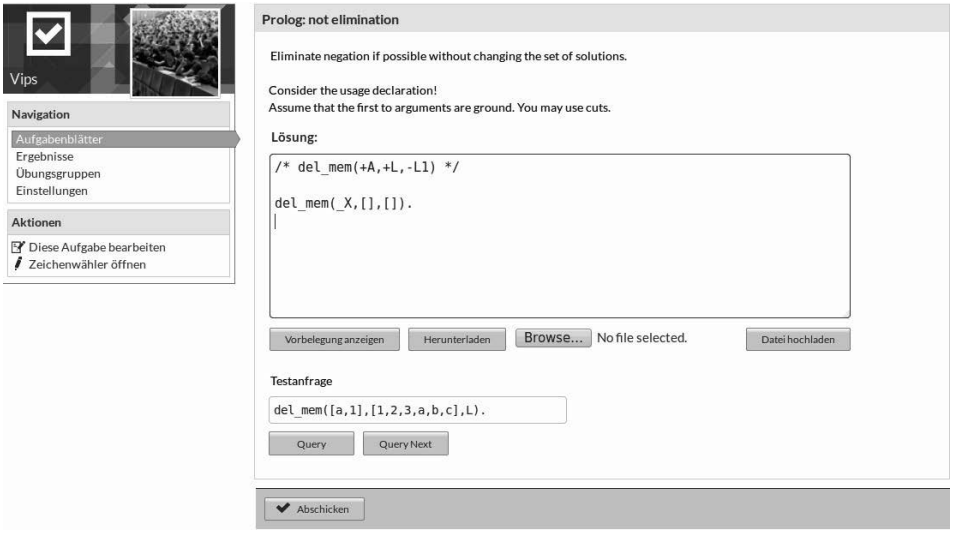


Abbildung 20.4: Bearbeitung einer Programmieraufgabe in Vips

Kursteilnehmer können in Arbeitsgruppen organisiert sein. Dann gilt bei einem Aufgabenblatt eine abgegebene Lösung für die gesamte Arbeitsgruppe, wobei jeder Teilnehmer der Gruppe die Gruppenlösung einsehen kann. Die jeweils letzte Lösung, die einer der Gruppenteilnehmer dabei abgegeben hat, zählt bei der Auswertung. Bei einer Klausur werden keine Gruppen berücksichtigt, hier arbeitet immer jeder Teilnehmer für sich.

Im Selbsttestmodus wird nach der Abgabe die Aufgabe automatisch ausgewertet und die Auswertung sowie die (erste) Musterlösung angezeigt. Ansonsten erfolgt die Auswertung durch den Dozenten und ist erst einsehbar, wenn die Bewertungen freigegeben wurden.

20.3.3 Auswertung eines Übungsblattes mit Programmieraufgaben

Alle abgegebenen Übungsblätter werden dem Dozenten in Vips in einer Übersicht präsentiert. Von hier aus gelangt er zu den einzelnen Übungsaufgaben bzw. den Lösungen der Teilnehmer.

Die Auswertung eines Übungsblattes beginnt normalerweise mit dem Aufruf der Autokorrekturfunktion, die alle automatisch auswertbaren Aufgabentypen zu bewerten versucht. Bei Programmieraufgaben liefert die automatische Auswertung aber oft nur einen Indikator für die Bewertung, daher stellt die Vips-Oberfläche im Korrektur- und Bewertungsmodus ähnliche Funktionen wie bei der Bearbeitung durch die Studierenden zur Verfügung – inklusive der Runtime-Umgebung zur Nutzung durch den Dozenten, hier aber noch erweitert um die Möglichkeiten zur Annotation der Lösung und zur Punktevergabe. Innerhalb des Auswertungsmodus einer Programmieraufgabe gibt es folgende Funktionen:

- Im abgegebenen Programmcode kann korrigiert und kommentiert werden. Dies kann beispielsweise nützlich sein, wenn die vom Teilnehmer abgegebene Lösung einen Syntaxfehler enthält, der eine automatische Bewertung des Progamms verhindert. Der Dozent oder Tutor kann dann diesen Fehler korrigieren und die Bewertung anschließend noch mal durchführen.

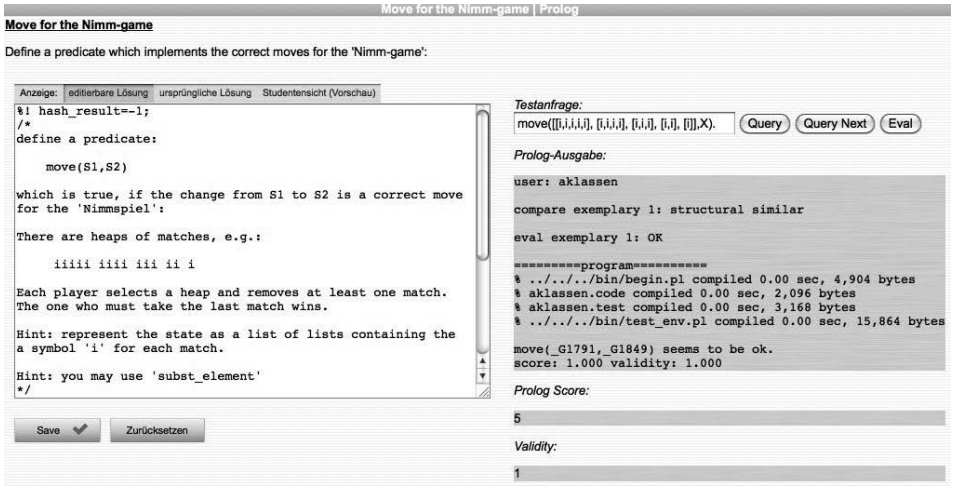


Abbildung 20.5: Auswertungsmodus einer Programmieraufgabe in Vips

Die Originallösung des Teilnehmers wird dabei aber nicht verändert – diese bleibt zur Dokumentation der abgegeben Lösung unverändert, die annotierte Lösung wird immer separat gespeichert.

- In einem Textfeld können allgemeine Kommentare zur Lösung oder Erläuterungen zur Bewertung hinzugefügt werden.

- Für die Lösung können Punkte vergeben werden. Dabei werden zunächst als Vorschlag die von der automatischen Auswertung vergebenen Punkte angezeigt.

- Der *Query*-Knopf sendet den Inhalt des Lösungsfelds an den Auswertungsserver und ruft dort die Hauptfunktion auf. Die Ergebnistexte der Kompilation und der Programmausführung werden anschließend als Text angezeigt. Für nichtdeterministische Sprachen wie Prolog gibt es noch den Knopf *Query Next*, der bei jeder Betätigung eine weitere Lösung des Programms abfragt und ausgibt.

- Der *Eval*-Knopf stößt die automatische Bewertung an (siehe Abbildung 20.5). Der Bewertungprozess selbst kann dabei über spezielle Schlüsselworte in der Musterlösung gesteuert werden, so kann z. B. die Bewertung nur auf einen Textvergleich beschränkt oder durch ein Testprogramm vorgenommen werden – die Details dazu sind im Kapitel 14 beschrieben.

  Als Ergebnis werden u. a. zwei Zahlen zwischen 0 und 1 angezeigt: Ein *Score*, der die Güte der Lösung repräsentiert und ein *Validitätswert*, der die Verlässlichkeit der Bewertung signalisiert. Bei einem Validitätswert von 1 kann die Bewertung als sicher angesehen werden. Als Bewertungskriterien können die I/O-Relation (also das Verhalten) des Programms sowie textuelle Vergleiche des Lösungscodes mit den Musterlösungen auf verschiedenen Normalisierungsebenen herangezogen werden.

20.4 Im- und Export

Zusätzlich zu der Möglichkeit, Aufgaben direkt im System zwischen Nutzern auszutauschen, lassen sich auch alle Aufgaben in verschiedenen Formaten importieren und exportieren. Für den Im- und Export von Programmieraufgaben werden im wesentlichen zwei Formate unterstützt:

Vips-Format

Das Vips-Format ist ein XML-basiertes Format speziell für den Datenaustausch von Aufgaben zwischen verschiedenen Vips-Installationen. Daneben kann auch das ältere, nicht XML-basierte Textformat weiterhin eingelesen werden, auf das hier nicht weiter eingegangen wird.

ProFormA-Task

Das ProFormA-Format ist ein Austauschformat speziell für Programmieraufgaben, das auch von anderen Werkzeugen unterstützt wird. Bisher ist Vips allerdings das einzige System, das Grader-Unterstützung für Aufgaben in Prolog enthält.

Daneben gibt es in Vips noch einen Export in das QTI 2.1 Format, der allerdings nicht alle Aufgabentypen darstellen kann – insbesondere werden von QTI keine Programmieraufgaben unterstützt.

20.4.1 Vips-Format

Das Vips-Format ist ein eigenes, XML-basiertes Austauschformat für alle Aufgabentypen. Das Schema ist auf Basis eines Aufgabenblattes definiert (d. h. einer Aufgabensammlung), der Export einzelner Aufgaben ist dabei eigentlich nicht vorgesehen. Daher eignet es sich auch nur bedingt zum Austausch isolierter Aufgaben mit anderen Werkzeugen bzw. Gradern – hierfür ist das im nächsten Kapitel dargestellte ProFormA-Format besser geeignet.

Generell besteht die Darstellung einer Programmieraufgabe aus verschiedenen Metadaten (Titel, Aufgabentext, Aufgabentyp) sowie dem enthaltenen Programmcode. Im Einzelnen sind das:

- Die Textvorgabe, die Grundlage für die Lösung der Teilnehmer ist.
- Eine oder mehrere Musterlösungen mit Angaben dazu, wie gut eine solche Lösung zu bewerten ist.
- Sowie optional auch Tests, die in die automatische Bewertung eingehen (in dem folgenden Beispiel nicht zu sehen).

Schließlich gibt es noch die Möglichkeit, einen Programmaufruf zu hinterlegen, mit dem die Teilnehmer in der interaktiven Runtime-Umgebung in Vips während der Entwicklung ihre Lösungen ausprobieren können.

Abbildung 20.6 zeigt exemplarisch den Export einer Programmieraufgabe in Prolog.

```
<exercise id="exercise-37636">
  <title>
    Reverse Elements of a List
  </title>
  <description>
    <text>
      Define a predicate 'rev(L1,L2)' which is true if L2
      contains the elements of L1 in reverse order.
    </text>
  </description>
  <items>
    <item type="program-prolog">
      <answers>
        <answer score="0" default="true">
          <text>
            rev( )
          </text>
        </answer>
        <answer score="1">
          <text>
            m_l_rev([], []).
            m_l_rev([X | Rest], Result) :-
            m_l_rev(Rest, RevRest),
            append(RevRest, [X], Result).
          </text>
        </answer>
      </answers>
      <evaluation-hints>
        <input-data type="prolog-query">
          time(rev([1,2,3,4,5,6,7,8,9,0], L)).
        </input-data>
      </evaluation-hints>
    </item>
  </items>
</exercise>
```

Abbildung 20.6: Beispiel für den Export einer Aufgabe im Vips-Format

20.4.2 ProFormA-XML

Das Exportformat ProFormA-XML wird in Vips primär zum Austausch von Aufgaben mit anderen Lernplattformen und zur Kommunikation mit der Middleware bei der zukünftigen Anbindung weiterer Grader verwendet – die Schnittstelle zum Grader VEA benutzt noch nicht das ProFormA-XML-Format. Dieses Format ist immer aufgabenbezogen, d. h. es kann (anders als beim zuvor beschriebenen Vips-Format) auch immer nur jeweils eine Aufgabe in einer Datei beschrieben werden.

```
<task xmlns="urn:proforma:task:v0.9.4" lang="en">
  <description>
    <h1>Reverse Elements of a List</h1>
    Define a predicate 'rev(L1,L2)' which is true if L2
    contains the elements of L1 in reverse order.
  </description>
  <proglang version="SWI-Prolog 5.10.1">
    prolog
  </proglang>
  <files>
    <file id="answerDefault" class="template" filename="ad.pl"
    type="embedded">
      rev( )
    </file>
    <file id="queryDefault" class="inputdata" filename="qd.pl"
    type="embedded">
      time(rev([1,2,3,4,5,6,7,8,9,0], L)).
    </file>
    <file id="m_l_0" class="internal" filename="m_l_0.pl" type="embedded">
      m_l_rev([], []).
      m_l_rev([X | Rest], Result) :-
      m_l_rev(Rest, RevRest),
      append(RevRest, [X], Result).
    </file>
    <file id="test" class="internal" filename="test.pl" type="embedded">
      m_test :-
      mu_test(mu_genlist(L1,5),
      m_l_rev(L1,L2),rev(L1,L2)).
    </file>
  </files>
  <model-solutions>
    <model-solution id="m_s_0" filename="m_s_0.pl" type="embedded">
      rev([], []).
      rev([X | Rest], Result) :-
      rev(Rest, RevRest),
      append(RevRest, [X], Result).
    </model-solution>
  </model-solutions>
  <tests>
    <test id="test_0" validity="1">
      <title>Test</title>
      <test-type>
        prolog-eval-similarity
      </test-type>
      <test-configuration>
        <filerefs>
          <fileref refid="answerDefault" />
          <fileref refid="m_l_0" />
          <fileref refid="test" />
        </filerefs>
      </test-configuration>
    </test>
  </tests>
</task>
```

Abbildung 20.7: Beispiel für den Export einer Aufgabe im ProFormA-Format

Weitere Informationen zum ProFormA-XML-Format selbst sind in Kapitel 24 zu finden.

Neben den bereits im Vips-Format gezeigten Metadaten zur Aufgabe kann in ProFormA-XML noch die Information zur Version des verwendeten Interpreters (hier: *SWI-Prolog 5.10.1*) explizit hinterlegt werden.

Dies ist wichtig, damit bei der Auswertung sichergestellt werden kann, dass auch ein mit dem Code in der Aufgabe kompatibler Interpreter verwendet wird.

Nach den Metadaten folgt eine Liste von Programmen, denen jeweils unterschiedliche Rollen zugewiesen sein können – eine Vorbelegung für die Teilnehmer, eine Liste von Musterlösungen sowie ein Beispielaufruf für die interaktive Runtime-Umgebung in Vips. Anschließend folgt die Liste der Tests für die automatische Programmbewertung. Generell sind bei aus Vips exportierten Aufgaben diese Programme immer in die XML-Struktur eingebettet (*type="embedded"*), auch wenn das Format die Auslagerung in separate Dateien ermöglichen würde.

Jeder aufgelistete Test kann dabei noch spezifische Optionen für die Auswertung enthalten, so gibt es z. B. folgende Prolog-spezifische Werte für den *test-type*:

prolog-similarity Auswertung ausschließlich anhand der Textähnlichkeit zu einer der Musterlösungen.

prolog-eval Auswertung ausschließlich anhand der Ausgabe des Testprogramms.

prolog-eval-similarity Auswertung anhand beider obiger Kriterien.

Abbildung 20.7 zeigt den Export einer Prolog-Programmieraufgabe in ProFormA-XML. Dieses Format wird zukünftig auch für die Anbindung an den ProFormA-Server (siehe Kapitel 22) verwendet.

Literatur für dieses Kapitel

[Hüg+05] Philipp Hügelmeyer u. a. „Integration des Virtuellen Prüfungssystems ViPS in die Lehr-/Lernplattform Stud. IP“. In: *Proceedings of the Workshop on e-Learning*. 2005, S. 187–196.

[Pey+00] Christoph Peylo u. a. „A Web-based intelligent educational system for PROLOG“. In: *Proceedings of the International Workshop on Adaptive and Intelligent Web-Based Education Systems held in conjunction with ITS 2000 Montreal, Canada.* 2000, S. 85–96.

21 Integration automatisierter Programmbewertung in ILIAS

Thomas Richter

Zusammenfassung
Elektronische Programmierübungen werden oft in dezidierten Entwicklungsumgebungen realisiert und sind daher selten in ein eLearning Gesamtkonzept integriert, welches Programmieraufgaben mit traditionellen Elementen von elektronischen Prüfungen wie Lückentexten, Multiple Choice oder Freitextaufgaben kombinieren kann. In diesem Kapitel wird eine ILIAS-Erweiterung diskutiert, welche die Durchführung einfacher Programmieraufgaben in mannigfaltigen Sprachen erlaubt, neben textuellen auch graphische Ausgabemöglichkeiten bietet und ebenso automatische Auswertung von Lösungen erlaubt. Neben der Softwarearchitektur wird ebenso auf die historische Entwicklung des Systems eingegangen und über die bisherigen Erfahrungen und geplante Entwicklungen berichtet.

21.1 Einführung und Hintergrund

Die Vermittlung elementarer Kenntnisse numerischer Mathematik sind schon seit geraumer Zeit Teil der Ingenieurausbildung. Je nach Studienfach erlernen hier die Studierenden numerische Algorithmen mit Hilfe von Computer-Algebra-Systemen (CAS) wie MATLAB oder implementieren die Algorithmen in höheren Programmiersprachen wie C++. Anders als im Studium der Informatik ist dabei die Kenntnis von Softwareentwicklungssystemen oder das Management größerer Softwareprojekte wenig relevant; ein niedrigschwelliger Einstieg in elementare Programmstrukturen und Algorithmen steht hier im Zentrum der Bedürfnisse.

Ein an der Universität Stuttgart nicht unübliches Vorgehen bestand nun darin, die Studierenden dazu aufzufordern, die notwendige Software – also etwa MATLAB oder ein C++ Compiler – herunterzuladen und auf dem eigenen Laptop oder dem heimischen PC zwecks der Durchführung der Übungsaufgaben zu installie-

ren. Obwohl die Universität Stuttgart über Rechnerpools verfügt, sind die Kapazitäten hier zu begrenzt, um allen Studierenden genügend Rechenzeit einräumen zu können; Übungsaufgaben wurden also typischerweise am eigenen Rechner durchgeführt und Lösungen manuell per E-Mail an das Lehrpersonal verschickt und dort korrigiert. Obwohl die Installation von Software auf dem heimischen Rechner eigentlich keine technische Hürde darstellen sollte, entstanden doch immer wieder erhebliche Verzögerungen durch Installationsprobleme, deren Lösung durch das Lehrpersonal viel Zeit in Anspruch nahm und zu Verzögerungen im Lehrplan führten.

Klausuren wurden und werden immer noch in Papierform durchgeführt, d. h. Programmbeispiele müssen mit Bleistift und Papier entwickelt werden, ohne dass die Möglichkeit besteht, diese ausprobieren zu können. Triviale Fehler wie fehlende Semikolons oder falsche Klammerung von Ausdrücken lassen sich so nur schwerlich verhindern, und die Klausurkorrektur ist zeitaufwendig und fehleranfällig.

Nicht zuletzt durch Studierende entstand eine Initiative, die Prozesse der manuellen Installation und manuellen Korrektur durch eine webbasierte Lösung zu ersetzen, die keinen Installationsaufwand erfordert und die eine sofortige Rückmeldung des Compilers oder CAS erlaubt. Dies war der Start des ViPLab-Projektes – kurz für Virtuelles Programmierlabor.

Anders als viele andere in diesem Buch besprochene Projekte sollte ViPLab von Anfang an ein von der Wahl der Programmiersprache unabhängiges System werden, die Anwenderwünsche reichten von MATLAB über C++ und C und Java bis zu universitären Eigenentwicklungen wie dem Numerikwerkzeug DuMux.

Die Entwicklungsziele von ViPLab beinhalteten dabei jedoch nicht, gängige Werkzeuge zur Softwareentwicklung – wie etwa Eclipse oder Visual Studio – zu ersetzen oder Kenntnisse im Management großer Softwareprojekte zu vermitteln. Die von Studierenden der Ingenieurwissenschaften zu erstellenden Programme sind oft kurz und umfassen selten mehr als eine Quelldatei. ViPLab ist darum nicht als Ersatz für ein graphisches Entwicklungssystem (IDE) gedacht.

Stattdessen soll ViPLab vor allen Dingen von Studierenden sofort vollumfänglich benutzbar sein und nur durch die Installation eines Webbrowsers nutzbar sein. Installation von zusätzlicher Software erzeugt eine Barriere, die es zu vermeiden galt. Zu guter Letzt sollte ViPLab sich in das eLearning-System der Universität eingliedern, welches weiterhin die zentrale Anlaufstelle für Übungsaufgaben, Übungen und Tests bleiben sollte.

21.2 Historische Entwicklung

ViPLab entstand als ein durch Studiengebühren finanziertes Projekt mit Zustimmung der Vertretung der Studierenden und wurde getragen vom Rechenzentrum der Universität Stuttgart (damals RUS, heute TIK), dem Institut für Wasser- und Umweltsystemmodellierung, dem Institut für angewandte Analysis und Numerische Simulation und dem Institut für Aero- und Gasdynamik als Projektleiter. Als externer Projektpartner liefert FreeIT eine Middleware zur Kommunikation zwischen den technischen Systemkomponenten, siehe auch Abschnitt 21.5 für weitere Details zum technischen Aufbau des Gesamtsystems.

Die erste Version von ViPLab basierte noch auf einem Java-basierten Plugin, welches als Teil des ViPLab-Benutzerinterfaces vom eLearning-System ausgeliefert wurde. Dieser Ansatz erlaubte zwar, die ViPLab-Benutzerschnittstelle als SCORM-Modul [Lea] zu realisieren und somit eine vom Learning-Management-System (LMS) unabhängige Lösung zu ermöglichen, die Betriebserfahrung zeigte jedoch, dass die Architektur diverse Einschränkungen mit sich brachte. Einerseits musste seitens der Studierenden eine Systemkomponente installiert werden – nämlich das Java-Plugin für den verwendeten Browser. Nicht zuletzt aufgrund mangelhafter Pflege seitens des Herstellers Oracle bei der Behebung auftretender Sicherheitslücken wurden Java-Applets unpopulär, und die Installation eines Plugins provoziert erneut Fehler seitens der Anwender, deren Vermeidung eines der Ziele des Projektes war.

Andererseits sind die technischen Möglichkeiten, über ein SCORM-Modul Einfluss auf ein Learning-Management-System zu nehmen, recht begrenzt. So ist in SCORM beispielsweise nicht vorgesehen, Berechnungsergebnisse zwecks manueller Nachkorrektur im Lernsystem zu hinterlegen; stattdessen beschränkt sich der Datensatz auf die Übermittlung einer bereits vom Plugin zu ermittelnden Punktzahl. Auch die Aufgabenerstellung durch das Lehrpersonal wurde häufig als zu komplex und unhandlich kritisiert, da Aufgaben außerhalb des eigentlichen Lernmanagementsystems erstellt und dann manuell in dieses hochgeladen werden mussten.

Viele dieser Kritikpunkte wurden in der zweiten Version von ViPLab beseitigt. Statt eines generischen SCORM-Plugins basiert die Version 2.0 auf einem proprietären Plugin für das ILIAS-System [Ili], des an der Universität Stuttgart verwendeten Lernmanagementsystems, welches sich jedoch tiefer und damit ohne weitere Barrieren in das System integriert.

Die Benutzerschnittstelle wird vollständig als Javascript an die Studierenden ausgeliefert und erfordert somit kein Browserplugin mehr. Durch die tiefere Integration in das Gesamtsystem werden die studentischen Lösungen im System

hinterlegt und können dort von dem Lehrpersonal zur Nachkorrektur eingesehen werden.

21.3 Die Benutzeroberfläche

Bild 21.1 zeigt eine ViPLab-Aufgabe aus Sicht eines Studierenden. Der zu bearbeitende Quellcode ist dabei zentral in der Mitte angeordnet und kann mit einem Editor bearbeitet werden. Der grau hinterlegte obere Teil ist dabei vom Dozenten als Teil der Aufgabenstellung vorgegeben und kann nicht verändert werden.

Über den „Berechnung starten"-Knopf unten wird der Code compiliert und ausgeführt, wobei dies auf dezidierten Servern des Rechenzentrums und nicht auf dem studentischen PC erfolgt. Eine Ausgabe in Textform oder als Plot erscheint rechts vom Quellcode.

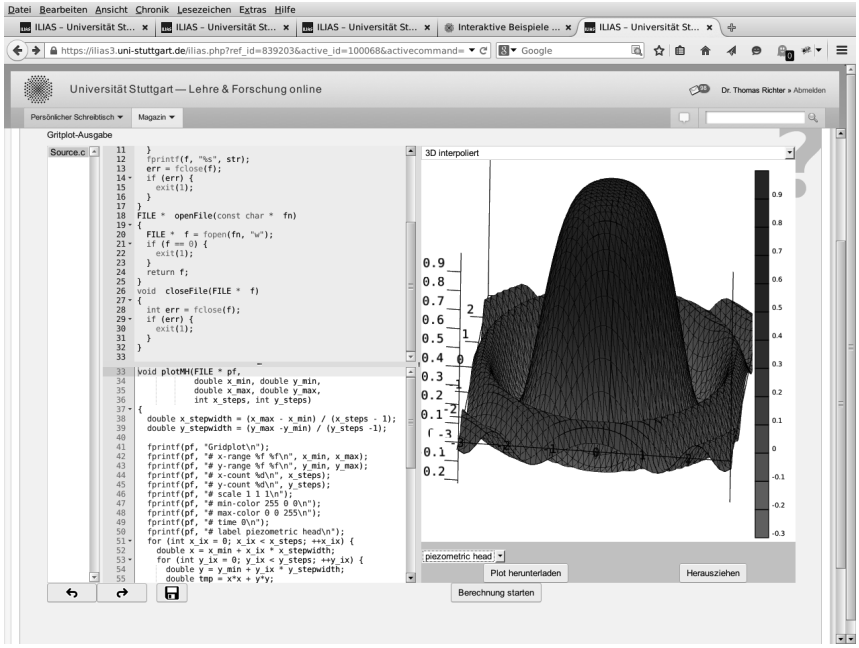


Abbildung 21.1: Eine typische ViPLab-Aufgabe aus Sicht eines Studierenden. Ganz links eine Leiste zur Auswahl des zu bearbeitenden Quellcodes, rechts daneben der Programmcode. Ganz rechts das Resultat der Berechnung, hier als Plot.

Die Visualisierung von Berechnungsergebnissen war dabei ein von den Vertretern der Studierenden schon früh angefragtes Feature von ViPLab; die graphische Ausgabe ist darüber hinaus nicht statisch, sondern erlaubt die Skalierung, Rotation und Ausrichtung des Plots über die Maus, ähnlich den nativen Plotfunktionen von MATLAB. Die graphische Ausgabe steht allen Programmiersprachen und nicht nur MATLAB zur Verfügung.

Das Benutzerfrontend für Lehrende entspricht weitgehend der Schnittstelle für Studierende, siehe Bild 21.2. Eine Aufgabe besteht dabei aus mehreren Quelldateien, die über ein Pop-Up-Menü (nicht im Bild) über der linken Auswahlleiste erzeugt oder gelöscht werden können. Eine Quelldatei wiederum besteht aus einem oder mehreren Abschnitten, hier zentral im Bild, die sichtbar, veränderbar, konstant oder unsichtbar sein können. Mehr zum Aufgabenmodell von ViPLab in Abschnitt 21.6.

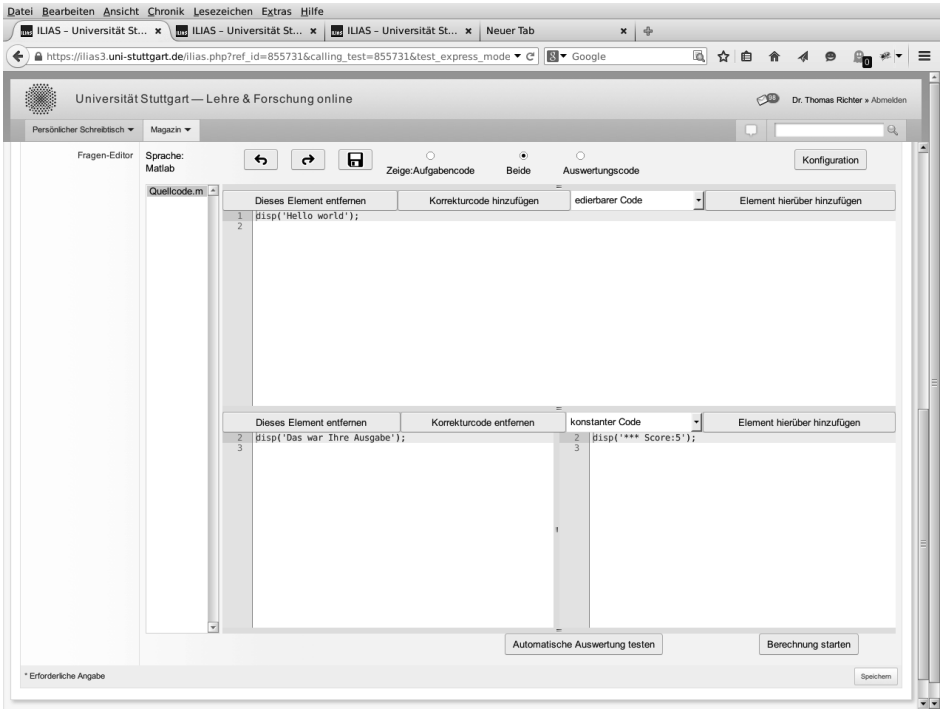


Abbildung 21.2: Die Entwicklung einer ViPLab-Aufgabe aus Dozentensicht. Eine Quellcodedatei ist hierbei in mehrere Abschnitte unterteilt, die veränderbar, konstant oder unsichtbar sein können.

Zur Aufgabenkorrektur können einer oder mehrere der Abschnitte durch einen Korrekturcode ersetzt werden. Details zur automatischen Aufgabenkorrektur sind in Abschnitt 21.7 beschrieben.

21.4 Automatische Korrektur studentischer Lösungen

In der VipLab-Version 1.0 erfolgte die Korrektur von Übungsaufgaben vollständig manuell: Studierende erarbeiteten ihre Lösungen am eigenen Laptop oder dem heimischen PC und verschickten die Lösungen am LMS vorbei an das Lehrpersonal, welches dann die Abgaben manuell bewerteten und die Punktzahl in das LMS eintrugen. Eine derartige Lösung skaliert nicht gut auf große Teilnehmerzahlen, so dass schon von Anfang an eines der Entwicklungsziele von ViPLab die automatische Korrektur von Übungsaufgaben war.

Mit der Version 2.0 entstand zunächst eine halbautomatische Lösung: Das System kann hierbei selbstständig eine Punktzahl ermitteln, jedoch muss diese Ermittlung manuell von einem Dozenten angestoßen werden. Dem Dozenten obliegt es, die Bewertung des Systems zu überprüfen und ggf. eine Nachbewertung durchzuführen. Dieses Vorgehen war einerseits durch die Systemarchitektur bedingt (siehe Abschnitt 21.5), andererseits herrschte seitens der ViPLab-Entwickler auch eine gewisse Skepsis gegenüber vollständig automatischen Bewertern und deren Einschätzungen von menschlichen Fehlern. Ein Syntaxfehler mag aus menschlicher Sicht zwar trivial sein, verhindert aber die maschinelle Ausführung eines Lösungsansatzes und somit die maschinelle Bewertung einer Abgabe. Andererseits sind derartige Fehler auch vom Studierenden durch Testen einer Lösung trivial zu finden, und so mehrten sich Stimmen von Anwendern, eine vollständig automatische Auswertung zuzulassen, die mit ViPLab 2.1 umgesetzt wurde.

Der in ViPLab verwirklichte Ansatz zur automatischen oder halbautomatischen Aufgabenbewertung unterscheidet sich dabei radikal von den typischerweise verfolgten Strategien, die auf sprachspezifischen Mechanismen aufbauen. Ein oft verfolgter Ansatz ist dabei die Anwendung von Unit-Tests, etwa von JUnit für Java. Aufgrund der angestrebten Sprachunabhängigkeit von ViPLab war diese Möglichkeit nicht gegeben. Genaueres über die Technik zur automatischen Auswertung findet sich in Abschnitt 21.7.

Das Fernziel von ViPLab ist auch, elektronische Klausuren durchführen zu können, die neben den üblichen Multiple- oder Single-Choice-Aufgaben ebenso Programmieraufgaben enthalten. Näheres zu den Konzepten für elektronische Klausuren ist in Abschnitt 21.9 beschrieben.

21.5 Softwarearchitektur

Die ViPLab-Softwarearchitektur besteht aus vier Schichten, siehe Bild 21.3: Zuoberst die Benutzerschnittstelle (siehe auch Bild 21.1), welche über Javascript im Browser des Nutzers abläuft. Die Benutzerschnittstelle wird dabei über `http` vom ILIAS System (dem LMS) ausgeliefert. Benutzercode wird auf den Backends (rechts in Bild 21.3) compiliert und zur Ausführung gebracht. Hierbei wird pro Aufgabe eine „Sandbox“ erzeugt, in der der studentische Code vom restlichen Betriebssystem abgeschirmt ablaufen kann.

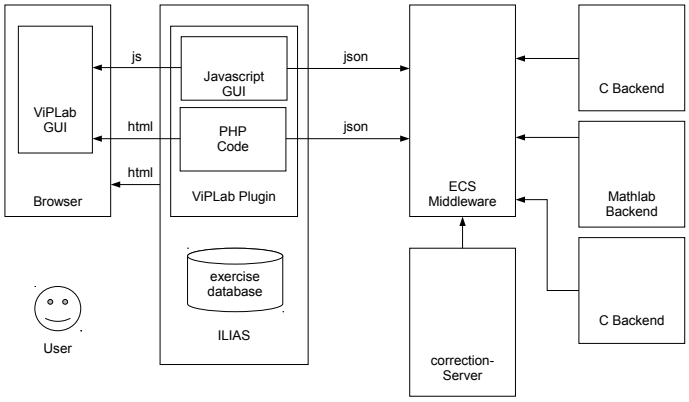


Abbildung 21.3: Gesamtarchitektur von ViPLab. Links die Benutzeroberfläche im Browser des Lernenden, in der Mitte ILIAS und die ECS-Middleware. Rechts die Backend-Server, die die eigentlichen Berechnungen ausführen.

Vermittelnd zwischen den beiden steht die ECS-Middleware, die Warteschlangen für Backends und Frontends zur Verfügung stellt und die Last zwischen den Backends verteilt. Eine Skalierung des Systems ist somit immer leicht durch den Anschluss weiterer Backends möglich.

Ebenfalls an der Middleware dockt der Korrekturserver an, der die Bewertung von studentischen Lösungen koordiniert und hierfür über die Middleware mit dem ECS und dem ILIAS kommuniziert.

Die Kommunikation zwischen den Systemkomponenten erfolgt dabei in JSON [Bra], der nativen Systemsprache von Javascript. Die Wahl fiel hierbei auf JSON, weil es deutlich einfacher zu verarbeiten ist als XML und für die Zwecke von ViPLab komplett ausreicht. In JSON werden sowohl Aufgaben, als auch studentische Lösungen und Berechnungsresultate codiert.

Möchte ein Studierender eine ViPLab Aufgabe bearbeiten, so legt im ersten Schritt das ILIAS-Plugin den Quellcode der Aufgabe auf der ECS-Middleware ab. Daraufhin liefert ILIAS die graphische Oberfläche an den Browser des Studierenden aus und übergibt ihr als Parameter die URL der Aufgabe auf dem ECS. Der im Browser ablaufende Javascript-Code zieht seinerseits wiederum die zu bearbeitende Aufgabe vom ECS; diese Aufgabe enthält den Korrekturcode nicht, es besteht also keine Gefahr, dass der Nutzer durch Beobachtung des Netzwerkverkehrs Hinweise auf eine korrekte Lösung bekommt.

Nach Bearbeitung der Aufgabe codiert das Javascript-Frontend im Browser den veränderten Codeabschnitt zurück an den ECS, der hierdurch wiederum ein Ereignis auf den Backends auflöst. Diese studentische Lösung enthält ihrerseits die URL der Aufgabe auf dem ECS. Ein Studierender kann also nicht durch Manipulation des Netzwerkverkehrs nachträglich den Aufgabentext verändern und so Punkte erschleichen.

Das Backend zieht die Lösung – und über den Verweis auf die Aufgabe auch den Quelltext – des Code-Templates aus dem Aufgabentext, kombiniert beides, und compiliert den hieraus erzeugten Quelltext. Der studentische Code wird dann innerhalb einer Sandbox, d. h. eines vom restlichen System getrennten Bereiches, zur Ausführung gebracht. Die Ausgaben der Lösung werden in JSON codiert und zurück auf den ECS geschickt, der nun seinerseits ein Ereignis auf dem Frontend auslöst. Das Frontend holt sich die Lösung, dekodiert sie und zeigt sie an.

Ähnlich funktioniert die Erstellung von Aufgaben, wobei das ILIAS Rechtemanagement die Aufgabendatenbank vor unberechtigtem Zugriff absichert. Die automatische Korrektur wird im Abschnitt 21.7 separat beschrieben.

21.6 Aufbau von ViPLab-Aufgaben

Eine ViPLab-Aufgabe besteht aus einer oder mehreren Übersetzungseinheiten, von der jede wiederum aus einem oder mehreren Abschnitten besteht, siehe hierzu Bild 21.4.

Eine Übersetzungseinheit kann man grob mit einer Quelldatei einer lokalen Compiler-Installation vergleichen; abhängig von der Wahl der Programmiersprache klassifiziert ViPLab diese Quelldateien auch nach Typ und unterscheidet Quellcode, Header und Datendatei. Letztere enthalten dabei vom studentischen Code zu bearbeitende Rohdaten.

Die Strukturierung einer Übersetzungseinheit in mehrere Abschnitte erlaubt, invariante Codeteile aus didaktischen Gründen vor dem Überschreiben durch Stu-

dierende zu schützen. Dies könnte etwa das Hauptprogramm sein, welches die Ansteuerung eines vom Studierenden zu erzeugenden Codeteils übernimmt.

Des Weiteren können ganze Codeteile verborgen werden; sie werden dann vom Frontend nicht dargestellt. Typischerweise werden hier Bibliotheksfunktionen ausgeblendet, die als solches nur verwirren würden und keinen didaktischen Wert haben. So enthält etwa die Beispielaufgabe aus Bild 21.1 einen solchen Bibliothekscode, der die graphische Ausgabe statt auf den (nicht existenten) Bildschirm des Backends in eine Datei umlenkt und durch das Frontend visualisieren lässt.

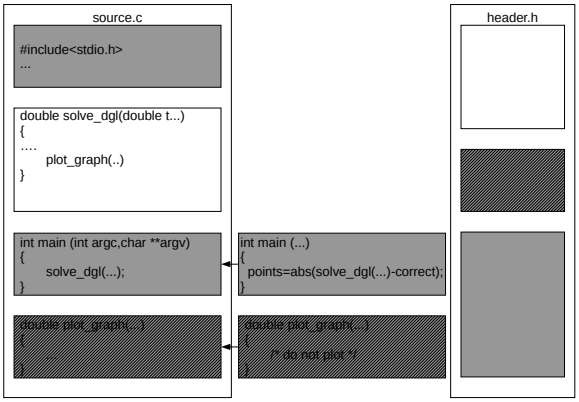


Abbildung 21.4: Der Aufbau einer ViPLab-Aufgabe: Sie besteht aus mehreren Übersetzungseinheiten, die sich wiederum in mehrere Abschnitte gliedern.

21.7 Automatische und halbautomatische Korrektur von Aufgaben

Anders als viele andere Zugänge zur automatischen Korrektur von Aufgaben ist der ViPLab-Ansatz vergleichsweise einfach; das Entwicklungsziel war eine vollständige Sprachunabhängigkeit, wie man sie durch die Nutzung von Frameworks wie JUnit nicht erreichen kann. Die Komplexität dieser Frameworks gefährdete in der Sicht des Entwicklungsteams auch die Akzeptanz des Gesamtsystems seitens der Lehrenden, so dass stattdessen ein möglichst einfacher und verständlicher Ansatz gewählt wurde. Die Korrektur von Aufgaben erfolgt in ViPLab einzig durch *Substitution von Codeabschnitten*.

Hierbei können konstante oder unsichtbare Codeabschnitte (siehe Kapitel 21.6, Bild 21.4) zum Zwecke der Korrektur durch anderen Code ersetzt werden, der über den Aufruf der studentischen Lösung eine Punktzahl ermittelt. Ein solcher Korrekturcode würde etwa ein vom Dozenten erstelltes Hauptprogramm ersetzen und einen studentischen Lösungsansatz mit Testdaten füttern, und die Rückgabewerte der Lösung auf Gleichheit mit einer Musterlösung überprüfen. Eine ausgefeiltere Strategie würde zufällige Eingabeparameter für die studentische Lösung erzeugen und parallel mit der abgegebenen Lösung und einem Musteralgorithmus Berechnungen durchführen. Stimmt die abgegebene Lösung mit der Musterlösung innerhalb einer gewissen Toleranz überein, so ist die Lösung korrekt. Schlägt die Auswertung in bestimmten Ausnahmesituationen fehl, so kann die Bepunktung entsprechend geringer ausfallen.

Im Falle einer halbautomatischen Korrektur erfolgt die Ersetzung der Codeteile durch das Frontend des Dozenten innerhalb von ILIAS, muss dann aber manuell angestoßen werden. Der Korrekturserver (siehe Bild 21.3) wird dann nicht verwendet.

Für den Einsatz in großen Vorlesungen oder elektronischen Klausuren (siehe Abschnitt 21.9) ist die halbautomatische Lösung allerdings immer noch zu aufwendig, da das Lehrpersonal jede Lösung einzeln einsehen muss. Deshalb erreichten uns nach der Veröffentlichung von ViPLab 2.0 sehr rasch Anwenderwünsche für eine vollautomatische Korrektur.

Wird die automatische Korrektur eingeschaltet, so wird bei der Abgabe einer studentischen Lösung diese genauso wie bei einer manuellen Auswertung zunächst auf dem ECS abgelegt; vergleiche hierzu den Vorgang bei der Berechnung einer Aufgabe in Abschnitt 21.5. Gemeinsam mit der Aufgabe und der studentischen Lösung legt das ILIAS-Plugin den Korrekturcode auf den ECS. Hierdurch wird am Korrekturserver (unten Mitte in Bild 21.3) ausgelöst, der nun statt des Backends die Lösung, die Aufgabe und den Korrekturcode vom ECS abholt. Dort erfolgt nun – statt im Dozenten-Frontend – die Ersetzung von Codeteilen der Aufgabe durch den Korrekturcode. Der Korrekturserver spielt über den ECS den Rechenjob an die Backends, die die Ergebnisse der Berechnung an den Korrekturserver zurück liefern und der nun seinerseits die Punktzahl aus dem Ergebnis extrahiert. Die Punktzahl wird über ein Ereignis an den ILIAS zurückgeliefert, der diese in seine Datenbank einträgt.

Eine detailliertere Analyse des studentischen Codes findet momentan nicht statt; so wäre durchaus der Einsatz von Codemetriken sinnvoll, oder von statischen Codeanalysewerkzeugen wie Lint zur Bewertung des Programmierstils. Derartige Funktionalitäten ließen sich als Teil des Korrekturservers implementieren. Ebenso würde die Systemarchitektur prinzipiell ermöglichen, durch den Korrekturcode

Aufrufe auf Unit-Test-Frameworks in die abgegebene Lösung zu injizieren und somit auch eine Prüfung des Codes durch Unit-Tests durchzuführen.

Alle diese Ansätze sind jedoch im Augenblick noch nicht im Einsatz; momentan beschränkt sich die Fähigkeit des Systems auf die oben eingeführte einfache Codesubstitution zur Bewertung einer Lösung.

21.8 Erfahrungen und Evaluation von ViPLab

ViPLab ist seit ca. 2010 an der Universität Stuttgart im Einsatz; neben einigen anfänglichen technischen Schwierigkeiten läuft der Betrieb aus der Sicht des Rechenzentrums reibungslos. Um nun auch Daten über die Nützlichkeit von ViPLab aus Anwendersicht zu sammeln, führte das Rechenzentrum im Jahr 2013 eine Vergleichsstudie [Van+13] durch, bei der Übungen mit ViPLab klassischen computergestützten Übungen gegenüber gestellt wurden; hierbei bearbeiteten Studierende an einem klassischen Editor mit vom Lehrpersonal bereitgestellten Codeabschnitten und einem lokal installierten Compiler oder mit ViPLab ihre Übungsaufgaben. Der Inhalt der Übungsaufgaben bestand aus dem Erstellen eines kurzen C++ Programms aus vorgegebenen Codeabschnitten. Die Übungen fanden bei ansonsten identischer Ausstattung im gleichen Computerpool der Universität statt. Das verwendete Betriebssystem im Pool war Debian-Linux, das Lehrpersonal war in beiden Gruppen ebenfalls identisch. Die Gruppengröße der ViPLab-Nutzer war 22, die der Kontrollgruppe mit einer klassischen Installation bestand aus 35 Studierenden.

Das Ziel dieser Vergleichsstudie war, festzustellen, ob zwischen den beiden Arbeitsumgebungen – lokale Installation, Compiler und Editor oder ViPLab – ein statistisch signifikanter Unterschied in Bezug auf die Zufriedenheit der Nutzer besteht, und inwieweit derartige Unterschiede mit der Computeraffinität der Studierenden korreliert.

Das Ziel der Untersuchung war es herauszufinden, wie sich der Einsatz von ViPLab auf die Zufriedenheit der Studierenden auswirkt. Dabei sollten zum einen das Vorwissen und die Vorkenntnisse der Studierenden erhoben werden um auf diesen Faktor zu kontrollieren. Die eigenen Selbtwirksamkeitsüberzeugungen der Studierenden im Bezug auf die Computerbenutzung sollten ebenfalls ermittelt werden.

Die Vorkenntnisse und die Selbtwirksamkeitsüberzeugungen wurden mit einer angepassten INCOBI-R Skala [RNG01; RNH10] erhoben. Die Zufriedenheit in Bezug auf das verwendete System mit einer Skala vermessen, die auf Gruber [Gru+10] basiert. INCOBI-R untersucht folgende Aspekte mittels eines Fra-

genkatalogs auf drei Skalen: Die praktische Computererfahrung (PRAECOWI), theoretisches Wissen (TECOWI), und das Selbstvertrauen der Nutzer im Umgang mit Rechnern (COMA). Zusätzlich wurden die Studierenden nach ihrem bislang verwendeten Betriebssystem auf ihren eigenen Rechnern befragt.

Das Resultat der Studie zeigte, dass zwischen der Zufriedenheit mit der lokalen Compilerinstallation und der Nutzung von ViPLab kein statistisch signifikanter Unterschied besteht. Ferner sollte die Studie klären, ob zwischen der Technikaffinität der Studierenden und der Zufriedenheit mit den zwei Lösungen ein Unterschied besteht. Hier konnte nur in einem Fall eine statistisch signifikante negative Korrelation nachgewiesen werden, nämlich zwischen Nutzern des Windows-Betriebssystems und der Zufriedenheit mit der lokalen Installation. Es zeigte sich, dass Windows-Nutzer mit der Unix Oberfläche im Computerpool weniger zufrieden waren als mit ViPLab. Bei Nutzern anderer Betriebssysteme (Apple und Linux) waren derartige Korrelationen statistisch nicht signifikant.

Das Resultat zeigt, dass ViPLab genauso gut angenommen wird wie eine klassische gut vorbereitete Installation auf lokalen Rechnern, wobei die Vorbereitungszeit für das Erstellen der Arbeitsumgebung durch die Dozenten und das Lehrpersonal für eine lokale Installation erheblich höher ist. Zusammenfassend kann man sagen, dass ViPLab von den Studierenden ebenso gut wie eine lokale Installation angenommen wird, aber die Arbeitsleistung beim Lehrpersonal erheblich reduziert.

21.9 Ausblick: Elektronische Klausuren

Ein möglicher zukünftiger Einsatz von ViPLab liegt in der Durchführung elektronischer Klausuren. ViPLab würde dabei ein Hauptproblem der bisherigen Papierklausuren lösen, nämlich dass Studierenden die Möglichkeit gegeben wird, Programmcode und Zwischenergebnisse direkt am Rechner zu validieren. ViPLab-Aufgaben würden hierbei mit klassischen Elementen elektronischer Prüfungen verknüpft werden, nämlich Multiple- und Single-Choice-Aufgaben, Freitext- und Formelfragen.

Leider verfügt die Universität Stuttgart über keinen geeignet großen Computerpool, um typische Bachelorstudiengänge aufnehmen zu können; hierbei sind Teilnehmerzahlen von bis zu 2000 Studierenden keine Seltenheit. Der größte verfügbare Pool verfügt hingegen nur über knapp 70 Plätze.

Zur Durchführung elektronischer Klausuren müssen stattdessen gewöhnliche Hörsäle mit mobilen Rechnern ausgestattet werden. Es wurden hierbei zwei mögliche Ansätze durchgespielt: Einerseits könnte man Studierende verpflichten, ei-

gene Geräte zur Klausur mitzubringen und die Klausur in einer geeignet abgesicherten Umgebung auf diesen Geräten durchzuführen. Andererseits könnte eine elektronische Klausur auch auf Rechnern der Universität durchgeführt werden, wobei hier auch je nach Wahl der Lösung ebenso eine abgesicherte Arbeitsumgebung erstellt werden müsste und die Geräte von Personal der Universität gewartet werden muss.

Um die Praktikabilität der ersten Lösung abzuwägen, führte das Rechenzentrum gemeinsam mit dem Institut für Wasser- und Umweltsystemmodellierung eine Probeklausur durch, wobei als gesicherte Umgebung ein selbstbootendes Linux auf Knoppix-Basis [Kno] eingesetzt wurde. Dieses wurde nur insofern verändert, als nach dem Systemstart automatisch ein Browser gestartet wurde, der sich selbsttätig mit dem LMS (ILIAS) der Universität verband. Alle weiteren Programme wurden aus der Installation entfernt.

Die Erfahrungen zeigten, dass dieser Ansatz wenig praktikabel ist: Neben typischen Hardwareproblemen – Knoppix unterstützt die WLAN-Chipsätze der neuesten Notebookgeneration nur unzureichend – gab es auch eine Reihe unerwarteter Probleme mit dem Nutzerinterface von Knoppix. Beispielsweise bootet das System grundsätzlich mit einer deutschen Tastaturbelegung, aber aufgrund einer Vielzahl ausländischer Kursteilnehmer entspricht dies nicht immer der am Gerät vorhandenen Tastatur. Da die problemlose Funktion von Knoppix nicht für jedes Laptopmodell garantiert werden konnte und somit die vom Gesetzgeber geforderte Chancengleichheit aller Prüfungsteilnehmer so nicht sichergestellt werden kann, wurde dieses Modell verworfen.

Die Alternative, Geräte für die Prüfung zu stellen, erfordert jedoch deren regelmäßige Wartung, wie etwa das Einspielen von Updates, durch Personal der Universität und ist deshalb mit wiederkehrenden Folgekosten verbunden. Ein in diese Richtung interessanten Ansatz verfolgt die Universität Utrecht: Statt gewöhnlicher Notebooks werden hier Chromebooks verwendet. Diese Geräte sind nicht nur relativ preisgünstig in der Anschaffung, sondern verfügen von Haus aus über eine abgesicherte Ausführungsumgebung – d. h. der Anwender kann keinerlei Software installieren – und sie können zentral administriert werden. Anders als bei typischen Laptops oder Netbooks läuft hier nur ein vorinstallierter Browser, der aber für die Zwecke elektronischer Prüfungen vollkommen ausreicht. Die Einstellungen aller Geräte, auch der Netzzugang und die für Nutzer erreichbaren Webseiten können zentral, also einmal für alle Geräte administriert werden. Bei der Verwendung des Kiosk-Modus verlassen Nutzerdaten auch nicht das universitäre Netz und die recht strengen Datenschutzrichtlinien in Deutschland werden eingehalten.

Im Zuge dieser Überlegungen beschaffte das Rechenzentrum der Universität im Sommer 2016 einen ersten Satz von 40 Chromebooks; die bisherigen Tests mit Studierenden im Rahmen einer Probeklausur verliefen positiv. Als eine der wenigen gewünschten Verbesserungen wird das Rechenzentrum noch Sichtschutzfilter auf den Chromebooks montieren, um die Gefahr des Abschreibens zu minimieren, und Mäuse als zusätzliche Eingabegeräte anschaffen.

Literatur für dieses Kapitel

[Bra] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. https://tools.ietf.org/html/rfc7159. (besucht am 23.05.2015).

[Gru+10] Thorsten Gruber u. a. „Examining student satisfaction with higher education services: Using a new measurement tool“. In: *International Journal of Public Sector Management* 23.2 (2010), S. 105–123. DOI: 10.1108/09513551011022474.

[Ili] *ILIAS Open Source e-Learning*. http://www.ilias.de. (besucht am 23.05.2016).

[Kno] Klaus Knopper. *KNOPPIX Website*. http://www.knoppix.org/. (besucht am: 23.05.2016).

[Lea] Advanced Distributed Learning. *SCORM Technical Specification*. https://adlnet.gov/wp-content/uploads/2011/07/SCORM_2004_4ED_v1_1_Doc_Suite.zip. (besucht am 23.05.2016).

[RNG01] Tobias Richter, Johannes Naumann und Norbert Groeben. „Das Inventar zur Computerbildung (INCOB): Ein Instrument zur Erfassung von Computer Literacy und computerbezogenen Einstellungen bei Studierenden der Geistes- und Sozialwissenschaften.“ In: *Psychologie in Erziehung und Unterricht* 48.1 (2001), S. 1–13.

[RNH10] Tobias Richter, Johannes Naumann und Holger Horz. „Eine revidierte Fassung des Inventars zur Computerbildung (INCOBI-R).“ In: *Zeitschrift für pädagogische Psychologie* 24.1 (2010), S. 23–37.

[Van+13] J. Vanvinkenroye u. a. „A Quantitative Analysis of a Virtual Programming Lab“. In: *Proc. of Multimedia (ISM), 2013 Intl. Symposium on* (2013), S. 457–461.

22 Integration mithilfe der Middleware ProFormA-Server

Oliver Rod

Zusammenfassung
An den Hochschulen gibt es häufig verschiedene Lernmanagementsysteme und spezialisierte Bewertungssysteme. Die ProFormA-Middleware verbindet diese beiden Typen von Systemen miteinander. Im folgenden Kapitel wird die Middleware vorgestellt. Der Fokus liegt auf den Schnittstellen und den Abläufen in und mit der Middleware. Dabei wird auf die Praktiken bei der Implementierung an der Ostfalia Hochschule verwiesen.

22.1 Einleitung

In diesem Buch werden in den Kapiteln 19, 20 und 21 verschiedene Lernmanagementsysteme mit ihren jeweiligen Besonderheiten und Schwerpunkten vorgestellt. Die Systeme haben unter anderem folgende Funktionen:

- Kursverwaltung (Inhaltsverwaltung, Kurse),
- verschiedene Kommunikationsmöglichkeiten zwischen Lehrenden und Lernenden (Forum, Chat, Nachrichten, Kalender),
- Unterstützung gängiger Aufgabentypen (Multiple Choice, Lückentext, Drag and Drop) und
- Verwaltung von Lernobjekten, Aufgaben, Tests.

Teile dieses Kapitels entstanden im Rahmen des Projekts eCULT, Teilvorhaben eAssessment, gefördert durch das Bundesministerium für Bildung und Forschung unter dem Förderkennzeichen 01PL16066H. Die Verantwortung für den Inhalt dieses Kapitels liegt bei dem Autor.

Die genannten Aufgabentypen der Lernmanagementsysteme können die Anforderungen der technischen Studiengänge möglicherweise nicht abdecken. Die Herausforderung besteht darin, hochgeladene Programmcodes oder Modellierungssprachen zu analysieren und zu testen. In den vorangegangenen Kapiteln dieses Buchs (siehe Abschnitt II) wurden bereits einige Systeme zur automatischen Programmbewertung vorgestellt. Diese Bewertungssysteme legen ihren Schwerpunkt hingegen auf:

- automatisierte Bewertung von Einreichungen,
- Sicherheit in der Ausführung der Einreichung für das ausführende System,
- angepasste Tests an die jeweiligen Aufgaben und Sprachen,
- Verwendung von Werkzeugen in der Softwareentwicklung wie Kompiler, Unit-Tests, Versionierung und Style-Tester,
- zusätzliches Feedback, welches über die Kompilerausgabe hinaus geht.

Trotz der Menge an Bewertungssystemen ist es schwierig ein Werkzeug zu finden, dass allen Anforderungen eines Lehrenden gerecht wird. Diese Anforderungen bestehen in den verwendeten Programmiersprachen, den jeweiligen Testarten und den verschiedenen Lehrstilen.

Die ProFormA-Middleware ermöglicht eine einfache Koppelung der Lernmanagementsysteme und der Bewertungssysteme miteinander und schafft somit die Voraussetzungen, um die genannten Vorteile der beiden Systeme zu verbinden. Die vorgeschlagene Middleware soll mehrere Bewertungssysteme mit jeweils einem Lernmanagementsystem koppeln, um den Lehrenden mehr Freiheiten in ihren jeweiligen Aufgabenstellungen zu gewährleisten.

Ein Fokus bei der Interoperabilität der Bewertungssysteme liegt im Austausch der Aufgaben. Diese sind in ihrer Erstellung viel komplexer und zeitintensiver im Vergleich zu gängigen Aufgabentypen (siehe [PJR12]). Als universelles Aufgabenformat für die Programmieraufgaben nutzt die Middleware das ProFormA-Aufgabenformat (vgl. Kapitel 24). Ein weiterer Vorteil bei der Verbindung von Lernmanagementsystem und Bewertungssystem für Studierende und Lehrende, basiert auf der Verwendung einer Benutzeroberfläche.

22.2 System und Ablauf

Die ProFormA-Middleware wurde an der Ostfalia Hochschule entwickelt und verbindet ein LMS, LON-CAPA (siehe Kapitel 19), mit einem Bewertungssystem, Praktomat (vgl. Kapitel 10). Die Middleware ist hierbei nach dem KISS-Prinzip[1] aufgebaut. Es wurde versucht die Komplexität und den Aufwand bei der Implementierung der Schnittstellen für das LMS und die jeweiligen Bewertungssysteme so gering wie möglich zu halten. Nur wenn viele Systeme die Schnittstellen unterstützen, kann die Middleware ihre Vorteile nutzbar machen. Denn auch das LMS und die Bewertungssysteme müssen eine Schnittstelle bereitstellen, welche die Middleware unterstützt.

Die Abbildung 22.1 zeigt exemplarisch die verbundenen Systeme und die jeweiligen Schnittstellen. Der Aufgabenpool (auch Repository) ist in der Abbildung als eigenständiges System dargestellt, wird aber an der Ostfalia derzeit mittels dem Aufgabenpool von LON-CAPA realisiert. Die zentralen Aufgaben der Middleware sind die Verwaltung der angeschlossenen Systeme und, bei Bedarf, eine Übersetzung der jeweiligen Antwortformate ineinander. Studierende und Lehrende verwenden ihren Browser für die Nutzung oder die Konfiguration des Lernmanagementsystems. Zusätzliche Zugänge zu den verwendeten Bewertungssystemen werden nicht benötigt. Die Middleware verwaltet die angeschlossenen Bewertungssysteme und bietet für das LMS eine definierte Schnittstelle (siehe Kapitel 22.3.1). Die Konfiguration der angeschlossenen Bewertungssysteme wird in einer Konfigurationsdatei auf der Middleware verwaltet. Die Bewertungssysteme

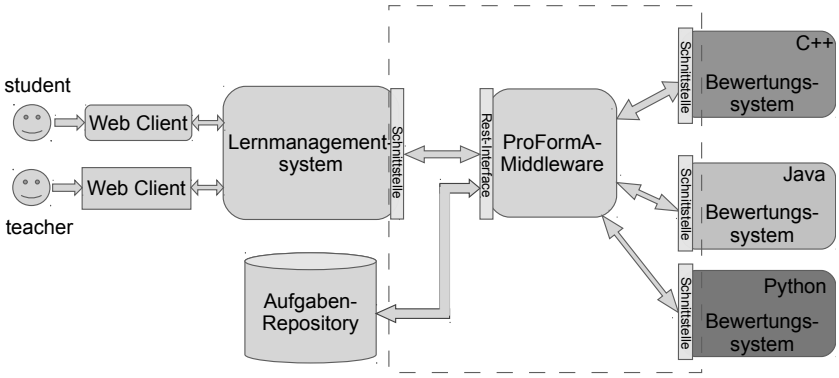


Abbildung 22.1: Architektur der Middleware

1 Keep it short and simple

können unterschiedliche Programmiersprachen, wie zum Beispiel C++, Python oder Java, aber auch unterschiedliche Testverfahren, wie Unit-Tests oder Blackboxtests unterstützen. Für die Bewertung einer Aufgabe sind nur die studentische Einreichung und die Programmieraufgabe im XML-basierten Austauschformat (ProFormA-XML) notwendig. Die Kommunikation über die Middleware verläuft derzeit immer synchron. Das Ergebnis einer studentischen Einreichung kann somit nicht zu einem späteren Zeitpunkt abgerufen werden. In Kapitel 22.3 folgt die Beschreibung der Schnittstellen.

22.3 Schnittstellen

Es gibt für die vorgeschlagene Architektur drei wichtige Schnittstellen, welche definiert werden müssen.

LMS verwendet ein einfaches Web-Interface, welches Daten versendet und über ein definiertes Antwortformat Ergebnisse auswerten kann.

Middleware REST-Schnittstelle für den definierten Informationsaustausch zwischen Bewertungssystemen und Lernmanagementsystemen.

Bewertungssystem benötigt ebenso eine REST-Schnittstelle für die Erstellung und Bewertung von Programmieraufgaben.

22.3.1 REST-API der Middleware

Die URI für die Einreichung ist detailliert in Abbildung 22.2 veranschaulicht. Es wird über die URI-Parameter *fw* und *fw-version* das benötigte Framework und die Version übermittelt. Hiermit ist die benötigte Programmiersprache gemeint. Die Middleware weist der Einreichung somit das benötigte Bewertungssystem zu. Grundsätzlich benötigt jede Einreichung drei allgemeine Informationen:

- die eingereichte Aufgabe im ProFormA-Aufgabenformat,
- das erwartete Antwortformat,
- die studentische Lösung.

Um die zu übermittelnden Daten so gering wie möglich zu halten, wird nur der Ort der Aufgabenressource mittels *task-repo* und *task-path* versendet. Eine direkte Übermittlung der Aufgaben erfolgt im XML-Format. Das LMS hat bestimm-

te Formvorgaben, in der es die Antwort erwartet. Das verwendete LMS, LON-CAPA, hat ein definiertes XML-Antwortformat[2]. Sollten die Lernmanagementsysteme keine externe Schnittstelle anbieten, könnte es sinnvoll sein, ein standardisiertes Antwortformat[3] für das LMS vorzugeben. Die studentische Einreichung kann entweder direkt im HTTP-POST-Parameter *submission* übergeben werden oder sie wird mittels *submission-uri* zusammen mit dem Ort der Einreichung übermittelt. Die Interaktionen zwischen LMS und Middleware und zwischen Middle-

POST api/v1/grading/prog-languages/**:fw/:fw-version**/submissions

- Einreichung der Problemlösung

URI-Parameter:

Feld	Typ	Beschreibung
fw	String	Programmiersprache bzw. Framework welches unterstützt werden soll
fw-version	String	Version des Frameworks

HTTP-Parameter:

Feld	Typ	Beschreibung
task-repo	String	IP bzw Domain des Repositories
task-path	String	Name der Aufgabe mit Pfad
answer-format	String	Name und Version des Antwortformates
submission	String	Inhalt der Einreichung
submission-uri	String	URI wo die Einreichung lieg

Abbildung 22.2: URI und Parameter für die Einreichung

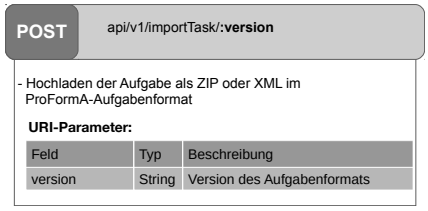
ware und Bewertungssystemen sollten immer über HTTPS verlaufen, um eine zusätzliche Sicherheit zu bieten. Des Weiteren müssen die verbundenen Systeme über ihre IP-Adresse beziehungsweise IP-Bereiche freigegeben werden. Es wurde von einer Schnittstelle zur Administration der Middleware abgesehen, zumal diese ein unnötiges Sicherheitsrisiko bei zu geringem Nutzen darstellen würde. Eine Entkopplung von IP-Freigabe und Freigabe neuer Bewertungssysteme sollte direkt auf dem Server der Middleware vorgenommen werden. Eine zusätzliche

2 https://vita.ostfalia.de/adm/help/Authoring_ExternalResponse.hlp
3 https://github.com/ProFormA/responsexml

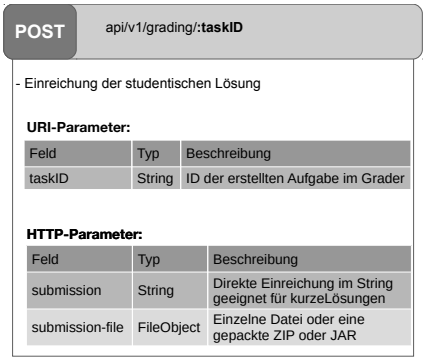
Authentifizierung über Token oder HTTP Authentication nach RFC 2617[4] ist geplant.

22.3.2 Schnittstelle von Bewertungssystemen

Bewertungssysteme laufen grundsätzlich autark und bieten meist keine externe Schnittstelle an. Der Vorschlag ist hier, eine externe Webschnittstelle zu implementieren. Über diese Schnittstelle kann die Middleware die Aufgabe mittels ProFormA-Aufgabenformates erstellen und anschließend die studentische Einreichung bewerten lassen. An der Ostfalia wurde das Bewertungssystem Praktomat[5] um diese vorgeschlagene REST-Schnittstelle mit zwei URIs erweitert. Einen Vorschlag für diese beiden URIs veranschaulicht Abbildung 22.3a.



(a) REST-URI vom Import im Bewertungssystem



(b) REST-URI vom Grading im Bewertungssystem

Abbildung 22.3: Übersicht der REST-URIs des ProFormA-Servers

Für den Import muss die Version der verwendeten ProFormA-XML angegeben werden. Bei erfolgreicher Erstellung wird die Identifikation des Tasks zurückgegeben (TaskID), welche für die eigentliche studentische Einreichung benötigt wird. Für die Antwort wird ein einfaches JSON-Schema mit *TaskID* und *message* empfohlen.

4 https://www.ietf.org/rfc/rfc2617.txt
5 https://github.com/KITPraktomatTeam/Praktomat

Die erhaltene *TaskID* wird für die *Grading-URI* benötigt und die eigentliche Einreichung wird entweder direkt als String im Feld *submission* übergeben oder als Datei gesendet. Nach dem erfolgten Testen der Einreichung wird auch hier die Antwort in einem JSON-Format erwartet. Die Antwort sollte sich hier an die Vorgaben des Antwortformates halten. Im Optimalfall erspart dies der Middleware eine Umwandlung verschiedener Antwortformate.

22.4 Aufbau der Middleware

Die Middleware wurde mit dem Webframework django[6] entwickelt und folgt dem Model-View-Controller-Schema. Der einfache Aufbau der Middleware wird in Abbildung 22.4 veranschaulicht. Alle Aufrufe über das Rest-Interface laufen über den Controller, welcher bei Bedarf mit den einzelnen Modulen interagiert. Zum derzeitigen Stand (Juli 2016) wurde zunächst ein Bewertungssystem (grader_praktomat) mit der Middleware verbunden. Dieses Modul behandelt die Interaktion mit dem Praktomaten sowie die Erstellung (*create_task*) und das Einreichen der studentischen Lösung (*grade_Task*). LON-CAPA dient übergangsweise auch als Repository und wird im *repo_handler* bearbeitet. Die Einreichungen können auch einen Verweis auf den Ort (submissions-uri) enthalten. Dieser externe Verweis wird innerhalb des Moduls *extern_submission* abgehandelt. Derzeit wird nur LON-CAPA für externe Verweise unterstützt.

Das Modul *answerformat_converter* soll sich um die Umwandlung der Antwortformate kümmern. Oftmals ist dies nicht notwendig, aber sobald mehrere Lernmanagementsysteme unterstützt werden, wird dies unausweichlich sein.

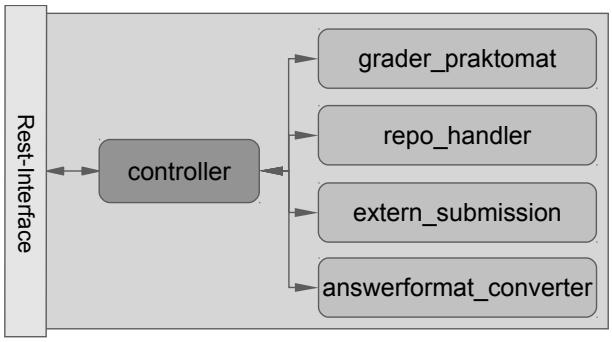


Abbildung 22.4: Systemübersicht der Middleware

6 https://www.djangoproject.com

22.5 Ablauf

In diesem Abschnitt soll auf einzelne Abläufe in der Verwendung der Middleware eingegangen werden. Die folgenden Phasen beschreiben den Prozess den ein Lehrender bei der Erstellung einer Programmieraufgabe durchläuft. Der Zyklus einer Programmieraufgabe wird in folgende Phasen unterteilt:

1. allgemeine Aufgabenstellung und Tests im eigenen System,
2. Formulierung der Aufgabe im ProFormA-XML-Format unter Zuhilfenahme des Editors[7],
3. Upload der XML- oder der ZIP-Datei in das Repository
4. Einrichtung der Aufgabe im LMS
5. Testen der Aufgabe im LMS

Aus der Praxis ist bekannt, dass diese Sequenz von Lehrenden mehrfach durchlaufen werden muss, um die Tests und die Aufgabenstellung so präzise wie möglich zu gestalten. Häufig werden Probleme erst nach der ersten Nutzung im Kurs sichtbar. Eine Optimierung der Aufgabenstellung ist zu jedem Zeitpunkt möglich, es müssen dann einzelne Schritte der Sequenz wiederholt werden. Für die Erstellung einer Programmieraufgabe wird die Middleware *nicht* benötigt. Es wird zudem zu keinem Zeitpunkt der direkte Zugriff auf das Bewertungssystem von Lehrenden oder Studierenden benötigt.

Ein direkter Zugriff auf die Middleware ist nur für die IP-Freischaltungen der verbundenen Lernmanagementsysteme, Bewertungssysteme und Aufgabenpools notwendig. Dieser Vorgang wird nicht von Lehrenden, sondern einem Administrator vorgenommen. Die Anzahl der angeschlossenen Systeme an die ProFormA-Middleware ist nicht begrenzt. Einzig die Anzahl der gleichzeitig zu bearbeitenden Anfragen ist limitiert. In der Konfigurationsdatei der Middleware erfolgt die Zuordnung der Bewertungssysteme zu einer Grading-Engine.

Die Abbildung 22.5 verdeutlicht den Ablauf einer studentischen Einreichung. In diesem Szenario gibt die Studierende ihren Programmcode in ein Textfeld ein und sendet ihre Einreichung ab. Das Lernmanagementsystem sendet die Einreichung mit dem Verweis auf die ProFormA-XML und dem zu verwendenden Bewertungssystem an die Middleware.

7 https://github.com/ProFormA/formatEditor

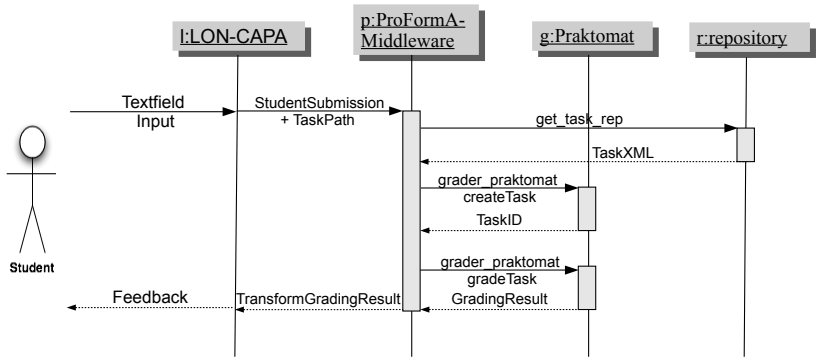


Abbildung 22.5: Sequenzdiagramm einer studentischen Einreichung

Die Middleware lädt die Aufgabe aus dem Aufgabenpool herunter und leitet diese an das gewählte Bewertungssystem, den Praktomaten, weiter. Anschließend erstellt das Bewertungssystem die Aufgabe und gibt die *TaskID* zurück. Jetzt kann die studentische Einreichung mit Hilfe der *TaskID* an die neu erstellte Aufgabe gesendet werden. Nach der automatisierten Bewertung der Einreichung gibt das Bewertungssystem das Ergebnis an die Middleware zurück. Diese übermittelt die Lösung im geforderten Antwortformat an das LMS.
Abschließend bekommt der Studierende das Ergebnis seiner Lösung in LON-CAPA dargestellt. Abbildung 22.6 zeigt beispielhaft das Ergebnis einer fehlerhaften Einreichung in LON-CAPA. Die Anzeige ist in drei Bereiche unterteilt:

- die Aufgabenstellung und die bereits getätigten Einreichungen,
- das Eingabefeld für die Einreichungen mit integriertem Syntax-Highlighting und
- das Ergebnis der Einreichung mit den einzelnen Testergebnissen.

Alle studentischen Daten, wie die Anzahl der Versuche, die getätigten Einreichungen und die Ergebnisse liegen im LMS vor. Die Kursleitung hat hierdurch alle notwendigen Daten innerhalb des Lernmanagementsystem vorliegen und muss keine weiteren Informationen aus den Bewertungssystemen abfragen. Die Kursleitung hat somit die Möglichkeit, die aktuellen Probleme und den aktuellen Kenntnisstand der Studierenden zu erkennen und in den Veranstaltungen gezielt darauf einzugehen.

Abbildung 22.6: Bildschirmfoto einer fehlerhaften Einreichung

22.6 Eigene Erfahrungen und Ausblick

Die vorgestellte Architektur ist seit dem Wintersemester 2014 an der Ostfalia Hochschule im Einsatz. Sie wurde anfangs nur in der Informatik und der Elektrotechnik für die Einführungsveranstaltungen der Programmierung mit Java genutzt. Mittlerweile werden auch Aufgaben im mathematischen Kontext mit SetlX (vgl. Kapitel 8) verwendet. Die Verwendung von LON-CAPA ermöglicht es auch

den Dozenten an anderen Standorten mit LON-CAPA Zugriff auf bereits erstellte Aufgaben der Ostfalia zuzugreifen. Der vorgeschlagene Austausch von Programmieraufgaben wurde somit in einigen Fällen bereits genutzt. Viele der an der Ostfalia betreuten Dozenten haben sich einverstanden erklärt ihre Aufgaben mit anderen Dozenten zu teilen. Gleichzeitig gab es die Bereitschaft Programmieraufgaben von anderen Lehrenden zu nutzen. Der Austausch von Aufgaben ist einer der zentralen Punkte von LON-CAPA (siehe [KC09]). Gerade in der akademischen Kultur ist das Teilen von Wissen essenziell um Erkenntnisse zu teilen und auf diesen aufzubauen.

Die exemplarische Implementierung der ProFormA-Middleware mit dem Lernmanagementsystem LON-CAPA und dem Bewertungssystem Praktomat ist erfolgreich verlaufen. Für die Zukunft ist die Integration weiterer Bewertungssysteme wie auch weiterer Lernmanagementsysteme angestrebt um so die Nutzerschaft und den Nutzen der Middleware weiter zu erhöhen. Die Themen Caching und Lastverteilung auf mehrere Bewertungssysteme werden somit in Zukunft an Bedeutung gewinnen. Hier bietet es sich an, diese in die weiterführenden Entwicklungen der Middleware zu implementieren.

Weiterhin ist geplant, die Middleware ProFormA-Server (Kapitel 22) und die Middleware Grappa (Kapitel 23) mit einer wechselseitigen Schnittstelle auf der Basis des Aufgaben-Austauschformats (Kapitel 24) auszustatten, so dass am Ende alle von einer Middleware unterstützten LMSe mit allen von der jeweils anderen Middleware unterstützten Gradern kombiniert werden können.

Literatur für dieses Kapitel

[KC09] Gert Kortemeyer und E. Cruz. „LON-CAPA – An Open-Source Learning Content Management and Assessment System“. In: *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications*. 2009, S. 1515–1520.

[PJR12] Uta Priss, Nils Jensen und Oliver Rod. „Software for Formative Assessment of Programming Exercises“. In: *elearning Baltics* (2012), S. 63–72.

23 Integration mithilfe der Middleware Grappa

Peter Fricke

Zusammenfassung
Die Vielzahl von Lernmanagementsystemen (LMS) hat in der Vergangenheit stark zugenommen. Ebenso werden immer mehr Programmiersprachen populärer, so dass der Bedarf an Programmen, die eine automatische Bewertung von Programmcode vornehmen („Grader"), zunimmt. Grappa[a] übernimmt dabei die Aufgabe einer Middleware – die Anbindung von Gradern an Lernmanagementsysteme und andersherum. Grappa spezifiziert die Schnittstelle zum LMS und die zum Grader und stellt dabei Tools zur effizienteren Anbindung zur Verfügung. Im Folgenden wird zunächst die Idee und danach die genaue Funktionsweise vorgestellt.

a Der Begriff entstand aus dem Kofferwort Grapper (Grader Wrapper), welches anschließend mit der Intention, Genuss zu assoziieren, abgewandelt wurde.

23.1 Motivation und Anforderungen

An der Hochschule Hannover (HsH) kommen seit 2010 für SQL und die Programmiersprache Java verschiedene Grader zum Einsatz. aSQLg („automated SQL grader" – Kapitel 12) untersucht und bewertet studentische Abgaben zu SQL-spezifischen Aufgaben. Graja („Grader for java programs" – Kapitel 11) prüft studentische Java-Programme auf deren Funktion und deren Ressourcenverbrauch. Beide Grader wurden über unterschiedliche Frontends angesprochen und den Lehrenden als auch Studenten zur Verfügung gestellt.

Dieses Vorhaben wird aus Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen 01PL11066D gefördert. Die Verantwortung für den Inhalt dieses Beitrags liegt bei dem Autor.

Bei mehreren Evaluationen der Hochschule Hannover (siehe auch [Stö+13] und [Stö+14]) wurde von Studierenden angemahnt, dass sie mehrere Systeme zum Hochladen ihrer Lösungen brauchen. Ebenso wurde die Darstellung und der Detailgrad des Feedbacks als „zu technisch“ benannt. Es entstand ein Bedarf die Benutzerschnittstelle zu vereinfachen und zu vereinheitlichen.

Ohne Middleware müsste für jeden vorhandenen oder gewünschten Grader eine Anbindung an das LMS konzeptioniert und programmiert werden. Das Kernkonzept von Grappa ist die Standardisierung der verschiedenen Grader-Schnittstellen auf eine einfache, für das Internet, und somit für LMS nutzbare Schnittstelle. Dabei sind über diese Schnittstelle gleich mehrere Grader ansprechbar. Der Programmieraufwand am LMS reduziert sich auf nur eine Erweiterung für Grappa. Über eine REST-Schnittstelle (siehe Kapitel 23.4) werden XML-Dateien zwischen Grappa und dem LMS ausgetauscht. Die Grader selbst werden von Grappa über eine Java-Schnittstelle angesprochen (näher beschrieben in Kapitel 23.5), welche für jeden Grader integriert werden muss.

Grappa soll unabhängig von **einem** konkretem LMS oder Grader arbeiten können. Um der zentralen Funktion gerecht zu werden, eine studentische Lösung zu einer Aufgabe vom LMS entgegennehmen und durch einen Grader bewerten zu lassen und das Ergebnis dem LMS zurückzuliefern, müssen verschiedene Anforderungen seitens der Grader als auch der LMS berücksichtigt werden.

Die Middleware soll die Beschaffenheit einer Aufgabe (Programmieraufgabe, Datenbankaufgabe, Modellierungsaufgabe, etc.) nicht explizit kennen. Grappa muss mögliche einfache und komplexe Eigenschaften von LMSen und Gradern handhaben können. Dabei müssen sowohl didaktische als auch technische Ausprägungen in einer Aufgabe und deren Bewertung dargestellt werden. Ein Grader sollte weitere über die Aufgabenstruktur hinausgehende Aspekte (bspw. syntaktische oder semantische Korrektheit), hinzufügen können. Grappa soll diese Aufgaben permanent kennen und speichern können. Die Bewertungsskala zwischen Grader und LMS kann sich unterscheiden. Grappa soll vorzugsweise mit einer offenen Wertung umgehen und ggf. mit Unterstützung des LMS eine entsprechende Umrechnung vornehmen.

Ein Grader benötigt oft über die Aufgabenbeschreibung hinausgehende Konfigurationsdateien. Grappa muss diese Dateien ebenfalls in Bezug auf die hinterlegten Aufgaben vorhalten und zur Verfügung stellen können.

Die Ergebnisse der Bewertung eines Graders können sehr einfach oder komplex gestaltet sein. Etwaige Bewertungskommentare können allgemein oder spezifisch sein. Dabei sollte zwischen Feedback für Lehrende und Studierende und dem Typ des Kommentars unterschieden werden. Ebenso muss Grappa zwischen LMS und

Grader die Darstellung der Bewertungskommentare in den Formaten PDF, XML, HTML und Text vermitteln, oder im besten Falle konvertieren.

Der Prozess der Bewertung von Programmieraufgaben kann unterschiedliche Zeitspannen in Anspruch nehmen. Eine Dauer von wenigen Sekunden (Prüfung einer simplen Textausgabe bspw. in Java), bis hin zu mehreren Minuten (komplexere JOIN-Abfragen mit SQL) ist dabei denkbar. Um eine zügige Verarbeitung der Bewertungsvorgänge zu gewährleisten, sollte Grappa mehrere Einreichungen parallel vom LMS annehmen und diese, wenn unterstützt, ebenso parallel an den Grader schicken. Sollte der Grader keine parallele Bewertung unterstützen, oder zu viele Bewertungsvorgänge auf einmal bearbeitet werden müssen, soll Grappa diese Vorgänge in einer Warteschlange dem Grader vorhalten und für das LMS asynchron aufrufbar machen.

Weitere Anforderungen um den Umgang mit der Parametrierung von Aufgaben zu spezifizieren wurden in [GHW15] vorgestellt.

23.2 Fachdatenmodell

In diesem Kapitel wird das Fachdatenmodell von Grappa ausführlich beschrieben. Um allen Anforderungen (siehe Kapitel 23.1) gerecht zu werden, unterscheidet Grappa in vier Hauptentitäten, welche in Tabelle 23.1 gezeigt und in den folgenden Unterkapiteln näher erläutert werden.

Entität	**Mögliche Interpretation**	**Querbezüge zu**
Problem	Ein Aufgabenblatt oder eine Aufgabe	*GrdCfg* (optional)
GrdCfg	Dateien für das GraderBackend zur Bewertung oder Konfiguration eines, mehrerer oder aller Aspekte des Problems	Graderabhängig: *Problem*
Submission	Studentische Lösung	*Problem* (nur temporär an Schnittstellen)
GradingResult	Ergebnis der Bewertung	*Problem*

Tabelle 23.1: Die vier Hauptentitäten von Grappa

23.2.1 Konfigurationsdateien (GrdCfg)

Um einen Grader zu benutzen bedarf es häufig verschiedener Konfigurationsdaten, in der Grappa Begriffswelt GraderConfigurations – *GrdCfgs*. Grappa persistiert diese *GrdCfgs* um eine Wiederverwendung zu ermöglichen. Abbildung 23.1 zeigt die Struktur einer *GrdCfg*. *GrdCfgs* werden von Grappa nicht interpretiert. Sie werden gespeichert und bei Bedarf an das *BackendPlugin* (siehe Kapitel 23.5) weitergeleitet.

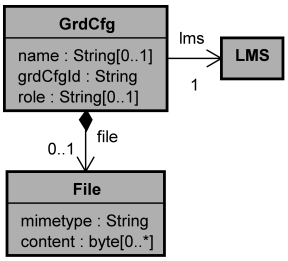


Abbildung 23.1: Fachdatenmodell der *GrdCfg*

name kann genutzt werden um eine Identifizierung der *GrdCfg* für den Benutzer zu ermöglichen.

grdCfgId und die LMS-Id beschreiben eine Konfigurationsdatei eindeutig.

role dient zur Identifikation der Funktion der *GrdCfg* für das BackendPlugin.

file In diesem Objekt befindet sich die eigentliche Konfigurationsdatei, diese kann, wenn es der Grader verlangt, mit einem MimeType versehen werden. *GrdCfg* können beliebige Dateien beinhalten, einfache Properties-Files, Java-Programme und benötigte Bibliotheken sind vorstellbar. Die Dateigröße variiert dabei zwischen wenigen Bytes und mehreren Megabytes.

23.2.2 Aufgabenstruktur (Problem)

Eine Aufgabe kann aus Sicht des LMS in mehrere Unteraufgaben oder Teilaspekte aufgeteilt sein. Diese Struktur spiegelt sich in der Bewertung der Aufgabe wider (siehe Kapitel 23.2.3). Der Grader kann dieser Struktur noch weitere Ebenen hinzufügen, indem zu einer Teilaufgabe bestimmte Aspekte, wie z. B. syntaktische oder semantische Korrektheit oder stilistische Fragen, hinzufügt werden. Grappa

soll diese Aufgabenstruktur kennen und persistent speichern. Ein *Problem* (Abbildung 23.2) definiert so eine Aufgabe mit verschiedenen Unteraufgaben oder Teilaspekten in *ProblemNodes*. Diese ProblemNodes können hierarchisch aufgebaut und konfiguriert werden. Ähnlich der *GrdCfg* ist eine Aufgabe über die *problemId* und die *lmsId* eindeutig bei Grappa identifiziert. Jede *ProblemNode* muss vom LMS einen *nodeKey* zugewiesen werden. Dieser ist für alle Kinder eines Knoten nur auf einer Ebene eindeutig. Der *nodeKey* ist eine Referenz auf eine Aufgabe, die das Backend und der Grader nutzen können, um Verknüpfungen zwischen bspw. einer Musterlösung oder einem speziellen Test und dieser (Teil-)Aufgabe (-Aspekt) herzustellen. Weiterhin wird für jede *ProblemNode* die maximale erreichbare Punktzahl, *scoreMax*, hinterlegt. Für unterschiedliche *ProblemNodes* bedarf es verschiedener *GrdCfg* für den Grader. Des Weiteren kann über *acceptedRDKinds* ein Wunsch an den Grader oder das *BackendPlugin* gestellt werden, wie das Ergebnis dargestellt werden soll. Schließlich können über die *computingResources* Einschränkungen für die maximale Laufzeit, Arbeitspeicher- und Festplattenbelegung der Teilaufgabe oder -aspekts an das GraderBackend gegeben werden.

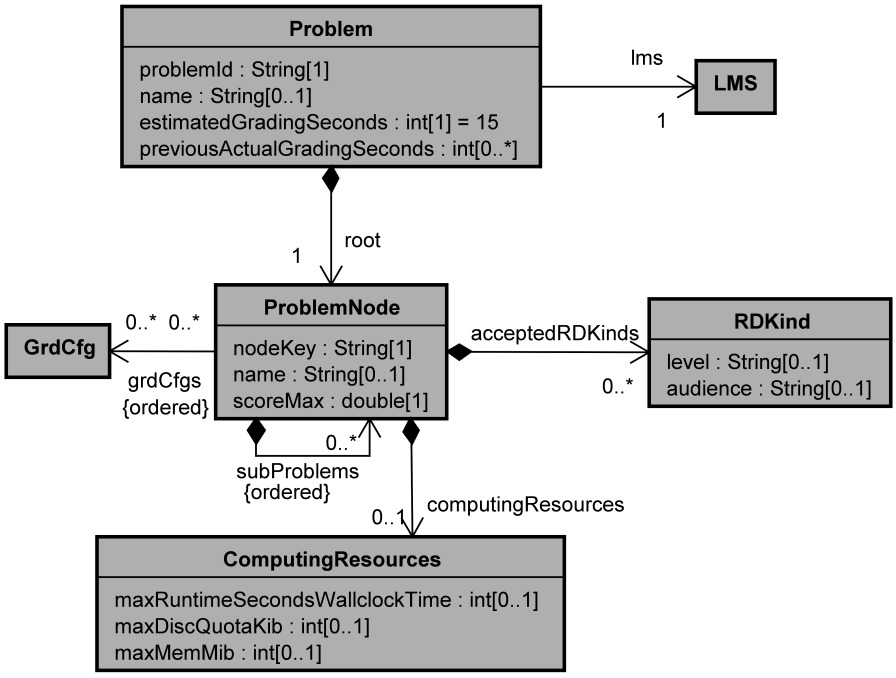


Abbildung 23.2: Fachdatenmodell eines Problem

Grappa persistiert diese *Problems* im derzeitigen System. Somit ist Grappa in der Lage, die Konsistenz zwischen *ProblemNodes* und *GrdCfgs* (23.2.1) zu erhalten. Es ist nicht möglich *GrdCfg*s zu löschen, die von einem oder mehreren *Problems* referenziert werden. Ein weiterer Vorteil ist die schnellere Wiederverwendbarkeit bei der asynchronen Bewertung (siehe Kapitel 23.6) wenn mehrere studentische Lösungen zu einer Aufgabe in kurzer Zeit abgegeben werden.

23.2.3 Studentische Lösung und Ergebnis der Bewertung (Submission & GradingResult)

Eine studentische Lösung kann aus einer Datei oder mehreren Dateien bestehen. Die Zuordnung zur richtigen Aufgabe, dem *Problem*, geschieht über den Aufruf der eindeutigen *problemId* innerhalb der *RunGrader*-Methode (siehe Kapitel 23.4).
Grappa stellt das Ergebnis des Gradens eines *Problems* als *GradingResult* dar. Abbildung 23.3 zeigt, dass ein *GradingResult* genauso hierarchisch aufgebaut ist wie ein *Problem*. Es gibt ein *root-Result*-Objekt, welches wiederum *Result*-Objekte in beliebiger Baumtiefe haben kann. Auf der oberste Ebene verweist jedes *Result*-Objekt auf die zugehörige *ProblemNode*. Darunterliegende *Results* dienen zur Darstellung von Teilaspekten, die gegebenenfalls vom Grader zusätzlich generiert wurden.

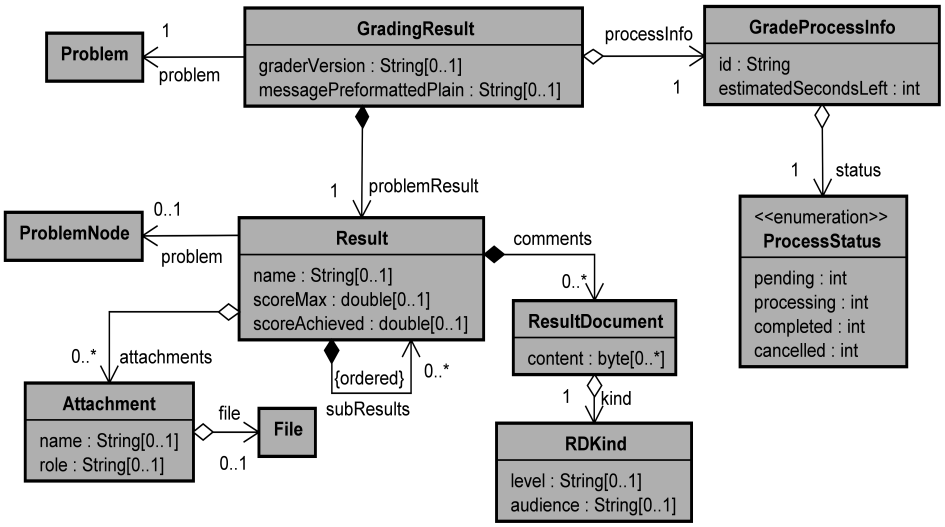


Abbildung 23.3: Darstellung eines GradingResults

name beschreibt die Art des Teilaspekts, oder den Aufgabennamen.

scoreMax zeigt die maximal erreichbare Punktzahl für diesen Teilbaum.

scoreAchieved zeigt die erreichte Punktzahl nach dem grade-Aufruf.

comments bestehen aus ein oder mehreren *ResultDocuments*. Diese dienen zur Darstellung des Grader-Feedbacks und können verschiedene Ausprägungen und Detailgrade haben.

attachments können optional bei Bedarf vom Grader an ein Result angehängt werden. Sie sind unspezifiziert und werden von Grappa nicht interpretiert und lediglich an das Frontend weitergegeben. Beispiele für *attachments* wären statistische Informationen über den *grade*-Aufruf oder ähnliches.

Dem *GradingResult* wird zusätzlich eine *graderVersion* zugewiesen. Sollte es bereits vor dem grade-Aufruf zu Fehlern innerhalb von Grappa oder des Backend-Plugins kommen, werden diese im *messagePreformattedPlain* Feld abgelegt. Somit kann dem LMS ein expliziter Fehler im BackendPlugin oder Grader selbst mitgeteilt werden.

Die *processInfo* enthält Informationen über den aktuellen Stand der Bearbeitung. Gültige Zustände sind:

Pending Die *Submission* befindet sich in der Warteliste und die Bearbeitung hat noch nicht begonnen.

Processing Die *Submission* wurde an das Backend gegeben und wird gerade bearbeitet.

Completed Der grade-Aufruf wurde beendet. Ob ein vollständiges Ergebnis vorliegt, ist den Result-Objekten und ggf. dem *messagePreformattedPlain* zu entnehmen.

Cancelled Die Bewertung wurde durch das Frontend abgebrochen (siehe *cancel*-Methode in Kapitel 23.4).

Der Status ist aus Sicht von LMS zu Grappa zu sehen und beschreibt lediglich den Verarbeitungszustand des *grade*-Befehls. Etwaige Fehler des Graders, oder BackendPlugins werden damit nicht beschrieben. GradingResults werden temporär von Grappa persistiert. Holt ein LMS das Ergebnis nicht innerhalb einer vorkonfigurierten Zeitspanne ab, wird es gelöscht und der grade-Aufruf muss erneut durchgeführt werden. Nähere Informationen darüber sind dem Kapitel 23.6 zu entnehmen.

RDKind

Das *RDKind* erfüllt zwei Aufgaben. Zum einen kann beim Anlegen eines Problems ein Wunsch gegenüber dem Grader ausgesprochen werden, in welchem Format und in welchen Detailgrad das Ergebnis zurückgegeben werden soll. Zum anderen wird ein Ergebnisdokument (*ResultDocument* Kapitel 23.3) mit einem *RDKind* versehen, um die Darstellung und Ausprägung zu spezifizieren.

level stellt dabei den Detailgrad des *ResultDocuments* dar. Mögliche Ausprägungen sind *fatal*, *error*, *warning*, *info* und *debug*.

audience beschreibt die Zielgruppe des *ResultDocuments*. Mögliche Ausprägungen sind *teacher*, *student* und *both*. Dadurch ist das LMS in der Lage das Feedback der richtigen Zielgruppe anzuzeigen.

representation dient zur Identifikation des Formates des *ResultDocuments*, derzeit sind drei Darstellungen möglich: *Html*, *Pdf* und *PlainText*.

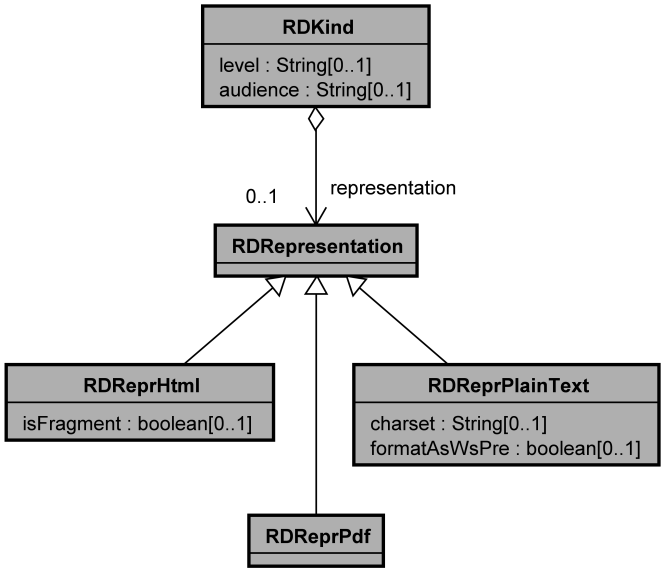


Abbildung 23.4: Arten von ResultDocuments

23.3 Komponenten

Die Middleware Grappa bietet dem Frontend insgesamt zwei und dem Backend eine Schnittstelle an. Mit diesen Schnittstellen müssen die jeweiligen Komponenten in Form von Erweiterungen (Plugins) kommunizieren. Abbildung 23.5 zeigt das Komponentendiagramm einer typischen Grappa-Installation.

LMS Das LMS dient dem Nutzenden als Benutzeroberfläche und sollte erweiterbar sein durch bspw. eine Plugin-Architektur.

LMSToGrappaPlugin Dieses Plugin nutzt die öffentlichen Schnittstellen von Grappa und übersetzt diese Informationen in die Domäne des LMS.

Setup Über die Setup-Schnittstelle erfolgt das Anlegen und Bearbeiten von Konfigurationsdateien und Programmieraufgaben in Grappa.

RunGrader Öffentliche Schnittstelle zum Einreichen und Bewerten studentischer Lösungen.

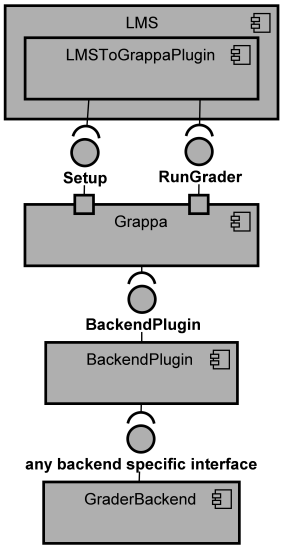


Abbildung 23.5: Fachdatenmodell eines Grappa-nutzenden Systems inklusive aller Komponenten

BackendPlugin Dieses Plugin übersetzt die Begriffswelt von Grappa in möglicherweise mehrere GraderBackend Aufrufe und übersetzt dessen Antwort zurück.

GraderBackend Denkbar sind hier mehrere GradingTools, oder nur ein Grader, die wiederum vom *BackendPlugin* aufgerufen werden.

Grappa wurde als Java-Servlet entwickelt und kann in einen typischen Application-Server, wie z. B. Tomcat, JBoss oder Glassfish, eingesetzt werden. Eine Anbindung an eine Datenbank ist ebenfalls notwendig, da Grappa *GrdCfgs* und *Problems* speichert. *GrdCfgs* können so mehrfach genutzt werden und das Bewerten mehrerer Lösungen zur gleichen Zeit kann schneller erfolgen.

23.4 Öffentliche Schnittstelle

Das Frontend von Grappa bietet zwei Schnittstellen für das LMS. Beide Schnittstellen werden über REST-Aufrufe angesprochen.

Die Setup-Schnittstelle dient zum Anlegen, Ändern und Löschen von *GrdCfg*s und *Problem*s. Das getrennte Anlegen von *GrdCfg*s und *Problem* wurde gewählt, um eine Wiederverwendbarkeit von *GrdCfg*s zu gewährleisten.

Einem LMS ist es möglich, eine *GrdCfg* anzulegen, die von mehreren Problems benutzt werden kann. Die interne Struktur von Grappa erlaubt kein Löschen einer *GrdCfg* die von einem *Problem* referenziert wird.

Für alle drei Operationen bekommt das LMS eine XML-Datei und einen HTTP-Status zurückgeliefert. Diese beinhaltet entweder eine Erfolgsmeldung, oder im Fehlerfall eine detaillierte Fehlerbeschreibung.

Die RunGrader-Schnittstelle kann erst genutzt werden, wenn ein *Problem* vollständig angelegt wurde. Sie dient dem LMS, um eine studentische Abgabe über Grappa dem Grader zur Verfügung zu stellen und zu bewerten. Der Schnittstellenparameter subm stellt den Bezug zum Problem her. Grappa kann intern mit einem asynchronen Bewertungsmechanismus laufen. Das LMS muss eine in der Warteschlange befindliche *Submission* per Polling abfragen. Grappa soll keine Kenntnisse über die Funktionsweise der verschiedenen LMS haben. Durch bspw. einen Callback-Mechanismus, gäbe es eine zu enge Kopplung zwischen LMS und Grappa.

Beim Grading, oder Polling (siehe 23.6) liefert Grappa dem LMS ein *GradingResult* in Form einer XML-Datei zurück. Diese beinhaltet ein vollstän-

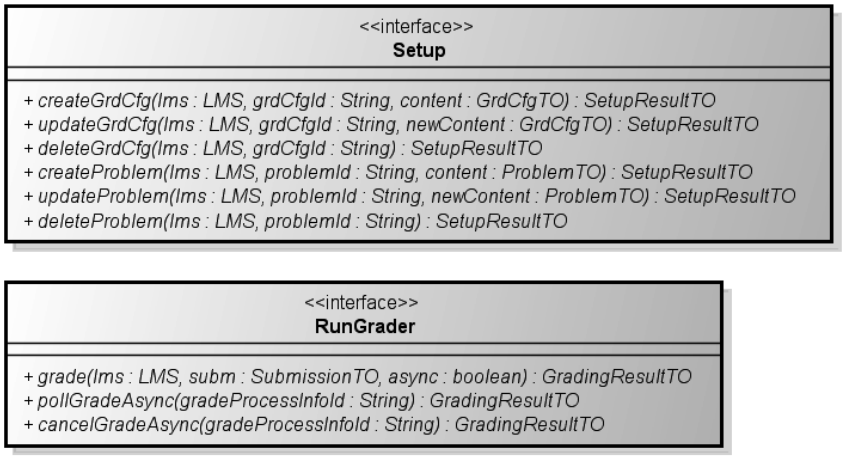


Abbildung 23.6: Methodenaufrufe der öffentlichen Schnittstellen;

diges Ergebnis, sofern der Vorgang abgeschlossen ist. Befindet sich die *Submission* weiterhin in der Warteschleife, so wird nur eine Prozessinformation (*Pending*, *Processing*, *Completed*, *Cancelled*) inklusive einer Id und einer voraussichtlichen Wartezeit übergeben. Das LMS kann auf diese Informationen dementsprechend reagieren und dem Studierenden oder Lehrenden eine Fehlermeldung anzeigen.

23.4.1 REST-Aufrufe

Die Kommunikation zwischen LMS und Grappa geschieht über HTTP-REST und XML-Dateien. Diese XML-Dateien unterliegen mehreren grappaspezifischen Schemadateien. Nehmen wir an, unter einer beliebigen URL läuft ein Grappa-Server (URL abgekürzt durch *Grappa*) und ein LMS mit der pseudo LMS-Id `hsh` wollte die in Abb. 23.6 gezeigten Methoden aufrufen. Dabei stellen Werte mit `_id_` den, vom LMS erzeugten, Identifikator für die jeweilige Ressource dar. Die *grdcfgsIds*, *problemIds* und *gradeProcessInfoIds* werden von Grappa erzeugt. Folgende Aufrufe sind für die Setup-Schnittstelle gültig[1]:

GrdCfg anlegen POST Grappa/hsh/grdcfgs/ new-grdcfg.xml

GrdCfg updaten PUT Grappa/hsh/grdcfgs/**_id_grdcfg** updated-grdcfg.xml

GrdCfg löschen DELETE Grappa/hsh/grdcfgs/**_id_grdcfg**

1 Derzeit nur vorgesehen, jedoch nicht implementiert sind GET-Aufrufe für eine komplette Liste der *GrdCfg*s und *Problem*s.

Problem anlegen POST Grappa/hsh/problems/ new-problem.xml

Problem updaten PUT Grappa/hsh/problems/**_id_problem** updated-problem.xml

Problem löschen DELETE Grappa/hsh/problems/**_id_problem**

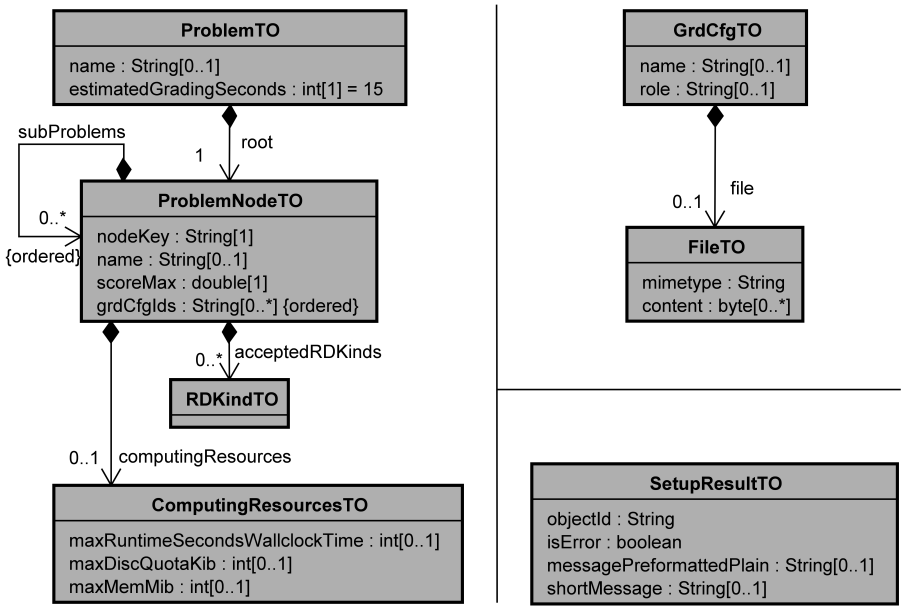


Abbildung 23.7: Darstellung aller XML-Dokumente der Setup-Schnittstelle

Dabei antwortet Grappa immer mit einem Ergebnisdokument in Form eines *SetupResults* (siehe Abbildung 23.7). Beim Anlegen von *GrdCfg*s oder Problems beinhaltet diese XML-Datei ebenfalls die erzeugte Objekt-ID, damit eine spätere Referenzierung möglich ist. Etwaiges Fehlverhalten oder Erfolgsmeldungen werden als unformatierter String in *messagePreformattedPlain* und ggf. *shortMessage* dem LMS zur Verfügung gestellt.
Die Grader-Schnittstelle:

Submission graden POST Grappa/hsh/gradeprocesses?async=*true* submission.xml

Grading pollen GET Grappa/hsh/gradeprocesses/**gradeProcessInfoId**

Grading löschen/abbrechen DELETE Grappa/hsh/gradeprocesses/**gradeProcessInfoId**

Grappa antwortet auch im Fehlerfall immer mit einem *GradingResult* in Form einer XML-Datei. Bei beiden Schnittstellen überträgt Grappa übliche HTTP-Statuscodes.

23.4.2 Tools

Im Rahmen der Entwicklung eines Moodle-Plugins für Grappa (Kapitel 18.3.1) wurde eine PHP-Clientbibliothek entworfen um den Programmieraufwand für LMS weiter zu vereinfachen. Der „Grappa-PHP-Client“ übernimmt die Übersetzung von PHP-Objekten in das XML-Format und zurück. Der Client stellt ebenfalls Methoden zur Kommunikation mit Grappa zur Verfügung. Dabei ist der komplette Polling-Mechanismus für den Apache-Webserver unter Linux und Windows integriert. Der Grappa-PHP-Client dient als weitere Schnittstelle zwischen *LMSToGrappaPlugin* und der *Setup* und *RunGrader*-Schnittstelle in der gezeigten Architektur (Abbildung 23.6).

23.5 Grader-Schnittstelle – BackendPlugin

Die Kommunikation zwischen einem Grader und Grappa wird über ein *Backend-Plugin* ermöglicht. Für jede Art von Grader muss ein Java-Plugin entwickelt werden, welches das in Abbildung 23.8 dargestellte Interface implementiert. Dieses Plugin übernimmt Übersetzungsaufgaben und die Kommunikation zwischen Grappa und Grader in beide Richtungen.

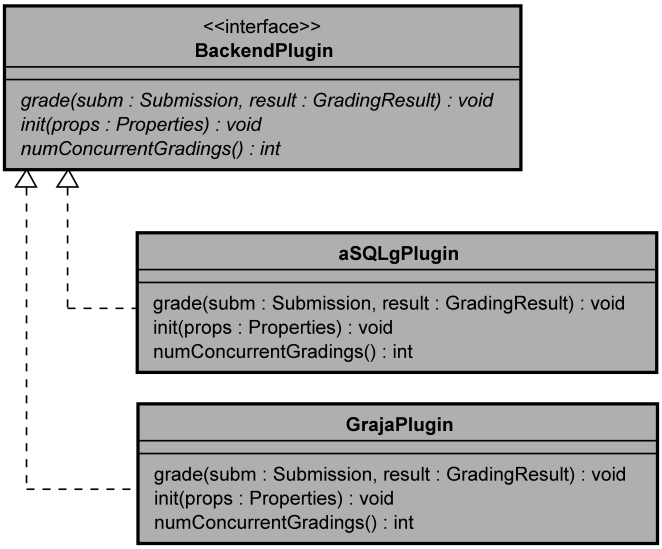


Abbildung 23.8: Fachdatenmodell der Plugin-Schnittstelle

Grappa liefert dem *BackendPlugin* die Rohdaten des Problems in Form eines *Result*-Objektes (erläutert in Kapitel 23.2.3). Das Plugin kann nun über die verschiedenen Teilaufgaben iterieren und je nach Einsatzmöglichkeit die komplette oder jede einzelne Teilaufgabe vom Grader bewerten lassen. Die Resultate des Graders werden anhand der verfügbaren *acceptedRDKinds* entweder direkt vom Grader oder vom *BackendPlugin* transformiert und an das entsprechende *Result*-Objekt angehängt. Es bleibt dem Grader und *BackendPlugin* überlassen, ob etwaige Teilaspekte des Resultates ebenfalls angehängt werden.

23.6 Ablauf

Bevor vom LMS eine studentische Lösung an Grappa übergeben werden kann, muss ein Lehrender alle Konfigurationsdateien (*GrdCfgs*) anlegen und eine Aufgabe (*Problem*) erstellen. Kapitel 18.3.1 zeigt diese Erstellung beispielhaft mit Moodle. Da diese Daten lediglich in der Grappa-Datenbank abgespeichert werden, bedarf es hier keiner detaillierten Beschreibung.

Sobald ein *Problem* erstellt wurde, kann ein Studierender über das LMS eine Lösung für diese Aufgabe abgeben. Abbildung **??** zeigt einen verkürzten jedoch genauen (asynchronen) Ablauf innerhalb des Grappasystems.

Sobald die studentische Lösung erfolgreich vom LMS entgegen genommen wurde, kann das LMS anhand der gespeicherten *problemId* und der expliziten Grappa-Grader-Instanz einen *grade*-Befehl an die *RunGrader*-Schnittstelle absenden. Grappa prüft nun das *Problem* inklusive aller Konfigurationsdateien am Datenbankserver und validiert somit Zuordnung der *Submission* an das *Problem*. Grappa stellt anhand der Größe der Warteschlange (*Queue*) die ungefähre Dauer des Gradingvorgangs fest und liefert dieses Ergebnis mit dem *ProcessStatus* „waiting“ und einer *GradeProcessInfoId* dem LMS zurück. Das LMS ist somit angewiesen, nach dieser Zeit ein Polling zum gegebenen Identifikator bei Grappa vorzunehmen.

Bei Grappa wird zeitgleich die Warteschlange von oben herab abgearbeitet. Dabei wird der nächste *GradingTask* bearbeitet und zunächst von der Datenbank abgerufen und per *grade*-Methode an das *BackendPlugin* weitergegeben. Das *BackendPlugin* kann nun das *Problem* untersuchen und die einzelnen *ProblemNodes* mit zugehörigen *GrdCfgs* für das Graderbackend unter Umständen umwandeln und bereitstellen. Mit *any Backend* können ebenfalls mehrere externe Tools aufgerufen werden, dies obliegt der Implementierung des BackendPlugins. Sobald das *BackendPlugin* die Ergebnisse der/des Grader/s verarbeitet und in die

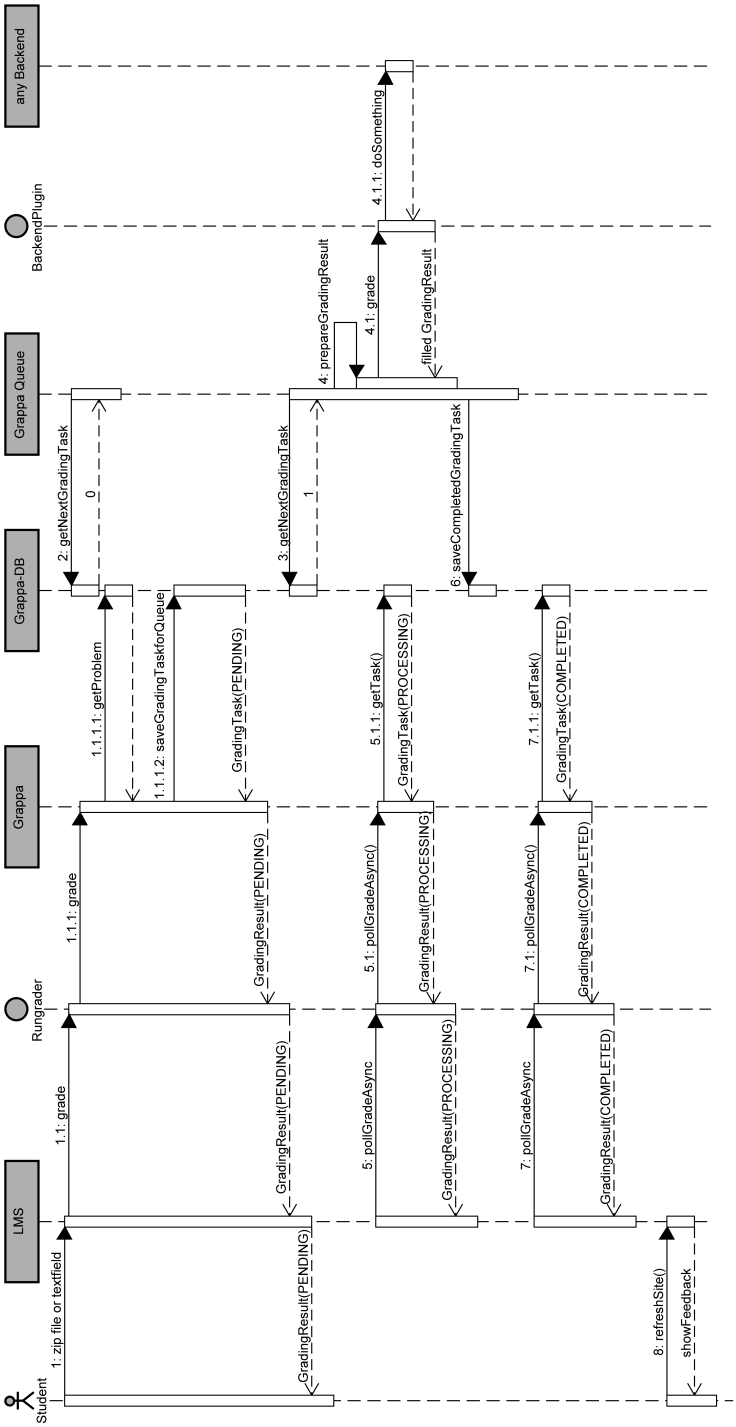


Abbildung 23.9: Verkürzte Darstellung des asynchronen Bewertungsablaufs

expliziten *Result*-Objekte ein- oder angefügt hat, speichert Grappa diese *GradingResult* in der Datenbank ab. Der Prozessstatus ist nun *completed*.

Das LMS könnte während dieser Phase durchgängig mit Grappa kommunizieren. Sämtliche Interaktionen mit Grappa wurden asynchron implementiert. Die dementsprechende Logik, wenn *GrdCfgs* oder *Problems* während eines *grade*-Vorganges geändert, oder gelöscht werden, wurde ebenfalls bedacht. Nachdem das LMS (womöglich nach mehreren Polling-Versuchen) den Status *completed* erhält, kann mit der Aufbereitung der Ergebnisdokumente begonnen werden (Abschnitt 23.4) und dem Studierenden zur Verfügung gestellt werden.

23.7 Zusammenfassung und Ausblick

Grappa kann durch die gezeigte Architektur alle Anforderungen von Gradern abbilden. Spezielle Anforderungen können durch Konfigurationsdateien und durch eine dementsprechende Funktion im BackendPlugin realisiert werden. Es ist gelungen eine stabile Middleware zu entwerfen und bisher an zwei Backends und ein LMS anzubinden. Die Anbindung weiterer LMS (derzeit LON-CAPA und ggf. Stud.IP) ist in Planung. Ebenso sollen weitere BackendPlugins für andere Grader erstellt werden, so dass die Unterstützung von weiteren Programmiersprachen erfüllt wird.

Derzeit laufen intensive Arbeiten um das in Kapitel 24 vorgestellte Austauschformat zu unterstützen. Durch die unterschiedliche Darstellung der Aufgaben in Grappa und im Austauschformat bedarf es entweder einer Anpassung im Austauschformat (vgl. [GFB16]) oder eines eigenen Namespaces in den dafür vorgesehenen XML-Räumen. Ein eigener Namespace hätte den Nachteil, dass Grappa-Aufgaben nur in andere Grappa-Instanzen mit den gleichen Gradern übertragen werden können, jedoch nur teilweise graderübergreifend funktionieren würden.

In eCULT+ soll ein Repository für Programmieraufgaben entwickelt werden. Ziel ist es, dieses Repository für Grappa ansprechbar zu machen, so dass der Lehrende im besten Fall nur eine Aufgabe auswählen muss, statt diese manuell anzulegen.

Weiterhin ist geplant, die Middleware ProFormA-Server (Kapitel 22) und die Middleware Grappa (Kapitel 23) mit einer wechselseitigen Schnittstelle auf der Basis des Aufgaben-Austauschformats (Kapitel 24) auszustatten, so dass am Ende alle von einer Middleware unterstützten LMSe mit allen von der jeweils anderen Middleware unterstützten Gradern kombiniert werden können.

Ebenso ist eine gemeinsame Definition der LMS und Grader-Schnittstellen für eCULT+ geplant. Dabei soll vorzugsweise eine Anbindungen weiterer bestehender Grader an Grappa stattfinden.

Grappa wurde zur Unterstützung von parametrierbaren Aufgaben bereits konzeptionell vorbereitet ([GHW15]) und in einer Beta-Version getestet. Es bedarf noch mehrerer Tests und Evaluationen, um diesen Ansatz fest zu integrieren.

Literatur für dieses Kapitel

[GFB16] Robert Garmann, Peter Fricke und Oliver Bott. „Bewertungsaspekte und Tests in Java-Programmieraufgaben für Graja im ProFormA-Aufgabenformat“. In: *DeLFI 2016 – Die 14. E-Learning Fachtagung Informatik*. Bd. 262. LNI. GI, 2016, S. 215–220.

[GHW15] Robert Garmann, Felix Heine und Peter Werner. „Grappa – die Spinne im Netz der Autobewerter und Lernmanagementsysteme“. In: *DeLFI 2015 – Die 13. E-Learning Fachtagung Informatik*. Bd. 247. LNI. GI, 2015, S. 169–181.

[Stö+13] Andreas Stöcker u. a. „Evaluation automatisierter Programmbewertung bei der Vermittlung der Sprachen Java und SQL mit den Gradern *aSQLg* und *Graja* aus studentischer Perspektive.“ In: *DeLFI 2013 – Die 11. E-Learning Fachtagung Informatik*. Bd. 218. LNI. GI, 2013, S. 233–238.

[Stö+14] Andreas Stöcker u. a. „Die Evaluation generischer Einbettung automatisierter Programmbewertung am Beispiel von Moodle und aSQLg.“ In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014, S. 301–304.

24 Ein XML-Austauschformat für Programmieraufgaben

Sven Strickroth, Oliver Müller und Uta Priss

Zusammenfassung
In diesem Kapitel wird die aktuelle Version 1.1 des ProFormA-Austauschformats beschrieben. Dabei wird insbesondere auf Änderungen zu vorhergehenden Versionen und aktuelle Erfahrungen aus vierjähriger Entwicklung und Einsatz eingegangen. Dieses Kapitel basiert auf den Publikationen [Str+14] und [Str+15], aktualisiert für die Version 1.1 und ergänzt um aktuelle Erfahrungen.

24.1 Einleitung

Es existiert eine Vielzahl von Unterstützungssystemen für die Programmierausbildung und für (automatische) Bewertungssysteme (vgl. Kapitel 2 bzw. [AM05; Iha+10; KJH16]). Es kann sogar angenommen werden, dass vermutlich jedes Institut für Informatik ein eigenes System zur Unterstützung des Übungsbetriebes entwickelt hat. Die Gründe für eine solche Entwicklung sind verschieden (vgl. Kapitel 2): Einerseits gibt es einen Trend zur Automatisierung, um auch größere Gruppen von Lernenden adäquat betreuen zu können, wobei viele Systeme dabei für einen konkreten Kontext entwickelt wurden. Andererseits sind forschungsgetriebene Entwicklungen zu nennen, aus denen innovative Unterstützungsansätze hervorgegangen sind. Daraus resultiert, dass es sich um abgeschlossene Systeme handelt, wobei viele Systeme zum Teil ähnliche Funktionen aufweisen (z. B. JUnit für Tests von Java-Programmen verwenden), sich aber in einigen Details (z. B. Softwarearchitektur, unterstützte Programmiersprachen sowie Art der Aufgaben,

Teile dieses Kapitels entstanden im Rahmen des Projekts eCULT, Teilvorhaben eAssessment, gefördert durch das Bundesministerium für Bildung und Forschung unter den Förderkennzeichen 01PL16066H und 01PL11066L. Die Verantwortung für den Inhalt dieses Kapitels liegt bei den Autoren und der Autorin.

der Bewertung oder des Feedbacks) unterscheiden. Aufgrund der speziellen, oftmals kontextbezogenen Anforderungen und aktueller Forschung in diesem Bereich ist auch nicht davon auszugehen, dass sich langfristig eines oder lediglich einige wenige Systeme global durchsetzen – unabhängig davon, ob einige Systeme eine größere Verbreitung erfahren.

Dennoch ist allen Systemen gemein, dass in irgendeiner Art und Weise Aufgaben erdacht und gestaltet werden müssen, die von den Lernenden bearbeitet werden sollen. Dabei muss neben didaktischen Aspekten vor allem darauf geachtet werden, dass keine Uneindeutigkeiten in der Aufgabenstellung entstehen. Dies gilt insbesondere für die Fälle, in denen Lösungen von Lernenden automatisch überprüft werden sollen (vgl. [AM05]). Hier muss mit größter Sorgfalt vorgegangen werden, da selbst kleinste Ungenauigkeiten in der Aufgabenstellung oder Fehler in Tests oder Musterlösungen zu großen Problemen führen können: Bei formativem Feedback können Lernende irritiert oder in eine falsche Richtung gelenkt und bei summativem Feedback im schlimmsten Fall falsche Noten vergeben werden. Dies sorgt für einen nicht zu unterschätzenden Aufwand für die Erstellung von Übungsaufgaben. In der Literatur finden sich Angaben zwischen mehreren Stunden und einer Personenwoche für die Erstellung einer einzigen Aufgabe (vgl. [KJH16]).

Trotz des teilweise enormen Aufwands für die Konzeption von Aufgabenstellungen, Musterlösungen und Tests gibt es bisher kaum etablierte Möglichkeiten, um Aufgaben und Testinhalte zwischen den Lehrenden auf einfache Art und Weise auszutauschen oder systemunabhängig zu nutzen. In diesem Kapitel wird das Austauschformat ProFormA vorgestellt, das diese Lücke eines fehlenden Austauschformats schließen soll und bereits von mehreren Systemen unterstützt wird. Für eine Übersicht über Anforderungen sowie weitere vorgeschlagene Austauschformate und deren Schwächen sei auf [Str+15] verwiesen.

24.2 ProFormA: Ein XML-Austauschformat für Programmieraufgaben

In diesem Abschnitt wird das XML-basierte Austauschformat ProFormA[1] in Version 1.1 beschrieben (in weiteren Publikationen finden sich Beschreibungen älterer Versionen: [Str+14] Version 0.9.1 und [Str+15] Version 0.9.4). Änderungen zwischen den Versionen werden im Folgenden kurz beschrieben. Das Austauschformat liegt sowohl als menschenlesbare Spezifikation mit Anmerkungen

1 Die genaue Spezifikation kann unter https://github.com/ProFormA/taskxml abgerufen werden.

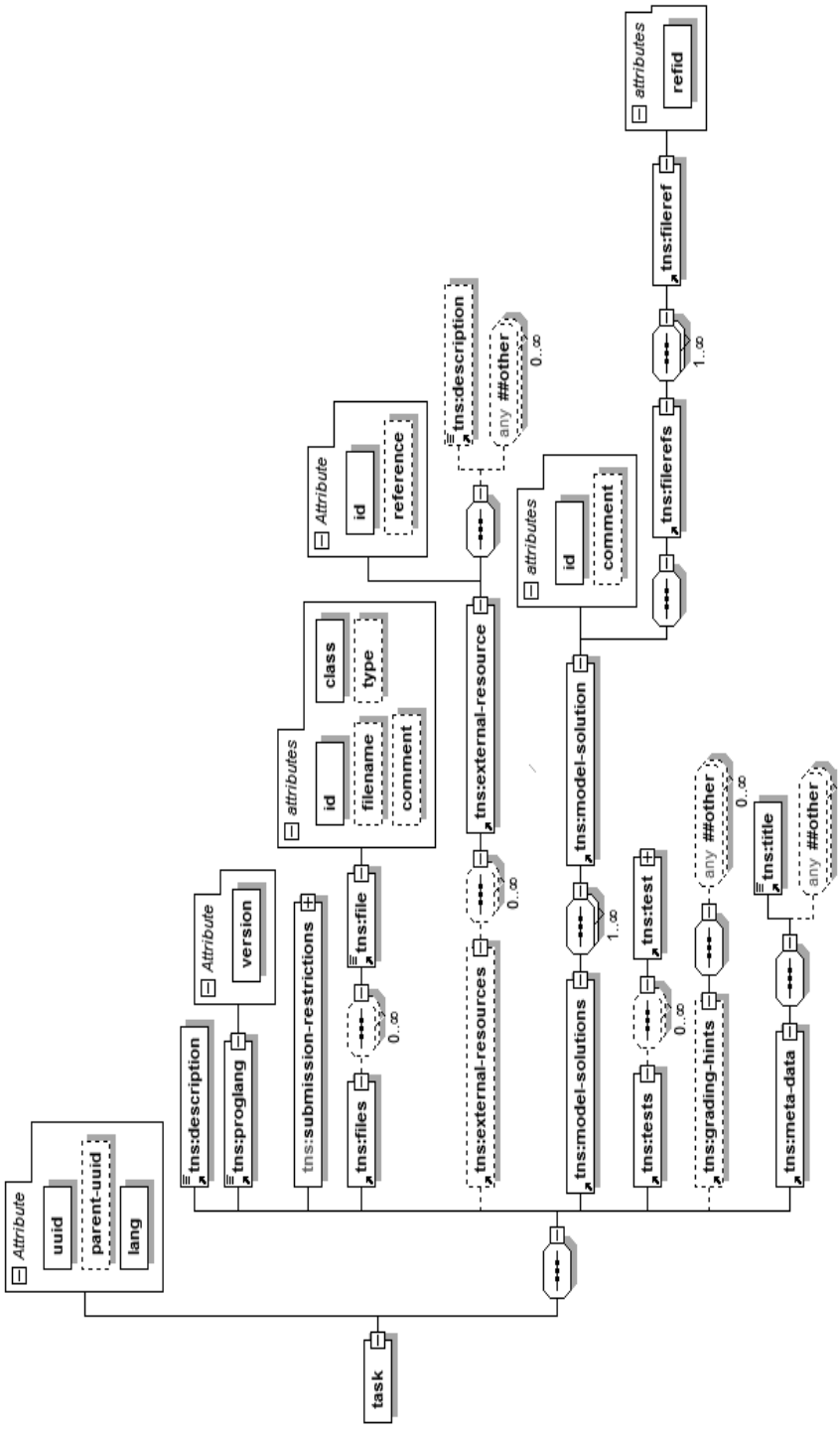


Abbildung 24.1: Strukturbaum des Austauschformats

zur genauen Bedeutung der einzelnen Tags als auch als maschinenlesbare XML-Schemadefinition vor, mit deren Hilfe erstellte Aufgaben auch automatisiert geprüft werden können.

Das Austauschformat spezifiziert für eine Aufgabe (*task*) einen Aufgabentext (Element */task/description*, inkl. Festlegung der Sprache zwecks Internationalisierung im Attribut *lang*), die Programmiersprache (Element */task/proglang* inkl. Version), zugehörige Dateien, technische Details zur Einreichung, Lösungsvorschläge, Metadaten und optional Hinweise zur Bewertung sowie Referenzen zu benötigten externen Ressourcen, die aufgrund ihrer Größe oder Struktur nicht in Verbindung mit der Aufgabe (z. B. als Anhang) ausgeliefert werden können. Mit Version 1.0.1 des ProFormA-Formats ist ein Universally Unique Identifier (UUID) zur eindeutigen Identifikation einer jeden im Austauschformat spezifizierten Aufgabe hinzugekommen. Dabei soll die UUID unter anderem eine Versionierung erleichtern, wobei bei modifizierten Aufgaben zusätzlich die UUID der originalen unmodifizierten Aufgabe als *parent-uuid* angegeben werden kann. Wie schon in vorherigen Versionen können zu einer Aufgabe mehrere Tests gehören, welche jeweils einen Testtyp (konkretes Bewertungsverfahren) und eine spezifische Testkonfiguration beinhalten. Tests werden in der Regel durch übliche Software-Engineering-Werkzeuge (Compiler, Unit-Tests, FindBugs, CheckStyle usw.) ausgeführt. Das Austauschformat selbst muss also im Wesentlichen die Bedingungen für die Ausführbarkeit der Tests festlegen, nicht aber den eigentlichen Testinhalt, welcher im Format der benutzten Werkzeuge gespeichert wird (z. B. skriptbasierte Blackbox-Tests oder XML-Konfigurationen für CheckStyle). Ein importierendes System kann folglich anhand der Testtypen entscheiden, welche Tests es direkt unterstützt und wie diese gegebenenfalls ausgeführt werden.

Abbildung 24.1 zeigt die Struktur des Austauschformats in der aktuellen Version 1.1. Dabei wird die Schachtelung der definierten Elemente und Attribute ersichtlich. Elemente, bei denen in der linken oberen Ecke drei stilisierte Linien zu sehen sind, können direkt mit Inhalten gefüllt werden (z. B. */task/description*). Optionale Elemente sind mit einer gestrichelten Linie umgeben dargestellt (z. B. */task/grading-hints*). XML-Sequenzen werden durch achteckige Kästen mit drei horizontal auf einer Linie befindlichen Punkten symbolisiert, wobei die Kardinalitäten direkt darunter angegeben sind, sofern diese von 1 abweichen. XML-Alternativen (xs:choice) werden ebenfalls durch achteckige Kästen mit drei vertikalen Punkten jedoch mit einem Auswahlzeiger (xs:choice, z. B. *submission-restrictions* in Abbildung 24.4) oder verbundenen horizontalen Linien (ebenfalls dort) dargestellt. Insbesondere sollen hier auch die Boxen mit „any ##other“ erwähnt werden: Das vorgeschlagene Austauschformat definiert lediglich das Grundgerüst, den gemeinsamen Nenner für Programmieraufgaben, und soll gleichzeitig

zum Beispiel für neue Programmiersprachen, Testverfahren und besondere Fähigkeiten von implementierenden Systemen erweiterbar sein. Zu diesem Zweck sind ähnlich zu Programmierframeworks sog. „Hotspots“ vorgesehen, an denen weitere Elemente aus anderen XML-Namespaces importiert und genutzt werden können – somit bleibt das Format für systemspezifische Konfigurationen und (neue) Bewertungsverfahren flexibel lokal erweiterbar. Zudem ist die vorgeschlagene Spezifikation versioniert (über die Namespace URI), um spätere Ergänzungen oder Änderungen der Spezifikation zu erlauben (die aktuelle Version trägt die URI „urn:proforma:task:v1.1“). Seit Version 0.9.4 des Formats müssen sämtliche Tags mit einem Namespace-Identifier voll qualifiziert werden. Diese Änderung wurde vorgenommen, da Parser für Aufgaben so einfacher implementiert werden können.

Listing 1 zeigt ein konkretes Minimalbeispiel einer Aufgabe im vorgeschlagenen Format, wie sie in einem typischen Einführungskurs in die Java-Programmierung zum Thema Rekursion oder Schleifen vorkommen kann: Die Berechnung der n-ten Fibonacci-Zahl.

```
<?xml version="1.0" encoding="UTF-8"?>
<p:task xmlns:p="urn:proforma:task:v1.1" lang="en" uuid="66217f7d-ca8e-4666-a6f8-934c79c24206">
    <p:description>Calculate the n-th Fibonacci number. As a reminder:
                   The first two Fibonacci numbers are 0 and 1.
                   The following numbers are the sum of the previous two numbers
                   (0, 1, 1, 2, 3, 5, 8, 13, 21...). The class should be named Fibonacci
                   and the method to be written fibonacci has an int as an input parameter.
    </p:description>
    <p:proglang version="1.4">java</p:proglang>
    <p:submission-restrictions>
        <p:files-restriction>
            <p:required filename="Fibonacci.java"/>
        </p:files-restriction>
    </p:submission-restrictions>
    <p:files>
        <p:file class="internal" filename="FibonacciTest.java" id="f1" type="embedded">
            import junit.framework.TestCase;
            public class FibonacciTest extends TestCase {
              public void testPos() { assertEquals(13, Fibonacci.fibonacci(7)); }
              public void testNull() { assertEquals(0, Fibonacci.fibonacci(0)); }
            }
        </p:file>
        <p:file class="internal" filename="Fibonacci.java" id="f2" type="embedded">
            public class Fibonacci {
              public int fibonacci(int i) {
              if (i &lt;= 0) return 0;
              else if (i == 1) return 1;
              return fibonacci(i - 2) + fibonacci(i - 1);}
            }
        </p:file>
    </p:files>
    <p:model-solutions>
        <p:model-solution id="m1">
            <p:filerefs><p:fileref refid="f2"/></p:filerefs>
        </p:model-solution>
    </p:model-solutions>
    <p:tests>
        <p:test id="t1">
            <p:title>Compilation test</p:title>
            <p:test-type>java-compilation</p:test-type>
            <p:test-configuration/>
        </p:test>
        <p:test id="t2">
            <p:title>Calculation test</p:title>
```

```
            <p:test-type>unittest</p:test-type>
            <p:test-configuration xmlns:u="urn:proforma:tests:unittest:v1.1">
                <p:filerefs>
                    <p:fileref refid="f1"/>
                </p:filerefs>
                <u:unittest framework="JUnit" version="3">
                    <u:entry-point>FibonacciTest</u:entry-point>
                </u:unittest>
            </p:test-configuration>
        </p:test>
    </p:tests>
    <p:meta-data>
        <p:title>Calculation of n-th Fibonacci number</p:title>
    </p:meta-data>
</p:task>
```

Listing 1: Beispielinstanz einer Programmieraufgabe

Zum besseren Verständnis wird im Folgenden genauer auf die zentralen Elemente des Austauschformats eingegangen.

24.2.1 Dateianhänge

Dateianhänge können entweder direkt in das XML-Dokument eingebunden oder über die Angabe des entsprechenden Dateinamens in einem ZIP-Archiv referenziert werden. Der letztere Fall ist vor allem für Binärdaten vorgesehen, damit das XML-Dokument nicht zu groß und unübersichtlich wird. Dabei ist lediglich vorgegeben, dass sich das XML-Dokument als „task.xml“ im Wurzelverzeichnis befinden muss, damit ein ZIP-Archiv korrekt als ProFormA-Aufgabe erkannt werden kann – die Organisation und Benennung weiterer Dateien ist nicht festgelegt.

Eine Datei, unabhängig ob es sich um eine Musterlösung, einen Test oder sonstiges handelt, wird durch ein *file*-Element repräsentiert, das unterhalb des Elements */task/files* eingefügt wird. Die Verwaltung der Dateien an einer zentralen Stelle soll zum einen für Übersichtlichkeit sorgen und zum anderen zur Vermeidung von Redundanzen beitragen. Für jede Datei muss eine innerhalb des XML-Dokuments eindeutige ID zur späteren Referenzierbarkeit (z. B. bei der Testdefinition) und eine Klasse (Attribut *class*) angegeben werden, die den Zweck und die Zugangsrechte beschreibt. Vorgesehen sind in diesem Zusammenhang Codevorlagen (template), Bibliotheken (library), Eingabedaten (inputdata), zusätzliche Informationen oder Anweisungen zur Bearbeitung einer Aufgabe (instruction), interne Dateien (z. B. Testtreiber oder Dateien zu einer Musterlösung, internal) sowie interne Bibliotheken (z. B. notwendige Bibliothek zur Ausführung eines Tests, internal-library, neu seit Version 0.9.4). „Intern“ bezieht sich dabei auf die Einschränkung, dass diese Dateien für Lernende grundsätzlich nicht zugänglich sind. Ob eine Datei direkt eingebunden ist oder im ZIP-Archiv referenziert wird, wird durch das Attribut *type* festgelegt. Zu jedem *file*-Element kann optional ein

Kommentar hinzugefügt werden (Attribut *comment*), um zusätzliche Informationen zur Datei anzugeben.

24.2.2 Musterlösungen

Für eine Aufgabe muss des Weiteren mindestens eine Musterlösung hinterlegt werden. Für diese doch recht strenge Anforderung gibt es mehrere Gründe: Zum einen sind diese technischer Natur, da es Systeme gibt, die eine Musterlösung benötigen, zum Beispiel um einen JUnit-Test zu übersetzen. Zum anderen hat dies auch praktische Gründe, da es eine Musterlösung zusammen mit der Aufgabenbeschreibung Lehrenden erlaubt die Schwierigkeit und Angemessenheit einer Aufgabe für den Einsatz im eigenen Kurs einzuschätzen.

Musterlösungen werden innerhalb des Elements */task/model-solutions* separat behandelt. Ein Element *model-solution* steht für eine konkrete Musterlösung, die aus einer Menge von Dateien bestehen kann (z. B. eine Musterlösung einer Java-Programmieraufgabe, bestehend aus mehreren .java-Dateien). Wie weiter oben bereits angedeutet, wird jede Datei, die zu einer Musterlösung gehört, in einem separaten *file*-Element definiert. Jedes dieser *file*-Elemente, wird über die Angabe seiner ID in einem */task/model-solution/filerefs/filref* -Element der jeweiligen Musterlösung zugeordnet.

Vor Version 0.9.4 konnte nur eine einzige Musterlösung angegeben werden. Zudem wurde noch nicht das bereits bestehende Konzept der *filerefs* genutzt, da Musterlösungen ursprünglich als separat angesehen wurden. Eine Änderung wurde vorgenommen, damit Redundanzen vermieden werden können, wenn zum Beispiel eine Datei zugleich Musterlösung und eine Eingabe für einen Test darstellt. Zur Angabe detaillierterer Informationen zu einer Musterlösung innerhalb eines *model-solution*-Elements lässt sich ebenfalls seit Version 0.9.4, wie bei einem *file*-Element, ein Attribut *comment* nutzen. Die Idee dahinter besteht darin, dass so unterschiedliche Musterlösungen menschenlesbar charakterisiert werden können (z. B. zur Unterscheidung von alternativen Lösungswegen, wie iterativ vs. rekursiv).

24.2.3 Tests und Bewertungsverfahren

Anzuwendende Tests oder Bewertungsverfahren werden innerhalb des Elements */task/tests* in einzelnen *test*-Elementen definiert (vgl. Abbildung 24.2). Neben der Angabe, welche Arten von Tests für eine Aufgabe zur Verfügung stehen (z. B.

Compile/Syntax- oder Unit-Tests, Element *test/test-type*) und der Angabe eines Titels (Element *test/title*), lässt sich die genaue Konfiguration eines Tests beschreiben (*test/test-configuration*). Zur Konfiguration eines Tests gehört, welche weiteren Dateien (z. B. Testtreiber) oder externen Ressourcen benötigt werden. Die für benötigte Dateien zu definierenden */files/file*-Elemente werden analog zu den Musterlösungen über *fileref* -Elemente referenziert. Darüber hinaus ist vorgesehen, dass direkt innerhalb eines *test-configuration*-Elements für jeden Testtyp ein eigener XML-Namespace genutzt werden kann, um testspezifische Parameter austauschen zu können (z. B. den Namen der aufzurufenden Testklasse für Unit-Tests, vgl. Listing 1).
Innerhalb des offiziellen ProFormA-Namensraumes sind aktuell die folgenden drei Testtypen vorgesehen:

java-compilation Der Java-Syntaxtesttyp benötigt keine weitere Testkonfiguration (vgl. Listing 1), da ein implementierendes System durch die abgegebenen Dateien und die referenzierten Dateien notwendige Bibliotheken automatisch laden kann.

unittest Dieser Testtyp ist seit Version 0.9.4 für allgemeine Unit-Tests vorgesehen. Vorher sollten für verschiedene Sprachen unterschiedliche Testtypen genutzt werden, was einen erhöhten Implementierungs- und vor allem auch Wartungsaufwand bedeutet hätte. Zur Konfiguration der Tests gibt es eine Spezifikation mit dem Namespace „urn:proforma:tests:unittest:v1.1“ (vgl. Listing 1). Dort ist ein einziges Element *unittest* spezifiziert (vgl. Abbildung 24.3), wo in den Attributen *framework* sowie *version* das zu verwendende Unit-Test-Framework (z. B. JUnit in Version 3) angegeben werden kann. Ferner bedürfen Unit-Tests der Angabe eines Einstiegspunkts oder Main-Klasse (Element *entry-point*). Zusammen mit den *filerefs* können die Unit-Tests ausgeführt werden.

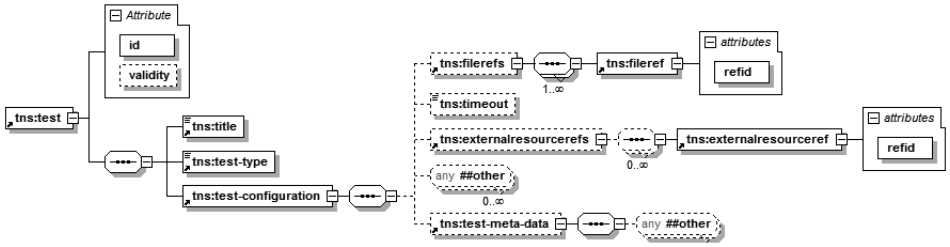


Abbildung 24.2: Strukturbaum des Elements */task/tests/test*

regexptest Dieser neueste spezifizierte Testtyp wird über das Element *regexp-test* des XML-Namespaces „urn:proforma:tests:regexptest:v0.9“ konfiguriert (vgl. Abbildung 24.3). Dieser Testtyp dient zur Überprüfung der Ausgaben einer Lernerlösung anhand von vorgegebenen regulären Ausdrücken. Dabei können mehrere reguläre Ausdrücke entweder als *regexp-allow* oder *regexp-disallow*-Elemente angegeben werden (inkl. weiterer Attribute um das Matching zu beeinflussen), die entweder alle zutreffen müssen beziehungsweise nicht zutreffen dürfen, um den Test zu bestehen. Wie die Lernerlösung ausgeführt werden soll, wird wie bei den oben genannten Tests über die Elemente *entry-point* und *parameter* für Kommandozeilenparameter festgelegt.

Zudem lassen sich innerhalb eines *test-configuration/test-meta-data*-Elements mit der Nutzung eines eigenen XML-Namespaces systemspezifische Metadaten (z. B. für Beschränkungen) für einen Test hinterlegen.
Grundsätzlich sollten Aufgabenspezifikationen in sich abgeschlossen sein, da nur auf diese Weise Aufgaben über einen langen Zeitraum ohne möglichen Verlust von Informationen oder Komponenten einsetzbar sind. Es haben sich jedoch Szenarien gezeigt, in denen die Nutzung von externen Ressourcen unvermeidbar ist, beispielsweise für Aufgaben, die automatisch bewertet werden sollen und dafür

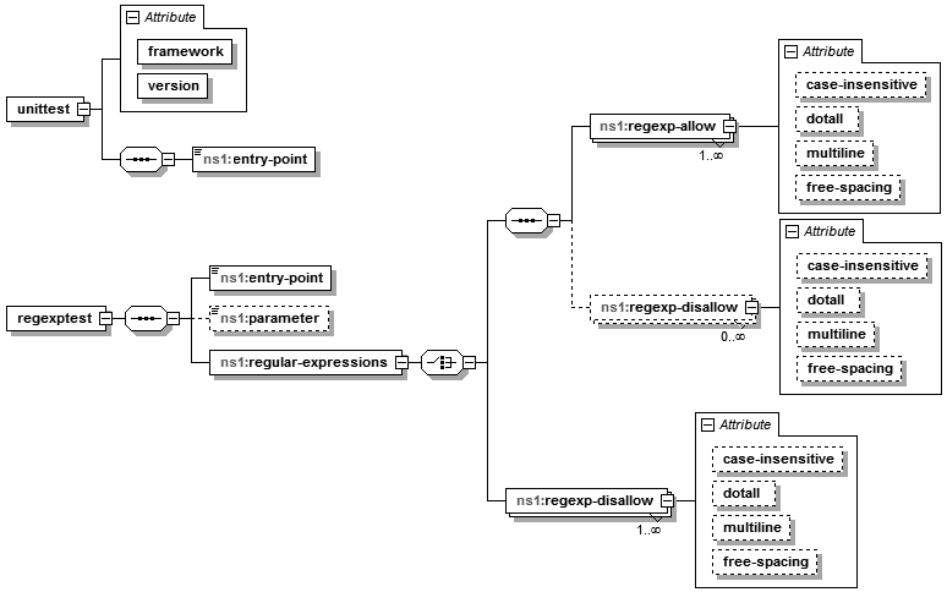


Abbildung 24.3: Strukturbäume der bisher spezifizierten Testtypen

sehr große Dateien als Abhängigkeiten benötigen, die nicht sinnvoll mit der Aufgabenspezifikation zusammengeführt werden können. Für diese Szenarien wurde das */task/external-resources* in Version 0.9.3 zur ProFormA-Spezifikation hinzugefügt, mit dem externe Ressourcen über eine eindeutige Referenz (*reference* Attribut) verknüpft werden können. Optional ist ein *description*-Element, in dem die Ressource näher beschrieben werden kann, und ein systemspezifischer XML-Namespace vorgesehen. Eine externe Ressource kann u. a. eine URL zu einem Datenbankabbild sein (z. B. ftp://ftp.fu-berlin.de/pub/misc/movies/database). Die Semantik und das genaue Format der Referenzen ist aktuell noch nicht genau spezifiziert, jedoch in dieser generischen Form integriert, um weitere Erfahrungen mit neuen Einsatzmöglichkeiten (vgl. Kapitel 23) sammeln zu können. Die Referenzierung externer Ressourcen, die innerhalb von *external-resource*-Elementen spezifiziert werden, erfolgt analog zu den *filerefs* unter Angabe ihrer jeweiligen ID in *externalresourcerefs/externalresourceref*-Elementen unterhalb von *test-configuration*-Elementen.

24.2.4 Abgabebeschränkungen

Das Element */task/submission-restrictions* bietet die Möglichkeit, Einschränkungen für die Einreichung von Lösungen zu definieren (vgl. Abbildung 24.4). In Version 0.9.1 hieß dieses Element noch *submission* und wurde zur Verbesserung des Verständnisses in nachfolgenden Versionen umbenannt. Zudem war es bis zur Version 0.9.4 lediglich möglich die maximale Größe (in Byte) der hochzuladenden Dateien festzulegen und über reguläre Ausdrücke Einschränkungen an den Dateinamen vorzunehmen. Mit Version 1.0.0 wurden diese Möglichkeiten deutlich feiner spezifiziert, wobei eine automatische Konvertierung sowohl zu älteren Versionen als auch zur Version 1.0.0 möglich ist. Gründe dafür bestanden darin, dass die Semantik der bisherigen Spezifikation sich als nicht klar genug herausgestellt hatte und Anforderungen von Systemen wie Praktomat (Limitierung der hochzuladenden Dateien über MIME-Type-Beschränkungen) und JACK (genaue Angabe von erforderlichen und optionalen Dateien, um im System zum Beispiel eine Abgabe auf Vollständigkeit prüfen zu können oder pro Datei ein eigenes Upload-Feld bereitzustellen) bisher nicht vollständig erfüllt wurden.

Beschränkungen oder Beschreibungen von abzugebenden Dateien können in einer von drei Formen erfolgen: *archive-restriction* (entspricht zu großen Teilen der Spezifikation aus Version 0.9.4), *files-restriction* oder *regexp-restriction* (entspricht fast vollständig der Spezifikation aus Version 0.9.4). Dabei kann in allen drei Fällen optional die maximale erwartete Größe (in Byte, *max-size*) einer

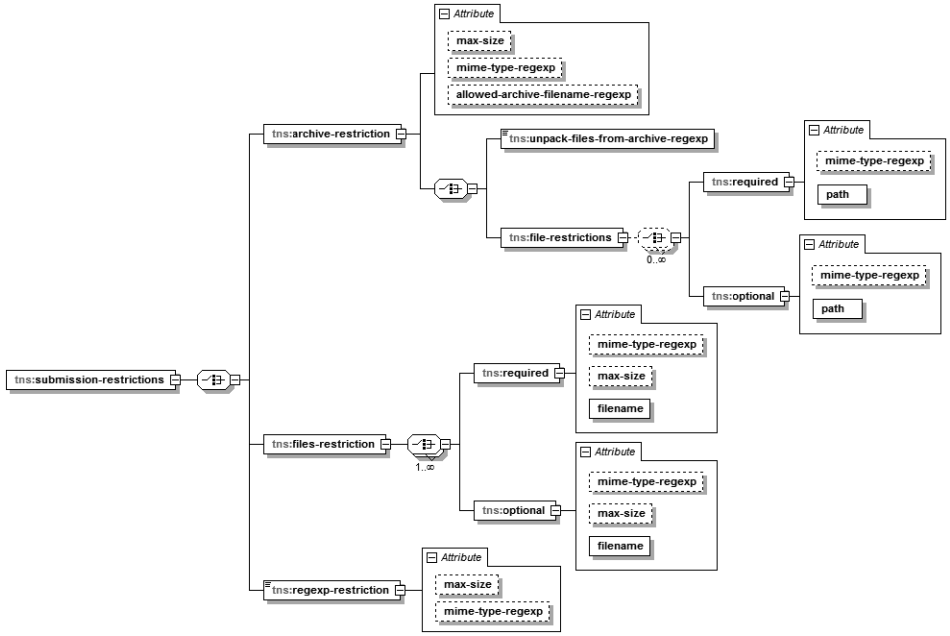


Abbildung 24.4: Strukturbaum des Elements */task/submission-restrictions*

zur Abgabe einer Lösung im System hochzuladenden Datei festgelegt werden. Natürlich kann ein implementierendes System eigene Größenbeschränkungen besitzen und den jeweils kleineren Wert anwenden. Die Möglichkeit, Einschränkungen für die Namen hochzuladender Dateien anzugeben, ist insbesondere für Java-Aufgaben relevant, da dort abhängig vom Klassennamen Anforderungen an den Dateinamen gestellt werden. Möchte ein Lernender eine Lösung einer Java-Aufgabe in Form eines Dateiuploads innerhalb eines Systems einreichen, so kann zunächst vom System verifiziert werden, ob der Name einer hochzuladenden Datei sämtliche Bedingungen eines vorgegebenen regulären Ausdrucks erfüllt beziehungsweise dem vorgegebenen Dateinamen entspricht. Ist dies nicht der Fall, so kann die Datei nicht hochgeladen und eine Warnung angezeigt werden. Somit lässt sich unter anderem sicherstellen, dass Tests nicht aufgrund eines falschen Dateinamens fehlschlagen.

Wird die Beschränkung *files-restriction* gewählt, so können die erwarteten Dateien einzeln spezifiziert werden. Dabei kann zwischen unbedingt erforderlichen (*required*) und optionalen Dateien (*optional*) einer Abgabe unterschieden werden. Es ist jeweils die Angabe des genauen Dateinamens (*filename*) sowie optional die maximale erwartete Dateigröße und eines regulären Ausdrucks als Anforderung an den MIME-Typ möglich. Dadurch kann insbesondere die Vollständigkeit einer

Abgabe frühzeitig erkannt werden. Für Systeme, welche die detaillierte Angabe von einzelnen Dateien nicht unterstützen und einen regulären Ausdruck erwarten (wie in früheren Versionen des Formats), kann dieser automatisch aus diesen präziseren Angaben generiert werden.

Die Beschränkung *regexp-restriction* erlaubt es über einen regulären Ausdruck Bedingungen an die Namen hochzuladender Dateien zu stellen, wobei zusätzlich mit Version 1.0 über ein Attribut auch Bedingungen an den MIME-Typ gestellt werden können.

Für komplexere Aufgaben, die zum Beispiel als Archiv abgegeben werden sollen, können über das Element *archive-restriction* Bedingungen an hochzuladende Archive festgelegt werden. Neben der Vorgabe eines erlaubten Archivnamens (Attribut *allowed-archive-filename-regexp*) und einer maximalen Archivgröße (Attribut *max-size*) können zusätzlich die Namen der automatisch zu extrahierenden Dateien entweder über einen regulären Ausdruck (Element *archive-restriction/unpack-files-from-archive-regexp*) oder der genauen Angabe von Dateipfaden innerhalb des hochzuladenden Archivs (Element *archive-restriction/file-restrictions*) angegeben werden (vgl. Element */task/submission-restrictions/files-restriction*).

24.2.5 Bewertungsschema und Metadaten

Für eine Aufgabe kann ein Bewertungsschema (Element */task/grading-hints*) sowie weitere Metadaten (Element */task/meta-data*) angeben werden. Innerhalb des Elements *meta-data* ist lediglich ein Element für die Angabe eines Titels für die Aufgabe verpflichtend vorgesehen (*title*). Weiterhin kann ein eigener XML-Namespace für systemspezifische Metadaten genutzt werden. Metadaten zur Kategorisierung wurden bisher bewusst nicht in das Format aufgenommen, da es für Metadaten zum einen etablierte Standards gibt (vgl. IEEE LOM, Dublin Core) und zum anderen diese oft nicht oder nur halbherzig benutzt werden [God04].

Analog zum Element *test-meta-data* kann im Element */task/grading-hints* ein eigener XML-Namespace für systemindividuelle Angaben zu Bewertungsschemata benutzt werden. Die Spezifikation von Bewertungsschemata wurde im ProFormA-Austauschformat bisher ebenfalls explizit ausgenommen, da diese oftmals sehr von den Vorstellungen und Vorlieben der Institutionen beziehungsweise Lehrenden abhängen (eine Untersuchung von [CR13] bestätigt eine sehr hohe Heterogenität). Dennoch können die Angaben von weiteren Lehrenden gelesen und eventuell adaptiert werden – jedoch in der Regel nicht automatisch.

Insgesamt können durch den expliziten Einsatz von individuellen XML-Namespaces, die nicht zur offiziellen ProFormA-Spezifikation gehören, aufgaben- oder

testbezogene Attribute spezifiziert werden, die für ein bestimmtes System benötigt werden, für andere Systeme aber nicht relevant sind. Systemindividuelle Angaben gehen dadurch beim Export nicht verloren. Ein vollständiger Export einer Aufgabe in das Austauschformat mit anschließendem verlustfreien Reimport in dasselbe System ist somit prinzipiell möglich. Wird eine aus einem System exportierte Aufgabe in ein anderes System importiert, so ist sichergestellt, dass nur relevante Daten importiert werden, die auch vom letztgenannten System berücksichtigt und verarbeitet werden können.

24.3 Erfahrungen mit dem Austauschformat und Diskussion

In diesem Abschnitt werden die aktuelle Unterstützung in verschiedenen Systemen und bisherige Erfahrungen mit dem Austauschformat beschrieben, Hinweise für Implementierungen gegeben sowie mögliche Probleme und Lösungen diskutiert.

24.3.1 Aktuelle Unterstützung von ProFormA

Die ProFormA-Spezifikation wird bereits seit ungefähr vier Jahren von mehreren Systemen (GATE (vgl. Kapitel 13), JACK (vgl. Kapitel 9), Graja (vgl. Kapitel 11) und einer Praktomat-Version (vgl. Kapitel 10)) zumindest zum Export von Java-Aufgaben unterstützt. Das System aSQLg (vgl. Kapitel 12) unterstützt die ProFormA-Spezifikation seit 2017 für SQL-Aufgaben.

Neben dem Export wurde für drei der aufgeführten Systeme (GATE, JACK, Praktomat) zudem ein Import implementiert. In diesem Zusammenhang wurden für diese Systeme systemübergreifende Importe (Export aus einem System und Import in ein anderes System) und systeminterne Importe (Export aus einem System und Reimport in dasselbe System) zu Testzwecken durchgeführt. Ein Export ist prinzipiell aus allen aufgeführten Systemen möglich. Beim JACK-System werden jedoch die notwendigen Musterlösungen zum aktuellen Zeitpunkt noch nicht exportiert (vgl. Abschnitt 24.2.2). GATE und Praktomat erzeugen vollständig valide XML-Dokumente gemäß Version 1.0.1 der Spezifikation. Ein Import ist sowohl von Version 0.9.4 als auch 1.0.1 in beiden Systemen möglich. Unterstützung für die Version 1.1 ist schon oder wird aktuell implementiert.

Ein systeminterner und systemübergreifender Import ist grundsätzlich für jedes der drei Systeme möglich. Die vollständige Ausführbarkeit einer importierten

Aufgabe, die zuvor aus einem anderen System exportiert wurde, ist jedoch nicht immer gewährleistet. Dies ist zum einen durch die unterstützten Bewertungsverfahren und zum anderen durch besondere Systemfeatures begründet. Beispielsweise werden alle drei Systeme für Java-Programmieraufgaben eingesetzt. GATE und Praktomat unterstützen beide JUnit-Tests – zwischen diesen beiden Systemen lassen sich daher Aufgaben samt Syntax- und JUnit-Tests vollständig austauschen und ausführen. Das JACK-System hingegen unterstützt keine JUnit-Tests, sondern setzt auf andere Bewertungsverfahren, die zurzeit exklusiv im JACK-System Anwendung finden. Im GATE-System findet sich (noch) keine vollständige Unterstützung für RegExp-Tests; sie erlauben bisher nur das Matching von korrekten Lösungen (*not-allowed-regexp* fehlt). Dennoch lassen sich durch die Möglichkeit systemspezifische Elemente in Form von Metadaten zu exportieren (Elemente */task/meta-data* und */task/tests/test/test-configuration/test-meta-data*), wie weiter oben bereits erwähnt, wesentliche Elemente von Programmieraufgaben systemübergreifend austauschen und zugleich systeminterne Importe verlustfrei durchführen.

24.3.2 Systemübergreifender Austausch in der Praxis

Wie im vorherigen Abschnitt beschrieben, ist ein Austausch von Aufgaben zwischen den Systemen GATE, JACK und Praktomat seit mehreren Jahren mithilfe der ProFormA-Spezifikation möglich. Einsetzende Institutionen besitzen zudem recht gut gefüllte Pools von Programmieraufgaben mit teilweise sehr ausführlichen Tests (GATE an der TU Clausthal: ca. 250 Java-Aufgaben, JACK an der Universität Duisburg-Essen: ca. 70 Java-Aufgaben, Praktomat an der Hochschule Ostfalia: 60 Aufgaben in Java, Python und einer mathematischen Programmiersprache). Dennoch wurde ein hochschulübergreifender Austausch bisher nur relativ selten praktiziert – auch wenn allgemeines Interesse daran besteht, neue, bewährte Aufgaben zu nutzen. Hauptgründe sind sicherlich, dass Lehrende bisher keinen Zugriff auf die Aufgabenpools der anderen Institutionen besitzen und sich folglich keinen Überblick über potenziell passende Aufgaben verschaffen können. Dies erfordert bisweilen einen persönlichen Kontakt, wobei Aufgaben schließlich über E-Mail ausgetauscht werden.

Zur Verbesserung des Austausches wäre daher eine Möglichkeit wünschenswert, direkt aus dem lokal genutzten System neue Aufgaben suchen und importieren zu können. Dies würde zudem die Attraktivität von ProFormA für andere Systeme erhöhen und sich wahrscheinlich auch wieder positiv auf den Aufgaben-

bestand sowie die Qualität der Lehre durch Nutzung bewährter Aufgaben auswirken.

Lösungsmöglichkeiten bestehen im Aufbau eines dezentralen Verbundes wie es zum Beispiel beim LON-CAPA-Verbund der Fall ist oder eines zentralen Repositories. Beide Ansätze haben charakteristische Vor- und Nachteile. In einem dezentralen Verbund müssten in allen Systemen Peer-to-Peer-Technologien implementiert und genutzt werden – ein Aufwand, der für jedes sonst unabhängige System durchgeführt und auch aktuell gehalten werden muss. Zudem ist es erforderlich, dass alle Server gut an das Internet angebunden und stets erreichbar sind. Dafür müssten jedoch keine Aufgaben auf einem „fremden“ System gespeichert werden und lokale Statistiken wären möglich. Ein zentrales Repository würde lediglich die Implementierung einer Schnittstelle in allen teilnehmenden Systemen erfordern. Die Schnittstelle könnte wiederum versioniert sein, so dass Änderungen nur sehr selten notwendig werden. Statistiken könnten global erhoben werden, jedoch müsste eine langfristige Bereitstellung sowie Finanzierung des Repositories sichergestellt werden. In beiden Fällen ist es zudem von enormer Bedeutung, dass ein Zugriff auf die vollständigen Aufgabenspezifikationen für Lernende ausgeschlossen ist (vgl. LON-CAPA [Kor+08]), da diese jeweils eine Musterlösung und auch die Tests beinhalten. Dadurch muss darüber hinaus auch sichergestellt werden, dass langfristig ebenfalls eine Instanz existiert, die über Aufnahme oder Zugriff entscheiden kann.

Nach Abwägung der Vor- und Nachteile, wird der Aufbau eines Repositories für Aufgaben angestrebt. Hierfür muss nicht unbedingt ein eigenes Repository aufgesetzt werden, sondern grundsätzlich kommt auch eine Mitnutzung bestehender Repositories in Betracht, wobei dabei sicherlich auch auf weitere verbreitete Standards wie zum Beispiel IMS Content Packaging[2] zurückgegriffen werden kann. Leider konnte bisher keine zufriedenstellende Lösung in Bezug auf die Mitnutzung unter Erreichung der oben genannten Anforderungen gefunden werden. Derweil laufen Gespräche mit von Universitäten unabhängigen Vereinen, die sich der Förderung von E-Learning widmen, um sowohl die Auswahlinstanz als auch das Repository dort anzusiedeln.

24.3.3 Authoring-Funktionalität

Eigentlich wird kein spezieller Editor für das Format benötigt, da jedes existierende System bereits eigene Authoring-Funktionalitäten bereitstellt. Dennoch kann ein systemunabhängiger Editor zentrale Vorteile bieten:

2 http://www.imsglobal.org/content/packaging/

- stete Unterstützung der aktuellsten ProFormA-Version durch unabhängige Entwicklung,
- Validierung, Einlesen und Bearbeiten von Aufgaben,
- Konvertierung von Aufgaben aus und in verschiedene Formatversionen.

Daher wurde ein eigener quelloffener Editor entwickelt[3] (vgl. Abbildung 24.5). Dieser Editor ist komplett in JavaScript geschrieben. Dadurch kann dieser sowohl systemunabhängig genutzt als auch relativ leicht in bestehende Systeme integriert werden. Aktuell wird der Editor produktiv an der Hochschule Ostfalia zur Erstellung von Aufgaben eingesetzt, weil dort der Praktomat, der die eigentliche Auswertungsfunktion für die Programmieraufgaben bereitstellt, komplett in das dort eingesetzte LMS LON-CAPA integriert ist und für die Lehrenden unsichtbar sein soll (vgl. Kapitel 22). Anstatt LON-CAPA um Authoring-Funktionalität für Programmieraufgaben zu ergänzen, erschien es sinnvoller, den Editor gleich so zu konzipieren, dass er in verschiedene Systeme integriert werden kann. Der Editor sieht vor, dass für jedes System eine JavaScript-Datei angelegt wird, welche die Metadaten und speziellen Namespaces, die nur von dem System benötigt werden, bereitstellt. Für die Benutzung mit LON-CAPA produziert der Editor außer dem Austauschformat auch noch eine zusätzlich von LON-CAPA benötigte Konfigurationsdatei. Eine derart vollständige Unterstützung fehlt im Editor noch für die anderen Systeme.

24.3.4 Erweiterbarkeit der Spezifikation

Beim Entwurf der Spezifikation wurden die Anforderungen unterschiedlicher Systeme (u. a. GATE (vgl. Kapitel 13), JACK (vgl. Kapitel 9), einer Praktomat-Version (vgl. Kapitel 10) und ViPS (vgl. Kapitel 14) sowie grundsätzlich 29 weitere) an Aufgabenspezifikationen beachtet (vgl. [Str+15]). Dadurch ist sichergestellt, dass das Format prinzipiell nicht nur die Anforderungen eines einzelnen Systems und lediglich die dort unterstützten Programmiersprachen implementiert, wie es für die meisten bestehenden Formate der Fall ist, sondern eine Obermenge der Anforderungen umfasst (vgl. [Str+15]). Zudem wird die vorgeschlagene Spezifikation nicht als festes „one-size-fits-all“-Format angesehen und systemspezifische Aspekte werden nicht ignoriert, sondern Erweiterungen einzelner Systeme oder neuer Testverfahren können über eigene, frei definierbare XML-Namespaces eingebettet werden. Dies soll zudem die Hürde senken ProFormA zu unterstützen, da auf diese Weise systemspezifische Aspekte nicht ausgelassen werden müssen.

3 https://github.com/ProFormA/formatEditor

Eine Erweiterbarkeit (über XML-Namespaces) birgt jedoch grundsätzlich auch die Gefahr einer „Zerfaserung“ des Austauschformats, wenn elementare Aspekte dort mehrfach oder unterschiedlich definiert werden. Diesem Aspekt wurde zum einen dadurch Rechnung getragen, dass wesentliche Aspekte bereits im vorgeschlagenen Format selbst definiert wurden. In der Praxis hat sich diese Annahme bisher bestätigt. Zum anderen können durch die Versionierung der Spezifikation später noch Änderungen und Erweiterungen vorgenommen werden, um „verbreitete“ Aspekte mit in das Format aufzunehmen. Doppeldefinitionen traten bisher nicht auf. Ein Grund dafür ist sicherlich, dass bisher alle Maintainer von Systemen mit ProFormA-Unterstützung an der Entwicklung von ProFormA (zumindest am Rande) beteiligt sind und mögliche Fälle direkt zur Diskussion stellen. Unabhängig davon haben sich seit der in [Str+14] veröffentlichten Version 0.9.1 einige Verbesserungsmöglichkeiten gezeigt (vgl. vorheriger Abschnitt). Zum Beispiel wurden Aspekte erweitert, die sich in älteren Versionen der Spezifikation nicht abbilden ließen (z. B. Modellierung von einzelnen Dateien bei der Abgabe, auch zur Vollständigkeitsprüfung) oder der Verbesserung des Verständnisses dienen (z. B. Umbenennungen) beziehungsweise die Handhabung vereinfachen (bspw.

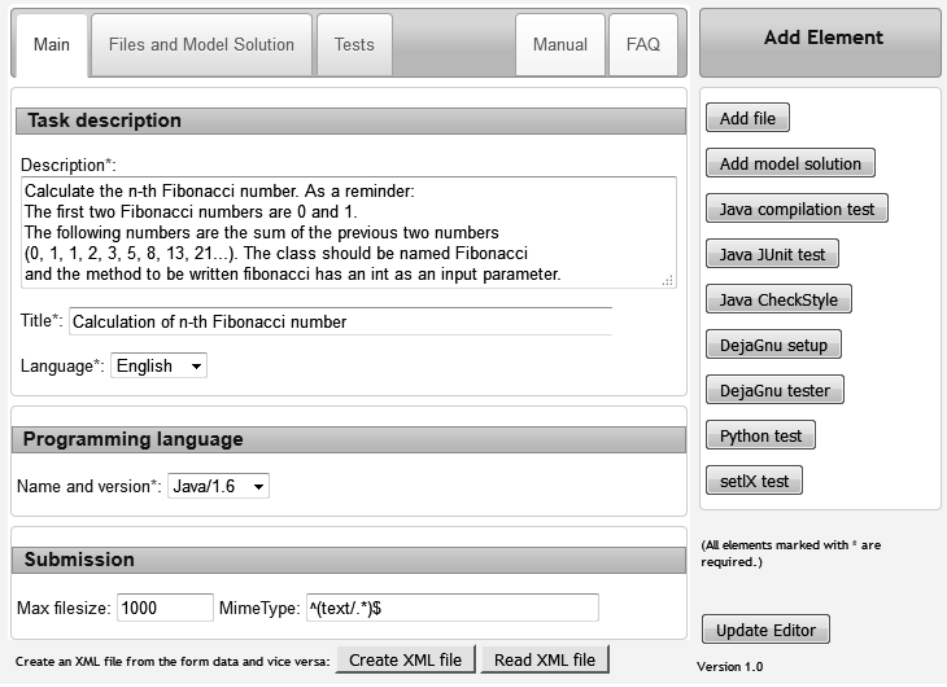


Abbildung 24.5: Benutzeroberfläche des ProFormA-Editors

Namespace-Präfixe für XML-Elemente). Bei frühen Versionen ergaben sich häufiger und größere Änderungen – da zu diesem Zeitpunkt auch nur recht wenige Aufgabenspezifikationen vorlagen, waren auch größere Veränderungen ohne Bedenken möglich – bei Versionen ab 0.9 hat sich bereits eine gewisse Stabilität mit jeweils ungefähr einem Jahr zwischen neuen Versionen gezeigt. Die Verwaltung von Änderungen an der Spezifikation mit Hilfe von Git (auf GitHub) hat sich allgemein als sehr hilfreich erwiesen – nicht nur um Änderungen verfolgen zu können, sondern auch um Vorschläge zu diskutieren und zu entwickeln.

Versionierung verhindert jedoch nicht, dass sich bei Änderungen an der Spezifikation notwendige Anpassungen an implementierenden Systemen ergeben, um Aufgaben importieren zu können, die mit der neuen Version beschrieben wurden. Es ist aber zu beachten, dass dies nur für Aufgaben gemäß der neuen Spezifikation gilt und ältere Aufgaben, bei entsprechender Unterstützung, weiterhin genutzt werden können. Es wird empfohlen bei Erweiterungen die Unterstützung für ältere Versionen nicht zu deaktivieren. Zudem sind für hinzukommende Testtypen oder andere Erweiterungen, die von einem System nicht unterstützt werden, keine Änderungen notwendig, so dass lediglich bei bestimmten Veränderungen an der ProFormA-Kernspezifikation und unterstützten Testtypen Änderungen notwendig werden.

Da es bisher kein Repository gibt, die Aufgaben größtenteils in den Systemen verwaltet und bei Bedarf von dort exportiert werden, ist es zurzeit ausreichend bei einer neuen Implementierung die aktuellste Version der Spezifikation zu unterstützen. Konvertierungen sind aktuell mit dem ProFormA-Editor möglich. Aufgrund der Nutzung von XML ist es aber auch möglich, Aufgaben mit Hilfe einer XSL-Transformation automatisch zu transformieren – bei der Nutzung eines zentralen Repositories könnte dies transparent vor dem Import in ein System durchgeführt werden.

Parser für das ProFormA-Format liegen für Praktomat in Python und GATE in Java vor. Da beide Systeme unter einer Open-Source-Lizenz veröffentlicht wurden, kann auf diese Implementierungen als Ausgangspunkt zurückgegriffen werden. Um eine eigene Implementierung der ProFormA-Spezifikation testen zu können, werden unter https://github.com/ProFormA/examples Beispielexporte aus verschiedenen Systemen bereitgestellt – zukünftig auch in verschiedenen Versionen der Spezifikation.

24.3.5 Weitere Nutzungsmöglichkeiten

Das Format erlaubt grundsätzlich neben dem Austausch von Aufgaben zwischen reinen Assessment-Systemen auch die Nutzung als Beschreibung für Aufgaben zwischen einem LMS und einem Evaluationssystem (serviceorientierte Architektur oder einer Middleware): Ein LMS kann die Verwaltung sowie Darstellung der Aufgaben übernehmen und zur Evaluation die studentischen Einreichungen zusammen mit der Aufgabenbeschreibung an einen Evaluationsservice beziehungsweise eine durch Plugins erweiterbare Middleware, die mehrere Programmbewertungssysteme einbinden kann, schicken (vgl. [Str+15] sowie Kapitel 22 und 23). An der Hochschule Ostfalia wird die Verbindung vom Praktomat mit LON-CAPA schon auf diese Art realisiert (vgl. Kapitel 22). Um einen synchronen Austausch auch zwischen anderen Systemen zu ermöglichen, sind aber noch eine kollaborativ erstellte Spezifikation und die Implementierung von Schnittstellen auch in anderen Systemen erforderlich. Dies ist derzeit bereits in Planung. Insbesondere erfordert dies auch die Spezifikation eines Antwortformats.

24.4 Zusammenfassung und Ausblick

In diesem Artikel wurde eine Spezifikation *ProFormA* zum systemübergreifenden Austausch von Programmieraufgaben vorgestellt. Dabei handelt es sich nicht um eine „one-size-fits-all"-Spezifikation für vorher festgelegte Sprachen und Testtypen, sondern um ein durch XML-Namespaces erweiterbares Format. Damit bietet das Format eine gute Grundlage zur Erhöhung der Interoperabilität zwischen verschiedenen Systemen – grundsätzlich könnte das Austauschformat von allen 33 im Review von [Str+15] enthaltenen Systemen genutzt werden. Ein Export und Import von Aufgaben mit der vorgeschlagenen Spezifikation ist bereits aus mehreren Systemen möglich. Ebenso lassen sich wesentliche Elemente von Aufgaben, die in das Austauschformat exportiert wurden, in diese Systeme importieren. Ein systemübergreifender Austausch von Aufgaben unter Verwendung des vorgeschlagenen Austauschformats ist durchführbar.

Die vorgeschlagene Spezifikation wird bereits seit ungefähr vier Jahren von mehreren Systemen unterstützt. Dabei wurden viele Erfahrungen gewonnen, die auch wieder in die Spezifikation eingeflossen sind. Zudem ist in weiteren Systemen eine Unterstützung für das ProFormA-Format geplant. Ebenfalls haben sich neue Nutzungsszenarien für ProFormA ergeben: Statt lediglich Aufgaben zwischen Lehrenden auszutauschen, kann das Format auch dazu genutzt werden die Kommunikation zwischen LMS und Gradern zu vereinfachen [Str+15].

Anhand der Versionsgeschichte (vgl. Abschnitt 24.2) lässt sich gut erkennen, dass die Verfeinerung, Verbesserung und Erweiterung der Spezifikation keineswegs abgeschlossen ist. Insbesondere gibt es einige offene Punkte (z. B. parametrisierte Aufgaben, Zusammenfassen von Aufgaben und weitere Aspekte hinsichtlich Sicherheit und Laufzeitbeschränkungen von Tests) und die Spezifikation weiterer oft genutzter Testtypen, die noch in das offizielle ProFormA-Format einfließen sollen. Unabhängig von der Arbeit der Spezifikation wird für einen einfachen Austausch der Aufbau eines Repositories für Aufgaben angestrebt, so dass Aufgaben zum Beispiel direkt aus einem System heraus gesucht und importiert werden können, um so den Austausch weiter zu vereinfachen und zu fördern.

Literatur für dieses Kapitel

[AM05] Kirsti M. Ala-Mutka. „A Survey of Automated Assessment Approaches for Programming Assignments“. In: *Computer Science Education* 15.2 (2005), S. 83–102. DOI: 10.1080/08993400500150747.

[CR13] Julio C. Caiza und José María del Álamo Ramiro. „Programming assignments automatic grading: review of tools and implementations“. In: *7th International Technology, Education and Development Conference (INTED2013)*. 2013, S. 5691–5700.

[God04] Jean Godby. „What do application profiles reveal about the learning object metadata standard?“ In: *Ariadne* 41 (2004). URL: http://www.ariadne.ac.uk/issue41/godby/.

[Iha+10] Petri Ihantola u. a. „Review of Recent Systems for Automatic Assessment of Programming Assignments“. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling '10. ACM, 2010, S. 86–93. DOI: 10.1145/1930464.1930480.

[KJH16] Hieke Keuning, Johan Jeuring und Bastiaan Heeren. *Towards a Systematic Review of Automated Feedback Generation for Programming Exercises – Extended Version*. Techn. Ber. UU-CS-2016-001. Department of Information und Computing Sciences, Utrecht University, März 2016. URL: http://www.cs.uu.nl/research/techreps/repo/CS-2016/2016-001.pdf.

[Kor+08] Gerd Kortemeyer u. a. „Experiences using the open-source learning content management and assessment system LON-CAPA in intro-

ductory physics courses“. en. In: *American Journal of Physics* 76.4 (2008).

[Str+14] Sven Strickroth u. a. „Wiederverwendbarkeit von Programmieraufgaben durch Interoperabilität von Programmierlernsystemen“. In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014, S. 97–108.

[Str+15] Sven Strickroth u. a. „ProFormA: An XML-based exchange format for programming tasks“. In: *eleed* 11.1 (2015). URL: https://eleed.campussource.de/archive/11/4138.

Teil IV

Abschluss und Ausblick

25 Automatisierte Bewertung in der Programmierausbildung – Ausblick

Nils Jensen

Zusammenfassung
Das vorliegende Buch hat Einblicke in aktuelle Konzepte und Tools zum halb- oder vollautomatisierten Assessment in der hochschulspezifischen Programmierausbildung gegeben. In diesem Kapitel soll ausgehend von den erreichten, erprobten Ansätzen gezeigt werden, welche Herausforderungen in der praktischen Anwendung liegen und welche neuen Entwicklungs- und Forschungsansätze sich daraus ergeben. Die folgenden Ausführungen sind dabei weder umfassend noch erschöpfend. Gleichwohl sollen sie der Leserschaft Inspiration und Impuls sein.

25.1 Zielsetzung der hochschulspezifischen Programmierausbildung

An einer Hochschule sollen die Grundlagenkenntnisse in einer Programmiersprache, meistens Java, vermittelt werden. Vielfach schließt das Vorlesungseinheiten und Übungen zu Debugging, Korrektheit und Stil ein. Es ist hierbei nicht Aufgabe der Hochschulen, eine abgeschlossene Ausbildung in der Programmierung zu bieten. In der Praxis müssen daher die Kenntnisse durch „Training-on-the-job“ und maßgeschneiderte Schulungen vertieft werden. Nur wenige Informatikerinnen und Informatiker werden im Beruf langfristig als Programmierer eingesetzt werden.

Gleichwohl sollen viele Studierende durch den Erwerb der Grundlagenkenntnisse die Kompetenz erlangen, Probleme rational erfassen, in Teilaufgaben zerlegen und algorithmisch verallgemeinert lösen zu können. Als Computational Thinking bezeichnet man hierbei das Vermögen, realweltliche Prozesse konsequent

algorithmisch zu erfassen (vgl. [Guz08]). Der Auftrag der Programmierausbildung an Hochschulen ist demnach, im engeren Sinne, eine Programmiersprache als Werkzeug begreifbar zu machen, und die Programmierausbildung im weiteren Sinne als Vehikel zu nehmen, um Computational Thinking zu befördern.

25.2 Didaktik

In einem konstruktivistischen Ansatz wird man versuchen, den Studierenden vielfältige Möglichkeit zur individuellen Schaffung eigener Lernerlebnisse zu ermöglichen. Die Information des zu Vermittelnden steht damit als notwendiges, aber für viele Lernende nicht hinreichendes Element im Zentrum. In dieser Hinsicht sind die folgend genannten Tools und Innovationen eine Möglichkeit, den Studierenden einen erweiterten, selbstbestimmten und vielfältigen Raum zur Selbsterprobung und Selbstreflexion zu geben, in dem sie Mut fassen und Fortschritte erzielen können.

Die Begleitforschung zum Einsatz der genannten Tools steht in der ProFormA-Gruppe im eCULT-Team am Anfang. Es müssen beispielsweise Untersuchungen zur gleichbleibenden oder höheren Wirksamkeit beim Lernerfolg ab 2017, d. h. in der zweiten Förderphase des Projekts eCULT, durchgeführt werden. Das Messen eines Lernerfolgs im Vergleich zu einer Kontrollgruppe oder zu vorherigen Kohorten ist schwierig, weil in der Lehrpraxis die Rahmenbedingungen unvorhergesehen variieren können. Es ist deshalb notwendig, mit Didaktikern gemeinsam formative und summative Studien zu konzipieren.

Abseits dessen sollten folgende Punkte konkret im Vordergrund stehen:

- Review der Online-Aufgaben, insbesondere um Diskrepanzen zwischen automatisierter Beurteilung und Aufgabentexten aufzudecken.
- Kontrollierte Usability-Studien, in denen der Umgang mit dem System beobachtet und Defizite im System identifiziert werden, z. B. unverständliche Bedienung oder nicht nachvollziehbares Systemverhalten.
- Qualitative Beurteilungen durch Studierende und Lehrende in mehreren Semestern, z. B. über ein Online-Feedbacksystem.

Die Verbindung zwischen Learning Analytics und didaktischer Innovation ist seit 2011 im Projekt eCULT erkannt worden und stellt einen immer wichtiger werdenden Ansatz zur empirisch basierten Lehr- und Lerninnovation dar. Die Notwendigkeit der mentoriellen und tutoriellen Betreuung scheint dadurch nicht geschmälert, sondern vielmehr bestätigt und um hochentwickelte Werkzeuge zur Daten- und Nutzungsanalyse erweitert worden zu sein (vgl. [Clo13]).

Es sollte mithilfe der Learning Analytics untersucht werden, ob die Tools geeignet sind, sehr schwachen Studierenden die nötige Hilfe zu geben, basale Konzepte des Programmierens zu verinnerlichen. Nach unserer Erfahrung benötigt insbesondere die Gruppe der schwächeren Studierenden, jenseits des Wiederholens des Stoffes nach erfolgloser Prüfung, weitere Betreuung. Wo hierbei die Grenzlinie zwischen automatisierter und manueller Intervention verlaufen kann, ist nach unserer Meinung unklar und bedarf weiterer Analysen. Eine hierbei möglicherweise nützliche Theorie des Lernens mathematischer Inhalte stellt die APOS-Theorie von [DM02] dar (vgl. Kapitel 8). Diese Theorie könnte auf die Programmierausbildung adaptiert werden. Gelänge dies, so wäre gezeigt, dass ähnliche mentale Prozesse erforderlich sind, um jeweils Mathematik oder das Programmieren zu erlernen. Der Vorteil der APOS-Theorie ist folgender: In ihr können typische Schwierigkeiten vorhergesagt, analysiert und zerlegt werden. Darauf abgestimmt kann Lernmaterial entwickelt und ein Vorgehensmodell in Anlehnung an [MP04] befolgt werden. Unter Zuhilfenahme der Learning Analytics ergäbe sich somit ein empirisch fundierter, wissenschaftlich nachvollziehbarer, Zyklus zur Optimierung der Lehre.

25.3 Operationalisierung und Rollen

Die Lehrenden wählen Inhalt und Ziele der Lehrveranstaltung, anhand der Ziele die zu ergreifenden Tätigkeiten und, dazu passend, die Tests. Die Lehrenden helfen den Studierenden, die zu erwerbenden Kompetenzen einzuüben. Die Lehrassistenz und Tutorenschaft unterstützt sie in den operativen Aspekten einer Lehrveranstaltung durch Aus- und Aufarbeitung von Materialien, Kontaktzeit mit den Studierenden, sowie Übungsaufgaben, Tests und Prüfungen. Entsprechende Collaboration Tools legen bisher ihren Fokus auf die Interaktion zwischen Studierenden und Lehrenden, nicht aber auf die Interaktion zwischen arbeitsteilig wirkenden Lehrenden. Terminierung, Qualitätsprüfung und Arbeitsfeedback finden bisher informell statt und können in Learning Management Systemen (LMS), zum Beispiel Moodle, nicht standardisiert und formal erfasst werden. Oft werden mehrere Tools gleichzeitig eingesetzt, die Verzahnung und die gemeinsame Datenbasis schwinden dadurch. Das gilt insbesondere beim gleichzeitigen Einsatz mehrerer LMS um deren Vorteile zu vereinen: Beispielsweise bieten nicht alle LMS ein gemeinsames Single-Sign-On-System.

25.4 Tools

Aspekte des Collaborative und Blended Learning können geeignet sein, erfolgreich Lehre zu befördern, indem sie beispielsweise zu einer stärkeren Aktivierung der Studierenden führen und ihren individuellen Bedarfen näher kommen. Insbesondere kann der Vorteil darin bestehen, jederzeit und jederorts mit den Lehrenden in Kontakt zu treten, die beste Zeit zum Lernen selbst zu bestimmen, ortsunabhängig auf die Lehrmaterialien zuzugreifen und gegebenenfalls alternative grafische Schnittstellen nutzen zu können. Dieser Punkt dient auch der Barrierefreiheit. Weiterhin erlaubt das Collaborative und Blended Learning den Studierenden zwecks besserer Aktivierung miteinander zu beraten (Peer-teaching PT, Peer-instruction PI) und technische Setups zu bedienen, in denen insbesondere vermittels des Problem-Based Learning (PBL) praxisnähere Lehr-/Lernszenarien angeboten werden können.

Hierfür bieten die diskutierten Konzepte und Tools vielfache Anknüpfungspunkte. Die Systeme sind webbasiert und daher prinzipiell jederzeit und von überallher erreichbar. Integrierte Foren- und Chatfunktionen können zielgerichtet von Lehrenden und Studierenden eingesetzt werden. Mit beispielsweise Peer Reviews sollen Studierende in die Lage versetzt werden, untereinander ihre Lösungen konstruktiv zu kritisieren und ihre Fähigkeit zu verbessern, fremden Quelltext zu analysieren. Dies kann ein angemessener Weg sein, eine Programmiersprache oder eine Programmiertechnik zu erlernen (s. [SM12]). Das Potenzial erscheint nach unserer Ansicht nicht ausgeschöpft.

Mehr noch als bisher müssen die Tools als Komponenten in komplexen Lehrszenarien konfigurier- und einsetzbar sein. Möglich ist zum Beispiel die kombinierte Tool-Bereitstellung in einer PBL-Umgebung zum Messen und Simulieren, so dass die Ergebnisse des Programms oder der Ablauf des Programms mittels Visualisierungskomponenten (Software Visualization, Information & Scientific Visualization) in die Umgebung integriert dargestellt werden. Obwohl die Tools webbasiert sind, sind sie untereinander noch eher schlecht verzahnt. Ein erster Ansatz können Web Mashups sein, in denen die entsprechenden Tools über eine zentrale Seite gesteuert und in eingebetteten IFrames in HTML laufen.

Mit Tools fallen zum Teil erstmalig digitale Daten zum individuellen Lernprozess oder Prüfungsverlauf von Studierenden an. Somit sind rechtliche Fragen zu klären. So muss die Prüfungsordnung den Einsatz von technischen Hilfsmitteln in der Lehre zwar nicht explizit erlauben, dennoch kommt es in der Praxis zu Unsicherheiten, ob E-Prüfungen erlaubt sind oder in der Prüfungsordnung gesondert zugelassen sein müssen. Die Prüfungsakten müssen dokumentiert und darin zweifelsfrei der Verlauf der Prüfung einsehbar sein. Die Begründung der Bewer-

tung muss aus den Daten hergeleitet werden können. Die Aufbewahrung der entsprechenden Akten erfolgt mehrere Jahre lang. Es ist nicht ersichtlich, weswegen diese Daten nur analog und auf Papier erhoben und aufbewahrt werden können. Es ist möglich, dass hierzu entsprechende Tools erst nach einem Audit zertizifiert und zugelassen werden müssten. Eine entsprechende Zertifizierungsstelle gibt es unseres Wissens in Deutschland nicht.

25.5 Community

Tools können didaktisch sinnvoll in Lehr-/Lernszenarien der Programmierausbildung eingesetzt werden. Einige Lehrende scheuen allerdings aufgrund mangelnder positiver Erfahrung den Einsatz und verzichten auf den Erfolg.

Den ersten Schritt in Richtung automatisierter Programmbewertung machen Lehrende in der Regel, um Zeit bei der Auswertung von Übungsaufgaben zu sparen und mehr Zeit für den qualitativen Diskurs mit den Studierenden über Inhalt und Systematik, nicht Form(alien), zu gewinnen. Daraus erwächst der Wunsch, auf bestehendes Material und bestehendes Wissen effektiv zuzugreifen und es in eigenen Lehrveranstaltungen individualisiert einzusetzen. Daher sind in Zukunft Netzwerke von Expertinnen und Experten gefragt, in denen in Fachzirkeln zur Programmierausbildung aktuelle Fragestellungen bekannt gemacht und diskutiert werden können (vgl. [ABP13]).

Erst durch den Austausch von Tools und Content vermittels ihrer kompatiblen Schnittstellen, Austauschformate und Bedienungsstandards einerseits und einer vitalen offenen Community andererseits können „Economies of Scale“ erreicht werden. Nutznießer und Betroffene sind dabei zentral die Studierenden, deren Feedback, Meinungen und Wünsche man einholen und in diesen Prozess einfließen lassen muss.

25.6 Aufgabenpools und Qualitätssicherung

Tools können helfen, bisher genutzte Übungen und Assessments wiederzuverwenden und ähnlich einer Tauschbörse interessierten Lehrenden zur Verfügung zu stellen. In dem an der Hochschule Ostfalia eingesetzten System LON-CAPA verfolgen wir dieses Paradigma („Tausch statt Bau“), siehe [Kor+03]. Die wichtigsten Herausforderungen liegen dabei in der Qualitätssicherung von heterogen zusammengestellten Online-Aufgaben und -Ressourcen, die auch nach vielen Jahren noch problemlos funktionieren müssen. Ein weiteres Problem betrifft die Er-

reichbarkeit und Auffindbarkeit dieser Ressourcen auch durch Nutzer jenseits der Grenze des eigenen Systems, zum Beispiel in LON-CAPA ([KDP14]), aus beispielsweise Moodle und Stud.IP heraus. Dies kann durch ein gemeinsames oder durch mehrere vernetzte, verteilte Repositories erfolgen. Die Anforderungen an solche Repositories umfassen:

- Versionierung: Eingestelle Ressourcen müssen versioniert und historisiert werden.

- Lizenzmodelle: Die Ressourcen müssen beim Einstellen mindestens mit einer Lizenze annotiert werden, z. B. Creative Commons, Public Domain, BSD oder GPL.

- Export in andere Formate und Pools: Die Ressourcen müssen in verschiedene LMS oder Grader importiert werden können. Hierzu dient die Unterstützung des Austauschformats (vgl. Kapitel 24).

- Probezugänge und Vorschau: Eine begrenzte Untermenge der im Pool angelagerten Ressourcen sollte öffentlich zugänglich sein, ohne den Schutz der anderen Teile zu gefährden. Registrierten, autorisierten Nutzenden sollte es möglich sein, die Ressourcen vorab zu betrachten und testweise zu bedienen.

- Zugang/Suche: Es sollten die relevanten Metadatenformate unterstützt werden.

- Technik: Die Bereitstellung der Tools erfordert mehrere Server, die beispielsweise in einem Peer-to-Peer-System vernetzt sind. Zum Beispiel bietet LON-CAPA diese Funktionalität seit längerem.

- Datenschutz und Sicherheit: Sowohl die Nutzerdaten (Verhalten, Passworte) als auch die Ressourcen müssen durch hohe IT-Sicherheitsstandards nach z. B. den Empfehlungen des Bundesamtes für Sicherheit in der Informationstechnik (BSI) abgesichert sein.

- Host/Provider: Die Bereitstellung und Administration der Server erfordert finanzielle Zuwendungen, die bisher nicht von den Hochschulen eingeplant sind.

- Bewertung/Feedback: Ressourcen sollten bewertet und empfohlen werden können.

- Zugangskontrolle: Nur registrierten, nachgewiesenen Lehrenden ist der Zugang zu gestatten. Authentifizierung, Autorisierung und gegebenenfalls Audits im Rahmen der Datenschutzgesetze sind erforderlich.

- Mehrsprachigkeit: Die Ressourcen sollten auf Deutsch und in andere Sprachen übersetzt sein. Es sollte die Möglichkeit bestehen, dass Übersetzungen von anderen als den Autorinnen und Autoren der Ressourcen eingepflegt und qualitätsgesichert werden. Insbesondere die Qualität und Mehrsprachigkeit der Aufgabentexte erfordert mehrfache Peer Reviews, mitunter durch die Studierenden.

- Bepunktung: Die Art der Bepunktung sollte den Lehrenden, nicht den Autorinnen und Autoren, obliegen, d. h. im LMS vorgenommen werden. Dennoch kann eine Bepunktungsempfehlung Teil der Ressource sein.

25.7 E-Assessments und Feedback

E-Assessments stellen eine elektronisch unterstützte Erhebung der Lernleistung der Studierenden dar. Hierbei kommen wie auch im nichttechnischen Fall formative (begleitende) und summative (abschließende) Tests zum Einsatz. Den formativen Assessments liegt im Gegensatz zu den Prüfungen nicht das Erfordernis zugrunde, eine Wertung zu vergeben, sondern den Lernfortschritt abseits einer quantitativen Beurteilung zu kennen, um wiederum selbst oder gemeinsam daraus zu lernen.

25.7.1 Die rechtliche Seite

Die niedersächsische Lehrverpflichtungsordnung (LVVO) sieht streng betrachtet nur den Fall der Präsenzlehre, d. h., die Kontaktzeit mit den Studierenden im selben Raum vor. Genauso sieht sie streng betrachtet nur die Prüfung ohne Zuhilfenahme technischer Hilfsmittel vor. Diese Rahmenbedingungen sind nicht mehr zeitgemäß und verbauen vielfach den effektiven, angemessenen Einsatz von Tools in Lehre und Prüfungswesen. Im Projekt eCULT etablieren wir seit 2011 den Einsatz von Vorlesungsaufzeichnungen, Response-Systemen, computerbasierten Lernräumen und E-Assessments sowie E-Prüfungen. Es ist zu erwarten, dass mehr als bisher der Gesetzgeber und die Fakultäten klar benennen müssen, welcher Einsatz von Tools rechtlich abgesichert ist. Auch ist hierbei wichtig zu benennen, welche Funktionen in Tools ohne Einwilligung der Nutzer laut Datenschutzgesetz

verboten sind. Hierbei bewegen sich zurzeit Tool-Entwickler und -Nutzer in einer rechtlichen Grauzone, zum Beispiel wenn die Aktivitäten eines Studierenden bei der Bearbeitung protokolliert und an die Lehrperson zur, durchaus wohlmeinenden, Auswertung gesendet werden.

25.7.2 Feedback

Eine Hemmschwelle und Ablehnung erfahren Tools, wenn Studierende durch deren Einsatz frustriert werden und studentische Anforderungen nicht oder unzureichend in die Softwareentwicklung einfließen. Insbesondere soll die GUI einfach benutzbar, widerspruchs- und fehlerfrei sein. Der Content soll nach und nach an die Bedienung des Tools heranführen. Die Ergebnisse im Tool müssen zu den manuellen Bewertungen und sonstigen Prozessen in der Lehrveranstaltung passen. Zum Beispiel ist es problematisch, zueinander widersprüchliche Tools einzusetzen, die die Studierenden nicht einwandfrei nachvollziehen lassen, ob ihre Einsendungen/Antworten richtig oder falsch sind. Die Unflexibilität der Tools oder auch „verletzende Härte“ ihrer Bewertungsergebnisse (objektiv, maschinell überprüfbar) kann dazu führen, dass sich Studierende mit „halbrichtigen“ Lösungen zurückgesetzt und demotiviert fühlen. Hier ist auf die richtige Begleitung und Erläuterung zu achten und auch eine Einzelfallprüfung nicht abzulehnen. Kryptische Feedbacks führen dazu, dass die Bedienung der Tools als Eingabesysteme zu einem Trial-and-Error-System transmutiert, dem man nur durch den richtigen „Zauberspruch“ die Akzeptanz der eingereichten Lösung zu entlocken versucht. Eine Erläuterung der wichtigsten Feedbackmeldungen ist erforderlich, um dem vorzubeugen. Es muss mehrmalige Versuche geben und gleichzeitig eine zu schnelle Wiedereinreichung durch Verzögerungen verhindert werden. Zu Anfang sollten Fehlermeldungen bewusst gemeinsam provoziert werden, um die Reaktionen des Systems einschätzen zu können. Kritisch sehe ich Versuche, dem System natürlichsprachlich wirkende Feedbacks zu entlocken (die zudem didaktisch Sinn ergeben müssen). Wir sind mindestens ein Jahrzehnt von einer solchen Entwicklung entfernt, aktuelle Versuche mit Chatbots machen dies noch einmal deutlich (vgl. [Gra16]).

Ein Feedbackformat muss erweiterbar sein, von Prüfelementen („Checkern“) erzeugt und vom LMS weiterverarbeitet werden können. Es enthält mindestens folgende Felder:

- ID der einreichenden Person: Die Matrikel-Nr. und gegebenenfalls Institution muss bei der Einreichung gespeichert werden und darf durch die Clients nicht verfälscht werden.

- Zeitstempel: Die Serverzeit der Einreichung ist festzuhalten.
- Versuch: Der aktuelle Versuch der Einreichung muss gespeichert und hochgezählt werden. Auch hier darf die Nummer nicht durch den Client verändert werden können. Versuche, die durch klare Fehlbedienung des Clients entstanden sind, sollten nicht gezählt werden.
- Maximale Anzahl Versuche (Max. Versuch): Nach Überschreiten dieses durch die Lehrenden gesetzten Feldes (d. h. Anzahl Versuche > Max. Versuch) ist die Einreichung abzuweisen.
- Einreichung: Dieses Feld enthält alle zur Einreichung erforderlichen Daten.
- Pass/Fail: Der Grader erzeugt ein Flag, das vom LMS als Pass/Fail gespeichert wird. Eine erfolgreiche Einreichung verhindert die Eingabe weiterer Versuche.
- Punktzahl (Score): Der Score wird auf Basis des Rohergebnisses des Graders vom LMS ermittelt und angezeigt.
- Mindestpunktzahl (Min. Score): Dieser Score wird von den Lehrenden gesetzt. Es gilt, dass Pass genau dann wahr ist, wenn Score $\geq$ Min. Score gilt.
- Höchstpunktzahl (Max. Score): Der Max. Score dient den Studierenden als Richtschnur, wieweit ihre Lösung vom Optimum entfernt ist. Auch der Max. Score wird auf Basis von Empfehlungen, die mit der Aufgabenressource verbunden sind, im LMS gesetzt.
- Ausführungsreport (formatiert): Hier erhält jede Studentin und jeder Student ihren bzw. seinen Nachweis der Einreichung und nachgewiesenen Prüfung der Einreichung durch das System.
- Grund der Ablehnung (nur bei Fail): Es muss klar erkennbar sein und somit als Fehlermeldung oder Empfehlungstext vorliegen, an welcher Stelle bei der Prüfung der Einreichung welcher Fehler vorliegt.
- Lösungshinweise (Hints – nur wenn Score < Max. Score): Die Hints sind auf Basis von Heuristiken von den Autorinnen und Autoren gesetzte Hilfstexte, die erläutern, welche Verbesserungen bei der Einreichung möglich sind. Es sollten auch Hints ausgegeben werden können, wenn die Einreichung akzeptiert worden ist.

Außer dem Zeitgewinn durch den Einsatz erprobter Ressourcen und der Teil- oder Vollautomatisierung des Bewertungsvorgangs (was nicht die Notenvergabe einschließt, diese obliegt den Prüfenden nach Einsicht der Ergebnisse) lässt sich beobachten, dass Tools einen Qualitätsgewinn in der Lehre nach sich ziehen können, denn nunmehr erhält der oder die Lehrende zeitnah Ergebnisse aus den automatisch bewerteten Programmieraufgaben und kann daran seine Lehre ausrichten, zum Beispiel Lehrstoff wiederholen oder Teile überspringen. Studierende erhalten schneller als bisher eine umfangreiche Auswertung ihrer Leistung und Hinweise zur Verbesserung. Die Optimierungen der formativen Feedbacks aus didaktischer Sicht und der datenschutzrechtlich abgesicherten und sabotage- und spionagegeschützten summativen Feedbacks (E-Prüfungen) sind nach unserem Ermessen ein offenes Forschungsgebiet.

Als weitere Forschungsperspektive können adaptive Feedbacks gelten, in denen das System intelligent entscheidet, in welcher Form es Feedback und Leseempfehlungen vergibt. Diese als Intelligent Tutoring Systems (ITS) bekannten Systeme sind kommerziell erhältlich. Sie fristen aus unserer Sicht eher ein Nischendasein, weil ihr Einsatz kostenpflichtig ist, Datenschutzbedenken bei Cloud-basierten ITS existieren und ITS das „klassische“ Zusammenspiel zwischen Lehrenden und Studierenden neu definieren, siehe [Bak16].

25.8 Plagiarismus

Man unterscheidet hier Plagiarismus in mindestens den folgenden Ausprägungen:

1. Plagiate von außerhalb des Studierendenkreises,
2. Plagiate einer anderen Lösung desselben Semesters,
3. Plagiate einer anderen Lösung aus vorhergehenden Semestern.

Plagiate umfassen neben wortwörtlichen Übernahmen auch strukturelle Plagiate, eine Übersicht und Diskussion geben [WWW06]. Dem weiteren Problem des Ghostwritings kann nur durch prüfungs- oder prüfungsähnliche Situationen entgegen gewirkt werden. Dem ersten Punkt kann durch geeignete Konzeption neuer Aufgaben und Ressourcen Abhilfe geschaffen werden. Auch kann unter Berücksichtigung der Datenschutz- und Urheberrechtsgesetzgebung Plagiatserkennungssoftware eingesetzt werden. Rechtlich gangbar ist allem Anschein nach nur eine Ähnlichkeitsprüfung zwischen studentischen Einreichungen an genau derselben Hochschule. Sie sollte hinreichend sein, da auch bei Plagiaten von anderen Hochschulen die Anzahl der „Incidents“ an derselben Hochschule hoch und damit

erkennbar sein müsste. Doch diese Prüfungen können erst im Nachhinein eingesetzt werden. Wünschenswerter sind konstruktive Verfahren, die ein Plagiat von vornherein hinreichend erschweren.

Dem Plagiarismus nach Punkt zwei entgegenwirken kann zum einen eine gemeinsame Eingabefrist für alle. Die Abgabe einer fremden Lösung wäre einerseits aufgrund des hohen Grads der Entdeckungswahrscheinlichkeit risikoreich, und zum anderen wäre sie keine Garantie auf ein korrektes Lösen der Aufgabe, da alle zum selben Zeitpunkt ihr Feedback erhielten. Zum anderen kann die Randomisierung von Aufgaben oder deren ständiger Austausch Abhilfe schaffen. Sie ist auch bei Punkt drei anwendbar.

Die Randomisierung von Programmieraufgaben ist diffizil. Einen möglichen Ansatz stellt die Parametrierung dar. Eine Ausprägung der Parametrierung hinterlegt bestimmte Felder mit variablen Ausdrücken, zum Beispiel „Schreiben Sie ein Programm, das zwei Zahlen des Datentyps <DATENTYP> <multipliziert | dividiert>“. Nehmen wir an, dass als Datentyp Float, Double und Integer hinterlegt sind, so ergeben sich 6 mögliche Ausprägungen. Entweder muss für jede dieser Ausprägungen eine andere Musterlösung hinterlegt werden, oder es existiert dieselbe Lösung, die mittels eines Präprozessors in die jeweilige Ausprägung gebracht und dann verwendet wird. Die Schwierigkeit, solche Varianten zu debuggen, ist erheblich. Es wäre hilfreich, eine Form der Randomisierung zu entwickeln, deren „Komplexität“ konstant oder logarithmisch mit der gewünschten Anzahl der Möglichkeiten wüchse. Eine andere Herausforderung ergibt sich aus der Wahrscheinlichkeit, dass mit der Zeit eine Sammlung an studentischen Lösungen entsteht, die außerhalb des Systems ebenfalls in einer Art Tauschbörse genutzt werden könnte.

Ein probabilistischer Ansatz dürfte ein durch Learning Analytics/KI ermitteltes Wahrscheinlichkeitsmodell sein, das die kognitiven Fortschritte jeder Studentin und jedes Studenten verzeichnen würde. Das Modell könnte nicht nur automatisch darauf hinweisen, wie wahrscheinlich es ist, dass die Einreichung von der Studentin oder dem Studenten selbst stammt, also zur Plagiatserkennung herangezogen werden. Es könnte darüber hinaus didaktisch sinnvoll dazu einsetzbar sein, Defizite und daraus abgeleitet erforderliche Lernwege zu prognostizieren. Hier schließt sich wieder der Kreis zu den ITS und zur eingangs zitierten APOS-Theorie, die eine Grundlage für ein Modell sein könnte. Auch andere technische Modelle, zum Beispiel künstliche neuronale Netze (KNN), können verwendet werden.

25.9 Fazit

Die Entwicklung von Tools in der akademischen Programmierausbildung hat Fortschritte gemacht und mehr Verbreitung an der Hochschule gefunden. Dennoch stehen wir erst am Anfang. Es ist zu hoffen, dass die Resultate verstetigt werden können und in Zukunft auf breite Akzeptanz bei Lehrenden und Studierenden treffen werden.

Literatur für dieses Kapitel

[ABP13] *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. (http://ceur-ws.org/Vol-1067/). 2013.

[Bak16] Ryan S. Baker. „Stupid Tutoring Systems, Intelligent Humans“. In: *International Journal of Artificial Intelligence in Education* 26.2 (2016), S. 600–614. DOI: 10.1007/s40593-016-0105-0.

[Clo13] Doug Clow. „An overview of learning analytics“. In: *Teaching in Higher Education* 18.6 (Aug. 2013), S. 683–695.

[DM02] Ed Dubinsky und Michael A. McDonald. „The Teaching and Learning of Mathematics at University Level: An ICMI Study“. In: Hrsg. von Derek Holton u. a. Dordrecht: Springer Netherlands, 2002. Kap. APOS: A Constructivist Theory of Learning in Undergraduate Mathematics Education Research, S. 275–282. DOI: 10.1007/0-306-47231-7_25.

[Gra16] Bernd Graff. „Rassistischer Chat-Roboter: Mit falschen Werten bombardiert“. In: *Süddeutsche Zeitung* (2016). URL: http://www.sueddeutsche.de/digital/microsoft-pro\-gramm-tay-rassistischer-chat-roboter-mit-falschen-werten-bombardiert-1.2928421 (besucht am 13. 05. 2016).

[Guz08] Mark Guzdial. „Education: Paving the Way for Computational Thinking“. In: *Commun. ACM* 51.8 (Aug. 2008), S. 25–27. DOI: 10.1145/1378704.1378713.

[KDP14] Gerd Kortemeyer, Stefan Dröschler und David E. Pritchard. „Harvesting latent and usage-based metadata in a course management system to enrich the underlying educational digital library – A case

study“. In: *Int. J. on Digital Libraries* 14.1-2 (2014), S. 1–15. DOI: 10.1007/s00799-013-0107-6.

[Kor+03] Gerd Kortemeyer u. a. „The Learning Online Network with Computer-Assisted Personalized Approach (LON-CAPA)“. In: *PGLDB 2003, Proceedings of the I PGL Database Research Conference, Rio de Janeiro, Brazil, April 10-11, 2003*. Hrsg. von Rubens N. Melo, Diva de Souza e Silva und Sean W. M. Siqueira. Bd. 70. CEUR Workshop Proceedings. CEUR-WS.org, 2003.

[MP04] Joan Middendorf und David Pace. „Decoding the disciplines: A model for helping students learn disciplinary ways of thinking“. In: *New directions for teaching and learning* 2004.98 (2004), S. 1–12.

[SM12] Harald Søndergaard und Raoul A. Mulder. „Collaborative learning through formative peer review: pedagogy, programs and potential“. In: *Computer Science Education* 22.4 (2012), S. 343–367. DOI: 10.1080/08993408.2012.728041.

[WWW06] Debora Weber-Wullf und Gabriele Wohnsdorf. „Strategien der Plagiatsbekämpfung“. In: *Information: Wissenschaft & Praxis* 57.2 (2006), S. 90–98.

Teil V

Verzeichnis der Autorinnen und Autoren

Oliver J. Bott (oliver.bott@hs-hannover.de) studierte Medizinische Informatik an der Universität Hildesheim. 2001 promovierte er am Institut für Medizinische Informatik der Technischen Universität Braunschweig zum Dr.-Ing. Nach Tätigkeiten in Forschung und Lehre sowie Privatwirtschaft 2007 dann Ruf als Professor für Medizinische Informatik an die Hochschule Hannover (HsH). Seine Arbeits- und Forschungsgebiete umfassen Medizinische Informationssysteme und E-Learning-Technologien, ein Schwerpunkt ist die automatisierte Programmbewertung in der Informatikausbildung. (Kapitel 4)

Joachim Breitner (mail@joachim-breitner.de) studierte Mathematik und Informatik auf Diplom in Karlsruhe und promovierte 2016 dort über Compilerbau und formale Verifikation mit Fokus auf funktionale Programmierung und Haskell. Während dieser Zeit betreute er den in Karlsruhe entwickelten Grader „Praktomat“. Seit Sommer 2016 arbeitet er als Post-Doc an der University of Pennsylvania in Philadelphia. (Kapitel 10)

Alexander Dallmann studierte bis 2012 Informatik an der Universität Würzburg. Schon während seiner Tätigkeit als studentische Hilfskraft arbeitete er an einem webbasierten E-Learning Tool zur automatischen Bewertung von Programmieraufgaben (PABS), das er bis 2014 aktiv entwickelte und welches auch in den von ihm betreuten Programmierpraktika sowie verschiedenen Vorlesungen eingesetzt wird. Heute arbeitet er an seiner Promotion im Bereich „Deep Learning“. (Kapitel 15)

Peter Fricke (peter.fricke@hs-hannover.de) schloss sein Informatikstudium 2014 an der Hochschule Hannover ab. Seit 2012, noch als studentische Hilfskraft, später als wissenschaftlicher Mitarbeiter im E-Learning Center der Hochschule Hannover, widmet er sich verstärkt der automatisierten Programmbewertung. Weiterhin beschäftigt er sich mit Lernmanagementsystemen (insb. Moodle, LON-CAPA) und die Anbindung dieser Systeme mit Autobewertungstools. (Kapitel 18 und 23)

Robert Garmann studierte und promovierte am Fachbereich Informatik der Universität Dortmund. Nach verschiedenen Softwareprojekten in der Privatwirtschaft wechselte er 2006 als Professor an die Fachhochschule Stralsund und 2009 an die Hochschule Hannover. Er lehrt zu allen Aspekten der Programmierung von Softwaresystemen. Forschend widmet er sich schwerpunktmäßig lehrbezogenen Fragestellungen der Softwareentwicklung. Insbesondere befasst er sich mit der automatischen Bewertung von Computerprogrammen. (Kapitel 3 und 11)

Helmar Gust (helmar.gust@uos.de) studierte Mathematik und Physik an den Universitäten Marburg und Göttingen. 1988 promovierte er am Fachbereich Informatik der Technischen Universität Berlin zum Dr.-Ing. und habilitierte sich 1994 in Computerlinguistik und Künstlicher Intelligenz an der Universität Osnabrück. Von 1993 bis 2001 war er Mitglied des Instituts für Semantische Informationsverarbeitung und seit 2001 ist er Mitglied des Instituts für Kognitionswissenschaft an der Universität Osnabrück. Seine Arbeits- und Forschungsgebiete umfassen logische Methoden in der Künstlichen Intelligenz, Logisches Programmieren, Constraints und analoges Schließen. (Kapitel 5 und 14)

Martin Hecker ist seit dem Abschluss seines Informatikstudiums an der Universität Münster wissenschaftlicher Mitarbeiter am Karlsruher Institut für Technologie (KIT). Dort beschäftigt er sich mit der statischen Sicherheitsanalyse verteilter und nebenläufiger Programme. Er ist seit 2011 Mitentwickler von Praktomat, einem System zur Qualitätskontrolle in Programmierpraktika. In der Lehre betreut er Veranstaltungen des Lehrstuhls für Programmierparadigmen sowie den Einsatz von Praktomat am KIT. (Kapitel 10)

Felix Heine (felix.heine@hs-hannover.de) ist seit 2010 Professor für Datenbanken und Informationssysteme an der Hochschule Hannover. Er studierte Informatik an der Universität des Saarlandes und an der Universität-GH Paderborn. 2006 promovierte er an der Universität Paderborn zum Dr. rer. nat. Zwischenzeitlich arbeitete er in verschiedenen Stationen als IT-Berater in der Privatwirtschaft. Prof. Heine forscht u.a. im Bereich der automatisierten Bewertung von SQL-Aufgaben und der Integration von Bewertern in Lernmanagementsysteme. (Kapitel 4 und 12)

Britta Herres studiert im Masterstudiengang Informatik an der Hochschule Trier und ist dort seit 2012 wissenschaftliche Mitarbeiterin im Fachbereich Informatik. Neben unterstützenden Tätigkeiten in der Lehre beschäftigt sie sich mit der Entwicklung von Systemen zur automatischen Bewertung von Software und zur Stundenplanung. (Kapitel 7 und 16)

Lukas Iffländer (lukas.ifflaender@uni-wuerzburg.de) schloss sein Informatikstudium 2016 an der Universität Würzburg ab. Derzeit ist er wissenschaftlicher Mitarbeiter und Doktorand am Lehrstuhl für Software Engineering an der Universität Würzburg. Seine Forschung fokussiert sich auf Verlässlichkeit und Widerstandsfähigkeit cyberphysischer Systeme. Sein Interesse an der Entwicklung automatischer Bewertungssysteme entspringt seiner langjährigen Betreuung des Programmierpraktikums und des Ziels der Aufwandsreduzierung. (Kapitel 15)

Marianus Ifland (ifland@informatik.uni-wuerzburg.de) studierte bis 2008 Informatik an der Universität Würzburg. Anschließend arbeitete er als wissenschaftlicher Mitarbeiter, wobei er sich hauptsächlich mit fallbasiertem Training beschäftigte und maßgeblich an der Entwicklung des „CaseTrain“-System beteiligt war. Er promovierte 2014 zur Feedbackgenerierung in E-Learning-Systemen. Seit 2014 ist er an der Universität Würzburg in der Lehre für das Programmierparaktikum für Informatiker verantwortlich. (Kapitel 6)

Nils Jensen ist seit 2010 Professor für Medieninformatik an der Ostfalia Hochschule. Er arbeitete unter anderem am Forschungszentrum L3S/RRZN und promovierte 2007 an der Leibniz Universität Hannover zum Doktor-Ingenieur. Seine Gebiete umfassen die Softwareentwicklung, die wissenschaftliche Visualisierung und das Autonomic Computing. Nils Jensen ist Mitglied der Gesellschaft für Informatik. (Kapitel 25)

Carsten Kleiner (carsten.kleiner@hs-hannover.de) studierte Mathematik an der Universität Hannover sowie Computer Science an der Purdue University. Vor und nach seiner Promotion (2003) zum Dr. rer. nat. an der Universität Hannover war er als Softwareentwickler, Technischer Berater und Projektleiter in Wirtschaftsunternehmen in Köln und Hamburg tätig. Seit Ende 2004 ist er Professor für Sichere Informationssysteme an der Hochschule Hannover. Seine Arbeits- und Forschungsgebiete umfassen neben Grundlagen und Anwendungen der Datenbank- und Informationssysteme die automatisierte Bewertung von Studierendenabgaben, insbesondere im Bereich der Datenbanksysteme sowie der theoretischen Informatik. (Kapitel 4 und 12)

Elmar Ludwig (elmar.ludwig@uni-osnabrueck.de) studierte Mathematik an der Universität Osnabrück und promovierte dort 2006 am Institut für Informatik. Seit 2006 arbeitet er am Zentrum für Informationsmanagement und virtuelle Lehre der Universität Osnabrück (virtUOS) und beschäftigt sich dort unter anderem mit der Weiterentwicklung der Lernplattform Stud.IP (als Mitglied der Core Group) und leitet die Entwicklung des an der Universität entstandenen Aufgabentools „Vips“. (Kapitel 5 und 20)

Oliver Müller studierte Wirtschaftsinformatik an der Technischen Universität Clausthal. Seit dem Ende seines Studiums im Jahr 2013 ist er im eCULT Projekt der TU Clausthal beschäftigt. Dort befasst er sich mit dem Einsatz und der Weiterentwicklung von Systemen zur automatisierten Bewertung von Programmier- beziehungsweise Modellierungsaufgaben und promoviert in diesem Themenbereich am Institut für Informatik der TU Clausthal. (Kapitel 13 und 24)

Rainer Oechsle studierte Informatik an der Universität Stuttgart und promovierte an der Universität Karlsruhe (heute KIT) im Bereich Betriebssysteme. Nach der Tätigkeit bei IBM am Europäischen Zentrum für Netzwerkforschung in Heidelberg und im IBM-Forschungslabor Rüschlikon bei Zürich folgte er 1994 einem Ruf auf eine Professur für Rechnernetze und Verteilte Systeme an der Hochschule Trier. Neben der automatischen Softwarebewertung umfassen seine Arbeitsgebiete parallele, verteilte und mobile Anwendungen sowie Softwarekomponenten. (Kapitel 7 und 16)

Niels Pinkwart (pinkwart@hu-berlin.de) studierte Mathematik und Informatik an der Universität Duisburg-Essen und promovierte dort im Jahr 2005. Nach Stationen an der Carnegie Mellon University und der TU Clausthal wechselte er im Jahr 2013 an die Humboldt-Universität Berlin. Dort leitet er den Lehrstuhl „Didaktik der Informatik/Informatik und Gesellschaft“ und das Zentrum für technologiegestütztes Lernen. Innerhalb der GI ist er in den Leitungsgremien der Fachgruppen eLearning und CSCW aktiv und ist Sprecher des Arbeitskreises zu Learning Analytics. Seine Arbeitsgebiete umfassen digitale Medien und Technologien, die menschliches Lernen, Kooperation, Kommunikation und soziale Interaktion unterstützen. (Kapitel 2)

Uta Priss hat im Bereich mathematischer Anwendungen in der Linguistik promoviert und anschließend 15 Jahre erst als Assistant Professor (Indiana University) und dann als Lecturer (Edinburgh Napier University) in der Informationswissenschaft und Informatik geforscht und gelehrt. Seit 2011 ist sie Didaktikerin im eCULT-Projekt der Ostfalia, wo sie sich mit dem Einsatz von eLearning-Technologien in der Lehre und insbesondere mit der automatisierten Bewertung von Programmieraufgaben in MINT-Fächern beschäftigt. (Kapitel 8 und 24)

Frank Puppe (frank.puppe@uni-wuerzburg.de) studierte bis 1983 Informatik an der Universität Bonn, promovierte bis 1986 an der Universität Kaiserslautern und habilitierte bis 1991 an der Universität Karlsruhe mit dem Thema „Problemlösungsmethoden in Experten“. Seit 1992 hat er den Lehrstuhl für Künstliche Intelligenz und Angewandte Informatik an der Universität Würzburg. Seine Forschungsgebiete umfassen Wissensbasierte Systeme, E-Learning, Data Mining, Deep Learning, Sprachverarbeitung, Bildverarbeitung und Medizinische Informatik. (Kapitel 6)

Thomas Richter (richter@tik.uni-stuttgart.de) studierte an der TU-Berlin Physik und Mathematik. 2000 promovierte er an der TU-Berlin am Institut für Mathematik zum Dr. rer. nat. Nach einer zweijährigen Tätigkeit in der Privatwirtschaft zum Thema Bilddatenkompression nahm er eine Stellung am Multimediazentrum der TU-Berlin an, bevor er 2007 an die Abteilung für „Neue Medien in Lehre und Forschung“ des Rechenzentrums der Universität Stuttgart (heute TIK) wechselte. Seine Arbeitsfelder sind E-Learning-Technologien, insbesondere virtuelle und ferngesteuerte Labore sowie elektronische Klausuren. (Kapitel 21)

Peter Riegler hat Physik an der University of New Mexico und der Universität Würzburg studiert mit Forschungstätigkeiten in Quantenoptik, Statistischer Physik und Maschinenlernen. Nach Industrietätigkeit in Telekommunikation, Sensorik und Automatisierungstechnik ist er seit 2002 Professor für Mathematik an der Ostfalia Hochschule. Seine Forschungsinteressen umfassen charakteristische studentische Verständnisschwierigkeiten, wirksame Lehre und Medieneinsatz in der Lehre. (Kapitel 8)

Oliver Rod studierte bis 2011 Elektrotechnik an der Ostfalia Hochschule. Seit 2011 arbeitet er im Zentrum für erfolgreiches Lehren und Lernen der Ostfalia Hochschule als Softwareentwickler und Didaktisch-Technischer Experte. Er entwickelte eine Middlewarelösung zwischen LON-CAPA und dem Praktomat. Sein Interesse liegt im formativen Assessment von Programmieraufgaben. (Kapitel 19 und 22)

David Schuster schloss sein Informatikstudium (B.Sc.) 2015 an der Hochschule Trier ab. Derzeit studiert er Informatik im Masterstudiengang und arbeitet als wissenschaftlicher Mitarbeiter im Fachbereich Informatik, ebenfalls an der Hochschule Trier. Neben seiner Lehrtätigkeit umfasst seine Arbeit die Entwicklung einer verteilten Anwendung zur automatisierten Analyse und Bewertung von Software. (Kapitel 7 und 16)

Gregor Snelting, Jahrgang 1958, schloss 1982 das Studium der Informatik und Mathematik mit Auszeichnung ab, und promovierte 1986 mit Auszeichnung zum Dr.-Ing. 1992 wurde er zum C3-Professor an der TU Braunschweig berufen, 1999 übernahm er den Lehrstuhl Softwaretechnik an der U. Passau; seit 2008 ist er Inhaber des Lehrstuhls für Programmierparadigmen am KIT und erhielt 2012 den Fakultätslehrpreis. Er forscht zu Programmiersprachen, Compilern, Programmanalyse und Softwaresicherheit. Seit 2008 ist er Sprecher des Beirates der Universitätsprofessoren in der Gesellschaft für Informatik. Ferner ist er gewähltes Mitglied des KIT-Senats und spielt als Lead-Gitarrist des „MetalMint"-Projektes jedes Jahr eine Live-Rockshow zur Begrüßung der Informatikerstsemester. (Kapitel 10)

Frauke Sprengel (frauke.sprengel@hs-hannover.de) studierte Mathematik an der Universität Rostock und promovierte dort 1997 über ein Thema aus der angewandten Analysis. Nach Tätigkeiten in der angewandten Forschung in Amsterdam und St. Augustin erfolgte 2001 der Ruf zur Professorin für Computergraphik und Mathematik an die Hochschule Hannover. Neben Themen aus Computergraphik und Bildverarbeitung gilt ihr Interesse dem Einsatz von E-Learning-Technologien in den Grundlagenfächern der Informatik, insbesondereL der automatisierten Bewertung von Aufgaben zur Theoretischen Informatik. (Kapitel 19)

Sven Strickroth studierte Informatik an der TU Clausthal. Danach arbeitete er an der TU Clausthal sowie an der Humboldt-Universität zu Berlin als Wissenschaftlicher Mitarbeiter, wo er im Bereich E-Learning forschte und Ende 2016 promoviert wurde. Er beschäftigte sich unter anderem mit Unterstützungssystemen für die Programmierausbildung, ist an der Entwicklung des Austauschformats ProFormA für Programmieraufgaben maßgeblich beteiligt und koodiniert aktuell ein universitätsweites E-Learning-Projekt an der Universität Potsdam als PostDoc. (Kapitel 2, 13 und 24)

Michael Striewe studierte Informatik an der TU Dortmund und Klassische Archäologie an der Ruhr-Universität Bochum. Derzeit arbeitet er an der Universität Duisburg-Essen, wo er auch 2014 in Informatik promovierte. Seine Forschung vereinigt E-Learning und Software Engineering und befasst sich sowohl mit dem technischen Aufbau von E-Learning- und E-Assessment-Systemen als auch mit dem Einsatz softwaretechnischer Methoden als Bewertungsmechanismen in solchen Systemen. (Kapitel 3, 9 und 18)

Tobias Thelen (tobias.thelen@uni-osnabrueck.de) studierte Computerlinguistik und Künstliche Intelligenz, Informatik und Philosophie an der Universität Osnabrück und promovierte dort 2009 zur automatischen Analyse orthographischer Leistungen von Schreibanfängern. Seit 2002 ist er wissenschaftlicher Mitarbeiter im E-Learning-Zentrum der Universität Osnabrück und befasst sich dort vor allem mit den Entwicklungen rund um die Lernplattform Stud.IP. Seit 2013 ist er in der Geschäftsführung des Zentrum für Forschungsfragen zuständig und lehrt an der Universität Osnabrück Künstliche Intelligenz, Logisches Programmieren, Webtechnologien und Medienbildung. (Kapitel 5)

UNSERE BUCHEMPFEHLUNG

Wolfgang Pfau, Caroline Baetge,
Svenja Mareike Bedenlier,
Carina Kramer, Joachim Stöter (Hrsg.)

Teaching Trends 2016

Digitalisierung in der Hochschule: Mehr Vielfalt in der Lehre

Digitale Medien in der Hochschullehre, Band 5, 2016, 248 Seiten, br., 29,90 €, ISBN 978-3-8309-3548-3

Die Digitalisierung an Hochschulen gewinnt immer mehr an Bedeutung. Digitalisierung kann einen wichtigen Beitrag für hochschuldidaktische Innovationen, für mehr Durchlässigkeit und die weitere Öffnung der Hochschulen für neue Zielgruppen leisten und so mehr Vielfalt in der Lehre generieren.

Im vorliegenden Sammelband zum ELAN e.V. Kongress „TEACHING-TRENDS16: Digitalisierung in der Hochschule: Mehr Vielfalt in der Lehre“ werden empirische Ergebnisse, Beispiele und Erfahrungsberichte zur Umsetzung und Integration didaktischer und technologischer Trends in der Hochschullehre in den Blick genommen. Schwerpunkte bilden hierbei die Diversität in der Lehre, individualisiertes Lehren und Lernen mit digitalen Medien sowie die Erfolgsfaktoren des Einsatzes digitaler Medien an Hochschulen.

UNSERE BUCHEMPFEHLUNG

David Kergel, Birte Heidkamp

Forschendes Lernen mit digitalen Medien. Ein Lehrbuch

#theorie #praxis #evaluation

Digitale Medien in der Hochschullehre, Band 4, 2015, 212 Seiten, br., 24,90 €, ISBN 978-3-8309-3383-0

E-Book: 21,99 €, ISBN 978-3-8309-8383-5

Das wissenschaftliche Feld und damit auch der Bildungsraum Universität sind durch den digitalen Wandel tiefgreifenden Änderungen und Transformationsprozessen ausgesetzt. In den verschiedensten Bereichen des Forschens, Lehrens und Lernens werden zunehmend digitale Medien eingesetzt, so dass sich auch die Anforderungen an eine zeitgemäße akademische Medienkompetenz ändern. Der hochschuldidaktische Ansatz des forschenden Lernens mit digitalen Medien bietet eine Perspektive, dem medialen Wandel im Feld der Hochschuldidaktik angemessen begegnen zu können.

Dieses Buch setzt sich gezielt mit dem forschenden Lernen in einer Zeit des digitalen Wandels und somit mit einem forschenden Lernen mit digitalen Medien auseinander. Im Sinne eines Lehrbuchs werden
– Praxisorientiert zentrale theoretische Überlegungen und
– konkrete Handlungs- sowie
– Evaluationsstrategien (einschließlich Evaluationsergebnissen)
zur Realisierung eines forschenden Lernens mit digitalen Medien integrativ vorgestellt. Dementsprechend wendet sich dieses Buch an Hochschullehrende, Dozent/inn/en, Didaktiker/innen, Lehrer/innen, Student/inn/en und Interessierte, die sich mit digitalen Lehr-/Lernszenarien und/oder forschendem Lernen mit digitalen Medien beschäftigen möchten.

www.waxmann.com

UNSERE BUCHEMPFEHLUNG

Janine Horn

Rechtliche Aspekte digitaler Medien an Hochschulen

Digitale Medien in der Hochschullehre, Band 3, 2015, 262 Seiten, br., 29,90 €, ISBN 978-3-8309-3320-5

E-Book: 26,99 €, ISBN 978-3-8309-8320-0

Digitale Medien spielen mittlerweile zur Erfüllung der Aufgaben einer Hochschule eine erhebliche Rolle. Zu Lehr- und Forschungszwecken werden u. a. Texte, Abbildungen, Filme, Videosequenzen, aufgezeichnete Lehrveranstaltungen und Software verwendet. Die Vermittlung der Inhalte erfolgt in Präsenzveranstaltungen sowie zeit- und ortsunabhängig auf Lernportalen im Internet. Auch Prüfungen werden zunehmend computergestützt durchgeführt. Mit dieser zunehmenden Durchdringung der Hochschullehre mit E-Medien und Blended-Learning-Konzepten sind zahlreiche Fragen des Urheber-, Datenschutz- und Haftungsrechts verbunden. Dieses Buch soll einen allgemeinen Überblick zu dieser Thematik bieten. Es geht insbesondere auf die wichtigen Fragen ein, ob Studierenden urheberrechtlich geschütztes Lehrmaterial zum Abruf im Intranet der Hochschule oder an PC-Arbeitsplätzen in Bibliotheken bereitgestellt werden darf und was bei Vorlesungsaufzeichnungen zu beachten ist.

www.waxmann.com