Software Engineering im Unterricht der Hochschulen

Ulrike Jaeger studierte Informatik an der TU München, promovierte an der Humboldt Universität zu Berlin und arbeitete bei Siemens, FORWISS München und Digital Equipment. Seit 1997 ist sie Professorin an der Hochschule Heilbronn im Studiengang Software Engineering. Dort setzt sie sich intensiv mit didaktischen und curricularen Fragen in der Informatik-Ausbildung auseinander.

Kurt Schneider studierte Informatik in Erlangen und promovierte bei Jochen Ludewig im Software Engineering. Er war Postdoktorand am Center for LifeLong Learning and Design (L3D) der University of Colorado at Boulder. Von 1996-2003 war er am Daimler-Chrysler Forschungszentrum Ulm beschäftigt und hat Projekte mit Unternehmensbereichen durchgeführt. Er hat für DaimlerChrysler auch das Software Experience Center (SEC) geleitet. Seit 2003 ist er Professor für Software Engineering an der Leibniz Universität Hannover und derzeit Studiendekan für Informatik.

Wir danken den Sponsoren der SEUH 2009

dpunkt.verlag

H&D
Hönigsberg & Düvel Datentechnik GmbH

IBM

Capgemini sd&m

Ulrike Jaeger · Kurt Schneider (Hrsg.)

Software Engineering im Unterricht der Hochschulen

SEUH 11 – Hannover 2009

dpunkt.verlag

Ulrike Jaeger
ulrike.jaeger@hs-heilbronn.de

Kurt Schneider
kurt.schneider@inf.uni-hannover.de

Herstellung: Birgit Bäuerlein
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: Koninklijke Wöhrmann B.V., Zutphen, Niederlande

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

ISBN 978-3-89864-619-2

1. Auflage 2009
Copyright © 2009 dpunkt.verlag GmbH
Ringstraße 19 B
69115 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.
5 4 3 2 1 0

Vorwort

Der vorliegende Tagungsband enthält die Beiträge zum 11. Workshop Software Engineering im Unterricht der Hochschulen (SEUH). Dieser Workshop wird vom German Chapter ACM, der GI-Fachgruppe Softwaretechnik, der Schweizerischen Informatik Fachgruppe Software Engineering durchgeführt und findet seit 1992 alle zwei Jahre statt. Software Engineering beschäftigt sich mit der systematischen und disziplinierten Entwicklung von Software. In der Hochschullehre sind nur schwer realistische Aufgaben und Beispiele vermittelbar; Erfahrung wäre wichtig, kann aber weder vorausgesetzt noch einfach »vermittelt« werden. Daher ist es wichtig, dass die Lehrenden des Fachs sich immer wieder austauschen und ihre Ideen besprechen. Dazu dient die SEUH.

Dieses Mal wurden von 21 Einreichungen 11 Beiträge ausgewählt und in vier Gruppen eingeteilt:

Realitätsnähe

Mittlerweile werden Praxisrelevanz und Realitätsnähe von Software-Praktika oft durch Kooperationen mit der Industrie unterstützt. Wir haben einen Experten auf diesem Gebiet eingeladen, der von seinen langjährigen Erfahrungen an der Universität Aalborg berichten wird. In Aalborg wird das gesamte Studium auf der Basis von großen industriebasierten Projekten aufgebaut. Solch eine Struktur ist an den Hochschulen in Deutschland nicht leicht nachzubauen, aber die Ergebnisse in Aalborg sind überzeugend. Zwei Beiträge der diesjährigen SEUH berichten von Erfahrungen mit Realitätsnähe. Im ersten Beitrag bilden ein Lehrender aus der Hochschule und ein Partner aus der Industrie ein »Pair Teaching« und beleuchten ein Projekt von den beiden unterschiedlichen Standpunkten aus. Ein anderer Beitrag diskutiert Auswirkungen eines echten externen Auftraggebers auf das Software-Praktikum.

Überfachliche Kompetenz

Unbestritten für professionelle Software-Entwicklung ist die Bedeutung von überfachlichen Kompetenzen. Nicht zufällig hat die SEUH oft die Arbeitssituation in den Blick genommen, für die wir die Studierenden vorbereiten wollen. Für diesen Komplex haben wir den Vortrag eines Projektberaters eingeladen, der von seinen Erfahrungen aus professioneller Projektarbeit und dem Transfer dieser Erfahrungen zurück in die Hochschulen berichtet. Zwei Beiträge befassen sich mit der entscheidenden Fähigkeit, sauber wissenschaftlich und technisch arbeiten zu können. Das beinhaltet auch das Schreiben von Dokumentationen. Die Bedeutung des kompetenten und verständlichen Schreibens (wie übrigens auch des Lesens und Sprechens) ist den Studierenden oft nicht klar. Sie geben den scheinbar unwichtigen Texten nicht die nötige Aufmerksamkeit. Zwei Beiträge befassen sich damit, wie diese wichtige, aber ungeliebte Aufgabe vermittelt werden kann.

Erlebter Software-Engineering-Prozess

Software Engineering ist ein komplexer Prozess, der über viele Phasen hinweg Teams, ihr Fachwissen, ihre Kreativität, ihre Sorgfalt bei der Ausführung, ihr Methodenwissen, ihre Kritikfähigkeit, Nachtarbeit und Schlafmangel, Entscheidungsfähigkeit und Wahrnehmung zusammenbringt. Viele Praktika entscheiden sich deshalb aus gutem Grund für einen Fokus auf einen bestimmten Bereich, der dann lebendig erlebt und erlitten wird. Wir haben dafür beispielhaft drei Beiträge zu den Themen »Rollen und ihre Wahrnehmung«, »Umgang mit Irrwegen« und »Relevanz von Computerspielen« ausgewählt.

Werkzeuge

Der letzte Bereich ist den Werkzeugen gewidmet. Qualitätssicherung ist eine von Studierenden leider oft vernachlässigte Aufgabe im Software-Engineering-Prozess, die durch Werkzeuge deutlich erleichtert werden kann. Spezielle Verständnisprobleme bei der Entwicklung von nebenläufigen Prozessen oder eingebetteten Systemen können durch Werkzeuge spürbar vermindert werden.

Zusätzlich gibt es noch eine Thesendiskussion zum Erfahrungsbericht des SEUH-Gründers, Jochen Ludewig.

Wir hoffen, mit dieser Auswahl einen interessanten Workshop zu erleben. Wir danken allen Autoren für die Einreichung ihrer Beiträge und dem Programmkomitee, das dann anschließend die nicht immer einfache Auswahl für den Workshop getroffen hat.

Die Programmorganisation wurde durch das kostenfreie Konferenz-Verwaltungssystem Easychair sehr erleichtert. Sebastian Meyer vom Fachgebiet Software Engineering der Leibniz Universität Hannover hat mit großem Einsatz aus vielen Einzelbeiträgen den vorliegenden Tagungsband zu einem vorzeigbaren

Ganzen zusammengefügt. Frau Preisendanz vom dpunkt.verlag war für den Tagungsband wie immer kompetent zur Stelle.

Der Workshop in Hannover wird von studentischen Hilfskräften tatkräftig unterstützt. Die Tagung ist auf finanzielle Unterstützung angewiesen, da den Hochschulen für die Lehre selten große Budgets zur Verfügung stehen. Wir danken deshalb den Sponsoren IBM, Hönigsberg & Düvel und Capgemini sd&m für ihren Beitrag zum Gelingen des Workshops.

Ulrike Jaeger und *Kurt Schneider*
Februar 2009

Das Programmkomitee der SEUH 2009

- Ralf Bruns, Fachhochschule Hannover
- Marcus Deininger, Hochschule für Technik Stuttgart
- Martin Glinz, Universität Zürich
- Dominikus Herzberg, Hochschule Heilbronn
- Ulrike Jaeger, Hochschule Heilbronn (Vorsitz)
- Barbara Paech, Universität Heidelberg
- Günther Riedewald, Universität Rostock
- Axel Schmolizky, Universität Hamburg
- Silke Seehusen, Fachhochschule Lübeck
- Debbie Weber-Wulff, FHTW Berlin
- Andreas Zeller, Universität des Saarlands
- Heinz Züllighoven, Universität Hamburg
- Olaf Zukunft, Hochschule für angewandte Wissenschaften Hamburg

Das ständige Organisationskommitee der SEUH

- Jochen Ludewig, Universität Stuttgart
- Günter Riedewald, Universität Rostock
- Andreas Spillner, Hochschule Bremen

Inhalt

**Eingeladener Vortrag:
Problem-Oriented and Project-Based Learning (POPBL) in
Software Engineering** **1**

Uffe Kjaerulff

**Eingeladener Vortrag:
Im glücklichen Tal der Ahnungslosen:
Kompetenzen für erfolgreiche Projektleitung** **3**

Markus Blaschka

**Erfahrungen mit Pair Teaching für Software Engineering:
Kooperation von Hochschule und Industrie** **5**

Axel Böttcher · Matthias Utesch · Austin Moore

Studentische Projekte: erfolgreich oder realistisch? **17**

Matthias Wetzel · Holger Röder

**Irrwege und Wegweiser im Praktikum der frühen
Softwareentwicklungsphasen** **29**

Robert Garmann

**Game Engineering – Computerspiele in der
Software-Engineering-Ausbildung** **43**

Daniel Volk

Das Projekt AntZeit **57**

*René Fröhlich · Erik Hänel · Markus Knopp · Daniel Nemschok ·
Alexander Schramm · Daniel Volk*

Von Häuptlingen und Indianern – Bachelor/Master als Chance **61**
Karsten Weicker · Nicole Weicker

Erfahrungen bei der Lehre des Software Engineering **75**
Jochen Ludewig

Ein integrativer interdisziplinärer Lehrversuch: Software Engineering und Technisches Schreiben **87**
Gabriele Schmidt · Gerlinde Hollweg

Lehr-Muster bei geringer Ausgangsmotivation für überfachliche Qualifizierung im Software Engineering **101**
Nicola Marsden

Entwicklung eingebetteter Software in einem Softwarepraktikum mit Lego Mindstorms **115**
André Heuer · Kim Lauenroth · Vanessa Stricker · Klaus Pohl

Threadnocchio – Einsatz von Visualisierungstechniken zum spielerischen Erlernen der parallelen Programmierung mit Java-Threads **131**
Dietrich Boles

Werkzeuge zur Qualitätssicherung in der Software-Engineering-Ausbildung **145**
Stephan Kleuker

Eingeladener Vortrag:

Problem-Oriented and Project-Based Learning (POPBL) in Software Engineering

Uffe Kjaerulff

Aalborg University, Denmark
uk@cs.aau.dk

Abstract

At Aalborg University (AAU) we have two software engineering programmes: bachelor and master. Both consist of modules (semesters) worth 30 ECTS credit points, where each module except the last consists of a problem-oriented, semester-long project and a number of courses. The projects are conducted in small student groups under a particular theme as defined in the curriculum and are often conducted in collaboration with industrial partners. The thesis project is, however, often conducted on an individual basis.

The semester project ranges from about 16 credits to 27 credits, and thus comprises the core study activity. The given courses within a module consist of 1-2 project supporting courses and 2-3 courses whose subjects are not tied to the semester theme.

This pedagogical scheme, which basically is applied to all bachelor's and master's programmes at AAU, has proven to be very effective, as dropout rates and completion times are significantly lower than under more traditional pedagogical schemes. Moreover, the Danish (software) industry prefers candidates from AAU over candidates from traditional Danish universities because they have good collaboration skills and can be effective from day one due to the similarity between the typical working style in industry and the POPBL study approach. Moreover, their theoretical skills are at least as good as candidates from classical universities.

Invited Speaker

Uffe Kjaerulff is an associate professor of Computer Science at Aalborg University (AAU). His area of research comprises probabilistic graphical models for decision making under uncertainty, comprising a key component in the broad area of research known as Uncertainty in Artificial Intelligence. He has been employed at AAU since 1990, and has extensive teaching experience under the POPBL paradigm. He has been Head of School of Science at AAU since 2005.

Eingeladener Vortrag:

Im glücklichen Tal der Ahnungslosen: Kompetenzen für erfolgreiche Projektleitung

Markus Blaschka

Dr. Blaschka GmbH, Raubling
info@drblaschka-consulting.de

Zusammenfassung

Sowohl in der Hochschulausbildung als auch in den Unternehmen wird bei der Qualifizierung und Auswahl von Projektleitern immer noch viel Gewicht auf die fachliche Kompetenz (z.B. Wissen über Programmiersprachen, Entwicklungsumgebungen, Implementierung) gelegt. Nach unserer Erfahrung ist die fachliche Kompetenz für einen erfolgreichen Projektleiter oft überbewertet. Zu viel fachliche Kompetenz des Projektleiters kann sogar schädlich für den Projekterfolg sein. Wir wünschen uns daher ein erweitertes Kompetenzmodell für die Ausbildung, das insbesondere die persönliche und soziale Kompetenz stärker berücksichtigt. Diese Kompetenzen entscheiden meist in viel stärkerem Maße über den Projekterfolg.

Anhand von konkreten Projekt-Beispielen aus Unternehmen und Hochschulausbildung verdeutlicht dieser Vortrag, warum der starke Fokus eines Projektleiters auf fachliche Kompetenzen wenig hilfreich ist und oft Projekte in Schwierigkeiten bringt. Wir zeigen, warum Projektleiter und Teams für erfolgreiche Projekte mehr denn je auch überfachliche Kompetenzen benötigen, und postulieren ein modifiziertes Kompetenzmodell, dessen Umsetzung wir uns auch in der Hochschulausbildung wünschen. Erste Ansätze finden sich in Unternehmen und Berufsverbänden, wir stellen einige davon vor.

Zur Person

Markus Blaschka ist Experte für professionelles Projektmanagement. Er wurde in das Qualitätsnetzwerk der Erfolgsgemeinschaft von Joachim Klein sowie in den Kreis der 100 besten Trainer im deutschsprachigen Raum aufgenommen. Sein Know-how bezieht er aus seiner langjährigen Tätigkeit in der Wirtschaft (u.a. Danet, BMW) und in der Forschung (TUM, FORWISS). Er leitet EU-Projekte zur Verbesserung der IT-Unterstützung in der europäischen Textilindustrie sowie Projekte in der Automobil- und IT-Branche. Als Dozent an verschiedenen Fachhochschulen begleitet er Software-Engineering-Projekte als Study Coach und kennt die Fragestellungen in der Lehre aus eigener Erfahrung.

Erfahrungen mit Pair Teaching für Software Engineering: Kooperation von Hochschule und Industrie

Axel Böttcher · Matthias Utesch

Hochschule für angewandte Wissenschaften (FH) München
Fakultät für Informatik und Mathematik

Austin Moore

namics (deutschland) gmbh, München

Zusammenfassung

In diesem Artikel stellen wir eine Variante des Team Teaching vor, die in Anlehnung an das Pair Programming als Pair Teaching bezeichnet wird. Während der vergangenen zwei Jahre setzten wir Pair Teaching im Rahmen unserer Software-Engineering-Ausbildung in dem Kurs Software-Architektur ein.

Wir lehren Software-Architektur mit verteilten Rollen: ein Hochschuldozierender als »Theoretiker« und ein Senior Software Developer aus der Industrie als »Praktiker«. Diese gestalten gemeinsam den Kurs bestehend aus einer Vorlesung und einem Praktikum.

Angeregt durch das Rollenspiel der beiden Dozierenden haben wir Synergieeffekte erlebt. Die Studierenden fühlten sich besser betreut und schätzten insbesondere die »super gute« Praxisnähe.

Gerade im Zuge der Umsetzung des Bologna-Prozesses sind praxisbezogene Lehrmuster im Hinblick auf neue Steuerungsmodelle für den Einsatz von Lehrleistung zu diskutieren.

1 Einleitung

Beginnend mit dem Wintersemester 2001/2002 starteten wir als eine der ersten Fakultäten für Informatik die Umsetzung des Bologna-Prozesses. Heute haben ca. 400 Absolventen ihren Bachelor und 60 Absolventen ihren Master erhalten.

Die neuen Abschlüsse waren für uns auch Anlass, unsere Pädagogik und Didaktik zu reflektieren.

Der Kurs Software-Architektur ist Teil der Software-Engineering-Ausbildung im Bachelor-Studiengang Informatik an der Hochschule (FH) München. Der Kurs besteht aus einer klassischen Vorlesung und einem Praktikum mit je zwei Semesterwochenstunden. Lernziel ist, dass die Studierenden fähig sind, komplexe Software-Systeme zu entwickeln, zu warten und zu betreiben. Sie lernen und trainieren, die dazu erforderlichen Werkzeuge einzusetzen.

Im didaktischen Design des Kurses [Flechsig/Haller 1987] gilt es unserer Erfahrung nach vor allem zwei Fragen zu beantworten:

1. Wie lehren wir in unserem Kurs die relevante Theorie über Architektur großer und komplexer Software-Systeme?
2. Wie lernen unsere Studierenden, sich nach Abschluss des Studiums in diesem dynamischen Gebiet in der Praxis zu bewähren [Schaller 1979]?

Unsere Antwort auf diese Fragen ist eine innovative Variante des Team Teaching, die wir in den vergangenen Semestern erprobt haben [Huber 2000]. Kern unseres didaktischen Designs ist das Lehren mit verteilten Rollen: Theoretiker und Praktiker. Die Rolle des Theoretikers wird von einem Hochschullehrer, die Rolle des Praktikers von einem Senior Software-Entwickler aus der Industrie wahrgenommen. Auf diese Weise wird der Unterricht bereichert durch eine wertvolle, nicht deterministische kommunikative Komponente [Rustemeyer 1985]. Unsere Lernumgebung ist multimedial [Kerres 1998]: Der Praktiker entwickelt live am Rechner in der Software-Entwicklungsumgebung Eclipse. Der Theoretiker kommentiert diesen Vorgang, formuliert Fragen aus studentischer Sicht an den Praktiker (Experteninterview, siehe [Waldherr/Walter 2008]) und regt damit die Kommunikation und Interaktion mit den Studierenden an. Zeitweise haben wir auch gemeinsam entwickelt und damit Pair Programming vorgelebt.

2 Der Spagat zwischen Theorie und Industriepraxis

> *Talented people are the most important element in any software orgnization ... The better and more experienced they are, the better the chance of producing first-class results. [Humphrey 1990]*

Talent benötigt mehrere Jahre Erfahrung, um sich zu entwickeln. Schlüssel dafür ist auch ein Umfeld mit guten und kooperativen Kollegen, von denen man lernen kann und mit denen gemeinsam die Fähigkeiten und Fertigkeiten ständig verbessert werden können. Technologien und Werkzeuge entwickeln sich zudem rasant weiter; gute Entwickler bleiben in dieser Landschaft auf dem Laufenden [Lehner 1991].

Auch der Hochschullehrer muss die Herausforderung bewältigen, in dieser Landschaft auf dem Laufenden zu bleiben. Selbst wenn Hochschullehrer aus dem

Umfeld der industriellen Software-Entwicklung kommen, sind sie in ihrem Alltag an der Hochschule oft verhältnismäßig weit davon entfernt. Auch dann, wenn man in Forschungsprojekten oder -Kooperationen mit Software-Entwicklung zu tun hat, kommt die Praxis häufig notgedrungen zu kurz.

Andererseits steht der Praktiker aus der Industrie, wenn er lehren will, unserer Erfahrung nach vor der Herausforderung, seinen Stoff didaktisch aufzubereiten. Aspekte dabei sind:

- Aufbereiten des Stoffes
- Finden geeigneter Übungsaufgaben
- Betreuung von Praktika
- Formulieren und korrigieren von Prüfungen

Dieses gerade ist die Kernkompetenz des Hochschullehrers, der im Laufe der Zeit sehr viel Erfahrung gewonnen und ein gutes Gefühl dafür entwickelt hat, wie man die Lernprozesse der Studierenden am besten befördern kann.

In der Vergangenheit haben wir mit zwei verschiedenen Lösungsansätzen gearbeitet: Eine Variante ist, dass ein Mitarbeiter aus einem Unternehmen einen Lehrauftrag für die Durchführung eines kompletten Kurses übernimmt. In der Praxis ist es oft schwierig, geeignete Lehrbeauftragte zu finden, die bereit sind, den großen Aufwand auf sich zu nehmen, den es bedeutet, jede Woche eine Vorlesung mit Praktikum abzuhalten – neben ihrer eigentlichen Industrietätigkeit.

In der zweiten Variante haben wir die klassische Unterrichtsform, durchgeführt von einem Dozierenden, durch einen gelegentlich teilnehmenden Praktiker aus einem Unternehmen bereichert. Unsere Erfahrungen damit sind nicht durchgängig positiv, ähnlich wie in [Waldherr/Walter 2008, S. 30] beschrieben. Hinzu kommt, dass es für diesen externen Dozenten nicht so gut möglich ist, seine didaktischen Fähigkeiten weiterzuentwickeln.

Als Ergebnis dieser Erfahrungen entstand unser didaktisches Design in Analogie zum *Pair Teaching* von Andersson und Bendix [Andersson/Bendix 2005].

3 Pair Teaching

> *The child who receives a hammer for Christmas will discover that everything needs pounding. [Weinberg 1985]*

Wir wollen unseren Studierenden bewusst das Werkzeug »Teamarbeit« im Software Engineering beibringen und leben dieses Element deshalb in unserer Lehre vor. Ein etablierter Ansatz dieser Teamarbeit ist das Pair Programming. Es wird z.B. für die Praxis beschrieben in [Beck 2000] und hinsichtlich verschiedener Effizienzkriterien wissenschaftlich untersucht in [Williams 2000]. Die Komplexität, die in unserem Fachgebiet Software Engineering aus rasantem Fortschritt sowohl in Theorie als auch der Praxis entsteht, wird personalisiert durch die Rollen

»Theoretiker« und »Praktiker« und durch deren Live-Kommunikation und Interaktion dargestellt und damit im Sinne des Systems-Thinking-Ansatzes nach Senge bewältigt [Senge 1990]. Jede der Rollen wird durch einen Lehrenden personalisiert [Andersson/Bendix 2005]. Wir haben diese Methode insofern erweitert, als wir je einen Dozierenden aus Hochschule und aus der Industrie zusammenbringen.

Mediale Präsentation von Inhalten durch zwei Personen ist eine in verschiedenen Bereichen durchaus übliche Praxis:

- Fernsehsendungen, die man dem Bildungsfernsehen zuordnen kann, werden gelegentlich von einem Paar bestehend aus einem Moderierenden sowie einem Experten gestaltet. Ein Beispiel hierfür ist die c't Magazin-Sendung (siehe *www.cttv.de*). Hier spielt der Moderator die Rolle eines Lernenden. Ähnlich verhält es sich in Sendungen des IT-Kompaktkurses im Bildungskanal BR-Alpha (siehe *www.br-alpha.de*).
- Die Pro- und Kontra-Debatte. Diese wird vor allem als Moderationstechnik eingesetzt. Sie wird zudem oft in Printmedien verwendet, um gegensätzliche Standpunkte zu verdeutlichen.

Wir beschreiben im Folgenden drei Beispiele für unsere Umsetzung von Pair Teaching im Kurs Software-Architektur.

3.1 Der Kurs Software-Architektur

Unser Kurs Software-Architektur besteht aus einer Vorlesung und einem Praktikum zu je zwei Semesterwochenstunden. Unser didaktisches Design in diesem Kurs orientiert sich an [Bass et al. 2003]. Wir behandeln zunächst Qualitätsmerkmale von Software [Liggesmeyer 2002]. Mit dem Qualitätsmerkmal Funktionalität sind die Studierenden bereits vertraut, und sie betrachten dies als das zentrale Merkmal. Deshalb führen wir – nach einer theoretischen Einführung – im Praktikum einige Refactoring-Übungen durch, um die Bedeutung von Merkmalen wie Testbarkeit, Erweiterbarkeit (Modifiability) und Benutzerfreundlichkeit (Usability) am praktischen Beispiel zu verdeutlichen.

Im weiteren Verlauf der Veranstaltung müssen die Studierenden Plug-ins für Eclipse[1] entwickeln. Ein vorhandenes Framework zu verwenden, hat den Vorteil, dass bereits komplexe Strukturen vorhanden sind, die durch eigenen Code erweitert werden müssen [Böttcher 2006]. Hinreichend komplexe Software selbst zu entwickeln, um daran Architektur studieren zu können, erschien uns zu wenig zielführend für ein Praktikum mit nur zwei Stunden pro Woche. Aufgrund des Pattern-orientierten Designs von Eclipse [Gamma/Beck 2004] lässt sich Software-Architektur anhand des Einsatzes von Patterns sehr praxisnah unterrichten.

1 *www.eclipse.org*

3.2 Beispiel 1: Vorlesung zum Thema »Refactoring«

Refactoring ist eine Basistechnik, um die Qualität von Software zu verbessern. Ziel der Vorlesung zum Refactoring ist, die Studierenden mit dem theoretischen Werkzeug und der praktischen Vorgehensweise beim Refactoring vertraut zu machen und ihnen ein Gefühl für die Beurteilung existierenden Quellcodes zu geben.

In der Refactoring-Vorlesung wird das Rollenmodell Theoretiker/Praktiker wie folgt umgesetzt:

Der Praktiker führt die Software-Entwicklung mit einer Entwicklungsumgebung vor, nimmt damit also die Rolle des Driver im Pair Programming wahr [Walther 2005]. Der Theoretiker hat das Auditorium im Blick und gibt zusätzliche Kommentare ab, die das Verständnis erleichtern. Gleichzeitig spielt er die Rolle des Navigators, also des zweiten Pair-Programmierers. Wie der Navigator beim Pair Programming Abstand zur aktuellen Arbeit hat, kann er die aktuelle Strategie zur Problemlösung kommentieren, gemeinsam mit dem Auditorium reflektieren, überprüfen und verbessern. Probleme werden so im Dialog zwischen Navigator und Driver in Interaktion mit dem Auditorium gelöst.

Unser Ansatz vermeidet damit die Ermüdung der Studierenden, die das Vorführen von Software-Entwicklung mit einer Entwicklungsumgebung für die Studierenden oft mit sich bringt. Denn insbesondere das dauernde Wechseln von Editor-Fenstern erfordert hohe Konzentration der Studierenden. Ebenso ermöglichen wir den Dozierenden die gleichzeitige Konzentration auf die Entwicklungsumgebung sowie das Auditorium. Unsere Vorlesung mittels Pair Teaching ist somit zugleich eine praktische Demonstration von Pair Programming.

3.3 Beispiel 2: Praktikum zum Thema »Refactoring«

Die Studierenden sollen in der Lage sein, Code zu beurteilen und eine gewisse Basisqualität herzustellen. Lernziel ist es, das Gefühl der Studierenden dafür zu entwickeln und Erfahrung zu vermitteln, wo Refactoring beendet werden kann, weil weitere Schritte nicht mehr wirtschaftlich sind oder weil es dafür keinen Kunden mehr gibt.

Ausgangspunkt ist ein Java-Programm, welches das Spiel Vier-Gewinnt realisiert. Dieser von einem Programmieranfänger geschriebene Code ist so offensichtlich von schlechter Qualität, dass die Notwendigkeit von Verbesserungen unmittelbar ins Auge springt. Die erste Übung besteht darin, die Software testbar zu machen. Dazu geben wir Unit-Tests vor, die aber erst nach einigen Basismodifikationen überhaupt zum Laufen kommen. Diese Abweichung von der reinen Lehre eines Test-Driven.Ansatzes nehmen wir bewusst in Kauf, da die Aufgabenstellung, eine Software testbar zu machen, durchaus praxisrelevant ist. Als Basis für die Beurteilung haben wir einen Katalog mit Punkten formuliert, der sich abprüfen lässt, der aber den Studierenden nicht vorab bekannt gegeben wird.

- Wir erwarten die Beseitigung statischer Methoden, soweit möglich.
- Wir erwarten eine Umstrukturierung gemäß objektorientierter Prinzipien.
- Die Software enthält einen Fehler, der im Refactoring zutage tritt.
- Wir erwarten die Beseitigung duplizierten Codes.
- Wir erwarten eine weitgehende Testabdeckung, die mit dem Coverage-Werkzeug Emma[2] gemessen wird.

Nach rein akademischer Ausbildung neigen die Studierenden dazu, Lösungen zu bauen, die unnötig kompliziert sind, wie das Patterns-Happy-Beispiel aus [Kerievsky 2004, S. 23 ff]. Der Dozierende in der Rolle »Praktiker« bremst hier die Studierenden rechtzeitig, so dass sie nicht eine vollständig flexible Lösung erarbeiten, die allen Prinzipien objektorientierten Designs genügen, aber unnötig viel Aufwand erfordern (dies ist als Yagni-Prinzip – you ain't gonna need it – bekannt). Der »Praktiker« wird dies auch aus seiner Erfahrung heraus sehr genau und glaubhaft begründen können. Der »Theoretiker« erörtert das Problem des Abbruchs von Refactoring mit dem »Praktiker« und den Studierenden. An dieser Stelle wird Bewusstsein dafür geschaffen, dass sich keine eindeutigen Aussagen formulieren lassen, vielmehr »Gespür« erforderlich ist.

3.4 Beispiel 3: Praktikum zum Thema »Eclipse Plug-ins«

Schwerpunkt unseres Praktikums »Eclipse Plug-ins« ist es, erste persönliche Erfahrungen mit komplexer Software zu sammeln: Die Studierenden sollen in der Lage sein, komplexe Software unter Berücksichtigung von Best Practices agiler Ansätze selber zu schreiben.

Ein erstes Beispiel zeigt unser Lernpfad für den Einsatz von Patterns in komplexer Software. In unserem Fall gilt es, ein Plug-in für das Verwalten und Abspielen von Bitströmen im MP3-Format zu erstellen. An einer Stelle kommt dabei das Composite Pattern ins Spiel. Abbildung 1 skizziert ein Klassendiagramm für das Composite Pattern [Gamma et al. 1995]. Das Bild verdeutlicht die beiden möglichen Platzierungen der Methoden für die Verwaltung der Kinder (`addChild`, `getChildren` etc.). Beide Lösungen haben spezifische Vor- und Nachteile; vgl. dazu etwa [Gamma et al. 1995] und [Gamma/Beck 2004].

Ein Dozierender kann dieses Pattern erläutern – aber: Unser Lernziel ist, den Studierenden nicht Musterlösungen, sondern in Diskussionskultur die Fähigkeit zu vermitteln, abzuwägen und zu entscheiden beizubringen. Deshalb diskutieren wir dieses Pattern, seine möglichen Ausprägungen und Konsequenzen im Dialog zweier Dozierender. So aktivieren wir die Studierenden, mit zu diskutieren.

Ein zweites Beispiel ist unser Lernpfad für die Schlüsselwörter `volatile` und `synchronized`. Die Rolle des Theoretikers ist, den Studierenden dafür die Defini-

2 *emma.sourceforge.net*

tion und Syntax zu vermitteln. Reichen aber diese Kenntnisse aus, damit unsere Studierenden konkrete Probleme lösen können? Hier zeigt der Praktiker aus seinem Alltag des Software Engineering beispielhafte Anwendungen auf. Gemeinsam diskutieren dann Theoretiker und Praktiker die jeweiligen Auswirkungen dieser Schlüsselwörter in konkreten Programmen. Anschließend üben die Studierenden selbstständig. Die Dozierenden unterstützen gemäß der Idee des Pair Programming.

Grundsätzlich macht das didaktische Design des Praktikums es möglich, den Studierenden am konkreten Beispiel zu zeigen, dass in der Arbeit mit einer Entwicklungsumgebung immer wieder nicht vorhersehbare Probleme auftreten können. Wir können so aufzeigen, dass Werkzeuge oft Spezifika können, die sich einem logisch denkenden Menschen nicht unmittelbar erschließen und dessen Bedienung Spezialkenntnisse durch praktische Erfahrung erfordert.

Hier beweist sich unser Pair Teaching durch die Kommunikation zwischen den Dozierenden und durch die Summe der Erfahrungen eines Theoretikers und eines Praktikers. Wenn solche Probleme auftreten, kann einer der beiden Dozierenden den Praktikumsbetrieb fortsetzen, während der andere versucht, diese Einzelprobleme zu lösen und zu dokumentieren.

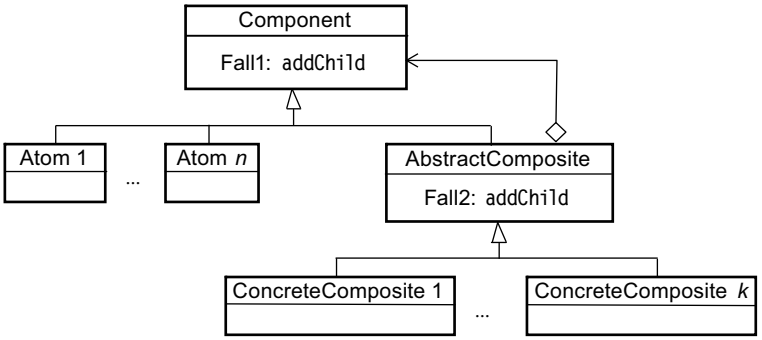


Abb. 1 *Skizze eines Klassendiagramms zum Composite Pattern. Die Methoden zur Verwaltung der Kinder können entweder in Component oder in AbstractComposite implementiert werden (mit Fall 1 und Fall 2 gekennzeichnet).*

4 Bewertung

Bei der Bewertung unseres didaktischen Designs des Pair Teaching orientieren wir uns im Grundsatz an den Punkten von [Williams 2000]. Wir verwenden auch dessen englische Bezeichnungen. Diese Kriterien wurden gewählt, da wir unsere Evaluierung derzeit auf qualitative Aussagen und den Dialog aller am Lehr-/Lernprozess Beteiligten stützen und erst im nächsten Schritt eine passende quantita-

tive Evaluierung entwickeln werden. Wir bewerten also das Pair Teaching mit Hilfe folgender Punkte:

- **Pair Pressure**
  Gemeinsames Arbeiten am selben Projekt erhöht die Motivation der Lehrenden, oder wie Williams formuliert: »*they do not want to let their partner down*« [Williams 2000, S. 39]. Diese Erfahrung haben wir selber gemacht: Man will den Kollegen nicht »hängen lassen«. Daraus entsteht ein positiver psychologischer Druck, gemeinsam die Qualität des Lehr-/Lernprozesses zu verbessern. Ein gutes Beispiel hierfür bietet das Thema der Synchronisation mehrerer Threads im Rahmen unserer MP3-Aufgabe. Wir konnten uns gegenseitig sehr gut helfen, die Fakten dazu ins Gedächtnis zu rufen und live gemeinsam den Studierenden beim Lösen der Probleme zu helfen.
- **Pair Think**
  Bei Aufgaben, die das konkrete Design einer Lösung nicht vorgeben, kommen verschiedene Gruppen von Studierenden zu unterschiedlichen Lösungen: »*A system with multiple actors possesses greater potential*« [Williams 2000, S. 41].

  Das Ändern des API bei einem Refactoring bietet hierfür ein konkretes Beispiel: Der Theoretiker erläutert, dass die Signaturen öffentlicher Methoden von existierender, laufender Software nicht geändert werden dürfen. Der Praktiker zeigt Fälle auf, in denen die Änderung sogar unabdingbar ist, nämlich dann, wenn ein API sehr schlecht ist. Die an solchen Stellen mit den Studierenden geführten Diskussionen erlebten alle Beteiligten als hilfreich.
- **Pair Relaying**
  Pair-Programmierer berichten, dass sie gemeinsam Probleme lösen konnten, die keiner der beiden allein hätte lösen können: »*together they can evolve solutions to unruly or seemingly impossible problems*« [Williams 2000, S. 42]. Dieser Effekt ähnelt den Erfahrungen, die über Brainstorming und Teamarbeit im Allgemeinen berichtet werden. Senge beschreibt dies als Team Learning [Senge 1990, S. 9]. Hier ergänzen sich Praktiker und Theoretiker durch ihre unterschiedlichen Sichtweisen.

  Wir haben im Pair Teaching diesen Effekt im Praktikum mehrmals an Stellen erfahren, an denen einer der Dozierenden einen Sachverhalt nicht schnell genug auf den Punkt bringen konnte. Dieser Aspekt wurde in den Evaluierungen mehrfach positiv erwähnt.
- **Pair Reviews**
  Begründet in den Arbeiten von Fagan ist das Pair Reviewing eine Standardmethode im Software Engineering [Fagan 1976], [Fagan 1986]. Wir Autoren dieses Artikels haben selber in unserer industriellen Praxis erfolgreich Reviews bei großen Systemen eingesetzt [Utesch/Thaler 1996]. Tom Gilb beschreibt den Kern eines Reviews: »*Reviews (...) are typically peer group dis-*

cussion activities« [Gilb/Graham 1993, S. 5]. Wie lehren wir Studierenden die Kultur der Diskussion? Unsere Antwort darauf ist: Mit den Rollen »Theoretiker« und »Praktiker« leben wir Diskussionen in unserem Kurs vor.

Das persönliche Fazit der Dozierenden ist, dass durch das sozial-emotionale Element »Dialog zweier Dozierender« ein positiveres Erleben der universitären Umwelt ermöglicht wird. Denn einer Längsschnittstudie zu den Bildungslebensläufen von über 2.300 deutschen Abiturienten zufolge, die den Bedingungen und subjektiven Begründungen von Studienabbrüchen nachgegangen ist, sind Gründe für einen Studienabbruch neben Lernschwierigkeiten und schlechten Berufsaussichten vor allem Praxisferne [Gold/Kloft 1991]. Nach dieser Studie wiesen die Nichtabbrecher insbesondere ein positiveres Erleben der universitären Umwelt und eine bessere Informiertheit über universitäre Angelegenheiten auf. Die genannten Themen »Praxisnähe« und »Berufsaussichten« werden von uns adressiert durch einen eigenen Dozierenden aus der Praxis. Wir bringen den »Beruf« und die »Praxis« personifiziert in die Hochschule.

Wir haben unsere Studierenden im Rahmen einer qualitativen Evaluierung gefragt, ob das Pair Teaching in dieser Form weiter gepflegt und entwickelt werden soll. Repräsentativ sind folgende Rückmeldungen:

- »Super gute Praxisnähe durch die Zusammenarbeit.«
- »Besonders manche Fallbeispiele/Kommentare sind interessant, wenn man hört, wie es Austin in seiner Firma bewerkstelligt.«
- »Ich fand, dass die Dozierenden sich sehr gut ergänzt haben.«
- »Die Dozierenden konnten sich mehr Zeit nehmen für Fragen oder um zu diskutieren, was in einem Feld wie der Software-Architektur durchaus notwendig ist.«

Am meisten beeindruckte die Lernenden, dass sie in die Kommunikation und Interaktion der dozierenden Experten mit einbezogen wurden – und zwar auf »Augenhöhe»: Angeregt durch das Rollenspiel zwischen Theoretiker und Praktiker äußerten die Studierenden frei ihre Meinung und begannen schrittweise selber die Verantwortung für das Gelingen der Lehrveranstaltung mit zu übernehmen. Wie bei [Utesch 2008] konstatiert, verstärkt das sowohl Engagement als auch Durchhaltevermögen der Lernenden.

5 Ausblick

Wir haben in dieser Arbeit das Pair Teaching als eine Variante des Team Teaching am Beispiel unseres Kurses Software-Architektur an der Hochschule München vorgestellt.

Pair Teaching wird aus Sicht der Dozierenden wie auch der Studierenden als wünschenswert und realitätsnah bewertet. Aus Sicht heute gültiger Verordnun-

gen zur Berechnung von Lehraufwänden mag eine Bewertung nicht so positiv ausfallen: Pair Teaching benötigt zwei Dozierende pro Kurs, was nach Kapazitätsverordnung [Kap-VO 2005] sowie Verordnung über die Lehrverpflichtung [LUFV 2007] nur in Ausnahmefällen zulässig ist bzw. nicht voll auf das jeweilige Lehrdeputat angerechnet werden kann. Vielleicht bietet hier die Diskussion um Skill-Modelle und Teaching Points [Handel et. al. 2005] im Zuge der Umsetzung des Bologna-Prozesses einen Rahmen, in dem statt Kapazitätsverordnung und Verordnung über die Lehrverpflichtung Erkenntnisse innovativer didaktischer Designs mit neuen Steuerungsmodellen für den Einsatz von Lehrleistung verbunden werden. Gerade in diesem Fall haben wir Pair Teaching als äußerst wertvoll erlebt, da diese Methode auf Kommunikation und Interaktion aller Beteiligten setzt und somit gemäß dem Bologna-Prozess den Erwerb von Kompetenzen und nicht den Erwerb von Fachwissen in den Vordergrund stellt. So gelingt es uns, unseren Studierenden schon an der Hochschule Praxis zu bieten. Deshalb sind wir überzeugt, dass auch andere Veranstaltungen von diesem Modell profitieren könnten. Neue Modelle zur Festlegung und zum Einsatz von Lehrleistung müssen dies künftig begleiten.

> *Nothing new ever works, but there's always hope that this time will be different. [Weinberg 1985, S. 142]*

Literatur

[Andersson/Bendix 2005] R. Andersson, L. Bendix: Pair Teaching – an eXtreme Teaching Practice. Proceedings of 4:e Pedagogiska Inspirationskonferensen, LTH, Lund University, Juni 2006

[Bass et al. 2003] L. Bass, P. Clements, R. Kazman: Software Architecture in Practice. SEI Series in Software Engineering, Addison-Wesley, Boston. 2003

[Beck 2000] K. Beck: eXtreme Programming explained. Addison Wesley, Boston, San Francisco. 2000

[Böttcher 2006] A. Böttcher: Eclipse in der Hochschul-Lehre. iX-Konferenz Bessere Software, Frankfurt, 2006.

[Fagan 1976] M. E: Fagan: Design and Code Inspections to Reduce Errors in Program Develpoment. IBM Systems Jounal, 15(3), 182-211. 1976

[Fagan 1986] M. E. Fagan: Advances in Software Inspections, IEEE Transactions on Software Engineering. SE-12(7), July, 744-751. 1986

[Flechsig/Haller 1987] K. H. Flechsig, H.-D: Haller: Einführung in didaktisches Handeln. Stuttgart: Klett. 1987

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley, Boston, 1995

[Gamma/Beck 2004] E. Gamma, K. Beck: Contributing to Eclipse. Addison-Wesley, Boston. 2004

[Gilb/Graham 1993] T. Gilb, D. Graham: Software Inspection. Addison-Wesley. 1993

[Gold/Kloft 1991] A. Gold, C. Kloft: Der Studienabbruch: Eine Analyse von Bedingungen und Begründungen. Zeitschrift für Entwicklungspsychologie und Pädagogische Psychologie 23, S. 265-279. 1991

[Handel et al. 2005] K. Handel, Y. Hener, L. Voegelin: Teaching Points als Maßstab für die Lehrverpflichtung und Lehrplanung. CHE-Arbeitspapier Nr. 69, 2005

[Huber 2000] B. Huber: Team-Teaching – Bilanz und Perspektiven, Verlag Peter Lang, Frankfurt, 2000

[Humphrey 1990] W. S. Humphrey: Managing the Software Process. Addison-Wesley New York. 1990

[KapVO 2005] Verordnung über die Kapazitätsermittlung, die Curricularnormwerte und die Festsetzung von Zulassungszahlen in Bayern (Kapazitätsverordnung – KapVO) vom 1. Januar 2005.

[Kerievsky 2004] J. Kerievsky: Refactoring to Patterns. Addison Wesley, Boston. 2004

[Kerres 1998] M. Kerres: Multimedia und telemediale Lernumgebungen. Oldenbourg Verlag, München. 1998

[Lehner 1991] F. Lehner: Softwarewartung – Management, Organisation und methodische Unterstützung. Carl-Hanser-Verlag, München, Wien 1991.

[Liggesmeyer 2002] P. Liggesmeyer: Software-Qualität – Testen, Analysieren und Verifizieren von Software. Heidelberg, Spektrum Akademischer Verlag, 2002.

[LUFV 2007] Verordnung über die Lehrverpflichtung des wissenschaftlichen und künstlerischen Personals an Universitäten, Kunsthochschulen und Fachhochschulen in Bayern (Lehrverpflichtungsverordnung – LUFV) vom 14. Februar 2007.

[Rustemeyer 1985] D. Rustemeyer: Kommunikation oder Didaktik? Aporien kommunikativer Didaktik und Konstruktionsprobleme kommunikativer Bildungstheorie. Pädagogische Rundschau 39, S. 61-85, 1985

[Schaller 1979] K. Schaller: Pädagogik der Kommunikation. In K. Schaller, Erziehungswissenschaft der Gegenwart. Prinzipien und Perspektiven moderner Pädagogik (S. 155-181). Bochum: Kamp. 1979.

[Senge 1990] P.M. Senge: The Fifth Discipline: the art & practice of the learning organization. London: Century Business – an imprint of Random House UK Ltd. 1990.

[Utesch/Thaler 1996] M. Utesch, M. Thaler: Effektivität und Effizienz von Softweare Inspektionen. In Liggesmeyer, P. (Hrsg.): Test, Analyse und Verifikation von Software. München: Oldenbourg Verlag. 1996.

[Utesch 2008] M. C. Utesch: Leistung braucht Netzwerke: Soziales Lernen und fachliche _Leistung. In B. S. Kultus, Werte machen stark – Praxishandbuch zur Werteerziehung (pp. 94-96). Augsburg: Brigg Pädagogik, 2008.

[Waldherr/Walter 2008] F. Waldherr, C. Walter: Methodensammlung für die Lehre an Hochschulen. Ingolstadt: Didaktikzentrum DIZ, 2008.

[Walther 2005] T. Walther: Pair Programming. Ausarbeitung im Rahmen des Seminars, Agile Softwareprozesse. FU Berlin, 2005
(online unter *www.tilman.de/uni/PairProgramming.pdf*).

[Weinberg 1985] G. M. Weinberg: The secrets of consulting, Dorset House Publishing, New York, 1985

[Williams 2000] L. Williams: The Collaborative Software Process. Dissertation an der University of Utah, 2000
(online unter *http://collaboration.csc.ncsu.edu/laurie/Papers/dissertation.pdf*).

Studentische Projekte: erfolgreich oder realistisch?

Matthias Wetzel · Holger Röder

Universität Stuttgart
Institut für Softwaretechnologie
Abteilung Software Engineering
{matthias.wetzel|holger.roeder}@informatik.uni-stuttgart.de

Zusammenfassung

Studentische Software-Projekte, die sowohl realistisch als auch erfolgreich sein sollen, sind mittlerweile fester Bestandteil der Softwaretechnik-Ausbildung. Anhand konkreter Beispielprojekte der vergangenen Jahre wird gezeigt, dass die beiden Ziele Erfolg und Realitätsnähe eng miteinander verwoben sind, und es werden Leitlinien entwickelt, wie sie sich verbinden lassen. Generell sollten diejenigen Aspekte umgesetzt werden, die die Realitätsnähe erhöhen, ohne die Erfolgsaussichten zu mindern. Dabei sind erfolgreiche Projekte realistischen, aber erfolglosen Projekten vorzuziehen.

1 Einleitung

Studentische Software-Projekte sind mittlerweile an vielen Universitäten und Hochschulen fester Bestandteil der Softwaretechnik-Ausbildung. Auch wenn sich die jeweiligen Projekte hinsichtlich Aufgabenumfang, Dauer, Teamgröße und Rahmenbedingungen teilweise deutlich voneinander unterscheiden, stellt sich doch bei allen die Frage nach der Realitätsnähe, die in den Projekten angestrebt wird. Gemeinhin gilt die Durchführung möglichst realistischer, also Projekten in der Wirtschaft möglichst ähnlicher studentischer Projekte als sinnvoll und empfehlenswert. Dabei wird jedoch teilweise außer Acht gelassen, dass Realitätsnähe als Ziel stets mit anderen Zielen, etwa der erfolgreichen Projektdurchführung, konkurriert.

Im Folgenden betrachten wir das Zusammenspiel von Realitätsnähe und Erfolg in studentischen Projekten. Grundlage für die Betrachtung sind Software-Projekte, die in den vergangenen Jahren im Studiengang Softwaretechnik an der Universität Stuttgart durchgeführt wurden. Dazu sollen zunächst die Rahmenbedingungen der Projekte und ihre Einbettung in den Studiengang Softwaretechnik vorgestellt werden. In Kapitel 2 werden die Ziele Erfolg und Realitätsnähe erläutert. Kapitel 3 befasst sich mit durchgeführten studentischen Projekten und den dort gemachten Erfahrungen mit Blick auf diese beiden Ziele. In Kapitel 4 wird die Frage nach der Priorität der Ziele diskutiert, und in Kapitel 5 werden Empfehlungen für Konzeption und Durchführung studentischer Software-Projekte in der Lehre gegeben. Kapitel 6 schließt mit einem Fazit.

1.1 Studentische Projekte im Studiengang Softwaretechnik

Seit 1996 existiert an der Universität Stuttgart ein eigenständiger Diplom-Studiengang Softwaretechnik [Ludewig 1999]. Der Studiengang ist als konstruktiver Informatik-Studiengang konzipiert und hat den Anspruch, die Studierenden optimal auf eine Tätigkeit in der Software-Industrie vorzubereiten [Ludewig 2003]. Projektarbeit ist ein zentraler Bestandteil des Studiums: Der Lehrplan sieht insgesamt drei studentische Software-Projekte vor.

Im Grundstudium führen Studierende im Rahmen des Softwarepraktikums ein kleines Software-Projekt in Dreiergruppen durch. Die Aufgabe ist dabei überschaubar, Projektablauf, Termine und Leistungen werden durch den betreuenden Lehrstuhl vorgegeben, die Vertiefung der Programmierkenntnisse nimmt einen wichtigen Platz ein [Wetzel 2007]. Die Dauer des Softwarepraktikums ist auf ein Semester begrenzt.

Bei den im Hauptstudium folgenden zwei Studienprojekten arbeiten Gruppen von sechs bis zwölf Studierenden gemeinsam an einer Aufgabe. Das erste Studienprojekt findet an einem Informatik-Lehrstuhl statt, das zweite Studienprojekt wird an einem Lehrstuhl des Anwendungsfachs (Automatisierung, Technologie oder Verkehr) durchgeführt. Jedes Studienprojekt dauert zwei, höchstens drei Semester. Die Studierenden führen das Projekt weitgehend eigenständig und eigenverantwortlich durch. Die Aufgabe ist im Vergleich zum Softwarepraktikum deutlich umfangreicher, häufig erfolgt die Aufgabenstellung in Kooperation mit einem externen Kunden aus der Industrie, der Interesse am Einsatz des entstehenden Software-Produkts hat. Daraus leitet sich der Anspruch ab, dass die in Studienprojekten entwickelte Software Produktreife besitzen soll.

Die nachfolgenden Betrachtungen zu Erfolg und Realitätsnähe studentischer Projekte beziehen sich auf Studienprojekte im Studiengang Softwaretechnik, sind aber auf ähnlich konzipierte Projekte in anderen Studiengängen übertragbar.

2 Erfolg und Realitätsnähe

Erfolg und Realitätsnähe sind zentrale Ziele bei der Durchführung studentischer Projekte; allerdings ist nicht immer klar, welches Verständnis den Begriffen zugrunde liegt.

Zur Erklärung soll zunächst der Erfolgsbegriff in industriellen Projekten betrachtet werden. Ausgehend davon kann ein Maßstab für den Erfolg studentischer Projekte erarbeitet werden. Für die Realitätsnähe studentischer Projekte, die ein Maß dafür darstellt, inwieweit die Rahmenbedingungen dieser Projekte denen industrieller Projekte ähneln, lassen sich ebenfalls Kriterien identifizieren.

Es ist offensichtlich, dass die für die Bewertung des Projekterfolgs angelegten Kriterien zu den Rahmenbedingungen gehören. Somit sind die beiden Ziele Erfolg und Realitätsnähe nicht unabhängig voneinander, sondern beeinflussen sich gegenseitig.

2.1 Erfolg

Der Erfolg eines Software-Projekts wird anhand des Erfüllungsgrads bestimmter Erfolgskriterien bestimmt. Im Allgemeinen werden folgende drei Erfolgskriterien herangezogen (vgl. auch [CHAOS 1995]):

- Der Auslieferungstermin wird eingehalten.
- Das Budget wird eingehalten.
- Die Anforderungen an das Software-Produkt werden erfüllt.

[Fahl 2003] nennt drei weitere Erfolgskriterien für Software-Projekte:

- Das Produkt wird verwendet.
- Der Kunde ist zufrieden.
- Das Projektteam hat etwas gelernt.

Der Erfolg eines Software-Projekts stellt sich damit als komplexes und facettenreiches Konstrukt dar, dessen Kriterien nicht immer einfach und objektiv zu erheben sind. Die Kriterien sind nicht unabhängig voneinander, sondern konkurrieren in der Praxis, wie sich etwa beim traditionellen Konflikt zwischen Termintreue und Anforderungserfüllung zeigt. Erschwerend kommt hinzu, dass – zumindest in industriellen Projekten – die Gewichtung der oben genannten sechs Kriterien durch die an einem Projekt beteiligten Parteien (beispielsweise Entwickler, Projekteigentümer oder Kunden) ganz unterschiedlich ausfallen kann. Ob das Projektteam seine Kenntnisse und Fähigkeiten erweitert hat, stellt für den Kunden kein Kriterium für die Beurteilung des Projekterfolgs dar. Umgekehrt spielt – zumindest bei einem Auftragsprojekt [Ludewig/Lichter 2007, S. 91] – der tatsächliche Einsatz des entstandenen Produkts beim Kunden für die Entwickler vermutlich eine untergeordnete Rolle, wenn diese nicht gleichzeitig an Betrieb und Wartung beteiligt sind.

Die Gewichtung und Priorisierung der genannten Erfolgsfaktoren in studentischen Projekten unterscheidet sich in aller Regel von der Gewichtung und Priorisierung in industriellen Projekten. Dies hat zwangsläufig auch Einfluss auf die Realitätsnähe studentischer Projekte.

2.2 Realitätsnähe

Von studentischen Projekten wird regelmäßig gefordert, dass sie realitätsnah sein sollen. Die Realität ist in diesem Fall die Praxissituation, in der Projekte mit realen Kunden für reales Geld durchgeführt werden. Realitätsnähe ist somit ein Maß für die Ähnlichkeit, die ein studentisches Software-Projekt zu Projekten in der Industrie besitzt. Die Ähnlichkeit bezieht sich sowohl auf die inhaltliche Zielsetzung des Projekts – es soll ein funktionsfähiges Produkt entstehen – als auch auf Rahmenbedingungen wie Umfang, Dauer und Teamgröße. Es ist klar, dass der Vergleichsgegenstand zwangsläufig Fiktion bleibt – das »typische Projekt« in der Industrie existiert in dieser Form nicht. Projekte divergieren gerade in Hinblick auf Umfang und Dauer erheblich. Trotzdem lassen sich bestimmte Eigenschaften industrieller Projekte identifizieren. Diese mögen zwar in der Praxis nicht in jedem Einzelfall vorhanden sein, können aber im Allgemeinen doch häufig angetroffen und somit als typisch für industrielle Projekte angenommen werden. Studentische Projekte sind in diesem Sinne umso realistischer, je eher sie diese Eigenschaften aufweisen. Typische Merkmale industrieller Projekte sind:

- Die Projekte haben eine gewisse Größe, sowohl hinsichtlich des budgetierten Aufwands als auch hinsichtlich der Projektdauer.
- Es existiert ein (zumeist unternehmensexterner) Kunde, der die Entwicklung des Produkts in Auftrag gegeben und Interesse am Einsatz des Produkts hat.
- Bei der Durchführung des Projekts müssen bestimmte verbindlich zugesicherte Termine eingehalten werden; Terminverschiebungen verursachen Kosten (nicht zwangsläufig monetärer Art).
- Die Bewertung des Projekterfolgs erfolgt durch den Kunden und ist zu jeder Zeit, nicht nur zum Ende der Projektlaufzeit, möglich. In der Konsequenz ist auch ein Abbruch des Projekts zu jeder Zeit möglich.

Studienprojekte weisen diese Merkmale in unterschiedlichem Grade auf und unterscheiden sich demzufolge in ihrer spezifischen Realitätsnähe. Zudem gibt es Aspekte, in denen sich Studienprojekte konstruktionsbedingt von Software-Projekten in der Industrie unterscheiden:

- Den studentischen Teilnehmern des Studienprojekts stehen Betreuer zur Seite. Diese Funktion wird üblicherweise von wissenschaftlichen Mitarbeitern wahrgenommen und hat in industriellen Projekten kein Pendant.

- Im Projektteam existiert keine natürliche Hierarchie; das Projektteam muss durch Wahl des Projektleiters eigenständig eine hierarchische Struktur schaffen. In industriellen Projekten ist die Hierarchie vorgegeben.
- Die studentischen Teilnehmer erhalten nach Abschluss des Projekts eine Note.

Weitere – ja nach Studienprojekt unterschiedlich stark ausgeprägte – Unterschiede zu industriellen Software-Projekten werden im folgenden Kapitel behandelt.

3 Erfolgreiche vs. realistische Studienprojekte

In der Abteilung Software Engineering des Instituts für Softwaretechnologie der Universität Stuttgart wurde seit 2003 eine Reihe von Studienprojekten[1] durchgeführt, in denen interaktive Softwaresysteme entwickelt wurden. Diese Projekte waren unterschiedlich erfolgreich und realistisch.

3.1 Erfolgreiche Studienprojekte in der Vergangenheit

Zwischen 2003 und 2007 wurden drei Studienprojekte durchgeführt:

- Im Studienprojekt *JUST* wurde ein webbasiertes Informationssystem zur Verwaltung von Anforderungen entwickelt. Das Projekt wurde von zehn studentischen Projektteilnehmern in Kooperation mit zwei externen Kunden aus der Software- und der Finanzbranche durchgeführt [Hampp et al. 2005]. Es hatte eine Laufzeit von zwölf Monaten; der Aufwand betrug 33 Personenmonate.
- Ergebnis des Studienprojekts *KoPrA* war ein webbasiertes Werkzeug zur Modellierung und Verwaltung von Prozessen. Kooperationspartner war ein Unternehmen aus der Finanzbranche. Das Projektteam bestand aus sechs studentischen Teilnehmern. Das Projekt dauerte elf Monate und hatte einen Gesamtaufwand von 22 Personenmonaten.
- Im Studienprojekt *CodeCover* entstand in 13 Monaten in Zusammenarbeit mit einem Partner aus der Finanzbranche ein Werkzeug zur Überdeckungsmessung im Glass-Box-Test. Durchgeführt wurde das Projekt von zehn studentischen Projektteilnehmern mit einem Aufwand von 27 Personenmonaten.

Bei der Bewertung des Projekterfolgs dieser Projekte stand die studentische Sicht im Mittelpunkt. Einigen der in Kapitel 2 genannten Kriterien wurde eine höhere Bedeutung zugemessen, andere wurden als weniger wichtig erachtet. Daraus resultierten die folgenden Erfolgskriterien, sortiert nach absteigender Wichtigkeit:

1 Weitere Informationen zu den durchgeführten Studienprojekten finden sich unter *http://www.iste.uni-stuttgart.de/se/research/*.

- Die Studierenden haben Kompetenzen im Bereich der Durchführung und des Managements von Software-Projekten erworben.
- Der vereinbarte Funktionsumfang des Software-Produkts ist weitgehend vorhanden.
- Die demonstrierte Qualität des Software-Produkts erscheint ausreichend.
- Aufwand und Termine wurden weitgehend eingehalten.

Im Sinne dieser Erfolgsdefinition waren alle drei Projekte sehr erfolgreich. Der direkte Vergleich mit den in Kapitel 2 genannten Erfolgskriterien zeigt, dass der tatsächliche Einsatz des entstandenen Produkts und die Zufriedenheit des Kunden keine wesentliche Rolle spielten. Die geringe Bedeutung dieser beiden eng mit der Rolle des Kunden verknüpften Erfolgskriterien scheint zunächst im Widerspruch zu stehen mit der etwa in [Hampp et al. 2005] beschriebenen durchaus als positiv einzustufenden Mitwirkung externer Kunden in Studienprojekten. Hier gilt es jedoch zu bedenken, dass die Entscheidung über den Einsatz des Produkts jeweils »kundenintern« und erst nach Abschluss des Projekts bzw. nach Abnahme des Produkts durch den externen Kunden fiel und somit tatsächlich keine Auswirkungen auf den Projekterfolg aus Sicht der Studierenden und Betreuer hatte. Zudem bewirkte schon die bloße Aussicht auf einen möglichen späteren Einsatz des Produkts durch den externen Kunden eine hohe zusätzliche Motivation der studentischen Projektteilnehmer.

Dieses eingeschränkte Erfolgsverständnis machte die Projekte jedoch zwangsläufig weniger realistisch. Zwar war die Größe der Projekte durchaus industrietypisch, ebenso existierte ein externer Kunde. Es fehlten jedoch in letzter Konsequenz verbindliche Terminvorgaben, und es bestand zu keiner Zeit die ernsthafte Gefahr des Scheiterns des Projekts.

3.2 Realistisch, aber erfolglos: das aktuelle Studienprojekt

Seit Frühjahr 2008 wird in der Abteilung Software Engineering ein weiteres Studienprojekt mit zwölf Teilnehmern und einer geplanten Laufzeit von zwölf Monaten durchgeführt. Aufgabe des Projekts ist die Entwicklung eines webbasierten Konferenzmanagementsystems. Im Unterschied zu den im vorigen Abschnitt vorgestellten Studienprojekten stand bei diesem Projekt von Beginn an der angestrebte Einsatz des entwickelten Produkts durch den externen Kunden im Vordergrund. Die Entscheidung für oder gegen den Einsatz des Produkts wurde vom Kunden bereits während der Projektlaufzeit getroffen: Der Kunde entschied sich, das Produkt nicht einzusetzen. Der Termin für die Entscheidung wurde vom Kunden vorgegeben und war nicht verhandelbar. Gleichzeitig musste bis zu diesem Termin eine bestimmte Mindestfunktionalität in akzeptabler Qualität bereitgestellt werden, was nicht gelang. Diese Rahmenbedingungen machten das Projekt deutlich realistischer als die bisherigen Studienprojekte.

In der Konsequenz änderte sich auch das Erfolgsverständnis. Die früher tendenziell weniger beachteten Erfolgskriterien Produkteinsatz und Kundenzufriedenheit traten in den Vordergrund. Gemessen an diesen Erfolgskriterien ist das Studienprojekt gescheitert. Diese Erfahrung hat uns veranlasst, erneut über das Verhältnis von Realitätsnähe und Erfolg nachzudenken.

4 Erfolg oder Realitätsnähe: Prioritäten in Studienprojekten

Wie in Kapitel 2 angeführt, stehen Erfolg und Realitätsnähe studentischer Projekte in verschiedener Hinsicht in Konkurrenz zueinander. Wir halten es nicht für möglich, entsprechende Projekte sowohl stets erfolgreich als auch vollständig realistisch zu konzipieren. Somit stellt sich die Frage, welchem der beiden Ziele welches Gewicht zugemessen werden soll. Zur Beantwortung dieser Frage soll zunächst herausgearbeitet werden, welche Auswirkungen das Erreichen bzw. das Nichterreichen dieser Ziele hat. Anschließend werden die Ziele gewichtet.

4.1 Auswirkungen von Erfolg

In aller Regel ziehen Studierende aus dem praktischen Einsatz der von ihnen im Rahmen von Studienprojekten entwickelten Software-Lösungen keinen direkten monetären Nutzen. Dieser Zustand ist ohne Zweifel verbesserungswürdig, soll aber an dieser Stelle nicht behandelt werden. Ein indirekter Nutzen durch eine Tätigkeit bei dem die Software-Lösung einsetzenden externen Kunden ist hingegen durchaus möglich.

Schwierig einzuschätzen ist die Bedeutung des Erfolgs in Hinblick z.B. auf die Bewerbung bei Softwarefirmen nach dem Studium. Generell gilt die bereits vorhandene Projekterfahrung als eine der großen Stärken der Absolventen unseres Studiengangs (vgl. auch [Röder 2007]). Die Zusammenarbeit mit einem externen Kunden und dessen (mutmaßlicher) Einsatz der entwickelten Software stellen auf jeden Fall einen Pluspunkt dar; Gleiches gilt aber z.B. auch für die Entwicklung eines praktisch eingesetzten Open-Source-Werkzeugs.

Die gewichtigste Auswirkung des Erfolgs eines Studienprojekts ist sicher der langfristige und positive Effekt auf das Selbstbewusstsein und die Methodenkompetenz der studentischen Teilnehmer. Zum einen wirkt das positive Feedback auf die erbrachte eigene Leistung bestätigend. Zum anderen wird den Studierenden gezeigt, dass sie mit Hilfe der in den Lehrveranstaltungen vermittelten und im Studienprojekt eingesetzten Methoden des Software Engineering in der Lage sind, die bei der Durchführung realistischer Projekte zwangsläufig auftretenden Schwierigkeiten zu lösen.

Dies zeigt auch, dass zumindest manche positiven Auswirkungen des Erfolgs durch eine große Realitätsnähe verstärkt und durch eine geringe Realitätsnähe geschwächt werden. So könnten Studierende bei einem wenig realistischen, aber

erfolgreichen Projekt zu der Auffassung gelangen, dass die an der Hochschule vermittelten Methoden nur fernab der Praxis funktionieren. Bei einem Scheitern wird die Effektivität der erlernten Methoden generell in Zweifel gezogen und zusätzlich das Selbstbewusstsein der Studierenden geschwächt. Erfolgt eine negative Einsatzentscheidung des externen Kunden während der Projektlaufzeit und wurde der Projekterfolg implizit oder explizit am Produkteinsatz festgemacht, ist das Resultat oft eine länger anhaltende Motivationskrise des Projektteams; diese Auswirkung ließ sich auch an dem in Kapitel 3.2 beschriebenen erfolgslosen Studienprojekt beobachten.

4.2 Auswirkungen von Realitätsnähe

Wie bereits im vorigen Abschnitt beschrieben, verstärkt eine größere Realitätsnähe bei einem erfolgreichen Projekt die positiven Auswirkungen des Erfolgs. Es ist deutlich schwieriger, die positiven Auswirkungen einer allgemein größeren Realitätsnähe zu bestimmen. Zwar erhöht das Vorhandensein eines externen Kunden sicher die Motivation der Studierenden, ein brauchbares Produkt zu entwickeln. Dieser Motivationsschub ist jedoch in erster Linie dem Vorhandensein eines »echten« Interessenten geschuldet und weniger der dadurch bewirkten größeren Realitätsnähe.

Auf den Studiengang Softwaretechnik bezogen erhöhen realistischere Projekte insgesamt die Wahrnehmung des Studiengangs in der Industrie als praxisorientiert. Für das einzelne Studienprojekt wirkt eine größere Realitätsnähe unter Umständen lindernd in Hinblick auf den »Praxisschock», den die Studierenden möglicherweise beim Einstieg ins Berufsleben erleiden.

Diesen Argumenten für eine größere Realitätsnähe muss jedoch entgegengehalten werden, dass die Hochschule ein berechtigtes Interesse daran hat, ihren Studierenden im Rahmen der Ausbildung auch Methoden zu vermitteln, die (noch) nicht überall Stand der Praxis sind. Weiterhin ist zu beachten, dass bestimmte realistische Rahmenbedingungen im Rahmen der Ausbildung weder umsetzbar noch gewünscht sind; dazu zählen z.B. eine hohe Fluktuation der Projektmitarbeiter, Budgetrestriktionen, unternehmerisches Risiko der Organisation und das Vorhandensein echter Hierarchien im Projektteam. Insgesamt sind der erreichbaren Realitätsnähe in Studienprojekten damit enge Grenzen gesetzt.

4.3 Gewichtung von Erfolg und Realitätsnähe

Aus der Betrachtung der Auswirkungen der Ziele Erfolg und Realitätsnähe ergeben sich drei wesentliche Aussagen:

1. **Erfolg ist maßgeblich.**
   Auch unabhängig von der Realitätsnähe hat der Erfolg in Studienprojekten positive Auswirkungen.
2. **Realitätsnähe wirkt im Erfolgsfall positiv.**
   Einige der positiven Auswirkungen erfolgreicher Studienprojekte werden durch eine größere Realitätsnähe verstärkt.
3. **Realitätsnähe allein ist nicht wesentlich.**
   Eine möglichst große Realitätsnähe ist kein Wert an sich, sondern gewinnt diesen Wert nur durch die Verstärkung anderer Auswirkungen und durch einzelne Maßnahmen, die auch zur Realitätsnähe beitragen.

Daraus lassen sich unmittelbar zwei sinnvolle Grundregeln für die Gewichtung der obengenannten Ziele Erfolg und Realitätsnähe in Studienprojekten ableiten:

1. Die Erfolgsaussichten sind zu maximieren, ohne die Realitätsnähe völlig aus dem Blick zu verlieren.
2. Eine hohe Realitätsnähe ist anzustreben, sofern dadurch die Erfolgsaussichten nicht (wesentlich) beeinträchtigt werden.

Beide Regeln sind nicht schrankenlos, sondern müssen in jedem Einzelfall für jeden Faktor, der eines der Ziele Erfolg und Realitätsnähe auf Kosten des jeweils anderen zu erreichen versucht, neu abgewogen werden.

5 Empfehlungen für studentische Projekte

Aus den Erfahrungen der in den vergangenen Jahren am Lehrstuhl durchgeführten Studienprojekte und der in den vorigen Kapiteln vorgestellten Überlegungen lassen sich Empfehlungen zur Erreichung der anzustrebenden Realitätsnähe und zum Umgang mit einem externen Kunden ableiten.

5.1 Gewährleistung einer optimalen Realitätsnähe

In Kapitel 4 wurde dargelegt, dass in studentischen Projekten zwar eine möglichst hohe Realitätsnähe angestrebt werden sollte, diese aber den Erfolg des Projekts nicht gefährden darf. Hierzu soll im Folgenden ein geeigneter Ansatz vorgestellt werden.

Wie beschrieben ist die Realitätsnähe ein Maß für die Ähnlichkeit, die ein studentisches Software-Projekt zu Projekten in der Industrie besitzt. Die Ähnlichkeit kann sich dabei auf die Rahmenbedingungen des Projekts, die inhaltlichen Ziele und das Vorgehen im Projekt beziehen. Unsere Hypothese ist, dass Projekteigenschaften wie z.B. ein mit typischen industriellen Projekten vergleichbarer Umfang sich einerseits auf die Realitätsnähe und andererseits auf die Erfolgsaussichten eines studentischen Projekts auswirken und dass eine positive Wirkung

auf die Realitätsnähe nicht zwangsläufig mit einer negativen Wirkung auf die Erfolgsaussichten korreliert (und umgekehrt). Durch eine gezielte Auswahl von Eigenschaften, die die Realitätsnähe verstärken, ohne die Erfolgsaussichten zu senken, sollte es möglich sein, studentische Projekte so zu gestalten, dass eine hohe Realitätsnähe bei gleichzeitig maximalen Erfolgsaussichten erreicht wird. Beispiele für Eigenschaften, die in studentischen Projekten die Realitätsnähe verstärken, ohne die Erfolgsaussichten (in Hinblick auf den Einsatz des Software-Produkts beim Kunden) wesentlich zu senken, sind aus unserer Sicht:

- Feste Endtermine
- Kostenbeschränkungen (in Form einer festen Teamgröße)
- Erhebung der Anforderungen beim Kunden, deren Prüfung und Priorisierung
- Eingeschränkte Verfügbarkeit des Kunden

Der Schlüssel zum richtigen Umgang mit den sich aus diesen Eigenschaften ergebenden Schwierigkeiten liegt in gutem Software Engineering. Bei einer Umsetzung der vorher in Lehrveranstaltungen vermittelten Methoden, etwa sorgfältiger Projektplanung, wirken sich diese Eigenschaften nicht negativ auf die Erfolgsaussichten aus.

Dagegen haben die folgenden beiden Eigenschaften auch bei gutem Software Engineering negativen Einfluss auf die Erfolgsaussichten des Projekts:

- Unsinnige Projektvorgaben durch Management oder Vertrieb (z.B. unrealistische Vorgaben für Termine und Leistungen)
- Entscheidungen über den Einsatz des entwickelnden Software-Produkts während der Projektlaufzeit

Studentische Projekte sollten daher auf diese Eigenschaften verzichten, auch wenn durch das Vorhandensein dieser Eigenschaften die Realitätsnähe weiter erhöht werden könnte.

5.2 Umgang mit externen Kunden

Ein externer Kunde erhöht zweifellos die Motivation der Studierenden und ist somit zu begrüßen. Allerdings ist seine Beteiligung am Projekt so zu gestalten, dass die Erfolgsaussichten des Projekts nicht gefährdet werden. Dies kann z.B. durch die Verlagerung der Einsatzentscheidung auf eine Zeit nach Projektende geschehen. Eine gleichzeitige Festlegung sowohl zwingend umzusetzender Funktionalität als auch harter Terminvorgaben ist zu vermeiden, insbesondere wenn es Zweifel an der Erreichbarkeit dieser Ziele gibt. Die negativen Auswirkungen auf die Motivation der Studierenden übersteigen im Falle eines Scheiterns die vorherigen positiven Auswirkungen auf Realitätsnähe und Motivation.

Um die Chancen auf einen Einsatz des entwickelten Systems bei einem externen Kunden zu erhöhen, sollten die Betreuer auf Lehrstuhlseite die – in aller

Regel noch über keine Projekterfahrung verfügenden – Studierenden bei der Kommunikation mit dem Kunden unterstützen, auch wenn dies mit einer verringerten Realitätsnähe einhergeht. Ansonsten ist die Gefahr groß, dass zu wenig Kommunikation mit dem Kunden stattfindet oder wesentliche Anforderungen verloren gehen.

Präsentationen und Demonstrationen des entwickelten Systems sind mit Vorsicht zu genießen und im Zweifelsfall zeitlich eher weiter nach hinten zu schieben. Es mag realistisch sein, dass Einsatzentscheidungen auf Basis halbfertiger Prototypen getroffen werden; es fördert aber nicht den Erfolg, wenn diese Prototypen das implizit gegebene Versprechen der baldigen Einsetzbarkeit nicht einlösen können.

6 Fazit

In diesem Artikel haben wir dargelegt, dass bei der Konzeption studentischer Projekte erfolgreiche Projekte realistischen, aber erfolglosen Projekten vorzuziehen sind. Dahinter steht nicht zuletzt die Überzeugung, dass in der akademischen Ausbildung nicht die Mängel der Praxis nachgeahmt, sondern stattdessen funktionierende – und erfolgreiche – Lösungen vorgestellt werden sollen. Studentische Projekte sollen als Vorbild für Projekte in der Praxis taugen, weniger als ihr Abbild.

Wir fordern, dass studentische Projekte zunächst in der Regel erfolgreich sein sollten. Denn wenn die Studierenden im Studium die Erfahrung gemacht haben, dass die Anwendung der erlernten Normen, Techniken und Mittel zum Erfolg führt, werden sie wahrscheinlich auch nach Eintritt ins Berufsleben so verfahren. Die oft geäußerte Meinung, die Studierenden sollten ruhig scheitern, das sei doch sehr lehrreich, teilen wir nicht. Wer weiß, dass ein bestimmter Weg nicht zum Ziel führt, hat dadurch nicht gelernt, wie man zum Ziel kommt. Wenn wir sagen, dass die Projekte »in der Regel« erfolgreich sein sollten, dann bedeutet die Einschränkung, dass die Möglichkeit des Scheiterns natürlich nicht ausgeschlossen werden darf, schon wegen der Realitätsnähe. Aber die Projektteilnehmer sollten nur scheitern, wenn es ihnen nicht gelingt, das Gelernte zu praktizieren, und wenn sie Hilfen und Ratschlägen nicht zugänglich sind.

Das Primat des Projekterfolgs bewirkt zwangsläufig eine Einschränkung der Realitätsnähe. Diese Einschränkung ist jedoch mit Blick auf die positiven Auswirkungen erfolgreicher Projekte gerechtfertigt. Eine hohe Realitätsnähe unter Einbezug derjenigen Projekteigenschaften, die den Projekterfolg nicht wesentlich beeinträchtigen, ist in jedem Fall anzustreben.

Literatur

[CHAOS 1995] Standish Group International: The CHAOS Report. 1995.

[Fahl 2003] W. Fahl: Anforderungsmanagement als Erfolgsfaktor der Software-Entwicklung. Softwaretechnik-Trends, 23 (1), 2003.

[Hampp et al. 2005] T. Hampp, S. Opferkuch., R. Schmidberger: Projekte der Lehre mit hochschul-externen Kunden. In: K.-P. Löhr, H. Lichter (Hrsg.): Software Engineering im Unterricht der Hochschulen (SEUH 9 – Aachen 2005). dpunkt.verlag, Heidelberg, 2005.

[Ludewig 1999] J. Ludewig: Softwaretechnik in Stuttgart – ein konstruktiver Informatik-Studiengang. Informatik-Spektrum, 22 (1). Springer, Berlin, 1999.

[Ludewig 2003] J. Ludewig: Zur Entwicklung des Studiengangs Softwaretechnik. In: L. Hieber (Hrsg.): Die Geschichte der Stuttgarter Informatik. Informatik-Forum Stuttgart e.V., 2003.

[Ludewig/Lichter 2007] J. Ludewig, H. Lichter: Software Engineering. dpunkt.verlag, Heidelberg, 2007.

[Röder 2007] H. Röder: Die Ausbildung der Software-Ingenieure in Indien und Deutschland – ein Vergleich. In: A. Zeller, M. Deininger (Hrsg.): Software Engineering im Unterricht der Hochschulen (SEUH 10 – Stuttgart 2007). dpunkt.verlag, Heidelberg, 2007.

[Wetzel 2007] M. Wetzel: Ein Experiment zur Ermittlung der Programmierproduktivität von Studenten. In: A. Zeller, M. Deininger (Hrsg.): Software Engineering im Unterricht der Hochschulen (SEUH 10 – Stuttgart 2007). dpunkt.verlag, Heidelberg, 2007.

Irrwege und Wegweiser im Praktikum der frühen Softwareentwicklungsphasen

Robert Garmann
Fachhochschule Stralsund
robert.garmann@fh-stralsund.de

Zusammenfassung

Studentische Irrwege beim Erwerb von Fertigkeiten in Analyse und Entwurf von Softwaresystemen sind zeitraubend. Andererseits wirken Fehler verständnisfördernd. Dieser Beitrag beschreibt Erfahrungen mit dem selbstgesteckten Ziel einer Balance zwischen einerseits ausreichendem Raum für Irrwege und andererseits genügendem Halt durch wegweisende Instruktionen des Betreuers. Wir beschreiben vier sich gegenseitig unterstützende Maßnahmen: Softwareentwicklung als Arbeit im Steinbruch, L-förmige Verzahnung von Theorie und Praxis, das Vorgehen »code first« sowie zuletzt einen Vorschlag zum vermittelnden Brückenschlag zwischen dem Vorgehen von Lernenden und Praktikern.

1 Einleitung

1.1 Didaktisches Ziel

Angesichts der Bestätigung durch unzählige SEUH-Beiträge ist es vermutlich nicht zu gewagt zu behaupten, dass Softwareentwicklung handlungsorientiert vermittelt werden muss, um überhaupt richtig verstanden werden zu können. Am besten geschieht das in einem im Umfang der Berufsrealität nahe kommenden Softwareprojekt. Wir betrachten in diesem Beitrag ein Praktikum der frühen Phasen der Softwarentwicklung im Bachelorstudium der Informatik. Idealerweise würde jedem teilnehmenden Studierenden ein Coach zur Seite gestellt, der bei jedweder Tätigkeit zeitnahes und kompetentes Feedback gibt. Hat man das nötige Personal in diesem Maßstab nicht zur Verfügung, besteht die Gefahr, dass

Studierende sich zeitweise »verrennen« – Irrwege beschreiten. Bis zu einem gewissen Grad ist das gewollt. Sackgassen beim Lernen können lehrreich sein, weil die Auseinandersetzung mit Fehlern verständnisfördernd wirkt. Leider sind Sackgassen und Irrwege zeitintensiv.

Um Zeit zu sparen, könnte man den Lernenden einen Idealweg vorzeichnen. Beispielsweise ein über das Semester durchzuarbeitendes Tutorial würde die Gefahr von Fehlern minimieren, und das bei immer noch vorhandener handlungsorientierter Unterrichtsweise. Hier liegt der Nachteil im geringen geistigen Aktivitätslevel der Lernenden, die in vergleichsweise passiver Weise vorgegebene Schritte praktisch umsetzen. Niemand garantiert uns, dass die Studierenden verstehen, was sie da praktisch umsetzen.

Die Lösungsvorschläge dieses Beitrags versuchen, ein Gleichgewicht zwischen Wegweisung und Laufenlassen zu finden. Individuelle Lernstile der Studierenden sollen dabei Berücksichtigung finden. Anders ausgedrückt: Wir suchen nach einer Balance zwischen konstruktivistischen und objektivistischen Lehrformen. Im vorliegenden Beitrag werden Vorschläge unterbreitet, wie eine solche Balance im genannten Praktikum aussehen könnte. Die beschriebenen Erfahrungen werden überwiegend nicht quantifiziert. Der vorliegende Beitrag soll qualitative Erfahrungen zweier im Sommer 2007 und 2008 durchgeführter Praktika zur Diskussion stellen.

1.2 Lehrveranstaltung

Im Curriculum des Bachelorstudienganges Informatik der Fachhochschule Stralsund wird im vierten Semester das Modul »Software Engineering II« (kurz SE2) angeboten (vgl. Abb. 1). Das Modul im Umfang von 5 ECTS-Punkten wird als Projektseminar durchgeführt und fokussiert auf die praktische Entwicklung von Software unter Berücksichtigung der Methodik des Software Engineering. Schwerpunkte liegen auf den frühen Phasen der Entwicklung: Anforderungen, Analyse und Entwurf. Der Entwurf wird durch die Implementierung erprobt.

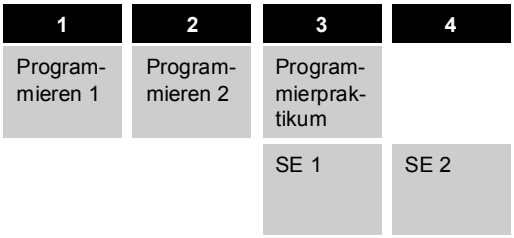


Abb. 1 *Ausschnitt aus dem Curriculum (1.-4. Semester, 5 ECTS-Punkte je Modul)*

Die Studierenden haben in einem zeitlich vorgelagerten Modul SE1 in Vorlesungen und Übungen Kenntnisse und erste Fertigkeiten zu den Themen Vorgehensmodelle, Requirements, OOA, OOD, UML, Softwarearchitektur und Qualitäts-

sicherung erworben. Die Programmier-Grundausbildung (C, C++, C#) ist bis einschließlich des dritten Semesters abgeschlossen.

Im auf den folgenden Seiten ausführlicher betrachteten Modul SE2 entwickeln die Studierenden ausgehend von einer groben Anforderungsbeschreibung über einen Zeitraum von 15 Wochen eine datenbankbasierte Anwendung mit grafischer Benutzeroberfläche und betrieblichem Fachhintergrund. Die jährliche Teilnehmerzahl des Moduls SE2 bewegt sich zwischen 25 und 40. Lernziele sind u.a.:

- Die Studierenden können präzise und widerspruchsfreie Anforderungen formulieren. Bei der Analyse entwickeln sie ein Gespür für Inkonsistenzen und Lücken in vom Anforderer formulierten Wünschen.
- Die Studierenden kennen die Vielseitigkeit und den Nutzen grundlegender Entwurfsprinzipien. Durch vielfache praktische Anwendung lernen sie, diese Prinzipien selbst beim Softwarentwurf einzusetzen.
- Die Studierenden können den Entwurf einer Software mit abstrakten Mitteln beschreiben (UML).

Nicht zu den Lernzielen gehört der Erwerb von Fertigkeiten in der systematischen Qualitätssicherung bzw. in der Projektorganisation. Diese Themen sind Gegenstand weiterer Veranstaltungen im 6. bzw. 7. Semester.

Die Studierenden arbeiten in Teams von je 3 bis 4 Teilnehmern, da diese Lernsituation das Training von Soft-Skills ermöglicht.

1.3 Überblick über diesen Beitrag

In diesem Beitrag beschreiben wir Erfahrungen und sich gegenseitig ergänzende Lösungsvorschläge für vier Aspekte. Kapitel 2.1 stellt unser Modell der zeitlichen Verschränkung von Theorie und Praxis dar. Kapitel 3 beschreibt Erfahrungen mit der Ermunterung der Studierenden, eine vorgegebene Beispielanwendung als Steinbruch für eigene Ideen zu nutzen. Gegenstand des Kapitels 4 ist der Vorschlag, Studierende sich den Entwurf »von unten«, d.h. vom Programmcode her, erarbeiten zu lassen. Das Kapitel 5 schlägt eine Möglichkeit zur Einordnung des lernenden Vorgehens in das Vorgehen eines Praktikers vor.

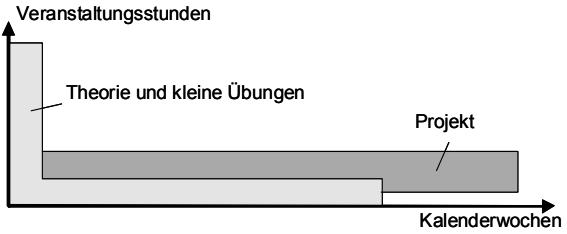


Abb. 2 *L-förmige Verzahnung von Theorie und Praktikum*

2 L-förmige Verzahnung von Theorie und Praxis

2.1 Das Problem

Damit Studierende ein Softwarentwicklungsprojekt durchführen können, müssen sie bereits am Anfang Wissen über Anforderungsanalyse und Entwurf besitzen (s. auch z.B. [Lindig/Zeller 2005]). Gegen ein einfaches zeitparalleles Angebot der Module SE1 (Theorie mit kleinen Übungen) und SE2 (Projekt) spricht, dass die Vermittlung des theoretischen Wissens einem anderen Zeitraster folgt, als es für die Arbeit im Projekt benötigt wird.

Eine Alternative ist, das Theorie-Modul ein Semester vor dem Projekt-Modul anzubieten (s. Abb. 1 oben). In früheren Jahrgängen wurde das bei uns praktiziert. Dabei war zu beobachten, dass die in SE1 im 3. Semester erworbenen Kenntnisse über die Semesterferien so stark verblasst waren, dass im 4. Semester in SE2 viel Zeit in eine Auffrischung des Theorie- und Faktenwissens investiert werden musste.

2.2 Lösungsvorschlag

Die Vorlesungszeit im Bachelorstudiengang Informatik der Fachhochschule Stralsund gliedert sich in eine Blockphase von in der Regel einer Woche und in eine anschließende 15-wöchige Phase regulären Veranstaltungsbetriebes. In der Blockphase wird ein großer Anteil der Inhalte des Moduls SE1 gelehrt. Nach der Blockphase wird SE1 mit 4 Veranstaltungsstunden pro Woche weitergeführt und endet schließlich etwa zur Mitte der Vorlesungszeit. Das Modul SE2 beginnt erst nach der Blockphase und endet mit der Vorlesungszeit. Abbildung 2 skizziert den Ablauf.

In unserem Informatik-Curriculum kann zu Beginn des 3. Semesters noch nicht mit hinreichenden Fertigkeiten in der objektorientierten Programmierung seitens der Studierenden gerechnet werden. Die Studierenden kommen erst im 3. Semester durch ein umfangreiches Programmierpraktikum mit etwas komplexeren Programmieraufgaben in Berührung (s. Abb. 1). Ohne diese Erfahrungen sehen die Studierenden nur schwer ein, dass Software-Engineering-Methoden ihre Berechtigung besitzen. Auch aus diesem Grund haben wir uns entschieden, SE1 erst nach dem Programmierpraktikum beginnen zu lassen[1].

2.3 Erfahrungen

Mit der beschriebenen Blockphase haben wir positive und negative Erfahrungen gemacht. Positiv ist, dass die Studierenden den in SE1 frisch erlernten Stoff prä-

1 Eine entsprechende Anpassung des offiziellen Curriculums ist geplant.

sent haben, wenn sie an die Bearbeitung des Projektes in SE2 gehen. Studentische Irrwege werden nun seltener als zuvor beobachtet. Als günstig hat sich außerdem der größere zeitliche Spielraum für SE2 zum Ende der Vorlesungszeit erwiesen. Ein SE-Projekt benötigt nach unserer Erfahrung gerade in der Endphase mehr Zeit als 4 Wochenstunden, so dass die beschriebene L-förmige Verzahnung als besonders geeignet für ein SE-Projekt gelten kann.

Negative Erfahrungen betreffen die Blockphase. Hier hatten wir bisher versucht, möglichst viel Stoff unterzubringen, um die Studierenden optimal auf das nachfolgende Projekt vorzubereiten. Die hohe zeitliche Belastung der Studierenden von je 8 Veranstaltungsstunden (4 Vorlesungen, 4 Übungen) über 5 Tage wollen wir in kommenden Jahren reduzieren und umverteilen, da sich gezeigt hat, dass zu wenig Zeit für Experimente und Selbststudium blieb. Im Sommersemester 2009 planen wir für die Blockwoche voraussichtlich insgesamt 32 Veranstaltungsstunden ein, wovon weniger als die Hälfte als Vorlesung gestaltet wird. Als Folge werden wir uns intensiver mit einer zeitlichen Abstimmung der jeweiligen, von verschiedenen Lehrenden beigesteuerten Veranstaltungsinhalte beschäftigen müssen, um sicherzustellen, dass Theoriewissen vor dem praktischen Einsatz vermittelt wird. Einige Themen (z.B. Dialogentwurf) werden bei dieser Neuordnung erst nach dem ersten praktischen SE2-Einsatz theoretisch in SE1 gelehrt werden und im weiteren Semesterverlauf wiederum ein zweites Mal in SE2 bei der Umsetzung weiterer Anwendungfälle praktisch eingesetzt werden (vgl. Spiral-Muster [Bergin 2007]).

3 Softwareentwicklung als Arbeit im Steinbruch

3.1 Das Problem

Erfahrungsgemäß geraten die ersten Entwürfe vieler Studierender mangelhaft, weil die Erfahrungen mit Entwurfsprinzipien fehlen. Verschiedene Analyse- und Entwurfsmuster sind den Studierenden als Fakten bekannt, sie besitzen jedoch nicht die Fertigkeit, die Muster problembezogen einzusetzen.

Wurden unsere Studierenden früherer Studienjahrgänge mit ihrem erworbenen Faktenwissen und einer Aufgabenstellung allein gelassen, so kam es immer mal wieder vor, dass ein Team nach mehreren Wochen enthusiastischer Arbeit einen völlig ungeeigneten Entwurf vorlegte. Dieser ließ sich in der Regel kaum inkrementell in vernünftiger Zeit korrigieren. Ein Neuanfang und Frust waren die Folge. Und nicht zuletzt verursachten Erläuterungen des Betreuers, warum der vorliegende Entwurf ungeeignet sei, erhebliche Betreuungsaufwände.

Das Lernverhalten vieler Studierender lässt sich in folgende drei Gruppen klassifizieren, wobei natürlich Mischformen nicht ausgeschlossen sind:

A. Musterstudierende können die erlernten Entwurfsprinzipien passgenau für ihr Problem anwenden.
B. Andere Studierende suchen im Internet nach Blogeinträgen, die dasselbe Problem beschreiben, vor dem sie selbst gerade stehen. Codefragmente werden zusammen kopiert. Am Ende entsteht ein bunt geknüpfter Flickenteppich verschiedener Entwurfsideen (s. auch »Google-And-Paste-Programming« [GAPP 2008]).
C. Kreative Studierende versuchen sich in der Kunst des Entwurfs, ohne sich an die gelernten Entwurfsmuster erinnern zu wollen. Sie erfinden das Rad lieber neu. Meist mit einem defizitären Ergebnis.

Die Herangehensweisen B und C bedingen ein aufwendiges, individuelles Feedback des Betreuers, um die Studierenden von ihrem Irrweg wieder auf den richtigen Pfad zu weisen.

Die Gruppen A, B, und C lassen sich recht gut mit den Lernstilen Converger (A), Assimilierer (B) und Akkomodierer (C) nach [Kolb 2005] vergleichen. Nach Kolb (und auch nach unserer Erfahrung) sind diese drei Gruppen etwa gleich stark unter Informatikern zu finden.

Eine zweite Beobachtung ist, dass viele Studierende Schwierigkeiten mit der Abgrenzung der Disziplinen Analyse, Entwurf und Implementierung haben. Häufig besteht das Problem nicht darin, dass zu früh zu detailliert voraus gedacht wird, sondern eher zu wenig. Die Analyse gerät in vielen Teams zur Alibiveranstaltung. Hauchdünne Analysedokumente machen es dem Betreuer schwer, alle Lücken aufzuzählen. Erst bei der Implementierung stellen die Studierenden fest, welche Antworten auf nicht gestellte Fragen ihnen nun fehlen. Frust ist häufig die Folge.

3.2 Lösungsvorschlag

Vom Veranstalter wird ein Beispielprogramm mittleren Umfangs bereit gestellt. Das Beispielprogramm verwendet die gleichen Technologien, wie sie von den Studierenden für ihr eigenes Problem eingesetzt werden sollen. Die Anwendungsdomäne ist jedoch deutlich verschieden. Das Beispielprogramm steht im Quelltext zur Verfügung und ist vorbildlich dokumentiert, sowohl im Quelltext als auch durch begleitende Analyse- und Entwurfsdokumente (vgl. auch Muster »Lay of the land« [Bergin 2007] bzw. eine ähnliche Vorgehensweise in [Hasselbring et al. 2007]). Die Idee dahinter besteht darin, dass Studierende, die komplexe Systeme nicht von Grund auf erstellen können, jedoch durchaus (wenn auch zu Beginn nur oberflächlich) in der Lage sind, ein komplexes System zu studieren und zu begreifen.

Das Beispielprogramm enthält Lösungen für möglichst viele Stolperfallen, die den Studenten vermutlich auf ihrem Weg begegnen werden. Um die Studenten zu Beginn nicht zu überfrachten, wird das Beispiel in zwei Ausbaustufen zur Verfü-

gung gestellt (vgl. Spiral-Muster [Bergin 2007]). Die größere Ausbaustufe hat eine mit dem von den Studierenden zu erstellenden Produkt vergleichbare Komplexität in der Tiefe, ist jedoch weniger breit angelegt (geringere Anzahl von Anwendungsfällen).

Das Veranstaltungsskript erläutert die Entwurfsentscheidungen der Beispielanwendung. Die Studierenden werden ermuntert, das Beispielprogramm als Steinbruch zu nutzen. Kopieren von Quelltextpassagen ist ausdrücklich erlaubt. Lernen durch Nachahmung ist ein sehr mächtiges Lernprinzip und bietet sich daher als ein Erfolg versprechender Weg an. Eine Gefahr besteht hier darin, dass die Studierenden Programmcode der Beispielanwendung kopieren, ohne ihn zu verstehen. Verzichteten wir jedoch auf die Beispielanwendung, würden diese Studierenden ebenfalls Programmcode kopieren. Nur nicht unseren eigenen, sondern irgendwelchen ergoogelten Code von vermutlich minderer Qualität, der die Studierenden auf einen Entwurfsirrweg führt. Es muss darauf geachtet werden, dass die Beispielanwendung nicht zu viel »verrät«. Irrwege müssen möglich sein, damit sich in begrenztem Maße Verstärkungen durch Versuch und Irrtum bilden können.

Im zur Verfügung gestellten Beispielprogramm haben die Studierenden immer die Möglichkeit, ihre aktuell durchgeführte Teilaufgabe architektonisch einzuordnen. Weiterhin bietet die Beispielanwendung die Möglichkeit der Einordnung von Tätigkeiten in Disziplinen wie Analyse und Entwurf. Mit den beigefügten Analysedokumenten gibt es konkretes Anschauungsmaterial, das es den Studierenden ermöglicht, den Reifegrad der eigenen Analysedokumente besser einzuschätzen. Das Veranstaltungsskript erläutert, wieso der gezeigte Detailgrad vernünftig ist und wann es sinnvoll wäre, mit leichterem Dokumentationsgepäck zu agieren.

Verwandter Ansatz

In [Pedroni/Meyer 2006] wird ein Ansatz beschrieben, bei dem schon im Erstsemester-Programmierkurs umfangreiche, vorbildlich entworfene und dokumentierte Bibliotheken bereit gestellt werden. Während dort die Nutzung über wohldefinierte Schnittstellen im Vordergrund steht, überwiegt bei unserem Vorgehen die Nachahmung der inneren Struktur. In beiden Fällen wirkt die zur Verfügung gestellte Software strukturierend auf das Denken des Studierenden.

3.3 Erfahrungen

Ein Teil der Studierendengruppe B nimmt das Angebot, nicht mehr alles ergoogeln zu müssen, dankbar an. Dadurch, dass sie sich beim Abschauen durch umfangreiche und die Vorlesungsinhalte von SE1 teilweise wiederholende Begleitdokumentation wühlen müssen, ist ein gewisser Lerneffekt kaum zu vermeiden.

Ebenso gibt es unter den kreativen Studierenden (Gruppe C) viele, die sich bereitwillig Denkanregungen holen und diese dann in ihrem eigenen Programm umsetzen.

Ein Teil der Studierenden jedoch sperrt sich zunächst gegen die Beispielanwendung. Einige scheuen den Aufwand, sich in eine große Menge[2] fremden Codes einzuarbeiten. Andere bringen als Argument, dass das Beispielprogramm sie in ihrer Kreativität einschränken würde. Beide nehmen das Angebot zunächst nicht an und verfolgen ihre oben beschriebenen Strategien B und C.

Nach einer kurzen Anlaufzeit, in der die Studierenden selbst einige lehrreiche Sackgassen ausprobiert haben, beobachten wir jedoch ein Umdenken. Insbesondere wenn die Studierenden merken, dass andere Teams schneller vorankommen, lenken sie häufig ein. Am Ende ist die Resonanz in Befragungen regelmäßig unisono, dass das Beispielprogramm einen tiefen Erkenntnisgewinn beim Einsatz guter Entwurfsprinzipien gebracht hat.

Häufig beobachten wir, dass schlechtere Studierende beim Übernehmen von Quellcode nicht immer sofort im Detail verstehen, weshalb die Beispielanwendung so und nicht anders entworfen wurde. Nach unserer Erfahrung ergibt sich in der Regel erst nach einigen Wochen ein Aha-Effekt, nämlich dann, wenn die Anwendung um neue Funktionen erweitert werden muss.

Nicht unerwähnt soll bleiben, dass schlechtere Studierende einen zu hohen Zeitaufwand beklagen. Einige resignieren frühzeitig angesichts der durchzuarbeitenden Informationsfülle. Der hohe Zeitaufwand resultiert nach unserer Einschätzung u.a. aus einer schlechten Arbeitsteilung, so dass wir uns vornehmen, hier in zukünftigen Semestern früher steuernd in die (Selbst-)Organisation des Teams einzugreifen.

4 Code first

4.1 Das Problem

In früheren Jahrgängen wurde das SE2-Projekt nach dem Wasserfallmodell durchgeführt. Dabei haben wir beobachtet, dass Studierende sich bei der strikten Befolgung der einzelnen Entwicklungsphasen schwer tun. Zum einen wurde die widerspruchsfreie Dokumentation und präzise Analyse von Anforderungen von vielen Teilnehmern nur unwillig und in schlechter Qualität vollzogen. Weiterhin waren viele Studierende mit der Aufgabe überfordert, einen Entwurf für ein später zu implementierendes Programm zu erstellen, noch dazu in der ungewohnten UML-Notation. Ähnliche Erfahrungen berichtet z.B. [Hasselbring et al. 2007].

2 Das von uns verwendete C#-Beispielprogramm hat ca. 5300 physische Codezeilen (davon 1500 logische Codezeilen [NDepend 2008]) in ca. 30 Klassen.

Der Übergang zu einem iterativ-inkrementellen Entwicklungsprozess verbesserte das Bild ein wenig. Dennoch ist auch bei diesem Vorgehen eine gewisse Hilflosigkeit der Studierenden bei der Analyse der Anforderungen und beim Entwurf zu beobachten gewesen. Insbesondere in den ersten Iterationen.

Würde man die Studierenden gewähren lassen, würden sie zuerst mit der Implementierung beginnen. Sie tun dies in der Regel nicht ohne Grund. Viele Studierende sind unerfahren im abstrakten Entwurf. Sie müssen sich den Entwurf »von unten« erarbeiten.

4.2 Lösungsvorschlag

»Code first« im Kleinen

Die Studierenden haben bis zum Beginn des SE2-Projektes überwiegend keine Erfahrungen mit dem abstrakten Entwurf von Software. Was sie jedoch verstehen, ist Programmcode. Die Studierenden setzen in der ersten Iteration eines iterativ-inkrementellen Entwicklungsprozesses einen einfachen Anwendungsfall direkt in Programmcode um, nachdem sie ihn vorher rudimentär beschrieben haben. Erst danach erstellen sie deskriptive UML-Entwurfsmodelle der bisher erstellten Anwendung und ein zugehöriges Anforderungsanalysedokument. Durch diese Reihenfolge erfahren die Studierenden durch entdeckendes Lernen, wie sich der Einsatz guter Entwurfsprinzipien auf den Programmcode auswirken kann.

»Model first« in der Breite

In der nun folgenden Iteration erstellen die Studierenden für alle übrigen Anwendungsfälle Anforderungsdokumente, Analyse- und Entwurfsmodelle. Der Zeitrahmen bis zum nächsten Meilenstein ist so eng gesteckt, dass die Studierenden kaum eine Chance haben zu »mogeln« und die Implementierung vorzuziehen. Erst die darauf folgende Iteration widmet sich der Implementierung.

Zeitnahes Feedback

Das beschriebene Vorgehen erzwingt ein zeitnahes Feedback des Betreuers. Die in der ersten Iteration erstellten Quelltexte werden vom Betreuer unmittelbar nach der Abgabe begutachtet. Feedback erfolgt innerhalb von 2-3 Tagen nach Abgabe in schriftlicher Form – sowohl teamindividuell als auch durch ein Feedback-Dokument für alle Teams, wenn sich systematisch Fehler eingeschlichen haben.

Weiterhin wird mündliches Feedback von der ersten Woche an im Labor gegeben. Da Faktenwissen nicht durch eine Vorlesung, sondern durch selbstständig durchzulesendes Material transportiert wird, haben die Betreuer mehr Zeit für ad hoc durchgeführtes, mündliches Feedback.

Insgesamt gibt es in der SE2-Veranstaltung schriftliches Feedback auf zwei Zwischenabgaben und eine Endabgabe. Die Abgabetermine sind in etwa gleichen Abständen über die Vorlesungszeit verteilt.

Verwandter Ansatz

Verwandt mit unserem Ansatz ist das in [Paech et al. 2005] beschriebene Vorgehen, ein Praktikum mit einer Reverse-Engineering-Phase starten zu lassen. Dort wird von dem Vorteil berichtet, dass die Beschäftigung mit dem Quellcode von Anfang an ohne SE-Vorwissen möglich war. Der dort beschriebenen Gefahr eines Motivationsproblems begegnen wir, indem wir die Studierenden selbst ein eigenes Programm konstruieren lassen.

4.3 Erfahrungen

Damit die erstellten Programme bereits vor der Begutachtung durch den Betreuer eine gewisse Qualität besitzen, müssen die Studierenden unbedingt Zugriff auf die bereits oben beschriebene Beispielanwendung erhalten. Im Sommersemester 2007 war die Beispielanwendung noch nicht so gut ausgearbeitet und so vollständig wie im Sommersemester 2008. Mit dem Ergebnis, dass im Sommersemester 2007 schlechtere Entwürfe und ein erhöhter Beratungsbedarf enstanden.

Das Vorgehen des »Code first« im Kleinen und der anschließenden Erstellung deskriptiver Entwurfsmodelle führt nach unserer Erfahrung dazu, dass die Studierenden bei der Erstellung der Modelle besser verstehen, was sie da tun. Da sie durch die vorhergehende Programmierung die Prinzipien besser verinnerlicht haben, können sie sich nun vermehrt der immer noch ungewohnten UML-Notation und dem noch nicht ganz so vertrauten UML-Werkzeug widmen.

In der folgenden Phase, in der nun für alle Anwendungsfälle zunächst Modelle erstellt werden, erkennen wir regelmäßig, dass die Studierenden beginnen, im Team auf abstrakter Ebene über die zu erstellende Software mit Mitteln der UML zu kommunizieren. Weiterhin beobachten wir im Vergleich zu früheren Jahrgängen eine deutlich verbesserte Qualität der Analyse- und Entwurfsmodelle. Vereinzelt erproben in dieser Phase sehr gute Studierende (und das wird auch nicht unterbunden) ihre abstrakten Entwürfen durch kurze Implementierungstätigkeiten.

Die Studierenden honorieren detailliertes schriftliches Feedback. Wir beobachten, dass die Studierenden die schriftlichen Feedbacks immer wieder bis zum Ende des Semesters hervorholen, um sich noch einmal zu erinnern, was sie beim ersten Mal falsch gemacht hatten.

5 Brücke von Lernenden zu Praktikern

5.1 Das Problem

Die Studierenden sollen in der Berufspraxis angewendete Vorgehensweisen wie z.B. den Unified Process kennenlernen. Durch die besondere Lernsituation gehen die Studierenden in ihrem eigenen Projekt jedoch völlig anders vor. Der Praktiker arbeitet zu Beginn in abstrakten Disziplinen Requirements, Analyse und Entwurf. In unserem Projekt dürfen die Studierenden – wie oben ausgeführt – mit der Implementierung beginnen. Wie lassen sich in dieser Situation Missverständnisse über das eigentlich übliche Vorgehen vermeiden?

5.2 Lösungsvorschlag

Die Studierenden lernen den Unified Process als ein Vorgehensmodell, das flexibel an die jeweilige Projektsituation anpassbar ist. Als zwei Beispiele werden einander gegenübergestellt: das Lehrveranstaltungsprojekt auf der einen Seite und ein Projekt aus der beruflichen Praxis des Dozenten auf der anderen Seite. Beide Projekte werden dargestellt als spezielle Zuschnitte des Unified Process. Durch Gegenüberstellung der beiden Zuschnitte in einem gemeinsamen Begriffsrahmen erreichen wir, dass die Studierenden ihre aktuelle Tätigkeit in das Vorgehen eines Praktikers, d.h. ihr eigenes zukünftiges Vorgehen, einordnen können.

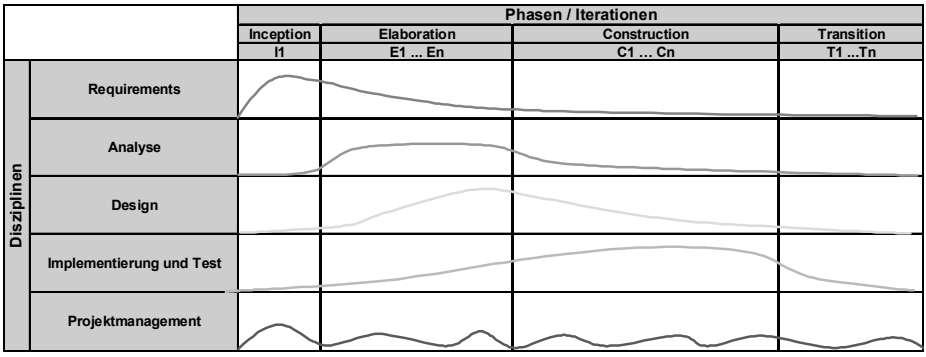


Abb. 3 *Zuschnitt des Unified Process für die Praxis. Die Kurven deuten den Arbeitsaufwand je Disziplin an.*

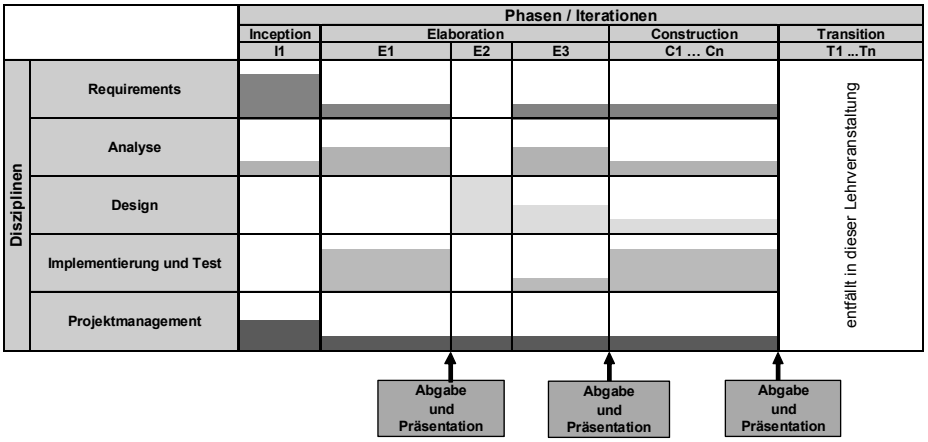


Abb. 4 *Zuschnitt des UP für die Lehrveranstaltung. Die Höhe der Balken deutet den Arbeitsaufwand je Disziplin an.*

Mit der in Abbildung 3 gezeigten Skizze erläutern wir den Studierenden einen üblichen Zuschnitt des Unified Process für ein reales Projekt der Berufspraxis. In frühen Iterationen überwiegen abstrakte Analyse- und Architekturüberlegungen, die durch geringer gewichtete Implementierungstätigkeiten erprobt werden.

Zum Vergleich geben wir den Studierenden in der Lehrveranstaltung den Ablauf in Abbildung 4 vor. Den Studierenden erläutern wir folgende Unterschiede:

- Die Elaboration startet in der Lehrveranstaltung mit einem einfachen Anwendungsfall, der auf einem »happy path« schnell zu einem Erfolgserlebnis führt. In der Realität wählt man hingegen zuerst riskante bzw. geschäftswichtige Anwendungsfälle.
- Die Iterationen E1 und E2 der Lehrveranstaltung laufen in der Realität in anderer Zusammensetzung und Reihenfolge ab. Insbesondere die Arbeitsaufwände in den Disziplinen Design und Implementierung verschieben sich auf der Zeitachse.
- Nachträgliche Modellierungsarbeiten, wie sie in der Iteration E2 der Lehrveranstaltung durchgeführt werden, sind in der Realität zwar auch anzutreffen, allerdings eher für Legacy-Code.

Neben den Unterschieden überwiegen die Gemeinsamkeiten. Ab Iteration E3 ist das Vorgehen in der Lehrveranstaltung vergleichbar mit dem Vorgehen in der Praxis.

5.3 Erfahrungen

Durch Gegenüberstellung der beiden Unified-Process-Zuschnitte fällt es uns nun leichter, Ähnlichkeiten und Unterschiede zwischen Lernvorgehen und Praktikervorgehen darzustellen. Die Teilnehmer des Lernprojektes und die Berufspraktiker sprechen die gleiche Sprache. Sie sprechen beide von Disziplinen und Iterationen.

Ein angenehmer Nebeneffekt: Die Studierenden verstehen, dass kein Projekt wie ein zweites ist und dass ein Prozessmodell immer auf die jeweilige Projektsituation zugeschnitten werden muss.

Gegenstand unserer Betrachtungen ist wie erwähnt ein Softwarepraktikum im 4. Semester. In diesem Kontext ist nach unserer Erfahrung ein gleichermaßen auf Modelle, Dokumente und Programmcode fokussierter Prozess einem dokumentenarmen Prozess wie XP vorzuziehen. Wir beobachten, dass die von uns abgeforderten UML-Diagramme die Studierenden veranlassen, ihre eigenen Programmstrukturen noch einmal zu durchdenken – mit der Folge eines tieferen Verständnisses. Höhere Semester könnten hingegen durchaus von agileren Vorgehensweisen mit größeren Freiheiten profitieren.

6 Diskussion

Dieser Artikel hat sich eine Balance zwischen einerseits ausreichendem Raum für Irrwege und andererseits genügendem Halt durch wegweisende Instruktionen zum Ziel gesetzt. Die beschriebenen vier Maßnahmen wirken hier zusammen.

Das »Code-first«-Vorgehen (Kapitel 4) bietet zu Beginn des Projektes Raum für entdekkendes Lernen und Lernen durch Versuch und Irrtum. Ohne das in Kapitel 3 beschriebene, als Steinbruch zu nutzende Beispielprogramm wären jedoch zeitaufwendige Irrwege vorbestimmt. Und ohne einen Vergleich des Lehrveranstaltungs-Vorgehens mit dem Vorgehen eines Praktikers (Kapitel 5) bestünde die Gefahr des irrigen Verständnisses, das »Code-first«-Vorgehen sei industrietauglich.

Mit dem »Code-first«-Ansatz ist es nach unserer Erfahrung besonders gut möglich, Theorie und Praxis in der in Kapitel 5 beschriebenen L-förmigen Verzahnung zu vermitteln. Würde der Praxisteil wasserfallähnlich mit abstrakten Analyse- und Entwurfstätigkeiten beginnen, wäre ein noch größerer zeitlicher Vorlauf für die Vermittlung von Theorie- und Faktenwissen erforderlich und man landete u.U. wieder bei einem Curriculum, bei dem die SE-Theorie ein ganzes Semester vor dem SE-Projekt (mit den in Abschnitt 2.1 beschriebenen Nachteilen) absolviert würde.

Danksagung

Ich danke den anonymen Reviewern für ihre wertvollen Anmerkungen zur Überarbeitung des Artikels.

Literatur

[Bergin 2007] J. Bergin: Fourteen Pedagogical Patterns (Version vom 5.03.2007). *http://csis.pace.edu/~bergin/PedPat1.3.html* (recherchiert am 20.11.2008)

[GAPP 2008] Google And Paste Programming (Version vom 13.05.2008) *http://c2.com/cgi-bin/wiki?GoogleAndPasteProgramming* (recherchiert am 20.11.2008)

[Hasselbring et al. 2007] W. Hasselbring, J. Matevska, H. Niemann, D. Geesen, H. Garbe, S. Gudenkauf, S. Kruse, C. Möbus, M. Gawunder: Projektorientierte Vermittlung von Entwurfsmustern in der Software-Engineering-Ausbildung. In: SEUH 10 – Software Engineering im Unterricht der Hochschulen, Stuttgart 2007, dpunkt.verlag, Heidelberg.

[Kolb 2005] A. Kolb, D. Kolb: The Kolb Learning Style Inventory Technical Specifications (Version 3.1 vom 15.05.2005) *http://www.learningfromexperience.com/images/uploads/Tech_spec_LSI.pdf* (recherchiert am 20.11.2008)

[Lindig/Zeller 2005] C. Lindig, A. Zeller: Ein Softwaretechnik-Praktikum als Sommerkurs. In: SEUH 9 – Software Engineering im Unterricht der Hochschulen, Aachen 2005, dpunkt.verlag, Heidelberg.

[NDepend 2008] NbLinesOfCode. *http://www.ndepend.com/Metrics.aspx#NbLinesOfCode* (recherchiert am 20.11.2008)

[Paech et al. 2005] B. Paech, L. Borner, J. Rückert, A. H. Dutoit, T. Wolf: Vom Code zu den Anforderungen und wieder zurück: Software Engineering in sechs Semesterwochenstunden. In: SEUH 9 – Software Engineering im Unterricht der Hochschulen, Aachen 2005, dpunkt.verlag, Heidelberg.

[Pedroni/Meyer 2006] M. Pedroni, B. Meyer: The Inverted Curriculum in Practice. Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, Houston, Texas, 1-5 March 2006, ACM.

Game Engineering – Computerspiele in der Software-Engineering-Ausbildung

Daniel Volk

Institut für Softwaretechnologie
Fakultät für Informatik
Universität der Bundeswehr München
daniel.volk@unibw.de

Zusammenfassung

Softwarepraktika in Projektform leisten einen wertvollen Beitrag zur Software-Engineering-Ausbildung im Unterricht der Hochschulen. Insbesondere hier können Studenten in einem der Realtät nahen aber dennoch sicheren Umfeld das im späteren Beruf unbedingt notwendige Handwerkszeug zur erfolgreichen Softwareentwicklung erwerben. Eine klassische Herausforderung in der Konzeption solcher Praktika stellt die Auswahl interessanter Themengebiete dar. Computerspiele sind genau eine solch interessante Domäne mit mittlerweile nicht unerheblicher Relevanz und haben zudem den ganz besonderen Ruf, als Themenfeld mit einem deutlichen Motivationsschub einherzugehen. Im Folgenden sind nun der Ansatz und erste Erfahrungswerte eines Game-Engineering-Praktikums beschrieben, das im Rahmen des Hauptstudiums an der Fakultät für Informatik der Universität der Bundeswehr München durchgeführt wurde.

1 Einleitung

Den Ausgangspunkt für erste Überlegungen in Hinblick auf das beschriebene Praktikum im Bereich *Game Engineering* (*GE*) stellte das an der Universität der Bundeswehr München und an der TU Dresden etablierte Konzept eines frameworkbasierten Softwarepraktikums [Dem99] [Dem05] dar, das an beiden Universitäten ein fester Bestandteil des Grundstudiums ist und bereits von zahlreichen

Studententeams erfolgreich absolviert wurde. In diesem Umfeld entstand die Idee eines fortgeschrittenen Praktikums im Hauptstudium, das die im vorangegangenen Programmierpraktikum erworbenen Fähigkeiten im Bereich Software Engineering (SE) vertiefen sollte. Insbesondere erschien es hierbei als wichtig, sowohl die zusätzliche Komplexität als auch die Motivation der Studenten auf natürliche Weise aus dem anvisierten Themengebiet abzuleiten. Die Umsetzung selbiger Idee als *GE*-Praktikum, ergo einer domänenspezifischen SE-Ausbildung [Jtf04, S.8] mit Fokus auf der Entwicklung von Computerspielen, erwies sich in Bezug auf obige Überlegungen aus den folgenden Gründen von Anfang an als idealer Kandidat: Obwohl die üblichen Projektparameter industrietypischer Spieleprojekte [Bad00] für ein Praktikum natürlich illusorisch sind und erheblich eingeschränkt werden müssen, weist die Entwicklung von Computerspielen zusätzlich auch eine qualitative Komplexität auf, die ein solches Projekt trotz Skalierung auf Praktikumsgröße interessant werden lässt [Swe05]. So ist der unerlässliche Anteil des *Game Design* (des theoretischen Spielentwurfs) aufgrund seiner kreativen Natur ein klassischer Reibungspunkt, welcher u.a. durch Betonung nicht funktionaler Anforderungen Friktionen im Bereich der Analyse zu verursachen vermag [Cal05]. Zudem erfordert die schwere Greifbarkeit des *Game Play* (des eigentlichen Spielerlebnisses) typischerweise auch ein prototypenorientiertes und evolutionäres Vorgehen, das nicht unerheblich auf das Projektrisiko ausstrahlt. Abgesehen von einem interessanten und komplexen Profil der Computerspiele-Entwicklung ist aber natürlich auch der zugehörige Motivationsschub nicht zu übersehen, der faktisch regelmäßig zu besseren Ergebnissen im Rahmen entsprechender Lehrveranstaltungen führt [Bur08] [Rez06]. Um allerdings den Motivationsbonus nicht verpuffen zu lassen, erschien es als eine Notwendigkeit, nicht nur einzelne Aspekte der Spieleentwicklung in ein klassisches SE-Praktikum zu übertragen, sondern zudem ein authentisches Umfeld zu schaffen. In Hinblick auf die konkrete Umsetzung des angedachten *GE*-Praktikums ergaben sich also die folgenden Zielsetzungen:

- **Das primäre Ziel – der Anteil Software Engineering**
  Das Praktikumskonzept sollte also zum einen eine Fortführung der bisherigen Lehranteile im Bereich SE darstellen und sich in diese integrieren. Die anvisierten Lerneffekte sollten sich in natürlicher Form aus der Domäne ergeben.
- **Das sekundäre Ziel – der Anteil Game Development**
  Um einen glaubwürdigen Kontext für die entsprechenden Studentenprojekte zu schaffen, sollte zum anderen der originäre Modus Operandi der Spieleentwicklung (bis zu einem vertretbarem Grad) simuliert werden.

2 Die Bestandsaufnahme

In Vorbereitung des Praktikums wurden entsprechend der oben erwähnten Zielsetzungen also zunächst die beiden Ausgangspunkte näher betrachtet. Einerseits waren dies der SE-Anteil des bestehenden Informatik-Curriculums, andererseits wurde hier das Curriculum der IGDA (*International Game Developer Association*) [Igd03] zu Rate gezogen, das die für die Spieleentwicklung relevanten Themen umreißt.

2.1 Das bestehende Software-Engineering-Curriculum

Der für diese Betrachtung relevante Anteil des Curriculums im Diplomstudiengang Informatik besteht zum einen aus den Vorlesungen Objekt-Orientierte Programmierung und SE sowie dem bereits erwähnten Programmierpraktikum. Um nun die Einbettung des Praktikums in das bestehende Curriculum zu erläutern, werden im Folgenden kurz die Inhalte der Veranstaltungen skizziert:

- **Objekt-Orientierte Programmierung** (3. Trimester)
  Aufbauend auf zwei initialen Vorlesungen zur Vermittlung der Grundfertigkeiten im Bereich der Softwareentwicklung, behandelt selbige Vorlesung zum einen objekt-orientierte Konzepte, das zugehörige Paradigma am Beispiel der Sprache Java sowie weiterführende Konzepte wie Wiederverwendung auf Basis von Klassenbibliotheken, Entwurfsmustern und Rahmenwerken. Zum anderen wird aber auch die objekt-orientierte Methode besprochen, am Beispiel ausgewählter Entwicklungsartefakte vertieft und in einfache Vorgehensmodelle eingebettet.
- **Das Programmierpraktikum** (4. Trimester)
  Aufbauend auf den Kenntnissen der Vorlesung Objekt-Orientierte Programmierung vermittelt nun das Programmierpraktikum einen ersten Einblick in die Organisation von Gruppenarbeit und erlaubt den Studenten, einen einfachen Entwicklungsprozess in der Realität zu erleben. Um dies zu erreichen wird die Situation eines Softwarehauses simuliert, das auf Basis kleiner Entwicklerteams einfache Projekte realisiert. In diesem Rahmen werden nun auch die wichtigsten Typen von Entwicklungsartefakten von den Studenten erstmalig in einem größeren Sinnzusammenhang genutzt. Um den Studenten hierbei Halt zu bieten, sind sowohl der Projektrahmen als auch das zu nutzende Framework (das SalesPoint-Framework, vgl. [Dem99]) vorgegeben. Weiterhin wird auch der konsequente Gebrauch von Softwarewerkzeugen im Gruppenrahmen eingeübt.
- **Software Engineering** (6. Trimester)
  Nach dem praktischen Einüben der Basisfertigkeiten des SE und dem Lösen typischer Problemstellungen im Rahmen des (ersten) eigenen Projekts, erfolgt

nun die theoretische Festigung und Erweiterung selbigen Wissensstands. So werden in der SE-Vorlesung nun u.a. komplexere Vorgehensmodelle (z.B. auch evolutionäre Modelle) aber auch weitere Modellierungskonzepte behandelt und bekannte Anteile detailliert. Zudem werden bis dato kaum behandelte Bereiche wie z.B. die der Projektplanung oder der Qualitätssicherung näher ausgeführt.

Das *GE*-Praktikum ist im Anschluss an die genannten Veranstaltungen einzuordnen und stellt damit eine Fortführung des Programmierpraktikums dar. Hierbei sollte der bekannte Modus Operandi zumindest im groben Rahmen beibehalten, die thematische Ausrichtung jedoch an den Wissenstand der SE-Vorlesung angepasst werden.

2.2 Das IGDA-Curriculum

In Hinblick auf die inhaltliche Ausrichtung fanden sich für das anvisierte Spektrum an SE-Fertigkeiten im vorliegenden IGDA-Curriculum die Anteile **Game Programming** und **Game Production**. Unter ersterem Bereich sind hierbei klassische Informatikthemen wie z.B. Datenstrukturen und Algorithmen zu verstehen, die weitestgehend durch Einführungsvorlesungen der ersten beiden Trimester abgedeckt werden. Weiterhin waren hier jedoch auch spezielle Einzelthemen der Informatik, wie z.B. selektive Anteile der Computergrafik, aufgeführt, die dem vorgesehenen Teilnehmerkreis (noch) nicht zur Verfügung standen. Der zweite Bereich umriss dagegen ein Mischspektrum aus klassischen SE-Themen, durchsetzt von Aspekten des Projektmanagements – beides Anteile, die zum Großteil durch die vorliegenden Vorlesungen des dritten und sechsten Trimesters behandelt wurden. Zusammengefasst erschienen also die bis hierhin beschriebenen Anteile eines *GE*-Praktikums als verhältnismäßig konventionell, wenn auch etwas »eingefärbt« durch die spezielle Domäne. Selbige Erkenntnis wurde auch durch eine erste Evaluation der zugehörigen Literatur im Kernbereich SE [Fly04] sowie im umschließenden Bereich des Projektmanagements [Cha06] bestätigt. Nachdem das angestrebte Praktikumsformat nun aber auch einen gewissen Grad an Authentizität aufweisen sollte, waren neben den puren Informatikanteilen auch Bereiche zu inkludieren, die den Studentenprojekten zudem den Charakter wirklicher Spieleentwicklungen verleihen würden. Eine umfangreiche inhaltliche Produktion erschien diesbezüglich als unangebracht, zumal in diesem Fall eine Ablenkung von den eigentlichen Kerninhalten des Praktikums zu erwarten gewesen wäre. In gewisser Weise erschien selbige Fragestellung der Initialisierung eines neuen Game-Development-Curriculums ähnlich zu sein, bezüglich dessen allgemeinhin ein »**Game-Design-first**«-Ansatz empfohlen wird [Raj05]. In diesem Sinne ist das Game Design als Kernanteil der Spieleentwicklung zu verstehen, von dem alle anderen Teile der Entwicklung abzuleiten sind und der diese zusammenhält [Rol04].

3 Das Praktikumskonzept

Ausgehend von der Evaluation der beiden Curricula ergab sich also für die Konzeption des *GE*-Praktikums folgende Ausprägung: Den Ausgangspunkt der Entwicklung sollte ein *Game-Design*-Anteil bilden, der als Aufgabenstellung für das jeweilige Projekt zu verstehen war. Ableitend hiervon sollte unter Verwendung domänenspezifischer SE-Techniken die Entwicklung der technischen Anteile auf Basis bekannter Implementierungstechniken erfolgen, was einen engen Fokus des Praktikums sicherstellen sollte (vgl. auch [Mcc04]). Die Verwendung allzu sehr spezialisierter Technologie sollte hierbei weitestgehend vermieden werden, ggfs. durch Einschränkungen des *Game Design*, am besten aber durch Nutzung eines geeigneten Frameworks. Den Rahmen des Praktikums sollte ein (bis zu einem gewissen Grad) authentisches Projektmanagement gemäß den üblichen Charakteristika der Spieleindustrie bilden.

3.1 Die Inhalte

Nach der abstrakten Formulierung des Praktikumskonzepts war es nun möglich, hieraus die konkreten Inhalte für das *GE*-Praktikum zu extrahieren, die sich aus den Anteilen des SE und des *Game Development* folgendermaßen zusammensetzen:

- **Die primären Inhalte – der Anteil Software Engineering**
  Ableitend aus dem Szenario der Spieleentwicklung sollten zum einen Aspekte vertieft werden, die bereits aus dem Programmierpraktikum bekannt waren, zum anderen sollten entsprechend der Vorlesung SE bestimmte Aspekte verändert werden. Hinsichtlich ersterer Forderung wurde so z.B. der bereits bekannte objekt-orientierte Analyse-Design-Implementierungszyklus aufgrund schwammiger Anforderungen des *Game Design* erschwert oder auch das Konfigurationsmanagement der Teams aufgrund des hinzugekommenen *Asset Management* (der Verwaltung der Spielinhalte) vor neue Herausforderungen gestellt. Zweiterer Forderung wurde z.B. durch Erweiterung des zuvor weitestgehend sequenziellen Prozessmodells auf Basis modellgetriebener Entwicklung durch evolutionäres Prototyping Rechnung getragen. Um die hiermit einhergehenden Risiken auszugleichen, wurden zudem Refactoring-Techniken und eine aktive Verwendung von Entwurfsmustern forciert. Die bekannten Phasen der Softwareentwicklung wurden zudem um eine Planungsphase bereichert.
- **Die sekundären Inhalte – der Anteil Game Development**
  Der charakteristische Kontext sollte sich zum einen aus der Strukturierung des Praktikums, gemäß des in der Spielebranche üblichen Projektmanagements, der Übernahme der entsprechenden Rollen bzw. Verfahrensweisen und der Integration eines *Game-Design*-Anteils ergeben.

3.2 Die Bewertungskriterien

Als Bewertungsrahmen für die erbrachten Leistungen im unbenoteten Hauptstudiums-Praktikum diente ein anvisierter Arbeitsaufwand von 180 Stunden pro Teilnehmer. Hinsichtlich der Wertigkeit der generierten Artefakte wurden die SE-Anteile den *Game-Design*-Anteilen gegenüber prinzipbedingt bevorzugt. Äquivalent wurde auch der Qualität des Entwicklungsprozesses und der technischen Umsetzung der Prototypen im Verhältnis mehr Beachtung beigemessen als dem Endergebnis im Sinne der technischen und inhaltlichen Quantität (es lagen keine Erfahrungswerte in Form konkreter Metriken aus früheren Praktika vor). So wurde z.B. keine möglichst facettenreiche Spielwelt eingefordert, die korrekte Umsetzung der Kernmechanik, also des zentralen Elements des Spielkonzepts, war dagegen obligatorisch. Die Bewertung selbst erfolgte hierbei auf Basis der im Projektverlauf erstellten Artefakte:

- Game-Development-Artefakte
  - 3× *High-Concept*-Dokument (2-3 Seiten)
  - 1× *Game-Design*-Dokument (abhängig vom Spieltyp 20-30 Seiten)
- Software-Engineering-Artefakte
  - Planungsanteile (1× Grobplanung sowie 2× Feinplanung)
    - Priorisierte Anforderungsliste (technisch/inhaltlich)
    - Risiko-/Aufwandsanalyse/Meilensteindefinitionen
  - Modellierungsanteile (konkrete Ausprägung abhängig vom Spieltyp)
    - Verhaltensdiagramme (primär Use-Case-, Aktivitäts-, Zustands- und Sequenzdiagramme)
    - Strukturdiagramme (primär Paket- und Klassendiagramme)
  - Implementierungsanteile
    - Wegwerf-Prototypen (abhängig vom konkreten Bedarf)
    - Vertikaler Prototyp (inklusive JavaDoc und JUnit-Tests)
  - Evaluationsanteile (in Form der Ergebnisse dokumentiert)
    - Überarbeitungen des *Game-Design*-Dokuments
    - Überarbeitungen der Planungsanteile
    - Re-Design der Modellierungs-/Refactoring der Code-Anteile

4 Die Umsetzung

Da es sich bei der Umsetzung um das Pilotprojekt eines neuen Praktikumkonzepts handelte, war nicht zu erwarten, dass die Planung in allen Aspekten der Realität entsprechen würde. Deswegen wurde von Anfang an ein hoher Grad an Flexibilität und Selbstständigkeit von Seiten der Teilnehmer mit eingeplant, was auch der Auslegung als ein fortgeschrittenes Praktikum im Bereich SE entsprach.

Dies erhöhte zwar deutlich die an die Studenten gestellten Anforderungen, erzeugte im Endeffekt aber genau das notwendige Level an Autonomie, um eine **wirkliche Identifikation** der Studenten mit dem jeweiligen Projekt zu erlauben [Fic05]. Um diesen Effekt noch zu verstärken, wurden (mit Auflagen) auch die Gruppeneinteilung und die Kontrolle über das *Game Design* an die Praktikumsteilnehmer übertragen. Weiterhin wurde den Projektteams bei den bereits aus dem Programmierpraktikum bekannten Anteilen Freiheiten in der Anwendung eingeräumt, die stichpunktartig überprüft wurden. Die Kontrolle der Arbeitsergebnisse erfolgte zudem weniger im Stil von Praktikumspräsentationen gegenüber der Praktikumsleitung, sondern eher im Verständnis von Zwischenstandsberichten gegenüber dem verantwortlichen Projektmanager, was zusätzlich die Illusion realer Spieleprojekte vertiefte. Nur die Beschränkung auf 2-D-Grafik verstand sich in Hinblick auf die Vorkenntnisse von selbst.

4.1 Der personelle Rahmen

Von den sich ursprünglich bewerbenden 30 (spieleaffinen) Studenten (ein typischer Informatik-Jahrgang umfasst an der Universität der Bundeswehr lediglich ca. 75 Studenten), wurden die 20 vielversprechendsten Teilnehmer ausgewählt, die sich in vier Teams aufteilten.

Innerhalb der Entwicklerteams waren zudem die speziellen Rollen (vgl. Abb. 1) des Teamchefs (*Developer Producer*), des *Game-Design*-Verantwortlichen (*Lead Designer*) und des Technikverantwortlichen (*Lead Engineer*) zu vergeben. Die Rolle der Praktikumsleitung war mit der eines Projektmanagers (*Publisher Producer*) gleichzusetzen. Zur Kompensation evtl. technischer Probleme existierte zudem die Rolle eines *Game Consultant*, die von einem studentischen Tutor übernommen wurde

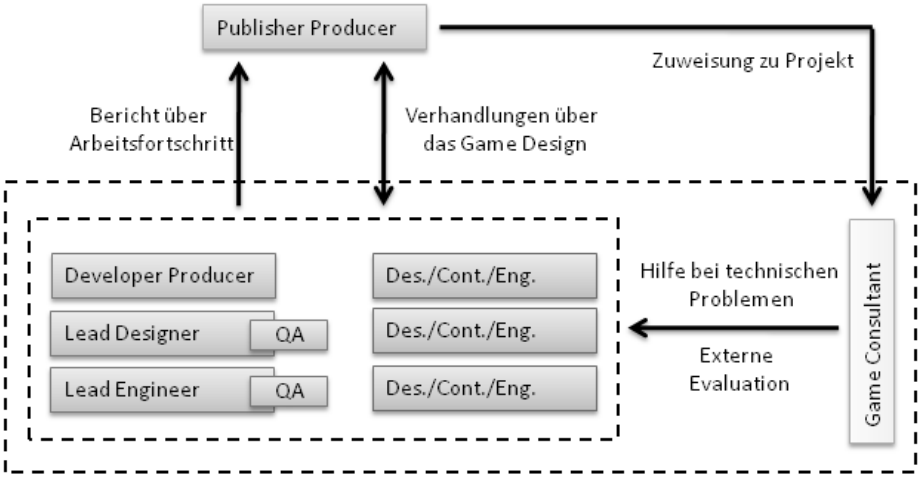


Abb. 1 *Die Entwicklungsrollen*

4.2 Der technische Rahmen

Gemäß dem vorliegenden Curriculum war Java als Sprache der Wahl gesetzt, was allerdings die Auswahl der verfügbaren Plattformen zur Spieleentwicklung erheblich einschränkte [Wal06]. Nach der Evaluation einer kleinen Anzahl möglicher Kandidaten fiel die Wahl auf das **Slick Game Framework**, eine relativ flexibel verwendbare Ansammlung einzelner Bibliotheken, die von einer Kernkomponente mit integrierter Spielschleife zusammengehalten werden. Zum abgebildeten Funktionsspektrum des Frameworks zählen z.B. eine einfache Ressourcen-Verwaltung und grafische Ausgabemöglichkeiten auf Sprite-Basis (vgl. hierzu auch [Tkd09]). Da es sich bei *Slick* allerdings nicht um ein Rahmenwerk mit Ausbildungshintergrund handelt, stand auch keine extensive Dokumentation zur Verfügung. Um dieses Problem etwas zu lindern, wurde die bereits erwähnte Rolle eines *Game Consultant* in das Praktikumskonzept aufgenommen, dessen Aufgabenfeld zum einen die Erstellung eines Tutorials zur Demonstration der Framework-Grundfunktionalität und zum anderen die technische Betreuung der Teams (z.B. durch einen Vortrag zum Framework) beinhaltete. Hinsichtlich der Entwicklungswerkzeuge waren die Versionsverwaltung *SVN* und die Entwicklungsumgebung *IBM Rational Systems Developer* vorgeschrieben.

4.3 Der Prozessrahmen

Organisatorisches

Entsprechend den üblichen Charakteristika eines Spieleentwicklungsprozesses zergliederte sich der Ablauf des Praktikums in drei grobe Abschnitte (vgl. Abb. 2): einer *Pre-Production*-Phase von vier Wochen Dauer, einer *Production*-Phase von acht Wochen Dauer und einer abschließenden *Post-Production*-Phase von zwei bis drei Tagen. Die erste Phase enthielt hierbei eine Konzeptions- (drei Iterationen à eine Woche) und eine Planungsphase (eine Woche). Die Produktionsphase beschränkte sich auf drei Iterationen des *Prototyping* (d.h. der technischen Umsetzung), eine *Full-Production*-Subphase war dagegegen aufgrund der Vernachlässigung der Inhaltsanteile zunächst nicht vorgesehen. Zur Intensivierung des Spieleentwicklungskontexts wurden zu Beginn der Vorproduktion Vortragsanteile mit Informationen zum Spielemarkt und der zugehörigen Industrie integriert. Den Start- bzw. Endpunkt des Praktikums bildeten die Industrie-übliche *High-Concept*-Präsentation (die Vorstellung der Spielidee) respektive eine Marketing-orientierte Endpräsentation. Weiterhin wurden vor der jeweiligen Phase Informationsveranstaltungen zu ausgewählten *Game-Design*-, Planungs- und SE-Anteilen sowie zum *Slick*-Framework angeboten. Die Praktikumskommunikation erfolgte auf der Projektmanagementebene mit Hilfe einer Praktikumsseite inklusive »Hot News«-Anteil, Anleitungen/Hilfestellungen für die Entwicklung und einem Google-Kalender zur Planung. Die Projektseiten der Studententeams spiegelten

die SE-Ebene wider und beinhalteten sowohl den ständig aktuellen Stand des Projektfortschritts als auch alle externen Dokumentationsartefakte.

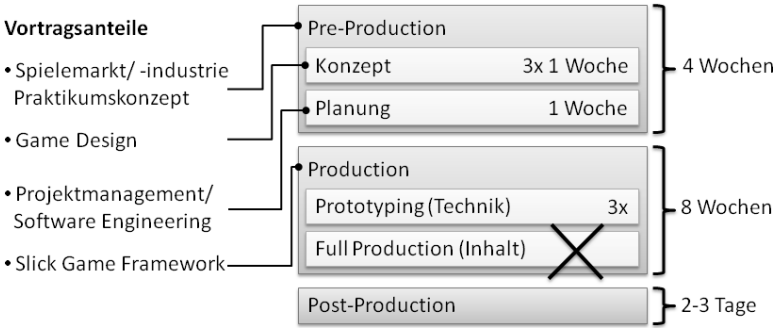


Abb. 2 *Die Entwicklungsphasen*

Alle übrigen Artefakte standen der Praktikumsleitung direkt über das Versionierungssystem zur Verfügung, was eine schnelle und effiziente Möglichkeit des Zugriffs erlaubte. Neben der eigenen Projektseite, die um übliche »Marketing-Anteile« bereichert werden durfte, wurden wöchentliche Präsentationen (60-90 Min.) vorgesehen. Diese dienten einerseits dazu, die erarbeiteten Anteile vorzustellen und zu besprechen, andererseits stand auf diese Weise aber auch jedem Team genügend Zeit zur Verfügung, Fragen zu klären bzw. Hilfestellungen bei Problemen zu erhalten. Zudem erhielten alle Teilnehmer aufgrund der noch spärlichen Dokumentation einen wöchentlichen »Praktikums-Newsletter«, der die Ergebnisse der vergangenen Woche besprach und Tipps und Anleitungen für die folgende Woche enthielt.

Pre-Production-Phase

Die Umsetzung der Konzeptphase, d.h. also der Erstellung eines *Game Design*, erschien zunächst als der schwierigste Anteil in Hinblick auf die Umsetzung des Praktikums, da die für eine zugehörige Einweisung zur Verfügung stehenden Zeit als zu knapp bemessen erschien. Dies wurde noch dadurch verstärkt, dass eine Eingrenzung der Thematik durch die Wahl von vier komplett unterschiedlichen Spielekonzepten erheblich erschwert wurde. Aus diesem Grund wurde an dieser Stelle ein iterativer Ansatz gewählt. So umrissen die einleitenden Vortragsanteile nur die Grundlagen des *Game Design* und gaben eine Starthilfe für die ersten Schritte. Nun war es an den Studententeams, auf Basis der zur Verfügung stehenden Literatur ([Rol03] erwies sich hier als geeignete Wahl) eine erste Iteration der eigenen Spielidee zu erstellen. Diese Version wurde nun in der darauf folgenden Wochenpräsentation detailliert besprochen und der Weg zur nächsten Iteration anskizziert. Selbige Diskussionen erfolgten hierbei primär auf der Ebene von Genre-typischen Aspekten bzw. *Game-Design-Pattern* [Bjö05], was aufgrund der nicht zu

unterschätzenden Vorkenntnisse der Teilnehmer im Bereich der Computerspiele ohne Weiteres möglich war. Hierbei wurden von jeder Gruppe zunächst drei Initialkonzepte erstellt, von denen dann eines in Richtung eines *Game-Design*-Dokuments verfeinert wurde (selbiges enthielt sowohl die Kernidee des Spiels als auch alle narrativen Anteile und die zugehörige Spielmechanik). Das Dokument wurde wiederum als »lebendige« Form der Dokumentation verstanden, die bis zum Ende des Praktikums weitergepflegt und bei Bedarf angepasst wurde. In der sich anschließenden Planungsphase erfolgte nun zunächst die Extraktion inhaltlicher und technischer Anforderungen aus den *Features* des *Game-Design*-Dokuments sowie deren Priorisierung in Hinblick auf das *Game Play*.

Auf Basis der sich ergebenden Liste wurde eine erste Aufwands-/Risikoanalyse vorgenommen, die wiederum die Grundlage für die Formulierung der Meilensteine (der Ausbaustufen des vertikalen Prototypen) bildeten. Auch Aufwands-/Risikoanalyse wurden hier nur als »erster Wurf« verstanden und im weiteren Verlauf des Projekts mehrfach aktualisiert. Der fehlenden Erfahrung der Studenten wurde hier u.a. mit Hilfe einer relativen Schätzmethode – eines Planungspokers – begegnet.

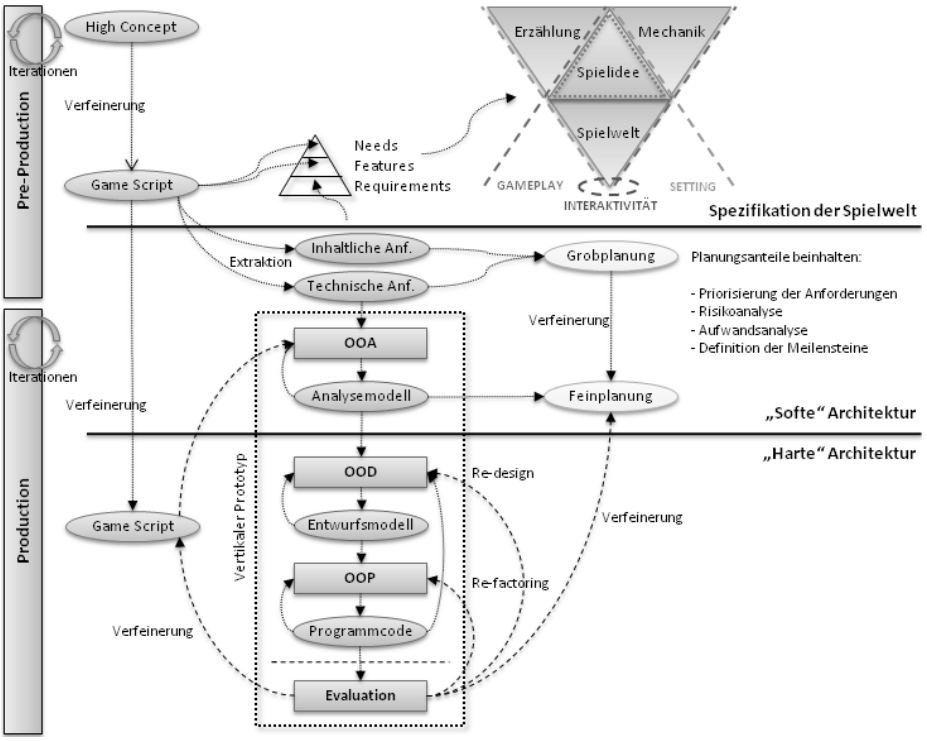


Abb. 3 *Der Entwicklungsprozess*

Production-Phase

Aufgrund der bestehenden Vorkenntnisse wurden den Projektteams nur geringfügige Vorgaben bzgl. des klassischen OOA-OOD-OOP-Zyklus gemacht. Allerdings ergaben sich einige domänenspezifische Eigenheiten, die im Folgenden besprochen werden: Während ursprünglich drei, rein technisch orientierte Prototyping-Iterationen vorgesehen waren, tendierte die Entwicklung (in der Realität dann zwei Iterationen) jedoch zum Ende hin teilweise in Richtung *Full Production*. Nur dadurch wurden die technischen Prototypen auch wirklich zu Spieleprotypen. Nichtsdestotrotz verblieb der Fokus auf der technischen Entwicklung, die auf zweierlei Wege ihren Anfang fand: Zum einen erfolgte die Erstellung kleiner Wegwerf-Prototypen, deren Zweck das schlichte Ausloten des Fähigkeitsprofils des gewählten Frameworks war, wobei hier explizit darauf geachtet wurde, dass diese niederqualitativen Fragmente keinen direkten Eingang in die eigentliche Codebasis fanden (die Codequalität der Projekte wurde regelmäßig per SVN kontrolliert). Zum anderen fand nun eine genauere Analyse der zuvor extrahierten Anforderungen (des ersten Meilensteins) statt, die mit Hilfe unterschiedlicher Analysetechniken zu einem Domänenmodell zusammengeführt wurden. Neben einem modellgetriebenen Ansatz, primär in Form von Kontext- und »60 Sekunden *Game Play*« Use Cases [Fly04] sowie Klassen-/Zustandsdiagrammen, erwiesen sich auch ein weniger formales Storyboarding und (in Teilen) CRC-Karten-Sitzungen als hilfreich. Allerdings wurden in dieser Phase auch die qualitative Komplexität und Andersartigkeit der Domäne offensichtlich, was sich in klaren Lücken der Analysemodelle äußerte. Hierbei waren der für Computerspiele typische Kontrollfluss in Form einer Ereignisschleife und die zustandsbasierte Spielweltsimulation die primären Problemfelder.

Nach einer Verfeinerung der zugehörigen Planungsanteile wurde nun versucht, die beiden Anteile in Form eines objekt-orientierten Entwurfsmodells zusammenzubringen. Die modellbasierte Verschmelzung zwischen dem durch die Prototypen erarbeiteten Framework-Know-how und des in eine technische Sicht transferierten Domänenmodells beschränkte sich aber auch im Entwurf auf einige wenige Diagrammanteile. Viele der relevanten Aspekte konnten von den Projektteams erst durch Implementierung des vertikalen Prototypen umrissen werden. Da die Fokussierung auf solch evolutionäre Prototypen und die damit einhergehende Bottom-up-Entwicklung aber bekanntermaßen mit einem nicht unerheblichen Risiko behaftet sind, wurde im Rahmen des Praktikums auch ganz besonderer Wert auf die regelmäßige Überarbeitung der Codebasis und Anwendung von Entwurfsmustern [Gam01] gelegt. Hierbei wurde den Teilnehmern laut eigener Aussage die Sinnhaftigkeit selbiger Maßnahmen auch deutlich klarer als bei deren Verwendung im Rahmen des deutlich sequenzielleren Programmierpraktikums. Die gleiche Aussage gilt hierbei auch für den Bereich der Regressionstests, die konkrete Umsetzung auf Basis von JUnit fiel den Teilnehmern aufgrund der ungewohnten Domäne allerdings erkennbar schwer.

Post-Production-Phase

Nach zwei Durchläufen des zuvor beschriebenen *Prototyping* bestand die letzte Phase des Praktikums dann zum einen aus der Reflektion des umgesetzten Projekts aus Sicht des jeweiligen Projektteams, die – wie in der Spieleindustrie üblich – in Form eines *Post Mortem* umgesetzt wurde. Zum anderen erfolgten noch die Endpräsentation der Projekte und eine Abschlussbewertung im Gesamtrahmen.

5 Die Evaluation

Endergebnis des *GE*-Praktikums waren vier lauffähige Prototypen aus komplett unterschiedlichen Spiele-Genres (im Bereich von ~8.000 – 16.000 SLOCs): So wurde mit dem Projekt *AntZeit* (vgl. [Tkd09]) ein isometrisches Echtzeitstrategiespiel im Mikrokosmos der Ameisen umgesetzt. *Operation: DIPLOMA* dagegen vertrat das Genre der *Point&Click Adventures* auf Basis statischer Szenerien. Der Sidescroller *Auftrag Bachelor* realisierte ein klassisches *Jump&Run* mit Studienhintergrund. Das Rollenspiel *Tales of Lavankor* letztlich setzte in Top-down-Sicht eine Fantasy-Welt um. Der herausragendste Aspekt bei der Entwicklung der Spiele war hierbei der hohe Grad an Selbstständigkeit und Motivation der Teilnehmer, der bei zweien der Projektteams so weit ging, dass diese auch nach Abschluss des Praktikums die Entwicklung fortführten. So stellten der neue Anteil des *Game Design* und der erhöhte Schwierigkeitsgrad im Prozess und der technischen Umsetzung keine allzu hohen Hürden dar, wären allerdings ohne die bestehenden Vorkenntnisse im Bereich SE kaum zu realisieren gewesen. Aber auch die üblicherweise sehr unbeliebten Planungsanteile wurden sorgsam angegangen – man wollte ja das **eigene Projekt** im Griff behalten. In diesem Sinne kam in der Rolle des *Publisher Producer* auch eher das Gefühl einer wirklichen Projekt- denn Praktikumsbetreuung auf, was aber auch durch die entsprechende Zeitinvestition und Flexibilität erkauft werden musste. Zudem war auch die Kreativität der Teilnehmer hinsichtlich der inhaltlichen Produktion etwas unterschätzt worden: Diese reichte letztlich von eigenen (einfachen) 3-D-Modellen bis hin zur Hintergrundmusik einer semiprofessionellen Rollenspielerband. In Hinblick auf die initiale Konzeption zeigte sich die geplante Dauer (2,5 Wochen) der drei angedachten Prototyping-Iterationen als zu kurz bemessen (dies wurde ad hoc auf zwei Iteration à vier Wochen angepasst). Die nicht eingeplante *Full-Production*-Phase fand in Teilen, parallel zur technischen Entwicklung und auf Basis von Eigeninitiative der Teams, doch statt. Bezüglich der reinen SE-Themen erwies sich der Analyseanteil als Herausforderung, was sich vor allem auf das noch spärliche Hilfsmaterial sowie fehlende Beispiele in Form von Vorgängerprojekten zurückführen ließ. Die mit fortschreitender Bearbeitung etwas vernachlässigten Entwurfsanteile sollten zukünftig stärker in Form einer speziellen Refactoring-/Re-Design-Sitzung eingefordert werden. Letztlich könnte auch die Authentizität

des Ansatzes noch (relativ einfach) durch »Bewerbungen« der Projektteams mit dem angedachten *High Concept* und (deutlich aufwendiger) durch interdisziplinäre Teams gesteigert werden. Für letzteren Part würde sich die Kooperation mit externen Partnern aus Bereichen der inhaltlichen Produktion anbieten.

Zusammenfassend formuliert, wurden im Projektverlauf die in Abschnitt 3.2 aufgeführten Bewertungskriterien von allen Teams erfüllt, allerdings war eine vom Spieltyp abhängige, unterschiedliche Betonung von genretypischen Eigenheiten unerlässlich. Im Ergebnis erwies sich der Pilot des *GE*-Praktikums für beide Seiten als ein sehr zeitintensives und forderndes, aber auch äußerst lehrreiches und damit erfolgreiches Projekt im Rahmen der SE-Ausbildung, das unbedingt fortgeführt werden sollte.

Literatur

[Bad00] Badcoe, I., 2000. The Computer Game Industry: Current State of Play. *Virtual Reality*, Volume 5, Number 4. Springer, London, 204-214

[Bjö05] Björk, S. and Holopainen, J., 2005. *Patterns in Game Design*. Charles River Media, Boston, MA

[Bur08] Burns, B., 2008. Teaching the Computer Science of Computer Games. *Journal of Computing Sciences in Colleges*, Volume 23, Issue 3. Consortium for Comp. Sciences in Colleges, USA, 154-161

[Cal05] Callele, D., Neufeld, E., Schneider, K., 2005. Requirements Engineering and the Creative Process in the Video Game Industry. In *Proc. of the 2005 13th IEEE International Conference on Requirements Engineering*. IEEE, 240-252

[Cha06] Chandler, H., 2006. *The Game Production Handbook*. Charles River Media, Inc., Hingham, MA

[Dem99] Demuth, B. Hußmann, H. Zschaler, S. Schmitz, L., 1999. Erfahrungen mit einem frameworkbasierten Softwarepraktikum. In *SEUH'99*. Teubner, Wiesbaden, Germany

[Dem05] Demuth, B. Schmitz, L. Wittek, B., 2005. Softwaretechnologie-Praktikum im Grundstudium: universitäres oder reales Projekt? In *SEUH'05*. Teubner, Wiesbaden, Germany

[Fic05] Ficocelli, L., Gregg, D., 2005. A Group Project – with a twist! In *Proc. of the 2005 Western Canadian Conference on Computing Education (WCCCE)*. At *http://web.unbc.ca/wccce05/html /proceedings.html* (last accessed 20.10.08)

[Fly04] Flynt, John P., Salem, O., 2004. *Software Engineering for Game Developers*. Thomson Course Technology, Boston, MA

[Gam01] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 2001. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley

[Igd03] IGDA Education Committee, 2003. *IGDA Curriculum Framework: The Study of Games and Game Development*. Version 2.3 beta, Feb 25, 2003

[Jtf04] The Joint Task Force on Computing Curricula, ACM, IEEE-CS, 2004. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. Aug 23, 2004

[Mcc04] McCallum, S., Mackie, J., Nacke, L., 2004. Creating a Computer Game Design Course. In Proc. of the New Zealand Game Developers Conference. NZGDC, Dunedin, New Zealand

[Raj05] Rajagopalan, M., Schwartz, D.I., 2005. Game Design and Game-Development Education. *Phi Kappa Phi Forum*, Volume 85, Summer 2005, at *http://www.highbeam.com/ doc/1G1-135022702.html* (last accessed 20.10.08)

[Rez06] Rezk-Salama, C., et al., 2006. Game Development as Part of the Computer Science Education. In *Proc. International Digital Games Conference (iDiG)*. Springer, 15-24

[Rol03] Rollings, A., Adams, E., 2003. *Andrew Rollings and Ernest Adams on Game Design*. New Riders Publishing, Berkeley, CA

[Rol04] Rollings, A., Morris, D., 2004. *Game Architecture and Design: A New Edition*. New Riders Publishing, Berkeley, CA

[Swe05] Sweedyk, E., Keller, R.M., 2005. Fun and Games: A New Software Engineering Course. *SIGCSE Bulletin*, Volume 37, Number 3. ACM, NY, 138-142

[Tkd09] Team KAROSHI [Dev]. Volk, D., 2009. Das Projekt AntZeit. In *SEUH´09*. dpunkt.verlag, Heidelberg, Germany

[Wal06] Wallace, S.A., Nierman, A., 2006. Addressing the Need for a Java Based Game Curriculum. *Journal of Computing Sciences in Colleges*, Volume 22, Issue 2. Consortium for Computing Sciences in Colleges, USA, 20-26

Das Projekt AntZeit

René Fröhlich · Erik Hänel · Markus Knopp · Daniel Nemschok · Alexander Schramm
Team KAROSHI [Dev]

Studentenjahrgang Informatik 2006
Universität der Bundeswehr München
{rene.froehlich|erik.haenel|markus.knopp|daniel.nemschok|alexander.schramm}@unibw.de

Daniel Volk

Institut für Softwaretechnologie
Fakultät für Informatik
Universität der Bundeswehr München
daniel.volk@unibw.de

1 Das Projekt AntZeit

Das im Folgenden beschriebene Softwareprojekt AntZeit des fünfköpfigen Studententeams *KAROSHI [Dev]* ist eines der Praktikumsprojekte, an den im Rahmen des Game Development Praktikums [Vol09] an der Universität der Bundeswehr München gearbeitet wurde. Als technische Plattform für die Java-basierte Umsetzung kam das *Slick Game Framework*[1] zum Einsatz. In Hinblick auf den Freiheitsgrad des Entwicklungsteams unterlag das Spieleprojekt einer Beschränkung auf 2-D-Grafik und der (indirekten) Begrenzung auf den Realisierungsgrad eines spielbaren Prototypen, was der Praktikumsdauer von drei Monaten geschuldet war.

In dem sich an übliche Verfahrensweisen der Spieleindustrie [Rol04] anlehnenden Entwicklungsprozess ergab sich in der Vorproduktionsphase, in Bezug auf das Game Design, das Thema eines Echtzeit-Strategiespiels im Mikrokosmos einer Ameisenkolonie, das in einer isometrischen 2-D-Ansicht umgesetzt wurde (vgl. Abb. 1). Aufgrund der zeitlichen Beschränkungen wurde hierbei bewusst auf

1 Slick – 2D Game Library based on LWJGL – (*http://slick.cokeandcode.com*)

Abb. 1 *Screenshot aus dem Spielgeschehen*

die Implementierung einer AI-Komponente verzichtet, im Gegenzug jedoch eine Mehrspielerkomponente auf Basis des *Sun DarkStar Frameworks*[2] integriert, das die Integration der notwendigen Kommunikations- und Persistenzanteile auf höherem Abstraktionsniveau erlaubte. Nach Erstellung des Game-Design-Dokuments und der sich anschließenden Extraktion von inhaltlichen und technischen Anforderungen erfolgte zum Ende der Vorproduktionsphase der Grobentwurf der Architektur, die Einteilung in Meilensteine inklusive zugehöriger Planungsanteile und eine Risikoabschätzung. Für die darauf folgende iterative Umsetzung ergab sich die folgende Aufteilung: Im Rahmen des ersten Meilensteins wurde zunächst das Weltmodell des Spiels entworfen, die für die Darstellung notwendige Isometrie-Schicht implementiert und ein erweiterter Ressourcenmanager auf der Basisfunktionalität des Slick-Frameworks aufgesetzt (vgl. Abb. 2). Der zweite Meilenstein konzentrierte sich auf die Spezifikation der notwendigen Spielzustände, die Realisierung der Spielmechanik in der primären Spielschleife und die Umsetzung der Spielerschnittstelle auf Basis des Slick-GUI-Modells. Weiterhin wurde der noch einfache Ressourcenmanager in Richtung eines in der Spieleent-

2 SUN Project Darkstar – (*http://www.projectdarkstar.com*)

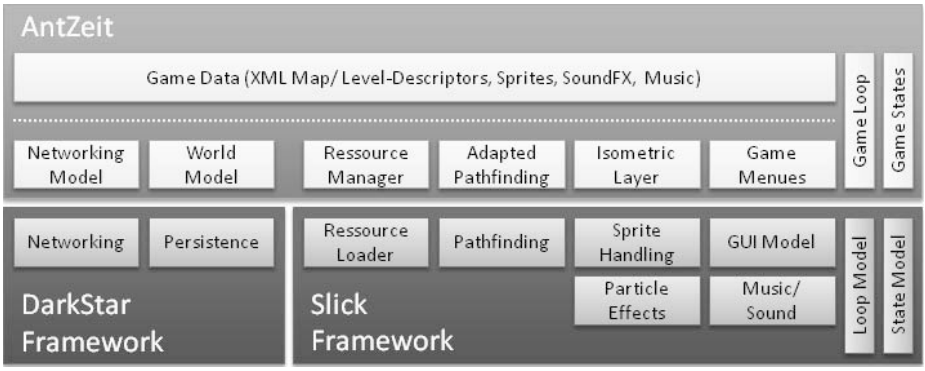


Abb. 2 *Der Überblick über den technischen Aufbau*

wicklung vorherrschenden, datengetriebenen Ansatzes erweitert, der eine XML-basierte Auslagerung statischer Spielwelt-Anteile ermöglichte. Der Fokus des dritten Meilensteins lag dann auf der Mehrspieler-Fähigkeit in Form eines einfachen Netzwerkmodells, einer Optimierung der zur Verfügung stehenden Pfadfinde-Algorithmen und der Integration einfacher Partikeleffekte.

In der Gesamtheit ergab sich ein lauffähiger Prototyp der Software, deren Realisierung zwar arbeitsintensiv (~16.000 SLOC), aber aufgrund des Spielethemas hoch motivierend und lehrreich war. Vor allem die neuartigen Herausforderungen der bis dato nur theoretisch bekannten evolutionären Vorgehensweise und der damit verbundenen Test- und Überarbeitungsphasen wurden nun direkt am eigenen Projekt erlebbar.

Literatur

[Rol04] Rollings, A., Morris, D., 2004. *Game Architecture and Design: A New Edition*. New Riders Publishing, Berkeley, California

[Vol09] Volk, D., 2009. Game Engineering – Computerspiele in der Software Engineering Ausbildung. In *SEUH'09*. dpunkt.verlag, Heidelberg, Germany.

Von Häuptlingen und Indianern – Bachelor/Master als Chance

Karsten Weicker

Hochschule für Technik
Wirtschaft und Kultur Leipzig (FH)
Fachbereich Informatik, Mathematik und Naturwissenschaften
karsten@weicker.info

Nicole Weicker

Pädagogische Hochschule Heidelberg
Fakultät III, Institut für Datenverarbeitung/Informatik
nicole@weicker.info

Zusammenfassung

Angelehnt an Hierarchien in Projekten der realen Welt wird eine neue Organisationsform des Softwarepraktikums vorgestellt, in der Bachelor- und Masterstudenten gemeinsam Projekte bearbeiten. Das Ziel ist, Lernen im Projekt auf zwei Ebenen aufzuteilen. Der Ansatz wird anhand von Fragebögen analysiert und entsprechend bewertet.

1 Motivation

Die zentrale Bedeutung des Softwarepraktikums (Sopra) im Informatikstudium liegt in der Möglichkeit, die gelernten Inhalte der Softwaretechnik praktisch umzusetzen und dabei neben fachlichen und methodischen Kompetenzen durch die Teamarbeit auch eine Reihe von Sozial- und Selbstkompetenzen zu erwerben. Letzteres wird durch die Akkrediteure der neuen Bachelor- und Masterstudiengängen immer stärker gefordert [ASIIN 2005, ASIIN 2006].

Dabei kommt der Durchführung und der Organisation des Softwarepraktikums besonders viel Bedeutung zu. Die Arbeit von [Fincher et al. 2001] liefert einen umfassenden Überblick, auf wie viele unterschiedliche Arten Projektarbeit im Informatikstudium integriert werden kann. Insgesamt kann ein Trend von

eher kleinen Gruppen [Schlimmer et. al. 1994] zu größeren Gruppen mit bis zu 12 Studenten beobachtet werden [Ludewig/Reißing 1998]. So wurde auch an der HTWK Leipzig bis 2004 das Softwarepraktikum mit individuellen Projekten von Gruppen mit 2-4 Studierenden durchgeführt. Im Zuge der anstehenden Akkreditierung der Studiengänge wurde danach die Teamgröße angehoben, und die individuellen Projekte wurden durch ein für alle Gruppen identisches Thema ersetzt. Allerdings sind im Bachelor/Master-System zeitintensive Konzepte wie zwei einjährige, überlappende Studienprojekte [Ludewig/Reißing 1998] nicht umsetzbar.

Im Weiteren wollen wir ein Konzept für die Durchführung von Softwarepraktika skizzieren, wobei die folgenden Anforderungen die Rahmenbedingungen vorgeben: Die Dauer beträgt ein Semester, und die Teams sollen 8-12 Personen umfassen. Ferner muss das Projekt »realitätsnah« sein, d.h., ein echter Kunde (sprich: extern oder ein Hochschullehrer, in keinem Fall ein Student) definiert ein komplexes und evtl. risikoreiches Produkt, und innerhalb des Teams sollen projektübliche Hierarchien abgebildet werden. Insbesondere legen wir Wert auf die üblichen Teamprozesse in einem Projekt mit großem Zeitdruck sowie auf ein individuelles Projektmanagement.

In einer Befragung wurde an der HTWK Leipzig untersucht, welche der im Sopra angestrebten fachlichen, methodischen, persönlichen und sozialen Kompetenzen die Studierenden im Rahmen der Lehrveranstaltung gelernt haben.

2 Organisation des Softwarepraktikums und seine Probleme

Für die Organisation des Softwarepraktikums steht eine ganze Bandbreite möglicher Betreuungsmodelle zur Verfügung. Diese reichen von der verbreiteten, weil mit wenig bis keinem Aufwand realisierbaren, Laissez-faire-Betreuung bis zu Formen, in denen das Projektmanagement den Studenten quasi durch wöchentliche, von allen zu bearbeitende Aufgaben abgenommen wird. Letzteres verfehlt dabei häufig viele der Lernziele, die wir mit einem Softwarepraktikum verbinden, da die erwünschten fachlichen Auseinandersetzungen und gruppeninternen Konflikte unterbleiben.

Aber auch die Laissez-faire-Betreuung ist kritisch zu hinterfragen, da bereits in [Weicker et al. 2006] argumentiert wurde, dass unbetreute Kompetenzvermittlung nur in seltenen Fällen gelingt. Das deckt sich insbesondere auch mit unseren Erfahrungen mit Softwareprojekten, in denen die Studenten die Projektleitung selbst organisiert haben. Die Wahl eines Projektleiters existiert meist nur auf dem Papier, da die gewählte Person häufig weder das notwendige Charisma besitzt noch über die technischen und organisatorischen Fähigkeiten bzw. Erfahrungen verfügt. Und selbst wenn diese Voraussetzungen gegeben sind, wird der Leiter nicht als Primus inter Pares akzeptiert, sondern vom Team immer wieder in seiner Autorität in Frage gestellt. So läuft es in der Regel auf das reine Konsensprinzip ohne echte Führung hinaus.

Daher sind all die Zwischenformen in der Betreuung und Organisation interessant, die sowohl Teamdynamik und reale Projektsituationen als auch eine gewisse Betreuung miteinander vereinbaren. Ein Hilfsmittel ist der Einsatz von Tutoren, die über ein gewisses Maß an Erfahrung verfügen, da sie die Veranstaltung bereits erfolgreich absolviert haben. [Stoyan/Glinz 2005] und [Lindig/Zeller 2005] setzen studentische Hilfskräfte als »Coachs« ein, die den Prozess begleiten und den Teams mit Ratschlägen zur Seite stehen. [Demuth et al. 2002] gehen noch einen Schritt weiter, da dort der Tutor zum größten Teil die Projektleitung übernimmt. Der Tutoreneinsatz ist in jedem Fall positiv zu werten. Allerdings bringt dies auch einen gewissen Unsicherheitsfaktor mit sich, da Tutoren die Aufgabe mit unterschiedlich starker Motivation angehen – dies reicht von einem reinen »Job« bis zu Tutoren, die auch als Coach das Team bevormunden. (Dies haben wir konkret im geringfügig anderen Kontext eines projektbasierten, fortgeschrittenen Programmierkurses im zweiten Semester beobachtet.) Was dabei fehlt, ist Verbindlichkeit in vielerlei Hinsicht.

Zur Motivationssteigerung haben [Stoyan/Glinz 2005] damit gearbeitet, dass die Tutortätigkeit mit ECTS-Punkten belohnt wird. Dies ging uns bei der Neukonzeption des Softwarepraktikums noch nicht weit genug, sondern wir wollten neben einer verbindlichen Aufgabenteilung auch eine echte Benotung der betreuenden Studenten. Unser Ansatz wird im nächsten Abschnitt beschrieben und ist angelehnt an ein Szenario der University of Leeds [Fincher et al. 2001], bei dem Studenten im dritten Studienjahr vierköpfige Teams aus dem ersten Studienjahr anleiten.

3 Softwarepraktikum an der HTWK

Dem politischen Willen Rechnung tragend, dass der Bachelor-Abschluss berufsqualifizierend sein soll, wurden an der HTWK (wie an vielen anderen Hochschulen auch [BMBF 2005]) die Kapazitäten entsprechend gestuft: Auf 80 Informatik-Bachelors kommen 20 Informatik-Masters. Dazu kommen an der HTWK noch 50 Medieninformatik-Bachelors. Ausgehend von diesen Zahlen wurde eine gemeinsame Veranstaltung konzipiert, in der die Masterstudenten die Projektleitung in den Softwarepraktika übernehmen. Damit haben wir Teams bestehend aus ausschließlich gleichberechtigten Studenten ersetzt durch viele Indianer mit wenigen echten Häuptlingen. Im »worst case« leiten zwei Master in ihrem zweiten Master-Semester (Projektleitung und Qualitätssicherung) ein Team mit 13 Bachelors. Studienabbrecher berücksichtigend sind es häufig Teams mit neun Bachelorstudenten. In den ersten drei Jahren Übergangszeit vom Diplom zum gestuften System variiert die Zahl der Masterstudenten noch stark, weshalb hier flexibler organisiert wird. Die Vorbildung der Masterstudenten umfasst während des Bachelorstudiums zwei Softwaretechnikvorlesungen, das Softwarepraktikum als Indianer und ein 3-monatiges Praktikum in einem Betrieb; begleitend zur Pro-

jektleitertätigkeit gibt es einen wiederholenden Crashkurs in Projektmanagement sowie 14-tägige Supervisionssitzungen mit allen Masterstudenten.

Konkret investiert jeder Bachelor 240 Stunden (8 ECTS-Punkte) und jeder Masterstudent 120 Stunden (4 ECTS-Punkte). Die Teams werden vom Betreuer zufällig zusammengestellt, wobei darauf geachtet wird, dass jedes Team ähnliche Voraussetzungen hat und die Interessengebiete der Studenten alle Tätigkeiten abdecken. Die Aufgabe wird vom Kunden vorgegeben – in den letzten beiden Jahren waren dies Webanwendungen. Diese Aufgabe wird im Sommersemester in 16 Wochen konkurrierend bearbeitet. Jeder Bachelorstudent übernimmt eine Rolle im Team (Analytiker, Architekt, Chefprogrammierer, Technologe, Tester, Integrator, PR, GUI-Spezialist, Weiterbildungsbeauftragter, Handbuch etc.), was bei uns bedeutet, dass er für diesen Aufgabenbereich der Projektleitung zuarbeitet (Arbeitspakete formulieren etc.). Die Rolle entspricht damit nicht dem Arbeitsgebiet des Studenten – so muss z.B. jeder einen Anteil an der Programmierung beitragen. Das bedeutet auch, dass wichtige Aufgaben wie die Architektur – analog zu realen Projekten – nicht ein Aufgabenpaket eines Bachelors, sondern mit Chefsache sind [Rainwater 2002]. Der Ablauf orientiert sich am Unified Process, und die vorgegebenen Meilensteine sind Anforderungsspezifikation, GUI-Prototyp, Entwurf, Unit-Test der Kernfunktionalität, 80 %-Systemtest und Abschlusspräsentation. Die Betreuung ist eher schlank und beschränkt sich auf Meilensteinabnahme, per Gutschein limitierte Kundengespräche [Schneider 2005], öffentliche Reviewsitzungen und 14-tägige Statussitzungen mit allen Masterstudenten. Neben der erstellten Software samt der dazugehörigen Dokumente wurden vom Team wöchentliche Projektberichte mit Zeitabrechnung abgegeben.

Die Benotung setzt sich bei den Bachelorstudenten zu gleichen Teilen zusammen aus der Bewertung der Dokumente (v.a. bzgl. Konsistenz) und Vorträge, der Produktbewertung durch den Kunden, einer Note für das persönliche Engagement (Arbeitsstunden, Arbeitspakete und einer Peer-Punktebewertung), einer Reflexion über das eigene Lernen und einer Zusammenfassung der Vorgänge im eigenen Rollenbereich. Bei den Masterstudenten gehen Dokumente, Produkt und Engagement zu je 20 %, der Projekt-/Qualitätsplan zu 30 % und eine Sesam-Simulation [Schneider 1994] zu 10 % ein.

Aus didaktischer Sicht sollen Studenten im ersten Durchgang als Bachelorstudent eigene Projekterfahrungen sammeln, sich in ihren Kompetenzen realistisch einschätzen lernen und durch Beobachten und Interaktion mit einem übergeordneten Projektleiter auch eine erste, eher passive Einschätzung des Projektmanagements entwickeln. Im zweiten Durchgang als Masterstudenten hingegen können sie auf diesen Erfahrungen aufbauen und durch aktive Betätigung Projektkompetenzen auf einer höheren Ebene erlangen. Dies entspricht dem klassischen didaktischen Spiralkonzept.

4 Bewertung der Lehrveranstaltung

Zur Beantwortung des Konzepts der Lehrveranstaltung wurden Fragebögen am Anfang und am Ende des Semesters ausgegeben. Durch einen individuellen Schlüssel können jeweils die beiden Fragebögen eines Studenten einander zugeordnet werden.

Konkret wurden die Befragungen in den Sommersemestern 2007 und 2008 durchgeführt. Von 2007 liegen 36 Bachelor- und 12 Master-Fragebögen vor. 2008 sind dies 50 bzw. 10 Fragebögen. Am Anfang wurden die Studenten gebeten, die Vorkompetenz bzgl. verschiedener Sozial-/Selbst- sowie fachlich-methodischer Kompetenzen auf einer Skala von 1 (sehr) bis 4 (kaum) einzuschätzen. Am Ende mussten sie nochmals die Vorkompetenz vor dem Projekt sowie den Lernerfolg beurteilen. Ferner wurden verschiedene Fragen zum Verlauf des Projekts gestellt.

Die Ergebnisse der beiden Vorher-Nachher-Befragungen sind vollständig in der am Ende stehenden Tabelle 3 enthalten. Im Weiteren werden die Ergebnisse dieser Befragungen vorgestellt und exemplarisch auch mit den Ergebnissen einer ähnlichen Befragung des Sopras 2006 ohne Masterprojektleiter verglichen.

4.1 Veränderte Wahrnehmung der Vorkompetenz

Zunächst interessiert uns an dieser Stelle, ob sich die Selbsteinschätzung der Vorkompetenz durch die Erfahrung im Softwarepraktikum ändert. Dies ist ein Indikator dafür, bei welchen Kompetenzen die Studierenden sich zunächst überschätzt haben bzw. welche Aspekte durch das Softwarepraktikum besonders gut gefordert werden, so dass die Vorkenntnisse über gemachte Erfahrungen verifiziert werden können. Die Umfrageergebnisse sind in Tabelle 3 dargestellt – Werte, auf die sich der folgende Text bezieht, sind grau hinterlegt.

Wenn man die beiden Bachelor-Jahrgänge gemeinsam auswertet, dann gibt es mit <5 % Fehlerwahrscheinlichkeit eine negativere Neueinschätzung bei Kostenschätzung, Schnittstellengestaltung, Begeisterungsfähigkeit, dem Zuhörenkönnen, Kommunikation mit Fachfremden und der Visualisierungskompetenz. Sogar mit <1 % Fehlerwahrscheinlichkeit gilt dies für die Oberflächengestaltung, Kundengespräche, andere Programmiersprachen als Java, Programmierrichtlinien, Unit-Tests, Abstraktionsvermögen, sichere Programmierkenntnisse, die Bereitschaft, Werkzeuge zu nutzen, selbstständiges Arbeiten, Neugier und die Bereitschaft, sich in Neues einzuarbeiten.

Bei den Masterstudenten liegen die Informationen nur von den zehn Studenten der 2008-Befragung vor. Mit <5 % Fehlerwahrscheinlichkeit erweisen sich hier Projektmanagement, Programmierrichtlinien und Einfühlungsvermögen als Kompetenzen mit realistischerer Korrektur der Selbsteinschätzung. Mit <10 % Fehler kann man noch die Fähigkeiten, selbstständig zu arbeiten und sich mitteilen zu können, hinzuzählen.

Deutlich erkennt man einen anderen Schwerpunkt bei den Masterstudenten. Interessant ist allerdings, dass alle Studenten ihre Fähigkeit, selbstständig arbeiten zu können, vor dem Projekt besser eingeschätzt haben, als sie dies rückblickend nach dem Projekt machen. Zusammen mit der guten Lerneinschätzung verdeutlichten die Ergebnisse, dass gerade die Teamarbeit auch das organisierte, zielstrebige Arbeiten des Einzelnen besonders gut schult.

4.2 Vergleich der Lernschwerpunkte

Beide Jahrgänge berücksichtigend wurden von den Bachelors die Oberflächengestaltung, Java, die Bereitschaft, Werkzeuge zu benutzen, die Fähigkeit selbstständig zu arbeiten, die Bereitschaft, sich in Neues einzuarbeiten, Hilfsbereitschaft und Verantwortungsbewusstsein besonders gut gelernt. (Diese Werte sind in Tabelle 3 ebenfalls grau markiert.)

Bei den Masterstudenten wurden besonders gute Werte bei den folgenden Kompetenzen erzielt: Projektmanagement, das Zuhörenkönnen, die Fähigkeit, sich mitteilen zu können, Hilfsbereitschaft, Durchsetzungsvermögen, die Fähigkeit zu leiten, Entscheidungsfähigkeit, Verantwortungsbewusstsein und die Fähigkeit zu deligieren.

Interessant ist, dass es sich auf den beiden Ebenen tatsächlich um unterschiedliche Lernschwerpunkte handelt – insbesondere sind dies bei den Masterstudenten Kompetenzen, die für Projektleiter wichtig sind.

Vergleicht man die Ergebnisse der Bachelors mit den Umfrageergebnissen von 44 Studenten einer ähnlichen Befragung 2006 bezüglich eines Softwarepraktikums ohne Masterstudenten, erkennt man, dass die meisten der Kompetenzen ähnlich gelernt wurden. Signifikante Unterschiede (Fehlerrate <11 %) sind lediglich bei der UML-Modellierung (2006: Mittelwert 2,62/Standardwert 0,85), Schichtenarchitektur (2006: 2,38/0,85) und Versionskontrolle (2006: 2,5/0,90) erkennbar, die in den neuen Veranstaltungen besser gelernt wurden. Lediglich die Kommunikation mit Fachfremden wurde mit <5 % Fehler schlechter gelernt. Bezüglich der Projektmanagementkompetenzen konnte kein signifikanter Unterschied zwischen 2006 und den Bachelors 2007/2008 festgestellt werden. Dies zeigt, dass die Einbeziehung von Masterstudenten das Lernen der Bachelors nicht verschlechtert – bei der Anwendung von relevantem fachlichen Wissen können sie sogar nachweislich davon profitieren.

Und zuletzt noch ein kurzer Blick auf die Unterschiede zwischen den Praktika 2007 und 2008 (in Tab. 3 fett dargestellt). Im jüngeren Projekt waren erstmals auch Medieninformatiker beteiligt, die Kunden waren externe Auftraggeber, und die Vorgaben für Präsentationen, Meilensteine etc. waren strenger. Vermutlich der letzte Punkt hat zu signifikant (<10 % Fehler) besserem Lernen von Unit-Tests, Visualisierung und Präsentation bei den Bachelors geführt. Ein schlechterer Lerneffekt beim sicheren Programmieren kann ebenfalls wie gewisse Turbulenzen

bei den Programmiersprachen darauf zurückgeführt werden, dass 2008 einige Teams mit Grails statt Java gearbeitet haben. Warum von den Bachelors Projektmanagement 2007 besser gelernt wurde, ist unklar. Bei den Masterstudenten wurden 2008 Präsentation, Kommunikation mit Fachfremden, Flexibilität und Begeisterungsfähigkeit signifikant besser gelernt.

4.3 Wichtigste Probleme im Projekt

Grundsätzlich ist das Softwarepraktikum so angelegt, dass die Aufgabe ein wenig zu komplex und zu umfangreich für die Bearbeitung in 15–16 Wochen ist. Dies soll Probleme und Konflikte im Team forcieren, da wir dadurch den Lerneffekt höher einschätzen. Tabelle 1 zeigt nun die prozentualen Antworten der Studenten auf die Frage, wo sie die größten Probleme bei der Arbeit am Projekt sahen.

Die meisten Nennungen hat der Punkt des fehlenden technischen Know-how erhalten. Dies ist einerseits verständlich, andererseits wurden die Studenten schon beim Start der Projekte dazu angehalten, sich durch den Cheftechnologen damit auseinanderzusetzen. Dies fand in vielen Projekten nicht im gewünschten Maß statt. Zusätzlich wurde allerdings beklagt, dass in der Anfangszeit nicht alle Studenten ausgelastet sind. Die zweithäufigste Nennung sind Kommunikationsprobleme: 2007 innerhalb des Teams, was 2008 durch die Probleme mit externen Kunden überlagert wurde.

	Befragung SS07		Befragung SS08	
Hauptprobleme	**Bachelor**	**Master**	**Bachelor**	**Master**
Zu komplexe Aufgabe	33%	33%	14%	40%
Kommunikation mit dem Kunden	3%	8%	70%	60%
Kommunikation im Team	50%	67%	30%	20%
Schnittstellenprobleme	17%	33%	36%	40%
fehlendes technisches Know-how	75%	50%	50%	60%
fehlende Motivation im Team	33%	42%	22%	20%
mangelhaftes Projektmanagement	3%	33%	20%	20%

Tab. 1 *Auswertung des Fragebogens bezüglich der größten Probleme in den einzelnen Projekten. Gezeigt wird die prozentuale Nennung durch die Teilnehmer.*

Bei den Bachelorstudenten folgt mit Abstand die fehlende Motivation der Teammitglieder, während die Kritik am Projektmanagement den letzten Platz einnimmt. Dies unterstreicht an dieser Stelle, dass die Projektleitung durch Masterstudenten bei den Bachelors sehr gut angenommen wird.

Bei den Masterstudenten sind die weiteren Punkte nahezu gleich gewichtet. Auffallend ist, dass die Komplexität der Aufgabe und Schnittstellenprobleme von

Mastern eher wahrgenommen werden. Und auch die selbstkritische Sicht der eigenen Leitungstätigkeit ist interessant.

4.4 Rollenverhalten der Masterstudenten

Um festzustellen, wie die Masterstudenten ihr Selbstverständnis als Projektleiter definieren, ließen wir diese Studenten ihren Führungsstil anhand von Eigenschaften charakterisieren. Diese sind den drei Prototypen »freundlicher Kollege«, »Choleriker« und »formaler Prozessler« aus [Vigenschow/Schneider 2007] entnommen. Tabelle 2 zeigt die Ergebnisse.

Auch hier sieht man, dass der Führungsstil sehr stark durch den freundlichen Kollegen bestimmt ist – eine Herangehensweise, die auch dem beobachteten Verhalten bei den früher durchgeführten selbstorganisierenden Softwarepraktika entspricht. Aber die Nennungen zu den anderen beiden Prototypen zeigen, dass darüber hinaus ein strengeres professionelles Leitungsverhalten angestrebt wird. Insbesondere eine genauere Auswertung der Arbeitszeiten und der Verteilung der Arbeitspakete zeigt, dass die Arbeit im Team großteils gleich verteilt wird – mit wenigen Ausreißern bei der Arbeitszeit nach oben und unten.

	SS07	SS08		SS07	SS08
Der freundliche Kollege			selbst aktiv mitmachen	17%	30%
verständig	42%	70%	appellierend	25%	20%
geduldig	75%	70%	»alle im selben Boot«	33%	60%
Der Choleriker					
fordernd	17%	0%	zielorientiert	83%	40%
bestimmt	0%	40%	»Nichts ist unmöglich«	25%	50%
Der formale Prozessler			bewährte Vorgehensw.	8%	20%
klare Vorgaben	17%	30%	Konflikte versachlichen	33%	30%
rational	58%	30%	»Erfolg durch Disziplin«	17%	10%

Tab. 2 *Auswertung des Fragebogens bezüglich des Selbstverständnisses der Projektleiter. Angaben sind in Prozent.*

Eine genauere Analyse zeigt, dass neun der 22 Masterstudenten laut Selbsteinschätzung überwiegend als freundlicher Kollege agiert haben. Jeweils einer fällt klar in die beiden anderen Prototypen. Die anderen 11 Studenten stellen Mischungen dar, wobei einer als ausschließliche Mischung des Cholerikers mit dem formalen Prozessler auffällt. Die Zuordnung der Eigenschaften zu den Prototypen war – wie auch die Prototypen selbst – den Studenten nicht bekannt.

5 Diskussion

Die hier beschriebene Veranstaltung ist so konzipiert, dass übliche Hierarchien aus der Arbeitswelt durch das Softwarepraktikum abgebildet werden. Die Ergebnisse zeigen die Akzeptanz der Struktur durch die Studenten. Die Häuptlinge werden tatsächlich als solche gesehen. Aus Gesprächen mit den Bachelors geht eine klare Erwartungshaltung an die Projektleiter hervor. In einem Fall war eine Mediationssitzung notwendig, weil der Master-Projektleiter dieser Erwartungshaltung nicht gerecht wurde. In den meisten Fällen haben die Projektleiter ihre Aufgabe ernst genommen und sich um eine professionelle Haltung bemüht. Da sie für dieses Projekt ebenfalls eine Note erhielten, haben wir subjektiv Einsatzbereitschaft und Verantwortungsgefühl erlebt, das weit über unsere Erfahrungen mit Tutoren in der Projektleitung hinausgeht.

Aus organisatorischer Sicht ließ sich die Veranstaltung unkompliziert durchführen. In der Übergangszeit der vergangenen zwei Jahre musste flexibel mit der Ressource der Projektleiter umgegangen werden. So gab es 2007 Projekte, in denen einzelne Masterstudenten sowohl Projektleitung als auch Qualitätssicherung übernahmen. 2008 wurde ein Qualitätssicherer immer zwei Projekten zugeordnet.

Es hat sich klar gezeigt, dass innerhalb einer Veranstaltung Häuptlinge und Indianer auf unterschiedlichen Ebenen lernen. Im Vergleich zu den gleichberechtigten Teams aus dem Jahr 2006 wurden keine substanziellen negativen Auswirkungen ersichtlich. Im Gegenteil profitieren die Bachelors fachlich von den Mastern.

Neben diesem Lernen auf unterschiedlichen Stufen, bei dem sich sowohl Bachelors als auch Masters auf ihre Ebene konzentrieren können und davon gegenseitig profitieren, sehen wir die Chance der vorgestellten Bachelor/Master-Kombination vor allem in der Tatsache, dass Studenten auf dem Weg zum Masterabschluss zwei Projekte in unterschiedlicher Funktion durchlaufen und ihr Agieren als Häuptling auf dem eigenen Erleben als Indianer aufbauen können.

Es sind viele Varianten des vorgestellten Konzepts denkbar. Insbesondere bei der Aufgabenverteilung zwischen den Masterstudenten eines Teams sehen wir noch Verbesserungspotenzial und werden im kommenden Semester mit einem Master für die fachliche Planung (Arbeitspakete) und einem für die Personalplanung arbeiten.

	Bachelorstudenten			Masterstudenten		
Kompetenz	**Start** Vorkmp	**Ende** Vorkmp	**Ende** Lerneff.	**Start** Vorkmp	**Ende** Vorkmp	**Ende** Lerneff.
Projekt-management	3,31/0,82 2,94/0,77	3,44/0,65 3,22/0,74	**2,08/0,65** **2,40/0,76**	2,50/0,85	3,25/0,97 3,40/0,70	2,08/0,90 1,80/0,63
Risiko-minimierung	3,53/0,70 3,12/0,66	3,50/0,56 3,21/0,73	2,75/0,87 2,94/0,76	3,40/0,52	3,67/0,65 3,20/0,91	2,75/0,87 2,78/0,44
Kostenschätzung	3,41/0,60 2,76/0,82	3,36/0,59 3,15/0,82	3,11/0,89 2.98/0,87	3,40/0,70	3,41/1,00 3,70/0,48	3,08/0,90 3,11/0,93
Prozessmodell	3,05/0,68 2,98/0,74	3,18/0,58 3,00/0,67	2,77/0,60 2,72/0,80	2,80/0,63	2,67/0,78 3,10/0,73	2,92/0,79 2,44/0,53
UML-Modellierung	2,61/0,73 2,36/0,75	2,72/0,70 2,5/0,66	2,33/0,89 2,31/0,83	2,50/0,53	1,92/0,79 2,40/1,07	2,75/0,87 2,67/0,87
GUI-Design	2,22/0,83 2,20/0,70	2,81/0,79 2,77/0,77	1,86/0,87 2,14/0,86	2,30/0,82	2,17/1,03 2,60/0,97	2,75/0,87 2,78/1,09
Kunden-gespräche	3,17/0,97 2,80/0,95	3,66/0,54 3,15/0,82	3,00/0,91 3,04/0,99	2,80/0,63	2,58/1,16 3,10/0,74	3,00/1,04 3,00/1,00
Schnittst.gest.	3,08/0,91 2,81/0,81	3,36/0,68 3,17/0,64	2,33/0,83 2,23/0,76	2,30/1,16	2,17/0,83 2,50/0,97	3,08/0,90 2,78/0,97
Schichtenar.	3,31/0,71 3,02/0,78	3,22/0,68 3,04/0,76	2,08/0,69 2,14/0,86	2,50/1,18	1,83/0,72 2,40/0,70	3,00/0,95 2,67/0,87
Entwurfsmuster	3,22/0,87 2,96/0,73	3,40/0,60 3,07/0,77	2,31/0,68 2,52/0,82	2,10/0,88	2,25/0,75 2,20/0,79	3,17/0,83 2,80/0,92
Java	3,11/0,75 2,39/0,86	3,22/0,80 2,66/0,79	**1,58/0,73** **2,04/0,73**	2,10/0,99	1,83/0,83 2,30/0,95	3,00/1,04 2,89/1,17
andere Prog.sp.	2,27/0,74 2,24/0,82	2,69/0,82 2,87/0,82	**2,92/0,87** **2,29/0,87**	1,50/0,71	1,92/0,51 1,90/0,74	3,17/1,03 2,89/1,17
Prog.richtlinien	2,86/0,76 2,55/0,84	3,11/0,75 2.98/0,74	2,31/0,75 2,39/0,73	1,80/0,42	1,92/0,67 2,70/0,67	2,75/1,14 2,78/1,09
Debugging	3,03/0,77 2,90/0,86	3,09/0,85 3,06/0,84	2,34/0,80 2,59/0,84	2,00/1,05	2,09/0,83 2,40/0,70	2,45/1,21 2,89/1,05
Unit-Tests	3,31/0,71 2,92/0,78	3,64/0,54 2,66/0,79	**3,03/0,84** **2,45/0,96**	3,20/0,63	3,42/0,67 3,20/0,63	2,83/0,94 2,30/1,06
Versionskontr.	3,42/0,81 3,12/0,80	3,56/0,73 2,87/0,82	2,06/0,95 2,33/0,94	2,30/0,95	2,08/0,792, 60/0,70	2,83/0,94 2,40/1,07
Abstraktionsver.	2,53/0,70 2,16/0,79	2,83/0,77 2,55/0,65	2,44/0,69 2,29/0,65	1,60/0,52	1,92/0,67 1,67/0,71	2,92/0,79 2,50/0,85
Formalisierung	2,97/0,65 2,78/0,82	3,03/0,65 2,85/0,59	2,67/0,63 2,71/0,74	2,20/0,63	2,00/0,74 2,00/0,50	2,83/0,93 2,60/0,97
Form. Meth. anwenden	2,67/0,682 ,64/0,75	2,81/0,67 2,81/0,61	2,47/0,88 2,67/0,66	2,20/0,42	2,17/0,83 2,22/0,44	2,83/0,83 2,80/0,79

	Bachelorstudenten			Masterstudenten		
Kompetenz	**Start** Vorkmp	**Ende** Vorkmp	**Ende** Lerneff.	**Start** Vorkmp	**Ende** Vorkmp	**Ende** Lerneff.
Exaktheit	2,60/0,74 2,62/0,67	2,81/0,79 2,79/0,66	2,47/0,81 2,51/0,71	2,20/0,63	2,17/0,83 2,11/0,78	2,58/0,90 2,40/0,97
sichere Progr.	2,50/0,88 2,38/0,88	2,83/0,74 2,87/0,80	**1,86/0,68** **2,24/0,75**	1,80/0,79	1,75/0,75 1,78/0,67	2,75/1,06 2,80/1,23
Werkzeuge nutzen	2,06/0,67 1,88/0,72	2,75/0,73 2,55/0,85	2,00/0,99 1,90/0,77	1,70/0,95	1,75/0,62 1,89/0,93	2,17/0,83 2,30/1,05
Problemlösen	2,58/0,65 2,10/0,74	2,40/0,74 2,57/0,71	1,97/0,70 2,24/0,72	1,90/0,88	2,41/0,67 1,78/0,67	2,50/0,80 2,10/0,74
Selbstständig arbeiten	1,94/0,79 1,70/0,54	2,08/0,73 2,24/0,77	1,92/0,87 2,02/0,70	1,40/0,84	1,50/0,67 2,22/0,67	2,83/1,19 3,00/0,94
Zeitmanagement	3,00/0,68 2,60/0,83	2,92/0,60 2,81/0,77	2,17/0,77 2,27/0,76	2,70/0,95	2,33/0,98 3,00/0,50	2,41/0,67 2,20/1,14
Geduld	2,36/0,932 ,24/0,82	2,19/0,58 2,34/0,94	2,19/0,98 2,08/0,81	1,60/0,97	2,58/0,67 2,33/0,87	2,67/0,89 2,20/0,92
Ausdauer	2,22/0,83 2,12/0,75	2,25/0,73 2,36/0,79	1,89/0,75 2,18/0,81	2,00/0,67	2,33/0,78 2,00/0,71	2,58/0,79 2,20/0,92
Neugier	1,47/0,56 1,84/0,79	1,97/0,94 2,15/0,86	2,17/0,91 2,20/0,97	1,50/0,53	1,92/0,67 1,67/0,87	2,75/0,87 2,10/0,99
Neues einarbeiten	1,75/0,60 1,94/0,89	2,03/0,74 2,42/0,83	1,67/0,59 1,88/0,78	1,40/0,52	1,50/0,52 1,89/0,93	2,25/0,97 2,30/0,82
Begeisterung	1,86/0,76 1,94/0,79	2,17/0,74 2,32/0,86	2,14/0,80 2,22/1,01	1,60/0,70	1,75/0,75 1,44/0,73	**2,75/1,06** **2,00/0,94**
Kreativität	2,36/0,87 2,26/0,80	2,50/0,94 2,30/0,78	2,39/0,96 2,45/0,89	2,30/0,95	2,58/0,67 2,11/0,78	2,92/1,00 2,70/0,95
Flexibilität	2,17/0,77 2,30/0,76	2,37/0,73 2,32/0,73	2,00/0,69 2,08/0,76	1,80/0,63	2,17/0,72 1,89/0,33	**2,67/0,89** **1,90/0,99**
zuhören können	1,77/0,68 1,80/0,72	2,14/0,88 2,09/0,69	2,11/0,76 2,10/0,78	1,60/0,70	2,00/0,74 2,22/0,97	2,17/0,58 1,90/0,99
sich mitteilen können	2,36/0,76 2,20/0,88	2,46/0,85 2,54/0,78	2,20/0,68 2,13/0,70	1,70/0,82	2,33/0,78 2,44/0,88	2,33/0,65 1,80/0,79
Kom. mit Fachfremden	2,39/0,90 2,34/0,87	2,69/0,68 2,61/0,74	2,91/0,85 2,58/0,92	1,90/0,88	2,42/0,90 2,33/0,71	**2,92/0,79** **2,20/0,92**
Einfühlungs-vermögen	2,44/0,65 2,46/0,97	2,43/0,65 2,50/0,69	2,60/0,88 2,42/0,85	1,70/0,67	2,42/0,79 2,56/0,88	2,67/1,07 2,10/0,99
Integration im Team	2,31/0,82 2,08/0,67	2,39/0,69 2,22/0,70	1,89/0,57 2,15/0,68	1,90/0,57	2,42/1,00 2,11/0,60	2,33/0,98 2,40/0,70
Kritikfähigkeit	2,28/0,70 2,16/0,65	2,33/0,76 2,20/0,65	2,14/0,59 2,19/0,82	2,10/0,74	2,42/1,00 2,22/0,97	2,67/0,78 2,10/0,88

	Bachelorstudenten			Masterstudenten		
Kompetenz	**Start** Vorkmp	**Ende** Vorkmp	**Ende** Lerneff.	**Start** Vorkmp	**Ende** Vorkmp	**Ende** Lerneff.
Hilfsbereitschaft	1,81/0,58 1,90/0,81	1,92/0,84 2,07/0,71	1,92/0,69 2,06/0,73	1,50/0,53	1,58/0,67 1,89/0,33	2,42/1,00 1,90/0,99
Durchsetzungs-vermög.	2,56/0,65 2,46/0,84	2,64/0,83 2,59/0,80	2,28/0,85 2,29/0,74	1,70/0,67	2,58/0,90 2,33/0,87	2,17/0,94 2,20/0,92
Kompromiss-bereit.	2,14/0,64 2,18/0,69	2,22/0,59 2,28/0,58	1,92/0,69 2,17/0,72	2,10/0,57	2,25/0,75 2,11/0,60	2,25/0,62 2,50/0,85
Visualisierungs-komp.	2,44/0,77 2,32/0,68	2,74/0,66 2,57/0,65	**2,66/0,73** **2,29/0,68**	2,20/1,03	2,08/0,67 2,56/0,88	2,83/0,94 3,00/0,94
Präsentations-komp.	2,78/0,80 2,42/0,86	2,92/0,60 2,63/0,71	**2,78/0,76** **2,41/0,82**	2,10/0,74	2,00/0,43 2,22/0,83	**3,17/0,83** **2,40/1,07**
Arbeit verteidigen	2,44/0,77 2,16/0,65	2,44/0,69 2,35/0,67	2,22/0,72 2,17/0,83	1,80/0,63	2,00/0,95 2,11/0,78	2,83/1,11 2,30/1,06
Fähigkeit zu leiten	2,92/0,91 2,72/0,88	2,97/0,70 2,93/0,74	2,89/0,89 2,73/0,96	2,30/0,95	3,00/0,95 2,33/0,71	1,92/0,90 2,00/0,82
Entscheidungs-fähigkeit	2,44/0,77 2,60/0,70	2,50/0,51 2,61/0,74	2,14/0,72 2,33/0,72	2,10/0,99	2,50/1,00 2,44/0,88	2,25/0,75 2,10/0,99
Deligieren können	2,89/0,82 2,66/0,82	3,03/0,51 2,78/0,66	2,56/0,81 2,60/0,84	2,30/0,95	2,67/0,78 2,78/0,83	2,17/0,58 1,90/0,99
Risikobereitschaft	2,89/0,67 2,56/0,70	2,89/0,71 2,72/0,66	2,67/0,89 2,53/0,83	2,70/0,82	2,92/0,51 3,00/0,87	2,58/0,90 2,40/0,70
Verantwortungs-bewusstsein	2,22/0,72 2,05/0,67	2,33/0,63 2,28/0,69	2,03/0,77 1,98/0,67	2,00/0,94	2,17/0,72 2,11/0,93	2,08/0,90 2,30/0,95
Systemdenken	2,56/0,65 2,33/0,77	2,53/0,70 2,52/0,72	2,11/0,85 2,29/0,74	2,10/0,57	2,18/0,98 2,67/0,71	2,73/0,79 2,50/0,71
Konflikte managen				2,30/0,95	2,75/0,97 2,22/0,97	2,17/0,94 2,30/1,05
Teambildung				2,20/0,92	3,08/0,79 2,44/1,01	1,92/0,90 2,40/1,07

Tab. 3 *Auswertung des Fragebogens bzgl. der Kompetenzen. Start = Befragung vor dem Softwarepraktikum, Ende = Befragung nach dem Softwarepraktikum, Vorkmp = Vorkompetenz, Lerneff. = Lerneffekt. Es ist jeweils in der ersten Zeile der Mittelwert und die Standardabweichung für die Befragung 2007 und in der zweiten Zeile analog für 2008 dargestellt. Werteskala: 1 (sehr) bis 4 (kaum). Auf die grau hinterlegten bzw. fett gedruckten Werte wird im Text direkt Bezug genommen.*

Literatur

[ASIIN 2005] ASIIN Fachausschuss Informatik: Fachspezifisch ergänzende Hinweise. (Akkreditierungsagentur für Studiengänge der Ingenieurwissenschaften, der Informatik, der Naturwissenschaften und der Mathematik e.V.), 2005.

[ASIIN 2006] ASIIN: Antrag auf Reakkreditierung durch den Akkreditierungsrat. (Akkreditierungsagentur für Studiengänge der Ingenieurwissenschaften, der Informatik, der Naturwissenschaften und der Mathematik e.V.), 2006.

[BMBF 2005] Bundesministerium für Bildung und Forschung: Bachelor- und Master-Studiengänge in ausgewählten Ländern Europas im Vergleich zu Deutschland, 2005.

[Demuth et al. 2002] B. Demuth, M. Fischer, H. Hussmann: Experience in early and late software engineering project courses. In Proc. 5th Conference on Software Engineering Education Training (CSEE&T 2002), S. 241–248, IEEE Computer Society, 2002.

[Fincher et al. 2001] S. Fincher, M. Petre, M. Clark (Herausgeber): Computer Science Project Work: Principles and Pragmatics. Springer, London, Berlin, 2001.

[Lindig/Zeller 2005] C. Lindig, A. Zeller: Ein Softwaretechnik-Praktikum als Sommerkurs. In: K.-P. Löhr, H. Lichter (Herausgeber), Software Engineering im Unterricht der Hochschulen SEUH 2005, S. 68-80, Heidelberg, dpunkt.verlag, 2005.

[Ludewig/Reißing 1998] J. Ludewig, R. Reißing: Teaching what they need instead of teaching what we like – the new software engineering curriculum at the University of Stuttgart. Information & Software Technology, 40(4):239–244, 1998.

[Rainwater 2002] J.H. Rainwater: Herding Cats: A Primer for Programmers Who Lead Programmers. Apress, Berkeley, 2002.

[Schlimmer et al.1994] J. C. Schlimmer, J. B. Fletcher, L. A. Hermens: Team-oriented software practicum. IEEE Trans. Educ., 37:212–220, 1994.

[Schneider 1994] K. Schneider: Komm, wir spielen Projektleiter! – Ein Lehrspiel für Software Engineering. In: H. Hußmann, B. Paech, Software Engineering im Unterricht der Hochschulen SEUH '94, S. 118-128, Stuttgart, Teubner, 1994.

[Schneider 2005] K. Schneider: Vier Formen der Erfahrungsvermittlung im Studium. In: K.-P. Löhr, H. Lichter (Herausgeber), Software Engineering im Unterricht der Hochschulen SEUH 2005, S. 16-25, Heidelberg, dpunkt.verlag, 2005.

[Stoyan/Glinz 2005] R. Stoyan, M. Glinz: Methoden und Techniken zum Erreichen didaktischer Ziele in Software-Engineering-Praktika. In: K.-P. Löhr, H. Lichter (Herausgeber), Software Engineering im Unterricht der Hochschulen SEUH 2005, S. 2-15, Heidelberg, dpunkt.verlag, 2005.

[Vigenschow/Schneider 2007] U. Vigenschow and B. Schneider: Soft Skills für Softwareentwickler. dpunkt.verlag, Heidelberg, 2007.

[Weicker et al. 2006] N. Weicker, B. Draskoczy, K. Weicker: Fachintegrierte Vermittlung von Schlüsselkompetenzen der Informatik . In: P. Forbrig, G. Siegel, M. Schneider (Herausgeber), HDI 2006: Hochschuldidaktik der Informatik, S. 51-62, Bonn, Gesellschaft für Informatik, 2006 .

Erfahrungen bei der Lehre des Software Engineering

Jochen Ludewig
Universität Stuttgart
ludewig@informatik.uni-stuttgart.de

Zusammenfassung

Dieser Beitrag stellt einige Erfahrungen zusammen, die ich in 25 Jahren Lehrpraxis im Software Engineering gesammelt habe. Die wichtigsten Punkte sind in zwölf Thesen formuliert. Ich hoffe, auf diese Weise jüngeren Kolleginnen und Kollegen Anregungen zu geben und der weiterhin notwendigen Diskussion über die Lehre des Software Engineering Nahrung zu liefern.

1 Hintergrund

Im Dezember 1983 erreichte mich die Anfrage, ob ich bereit sei, im Neu-Technikum Buchs, einer Fachhochschule in der Ostschweiz, eine Vorlesung über Software Engineering zu halten. Das habe ich gern getan. Zwei Jahre später war ich Assistenzprofessor für das Fachgebiet Software Engineering an der ETH Zürich, und seit 1988 sitze ich auf einem Lehrstuhl für Software Engineering an der Universität Stuttgart.

Damit blicke ich auf ein Vierteljahrhundert Lehre im Software Engineering (SE) zurück. Die interessanteste Zeit begann 1995, als in Stuttgart der neue Studiengang Softwaretechnik konzipiert, eingeführt und verbessert wurde [Ludewig 2008]. Nachfolgend trage ich einige positive und negative Erfahrungen zusammen. Vielleicht sind sie für jüngere Kolleginnen und Kollegen nützlich; mindestens sollten sie eine Diskussion provozieren.

Es liegt in der Natur einer Erfahrungssammlung, dass die Feststellungen auf Beobachtungen beruhen, die reflektiert, aber nicht wissenschaftlich, also durch eine sie stützende Literatur oder empirische Untersuchungen, begründet sind.

Auch erhebe ich nicht den Anspruch, völlig neue Einsichten zu präsentieren oder solche, die *nur* für unser Fachgebiet gelten. Die Aussagen scheinen mir wichtig und bislang nicht allgemein anerkannt; das ist Grund genug, sie vorzutragen.

2 Der Reiz eines jungen Fachs

Es ist üblich geworden, das Jahr 1968, genauer: die NATO-Konferenz in Garmisch, als Geburtsdatum des Software Engineering zu betrachten [Randell 2001]. Genau genommen fand aber in Garmisch nur die Zeugung statt; geboren wurde das SE einige Jahre später, als Boehms wegweisende Artikel erschienen [Boehm 1973, 1976]. Vorher war der Begriff kaum ernst genommen worden. Wer die Berichte von Brian Randell und anderen Pionieren des Gebiets auf der ICSE 2008 in Leipzig gehört hat, wird sich erinnern, dass es zunächst gar nicht danach aussah, als ob das SE Zukunft hätte. Die Nachfolgekonferenz in Rom 1969 war ein Flop und veranlasste viele der Beteiligten, die Sache nicht weiter zu verfolgen. Dass ein paar Jahre später trotzdem ein neues Forschungs- und Lehrgebiet mit Zeitschriften, Tagungen und Lehrstühlen entstand, lag vor allem am starken Bedarf der Praxis, aber auch am Wunsch einiger der Pioniere, einen Heimathafen für ihre Forschung zu haben, die sonst nirgends richtig hineinpasste. (Man schaue sich an, in welchen Zeitschriften und auf welchen Konferenzen die frühen, heute zum SE gerechneten Arbeiten erschienen sind!)

Ungeklärt blieb dabei, was SE eigentlich sein sollte. Diese Unklarheit war kein Mangel, sondern ein Glück. Denn so konnten sich mathematisch geprägte Forscher wie Bauer, empirisch arbeitende wie Boehm und konstruktiv denkende wie Parnas unter dem Schirm des SE versammeln.

Daraus folgt **These 1**:
SE vereint ganz unterschiedliche Auffassungen, Problemsichten und Lösungsansätze. Wir sollten nicht versuchen, eine klare Definition zu schaffen, die Teile der Gemeinschaft ausgrenzt. SE ist ein Vielvölkerstaat. Wir sollten auch die fruchtlose Diskussion einstellen, was SE *wirklich* ist. SE ist, was wir, seine *Professionals*, darunter verstehen. Dazu gehört natürlich auch, dass wir, trotz unterschiedlichen Positionen und Schwerpunkten, einander respektieren und akzeptieren.

Die Möglichkeit, die allermeisten Pioniere persönlich zu treffen und zu sprechen, zeichnet das SE vor fast allen anderen Teilgebieten der Informatik und noch weit deutlicher vor den übrigen Wissenschaften aus. Dies wurde nirgends besser sichtbar als 2002 in Bonn, wo sich auf Einladung von sd&m viele der Pioniere trafen [Broy/Denert 2002].

Der persönliche Kontakt mit den Großen des Fachs bewirkt vor allem, dass man in allen Entwicklungen die menschliche Dimension sieht. Die Erfindungen,

Ideen und Irrtümer sind nicht vom Himmel gefallen, sondern in den Köpfen denkender Personen entstanden. Das zu wissen, gibt mir selbst die Zuversicht, Beiträge leisten zu können und irren zu dürfen.

Dies führt zur **These 2**:
Wer aktiv im SE mitmischen will, sollte wissen, dass es überall menschelt, auch bei den Koryphäen. Darum sollte keine Lehre des SE darauf verzichten, die Protagonisten vorzustellen und ihre Leistungen und Irrtümer zu diskutieren. Am besten ist es natürlich, sie in den eigenen Hörsaal zu holen. Das gilt ebenso für die Urheber aktueller Ideen und Konzepte, z.B. Entwurfsmuster, agile Prozesse und Model Driven X. Sollte sich dabei zeigen, dass *der Kaiser gar keine Kleider anhat*, ist auch das sehr erhellend.

Die Möglichkeit, die Pioniere zu treffen, schwindet; seit dem Bonner Symposium sind wichtige Leute gestorben (Dijkstra, Dahl, Nygaard). Aber andere (wie Parnas, Brooks, Boehm, Jackson und auch Bauer) sind weiterhin aktiv und oft auch auf Konferenzen anzutreffen.

3 Praktika und Projekte

Praktika und Projekte sind für die SE-Lehre zentral. Sie sollten präzise auf die übrigen Lehrveranstaltungen abgestimmt sein (oder besser: die anderen auf sie). In Stuttgart gibt es die *Studienprojekte*, in denen acht bis zehn Studenten zwölf Monate lang an einer konkreten Aufgabe arbeiten, von der Erhebung der Anforderungen bis zur Ablieferung des Produkts.

Das halbe Dutzend solcher Projekte, die unsere Abteilung bislang angeboten hat, war für alle Beteiligten sehr lehrreich. Wir sind uns heute einigermaßen sicher, dass

- Studenten mit ausreichenden SE-Vorkenntnissen auch anspruchsvolle Probleme lösen können und in der Regel tatsächlich lösen,
- die Aufgabe unbedingt von außen vorgegeben sein muss, im besten Fall aus einem Bereich, der den Teilnehmern völlig fremd ist,
- die Rolle des Kunden mit der eines Betreuers definitiv inkompatibel ist,
- ein Vorgehensmodell auf der Basis eines »iterierten Wasserfalls« (7 Monate bis zur Ablieferung eines Kernprodukts, 5 Monate für die Verbesserung und Ergänzung) in den meisten Fällen gute Aussichten auf Erfolg bietet,
- die Projekte meist nach einem Vierteljahr in eine Krise geraten, weil sich zeigt, dass die Kooperationstechniken aus kleinen Projekten nicht mehr ausreichen,
- jedes Projekt durch die Teilnehmer, vor allem durch den Projektleiter, eine andere Kultur entwickelt, die sich im Selbstbewusstsein der Gruppe, in der Kooperation und in der Kommunikation niederschlägt,

- technologische Probleme keine große Rolle spielen, da die Studenten in der Regel auch mit neuer Technologie nach kurzer Zeit klarkommen,
- das Verhältnis zum Kunden dagegen schwierig bleibt, weil die Studenten unsicher sind, was sie fragen, vorschlagen oder selbst entscheiden sollen,
- vielen Studenten die Vermarktung ihrer Lösung abwegig erscheint,
- es keineswegs notwendig, sondern eher schädlich ist, als Veranstalter des Projekts schon vorher *eine* oder gar *die* Lösung der Aufgabe zu kennen.

These 3:
Projekte müssen sein, sukzessive vom Programmierkurs über Kleingruppen-Projekte bis hin zu solchen, in denen Projektmanagement und geregelte Kommunikation erforderlich sind. Die Studenten können dabei alle Rollen übernehmen außer der des Kunden. Auch Betreuer werden benötigt; sie schauen den Teilnehmern auf die Finger und geben ihnen gelegentlich, gefragt oder ungefragt, Ratschläge.

Unser aktuelles Projekt läuft anders als die früheren, weil wir diesmal im Gegentakt liegen, also nicht im Herbst, sondern im Frühjahr begonnen haben. Praktisch heißt das: Die Teilnehmer haben die zweite SE-Vorlesung mit dem Schwerpunkt Projektmanagement noch nicht gehört. Das erweist sich als Handicap. Vorlesung und Projektbeginn im selben Semester sind optimal. Es ist nicht klar, ob die technikzentrierte Sicht der Teilnehmer, die wir in diesem Projekt beobachten, ebenfalls mit dem Mangel an Managementwissen zu tun hat.

These 4:
Projekte und Vorlesungen zum Projektmanagement und zu den eingesetzten Techniken sollten simultan stattfinden. Die »Einheit von Theorie und Praxis«, im real existierenden Sozialismus gern beschworen: Hier ist sie möglich und nötig.

Wir haben außerdem zum ersten Mal ein Projekt, bei dem die Teilnehmer einem extremen Zeitdruck ausgesetzt waren. Es hat sich gezeigt, dass sie darauf nicht vorbereitet sind und in dieser Situation nicht immer sinnvoll entscheiden.

Eine in diesem Zusammenhang aufgeflammte Diskussion in unserer Abteilung befasste sich mit der Frage, wie realistisch Projekte sein können und sollten. Die spontane Antwort »So realistisch wie möglich« trägt nicht. Wir müssen uns darüber klar sein, dass wir die Realität nicht nachstellen können und auch nicht *wollen*. Praktisch kein Softwaresystem ist wirklich fertig, wenn es ausgeliefert wird. Für den Erfolg der Software ist gerade wichtig, wie sie nach der Auslieferung betreut und gewartet wird. Darum ist ein Projekt, an dessen Ende eine fertige, also korrekte und brauchbare, Software steht, offensichtlich unrealistisch.

Aber die Wartungsphase lässt sich beim besten Willen nicht in unsere Projekte integrieren. Wo also liegt das Optimum?

Und natürlich wollen wir all die – nach unserer festen Überzeugung – kontraproduktiven Randbedingungen *nicht* kopieren, die in der Praxis nicht selten vorkommen, beispielsweise unsinnige Vorgaben des Managements, hohe Fluktuation usw.

These 5:
Projekte in der Hochschule bleiben selbst im Idealfall verschieden von realen Projekten außerhalb der Hochschule. Es ist Sache der Lehrenden, die Ähnlichkeiten – so weit möglich und sinnvoll – herzustellen und die Unterschiede zu reflektieren und mit den Teilnehmern zu diskutieren.

Im Bachelor-Studienplan (ab 2009) gibt es, wenn unsere Vorschläge die Gremien passieren, die Studienprojekte unverändert, allerdings folgt nicht mehr wie bisher nach dem ersten in der Informatik-Fakultät noch ein zweites in einem Anwendungsfach.

4 Prüfungen und Noten

Software Engineering ist sehr einfach. Was wollen wir denn vermitteln? Es sind doch banale Weisheiten, beispielsweise die, dass man ein neues Artefakt nicht weitergeben sollte, ohne es zuvor geprüft zu haben. Das ist *schrecklich* einfach, weil jedes Kind die Aussage begreift; dagegen braucht das tiefe Verständnis und die Aufnahme in den eigenen Wertekanon Monate oder Jahre. Ähnliches kann man über Planung, Dokumentation, Strukturierung und andere Aspekte sagen. Ob jemand ein Integral berechnen kann, lässt sich leicht prüfen. Aber wie prüft man, ob ein Student begriffen hat, dass Planung wichtig ist? Wenn wir so etwas abfragen, bekommen wir Absolventen, die den SE-Katechismus auswendig gelernt haben, aber keine Software-Ingenieure.

Diese Situation führt dazu, dass wir entweder nicht prüfen, was wir lehren, oder nicht lehren, was wir lehren sollten. Tatsächlich habe ich von einem Kollegen als Begründung für seine Stoffauswahl gehört: »Schließlich muss das ja auch prüfbar sein!« Hier wedelt offensichtlich der Schwanz mit dem Hund.

These 6:
Wir brauchen neue Verfahren, um Studenten zu prüfen und ihre Leistungen zu bewerten. Denn Klausuren und mündliche Prüfungen, wie wir sie bislang praktizieren, sind ungeeignet, das tiefe Verständnis zu kontrollieren. Vielleicht können wir (in Anlehnung an Prüfungen in Maschinenkonstruktion) in der Prüfung komplexe Szenarien vorgeben, die mehr als eine schematische Reaktion erfordern.

Bei Projekten fällt es leicht, die Leistung der Gruppe insgesamt zu bewerten. Viel schwieriger ist die (durch die Prüfungsordnung gestellte) Aufgabe, individuelle Noten zu verteilen. Wir haben das Problem nicht gelöst, aber entschärft, indem wir jeweils zu Beginn das Verfahren erklären: Eine kollektive Note wird festgelegt und dann für Teilnehmer, die (positiv oder negativ) aufgefallen sind, durch ein angemessenes Delta korrigiert. Das ergibt die individuellen Noten.

5 Industrie und Software-Engineering-Lehre

Auf den ersten Blick scheint es geradezu ideal, den Studenten Praxiserfahrungen zu vermitteln, indem man ihnen Prüfungsarbeiten (insbesondere Diplomarbeiten) gibt, die ganz oder zu einem erheblichen Teil in der Industrie stattfinden. (»Industrie« steht hier für alle Organisationen, die gewinnorientiert arbeiten und nicht der Forschung oder Lehre dienen.)

Unsere Erfahrungen mit solchen Arbeiten sind überwiegend negativ. Vor allem drei Punkte haben immer wieder Schwierigkeiten gemacht:

1. Viele Studenten suchen gezielt nach Industrie-Arbeiten (vor allem Diplomarbeiten), weil sie die Bezahlung brauchen, die die Industrie meist in Aussicht stellt. Damit stehen außerfachliche Gesichtspunkte bei der Wahl der Arbeit im Vordergrund. Und aus Sicht der Studenten gibt es Diplomarbeiten erster und zweiter Klasse.
2. Die Industrie will vor allem verwertbare Resultate. In vielen Fällen ist dies ausführbarer Code. Kommt die Arbeit nicht so schnell voran wie ursprünglich geplant (eine sehr oft auftretende Situation!), so wird die Codierung von Industrieseite forciert, auch dann, wenn die Grundlagen dafür längst nicht geklärt sind.
3. Selbst wenn völlig klar ist, dass die Industrieseite keinen Einfluss auf die Benotung der Arbeit hat, wollen die Diplomanden es ihren Betreuern (und in der Regel Gönnern) recht machen. Sie treffen darum Entscheidungen, die nicht sachlich begründet, aber von den Partnern in der Industrie gewünscht sind. (Typische Formulierungen: »Das habe ich nicht näher untersucht, weil sich das dort ohnehin nicht durchsetzen lässt.«)

Diese Erfahrungen, die nicht auf meinen Lehrstuhl beschränkt sind, haben dazu geführt, dass die gesamte Stuttgarter Informatik inzwischen Arbeiten mit externen Partnern sehr kritisch gegenüber steht.

Wir haben daraus die Konsequenz gezogen, dass wir extern durchgeführte Arbeiten in der Regel ablehnen (also das Thema nicht ausgeben und die Arbeit entsprechend auch nicht prüfen). Das gilt insbesondere, wenn ein Student mit einem Thema hausieren kommt. Dagegen führen wir solche Arbeiten durch, wenn die Industrie uns anspricht, wir am Thema interessiert sind, die Bedin-

gungen der Arbeit vollständig vorgegeben sind und wir selbst eine Person suchen, die das Thema bearbeiten will und kann.

These 7:
Bei Prüfungsarbeiten, die außerhalb der Hochschule durchgeführt werden, ist das Risiko groß, dass Aspekte, die nichts mit dem Studium zu tun haben (sollten), einen negativen Einfluss auf den Verlauf und die Ergebnisse haben. Prüfer sollten ihrer Verantwortung gerecht werden, indem sie sicherstellen, dass solche Einflüsse ausgeschlossen bleiben. Das läuft auf eine sehr restriktive Ausgabepraxis hinaus.

Ganz anders ist die Lage, wo die Kooperation von vornherein vorgesehen ist. Unsere sogenannten Fachstudien (Untersuchung einer zu klärenden Frage, meist in der Industrie, durch eine Gruppe von drei Studenten) sind ein Schlager geworden. Die Industrie freut sich über die meist wirklich kompetente Beratung und über den Kontakt zu Studenten, die bald eine Stelle suchen werden. Die Studenten können das Gelernte anwenden und finden interessierte Zuhörer, die das Ergebnis wirklich brauchen. Obwohl es keine Note, sondern nur einen Schein gibt, haben wir noch nicht erlebt, dass dünne Bretter gebohrt werden.

6 Entscheiden lernen

Wissenschaft ist radikal. Das ist eine ihrer wichtigsten Eigenschaften. Ein Resultat ändert sich nicht dadurch, dass es erwünscht oder bedrohlich ist. Im SE fehlt es oft an Radikalität; immer wieder hört man in der Praxis Aussagen wie: »Ja, wir wissen, dass wir das anders machen sollten. Aber es lässt sich nicht so einfach durchsetzen.« Und Resultate, die klare Implikationen haben, sind keineswegs populär. Wir haben immer wieder gesehen, dass in Industriekooperationen Analysen willkommen, klare Worte aber unerwünscht sind.

Auch unsere Studenten schrecken – aus ganz anderen Gründen – vor klaren Aussagen zurück, denn sie setzen sich damit der Kritik aus. Aber wenn sie die Verhältnisse in der Praxis verändern, verbessern sollen, müssen sie genau solche Feststellungen treffen und vertreten können. Wir sollten uns also nicht mit lauwarmen Formulierungen abfinden, die alle Entscheidungen offenlassen. Die Studenten müssen schon bei uns lernen zu entscheiden.

These 8:
Es reicht nicht aus, das Analysieren der Probleme zu lehren und einzuüben. Wir müssen auch verlangen (und selbst vormachen), dass am Ende einer Analyse eine Wertung steht, die zu einer Entscheidung führt.

7 Deutsch

Da ich seit Jahren auch die Funktion habe, als »das Praktikantenamt« die Berichte der Studenten entgegenzunehmen und zu prüfen, sehe ich viele von Studenten verfasste Texte, meist auf Deutsch. Jedenfalls halten die Urheber das, was sie abliefern, für Deutsch. Ich sehe – bei beträchtlicher Streuung – eine stetige Abnahme der sprachlichen Kompetenz. Am schlimmsten ist die Interpunktion, dann kommen Rechtschreibung und Satzbau. Von Stil zu sprechen, erscheint fast als Witz. Ich erinnere mich noch an meinen Schock, als ich zum ersten Mal einen Satz der Art »Mir viel auf das ...« sah.

Solche Fehlleistungen beschränken sich nicht auf die Studenten; mangelhaftes Deutsch hören und sehen wir heute auch bei Politikern, Journalisten und Professoren. In vielen Briefen werde ich mit »sie« angeredet, und »wegen« mit Dativ ist ebenso wie »dank« mit Genitiv Standard (oder, wie es immer öfter heißt, »Standart«).

Der beliebte Hinweis auf Unsicherheiten, die durch die Rechtschreibreform entstanden sind, geht völlig an der Realität vorbei: Die Texte sind so falsch, dass sie keiner deutschen Rechtschreibung seit Luther zugeordnet werden können.

Ich weiß einfach nicht, wie ich damit umgehen soll. Es gibt offensichtlich zwei klare, einfache Standpunkte. Wir können entweder verlangen, dass die Regeln eingehalten werden. Oder wir können alles akzeptieren, was bei geeigneter Transkription irgendeinen Sinn ergibt. Die zweite Möglichkeit erscheint mir indiskutabel. Über Jahrhunderte gab es einen Konsens, dass der Schreiber Aufwand treibt, um dem Leser das Leben zu erleichtern. Ist das heute obsolet? Sprache ist ein Spiegel des Denkens. Akzeptieren wir, dass Gedanken nicht geordnet und geprüft, sondern unverdaut ausgewürgt werden?

Informatik ist eine Kommunikationswissenschaft. Wer die Syntax einer Programmiersprache lernen kann, sollte mit dem Deutschen keine größeren Probleme haben. Ich neige also dazu, die Einhaltung der Regeln zu fordern. Dass es dabei nicht auf ein paar Fehler ankommt, ist klar; die unterlaufen uns allen.

Damit lautet meine **These 9**:
Wir sollten uns bewusst um präzise, korrekte Sprache bemühen, sie auch einfordern und in der Bewertung von Texten, auch Dissertationen und Publikationen, die sprachliche Qualität entsprechend gewichten.

8 Alternde Professoren

In den Achtzigerjahren habe ich gern und viel programmiert. Später gab es dazu immer weniger Gelegenheiten, und ich war (und bin bis heute) umgeben von jungen Leuten, die sehr routiniert mit der aktuellen Technologie umgehen, ob es nun neue Programmiersprachen oder irgendwelche Umgebungen und Frameworks

sind. Ich glaube noch immer, die Konzepte zu kennen und zu verstehen, aber ich weiß auch, dass es große Vorteile hat, sich immer wieder selbst mit den Details der Softwareentwicklung zu befassen.

2012 wird meine Pensionierung dieses Problem für mich lösen. Aber ich kann mir nicht vorstellen, dass ich der Einzige bin, der es hat. Wir, die Hochschullehrer, sollten darüber nachdenken, ob es nicht auch für uns eine Art Life Cycle gibt, der unsere sich ändernden Stärken und Schwächen berücksichtigt. Es hätte viele Vorteile, wenn es für meinesgleichen ein Altenteil gäbe, auf dem ich die letzten Jahre mit Einführungs- und Überblicksvorlesungen verbringen könnte (die meine jüngeren Kollegen offensichtlich fürchten). Dazu müsste aber heute schon eine Nachfolgerin oder ein Nachfolger da sein, um die spezielleren Lehrveranstaltungen zu übernehmen.

Meine **These 10** ist damit:
Die Stärken und Schwächen der Professoren ändern sich; eine Fakultät sollte wie eine Familie funktionieren, in der sich die Rollen ebenfalls mit den Jahren ändern. Das Visionäre, das Gestalten der Zukunft ist Sache der Jungen, ebenso die Präsentation der jüngsten wissenschaftlichen Resultate in der Lehre. Die Älteren können sehr gut die Grundlagen und den fachlichen Überblick vermitteln; durch die Übernahme wichtiger Ämter (Studiendekan, Vorsitz im Prüfungsausschuss usw.), in denen die lange Erfahrung nützt, halten sie den Jüngeren den Rücken frei.

9 Ingenieure für Software

In Deutschland wohnt das SE wie in den meisten anderen Ländern bei der Informatik zur Untermiete. Damit sind die Rahmenbedingungen der Lehre durch die Informatik vorgegeben. Das bedeutet vor allem: Die Grundausbildung in Programmierung erfolgt bottom-up, von Micky-Maus-Programmen zur komplexen Software (wobei man dorthin in der Hochschule kaum kommt), das Grundstudium hat eine starke Prägung durch die Theorie (das Pumping-Lemma muss sein), und im Übrigen bekommen die Studenten eine breite Übersicht der Gebiete, nicht am Bedarf orientiert, sondern an der realen Mischung der Lehrstühle.

Ich behaupte nicht, dass das falsch ist. Wahrscheinlich ist es besser, zwei Vorlesungen eines begeisterten Datenbankers zu hören als eine gute über Datenbanken und eine wirklich *vorgelesene* über Software Engineering. Aber ich sehe auch nicht, dass dieses Vorgehen didaktisch legitimiert oder durch die berufliche Zukunft unserer Studenten gerechtfertigt ist. Eine offene Diskussion über die Inhalte und den Aufbau der Lehre ist, wenigstens in der deutschen Universität, wie ich sie kenne, nicht möglich. Die Entscheidungen werden von Leuten getroffen, die sich mehrheitlich als Lobbyisten ihrer Disziplinen verstehen. In der Dis-

kussion über die Umstellung auf Bachelor und Master (die von Politikern ohne jede fachliche Kompetenz erzwungen wurde) war das meistgenannte Argument meiner Kollegen: »Wir wollen die besten Leute in unserem Masterstudiengang haben!« Die Frage, wie denn Leute *ohne* Master ausgerüstet sein werden, wurde regelmäßig mit der Gegenfrage beantwortet: »Glaubt wirklich irgendjemand, dass das eine sinnvolle Qualifikation ist?«

In dieser Umgebung haben wir, die Software-Ingenieure, zwar die Möglichkeit, unsere Sicht der Dinge zu präsentieren, aber keine Chance, aus unseren Studenten Ingenieure zu machen. Chinese wird man nicht durch den Besuch eines Sprachkurses oder eines Diavortrags, sondern indem man in China unter Chinesen lebt.

Die Existenz eines Studiengangs Softwaretechnik in Stuttgart widerlegt diese Einschätzung nicht. Die Chance, ihn einzurichten, war 1995 durch eine Reihe glücklicher Umstände entstanden und ist nicht reproduzierbar. Wir haben damit wenigstens, um im Bild zu bleiben, ein Stück China, aber leider viel zu wenig Chinesen.

Das bedeutet (**These 11**):
Wer SE lehren will, darf sich nicht damit begnügen, die für ihn vorgesehene Nische zu gestalten. SE erhält seine Motivation und seine Notwendigkeit aus den Erfahrungen bei der Lösung realer, praktischer Probleme, es ist also – wie jedes Engineering – nicht an der Erkenntnis orientiert, sondern an der Problemlösung. Wir müssen dafür sorgen, dass unsere Studenten von Beginn an mit diesem Denkansatz vertraut sind. Das alte, heute leider vergessene Motto des Gymnasiums »Jede Stunde eine Deutschstunde« sollte auf unser Fach übertragen werden: »Jede Vorlesung eine Lehrstunde in SE!« Das klappt natürlich nur, wenn sich die Dozenten als Software-Ingenieure verstehen.

Konkrete Konsequenzen dieser Überlegung waren in Stuttgart der frühe Einsatz der SE-Vorlesungen und die Umstellung des Programmierkurses im 1. Semester auf ein Praktikum, in dem die Auseinandersetzung mit einem großen Programm (20 kLOC) geübt wird. Eine spezielle Einführungsvorlesung für Softwaretechniker hat sich leider nicht realisieren lassen. Allerdings hatten SE-Veranstaltungen positive Rückwirkungen auf die Lehre im Informatik-Studiengang. So führten Erfahrungen im zusätzlichen Programmierkurs dazu, dass ihn nun auch die Informatiker bekommen. Im Bachelor-Studiengang Informatik werden eine SE-Einführung und ein Projekt obligatorisch.

10 Institutionalisierung

Gerade Ingenieure neigen zu der Annahme, dass sich die bessere Lösung durchsetzt. Dabei ist die Welt voll von schlechten Lösungen, die die guten aus dem Rennen geworfen haben. Ich selbst war Jahrzehnte lang überzeugt, dass sich die offensichtlichen Vorteile einer sinnvollen SE-Ausbildung in einem Zuwachs an Lehrstühlen und Mitteln niederschlagen würden.

Das war ein Irrtum. Anders als die künstliche Intelligenz, die in den Achtzigerjahren so viel Lärm gemacht hat, dass sie noch heute vom Echo leben kann, ist das SE ein Aschenbrödel geblieben. Anders als im Märchen hat die Sache aber kein glückliches Ende, Aschenbrödel sitzt noch immer an der Feuerstelle, während ihre Stiefschwestern mit dem Prinzen (dem Herrn aus dem Ministerium) flirten.

Wenn es uns nicht gelingt, ein Lehr- und Forschungsgebiet SE zu etablieren, das strukturell nicht von der Informatik abhängt, müssen wir uns damit abfinden, dass unsere Arbeitsbedingungen und Gestaltungsmöglichkeiten von Leuten vorgegeben werden, die sich durch das Software Engineering vor allem bedroht fühlen. Natürlich ist diese Unabhängigkeit fern und alles andere als leicht erreichbar. Aber die Geschichte hat immer wieder gezeigt, dass ferne Ziele nicht utopisch bleiben, wenn sie nur mit großer Hartnäckigkeit verfolgt werden.

These 12:
Nur in einem *eigenständigen* Fachbereich, der mit der Informatik eng zusammenarbeitet und Module der Lehre austauscht, ihr aber nicht untergeordnet ist, wird sich das SE entfalten. Die Informatik stellt den Software-Ingenieuren die theoretischen Grundlagen im weitesten Sinne bereit.

11 Fazit

Unsere Lehre des Software Engineering ist, gemessen an der Nachfrage der Studenten und dem Echo aus der Industrie, recht erfolgreich. Aber wir schöpfen die Möglichkeiten bei weitem nicht aus. Verbesserungen sind auf allen Ebenen möglich: bei den Rahmenbedingungen, Studienplänen und Lehrveranstaltungen, bei der Beurteilung und Notengebung und an vielen anderen Stellen. Dass wir nicht alle Probleme lösen können, sollte uns nicht davon abhalten, wenigstens die einfacheren anzugehen.

Und wir sollten daran arbeiten, dass das Software Engineering nicht länger das falsche Etikett eines Spezialgebiets der Informatik trägt, sondern sich als Technik neben anderen Techniken versteht.

Danksagung

Es versteht sich von selbst, dass in einer solchen Sammlung von Erfahrungen viele Gespräche mit Studenten, Mitarbeitern und Kollegen verarbeitet sind. Sie alle haben Anteil an diesem Text.

Drei anonyme Gutachter haben sich die Mühe gemacht, mein Manuskript zu lesen und detailliert zu kommentieren. Dafür danke ich ihnen sehr herzlich.

Frau Kuhle hat die (vermeintliche) Endfassung geprüft, wie immer sehr effektiv.

Literatur

[Boehm 1973] B.W. Boehm: Software and its impact: a quantitative assessment. DATAMATION, 19, No.5, 48-59.

[Boehm 1976] B.W. Boehm: Software Engineering. IEEE Trans.s on Computers, C-25, pp.1226-1241.

[Broy, Denert 2002] M. Broy, E. Denert (eds.): Software Pionieers: Contributions to Software Engineering. Springer Verlag, Berlin. 2002

[Ludewig 2008] J. Ludewig: Software Engineering at Full Scale: A Unique Curriculum. Chapter XIV in H.J.C. Ellis, S.A. Demurjian, J.F. Naveda (eds.): Software Engineering: Effective Teaching and Learning Approaches and Practices. IGI Global, Hershey, PA, USA. 2008

[Randell 2001] B. Randell: Webseite mit Quellen zu den NATO-Konferenzen 1968 und 1969. *http://homepages.cs.ncl.ac.uk/brian.randell/NATO/*

Ein integrativer interdisziplinärer Lehrversuch: Software Engineering und Technisches Schreiben

Gabriele Schmidt

Fachbereich Informatik und Medien
Fachhochschule Brandenburg
gabriele.schmidt@fh-brandenburg.de

Gerlinde Hollweg

Studienrätin, Schreibtrainerin
Berlin, Brandenburg
hollweg@besser-schreiben.de

Zusammenfassung

Interdisziplinäre Schlüsselkompetenzen stellen neben den fachlichen Kompetenzen wichtige Voraussetzungen für den beruflichen Erfolg dar. Bei dem durch den Bologna-Prozess verkürzten ersten berufsqualifizierenden Studium sind es häufig gerade diese Schlüsselkompetenzen, die zugunsten der fachlichen Ausbildung wegfallen. Dies führt zu einem Dilemma in der Ausbildung, aus dem ein integrativer interdisziplinärer Ansatz einen Ausweg bietet.

Im Folgenden wird die schriftsprachliche Kompetenz als eine für die Ausbildung in technischen Berufen unerlässliche interdisziplinäre Schlüsselkompetenz motiviert und in einem interdisziplinären Lehrversuch in die Software-Engineering-Ausbildung integriert. Erste Ergebnisse nach zweimaliger Durchführung des Versuchs werden vorgestellt und diskutiert.

1 Einleitung

Schreiben gehört zum Beruf. Allerdings wird es oft als lästige Pflicht im Nachhinein erledigt, wenn das eigentliche, das technische Problem gelöst ist. Falls der Leser den Text nicht versteht, liegt das am Leser und nicht am Text. Der Leser

gibt seine vermeintliche Inkompetenz lieber nicht zu, und so tradiert sich eine Vorstellung von Schreiben in technischen Berufen, die hinter dem heutigen Stand der Schreibprozessforschung zurückbleibt und dazu beiträgt, dass Studierende in technischen Studiengängen nicht hinreichend auf ihren zukünftigen Beruf vorbereitet werden.

Das sichere Beherrschen kommunikativer Kompetenzen ist für den Erfolg des Arbeitsprozesses und den persönlichen Aufstieg im Unternehmen von entscheidender Bedeutung. Nicht von ungefähr hat Denert in seinem eingeladenen Vortrag zum Thema »Was wir vom Software-Ingenieur erwarten: Gleichgewicht von Fachwissen und Persönlichkeit« [Denert 2007] auf der SEUH 2007[1] unter den vier Muss-Anforderungen an das Fachwissen »gutes Deutsch« aufgelistet.

Dass die fachliche Aussage eines Textes im Berufsalltag richtig sein muss, versteht sich von selbst. Sie muss aber darüber hinaus auf einen konkreten Adressaten und ein gegebenes Ziel hin formuliert sein. Das ist in der Regel bedeutend schwieriger, als im Fachdiskurs einen Zusammenhang korrekt auszudrücken. Das Erlernen der Fachsprache und ihre korrekte Anwendung ist Bestandteil des Studiums. Im Beruf ist es darüber hinaus notwendig, komplexe fachliche Zusammenhänge für unterschiedliche Adressaten so gut verständlich zu machen, dass diese über ein Projekt verantwortlich entscheiden oder ausgehend von dem Text eindeutig handeln können.

Diese sprachliche Kompetenz bringen Studierende nicht von der Schule mit. Es ist Aufgabe der Hochschulen, diese im jeweiligen Fach für die spezifische Berufspraxis zu entwickeln.

Bei den durch Studienreformen im Rahmen des Bologna-Prozesses vorgeschriebenen Kürzungen im Lehrumfang wurden Inhalte, die den Soft Skills zugeordnet werden, am Fachbereich Informatik und Medien der Fachhochschule Brandenburg zugunsten fachlicher Inhalte weitgehend gestrichen. Den ersten umfangreichen Text müssen die Studierenden mit der Abschlussarbeit erstellen. Sie stehen dieser Aufgabe mehr oder weniger hilflos gegenüber. Selbst wenn sie über das theoretische Know-how verfügen, fehlt ihnen häufig die praktische Erfahrung, um unter dem gegebenen Zeitdruck eine gute bis sehr gute Leistung erbringen zu können.

Aus dieser Ausgangslage heraus entstand die Idee eines integrativen interdisziplinären Ansatzes. Das schon frühzeitig professionell angeleitete Schreiben von Texten in Phasen der fachlichen Ausbildung, in denen das Erstellen von spezifischen Dokument erforderlich ist, erscheint als ein Weg, Schlüsselkompetenzen, zu denen in einem erweiterten Verständnis auch Schreibkompetenzen gehören, in die fachliche Ausbildung einzubeziehen (vgl. [Chur 2005]). Der Vorteil des integrativen interdisziplinären Ansatzes liegt erstens darin, dass die Studierenden von

1 SEUH 2007 steht für den 10. Workshop Software Engineering im Unterricht der Hochschulen, der 2007 in Stuttgart stattfand.

einem doppelt kompetenten Team angeleitet werden. Zweitens gelingt es leichter, den Studierenden die Relevanz und den praktischen Anwendungsbezug überfachlicher Kompetenzen zu verdeutlichen, da sie unmittelbar auf die Inhalte und Methoden ihres Studienfaches zugeschnitten werden können (vgl. [Weinert 2001]).

2 Ziele des integrativen interdisziplinären Lehrversuchs

Im Folgenden wird ein integrativer interdisziplinärer Lehrversuch beschrieben, in dem die professionelle Ausbildung im Erstellen von Texten mit einem Software-Engineering-Modul kombiniert wird. Dieser Lehrversuch wird gemeinsam von einer Informatikerin (Professorin für Software Engineering) und einer Schreibpädagogin (Lehrbeauftragte) durchgeführt.

Die Lernziele zur Fachkompetenz im Software Engineering sind:

- Es werden Kompetenzen in der Erhebung und insbesondere Strukturierung von Sachverhalten, Wissen, Abläufen etc., d.h. Anforderungen, erworben.
- Schwerpunktmäßig werden Kompetenzen der objektorientierten Modellierung im Bereich Software Engineering erworben.
- Die Studierenden werden die Fähigkeit zur Teamarbeit erlernen bzw. weiterentwickeln und den Bezug zu ihrer Arbeitsweise im Beruf reflektieren.

Diese eher gängigen Lernziele werden durch die folgenden Lernziele des Technischen Schreibens erweitert:

- Das Bewusstsein wird geweckt, dass Schreiben Bestandteil fachlichen und beruflichen Handelns und Problemlösens ist.
- Es wird das Bewusstsein für die Korrelation zwischen fachlicher und stilistischer Güte geweckt. Hier wird besonders Wert darauf gelegt, dass die Studierenden die Fähigkeit zur sprachlichen Strukturierung und Präzision entwickeln und den Zusammenhang zur fachlichen Strukturierung und Präzision erkennen.
- Weiterhin soll deutlich werden, dass schriftsprachliche Kompetenz zum einen eine wesentliche Voraussetzung dafür ist, dass ein Projekt erfolgreich durchgeführt werden kann, und zum anderen eine Voraussetzung für persönlichen Erfolg im Beruf.

Da Arbeits-, Entwicklungs- und Forschungsprozesse anfangs als sprachliche Artefakte existieren bzw. aus diversen, oft kooperativen Iterationsprozessen heraus entstehen, ist das Produzieren von Texten Handeln im fachlichen Kontext und keine nachgeordnete Tätigkeit im Sinne eines »Aufschreibens« von etwas, was man zuvor richtig gedacht hat. Berufliche Texte müssen eindeutige Handlungen von konkreten Personen ermöglichen. Daher ist es Aufgabe des Verfassers, diese Texte eindeutig und in Bezug auf den konkreten Adressaten zu formulieren.

Die Genauigkeit einer Dokumentation soll dazu führen, dass Adressaten und Beteiligte (Stakeholder) den Text vollständig erfassen und von diesem ausgehend zielorientiert handeln können. Beispielsweise soll der Auftraggeber als Adressat einer Spezifiktion der Anforderungen in die Lage einer Abnahme versetzt werden. Gleichzeitig sollen die Softwareentwickler, die ebenfalls Adressaten dieser Spezifikation sind, auf dieser Basis den Entwurf und die Software erstellen können.

Die Studierenden sollen »Schreiben im Beruf als Handeln im Fach« [Pogner 1999] begreifen und ernst nehmen.

Langfristig versprechen sich die Autorinnen als Ziele des integrativen interdisziplinären Lehrversuchs nicht nur eine verbesserte Schreibkompetenz der Studierenden für zukünftige Aufgaben in Studium und Beruf, sondern auch Synergieeffekte zwischen den fachlichen und den stilistischen Lernzielen. Beispielsweise wirken sich Fähigkeiten wie Präzision und Strukturierung unmittelbar auf die Qualität der Darstellung aus. Sie sind aber zugleich grundlegende Fähigkeiten, die für die objektorientierte Modellierung im Besonderen und für das Software Engineering im Allgemeinen benötigt werden.

3 Beschreibung des Lehrversuchs

Das Modul Software Engineering des Bachelor-Studienganges Informatik wird im vierten Semester des Curriculums angeboten und ist für alle Studierenden eine Pflichtveranstaltung. Das Modul ist unterteilt in zwei Semesterwochenstunden (SWS) Vorlesung und zwei SWS-Übungen.

Die Studierenden des vierten Semesters haben Erfahrung in der objektorientierten Programmierung und der Dokumentation von Programmcodes. Sie haben dagegen keine oder kaum Erfahrung in Projektmanagement, Qualitätsmanagement und in über die Programmierung hinausgehenden Schritten bei der Software-Entwicklung.

3.1 Themen der Vorlesung

Die Professorin stellt Prozessmodelle, die Erhebung von Anforderungen (siehe z.B. [Rupp 2007]) und objektorientierte Modellierung mit der Unified Modeling Language (kurz UML, siehe z.B. [Balzer 04] oder [Rupp et al. 2007]) unterteilt in objektorientierte Analyse (OOA) und objektorientierten Entwurf (OOD[2]) vor. Der hauptsächliche Schwerpunkt wird auf die OOA gelegt. Im OOD werden Entwurfsmuster nach [Gamma et al. 1994] besonders betont.

Ergänzend liest die Schreibpädagogin früh – meist nach einer Motivation und der Vorstellung der Prozessmodelle – eine Einheit zur Einführung.

2 Das »D« steht für »Design«.

Diese Einheit zum Technischen Schreiben umfasst zwei Schwerpunkte. Zum einen wird ein Verständnis von Schreiben als problemlösendem Handeln in Studium und Beruf vorgestellt, zum anderen wird die Notwendigkeit eines bewussten Umgangs mit Sprache betont, wenn das fachliche Ziel erreicht werden soll.

Ausgehend von Textbeispielen aus Arbeiten von Studierenden und wissenschaftlichen Veröffentlichungen wird gezeigt, wie die Verwendung unterschiedlicher Begriffe für dieselbe Sache, unklare Satzkonstruktionen, fehlerhafte grammatikalische Bezüge oder die Verwendung einer aufgeblähten, vermeintlichen Fachsprache dazu führen, dass Texte unverständlich werden oder sogar den Sachverhalt falsch darstellen.

3.2 Beschreibung der Übung

Im Rahmen der Übung, die den Schwerpunkt des Lehrversuches darstellt, bilden die Studierenden Teams.

Die Studierenden sollen die Modellierungsaufgaben in drei Arbeitsschritten bearbeiten. Diese Arbeitsschritte stehen exemplarisch für das sequenzielle Durchlaufen von Phasen eines Software-Engineering-Prozessmodells, z.B. Unified Process (UP): Inception, Elaboration and Construction [Jacobson et al.1999][3]. Die drei Arbeitsschritte führen zu drei Dokumenten: Lastenheft, Pflichtenheft und Entwurf.

Das Lastenheft wird im Sinne von [Balzert 1998] als eine grobe Beschreibung der fachlichen Anforderungen an die zu erstellende Software aus Sicht des Auftraggebers, der häufig ein Laie in der Software-Entwicklung ist, verstanden. Adressat ist der Auftragnehmer, der in Bezug auf die Domäne des Problems und die mit der Aufgabe verknüpften Anforderungen ein Laie ist. Die Aufgabe hat noch keinen objektorientierten Modellierungsteil. Daher können sich die Studierenden auf die stilistische Darstellungsform konzentrieren.

Das Pflichtenheft stellt sich als ein ausführliches (vollständiges) Lastenheft dar [Balzert 1998]. Es wird vom Auftragnehmer, d.h. dem Systemanalytiker oder Softwareentwickler, aus Sicht des Auftraggebers verfasst.

Als fachliche Aufgabe werden UML-Diagramme der OOA (Anwendungsfall-, Aktivitäts-, Klassen-, Interaktionsdiagramm und Objektlebenszyklus) gefordert.

In Bezug auf die stilistische Darstellungsform sollen sich die Studierenden auf drei Punkte konzentrieren:

- **Gliederung:**
  Ist diese vollständig und sind die einzelnen Positionen richtig zugeordnet?

3 Die Phasen können aus zeitlichen Gründen leider nicht, wie in UP oder anderen modernen (agilen) Prozessmodellen vorgesehen, inkrementell/zyklisch durchlaufen werden.

- **Feingliederung:**
  Sind die einzelnen Abschnitte schlüssig formuliert?
- **Wortwahl:**
  Sind die Fachbegriffe eindeutig angewendet und werden Worthülsen vermieden?

Als Hilfestellung werden den Studierenden zwei Gliederungsvorschläge vorgegeben: entweder das kommentierte Beispiel aus [Balzert 2004] oder der Standard IEEE SRS (Software Requirements Specification).

Der Adressatenbezug gestaltet sich beim Pflichtenheft besonders schwierig. Das Dokument wird einerseits für den Auftraggeber, höchstwahrscheinlich ein Laie im Software Engineering, erstellt. Der Auftraggeber muss das Dokument abnehmen, d.h., er muss in die Lage versetzt werden, entscheiden zu können, ob die im Dokument beschriebene Lösung (Software) sein Problem richtig löst. Andererseits wird das Dokument für die Softwareentwickler erstellt, die auf der Basis dieses Dokumentes und insbesondere seiner UML-Diagramme die Software im Detail entwerfen, implementieren und verifizieren[4].

Als zusätzliche Anleitung erhalten die Studierenden den helfenden Hinweis, dass der Bezug zum Laien (Auftraggeber) als Adressat eher im textuellen Teil und der Bezug zum Experten (Softwareentwickler) als Adressat eher durch die Diagramme hergestellt werden soll.

Insgesamt ist diese Aufgabe für die Studierenden wesentlich anspruchsvoller als der erste Arbeitsschritt, da sie nun sowohl die neu in der Vorlesung vorgestellten Konzepte in der Modellierung (OOA) als auch die neu erworbenen Schreibkenntnisse umsetzen sollen. Hinzu kommt, dass die Studierenden beides sinnvoll und unter Berücksichtigung des Adressatenbezugs integrieren sollen, d.h., die UML-Diagramme sind in die textuelle Gliederung richtig einzufügen und sprachlich zu integrieren.

Der Entwurf als Modell der zu entwickelnden Software ist ein in Fachsprache geschriebenes Dokument von Softwareentwicklern für Softwareentwickler, d.h., der Adressat ist klar gegeben. Der Schwerpunkt liegt auf der Modellierung (OOD).

In Bezug auf den Schreibstil wird die Gliederung beurteilt, da im Gegensatz zu den vorangegangenen beiden Arbeitsschritten keine Struktur vorgegeben wird. Die Studierenden sollen das bisher Erlernte, d.h. die Befähigung, Dokumente strukturieren zu können, selbstständig umsetzen. Des Weiteren wird auf die korrekte Verwendung von Fachbegriffen besonderen Wert gelegt.

---

4 Dieser doppelte Adressatenbezug stellt nach Ansicht der Autorinnen eine der größten Herausforderungen des Software Engineering dar und ist in keinem Prozessmodell – auch nicht durch die »On-site-customer«-Forderung im Extreme Programming (XP) – bisher befriedigend gelöst.

Feedbacktechnik und Vergabe der Leistungspunkte

In jeweils einer Übungsstunde werden sowohl Lasten- als auch Pflichtenhefte in einer Feedbackrunde bearbeitet, indem jeweils zwei Gruppen ihre Dokumente austauschen. Jede Gruppe erhält das Heft ihrer Partnergruppe zur Textüberarbeitung.

Da Textüberarbeitung häufig nicht als inhaltliche Arbeit, sondern als Korrektur formaler Fehler verstanden wird, ist hier die Aufgabe so organisiert, dass jede Gruppe im Text ihrer Partnergruppe Ausrufungs- und Fragezeichen anbringen und diese schriftlich kommentieren soll. Die Ausrufungszeichen sollen Textstellen markieren, die besonders aussagekräftig oder gut formuliert sind, die Fragezeichen solche, die fraglich oder unverständlich erscheinen. In den Kommentaren werden die Begründungen für diese Markierungen schriftlich festgehalten, bevor die Gruppen zum Austausch ihrer Beobachtungen zusammenkommen [Schnetzer 2006].

Insbesondere beim Pflichtenheft erhalten die Gruppen Zeit, nach dem Feedback Fehler zu korrigieren. Erst die korrigierte Version wird bewertet. Dies ist den Studierenden zuvor nicht bekannt.

Als Motivation zur Teilnahme an den Übungen werden Leistungspunkte, die neben der Klausur in die Bewertung der Modulleistung eingehen, teamweise vergeben. In der Übung können insgesamt 20 Punkte erzielt werden. Bei den Kriterien für die Punktevergabe wird zwischen fachlichem Inhalt und Darstellungsstil unterschieden. Für das Lastenheft als Einarbeitungsschritt werden nur vier Punkte vergeben (zwei für Inhalt und zwei für Stil). Für den zweiten Arbeitsschritt werden acht Punkte vergeben (vier für Inhalt und vier für Stil). Beim Entwurf liegt der Schwerpunkt auf der Modellierung (sechs Punkte) und weniger auf dem Schreibstil (zwei Punkte).

4 Durchführung des Lehrversuchs und Beispiele

Der Lehrversuch wurde bisher in zwei Varianten durchgeführt. Diese beiden Varianten werden im Folgenden in Bezug auf ihre jeweiligen Besonderheiten beschrieben.

4.1 Variante 1: Sommersemester 2007

Der Versuch fand im Sommersemester 2007 statt, das 15 Wochen Lehre zur Verfügung stellte. Es haben insgesamt 74 Studierende an der Übung teilgenommen, die sich auf 19 Teams aufteilten. Die Teams hatten zwischen zwei und sechs Mitglieder. Bei der Erstellung des Entwurfs schieden einige Teams aus, so dass hier nur noch 13 Teams teilnahmen, in denen insgesamt 51 Studierende waren[5].

5 Das Ausscheiden der Teams kann als Aufwand-Nutzen-Optimierungsstrategie der Studierenden erklärt werden, da gegen Ende des Semesters in vielen Modulen Abgaben verlangt werden und bereits für Klausuren gelernt werden muss.

Zu Beginn erhalten die Studierenden eine kurze Aufgabenbeschreibung zu »Mikes Internet Kneipe«: Eine fiktive Person – Mike – möchte ihre Kneipe in den USA zu einer voll elektronisch durchorganisierten Internetkneipe umfunktionieren. Die Aufgabenbeschreibung enthält bereits einige Anforderungen an die Software auf unterschiedlichen Detailebenen.

Da kein realer Auftraggeber als Verfasser zur Verfügung steht, sollen sich die Studierenden in die Rolle von Mike versetzen und ein Lastenheft formulieren, in dem sie selbst Anforderungen beschreiben und ergänzen. Da mögliche Fehler aus dem Lastenheft nicht unreflektiert ins Pflichtenheft übernommen werden sollen, müssen die Studierenden die Lastenhefte mit einer Partnergruppe tauschen. Das getauschte Lastenheft bildet die Ausgangsbasis für den zweiten Arbeitsschritt, die Erstellung eines Pflichtenheftes. Die Studierenden werden angeleitet, die Perspektive zu wechseln und sich nun als Mitarbeiter einer Softwarefirma zu verstehen.

4.2 Beispiele aus Variante 1 für Variante 2

Die folgenden Beispiele wurden den Pflichtenheften, die durch die Studierenden in Variante 1 verfasst wurden, entnommen und neben weiteren Beispielen aus der Literatur in der Vorlesung (Variante 2) von der Schreibpädagogin präsentiert.

In der Auswertung der Pflichtenhefte zeigte sich eine Auffälligkeit, die als ein »ungefährer Gebrauch von Sprache« bezeichnet werden kann. So findet sich in einem Pflichtenheften die Formulierung: »Durch ein mehrsprachiges System sollen es möglichst viele Personen nutzen können.« Es scheint unklar zu sein, was das Subjekt des Satzes ist und wodurch es bestimmt ist. Oder ein Subjekt wird innerhalb eines Satzes mit zwei unterschiedlichen Prädikaten verknüpft, ohne dass erkannt wird, dass dies den Sinn entstellt. »Die verfügbaren Speisen und Getränke können über Terminals am Sitzplatz abgerufen und bestellt werden oder bei einer Servicekraft aufgegeben werden.« In folgender Formulierung wird das Wort »Bestellung« für die Bestellung selbst und die bestellte Ware verwendet. »Bei Bestellung über das Intranet muss der Kunde seine Bestellung selbst abholen.«

Auf Nachfrage zeigte sich, dass die Studierenden davon ausgingen, das von ihnen Gemeinte formuliert zu haben. Ein wohlmeinender Leser kann das Gemeinte auch ermitteln, wenn er die Wörter kombiniert und dabei zumindest teilweise von der Satzstruktur abstrahiert. Dies setzt voraus, dass der Leser mit der Thematik vertraut ist. Es ist anzunehmen, dass die Studierenden in ihrer Alltagskommunikation ähnlich verfahren und diese Ausdrucksweise auf geschriebene Texte übertragen. Hier ist es notwendig, diesen »ungefähren Gebrauch von Sprache« bewusst zu machen und auf die Gefahren hinzuweisen, die ein solcher Sprachgebrauch in schriftlichen Vereinbarungen mit Kunden oder in Anweisungen zum Programmieren hat.

Werden solche grammatisch falschen Formulierungen aus ihrem Kontext gelöst und anonymisiert in der Vorlesung präsentiert, sind die Studierenden

schnell bereit, Fehler zu suchen und zu benennen. Auf Kritik an ihren eigenen Texten dagegen reagieren sie eher empfindlich und beginnen, sich zu rechtfertigen. Über die Arbeit an Texten und deren Schwächen hinaus ist es im Rahmen dieses Projekts notwendig, den Umgang mit Feedback zu üben und Schreiben als Arbeitsprozess im Team einzuführen und zu üben. Das Erstellen und Überarbeiten von Texten im Team entspricht weitgehend der Praxis im Berufsleben, und so kann diese Arbeitsform im Studium eine wichtige Vorbereitung auf die spätere Berufspraxis sein.

4.3 Variante 2: Sommersemester 2008

Der zweite Versuch ist bzgl. Anzahl der teilnehmenden Studierenden, der Teambildung und Teamgröße mit dem Vorjahr vergleichbar. Gegenüber dem Vorjahr wurde eine neue Aufgabe gestellt und der Arbeitsschritt Lastenheft verändert.

Das fiktive »Hotel Oase« möchte eine Informations- und Buchungssoftware über die Angebote im Sport- und Unterhaltungsbereich für seine Gäste. Des Weiteren soll den Gästen ein Internetzugang gegen Gebühr ermöglicht werden und Daten mit dem vorhandenen Hotelsystem ausgetauscht werden können.

Um den Schreibaufwand geringer zu halten und der Situation im Beruf näherzukommen, wird ein Lastenheft vorgegeben. Es enthält gezielt Schwachstellen, die sowohl sprachlicher Art (Verletzung des Adressatenbezugs, ungefährer Gebrauch von Sprache, Fehler in Gliederung und Feingliederung) als auch inhaltlicher Art sind. Die Studierenden sollen diese Schwachstellen finden und erkennen, zu welchen inhaltlichen Missverständnissen diese führen können. Sie sollen die Schwachstellen schriftlich festhalten (Mängelliste) und Fragen an den Kunden für ein mögliches Kundengespräche (Fragebogen) sowie Glossareinträge formulieren. Nach der Bepunktung erläutert die Professorin als Feedback die Schwachstellen und beantwortet in der Rolle des Kunden die Fragen der Studierenden.

Eine umfassende aktive Auseinandersetzung mit dem Schreibprozess konzentriert sich auf das Pflichtenheft in seiner besonderen Anforderung an den doppelten Adressatenbezug. Hier werden keine Änderungen vorgenommen, und auch die Feedbacktechnik wird wie bisher eingesetzt. Beim Entwurf wird ebenfalls nichts verändert.

5 Evaluation des Lehrversuchs

Die Evaluation wird einerseits durch die Studierenden durchgeführt und andererseits durch die beiden Lehrenden. Dabei handelt es sich um subjektive Bewertungen. Die Studierenden werten aus der aktuellen Stimmung während der Erhebung heraus. Die Lehrenden versuchen, die gemachten Beobachtungen zu interpretieren.

Leider ergaben sich zwei unterschiedliche Ergebnisse, die sich nicht nur auf die leichte Variation zurückführen lassen, sondern auch einen genaueren Vergleich erfordern.

5.1 Evaluation durch die Studierenden

Neben der kontinuierlichen semesterbegleitenden Evaluation durch den Fachbereich werden die Studierenden gesondert zu dem Versuch befragt. Es werden insgesamt neun Fragen gestellt, u.a. zum Stellenwert von technischen Dokumenten in der Informatik, zum Praxisbezug der Übung, zur Teamarbeit, zur Feedbacktechnik und zum Aufwand der Aufgabenbearbeitung.

In Variante 1 wurde das Tauschen der Lastenhefte als Vorgabe für die Weiterarbeit von den 28 an der Evaluation teilnehmenden Studierenden als negativ empfunden, da bereits unterschiedlich hoher Aufwand in die Aufgabe investiert worden war. Die Teilnahme an den Feedbackrunden und die Teamarbeit wurden am höchsten bewertet. Die Teamarbeit wird überwiegend als produktiv und sogar sehr produktiv eingestuft. Der Praxisbezug der Übung wird indifferent gesehen. Es scheint, dass die Studierenden des vierten Semesters zu wenig eigene Praxiserfahrung haben, um diesen einschätzen zu können. Die Frage bzgl. des Aufwandes ist durchschnittlich am negativsten bewertet worden. Die Aufgabe wird von den Studierenden trotz der als produktiv gesehenen Teamarbeit als zu aufwendig empfunden.

An der Evaluation zur Variante 2 nehmen 27 Studierenden teil. Die Bewertung fällt nahezu gleich aus. Der Bezug zur späteren beruflichen Praxis wird allerdings höher, der Arbeitsaufwand wird dagegen durchschnittlich noch negativer bewertet. In mehreren Freitext-Anmerkungen wurde das Verhältnis der Punkte zu der zu leistenden Arbeit als zu gering angesehen. Dieses Ergebnis ist umso verblüffender, da der Aufwand um die Formulierung des Lastenheftes reduziert worden war.

5.2 Auswertung der Lehrenden

Auffallend bei der nachträglichen Betrachtung der vergebenen Leistungspunkte bei Variante 1 ist, dass sich zwischen Lasten- und Pflichtenheft eine Verbesserung der Punkte für die stilistische Darstellung (von 68,5 % auf 81,75 % der jeweils erreichbaren Punkte) bei einer parallelen Verschlechterung der Punktzahl für die inhaltliche Arbeit (von 85 % auf 66,25 %) erkennen lässt, während die Gesamtpunktzahl prozentual im Vergleich beider Schritte fast gleich bleibt.

Die Verbesserung in der stilistischen Darstellung lässt sich zunächst damit erklären, dass die Studierenden sich durch die Bekanntgabe der Punkte, der Erläuterung der Kriterien und eine zunehmende Erfahrung in der stilistischen Darstellung verbessern konnten. Die gleichzeitige Verschlechterung der inhaltli-

chen Arbeit kann auf den steigenden Schwierigkeitsgrad durch die hinzukommende Modellierungsaufgabe und auf die noch mangelnde Praxis zurückgeführt werden.

Die Verbesserung des Schreibstils wurde von beiden Lehrenden, unabhängig voneinander, festgestellt. Im Nachhinein hatten sie den Eindruck, dass die Gruppen sehr motiviert und dem neuen Aspekt, Vermittlung von Schreibkompetenz, gegenüber offen waren.

Trotz dieses insgesamt positiven Gesamteindrucks empfinden sowohl die Studierenden als auch die Lehrenden den Aufwand für das Projekt als zu hoch.

Durch die in Variante 2 beschriebene Vorgabe eines Lastenhefts ist tatsächlich für die Lehrenden eine leichte Reduktion des Korrekturaufwands entstanden. Leider hat sich gleichzeitig bei den Studierenden eine Verschlechterung gegenüber dem Vorjahr von 76 % auf 60,45 % der Gesamtpunkte ergeben.

Überraschenderweise ist die Verschlechterung bei dem geänderten Schritt 1 (Lastenheft vs. Mängelliste, Fragenbogen und Glossar) besonders hoch beim Darstellungsstil (68,5 % im Vorjahr gegenüber 49 %). Die Mängellisten waren sehr unvollständig, konzentrierten sich häufig auf Tippfehler und konnten selten die einzelnen Sprachkonzepte benennen. Häufig sind die Fragen des Fragebogens stilistisch so schlecht formuliert, dass sie nicht an einen Kunden weitergegeben werden könnten. So werden Aufgaben, die von den Studierenden als Auftragsnehmer zu erledigen wären, an den Kunden delegiert und darüber hinaus in einem »Befehlston« formuliert.

Beim Pflichtenheft wurde abgesehen von formalen Fehlern wie fehlende Beschriftungen der Diagramme die Einbettung der Diagramme in den Text mehr vernachlässigt. Die Steigerung von Pflichtenheft zu Entwurf ist der des Vorjahres ähnlich.

Die Lehrenden waren beide mit den Leistungen der Studierenden nicht zufrieden und empfanden eine deutliche Verschlechterung gegenüber dem Vorjahr. Leider konnte das sich schon früh abzeichnende schlechte Gesamtniveau trotz der intensiven Feedbackrunde durch die Professorin nicht mehr verbessert werden.

Obwohl in der Evaluation die Bedeutung des Schreibens für die berufliche Praxis höher eingeschätzt wird, erscheint den Lehrenden die Motivation und Eigeninitiative der Studierenden geringer. Ferner hatte die Professorin den Eindruck, dass die Zeit zur Verbesserung des Pflichtenheftes nach der Feedbackrunde kaum genutzt wurde.

5.3 Vergleich der beiden Varianten

In beiden Evaluationen werden die Werte zur Feedbackrunde nahezu gleich eingeschätzt, trotzdem wird sie in der Diskussion durch die Studierenden des Sommersemesters 2008 stärker kritisiert. Es wird ein Feedback von den Lehrenden anstelle der Kommilitonen verlangt. Die nachfolgende Korrekturphase wurde dementsprechend nicht so ernst genommen und nicht so effektiv genutzt. Außerdem wurde die Information, dass die Möglichkeit zur Nachbesserung besteht und es sich noch nicht um den endgültigen Abgabetermin handelt, schon von den vorangegangenen Übungsgruppen weitergegeben. Dies wirkt sich auf die Sorgfalt der vorliegenden Dokumente negativ aus. Damit ist dieser methodische Schritt infrage gestellt.

Von der Evaluation ausgehend, wurde das Schreiben im zweiten Versuch etwas besser motiviert. Trotz dieser Verbesserung und etwa gleichem Teamverhaltens – bei beiden Versuchen stellt die Teamarbeit kein Problem dar – war die Leistung der Studierenden in Variante 2 geringer. Dies kann mit den Unterschieden zwischen den Jahrgängen begründet werden. Der Jahrgang der Variante 1 zeigte gegenüber den bis dahin gesammelten Erfahrungen der Professorin in diesem Modul eine insgesamt auffallend bessere Leistung. Anderseits muss die Frage gestellt werden, ob durch die Umstellung des Arbeitsschrittes in Bezug auf das Lastenheft nicht ein erheblicher Übungsanteil zum Sammeln von Erfahrungen weggelassen wurde.

Zusammenfassend stellen sich die beiden Lehrenden die Frage, ob der Jahrgang der Variante 2, der nach den bisherigen Erfahrungen eher dem durchschnittlichen Niveau der Jahrgänge entspricht, nicht überfordert war, obwohl z.B. die Rollenwechsel wegfielen. Hier würde eine weitere Unterteilung der Aufgabe mit mehr Einzelschritten, mehr Anleitungen und Feedback zu den Einzelschritten durch die beiden Lehrende Abhilfe schaffen. Gezieltere, kleinere Aufgabenstellungen könnten auf das Ganze gesehen trotzdem eine Reduktion des Korrekturaufwands nach sich ziehen.

6 Zusammenfassung und Ausblick

Ein integrativer interdisziplinärer Lehrversuch wurde vorgestellt, in dem die professionelle Ausbildung im Erstellen von Texten mit einem Software-Engineering-Modul kombiniert wird. Der Versuch wurde in zwei Varianten durchgeführt, wobei durch die zweite Variante im Wesentlichen der Aufwand sowohl für die Studierenden als auch die Lehrenden etwas reduziert werden sollte. Während in der ersten Durchführung eine sehr gute Leistungssteigerung erzielt werden konnte und die gesetzten Ziele weitgehend erreicht wurden, konnte dies in der zweiten Variante nicht wiederholt werden. Dies lässt sich nicht ausschließlich mit der Variation des Versuchs erklären.

Für die Fortsetzung des Projekts planen die Lehrenden, den Aufwand weiter einzugrenzen, die Übungen in mehr angeleitete Einzelschritte zu splitten, die Bewertungskriterien klarer zu definieren und eindeutiger der fachbezogenen und sprachbezogenen Korrektur zuzuordnen.

Inwieweit die Nachhaltigkeit dieses Ansatzes, ein Training zum Technischen Schreiben in eine Lehrveranstaltung zum Software Engineering zu integrieren, überprüfbar sein wird, ist gegenwärtig nur schwer absehbar. Weitere Versuche sollen durchgeführt, durch Umfragen unter den Studierenden evaluiert und verglichen werden. Denkbar ist, Absolventen, die an diesem Programm teilgenommen haben, zu befragen, inwieweit die Übung während des Studiums für das Verfassen der Abschlussarbeit hilfreich war.

Solange eine solche Arbeitsweise der Verknüpfung der fachlichen Erarbeitung mit ihrer schriftlichen Dokumentation auf eine einzelne Lehrveranstaltung im Laufe des Informatikstudiums beschränkt bleibt, wird auch der Effekt begrenzt bleiben. Für eine optimale Vorbereitung auf die spätere Tätigkeit im Beruf wäre eine durchgängige Integration und Verknüpfung der sogenannten Soft Skills mit den fachlichen Studieninhalten notwendig.

Danksagung

Der Dank der Autorinnen gilt den Studierenden des vierten Semesters im Modul »Software Engineering« im Sommersemester 2007 und 2008 im Studiengang Informatik (B. Sc.) am Fachbereich Informatik und Medien der Fachhochschule Brandenburg. Besonderen Dank schulden wir Susanne Draheim für ihre Motivation und Unterstützung durch Ideen.

Literatur

[Balzert 1998] Balzert, Helmuth: Lehrbuch der Software-Technik. Spektrum, Akad. Verl.; 1998

[Balzert 2004] Balzert, Heide: Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2. Spektrum, Akad. Verl.; Auflage: 2., 2004

[Chur 2005] Chur, Dietmar: (Aus-) Bildungsqualität durch Schlüsselkompetenzen – zur Konkretisierung eines integrativen Bildungsverständnisses. In: Nicole Colin/Alain Latard/ Joachim Umlauf (Hrsg.): Germanistik – eine europäische Wissenschaft? 2005.

[Denert 2007] Denert, Ernst: Was wir vom Software-Ingenieur erwarten: Gleichgewicht von Fachwissen und Persönlichkeit, In: Zellner, A. und Deiniger, M. (Hrsg.): Software Engineering im Unterricht der Hochschulen. dpunkt.verlag, 2007.

[Gamma et al. 1994] Gamma, Erich, Helm, Richard, Johnson, Ralph und Vlissides. John: Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software, Addison-Wesley, 2004

[Jacobson et al.1999] Jacobson, Ivar, Booch Grady and Rumbaugh James: The Unified Software Development Process, Addison-Wesley, 1999

[Pogner 1999] Pogner, Karl-Heinz: Schreiben im Beruf als Handeln im Fach. Narr 1999

[Rupp 2007] Rupp, Chris: Requirements-Engineering und – Management: Professionelle, iterative Anforderungsanalyse für die Praxis. Hanser Fachbuchverlag; Auflage: 4., 2007

[Rupp et al. 2007] Rupp, Chris, Queins, Stefan und Zengler, Barbara: UML 2 glasklar. Praxiswissen für die UML-Modellierung. Hanser Fachbuchverlag; Auflage: 3, 2007

[Schnetzer 2006] Schnetzer, Adrian: Peer-Feedback auf Texte an Mittel- und Hochschule. In: Kruse, Otto/ Berger, Katja/ Ulmi, Marianne: Prozessorientierte Schreibdidaktik, Schreibtraining für Schule, Studium und Beruf. Haupt-Verlag 2006, S. 195-214

[Weinert 2001] Weinert, Franz E.: Concept of Competence: A Conceptual Clarification. In: Dominique S. Rychen/Laura H. Salganik, (Hg.): Defining and Selecting Key Competencies. Seattle/Toronto/Bern/Göttingen 2001, S.45-65

Lehr-Muster bei geringer Ausgangsmotivation für überfachliche Qualifizierung im Software Engineering

Nicola Marsden
Hochschule Heilbronn
marsden@hs-heilbronn.de

Zusammenfassung

Bei der Vermittlung von Arbeitsgrundlagen wie Recherchieren und wissenschaftliches Arbeiten im Studium Software Engineering sehen Lehrende sich insbesondere in der Studieneingangsphase häufig mit ausgesprochen geringer Motivation der Studierenden für diese Themen konfrontiert.

Auf Basis des theoretischen Motivationskonzepts der Lernziel- und Performanzzielorientierung und der empirischen Erkenntnisse im Rahmen dieser Herangehensweise lässt sich zeigen, dass die Induktion von Performanzzielorientierung die Erfolg versprechendere Herangehensweise ist. Auf dieser theoretischen Basis wurden die Lehr-Muster SINN AUFZEIGEN, MONITORING, GEZIELTE ÜBERFORDERUNG *und* ENTWICKLUNG SPÜRBAR MACHEN *entwickelt und im Studiengang Software Engineering an der Hochschule Heilbronn in der Erstsemesterveranstaltung »Arbeitstechniken in der Informationstechnologie« eingesetzt.*

1 Studentische Motivation für überfachliche Arbeitsgrundlagen

»Software-Engineering-Firmen sind in erster Linie textproduzierende Unternehmen« sagte Ernst Denert, Gründer von sd&m, auf der SEUH 2007. Überfachliche Arbeitsgrundlagen wie wirkungsvolles Kommunizieren, Recherchieren und eine sprachlich angemessene Darstellung von Inhalten sind unabdingbar für den Erfolg im Software Engineering. Einige Informatik- und Software-Engineering-Studiengänge haben mittlerweile auch Veranstaltungen integriert, die die Vermittlung von sozialen, personalen und methodischen Kompetenzen fokussieren.

Die Studierenden sind über diese Ansprüche an Berufsanfängerinnen und -anfänger informiert, gleichwohl lässt sich im Alltag der Lehre immer wieder beobachten, dass die Studierenden wenig motiviert sind, die für das Erlernen der überfachlichen Qualifizierung nötigen Anstrengungen und Aufwände zu investieren. Vielmehr scheint aus der Perspektive ihrer Studienfachwahl die Vermittlung technischer Inhalte als die »eigentliche« Lehre, während für die überfachliche Qualifizierung die Erwartung gilt, dass diese eine geringe Rolle spielt oder »en passant« vermittelt wird.

Entsprechend muss von einer geringen intrinsischen Motivation in entsprechenden Lehrveranstaltungen ausgegangen werden. Um in diesem Zusammenhang didaktische Situationen erfolgreich zu gestalten, kann das Konzept der verschiedenen Lernzielorientierungen eine geeignete Basis liefern. Es geht dort um die Unterscheidung zwischen Performanzziel- und Lernzielorientierung als motivationale Grundunterscheidung. Diese wird im Folgenden dargestellt, und es wird die Umsetzung in einer Pilotveranstaltung im ersten Semester des Bachelorstudiengangs Software Engineering an der Hochschule Heilbronn geschildert. Anschließend werden Lehr-Muster für die Gestaltung von Lehrveranstaltungen extrahiert, die Performanzzielorientierung induzieren.

2 Performanz- und Lernzielorientierung

Voraussetzung für motiviertes Handeln ist, dass die Person ein Ziel hat [Heckhausen/Heckhausen 2006]. Erreicht man durch eine Handlung mehrere Ziele, so spricht man von Multifinalität [Shah/Kruglanski 2000]. Diese Multifinalität spielt insbesondere bei der Lernmotivation eine wichtige Rolle [Heckhausen/Rheinberg 1980; Rheinberg 1989] – ein gutes Lernergebnis kann mit ganz verschiedenen Zielsetzungen angestrebt werden.

Die verschiedenen Arten von Lernzielen lassen sich auf unterschiedliche Art und Weise klassifizieren. Eine Unterscheidung stellt *learning* bzw. *mastery goal orientation* der *performance goal orientation* gegenüber [Dweck/Leggett 1988; Nicholls 1984]: Lernzielorientierung ist dadurch gekennzeichnet, dass die Person danach strebt, die Lerninhalte zu kennen und zu verstehen. Sie möchte den Lernstoff beherrschen, ihr Ziel ist also der Erwerb von Wissen und Kompetenzen. Bei Performanzzielorientierung fokussiert die Person weniger auf den Lernzuwachs als auf das Leistungsziel, d.h. auf den Maßstab, an dem die Zielerreichung gemessen wird. Es geht der Person darum, eine positive Beurteilung der eigenen Kompetenz zu erhalten bzw. negative Einschätzungen über die eigene Kompetenz zu vermeiden.

Im Kontext akademischer Lehre untersuchen Husman und ihre Kollegen [Husman/Lens 1999; Husman et al. 2004] die verschiedenen Zielorientierungen als Instrumentalität vom Erfolg in Lehrveranstaltungen: Sie zeigen auf, dass die Lernzielorientierung eher dadurch gekennzeichnet ist, dass die künftige Berufstä-

tigkeit in dem gewählten Fachgebiet aus sich selbst heraus für die Person attraktiv ist und der jeweilige Lerninhalt in einem inneren Zusammenhang mit dem avisierten Berufsbild gesehen wird. Performanzzielorientierung von Studierenden dagegen ist auf das Erreichen einer guten Benotung bzw. an der direkten Instrumentalität des dokumentierten Lernerfolgs für die Zulassung zu einem weiterführenden Studienprogramm oder den Einstieg in einen guten Job fokussiert.

In Lernziel- und Performanzzielorientierung findet sich auch die Unterscheidung zwischen intrinsischer und extrinsischer Motivation wieder [Rheinberg 2006]. Lernzielorientierung, d.h. das Streben danach, Kompetenzen zu erwerben, ist dabei eine intrinsische Lernmotivation, Performanzzielorientierung, d.h. der Wunsch, seine Kompetenzen zu demonstrieren, externe Lernmotivation. Nun gilt gemeinhin die interne Motivation als die echte, nachhaltige, wirkliche Motivation, während externe Motivation meist weniger positiv als »entfremdet« [Csikszentmihalyi 2005] angesehen wird. Nach wie vor unbeantwortet ist die Frage, wie intrinsische oder extrinsische Motivation interagiert und welche der beiden Motivationsformen leistungsförderlicher ist [Cameron et al. 2001; Deci et al. 1999; Eisenberger/Cameron 1996; Eisenberger et al. 1999]. Dies liegt nicht zuletzt daran, dass die Konzeptionalisierungen von intrinsischer und extrinsischer Motivation so unterschiedlich sind, dass der gemeinsame Kern, der dann auch als Basis für eine empirische Überprüfung in Form von Metaanalysen zugrunde gelegt werden könnte, nicht ausmachbar ist [Rheinberg 2006; Sansone/Harackiewicz 2000].

Bezogen auf die verschiedenen Zielorientierungen im Lernkontext argumentieren einige, dass auch hier intrinsische Motivation, in diesem Fall also Lernzielorientierung, positive Effekte auf Lernen und Lernerfolg habe, Performanzzielorientierung dagegen korrumpiere Lernerfolg und Leistung [Dweck 1986; Dweck/Leggett 1988, Nicholls 1984]. Allerdings ließ sich dieser positive Effekt von Lernzielorientierung auf die Leistung in vielen Studien nicht nachweisen [Covington/Omelich 1984; Yates 2000]. Vielmehr zeigen sich wiederholt positive Einflüsse von Performanzzielorientierung auf Leistung und Lernerfolg im akademischen Kontext [Barron/Harackiewicz 2001; Elliot/McGregor 2001; Elliot et al. 1999; Pintrich 2000; Hidi/Harackiewicz 2000; zur Debatte siehe Sansone/Harackiewicz 2000]. Performanzzielorientierung, so argumentieren [Harackiewicz et al. 2002], wirkt sich positiv auf die Motivation in Hochschulveranstaltungen aus, weil sie die eigene Kompetenz salient werden lasse und zur Mobilisierung von Anstrengung führt.

Auch eigene Untersuchungen im Rahmen von hochschuldidaktischen Untersuchungen zeigen wiederholt, dass ein kurzfristiger Fokus auf das Performanzziel, in diesem Fall das erfolgreiche Absolvieren einer Lehrveranstaltung, zu positiven Effekten führt: So konnte beispielsweise in einem quasi experimentellen Design gezeigt werden, dass der explizite Hinweis auf die Notwendigkeit, für eine gute Note auch gute Qualität abzuliefern, zu subjektiv besseren Leistungen führt

[Kasparek/Marsden 2007]. In einer weiteren Untersuchung zeigte sich als zentraler Effekt der Performanzzielorientierung eine deutlich positive Einschätzung der Lernenden vom Organisiertheitsgrad ihres Teams und der konstruktiven Zusammenarbeit innerhalb des Teams. Auch die Zufriedenheit mit den von einem selbst und vom Team produzierten Ergebnissen war bei Performanzzielorientierung höher als bei Lernzielorientierung [Marsden 2008].

3 Induktion von Performanzzielorientierung im Studiengang Software Engineering

Zur Vermittlung von überfachlicher Qualifizierung gibt es im Studiengang Software Engineering an der Hochschule Heilbronn seit Beginn des Studiengangs vor über zehn Jahren eine Lehrveranstaltung im ersten Semester. Es handelt sich bei der Lehrveranstaltung »Arbeitstechniken in der IT« um eine Pflichtveranstaltung, d.h., im Schnitt nehmen circa 40 Studierende teil, dabei sind meist weniger als 10 % Studentinnen. Die Veranstaltung umfasst 2 Semesterwochenstunden und 2 ECTS, d.h., die Studierenden haben rund 20 Stunden Präsenzzeit und arbeiten rund 30 Stunden außerhalb der Veranstaltung für das Fach. Inhaltlich deckt diese Veranstaltung die Themen Recherche, schriftliche Darstellung von Sachverhalten, Präsentation, Lerntechniken und Kommunikation ab. Es handelt sich um eine sogenannte Prüfungsvorleistung, d.h., es gibt keine Note, sondern nur die Möglichkeit zu bestehen oder nicht zu bestehen. Der Leistungsnachweis erfolgt anhand einer Ausarbeitung und eines schriftlichen Abschlusstests. Eine Quote von fast 50 % der Studierenden, die die Veranstaltung beim ersten Versuch nicht erfolgreich absolvierten, führte die Autorin in den Jahren 2005 und 2006 zunächst dazu, in kollegialer Supervision den Schwierigkeitsgrad der Inhalte zu überprüfen. Im Rahmen dieses Prozesses stellte es sich für beteiligten Lehrenden jedoch so dar, dass es nicht an zu schwierigen Inhalten, sondern an der aus oben genannten Gründen zu geringen Ausgangsmotivation der Studierenden lag, dass diese die geforderten Leistungen nicht erbrachten.

In Folge wurde im Jahr 2007 eine Überarbeitung der Lehrveranstaltung pilotiert, um der motivationalen Ausgangssituation der Studierenden für überfachliche Arbeitstechniken in der Studieneingangsphase Rechnung zu tragen. Die Erstsemesterveranstaltung »Arbeitstechniken in der Informationstechnologie« wurde so überarbeitet, dass sie die Induktion von Performanzzielorientierung fördert. Aufbau und Instruktionen wurden in der im Folgenden dargestellte Art und Weise eingesetzt.

3.1 Sinn aufzeigen

Bei der Veranstaltung »Arbeitstechniken« handelt es sich um eine Prüfungsvorleistung, d.h., die Leistung der Studierenden wird nicht benotet, sondern die Veranstaltung kann nur bestanden bzw. nicht bestanden werden. Sinnhaft und relevant im Sinne der Performanzzielorientierung sind Lerninhalte entsprechend nur dann, wenn sie unmittelbar zum Bestehen der Veranstaltung beitragen. Insofern wurden die im Rahmen der Veranstaltung angebotenen Lerninhalte so aufbereitet und dargestellt, dass den Studierenden die Relevanz zum erfolgreichen Absolvieren der Lehrveranstaltung unmittelbar klar war. Hierzu wurde beispielsweise für jeden inhaltlichen Baustein aufgezeigt, dass und in welcher Form er für das Bestehen der Veranstaltung relevant ist: Entweder es handelte sich um einzelne Aufgabenstellungen, wie etwa die Dokumentation einer Recherche, die die Studierenden veranstaltungsbegleitend termingerecht absolvieren mussten, oder es handelte sich um Inhalte, die im Rahmen der Veranstaltung geübt wurden und dann für den Abschlusstest relevant waren.

3.2 Monitoring

Die Struktur der Lehrveranstaltung, die Veranstaltungsmaterialien wie Folien, Unterlagen, Links etc., der für die einzelnen Aufgaben kalkulierte Arbeitsaufwand für die Studierenden und die Aufgaben zu den jeweiligen Lerninhalten waren auf der Learning-Managment-Plattform Ilias[1] hinterlegt.

Die Aufgaben bestanden in einer Umsetzung und einem Transfer der in der Lehrveranstaltung bearbeiteten Inhalte, sie forderten eine zeitnahe Erstellung und Abgabe der Arbeitsergebnisse auf der Lernplattform. Nach einer Durchsicht der individuellen Abgaben gab es für die Studierenden ein Feedback im Rahmen der Veranstaltung über Fehler und Verbesserungsmöglichkeiten, die bei den meisten Studierenden aufgetreten waren. Es gab dann für jede einzelne Studentin und jeden einzelnen Studenten individuelle Hinweise über die Spezifika – positiv wie negativ – ihres oder seines Arbeitsergebnisses. Darüber hinaus wurde festgehalten, ob er oder sie die Aufgabe bestanden hat. War dies nicht der Fall, so hatten die Studierenden die Möglichkeit, eine neue oder überarbeitete Fassung abzugeben. Bei einzelnen Aufgaben gab es dann, wenn auch die zweite Fassung noch mangelhaft war, die Möglichkeit und Aufforderung, in einem individuellen Gespräch mit einer Tutorin die nötigen Verbesserungen zu besprechen. Außerdem gab es die Möglichkeit, jederzeit in einem Forum auf der Lernplattform Fragen an die Lehrenden zu stellen, die binnen eines Tages beantwortet wurden, oder face-to-face bei der Professorin oder Tutorin nachzufragen.

1 *www.ilias.uni-koeln.de*

3.3 Gezielte Überforderung

Die Themen der Lehrveranstaltung »Arbeitstechniken« setzen inhaltlich und vom Schwierigkeitsgrad auf einem Niveau auf, dessen Erreichen auch in den Lehrplänen der weiterführenden Schulen vorgesehen ist. Es ist somit eine Verzahnung mit dem erreichten Kompetenzniveau angestrebt, die eine optimale Passung der Kompetenz der Studierenden mit dem Schwierigkeits- und Komplexitätsgrad der Aufgabenstellungen sicherstellt. Beim Thema Kommunikation beispielsweise wird an das schon bekannte Vier-Seiten-Modell [Schulz von Thun et al. 2003] angeknüpft, um auf dessen Basis Gesprächstechniken zu entwickeln und auf für die Studierenden im Studium und in ihrem späteren Berufsleben relevante Situationen zu adaptieren. Im Sinne der Performanzzielorientierung wird auch hier in der Instruktion immer wieder verdeutlicht, dass und in welcher Form das Beherrschen dieser Gesprächstechniken für das Erlangen des Scheins relevant ist. Bei der Vermittlung der Lerninhalte soll der Schwierigkeitsgrad dabei zu einer gezielten Überforderung führen, d.h., er soll genau oberhalb des schon vorhandenen Kompetenzniveaus liegen. Um die interindividuellen Unterschiede im Kompetenzniveau zu berücksichtigen, werden die kurzen Feedbackschleifen im Rahmen des Monitorings genutzt, um bei absoluter Unangemessenheit der vorhandenen Kompetenz bei einzelnen Studierenden individuell nachzusteuern.

3.4 Entwicklung spürbar machen

Dort, wo im Rahmen der Lehrveranstaltung »Arbeitstechniken« aufeinander aufbauende Inhalte vermittelt werden, bekommen die Studierenden die Möglichkeit, für die einzelnen Schritte schon Rückmeldung über den eigenen Erfolg oder Misserfolg zu bekommen, um dann im weiteren Lern- und Arbeitsprozess im Rahmen der Veranstaltung ein Gefühl dafür zu haben, wo sie derzeit stehen. Der Aufbau der Veranstaltung ist darauf angelegt, dass das Vorankommen für die Studierenden transparent und Dynamik damit erlebbar wird. So war es zum Beispiel so, dass vor der Optimierung der Veranstaltung »Arbeitstechniken« das Thema wissenschaftliches Arbeiten vermittelt wurde und die Studierenden dann selbstständig eine Recherche und eine kurze wissenschaftliche Ausarbeitung anfertigten. Nicht zuletzt aufgrund der Beobachtung, dass die Studierenden auf der Basis anfangs gemachter Fehler, grundsätzlicher Missverständnisse oder einer Fehleinschätzung der geforderten Gewissenhaftigkeit Ergebnisse ablieferten, die erkennbar schon zu Beginn der Arbeit angelegt waren, wurde die Gesamtdramaturgie in der hier beschriebenen Pilotveranstaltung im Jahr 2007 geändert und der Aufgabenblock unterteilt: Zunächst recherchierten die Studierenden ihr Thema in entsprechenden Datenbanken, dokumentierten diese Recherche und erhielten dann Rückmeldung über die Güte des Rechercheprozesses und der -ergebnisse, hatten dann bei Bedarf die Möglichkeit, diese Recherche zu wiederholen und zu optimieren. Dann formulierten sie ihre Fragestellung für die Aus-

arbeitung und gaben die zentralen Quellen ihrer Arbeit im verabredeten Zitierstil an. Auch hierzu erhielten sie Rückmeldung und die Möglichkeit, eine verbesserte Variante zu erstellen. Erst dann bearbeiteten die Studierenden die umfassendere Ausarbeitung, in der sie dann nach bis zu zweifacher individueller Rückmeldung wiederum optimierte Fassungen einreichen konnten. Im Rahmen dieser kontinuierlichen Rückmeldungsschleifen für Studierenden und Lehrende konnte also die eigene Performanz im Rahmen der Lehrveranstaltung nachgesteuert und die eigene Entwicklung damit spürbar werden.

4 Lehr-Muster

Für die Lehre im Software Engineering ist die Unterscheidung in Performanz- und Lernzielorientierung außerordentlich bedeutsam: Im Kontext der Vermittlung von überfachlichen Arbeitsgrundlagen im Software Engineering kann wie oben dargestellt von einer geringen Motivation der Studierenden für diese Themen ausgegangen werden, somit ist die Induktion von Performanzzielorientierung Erfolg versprechender als die Erwartung, dass didaktisch auf Lernzielorientierung aufgesetzt werden kann. Wie sehen nun die Lehr-Muster aus, die zur Induktion von Performanzzielorientierung in der Veranstaltung eingesetzt wurden? Um die theoretischen Vorüberlegungen und empirischen Erkenntnisse auf den Alltag der Lehre im Software Engineering übertragbar zu machen, wurden Lehr-Muster abgeleitet, die bei geringer Ausgangsmotivation eingesetzt werden können.

Lehr-Muster stellen den Versuch dar, pädagogische Konzepte pointiert und griffig darzustellen und somit für eine möglichst großen Nutzergruppe zugänglich zu machen. Ziel von Mustern ist es, erfolgreiche Herangehensweisen in ihrer Essenz darzustellen und wiederverwerten zu können. Die Herangehensweise, didaktisches Wissen in Form von Lehr-Mustern aufzubereiten, wird bevorzugt in der Informatik eingesetzt [Booch 2008, Derntl/Buttori 2006]. Hier wird meist auf die Architektur [Alexander 1979] als Inspiration für den Einsatz von Muster referenziert, allerdings gibt es sehr viel ältere Wurzeln des Konzepts aus anderen Disziplinen [Coplien/Harrison 2004]. Die hier entwickelten Lehr-Muster werden in Anlehnung an die Darstellung des Pedagogical Pattern Project [Sharp et al. 2003] notiert und kann als Ergänzung zu den dort vorgestellten Mustern verstanden werden. Die Notation des Pedagogical Pattern Project baut auf der Darstellung von [Alexander et al. 1977] auf, die folgendermaßen aussieht: Es gibt vier Abschnitte in der Beschreibung des Musters – zunächst wird der

- **Kontext** zum Einsatz des Musters skizziert, dann werden die
- **Einflusskräfte** und das **Kernproblem** vorgestellt. Es folgt die
- **Lösung** mit den Konsequenzen, Limitierungen und Einschränkungen. Abschließend kommt die
- **Diskussion** mit weiteren Informationen und Beispielen.

Die in Abbildung 1 – 4 dargestellten Lehr-Muster sind für die Vermittlung von überfachlichen Qualifikationen im Bereich Software Engineering entwickelt worden und speziell darauf ausgerichtet, bei Studierenden mit geringer Ausgangsmotivation zum Einsatz zu kommen. Gleichwohl sind sie anzusiedeln im Kontext der Pattern Languages des Pedagogical Pattern Project [Sharp et al. 2003] und sollten in Kombination mit den dort vorgestellten Lehr-Mustern eingesetzt werden. Verwandte Lehr-Muster sind also Muster für aktives Lernen [Eckstein et al. 2002a], Muster für Erfahrungslernen [Eckstein et al. 2002b], Muster, um eine andere Perspektive zu erlangen [Bergin et al. 2001] und insbesondere Feedback-Muster [Eckstein et al. 2003].

SINN AUFZEIGEN

Sie führen eine Lehrveranstaltung durch, in der den Studierenden der Sinn des zu Lernenden für ihre berufliche Zukunft nicht klar ist.

* * *

Die Studierenden zeigen nur wenig Engagement und sind nur **wenig motiviert, weil sie das Gefühl haben, dass die Lerninhalte wenig zu tun haben mit dem, was sie »eigentlich« im Studium lernen möchten**.

* * *

In dem Ausmaß, in dem das Thema an sich für die Studierenden nicht motivierend ist, muss diese Sinnhaftigkeit darüber vermittelt werden, dass den Studierenden für jeden Lerninhalt verdeutlicht wird, wie sie hier ihre Kompetenzen demonstrieren und damit ihren Studienerfolg beeinflussen können. Im Rahmen der Lehrveranstaltung und in den Instruktionen für die einzelnen Aufgaben wird **immer wieder explizit auf die Sinnhaftigkeit und Relevanz der Lerninhalte hingewiesen** – und zwar mit einem **Fokus auf der kurzfristigen Relevanz zum erfolgreichen Absolvieren der Lehrveranstaltung und des Studiums**.

* * *

Wahrscheinlich widerstrebt es Ihnen, die Studierenden kontinuierlich darauf hinzuweisen, wie sie mit den Lerninhalten umgehen müssen, um hier eine gute Leistung zu zeigen. Sie wünschen sich, dass die Studierenden die Relevanz »Ihres« Faches erkennen bzw. dass es Ihnen gelingt, die Begeisterung dafür auf die Studierenden zu übertragen und diese intrinsisch motiviert sind. Aber: Lernerfolg kann sich auch durch Performanzzielorientierung einstellen – und zwar im beschriebenen Kontext mit weniger Zeitaufwand und größerem Erfolg.

Abb. 1 *Lehr-Muster SINN AUFZEIGEN (Version 1.0)*

MONITORING

Sie führen eine Lehrveranstaltung durch und nutzen die Lehr-Muster des aktivierenden Lernens und Feedback-Muster.

* * *

Die Studierenden halten überfachliche Qualifikationen teilweise für trivial und unterschätzen die Notwendigkeit, sie systematisch zu erlernen bzw. **überschätzen die eigenen Kompetenzen in diesem Bereich.**

* * *

Bieten Sie die Lerninhalte klar strukturiert dar und lassen Sie die Studierenden zu den jeweiligen Lerninhalten Aufgaben bearbeiten, die eine direkte Umsetzung des Gelernten ermöglichen und fordern. Geben Sie den Studierenden veranstaltungsbegleitend **kontinuierlich individuell und zeitnah Rückmeldung** zu ihren Aufgaben, so dass sie **über ihren Erfolg bzw. Misserfolg im Bilde sind und die Gelegenheit haben, durch Optimierungen entsprechend nachzusteuern.**

* * *

Um ein wirkliches Monitoring zu ermöglichen und nicht nur eine Summe von Rückmeldungen zu geben ist es wichtig, dass die Aufgaben aufeinander aufbauen. Größere Aufgaben wie zum Beispiel das Schreiben einer wissenschaftlichen Ausarbeitung sollten in kleinen Chunks heruntergebrochen werden, zum Beispiel in Recherche, Fragestellung, Gliederung und Fertigstellung. Wichtig ist auch, dass die Abgaben intensiv geund auch überprüft werden: Plagiate müssen aufgedeckt werden, das Durchführen der Recherche muss durch Screenshots belegt werden – da dieses Muster in einem Kontext mit geringer intrinsischer Motivation zum Einsatz kommt, ist es zudem wichtig, für jeden Chunk darzustellen, dass und wie er zum Performanzziel beiträgt.

Abb. 2 *Lehr-Muster MONITORING (Version 1.0)*

GEZIELTE ÜBERFORDERUNG

Sie führen eine Lehrveranstaltung durch, bei der die intrinsische Motivation für das Thema gering ist. Deshalb strengen die Studierenden sich wenig an. Somit ist auch das Leistungsniveau im Rahmen der Lehrveranstaltung relativ niedrig.

* * *

Aufgrund des geringen Leistungsniveaus besteht die Gefahr, dass Sie den **Leistungsanspruch an dieses (geringe) Niveau anpassen** – nicht zuletzt, um eine zu hohe Durchfallerquote zu vermeiden.

* * *

Setzen Sie den Schwierigkeits- und Komplexitätsgrad der Aufgaben so, dass die **Studierenden massiv gefordert, tendenziell überfordert werden.** Wichtig ist eine entsprechende Passung, es darf nicht zu einer völligen Überforderung der Lernenden kommen, denn das Erleben der absoluten Unangemessenheit der eigenen Kompetenz oder ein völliges Infragestellen der bisherigen Herangehensweisen würde zu einem systematischen Ausschluss von Informationen führen.

* * *

* * *

Natürlich besteht neben der Gefahr des Milde-Effekts (»Das kann ich ja nicht machen, so viele durchfallen zu lassen«) auch die Möglichkeit, dass Sie zum Strenge-Effekt neigen (»Das darf ja wohl nicht wahr sein, dass die so eine simple Aufgabe so schlurig bearbeiten«). Um gezielt zu überfordern, brauchen Sie auf der einen Seite den Mut, den Leistungsanspruch auf ein hohes Niveau zu setzen, auf der anderen Seite eine realistische Einschätzung (basierend auf Kenntnis der bisherigen Schulbildung etc.), was von den Studierenden verlangt werden kann. Und nicht zuletzt brauchen Sie eine gemeinsame Herangehensweise der Lehrenden des Studiengangs, wie mit dem Anspruchsniveau und Studierenden, die diese Ansprüche nicht erreichen, umgegangen wird.

Abb. 3 *Lehr-Muster GEZIELTE ÜBERFORDERUNG (Version 1.0)*

ENTWICKLUNG SPÜRBAR MACHEN

Sie führen eine Lehrveranstaltung durch, deren Inhalte die Studierenden für sich selbst (noch) nicht als relevant erkannt haben.

* * *

Die Studierenden entwickeln **wenig Ehrgeiz**, sich mit den Themen auseinanderzusetzen, da sie den Bezug zu den eigenen Zielen nicht wahrnehmen.

* * *

Sprechen Sie grundlegendere Ziele und Motive der Studierenden an, wie zum Beispiel den Wunsch, sich als Person weiterzuentwickeln und besser zu werden. Legen Sie die **Dramaturgie der Lehrveranstaltung** so an, dass die Studierenden erleben, wie die eigene **Performanz im Laufe der Zeit besser** wird, die Aufgaben komplexer werden und die **eigene Entwicklung** dadurch **spürbar** wird.

* * *

Menschen haben grundsätzlich den Wunsch nach Konsistenz, d.h., sie vermeiden Dissonanz zwischen dem, was sie tun und was sie über sich selbst annehmen. Wenn Sie die Studierenden dazu bringen, sich für ein Fach anzustrengen und gute Leistungen zu erbringen, ist die Wahrscheinlichkeit hoch, dass die Studierenden dieses Fach dann auch für sich selbst als wertvoll anerkennen, um kognitive Dissonanz zu reduzieren. Insofern ist es durchaus Erfolg versprechend, nicht an der Einstellung zum Thema anzusetzen, sondern am Aufwand, den sie für das Thema betreiben.

Abb. 4 *Lehr-Muster ENTWICKLUNG SPÜRBAR MACHEN (Version 1.0)*

5 Ergebnisse aus der Pilotveranstaltung

Die Veränderung der Lehrveranstaltung »Arbeitstechniken in der Informationstechnologie« im Studiengang Software Engineering entsprechend der Lehr-Muster SINN AUFZEIGEN, MONITORING, GEZIELTE ÜBERFORDERUNG und ENTWICKLUNG SPÜRBAR MACHEN hat zu deutlichen motivationalen Veränderungen bei den Studierenden hinsichtlich der Veranstaltungsinhalte geführt.

Die Veranstaltung wird seit der Pilotierung im Jahr 2007 nun zum vierten Mal in der dargestellten Form durchgeführt. Die dargestellten Änderungen mit dem extrem hohen Korrektur- und Rückmeldeaufwand während des Semesters wurden zunächst von der Dozentin selbst ohne Tutorinnen oder Tutoren durchgeführt. Dies erwies sich jedoch auf Dauer als nicht tragbar: Mittlerweile gibt es eine wissenschaftliche Mitarbeiterin, die aus Studiengebühren finanziert ist und einen Teil ihrer Kapazität dieser Veranstaltung zur Verfügung stellt.

Der offensichtlichste Effekt besteht darin, dass die Studierenden durch eine rigorose Performanzzielorientierung in Dramaturgie und Instruktion ein deutlich höheres Maß an Anstrengung und Aufwand in den Erwerb von überfachlichen Arbeitsgrundlagen investiert haben. Ob damit auch ein höherer Lernerfolg einhergeht, kann nur teilweise überprüft werden: Die Leistungsanforderungen sind im Rahmen der Überarbeitung der Lehrveranstaltung deutlich profilierter geworden, so dass die Kriterien für das Bestehen der Veranstaltung transparenter – und damit auch leichter verfehlbar – geworden sind. Einerseits ist anhand der Qualität der resultierenden Ausarbeitungen und Abschlusstests deutlich nachweisbar, dass der Lernerfolg insgesamt höher ist als vor der Überarbeitung der Lehrveranstaltung, es sind mehr Kompetenzen erworben und mehr Lerninhalte verinnerlicht worden. Andererseits haben dieses Anziehen der Leistungsanforderungen und das konsequente Überprüfen der Performanz der Studierenden zur Folge, dass mehr Studierende als vorher an diesen Ansprüchen scheitern. Dies ist, so muss im Sinne einer kritischen Selbstreflexion von Seiten der Lehrenden festgestellt werden, nicht zuletzt eine Folge davon, dass die für das Bestehen der Veranstaltung nötige Performanz von vorneherein klar strukturiert und verdeutlicht wird: Denn dann müssen sich nicht nur die Studierenden an diesem Maßstab messen lassen, sondern auch die Lehrenden haben keine andere Wahl, als die avisierte Performanz abzuverlangen.

Danksagung

Die Untersuchung der motivationalen Verankerung von überfachlichen Arbeitsgrundlagen in der Studieneingangsphase ist vom Land Baden-Württemberg im Rahmen des LARS-Programms (Leistungsanreizsysteme in der Lehre) gefördert worden.

Literatur

[Alexander 1979] C. Alexander: The timeless way of building. New York: Oxford University Press, 1979

[Alexander et al. 1977] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel: A pattern language—towns, buildings, construction. New York: Oxford University Press, 1977

[Barron/Harackiewicz 2001] K.E. Barron, J.M. Harackiewicz: Achievement goals and optimal motivation: testing multiple goal modes. Journal of Personality and Social Psychology, 80, 706-722, 2001

[Bergin et al. 2001] J. Bergin, J. Eckstein, M.L. Manns, E. Wallingford: Patterns for Gaining Different Perspectives. In: Proceedings of PLoP 2001, 2001

[Booch 2008] G. Booch: Architectural Organizational Patterns. Software IEEE, 25(3), 18-19, 2008

[Cameron et al. 2001] J. Cameron, K.M. Banko, W.D. Pierce: Pervasive negative effects of rewards on intrinsic motivation: The myth continues. The Behavior Analyst, 24, 1-44, 2001

[Coplin/Harrison 2004] J.O. Coplin, N.B. Harrison: Organizational Patterns for Agile Software Development. Upper Saddle River: Pearson Prentice Hall, 2004

[Covington/Omelich 1984] M.V. Covington, C.L. Omelich: Task-oriented versus competitive learning structures: Motivational and performance consequences. Journal of Educational Psychology, 76, 1038-1150, 1984

[Csikszentmihalyi 2005] M. Csikszentmihalyi: Das Flow-Erlebnis. Jenseits von Angst und Langeweile[9. Aufl.]. Stuttgart: Klett-Cotta, 2005

[Deci et al. 1999] E.L. Deci, R. Koestner, R.M. Ryan: A meta-analytic review of experiments examining the effects of extrinsic rewards on intrinsic motivation. Psychological Bulletin, 125, 627-668, 1999

[Derntl/Botturi 2006] M. Derntl, L. Botturi: Essential use cases for pedagogical patterns. Computer Science Education, 16(2), 137-156, 2006

[Dweck 1986] C.S. Dweck: Motivational processes affecting learning. American Psychologist, 41, 1040-1048, 1986

[Dweck/Leggett 1988] C.S. Dweck, F.L. Leggett: A social-cognitive approach to motivation and personality. Psychological Review, 95, 256-273, 1988

[Eckstein et al. 2002a] J. Eckstein, J. Bergin, H. Sharp: Patterns for Active Learning. In: Proceedings of PLoP 2002, 2002

[Eckstein et al. 2002b] J. Eckstein, M.L. Manns, E. Wallingford, K. Marquardt: Patterns for Experiential Learning. In: Proceedings of EuroPLoP 2001, Universitätsverlag Konstanz, Konstanz, 2002

[Eckstein et al. 2003] J. Eckstein, J. Bergin, H. Sharp: Feedback Patterns. In: Proceedings of EuroPLoP 2002, 2003

[Eisenberger/Cameron 1996] R. Eisenberger, J. Cameron: The detrimental effects of reward: Myth or reality? American Psychologist, 51, 1153–1166, 1996

[Eisenberger et al. 1999] R. Eisenberger, W.D. Pierce, J. Camron: Effects of reward on intrinsic motivation: Negative, neutral, and positive. Psychological Bulletin, 125, 677–691, 1999

[Elliot/McGregor 2001] A.J. Elliot, H.A. McGregor: A 2 x 2 achievement goal framework. Journal of Personality and Social Psychology, 80, 501-519, 2001

[Elliot et al. 1999] A.J. Elliot, H.A. McGregor, S. Gable: Achievement goals, study strategies, and exam performance: A mediational analysis. Journal of Educational Psychology, 91, 549-563, 1999

[Harackiewicz et al. 2002] J.M. Harackiewicz, K.E. Barron, P.R. Pintrich, A.J. Elliot, T.M. Thrash: Revision of achievement goal theory: Necessary and illuminating. Journal of Educational Psychology, 94, 638-645, 2002

[Heckhausen/Heckhausen 2006] J. Heckhausen, H. Heckhausen: Motivation und Handeln. Heidelberg: Springer, 2006

[Heckhausen/Rheinberg 1980] H. Heckhausen, F. Rheinberg: Lernmotivation im Unterricht, erneut betrachtet. Unterrichtswissenschaft, 8, 7-47, 1980

[Hidi/Harckiewicz 2000] S. Hidi, J.M. Harackiewicz: Motivating the academically unmotivated. A critical issue for the 21st century. Review of Educational Research, 70, 151-179, 2000

[Husman/Lens 1999] J. Husman, W. Lens: The role of the future in student motivation. Educational Psychologist, 34 , 113–125, 1999

[Husman et al. 2004] J. Husman, W.P. Derryberry, H.M. Crowson, R. Lomax: Instrumentality, task value, and intrinsic motivation: making sense of their independent interdependence. Contemporary Educational Psychology, 29, 63-76, 2004

[Kasparek/Marsden 2007] M. Kasparek, N. Marsden: Einfluss von Qualitätsdruck und Kontinuität der Zusammenarbeit auf virtuelle Teamarbeit. In Andreas Zeller & Marcus Deininger[Hrsg.]: Software Engineering im Unterricht der Hochschulen – SEUH 10 [S. 83-85]. Heidelberg: dpunkt.verlag, 2007

[Marsden 2008] N. Marsden: Motivation durch Instruktion? Effekte der Induktion von Performanzzielorientierung. Zeitschrift für E-Learning, Nr. 1, 3. Jahrgang, S. 32-44, 2008

[Nicholls 1984] J.G. Nicholls: Achievement motivation: Conceptions of ability, subjective experience, task choice, and performance. Psychological Review, 91, 328-346, 1984.

[Pintrich 2000] P.R. Pintrich: Multiple goals, multiple pathways: The role of goal orientation in learning and achievement. Journal of Educational Psychology, 92, 544-555, 2000

[Rheinberg 1989] F. Rheinberg: Zweck und Tätigkeit. Göttingen: Hogrefe, 1989

[Rheinberg 2006] F. Rheinberg: Intrinsische Motivation und Flow-Erleben. In Jutta Heckhausen & Heinz Heckhausen [Hrsg.], Motivation und Handeln [S. 331-354]. Heidelberg: Springer, 2006

[Sansone/Harackiewicz 2000] C. Sansone, J.M. Harackiewicz: Intrinsic and extrinsic motivation. San Diego: Academic Press, 2000

[Schulz von Thun et al. 2003] F. Schulz von Thun, J. Ruppel, R. Stratmann: Miteinander reden – Kommunikationspsychologie für Führungskräfte. Reinbek bei Hamburg: Rowohlt-Taschenbuch-Verlag, 2003

[Shah/Kruglanski 2000] J.Y. Shah, A.W. Kruglanski: The structure and substance of intrinsic motivation. In C. Sansone & J. M. Harackiewicz [Eds.], Intrinsic and extrinsic motivation [pp. 105-127]. San Diego: Academic Press, 2000

[Sharp et al. 2003] H. Sharp, M.L. Manns, J. Eckstein: Evolving Pedagogical Patterns: The Work of the Pedagogical Patterns Project. Computer Science Education, 13(4), 315-330, 2003

[Yates 2000] Task involvement and ego orientation in mathematics achievement: A three year follow-up. Issues in Educational Research, 10, 77-91, 2000

Entwicklung eingebetteter Software in einem Softwarepraktikum mit Lego Mindstorms[1]

André Heuer · Kim Lauenroth · Vanessa Stricker · Klaus Pohl

Software Systems Engineering
Institut für Informatik und Wirtschaftsinformatik (ICB)
Universität Duisburg-Essen
{andre.heuer|kim.lauenroth|vanessa.stricker|klaus.pohl}@sse.uni-due.de

Zusammenfassung

In der Bachelor-Veranstaltung Softwareentwicklung und Programmierung (SEP) führen die Studierenden in einem Softwareprojekt die wesentlichen Aktivitäten der Softwareentwicklung vom Requirements Engineering über die Implementierung bis zum Softwaretesten durch. Um den Studierenden einen Einblick in die Besonderheiten und Herausforderungen der Softwareentwicklung für eingebettete Systeme zu geben, wurden mit Hilfe von Studiengebühren Lego-Mindstorms-Baukästen für diese Veranstaltung erworben und eingeführt.

In diesem Beitrag werden die Herausforderungen, deren Lösungsmöglichkeiten und Erfahrungen bei der Durchführung des SEPs mit Lego Mindstorms beschrieben und nötige Anpassungen der Veranstaltung präsentiert. Die Akzeptanz der Einführung der Lego Mindstorms wird anhand einer Umfrage belegt. Zudem vermittelt der Beitrag hilfreiche Informationen zur Durchführung ähnlicher Lehrveranstaltungen mit Lego Mindstorms.

1 Dieser Beitrag wurde teilweise gefördert durch das BMBF-Projekt REMsES, Fördernummer 01 IS F06 D.

1 Einleitung

Die Veranstaltung Softwareentwicklung und Programmierung (SEP) ist nach den aktuellen Prüfungsordnungen ein Bestandteil der praktischen Ausbildung in den Bachelor-Studiengängen Angewandte Informatik – Systems Engineering, Wirtschaftsinformatik sowie im Lehramtsstudiengang Informatik an der Universität Duisburg-Essen. Das SEP wird üblicherweise von Studierenden im 2. oder 3. Semester besucht und verfolgt drei wesentliche Zielsetzungen[2]:

- Die Vertiefung von Programmierkenntnissen
- Die Vermittlung von Soft-Skills, insb. die teamorientierte Arbeitsweise
- Die Vermittlung elementarer, methodischer Vorgehensweisen zur Softwareentwicklung

1.1 Motivation für die Einführung von Lego Mindstorms

Ein wesentliches Ziel des Studiengangs Angewandte Informatik – Systems Engineering ist die Vermittlung einer systemorientierten Sichtweise auf die Aufgabenstellungen bei der Entwicklung von Software und softwaregestützten Diensten. Eine Systemorientierung in der Softwareentwicklung trägt unter anderem der Tatsache Rechnung, dass die Entwicklung von Software für eingebettete Systeme in den vergangenen Jahren erheblich an Bedeutung gewonnen hat (vgl. [Liggesmeyer und Rombach 2005]). Zum Beispiel ist eingebettete Software ein wesentlicher Bestandteil von modernen Fahrzeugen (vgl. z.B. [Broy 2006]), Systemen in der Avionik, in der Automatisierungstechnik und in der Medizintechnik. Um die Systemorientierung in der Veranstaltung SEP zu verstärken und den Studierenden erste Einblicke in die Entwicklung eingebetteter Software zu bieten, werden seit dem Sommersemester 2008 Lego-Mindstorms-Baukästen in der Veranstaltung eingesetzt. Die Analogien zwischen typischen Bestandteilen eingebetteter Systeme und Lego-Mindstorms-Elementen (siehe Abschnitt 2.2 für eine Beschreibung der Elemente) sind in Tabelle 1 dargestellt.

Typische Bestandteile eingebetteter Systeme	Lego-Mindstorms-Elemente
Elektronische Steuerung	NXT-Baustein
Sensoren	Berührungs-, Licht-, Ultraschall-, Geräuschssensor
Aktuatoren	Servomotoren
Mechanik	Lego-Technik-Bausteine

Tab. 1 *Analogien zwischen eingebetteten Systemen und Lego Mindstorms*

2 Für weitere Details zur Veranstaltung SEP verweisen wir auf [Lauenroth et al. 2007].

1.2 Ziel und Aufbau des Beitrags

Der vorliegende Beitrag zeigt am Beispiel SEP die wesentlichen Herausforderungen bei der Einführung von Lego Mindstorms (LM) auf, stellt Lösungen für diese Herausforderungen vor und fasst die Erfahrungen bei der Verwendung von LM in einem Softwarepraktikum zusammen. Der Beitrag soll damit Verantwortliche von Softwarepraktika bei der Entscheidung über die Einführung von LM-Baukästen in der Lehre unterstützen.

Kapitel 2 gibt zu diesem Zweck einen kurzen Überblick über die Kosten der Anschaffung, die Bestandteile eines LM-Baukastens und die im SEP eingesetzte Programmierumgebung. Kapitel 3 zeigt, wie im SEP den folgenden Herausforderungen bei der Durchführung eines Softwarepraktikums mit LM begegnet wurde:

(H1) Einführung der Studierenden in die Softwareentwicklung für LM

(H2) Definieren einer geeigneten Aufgabenstellung für die Studierenden, welche die Besonderheiten der Softwareentwicklung für LM berücksichtigt

(H3) Anpassung des Ablaufs der Veranstaltung bedingt durch den Einsatz von LM

In Kapitel 4 wird ein Resümee aus dem Einsatz von Mindstorms in der Lehrveranstaltung gezogen. Dazu werden die Ergebnisse der im Anschluss an die Veranstaltung durchgeführten Evaluierung vorgestellt und die aufgrund der Verwendung von LM beobachteten Lerneffekte der Studierenden und der Veranstaltungsbetreuer beschrieben. Der Beitrag wird in Kapitel 5 mit einer Zusammenfassung und einem Ausblick abgeschlossen.

2 Lego-Mindstorms-Baukästen und Programmierumgebung

In diesem Abschnitt wird ein Überblick über Anschaffungskosten von LM-Baukästen gegeben (Abschnitt 2.1) und der Inhalt eines LM-Baukastens (Abschnitt 2.2) sowie die verwendete Programmierumgebung (Abschnitt 2.3) kurz vorgestellt.

2.1 Kosten für die Anschaffung

In der Veranstaltung SEP werden pro Semester zwischen 10 und 15 Gruppen bzw. Projektteams betreut. Für den Einsatz von LM in der Veranstaltung waren folgende Anschaffungen nötig:

- 18 LM-Baukästen
- Bluetoothdongles für jeden Baukasten, um die Kommunikation zwischen PCs und LM-Robotern zu ermöglichen.
- Sortierkästen für jeden Baukasten zur Aufbewahrung.

Insgesamt wurden für Baukästen, Bluetoothdongles und Sortierkästen ca. *4.000 Euro* investiert.

2.2 Inhalt des Baukasten

Der LM-Baukasten beinhaltet verschiedene Hardwareelemente, eine Software zur Programmierung der Roboter sowie Materialien, die die Inbetriebnahme unterstützen sollen. Diese Elemente werden im Folgenden beschrieben:

- **NXT-Baustein:**
  Das Kernstück des LM-Baukastens bildet der programmierbare NXT-Baustein. Dieser Baustein enthält einen 32-Bit-Mikroprozessor, 64-KB-Arbeitsspeicher, 256-KB-Flash-Speicher, Lautsprecher sowie Anschlüsse für bis zu vier Sensoren und drei Servomotoren. Weiterhin verfügt er über je eine USB 2.0- und eine Bluetooth 2.0-Schnittstelle, die die Kommunikation mit dem Baustein erlauben, ein LCD zur Darstellung von Ausgaben und vier Bedienknöpfe.
- **Sensoren:**
  Der LM-Baukasten enthält vier verschiedene Sensoren:
  - **Berührungssensor:**
    Der Berührungssensor erlaubt dem NXT zu erkennen, ob in der Umgebung des Roboters ein Hindernis erreicht wurde.
  - **Ultraschallsensor:**
    Der Ultraschallsensor ermöglicht es, Distanzen zu Objekten in Zentimeter oder Zoll mittels Ultraschall mit einer Reichweite von bis zu 255 cm und einer Präzision von +/- 3 cm zu messen, um somit auch auf Bewegungen in der Umgebung reagieren zu können.
  - **Lichtsensor:**
    Der Lichtsensor ist der zweite Sensor, der dem Roboter das Erkennen seiner Umgebung erlaubt, indem Farben anhand der Lichtintensität erkannt werden können.
  - **Geräuschsensor:**
    Der Geräuschsensor misst die Lautstärke in der Umgebung des Roboters in Dezibel (bis zu 90 db) und kann zum Beispiel auf lautes Klatschen reagieren.
- **Servomotoren:**
  Drei Servomotoren sind mit eingebauten Rotationssensoren sowie alle benötigten Kabel zur Verbindung der Motoren und Sensoren mit dem NXT-Baustein im Baukasten vorhanden. Die Servomotoren verwenden eingebaute Drehsensoren, die es erlauben, die Achsendrehung auf 1° genau zu beobachten und somit z.B. durch präzise Messung der Roboterbewegungen jederzeit in die ursprüngliche Startposition zurückzukehren [Bagnall 2007].

- **Legobausteine:**
  Der LM-Baukasten enthält über 500 Lego-Technik-Bausteine, u.a. einfache Bausteine, Stangen, Verbindungselemente, Zahnräder und Reifen, wodurch Roboter mit unterschiedlichen Funktionen und Mechaniken gebaut werden können. Darüber hinaus sind zwei Bälle und Zangenelemente enthalten.
- **Dokumentationsmaterialien:**
  Der Baukasten enthält verschiedene Dokumentationsmaterialen, die die Inbetriebnahme des NXT erleichtern. Neben einem Quickstart Guide wird außerdem eine Bedienungsanleitung bereitgestellt, die eine Einführung in die NXT-Technik enthält und das Zusammenbauen von vorgefertigten Modellen mit Hilfe der mitgelieferten Teile erläutert. Ein solches Model, genannt Tribot (siehe Abb. 1), wurde im SEP verwendet, da es durch die Verwendung von drei Rädern, aller Sensoren sowie einer Greifzange den Anforderungen aus der Aufgabenstellung (siehe Abschnitt 3.2) gerecht wird.

Abb. 1 *Foto eines Tribots*

- **Programmierumgebung:**
  Zur Programmierung des NXT-Bausteins wird die Software NXT-G (für PC oder Mac) zur Verfügung gestellt. NXT-G basiert auf der grafischen Programmiersprache labVIEW (vgl. [Georgi und Metin 2007]) und erlaubt die Programmierung des NXT-Bausteins mit Hilfe von grafischen Elementen.

2.3 Programmierumgebung leJOS

Die Programmiersprache Java (siehe z.B. [Horstmann und Cornell 2007]) wird im 1. Semester für alle Informatikstudiengänge gelehrt und ist daher den Studierenden im SEP bekannt. Um an die im ersten Semester vermittelten Java-Kenntnisse der Studierenden anzuknüpfen, wurde im SEP die Programmierumgebung leJOS (vgl. [Bagnall 2007]) zur Entwicklung von Software für den NXT-Baustein eingesetzt. LeJOS portiert einen Teil der Java Virtual Machine und stellt ein reduziertes Java für die Programmierung des NXT-Bausteins zur Verfügung. Es ist mit Windows, Linux und Macintosh kompatibel, so dass für die Studierenden keine Einschränkungen bezüglich ihres gewohnten Entwicklungssystems entstanden. Das Ausführen und Hochladen von bereits geschriebenem leJOS Programmcode ist über die Konsole möglich. Darüber hinaus wird ein Eclipse Plug-in für leJOS zur Verfügung gestellt, das die Programmierung der LM mit leJOS in Eclipse (siehe z. B. [Holzner 2005]) ermöglicht.

3 Einbindung von Lego Mindstorms in das Veranstaltungskonzept des SEP

In diesem Abschnitt werden die in Abschnitt 1.2 genannten Herausforderungen bei der Durchführung eines Softwarepraktikums mit LM aufgegriffen. Abschnitt 3.1 beschreibt, wie die Studierenden in die Programmierung von LM eingeführt wurden (H1). In Abschnitt 3.2 wird die Aufgabenstellung für den Einsatz von LM vorgestellt (H2). In Abschnitt 3.3 werden die wesentlichen Unterschiede in der Durchführung der Veranstaltung mit LM gegenüber früheren Veranstaltungen skizziert (H3).

3.1 Einführung der Studierenden in die Programmierung von Lego Mindstorms

Um den Studierenden einen möglichst einfachen Einstieg in die Programmierung der LM zu geben, wurden die erweiterten Lehrinhalte didaktisch in mehrere Schulungsteile aufbereitet. Die begleitenden Schulungsmaterialien, in Form von Powerpointfolien, wurden teilweise durch Beispielprogramme ergänzt. Die gesamte Einführung in die LM wurde in die folgenden drei Schulungsteile unterteilt:

- **Allgemeine Einführung in den LM-Baukasten**:
  Zu Beginn des ersten Schulungsteils haben alle Studierende eine Einführung in die Bestandteile des LM-Baukastens erhalten. Diese Einführung hat die Funktionalitäten des NXT-Bausteins, der Sensoren und der Servomotoren erläutert. Das begleitende Schulungsmaterial umfasst acht Powerpointfolien.
- **Programmierung der LM mit leJOS**:
  Dieser Teil der Schulung gab den Studierenden eine Einführung in die leJOS-API. Anhand von einfachen Programmbeispielen wurden die Ansteuerung der Servomotoren und das Auslesen von Sensordaten erläutert. Das begleitende Schulungsmaterial umfasst 31 Powerpointfolien und ein Beispielprogramm, das sämtliche Sensoren und Servomotoren des LM-Baukastens ansteuert.
- **Wegverfolgungsalgorithmus für LM**:
  Im letzten Teil der Schulung wurde den Studierenden ein einfacher Algorithmus zur Verfolgung von Wegen mit Hilfe des Lichtsensores vermittelt. Dieser Schulungsteil war speziell auf die Aufgabenstellung zugeschnitten, um den Studierenden eine mögliche Lösung für das Wegverfolgungsproblem anzubieten. Das Schulungsmaterial umfasst zehn Powerpointfolien und ein Beispielprogramm, das den Wegverfolgungsalgorithmus umsetzt.

Das vorgestellte Schulungsmaterial wurde hauptsächlich durch die beiden studentischen Hilfskräfte der Veranstaltung in der vorlesungsfreien Zeit erstellt. Insgesamt wurden dafür ca. 80 Arbeitsstunden aufgewandt.

3.2 Aufgabenstellung »TanSEPort«

Die bisherigen Aufgabentypen, die im SEP zum Einsatz kamen, umfassten die Anwendungs- und Spieleentwicklung (vgl. [Lauenroth et al. 2007]). Für beide Arten wurden die Aufgaben so gestellt, dass sie eine hinreichende Komplexität boten, um den gewünschten Lerneffekt bei den Studierenden zu erreichen, ohne sie dabei zu überfordern. Weiterhin wurden Szenarien und Features in den Softwareentwicklungsprozess eingeführt, um die Qualität der einzelnen Entwicklungsaktivitäten zu erhöhen (vgl. [Lauenroth et al. 2007]). Die neue Aufgabenstellung, die den Einsatz von LM vorsieht, sollte eine ähnliche Komplexität aufweisen und der gleichen Methodik folgen. Die gestellte Aufgabe wird im Folgenden erläutert.

Im Sommersemester 2008 bestand die Aufgabe des SEPs in der Durchführung des Projektes »TranSEPort«. Die Aufgabenstellung sah es vor, ein System zu entwickeln, das den Transport von Waren in einer bestimmten vorgegebenen Umgebung erlaubt. Diese Umgebung enthält Städte und Straßen, wobei die Straßen auch in Kreuzungen münden können. Bestandteil der Aufgabenstellung war es, bis zu drei Roboter gleichzeitig durch einen Server zu koordinieren, um Aufträge zum Transport der Waren zwischen den Städten abzuarbeiten. Dabei sollen die Roboter möglichst selbstständig ihren Weg zur Ausführung des ihnen zugeteilten Auftrags finden.

Entsprechend wurde durch die Aufgabenstellung vorgegeben, dass eine Client-Server-Struktur von den Studierenden zu entwickeln war. Der Server musste folgende Aufgaben übernehmen:

- Verwaltung des Warenbestands in den einzelnen Städten
- Empfang neuer Transportaufträge
- Koordinierte Transportabwicklung inkl. Routenplanung und Kollisionsvermeidung

Der Client musste folgende Funktionen umsetzen:

- Eigenständige Funktion zur Straßenerkennung
- Erkennung von Kreuzungen und Städten
- Aufnahme und Ablage von Waren an Städten unter Verwendung der Greifzange (Waren wurden durch die mitgelieferten Bälle repräsentiert)
- Kommunikation zwischen Server und Roboter inkl. Auftragserteilung des Servers an den Roboter und Positions- und Sensordatenüberwachung über Bluetooth
- Auftragserfüllung

3.3 Durchführung der Veranstaltung mit Lego Mindstorms

Der Ablauf des SEPs orientiert sich an den Phasen des in [Lauenroth et. al. 2007] beschriebenen Softwareentwicklungsprozesses (Requirements Engineering, Design, Protoypentwicklung, Implementierung und Test). Im Folgenden werden zusätzliche Lerneffekte aufgelistet, die bei den Studierenden hervorgerufen wurden, indem einige Informationen und Ressourcen nicht sofort zur Verfügung gestellt wurden:

- **Ressourcenknappheit:**
  Die Ressourcenknappheit in Softwareprojekten ließ sich beim Einsatz der LM gut vermitteln, indem beispielsweise die LM-Baukästen erst während der Designphase herausgegeben wurden. Das heißt, die Studierenden mussten sich bereits mit den Anforderungen an das zu entwickelnde System zu einer Zeit auseinandersetzen, zu der sie noch keinen Zugriff auf die einzusetzende Hardware hatten. Dies hat auch dazu geführt, dass die Studierenden erlernt haben, wie sie die Spezifikation eines Hardwaresystems nutzen können, um ausreichend Informationen für die Ableitung von Anforderungen gewinnen zu können. Weiterhin waren die Studierenden somit gezwungen, die Erarbeitung des Designs mit der Einarbeitung in die LM zu koordinieren.
- **Hilfestellung zu Programmierinhalten:**
  Die Bereitstellung der Hilfestellung erfolgte erst während der Erstellung des Prototypens. Dadurch wurde zum einen darauf abgezielt, dass die Studierenden sich während der Erstellung des Designs nicht von der Implementierung leiten lassen. Zum anderen haben die Studierenden erfahren, dass eine systematische und geplante Vorgehensweise wichtig ist, um das doppelte Durchführen von Aktivitäten zu vermeiden und existierende Ergebnisse effizient wiederverwenden zu können.
- **Zeitdruck:**
  Um Zeitdruck bei den Studierenden entstehen zu lassen, wurde die Testphase durch mehrere Faktoren beeinflusst. Zum einen sah die Aufgabenstellung vor, dass das System bis zu drei LM gleichzeitig steuern sollte, so dass den Studierenden ein sehr restriktiver Zugang zu weiteren LM-Robotern gewährt werden musste. Dieser Zugang musste zusätzlich mit den anderen Gruppen, die an dem SEP teilnahmen, koordiniert werden. Dies galt ebenfalls für die Testumgebung. Die Testumgebung wurde erst während der Testphase selbst zur Verfügung gestellt. In der Woche vor Beginn der Tests wurde allerdings eine Karte der Testumgebung in Form einer Grafikdatei zur Verfügung gestellt (nicht maßstabsgetreu), die Auskunft über die Lage der Städte und Straßen gab. Durch diesen restriktiven Zugang zu der Testumgebung, waren die Studierenden gezwungen, ihre Testfälle detailliert zu entwerfen und effizient durchzuführen.

- **Späte Änderung der Testumgebung:**
  Um den Studierenden zu vermitteln, dass späte Änderungen an der Spezifikation der Testumgebung keine Seltenheit sind, wurden gezielt die Farben, die zur Kennzeichnung der Städte und Kreuzungen verwendet wurden, während der laufenden Tests geändert. Somit mussten zum einen die entsprechenden Komponenten der Implementierung kurzfristig angepasst werden und die entsprechenden Tests wiederholt durchgeführt werden. Zum anderen haben die Studierenden durch die ausgeführten Tests bereits vor der Änderung die Notwendigkeit der Anpassung erkannt, da das System mit den vorgegebenen Markierungen nicht realisierbar war.

4 Evaluierung der Verwendung von Lego Mindstorms im SEP

In diesem Abschnitt wird die Evaluierung der Verwendung von LM-Baukästen im SEP beschrieben. Zunächst wird in Abschnitt 4.1 auf die Evaluierung der Veranstaltung durch die Studierenden mittels Fragebögen eingegangen. Im Anschluss daran werden im darauffolgenden Abschnitt die Lerneffekte seitens der Studierenden beschrieben. In Abschnitt 4.3 werden die Erfahrungen der Betreuer in Hinblick auf die beobachteten Lerneffekte beschrieben.

4.1 Veranstaltungsevalierung durch die Studierenden

Die Evaluierung der Veranstaltung durch die Studierenden ist in drei Teile aufgeteilt. Zunächst wird der Aufbau der Evaluierung erläutert, anschließend die Durchführung der Evaluierung. Abgeschlossen wird mit der Auswertung und der Interpretation der Ergebnisse der Evaluierung aus Sicht der Studierenden.

Aufbau der Evaluierung

Um die Einführung der LM-Baukästen zu evaluieren, wurde von den Studierenden die Lehrveranstaltung anhand von Fragebögen am Ende der Veranstaltung bewertet. Um den Erfolg der Veranstaltung zu bewerten, waren die folgenden Fragestellungen von maßgebender Bedeutung:

1. Wurden die definierten Ziele aus Kapitel 1 weiterhin erreicht:
   a) Vertiefung von Programmierkenntnissen
   b) Vermittlung von Soft-Skills, insb. die teamorientierte Arbeitsweise
   c) Vermittlung elementarer, methodischer Vorgehensweisen zur Softwareentwicklung
2. Ist die Verwendung von Studiengebühren zum Erreichen der Ziele angebracht?

Um die beiden genannten Fragestellungen zu beantworten, wurden die folgenden sieben Fragen mit entsprechenden Kontrollfragen in den abschließenden Mei-

nungstest integriert (in den Klammern ist die Fragestellung angegeben, welche der Fragestellungen aus dem vorherigen Absatz evaluiert werden sollte). Sie sollten auf einer Skala (sehr zutreffend, zutreffend, mittel, unzutreffend, sehr untreffend) beantwortet werden:

(1) Die vorgestellten theoretischen Inhalte wurden durch die Entwicklung mit Mindstorms-Roboter verdeutlich (Fragestellung 1.c).
(2) Die Entwicklung der Lejos-Software zeigte mir, dass eine Software ohne die Spezifikation von Anforderungen schwierig zu realisieren ist (Fragestellung 1.c).
(3) Das Arbeiten mit Mindstorms-Robotern hat mir Spaß gemacht.
(4) Das Arbeiten mit Mindstorms-Robotern motivierte mich, meine Programmierkenntnisse zu vertiefen (Fragestellung 1.a).
(5) Die Verwendung von Studiengebühren zur Anschaffung der Mindstorms-Roboter halte ich für sinnvoll (Fragestellung 2).
(6) Nach diesem Projekt hat sich mein Interesse an der Softwareentwicklung gesteigert (Fragestellung 1.a).
(7) Das Arbeiten in einem Softwareentwicklungsteam ist entscheidend für den Erfolg des Projektes (Fragestellung 1.b).

Die Fragestellung (3) dient der Prüfung der Motivation der Studierenden, die durch den Einsatz der LM-Baukästen erreicht wurde.

Durchführung der Evaluierung

Zur Befragung der Studierenden wurden 130 Fragebögen ausgeteilt. Die Studierenden hatten ca. 20 Minuten Zeit, den Fragebogen auszufüllen. Es wurden 109 ausgefüllte Fragebögen zurückgegeben, was einer Rücklaufquote von ca. 77 % entspricht.

Auswertung und Interpretation

Die Ergebnisse der Befragung sind in Abbildung 2 als Balkendiagramme zu finden. Zur folgenden Interpretation der Ergebnisse werden nicht negative Bewertungen als Zustimmung der Studierenden gewertet, d.h., die Kategorien »mittel«, »zutreffend« und »sehr zutreffend« werden als Zustimmung zu den vorgestellten Items gewertet:

(1) 90 Studierende (88 %) äußerten ihre Zustimmung. Wir schließen daraus, dass die von uns vermittelten theoretischen Inhalte trotz der Einführung der LM klar und deutlich vermittelt wurden. Dieses Bild gilt sowohl für die Studierenden des Studiengangs Angewandte Informatik sowie auch der anderen Studiengänge.
(2) 92 Studierende (92 %) äußerten ihre Zustimmung. Wir schließen daraus, dass den Studierenden anhand des entwickelten Spieltriebs für den LM die

Wichtigkeit zur Spezifikation von Anforderungen in Hinblick auf ein methodisches Vorgehen deutlich wurde.

(3) Fast alle Studierenden (97 %) äußerten ihre Zustimmung. Wir schließen daraus, dass die Motivation der Studierenden durch den Einsatz der LM-Baukästen hoch war.

(4) Fast alle Studierende (97 %) äußerten ihre Zustimmung. Wir schließen daraus, dass das Veranstaltungsziel zur Vertiefung der Programmierkenntnisse erreicht wurde.

(5) 99 Studierende (94 %) äußerten ihre Zustimmung. Wir schließen daraus, dass die Studierenden der Investition ihrer Studiengebühren in die LM-Baukästen positiv gegenüberstanden.

(6) 86 Studierende (82 %) äußerten ihre Zustimmung. Wir schließen daraus, dass das Interesse der Studierenden an Softwareentwicklung weiter gestiegen ist.

(7) Fast alle Studierenden (98 %) äußerten ihre Zustimmung. Wir schließen daraus, dass die Studierenden die Notwendigkeit der Arbeit in einem Team und die Gruppendynamik innerhalb des Teams erfahren haben.

Insgesamt lässt sich durch die Evaluierungsergebnisse festhalten, dass aus studentischer Sicht die Einführung der LM-Baukästen einen positiven Einfluss auf die Veranstaltung hatte. Auch die Motivation der Studierenden ist im Vergleich zum Sommersemester 2007 gestiegen, wo 90 % (118 von 131 Studierenden) Zustimmung zur Frage »Mir hat das SEP Spaß gemacht« gaben. Insgesamt wurden die Ziele der Veranstaltung erreicht, und den Studierenden konnte ein Einblick in die Entwicklung von Software für eingebettete Systeme gegeben werden.

4.2 Lerneffekte durch den Einsatz der Lego Mindstorms

Durch die Einführung der LM-Baukästen ergaben sich weitere neue Herausforderungen und zusätzliche Lerneffekte für die Studierenden, die durch die Spiele- und Anwendungsentwicklung nicht erzielt werden konnten. Im Folgenden werden diese Lerneffekte anhand der einzelnen Phasen der Veranstaltung vorgestellt:

- **Requirements Engineering**:
  Durch den Einsatz von Hardware im SEP haben die Studierenden während des Requirements Engineering die Erfahrung gemacht, dass nicht allein die Aufgabenstellung als Anforderungsquelle zu berücksichtigen war, sondern auch die Eigenschaften und Fähigkeiten der zur Verfügung stehenden Hardware (z.B. Sensoren und Servomotoren) und die Umgebung des Systems. Hierzu mussten die Studierenden u.a. in den Szenarien für das geplante System auf die Umgebung und auf die verwendeten Hardwarebestandteile eingehen. Zum Beispiel muss das Szenario »Aufnehmen von Ware« beschreiben,

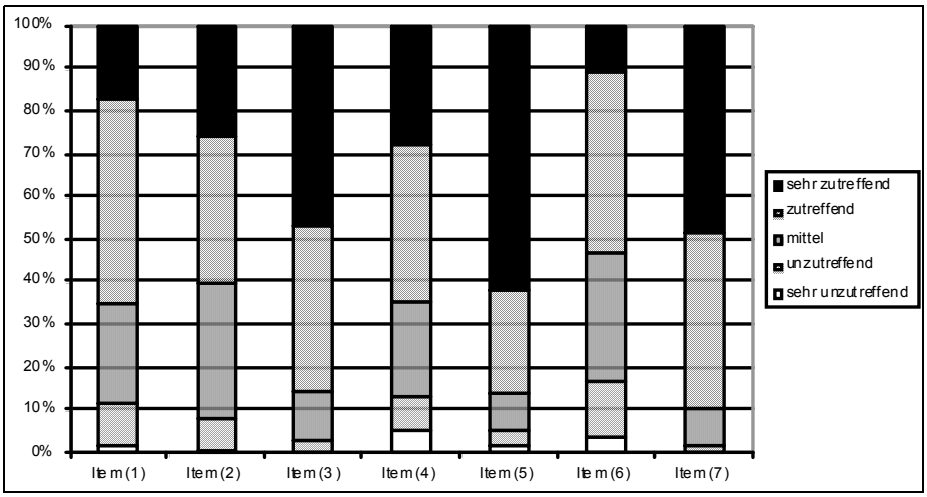


Abb. 2 *Verteilung der Antworten der Studierenden zu den sieben Items*

wie das System mit Hilfe der Sensoren und Motoren zum Ablageplatz der Ware findet und wie die Ware aufgenommen wird.

- **Design:**
  Ziel des Designs mit den LM war es, den Studierenden den Unterschied zwischen Hardware- und Software-Design zu vermitteln. Im Gegensatz zu früheren Aufgaben waren nicht mehr ausschließlich zu kontrollierende Softwarekomponenten sowie deren Schnittstellen zu berücksichtigen, sondern auch zu kontrollierende Hardwarekomponenten. Weiterhin konnten die Studierenden auch erkennen, dass die sorgfältige und frühzeitige Planung des Deployment der einzelnen Komponenten auf der verfügbaren Hardware ein gutes Design ausmacht. Während des Designs wurden unter anderem Message Sequence Charts zur Konkretisierung der Szenarien aus den Anforderungen verwendet, wobei die Studierenden z.B. die Sensoren der Roboter in den MSCs berücksichtigen sollten.
- **Prototyp:**
  Der zu entwickelnde Prototyp zeigte den Studierenden, dass dieser nicht nur die grafische Gestalt des Systems darstellen kann, sondern zum Beispiel auch für eine frühzeitige Erprobung einer rudimentären Kommunikation zwischen Server und Robotern verwendet werden konnte. Hierzu wurde für den Prototypen gefordert, eine manuelle Steuerung für den Roboter zu realisieren. Damit wurde bereits durch den Prototyp sichergestellt, dass sich die Studierenden möglichst frühzeitig mit der Kommunikation zwischen Server und Robotern auseinandersetzten, da dies aus Sicht der Betreuer den schwierigsten Teil der Aufgabenstellung darstellte.

- **Implementierung:**
  Durch die Programmierung mit leJOS waren die Studierenden gezwungen, sich in eine neue Entwicklungsumgebung und die wichtigsten Elemente der zugrunde liegenden API in möglichst kurzer Zeit einzuarbeiten. Um den Studierenden den Einstieg in die Implementierung zu erleichtern, wurden in dieser Phase verschiedene Quellcodebeispiele zur Verfügung gestellt und erläutert, z.B. eine Beispielimplementierung für den Algorithmus zur Straßenverfolgung.
- **Test:**
  Beim Test konnten die Studierenden erfahren, welche Schwierigkeiten eine fremde und nicht frei verfügbare Testumgebung mit sich bringt, da es nicht möglich war, das entwickelte System ausschließlich in einer von ihnen erzeugten Testumgebung ausreichend zu testen. Stattdessen musste auch die Laufzeitumgebung in den Fokus des Tests gerückt werden. Hierzu wurde rechtzeitig zum Testbeginn eine geeignete Testumgebung bereitgestellt.

Ein Video, das die entwickelten Systeme während der Projektabnahme in Aktion zeigt, ist unter *http://sep.icb.uni-due.de* zu finden.

4.3 Erfahrungen

Aus Sicht der Veranstaltungsbetreuung sind die folgenden Lerneffekte im Zusammenhang mit der Einführung der LM eingetreten:

- **Fehlerhafte leJOS-Bibliotheken:**
  Während der Implementierung der Software hat sich herausgestellt, dass die verwendete Version von leJOS und die entsprechenden Java-Bibliotheken teilweise fehlerhaft waren und durch ältere Versionen ersetzt werden mussten. Dieser Umstand stellte einen guten Lerneffekt für die Studierenden dar, da sie auf die Änderung der Entwicklungsumgebung reagieren mussten.
- **Schwierigkeiten mit der Eclipse-Umgebung:**
  Zur Entwicklung von Software mit leJOS werden Plug-ins für Eclipse zur Verfügung gestellt, die Aufgaben wie das Kompilieren und Kopieren der Dateien auf den NXT übernehmen. Die Verwendung von Eclipse wurde im SEP empfohlen. Für einige Studierende im 2. Semester stellte Eclipse durch seinen Funktionsumfang jedoch eine zu große Herausforderung dar und wurde daher nicht von allen Gruppen genutzt.
- **Unerwartete Umgebungseffekte:**
  Während der Nutzung der Testumgebung durch die Studierenden hat sich herausgestellt, dass die Lichtsensoren empfindlich auf wechselnde Lichtverhältnisse innerhalb der Testumgebung reagierten. Für die Studierenden hatte

dies zur Konsequenz, dass z.B. Kalibrierungsroutinen für die Lichtsensoren in das System integriert werden mussten.

- **Seiteneffekte in der Hardware:**
  Die Roboter zeigten unerwartete Seiteneffekte, die erst während des Betriebs auftraten. So wurden z.B. bei schwachen Batterien die Servomotoren mit der Drehungsmessung sehr unzuverlässig.

5 Zusammenfassung und Ausblick

In diesem Beitrag haben wir die Einführung von Lego-Mindstorms-Baukästen in ein Softwarepraktikum beschrieben. Das Ziel dieser Einführung bestand unter anderem darin, das Aufgabenspektrum der Veranstaltung um die Programmierung von Software für eingebettete Systeme zu erweitern. Für die Anschaffung der Baukästen sind Kosten in Höhe von ca. 4.000 Euro angefallen. Für die Erstellung der Schulung und der Schulungsmaterialen wurden ca. 80 Arbeitsstunden aufgewendet. Wir bewerten diese Kosten für die Einführung der Baukästen im Vergleich zu den Personalkosten im Lehrbetrieb einer Hochschule als vergleichsweise gering.

Die Akzeptanz dieser Maßnahme durch die Studierenden wird durch die positiven Evaluierungsergebnisse deutlich. Hervorhebenswert ist, dass über 94 % der Studierenden die Verwendung von Studiengebühren für die Anschaffung der Baukästen als sinnvoll erachtet haben. Aus Sicht der Betreuer konnte durch die Anschaffung der LM-Baukästen Lerneffekte bei den Studierenden erzeugt werden, die typisch für eingebette Systeme sein können (z.B. unerwartete Einflüsse aus der Systemumgebung).

Für die kommenden Veranstaltungen planen wir, den Studierenden die Art der zu bearbeitenden Aufgabenstellung zur Wahl zu stellen, d.h. Programmierung einer Anwendung, Programmierung eines Spiels und Programmierung mit LM-Baukästen. Wir wollen im Rahmen der Anmeldung zur Veranstaltung die Interessen der Studierenden für einen Aufgabentyp abfragen und dann entsprechend der geäußerten Interessen viele Gruppen mit verschiedenen Aufgabentypen anbieten. Den Fokus unserer weiteren Forschungsarbeit werden wir insbesondere darauf richten, ob die Art der Aufgabenstellung einen Einfluss auf den Lerneffekt ausübt.

Danksagung

Wir möchten einen besonderen Dank an unsere studentischen Hilfskräfte Florian Buchloh und Thomas Keller für ihre wertvollen Beiträge zur Gestaltung des SEPs sowie für ihren unermüdlichen Einsatz bei der Betreuung von SEP-Gruppen und bei der Erfassung der Evaluierungsbögen aussprechen.

Literatur

[Bagnall 2007] B. Bagnall: Maximum LEGO NXT: Building Robots with Java Brains, Variant Press, Winipeg, 2007.

[Broy 2006] M. Broy: Challenges in automotive software engineering. In Proceedings of the 28th International Conference on Software Engineering; ICSE '06. ACM, S. 33-42, 2006.

[Georgi und Metin 2007] W. Georgi, E. Metin: Einführung in LabVIEW, Hanser Fachbuchverlag, 2007.

[Holzner 2005] S. Holzner: Eclipse Cookbook, O'Reilly Media, 2005.

[Horstmann und Cornell 2007] C. S. Horstmann, G. Cornell: Core Java 2 Vol.1: Fundamentals, Prentice Hall International, 8. Auflage, 2007.

[Lauenroth et al. 2007] K. Lauenroth, E. Sikora, K. Pohl: Positive Effekte von Szenarien und Features in einem Softwarepraktikum. In: Software Engineering im Unterricht der Hochschulen, SEUH 10, dpunkt.verlag, 2007.

[Liggensmeyer und Rombach 2005] P. Liggesmeyer, D. Rombach (Hrsg.): Software Engineering eingebetteter Systeme – Grundlagen – Methodik – Anwendungen. Elsevier, Heidelberg, 2005.

Threadnocchio – Einsatz von Visualisierungstechniken zum spielerischen Erlernen der parallelen Programmierung mit Java-Threads

Dietrich Boles

Universität Oldenburg
boles@informatik.uni-oldenburg.de

Zusammenfassung

Die parallele Programmierung, d.h. die Entwicklung nebenläufiger Systeme, gewinnt auch außerhalb von Hochleistungsrechenzentren immer mehr an Bedeutung. Insbesondere wird die volle Leistungsstärke moderner Multicore-Rechner nur dann erreicht werden können, wenn die Programme den angebotenen Parallelismus auch nutzen. Allerdings pflegt die parallele Programmierung in heutigen Curricula eher ein Nischendasein. Viele Programmierer sind nicht in der Lage, Software zu entwickeln, die parallel verarbeitet werden kann.

In diesem Artikel wird Threadnocchio vorgestellt, ein Tool zum spielerischen Erlernen der Konzepte der parallelen Programmierung, insbesondere der Programmierung mit Java-Threads. In Threadnocchio werden Threads durch Bilder bzw. Icons visualisiert. Dadurch lässt sich die Ausführung paralleler Programme sehr gut nachvollziehen, und die Auswirkungen von bspw. Kommunikations- und Sychronisationsmechanismen werden dem Programmierer unmittelbar vor Augen geführt.

1 Einleitung

Die parallele Programmierung – genauer die Entwicklung paralleler Anwendungen – führte in den vergangenen Jahrzehnten eher ein Nischendasein. Parallele Anwendungen wurden vor allem für numerische Probleme auf Hochleistungsrechnern entwickelt. Durchaus Nutzen findet der Einsatz paralleler Programmierkonzepte allerdings auch heute schon in gängigen Anwendungen, wie etwa Multimedia-Anwendungen, Simulationen, Webanwendungen oder Computerspielen [Boles 2008].

In bereits naher Zukunft wird die parallele Programmierung jedoch immens an Bedeutung gewinnen müssen, denn auf modernen Multicore-Rechnern mit vielen Prozessoren mit Multithreading-Fähigkeiten (siehe bspw. [Bode 2006] oder [Bengel et al. 2008]) werden zwar auch herkömmliche sequenzielle Programme weiterhin laufen. Die volle Leistungsfähigkeit der Multicore-Technologien wird allerdings nur dann ausgeschöpft werden können, wenn auch Standard-PC-Anwendungen parallel implementiert sind.

Problem ist, dass die Entwicklung korrekter paralleler Programme deutlich komplizierter ist als die Entwicklung sequenzieller Programme. Insbesondere bei der Synchronisation der Prozesse kommt es häufig zu Programmierfehlern, die schwer zu finden sind, weil sie nicht immer, sondern nur gelegentlich auftreten, aber zu fehlerhaften Ergebnissen oder Deadlocks führen können. Weiterhin wird der parallelen Programmierung in den Curricula der Hochschulen bisher zu wenig Platz eingeräumt. David Cearly, Vice President des Analystenhauses Gartner, schätzt daher, »dass weniger als die Hälfte der Programmierer in der Lage ist, Software zu schreiben, die parallel verarbeitet werden kann« [Cearly 2008] und Erhard Plöderer, Universität Stuttgart, konstatiert: »Manche Informatikabsolventen, die ganz gut sequenziell programmieren können, haben von Parallelität keine Ahnung« [Killer 2008].

Notwendig ist es also zum einen, die Forschung im Bereich des Software Engineering für parallele Systeme voranzutreiben, um die Entwicklung paralleler Anwendungen zu vereinfachen. In aktuellen Projekten wird bspw. nach weniger komplexen Programmiermechanismen gesucht, es werden neuartige Debugger für die Entdeckung von Synchronisationsfehlern entwickelt und Pattern für die Strukturierung paralleler Anwendungen definiert (siehe auch [Pankratius et al. 2008]). Walter Tichy, Universität Karlsruhe, sieht in der Parallelisierung »die große neue Herausforderung für die Softwaretechnik« [Tichy 2008].

Zum anderen muss die parallele Programmierung in den Curricula der Hochschulen in Zukunft sehr viel stärkere Berücksichtigung finden. Für die Lehre müssen neuartige didaktische Hilfsmittel konzipiert und entwickelt werden, die das Erlernen der parallelen Programmierung vereinfachen.

Arndt Bode, TU München, fasst diese Forderungen wie folgt zusammen: »In weniger als 10 Jahren werden Standardmikroprozessoren mehr als 128 Prozessoren auf dem Chip aufweisen. Die effiziente Nutzung solcher Hardwarearchitekturen stellt neue Anforderungen an die Programmierung. Programmentwickler werden in Zukunft parallele Programme entwickeln, testen und warten müssen. Neue Programmiersprachen und -modelle sind gefordert, aber auch die Ausbildung für künftige Informatiker muss das Thema Parallelismus stärker berücksichtigen. als das in bisherigen Curricula der Fall war« [Bode 2006].

In diesem Artikel wird mit *Threadnocchio* – auch *Thread-Theater* genannt – ein Tool vorgestellt, das der zweiten der beiden oben angeführten Forderungen gerecht werden soll. Threadnocchio unterstützt durch den Einsatz spielerischer

und visualisierender Elemente das Erlernen der Basiskonzepte der parallelen Programmierung. Es nutzt dabei das Thread-Konzept der Programmiersprache Java. Das Tool, das aus einem grafischen Simulator und einer Klassenbibliothek bzw. API besteht, kann als Open-Source-Software von der Website *http://www.programmierkurs-java.de/threadnocchio* heruntergeladen werden.

Grundlegende Idee von Threadnocchio ist die Visualisierung von Threads durch Bilder bzw. Icons. Schaut man nämlich in gängige Lehrbücher zur parallelen Programmierung werden in den Programmbeispielen die Aktivitäten von Prozessen im Allgemeinen durch Ausgaben auf die Konsole (`System.out.println`) dargestellt. Lösungen bspw. für das bekannte Philosophenproblem dadurch vor Augen geführt zu bekommen, dass auf der Konsole die Ausgabe »Philosoph 1 isst gerade« erscheint, ist für die Lernenden zum einen wenig motivierend und zum anderen durch die Eindimensionalität der Ausgabe auch unübersichtlich. In Threadnocchio dahingegen kann man den Philosophen im Simulator auf einer grafischen, zweidimensionalen Oberfläche beim Essen zuschauen. Und dabei muss sich der Programmierer des Philosophenproblems durch die Nutzung der Threadnocchio-API nur unwesentlich mit der Programmierung der grafischen Ausgabe beschäftigen, sondern kann sich voll und ganz der korrekten Synchronisation der Philosophen widmen.

Dieser Artikel ist so aufgebaut, dass nach dieser motivierenden Einleitung im zweiten Abschnitt zunächst die Grundidee von Threadnocchio sowie der Threadnocchio-Simulator vorgestellt werden. Die Beschreibung der Threadnocchio-API ist Gegenstand von Abschnitt 3. Durch ein konkretes Beispielszenario wird der Einsatz und Nutzen von Threadnocchio in Abschnitt 4 verdeutlicht. Abschnitt 5 beschließt den Artikel mit einer Zusammenfassung und einer Übersicht über weitere Aktivitäten rund um Threadnocchio.

2 Threadnocchio

Threadnocchio führt in die parallele Programmierung anhand des Thread-Konzepts der Programmiersprache Java ein. In Java ist das Thread-Konzept sehr harmonisch in die objektorientierten Konzepte der Sprache integriert worden. Neue Threads werden ausgehend von Objekten einer vordefinierten Klasse `Thread` gestartet, wobei die Objekte zum Kapseln des Zustands des ihnen zugeordneten Threads genutzt werden können. Kommunikation betreiben Threads in Java über global definierte Objekte bzw. Variablen. Zur Absperrung kritischer Abschnitte gibt es die `synchronized`-Anweisung. Zum Warten auf die Erfüllung von Bedingungen durch andere Threads bzw. zum Signalisieren des Erfülltseins von Bedingungen kennt jedes Objekt in Java die Methoden `wait`, `notify` und `notifyAll`. Seit der Version 1.5 stellt Java über diese Basissynchronisationsmechanismen hinaus weitergehende Konstrukte wie explizite Locks, Semaphore und spezielle Collection-Klassen zur Verfügung (siehe auch [Boles 2008] oder [Oechsle 2007]).

2.1 Analogien

Zumindest andeutungsweise lässt sich die objektorientierte Programmierung – nicht nur in Java – mit einem Marionettentheater vergleichen:

- Die Objekte sind die Marionetten.
- Ein objektorientiertes Programm ist ein Marionettentheaterstück.
- Der Programmierer ist der Autor des Marionettentheaterstücks.
- Die Ausführung des Programms entspricht der Aufführung des Stücks.
- Der Marionettenspieler fungiert als Prozessor. Er führt die im Theaterstück gegebenen Handlungssequenzen nacheinander aus.

Bei einem objektorientierten Programm sind die Objekte, was ihre Handlungsfreiheiten angeht, genauso passiv wie die Marionetten im Marionettentheater. Bereits der Programmierer steuert ihr Zusammenspiel. Er gibt vor, welche Aktionen in welcher Reihenfolge ausgeführt werden sollen, auch wenn prinzipiell durch Benutzerinteraktionen die Ausführung in unterschiedliche Richtungen gelenkt werden kann. Der Programmierer hält im wahrsten Sinne des Wortes »alle Fäden in der Hand«.

Im Unterschied dazu können in der Thread-Programmierung in Java die Threads in Bezug auf ihre Koordination als aktive Objekte angesehen werden. Der Programmierer erschafft sie zwar, gibt jedoch bei der Ausführung eines parallelen Programms die Fäden aus der Hand, die Threads agieren nach ihrem Start selbstständig und müssen sich mit anderen Threads koordinieren. Threads sind also vergleichbar mit Pinocchio, der bekannten Kinderbuchfigur des italienischen Autors Carlo Collodi, die – vom Holzschnitzer Geppetto geschaffen – plötzlich lebendig wird und phantastische Abenteuer erlebt. Daher hat das hier vorgestellte Tool auch seinen Namen: *Threadnocchio* ist ein Marionettentheater für Java-Threads als selbstständige Marionetten.

2.2 Threadnocchio-Simulator

Diese Analogien spiegeln sich auch im Aufbau und Erscheinungsbild des Threadnocchio-Simulators wider (siehe auch Abb. 1):

- Ein paralleles objektorientiertes Programm entspricht einem Theaterstück (play).
- Handlungsumfeld für die Threads ist eine Bühne (stage). Dabei kann es in Threadnocchio zu einem Stück durchaus mehrere verschiedene Bühnenaufbauten geben.
- Eine Menge an Threads agiert in einem Threadnocchio-Theaterstück wie selbstständige Marionetten. Sie werden als Akteure (actor) bezeichnet.
- Neben den selbstständigen Threads existieren normale passive Objekte, die als Requisiten (prop) auf der Bühne platziert und von den Marionetten genutzt werden können.

- Die Aufführung (performance) eines Threadnocchio-Theaterstücks, also die Ausführung des Programms, kann in bestimmten Details variieren.

Im linken Bereich des Threadnocchio-Simulators ist es möglich, entsprechende Klassen auf der Grundlage der in Abschnitt 3 vorgestellten Threadnocchio-API zu definieren. In der Lösung des Philosophenproblems aus Abbildung 1 definiert eine von der Threadnocchio-Klasse `Stage` abgeleitete Klasse `Raum` den Aufbau und die Gestaltung der Bühne. Einer Bühne kann ein Bild zugeordnet werden, das als Hintergrund im rechten Teil des Simulators angezeigt wird.

Akteure, also Threads, in dem Beispiel sind die Philosophen. Ihr Verhalten wird in der Klasse `Philosoph` definiert, die von der Threadnocchio-Klasse `Actor` abgeleitet ist. Die Klasse `Actor` selbst ist wiederum (indirekt) von der Java-Klasse `Thread` abgeleitet. Akteuren kann ein Bild oder Icon zugeordnet werden, das bei einer Instanziierung der entsprechenden Actor-Klasse auf der Bühne im rechten Teil des Simulators erscheint (hier die Gesichter).

Als Requisiten, also Objekte ohne eigenen Handlungsstrang, werden in dem Beispiel ein Tisch und mehrere Gabeln benötigt, denen als Objekte von den als Unterklassen der Threadnocchio-Klasse `Prop` definierten Klassen `Tisch` und `Gabel` ebenfalls ein Bild zugeordnet und auf der Bühne platziert werden kann.

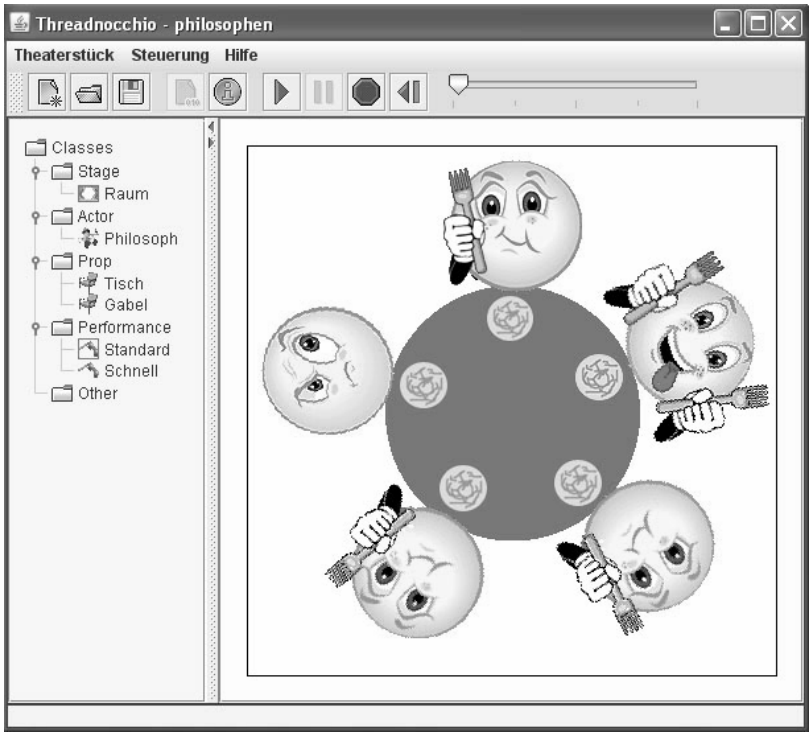


Abb. 1 *Threadnocchio-Simulator mit Lösung des Philosophenproblems*

In dem Beispiel werden weiterhin die zwei Klassen `Standard` und `Schnell` als von der Threadnocchio-Klasse `Performance` abgeleitete Klassen definiert. In Performance-Klassen können Variationen der Ausführung des Programms wie bspw. die Geschwindigkeit festgelegt werden.

Das Vorgehen bei der Entwicklung eines Threadnocchio-Theaterstücks hat folgende Form: Zunächst werden alle benötigten Klassen auf der Grundlage der Threadnocchio-API definiert. Anschließend werden über Popup-Menüs interaktiv Akteure und Properties instanziiert und deren Icons auf der Bühne in der gewünschten Form und an der gewünschten Stelle platziert (alternativ kann das auch fest in einer entsprechenden Stage-Klasse programmiert werden). Danach kann mit den Steuerungsbuttons in der Toolbar des Simulators die Aufführung des Theaterstücks bzw. die Ausführung des Programms gesteuert und verwaltet werden, wobei die im linken Teil selektierte Performance-Klasse die genaue Form der Aufführung definiert (hier die Klasse `Standard`).

3 Threadnocchio-API

Threadnocchio nutzt als Klassenbibliothek bzw. API die sogenannte *Theater-API*, die auch noch in anderen Werkzeugen eingesetzt wird (siehe Abschnitt 5). Die Theater- bzw. Threadnocchio-API umfasst eine Menge an vordefinierten Klassen, die das Grundverhalten aller beteiligten Objekte implementieren und weitere nützliche Features zur Verfügung stellen. Bei der Konzeption der API standen dabei die Grundsätze »so klein wie möglich«, aber »so mächtig wie nötig« und »einfach erweiterbar« im Vordergrund. Ziel der API ist die schnelle und einfache Entwicklung »kleiner« Anwendungen, um einzelne Konzepte der parallelen Programmierung zu visualisieren. Sie ist nicht in erster Linie für die Entwicklung aus Sicht der Softwaretechnik sauber entworfener komplexer paralleler Anwendungen gedacht. Die wichtigsten Klassen der API sind: `Stage`, `Component`, `Actor`, `Prop` und `Performance`.

3.1 Stage

Die Gestaltung einer konkreten Bühne kann durch die Definition einer von der Klasse `Stage` abgeleiteten Klasse erfolgen. Eine Bühne besteht dabei aus einem rechteckigen Gebiet, das sich aus gleichförmigen quadratischen Zellen zusammensetzt. Die Größe der Bühne wird über einen Konstruktor durch die Anzahl an Spalten und Reihen sowie die Größe der Zellen in Pixeln festgelegt. Hierdurch wird ein Koordinatensystem definiert, das zum Platzieren von Akteuren und Requisiten (im Folgenden durch den Begriff *Komponente* zusammengefasst) auf der Bühne dient. Das Koordinatensystem ist nicht endlich, so dass sich Komponenten auch außerhalb der Bühne (»backstage«) befinden können, also (zwischenzeitlich) nicht sichtbar sind.

Neben einer Menge von Getter-Methoden zum Abfragen des Zustands einer Bühne sowie Methoden zur Verwaltung von Maus- und Tastatur-Events (siehe Abschnitt 3.4) lassen sich die Methoden der Klasse `Stage` einteilen in Methoden zur Gestaltung der Bühne und Methoden zur »Kollisionserkennung«.

Zu den Gestaltungsmethoden gehören `add`- und `remove`-Methoden zum Platzieren und Entfernen von Komponenten auf bzw. von der Bühne. Neben der Spalte und Reihe, in die eine Komponente platziert werden soll, kann bei den `add`-Methoden zusätzlich eine z-Koordinate angegeben werden, die eine dritte Dimension auf der eigentlich zweidimensionalen Bühne simuliert. Weiterhin existieren Methoden zum Festlegen eines Hintergrundbildes für die Bühne.

Über die Kollisionserkennungsmethoden lässt sich zur Laufzeit u.a. ermitteln, welche Komponenten sich aktuell in bestimmten Bereichen der Bühne aufhalten oder welche Komponenten (genauer gesagt deren Icons) sich berühren oder überlappen.

3.2 Component, Actor und Prop

Die Klasse `Component` ist von der Java-Klasse `Thread` abgeleitet, erlaubt also die Erzeugung von Threads. Sie definiert Methoden zur Verwaltung von Akteuren und Requisiten, die sie an die von ihr abgeleiteten Klassen `Actor` und `Prop` vererbt. Die Klassen `Actor` und `Prop` unterscheiden sich nur dadurch, dass von `Actor` abgeleitete Klassen die Thread-Methode `run` überschreiben, also Threads definieren können. Bei Unterklassen von `Prop` wird dies dadurch verhindert, dass die `run`-Methode in der Klasse `Prop` als leere `final`-Methode definiert wird.

Die wichtigsten Methoden der Klasse `Component` sind Methoden, um Akteuren und Requisiten ein Icon zuzuordnen und sie auf der Bühne bewegen, also umplatzieren oder bspw. rotieren, zu können. Weiterhin ist eine Menge von Getter-Methoden definiert, um zum einen ihren Zustand abfragen und zum anderen das aktuelle Bühnen- sowie Performance-Objekt ermitteln zu können.

Wie die Klasse `Stage` enthält die Klasse `Component` darüber hinaus Kollisionserkennungsmethoden zum Entdecken von Kollisionen der entsprechenden Komponente mit anderen Komponenten sowie Methoden zur Verwaltung von Maus- und Tastatur-Events.

3.3 Performance

Die Klasse `Performance` definiert insbesondere Methoden zur Steuerung und Verwaltung der Ausführung von Threadnocchio-Programmen: `stop`, `suspend`, `setSpeed`, `started`, `stopped`, `suspended`, `resumed` und `playSound` sowie Getter-Methoden zur Zustandsabfrage.

Möchte ein Programmierer zusätzliche Aktionen implementieren, wenn die entsprechenden Steuerungsbuttons in der Toolbar angeklickt werden, kann er

eine Unterklasse der Klasse `Performance` definieren und hierin die entsprechende Methode überschreiben.

3.4 Maus- und Tastatur-Events

Sowohl die Klasse `Stage` als auch die Klasse `Component` definieren die von der Java-GUI-Programmierung bekannten Methoden zur Verarbeitung von Maus- und Tastatur-Events (`keyPressed`, `keyRealeased`, `mouseClicked`, `mouseEntered`, ...). Soll ein Akteur, eine Requisite oder die Bühne darauf reagieren, wenn während der Programmausführung bspw. eine bestimmte Taste gedrückt oder das Icon des Akteurs mit der Maus angeklickt wird, kann der Programmierer die Methode in der jeweiligen Klasse entsprechend überschreiben. Den Methoden werden Objekte der Theater-Klassen `KeyInfo` bzw. `MouseInfo` übergeben, über die weitergehende Informationen zu dem Event abgefragt werden können.

3.5 Weitere Klassen

`TheaterImage` ist eine Theater-Klasse mit vielfältigen Methoden zum Erzeugen und Manipulieren von Bildern bzw. Icons, die dann Akteuren, Requisiten oder der Bühne zugeordnet werden können. Die Bilder lassen sich dabei auch noch zur

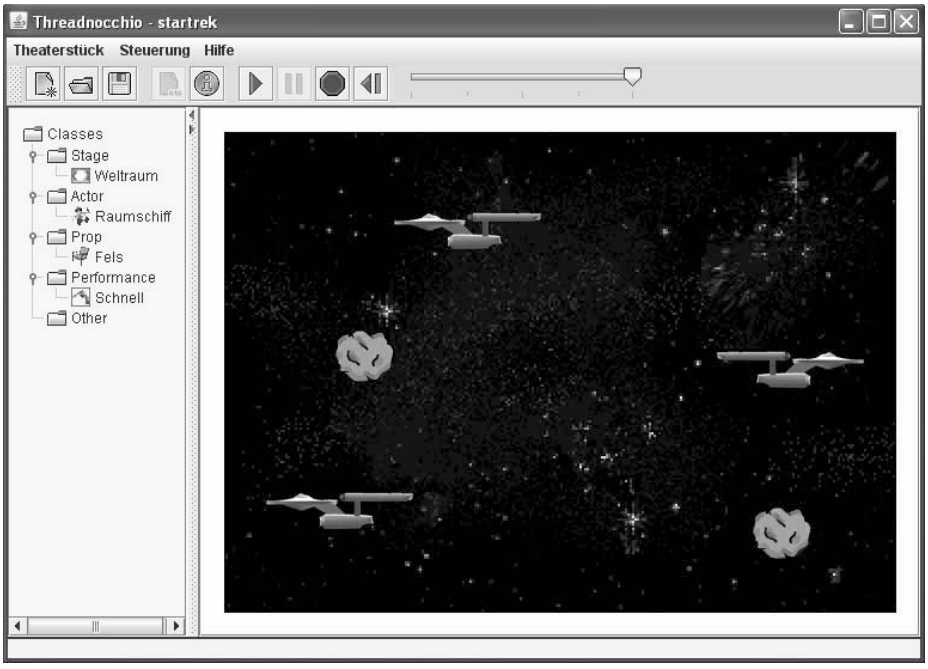


Abb. 2 *Threadnocchio-Simulator mit Startrek-Theaterstück*

Laufzeit verändern, so dass mit Hilfe der Klasse `TheaterImage` bspw. Punktezähler für kleinere Spiele implementiert werden können.

`PixelArea` ist ein Interface, das die Grundlage der Kollisionserkennungsmethoden darstellt. Es definiert hierzu entsprechende Methoden `contains`, `isInside` und `intersects`. Neben einigen zur Verfügung gestellten Standardklassen (`Point`, `Rectangle`, `Cell`, `CellArea`) implementieren auch die Klassen `Stage` und `Component` das Interface. Dadurch sind nur sehr wenige Methoden zur Kollisionserkennung notwendig, die jedoch sehr flexibel und umfassend eingesetzt werden können.

4 Beispiel-Theaterstück Startrek

In dem im Folgenden vorgestellten Theaterstück namens *Startrek* wird der Einsatz und Nutzen von Threadnocchio demonstriert. Threads werden in dem Beispiel durch Raumschiffe visualisiert, die gleichzeitig durch den Weltraum gleiten (siehe Abbildung 2). Treffen die Raumschiffe auf Felsbrocken, werden diese »weggebeamt«, d.h. entfernt. Über Mausklicks auf die Raumschiffe kann der Nutzer die Priorität der entsprechenden Threads verändern und unmittelbar sehen, welche Auswirkungen dies auf die Ausführung eines Programms hat.

Die Bühne im Startrek-Theaterstück wird über die folgende Klasse `Weltraum` realisiert:

```
import java.util.List;
import theater.*;

public class Weltraum extends Stage {

   public Weltraum() {
          // Buehne entspricht Groesse des Bildes, Zellen 1 Pixel
          super(560, 400, 1);
          // Zuordnung des Hintergrundbildes
          setBackground("weltraum.gif");
   }

   public void mousePressed(MouseInfo info) {
          // bei Mausklick auf Buehne wird allen Raumschiffe wieder die
          // Standard-Thread-Prioritaet zugeordnet
          List<Component> schiffe = getComponents(Raumschiff.class);
          for (Component raumschiff : schiffe)
                raumschiff.setPriority(Thread.NORM_PRIORITY);
   }
}
```

Raumschiffe sind Instanzen der Actor-Klasse `Raumschiff`. Wird der Start-Button in der Toolbar des Simulators gedrückt, werden alle Raumschiffe – also Threads –, die der Nutzer vorher über ein spezielles Menü auf der Bühne platziert hat, gestartet und führen ihre `run`-Methode aus. Auch während der Ausführung lassen sich interaktiv weitere Raumschiffe erzeugen. In der Methode `ueberpruefeFels-`

Kollision wird eine Synchronisation vorgenommen, um sicherzustellen, dass die bei der Kollisionsentdeckung ermittelten Felsbrocken noch nicht von anderen Raumschiffen weggebeamt worden sind. Das könnte ansonsten theoretisch passieren, wenn zwei Raumschiffe gleichzeitig auf einen Felsbrocken treffen und der Scheduler nach der Kollisionsentdeckungsanweisung einen Thread-Wechsel anstößt.

```
import java.awt.event.MouseEvent;
import java.util.List;
import theater.*;

public class Raumschiff extends Actor {

   public Raumschiff() {
          // Icon zuordnen
          setImage("raumschiff.gif");
   }

   public void run() {

          // Raumschiff gleitet im Weltraum hin und her
          int step = 1; // Richtung Ost

          while (true) {
                // befindet sich das Icon vollstaendig auf der Buehne?
                if (!isInside(getStage())) {
                      step *= -1; // Richtunsgwechsel
                      getImage().mirrorHorizontally(); // Icon-Spiegelung
                }
                // eine Celle in der aktuellen Richtung weiter platzieren
                setLocation(getColumn() + step, getRow());
                ueberpruefeFelsKollision();
          }
   }

   private void ueberpruefeFelsKollision() {
          synchronized (Fels.class) {
                List<Component> felsen =
                   getStage().getIntersectingComponents(this, Fels.class);
                for (Component fels : felsen) {
                      getStage().remove(fels); // Fels wegbeamen
                }
          }
   }

   public void mousePressed(MouseInfo event) {
          // Veraenderung der Thread-Prioritaet bei Mausklick auf Icon
          event.consume();
          if (event.getButton() == MouseEvent.BUTTON1) {
                setPriority(Math.min(getPriority() + 1,
                                     Thread.MAX_PRIORITY));
          } else {
```

```
                setPriority(Math.max(getPriority() - 1,
                                     Thread.MIN_PRIORITY));
            }
        }
    }
```

Felsbrocken werden als Requisiten, d.h. als passive Objekte der von der Klasse `Prop` abgeleiteten Klasse `Fels`, umgesetzt. Genauso wie Raumschiffe können auch Felsbrocken durch den Nutzer interaktiv im Weltraum, also auf der Bühne, platziert werden.

```
import theater.*;

public class Fels extends Prop {

   public Fels() {
      // Icon zuordnen
      setImage("fels.gif");
   }
}
```

Von der Klasse `Performance` wird die folgende Klasse `Schnell` abgeleitet, die dafür sorgt, dass nach dem Start die maximale Ausführungsgeschwindigkeit eingestellt wird.

```
import theater.*;

public class Schnell extends Performance {

   public void started() {
          // beim Start wird die Geschwindigkeit hochgesetzt
          setSpeed(Performance.MAX_SPEED);
   }
}
```

5 Schlussbemerkungen

Der parallelen Programmierung wird bereits in naher Zukunft bedingt durch die Multicore-Technologien moderner Rechner eine stark wachsende Bedeutung zukommen. Festzustellen ist, dass mit der parallelen Programmierung zusätzliche Programmierprobleme wie die Synchronisation einhergehen, denen viele Programmierer (noch) nicht gewachsen sind. Gefordert werden damit zum einen neuartige Konzepte, die die Komplexität paralleler Programmiermechanismen an sich reduzieren, sowie Werkzeuge, die den Umgang mit den Konzepten der parallelen Programmierung sowie das Erlernen dieser Konzepte vereinfachen.

Letzteres ist das Ziel des in diesem Artikel vorgestellten Tools *Threadnocchio*. In Threadnocchio werden Threads durch Icons visualisiert, so dass Programmierer, die die parallele Programmierung mit Java-Threads erlernen wollen, die Ausführung ihrer Programme sehr gut nachvollziehen können und ihnen die

Auswirkungen der eingesetzten Parallelitätsmechanismen unmittelbar vor Augen geführt werden.

Threadnocchio wurde im Rahmen eines Programmierpraktikums an der Universität Oldenburg konzipiert und anschließend weiterentwickelt. Einige Ideen und Konzepte des Tools *Greenfoot* sind dabei in die Entwicklung eingeflossen. Greenfoot unterstützt das Erlernen der objektorientierten Programmierung, indem Objekte durch Icons repräsentiert werden (siehe *www.greenfoot.org*). Ebenfalls Einfluss hatte das parallele Hamster-Modell [Boles 2008]. Während Java-Threads im Hamster-Modell durch (visuelle) Hamster repräsentiert werden, ist die Visualisierung von Threads in Threadnocchio frei wählbar.

Threadnocchio ist nur eine Instanz einer Reihe von Werkzeugen für die Programmierausbildung, die derzeit an der Universität Oldenburg entwickelt werden. Ziel aller dieser Werkzeuge ist die Visualisierung der Ausführung von Programmen. Sie bedienen sich dabei alle der Theatermetapher und nutzen die Theater-API.

So ermöglicht das Tool *Solist* die Generierung von Simulatoren für sogenannte *Miniwelten* wie Karel, Niki, Hamster, Kara oder Turtle (siehe [Freiberger 2002]). Aufbauend auf der Theater-API muss dabei ein Programmierer lediglich die Eigenschaften der entsprechenden Miniwelt implementieren, was für die genannten Miniwelten innerhalb weniger Stunden möglich ist.

Ein weiteres Werkzeug namens *Kolosseum* erlaubt die Generierung von Simulatoren für sogenannte *Educational Programming Games* wie Robocode, Robocup-Varianten oder CodeRally (siehe [Thevissen 2008]), bei denen Programme entwickelt werden müssen, die Roboter, Fußballspieler oder Rennfahrer implementieren und gegen andere Programme Wettkämpfe austragen.

Letztendlich existiert mit dem Werkzeug *Objekt-Theater* auch eine Alternative zu Greenfoot, bei der die Greenfoot-API durch die Theater-API ersetzt ist.

Literatur

[Bengel et al. 2008] G. Bengel, C. Baun, M. Kunze, K.-U. Stucky: Masterkurs Parallele und Verteilte Systeme: Grundlagen und Programmierung von Multicoreprozessoren, Multiprozessoren, Cluster und Grid, Vieweg+Teubner-Verlag, Wiesbaden, 2008

[Bode 2006] A. Bode: Multicore-Architekturen. In: Informatik-Spektrum, Band 29, Heft 5, Springer-Verlag, Heidelberg, Oktober 2006

[Boles 2008] D. Boles: Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell: Programmierung mit Java-Threads, Vieweg+Teubner-Verlag, Wiesbaden, 2008

[Cearly 2008] D. Cearly: Clients haben zuerst Probleme. In: Computerzeitung, Konradin IT-Verlag, Leinfelden-Echterdingen, 11. August 2008

[Freiberger 2002] U. Freiberger: Karel – Eine Übersicht über verschiedene Entwicklungen, die auf der Idee von »Karel, the Robot« basieren. Luitpold Gymnasium München, 2002. *http://www.schule.bayern.de/karol/data/uebersicht.pdf*

[Killer 2008] A. Killer: Software-Industrie verheddert sich in vielen Programmfäden. In: Computerzeitung, Konradin IT-Verlag, Leinfelden-Echterdingen, 11. August 2008

[Oechsle 2007] R. Oechsle: Parallele und verteilte Anwendungen in Java, Hanser-Verlag, München, 2007

[Pankratius et al. 2008] V. Pankratius, C. Schaefer, A. Jannesari, W. F. Tichy: Software Engineering for Multicore Systems – An Experience Report. In Proceedings of the 1st international Workshop on Multicore Software Engineering (Leipzig, Germany, May 11 – 11, 2008). IWMSE '08. ACM, New York, NY, 53-60, 2008. DOI= *http://doi.acm.org/10.1145/1370082.1370096*

[Thevissen 2008] C. Thevissen: Ein visuelles Framework zur Entwicklung von Educational Programming Games. Diplomarbeit, Universität Oldenburg, Januar 2008.

[Tichy 2008] W. Tichy: Multicore fordert die Informatik. In: Computerzeitung, Konradin IT-Verlag, Leinfelden-Echterdingen, 19. Mai 2008

Werkzeuge zur Qualitätssicherung in der Software-Engineering-Ausbildung

Stephan Kleuker

Fachbereich Design Informatik Medien
Fachhochschule Wiesbaden
kleuker@informatik.fh-wiesbaden.de

Zusammenfassung

Qualitätssicherung wird häufig als notwendiges Übel der vermeintlich kreativeren Softwareentwicklung angesehen. Mit den richtigen Werkzeugen und dem Wissen über passende Verfahren kann die Qualitätssicherung zur spannenden Aufgabe bei der Suche nach potenziellen Fehlern werden. Im hier vorgestellten Forschungsprojekt werden zur Zeit freie Werkzeuge zur Analyse von Java-Programmen für ihren Einsatz in der Software-Engineering-Lehre und der Praxis bewertet.

1 Einleitung

Die Qualitätssicherung (QS) als Teil der Softwareentwicklung spielt für den Projekterfolg eine immer größere Rolle, da unzuverlässige Softwaresysteme von Kunden immer weniger akzeptiert werden. Da allerdings die Mittel für die Qualitätssicherung nicht beliebig erhöht werden können, stellt sich die Frage, welche Ansätze der Qualitätssicherung unter welchen Randbedingungen die größten Erfolgswahrscheinlichkeiten bei der Suche nach gravierenden Fehlern haben.

QS ist damit ein wichtiges Thema in der Software-Engineering-Ausbildung im Studium. Leider verfolgt dieses Thema das Vorurteil der langweiligen, monotonen Arbeit, was durchaus berechtigt ist, wenn man an sich immer wiederholende Tests denkt, die nach jeder Softwareänderung erneut mit genau der gleichen Vorgehensweise manuell durchgeführt werden müssen. Interessant werden die Aufgaben mit neuen Werkzeugen, die das Aufspüren von Fehlern unterschiedlicher Art wesentlich erleichtern und die Testwiederholung automatisieren. Mit

der Idee, die Werkzeuge der »Computer Software Investigation«, kurz CSI, kennenzulernen, besteht die Möglichkeit, das Interesse auf das Thema QS zu leiten.

Allein im Bereich der Java-Programmierung wurde in den letzten Jahren eine große Anzahl von QS-Werkzeugen entwickelt, wobei es praktisch unmöglich ist, sich einen annähernd vollständigen Überblick zu verschaffen. Zentrale Anlaufpunkte sind die folgenden Seiten: [@oss], [@tut] und [@ecp]. Einen guten Überblick über kommerzielle Werkzeuge liefert [@imb]. Neben kurzen Werkzeug-Beschreibungen und Bewertungen einzelner Nutzer findet man wenig bis keine Informationen zu ihren Einsatzbereichen und der Integration mit anderen Werkzeugen. Oft sind die Werkzeuge schwer nutzbar, da die Nutzungsmöglichkeiten vielfältig sind und Einführungen bei Open-Source-Werkzeugen von Entwicklern geschrieben wurden, die nicht die Sicht von Außenstehenden einnehmen können. Es stellt sich damit die Frage, welche Werkzeuge sich für den Einsatz im Bereich QS der Software-Engineering-Ausbildung und der Praxis eignen. Diese Frage wird zunächst eingeschränkt auf ein enges Gebiet, pure Java-Programme und Open-Source-Werkzeuge, im Projekt »Überprüfung von freien QS-Werkzeugen in der Softwareentwicklung auf ihre Anwendbarkeit« (QSQS) untersucht. QSQS findet unter dem Projektdach mit dem Namen »Kombination von Qualitätssicherungsmaßnahmen« (KombiQu) [@kbq] statt. Die Fachhochschule Wiesbaden unterstützt die Forschungsbemühungen mit einer internen Finanzierung. In QSQS werden in Zusammenarbeit mit studentischen Hilfskräften QS-Werkzeuge für Standard-Java-Programme auf ihre Funktionalität, Nutzbarkeit und längerfristige Verwendbarkeit analysiert. Die bisher erreichten Ergebnisse werden kontinuierlich im Internet veröffentlicht.

Im folgenden Text werden zunächst einige Grundlagen der QS wiederholt, dann Bewertungskriterien für Werkzeuge vorgestellt, um darauf aufbauend Werkzeuge für unterschiedliche Aufgabenbereiche für ihre Nutzung in der Ausbildung zu analysieren. Diese Kenntnisse können leicht auf die Nutzbarkeit in Unternehmen übertragen werden, da sich Mitarbeiter häufig individuell im Selbststudium mit QS-Werkzeugen beschäftigen, um ihre Einsatzmöglichkeiten in der Praxis zu evaluieren. Abschließend wird ein erstes Fazit mit einem konkreten Vorschlag für eine Entwicklungsumgebung aus Sicht der QS gegeben.

2 Wesentliche Grundlagen der Qualitätssicherung

Innerhalb der Qualitätssicherung gibt es verschiedene Teilgebiete, die z.B. konstruktive Maßnahmen, wie Guidelines und die Auswahl passender Schulungen, und analytische Maßnahmen, wie das klassische Testen und den Einsatz von Metriken [Hen96], umfassen. Weitere Klassifizierungen der QS-Maßnahmen sind möglich, wie es z.B. in [Lig02] oder [Hof08] gezeigt wird. Im Mittelpunkt von QSQS stehen analytische Maßnahmen für einfache Java-Programme, die in der bisherigen Literatur nur punktuell bezogen auf einen Ansatz, ein Werkzeug

oder eine Java-Technologie beschrieben werden. Die genauer betrachteten analytischen Verfahren können generell wie folgt klassifiziert werden:

- Funktionale Tests:
  Ansätze zur Prüfung, ob ein in Entwicklung befindliches System die geforderte Funktionalität liefert.
  - Äquivalenzklassenanalyse:
    Ansatz zur systematischen Auswahl möglichst weniger Testfälle, die möglichst viele potenzielle Fehler aufdecken können.
  - Überdeckungstests:
    Ansätze zur Prüfung, ob bestimmte Anweisungen, Ablaufpfade oder Boolesche Bedingungen garantiert in Testfällen berücksichtigt werden.
- Prüfung nichtfunktionaler Anforderungen:
  - Performance-Analyse (Lasttest):
    Prüfung, ob ein in Entwicklung befindliches System auch unter bestimmten Lasten die erwartete Reaktionsgeschwindigkeit beibehält.
  - Speichernutzung:
    Prüfung, ob ein in Entwicklung befindliches System auch unter bestimmten Lasten Grenzen für die maximale Speichernutzung beibehält.
  - Codequalitätsanalyse durch Metriken:
    Berechnung von speziellen Kennzahlen einer Software, um Indikatoren für die Qualität des Programmcodes hinsichtlich Wartbarkeit und Änderbarkeit zu erhalten.
  - Usability-Prüfung:
    Analyse, ob ein in Entwicklung befindliches System benutzbar ist.
- Manuelle Prüfverfahren:
  Menschen prüfen (Teil-)Produkte eines in Entwicklung befindlichen Systems nach bestimmten Kriterien, die typischerweise nicht automatisch geprüft werden können, wie z.B. die Lesbarkeit von Dokumenten und die vollständige Umsetzung von Anforderungen.

Betrachtet man die aktuelle Literatur, z.B. [SW02], [HT05] und [Lin05], so konzentrieren sich die Möglichkeiten meist auf den Unit-Test-Ansatz von Kent Beck und Erich Gamma [Bec00] [Bec06], der es erlaubt, Tests in der Programmiersprache des Softwareprojekts zu formulieren. Ausgehend von diesem zentralen Ansatz sind viele weitere Werkzeuge entstanden, die die Ideen von Unit-Tests um weitere Varianten und auf spezielle Anwendungsbereiche erweitern. Die zugehörigen Ansätze und andere frei zugängliche Testwerkzeuge sind in Unternehmen zu wenig verbreitet [@ueb06], da sie oft nicht bekannt sind oder sich die Einarbeitung, die meist durch Mitarbeiter »nebenbei« passieren muss, als zu komplex

herausstellt. Kommerzielle Werkzeuge erschrecken üblicherweise mit ihren enormen Lizenz- und Folgekosten.

Die generellen Ausführungen zu den verschiedenen Testarten in den folgenden Kapiteln sind eng an [Kle08] angelehnt.

3 Bewertungskriterien

Der Ausgangspunkt für die Erstellung der Bewertungskriterien sind die Erfahrungen des Autors mit verschiedenen Vorlesungen zu den Themengebieten Software Engineering, Programmierung und Qualitätssicherung an den Fachhochschulen Wiesbaden und Nordakademie in Elmshorn. In den Praktika zu den Vorlesungen wurden verschiedene QS-Werkzeuge eingesetzt, zu denen im Vorlesungsteil nur die dahinter liegenden Ansätze vorgestellt wurden. Dies führt dazu, dass sich Studierende in den Übungen mit der neuen Methodik und der Nutzung des Werkzeugs im engen studentischen Zeitrahmen auseinander setzen müssen. Um hier erfolgreich mit dem Werkzeug arbeiten zu können, gibt es zwei zentrale Qualitätskriterien für die QS-Werkzeuge: Sie müssen intuitiv bedienbar und ihre Ergebnisse auf den ersten Blick interpretierbar sein. Eine intuitive Bedienbarkeit liegt z.B. vor, wenn man in einer Entwicklungsumgebung maximal eine neue Einstellung und dann in einem Kontextmenü zum Starten eine neue Auswahl vornehmen muss. Ergebnisse sind schnell interpretierbar, wenn sie visuell durch Farben oder erzeugte Tabellen angezeigt werden.

Markantes Beispiel für ein Werkzeug, das diese Anforderungen nicht erfüllt, ist Jester (JUnit Testtester, *http://jester.sourceforge.net*), ein Werkzeug, mit dem man zu testende Programme mutieren kann; genauer werden Bedingungen in if- und while-Befehlen durch false und true ersetzt sowie Zahlen verändert. Führt man dann seine Tests mit dem mutierten Programm durch, sollten die Tests bei genügender Präzision, z.B. detaillierter Äquivalenzklassenanalyse, diese Mutationen finden und Fehler ausgeben. Da das Werkzeug nicht ganz einfach installierbar ist, man außerdem aufpassen muss, dass nicht der ursprüngliche Programmquellcode verändert wird und als kritisch ausgegebene Mutationen von Hand analysiert werden müssen, kann es nicht »nebenbei« im Praktikum genutzt werden.

Basierend auf diesen Lehrerfahrungen stellt sich die Frage nach einer Werkzeugsammlung, die Studierende in Praktika einfach nutzen können und mit denen wesentliche Aspekte der QS behandelt werden. Zur Beantwortung müssen aktuelle Werkzeuge analysiert werden. Da die derzeitige Prüfungsordnung keine Integration dieser Aufgabenstellung in eine Projektform im Studienplan ermöglichte, findet die Arbeit in einem kleinen Forschungsprojekt statt. Die Untersuchungen werden dabei von Bachelor-Studierenden durchgeführt, die als generelle Aufgabe ein Werkzeug nach den genannten Kriterien untersuchen. Konkretes Ziel ist es dabei, möglichst schnell ein eigenes Beispiel zu konstruieren, mit dem man die wesentliche Funktionalität des untersuchten Werkzeugs vorstellen kann.

Parallel zu den Lehrerfahrungen wird auch der Bedarf in Unternehmen mehr und mehr erkannt, QS-Werkzeuge in die eigenen Entwicklungsprozesse zu integrieren. Dies wird durch verschiedene Abschlussarbeiten deutlich, in denen sich Studierende mit QS-Werkzeugen für konkrete Teilaufgaben beschäftigen. Aus diesem Grund wurde die Werkzeugbewertung um Kriterien erweitert, die für Unternehmen, aber auch für Studierende in studentischen Projektgruppen, in denen das Thema QS intensiver betrachtet wird, relevant sind.

Folgende Dokumentationsschablone wurde ab dem Punkt »Einsatzgebiete« zur Bewertung der Werkzeuge genutzt. Dabei wurde neben der Funktionalität besonders auf die Installierbarkeit, Nutzbarkeit und die Support-Seiten geachtet. Die Kriterien können für bestimmte Anwendungsszenarien gewichtet werden. Die letzte Spalte gibt mit einem Kreuz an, ob das Kriterium für die Grundlagenausbildung besonders relevant ist.

Kriterium	Beschreibung	LV
Name	Name des Werkzeugs	
Homepage	Ausgangsseite zur Homepage des Werkzeugs	
Lizenz	Unter welcher Lizenz ist das Werkzeug nutzbar?	
Untersuchte Version	Genaue Angabe der Versionsnummer und der genutzten Dateien	
Zeitpunkt	Wann wurde die Analyse zuletzt aktualisiert?	
Kurzbeschreibung	Beschreibung in ein bis vier Sätzen, welche Hauptaufgabe(n) das Werkzeug lösen soll	
Fazit	Ist das Werkzeug für Einsteiger in die Qualitätssicherung nutzbar, lohnt sich der Einstieg in die Einarbeitung für erfahrene Nutzer, gibt es besondere Highlights/Lowlights bei der Nutzung, Dokumentation oder Installation? Wie gut ist das Werkzeug im Vergleich zu Werkzeugen mit vergleichbarer Funktionalität?	
Einsatzgebiete	Wo kann das Werkzeug eingesetzt werden? Gibt es Bereiche, in denen das Werkzeug nicht nutzbar ist?	X
Einsatzumgebungen	Kann das Werkzeug ohne andere Werkzeuge benutzt, kann es (zusätzlich) als Plug-in in Eclipse oder Netbeans installiert werden?	X
Installation	Wie wird installiert bzw. wo (Link) steht eine Installationsanleitung? Kurze Beurteilung, ob Installation einfach für Programmierer durchführbar.	X
Dokumentation	Macht die Dokumentation für einen Entwickler mit Programmier-, aber wenig bis keiner QS-Erfahrung einen lesbaren Eindruck, gibt es einen Schnelleinstieg, gibt es die Dokumentation in deutsch/englisch, ist sie leicht herunterladbar (z. B. PDF oder/und Teil des Downloads), gibt es besondere Highlights (z. B. Videos)?	X
Wartung der Projektseite	Machen die Seiten des Werkzeugs den Eindruck, dass sie häufiger gepflegt werden, gibt es eine kontinuierliche Versionsgeschichte, erkennt man, dass neue Features (z.B. Generics und Annotationen in Java) die Entwicklung beeinflussen?	→

Kriterium	Beschreibung	LV
Nutzergruppen und Support	Gibt es eine detaillierte Fragen- und Antwortseite, gibt es Foren und Nutzergruppen zur Diskussion über das Werkzeug? Gibt es eine E-Mail-Adresse für Anfragen und Kommentare?	
Intuitive Nutzbarkeit	Kann sich ein erfahrener Entwickler anhand der Dokumentation leicht einarbeiten, kann er schnell zu eigenen Beispielen kommen? Werden besondere Fähigkeiten bzw. Kenntnisse gefordert? Gab es besondere Beobachtungen bei der Einarbeitung?	X
Automatisierung	Kann das Werkzeug ohne Nutzer nach dem Start laufen, werden die gewonnenen Ergebnisse lesbar aufbereitet (z. B. HTML-Export), kann der Nutzungsprozess einfach automatisiert werden (z. B. mit Shell-Skripten oder als Ant-Task)?	

Tab. 1 *Dokumentationsschablone*

4 Funktionstest

Die Funktionalität eines Systems lässt sich sehr gut mit Unit-Tests prüfen. Dabei kann man den Code sehr genau analysieren (White-Box-Tests) oder Zusammenhänge zwischen Komponenten (Gray-Box-Test) oder ganze Systeme (Black-Box-Test). Als Standardwerkzeug kann hier JUnit (*http://www.junit.org/*) angesehen werden. Mit JUnit ist es möglich, Tests direkt in Java zu schreiben, Studierende lernen dabei, warum es wichtig ist, für jeden Test die gleiche Umgebung zu schaffen und dass Tests automatisierbar sein können.

Nach einer durchaus aufwendigeren Einarbeitung, in der das Grundverständnis für die Vorgehensweise dem Entwickler verständlich werden muss, ist es sehr leicht, systematisch Tests zu schreiben. JUnit ist in alle wichtigen Entwicklungsumgebungen integriert, kann aber auch sehr gut ohne Umgebung und integriert in eine Automatisierungsumgebung ablaufen. Die systematische Testerstellung mit JUnit ist damit immer der Einstiegspunkt in das systematische Testen.

JUnit existiert zurzeit in den Versionen 3.8 und 4.x, wobei 3.8 »klassisches« Java ohne Annotationen nutzt. JUnit 4.x ist etwas flexibler durch die Verwendung von Annotationen, was die Nutzbarkeit aber vergleichbar lässt. Welche Variante man nutzt, ist fast egal, hängt davon ab, was man noch weiter im Testumfeld machen will. Der Übergang von JUnit 3.8 zu Unit-Tests anderer Programmiersprachen wie C++ gestaltet sich etwas einfacher, weiterhin gibt es weitere QS-Werkzeuge, die JUnit-Tests zur Testwiederholung generieren, wobei diese Tests teilweise nur unter JUnit 3.8 laufen.

Eine Alternative zu JUnit stellt TestNG (*http://www.testng.org/*) dar, das genau den gleichen Prinzipien von JUnit folgt. JUnit-Tests können leicht in TestNG-Tests verwandelt werden. TestNG bietet etwas mehr Unterstützung bei der systematischen Gruppierung von Tests in Teilgruppen und bei der Ausführung gleichartiger Tests mit unterschiedlichen Parametern, die auch aus Dateien

eingelesen werden können. Die Dokumentation von TestNG ist noch verbesserungswürdig, gerade in den Bereichen, in denen mehr Funktionalität als JUnit geboten wird.

Einen vollständig anderen Ansatz verfolgt das Werkzeug FIT (*http://fit.c2.com/*), dass davon ausgeht, dass Tests von Leuten geschrieben werden, die diese nicht ausprogrammieren wollen oder können. FIT geht einen eher unkonventionellen Weg, indem die Tests nicht in der jeweiligen Programmiersprache geschrieben werden, sondern von Entwicklern Methoden zur Verfügung gestellt werden, die FIT aufruft. Die Aufrufparameter stehen dann in einer vom Tester geschriebenen HTML-Tabelle. Das Ergebnis wird ebenfalls in eine HTML-Datei geschrieben. Nach kurzer Einarbeitung ist auch dieser Ansatz sehr intuitiv nutzbar. Wichtig ist die Gestaltung der Testschnittstelle, da sie wesentlich beeinflusst, was überhaupt testbar wird.

Als Fazit für eine minimale Umgebung wird JUnit 3.8 vorgeschlagen, wobei alle anderen genannten Werkzeuge für eine vertiefte Betrachtung des Themengebiets sehr interessant sind.

5 Überdeckungstest

Zu jedem Programm kann man einen gerichteten Kontrollflussgraphen angeben, der die möglichen Programmausführungen beschreibt. Dabei sind die Knoten des Graphen ausgeführte Befehle, die durch gerichtete Kanten verbunden werden. If-Befehle, Switch-Befehle und Schleifen führen dabei zu Verzweigungen im Graphen, der Graph kann durch Normalisierung unabhängig von der Programmformatierung gemacht werden. Abbildung 1 zeigt eine Methode und den zugeordneten Graphen.

```
public int aha(int ein){
  int erg=-ein/2;          // 0
  int i=ein;               // 1
  while(i>0){              // 2
    erg=erg+(i--);         // 3
  }
  if (ein <0 || ein%2==1){// 4
    erg=0;                 // 5
  }
  return erg*2;            // 6
}
```

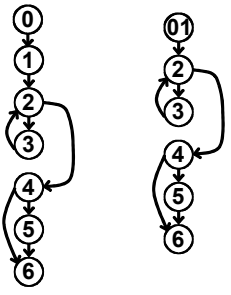


Abb. 1 *Methode mit Kontrollflussgraph und normalisierter Variante [Kle08]*

Der Ansatz bei den vorgestellten Überdeckungsmaßen ist, dass man fordert, dass mit einer Menge von Testfällen das jeweilige Maß möglichst groß, genauer möglichst der Wert Eins, ist. Für die sogenannte C0-Überdeckung wird gezählt, wie

viele Knoten von den Tests im Verhältnis zur Anzahl aller Knoten besucht wurden. Bei der C1-Überdeckung, auch Zweig- oder Kantenüberdeckung genannt, wird gefordert, dass alle Kanten überdeckt werden. In verwandten Maßen werden nur die Boole'schen Bedingungen der Programme betrachtet und gefordert, dass alle Bedingungen oder gar alle Teilbedingungen mit Tests einmal nach wahr und einmal nach falsch ausgewertet werden.

Generell kann die Aufforderung an Studierende, ihre Programme zu testen, dadurch konkretisiert werden, dass die Erreichung bestimmter Überdeckungsmaße gefordert wird. Im Rahmen von QSQS wurden folgende Werkzeuge untersucht.

Coverlipse (*http://coverlipse.sourceforge.net/index.php*) ist einfach als Plug-in in Eclipse installierbar und erlaubt die Prüfung auf die Anweisungsüberdeckung. Das Werkzeug prüft die Überdeckung bezüglich vorher zu entwickelnder JUnit-Tests. Die Nutzung ist recht einfach, statt die Tests als JUnit-Tests zu starten, existiert eine Startmöglichkeit mit dem Menü-Punkt »JUnit w/Coverlipse«. Nach der Programmausführung werden die ausgeführten Zeilen markiert und das Überdeckungsmaß angezeigt. Sehr interessant ist eine zweite Testart, mit der geprüft wird, ob einzelne Zuweisungen Auswirkungen haben, also der Wert der zugewiesenen Variable später noch mal genutzt wird. Aktuell scheint es keine Weiterentwicklung zu geben.

EclEmma (*http://www.eclemma.org/*) ist einfach als Plug-in in Eclipse installierbar und erlaubt die Prüfung auf eine Anweisungsüberdeckung. Die Nutzung ist optimal vereinfacht, statt das Programm über den Eclipse-Startknopf laufen zu lassen, existiert ein neuer Knopf, um das Programm mit Coverlipse zu starten. Natürlich sind auch JUnit-Tests startbar. Nach der Programmausführung werden die ausgeführten Zeilen und das Überdeckungsmaß angezeigt. Leider kann man selten eine 100 %-Überdeckung erreichen, da das Werkzeug Schwierigkeiten mit Zeilen hat, die z.B. Klassen-Deklarationen enthalten. Eine Testautomatisierung außerhalb von Eclipse ist nur über das verwandte Werkzeug Emma möglich.

CodeCover (*http://www.codecover.org/*) wurde an der Uni Stuttgart entwickelt und kann allein oder als Eclipse-Plug-in genutzt werden. Nachdem man die etwas aufwendige Projekteinrichtung verstanden hat, CodeCover muss aktiviert und dann für die zu analysierenden Klassen eingeschaltet werden, ist es leicht nutzbar. CodeCover ist sehr interessant, da es neben der reinen Anweisungsüberdeckung die Möglichkeit bietet, für Bedingungen zu analysieren, ob sie nach wahr und falsch ausgewertet werden (verwandt zur Zweigüberdeckung, die allerdings nicht berechnet wird). Weiterhin besteht die Möglichkeit, für Boole'sche Ausdrücke genau die Werte der einzelnen Ausdrücke zu analysieren, wobei eine Form der Bedingungsüberdeckung, die Modifizierte Bedingungs-/Entscheidungsüberdeckung (MC/DC) geprüft wird. Unschönes Detail ist, dass return-Anweisungen nicht als eigene Programmzeilen aufgefasst werden. Durch die Auswertungsvarianten dauert es, bis Studierende die Möglichkeiten des Werkzeugs

begriffen haben, danach bleibt es ein wichtiger Baustein auch bei weiteren Softwareprojekten.

Als Fazit lässt sich feststellen, dass CodeCover sehr gut für die Lehre geeignet ist, da es nach kleiner Einarbeitungshürde intuitiv nutzbar ist und verschiedene Maße liefert, evtl. findet in weiteren Releases eine Orientierung an klassischen C0-, C1-, C2-Maßen statt. Für vertiefende Betrachtungen ist auch Coverlipse interessant, da es zusätzlich die Möglichkeit bietet, die Abhängigkeit von Programmfragmenten zu nutzen.

6 Oberflächengesteuerte Tests

Generell sind Oberflächen normale Softwarekomponenten und können deshalb in der Entwicklung und Nutzung genau wie andere Software auch getestet werden. Dabei wird das Design getrennt betrachtet. Insbesondere die Äquivalenzklassenmethode eignet sich zur Erstellung von Testklassen.

Ein eher technisches Problem ist, dass die Ausführung der Tests meist nicht mit den Standardtestwerkzeugen ausgeführt werden kann. Aus diesem Grund werden hierzu häufig sogenannte Capture-and-Replay-Werkzeuge genutzt. Die zentrale Idee ist dabei, dass die Nutzung der Oberfläche, also z. B. das Eintragen von Werten und das Ziehen sowie Klicken der Maus, aufgezeichnet wird. In das Protokoll dieser Aufzeichnung kann man die geforderten Eigenschaften, z.B. Ausgaben, angezeigte Texte und Anzeigefarben, eintragen. Teilweise besteht auch die Möglichkeit, dass Teile des Bildschirms als Bild mit vorher gemachten Bildern, also Screenshots, in Teilen oder als Ganzes verglichen werden.

Die so aufgezeichneten Testfälle können dann immer wieder, bei jedem neuen Software-Release, möglichst automatisch durchgeführt werden. Testwerkzeuge mit Replay-Möglichkeit führen dann die aufgezeichneten Mausbewegungen aus. Weiterhin ist eine spätere Anpassung der Testfälle möglich, so dass z.B. geprüft werden kann, ob in einem Ausgabefeld ein erwarteter Wert steht.

Abbot/Costello (*http://abbot.sourceforge.net/*) erlaubt die Aufzeichnung von Mausbewegungen und Klicks auf Standard-Java-Oberflächen, die AWT und Swing nutzen. Die Aufzeichnungen können erneut abgespielt werden, weiterhin kann man die Prüfung von Eigenschaften der GUI-Elemente einbauen. Gerade beim ersten Kontakt ist die Handhabung recht trickreich, und Studierende können so leicht den Spaß am Experimentieren verlieren. Ist die erste Aufzeichnung gelungen, lehrt das Werkzeug, dass man Oberflächen sauber strukturieren und allen Elementen lesbare Namen zuordnen muss, damit man in einer hierarchisch strukturierten Oberfläche das Element identifizieren kann, dessen Eigenschaft man prüfen möchte. Die Handhabung von Testskripten ist nicht immer intuitiv, kleine Fehler werden nicht verziehen, so dass dieses generell interessante Werkzeug für den Einstieg nicht geeignet ist.

UISpec4J (*http://www.uispec4j.org/*) basiert auf JUnit und hat als Konzept, dass die GUI-Klassen aus Java durch eigene GUI-Klassen ersetzt werden, die die gleiche Funktionalität nach außen haben, nach innen aber mehr Testmöglichkeiten bieten. Dieser interessante Ansatz hat als Nachteil, dass es lange dauert, bis Java-Aktualisierungen in UISpec4J-Klassen übernommen werden. Weiterhin ist der Ansatz für ineinandergeschachtelte GUIs sehr aufwendig, da man immer im Detail alle GUI-Komponenten bei der Testerstellung kennen muss, wodurch leider der Vorteil der schnellen Testerstellung wieder verloren geht. Da gewisse Funktionalität unter Linux nicht lauffähig ist und man in der Wahl der Java-Version eingeschränkt wird, ist das Werkzeug zum Einsatz in der Lehre nicht zu empfehlen.

FEST (*http://fest.easytesting.org/wiki/pmwiki.php*) legt sogenannte Fixtures an, die eine Schnittstelle zu den nativen Java-Swing-Komponenten zur Verfügung stellen, um diese entsprechend zu testen. Als Grundlage werden JUnit oder TestNG genutzt. FEST ist modular aufgebaut, kann einfach als Jar-Datei in einem Projekt genutzt werden und unterstützt fast alle GUI-Komponenten beim Capture-and-Replay. Mit Hilfe der Dokumentation ist die hier nicht sehr aufwendige Einarbeitung möglich, wieder sollte das zu untersuchende GUI gut strukturiert und dokumentiert sein. Mit etwas Einarbeitungszeit ist das Werkzeug durch Studierende gut nutzbar.

Jacareto (*http://jacareto.sourceforge.net/*) ist ebenfalls ein Werkzeug, mit dem Klicks- und Mausbewegungen aufgezeichnet und abgespielt werden können. Da Jacareto GUI-Komponenten erkennt, können auch auf der GUI ausgegebene Ergebnisse mit einem erwarteten Ergebnis verglichen werden. Es fehlt allerdings die Möglichkeit, komplexere Prüfungen einfach zusammenzufassen. Weitere Tests haben die Nutzbarkeit für kleine Beispiele gezeigt, allerdings traten bei etwas komplexeren Anwendungen einige Fehler auf, weiterhin ist die Nutzung wenig intuitiv. Die Integrationsmöglichkeit in komplexere Testumgebungen ist praktisch nicht gegeben, die Nutzung unter Linux machte Probleme. Die Versionsnummer 0.7.12 trägt die betrachtete Version als Vorstufe zu einem einsetzbaren Werkzeug zu Recht.

Bei *Marathon* (*http://marathontesting.com/Home.html*) muss der Tester ebenfalls keinen Programmcode schreiben, Testfälle werden wieder aufgezeichnet und können mit Prüfungen auch während der Ausführungen annotiert werden. Das Werkzeug basiert auf JUnit 3.8. Die Entwicklungsumgebung von Marathon ist sehr strukturiert. Kennt man sich mit Eclipse bereits aus, findet man sich auch in dieser Umgebung schnell zurecht. Marathon kann unabhängig von Entwicklungsumgebungen genutzt werden und ist auch für unerfahrene Nutzer nach kurzem Studium der guten Dokumentation einsetzbar.

Die untersuchten Werkzeuge benötigen eine recht große Einarbeitungszeit oder müssen von Experten in Übungen sehr detailliert erklärt werden, da es hier leicht zu Misserfolgen kommen kann, wenn z.B. eine bei der ersten Nutzung

nicht intuitive und deshalb vergessene Einstellung die Werkzeugnutzbarkeit beeinflusst. Letztendlich können FEST, Marathon und auch Abbot für eine genauere Betrachtung von GUI-Testern genutzt werden.

In studentischen Softwareprojekten wurde deutlich, dass GUI-Tester nur sehr vereinzelt genutzt werden können, da sich die grafische Oberfläche nicht oder nur marginal ändern darf, damit vorher aufgezeichnete Tests wiederverwendet werden können. Dies ist bei kurzen studentischen Projekten schwer erreichbar.

Betrachtet man »außer Konkurrenz« GUI-Tester, die nicht für Standard-Java-Oberflächen geschrieben wurden, so sticht für Webapplikationen das Werkzeug *Selenium* (*http://selenium.openqa.org*) heraus, mit dem man nach relativ kurzer Einarbeitungszeit sehr intuitive Tests für Weboberflächen erstellen und wiederholen kann [KG08].

7 Mocking

Häufig tritt bei der parallelen Entwicklung durch mehrere Personen der Fall auf, dass man zum Test seiner eigenen Klassen die Klassen anderer Leute benötigt, die diese aber noch nicht implementiert haben. Statt zu warten, gibt es dann den Ansatz, sich diese Klassen in minimaler Form selbst zu schreiben. Dieser Ansatz wird in der Literatur die Erstellung sogenannter Mocks (teilweise auch Stubs) genannt. Man sucht dabei nach einer möglichst einfachen Implementierung, so dass man seine eigenen Tests sinnvoll ausführen kann.

Die einfachst mögliche Implementierung besteht darin, immer einen Default-Wert zurückzugeben. Für den Rückgabetypen void kann man eine vollständig leere Implementierung nutzen. Soll ein echtes Objekt zurückgegeben werden, reicht es für eine lauffähige Implementierung aus, dass null zurückgegeben wird. Bei Rückgabetypen wie boolean oder int muss man sich auf Standardwerte wie false und 0 einigen.

Mit der einfachst möglichen Implementierung kann man dann feststellen, ob die eigene Klasse implementierbar ist. Dies reicht aber häufig nicht aus, um die eigene Klasse vollständig zu testen, wenn man z.B. daran interessiert ist, ein bestimmtes Überdeckungsmaß zu erfüllen.

Generell ist es eine sehr gute Übungsaufgabe, Mocks zur Erreichung eines bestimmten Überdeckungsgrades von Hand zu schreiben, da dies eine intensive Auseinandersetzung mit dem Programmcode fördert. Danach stellt sich aber die Frage, welche Automatisierungsmöglichkeiten es gibt. Insgesamt wurden hierzu die folgenden Werkzeuge betrachtet.

JMock (*http://www.jmock.org/*) bietet eine einfache Möglichkeit, nicht vorhandene Klassen durch sogenannte Mock-Objekte zu testen. Es ist eine Klassenbibliothek, die zusammen mit JUnit nutzbar ist. Über Reflection ist es möglich, für eine fehlende Klasse, deren Interface bekannt sein muss, passende Objekte anzulegen. Neben der Objekterzeugung kann das weitere Verhalten eines Objek-

tes nach außen festgelegt werden, so dass echte Überdeckungstests der zu prüfenden Klasse möglich sind. Mit etwas Einarbeitungszeit ist die Nutzung auch für Anfänger intuitiv möglich.

EasyMock (*http://www.easymock.org/*) ist von seiner Arbeitsweise fast identisch wie JMock, stellt aber eine eigene Entwicklung dar. Wieder können Erwartungen an das Verhalten des entstehenden Mock-Objekts recht einfach festgelegt werden.

MockLib (*http://mocklib.sourceforge.net/mocklib3/index.html*) funktioniert im Prinzip wie die beiden anderen Mock-Werkzeuge, ist dabei aber komplexer zu bedienen. Zahlreiche kleine und nur unzureichend dokumentierte Fallstricke machen den Einstieg zäher und langwieriger, als er sein müsste. Die gebotene Funktionalität ist dagegen gut, trotzdem sind die anderen Werkzeuge MockLib vorzuziehen.

Zusammenfassend kann festgestellt werden, dass JMock und EasyMock generell gut nutzbar sind, man aber auf die Nutzung expliziter Mock-Werkzeuge in der Grundausbildung verzichten kann, da sie eine nicht triviale Einarbeitung benötigen und der Erkenntnisgewinn nicht sehr hoch ist, wenn Studierende gelernt haben, Mock-Objekte selbst zu schreiben. In vertiefenden Veranstaltungen z.B. zu verteilten Systemen, sollte die Mocking-Idee wieder aufgegriffen werden, da man dann ganze Komponenten, wie Server oder Datenbankverbindungen, durch Mocks ersetzen kann.

8 Weitere QS-Werkzeuge

Neben den testorientierten QS-Werkzeugen gibt es noch andere Ansätze, die die Qualität erhöhen können. Ein Beispiel sind Metriken, die auf dem Programmcode basierend berechnet werden und so gewisse Indizien für die Codequalität liefern. Typischer Vertreter einer Softwaremetrik ist die McCabe-Zahl, mit der die Anzahl der Verzweigungen einer Methode gemessen wird. Dieses Maß ist ein Indikator für die Komplexität und den Testaufwand, wobei allerdings implizite Verzweigungen, die z.B. durch Polymorphie möglich sind, nicht betrachtet werden. Ein Indikator für gute Objektorientierung heißt »Lack of Cohesion in Methods« (LCOM*) und prüft, ob Exemplarvariablen in möglichst vielen verschiedenen Methoden genutzt werden. Recht viele Maße sind intuitiv, und die Aussagekraft sowie Probleme von Metriken generell können so sehr gut mit Studierenden erarbeitet werden. Als Werkzeug kann hier *Metrics* (*http://metrics.sourceforge.net/*) als Plug-in für Eclipse empfohlen werden, da es einige Maße anbietet und sehr einfach bedienbar ist. Das Werkzeug wurde zwar seit Eclipse 3.1 nicht weiterentwickelt, ist aber auch in Eclipse 3.4 nutzbar. Als hilfreich stellen sich auch Werkzeuge zum Prüfen der Codeformatierung wie *CheckStyle* (*http://eclipse-cs.sourceforge.net/*) und zur statischen Analyse auf potenzielle Fehler wie *Findbugs* (*http://findbugs.sourceforge.net/*) heraus.

Das Projekt QSQS konzentriert sich zunächst auf einfache Standard-Java-Programme, wobei in den nächsten Schritten Werkzeuge für den Last- und Performance-Test betrachtet werden, auch wenn es in den betrachteten Themengebieten weitere interessante Kandidaten gibt. Für fortgeschrittene Java-Technologien gibt es eine Vielzahl weiterer Testwerkzeuge, die teilweise auf den hier genannten Werkzeugen basieren. Die Analyse der hier vorgestellten Werkzeuge und der neuen Werkzeuge ist ein offenes Forschungsfeld. Dabei müssen die Werkzeuge nicht nur einzeln betrachtet werden; es stellt sich die Frage, mit welcher Kombination von Werkzeugen man die größte Chance hat, möglichst viele kritische Fehler zu finden.

9 Fazit und Ausblick

Eine funktionierende Qualitätssicherung benötigt Werkzeuge, mit denen man in möglichst geringer Zeit möglichst viele Fehler finden kann. Qualitätssicherung muss damit im Rahmen der Grundausbildung im Software Engineering ein wichtiger Baustein sein. Für einen kompakten Bachelor-Studiengang ergeben sich damit folgende notwendigen Inhalte:

- Einführung von Testverfahren: Äquivalenzklassentests und Überdeckungen
- Programme testbar machen; Erstellung von Mocks
- Konzepte der White-, Gray- und Black-Box-Tests
- Integration der Testansätze in den gesamten Entwicklungsprozess, Regressionstests, Änderungsmanagement [Wal01]

Um diese sehr trockenen Ideen mit Leben zu füllen, muss die Theorie in praktischen Übungen an konkreten Beispielen vertieft werden. Mittlerweile gibt es im Java-Umfeld eine unübersichtliche Anzahl von Werkzeugen, die für Tests frei zur Verfügung stehen. Wesentliches Teilergebnis des Projekts QSQS ist ein Vorschlag für eine minimale CSI-Ausstattung einer Software-Entwicklungsumgebung mit Testwerkzeugen. Dieser Vorschlag umfasst folgende Werkzeuge:

- JUnit 3.8, nach kurzer Einarbeitung eine sehr intuitive Möglichkeit, Tests in Java für Java-Programme zu erstellen
- CodeCover, zur Überprüfung der Anweisungsüberdeckung und zur Analyse Boole'scher Verzweigungsmöglichkeiten
- Metrics, zur statischen Analyse des Programmcodes, u. a. mit zyklomatischer Zahl nach McCabe und LCOM*

Möchte man das Thema Qualitätssicherung intensiver betrachten, gibt es vielfältige Möglichkeiten zur Vertiefung. Ein sehr interessantes Gebiet ist dabei der Test ausgehend von Oberflächen. Die dazu untersuchten Werkzeuge können als »nutzbar, aber verbunden mit recht hoher Einarbeitungszeit« klassifiziert werden.

Insgesamt erlaubt das Thema Qualitätssicherung mit seiner Integration in den Entwicklungsprozess hervorragend den Aufbau eigener Vertiefungsveranstaltungen. Ausgehend von der Betrachtung von Standard-Java-Programmen ergeben sich für die mannigfaltigen Java-Technologien viele weitere Analysemöglichkeiten für vorhandene Werkzeuge. Bei der Entwicklung einer Vertiefung kann die Möglichkeit zur Integration des ersten Zertifikats »Foundation Level« zum »Certified Tester« [@gtb] [@ist], wie z.B. in der GI-Fachgruppe »Test, Analyse und Verifikation von Software« [@stt] diskutiert, geschaffen werden. Durch die enorme Tool-Vielfalt gerade auch für neuere Java-Technologien eröffnet sich ein wunderbares Feld für unterschiedliche Arten von Experimenten zur Einrichtung eines spezialisierten CSI-Labors. Die aktuellen Analyseergebnisse stehen unter [@kbq] der interessierten Öffentlichkeit zur Verfügung.

Danksagung

Zu guter Letzt sei Andreas Himmighofen und Daniel Knott für die sehr gute Zusammenarbeit gedankt.

Literatur und Webadressen

Links zuletzt am 1.12.2008 geprüft.

[@ecp] Eclipse Plug-in Central, *http://www.eclipseplugincentral.com/*

[@gtb] German Testing Board, *http://german-testing-board.info/*

[@imb] imbus AG – Testlabor, *http://imbus.de/tool-list.shtml*

[@ist] ISTQB:Home, *http://www.istqb.org/*

[@kbq] Kombination von Qualitätssicherungsmaßnahmen, *http://www.informatik.fh-wiesbaden.de/~kleuker/KombiQu/kombiqu.html*

[@oss] open source software testing tools, news and discussion, *http://www.opensourcetesting.org/functional.php*

[@stt] GI Softwaretechnik.Trends, *http://pi.informatik.uni-siegen.de/stt*

[@tut] Eclipse-Plug-ins, die's bringen, *http://www.tutego.com/java/eclipse/plugin/eclipse-plugins.html#Testen*

[@ueb06] S. Ueberhorst, Wer zu spät testet, verschleudert Geld, *http://www.tecchannel.de/test_technik/software/448205, 22.09.2006*

[Bec00] K. Beck, Extreme Programming, Addison-Wesley, München, 2000

[Bec06] K. Beck, JUnit die Chance geben, ein wenig länger zu leben, Javamagazin 08/06, Seiten 23-24

[Hen96] B. Henderson-Sellers, Object-Oriented Metrics, Measures of Complexity, Prentice Hall, USA, 1996

[Hof08] D. W. Hoffmann, Software-Qualität, Springer-Verlag, Berlin Heidelberg, 2008

[HT05] A. Hunt, D.Thomas, Pragmatisch Programmieren: Unit-Tests mit JUnit, Hanser, München, Wien, 2005

[KG08] T. Kamann, M. Groh, Webtests mit Selenium, Groovy, TestNG und Maven, Javamagazin 08/08, Seiten 65-69

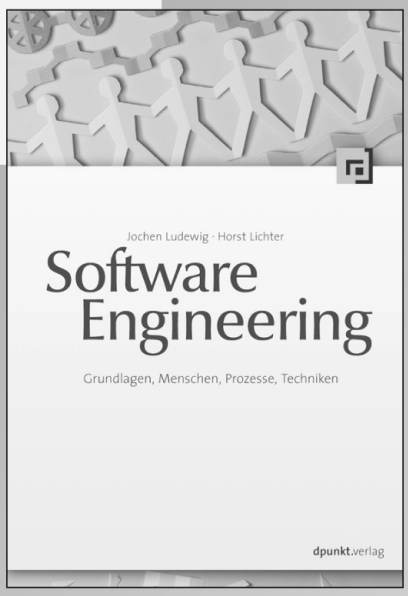
[Kle09] S. Kleuker. Grundkurs Software-Engineering mit UML, Vieweg+Teubner, Wiesbaden, 2009

[Lig02] P. Liggesmeyer, Software-Qualität. Testen, Analysieren und Verifizieren von Software, Spektrum Akademischer Verlag, Heidelberg Berlin Oxford, 2002

[Lin05] J. Link, Softwaretests mit JUnit, 2. Auflage, dpunkt.verlag, Heidelberg, 2005

[SW02] H. M. Sneed, M. Winter, Testen objektorientierter Software, Hanser, München, Wien, 2002

[Wal01] E. Wallmüller, Software-Qualitätsmanagement in der Praxis, 2. Auflage, Hanser, München Wien, 2001



2007, 640 Seiten, gebunden
€ 39,00 (D)
ISBN 978-3-89864-268-2

Jochen Ludewig · Horst Lichter

Software Engineering

Grundlagen, Menschen, Prozesse, Techniken

Das Buch vermittelt Grundelemente des Software Engineerings, um die Studenten für Projekte im Studium und für das Berufsleben auszurüsten.

Es ist in sechs Teile gegliedert:

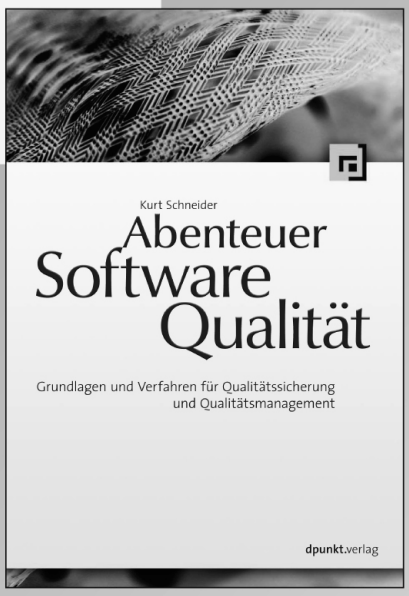
- Grundlagen
- Menschen und Prozesse
- Daueraufgaben im Softwareprojekt
- Techniken der Software-Bearbeitung
- Verwaltung und Erhaltung der Software
- Nachwort, Literatur und Index

Die Autoren betonen dabei oft vernachlässigte Faktoren wie Motivation und die Vermittlung von Zusammenhängen und gehen damit über die reine Darstellung von Fakten und Erfahrungen deutlich hinaus.

Der Inhalt des Buches wird durch zusätzliche Informationen auf den Webseiten der Autoren ergänzt.

dpunkt.verlag

Ringstraße 19 · 69115 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
http://www.dpunkt.de



2007, 224 Seiten, Broschur
€ 29,00 (D)
ISBN 978-3-89864-472-3

Kurt Schneider

Abenteuer Softwarequalität

Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement

Software ist aus Schule, Beruf und Privatleben nicht mehr wegzudenken. Entsprechend wächst die Bedeutung guter Softwarequalität.

Dieses Buch vermittelt ein fundiertes Grundwissen und einen breiten Überblick. Kann man Qualität messen, und wie kommt man zu sinnvollen Qualitätskriterien? Wie findet man Fehler in Programmen, und wie geht man beim systematischen Testen vor? Auch organisatorische Fragen, Bedienbarkeit von Software und agile Methoden werden betrachtet, ihr Bezug zur Softwarequalität wird erklärt.

Das Buch eignet sich als Begleitlektüre zu einer Vorlesung und zum Selbststudium. Die Darstellung führt einfach und verständlich in alle Themen ein. Dabei begleitet die Geschichte von »Q« den Leser. Q ist soeben Qualitätsbeauftragter geworden und erlebt Zweifel und Erfolge im »Abenteuer Softwarequalität«.

dpunkt.verlag

Ringstraße 19 · 69115 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
http://www.dpunkt.de

2007, 360 Seiten, Broschur
€ 36,00 (D)
ISBN 978-3-89864-433-4

Uwe Vigenschow · Björn Schneider

Soft Skills für Softwareentwickler

Fragetechniken, Konfliktmanagement, Kommunikationstypen und -modelle

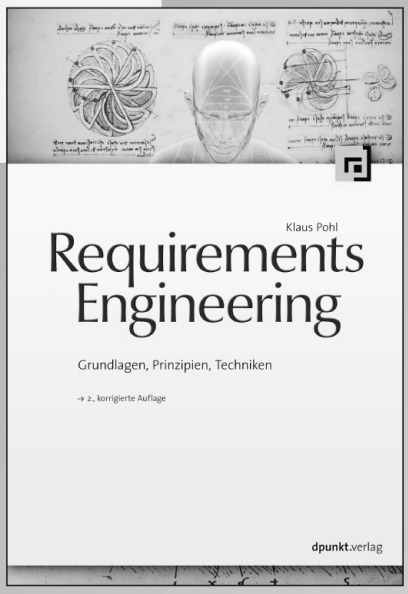
Unter Mitarbeit von Ines Meyrose

Viele Softwareprojekte scheitern nicht aus technischen Gründen, sondern oftmals aufgrund mangelnder Kommunikation.

Die Autoren zeigen praxisnahe Wege auf, im Arbeitsumfeld besser miteinander zu kommunizieren und Konflikte frühzeitig zu erkennen, um sie erfolgreich zu lösen. Aus ihrer langjährigen Entwickler- und Projektleiterpraxis heraus vermitteln sie die verschiedensten arbeitspsychologischen Modelle und Techniken mit konkreten Beispielen aus der IT. Behandelt werden Projektumfeldanalyse, Fragetechniken, hilfreiche Kommunikationsmodelle und Konfliktmanagement. Standardmethoden werden in die Software-Entwicklung transferiert und anwendbar gemacht. Anhand von zwölf Kommunikationsmustern werden konkrete Handlungswege aus typischen Sackgassen aufgezeigt.

dpunkt.verlag

Ringstraße 19 · 69115 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
http://www.dpunkt.de



2., korrigierte Auflage 2008,
764 Seiten, gebunden
€ 49,00 (D)
ISBN 978-3-89864-550-8

Klaus Pohl

Requirements Engineering

Grundlagen, Prinzipien, Techniken

2., korrigierte Auflage

Unter Mitarbeit von Kim Lauenroth, Ernst Sikora, Thorsten Weyer, Andreas Froese, Günter Halmans, Andreas Metzger

Requirements Engineering (RE) ist der Schlüssel zur erfolgreichen Entwicklung von softwareintensiven Systemen. Das moderne RE gewinnt neue, innovative Anforderungen an solche Systeme und überführt sie in detaillierte, möglichst vollständige Anforderungsspezifikationen.

Das Buch bietet eine umfassende Einführung in die Grundlagen, Prinzipien und Techniken des RE. Die kompakte Darstellung der Inhalte sowie deren Strukturierung anhand eines Rahmenwerks machen es zugleich zu einem Nachschlagewerk für Praxis, Lehre und Forschung.

Mit zahlreichen Beispielen, Checklisten und praktischen Hinweisen.

dpunkt.verlag

Ringstraße 19 · 69115 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
http://www.dpunkt.de

2., überarbeitete und erweiterte Auflage
2009, 575 Seiten, gebunden
€ 59,00 (D)
ISBN 978-3-89864-559-1

Ralf Reussner
Wilhelm Hasselbring (Hrsg.)

Handbuch der Software-Architektur

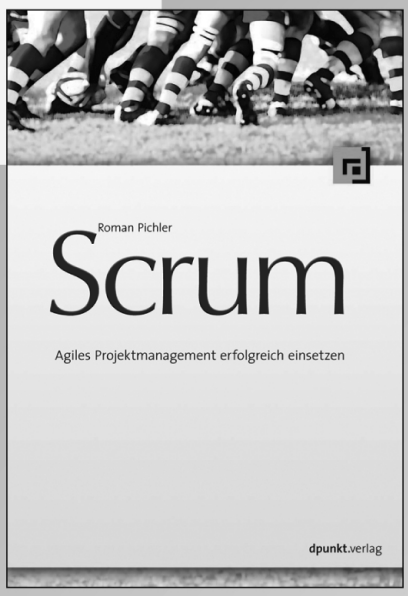
2., überarbeitete und erweiterte Auflage

Dieses Handbuch liefert einen fundierten Einstieg und Überblick über den Stand der Technik und zukunftsweisende Entwicklungen. Ausgehend von der Rolle des Software-Architekten werden Konstruktion und Evolution von Software-Architekturen systematisch aufbereitet. Aspekte des Managements, der Bewertung und Wiederverwendung von Architekturen werden im Detail erläutert und an Beispielen verdeutlicht. Die 2. Auflage wurde wesentlich überarbeitet und erweitert. Insbesondere werden serviceorientierte Architekturen und die modellgetriebene Softwareentwicklung nun umfassend behandelt.

»Das Handbuch wird seinen Namen voll und ganz gerecht und liefert eine umfassende theoretische Abhandlung zum Thema Software-Architektur, die ich in dieser Form bisher vermisst habe.« (Javamagazin 8.2006)

dpunkt.verlag

Ringstraße 19 · 69115 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
http://www.dpunkt.de



2008, 200 Seiten, Broschur
€ 29,00 (D)
ISBN 978-3-89864-478-5

Roman Pichler

Scrum

Agiles Projektmanagement erfolgreich einsetzen

Scrum ist ein agiles Projektmanagement-Framework, das sich auf alle Arten der Softwareentwicklung anwenden lässt. Richtig eingesetzt hilft es, Kundenzufriedenheit und Wertschöpfung nachhaltig zu steigern.

Dieses Buch vermittelt Ihnen das notwendige Wissen, um Scrum erfolgreich anzuwenden. Es beschreibt das Framework umfassend, systematisch und leicht verständlich.

Das Buch wendet sich an Einsteiger in Scrum und agiles Projektmanagement sowie an Fortgeschrittene bzw. an Leser, die sich auf Scrum-Zertifizierungen wie Certified ScrumMaster und Certified Scrum Product Owner vorbereiten.

dpunkt.verlag

Ringstraße 19 · 69115 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
http://www.dpunkt.de