Software Engineering im Unterricht der Hochschulen

Andreas Zeller ist Professor für Softwaretechnik an der Universität des Saarlandes. In der Forschung beschäftigt er sich mit der Diagnose und Vorhersage von Software-Fehlern; in der Lehre hat er sich durch innovative Organisation von Software-Praktika profiliert. Sein Buch »Why Programs Fail«, erschienen im dpunkt.verlag, wurde 2006 mit dem Software Development Magazine Productivity Award ausgezeichnet.

Marcus Deininger hat Informatik an der Universität Stuttgart studiert. Anschließend war er als wissenschaftlicher Mitarbeiter in der Abteilung Software Engineering der Universität Stuttgart tätig. Von 1996 bis 2004 war er Projektleiter und Systemberater bei debis Systemhaus und T-Systems. Seither ist er Professor für Informatik an der Hochschule für Technik Stuttgart mit dem Schwerpunkt Software Engineering.

Wir danken den Sponsoren der SEUH 2007

dpunkt.verlag

IBM

T · · Systems · · ·

vector

Andreas Zeller · Marcus Deininger (Hrsg.)

Software Engineering im Unterricht der Hochschulen

SEUH 10 – Stuttgart 2007

dpunkt.verlag

Andreas Zeller
zeller@cs.uni-sb.de

Marcus Deininger
marcus.deininger@hft-stuttgart.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 978-3-89864-458-7

1. Auflage 2007

Ringstraße 19
69115 Heidelberg

5 4 3 2 1 0

Vorwort

Der vorliegende Tagungsband enthält die Beiträge zum 10. Workshop *Software Engineering im Unterricht der Hochschulen* (SEUH). Ausbildung in Softwaretechnik an Universitäten und Fachhochschulen – wie ist sie in das Informatik-Studium eingebettet, wie sollte sie gestaltet werden, welche Erfahrungen gibt es? Dies sind die Themen, um die es bei dem SEUH-Workshop geht. 1992 von Jochen Ludewig ins Leben gerufen, findet der Workshop alle zwei Jahre statt, durchgeführt von der Gesellschaft für Informatik und dem German Chapter of the ACM gemeinsam mit der Schweizer Informatik-Gesellschaft.

Ausbildung in Softwaretechnik ist schwierig, weil das Gebiet einerseits nicht scharf umrissen und andererseits ohne intensive praktische Projektarbeit nicht gut zu vermitteln ist. In den letzten 15 Jahren hat es zahlreiche Versuche gegeben, die Softwaretechnik-Lehre zu verbessern; die SEUH ist das Forum, auf dem Erfolge und Erfahrungen vorgestellt, diskutiert und verfeinert werden.

Für die SEUH 2007 wurden 18 Beiträge eingereicht, aus denen das Programmkomitee 10 Beiträge zur Präsentation ausgewählt hat. Die Beiträge gliedern sich in fünf Sitzungen:

- Globale Ausbildung
- Software-Praktika
- Entwurf und Architektur
- Teamarbeit und Simulation
- Programmierung

Auch in diesem Jahr beschäftigte sich der überwiegende Teil der eingereichten Beiträge mit Fragen der Software-Praktika – offensichtlich ist dies ein Dauerbrenner für die Softwaretechnik-Lehre. Wir haben uns sehr über Beiträge mit fundierter empirischer Auswertung gefreut, da sie den Nutzen (oder Nicht-Nutzen) der diskutierten Konzepte belegen.

Die Konzeption der SEUH bietet viel Raum für Diskussionen, und dies ist in der Vergangenheit ausgiebig – und auch mit lebhaften, doch konstruktiven Kon-

troversen – genutzt worden. Entsprechend haben wir auch darauf geachtet, dass die ausgewählten Beiträge der Diskussion förderlich sind und dass das Programm genügend Raum für Meinungsaustausch lässt. Viele Dozenten der Softwaretechnik haben von der SEUH entscheidende Impulse für ihre Lehre erhalten. Wir hoffen, dass auch die SEUH 2007 solche Impulse gibt und so eine engagierte und erfolgreiche Lehre der Softwaretechnik an den Hochschulen fördert.

Wir bedanken uns bei allen, die bei der Organisation und Durchführung der SEUH 2007 mitgewirkt haben: dem Programmkomitee, der Mannschaft des Konferenz-Verwaltungssystems EasyChair, unseren Sponsoren – und natürlich den Autoren der eingereichten Beiträge, die die SEUH mit Leben füllen.

Januar 2007
Andreas Zeller und Marcus Deininger

Das Programmkomitee der SEUH 2007

- Ralf Bruns, Fachhochschule Hannover
- Marcus Deininger, Hochschule für Technik Stuttgart (Vorsitz)
- Gregor Engels, Universität Paderborn
- Martin Glinz, Universität Zürich
- Jens Krinke, Fernuniversität Hagen
- Lutz Prechelt, Freie Universität Berlin
- Günter Riedewald, Universität Rostock
- Silke Seehusen, Fachhochschule Lübeck
- Kurt Schneider, Universität Hannover
- Debora Weber-Wulff, Fachhochschule für Technik und Wirtschaft Berlin
- Andreas Zeller, Universität des Saarlandes (Vorsitz)
- Olaf Zukunft, Hochschule für Angewandte Wissenschaften Hamburg

Das ständige Organisationskomitee der SEUH

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

Inhalt

**Eingeladener Vortrag:
Was wir vom Software-Ingenieur erwarten:
Gleichgewicht von Fachwissen und Persönlichkeit** **1**

Ernst Denert

**Die Ausbildung der Software-Ingenieure in Indien und Deutschland –
ein Vergleich** **3**

Holger Röder

**Ein verbessertes Softwaretechnikpraktikum:
zwischen grüner Wiese und Legacy-Systemen** **13**

Björn Axenath · Stefan Henkler

Positive Effekte von Szenarien und Features in einem Softwarepraktikum **27**

Kim Lauenroth · Ernst Sikora · Klaus Pohl

**Selbstbestimmung oder Anleitung:
Erfahrungen mit einem Softwaretechnikpraktikum im
Bereich Qualitätssicherung** **41**

Sebastian Jekutsch · Christopher Oezbek · Stephan Salinger

**Projektorientierte Vermittlung von Entwurfsmustern in der
Software-Engineering-Ausbildung** **45**

*Wilhelm Hasselbring · Jasminka Matevska · Heiko Niemann · Dennis Geesen ·
Hilke Garbe · Stefan Gudenkauf · Steffen Kruse · Claus Möbus · Marco Grawunder*

Software Engineering moderner Anwendungen **59**

Jürgen Rückert · Barbara Paech

Software-Engineering-Simulation als Brücke zwischen Vorlesung und Praktikum **73**

Tilmann Hampp · Stefan Opferkuch

Einfluss von Qualitätsdruck und Kontinuität der Zusammenarbeit auf virtuelle Teamarbeit **83**

Marc Kasparek · Nicola Marsden

Einführung in die Softwareentwicklung – Softwaretechnik trotz Objektorientierung? **87**

Axel Schmolitzky · Heinz Züllighoven

Ein Experiment zur Ermittlung der Programmierproduktivität von Studenten **101**

Matthias Wetzel

Eingeladener Vortrag:

Was wir vom Software-Ingenieur erwarten: Gleichgewicht von Fachwissen und Persönlichkeit

Ernst Denert

IVU Traffic Technologies AG, Berlin
denert@ivu.de

Zusammenfassung

Was erwarten wir von jungen Mitarbeitern, die neu zu uns kommen, meist frisch von der Hochschule? Was brauchen die Software-Ingenieure in der Praxis an Kenntnissen, Fähigkeiten und Persönlichkeit, um Software zu entwickeln und damit unsere Kunden zufrieden zu stellen?

Die IVU bietet IT-Systeme zur Planung, Steuerung und Optimierung von Fahrzeugflotten, vor allem im öffentlichen Personenverkehr. Sie basieren auf IVU-eigenen Produkten, einschließlich Bordcomputer-Hardware, die in Projekten zum Einsatz gebracht werden. Dazu werden die Produkte weiterentwickelt und angepasst, die Systeme installiert, Daten aufbereitet, Anwender geschult etc., nicht zu vergessen das Projektmanagement.

Das bedeutet, dass unsere Mitarbeiter nicht nur im Labor entwickeln, sondern auch eng mit den Kunden zusammenarbeiten. Auch junge Mitarbeiter haben intensiven Kundenkontakt. Projektarbeit ist Teamwork – IVU- und Kundenmitarbeiter müssen harmonieren, damit ein sachlich gutes Ergebnis zustande kommt. Aber ebenso ist die schöpferische Einzelleistung gefragt, mit geistiger Disziplin und kreativ zugleich. Methoden und Werkzeuge des Software Engineering helfen, allerdings nur begrenzt. All das fasst unser Motto zusammen:

Menschen machen Projekte.

Unsere Produkte und Projekte sind interessant und anspruchsvoll, jedes eine Herausforderung für jeden Mitarbeiter, insbesondere für die jungen und neuen. Sie müssen schon einiges mitbringen, um dem gewachsen zu

sein. Was das bedeutet, wird im Vortrag ausgeführt: welche fachlich-technische Qualifikation einerseits nötig ist, also welche Kenntnisse und Fähigkeiten, und andererseits welche Persönlichkeitsmerkmale für erfolgreiche Produkte und Projekte besonders wichtig sind.

Die Ausbildung der Software-Ingenieure in Indien und Deutschland – ein Vergleich

Holger Röder

Universität Stuttgart, Institut für Softwaretechnologie
Universitätsstraße 38, 70569 Stuttgart
holger.roeder@informatik.uni-stuttgart.de

Zusammenfassung

Ländergrenzen und Sprachbarrieren haben bei der Entwicklung von Software an Bedeutung verloren. Software wird heute oftmals dort entwickelt, wo qualifizierte Fachkräfte kostengünstig zur Verfügung stehen. Software-Ingenieure sehen sich somit nicht nur einer Konkurrenzsituation auf dem nationalen Arbeitsmarkt gegenüber, sondern stehen zusätzlich im globalen Wettbewerb mit Software-Ingenieuren aus aller Welt. Die Qualität der Ausbildung, die Software-Ingenieure erhalten, gewinnt dadurch an Bedeutung.

Dieser Beitrag beschreibt die akademische Ausbildung von Software-Ingenieuren in Indien. Anhand der von der Software-Industrie geforderten Kenntnisse und Fähigkeiten, über die Absolventen verfügen sollten, werden die wichtigsten Schwächen der Ausbildung identifiziert: mangelnde Praxiserfahrung der Absolventen, zu wenig Software Engineering, zu wenig Soft Skills. Dem gegenüber stehen die kurze Studiendauer und die hohe Motivation der Studenten. Zudem bemüht sich die indische Software-Industrie stärker als die deutsche, Schwächen der akademischen Ausbildung durch intensive Schulungsprogramme auszugleichen. Für die Verbesserung der Ausbildung von Software-Ingenieuren an deutschen Universitäten finden sich in Indien jedoch kaum Anregungen. Stattdessen gilt es, den vorhandenen Vorsprung der deutschen Universitäten im Ausbildungsbereich durch eine stärkere Berücksichtigung der Themen Software Engineering und Soft Skills in den Informatik-Studiengängen auszubauen und dadurch einen Beitrag zur Stärkung der Konkurrenzfähigkeit der Software-Entwicklung in Deutschland zu leisten.

1 Einführung

Länder wie Indien und China sind in den vergangenen zwei Jahrzehnten zu bedeutenden Mitspielern auf dem weltweiten Softwareentwicklungsmarkt herangewachsen. Zu den wichtigsten Gründen für ihren Aufstieg zählen die im Vergleich zu den westlichen Industriestaaten deutlich niedrigeren Löhne und ein nahezu unerschöpfliches Reservoir an jungen und motivierten Software-Ingenieuren [Aro 02]. So werden beispielsweise in Indien im Jahr 2007 voraussichtlich mehr als 160.000 Absolventen ihr Studium der Informatik, Elektronik oder Telekommunikation abschließen [NAS 06] und dem IT-Arbeitsmarkt zur Verfügung stehen. Auf den beiden Feldern Lohnniveau und Fachkräftezahl hat Deutschland diesen Ländern auf absehbare Zeit nichts entgegenzusetzen. Für die deutsche Software-Industrie als Ganzes wie auch für den einzelnen Software-Ingenieur hierzulande stellt sich darum zwangsläufig die Frage, auf welche andere Weise dieser globalen Konkurrenzsituation begegnet werden kann.

Die akademische Ausbildung, die Software-Ingenieure erhalten, hat ohne Zweifel Einfluss auf die beruflichen Perspektiven des Einzelnen und auf die Qualität und damit die Konkurrenzfähigkeit der Software-Entwicklung eines Landes als Ganzes. Es liegt darum nahe, die Ausbildung in verschiedenen miteinander im Wettbewerb stehenden Ländern zu betrachten und zu vergleichen. In vielen Ländern besitzen Software-Ingenieure ähnlich titulierte Abschlüsse in Fächern wie Informatik, Software Engineering oder auch Ingenieurwissenschaften. Ein Blick in die Literatur zeigt jedoch, dass über die Unterschiede und Gemeinsamkeiten und damit auch über die Stärken und Schwächen dieser Studiengänge und der Ausbildung von Software-Ingenieuren in verschiedenen Ländern nur wenig bekannt ist.

Im Rahmen einer Diplomarbeit [Röd 06] wurde die Ausbildung von Software-Ingenieuren in Indien und Deutschland am Beispiel der Informatik-Studiengänge am Indian Institute of Technology, Kanpur, und der Informatik- und Softwaretechnik-Diplomstudiengänge an der Universität Stuttgart untersucht. Die dabei gewonnenen Erkenntnisse über die Situation in Indien sind in diesem Beitrag zusammengefasst. Dabei wird die Kenntnis der universitären Ausbildung in Deutschland vorausgesetzt. Der Beitrag ist wie folgt aufgebaut: Kapitel 2 beschreibt kurz das Universitätssystem in Indien. Kapitel 3 erläutert die wesentlichen Merkmale der Ausbildung von Software-Ingenieuren an Universitäten in Indien mit Blick auf Aufbau und Inhalt der dortigen Informatik-Studiengänge. Kapitel 4 beschäftigt sich mit den Anforderungen der Software-Industrie an angehende Software-Ingenieure und mit der Frage, inwieweit Absolventen in Deutschland und Indien diese erfüllen. In Kapitel 5 wird die Ausbildung indischer Software-Ingenieure unter anderem anhand dieser Anforderungen bewertet. In Kapitel 6 wird ein abschließendes Fazit gezogen.

2 Das Universitätssystem in Indien

Das indische Universitätssystem orientiert sich am angelsächsischen Bachelor-Master-Modell. Das Bachelor-Studium dauert regulär vier Jahre und führt in den technischen Studiengängen zum berufsqualifizierenden Abschluss Bachelor of Technology (B. Tech.; seltener auch: Bachelor of Engineering, B. Eng.). Ein Bachelor-Abschluss ist Voraussetzung für ein Master-Studium, das weitere zwei Jahre dauert und mit dem Abschluss Master of Technology (M. Tech., seltener: Master of Engineering, M. Eng.) beendet wird. Vereinzelt werden auch integrierte Bachelor-Master-Studiengänge angeboten, in denen beide Abschlüsse nach insgesamt fünf Jahren vergeben werden.

Indische Studenten beginnen ihr Studium im Alter von 17 bis 18 Jahren und damit deutlich früher als Studenten in Deutschland. Das Studium folgt in aller Regel direkt im Anschluss an die zwölfjährige Schulzeit. Eingeschobene Aktivitäten, etwa Wehr- oder Sozialdienst, Praktika oder berufliche Tätigkeiten, wie sie in Deutschland durchaus üblich sind, sind in Indien die Ausnahme. Die Regelstudienzeit wird von den meisten Studenten eingehalten, da die Studienpläne im Normalfall nur wenig Flexibilität bieten. Entsprechend sind Bachelor-Absolventen im Durchschnitt 21 oder 22 Jahre alt, Master-Absolventen zwei Jahre älter.

Die Unterschiede zwischen den einzelnen Universitäten sind in Indien deutlich größer als in Deutschland. Die Universitäten unterscheiden sich teilweise erheblich in der Qualität von Forschung und Lehre, der Infrastruktur und den allgemeinen Studienbedingungen. Wenigen Spitzenuniversitäten mit hoher Reputation steht eine breite Masse von Universitäten mit schlechter ausgebildeten Dozenten und unzureichender Ausstattung gegenüber. Im technischen Bereich stehen die sieben staatlich finanzierten Indian Institutes of Technology (IITs) an der Spitze der indischen Universitätspyramide. Sie gelten als Eliteschmieden und schneiden auch im internationalen Vergleich gut ab. Etwas schwächer, aber immer noch überdurchschnittlich, werden die National Institutes of Technology (NITs) und die privaten International Institutes of Information Technology (IIITs) eingeschätzt. Schließlich existiert noch eine unübersichtliche Vielzahl regionaler Universitäten und privater Colleges, die ebenfalls Informatik- und Ingenieurabschlüsse anbieten, deren Qualität aber häufig als nicht ausreichend kritisiert wird [Man 02], [Bag 99]. Ein wichtiger Grund hierfür ist die mangelnde Qualifikation der Lehrenden. Das Durchschnittsgehalt eines Professors liegt deutlich unter den Einstiegsgehältern in der Softwareindustrie, was den Beruf insbesondere für hochqualifizierte Software-Ingenieure wenig attraktiv macht. An vielen Universitäten lehren deshalb Dozenten, die selbst nur über einen Master-Abschluss, nur wenig oder keine Berufserfahrung und lediglich eingeschränkte didaktische Fähigkeiten verfügen. Das Problem der mangelnden Qualifikation der Lehrkräfte ist seit langem bekannt (siehe hierzu auch den etwas älteren, aber durchaus noch aktuell erscheinenden Beitrag [Ram 97]). Mittlerweile existieren

Kooperationen zwischen der Industrie und den Universitäten mit dem Ziel, die Situation zu verbessern. Dennoch existiert das Problem auch heute noch an vielen Universitäten.

Der Zugang zu den Universitäten wird über Zulassungsprüfungen geregelt. Die Möglichkeit, an einer der renommierten Universität zu studieren, wird durch die große Zahl der Bewerber und die entsprechend anspruchsvollen Zulassungsprüfungen sehr stark eingeschränkt: So liegt die Zulassungsquote für ein Studium an einem IIT typisch bei unter zwei Prozent. Dennoch steigt die Bewerberzahl seit Jahren stetig an.

3 Überblick über die akademische Ausbildung von Software-Ingenieuren in Indien

Indische Universitäten bieten keine spezielle akademische Ausbildung für Software-Ingenieure an. Prinzipiell steht auch in Indien jedem Bewerber die Möglichkeit offen, als Software-Ingenieur zu arbeiten. In der Praxis hat jedoch die überwiegende Mehrheit der jährlich neu eingestellten Software-Ingenieure in Indien ein technisches Studium abgeschlossen. Da der Personalbedarf der Software-Industrie durch Informatikabsolventen allein quantitativ nicht gedeckt werden kann, werden neben Informatikern vor allem auch Ingenieure, etwa Absolventen der Fachrichtungen Elektrotechnik oder Elektronik, eingestellt. Absolventen nichttechnischer Studiengänge und Berufseinsteiger ohne abgeschlossenes Studium spielen zahlenmäßig kaum eine Rolle. Die folgenden Aussagen zur akademischen Ausbildung von Software-Ingenieuren beziehen sich im Wesentlichen auf Informatik-Studiengänge in Indien.

Im Unterschied zu vielen Informatik-Studiengängen in Deutschland, die in der Tradition der Mathematik stehen, weisen Informatik-Bachelor-Studiengänge in Indien in Bezug auf Inhalt und Aufbau des Studiums deutlich mehr Gemeinsamkeiten mit ingenieurwissenschaftlichen Fächern auf. Häufig drückt sich dies auch in den Namen der Studiengänge (»Computer Science and Engineering«) aus. Die Studiengänge sind typisch zweigeteilt. Der allgemeine Teil umfasst Grundlagenveranstaltungen aus den Bereichen Mathematik, Natur-, Ingenieur- und Geisteswissenschaften und soll ein möglichst breites Grundlagenwissen vermitteln, für das der Begriff Allgemeinbildung treffend erscheint. Niveau und Inhalt dieser Veranstaltungen entsprechen insbesondere bei den Natur- und Geisteswissenschaften eher der deutschen gymnasialen Oberstufe als Vorlesungen entsprechender deutscher Magister- oder Diplomstudiengänge, was durch die kürzere Schulzeit indischer Studenten und vor allem durch das allgemein niedrigere Niveau der schulischen Ausbildung erklärt werden kann. Der allgemeine Teil steht typisch am Anfang des Bachelor-Studiums und ist innerhalb einer Universität für alle technischen Studiengänge ähnlich oder gleich.

Der fachspezifische Teil des Studiums umfasst Veranstaltungen, in denen Kenntnisse im eigentlichen Studienfach vermittelt werden. In der Informatik gleichen die Titel der Grundlagenvorlesungen weitgehend denen deutscher Informatik-Studiengänge: Einführungen in die praktische, theoretische und technische Informatik, Betriebssysteme, Datenbanken, Programmiersprachen, Verteilte Systeme und Wissensverarbeitung gehören auch in Indien zum Standardrepertoire. Auch zum Thema Software Engineering werden üblicherweise Einführungsvorlesungen angeboten. Darüber hinaus werden vertiefende Vorlesungen zu Themen der Informatik angeboten, deren Inhalt und Umfang jedoch zu stark von der jeweiligen Fakultät und den verantwortlichen Dozenten abhängt, als dass darüber allgemeine Aussagen gemacht werden könnten. [Röd 06] untersucht beispielhaft die Situation am Indian Institute of Technology, Kanpur (IITK), und kategorisiert die dort angebotenen Lehrveranstaltungen nach [IEEE 01]: Es zeigt sich, dass sich neben der Grundlagenvorlesung nur drei (unregelmäßig angebotene) Veranstaltungen mit Themen des Software Engineering beschäftigen (zum Vergleich: an der Universität Stuttgart werden zehn vertiefende Vorlesungen angeboten). Da das IITK zu den führenden technischen Universitäten gehört, liegt die Vermutung nahe, dass Software Engineering auch an anderen indischen Universitäten nicht zu den stark vertretenen Themen gehört und über die Grundlagen hinausgehende Veranstaltungen eher die Ausnahme sind. Ein Blick in die Vorlesungsverzeichnisse einiger anderer indischer Universitäten unterstützt diese Annahme. Dasselbe gilt für größere Software-Projekte innerhalb des Studiums: Zwar enthalten viele Informatikvorlesungen Übungseinheiten, in denen kleinere Programmieraufgaben gelöst werden müssen, umfangreichere Projekte, in denen Studenten ihr spezifisches Software-Engineering-Wissen (etwa über Projektmanagement, Kostenschätzung oder Konfigurationsverwaltung) praktisch anwenden, werden jedoch nicht angeboten [Mah 05].

Die zweijährigen Informatik-Master-Studiengänge bieten Studenten die Möglichkeit, sich auf den von ihnen bevorzugten Gebieten der Informatik zu spezialisieren. Der Studienplan sieht zumeist Wahlfächer vor und ist somit inhaltlich flexibel, weshalb an dieser Stelle keine weiteren Aussagen zum Master-Studium gemacht werden.

4 Anforderungen der Industrie

Traditionell ist die Zielsetzung der universitären Ausbildung zweigeteilt: Neben der Vermittlung der grundsätzlichen Fähigkeit zur Forschung auf dem Gebiet der Informatik sollen Universitäten Software-Ingenieure auch für die berufliche Praxis ausbilden. Die Anforderungen, die in der Praxis an Absolventen gestellt werden, können somit als eine wesentliche Grundlage für die Bewertung der Ausbildung angesehen werden. Im Rahmen des Vergleichs der Studiengänge in Stuttgart und Kanpur [Röd 06] wurden aus diesem Grund Verantwortliche in jeweils sechs

indischen und deutschen Firmen befragt, über welche Kenntnisse und Fähigkeiten angehende Software-Ingenieure verfügen sollten. Dabei zeigte sich, dass die Anforderungen, die die Firmen an ihre neuen Mitarbeiter stellen, in beiden Ländern ähnlich sind. Im Wesentlichen wird auf vier Punkte Wert gelegt:

- Fundierte Informatik-Kenntnisse werden als Grundvoraussetzung für eine Tätigkeit als Software-Ingenieur gesehen. Insbesondere anwendungsnahes Wissen über Datenstrukturen und Algorithmen, Datenbanken und Verteilte Systeme wird vorausgesetzt.
- Ausdrücklich verlangt (und deshalb an dieser Stelle und im Weiteren als eigener Punkt aufgeführt) werden Software-Engineering-Kenntnisse. Angehende Software-Ingenieure benötigen aus Sicht der Firmen ein grundlegendes Verständnis der wichtigsten Konzepte und Vorgehensweisen des Software Engineering und der verschiedenen Tätigkeiten im Rahmen der Software-Entwicklung.
- Neben den aufgeführten fachlichen Fähigkeiten wird auch den so genannten Soft Skills eine große Bedeutung zugemessen. Dazu gehören unter anderem gutes mündliches und schriftliches Ausdrucksvermögen, Teamfähigkeit und professionelles Auftreten gegenüber Vorgesetzten und Kunden.
- Ebenfalls einhellig gewünscht wird, dass die Absolventen bereits über möglichst viel praktische Erfahrung in der Software-Entwicklung verfügen. Diese Praxiserfahrung sollte über reine Programmiererfahrung (etwa im Programmierkurs an der Universität) hinausgehen und sich aus der Mitarbeit an »echten« Software-Projekten speisen.

Die genannten Fähigkeiten entsprechen in weiten Teilen den in [GI 05] genannten Kompetenzen, deren Vermittlung die Gesellschaft für Informatik e.V. als Ausbildungsziel für Informatik-Studiengänge empfiehlt. Auffällig ist, dass deutsche und indische Firmen wie oben erwähnt bezüglich der gewünschten Fähigkeiten weitgehend übereinstimmen. Deutliche Unterschiede gibt es jedoch hinsichtlich der Bewertung, in welchem Maße Absolventen, die sich als Software-Ingenieure bewerben, diese Fähigkeiten besitzen.

In Deutschland stellen Unternehmen zumeist Absolventen der Informatik oder verwandter Studienfächer ein, wenn sie Positionen im Bereich Software-Entwicklung mit Absolventen besetzen möchten. Offenbar besteht zurzeit kein Mangel an Informatik-Absolventen, wie es noch vor einigen Jahren der Fall war. Die Informatik-Absolventen verfügen nach Aussage der Firmen in der Regel über gute Informatik-Kenntnisse und ausreichende praktische Erfahrung durch Nebentätigkeiten oder Praktika. Schwächen sehen die Verantwortlichen bei den Software-Engineering-Kenntnissen und besonders bei den Soft Skills. Diese Aussage deckt sich mit den Ergebnissen vergleichbarer Untersuchungen [Bur 03]. Die Mehrheit der Bewerber verfügt über Grundkenntnisse im Software Engineering, hat dieses Gebiet im Studium jedoch nicht vertiefend behandelt. Zudem wird

bemängelt, dass Soft Skills an den Universitäten üblicherweise nicht explizit vermittelt werden, etwa in speziellen Präsentationskursen. Zusammenfassend wird die Ausbildung der Informatik-Absolventen dennoch als ausreichend für einen direkten Einstieg in die Arbeitswelt angesehen. Weitergehende Fähigkeiten, etwa Kenntnisse spezieller Technologien oder Werkzeuge, werden üblicherweise in kurzen, ein- bis zweitägigen Schulungen vermittelt.

In Indien ist die Situation anders. Zwar verfügen die meisten Bewerber mit Informatik-Abschluss auch über ausreichende Informatik-Kenntnisse, doch vielen Absolventen ingenieurwissenschaftlicher Studiengänge mangelt es selbst daran. Bei beiden Gruppen vermissen die Firmen darüber hinaus Software-Engineering-Kenntnisse ebenso wie praktische Erfahrung in Software-Projekten und Soft Skills. Aus diesem Grund werden junge Software-Ingenieure nach ihrem Einstieg nicht direkt mit Projektarbeit betraut, sondern stattdessen zunächst in firmeninternen Schulungsprogrammen bis zu vier Monate lang auf ihre spätere Tätigkeit vorbereitet. Struktur und Inhalt dieser Schulungsprogramme erinnern in Ansätzen an vollwertige Informatik-Studiengänge. Auf dem Stundenplan stehen im technischen Teil Informatik-Grundlagen, darunter auch Software Engineering, sowie konkrete Technologien, die in späteren Projekten voraussichtlich zum Einsatz kommen. Der Fokus liegt bei allen Themen auf der praktischen Anwendung, nicht auf einer detaillierten theoretischen Betrachtung. Neben den technischen Themen ist die Vermittlung von Soft Skills Schwerpunkt der Schulungsprogramme, angefangen bei Präsentationsfähigkeiten bis hin zu geschäftlicher Etikette und Kleiderordnung. Einen detaillierten Einblick in das Schulungsprogramm eines großen indischen IT-Unternehmens liefert etwa [Nar 01]. Erst nachdem die Absolventen das Schulungsprogramm erfolgreich abgeschlossen haben, beginnen sie ihre Tätigkeit als »vollwertige« Software-Ingenieure.

5 Bewertung

Ganz offensichtlich unterscheidet sich die Ausbildung von Software-Ingenieuren in Indien an vielen Stellen von der Ausbildung in Deutschland. Diese Unterschiede sind sowohl außerhalb als auch innerhalb der Universitäten begründet.

Aufgrund des hohen jährlichen Bedarfs an neuen Fachkräften muss die indische Software-Industrie in großem Umfang auf Universitätsabsolventen mit ingenieurwissenschaftlichem Abschluss zurückgreifen, die in ihrem Studium nur wenig über Informatik und sehr wahrscheinlich nichts speziell über Software Engineering gelernt haben. Hinzu kommt, dass nur wenige Absolventen, seien es Informatiker oder Ingenieure, über nennenswerte praktische Erfahrung in Software-Projekten verfügen. Nebentätigkeiten oder Praktika in der Software-Entwicklung, die viele Studenten in Deutschland während ihres Studiums absolvieren, sind in Indien selten. Diese fehlenden Eindrücke aus dem Berufsleben mögen auch einer der Gründe sein, weshalb die Firmen in Indien über mangelnde

Soft Skills und Teamfähigkeit der Absolventen klagen. Ein weiterer Aspekt hierbei ist sicherlich die Tatsache, dass Absolventen in Indien bei Abschluss ihres Studiums vergleichsweise jung sind. Der Großteil der Studenten folgt dem vorgesehenen Studienplan und schließt das Bachelor-Studium innerhalb von vier Jahren (und gegebenenfalls das anschließende Master-Studium in weiteren zwei Jahren) erfolgreich ab. Auch wenn, wie beispielsweise in der deutschen Debatte um die Einführung von Bachelor-Studiengängen, ein schnellerer Berufseintritt von vielen als erstrebenswert betrachtet wird, bedeutet er in aller Regel doch, dass die Absolventen unreifer und unerfahrener sind.

Auch mit Blick auf die Studieninhalte weist das Informatik-Studium an indischen Universitäten Unterschiede zu einem vergleichbaren Studium in Deutschland auf. Hier fällt (beim Bachelor-Studium) zunächst der hohe Anteil an informatikfremden Lehrinhalten auf: Das Studium dient in Indien in weitaus größerem Maße als in Deutschland der Vermittlung einer umfassenden Allgemeinbildung, da das indische Schulsystem hierzu nicht in der Lage ist. Entsprechend ist der Umfang der im Studium behandelten Informatik-Themen üblicherweise geringer als etwa in Diplomstudiengängen an deutschen Universitäten. In Schulungsprogrammen versucht die indische Software-Industrie, diesen Mangel auszugleichen. Abgesehen von renommierten Universitäten wie den IITs leidet die Lehre zudem oftmals unter der mangelnden Qualifikation der Dozenten, die sich auf die fehlende Berufserfahrung im Bereich Software-Entwicklung zurückführen lässt. Das praxisbezogene Thema Software Engineering, für dessen Vermittlung persönliche Kenntnis der täglichen Praxis in der Software-Industrie unerlässlich ist, ist von diesem Problem in besonderem Maße betroffen. Aus diesem Grund nimmt das Thema im Studium der meisten indischen Informatik-Studenten nur eine Nebenrolle ein.

Zusammenfassend lässt sich sagen, dass die formale akademische Ausbildung von Software-Ingenieuren in Indien eine ganze Reihe von Schwachpunkten aufweist und in vielen Belangen der Ausbildung in Deutschland unterlegen ist. Dies wird teilweise ausgeglichen durch die hohe Motivation, die bei indischen Studenten erkennbar ist. Der Beruf des Software-Ingenieurs gehört zurzeit zu den bevorzugten beruflichen Perspektiven vieler Schüler und Studenten in Indien und genießt ganz allgemein ein hohes gesellschaftliches Ansehen. Wie bereits erwähnt beenden die meisten Studenten ihr Studium in der Regelstudienzeit, der Anteil der Studienabbrecher ist minimal. Zudem wird von verschiedenen Seiten an einer weiteren Verbesserung der Ausbildung gearbeitet. Unternehmen der Software-Industrie finanzieren die Weiterbildung von Universitätsdozenten oder schicken direkt erfahrene Mitarbeiter als Dozenten an die Hochschulen. Neue staatliche und private Universitäten werden gegründet und bieten Informatik-Studiengänge an, was die Zahl qualifizierter Absolventen in Zukunft ansteigen lassen wird. Weiterhin sollen strengere Prüfungen bei der Erteilung staatlicher Akkreditierungen für Universitäten die Qualität der akademischen Ausbildung erhöhen. Es bleibt abzuwarten, ob diese Maßnahmen die erhoffte Wirkung bringen.

6 Fazit

Der bemerkenswerte Aufstieg der indischen Software-Industrie lässt sich nicht bestreiten. Gleiches gilt für die eingangs genannten Gründe für den Aufstieg, also das niedrige Lohnniveau und die große Zahl verfügbarer Fachkräfte. Die aus der akademischen Ausbildung resultierende Qualifikation dieser Fachkräfte kann jedoch nicht zu den Gründen gezählt werden. Wie in diesem Bericht dargelegt wurde, weist die akademische Ausbildung von Software-Ingenieuren in Indien deutliche Schwächen gegenüber der Ausbildung in Deutschland auf. Wenn also in der Debatte um Vor- und Nachteile einer Verlagerung der Software-Entwicklung nach Indien die angeblich überlegenen Fähigkeiten indischer Software-Ingenieure im Vergleich zu ihren deutschen Kollegen genannt werden, ist dieses Argument in vielen Fällen unberechtigt.

Die ursprüngliche Motivation für die Untersuchung der Ausbildungssituation in Indien lag unter anderem darin, Anregungen und Verbesserungsvorschläge für die Software-Engineering-Ausbildung an deutschen Universitäten zu finden. Diese Suche blieb jedoch weitgehend erfolglos. Software Engineering wird trotz der Tatsache, dass die meisten Absolventen eine berufliche Karriere in der Software-Industrie anstreben, in den Informatik-Studiengängen indischer Universitäten nur selten speziell berücksichtigt. Für indische Universitäten lautet deshalb die – zugegebenermaßen pauschale – Empfehlung, die offenbar vorhandene breite Kluft zwischen Studieninhalten und Industrieanforderungen zu verkleinern.

Für Universitäten hierzulande besteht die wesentliche Erkenntnis darin, dass die akademische Ausbildung von Software-Ingenieuren in Deutschland heute durchaus konkurrenzfähig und in vielen Belangen der Ausbildung in Indien sogar überlegen ist. Angesichts der rapiden Veränderungen in Indien in den vergangenen zwei Jahrzehnten muss jedoch davon ausgegangen werden, dass die momentan noch existierenden Probleme in der akademischen Ausbildung nach und nach beseitigt werden. Zudem gilt es zu beachten, das die Ausbildung vieler indischer Software-Ingenieure wie beschrieben nicht nur das Studium an einer Universität, sondern auch ein intensives und zielgerichtetes Schulungsprogramm in der Industrie umfasst. Es besteht in Deutschland also wenig Anlass, sich mit dem Status quo zufriedenzugeben. Stattdessen sollte darüber nachgedacht werden, die von den befragten Firmen genannten Punkte, also das auch in vielen Informatik-Studiengängen in Deutschland offenbar eher vernachlässigte Thema Software Engineering und die explizite Vermittlung von Soft Skills, bei der zukünftigen Ausbildung von Software-Ingenieuren in Deutschland stärker zu berücksichtigen.

Literatur

[Aro 02] A. Arora und S. Athreye: The Software Industry and India's Economic Development. Journal of Information Economics and Policy, Elsevier, Vol. 14, Issue 2, 2002. 253–273.

[Bag 99] S. Bagchi: India's Software Industry: The People Dimension. IEEE Software, Vol. 16, Issue 3, 1999. 62–65.

[Bur 03] W. Burhenne: Bachelor/Master im Informatikstudium und im Beruf. Informatik-Spektrum, Springer, Vol. 26, Ausgabe 3; 2003. 206–209.

[GI 05] Gesellschaft für Informatik e.V. (GI): Empfehlungen für Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen, 2005.

[IEEE 01] Gerald Engel und Eric Roberts (Hrsg.): IEEE/ACM Computing Curricula 2001 – Computer Science, 2001.

[Mah 05] R. Mahanti und P. K. Mahanti: Software Engineering Education from Indian Perspective. Proceedings of the 18th Conference on Software Engineering Education and Training, 2005. 111–117.

[Man 02] A. Mansingh: Role of Universities in IT Education in India. Journal of AI & Society, Springer, Vol. 16, Issue 1, 2002. 138–147.

[Nar 01] R. Narayanan und S. Neethi: Building Software Engineering Professionals: TCS Experience. Proceedings of the 14th Conference on Software Engineering Education and Training, 2001. 162–171.

[NAS 06] National Association of Software and Service Companies (NASSCOM): Knowledge Professionals in India Factsheet, 2006. *http://www.nasscom.in/*

[Ram 97] K. Ramamritham: Computer Science Research in India. IEEE Computer, Vol. 30, Issue 6, 1997. 40–47.

[Röd 06] H. Röder: Software Engineering Education at University Level in India and Germany. Diplomarbeit. Institut für Softwaretechnologie, Universität Stuttgart, 2006.

Ein verbessertes Softwaretechnikpraktikum: zwischen grüner Wiese und Legacy-Systemen

Björn Axenath · Stefan Henkler

Universität Paderborn, Fachgebiet Softwaretechnik
Warburgerstraße 100, 33098 Paderborn
{axenath|shenkler}@uni-paderborn.de

Zusammenfassung

An der der Universität Paderborn soll Studenten im Grundstudium die Entwicklung von komplexen Softwaresystemen im Rahmen des Softwaretechnikpraktikums vermittelt werden. Dazu führen sie in kleinen Teams gemeinsam einen Entwicklungsprozess durch, um das geforderte Produkt zu erstellen. Aufgrund der geringen Dauer eines Praktikums werden die Schritte des Entwicklungsprozesses i. Allg. nur einmal durchgeführt.

In der Praxis wird Software selten auf der grünen Wiese entwickelt: In der Regel werden existierende Systeme, sog. Legacy-Systeme, erweitert. Für einen Studenten im Grundstudium birgt jedoch schon das Entwickeln von Software ohne die Erschwernis eines Legacy-Systems mannigfache Herausforderungen. Für heutige Projekte ist zusätzlich der Einsatz von Technologien erfolgsentscheidend; auch dieser Aspekt sollte daher im Rahmen eines Praktikums vermittelt werden.

Um Studenten dennoch bereits im Grundstudium an die Methoden heranzuführen, die in der Praxis verlangt werden, und durch Wiederholung der Tätigkeiten ihren Lernerfolg zu verbessern, wurde ein neues Konzept für Aufgabenstellungen erstellt und erprobt. Kern dieses Konzeptes ist es, jedes Team des Softwaretechnikpraktikums mehrere Teilaufgaben bearbeiten und durch die Übernahme von Software aus anderen Teams ein komplexes Softwaresystem entwickeln zu lassen.

1 Einleitung

Das Softwaretechnikpraktikum an der Universität Paderborn, eine Veranstaltung für Studenten des Studiengangs Informatik oder des Studiengangs Ingenieurinformatik mit dem Schwerkunkt Informatik im ersten Studienabschnitt, hat das Ziel, die Entwicklung komplexer Systeme zu vermitteln. Hierbei sollen UML und Java verwendet werden. Diese Aufgabe soll in Teams mit jeweils ca. zehn Studenten durchgeführt werden. Da es sich bei industrieller Softwareentwicklung nur selten um Neuentwicklungen handelt, wird im Paderborner Softwaretechnikpraktikum seit mehreren Jahren dabei ein Reengineering eines Legacy-Systems durchgeführt (vgl. [Bot01]).

In den letzten Jahren wurde den Studenten der undokumentierte Code eines komplexen, verteilten Systems gegeben. Die Studenten mussten zunächst ein Reverse Engineering durchführen, um die Software zu verstehen. Erst danach konnte mit dem Forward Engineering begonnen werden.

Für einen Großteil der Studenten war dies die erste Begegnung mit einem großen Softwaresystem. Da es sich um ein Legacy-System handelte, war die Qualität bewusst niedrig gehalten. Die Erkenntnis für die Studenten beschränkte sich deshalb oft darauf, dass sie den Bedarf an Entwicklungsdokumentation für Softwaresysteme erkannten. Es war ihnen nicht möglich, an diesem System zu erkennen, ob es sich um gute oder schlechte Architektur bzw. gutes oder schlechtes Design handelte. Folglich übernahmen sie schechte Entwurfsmuster [Gam95] aus dem Legacy-System in ihre eigene Entwicklung.

Ein weiteres Problem der ursprünglichen Aufgabenstellung war das Vorgehensmodell. Wie viele andere Praktika [Dem99] lag dem Paderborner Softwaretechnikpraktikum das Wasserfallvorgehensmodell zugrunde, das es den Studenten nicht ermöglicht, aus Fehlern zu lernen und sofort das Gelernte erneut anzuwenden. Ziel war es, mit der neuen Konzeption des Praktikums unter den gegebenen Umständen ein iteratives Prozessmodell einzuführen, wie es auch bereits für die Universtität Dortmund beschrieben wurde (vgl. [Kop04]).

Bei dem bisherigen System wurde als einzige Technologie RMI [RMI06] eingesetzt. Unserer Meinung nach ist diese Technologie zu einfach im Vergleich zu den Technologien, die in den uns bekannten Industrieprojekten eingesetzt werden. Der effektive und effiziente Umgang mit Technologien, d.h. der Einsatz von Werkzeugen und Rahmenwerken zur Erstellung eines Produkts, ist unserer Meinung nach ein entscheidender Erfolgsfaktor für heutige Softwareprojekte. Um diesen Anfoderungen später gerecht werden zu können, müssen Studenten z.B. lernen, wie man sich zeitökonomisch in Technologien einarbeitet.

Diese Missstände veranlassten uns, ein neues Konzept für die Aufgabenstellungen des Softwaretechnikpraktikums zu erarbeiten. Dabei war es vorteilhaft, dass das Softwaretechnikpraktukum in den letzten Jahren von unserem Lehrstuhl betreut wurde. In den folgenden Unterabschnitten beschreiben wir die Lernziele

des Praktikums, gefolgt von den Rahmenbedingungen, die wir für die Gestaltung des Softwaretechnikpraktikums haben. In Abschnitt 2 beschreiben wir das neue Konzept der Aufgabenstellung. Die von uns gemachten Beobachtungen werden in Abschnitt 3 evaluiert. Zuletzt, in Abschnitt 4, ziehen wir ein Fazit aus dem in diesem Jahr erstmals nach dem neuen Konzept verlaufenen Praktikum.

1.1 Lernziele des Paderborner Softwaretechnikpraktikums

Wie oben bereits ausgeführt, soll das Softwaretechnikpraktikum die Entwicklung komplexer Systeme vermitteln. Dazu sollte wie bereits in den letzten Jahren auch das Reengineering behandelt werden. In diesem Jahr sollen die Themen Entwicklungsprozesse, Projektmanagement und Technologieeinsatz besser vermittelt werden. Selbstverständlich werden auch die Themen Konfigurationsmanagement, Qualitätssicherung und Soft Skills behandelt.

Der Entwicklungsprozess soll ein geplantes Vorgehen bei der Softwareentwicklung vermitteln. Hierbei ist das affektive Lernziel die Erstellung von Lastenheft, Pflichtenheft, Analyse- und Entwurfsdokumentation sowie eines Testberichts. Im Rahmen des Projektmanagements sollen dazu Verantwortlichkeiten festlegt, Aktivitäten geplant sowie ein Controlling der Arbeitsstunden durchgeführt werden, sodass die Studenten ihr prozessbasiertes Vorgehen reflektieren können.

Im Umgang mit Technologien liegt der Schwerpunkt auf den kognitiven Lernzielen. Die Studenten sollen sich beim Einsatz von Technologien über die Vor- und Nachteile bewusst sein, die in Bezug zu den anderen Lernzielen stehen. Aus Sicht des Entwicklungsprozesses muss z.B. eine Einarbeitungsphase aufgenommen werden, die auch vom Projektmanagement berücksichtigt werden muss. Die Qualitätssicherung muss den Einfluss der Qualität der eingesetzten Technologie auf das gesamte zu entwickelnde Produkt berücksichtigen.

Im Bereich des Reengineering ist folgendes Lernziel zu erreichen: Die Studenten sollen aus dem Reverse Engineering für das folgende Forward Engineering Schlüsse ziehen. Im affektiven Bereich sollen sie den Nutzen der Reverse-Engineering-Modelle wahrnehmen und daraufhin die Qualität der eigenen Modellierung verbessern.

1.2 Voraussetzungen des Softwaretechnikpraktikums

Die Studenten haben u.a. die folgenden relevanten Vorkenntnisse im Programmieren im Kleinen am Beispiel von Java erworben sowie im Einsatz der UML. Eine detaillierte Beschreibung der vorausgesetzten Kenntnisse findet sich im Modulhandbuch des Studiengangs [Upb06].

Im Sommersemester 2006 hatten sich ca. 150 Studenten zum Softwaretechnikpraktikum angemeldet, sodass 16 Teams mit 8 bis 10 Teilnehmern gebildet werden konnten. Die Arbeitslast ist durch 10 ECTS Punkte definiert.

Für die Betreuung des Praktikums wurden vier halbe Stellen wissenschaftlicher Mitarbeiter und 14 halbe SHK-Stellen zur Verfügung gestellt. Jedem Team stand ein Betreuer zur Seite, der an den wöchentlichen Teamsitzungen teilnahm. Während der Woche waren die Betreuer per E-Mail zu erreichen und sollten auf E-Mails innerhalb von 24 Stunden antworten. Die Betreuer mussten zwei Rollen einnehmen. Zum einen sollten sie die Teams inhaltlich unterstützen, indem sie Fragen beantworteten, aber keine Vorgaben machten. Zum anderen sollten sie die Teams kontrollieren und die Organisatoren des Praktikums über mögliche Probleme informieren. Darüber hinaus stand allen Teams eine Programmierberatung zur Verfügung, die ebenfalls eine wöchentliche Sprechstunde hatte und per E-Mail kontaktiert werden konnte. Fragen zur Aufgabenstellung wurden von den Organisatoren stellvertretend für einen virtuellen Kunden per E-Mail beantwortet. Die Betreuer der Teams trafen sich mindestens einmal wöchentlich, um auf die nächste Woche vorbereitet zu werden und über den Stand ihrer Teams zu berichten.

2 Die Konzeption des Praktikums

Das Praktikum wird hauptsächlich durch die Aufgabenstellung geprägt. In diesem Jahr wurde beschlossen, neben der Vorlesung und den wöchentlichen Treffen der Teams weitere untestützende Veranstaltungen anzubieten.

Zunächst beschreiben und erläutern wir die Konzeption der Aufgabenstellung, die sich in das zu entwickelnde Produkt und das Vorgehen aufteilt. Dabei begründen wir auch die konkrete Auswahl der eingesetzten Technologien, da diese zur Machbarkeit des Praktikums beigetragen haben. Abschließend geben wir an, welche Unterstützung die Studenten bei der Durchführung des Praktikums erfahren haben.

2.1 Eine integrierte Entwurfsumgebung

Jedes Team des Softwaretechnikpraktikums hatte die Aufgabe, eine Entwurfsumgebung für eingebettete Systeme zu entwickeln. Als Anwendungsbereich wählten wir automotive Systeme. Die Entwurfsumgebung sollte aus drei Werkzeugen bestehen: einem Komponenteneditor, einem Verteilungseditor und einem Diagnosewerkzeug. Mit dem Komponenteneditor sollten Software-Komponenten eines eingebetteten Systems durch ihre Schnittstelle und ihr Verhalten beschrieben werden. Mit dem Verteilungseditor sollte ein Netzwerk aus Rechnern erstellt werden können, auf dem die zuvor definierten Komponenten instanziiert werden können. Das Diagnosewerkzeug sollte das System ausführen und zur Diagnose über-

wachen können. Alle drei Komponenten sollten eine grafische Benutzeroberfläche haben. Die Teams mussten nur eine der drei Komponenten selber entwickeln, die anderen beiden Komponenten sollten von anderen Teams übernommen werden. Die vollständige Aufgabenstellung findet man unter [Swt06].

Als Datenmodell haben wir den Studenden Code zur Verfügung gestellt, der aus EMF-Modellen [Emf06] generiert wurde. Eine Besonderheit dieses Codes ist, dass die Daten durch Serialisierung in XMI persistent gemacht werden können. Dadurch reduziert sich der Implementierungsaufwand einer Applikation deutlich.

Das Modell enthielt keine Funktionalität zur Ausführung des Systems. Die Ausführungssemantik war textuell in der Aufgabenstellung beschrieben und musste von den Studenten implementiert werden. Im Folgenden bezeichnen wir diesen Code als Simulationsalgorithmus. In der Entwurfsumgebung ist er Teil des Diagnosewerkzeugs.

Durch den Anwendungsbereich werden den Studenten implizit elementare Fragestellungen aus dem Bereich des Software Engineering für eingebettete Systeme vermittelt, die Forschungsschwerpunkt der Universität Paderborn sind. Des Weiteren verfolgen wir, wie bereits in den Aufgabenstellungen der letzten Jahre, das Ziel, die Synergieeffekte aus kombinierter Forschung und Lehre zu nutzen [Geh02, Geh03].

Als Rahmenwerk für die Werkzeuge und deren Integration haben wir die Eclipse-Plattform vorgegeben [Ecl06]. Diese ermöglicht es, auf Grundlage einer Plug-in-basierten Architektur effizient eigene Werkzeuge zu erstellen, da sie Grundfunktionalitäten, wie z.B. das Verwalten von Konfigurationsdaten, bereits anbietet. Des Weiteren sind dort bereits viele Entwurfsmuster [Gam95], wie z.B. das Command Pattern für die Undo/Redo-Funktionalität vorhanden, sodass die Studenten einen qualitativ hochwertigen Code analysieren und daraus lernen können. Dies ermöglicht es den Betreuern der Teams, eine Binnendifferenzierung vorzunehmen, indem im Rahmen der Teamsitzung z.B. Entwurfsmuster behandelt werden können, falls das Team dieses Niveau erreicht hat. Dadurch wird ein iterativer Lernprozess ermöglicht, da diese Themen im folgenden Studienabschnitt vertieft werden können.

Die grafische Benutzeroberfläche sollte mit GEF (Graphical Editing Framework) erstellt werden [Gef06]. Dieses Rahmenwerk verfolgt die Model-View-Controller-Architektur und schafft somit ein Bewusstsein für Softwarearchitekturen. Die Kenntnis über Eclipse und GEF soll es den Studenten erleichtern, Software im Rahmen von Bachelor- und Masterarbeiten zu entwicklen.

2.2 Ablauf des Praktikums

Die wesentliche Neuerung des Softwaretechnikpraktikums bestand darin, dass die Aufgabe sich aus mehreren Teilaufgaben zusammensetzt. Für jede Teilaufgabe

wird ein vollständiger Entwicklungsprozess mit Lastenheft, Pflichtenheft, Analyse und Entwurf, Implementierung und Test durchgeführt (siehe Abb. 1). Diese Entwicklungsprozesse starten im Abstand von einigen Wochen. Zuerst wurde das eigene Werkzeug entwickelt, das erst später mit den beiden erworbenen Werkzeugen zu einer Entwurfsumgebung integriert wurde. Durch diese Aufteilung können wir unsere Ziele erfüllen.

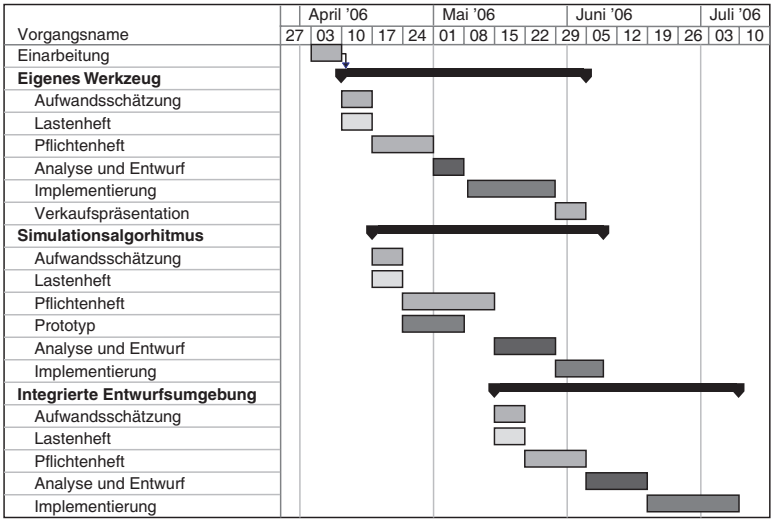


Abb. 1 *Ausschnitt aus der Projektplanskizze*

Die erste Teilaufgabe war das Erstellen eines eigenen Werkzeugs. Die zu erstellenden Werkzeuge wurden anhand der Teamgröße und Schwierigkeitsgrad des Werkzeugs auf die Teams verteilt. Den Teams wurden spezielle Rahmenwerke für die grafische Benutzeroberfläche und das Grundgerüst der Applikation vorgegeben. Folglich mussten sie sich zunächst in eine ausgereifte Technologie einarbeiten.

Da der Standardprozess mehrfach durchlaufen wurde, konnten die Studenten das Erstellen der Dokumente mehrfach üben. Bevor im nächsten Zyklus ein Dokument erstellt wurde, haben wir eine korrigierte Version des vorher erstellten Dokuments gleichen Typs zurückgegeben. Die Aufgabenstellung sah vor, dass neben der Erstellung des eigenen Werkzeugs und der Integration der Entwurfsumgebung auch die Ausführungssemantik durch einen Simulationsalgorithmus implementiert wurde. Die Dokumente zum eigenen Werkzeug und der Implementierung waren sich ähnlich, weil sie beide auf die Benutzungoberfläche ausgerichtet waren. Im Gegensatz hierzu war für den Simulationsalgorithmus keine Oberflächenbeschreibung notwendig. Diese unterschiedlichen Ausrichtungen der zu spezifizierenden Werkzeuge erforderten, dass die Dokumente nicht stereotyp geschrieben werden konnten und die Studenten sich über deren Zweck jedesmal erneut bewusst werden mussten.

Durch die Integration des eigenen Werkzeugs mit zwei fremden Werkzeugen zu einer einheitlichen Entwurfsumgebung sollen die Grundzüge des Reengineering vermittelt werden. Das Reengineering war für die Integration notwendig, weil trotz Aufgabenstellung mit einem Metamodell und vorgegebenem Rahmenwerk noch einige Anorderungen offen gelassen waren. Die Aufgabenstellung sah nicht explizit vor, eine Konsistenzprüfung für die Editoren zu erstellen. Beispielsweise erlaubte das Modell, eine Kompontente zu instanziieren, ohne ihr dabei einen Knoten zuzuweisen. Diese Eigenschaft konnte von dem Verteilungseditor oder dem Diagnosewerkzeug geprüft werden. Zudem hatten wir den Studenten empfohlen, dass die integrierte Entwurfsumgebung z.B. ein einheitliches Aussehen haben sollte. Dies bezog sich nicht nur auf Farben und Formen, sondern auch auf die funktionalen Aspekte des Layouts. Beispielsweise bietet das Applikationsrahmenwerk ein Fenster für Fehler oder Warnungen an, das anfänglich nur von einigen Teams genutzt wurde, während andere eine eigene Ausgabe für ihre Warnmeldungen entwickelten.

Wie bereits in der Einleitung erwähnt, ist das Reengineering durch die in der Praxis häufig vorkommenden Arbeiten mit Lagacy-Systemen motiviert. Die in diesem Praktikum konstruierte Situation deckt nicht alle Eigenschaften eines Legacy-Systems ab. Insbesondere entspricht die Größe der übernommenen Komponenten nicht der Größe realer Systeme, und die vorgebene Systemarchitektur ist nicht veraltet. Die Studenten mussten folglich nur den Umgang mit einem unzureichend dokumentierten System erfahren.

Für die Entwicklung des eigenen Werkzeugs und des Simulationsalgorithmus ist es notwendig, sich in die vorgegebenen Rahmenwerke und das Metamodell einzuarbeiten. Damit der Einarbeitungsaufwand in die Technologien rechtzeitig wahrgenommen wird, haben wir in der Projektplanskizze in der ersten Woche eine Einarbeitungszeit vorgesehen. Zur Unterstützung dieser Aufgabe haben wir auf der Homepage der Veranstaltung einen einfachen Editor veröffentlicht, der auf einem ähnlichen Modell basiert. Um die Semantik des Modells besser zu verstehen, haben wir verlangt, dass bereits während der Entwicklung des Pflichtenhefts für den Simulationsalgorithmus ein Prototyp erstellt wird.

2.3 Vorlesung, Tutorien und Betreuung

Ziel der Vorlesung ist es, die Konzepte des Software Engineering, wie z.B. Qualitätssicherung oder Konfigurationsmanagement, zu vermitteln. Übungen, in denen diese Konzepte angewendet werden, sind nicht vorgesehen. Wir haben dieses Jahr Tutorien zu unterschiedlichen Fragestellungen, wie z.B. Qualitätssicherung, Konfigurationsmanagement, Eclipse/GEF etc., angeboten, in denen wir die Studenten in die praktische Anwendung der Konzepte eingeführt haben. Aus jedem Team durfte abwechselnd ein Mitglied teilnehmen. Jedes Teammitglied hatte ein Rolle, z.B. Qualitätsmanagementbeauftragter, die seine Verantwortlich-

keit definierte. Durch die Tutorien konnten sich die Teammitglieder schulen lassen und waren dann u.a. dafür verantwortlich, das Gelernte in dem Team zu verbreiten.

2.4 Motivation der Studenten

Bei der Konzeption des Praktikums spielte auch die Motivation eine zentrale Rolle. Wir diskutieren diesen Aspekt des Praktikums, weil durch die motivationssteigernden Maßnahmen den Studenten die Möglichkeit zur Selbstreflexion gegeben wurde.

Nach Fertigstellung des eigenen Werkzeugs haben wir eine Produktmesse veranstaltet. Die Teams präsentierten dazu ihre Produkte an kleinen Messeständen. Während der Produktmesse konnten die Teams ihre Arbeit evaluieren, um so eine Auswahl für ihre integrierte Entwurfsumgebung zu treffen. Die Betreuer des Softwaretechnikpraktikums bewerteten ebenfalls die Produkte. Auf Basis dieser Bewertung wurde entschieden, welches Team welche anderen Werkzeuge einkaufen durfte.

Bei einer Produktmesse können neben der technischen Begutachtung auch die Soft Skills erkannt und bewertet werden. Da wir keine Vorgaben über die Art der Messestände gemacht haben, waren die Studenten gezwungen, kreativ zu werden, und konnten auf der Produktmesse ihre Präsentationsideen direkt mit den Ideen der Konkurrenten vergleichen.

Am Ende der Vorlesungszeit wurde ein Modellierungswettbewerb veranstaltet, bei dem jeweils zwei Teams gegeneinander die Software eines mechatronischen Systems modellieren oder analysieren mussten.

Die Industrierelevanz der Aufgabe wurde durch zwei Maßnahmen verdeutlicht. Zum einen dürfen die besten drei Teams ihr Produkt vor Vertretern der Automobilindustrie präsentieren und zum zweiten wurde im Rahmen der Vorlesung ein Vortrag eines Mitarbeiters aus der Softwareindustrie gehalten.

3 Evaluation des Praktikums

Die Evaluation des Praktikums erfolgt durch die Ermittlung des Lernerfolgs der Studenten, den Vergleich der Ergebnisse des diesjährigen Praktikums mit vorherigen und die Reflexion des Organisationsaufwands. Im Folgenden werden wir zuerst auf den Lernerfolg der Studenten eingehen. Dazu spezifizieren wir, was wir unter Lernerfolg verstehen, und gehen hierbei auf Beobachtungspunkte ein. Dann werden wir in Abschnitt 3.1 bis Abschnitt 3.7 den Lernerfolg der Studenten bei den gesetzten Zielen erläutern. Abschließend werden wir in Abschnitt 3.8 auf einen Vergleich mit den vorherigen Softwaretechnikpraktika eingehen und hierbei konkret die Vor- und Nachteile reflektieren. Zudem werden wir auf den Organisationsaufwand eingehen.

Den Lernerfolg des Praktikums messen wir durch Beobachtung der gesetzten Ziele (siehe Abschnitt 1.1). Um dies zu ermöglichen haben wir diverse Instrumentierungen des Praktikums vorgenommen, um die Beobachtbarkeit zu erhöhen. Neben den klassischen Beobachtungspunkten, wie Klausur und Endabgabe der Projektarbeit, haben wir zusätzlich folgende Instrumentierungen vorgenommen:

- Feedbackschleifen: Jedes Team wurde aufgefordert, zu jedem Vorlesungstermin Fragen an den Veranstalter zu stellen.
- Tutorien: Neben den Feedbackschleifen wurden zu den Vorlesungsthemen Tutorien gehalten. Für die verschiedenen Themen wurden Verantwortliche in den Teams bestimmt, die entsprechend die Tutorien besuchen mussten.
- Wöchentliche Tutorentreffen: Feedback über den Stand der Teams durch die Tutoren.
- Abgaben: Dokumente wie Pflichtenheft oder Analyse und Entwurf wurden kontrolliert.
- Präsentationen: Neben der Abschlusspräsentation gab es noch bis zu vier weitere Präsentationen. Hier wurden gezielt unterschiedliche Rollen (Käufer, Lehrender, Fachmann) durchgespielt, um die verschiedenen Anforderungen, wie Soft Skills oder Technologieverständnis, zu kontrollieren.
- Feedbackbögen: Bei der Endabgabe zum Softwaretechnikpraktikum wurden zusätzlich zu den Dokumenten und der erstellten Software ein Feedbackbogen und ein eigenes Resümee eingefordert. Das Resümee teilt sich in ein Grupenresümee und ein Resümee einzelner Personen auf. Der Feedbackbogen besteht aus den Kategorien Teamarbeit, Projektmanagement, Unterstützung durch die Veranstaltung, allgemeiner Lernerfolg, Lernerfolg bei Tätigkeiten und Lernerfolg bei Technologien.

Eine interessante Beobachtung bei der Auswertung der Resümees und Feedbackbögen ist, dass die Selbsteinschätzung aus dem Team oder einer Person in den meisten Fällen mit der Einschätzung des Veranstalters und der Betreuer korrelieren.

Im Folgenden werden wir auf die verschiedenen Ziele des Praktikums eingehen und mit Hilfe der Beobachtungspunkte den Lernerfolg des Praktikums aufzeigen.

3.1 Entwicklungsprozesse

Um den Lernerfolg des Entwicklungsprozesses zu beschreiben, müssen die verschiedenen Phasen des Entwicklungsprozesses beobachtbar sein. Dies haben wir zum einen durch Abgaben der verschiedenen Dokumente und deren Bewertung sowie durch Präsentationen ermittelt. Durch die drei parallelen Prozesse, die versetzt zueinander durchlaufen wurden, hatten wir zudem die Möglichkeit, die Abgaben zueinander in Vergleich zu stellen. Dies hat uns ermöglicht, einen konti-

nuierlichen Lernerfolg zu beobachten. Der Vergleich der Abgaben der parallelen Prozesse lässt stark darauf schließen, dass die Studenten einen Lernerfolg durch die Iteration durchlebt haben. Während beim ersten Durchlauf häufig Abhängigkeiten zwischen unterschiedlichen, aufeinanderfolgenden Phasen nur schwach erkannt und entsprechend nur kaum erkennbar in den Dokumenten zu sehen waren, war in dem zuletzt beginnenden Prozess eine deutliche Verbesserung zu erkennen. Die Qualität der Dokumente hat sich wie erwartet verbessert.

3.2 Technologien

Den Lernerfolg bezüglich der Technologien lässt sich mit Hilfe des entwickelten Produktes feststellen.

Alle Teams haben eine integrierte Entwufsumgebung mit den geforderten Funktionen und Eigenschaften abgegeben. Mit der Vorgabe des Metamodells waren alle Teams in der Lage, ihr Werkzeug zu entwickeln und es in die geforderte Entwurfsumgebung zu integrieren. Die Mehrzahl der Produkte übertraf in Bezug auf den Funktionsumfang und die Stabilität unsere Erwartungen.

Letztendlich hatten zwei von 16 Teams zu der gegebenen Deadline für das eigene Werkzeug noch keine lauffähige Version. Eine Woche später konnte eines der Teams sein Werkzeug nachliefern. Damit konnte dieses Team rechtzeitig bis zur Produktmesse sein Werkzeug fertigstellen und dort präsentieren. Das andere Team musste mit einem fremden Werkzeug das Praktikum fortsetzen.

Weiterhin konnten wir den Lernerfolg in Bezug auf die eingesetzten Technologien durch Zwischenpräsentationen beobachten. Hierbei wurde ein sehr gutes Verständnis der eingesetzten Technologien festgestellt. Dies wurde letztendlich durch die Feedbackbögen am Ende der Veranstaltung bestätigt. Alle Teams hatten den Lernerfolg bezüglich der Technologien im Mittel mit gut bis sehr gut eingestuft. Eine positive Tendenz ließ sich ebenfalls durch die wöchentlichen Treffen mit den Tutoren ziehen.

3.3 Reengineering

Etwa nach der Hälfte des Praktikums mussten die Teams aus ihrem selbst entworfenen Plug-in und mit zwei weiteren Plug-ins eines anderen Typs eine integrierte Entwurfsumgebung entwickeln. Wir haben hierbei sehr hohen Wert auf gleiche Benutzbarkeit und Funktionalität (z.B. Konsistenzmanagement einheitlich durch restriktive Eingaben, einheitlich durch Fehlermeldungen etc.) der verschiedenen Editoren gelegt. Damit die Studenten diese Aufgabe erfolgreich bestehen, mussten sie mit den eingekauften Produkten ein Reengineering durchführen. Die entsprechenden Dokumentabgaben hierzu waren im Durchschnitt nur befriedigend. Da das Reengineering nur in einem Prozess durchgeführt wurde, kann eine Verbesserung über den Zeitraum des Praktikums nicht festgestellt werden.

Trotz der durchschnittlichen Abgaben wurden in dem Endprodukt bis auf wenige Ausnahmen die geforderten Eigenschaften umgesetzt. Der selbstbewertete Lernerfolg der Studenten wurde im Schnitt mit gut angegeben.

3.4 Projektmanagement

Beobachtungspunkte für das Projektmanagement waren die Abgaben der Projektpläne und die wöchentlichen Treffen mit den Tutoren. Aus beiden lässt sich durch die parallelen Prozesse erkennen, dass hier eine kontinuierliche Verbesserung stattgefunden hat. Während anfänglich überwiegend alle Teams nur rechtzeitig die Deadlines der Abgaben einhalten konnten, hatte sich im Lauf der Zeit ein verbessertes Management der zu erledigten Arbeiten eingestellt, wodurch sogar die Abgaben einige Tage vor der Deadline erfolgt sind. Die Aufgabe, dass jeder Teilnehmer während des Praktikums möglichst alle Tätigkeiten durchlaufen hat, wurde in der Regel erfüllt. Weiterhin ist positiv zu bewerten, dass die meisten Teams in der Lage waren, sich geeignet für mehrere parallele Aufgaben aufzuteilen, und hierbei auch einen Projektplan erstellt haben. Das Feedback der Teams war zum Projektmanagement gut: Die Teamarbeit war für die Studenten eine gute Erfahrung und ein guter Lernerfolg. Entsprechend war das Projektmanagement für die meisten Teams positiv und ein guter Lernerfolg. Einige Teams merkten in den Feedbackbögen zu knappe Deadlines an. Da diese Termine aber von anderen Teams unterschritten wurden, ist dieser Kritikpunkt eher weniger stark zu gewichten.

3.5 Inhalte und Anwendung der Vorlesung

Um den Lernerfolg bei diesem Thema festzustellen nutzen wir die Beobachtungspunkte Klausur, Feedbackschleifen, Tutorien und Feedbackbögen. Die Klausur wurde im Vergleich zum letzten Jahr besser abgeschlossen. Die Feedbackschleifen zur Vorlesung wurden von den Studenten noch nicht im gewünschten Rahmen genutzt. Hier muss offenbar noch vermittelt werden, welche Vorteile diese Feedbackschleifen für die Umsetzung der Vorlesungstheorie in die Praxis haben. In den Tutorien, wo Vorlesungsstoff praktisch angewandt wurde, sind überwiegend keine Diskussionen aufgekommen. Die Tutoren berichteten zum Teil, dass die Inhalte und Anwendung der Tutorien, eine Woche nach der dazugehörigen Vorlesung, nicht immer präsent bei den Studenten waren. Hier wird erneut deutlich, dass die Feedbackschleifen zur Vorlesung stärker ins Bewusstsein der Studenten gerückt werden müssen. Trotz dieser Schwierigkeiten ist in den Abgaben und Produkten sowie durch die Kontrollen der Teambetreuer deutlich geworden, dass die Inhalte von den Studenten umgesetzt werden konnten. Hierzu passen dann auch die überwiegend positiven Feedbackbögen.

Der allgemeine Lernerfolg wurde durch die Studenten mit gut bewertet. Durch die Resümees wurde auch deutlich, dass der Vorlesungsstoff interessant und hilfreich war.

3.6 Qualitätssicherung

Zur Qualitätssicherung wurden Reviews und Inspektionen für die Dokumenterstellung (Lastenheft, Pflichtenheft, Analyse und Entwurf) verlangt. Während der Implementierungsphase wurden Reviews und JUnit-Tests mit Überdeckungsbericht mit Hilfe von Emma zur Qualitätssicherung eingesetzt. Die Reviews und Inspektionen wurden innerhalb der Teambesprechung thematisiert bzw. durchgeführt. Hier lässt sich ähnlich wie bei den anderen Tätigkeiten, die mehrfach aufgrund der parallelen Prozesse durchgeführt wurden, eine Verbesserung erkennen.

3.7 Soft Skills

Hierunter betrachten wir, ob und inwiefern Studenten teamfähig sind und wie sie sich in verschiedenen Rollen, wie zum Beispiel in der eines Verkäufers, präsentieren können. Um die Teamfähigkeit zu bewerten, haben wir die Möglichkeit genutzt, entsprechende Informationen von den Tutoren zu erfragen. Weiterhin haben wir in den Fragebögen bei der Endabgabe diesen Punkt sowohl in dem Bereich des Team- als auch im Bereich des Einzelfeedbacks erhoben. Bis auf zwei Teams kann hierbei von einem positiven Lernerfolg gesprochen werden. Hierbei haben die Berichte der Tutoren mit den Fragebögen korreliert. Bei den zwei Teams, in denen die Fähigkeiten im Bereich der Soft Skills negativ beurteilt wurden, ist ebenfalls eine Korrelation mit dem erstellten Produkt zu erkennen.

3.8 Vergleich mit vorherigen Praktika

Ziel der Umstellung ist, das bisherige Praktikum zu verbessern. Berücksichtigt man die Vorgaben im Bereich der Lernziele und die gewählten Evaluationsmaßnahmen, ist festzustellen, dass tatsächlich eine Verbesserung erzielt werden konnte. Die parallelen Prozesse haben sich sehr positiv auf den Lernerfolg für zentrale Themen der Softwaretechnik, wie beispielsweise systematisches Vorgehen vom Lastenheft bis hin zur Implementierung, ausgewirkt.

Der Organisationsaufwand im Vergleich zu den vorherigen Jahren ist gestiegen. Durch die parallelen Prozesse sind zusätzliche Abgaben angefallen, die kontrolliert werden mussten, und zudem ist der Aufwand, die parallelen Prozesse zu koordinieren, größer geworden.

Der Korrekturaufwand ist gestiegen, da nun die dreifache Anzahl von Dokumenten geprüft werden musste. Da der Umfang der Dokumente deutlich geringer war als im vorherigen Jahr, relativierte sich dieser Aufwand.

4 Fazit

Um die Ziele des diesjährigen Softwaretechnikpraktikums zu erreichen haben wir drei parallele Entwurfsprozesse eingeführt, die versetzt zueinander gestartet wurden. Wie in den vorherigen Kapiteln gezeigt, ist dies eine gute Entscheidung, da hierdurch ein größerer Lernerfolg bei den Studenten erreicht wird. Die zusätzliche Herausforderung, eine komplexe Technologie zu erlernen, gelingt ebenfalls. Es hat sich allerdings herausgestellt, dass beim Reengineering ein geringerer Lernerfolg erreicht wurde, der aber noch akzeptabel ist. Aufgrund der neuen Aufgabenstellung war es nicht möglich, von Anfang an ein geeignetes Legacy-System bereitzustellen. Für die folgenden Praktika stellen die diesjährigen Entwurfsumgebungen eine gute Grundlage für ein anspruchsvolleres Reverse Engineering dar.

Bezüglich des Arbeitsaufwands der Betreuung lässt sich abschließend folgern, dass dieser nicht proportional mit den parallelen Prozessen gestiegen ist. Im Verhältnis zu den erreichten Zielen ist der erhöhte Aufwand vertretbar.

Der Erfolg des diesjährigen Praktikums hat uns dazu bewogen, auf Basis dieser Konzeption das Folgepraktikum zu gestalten.

Literatur

[Bot01] Klaus Bothe, Ulrich Sacklowski: Praxisnähe durch Reverse Engineering-Projekte: Erfahrungen und Verallgemeinerungen. Software Engineering in den Hochschulen, SEUH 7, dpunkt, 2001.

[Dem99] Birgit Demuth, Heinrich Hussmann, Lothar Schmitz, Steffen Zschaler: Erfahrungen mit einem frameworkbasierten Softwarepraktikum. In: Tagungsband des 6. Workshops Software-Engineering im Unterricht der Hochschulen, Teubner-Verlag, 1999.

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Ecl06] Eclipse Projekt Homepage, *http://www.eclipse.org*.

[Emf06] Eclipse Modeling Framework Project Homepage, *http://www.eclipse.org/emf*.

[Gef06] GEF Projekt Homepage, *http://www.eclipse.org/gef*.

[Geh02] M. Gehrke, H. Giese, U.A. Nickel, J. Niere, M. Tichy, J.P. Wadsack, A. Zündorf: Reporting about Industrial Strength Software Engineering Courses for Undergraduates. In: Proc. of the 4th International Conference on Software Engineering (ICSE), Orlando, Florida, USA, 95–405, 2002.

[Geh03] Matthias Gehrke, Holger Giese, Ekkart Kindler, Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals: Software Engineering Education: The Synergy of Combined Research and Teaching, Tech. Rep. tr-ri-03-237, University of Paderborn, Paderborn, January 2003.

[Kop04] Corina Kopka, Doris Schmedding, Jens Schröder: Der Unified Process im Grundstudium – Didaktische Konzeption, von Lernmodulen und Erfahrungen. DeLFI 2004: 127-138.

[RMI06] Remote Method Invocation, *http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp*.

[Upb06] Modulhandbuch Bachelor-Master-Studienprogrammm Informatik. Fakultät für Elektrotechnik, Informatik und Mathematik an der Universität Paderborn, Juni 2006, *http://wwwcs.uni-paderborn.de/cs/studium/material/modulhandbuch.pdf*, 2006.

[Swt06] Homepage Softwaretechnikpraktikum, Universität Paderborn, Fachgebiet Softwaretechnik: *http://ag-schaefer.upb.de/Lehre/Lehrveranstaltungen/Praktika/Softwaretechnikpraktikum/SS06*.

Positive Effekte von Szenarien und Features in einem Softwarepraktikum[1]

Kim Lauenroth[1] · *Ernst Sikora*[1, 2] · *Klaus Pohl*[1, 2]

1. Software Systems Engineering
   Institute for Computer Science and Business Information Systems (ICB)
   University of Duisburg-Essen, 45117 Essen
   {kim.lauenroth|ernst.sikora|klaus.pohl}@sse.uni-due.de
2. Lero – The Irish Software Engineering Research Centre
   University of Limerick, Limerick, Ireland
   {ernst.sikora|pohl}@lero.ie

Zusammenfassung

In der Bachelor-Veranstaltung Softwareentwicklung und Programmierung (SEP) führen die Teilnehmer die wesentlichen Aktivitäten der Softwareentwicklung vom Requirements Engineering über die Implementierung bis zum Softwaretesten durch.

In diesem Beitrag beschreiben wir die von uns beobachteten Probleme bei der Durchführung der Softwareentwicklungsaktivitäten durch die Studierenden und präsentieren einen szenario- und featurebasierten Ansatz zur Verminderung dieser Probleme. Wir erläutern die positiven Effekte von Szenarien und Features auf das SEP und belegen die Akzeptanz von Szenarien und Features anhand einer Umfrage unter den Teilnehmern.

1 Einleitung

Die Veranstaltung Softwareentwicklung und Programmierung (SEP) ist ein Bestandteil der praktischen Ausbildung in den Studiengängen Angewandte Informatik – Systems Engineering, Wirtschaftsinformatik, Mathematical Engineering und Lehramt Informatik an der Universität Duisburg-Essen. Das SEP hat pro

1 Das Verfassen dieser Arbeit wurde teilweise gefördert durch SFI Grant No. 03/CE2/I303_1.

Semester ca. 150 bis 250 Teilnehmer. Das SEP verfolgt drei wesentliche Zielsetzungen:

- Die Vertiefung von Programmierkenntnissen
- Die Entwicklung von Soft Skills (z.B. teamorientierte Arbeitsweise oder die angemessene Präsentation von Arbeitsergebnissen)
- Die Vermittlung eines methodischen Vorgehens zur Entwicklung von Software

Über vergleichbare Zielsetzungen in einem Softwarepraktikum wurde bereits berichtet, vgl. z.B. [Göh 05]. Schwerpunkt dieses Beitrags ist die Umsetzung des dritten Ziels, d.h. die Vermittlung methodischer Vorgehensweisen zur Softwareentwicklung.

In der industriellen Praxis hat die Bedeutung methodischer Vorgehensweisen zur Entwicklung von Software in den letzten Jahren erheblich zugenommen. Themen wie Requirements Engineering, Softwareentwurf und Softwaretesten spielen in IT-Unternehmen eine zentrale Rolle. Empirische Untersuchungen schreiben dabei insbesondere dem Requirements Engineering eine Schlüsselrolle für den Erfolg oder Misserfolg von Softwareprojekten zu (vgl. z.B. [Ha 02; Stan 04]).

Im SEP werden alle wesentlichen Aktivitäten der Softwareentwicklung durchgeführt, um den Studierenden die Zusammenhänge zwischen den einzelnen Aktivitäten anhand eines konkreten Beispiels zu verdeutlichen und ihnen somit wichtige Grundlagen der Softwareentwicklung anwendungsorientiert zu vermitteln. Um dies zu erreichen werden beispielsweise die Anforderungen an das zu entwickelnde System den Teilnehmern nicht im Detail vorgegeben, sondern müssen von den Teilnehmern (unter Berücksichtigung der gegebenen Aufgabenstellung) entwickelt werden.

Im Folgenden beschreiben wir den Aufbau des SEPs und legen die von uns beobachteten Probleme bei der Durchführung von Requirements-Engineering-, Entwurfs- und Testaktivitäten in früheren SEP-Veranstaltungen dar. In Abschnitt 2 erläutern wir die Umstellung des SEPs auf ein szenario- und featurebasiertes Vorgehen sowie die dadurch erzielten positiven Effekte. In Abschnitt 3 präsentieren wir die Ergebnisse einer Umfrage unter den Teilnehmern zu der Verwendung von Szenarien und Features im Requirements Engineering.

1.1 Aufbau und Durchführung des SEPs

Die Veranstaltung Softwareentwicklung und Programmierung baut auf einer Vorlesung »Programmierung« mit einer begleitenden Übung auf, in der u.a. die Programmiersprache Java vermittelt wird. Die Veranstaltung besteht aus den Teilen Einstiegsaufgabe (4 Wochen) und Hauptaufgabe (9 Wochen). Die Veranstaltung ist mit 3 ECTS-CP bewertet.

Die Einstiegsaufgabe beinhaltet Programmierübungen sowie eine darauf aufbauende Testaufgabe, mit der die Programmierkenntnisse der Teilnehmer über-

prüft werden. Nur die Teilnehmer, die die Testaufgabe bestehen (ggf. im Wiederholungsversuch), werden zur Teilnahme an der Hauptaufgabe zugelassen. Eine typische Testaufgabe beinhaltet die Ergänzung eines vorgegebenen, den Teilnehmern bekannten Programms um eine zusätzliche Funktionalität.

In der Hauptaufgabe führen Teams von jeweils 6 bis 10 Studierenden, begleitet durch die SEP-Betreuer, einen vorgegebenen Softwareprozess (im Folgenden als SEP-Prozess bezeichnet) durch. Der SEP-Prozess orientiert sich an bekannten Vorgehensmodellen, wie z.B. dem Rational Unified Process [Kru 03] oder dem V-Modell XT [Rau 07]. Da das SEP von Studierenden im 2. Semester belegt wird, können bei den Teilnehmern keine fundierten Kenntnisse von Softwareprozessmodellen, Modellierungstechniken oder CASE-Tools vorausgesetzt werden. Somit kommen für das SEP nur Techniken und Werkzeuge in Frage, die die Teilnehmer nach einer geringen Einarbeitungszeit anwenden können. Um den SEP-Prozess zu etablieren, finden wöchentliche Gruppensitzungen statt. Der Betreuer stellt in einer Gruppensitzung das Ziel für die nächste Woche vor (z.B. Durchführung des Requirements Engineering) und erläutert die zur Zielerreichung benötigten Techniken anhand eines durchgängigen Beispiels. Die Teilnehmer sind dafür verantwortlich, Teilgruppen zu bilden, die die jeweilige Aufgabenstellung bearbeiten, die Arbeitsergebnisse in eine vorgegebene Dokumentationsstruktur integrieren und diese dem Betreuer zur Verfügung stellen. Zur Vorbereitung der folgenden Gruppensitzung unterzieht der Betreuer die Ergebnisse einem Review. Die Teilnehmer werden in der folgenden Gruppensitzung über die identifizierten Fehler und die Verbesserungsvorschläge des Betreuers in Kenntnis gesetzt und müssen die notwendigen Korrekturen so durchführen, dass die Konsistenz von Anforderungen, Entwurf etc. gewahrt wird.

Zum Ende der Hauptaufgabe findet eine Abnahme statt, bei der jeweils ein Projektteam den SEP-Betreuern und den anderen Projektteams die eigenen Arbeitsergebnisse vorstellt. Eine solche Abschlusspräsentation besteht aus einem Folienvortrag, in dem die zentralen Entwicklungsartefakte kurz vorgestellt werden, sowie einer Programmdemonstration. Alle SEP-Teilnehmer erhalten die Gelegenheit, die erstellte Software auszuprobieren. In die Entscheidung über die Projektabnahme durch die Betreuer fließen die Begutachtung des lauffähigen Systems, der Projektdokumentation sowie der Abschlusspräsentation durch die SEP-Betreuer ein.

Tabelle 1 enthält einen Überblick über einige bisher bearbeitete Aufgabenstellungen in der Hauptaufgabe. Die Komplexität der Aufgabenstellungen wurde so gewählt, dass die Lösung der Aufgabe für die Teams eine Herausforderung darstellte, die Teams jedoch nicht überforderte (vgl. [Göh 05]). Eine angemessene Komplexität der Aufgaben ist insbesondere vonnöten, um ein methodisches Vorgehen zu motivieren.

Bezeichnung	Kurzbeschreibung
WikiSEPedia	Einfaches Wikisystem
BSepCW	Dokumenten- und Versionsmanagementsystem
BoulderSEP	Geschicklichkeitsspiel in Boulder-Dash-Manier
SEPTakis	Action-Spiel (Horizontalscroller) in Katakis-Manier

Tab. 1 *In der SEP-Hauptaufgabe bearbeitete Aufgabenstellungen*

1.2 Beobachtete Probleme mit Entwicklungsaktivitäten

In früheren SEP-Veranstaltungen wurden verschiedene Probleme bei der Durchführung von Requirements-Engineering-, Entwurfs- und Testaktivitäten beobachtet. Ein signifikanter Teil der Probleme wurde dabei durch Mängel in Requirements-Engineering-Aktivitäten verursacht. Diese Probleme stellen wir im Folgenden vor.

Problem 1: Mangelnde Detaillierung von Anforderungen

Eine häufig anzutreffende Fehlannahme der Teilnehmer war, dass die Aufgabenstellung bereits die Anforderungen an das zu entwickelnde System in einem hinreichenden Detailgrad enthalten würde. Die Teilnehmer setzten sich folglich nur unzureichend mit den Anforderungen an das System auseinander. Daten, Funktionen und Verhalten des geplanten Systems wurden somit zu Beginn des Projekts nur unzureichend erarbeitet und dokumentiert. Die Zeit, die für die Entwicklung von Anforderungen zur Verfügung stand, wurde nicht effektiv genutzt.

Problem 2: Vermischung von Anforderungen und Entwurf

In anderen Fällen vermischten die Teilnehmer Anforderungen und Entwurf. Dies führte u.a. dazu, dass die Teilnehmer Entwurfsdetails festlegten, ohne dass sie sich eingehend mit den Anforderungen an das System befasst hatten. Zeit, die für die Entwicklung von Anforderungen zur Verfügung stand, wurde von diesen Teilnehmern genutzt, um sich mit Entwurfsdetails zu befassen. In mehreren Fällen mussten die entwickelten Entwürfe jedoch wieder verworfen werden, nachdem die Anforderungen geklärt worden waren.

Problem 3: Entwurf und Implementierung mit unklaren Anforderungen

Die Teilnehmer gingen mit unklaren Vorstellungen darüber, was das geplante System genau können sollte, in die Entwurfs- und die Implementierungsphase ein. Folglich hatten die einzelnen Teilnehmer entweder sehr vage oder untereinander widersprüchliche Vorstellungen von dem zu entwickelnden System. Der einzelne Teilnehmer wusste somit nicht, was er entwerfen oder implementieren sollte. Die

Planung und arbeitsteilige Durchführung der Entwicklungsaktivitäten wurde daher erschwert. Anforderungen und Entwurf wurden letztlich nicht in der Gruppe abgestimmt, sondern von einem Kern von Projektmitgliedern festgelegt, die das System implementierten. Die weiteren Teammitglieder konnten keinen Einfluss auf die Anforderungen und den Entwurf nehmen, sondern diese nur noch nachträglich in die Anforderungsdokumentation übernehmen. Die Entwicklung des Systems war für die Gruppe insgesamt nicht mehr transparent und daher auch nicht kontrollierbar.

Problem 4: Fehlende Basis für Testfallerstellung

Für die Erstellung von Testfällen stand keine geeignete Ausgangsbasis, d.h. keine Testreferenzen z.B. in Form von Anforderungen, zur Verfügung. Ein systematisches Vorgehen zur Testfallerstellung, Testdurchführung und Testprotokollierung konnte den Teilnehmern folglich nicht vermittelt werden. Den Teilnehmern war es somit z.B. nicht möglich, (detaillierte) Testfälle zu erstellen, bevor die Implementierung beendet war. In der Zeit, die eigentlich für die Testdurchführung zur Verfügung stand, mussten daher auch alle Testfälle entwickelt werden.

2 Szenarien und Features im SEP

Zur Abmilderung der von uns beobachteten Probleme wurde das SEP im Wintersemester 2005/2006 auf ein auf Szenarien und Features basierendes Vorgehen umgestellt. Durch die Verwendung von Szenarien und Features sollte im SEP eine verbesserte Unterstützung der Requirements-Engineering-, Entwurfs- und Testaktivitäten erreicht und nahtlose Übergänge zwischen den einzelnen Aktivitäten ermöglicht werden.

Szenarien dokumentieren Interaktionsfolgen, die zu einem Mehrwert für den Systemnutzer (oder einen anderen Stakeholder) führen. Das Entwickeln von Szenarien unterstützt ein Projektteam bspw. darin, über zusammenhängende Abläufe nachzudenken und diese im Detail zu dokumentieren. Zahlreiche Autoren berichten von positiven Erfahrungen mit Szenarien (vgl. z.B. [Hau 98; Car 00]). Details zur Verwendung von Szenarien im Requirements Engineering sind u.a. in [Poh 07] zu finden. Features sind nach Kang für den Endbenutzer sichtbare Eigenschaften eines Systems (vgl. z.B. [Kan 90]). Features sind bspw. in Form von Produktmerkmalen in zahlreichen Produktbeschreibungen enthalten (z.B. von Mobiltelefonen oder von Fotokameras).

Über die Anwendung von Szenarien und Features in einem Softwarepraktikum liegen unseres Wissens keine Berichte vor. Daher beschreiben wir im Folgenden, wie Szenarien und Features im SEP eingesetzt werden. Die Verwendung von Szenarien und Features wird dabei anhand der von einem SEP-Team erstellten Entwicklungsartefakte zu der Aufgabenstellung »SEPTakis« veranschaulicht.

2.1 Requirements Engineering

Abbildung 1 gibt einen Überblick über das Requirements Engineering mit den Eingaben und Ausgaben im SEP-Prozess. Den Teilnehmern wird eine (bewusst vage gehaltene) Beschreibung des zu entwickelnden Systems (Aufgabenstellung) zur Verfügung gestellt. Ausgehend von dieser Beschreibung definieren die Teilnehmer die Anforderungen an das zu entwickelnde System in Form von Features, (Interaktions-)Szenarien und einem Datenmodell (siehe Abb. 1).

Einige Features können direkt aus der Aufgabenstellung abgeleitet werden. Die Teilnehmer müssen die aus der Aufgabenstellung übernommenen Features jedoch detaillieren sowie die Verwendung der Features durch den Systemnutzer mittels Szenarien konkretisieren. Die Formulierung von Szenarien verbessert zum einen das Verständnis der Teilnehmer für die vorhandenen Features und stimuliert zum anderen das Formulieren neuer Features. Somit decken die Teilnehmer im Verlauf der Szenarioerstellung weitere Features auf bzw. detaillieren die zuvor identifizierten Features. Für diese Features müssen wiederum neue Szenarien formuliert werden. Auf diese Weise werden Features und Szenarien für das geplante System iterativ weiterentwickelt. Die iterative Entwicklung von Szenarien und Features trägt u.a. zur Verminderung von Problem 1 (»Mangelnde Detaillierung von Anforderungen«; siehe Abschnitt 1.2) bei. Problem 2 (»Vermischung von Anforderungen und Entwurf«) konnte aufgrund der Ausrichtung von Features und Interaktionsszenarien auf für den Endbenutzer sichtbare Systemeigenschaften ebenfalls vermindert werden.

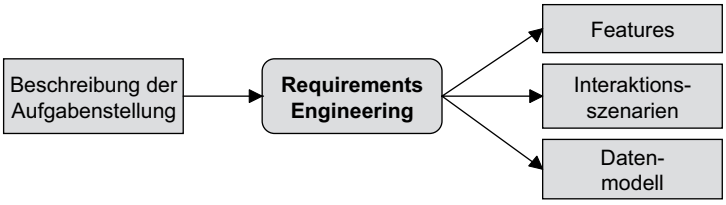


Abb. 1 *Requirements-Engineering-Aktivität mit Ein- und Ausgaben im SEP-Prozess*

Beispiel: Featuremodell

Basierend auf der Aufgabenstellung haben die Teilnehmer unter anderem die in Abbildung 2 dargestellten Features identifiziert. Die Features wurden von den Studierenden in einem Feature-Baum dokumentiert, der eine hierarchische Strukturierung von Features ermöglicht. Das Spiel SEPTakis als Wurzel-Feature wird unterteilt in die groben Features Grafik, Funktion, Steuerung und Sound. Diese Features werden wiederum weiterverfeinert, wie in Abbildung 2 dargestellt. Zusätzlich wird jedes Feature textuell erläutert, um Missverständnisse über die Bedeutung der einzelnen Features zu vermeiden.

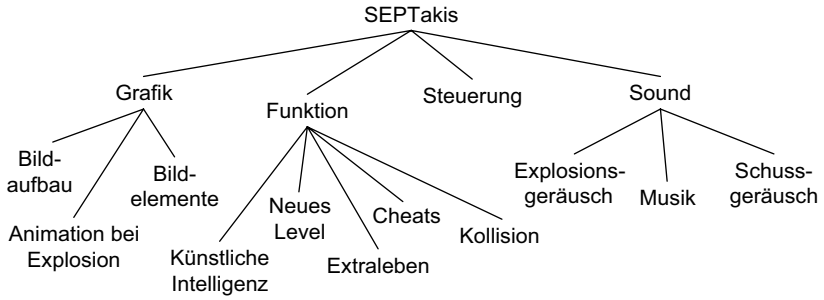


Abb. 2 *Auszug aus einem Feature-Baum für SEPTakis*

Beispielszenario

Szenarien werden im Requirements Engineering von den Studierenden in textueller Form dokumentiert. Um nachvollziehen zu können, ob jedes Feature mindestens einem Szenario zugeordnet ist, werden die Teilnehmer angehalten, die Features zu nummerieren und für jeden Szenarioschritt zu dokumentieren, welche Features in diesem Schritt zur Anwendung kommen. Abbildung 3 zeigt ein Beispielszenario zum Spiel SEPTakis.

Szenario:	Features:
1. Der Spieler steuert das Raumschiff gegen einen großen Asteroiden,die Kollision (2.6.1) zerstört das Raumschiff.	2.6.1: Kollision mit Objekten
2. Die Explosion des Raumschiffes wird durch ein Explosionsgeräusch (4.1) und Explosions-Animation (1.2) dargestellt.	4.1: Explosionsgeräusche 1.2: Animation bei Explosionen
3. Der Spieler verliert ein Leben (2.3).	2.3: Raumschiffzerstörung
4. Das Level beginnt am Levelanfang (2.2).	2.2: Levelneustart

Abb. 3 *Beispielszenario mit zugeordneten Features*

2.2 Entwurf

Der Entwurf ist im SEP in den Grobentwurf und den Feinentwurf untergliedert (siehe Abb. 4). Beim Grobentwurf sollen die Teilnehmer die Systemarchitektur skizzieren, d.h. das System in grobe Systemkomponenten strukturieren. Beim Feinentwurf erstellen die Teilnehmer für jede identifizierte Komponente im Grobentwurf ein (Entwurfs-)Klassendiagramm. Architekturszenarien unterstützen die Teilnehmer darin, Komponenten, Klassen und Methoden zu identifizieren. Die Teilnehmer erstellen für jedes Interaktionsszenario ein Architekturszenario, das die zur Ausführung des Interaktionsszenarios erforderlichen Interaktionen zwischen Systemkomponenten beschreibt. Die Komponenten und deren Interaktionen müssen wiederum durch Klassen und Methodenaufrufe realisiert werden.

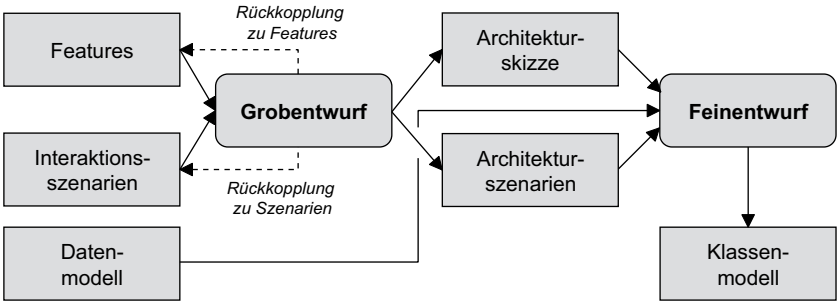


Abb. 4 *Entwurfsaktivitäten im SEP-Prozess*

Architekturszenarien bilden eine Brücke zwischen Anforderungen und Entwurf, d.h., sie erleichtern es den Teilnehmern, Beziehungen zwischen Anforderungen und Entwurf herzustellen. Bei der Entwicklung von Architekturszenarien reflektieren die Teilnehmer die zuvor formulierten Szenarien und Features. Häufig stellen die Teilnehmer fest, dass bestimmte Szenarien oder Features in der formulierten Weise nicht umzusetzen sind, oder dass Szenarien oder Features fehlen. Die zusätzlichen Erkenntnisse führen zu Anpassungen der bereits definierten Szenarien und Features. Diese stehen wiederum für die weiteren Entwurfsaktivitäten zur Verfügung. Durch die sukzessive Verfeinerung und Verbesserung der Anforderungen wird Problem 3 vermindert (»Entwurf und Implementierung mit unklaren Anforderungen«).

Beispiel: Komponenten von SEPTakis

Die Architektur des Systems wird im SEP mittels eines UML-Komponentendiagramms dokumentiert (siehe [Rum 05]). Abbildung 5 zeigt exemplarisch ein Komponentendiagramm für SEPTakis.

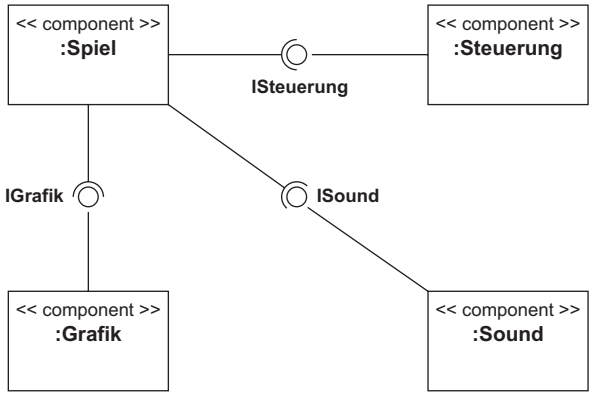


Abb. 5 *SEPTakis-Architektur als UML-Komponentendiagramm*

Beispiel: Architekturszenario

Architekturszenarien werden im SEP mittels UML-Sequenzdiagrammen (siehe [Rum 05]) dokumentiert. In Abbildung 6 wird ein Architekturszenario dargestellt, das das Szenario aus Abbildung 3 detailliert. Die Akteure des Architekturszenarios sind die in Abbildung 5 dargestellten Komponenten.

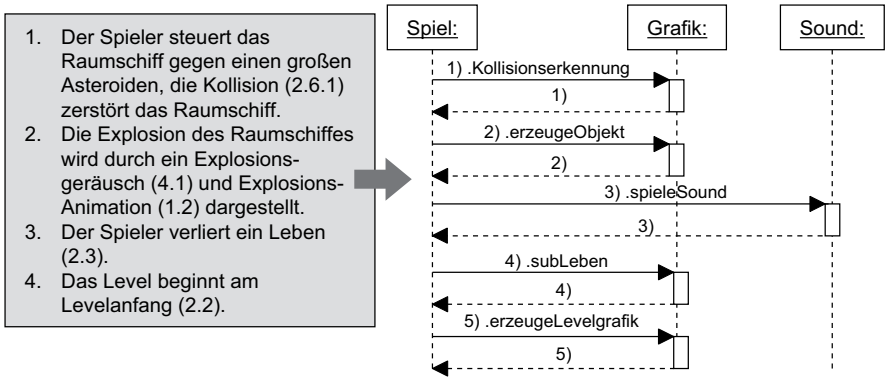


Abb. 6 *Architekturszenario von SEPTakis (rechts)*

2.3 Implementierung

Bei der Implementierung erstellen die Teilnehmer basierend auf den zuvor entwickelten Artefakten das System. Szenarien und Features unterstützen die Teilnehmer bei der Koordination der Implementierungsaktivitäten. Zum Beispiel können die Teilnehmer anhand der bereits durchführbaren Szenarien oder der realisierten Features den erreichten Projektfortschritt messen.

2.4 Testen

Bei der Entwicklung von Testfällen dienen Interaktionsszenarien als Grundlage für die Formulierung von Systemtestfällen und Architekturszenarien als Basis für die Entwicklung von Integrationstestfällen (siehe Abb. 7).

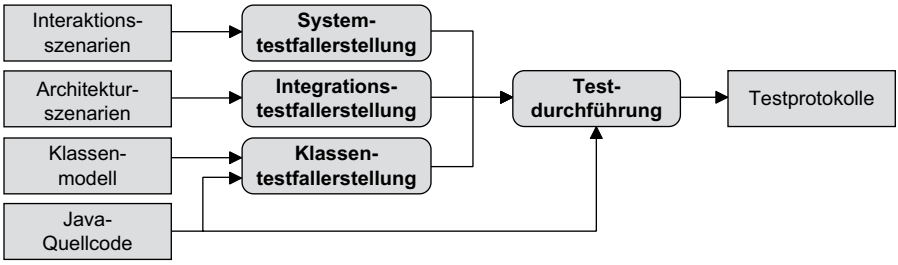


Abb. 7 *Testaktivitäten im SEP-Prozess*

Den Teilnehmern wird vermittelt, dass Testfälle bereits unmittelbar im Zusammenhang mit der Erstellung der jeweiligen Anforderungs- und Entwurfsartefakte entwickelt werden können. Durch das beschriebene Vorgehen wird Problem 4 (»Fehlende Basis für Testfallerstellung«) vermindert. Testdurchführung und Protokollierung erfolgen möglichst zeitnah zur Fertigstellung der für den Test benötigten Systemteile.

Beispiel: Systemtestfall

Von den SEP-Teilnehmern wird lediglich gefordert, exemplarisch jeweils zwei Modul-, zwei Integrations- und zwei Systemtestfälle zu erstellen. Darüber hinaus ist es den Teilnehmern selbst überlassen, wie intensiv sie das implementierte System testen. Abbildung 8 zeigt auszugsweise einen von SEP-Teilnehmern erstellten Systemtestfall.

Testziel:	Test des Szenarios »Raumschiff trifft auf Jäger«
Testumgebung:	Windows XP, JRE 1.5, JCreatorLE, SEPTakis
Vorbedingungen:	SEPTakis, kompiliert, noch nicht gestartet
Nachbedingungen:	Spiel läuft
Testfallszenario	
Tester	**System**
Der Tester startet das Spiel, indem er auf das Desktop-Icon klickt.	
	Das System stellt das Spielfenster mit allen Elementen dar, und das Spiel beginnt.
Der Tester prüft, ob • die Objekte (Raumschiff, Jäger, Asteroiden) im Spielfeld vorhanden sind, • die Lebenspunkte angegeben sind, • der Punktezähler vorhanden ist.	
Der Tester bewegt das Raumschiff über die Pfeiltasten.	
	Das System prüft die Eingaben auf Gültigkeit und führt diese aus. Das System bewegt das Raumschiff entsprechend der Eingabe.
Dem Raumschiff kommt ein flugbahnkreuzender Jäger entgegen. Der Tester versucht, dem Jäger auszuweichen, indem er die Pfeiltasten betätigt.	
	Das System bewegt das Raumschiff entsprechend der Eingabe.
Erfolgskriterien • Das System stellt das Spielfenster mit allen Elementen korrekt dar, und das Spiel beginnt. • Das System erkennt die Kollision zwischen Raumschiff und Objekten (Jäger, Asteroiden). • Das System erkennt die Benutzereingabe (Pfeiltasten für Richtungen und Leertaste für Schuss) und führt diese aus. • Das System zählt die erzielten Punkte im Punktezähler. • Das System zeigt die aktuellen Lebenspunkte an. • Das System reagiert nicht auf falsche Tastatureingaben. • Das System erkennt die Kollision mit dem Spielfeldrand und verhindert ein Herausfliegen.	

Abb. 8 *Exemplarischer Systemtestfall von SEPTakis*

3 Evaluierung

Die positiven Effekte von Szenarien und Features wurden an den Arbeitsergebnissen der Studierenden deutlich. Beispielsweise waren in den von Studierenden erstellten Features und Szenarien deutlich weniger Entwurfsinformationen enthalten als in den natürlichsprachlichen Anforderungen, die in früheren Semestern erstellt wurden. Die Entwicklung von Testfällen (insbesondere für den System- und Integrationstest) fiel den Studierenden deutlich leichter, da die erstellten Interaktions- und Architekturszenarien als Basis verwendet werden konnten. Für die Beurteilung des mit der Verwendung von Szenarien und Features erreichten Erfolgs war für uns die Meinung der SEP-Teilnehmer von Interesse. Maßgeblich waren für uns die folgenden Fragestellungen:

1. Inwieweit regten Szenarien und Features inhaltliche Diskussionen über Anforderungen an?
2. Inwieweit haben Wechselwirkungen zwischen Szenarien und Features zu einem verbesserten, inhaltlichen Verständnis des Systems beigetragen?

Um diese beiden Fragestellungen zu beantworten wurden vier Items formuliert und in einen Meinungstest zur Evaluierung der Veranstaltung integriert:

1. Die Szenarien des geplanten Spiels waren eine Grundlage für viele Diskussionen.
2. Die Features des geplanten Spiels waren eine Grundlage für viele Diskussionen.
3. Durch die Formulierung von Szenarien zu den einzelnen Features habe ich die Features besser verstanden.
4. Bei der Formulierung von Szenarien habe ich Ideen für weitere Features bekommen.

Die Items wurden von insgesamt 94 Teilnehmern auf folgender Skala bewertet: sehr zutreffend, zutreffend, mittel, unzutreffend, sehr unzutreffend. Abbildung 9 stellt die Resultate des Meinungstests zu den vier vorgestellten Items als Säulendiagramm dar.

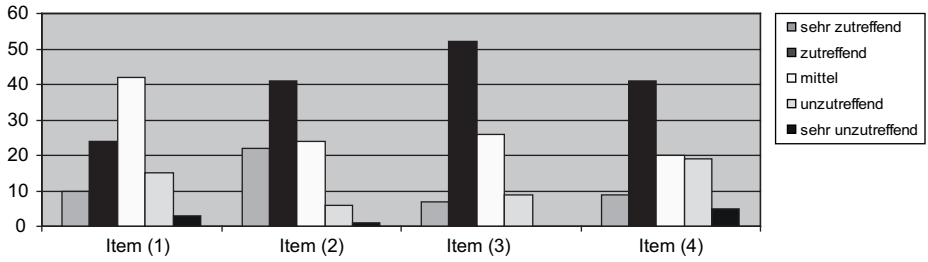


Abb. 9 *Antworthäufigkeiten zu den vier Items*

Im Folgenden interpretieren wir die Resultate der einzelnen Items in Hinblick auf unsere Ausgangsfragestellungen. Wir werten eine nicht negative Bewertung eines Teilnehmers, d.h. die Kategorien »mittel«, »zutreffend« und »sehr zutreffend«, als Zustimmung zu dem Item:

1. Zu diesem Item äußerten 76 Teilnehmer (81%) ihre Zustimmung. Wir schließen daraus, dass die Gruppen durch das Formulieren von Szenarien zu inhaltlichen Diskussionen angeregt wurden.
2. Zu diesem Item äußerten 87 Teilnehmer (93%) ihre Zustimmung. Wir schließen daraus, dass Features ein zentraler Aspekt von inhaltlichen Diskussionen innerhalb der Gruppen waren.
3. Zu diesem Item äußerten 70 Teilnehmer (74%) ihre Zustimmung. Wir schließen daraus, dass die Teilnehmer das Konkretisieren von Features durch Szenarien als positiv empfunden und im Projekt genutzt haben.
4. Zu diesem Item äußerten 85 Teilnehmer (90%) ihre Zustimmung. Wir schließen daraus, dass die Teilnehmer Szenarien als sehr hilfreich empfanden, um Features zu identifizieren.

Insgesamt lässt die Auswertung des Meinungstests auf einen guten bis sehr guten Erfolg der Verwendung von Szenarien und Features im SEP schließen. Szenarien und insbesondere Features regten einen signifikanten Anteil der Teilnehmer zu inhaltlichen Diskussionen an. Zudem erwiesen sich Szenarien als sehr hilfreich, um neue Features zu identifizieren sowie das Verständnis über bekannte Features zu vertiefen.

4 Fazit und Ausblick

In diesem Beitrag wurde die Verwendung von Szenarien und Features in einem Softwarepraktikum dargestellt. Die Arbeitsergebnisse der Teilnehmer zeigen, dass Szenarien und Features zu einer Verbesserung gegenüber früheren Veranstaltungen beigetragen haben. Die positive Bewertung von Szenarien und Features im Meinungstest ist ein Indiz dafür, dass die Teilnehmer die positiven Effekte von Szenarien und Features selbst erfahren haben. Diese Ergebnisse motivieren uns, weitere Evaluierungen zur Untermauerung unserer bisherigen Erkenntnisse durchzuführen.

5 Danksagung

Wir möchten an dieser Stelle einen besonderen Dank an unsere studentischen Hilfskräfte André Heuer und Sebastian Jackels für ihre wertvollen Beiträge zur Gestaltung des SEPs sowie für ihren unermüdlichen Einsatz bei der Betreuung von SEP-Gruppen aussprechen.

Literatur

[Car 00] J. M Carrol (Hrsg.): Making Use: Scenario-Based Design of Human Computer Interactions. MIT Press, Cambridge, 2000.

[Göh 05] P. Göhner, F. Bitsch, H. Mubarak: Softwaretechnik live – im Praktikum zur Projekterfahrung. In: SEUH '05, dpunkt.verlag, Heidelberg, 2005, S. 41-55.

[Ha 02] T. Hall, S. Beecham, A. Rainer: Requirements Problems in Twelve Companies – An Empirical Analysis. In: Proceedings of the 6th International Conference on Empirical Assessment and Evaluation in Software Engineering (EASE 2002), Keele University, 2002.

[Hau 98] P. Haumer, K. Pohl, K. Weidenhaupt: Requirements Elicitation and Validation with Real World Scenes. IEEE Transactions on Software Engineering, Vol. 24, Nr. 12, 1998, S. 1036-1054.

[Kan 90] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, Pittsburg, 1990.

[Kru 03] P. Kruchten: The Rational Unified Process – An Introduction. Addison-Wesley, Reading, 2003.

[Poh 07] K. Pohl: Requirements Engineering – Grundlagen, Prinzipien und Techniken. dpunkt.verlag, Heidelberg, 2007.

[Rau 07] A. Rausch, M. Broy: Das V-Modell XT – Grundlagen, Erfahrungen, Werkzeuge. dpunkt.verlag, Heidelberg, erscheint 2007.

[Rum 05] J. Rumbaugh, I. Jacobson, G. Booch: The Unified Modeling Language Reference Manual. 2. Aufl., Addison-Wesley, Boston, 2005.

[Stan 04] The Standish Group: Chaos Demographics, 2004.

Selbstbestimmung oder Anleitung: Erfahrungen mit einem Softwaretechnikpraktikum im Bereich Qualitätssicherung

Sebastian Jekutsch · Christopher Oezbek · Stephan Salinger
Institut für Informatik, Freie Universität Berlin
{jekutsch|oezbek|salinger}@inf.fu-berlin.de

Zusammenfassung

Wir berichten von zwei ähnlichen Softwaretechnikpraktika. In der ersten Version wurde ein freies Softwaresystem anhand wöchentlicher Übungsblätter hinsichtlich Qualitätsmängel untersucht. Im Kontrast zu diesem sehr angeleiteten Vorgehen gab es in der zweiten Version lediglich die eine Zielvorgabe, möglichst viele Schwachstellen zu finden. Der Weg dorthin blieb den Studierenden überlassen.

1 Einführung

Im Folgenden werden zwei Durchführungen des Praktikums Softwaretechnik an der Freien Universität Berlin vergleichend beschrieben. Das Praktikum behandelte in beiden Fällen Themen der analytischen Qualitätssicherung anhand eines freien Softwaresystems, einem Open-Source-Content-Management-System (CMS).

Die Durchführungen der Jahre 2004 und 2005 unterschieden sich kaum in den Lernzielen und den fachlichen Inhalten, sondern lediglich in der Vorgehensweise: Während das Praktikum 2004 eher Vorlesungscharakter mit praktischen, aufeinander aufbauenden Übungsaufgaben hatte, fehlten 2005 Übungsblätter und Anweisungen beinahe gänzlich. Stattdessen wurden lediglich die Erfolgskriterien für die ansonsten eigenständig zu planende Arbeit vorgegeben.

Beide Versionen des Praktikums enthielten die folgenden Inhalte:

- Nach einer Einführung in allgemeine Anforderungen an Content-Management-Systeme galt es zunächst, sich in die Software einzuarbeiten.
- Selbst auszuwählende Teile der Software wurden anhand verschiedener Qualitätskriterien untersucht.

- Wurden funktionale Schwächen entdeckt, so waren sie in die für uns zugängliche Fehlerdatenbank einzutragen. Die Entwickler des CMS haben die neuen Einträge daraufhin bewertet.
- Zum Abschluss des Praktikums wurde von den Studierenden ein Paket von automatisierten Testfällen, Analysen und gemeinsam erstellten Ratschlägen an die Entwickler des CMS übergeben.

Bei der Konzeption lagen folgende Kriterien zugrunde, die ein Softwaretechnikpraktikum aus unserer Sicht erfüllen sollte (siehe auch [Prechelt93], [Shaw00]):

1. Theoretisch vermittelte Techniken und Methoden sollen angewendet und die Konsequenzen einer Nichtanwendung im Vergleich zu einem unreflektierten Ad-hoc-Vorgehen erkannt werden.
2. Das Anwendungsbeispiel sollte eine realistische Größe haben und nicht lediglich akademisches »Spielzeug« sein.
3. Die Aufgabe soll nicht von einer Person alleine lösbar sein.
4. Zeitmanagement, Kommunikation, Konfliktlösung und Entscheidungsfindung sollen geübt werden.

Voraussetzung war der Abschluss der Vorlesung Softwaretechnik. Die Studierenden arbeiteten in Teams von drei bis vier Personen. Es gab rotierend einen Gruppenleiter, der Ansprechpartner für uns war.

2 Angeleitete vs. selbstbestimmte Version

Die erste Durchführung des Praktikums besaß Vorlesungscharakter:

- Es gab zwei Präsenztermine in der Woche: einen von uns durchgeführten Vortragstermin und eine Praxisstunde. Die restliche praktische Arbeit war in Eigenregie zu leisten.
- Die Vorträge wiederholten relevante Themen aus der Softwaretechnikvorlesung mit speziellem Bezug auf die Praktikumssituation. Im wöchentlichen und zweiwöchentlichen Rhythmus wurden Übungsblätter mit festem Fertigstellungstermin verteilt.
- In der Praxisstunde wurden die Ergebnisse der vergangenen Übungsblätter vorgestellt und diskutiert.

Es gab vier große Themenblöcke: Modultests für einen kleinen Bereich der Software, Durchsichten und statische Codeanalyse für einen anderen Teil sowie Abnahme einer speziellen Funktionalität der Software[1].

Letztlich blieben die Ergebnisse des Praktikums hinter den Erwartungen zurück. Es wurden nur wenige Defekte gefunden, die erstellten Testfälle waren tri-

1 Wir gehen auf die Inhalte im Folgenden nicht weiter ein. Genaueres kann auf der Veranstaltungswebsite *http://projects.mi.fu-berlin.de/w/bin/view/SWTprak/* nachgelesen werden.

vial, die Durchsichten schienen dem Ziel zu folgen, möglichst einfache Perspektiven einzunehmen und bei der funktionalen Abnahme äußerten sich Fehldeutungen, was die Software leisten müsse und in den Abnahmekriterien auftauchen solle.

Für die schwachen Ergebnisse machten wir vor allem den Charakter des Praktikums verantwortlich:

- Vordergründiges Ziel der Studierenden schien es gewesen zu sein, das Übungsblatt zwar rechtzeitig, aber ohne viel Aufwand zu lösen. Die Ergebnisse bzgl. der impliziten Zielsetzung, die Qualität der Software durch Aufdecken von Defekten und Versagen zu erhöhen, blieben hinter den Möglichkeiten zurück.
- Bei der Besprechung wurde das schwache Ergebnis entweder mit der schlecht strukturierten und dokumentierten Software begründet oder mit den ungenauen Vorgaben in den Übungsblättern.

Dies geschah, obwohl wir regelmäßig die Kriterien an ein gutes Ergebnis wiederholten.

Aufgrund der gewonnenen Erfahrung versuchten wir, die zweite Durchführung im nachfolgenden Jahr am zu erreichenden Ergebnis zu orientieren.

Das Praktikum bestand nun im Wesentlichen aus zwei Arbeitsaufträgen:

1. Es sollten so viele relevante Versagensfälle der Software provoziert werden wie möglich. Die Relevanz errechnete sich hierbei vor allem aus der Einstufung (severity) des Versagensberichts durch die Entwickler des CMS.
2. Jede Gruppe musste wöchentlich auf der Webseite der Veranstaltung einen Bericht hinterlegen, der den beschrittenen Weg, die zugrundeliegende Überlegung, die erzielten Ergebnisse und die zukünftigen Arbeitsschritte beschrieb.

Wie man die Versagensfälle findet, blieb den Studierenden überlassen. Es gab in der großen Runde regelmäßig Rückmeldung und Diskussionen zu angemessenen und gewählten Vorgehensstrategien. Für die Endnote war eine gut begründete Vorgehensweise genauso wichtig wie ihr Erfolg.

3 Vergleich der Versionen und Schlussfolgerungen

Zum Vergleich interessant sind

- der Erfolg im Sinne der Qualitätssicherung. Hier zeigte die zweite Durchführung bessere Ergebnisse bei vergleichbaren Rahmenbedingungen. Der Grund lag unter anderem schlicht im erhöhten Arbeitseinsatz.
- der Lernerfolg im Sinne der Softwaretechnikausbildung. Dieser ist schwer zu benennen, da es keine Abschlusskontrolle gab. Unbestritten ist, dass sich deutlich verschiedene Schwerpunkte ergaben.
- das Interesse der Studierenden im Verlauf der Veranstaltung. Hier polarisierte die zweite Version mehr: Während die erste Version durchschnittliche Noten

von Seiten der Studierenden bekam, fielen bei der selbstgeleiteten Version die Stimmen sowohl stark positiv als auch negativ aus. Wir sahen also sowohl mehr Spitzenleistung als auch mehr demotivierte Studenten in der zweiten Version.

Die Studierenden haben in der zweiten Version sowohl im Umfang (Anzahl gefundener Versagensfälle und Menge der zumindest durchdachten Techniken) als auch in der Güte der Ergebnisse (Relevanz der Versagensfälle gemessen an der vergebenen »severity« im Defektverfolgungssystem und deren Dokumentation, bevorzugt in Form von automatisierten Testfällen) eine bessere Leistung gezeigt.

Durch die Pflicht, sich selbst um die einzusetzenden Methoden und Techniken zu kümmern, stand dabei natürlich zunächst nicht deren Durchführung im Vordergrund, sondern deren Ökonomie, also die Frage, ob es sich überhaupt lohnen wird, die Technik einzusetzen.

Die Unzufriedenheit der Studierenden mit dem zweiten Praktikum resultierte aus der Unklarheit der Vorgehensweise. Die gestellte Aufgabe war weder intuitiv noch nach »Schema F« zu lösen, die potenziell defekte Software sehr umfangreich und das Ziel nicht schon nach ein paar Tagen zu erreichen. Dies traf nicht das gewohnte Lern- und Arbeitsschema an einer Universität.

Darüber hinaus offenbarte sich die Gruppenarbeit als schwierig. Ein verbreiteter Irrtum war, dass Gruppenarbeit gemeinsame gleichzeitige Arbeit an einem Problem bedeutet. Zudem entstand ein Wettbewerb zwischen den Gruppen, weil ein einmal berichteter Mangel natürlich für die anderen Gruppen »verloren« war. Dieser Wettbewerbscharakter hat vielen Teilnehmern missfallen.

Trotzdem sind wir Veranstalter zufriedener mit dem (auch objektiv besseren) Ergebnis der zweiten Version, denn wir sind uns sicher, in dieser freien Form praxisrelevantere Lernziele erreicht zu haben als bei der Orientierung an Übungsblättern.

Allerdings erscheint es uns notwendig, eine Balance zwischen dem bequemen Wochentakt der angeleiteten Version und dem elfwöchigen Alleingang des zweiten Durchlaufs zu erreichen. Wir wollen Zwischenstopps einführen, bei denen man Teilziele zu erreichen hat, die für sich einen Wert haben. Am Ende eines solchen zeitlich klar begrenzten Blocks bleibt dann die Zusammenfassung der einzelnen Ergebnisse, die jede Gruppe für ihre eigene Weiterarbeit nach Belieben verwenden kann.

Zudem werden wir die Gruppen und ihr inneres Funktionieren genauer beobachten und in Zweifelsfällen deeskalierend eingreifen müssen.

Literatur

[Prechelt93] Lutz Prechelt: Ziele und Wege für Softwaretechnik-Praktika. *Proceedings Workshop SEUH'93*, B.G Teubner, 1993, S. 78 – 82.

[Shaw00] M. Shaw. Software engineering education: A roadmap. *Proceedings of the Conference on the Future of Software Engineering* (Limerick, Ireland, June 04 – 11, 2000). ICSE '00. New York, S. 371 – 380.

Projektorientierte Vermittlung von Entwurfsmustern in der Software-Engineering-Ausbildung

Wilhelm Hasselbring · Jasminka Matevska · Heiko Niemann · Dennis Geesen

Abteilung Software Engineering, Department für Informatik
Universität Oldenburg
{hasselbring|matevska|heiko.niemann}@informatik.uni-oldenburg.de

Hilke Garbe · Stefan Gudenkauf · Steffen Kruse · Claus Möbus

Abteilung Lehr- und Lernsysteme, Department für Informatik
Universität Oldenburg
{hilke.garbe|stefan.gudenkauf}@informatik.uni-oldenburg.de

Marco Grawunder

Software Labor, Department für Informatik
Universität Oldenburg
{marco.grawunder}@informatik.uni-oldenburg.de

Zusammenfassung

Die projektorientierte Ausbildung im Software Engineering ist inzwischen etabliert, z.B. in Form von Softwarepraktika. Im vorliegenden Beitrag berichten wir über unsere Erfahrungen mit der Vermittlung von Entwurfsmustern in diesem Kontext. Wir konzentrieren uns dabei auf die Lehre im Grundstudium und erläutern wie wir hier ein nicht triviales Softwaresystem vorlesungsbegleitend als Lehrbeispiel einsetzen. Ergänzt wird die Präsenzlehre durch ein auf Entwurfsmuster spezialisiertes E-Learning- und Assistenzsystem.

1 Einleitung

Ein zentraler Aspekt in der Ausbildung im Software Engineering ist die Vermittlung von Techniken zur Modellierung und der anschließenden programmiertechnischen Umsetzung, beispielsweise in der Kombination von UML und Java [AHM99]. Insbesondere für die Vermittlung von Modellierungstechniken ist es

jedoch nicht ausreichend, nur die Notationen zu behandeln. Das Erlernen von Software Engineering hat viel damit zu tun, Erfahrungen zu sammeln. Dies kann durch eigenes »Erleben« z.B. in Form von Übungsaufgaben geschehen, aber auch in der Vermittlung explizit dokumentierten Erfahrungswissens. Ein bewährtes Mittel dazu sind Entwurfsmuster, die bewährte Lösungsstrukturen für häufig wiederkehrende Problemstellungen liefern [GHJV96]. Diese Muster kommen im Allgemeinen erst in Softwaresystemen mit einer gewissen Komplexität richtig zur Geltung, womit die Studierenden jedoch gerade in den ersten Semestern kaum in Berührung kommen. Ein Ansatz, die Studierenden trotzdem mit nicht trivialen Softwaresystemen zu konfrontieren, ist die Arbeit mit vorgefertigten Beispielen, die die Studierenden (1) verstehen, (2) nachdokumentieren und (3) selbst erweitern sollen. In den ersten zwei Semestern ist es kaum möglich, durch die Studierenden selbst ein komplexes Softwaresystem konstruieren zu lassen. Gleichzeitig ist es sehr schwierig, die Notwendigkeit zur Modellierung von Entwürfen zu motivieren, wenn die Studierenden bisher nur kleinere Programme kennengelernt haben (geschweige denn die Notwendigkeit zur Modellierung von Anforderungen zu motivieren, worauf wir in diesem Beitrag jedoch nicht eingehen können). Unser Ansatz besteht nun darin, hierzu ein speziell für die Ausbildung in Software Engineering realisiertes, nicht triviales Softwaresystem für die vorlesungsbegleitenden Übungen zu nutzen, im dem insbesondere einige Entwurfsmuster zur Lösung genutzt wurden. Unser Vorgehen besteht hier darin, den Studierenden ein positives Beispiel zu geben. Alternativ könnte auch ein schlecht strukturiertes Beispiel genutzt werden, um die Notwendigkeit zu einer besseren Strukturierung darzustellen, was wir jedoch nicht als geeignetes Motivationsmittel ansehen.

Die Struktur der Lehrmodule im Informatikstudium an der Universität Oldenburg mit direktem Bezug zu Software Engineering ist in Abbildung 1 skizziert. Im Programmierkurs werden die Grundlagen der objektorientierten Programmierung mit Java vermittelt [Bol06]. Auf die Vorlesung (VL) Software Engineering, in der Entwurfsmuster eingeführt werden, wird in Abschnitt 2 näher eingegangen sowie auf das zweisemestrige Softwareprojekt in Abschnitt 3. Das als Kombination von Vorlesung und Seminar konzipierte Modul Software System Engineering (Lehrsprache Englisch) vertieft u.a. ausgewählte Muster (beispielsweise für JavaEE), und in der zweisemestrigen Projektgruppe bearbeiten 12 Studierende im Hauptstudium ein Jahr lang ein größeres Projekt. In diesem Beitrag betrachten wir nur die Module des ersten bis vierten Semesters (unabhängig davon, ob es im auslaufenden Diplomstudium oder im seit dem Wintersemester 2000 angebotenen Bachelorstudium stattfindet). Daher gehen wir auch nicht auf die weiteren Module der Praktischen Informatik bzw. auf speziellere Angebote im Software Engineering ein. Das auf Entwurfsmuster spezialisierte E-Learning- und Assistenzsystem InPULSE wird in Abschnitt 4 vorgestellt, bevor wir diesen Beitrag in Abschnitt 5 zusammenfassen und zukünftige Aktivitäten diskutieren.

2 Das Modul Software Engineering

In diesem Vorlesungsmodul werden zunächst Vorgehens- und Prozessmodelle [LL06] sowie das Konfigurationsmanagement behandelt. Das Konfigurationsmanagement wird als grundlegende Technik zur Arbeit im Team bereits so früh eingeführt, damit die Lösungen zu den vorlesungsbegleitenden Übungen über das Repository abgegeben werden können (wir verwenden Subversion, *http://subversion.tigris.org/*). Nach einer Einführung in die Anforderungsanalyse werden Modellierungskonzepte zur Beschreibung von Struktur und Dynamik mittels der UML-Notation erläutert. Auf dieser Grundlage können dann ausgewählte objektorientierte Entwurfsmuster [GHJV96] behandelt werden. Dies geschieht bereits mit Bezug auf das Lehrbeispiel, soweit sinnvoll. Die Vorlesung wird abgeschlossen mit einer Einführung in Techniken zur Qualitätssicherung und zum Projektmanagement. Im Folgenden stellen wir das Lehrbeispiel, die vorlesungsbegleitenden Aufgaben sowie eine spezielle Zusatzaufgabe zur Erweiterung des Softwaresystems durch die Studierenden vor.

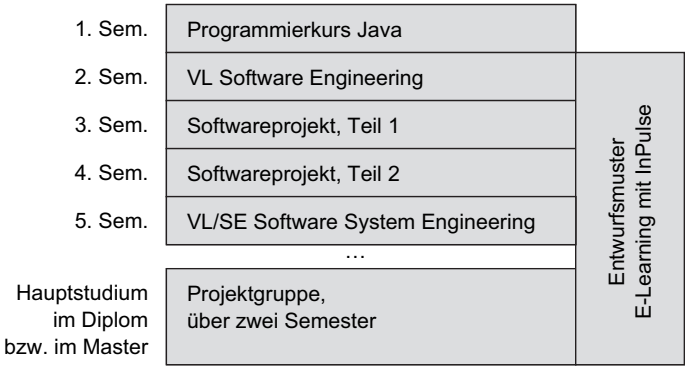


Abb. 1 *Lehrmodule mit direktem Bezug zu Software Engineering*

2.1 Das Lehrbeispiel File Manager

Im Rahmen der Veranstaltung Software Engineering wird als Lehrbeispiel und Grundlage für vertiefende Übungsaufgaben die Fallstudie »File Manager« eingesetzt. Diese Fallstudie wurde im Rahmen eines Projekts entwickelt, das die Schaffung eines praxisnahen didaktischen Lehrkonzeptes zur Ausbildung im Software Engineering zum Thema hatte [Gud04]. Dazu wurden der Softwareentwicklungsprozess und die verschiedenen fachspezifischen Konzepte am Beispiel der Konstruktion eines mittelgroßen Softwaresystems verdeutlicht. Die Entwicklung des File Manager erfolgte mithilfe der Entwicklungswerkzeuge Eclipse (*www.eclipse.org*) und EclipseUML (*www.omondo.de*).

Abbildung 2 stellt einen Screenshot des File Manager dar. Er besteht aus einer Menüleiste (1) und einer Schaltflächenleiste (2), über die sämtliche Funktionalitäten des File Manager zugänglich sind. Zusätzlich können dateispezifische Funktionalitäten über ein Kontextmenü (3) angesprochen werden. Das vom Programm verwaltete Dateisystem wird in einer geteilten Baumansicht (4) visualisiert.

Neben der eigentlichen Implementierung umfasst die Fallstudie u.a. die Ausarbeitung einer exemplarischen Anforderungsdefinition, ein Entwurfsdokument inklusive der verschiedenen Diagramm- und Planhilfen, einen Aufgabenkatalog für Arbeitsblätter und Gruppenübungen, Foliensätze sowie eine Betrachtung der zum Einsatz gekommenen Entwurfsmuster und Konzepte. Dabei wurde besonderer Wert auf die Berücksichtigung didaktischer Erkenntnisse gelegt. Insbesondere wurde ein allgemeines Planungsmodell ausgearbeitet, das die Besonderheiten einer Lehrveranstaltung zum Software Engineering berücksichtigt [Gud04].

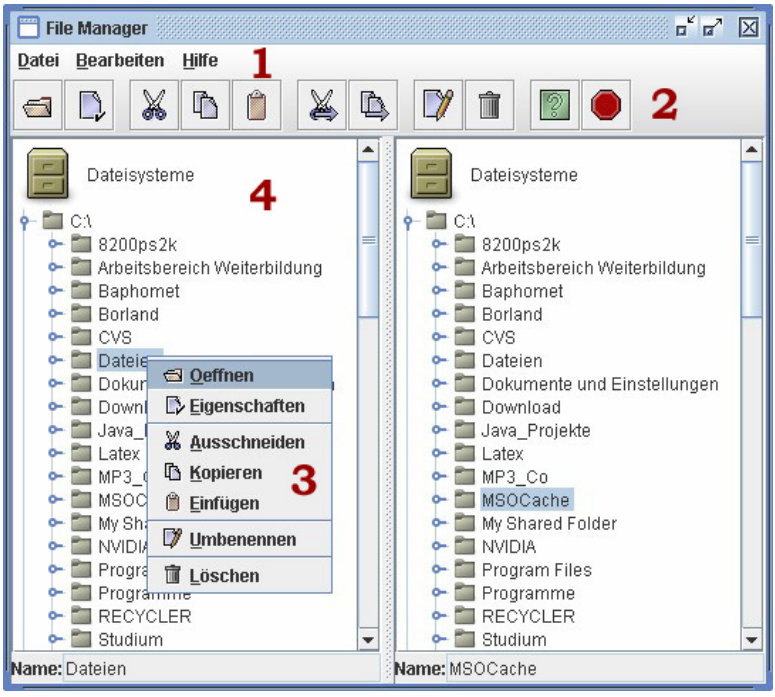


Abb. 2 *Screenshot der Fallstudie File Manager*

Die Architektur des File Manager

Die Softwarearchitektur des File Manager stellt sich als Zwei-Schichten-Architektur dar. Der wesentliche Aspekt ist die Trennung von Zuständigkeiten in Form der Trennung der Programmvisualisierung von der Programmlogik. Der struktu-

relle Aufbau ist in Abbildung 3 dargestellt. Die Architektur ist folgendermaßen aufgebaut:

- Die Benutzungsschnittstelle ist im Paket `presentation` wiederzufinden. Sie besitzt das Unterpaket `swing`, in dem Java-Swing-Elemente zur grafischen Darstellung gekapselt sind.
- Die Dateilogik ist die Anwendungsebene des Systems. Sie setzt direkt auf dem Datei- und Verzeichnissystem der Zielumgebung auf und stellt Operationen auf diesem bereit. Die Dateilogik ist im Paket `logic` realisiert. Sie besitzt das Unterpaket `file`, das die direkte Datei- und Verzeichnisbehandlung übernimmt.

Dieser Aufbau erlaubt den einfachen Austausch sowohl der Benutzungsschnittstelle als auch der Dateilogik. Eine denkbare alternative Realisierung der Dateilogik wäre z.B. die Verwaltung des Repositories eines Versionsmanagementsystems, wie Subversion. Kern der Trennung von Visualisierung und Logik bildet die Schnittstelle `EntityInterface`, die generische Entitäten darstellt, die von der Benutzungsschnittstelle darzustellen und von der Dateilogik zu behandeln sind.

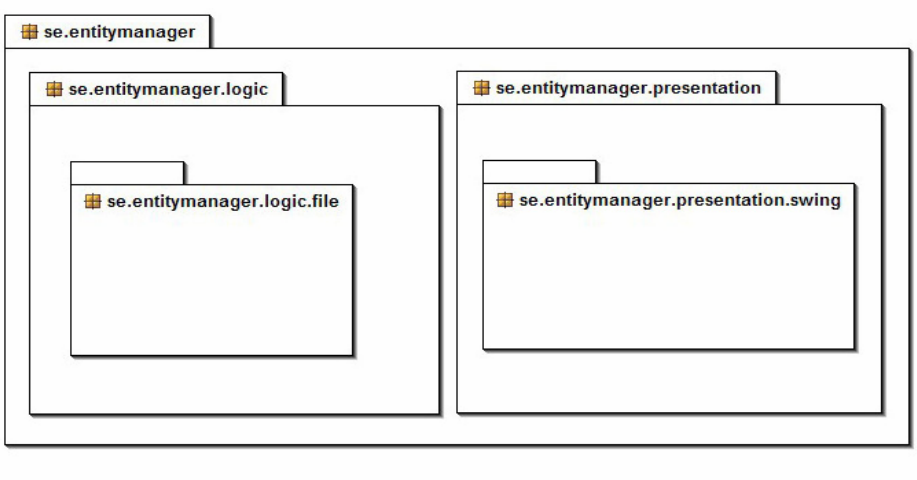


Abb. 3 *UML-Paketdiagramm des File Manager*

Entwurfsmuster in der Architektur des File Manager

Obwohl die Fallstudie File Manager kein extrem großes System ist, enthält dessen Implementierung zahlreiche Entwurfsmuster zur Anschauung und zur Lehre. Diese sollten u.a. von den Teilnehmern der Veranstaltung Software Engineering im Programmcode entdeckt und beschrieben werden (siehe Abschnitt 2.2). Tabelle 1 stellt eine Auswahl der zu findenden Entwurfsmuster und deren Verwendung vor. Aus Platzgründen können wir in diesem Beitrag nicht näher auf den Detailentwurf eingehen.

Entwurfsmuster	Verwendung
Facade	*Fassaden* ermöglichen sehr schmale Kommunikationsschnittstellen zwischen Benutzungsoberfläche und Dateilogik und kapseln damit beide Schichten ein.
Singleton	*Singleton*-Muster stellen die einmalige Instanziierung von Klassen sicher und sorgen dafür, dass die *Singleton*-Klassen wohlbekannte Ansprechpunkte für Clients darstellen. *Singleton*s treten im File Manager u.a. im Zusammenspiel mit Fassaden auf.
Abstract Factory	Durch die Bereitstellung einer Schnittstelle, die die Erstellung von Symbolen beschreibt, ohne eine konkrete Implementierung zu spezifizieren, und der zugehörigen implementierenden Klassen werden die Austauschbarkeit und die Konfigurierbarkeit der Symbolverwendung innerhalb des File Manager gewährleistet.
Factory Method	Mit Hilfe des *Factory Method*-Musters werden die Verzeichnis- und Dateistrukturen repräsentierenden Entitäten generiert, ohne dass bekannt ist, zu welcher (Datei-)Art diese gehören.
Observer	Die Verwendung des *Observer*-Musters ermöglicht, auf Änderungen der Entitäten automatisch zu reagieren. Ein explizites Aktualisieren der Benutzungsoberfläche ist nicht notwendig.
Composite	Mit Hilfe des *Composite*-Musters wird die Eigenart der Dateien, andere Dateien enthalten oder nicht enthalten zu können (z.B. Dateiarchive), sowie die unterschiedliche Behandlung von Dateien und Ordnern optimal wiedergegeben.

Tab. 1 *Auswahl einiger Entwurfsmuster in der Architektur des File Manager.*

2.2 Begleitende Aufgaben

Im Modul Software Engineering werden wöchentlich Übungsaufgaben gestellt, die in Tutorien mit ca. 20 Teilnehmern besprochen werden. Die Modellierungsaufgaben werden dabei mit UML-Werkzeugen gelöst: In 2006 wurde Poseidon (*www.gentleware.com*) eingesetzt, in den Jahren davor Together (*www.borland.com*). Neben Aufgaben aus unterschiedlichen Anwendungsbereichen werden auch Aufgaben zur Fallstudie File Manager (siehe Abschnitt 2.1) und zu Entwurfsmustern gestellt, die nach folgender Klassifizierung vorgestellt werden sollen:

- Allgemeine Aufgaben zum File Manager (ohne Entwurfsmuster)
- Aufgaben zu Entwurfsmustern
- Aufgaben zu Entwurfsmustern im File Manager

Zunächst werden Anwendungsfälle und Interaktionen hinsichtlich der Fallstudie File Manager modelliert. Hierbei geht es in den Tutorien darum, die studentischen Lösungen mit dem Entwurf des File Manager zu vergleichen, um, wie bereits erwähnt, an einem guten Beispiel zu lernen. Weiterhin wird auch ein Reverse Engineering des File Manager bzw. von Teilen des File Manager vorgenommen.

Bei den »einfachen« Aufgaben zu Entwurfsmustern wird das benötigte Muster in der Aufgabenstellung vorgegeben. Hierbei wird die Verwendung des Entwurfsmusters in einer konkreten Problemstellung geübt, z.B. wurde das Entwurfsmuster Singleton bei der Implementierung einer Komponente für eindeutige Schlüsselgenerierung oder das Entwurfsmuster Proxy bei der Modellierung der Stellvertreterrolle eines Managers gegenüber seinen Sportlern genutzt. Bei den »komplexen« Aufgaben konnte auf das E-Learning-System InPULSE (siehe Abschnitt 4) zurückgegriffen werden. Hiermit kann über das Dialogsystem ein passendes Entwurfsmuster bestimmt werden, z.B. für eine Playlist von Musiktiteln das Entwurfsmuster Flyweight oder für einen Stecker eines Föns, der im Ausland genutzt werden soll, das Entwurfsmuster Adapter. Nach der Bestimmung des Entwurfsmusters kann InPULSE bei der folgenden Modellierung durch hinterlegte Informationen zu den Mustern helfen, was z.B. in einer Aufgabe zur Modellierung einer Plastikartikelproduktion mittels des Entwurfsmusters Abstract Factory geschehen ist. Bei den Aufgaben zu Entwurfsmustern im File Manager wird z.B. auf die Ergebnisse aus der Aufgabe »Reverse Engineering« (siehe oben) zurückgegriffen. Die verwendeten Entwurfsmuster sollen identifiziert und der Zweck ihres Einsatzes diskutiert werden. In 2006 wurde auch eine Zusatzaufgabe für besonders interessierte Studierenden gestellt, die über den Rahmen der normalen Aufgaben hinwegging. Daher wurde für diese Aufgabe eine vierwöchige Bearbeitungszeit gestattet. Zur Motivation wurde eine Prämie für die beste Lösung ausgelobt. Diese Zusatzaufgabe wird im folgenden Abschnitt dargestellt. Zukünftig könnte dies zur Pflicht werden.

2.3 Zusatzaufgaben zur Erweiterung des File Manager

Die Zusatzaufgabe bestand darin, die Funktionalität des File Manager sinnvoll zu erweitern. Die Erweiterung soll durch den Einsatz weiterer Entwurfsmuster modelliert und implementiert werden. Die Lösung sollte eine entsprechende Dokumentation enthalten. Das Interesse an dieser Aufgabe war recht groß, und dabei wurden verschiedene Lösungen vorgestellt. Sie umfassten z.B. die Erweiterung der Funktionalität in Form einer Vorschauansicht (z.B. für verschiedene Bildformate) mit Nutzung des Erbauer-(Builder-)Musters, Unterstützung zur Behandlung von versteckten Dateien unter Einsatz des Singleton-Musters, Suche nach doppelten Dateien (Memento-Muster), Sortierfunktion mit Hilfe des Decorator-Musters und schließlich eine Erweiterung um einen Undo/Redo-Mechanismus unter Einsatz der Muster Memento und Singleton, die im Folgenden ausführlicher beschrieben wird.

Um die gewünschte Funktionalität eines Undo/Redo-Mechanismus zu realisieren eignet sich das Memento-Muster. Weiterhin sollte es nur eine Instanz des Memento-Musters geben, damit sichergestellt wird, dass das zuletzt gespeicherte Memento eindeutig ist. Deswegen wurde zusätzlich das Singleton-Muster verwendet.

Das Memento-Muster dient im Allgemeinen dazu, einen Zustand eines Objekts zu speichern, so dass es nach einer Änderung wieder in den alten Zustand zurückversetzt werden kann. Das Muster besteht aus einem Urheber, einem Aufbewahrer und einem oder mehreren Mementos. Der Urheber verwaltet die Mementos, erzeugt diese und gibt diese weiter nach außen. Der Aufbewahrer fordert das Memento vom Urheber an und kommuniziert nur über den Urheber mit den Mementos. Der Aufbewahrer enthält ebenso alle Mementos. Das Singleton-Muster ist anzuwenden, wenn man sicherstellen will, dass es nur eine einzige Instanz einer Klasse geben darf. Ein von außerhalb geschützter Konstruktor und eine statische Methode, die die einzige Instanz wiedergibt, stellen diese Funktionalität bereit. Für die Verwendung des Memento-Musters musste dieses den Anforderungen der Erweiterung angepasst werden. Da es aufgrund der verschiedenen Aktionen wie Kopieren, Umbenennen oder Löschen und deren unterschiedlichen Anforderungen an das Memento auch mehrere unterschiedliche Mementos geben muss, dient hier das Memento als abstraktes Interface. Der Entwurf der Erweiterung ist in Abbildung 4 dargestellt.

Die speziellen Mementos implementieren das Interface Memento, damit diese vom Aufbewahrer, der Klasse MementoManager, gespeichert werden können. Als Datenstruktur wurde hier ein Keller (Stack) vom Datentyp Memento gewählt, da diese Struktur die chronologische Verwaltung der Mementos vereinfacht. Weiterhin wurden zwei Actions hinzugefügt, eine UndoAction und eine RedoAction, die je dafür Sorge tragen, eventuelle Undo- oder Redo-Mementos wiederherzustellen. Diese Actions gehören im Vergleich zu der Mustervorlage zu dem Urheber, da nur der Urheber direkt auf die Mementos zugreifen darf. Diese Actions setzen und erzeugen die Mementos. Weiterhin darf es nur eine Instanz des MementoManager (Aufbewahrer) geben, um den Kontrollfluss über die Mementos eindeutig zu halten. Daher wird der MementoManager mit Hilfe des Singleton-Musters erstellt. Diese Vorgehensweise hat ebenfalls den Vorteil, dass die Klassen UndoAction und RedoAction immer auf dieselbe Instanz des MementoManager zugreifen. Den Nutzerinnen und Nutzern des File Manager stellen sich die Mementos als durchgeführte Aktionen dar, die wieder rückgängig gemacht werden können.

Das geänderte Muster konnte durch die Studierenden in die durch den File Manager vorgegebene Struktur integriert werden. Dazu wurde die Logik des File Manager erweitert, indem jede existierende Funktion bei ihrem Aufruf ein entsprechendes Memento erstellt und dieses dem MementoManager übergibt. Ebenso wurde die grafische Benutzungsoberfläche um zwei Menüeinträge und zwei Schaltflächen für die Undo- und Redo-Funktion erweitert. Weiterhin erfolgten einige Änderungen an der bestehenden Logik, damit die Funktionalität der Mementos sichergestellt ist. Neben den neuen Methoden musste dabei beachtet werden, dass diese auch den vorher gegebenen Entwurfsmustern des File Manager genügen.

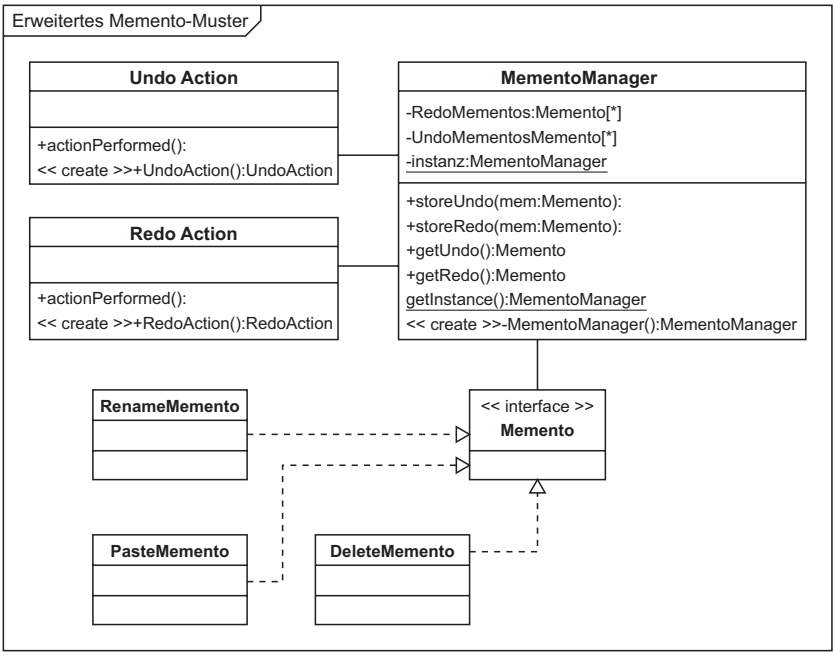


Abb. 4 *Abstraktes, angepasstes Memento- und Singleton-Muster*

3 Das anschließende Softwareprojekt

Das Softwareprojekt schließt sich im SE-Kurrikulum direkt an die Veranstaltung Software Engineering an und führt Wissen aus verschiedenen Grundstudiums-Veranstaltungen zur praktischen Anwendung. Diese Pflichtveranstaltung im Umfang von 4 SWS findet, wie in Abbildung 1 dargestellt, im 3. und 4. Semester statt und hat zum Ziel, Techniken der systematischen Softwareentwicklung im Team zu vermitteln. Regelmäßig findet diese Veranstaltung auch standortübergreifend statt [BHASV03].

Im Softwareprojekt entwickeln Gruppen mit einer Stärke von 8-12 Personen ein komplexes softwaretechnisches System, i.d.R. mindestens Client-/Server-basiert. Die Aufgabe ist dabei so komplex gewählt, dass sie nur durch die geschickte Aufteilung in Teilaufgaben effektiv gelöst werden kann. Auf diese Weise ist die Gruppe gezwungen, alle Teilnehmer in die Bearbeitung der Aufgabe zu involvieren. Die Gruppen werden durch einen Tutor unterstützt und durchlaufen alle klassischen Phasen der Softwareentwicklung in einem auf dem Wasserfallmodell basierendem Vorgehensmodell [LL06]. Neben der eigentlichen Softwareentwicklung müssen die Studierenden im ersten Semester basierend auf groben Vorgaben und im zweiten Semester selbstständig auch eine Projektplanung und -überwachung durchführen.

Die Veranstaltung beinhaltet einen Vorlesungsblock, Gruppensitzungen, Präsentationen sowie ein Proseminar. Im Vorlesungsblock zu Beginn des Semesters werden wesentliche, für die Bearbeitung der jeweiligen Aufgabenstellung notwendige Grundlagen, wie die Java-basierte Webentwicklung, vermittelt. Hierbei wird bewusst auf eine Wiederholung softwaretechnischer Grundlagen verzichtet und lediglich ein Handlungsrahmen vorgestellt, der beim eigentlichen Vorgehen helfen soll. Die Gruppen führen mindestens einmal pro Woche zusammen mit einem Tutor eine Sitzung durch, in der wichtige inhaltliche und organisatorische Entscheidungen getroffen, Vorträge zu Themen des Praktikum gehalten und Ergebnisse mit dem Tutor diskutiert werden. Ein weiterer wesentlicher Block ist das Einüben von Präsentationen. Jedes Mitglied der Gruppe kann sich zu Beginn des Semesters eine Spezialaufgabe aus einer Liste vordefinierter Aufgaben (Projektmanagement, Versionsverwaltung, Datenbank, GUI etc.) aussuchen und muss die Gruppe bezüglich dieses Themas beraten und einen Vortrag zu diesem Thema vorbereiten. Weiterhin findet eine Art Messe etwa nach 2/3 des Projekts statt, auf der die Gruppen ihre bis dahin vorliegenden Ergebnisse einem größeren Publikum im Rahmen des Hochschulinformationsstages vorstellen. Jeweils zum Abschluss eines Semesters erfolgt eine Abnahme des entwickelten Produkts.

Generell ergeben sich beim Softwareprojekt für die Studierenden Probleme, die sich im Wesentlichen auf mangelnde Erfahrung bei der Softwareentwicklung zurückführen lassen. So haben sie häufig Probleme, die komplexe Aufgabe in kleinere Teilaufgaben herunterzubrechen und entsprechende Abstraktionen durchzuführen. Es fehlt ein Gefühl dafür, was ein guter Entwurf ist. Kopplung und Kohäsion scheinen eine zu abstrakte Metrik zu sein. Eine Fassade war praktisch ein Fremdwort. Gerade bei diesen Problemen hat sich in den letzten beiden Durchläufen des Softwareprojekts eine entscheidende Verbesserung ergeben. Durch die stärkere Konzentrierung auf den Aspekt der Entwurfsmuster in der Software-Engineering-Veranstaltung fällt es den Studierenden deutlich leichter, eine sinnvolle Aufteilung und Kapselung von Modulen durchzuführen, was insgesamt zu deutlich besseren Entwürfen und einer effektiveren Untergliederung der Aufgabe führte. Das im nächsten Abschnitt vorgestellte InPULSE-System kann von den Studierenden kontinuierlich zum Selbststudium und zur Vertiefung genutzt werden.

4 Das E-Learning- und Assistenzsystem InPULSE

Das E-Learning- und Assistenzsystem InPULSE (*http://inpulse.uni-oldenburg.de*) wurde in den vom BMBF geförderten Projekten estat [MAHTZ02] und InPULSE entwickelt. Es dient zum einen als E-Learning-System, in dem Informationen zu Entwurfsmustern abrufbar sind. Zum anderen enthält es ein Assistenzsystem, das einen Dialog mit den Benutzern über ihr aktuelles Entwurfsvorhaben führen und anschließend Vorschläge zum Einsatz von Entwurfsmustern geben kann [GJMSV06]. Während des Dialogs werden den Benutzern sukzessiv Fragen zu ihrem Entwurf gestellt, um ihre Modellierungssituation zu erfassen. Abbildung 5 zeigt exemplarisch eine Seite aus dem System. Im linken Bildschirmbereich sind die vorhandenen Muster über eine Baumdarstellung zugriffsbereit. Im rechten Bereich wird über die Karteikarten-Reiter-Darstellung ein schneller Zugriff auf die einzelnen Elemente ermöglicht, die in allen Musterbeschreibungen vorhanden sind. Die Elemente sind an die Beschreibung der Entwurfsmuster nach [GHJV96] angelehnt. Besonderer Wert wurde bei den Inhalten darauf gelegt, dass sie für Software-Engineering-Anfänger geeignet und verständlich sind, dies gilt insbesondere für die enthaltenen Beispielanwendungen für die Muster. Um die Anwendung der Muster zu unterstützen, besteht die Möglichkeit, die Struktur des Musters und des Beispiels als XMI-Dateien im System zu hinterlegen. So können die Nutzer diese Diagramme in die von ihnen verwendeten CASE-Tools importieren und als Vorlage für ihren Entwurf verwenden. Ebenso können die Quelltexte der Beispiele als Java-Dateien heruntergeladen werden, die mit wenig Aufwand zu ablauffähigen Programmen vervollständigt werden können. Bei einer durchgeführten Evaluation des InPULSE-Systems mit Studierenden der Vorlesung Software Engineering wurden die Download-Möglichkeiten sowohl der Diagramme als auch des Quelltextes als positiv bewertet: »Die Muster sind gut erklärt/vorgestellt, vor allem da Diagramme und Beispielcode zur Verfügung stehen« und »Schön, übersichtlich und gute Beispiele sowie Beispielcode«.

Weiterhin bietet InPULSE die Möglichkeit, aus vorhandenen Elementen Kurse zusammenzustellen, z.B. speziell für unser Fallbeispiel. Lehrende können den Umfang und die Reihenfolge der Elemente festlegen, die von den Studierenden bearbeitet werden sollen. Um eine Adaption an die Lehrveranstaltung vorzunehmen, können diese Elemente im Kurs durch eigene veranstaltungsspezifische Elemente ergänzt werden. Zum Beispiel kann die Einbettung der einzelnen Muster in ein größeres Softwaresystem als Fallstudie durch zusätzliche Kurselemente gezeigt werden. So kann InPULSE zur gezielten Unterstützung einer Veranstaltung genutzt werden oder als freies Nachschlagewerk Projektarbeiten von Studierenden unterstützen.

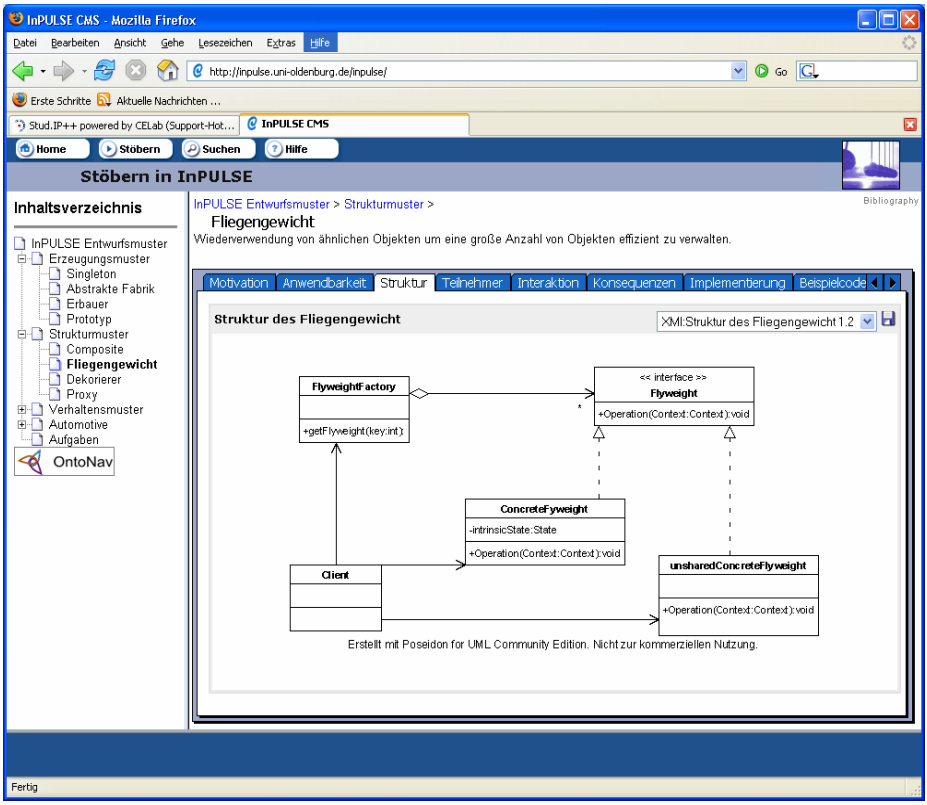


Abb. 5 *Screenshot von InPULSE*

5 Schlussbetrachtung

Die projektorientierte Vermittlung von Entwurfswissen in Form von Entwurfsmustern, die in einem nicht trivialen Beispielsystem umgesetzt und durch die Studierenden nachdokumentiert sowie erweitert werden, hat sich in der Informatikausbildung an der Universität Oldenburg bewährt. Insbesondere im Softwareprojekt und in Projektgruppen konnte festgestellt werden, dass die Studierenden über mehr Entwurfswissen verfügen, als dies zuvor der Fall war. Die Nutzung von Entwurfsmustern ist für die Studierenden zu einer Selbstverständlichkeit geworden.

Inzwischen haben wir den File Manager öffentlich über Sourgeforge zugänglich gemacht, so kann zukünftig in den Aufgaben im Lehrmodul Software Engineering auf *http://filemanager-uol.sourceforge.net/* verwiesen werden. Dies ermöglicht es den Studierenden, ihre Erweiterungen auch dort zu veröffentlichen und so Erfahrungen mit der Arbeit an Softwaresystemen in einem größeren Kontext zu gewinnen.

Der Inhalt in InPULSE kann für spezielle Veranstaltungen angepasst werden. Die Aufnahme umfassenden Kursmaterials zum File Manager in das InPULSE-Kurssystem kann das vorhandene Lehrangebot abrunden. Ein solcher Kurs würde die für den File Manager relevanten Entwurfsmuster einführen, deren Verwendung in einer in sich abgeschlossenen Anwendung aufzeigen und einen beispielhaften Anwendungskontext für das Dialogsystem bieten. Weiterhin werden in InPULSE neben den klassischen objektorientierten Entwurfsmustern [GHJV96] zurzeit auch Muster auf Architekturebene [Has06] für das E-Learning aufbereitet. Wir nehmen an, dass die Nutzung des Dialogsystems das Lernen der Entwurfsmuster unterstützt, da das System gezielt entwurfsrelevante Aspekte der Muster abfragt und so zur Reflexion über diese Aspekte und den eigenen Entwurf anregt. Für das nächste Jahr ist eine entsprechende Evaluation vorgesehen.

Literatur

[AHM99] K. Alfert, W. Hasselbring, A. Mester: Von Analyse und Entwurf zur Programmierung in einem Semester: UML, Java und deren Anwendung in Projektteams. In: Software Engineering im Unterricht der Hochschulen SEUH'99. Teubner, 1999.

[BHASV03] L. Bischofs, W. Hasselbring, H.-J. Appelrath, J. Sauer, O. Vornberger: Erste Erfahrungen mit dem Virtuellen Softwareprojekt. In: Software Engineering im Unterricht der Hochschulen SEUH'8. dpunkt.verlag, 2003.

[Bol06] D. Boles: Programmieren spielend gelernt mit dem Java-Hamster-Modell, 3. Auflage, Teubner, 2006.

[GHJV96] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1996.

[GJMSV06] H.Garbe, C. Janssen, C. Möbus, H. Seebold, H. de Vries: KARaCAs: Knowledge Acquisition with Repertory Grids and Formal Concept Analysis for Dialog System Construction. In: 15th International Conference on Knowledge Engineering and Knowledge Management – Managing Knowledge in a World of Networks (EKAW 2006), to appear 2006.

[Gud04] S. Gudenkauf: Entwicklung eines Anwendungsbeispiels für die Ausbildung im Software Engineering mittels EclipseUML. Individuelles Projekt, Carl von Ossietzky Universität Oldenburg, Department für Informatik, Abteilung Software Engineering, 2004.

[Has06] W. Hasselbring: Software-Architektur. *Informatik-Spektrum*, 29(1): 48–52, Februar 2006.

[LL06] J. Ludewig, H. Lichter: Software Engineering. dpunkt.verlag, 2006.

[MAHTZ02] C. Möbus, B. Albers, S. Hartmann, H.J. Thole, J. Zurborg: Towards a Specification of Distributed and Intelligent Web Based Training Systems. In: Intelligent Tutoring Systems, Proceedings of the 6th International Conference (ITS2002), Biarritz, France and San Sebastian, Spain, June, 2002, S. 291–300, Berlin: Springer, Lecture Notes in Computer Science, LNCS 2363.

Software Engineering moderner Anwendungen

Jürgen Rückert · Barbara Paech

Universität Heidelberg, Fakultät für Mathematik und Informatik, AG Software Engineering
Im Neuenheimer Feld 326, 69120 Heidelberg
{rueckert | paech}@informatik.uni-heidelberg.de

Zusammenfassung

An der Universität Heidelberg wurde die Veranstaltung »Software Engineering moderner Anwendungen: komponentenbasierte, serviceorientierte oder mobile Systeme« erstmals im Wintersemester 2005/2006 angeboten. Die innovative Idee zur Lehre war dabei die Entwicklung eines verteilten Software-Systems in vier Architekturen – mit Hilfe von vier verschiedenen Technologien. Einerseits konnten die Studierenden die Vor- und Nachteile der Architekturen durch Bewertung von nicht funktionalen Anforderungen diskutieren. Andererseits konnten sie praktische Erfahrung mit neuen Technologien sammeln, die für die Entwicklung von modernen Geschäftsanwendungen nützlich sind.

In diesem Beitrag stellen wir das Konzept und dessen Umsetzung, das begleitende Anwendungsbeispiel, die verwendeten Technologien und Werkzeuge sowie abschließend unsere Erfahrungen vor.

1 Einleitung

Die Vorlesung und Übung SWE 2B »Software Engineering moderner Anwendungen: komponentenbasierte, serviceorientierte oder mobile Systeme« wurde an der Universität Heidelberg erstmals im Wintersemester 2005/2006 angeboten. Studierende der anwendungsorientierten Informatik mit Abschluss Bachelor nehmen im 4. Fachsemester daran teil, Studierende mit Abschluss Master im 1. oder 2. Fachsemester. Die Wahlpflichtveranstaltung SWE 2B umfasst 6 Semesterwochenstunden (SWS), davon 3 SWS Vorlesung und 3 SWS Übungen. SWE 2B erfordert die Kenntnisse aus SWE 1 [PBR+05] [Häf05].

Ziel der Veranstaltung ist einerseits der Wissensaufbau über Architektursichten und -muster sowie die Bewertung von funktionalen und nicht funktionalen Anforderungen (NFR) zur Auswahl und Anpassung geeigneter Architekturen. Andererseits sollen die ausgewählten Architekturen umgesetzt werden, um praktische Kenntnisse über komponentenbasierte und serviceorientierte Systeme aufzubauen.

Kernidee unseres Vorgehens ist die Realisierung einer verteilten Anwendung mit Hilfe von vier Architekturen. Die Anwendung wird einmalig funktional spezifiziert. Die Architektur wird durch die jeweils eingesetzte(n) Technologie(n) bestimmt und durch die zu unterstützenden NFRs. Die Anwendung wird für AnwenderInnen zugänglich durch einen Web Client, der weitestgehend konstant gehalten wird, d.h., nur dessen Dialogkern, eine Komponente, die den Web Client mit einer Server-Komponente oder einem Web Service verbindet, muss an die jeweilige Technologie angepasst werden.

Schwerpunkte in den praktischen Übungen bilden Requirements Engineering, Entwurf (inklusive Architekturentwurf), Implementierung und Qualitätssicherung (Systemtest und Rationales).

Auf die Ausbildung von Soft Skills wird dadurch Wert gelegt, dass die Studierenden im Team arbeiten und die verschiedenen Ergebnisse präsentieren. Der überraschende Tausch von Ergebnissen zwischen den Teams, inklusive der anschließenden Weiterentwicklung von fremdem Quellcode, hat das Bewusstsein zur Entwicklung von qualitativ hochwertiger Software geschärft.

In Abschnitt 2 präsentieren wir Vorlesung und Übung im Detail: Die funktionalen Anforderungen werden in 2.1 vorgestellt, die vier verschiedenen Anwendungen kategorisieren wir in 2.2, den Ablauf der Vorlesung und Übung erklären wir in 2.3, nicht funktionale Anforderungen und Architekturen stellen wir in 2.4 vor, und die verwendeten Technologien und Werkzeuge nennen wir in 2.5. In Abschnitt 3 beschreiben wir unsere Erfahrungen. Abschließend geben wir in Abschnitt 4 eine Zusammenfassung und einen Ausblick für die Wiederholung der Veranstaltung im kommenden Wintersemester 2006/2007.

2 Vorlesung und Übung

2.1 Funktionale Anforderungen

Als Anwendungsbeispiel haben wir versucht, ein anschauliches Beispiel zu wählen: eine Anwendung »Auctions«, mit der AnwenderInnen an Auktionen mehrerer Auktionshäuser mitbieten können. AnwenderInnen sollen in der Rolle »AuktionsteilnehmerIn« in der Lage sein, einen Auktionsgegenstand zu ersteigern. Aktoren, Aufgaben, Nutzungsfälle und Systemfunktionen sind in Abbildung 1 dargestellt. Die beiden Nutzungsfälle »Kontaktiere ein Auktionshaus« und

»Nimm an einer Auktion teil« sollen durch eine Benutzungsschnittstelle grafisch zugänglich werden. Die Aufgabe »Verwalte Auktionen« muss dagegen nicht durch eine grafische Benutzungsschnittstelle unterstützt werden.

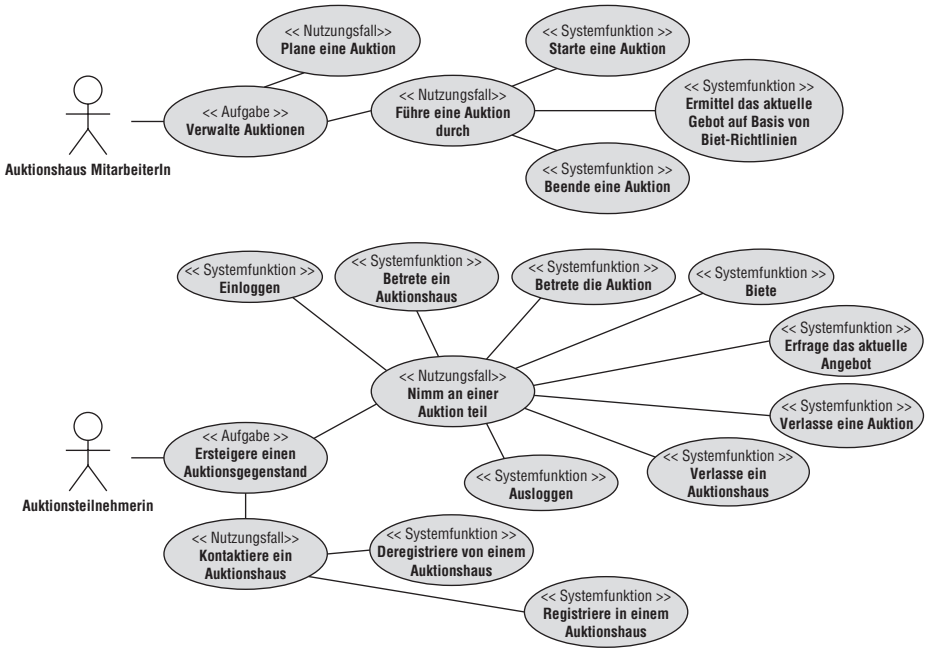


Abb. 1 *Nutzungs-Diagramm der Anwendung »Auctions«*

2.2 Anwendungen

Die in Abschnitt 2.1 genannten funktionalen Anforderungen sind fix und sollen in vier vorgegebenen Technologien umgesetzt werden. Es ist die Aufgabe der Studierenden, die unterschiedlichen NFRs herauszuarbeiten und die sich daraus ergebende Architektur abzuleiten:

Anwendung 1 mit Server-Komponente auf Basis der Java 2 Standard Edition [J2SE50]

Anwendung 2 mit Server-Komponente auf Basis der Java 2 Enterprise Edition [J2EE14]

Anwendung 3 mit Web Service auf Basis von Apache AXIS [Axis13]

Anwendung 4 mit Web Service auf Basis der Java 2 Enterprise Edition [J2EE14]

2.3 Ablauf

In einer wöchentlich stattfindenden Vorlesung (an zwei Tagen: 1 SWS plus 2 SWS) wird den Studierenden das Fachwissen vermittelt, das die Basis bildet für die Übungsstunden (3 SWS). Ein durchgehender Projektkontext in den Übungen wird durch die Verwendung eines einzigen Anwendungsbeispiels (siehe 2.1) geschaffen. Die TeilnehmerInnen der Veranstaltung werden für die Übung in Teams (in diesem Fall 4 Teams à 4 Personen) eingeteilt. Alle Aufgaben werden in Teamarbeit gelöst, da neben technischen und fachlichen Erkenntnissen der Umgang mit organisatorischen und sozialen Problematiken erlernt werden soll [FSN05]. Die Studierenden lernen, miteinander zu kommunizieren und gemeinsam schwierige, komplexe Aufgaben zu verteilen und zu lösen. Der Aufbau der Entwicklungsumgebung erfolgt jedoch individuell. Tabelle 1 gibt einen Überblick über die Vorlesungs- und Übungsinhalte. Erläuterungen zu Tabelle 1 folgen im Anschluss.

	Vorlesung	**Übung**
1	Architektur-sichten, -modellierung, -muster	**Projektmanagement**: Fragebogen und Teambildung **Projektmanagement**: Aufbau der Entwicklungsumgebung
		Anwendung: Spezifikation der Funktionalität (Aktoren, Aufgaben, Nutzungsfälle, Systemfunktionen) auf Basis einer vorgegebenen Problembeschreibung **Anwendung**: Spezifikation des funktionalen Systemtests (logische und konkrete Testfälle)
		Web Client: Spezifikation der Funktionalität (Arbeitsbereiche, Sichten, Sichten-Übergänge)
2	Komponenten	**Anwendung J2SE**: Spezifikation von NFRs **Anwendung J2SE**: Entwurf der Architektur **Anwendung J2SE**: Diskussion der Architektur anhand der NFRs **Anwendung J2SE**: Diskussion der Architektur anhand von Architektursichten
		J2SE-Server: Entwurf der Server-Schnittstelle **J2SE-Server**: Entwurf des Servers
		Web Client: Entwurf der Architektur und des J2SE-Dialogkerns (Komponenten, Klassen, Pakete) **Web Client**: Implementierung
3	Verteilte Systeme, Middleware, Enterprise-Architekturen	**J2SE-Server**: Implementierung
		Anwendung: Spezifikation des nicht funktionalen Systemtests (hauptsächlich für Usability, Sicherheit und Plattform-Unabhängigkeit) **Anwendung**: Implementierung (Aufzeichnung) des Systemtests mit Hilfe des Capture-and-Replay-Werkzeugs [MaxQ098]
		Anwendung J2SE: Ausführung des Systemtests **Anwendung J2SE**: Aktualisierung der Spezifikation und des Entwurfs nach Implementierung und Fehlerbeseitigung

	Vorlesung	Übung
4	CORBA	**Anwendung J2EE**: Spezifikation von NFRs **Anwendung J2EE**: Entwurf der Architektur
		J2EE-Server: Entwurf der Server-Schnittstelle
		Web Client: Entwurf des J2EE-Dialogkerns des Web Clients
5	J2EE	**J2EE-Server**: Entwurf des Servers **J2EE-Server**: Implementierung 1 (ohne Persistenz)
		Anwendung: Aktualisierung der Spezifikation des funktionalen Systemtests **Anwendung**: Aktualisierung der Spezifikation des nicht funktionalen Systemtests
		Anwendung J2EE: Implementation eines Application Clients
6	.NET	**J2EE-Server**: Implementierung 2 (mit Container-Managed Persistenz)
		Web Client: Implementierung des J2EE-Dialogkerns des Web Clients
		Anwendung J2EE: Ausführung des Systemtests auf Basis CMP **Anwendung J2EE**: Aktualisierung des Entwurfs nach Implementierung
7	NFR, Rationale	**Anwendungen J2SE und J2EE**: Aktualisierung der Spezifikation und des Entwurfs
		J2EE-Server: Implementierung 3 (mit Bean-Managed Persistenz)
		Anwendung J2EE: Ausführung des Systemtests auf Basis BMP
		Projektmanagement: Präsentation der J2SE- und J2EE-Anwendungen

8	CodeCamp

9	Web Services	**Anwendung AXIS**: Entwurf der Architektur
		AXIS-Service: Entwurf der Server-Schnittstelle nach der Contract-first-Methode **AXIS-Service**: Implementation eines Java Web Service (JWS) nach der Implement-first-Methode
10	XML, SOAP	**Web Service Client**: Diskussion von Client-Entwicklungsarten **Web Service Client**: Implementation eines Google Web Service Client **Web Service Client**: Implementierung eines Dynamic Invocation Client für JWS
		AXIS-Service: Implementierung des Web Service nach der Contract-first-Methode
11	WSDL, UDDI	**UDDI Client**: Entwurf **UDDI Client**: Implementierung
		Anwendung AXIS: Implementierung des AXIS-Dialogkerns des Web Clients **Anwendung AXIS**: Ausführung des Systemtests

	Vorlesung	Übung
12	Orchestrierung	**Projektmanagement**: Tausch des Quellcodes
		Anwendung J2EEWS: Diskussion von Architekturoptionen **Anwendung J2EEWS**: Entwurf der Architektur **Anwendung J2EEWS**: Implementierung
13	Sicherheit	**Anwendung J2EE**: Diskussion der Architekturänderungen durch EJB 3.0
		Anwendung J2EEWS: Implementierung des J2EEWS-Dialogkerns des Web Clients **Anwendung J2EEWS**: Ausführung des Systemtests
14	Semantic Web	**Projektmanagement**: Abschlussfragebogen **Projektmanagement**: Abschlusspräsentationen der AXIS- und J2EE-basierten Web-Service-Anwendungen **Prüfungen**

Tab. 1 *Inhalt von Vorlesung und Übung pro Semesterwoche*

- In *Woche 1* beginnt die Spezifikation der Funktionalität der Anwendung.
- In *Woche 3* ist die *Anwendung 1* fertig implementiert.
- In *Woche 4* beginnt die Entwicklung der *Anwendung 2*. Basierend auf NFRs wird die Architektur entworfen, wie auch die Schnittstelle des J2EE-Servers, die sich aufgrund des J2EE-Rahmenwerks von der J2SE-Schnittstelle technisch sehr unterscheidet (fachlich dagegen nicht).
- In *Woche 5* wird der J2EE-Server entworfen und implementiert. Die einzelnen (prinzipiell verteilbaren) Enterprise JavaBeans (EJB) [EJB21] werden von einem lokal installierten JBoss Application Server [JBossAS403] instanziiert. Aufgrund der Veränderung der NFRs gegenüber der *Anwendung 1* und der Schnittstelle des J2EE-Servers wird die Spezifikation der Systemtestfälle auf den neuesten Stand gebracht. Um die J2EE-Technologie besser zu erlernen, d.h. konkret den Aufruf von EJBs, hat es sich bewährt, zuerst einen »standalone Application Client« zu entwickeln, der den gesamten Nutzungsfall »Nimm an einer Auktion teil« ohne Interaktion des Benutzers ablaufen lässt.
- In *Woche 6* wird die Implementierung des J2EE-Servers um Persistenz erweitert, zunächst mit der Container Managed Persistence (CMP) [EJB21, Kapitel 10].
- In *Woche 7* wird der J2EE-Server mit Hilfe der Bean-Managed Persistence (BMP) [EJB21, Kapitel 12] implementiert und getestet, um auch diese Technologie kennenzulernen.
- In *Woche 8* präsentieren die Teams ihre beiden Lösungen in je einer halben Stunde im Rahmen eines »CodeCamps«: Spezifikation, Entwurf, Implementierung (Quellcode), Systemtest (Spezifikation und Ausführung). Hier wird ausführlich Feedback gegeben, damit Irrtümer bei den beiden letzten Anwendungen nicht wiederholt werden. Ab Woche 12 werden die Teams auf Basis

des Quellcodes eines anderen Teams arbeiten. Das CodeCamp dient somit zum gezielten Auffrischen der Kenntnisse über die Implementierung des »neuen« Systems.

- In *Woche 9* beginnt die Entwicklung der ersten Web-Service-Anwendung (*Anwendung 3*). Die Schnittstelle des Web Service wird mit der Contract-first-Methode entworfen, d.h., es entsteht eine Beschreibung mit WSDL [WSDL11]. Gleichzeitig wird der für *Anwendung 1* entwickelte J2SE-Server als Java Web Service (JWS) [Axis13, User Guide] gekapselt. Dies dient zum Kennenlernen der Implement-first-Methode und der AXIS SOAP Engine selbst.
- In *Woche 10* werden die NFRs verschiedener Typen von Web Service Clients (static stub, dynamic invocation) bewertet. Der allererste Web Service Client wird für einen produktiven Web Service im Internet (Google Web Service) entwickelt, und nicht für einen selbst entwickelten Web Service, um die tatsächliche Anwendbarkeit von Schnittstellenbeschreibungen aufzuzeigen.
- In *Woche 11* wird ein UDDI-Client [UDDI2] entworfen und implementiert. Dabei werden öffentlich zugängliche UDDI-Registraturen verwendet.
- In *Woche 12* tauschen die Teams überraschend die entwickelten Spezifikationen sowie die entwickelte Software und beginnen, die *Anwendung 4* zu realisieren.
- In *Woche 13* wird die *Anwendung 4* vervollständigt und die Vor- und Nachteile der neuen EJB 3.0-Technologie [EJB30] diskutiert, um zumindest einen kurzen Blick auf diese neue Komponententechnologie zu werfen.
- In *Woche 14* präsentieren die Teams die beiden *Anwendungen 3* und *4*, d.h. Spezifikation, Entwurf, Quellcode und Systemtest (Spezifikation und Ausführung).

2.4 Nicht funktionale Anforderungen und Architekturen

Neben den Technologien sollen die Studierenden vor allem die Auswirkungen von Architekturentscheidungen verstehen lernen.

Dies wird zum einen durch die Variation der Architektur erreicht. Die vom System automatisch ausgeführten Aufgaben, d.h. Anwendungslogik und Datenhaltung, ermöglichen entweder Server-Komponenten (*Anwendungen 1* und *2*) oder Web Services (*Anwendungen 3* und *4*). Der Web Client wird zuerst entwickelt und bleibt weitestgehend konstant. Weitestgehend deshalb, da der Web Client an die jeweilige Technologie angepasst werden muss. Dabei wird aber nur die Dialogkern-Komponente ausgetauscht, die den Web Client an eine Server-Komponente oder an einen Web Service anbindet. Die Architektur der Web-Oberfläche basiert auf [Sie05]. Die Server-Komponenten und Web Services werden in der jeweiligen Technologie nach und nach entwickelt.

Zum anderen werden die Auswirkungen der Architekturentscheidungen durch ständige Diskussion der NFRs verdeutlicht. Tabelle 2 zeigt, welche NFRs bei welchen architektonischen Komponenten eine Rolle spielen. Lernziel hierbei ist die Erkenntnis, in welchem Maße sich eine bestimmte Technologie eignet zur Erfüllung bestimmter NFRs. Wird »Persistenz« gefordert, so hat man die Möglichkeit, einen J2SE-Server auf Basis JDBC zu implementieren, oder einen J2EE-Server auf Basis von EJBs. Zur Auswahl kann das NFR »Transparente Einbindung von Persistenz (kein SQL)« dienen, das dann den Ausschlag für den J2EE-Server gibt. Die gleichzeitige Forderung von »Verteilung« und »Interoperabiliät« schließt die Verwendung der J2SE- und J2EE-Technologie (d.h. RMI und EJB) aus und resultiert in einer der Web-Service-Technologien (d.h. SOAP). In Tabelle 2 sind die NFRs der obersten Abstraktionsstufe dargestellt, diese werden von den Studierenden zu einer Vielzahl von NFRs auf Ebene von u.a. Nutzungsfall, Systemfunktion und Benutzungsschnittstelle verfeinert.

	J2SE-Server (Anw. 1)	J2EE-Server (Anw. 2)	AXIS Web Service (Anw. 3)	J2EE Web Service (Anw. 4)
Interoperabilität			✓	✓
Logging	✓	✓	✓	✓
Multi-User	✓	✓	✓	✓
Persistenz	✓	✓		✓
Sicherheit		✓	✓	✓
Verteilung	✓	✓	✓	✓

Tab. 2 *Zusammenhang zwischen nicht funktionalen Anforderungen und architektonischen Komponenten*

2.5 Technologien und Werkzeuge

Wir unterscheiden die Technologien und Werkzeuge, die zur Realisierung der vier Architekturen dienen, von denen, die zur ständigen Qualitätssicherung eingesetzt werden. Die verwendeten Technologien und Werkzeuge sind in Tabelle 3 dargestellt.

	Technologien	Werkzeuge
Web Client	■ HTML oder XHTML ■ Java Servlets [JST24] oder Java Server Pages [JSP20]	■ Apache Tomcat [Tomcat55] ■ MAXQ für Capture-Replay-Systemtest [MaxQ098] ■ NVU als HTML-Editor [NVU10]
J2SE-Server	■ Java 2 Standard Edition (J2SE) 5.0 und Dokumentation [J2SE50]	■ Eclipse 3.1 [Eclipse31]
J2EE-Server	■ Java 2 Enterprise Edition (J2EE) 1.4 und J2EE Tutorial [J2EE14] ■ JBoss 4.0 [JBossAS403]	■ Eclipse JBoss IDE 1.5 [JBossIDE15]
AXIS Web Service	■ Java 2 Standard Edition 5.0 (J2SE) [J2SE50] ■ AXIS 1.3 [Axis13]	■ Eclipse 3.1 WTP [Eclipse31] ■ AXIS SOAP Monitor [Axis13, User Guide, Appendix]
J2EE Web Service	■ Java 2 Enterprise Edition 1.4 [J2EE14] ■ JBoss 4.0 [JBossAS403]	■ Eclipse JBoss IDE 1.5 [JBossIDE15]

Tab. 3 *Zur Realisierung der Anwendungen und zur Qualitätssicherung verwendete Technologien und Werkzeuge*

Als Entwicklungsumgebung dienen zwei Eclipse-Plattformen (in der Ausprägung 3.1 WTP und in der Ausprägung JBoss IDE 1.5), die beide den Vorteil haben, die Entwicklung von J2EE-Anwendungen und Web Services durch spezielle Plug-ins zu erleichtern. Zur Versionskontrolle der kollaborativ entwickelten Software wird CVS der Vorzug gegenüber Subversion gegeben, da das Subversion Eclipse Plug-in »Subclipse« nicht stabil genug zur Verfügung steht. Zur Erstellung von UML-Diagrammen dient das UML-Modellierungswerkzeug JUDE [JUDE251]. Anforderungen werden mit Hilfe des Werkzeugs Sysiphus [Sysiphus] dokumentiert, genauer gesagt unter Verwendung dessen Web-Oberfläche »REQuest«.

3 Erfahrungen

Kenntnisstand:
TeilnehmerInnen waren Studierende der anwendungsorientierten Informatik (Bachelor und Master), Mathematik, Computerlinguistik, Physik, Geografie und des wissenschaftlichen Rechnens. Darunter erfahrene Software Engineering 1 [PBR+05] Teilnehmer und Studienortwechsler von anderen Universitäten, Fachhochschulen und Berufsakademien, bei denen Wissen über Software Engineering teilweise nur gering vorhanden war – diese Lücke musste durch eine geeignete Teameinteilung ausgeglichen werden. Ebenso verhielt es sich mit Programmierkenntnissen – im Vorfeld wurden die Erfahrungen und Kenntnisse durch einen Fragebogen erfasst, damit die Teams gleichmäßig eingeteilt werden konnten. Programmierkenntnisse wurden dennoch in den ersten zwei Wochen des Semesters nachgeschult in zwei Vorlesungen zu Java und Java-Web-Anwendungen und in

ergänzenden Hausaufgaben, die individuell korrigiert und besprochen wurden. Im Laufe des Semesters konzentrierten sich die TeilnehmerInnen mit geringeren Programmierkenntnissen auf die Spezifikation mit Hilfe von [Sysiphus] und die Synchronisation von Entwurf und Quellcode.

Ablauf der Übungen:
Die Übung bestand jeweils aus drei Teilen: Im ersten Teil wurde in einem Vortrag des Übungsleiters die jeweils neuen Technologien und Werkzeuge vorgestellt (Java, Java-Web-Anwendungen, Java 5 Neuheiten, Eclipse, J2EE-Details, JBoss, AXIS, Build und Deployment). Im zweiten Teil wurde das neue Aufgabenblatt erklärt. Im dritten Teil haben die Teams mit der besten Lösung ihre Ergebnisse präsentiert.

Anwendung und Web Client:
Die Anforderungsspezifikation der Anwendung und die Entwicklung des Web Clients stellten einen angenehmen Einstieg zu Beginn des Semesters dar, da die Studierenden die vorausgesetzten Kenntnisse von SWE 1 entweder auffrischen oder punktuell neu aufbauen konnten. Gleichzeitig konnte die Vorlesung in der Zeit neues Wissen vermitteln, das ja erst bekannt sein muss, bevor es in den Übungen vertieft werden kann.

J2SE-Server:
Die Entwicklung eines nicht persistierenden J2SE-Servers hat sich aufgrund der frühzeitigen Modellierung der Web-Client-Schnittstelle bewährt. Diese Schnittstelle kann bei den folgenden Anwendungen als Startpunkt dienen. Allerdings kann diese Schnittstelle nicht vollständig wiederverwendet werden, da AXIS Web Services keine Java-Kollektionen unterstützen. Die gesamte Anwendung ist zudem schnell lauffähig. Der Systemtest ist damit frühzeitig implementier- und ausführbar und kann bei nachfolgenden Anwendungen als Regressionstest wiederverwendet werden. Eine frühzeitige, genaue Kontrolle und Korrektur der Studierenden-Ergebnisse ist sehr zu empfehlen, da falsche Entwurfsentscheidungen aufwendige Änderungen der Schnittstelle bei Verwendung anderer Technologien nach sich ziehen.

J2EE-Server:
Diese Technologie hat sich als sehr umfangreich und schwer zu erlernen dargestellt, insbesondere die Verwendung der Bean Managed Persistence, die der Vollständigkeit halber mit einbezogen wurde. Hierbei ist eine individuelle, sehr zeitaufwendige Betreuung notwendig.

Web Services:
Die Verwendung der einstufigen Implement-first-Methode und der zweistufigen Contract-first-Methode hat sich in dieser Reihenfolge nicht bewährt, da die Studierenden mit Hilfe von Werkzeugen den Contract aus der Implementierung

generierten und diesen nicht mehr unbefangen modellierten, insbesondere die Datenstrukturen. In der Zukunft ist der Vorrang eindeutig der Contract-first-Methode zu geben, damit die Modellierung in WSDL [WSDL11] gründlich erlernt wird.

Technologien:
Die Entwicklung einer verteilten Anwendung in mehreren Technologien mit mehreren Werkzeugen erfordert eine genaue Abstimmung der Versionen aller eingesetzten Software-Produkte. Diese Abstimmung muss im Vorfeld herausgefunden werden und sollte den Studierenden als Gesamt-Installationspaket (in Form einer CD) zur Verfügung gestellt werden, damit durch Verwendung eigener Downloads der Lernerfolg durch inkompatible Versionen nicht verhindert wird. Die xdoclet-Technologie [XDoclet], die zur Entwicklung aller vier Anwendungen verwendet werden kann, zur Entwicklung von *Anwendung 3* sogar unentbehrlich ist, hat sich als schwer zu erlernend herausgestellt, da wenig gute Dokumentation vorhanden ist.

Betreuungsaufwand:
Der Betreuungsaufwand ist außerordentlich hoch und steigt linear mit der Anzahl Teams, da pro Team verschieden granulare Spezifikationen und vor allem sehr unterschiedliche Entwürfe und Implementierungen entstehen. Aufgrund der neuen Technologien und Werkzeuge ist individuelle Unterstützung, per E-Mail oder durch Vorführungen, unumgänglich. Die 16 TeilnehmerInnen wurden durch den Übungsleiter und zwei wissenschaftliche Hilfskräfte betreut.

Feedback:
Die Studierenden zeigten sich bis zum Schluss sehr motiviert. In einem Abschlussfragenbogen wurde als Ursache ermittelt, dass sie umfangreiche theoretische Kenntnisse erhalten haben und gleichzeitig Erfahrung mit vielen aktuellen Technologien sammeln bzw. vertiefen konnten. Die meisten neuen Erkenntnisse haben die Studierenden bei Entwicklung der J2EE-Anwendung (aufgrund Enterprise JavaBeans) und der AXIS-Anwendung (aufgrund Web Services) erhalten. Der theoretische Vorlesungsanteil in Verbindung mit praktischen Übungen wurde als sehr lehrreich beurteilt. Insbesondere die Reihenfolge der Vorlesungsinhalte, die Abfolge der Anwendungen und die Synchronisation von Vorlesung und Übung wurden gelobt. Die Einarbeitung in den Quellcode des anderen Teams hat viel Arbeit verursacht. Ebenso viel Zeit hat die Erstellung von Konfigurationen zum Build und Deployment jeder Anwendung gekostet. Durch die Entwicklung mehrerer Anwendungen hatten die Studierenden mehrere Erfolgserlebnisse.

4 Zusammenfassung und Ausblick

In diesem Beitrag haben wir unser Vorgehen zur Lehre von modernen Technologien und Architekturen vorgestellt. Innerhalb eines Semesters haben wir Konzepte über verteilte Systeme und Architekturen vermittelt, diese in Zusammenhang mit Software-Engineering-Methoden gebracht und in einem Forward-Engineering praktisches Entwicklungswissen über J2EE- und Web Service basierte Geschäftsanwendungen weitergegeben.

Die Lehrveranstaltung wird im Wintersemester 2006/2007 wiederholt. Da sich in der Zwischenzeit die Technologien weiterentwickelt haben, werden wir die neue Komponententechnologie Enterprise JavaBeans 3.0 [EJB30] einsetzen und die Orchestrierung von Web Services mit BPEL [BPEL11] einüben. Damit der Entwicklungsaufwand zur Anpassung des Web Client an die unterschiedlichen Technologien von Server-Komponenten und Services entfällt, kann ein durch Konfiguration adaptierbarer Dialogkern eingesetzt werden, der in [RHN+06] vorgestellt wurde. Weiterhin wollen wir nicht funktionale Anforderungen verstärkt modellieren und zur Bewertung der Architekturoptionen einsetzen.

Wir danken Jerko Horvat und Nick Meier für die Unterstützung der Übung und allen TeilnehmerInnen für die konstruktive Mitarbeit.

Literatur

[Axis13] SOAP Engine Apache AXIS 1.3 Final. Apache Web Service Project, 5 October 2005. *http://ws.apache.org/axis*

[BPEL11] Business Process Execution Language (BPEL) 1.1. IBM, May 2003. *http://www-128.ibm.com/developerworks/library/specification/ws-bpel*

[Eclipse31] Eclipse Platform 3.1 inklusive Web Tools Project (WTP) Plug-in. *http://www.eclipse.org* und *http://www.eclipse.org/webtools*

[EJB21] Enterprise JavaBeans Technologie 2.1 Final Release. Sun Microsystems, 24 November 2003. *http://java.sun.com/products/ejb*

[EJB30] Enterprise JavaBeans Technologie 3.0 Proposed Final Draft. Sun Microsystems, December 2005. *http://java.sun.com/products/ejb*

[FSN05] A. Fleischmann, K. Spies, K. Neumeyer: Teamtraining für Software-Ingenieure. Software Engineering im Unterricht der Hochschulen. Aachen, 2005.

[Häf05] P. Häfele: Softwareentwicklung mit dem TRAIN-Prozess. Bachelor-Arbeit. Universität Heidelberg, AG Software Engineering, 2005. *http://www-swe.informatik.uni-heidelberg.de/research/publications/TRAIN.pdf*

[J2EE14] Java Platform Enterprise Edition (J2EE). Version 1.4. Sun Microsystems. *http://java.sun.com/javaee*

[J2SE50] Java Platform Standard Edition (J2SE). Version 5.0. Sun Microsystems. *http://java.sun.com/javase*

[JBossAS403] JBoss Application Server 4.0.3, Service Package 1. JBoss, 24 October 2005. *http://labs.jboss.com/portal/jbossas*

[JBossIDE15] JBoss IDE für die Eclipse Plattform 1.5.0. JBoss, October 2005. *http://labs.jboss.com/portal/jbosside*

[JSP20] JavaServer Pages Technologie. Version 2.0. Sun Microsystems. *http://java.sun.com/products/jsp*

JST24] Java Servlet Technologie. Final Release 2.4. Sun Microsystems, 24 November 2003. *http://java.sun.com/products/servlet*

[JUDE251] UML Modellierungswerkzeug JUDE/Community 2.5.1. Change Vision, October 2005. *http://jude.change-vision.com/jude-web*

[MaxQ098] MaxQ Capture-and-Replay Testwerkzeug für HTTP-Clienten. Tigris.org (Open Source Software Engineering Tools), October 2005. *http://maxq.tigris.org*

[NVU10] NVU HTML-Editor 1.0. Linspire Inc., 28 June 2005. *http://www.nvu-composer.de*

[PBR+05] B. Paech, L. Borner, J. Rückert, A.H. Dutoit, T. Wolf: Vom Kode zu den Anforderungen und zurück: Software Engineering in 6 Semesterwochenstunden. Software Engineering im Unterricht der Hochschulen. Aachen, 2005.

[RHN+06] J. Rückert, J. Horvat, D.-K. Nguyen, S. Becker, B. Paech: Modell zur Adaption eines Dialogkerns. Modellierung 2006, Workshop »Qualität von Modellen«. Innsbruck, 2006.

[Sie05] J. Siedersleben. Moderne Software-Architektur. Umsichtig planen, robust bauen mit Quasar. dpunkt.verlag, August 2004.

[Sysiphus] Sysiphus Requirements-Engineering-Werkzeug. Build October 2005. *http://wwwbruegge.in.tum.de/Sysiphus*

[Tomcat55] Apache Tomcat 5.5.12. Apache, October 2005. *http://tomcat.apache.org*

[UDDI2] Universal Description, Discovery and Integration Specification (UDDI) Version 2. Organization for the Advancement of Structured Information Standards (OASIS). *http://www.oasis-open.org*

[WSDL11] Web Services Definition Language (WSDL) 1.1. W3C Note 15 March 2001. *http://www.w3.org/TR/wsdl*

[XDoclet] XDoclet Open Source Generation Engine. *http://xdoclet.sourceforge.net/xdoclet*

Software-Engineering-Simulation als Brücke zwischen Vorlesung und Praktikum

Tilmann Hampp · Stefan Opferkuch

Abteilung Software Engineering, Institut für Softwaretechnologie, Universität Stuttgart
Universitätsstraße 38, 70569 Stuttgart
{hampp|opferkuch}@informatik.uni-stuttgart.de

Zusammenfassung

In der Ausbildung angehender Projektleiter müssen nicht nur theoretische Kenntnisse vermittelt werden, sondern auch deren praktische Anwendung. Die Kombination aus Vorlesung und Praktikum vermittelt die Theorie und verlangt ihre Umsetzung. Sie erlaubt es aber nicht, die Zusammenhänge zwischen den theoretischen Grundlagen und der Umsetzung nachzuvollziehen. Diese Schwäche lässt sich ausgleichen, indem während der Ausbildung eine Projektsimulation mit detaillierter Auswertung durchgeführt wird. In diesem Beitrag zeigen wir, wie die simulierten Projekte ausgewertet werden, stellen Beispielauswertungen vor und diskutieren den dadurch erzielten Lernerfolg.

1 Einleitung und Motivation

Der Erfolg eines Softwareentwicklungsprojekts wird maßgeblich durch die Projektleitung bestimmt. Empirische Untersuchungen [Lic98] [Aho03] zeigen typische Fehler von (angehenden) Projektleitern: Die Planung ist unzureichend, Qualitätssicherung und Fortschrittskontrolle werden vernachlässigt, Termine und Budget überschritten. Daraus ergibt sich der Bedarf für eine Verbesserung der Projektleiter-Ausbildung.

1.1 Projektleiter-Ausbildung durch Vorlesungen

Die für die Leitung eines Softwareprojekts erforderlichen theoretischen Kenntnisse lassen sich durch Vorlesungen vermitteln. Vorteilhaft dabei ist, dass in Vorlesungen einer großen Anzahl von Teilnehmern Inhalte vermittelt werden können und sich die Vorlesungen jedes Semester einfach wiederholen lassen. Für die Projektleiter-Ausbildung haben diese konventionellen Lehrveranstaltungen jedoch Nachteile: Die praktische Anwendung der Inhalte muss geübt werden. Die Studierenden haben eine unrealistische Vorstellung von der Projektarbeit, weil diese durch Programmieren im Kleinen geprägt ist und nicht reale Softwareprojekte widerspiegelt.

1.2 Projektleiter-Ausbildung durch praktische Lehrveranstaltungen

Praktische Projekte in der Ausbildung ermöglichen es dagegen, theoretische Kenntnisse anzuwenden. Dazu muss die Problemstellung realistisch sein: Die Projekte müssen so umfangreich sein, dass die Arbeit nur im Team geleistet werden kann. Planung, Qualitätssicherung, Projektleitung und Organisation dürfen nicht vorgegeben werden, damit die Teilnehmer die typischen Schwierigkeiten kennenlernen und dadurch eine deutlichere und realistischere Vorstellung von Softwareprojekten und ihren Problemen gewinnen. Dabei ist nachteilig, dass nur ein Teilnehmer die Rolle des Projektleiters übernehmen kann. Eine Rotation der Rolle des Projektleiters zwischen den Teilnehmern während des Projekts hat sich nicht bewährt.

Der Dozent steht aufgrund der komplexen Zusammenhänge in Projekten vor Problemen: Er kann den Erfolg nicht selbst herbeiführen oder erzwingen, sondern höchstens beratend eingreifen. Nachträglich kann das Vorgehen kaum analysiert werden, weil sich der Zusammenhang zum Projektergebnis nicht nachvollziehen lässt und durch andere Einflüsse überlagert wird. Für eine Analyse wäre eine Art gläsernes Projekt notwendig, so dass der Dozent die einzelnen Handlungen des Projektleiters und ihre Auswirkungen identifizieren kann. Selbst wenn Fehler im Vorgehen erkannt werden, kann das Projekt nicht wiederholt werden. Die Teilnehmer haben keine Möglichkeit, diese Fehler zu vermeiden und zu sehen, wie sich die Ergebnisse verbessern.

1.3 Projektleiter-Ausbildung durch Simulation

Die Nachteile der praktischen Lehrveranstaltungen versuchen wir an der Universität Stuttgart im Diplomstudiengang Softwaretechnik [Lud99] durch die Simulation von Softwareprojekten mit dem SESAM-System (Software Engineering durch Animierte Modelle) auszugleichen [Lud94]. Simuliert werden Softwareprojekte von der Analyse bis zur Übergabe an den Kunden mit allen für die Aus-

bildung relevanten Aspekten. Ausnahme ist der Projektleiter, der nicht simuliert wird, sondern das Projekt am Simulator steuert und kontrolliert. Modelliert werden kleine und mittelgroße Projekte, die ein Kunde als Auftrag für ein Informationssystem vergibt [Dra98].

Die Simulation bietet den wichtigsten Vorteil der praktischen Arbeit, indem sie es ermöglicht, die Theorie anzuwenden. Sie vermeidet die Nachteile praktischer Lehrveranstaltungen, weil jeder Teilnehmer sein eigenes Projekt leitet. Der Dozent kann die Stärken und Schwächen im Vorgehen des Projektleiters analysieren und die Auswirkungen auf das Projektergebnis zeigen. Diese Analyse basiert auf den im Simulationsmodell enthaltenen Zusammenhängen in Softwareprojekten.

Damit Simulationen mit SESAM auf die Realität übertragbar sind, müssen die enthaltenen Zusammenhänge valide sein. Sie sind durch Literatur belegt, die Quantifizierung erfolgte mit empirischen Daten, vorwiegend aus [Jon96]. Weil kaum andere zuverlässige empirische Daten veröffentlicht wurden, ist das Modell auch gegen das Kostenschätzmodell Cocomo II [Boe00] validiert worden. Die Plausibilität wurde durch Vergleich von Simulationsergebnissen mit unterschiedlichem Vorgehen geprüft.

2 Schulungen mit dem SESAM-System

[Dra98] untersuchte, ob sich Studierende durch die Leitung simulierter Projekte verbessern. Dabei zeigte sich, dass die Zusammenhänge zwischen dem Vorgehen des Projektleiters, den Effekten des Modells und den Projektergebnissen zu komplex sind, als dass man ohne Analyse ausschließlich aus der Projektdurchführung lernen könnte. Darum wird die Simulation in ein Schulungskonzept eingebettet [Man01]. Dazu gehört eine Einführungsveranstaltung, in der SESAM und seine Bedienung vorgestellt werden. Anschließend führen die Teilnehmer ein simuliertes Projekt durch, das der Dozent auswertet. In der Analyseveranstaltung werden Probleme im Vorgehen anhand dieser Auswertung mit den Teilnehmern durchgesprochen und diskutiert. Danach wiederholen die Teilnehmer das Projekt mit den gleichen Vorgaben und können ihre Fehler aus dem ersten Spiel vermeiden oder alternative Lösungsstrategien ausprobieren. Sie sehen als Abschluss ihre Projektergebnisse und können daran Verbesserungen erkennen.

Mit diesem Konzept werden seit dem Wintersemester 2000/2001 Schulungen durchgeführt. Sie erfolgen für alle Studierenden des Studiengangs Softwaretechnik im dritten Semester nach der Vorlesung »Einführung in die Softwaretechnik 1« und vor den praktischen Lehrveranstaltungen [Lud06].

3 Analyse simulierter Projekte

Für den Lernerfolg der Teilnehmer ist die Analyse der simulierten Projekte ausschlaggebend, weil sie dem Dozenten erlaubt, konkrete Hilfestellung für Verbesserungen im Projekt zu geben. Für die Analyse werden vom Simulator gespeicherte Protokolldateien mit zwei Werkzeugen ausgewertet. Das Werkzeug Sesamalyzer analysiert den Projektzustand über die simulierte Zeit, das Werkzeug Sesamscore ermöglicht die Auswertung der Interaktion zwischen Studierendem und Simulator.

Zu Beginn der Analyse prüft der Dozent, welche Zielvorgaben des Projekts jeder Studierende erreicht hat. Wenn die Zielvorgaben nicht eingehalten wurden, wird der Projektverlauf anhand eines Ganttdiagramms betrachtet. Dies liefert einen ersten Anhaltspunkt, welche Probleme bei der Projektdurchführung aufgetreten sind. Ausgehend von diesen Erkenntnissen erarbeitet der Dozent Hypothesen über die Handlungen und Effekte, die zu dem vorliegenden Ergebnis geführt haben könnten. Durch weitere Analysen wird versucht, die Hypothesen zu widerlegen oder zu bestätigen. So kommt der Dozent zu einem Ergebnis, welche Handlungsweisen eines Studierenden sinnvoll oder problematisch waren. Die Ergebnisse aller Studierenden werden für die Analyseveranstaltung verdichtet, aufbereitet und in der Veranstaltung diskutiert. Die Studierenden sollen sich dabei aktiv an der Ursachenforschung beteiligen. Durch den Vergleich mit dem Vorgehen anderer Projektleiter lernen die Studierenden Alternativen zu den von ihnen getroffenen Maßnahmen und Entscheidungen kennen.

Wie eine Auswertung zu erarbeiten ist, wird in [Opf06] beschrieben. Im Folgenden werden drei Auswertungen realer Schulungen exemplarisch vorgestellt.

3.1 Erstes Auswertungsbeispiel

Im ersten Beispiel überschritt der Studierende die Zielvorgaben bezüglich Dauer, Korrektheit und Vollständigkeit. Abbildung 1 zeigt das Ganttdiagramm des Projekts.

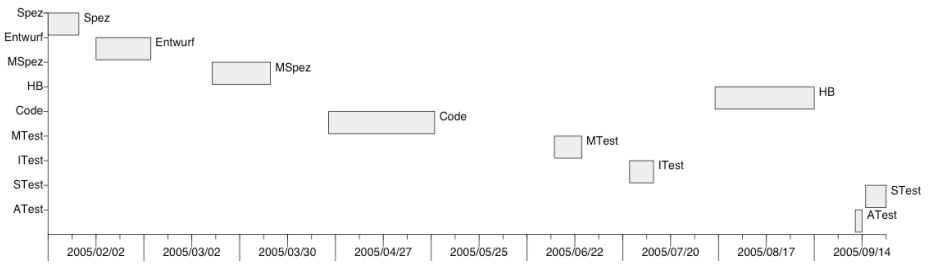


Abb. 1 *Ganttdiagramm eines Projektverlaufs in SESAM*

Neben den Tätigkeiten zur Erstellung der Spezifikation (*Spez* in *Abb.1*), des Entwurfs (*Entwurf*), der Modulspezifikation (*MSpez*), des Codes (*Code*) und des Handbuchs (*HB*) wurden mehrere Tests, nämlich Modultest (*MTest*), Integrationstest (*ITest*), Systemtest (*STest*) und Abnahmetest (*ATest*), durchgeführt. Keines der erstellten Artefakte wurde jedoch durch Reviews überprüft und korrigiert; diese wären sonst im Ganttdiagramm als Aktivitäten zu sehen. Dadurch erklärt sich die Überschreitung der Zielvorgaben.

Dieser Fall deutet auf ein grundlegendes Missverständnis über die Bedeutung von Reviews hin. Offensichtlich ist es dem Studierenden nicht gelungen, sein Wissen über Reviews aus der Vorlesung in Handlungen im simulierten Projekt umzusetzen. Der Dozent muss in der Analyseveranstaltung nochmals die Bedeutung von Reviews herausstellen. Das nächste Beispiel zeigt dies anschaulich.

3.2 Zweites Auswertungsbeispiel

Im zweiten Beispiel werden die Spielverläufe zweier Studierender gegenübergestellt, um die Auswirkungen unterschiedlicher Handlungsweisen zu zeigen. In Abbildung 2 sind die frühen Phasen in zwei Projekten über die Zeit dargestellt.

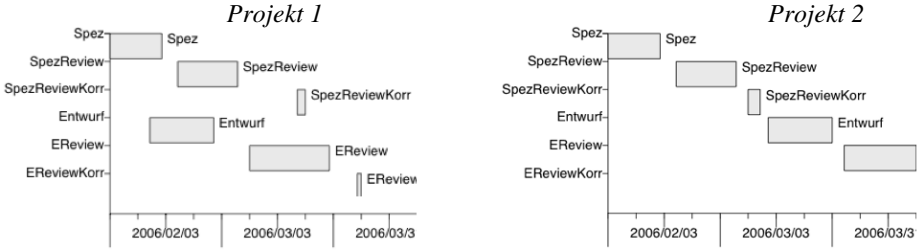


Abb. 2 *Frühe Phasen in den simulierten Projekten zweier Studierender*

In Projekt 1 wurde mit dem Entwurf direkt nach der Fertigstellung der Spezifikation begonnen. In Projekt 2 hingegen wurde die Spezifikation zunächst einem Review (*SpezReview*) unterzogen und dann korrigiert (*SpezReviewKorr*), bevor mit dem Entwurf begonnen wurde.

Die Auswirkungen dieser beiden unterschiedlichen Vorgehensweisen sind in Abbildung 3 grafisch dargestellt. Da der Entwurf auf der Spezifikation basiert, werden unentdeckte Fehler von der Spezifikation in den Entwurf übertragen.

Abbildung 3 zeigt die im Entwurf enthaltenen Analysefehler für Projekt 1 und 2. Die Plateaus der beiden Kurven (*1* und *2* in der Abbildung) entsprechen der maximalen Anzahl an Fehlern, die in den Entwürfen enthalten waren. Der Rückgang der Fehlerzahl im Laufe des Projekts (*3* und *4* in der Abbildung) wird durch Entwurfsreviews erreicht.

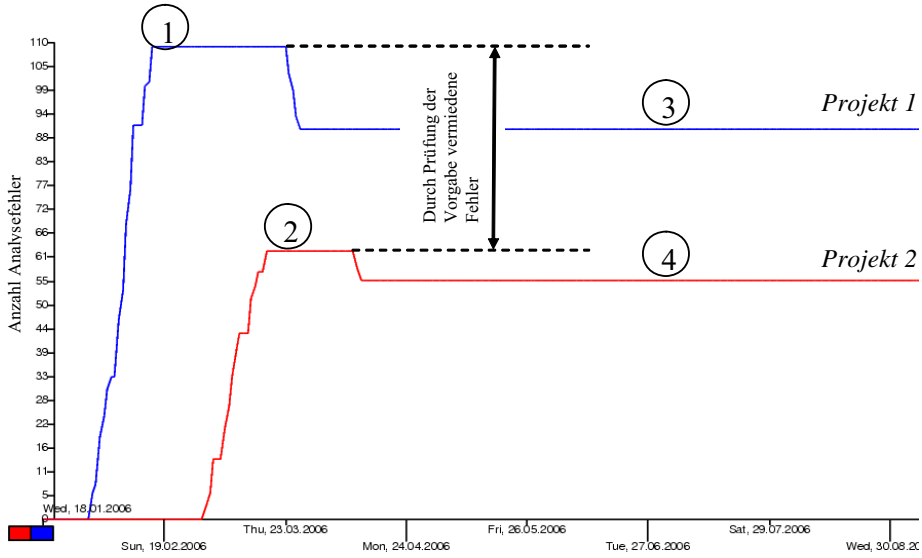


Abb. 3 *Analysefehler im Systementwurf*

Der Dozent sollte hier herausstellen, dass durch die Verwendung geprüfter Ergebnisse das Übertragen von Fehlern von einem Dokument zum nächsten deutlich reduziert werden kann. Zwar können Fehler auch durch spätere Reviews entdeckt und beseitigt werden – Abbildung 3 zeigt, dass dies in Projekt 1 sogar besser gelang als in Projekt 2, da der Rückgang der Fehleranzahl in Projekt 1 deutlich größer ist –, aber es gilt: Wenn Fehler erst gar nicht gemacht werden, entstehen auch keine Kosten für ihre Behebung.

3.3 Drittes Auswertungsbeispiel

Im dritten Beispiel hat der Studierende die Budgetvorgaben überschritten. In der Simulation sind wie in der Realität die Mitarbeiter der größte Kostenfaktor. In Abbildung 4 sind die im Projekt eingestellten Mitarbeiter über die Zeit dargestellt.

Die Einstellungsdauer ist durch den unteren durchgehenden Balken neben dem Namen des jeweiligen Mitarbeiters dargestellt. Der Balken beginnt beim Einstellungs- und endet beim Entlassungsdatum. Darüber sind durch mehrere Balken die Zeiten dargestellt, in denen dieser Mitarbeiter mit einer Aufgabe beschäftigt war. Die dunklen Balken zeigen die Zeiten, in denen der Mitarbeiter ohne Aufgabe war. Bei der Analyse muss beachtet werden, dass eine durchgängige Beschäftigung eines Mitarbeiters in SESAM nicht erreicht werden kann, weil es mit einer Simulationsschrittweite von einem Tag zu Leerlauf bei der Aufgabenverteilung an die Mitarbeiter kommen kann.

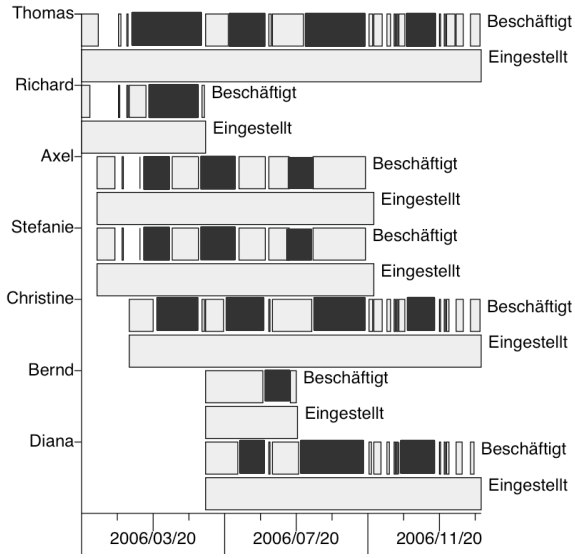


Abb. 4 *Beschäftigte Mitarbeiter*

Abbildung 4 zeigt jedoch Zeiträume von bis zu zwei Monaten, in denen ein Mitarbeiter zwar eingestellt war, er aber keine vom Projektleiter zugewiesene Aufgabe hatte. Dies lässt auf eine ungenügende Personalplanung schließen. Der Dozent muss betonen, dass eine detaillierte Personalplanung bereits vor Projektbeginn durchgeführt werden muss. Dabei muss der Projektleiter planen, welche Mitarbeiter in welchen Zeiträumen im Projekt für welche Aufgaben benötigt werden. Diese Planung muss natürlich im Projektverlauf überprüft und angepasst werden.

4 Lernerfolge in der Schulung mit SESAM

Der Lernerfolg zeigt sich deutlich in empirischen Untersuchungen [Opf02] und in den Ergebnissen, die die Schulungsteilnehmer in den simulierten Projekten erzielen.

4.1 Verbesserungen der Studierenden nach der Analyseveranstaltung

Die Verbesserungen der Projektergebnisse zeigen sich durch den Vergleich der erreichten Zielvorgaben zwischen erstem und zweitem Spiel. Abbildung 5 stellt für die Schulung im Wintersemester 2004/2005 dar, mit welcher Häufigkeit welche Anzahl der Zielvorgaben von den 60 Studierenden erreicht wurde. Im Spiel vor der Analyseveranstaltung haben 5 Studierende alle sechs Zielvorgaben erreicht. Das zweite Spiel haben 24 Studierende innerhalb aller Zielvorgaben beendet.

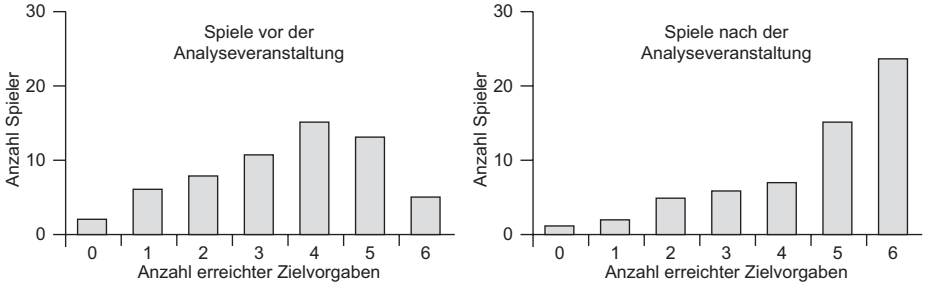


Abb. 5 *Anzahl erreichter Zielvorgaben aller Studierenden im Wintersemester 04/05*

In den Projekten vor der Analyseveranstaltung hat mehr als die Hälfte der Studierenden weder den Termin noch die Kosten eingehalten. Die Überschreitungen sind sehr deutlich: Ein Teilnehmer hat das vorhandene Budget um mehr als das Doppelte überzogen und den Termin um 5 Monate verfehlt. Nur ein Drittel konnte den Code mit für den Kunden zufriedenstellender Korrektheit ausliefern. Nach der Analyserunde konnten über zwei Drittel der Studierenden Budget und Termin einhalten. Auffällig ist, dass die Überschreitungen deutlich abnehmen: Der Termin wurde maximal um 40 Tage überschritten, das Budget maximal um rund 80.000 €. Immerhin 36 der 60 Teilnehmer konnten den Code mit zufriedenstellender Korrektheit abliefern.

Diese Verbesserungen sind nach unserer Beobachtung nicht durch die Gewöhnung an den Simulator und seine Bedienung geprägt: Die Studierenden haben für das erste Spiel ausreichend Zeit und stellen häufig Fragen an den Dozenten. Aus diesen Kontakten wissen wir, dass sich die Studierenden für das erste Spiel zuerst eine Sicherheit im Umgang mit dem Simulator antrainieren und dann das Projekt durchführen. Wir führen darum die Verbesserung im zweiten Spiel auf die Analyseveranstaltung zurück.

Die Schulungsteilnehmer nehmen die Projektsimulation ernst und identifizieren sich mit der Rolle des Projektleiters. Die Diskussionen in den Analyseveranstaltungen zeigen, dass sie die Simulation als Realität erleben. Teilnehmer von Schulungen in der Industrie bestätigen, dass die Simulation ihre Realität widerspiegelt. Sie übertragen ihre Wertungen auf das simulierte Projekt: Während Studierende eher bereit sind, für bessere Qualität den Termin oder die Kosten zu überschreiten, wird in Schulungen in der Industrie verstärkt auf Termin- und Kosteneinhaltung geachtet, auch zu Lasten der Qualität.

4.2 Auswirkungen der Schulung auf praktische Projektarbeit

Wir können kaum bewerten, ob sich die Schulung auf die praktische Projektarbeit auswirkt: Eine empirische Untersuchung von Projektleitern in der Industrie, unter denen auch ehemalige Teilnehmer der SESAM-Schulung sind, kann keine belastbaren Ergebnisse erbringen, denn alle neu erworbenen Fähigkeiten und die

Projektkultur ihrer Firma würden die Einflüsse überlagern. Ein kontrolliertes Experiment, das den Unterschied zwischen Studierenden mit und ohne Schulung zeigt, ist nicht möglich. Der Kontrollgruppe müsste die Schulung verweigert werden, dieses Vorgehen ist nicht vertretbar.

5 Fazit und Ausblick

Die SESAM-Schulung ist integraler Bestandteil der Softwaretechnik-Ausbildung in Stuttgart. Dabei haben wir den Eindruck gewonnen, dass die Studierenden besser auf Praktika vorbereitet werden als einzig durch Software-Engineering-Vorlesungen. Aufgrund des Interesses an SESAM im akademischen und industriellen Umfeld haben wir bereits Erfahrungen mit Schulungen außerhalb der Universität Stuttgart gesammelt, die unseren Eindruck bestätigen.

Der anspruchsvollste Teil der Schulung ist die Auswertung. Um diese zu erleichtern, erstellen wir eine Anleitung, die das Vorgehen bei einer Auswertung beschreibt und anhand verschiedener Beispiele illustriert. Die Auswertungswerkzeuge werden verbessert, um die Routineaufgaben automatisiert auszuführen.

Literatur

[Aho03] J. J. Ahonen, T. Junttila: A Case Study on Quality-Affecting Problems in Software Engineering Projects. Proc. of the IEEE Intl. Conf. on Software-Science, Technology & Engineering. 2003.

[Boe00] B. W. Boehm et al.: Software Cost Estimation with Cocomo II. Prentice Hall PTR. 2000.

[Dra98] A. Drappa: Quantitative Modellierung von Softwareprojekten. Dissertation. Universität Stuttgart. Shaker Verlag. 1998.

[Jon96] C. Jones: Applied Software Measurement. 2. Auflage. McGraw-Hill. 1996.

[Lic98] H. Lichter, P. Mandl-Striegnitz: A Case Study on Software Project Management in Industry – Experiences and Conclusions. Proc. of the European Software Measurement Conference. Antwerpen. 1998.

[Lud94] J. Ludewig (Hrsg.): SESAM – Software Engineering Simulation durch Animierte Modelle. Bericht der Fakultät Informatik 5/94. Universität Stuttgart. 1994.

[Lud99] J. Ludewig: Softwaretechnik in Stuttgart – ein konstruktiver Informatik-Studiengang. Informatik-Spektrum 22(1). 1999.

[Lud06] J. Ludewig (Hrsg.): Praktische Lehrveranstaltungen im Studiengang Softwaretechnik. Bericht der Fakultät Informatik. Universität Stuttgart. 4. Auflage. 2006.

[Man01] P. Mandl-Striegnitz: How to Successfully Use Software Project Simulation for Educating Software Project Managers. Proc. of the 31st Frontiers in Education Conference. 2001.

[Opf02] S. Opferkuch: Eine Untersuchung zum Einsatz komplexer Simulationsmodelle in der Projektmanagement-Ausbildung. Diplomarbeit. Universität Stuttgart. 2002.

[Opf06] S. Opferkuch (Hrsg.): Auswertungen von SESAM-Spielen. Informatikbericht der Fakultät 5. Universität Stuttgart. 2006.

Einfluss von Qualitätsdruck und Kontinuität der Zusammenarbeit auf virtuelle Teamarbeit

Marc Kasparek · Nicola Marsden

Studiengang Software Engineering, Hochschule Heilbronn
marc@kasparek.info · marsden@hs-heilbronn.de

Zusammenfassung

Im Rahmen der Qualifizierung zur virtuellen Teamarbeit im Software Engineering wurden die Auswirkungen von Qualitätsdruck und Kontinuität der Zusammenarbeit untersucht.

1 Qualifizierung zur verteilten Zusammenarbeit

Software Engineering findet zunehmend in weltweit verteilten Teams statt. Für die Ausbildung im Bereich Software Engineering bedeutet dies, dass die Studierenden auf diese Form der Zusammenarbeit vorbereitet werden und die Erfolgsfaktoren kennenlernen müssen.

Um dieser Entwicklung Rechnung zu tragen, wird im Studiengang Software Engineering an der Hochschule Heilbronn die Lehrveranstaltung »Computer-Mediated Communication« angeboten. Im Rahmen dieser Veranstaltung werden die Studierenden für die Zusammenarbeit in virtuellen internationalen Teams qualifiziert. In dieser zweisemesterwochenstündigen Vorlesung arbeiten Studierende aus unterschiedlichen Hochschulen (Indian Institute of Technology in Bombay/Indien, Transylvanische Universität in Brasov/Rumänien, Dundalk Institute of Technology/Irland, Hochschule Heilbronn/Deutschland) in Teams an kleineren Softwareentwicklungsprojekten zusammen. Die Veranstaltung findet online auf einer Groupwareplattform statt, die die benötigten Funktionen für die Teamarbeit in einer virtuellen Umgebung anbietet. Die Studierenden kommunizieren rein textbasiert – synchron während der Online-Meetings und asynchron in der Projektphase außerhalb der Lehrveranstaltungen. Face-to-face-Treffen finden nicht statt. Die Kommunikation innerhalb der internationalen studentischen

Teams findet ausschließlich auf der Plattform statt. Die gesamte Kommunikation wird geloggt und bildet die Basis für Notengebung und weitere Auswertungen.

2 Untersuchung von Einflussfaktoren auf den Erfolg

Um Erfolgsfaktoren der virtuellen Zusammenarbeit im Software Engineering zu identifizieren, wurde im Rahmen der Lehrveranstaltung im Wintersemester 2004 eine Reihe von Hypothesen quasi-experimentell überprüft. In acht Gruppen nahmen 62 Studierende teil. Die Teams bekamen drei Gruppenaufgaben für je eine Online-Sitzung und eine Projektaufgabe für einen längeren Zeitraum, die entsprechend dem Untersuchungsdesign in unterschiedlichen Konstellationen bearbeitet wurden, um die Hypothesen zu überprüfen [Ka06].

Theoretische Grundlage der Untersuchungen für die Hypothesen zur Auswirkung von Qualitätsdruck auf virtuelle Zusammenarbeit war das Flow-Konzept von Csikszentmihalyi [Csik05], in dem die Wichtigkeit von anspruchsvollen Aufgaben expliziert wird. Die Hypothesen zu den Auswirkungen der Kontinuität der Zusammenarbeit basieren auf Joseph B. Walthers Theorie zur hyperpersonalen Kommunikation [Wal00], in der aufgezeigt wird, unter welchen Bedingungen virtuelle Kommunikation zu noch größerer Nähe (»Hyperpersonalität«) der Teammitglieder untereinander führen kann. Die Teams wurden zufällig verteilt und den verschiedenen Versuchsbedingungen zugeordnet. Hier wurden Qualitätsdruck und Kontinuität der Zusammenarbeit systematisch variiert.

3 Qualitätsdruck und Kontinuität: Positiver Einfluss

Die Untersuchung der studentischen Software-Engineering-Teams zeigte bezogen auf die Auswirkungen von Qualitätsdruck in der virtuellen Zusammenarbeit, dass ein höherer Qualitätsdruck zu einer größeren Zufriedenheit mit dem Gruppenergebnis führt. Diese Zufriedenheit scheint auch gerechtfertigt: Der ausgeübte Qualitätsdruck führte tatsächlich zu einer höheren Qualität der Arbeitsergebnisse der international verteilten Teamarbeit. Hypothesenkonform führt der Qualitätsdruck ebenfalls dazu, dass der persönliche Anteil der Einzelnen am Gruppenergebnis tendenziell überschätzt wird.

Wussten die Studierenden, dass sie im bestehenden Team nicht nur einmalig, sondern kontinuierlich zusammenarbeiten würden, so führte dies zu weniger Selbstdarstellung und einer realistischen Einschätzung des eigenen Anteils am Gesamtgruppenergebnis. Insgesamt führte die Kontinuität der Zusammenarbeit ebenso wie der Qualitätsdruck zu einer höheren Qualität der international verteilten Softwareprojekte.

Literatur

[Csik05] Csikszentmihalyi, Mihaly: Flow – Das Geheimnis des Glücks. Stuttgart, Klett-Cotta, 2005.

[Ka06] Kasparek, Marc: Einfluss von Qualitätsdruck und von Kontinuität der Zusammenarbeit auf virtuelle Teamarbeit. Unveröffentlichte Diplomarbeit im Studiengang Software Engineering, Hochschule Heilbronn, 2006.

[Wal00] Walther, Joseph B.: Die Beziehungsdynamik in virtuellen Teams. In: M. Boos, K. J. Jonas & K. Sassenberg (Hrsg.): Computervermittelte Kommunikation in Organisationen. Göttingen, Hogrefe, 2000, S. 11 – 25.

Einführung in die Softwareentwicklung – Softwaretechnik trotz Objektorientierung?

Axel Schmolitzky · Heinz Züllighoven

Arbeitsbereich Softwaretechnik, Department Informatik, Universität Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg
{schmolitzky|zuellighoven}@informatik.uni-hamburg.de

Zusammenfassung

Kaum ein Thema ist so umstritten wie der richtige Weg bei der grundständigen Programmierausbildung. Dieser Beitrag beschreibt einen neuen Ansatz zur Einführung in die Softwareentwicklung, der in den letzten Jahren im Arbeitsbereich Softwaretechnik an der Universität Hamburg entwickelt wurde. Er orientiert sich an einem Objects First-Ansatz, macht aber gleichzeitig einen Einstieg in klassische Themen wie Algorithmen und Datenstrukturen früher als allgemein üblich und geht einen ungewöhnlichen Weg bei der Vermittlung von Interfaces und Vererbungskonzepten.

1 Einführung

Programmierausbildung im Informatikstudium: Zuerst funktional oder zuerst imperativ? *Objects First* oder *Algorithms First*? *Modeling First* oder *Interaction First*? Was ist mit *Test First*? Kaum ein Thema ist so umstritten wie der richtige Weg bei der grundständigen Programmierausbildung. Einigkeit besteht wohl nur bei der Aussage, dass Programmieren als das Handwerkszeug der Softwareentwicklung nach wie vor eine zentrale Rolle in der Ausbildung spielen sollte.

Zumindest aus Sicht der Softwaretechnik lassen sich einige der eingangs gestellten Fragen leichter beantworten. Da sich in der praktischen Softwareentwicklung das objektorientierte Paradigma in den letzten Jahren durchgesetzt hat, steht außer Zweifel, dass gute Kenntnisse objektorientierter Programmierung notwendig sind. Die funktionale Programmierung hingegen verliert zunehmend an Stellenwert, was einige bedauern; sie wird aus softwaretechnischer Sicht eher als eine hilfreiche Ergänzung denn als elementar angesehen.

Auch bei eher konservativen Studiengängen, die in den letzten beiden Jahrzehnten beim Einstieg in die Programmierung nicht vom imperativen auf das funktionale Paradigma umgestellt haben, wird jetzt häufiger gefragt: Klassisch imperativ, beginnend mit Kontrollstrukturen und Prozeduren (in der Tradition imperativer Sprachen wie Pascal, Modula-2 und C) oder konsequent objektorientiert mit Objekten vom ersten Tag an, unter Nutzung der Möglichkeiten von Sprachen wie Java und C#?

Und schließlich: Bedeutet »konsequent objektorientiert« nicht insbesondere eine frühe Vermittlung des für die Objektorientierung laut Wegner [Weg87] so zentralen Konzeptes der Vererbung? Dann stellt sich aus Sicht der Softwaretechnik schon die Frage, ob dieses leicht zu missbrauchende Instrument (Systeme mit großen Vererbungshierarchien gelten als sehr schwierig zu warten) nicht zu früh in die Hände noch unerfahrener Programmierer gegeben wird; nicht zu unrecht wurde Vererbung auch schon als das GoTo der 90er Jahre bezeichnet.

Unser Eindruck ist, dass statt polarisierender Alternativen eine geeignete Kombination gewählt werden kann. Dieser Beitrag beschreibt einen Ansatz, den wir in den letzten Jahren im Arbeitsbereich Softwaretechnik an der Universität Hamburg entwickelt haben. Darin sind viele Erfahrungen eingeflossen, die wir in etlichen Jahren der praktischen Programmierausbildung an der Hochschule und im industriellen Umfeld gesammelt haben.

Der Text gliedert sich wie folgt: In Abschnitt 2 wird kurz der Hintergrund dargestellt, vor dem der neue Ansatz entstanden ist. Abschnitt 3 diskutiert verschiedene Entwurfsprinzipien, die bei der Gestaltung einer Programmiereinführung berücksichtigt werden können. Die Abschnitte 4, 5, 6 und 7 bilden den Kern des Beitrags, in dem der inhaltliche und didaktische Aufbau zweier konsekutiver Veranstaltungen zur Einführung in die Softwareentwicklung vorgestellt wird, samt einer ersten Bewertung. Abschnitt 8 fasst den Beitrag zusammen.

2 Hintergrund

Seit dem Wintersemester 2005/06 löst der *Bachelor of Science in Informatik* das Informatik-Diplom an der Universität Hamburg ab. Im Zuge der Umstellung wurde die Chance genutzt, die einführenden Veranstaltungen substanziell neu zu strukturieren. Die in [BBSZ03] beschriebenen P-Veranstaltungen (einführende Veranstaltungen zur Programmierung) mit funktionaler Programmierung als Einstieg wurden abgelöst durch eine Modulfolge zur Softwareentwicklung (SE), die mit der imperativen Programmierung beginnt und erst im dritten Semester (SE3) wahlweise die funktionale oder die logische Programmierung anbietet. Sie wird ergänzt durch das Modul »Algorithmen und Datenstrukturen« im dritten Semester. Der Arbeitsbereich Softwaretechnik ist derzeit verantwortlich für die ersten beiden Module, Softwareentwicklung I (SE1) und Softwareentwicklung II (SE2). SE1 besteht aus Vorlesung und Übungen (jeweils zweistündig) und soll die impe-

rative und objektorientierte Programmierung vermitteln. SE2 bietet zwei parallele zweistündige Vorlesungen mit einer gemeinsamen Übung (4+2); eine Vorlesung behandelt objektorientierte Programmierung und Modellierung (OOPM), die andere thematisiert vor allem Softwareentwicklungsprozesse unter dem Titel »Softwaretechnik und -Ergonomie«.

Im Übungsbetrieb beider Module haben wir bewährte Konzepte aus der alten P-Veranstaltung P2 übernommen, die wir in verschiedenen Kontexten bereits beschrieben haben [Sch06a]: ein betreuter Laborbetrieb, Arbeiten in Paaren, möglichst viel direktes Feedback. Auf diese eher prozessbezogenen Aspekte wollen wir hier nicht näher eingehen. In diesem Beitrag konzentrieren wir uns auf die inhaltliche Gestaltung von SE1 und dem darauf unmittelbar aufbauenden OOPM-Anteil von SE2.

3 Gestaltungsmuster

Bevor wir unseren neuen Ansatz ausführlich beschreiben, stellen wir in diesem Abschnitt einige grundsätzliche Möglichkeiten dar, den Einstieg in die Programmierausbildung zu gestalten. Teilweise haben wir diese Ansätze in Lehrbüchern vorgefunden, teilweise in den Beschreibungen von Informatik-Studiengängen. Einige sind bereits in der wissenschaftlichen Gemeinde diskutiert worden [Bru04]. Die hier vorgestellte Auswahl erhebt dabei keinen Anspruch auf Vollständigkeit.

Einige der beschriebenen Muster tragen ein »First« in ihrem Namen. Sie folgen damit dem didaktischen Muster des »frühen Vogels« (siehe das *Early Bird Pattern* im Pedagogical Pattern Projekt [PPP]), das besagt, dass zentrale Konzepte früh vermittelt werden sollten, damit sie länger wirken und auch häufiger wiederholt werden können als spät vermittelte. Offensichtlich führt ein Befolgen aller First-Ansätze, falls vereinbar, zu einer unrealistischen Stoffdichte in den ersten Wochen.

3.1 Algorithms First

Die meisten Einführungen beruhen auf der Annahme, dass Algorithmen im Kern der Informatik stehen: ohne ein solides Verständnis von grundlegender Algorithmik keine soliden Programmierkenntnisse. Für einen »sanften« Einstieg kann dafür das klassische Paar »Algorithmen und Datenstrukturen« zuerst ohne Datenstrukturen betrachtet werden: Prozeduren oder Funktionen mit festen Ein- und Ausgabegrößen berechnen Lösungen für kleinere und mittlere Probleme. Wir bezeichnen diesen Ansatz hier verkürzt als *Algorithms First*-Ansatz.

In der imperativen Welt wird so das Prozedur-Konzept früh eingeführt. Beim Einstieg mit einer funktionalen Sprache wird die reine Algorithmik noch stärker betont. So lässt sich beispielsweise ein Quicksort in ML deutlich einfacher (in einer Zeile) formulieren als in jeder imperativen Umsetzung.

3.2 Objects First

Der Objects First-Ansatz basiert auf einer simplen Grundannahme: Moderne Softwaresysteme bestehen aus einer großen Anzahl von Objekten, die miteinander auf vielfältige Weise interagieren; wenn Objekte die elementaren Bausteine solcher Systeme sind, dann sollten diese Bausteine von Beginn an behandelt werden. So eingängig die Grundidee ist, so schwierig ist ihre Umsetzung. In einer Diskussion auf einer SIGCSE-Mailingliste wurde beklagt, dass der Ansatz ohne Werkzeugunterstützung kaum gangbar ist [Bru04]. Entsprechend existieren zahlreiche Unterstützungen, von Mikrowelten über didaktisch motivierte Werkzeuge hin zu Bibliotheken einfacher grafischer Objekte.

Eines der Werkzeuge mit didaktischem Anspruch ist BlueJ [KQPR03], eine Entwicklungsumgebung für Java, die explizit für die Vermittlung objektorientierter Programmierung entwickelt wurde. Hier können auf einer Objektleiste einzelne Objekte interaktiv erzeugt und untersucht werden; die Klassenstruktur des aktuellen Projektes wird in einem interaktiven UML-Diagramm visualisiert und ermöglicht das direkte Erzeugen von Exemplaren dieser Klassen.

Da ein Werkzeug ohne begleitende Kursstruktur zu leicht »missbraucht« werden kann, habe die BlueJ-Designer ein Lehrbuch [BaK06] geschrieben, das BlueJ mit einem konsequenten Objects First-Ansatz nutzt. Die dritte Auflage dieses Buches basiert auf Java 1.5 und ist derzeit eines der erfolgreichsten Lehrbücher zu diesem Ansatz (es wurde u.a. bereits in fünf Sprachen übersetzt).

3.3 Modeling/Design First

Der Modeling First-Ansatz folgt dem Grundsatz, dass Programmieren primär dem Problemlösen dient. Um ein Problem zu lösen, muss zuerst der Problembereich geeignet modelliert werden. Hier spielt eine wichtige Rolle, dass die objektorientierte Programmierung auch deshalb so erfolgreich wurde, weil sie ohne Strukturbruch einen Ausschnitt der Welt objektorientiert als ein System interagierender Exemplare von Klassen repräsentieren kann.

Eine Einführung in diesen Ansatz zielt also darauf ab, Gegenstände der natürlichen Welt objektorientiert zu modellieren und idealerweise auch zu simulieren. Klassisch ist das Restaurant-Beispiel aus dem COOL-Projekt (u.a. von Kristen Nygaard) [COOL], in dem ein Restaurant mitsamt Einrichtung, Gästen und Kellnern als ein System interagierender Objekte modelliert werden soll. Bewusst wird zum Einstieg kein simples Beispiel gewählt, sondern ein ausreichend komplexes Beispiel, mit dem alle Facetten der objektorientierten Programmierung ausgelotet werden können. Erste Erfahrungen mit diesem Ansatz sind in [BeC04] beschrieben.

3.4 Interactive Programming

Interactive Programming ist ein Ansatz für das erste Semester der Programmierausbildung, der von Lynn Andrea Stein seit über zehn Jahren vertreten und seit mehreren Jahren praktiziert wird, anfangs am MIT, inzwischen am Franklin W. Olin College of Engineering [RCS101]. Ihr Ansatz geht von interagierenden Objekten aus, jedoch nicht nach einem sequenziellen Programmiermodell, sondern als nebenläufige Einheiten. Stein betont den Bezug zur realen Welt mit ihren parallel agierenden Einheiten und erwartet so einen intuitiven Zugang für Programmieranfänger. Sie wendet sich bewusst gegen das Paradigma *Computation as Calculation* und betont die Interaktion zwischen Programmeinheiten innerhalb fortlaufender Prozesse. Einen positiven Erfahrungsbericht liefert Weber-Wulff in [Web00].

3.5 Programmiersprache zuerst, Konzepte danach

Ein häufig anzutreffender Ansatz in vielen Studiengängen ist, eine Programmiersprache als Handwerkszeug zu sehen und sie vollständig im ersten Semester zu vermitteln; im zweiten Semester folgt dann häufig eine Programmierveranstaltung mit Algorithmen und Datenstrukturen als explizitem Thema. Dabei werden die Sprachkonstrukte des ersten Semesters auf klassische algorithmische Probleme der Informatik angewendet.

Problematisch an diesem Ansatz ist, dass eine hohe »Konzeptdichte« im ersten Semester entsteht: Java ist zwar einfacher als vergleichbare Sprachen (C++, C#), umfasst aber seit der Java-Version 1.5 anspruchsvolle Konzepte wie Generizität, Vererbung und Nebenläufigkeit; aus unserer Sicht ein zu hartes Themenpaket für Programmieranfänger.

3.6 Der historischen Entwicklung folgen

Eine weitere häufige Vorgehensweise könnte so formuliert werden: Programmiersprachen haben sich über mehrere Jahrzehnte entwickelt; vermittle die Sprachkonzepte entlang ihrer historischen Entwicklung.

Zwei Beispiele sollen dies verdeutlichen:

- C++ wurde als Nachfolger der Programmiersprache C entworfen, indem vor allem objektorientierte Konzepte hinzugefügt wurden. Das Klassenkonzept ist auf diese Weise in eine imperative Sprache hineingewachsen, so dass neben den Verbünden mit dem Schlüsselwort `struct` auch Klassen mit dem Schlüsselwort `class` definiert werden konnten. Die Ähnlichkeiten sind groß, und Klassen können auf diese Weise als eine Erweiterung des Verbund-Konzeptes angesehen werden. Für den Weg von C zu C++ ist diese Entwicklung relevant, für die Vermittlung der Idee von Klassen und Objekten scheint sie uns eher

hinderlich. Ein Beispiel für eine aktuelle Anwendung dieses Prinzips für Java ist in [Mös05] zu finden.

- Das Sprachkonstrukt *Interface*, das durch Java in den Mainstream der Programmiersprachen eingeführt wurde, ist historisch gesehen eine Weiterentwicklung der *abstrakten Klasse*. Eine Ausprägung abstrakter Klassen sind *vollständig abstrakte Klassen*, die als reine Spezifikation der zu implementierenden Funktionalität dienen; Interfaces sind letztendlich genau dies. Interfaces können aber auch als eigenständiges Konzept angesehen werden, das unabhängig von Fragen der Implementationsvererbung ist. In Lehrbüchern zu Java werden aber üblicherweise Interfaces nach Vererbung und abstrakten Klassen eingeführt [Sch06a].

Diese Herangehensweise ist meist motiviert durch profundes (technisches) Hintergrundwissen der Dozenten über Programmiersprachen, das den meisten Programmieranfängern jedoch üblicherweise fehlt.

3.7 Erst Konsumieren, dann Produzieren

Eines unserer eigenen Entwurfsprinzipien beim Kursaufbau lautet: *Konsumieren* ist einfacher als *Produzieren*, vermittle deshalb das Konsumieren eines Konzeptes, ehe es produktiv eingesetzt wird. Nehmen wir z.B. das Konzept »geschriebener Text«: Das Lesen eines Textes (das Konsumieren) ist zwar eine anspruchsvolle Tätigkeit, die eine solide Ausbildung erfordert; das Schreiben eines Textes (das Produzieren) ist jedoch noch anspruchsvoller und wird üblicherweise später vermittelt. Um einen Text zu lesen müssen darüber hinaus nicht alle Details eines geschriebenen Textes (Grammatik, Stil, Argumentationslinien etc.) bekannt sein; man kann Texte auch mit partiellen Kenntnissen dieser Details lesen.

Angewendet auf Programmiersprachkonzepte lässt sich das Prinzip an zwei Beispielen veranschaulichen:

- Ein *konsumierendes* Verständnis von Paketen in Java (packages) bedeutet, dass bereits implementierte Dienstleistungen von Klassen in benannten Paketen liegen und von dort importiert werden können; dieser Umstand kann relativ früh vermittelt werden. Pakete zu produzieren schließt die Fähigkeit ein, größere Systeme in Subsysteme zu zerlegen; dazu sollten alle Konzepte der Paketsichtbarkeit (für Klassen und Methoden) verstanden sein, die aber erst beim eigenen Umgang mit vergleichsweise großen Systemen nachvollziehbar werden und somit später vermittelt werden sollten.
- Das *Konsumieren* von Generizität in Java setzt voraus, dass beim Umgang mit Sammlungen des Java Collection Frameworks der Typ der Elemente deklariert werden muss; dies kann sehr früh in der Ausbildung vermittelt werden. Das produzierende Definieren einer generischen Klasse erfordert etliches mehr: das Wissen über formale und aktuelle Typparameter, über Einschränkungen bei der Benutzung formaler Typparameter, über Type Erasure u.Ä.

Für Lehrende bietet dieses Prinzip den Vorteil, dass vermeintlich komplizierte Themen früh angesprochen werden können, wenn die Voraussetzungen für ein konsumierendes Verständnis geeignet ausgewählt werden. Aus didaktischer Sicht ist dies eine Anwendung des *Spiralmusters* aus dem Pedagogical Pattern Project [PPP]; dieses besagt, dass nicht alles zu einem Thema auf einmal gesagt werden muss, sondern dass nur ein Teil vermittelt und das Thema später wieder aufgegriffen und vertieft werden kann.

4 Unsere Einführung in die Softwareentwicklung

Für unsere zweisemestrige Einführung in die objektorientierte Softwaretechnik haben wir nach dem gerade genannten Prinzip einige Themen ausgewählt, deren Spannungsbogen von den Objekten (zur Laufzeit) über die Klassen (als grundlegende Konstruktionseinheiten) zu den Softwaretechnikkonzepten hinter dem Entwurf reicht. Dabei machen wir deutlich, dass Softwaretechniker beim Entwurf üblicherweise nicht die ganze Welt modellieren, sondern bewusst das Systemmodell (Was bauen wir in Software?) vom Anwendungsmodell (In welchem Kontext läuft die Software?) trennen sollten. Eine Übersicht der beiden Semester lässt sich so charakterisieren:

- Semester 1: Objects First und Interactive Programming über (nicht grafische) Schnittstellen von Objekten mit BlueJ, frühes Testen über Schnittstellen mit Interfaces, früher Einstieg in Algorithmen und Datenstrukturen am Beispiel von Sammlungsimplementationen.
- Semester 2: Abstraktionen über die Programmierung mit abstrakten Datentypen, dem Vertragsmodell und verschiedenen Formen der Polymorphie (Subtyping, Generizität); Implementationsvererbung und objektorientierte Modellierung von Werten und Objekten; Grundlagen grafischer Benutzungsschnittstellen, von Entwurfsmustern und Refactorings.

Im ersten Semester beschränken wir uns bewusst auf die handwerklichen Aspekte der Programmierung und bleiben im objektbasierten Sprachmodell einer Sprache (Java). Erst auf Basis dieser Programmiererfahrung abstrahieren wir im zweiten Semester stärker von der verwendeten Sprache und stellen weitergehende Konzepte vor (Pass-by-Reference, benutzerdefinierte Wert- und Referenzsemantik etc.).

Was haben wir bewusst ausgelassen?

Zunächst erläutern wir hier, welche der in Abschnitt 3 genannten Grundsätze wir bewusst ausgelassen oder weniger stark berücksichtigt haben.

Wie bereits in [BBSZ03] beschrieben, bringen wir in unseren Programmierveranstaltungen unseren Industriehintergrund mit ein: Beim »Training on the Job« hat sich eine enge Verbindung von Konzeptvorstellung und praktischer Um-

setzung bewährt; häufig lassen wir in Schulungen erst praktische Erfahrung sammeln, um dann die Theorie aufzuarbeiten. Dabei zeigt sich immer wieder: Erfahrungsgrundiertes Konzeptwissen benötigt Zeit. Eine zu hohe Verdichtung oder mangelnde praktische Erfahrung verhindert ein nachhaltiges Verständnis. Eine vollständige Behandlung von Java in nur einem Semester (siehe 3.5) kam für uns deshalb nicht in Frage.

Zum Thema *Modeling First* haben wir bereits in früheren P2-Veranstaltungen Erfahrungen gesammelt, die nicht ermutigend waren: Bereits sehr früh sollte dort, ausgehend von Szenarios der realen Welt, ein System für ein Taxiunternehmen modelliert werden. Da die Studierenden noch kein Gefühl dafür hatten, was von einem objektorientierten System geleistet werden kann (und was nicht), waren die Modelle entsprechend haarsträubend. Wir sind deshalb inzwischen der Meinung, dass vor den ersten Modellierungsversuchen eine handwerkliche Vermittlung der grundlegenden Bausteine (Objekte und Klassen) mit ihren technischen Eigenschaften stehen sollte, um die Zielwelt einer objektorientierten Modellierung zu verdeutlichen.

Im Entwurf der ersten beiden Semester haben wir auch bewusst die nebenläufige Programmierung ausgelassen (sie wird in unserem Bachelor-Lehrplan frühestens im dritten Semester in den Modulen zu Datenbanken und zu Systemsoftware thematisiert). Während wir, ähnlich wie Wegner in [Weg97], beim Umgang mit Computern einen Trend zu interaktiven Systemen erkennen, der die Bedeutung reiner Berechnungen durch Algorithmen in den Hintergrund treten lässt, und somit auch das Interactive Programming als einen lohnenswerten Ansatz sehen, schätzen wir die explizite Berücksichtigung nebenläufiger Prozesse als zu ambitioniert für die ersten Semester ein.

5 Semester 1: Grundlagen der objektorientierten Programmierung

Der Anspruch im ersten Semester ist, *Programmieranfänger* an das systematische Programmieren heranzuführen. Wir nehmen dabei bewusst in Kauf, dass einige Studierende bereits vor dem Studium Erfahrungen mit imperativer Programmierung gesammelt haben und bei Teilen der Veranstaltung unterfordert sind. Wir versuchen jedoch, durch das Laborkonzept diese Vorkenntnisse zu nutzen, indem die erfahreneren Studierenden ihr Wissen in einem Programmierpaar an Anfänger weitergeben sollen. Unsere Erfahrungen damit sind jedoch gemischt, denn nicht jedem Studierenden mit Vorerfahrung liegt es, sein Wissen in didaktisch geeigneter Form weiterzugeben.

Wir haben, insbesondere für das erste Semester, sogenannte *Komplexitätsstufen* für die Lehre entworfen, um kleine Objektsysteme einzuordnen. Jede Stufe definiert dabei Konzepte (samt reservierter Wörter), die auf den Konzepten niedrigerer Stufen aufbauen. SE1 strukturieren wir in vier Komplexitätsstufen. Wir

wollen auf diese Stufen im Rahmen dieses Beitrags nicht näher eingehen, erwähnen sie jedoch an einigen Stellen im nachfolgenden Text. Ein erster Ausblick wurde für einen Workshop der ECOOP 2005 gegeben [Sch05].

5.1 Objects First mit BlueJ

Beim Einstieg mit BlueJ werden Klassen vorgegeben, die anfangs so konsumiert werden, dass interaktiv Exemplare erstellt und manipuliert werden. Wir nutzen dabei eine Besonderheit von BlueJ: Beim Doppelklick auf eine Klasse lassen wir nicht, wie üblich, den Editor in der Quelltextansicht erscheinen, sondern in der von javadoc erzeugten Schnittstellenansicht der Klasse. Die Studierenden haben so früh Kontakt mit der Idee einer *Schnittstelle*, die die benutzbaren Operationen beschreibt, ohne Details der Realisierung preiszugeben.

5.2 Java Level 1: Vereinfachte Syntaxbeschreibung

Ein wichtiges Detail in der Programmierausbildung sind formale Syntaxbeschreibungen. Hier sollen die Studierenden praktisch erfahren, dass solche Beschreibungen für den Umgang mit einer Programmiersprache nützlich sind. Wir hoffen, dass die Studierenden anschließend in der theoretischen Informatik der Behandlung formaler Sprachen leichter folgen können. Problematisch ist dabei, dass aufgrund der neuen Spracheigenschaften die vollständige Syntaxbeschreibung von Java 1.5 so kompliziert geworden ist, dass sie Anfängern kaum noch vermittelbar ist. Wir haben deshalb für unsere erste Komplexitätsstufe eine abgespeckte Syntaxbeschreibung erstellt, die aufgrund der wenigen Konzepte (u.a. noch keine Schleifen) auf einer A4-Seite Platz findet.

5.3 Interfaces zur Modellierung von Schnittstellen

Während die zweite Komplexitätsstufe Referenztypen einführt und sich mit Iteration und Rekursion beschäftigt, stellt die dritte Stufe die Interfaces von Java als eine Möglichkeit vor, die Schnittstelle einer Klasse explizit zu beschreiben. Motiviert wird dies über den Anspruch, in einer Testklasse Black-Box-Tests zu implementieren; wenn die Testklasse nur ein Interface benutzt, wird die Black-Box-Eigenschaft eher garantiert als in einem direkten Test der Klasse. In einer weiteren Woche benutzen die Studierenden die Interfaces `Set` und `List` des Java Collection Frameworks, um kleine Probleme mit Hilfe von Sammlungen zu lösen. Sie lernen konsumierend, dass es für `List` zwei Implementationen geben kann (`LinkedList` und `ArrayList`), die beliebig austauschbar sind.

Wir zeigen, dass ein Interface/Typ auf verschiedene Weise implementiert werden kann, und unterscheiden dabei den statischen und dynamischen Typ einer Variablen. So stellen wir ein erstes Beispiel für Polymorphie vor, ohne auf die

volle Komplexität von Implementationsvererbung eingehen zu müssen (Konsumieren von Vererbung, aber noch kein Produzieren mit Vererbung). Diesen Ansatz haben wir bereits an anderer Stelle beschrieben [Sch06b].

5.4 Sammlungen: Mengen und Listen implementieren

Auf der vierten Stufe stellen wir nur noch wenige neue Sprachelemente vor; stattdessen vertiefen wir in den letzten fünf Wochen die auf den ersten drei Stufen vermittelten Konzepte anhand eines zentralen Themas: Implementierungen von Sammlungen. Hier werden dynamische Datenstrukturen (verkettete Listen, wachsende Arrays, Bäume, einfache Hash-Verfahren) erklärt und teilweise mit den objektbasierten Mitteln von Java implementiert. Suchen und Sortieren auf diesen Strukturen werden in ihrer Zeitkomplexität für die Studierenden unmittelbar erfahrbar. Wir erläutern dann passend die O-Notation. Abschließend werden anhand einfacher Graph-Probleme die Vorteile von rekursiven Algorithmen verdeutlicht.

6 Semester 2: Abstraktion – der Weg zu objektorientierter Modellierung

Während wir uns im ersten Semester knapp oberhalb der »Grasnarbe« der Programmierung bewegen, stellen wir im zweiten Semester deutlich höhere Ansprüche an die Abstraktionsfähigkeiten der Studierenden. Wir diskutieren zu Anfang die Korrektheit von Software, untersuchen Strategien zur Fehlerbehandlung mit Exceptions und stellen das Vertragsmodell vor. Anschließend folgen zwei getrennte Themenblöcke: Polymorphie und Implementationsvererbung.

Polymorphie: Subtyping und Generizität

Wir stellen die Taxonomie von Polymorphie nach Cardelli/Wegner [CaW85] vor und diskutieren Merkmale der objektorientierten Typisierung: Subtyping durch Typhierarchien, Probleme mit Ko- und Kontravarianz, Generizität in Java. Hier zahlt sich aus, dass wir den Typbegriff anhand der Interfaces von Java ausführlich in SE1 eingeübt haben. Der Schritt zu Hierarchien von Interfaces fällt so leicht.

Implementationsvererbung

Bewusst spät (ganz im Sinne der Umkehrung des »frühen Vogels«) behandeln wir die Implementationsvererbung als ein Konzept zur inkrementellen Modifikation von Implementationen. Hier erklären wir die bisher nur konsumierte Methodenauswertung beim dynamischen Binden, abstrakte Methoden, das Redefinieren von Methoden, die Konstruktoren-Verkettung in Java und das Konzept der Erbenschnittstelle. All diese Themen sollten Softwaretechniker beherrschen; aber

wir legen großen Wert darauf, dass die Unterscheidung von Subtyping und Implementationsvererbung deutlich wird. Letzere sollte überwiegend konsumiert und nur mit Vorsicht aktiv produziert werden.

Auf der Basis der bisher vorgestellten technischen Grundlagen können objektorientierte Rahmenwerke kompetent genutzt werden. Dies wird am Beispiel der Swing-Bibliothek von Java geübt. Die zweite Hälfte der Lehrveranstaltung behandelt Analyse, Modellierung und Entwurf sowie Refactorings in Form eines kleinen Projekts.

7 Feedback der Studierenden

Jedes Lehrkonzept muss sich an seinem Erfolg messen lassen. Bisher, nach nur einem Durchlauf, haben wir keine belastbaren Zahlen zur Effektivität des neuen Ansatzes. Bisher liegen nur die Ergebnisse der allgemeinen anonymen Studierendenbefragung nach SE1 (regelmäßig, einheitlich und für alle Pflichtveranstaltungen durchgeführt von der studentischen Fragebogen AG) und die Antworten auf unsere gezielte Umfrage nach SE2 vor. Die Ergebnisse der Fragebogen AG für SE2 lagen uns beim Verfassen dieses Beitrags noch nicht vor.

Das Feedback zu SE1 über die Fragebogen AG war sehr positiv. Das Modul hat exzellente Noten bekommen, insbesondere für seine klare Strukturierung (über 80% wählten »sehr gut« oder »gut«). Den Studierenden war bewusst, dass wir für Interfaces einen ungewöhnlichen Weg gewählt haben. Es gab keine Beschwerden über die fehlende Behandlung von Vererbung, und für keinen Studierenden war unsere Verwendung von Interfaces problematisch.

Nach SE2 haben wir per E-Mail eine Umfrage zu unserer Vermittlung von Vererbungskonzepten durchgeführt. Die Umfrage bestand aus den folgenden Aussagen mit den jeweiligen Antworten; die Prozentzahlen geben die Verteilung der gewählten Antworten wieder:

- »Vererbung ist ein wichtiges Thema, das für das volle Verständnis objektorientierter Programmierung zentral ist«:

  Ja, absolut (57%) Ja (33) eher ja (5%) eher nein (5%) gar nicht ( )

- »Mir ist der Unterschied zwischen Subtyping und Implementationsvererbung klar geworden«:

  Ja (71%) ein bisschen (29%) eher nicht ( ) gar nicht ( )

- »Die Unterscheidung hat mir geholfen, das komplizierte Thema Vererbung zu strukturieren«:

  Ja (33%) **eher ja (57%)** eher nicht (5%) nein, gar nicht (5%)

- »Ich finde die Unterscheidung relevant, sie sollte gelehrt werden«:

  Ja (62%) eher ja (38%) eher nicht ( ) nein, gar nicht ( )

- »Ich finde die Reihenfolge der Vermittlung (erst Subtyping, dann Implementationsvererbung) gut«:

  Ja, gut nachvollziehbar (86%) nein, besser andersrum (14%)

- »Ich habe auch an SE1 teilgenommen und halte es für sinnvoll, Interfaces dort zu thematisieren, bevor Vererbung vollständig erklärt wurde, weil Interfaces auch ohne ausführliche Behandlung von Vererbung verstanden werden können:«

  Ja (75%) eher ja (15%) eher nicht (10%) nein, gar nicht ( )

8 Zusammenfassung

In diesem Beitrag haben wir unsere Überlegungen und Entwurfsprinzipien zur Gestaltung des Einstiegs in die Softwareentwicklung im neuen Bachelor-Studiengang Informatik an der Universität Hamburg dargestellt. Das Ergebnis ist eine Kursfolge, in der Objekte vom ersten Tag an behandelt und Schnittstellen und Interfaces früh angesprochen werden. Vererbung wird bewusst erst vollständig im zweiten Semester behandelt, um im ersten Semester grundlegende Algorithmen auf Basis objektbasierter Sprachkonzepte zu verdeutlichen. Die Bewertung des neuen Ansatzes durch die Studierenden ist ermutigend.

Danksagungen

Wir danken allen Kolleginnen und Kollegen im Arbeitsbereich Softwaretechnik an der Universität Hamburg für die exzellente Zusammenarbeit der letzten Jahre, die diesen Ansatz erst ermöglicht hat.

Literatur

[BaK06] Barnes, D., Kölling, M.: *Objects First with Java – A Practical Introduction Using BlueJ (3rd Edition)*, Pearson Education, UK, 2006.

[BBSZ03] Becker-Pechau, P., Bleek, W.-G., Schmolitzky, A., Züllighoven, H.: »Integration Agiler Prozesse in die Softwaretechnik-Ausbildung im Informatik-Grundstudium«, in Siedersleben, J. and Weber-Wulff, D. (Eds.), *Software Engineering im Unterricht der Hochschulen (SEUH),* dpunkt.verlag, S. 8 – 21, 2003.

[BeC04] Bennedsen, J., Caspersen, M. E.: »Programming in context: a model-first approach to CS1«, *Proc. 35th SIGCSE technical symposium on Computer Science Education*, Norfolk, Virginia, USA; ACM Press, S. 477 – 481, 2004.

[Bru04] Bruce, K. B.: »Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list«, *Inroads (newsletter of SIGCSE),* 36:4, S. 29 – 34, 2004.

[CaW85] Cardelli, L., Wegner, P.: »On Understanding Types, Data Abstraction and Polymorphism«, *ACM Computing Surveys,* 17:4, S. 471 – 522, 1985.

[COOL] Nygaard, K.: *COOL – Comprehensive Object-Oriented Learning*, *http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_COOL1.html* (zuletzt besucht am 15.10.2006)

[KQPR03] Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: »The BlueJ system and its pedagogy«, *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology,* 13:4, S. 249 – 268, 2003.

[Mös05] Mössenböck, H.-P.: *Sprechen Sie Java? Eine Einführung in das systematische Programmieren (3. überarb. Auflage)*, dpunkt.verlag, 2005.

[PPP] Bergin, J.: *The Pedagogical Patterns Project*, *http://www.pedagogicalpatterns.org* (zuletzt besucht am 15.10.2006)

[RCS101] Stein, L. A.: *The Rethinking CS101 Project*, *http://www.cs101.org* (zuletzt besucht am 15.10.2006)

[Sch05] Schmolitzky, A.: »Towards Complexity Levels of Object Systems Used in Software Engineering Education (position paper)«, *Proc. Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, ECOOP 2005*, Glasgow, UK, 2005.

[Sch06a] Schmolitzky, A.: »Hochschullehre im Umbruch – Neue Lehrmethoden im softwaretechnischen Anteil des Informatikstudiums«, *LOG IN,* Heft 138/139, S. 48 – 54, 2006.

[Sch06b] Schmolitzky, A.: »Teaching Inheritance Concepts with Java«, *Proc. Principles and Practices of Programming in Java (PPPJ)*, Mannheim, Germany; ACM Press, 2006.

[Web00] Weber-Wulff, D.: »Combating the code warrior: a different sort of programming instruction«, *Proc. 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, Helsinki, Finland; ACM Press, S. 85 – 88, 2000.

[Weg87] Wegner, P.: »Dimensions of Object-Based Language Design«, *Proc. OOPSLA '87*, Orlando, Florida; ACM SIGPLAN Notices, Vol. 22, 12, 1987.

[Weg97] Wegner, P.: »Why Interaction is More Powerful than Algorithms«, *Communications of the ACM,* Vol. 40:5, S. 80 – 91, 1997.

Ein Experiment zur Ermittlung der Programmierproduktivität von Studenten

Matthias Wetzel

Universität Stuttgart
Universitätsstraße 38, 70569 Stuttgart
wetzelms@informatik.uni-stuttgart.de

Zusammenfassung

Unterschiede in der Programmierproduktivität zwischen einzelnen Programmierern sind ein wichtiger Gegenstand der Forschung. In diesem Beitrag wird ein zweiteiliges Experiment mit Softwaretechnikstudenten der Universität Stuttgart zur Ermittlung der individuellen Programmierproduktivität beschrieben. Im Ergebnis liegt zwischen der Programmierproduktivität der langsamsten und der schnellsten Studenten der Faktor 2,7. Nach einer Beschreibung des Experimentaufbaus und der Resultate wird auf Verbesserungsmöglichkeiten im Experimentdesign und Schlussfolgerungen aus den Ergebnissen im Hinblick auf mögliche Ansätze zur Verbesserung der Programmierausbildung im Studium eingegangen. Ein mittlerweile bereits umgesetzter Ansatz ist die Einrichtung zusätzlicher betreuter Programmierübungen parallel zu bestehenden Lehrveranstaltungen.

1 Einleitung

Spätestens seit der Studie von Sackman [Sac68] sind Unterschiede in der Programmierproduktivität zwischen einzelnen Programmierern ein wichtiger Gegenstand der Forschung. Auch wenn neuere Untersuchungen [Dic81] [Pre01] die Ergebnisse von Sackman zum Teil relativiert haben, sind in der Praxis zwischen schnellen und langsamen Programmierern Unterschiede von mindestens 4:1 im Hinblick auf die reine Programmierproduktivität zu erwarten [Pre01].

Im akademischen Umfeld stellt sich die Frage, welche Unterschiede zwischen Studienanfängern bestehen und ob sich diese Unterschiede im Lauf des Studiums

tendenziell vergrößern oder verkleinern. Die Beantwortung dieser Frage lässt u.a. Rückschlüsse auf eine optimale Gestaltung der praktischen Lehrveranstaltungen im Studium zu, insbesondere im Hinblick auf die gezielte Förderung schwächerer Studenten.

Dieser Beitrag beschreibt ein im Jahr 2005 an der Universität Stuttgart im Studiengang Softwaretechnik durchgeführtes Experiment, in dem die individuelle Programmierproduktivität von Studenten vor und nach einem Softwarepraktikum im Grundstudium verglichen wurde. Im Unterschied zu z.B. der Studie in [McC01], bei der ebenfalls die Programmierfähigkeiten von Studenten untersucht wurde, wurde dabei keine Bewertungsskala vorgegeben, sondern lediglich die Leistung der Studenten bei der Implementierung von Funktionalität miteinander verglichen.

Folgende Struktur liegt dem Beitrag zugrunde: In Kapitel 2 wird kurz die Programmierausbildung im Studiengang Softwaretechnik beschrieben. Kapitel 3 beschreibt die Durchführung und Kapitel 4 die Resultate des Experiments. In Kapitel 5 wird auf einige Probleme des Experimentdesigns eingegangen, und Kapitel 6 fasst die Ergebnisse zusammen und schließt den Beitrag mit einem Ausblick ab.

2 Programmierausbildung im Studiengang Softwaretechnik

Die im Studiengang Softwaretechnik durchgeführten praktischen Lehrveranstaltungen werden in [Lud97] ausführlich beschrieben. Für das durchgeführte Experiment ist neben den praktischen Lehrveranstaltungen *Programmierkurs* im ersten und *Softwarepraktikum* im dritten und vierten Semester insbesondere die Vorlesung *Programmentwicklung* im dritten Semester von Bedeutung.

Im Rahmen des Programmierkurses sollen den Studenten im ersten Semester die Grundlagen des »Programming in the small« gemäß softwaretechnischen Prinzipien vermittelt werden. Dabei müssen die Studenten in Dreiergruppen zunehmend komplexer werdende Aufgabenstellungen in Ada-95 implementieren. Besonderer Wert wird dabei auf die Einhaltung eines Styleguides und eine gute Kommentierung gelegt. Ergänzend zum Programmierkurs wird ein freiwilliger Stützkurs für Studienanfänger ohne Vorkenntnisse im Programmieren angeboten, der diesen im ersten Semester die notwendigen Grundkenntnisse des Programmierens vermitteln soll.

Das Softwarepraktikum beginnt in der Mitte des dritten Semesters und dauert bis zum Ende der Vorlesungszeit des vierten Semesters. Im Rahmen dieser Veranstaltung führen die Studenten – wiederum in Dreiergruppen – ein komplettes Softwareprojekt durch, von der Analyse über Spezifikation, Entwurf, Implementierung in der Programmiersprache Java, Test und Abnahme. Sowohl hier als auch im Programmierkurs ist der Schein Voraussetzung für das Vordiplom.

In der Vorlesung Programmentwicklung werden den Studenten im dritten Semester die Grundlagen der objektorientierten Programmierung vermittelt.

Neben einer Einführung in Java wird dabei u.a. auf Design Patterns, Architektur und Test von OO-Programmen sowie die UML eingegangen. Die Bearbeitung der Java-Implementierungsaufgabe für den – nicht zwingend für das Vordiplom erforderlichen – Schein kann individuell oder in Kleingruppen erfolgen.

Begleitend zu diesen praktischen Lehrveranstaltungen vermitteln die Vorlesungen *Einführung in die Informatik I & II* die theoretischen Grundlagen des Programmierens.

3 Beschreibung des Experiments

3.1 Ziele des Experiments

Mit Hilfe des Experiments sollten die folgenden Fragen untersucht werden:

- Welchen Stand haben die Programmierfähigkeiten der Studenten zu Beginn des vierten Semesters (also nach dem Programmierkurs und der Vorlesung Programmentwicklung, aber vor dem Implementierungsteil des Softwarepraktikums) allgemein und im Vergleich miteinander?
- Wie entwickeln sich diese Fähigkeiten bis zum Ende des vierten Semesters nach Abschluss des Softwarepraktikums?

Diese Fragen sollten insbesondere unter dem Gesichtspunkt der unterschiedlichen Vorkenntnisse bei Studienbeginn untersucht werden. Da einige Studenten während des Softwarepraktikums ihr Studium abbrechen, sollte auch untersucht werden, ob ein Zusammenhang zwischen Abbruchwahrscheinlichkeit und vorhandenen Programmierkenntnissen bei Studienbeginn besteht.

Während die Beantwortung der ersten beiden Fragen Aufschluss über den Beitrag verschiedener Lehrveranstaltungsformen zur Vermittlung von Programmierfähigkeiten im Rahmen des Studiums geben soll, könnte zusätzliches Wissen über die Ursache von Studienabbrüchen dabei helfen, die bei Informatikstudiengängen relativ hohe Zahl von Studienabbrechern zu reduzieren.

3.2 Aufbau des Experiments

Um die Programmierfähigkeiten der Studenten vor und nach dem Softwarepraktikum messen zu können, fand das Experiment in zwei Teilen statt, die ab hier mit Teil 1 und Teil 2 bezeichnet werden:

- Teil 1 wurde zu Beginn des vierten Semesters im April und damit nach Abschluss des Spezifikationsteils, aber vor Beginn der Implementierungsarbeiten im Softwarepraktikum durchgeführt.
- Teil 2 wurde nach Ende der Vorlesungszeit des vierten Semesters im Juli und damit nach dem vollständigen Abschluss des Softwarepraktikums durchgeführt.

Bei beiden Teilen ging es um die Implementierung einer vorgegebenen Aufgabenstellung in der Programmiersprache Java mit Hilfe der IDE Eclipse. Um in dem gegebenen Zeitraum von sieben Stunden eine etwas umfangreichere Aufgabenstellung bearbeiten zu können, standen den Teilnehmern bei beiden Teilen einige Java-Klassen mit bereits fertig implementierten Basisfunktionalitäten wie z.B. Schreib- und Leseoperationen für Dateien zur Verfügung. Nach einer kurzen Präsentation, bei der den Studenten die allgemeinen Rahmenbedingungen erläutert wurden, fand die individuelle Bearbeitung der Aufgabenstellung in einem Computerpool statt, wobei jedem Teilnehmer ein eigener Rechner zur Verfügung stand. Die Rechner liefen unter Linux, die Entwicklungsumgebung Eclipse mit den vorgegebenen Klassen und die Beschreibung der Aufgabenstellung waren vorinstalliert. Ein Internetzugang war vorhanden, und die Nutzung des Internets zur Recherche war ebenso gestattet wie das Mitbringen von Programmierbüchern; lediglich der Austausch zwischen den Studenten war verboten.

Die erfolgreiche Teilnahme, nachgewiesen durch eine Unterschrift nach Beendigung jedes Teils, war eine Voraussetzung für den Erhalt des Softwarepraktikum-Pflichtscheins; falls nach Ablauf von sieben Stunden absehbar war, dass dieser Teil des Experiments nicht erfolgreich abgeschlossen werden würde, war ein Abbruch ohne Verlust des Scheins aber zulässig. Erfolg bedeutete dabei jeweils das Bestehen eines vorgegebenen Abnahmetests. Dieser stand den Teilnehmern während des Experiments zur Verfügung; sie konnten also jederzeit selbst feststellen, ob ihre Lösung den Abnahmekriterien genügte. Nach Abschluss jedes Teils musste von jedem Studenten unabhängig vom Erfolg ein Fragebogen ausgefüllt werden. Zur Bewertung der Qualität der erfolgreichen Abgaben wurden nach Ende des Experiments weitere Tests durchgeführt, die den Teilnehmern nicht im Voraus bekannt waren.

Die Aufgabe bestand bei Teil 1 aus dem Einlesen einer Datei mit Datensätzen und der Ausgabe der sortierten und umformatierten Datensätze. Die Klassen mit Lese- und Schreiboperationen zum Einlesen und Ausgeben der Dateien waren vorgegeben. Bei Teil 2 musste eine Musterlösung aus Teil 1 um eine Funktion zur Filterung der Datensätze anhand von Kriterien, die aus einer weiteren Datei eingelesen werden mussten, erweitert werden. Außerdem wurden die Teilnehmer in vier Gruppen eingeteilt, die unterschiedliche Kombinationen von Begleitdokumenten erhielten (Algorithmusbeschreibung mit Klassendiagramm der vorgegebenen Klassen und Ablaufschilderung einerseits sowie aus zusätzlich eingefügten Javadoc-Kommentaren generierte HTML-Dateien andererseits). Ziel der Gruppenaufteilung war es, mögliche Auswirkungen vorhandener Dokumentation auf die zur Lösung der Aufgabe benötigte Zeit und die Qualität der erzeugten Programme zu untersuchen.

4 Resultate des Experiments

4.1 Teil 1

Teilnehmer

Insgesamt nahmen 67 Studenten an Teil 1 des Experiments teil, von denen allerdings 15 Sonderfälle aus verschiedenen Gründen bei der Auswertung nicht berücksichtigt werden konnten. Gründe dafür waren u.a. die Teilnahme an einem Ersatztermin unter nicht kontrollierten Bedingungen wegen Krankheit, Täuschungsversuchen, nicht erfolgter Abgabe des Fragebogens oder unerlaubtem Abbrechen des Experiments. Da zumindest die Teilnehmer aus den letzten drei Gruppen eine unterdurchschnittliche Leistung aufwiesen, sind die Leistungsdurchschnitte und -quantile tendenziell eher schlechter als im Folgenden angegeben. Für die Auswertung verbleiben damit 52 von 67 Studenten, was einer Quote von 78% entspricht.

Coderesultate

Von den 52 betrachteten Studenten haben 34 Studenten den Abnahmetest von Teil 1 bestanden, das entspricht einer Quote von 65%. Demnach haben es 18 bzw. 35% der Studenten – also etwas mehr als ein Drittel – nicht geschafft, die gestellte Aufgabe innerhalb der zur Verfügung stehenden Zeit von sieben Stunden so weit zu lösen, dass zumindest der Abnahmetest bestanden wurde. Drei Teilnehmer haben überhaupt keinen Programmcode geschrieben, sondern lediglich leere Klassen mit einer main-Methode generiert. Von den 18 nicht erfolgreichen Abgaben waren bei Experimentabbruch neun nicht kompilierbar, enthielten also syntaktische Fehler.

Zur besseren Einschätzung des Aufwands wurde die gestellte Aufgabe im Vorfeld von zwei Mitarbeitern des Lehrstuhls gelöst. Ein Mitarbeiter mit sehr guten Java-Kenntnissen benötigte für eine etwas komplexere Version der Aufgabenstellung 3:10 Stunden, woraufhin die Aufgabenstellung durch Verzicht auf einige Sonderfälle vereinfacht wurde. Der andere Mitarbeiter mit mittleren Java-Kenntnissen benötigte für die Bearbeitung der späteren Aufgabenstellung 3:00 Stunden.

Die erfolgreichen Teilnehmer benötigten zwischen 2:10 Stunden und 8:40 Stunden für die erfolgreiche Bearbeitung der Aufgabe (siehe Abb. 1). Der Median lag bei 4:25 und der Mittelwert bei 4:42 Stunden. Berücksichtigt man alle 52 Studenten, liegt der Median bei 5:10 Stunden. Aufgrund der großen Zahl nicht erfolgreicher Teilnehmer können die *LS50*- und *LS25*-Verhältnisse nach [Pre01] nicht sinnvoll gebildet werden.

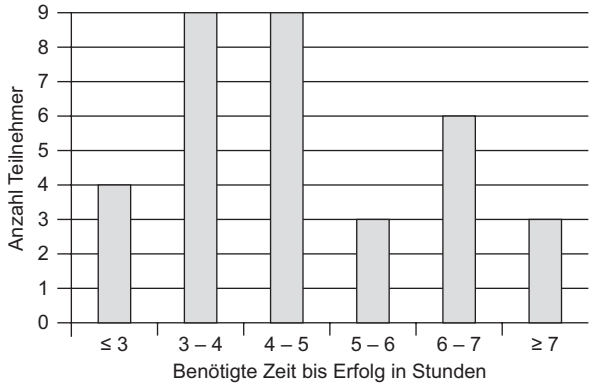


Abb. 1 *Benötigte Zeit der 34 erfolgreichen Teilnehmer in Std. / Anzahl Teilnehmer*

Zur differenzierten Bewertung der Qualität der Abgaben wurden acht zusätzliche Testfälle definiert und jede erfolgreiche Abgabe damit getestet. Die 34 erfolgreichen Abgaben ergaben in 0 – 8 Testfällen korrekte Ergebnisse; der Median lag bei 6 und der Mittelwert bei 5,12. Ursachen für die falschen Ergebnisse waren teils Programmierfehler, teils aber auch Täuschungsversuche wie die statische Festlegung bestimmter Werte im Programm, die eigentlich dynamisch hätten eingelesen werden sollen. Das Streudiagramm in Abbildung 2 zeigt für jede Abgabe die benötigte Zeit bis zum erfolgreichen Abschluss des Experiments in Stunden und die jeweilige Anzahl der Testfälle mit korrektem Ergebnis. Die Korrelation zwischen der benötigten Zeit und der Anzahl der Testfälle mit korrektem Ergebnis ist nach Pearson mit einem Wert von -0,455 auf dem Niveau von 0,01 (2-seitig) signifikant. Die schnelleren Teilnehmer produzierten also zumindest tendenziell auch eine bessere Codequalität.

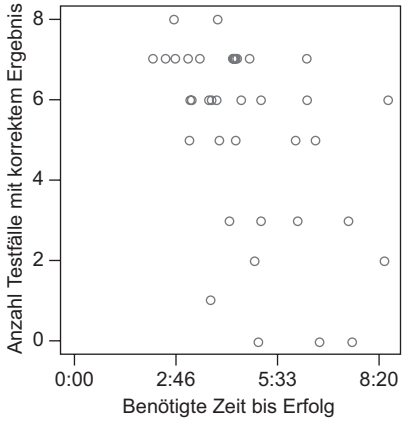


Abb. 2 *Bis zum Erfolg benötigte Zeit in Std. / Anzahl Testfälle mit korrektem Ergebnis*

Fragebogen

In dem nach Beendigung von Teil 1 auszufüllenden Fragebogen wurde u.a. gefragt, wie die Studenten Programmieren gelernt haben; Mehrfachnennungen waren dabei zulässig. Demnach haben sich 31 Studenten das Programmieren weitgehend selbst beigebracht. Jeweils 15 Studenten gaben an, in Schule oder Hochschule Programmieren gelernt zu haben, bei 4 Studenten traf dies für eine Beschäftigung zu, der sie nachgegangen sind.

Zwischen dem Vorhandensein von Programmierkenntnissen vor dem Studium und dem Erfolg bei Teil 1 des Experiments besteht eine schwach positive Korrelation, die nach Pearson mit einem Wert von 0,308 auf dem Niveau von 0,05 (2-seitig) signifikant ist. Tendenziell waren Studenten, die bereits vor dem Studium Programmiererfahrungen gesammelt hatten, also erfolgreicher bei Teil 1. Eine ähnliche schwach positive Korrelation besteht zwischen der Programmiererfahrung in bereits geschriebenen LOC und dem Erfolg bei Teil 1; diese ist nach Pearson mit einem Wert von 0,286 auf dem Niveau von 0,05 (2-seitig) signifikant. Zwischen dem Vorhandensein von Programmiererfahrungen vor dem Studium und den geschriebenen LOC besteht keine statistisch signifikante Korrelation, ebenso wenig wie zwischen den Programmiererfahrungen und der bei Teil 1 benötigten Zeit bis zum Erfolg.

Weiterhin wurde im Fragebogen nach der Anzahl bisher im Studium erworbener Scheine gefragt. Dabei gab es einen Unterschied zwischen den bei Teil 1 erfolgreichen Studenten und den nicht erfolgreichen: Letztere hatten im Mittel 8,72 Scheine erworben (Standardabweichung 1,406), Erstere hingegen im Mittel 9,44 Scheine (Standardabweichung 1,795). Die Differenz von 0,72 ist zwar nicht statistisch signifikant, aber praktisch bedeutsam, weil die absolute Minimalanzahl von Scheinen zum Erreichen des vierten Semesters fünf beträgt, die sinnvolle Minimalzahl sieben. Die bei Teil 1 erfolgreichen Studenten haben damit 30% mehr Scheine über das sinnvolle Mindestmaß hinaus freiwillig erworben als die nicht erfolgreichen und scheinen damit auch über das Programmieren hinaus im Studium tendenziell etwas engagierter zu sein.

4.2 Teil 2

Teilnehmer

Die Teilnahme und der Erfolg der 52 betrachteten Studenten ergab sich bei Teil 2 wie folgt:

- 62% (32 Studenten) haben bei beiden Teilen erfolgreich teilgenommen.
- 4% (2 Studenten) haben bei Teil 1 erfolgreich teilgenommen und bei Teil 2 nicht teilgenommen.

- 13% (7 Studenten) haben bei Teil 1 ohne Erfolg und bei Teil 2 erfolgreich teilgenommen.
- 6% (3 Studenten) haben bei beiden Teilen ohne Erfolg teilgenommen.
- 15% (8 Studenten) haben bei Teil 1 ohne Erfolg teilgenommen und bei Teil 2 nicht teilgenommen.
- Es gab keine Studenten, die bei Teil 1 erfolgreich teilgenommen haben und bei Teil 2 ohne Erfolg teilgenommen haben.

Da die Teilnahme am gesamten Experiment Voraussetzung für den Erhalt eines Pflichtscheins für das Vordiplom war, kann davon ausgegangen werden, dass die zehn Studenten, die nicht an Teil 2 teilgenommen haben, das Studium abgebrochen haben. Drei davon haben zwischen den beiden Experimentteilen aufgrund des Nichtbestehens von Prüfungen (Mathematik bzw. Praktische Informatik) endgültig den Prüfungsanspruch verloren; die Ursache für den Studienabbruch der anderen sieben ist unbekannt, könnte aber mit Schwierigkeiten im SoPra zusammenhängen.

Um die Leistung der mit verschiedenen Begleitdokumenten ausgestatteten Gruppen (siehe Abschnitt 3.2) vergleichen zu können, wurde versucht, die teilnehmenden Studenten möglichst homogen aufzuteilen, wobei als Zuteilungskriterium die bis zum Erfolg bei Teil 1 benötigte Zeit verwendet wurde. Die bei Teil 1 nicht erfolgreichen Studenten wurden ebenfalls gleichmäßig auf die Gruppen verteilt. Aufgrund von Studienabbrüchen und Krankmeldungen konnte aber keine vollständig homogene Verteilung erzielt werden.

Coderesultate

Von den 42 berücksichtigten Teilnehmern an Teil 2 haben 39 den Abnahmetest bestanden, was einer Quote von 93% entspricht. Von den drei nicht erfolgreichen Abgaben waren zwei gegenüber den vorgegebenen Klassen in höchstens 20 Zeilen verändert, die dritte war nicht kompilierbar.

Auch die Aufgabe des zweiten Teils war im Vorfeld von zwei Mitarbeitern des Lehrstuhls bearbeitet worden, wobei zur Lösung 1:20 bzw. 1:00 Stunden (mittlere bzw. sehr gute Java-Kenntnisse) benötigt wurden. Bei den Studenten lag der Median der bis zum Erfolg benötigten Zeit in allen Gruppen mit verschiedener Begleitdokumentation (siehe Abschnitt 3.2) unter 2:00 Stunden; bis auf zwei Ausreißer haben alle erfolgreichen Teilnehmer weniger als vier Stunden zur Lösung der Aufgabe benötigt. Die Unterschiede zwischen den einzelnen Gruppen hinsichtlich der benötigten Zeit sind statistisch nicht signifikant. Da nur drei von 42 Studenten diesen Teil nicht erfolgreich abgeschlossen haben, können zum Vergleich der Programmierleistung die von [Pre01] empfohlenen Maße *LS50* und *LS25* herangezogen werden. *LS* steht dabei für Langsam-Schnell-Verhältnis und zeigt anschaulich, wie viel Mal länger eine mittlere »langsame« Versuchsperson im Verhältnis zur mittleren »schnelleren« Versuchsperson benötigt:

- *LS50* ist der Quotient von 75- und 25-Percentil (q_{75}/q_{25}) und entspricht damit dem Verhältnis der langsameren zur schnelleren Hälfte.
- *LS25* ist der Quotient von 87,5- und 12,5-Percentil ($q_{87,5}/q_{12,5}$) und entspricht damit dem Verhältnis des langsamsten zum schnellsten Viertel.

Mit q_{75} = 2:00 Stunden und q_{25} = 1:15 Stunden ergibt sich *LS50* = 120 min / 75 min = 1,6. Die langsamere Hälfte brauchte demnach im Mittel etwas mehr als anderthalb mal so lange wie die schnellere Hälfte. Aus $q_{87,5}$ = 2:52 Stunden und $q_{12,5}$ = 1:04 Stunden ergibt sich *LS25* = 172 min / 64 min = 2,7. Das langsamste Viertel brauchte also im Mittel etwas mehr als zweieinhalb mal so lang wie das schnellste Viertel.

Zur Bewertung der Qualität der Abgaben wurden 13 Testfälle definiert und die erfolgreichen Abgaben damit getestet. Im Unterschied zu den Testergebnissen bei Teil 1 war die Qualität der Abgaben deutlich homogener:

- Der Median von Testfällen mit korrektem Ergebnis lag in Gruppe 1 bei 12 und in allen anderen Gruppen bei 13 Testfällen.
- Bis auf zwei Ausreißer bestanden alle erfolgreichen Abgaben mindestens acht Testfälle.

Die Korrelation zwischen bis zum Erfolg benötigter Zeit und bestandenen Testfällen war im Unterschied zu Teil 1 nicht statistisch signifikant.

Fragebogen

Mit dem Fragebogen nach Teil 2 sollte untersucht werden, wie sich die Programmierkenntnisse der Studenten während des Softwarepraktikums entwickelt haben. Insbesondere stand dabei die Frage im Mittelpunkt, ob sich vor dem SoPra bestehende Unterschiede während des SoPras verkleinern oder vergrößern würden.

Zwischen der Anzahl vor dem SoPra geschriebener LOC und der Anzahl LOC, die während des SoPras geschrieben wurden, gibt es eine Korrelation, die nach Pearson mit einem Wert von 0,607 auf dem Niveau von 0,01 (2-seitig) statistisch signifikant ist. Die Teilnehmer, die bereits vor dem SoPra mehr Programmiererfahrung gesammelt hatten, haben also auch tendenziell während des SoPras innerhalb des Teams einen größeren Teil der Programmieraufgaben übernommen. Zwischen der Anzahl während des SoPras geschriebener LOC und der bei Teil 2 bis zum Erfolg benötigten Zeit gibt es aber keine statistisch signifikante Korrelation, ebenso wenig wie zwischen der Anzahl vor dem SoPra geschriebener LOC und der bei Teil 2 bis zum Erfolg benötigten Zeit.

Zusätzlich wurde im Fragebogen nach Verwendung und Einschätzung der Nützlichkeit der Algorithmusdokumentation und der aus zusätzlichen Javadoc-Kommentaren generierten HTML-Seiten gefragt. Demnach haben nur zwei Drittel der Studenten eine vorhandene Algorithmusdokumentation benutzt. Von diesen

zwei Dritteln empfanden wiederum zwei Drittel und damit 44% der Ausgangsmenge die Algorithmusdokumentation als nützlich. Zur Verfügung gestelltes Javadoc wurden ebenfalls von zwei Dritteln genutzt, von diesen allerdings nur zur Hälfte und damit von 33% der Ausgangsmenge als nützlich bewertet. Von den Teilnehmern, denen keine Algorithmusdokumentation zur Verfügung stand, gaben 45% an, dass eine solche Dokumentation die Aufgabe vermutlich erleichtert hätte. Bei Javadoc waren 35% dieser Ansicht. Setzt man voraus, dass Studenten, die eine zur Verfügung stehende Dokumentation nicht benutzten, diese implizit als nicht nützlich bewerteten, ergibt sich eine nahezu perfekte Übereinstimmung der Beurteilung der Nützlichkeit von Dokumentationstypen unabhängig von deren tatsächlichem Vorliegen in der konkreten Situation. Diese Übereinstimmung könnte möglicherweise auf die Existenz grundlegend verschiedener und individuell bereits im Studium gefestigter Herangehensweisen an Programmierprobleme hindeuten. Auch wenn in diesem Experiment keine statistisch signifikante Korrelation zwischen benutzter und als nützlich oder nicht nützlich eingestufter Dokumentation und Erfolg aufgetreten ist, erscheint eine weitergehende Forschung in diesem Bereich dennoch lohnend.

5 Schwachpunkte im Experimentdesign

Die folgenden vier Punkte haben sich bei Durchführung und Auswertung des Experiments als mögliche Schwachstellen gezeigt, die bei einer Wiederholung des Experiments genauer betrachtet und evtl. anders gehandhabt werden sollten:

5.1 Code&Fix anstelle von Software Engineering

Die Rahmenbedingungen des Experiments führten dazu, dass die reine Programmierleistung im Sinne der Implementierung von Funktionalität im Mittelpunkt stand: Die im Studiengang Softwaretechnik vermittelten Grundsätze guten Software Engineering wie die Erstellung einer Spezifikation, der gründliche Entwurf einer tragfähigen Architektur, sorgfältige Implementierung mit guter Kommentierung sowie planvolles Testen spielten keine wesentliche Rolle. Im Gegenteil begünstigte der den Teilnehmern ständig zur Verfügung stehende Abnahmetest – bei dem als einziges Erfolgskriterium eine bestimmte Funktionalität verlangt wurde und die Programmqualität nicht bewertet wurde – einen Code&Fix-Ansatz. Allerdings wurde bei dem Experiment auch die Anzahl fehlgeschlagener Abnahmetests mitprotokolliert, die von den Teilnehmern selbst durchgeführt worden waren: danach war mit 13 von 52 Teilnehmern bei einem Viertel bereits der erste durchgeführte Abnahmetest erfolgreich, mehr als drei Viertel haben nur acht oder weniger fehlgeschlagene Abnahmetests durchgeführt. Lediglich bei zwei Teilnehmern ist mit 41 und 44 fehlgeschlagenen Abnahmetests von einem extremen Code&Fix-Ansatz auszugehen.

Die Vernachlässigung der Programmqualität und der Nicht-Implementierungsaspekte der Softwareentwicklung bei einem solchen Experiment führen sicherlich dazu, dass die Ergebnisse keine direkten Rückschlüsse auf die Gesamtleistung der Teilnehmer als Softwareentwickler zulassen. Andererseits sollte jeder Entwickler das grundlegende Handwerkszeug des Programmierens beherrschen, auch wenn er oder sie später hauptsächlich spezifiziert oder testet. Während also ein gutes Abschneiden bei einem derartigen Experiment nicht zwangsläufig einen guten Software-Ingenieur ausweist, deutet ein stark unterdurchschnittliches Ergebnis auf Lücken in der bisherigen Ausbildung hin.

5.2 Anonymisierung

Um datenschutzrechtliche Bedenken von vornherein auszuschließen, wurden die Ergebnisse vollständig anonymisiert erhoben. Zu diesem Zweck wurde von jedem Studenten bei der Präsentation der Aufgabenstellung von Teil 1 ein Linux-Benutzername in der Form exp*Zufallsnummer* mitsamt zugehörigem Benutzerpasswort zufällig gezogen. Die Programmabgaben und die ausgefüllten Fragebögen konnten damit nur einem bestimmten Benutzernamen bzw. der Zufallsnummer zugeordnet werden. Die Unterschrift unter eine Teilnehmerliste zum Zweck der Überprüfung der Mitwirkung für den Schein erfolgte bei Verlassen des Poolraums ohne Zuordnung zu einer speziellen Abgabe.

Diese Vorgehensweise sicherte zwar weitgehende Anonymität zu (abgesehen allenfalls von Krankheitsfällen), sie brachte aber auch eine Reihe von Problemen mit sich. So konnten sich viele Studenten bei Teil 2 trotz im Vorfeld erfolgter Hinweise nicht an ihre Zufallsnummer erinnern, was die Auswertung erschwerte. Ohne die Zuordnung der Zufallsnummern zu Namen war es auch nicht möglich, Studenten, die das Studium aufgrund des Nichtbestehens von Prüfungen aufgeben mussten, auf Experimentteilnehmer abzubilden. Rückblickend erscheinen eine im Vorfeld garantierte, strikte Geheimhaltung von Experimentdaten und eine anonymisierte Veröffentlichung der Ergebnisse unter Verzicht auf Maßnahmen zur vollständigen Anonymisierung die bessere Alternative zu sein.

5.3 Effekte zusätzlicher Dokumentation als Experimentvariable

Durch die Aufteilung in vier Gruppen bei Teil 2 des Experiments sollten die Effekte zusätzlich zur Verfügung stehender Dokumentation auf die Programmierproduktivität bei einer Wartungsaufgabe untersucht werden. Dies ist unter anderem wegen der zu kleinen Fallgruppen nicht gelungen. Bei einer Teilnehmermenge von ca. 40 sollte maximal ein zusätzliches Unterscheidungsmerkmal eingeführt werden, damit die interessierenden Fallgruppen groß genug sind. Hinzu kommt, dass die Einführung zusätzlicher Variabilität in nur einem Experimentteil den Vergleich erschwert.

5.4 Umfang und Art der Aufgabenstellung

Der Umfang der Aufgabenstellung bei Teil 1 des Experiments war zu groß gewählt. Im Nachhinein lässt sich zumindest für Teil 2 zeigen, dass das schnellste Viertel der Studenten im Mittel etwa so lange bis zum Erfolg gebraucht hat wie die Institutsmitarbeiter, die die Aufgabe im Vorfeld bearbeitet hatten. Wegen *LS25* = 2,7 braucht das langsamste Viertel der Studenten im Mittel also mehr als zweieinhalb mal so lange wie die Institutsmitarbeiter. Um mindestens 7/8 der Studenten die Möglichkeit zu geben, das Experiment erfolgreich zu beenden – was gleichzeitig auch die Aussagekraft der gewonnenen Werte erhöht –, sollte bei der Aufgabenstellung also darauf geachtet werden, dass Institutsmitarbeiter, die die Aufgabe im Vorfeld lösen, nicht mehr als 1/4 bis 1/3 der den Studenten maximal zur Verfügung stehenden Zeit dafür benötigen (mittlere Kenntnisse der Institutsmitarbeiter in der verwendeten Programmiersprache vorausgesetzt). Darüber hinaus wäre überlegenswert, den Erfolg bei einem derartigen Programmierexperiment nicht nur an der Lösung einer einzigen Aufgabenstellung festzumachen, sondern stattdessen mehrere, nach Schwierigkeitsgrad abgestufte Aufgaben zu stellen. Eine solche feinere Skala für den Erfolg hätte den Vorteil, auch schwächeren Teilnehmern ein erreichbares Ziel zu bieten, würde allerdings auch die Verwendung der benötigten Zeit als einfachen Vergleichsmaßstab erschweren.

6 Ergebnis und Ausblick

Das durchgeführte Experiment hat gezeigt, dass die Leistungsunterschiede der Studenten beim Programmieren vor der Durchführung des Softwarepraktikums erheblich sind. Etwas mehr als ein Drittel der Studenten war zu Beginn des vierten Semesters trotz der Lehrveranstaltungen in den ersten drei Semestern nicht in der Lage, die gestellte Java-Programmieraufgabe zu lösen; ein Sechstel der Studenten beherrschte nicht einmal die grundlegende Java-Syntax.

Erst die Implementierungsarbeit im SoPra scheint zu einer Angleichung der Programmierproduktivität zu führen: Nach dem SoPra sind nur noch 7% der dann noch verbleibenden Studenten nicht in der Lage, die gestellte Java-Programmieraufgabe zu lösen. Der *LS25*-Wert von 2,7 bei Teil 2 liegt unterhalb des in der Literatur angegebenen Verhältnisses von 4:1, so dass es sich bei den Studenten nach dem SoPra um eine relativ homogene Gruppe zu handeln scheint. Auch wenn sich die vor dem SoPra vorhandene Programmiererfahrung auf die im SoPra geleistete Programmierarbeit auswirkt, scheint sie auf die Programmierproduktivität nach dem SoPra keinen wesentlichen Einfluss mehr auszuüben.

Allerdings scheint die Aneignung der notwendigen Programmierpraxis im SoPra parallel zur Bearbeitung der eigentlichen Aufgabe des SoPras einige Studenten vor größere Probleme zu stellen. Von den bei Teil 1 des Experiments nicht erfolgreichen Teilnehmern haben 44% das Studium zwischen Teil 1 und Teil 2

abgebrochen, während diese Quote bei den erfolgreichen Teilnehmern von Teil 1 nur 6% beträgt. Nicht bestandene Prüfungen können dafür nur teilweise die Ursache sein, so dass die Vermutung naheliegt, dass Schwierigkeiten mit der Programmierung im SoPra mit zum Studienabbruch beitragen. Unter anderem um diesen Schwierigkeiten vorzubeugen, werden jetzt im Studiengang Softwaretechnik begleitend zur Vorlesung Programm-entwicklung im dritten Semester betreute Programmierübungen in Java angeboten – damit alle Studenten schon vor dem SoPra eine ausreichende Programmierpraxis haben und im SoPra nicht plötzlich vor unlösbaren Schwierigkeiten stehen.

Literatur

[Dic81] T. E. Dickey, Programmer variability, Proc. IEEE 69, 7 (Jul. 1981), 844 – 846.

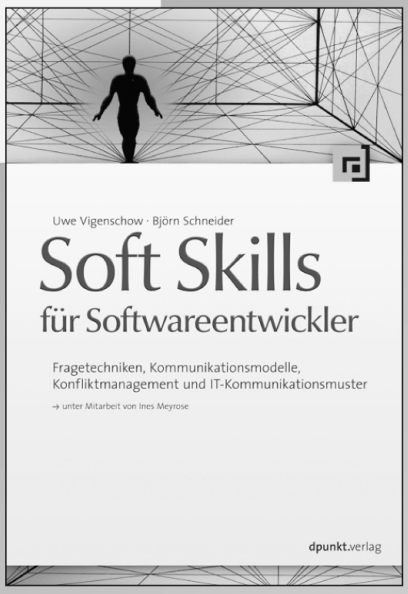
[Lud97] Ludewig, J. Der Modellstudiengang Softwaretechnik. in P. Forbrig, G. Riedewald (Hrsg.): SEUH'97 (Software Engineering im Unterricht der Hochschulen). Berichte des German Chapter of the ACM, Band 48, Teubner, Stuttgart, 9 – 23.

[McC01] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. SIGCSE Bull. 33, 4 (Dec. 2001), 125 – 180.

[Pre01] Lutz Prechelt. Kontrollierte Experimente in der Softwaretechnik. Springer Verlag, 2001.

[Sac68] Sackman, H., Erikson, W. J., and Grant, E. E. Exploratory experimental studies comparing online and offline programming performance. Commun. ACM 11, 1 (Jan. 1968), 3 – 11.

[SoPra] Webseite der Abteilung SE der Universität Stuttgart zum SoPra-Experiment 2005: *http://www.iste.uni-stuttgart.de/se/teaching/courses/sopra/experiment/index.html*



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

2006, 342 Seiten, gebunden
€ 39,00 (D)
ISBN 978-3-89864-275-0

Andreas Spillner · Thomas Roßner
Mario Winter · Tilo Linz

Praxiswissen Softwaretest – Testmanagement

Aus- und Weiterbildung zum Certified Tester – Advanced Level nach ISTQB-Standard

In diesem Buch werden praxiserprobte Testmanagement-Techniken vorgestellt und anhand eines durchgängigen Beispiels erklärt. Behandelt werden u.a. Dokumentation, Testplanung, risikoorientiertes Testen, Fehler und Abweichungsmanagement, Optimierung des Testprozesses und Teamzusammensetzung. Fallstudien aus der Industrie illustrieren, wie Testmanager in verschiedenen Zweigen der Softwarebranche in großen und kleinen Projekten die täglichen Aufgaben und Herausforderungen des Testmanagements erfolgreich lösen.

Das Buch ist so aufbereitet, dass es für das Selbststudium geeignet ist. Der Inhalt umfasst den prüfungsrelevanten Stoff des Moduls »Test Manager« des Certified Tester (Advanced Level) nach dem Standard des ISTQB.

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