SEUH 2013
aachen

Tagungsband 13. Workshop

SEUH

Software Engineering im Unterricht der Hochschulen

28. Februar – 1. März 2013
RWTH Aachen

Herausgeber:
Andreas Spillner (Hochschule Bremen)
Horst Lichter (RWTH Aachen)

http://ceur-ws.org/Vol-956/

Vorwort

Seit über 20 Jahren wird auf den SEUH-Workshops über Schwierigkeiten und Möglichkeiten diskutiert und beraten, wie wir Studierenden die Vorgehensweisen bei der Softwareentwicklung am besten vermitteln. Seit den ersten Workshops hat sich einiges verändert, es gibt aber auch vieles, was unverändert geblieben ist.

Wenn wir Software Engineering unterrichten, stehen wir immer wieder vor der Herausforderung, wie wir den Studierenden auch in kleinen Projekten die Vorteile von methodischem Vorgehen deutlich machen können. Da die meisten Studierenden keine Erfahrung mit großen Softwaresystemen haben, sehen sie nicht immer ein, warum der durch ein methodisches Vorgehen begründete Aufwand auch bei kleinen Ausbildungsprojekten gerechtfertigt und lohnend ist. Diese Problematik hat uns bei der SEUH von Anfang an beschäftigt. Studentische Ausbildungsprojekte waren ein Ansatz, dieses Problem zu überwinden.

Die Beiträge zur 13. SEUH haben wir in fünf Themenschwerpunkte gruppiert.

Im Schwerpunkt *Just-in-Time Teaching* greifen einige Autoren dieses SEUH-Generalthema auf und stellen neue Lösungsmöglichkeiten vor. Dabei soll den Studierenden durch schnelle Rückkopplung und Diskussion die Problematik verdeutlicht und näher gebracht werden.

Dass die Ausbildung in den Bereichen Programmierung und Software Engineering viele Schnittstellen hat, ist unbestritten. Diesem Thema widmen sich die Beiträge unter dem Stichpunkt *Programmierausbildung*. Sie diskutieren, wie diese Schnittstellen zum Vorteil von beiden Ausbildungsschwerpunkten genutzt werden können und wie die enge Verzahnung vorteilhaft umgesetzt werden kann.

Agile Methoden haben in den letzten Jahren ihren Weg in die Praxis gefunden, sie werden dort erprobt und eingesetzt. Die SEUH greift dieses Thema im Schwerpunkt *Agile Methoden im Unterricht* auf. Es werden erste interessante Erfahrungen berichtet, über die auf dieser SEUH sicherlich intensiv diskutiert werden wird.

Werkzeuge spielen in der Ausbildung eine wesentliche Rolle, entweder zur Entwicklung von Software oder aber als Mittel, um Lerninhalte effizient und effektiv vermitteln zu können. Dieses spiegeln die Beiträge zum Thema *Werkzeuge in und für die Ausbildung* wider. Dort werden ein Werkzeug zur spielerischen Vermittlung von Entwicklungsprozessen und ein Werkzeug zur Auswahl und zum Einsatz von Metriken vorgestellt.

Daneben gibt es auch in diesem Jahr wieder neue Ideen, wie wir unser Wissen im Software Engineering besser und nachhaltiger an Studierende vermitteln können. Unter dem Punkt *Zur Diskussion gestellt* werden diese Vorschläge vorgestellt und erörtert.

Für die SEUH konnten wir mit Bernd Brügge und Peter Zimmerer zwei sehr interessante Persönlichkeiten als eingeladene Vortragende gewinnen. Bernd Brügge, TU München, berichtet über seine Erfahrungen beim Multimedia-Einsatz in Software Engineering Projektkursen. Peter Zimmerer von der Siemens AG präsentiert das Software Engineering Curriculum in der betrieblichen Weiterbildung bei der Siemens AG und berichtet über die gemachten Erfahrungen sowie über kritische Erfolgsfaktoren.

Organisatorisch gibt es 2013 eine Neuerung: Die SEUH wird als Teil der »Multikonferenz Software Engineering« durchgeführt. Diese bündelt die deutschsprachigen Tagungen im Bereich Software Engineering: SEUH, SE-2013 (Motto Software – Innovator für Wirtschaft und Gesellschaft) und SEE (Software & Systems Engineering Essentials).

Dadurch wird die RWTH Aachen Ende Februar 2013 zum Treffpunkt der deutschsprachigen Software Engineering Community.

Unser besonderer Dank gilt den Autoren, die eine Vielzahl von Beiträgen eingereicht und somit erst eine Auswahl ermöglicht haben. Die Auswahl der Beiträge und die thematische Struktur der diesjährigen SEUH wurden wie immer von einem Programmkomitee vorgenommen. Folgende Hochschullehrerinnen und Hochschullehrer haben daran mitgearbeitet:

- Axel Böttcher (Hochschule München)
- Marcus Deininger (Hochschule für Technik Stuttgart)
- Ulrike Jaeger (Hochschule Heilbronn)
- Dieter Landes (Hochschule Coburg)
- Horst Lichter (RWTH Aachen, lokale Organisation)
- Barbara Paech (Universität Heidelberg)
- Lutz Prechelt (Freie Universität Berlin)
- Kurt Schneider (Leibniz Universität Hannover)
- Andreas Spillner (Hochschule Bremen, Vorsitz)
- Andreas Zeller (Universität des Saarlandes)

Ana-Maria Dragomir, RWTH Aachen, hat vielfältige Aufgaben in der Vorbereitung der SEUH übernommen und dazu beigetragen, dass das Konferenzsystem EasyChair verwendet werden konnte. Ferner hat sie aus den akzeptierten Beiträgen den Tagungsband zum Workshop erstellt. Vielen Dank dafür!

Schlussendlich möchten wir den Sponsoren ANECON Software Design und Beratung G.m.b.H. und iSQI International Software Quality Institute GmbH für die großzügige Unterstützung der 13. SEUH danken.

Wir freuen uns auf das gemeinsame Treffen,
spannende Vorträge
und anregende Diskussionen.

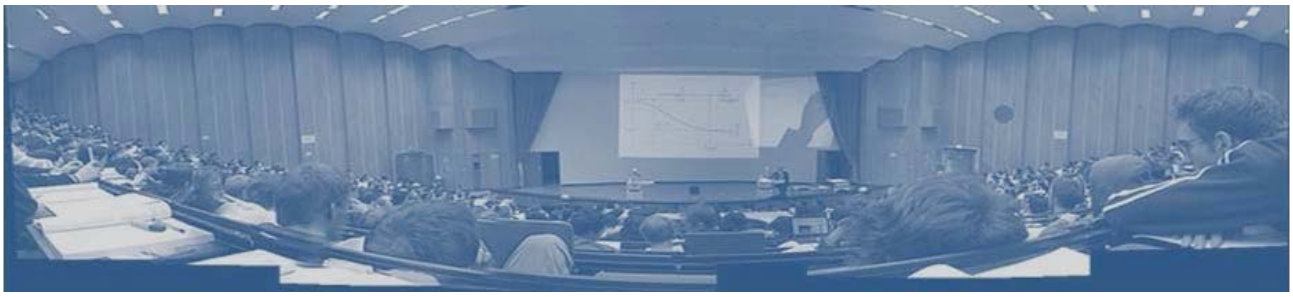
Bremen und Aachen, Januar 2013

Andreas Spillner
Hochschule Bremen

Horst Lichter
RWTH Aachen

Inhaltsverzeichnis

Session 5: Werkzeuge in und für die Ausbildung



Eingeladene Vorträge

Neue Wege in Software Projektkursen

Bernd Brügge, Technische Universität München

bruegge@in.tum.de

Zusammenfassung

Projektkurse mit realen Kunden aus der Industrie sind ein wichtiger Bestandteil vieler Software Engineering Vorlesungen, da sie es ermöglichen, relevante Konzepte und Praktiken gründlich zu vermitteln.

In meinem Vortrag zeige ich, wie man solche Kurse mit bis zu 100 Bachelorstudenten innerhalb eines Semesters durchführen kann. Die Studenten arbeiten in Teams mit mehreren Kunden aus der Industrie, bekommen reale Probleme, die mit agilen Methoden innerhalb eines Semesters zu lösen sind, wobei alle Phasen des Softwarelebenszyklus von der Anforderungsanalyse bis zur Lieferung und Demonstration des Systems durchlaufen werden.

Ein wesentlicher Bestandteil meiner Kurse ist die durchgehende Verwendung von Video beim Anforderungsmanagement, szenariobasierten Entwurf, team-aufbauenden Maßnahmen, Projektreviews, Präsentationen und bei Demos. Anhand von aktuellen Beispielen aus durchgeführten Projektkursen zeige ich den Einsatz von Video, illustriere die Herausforderungen beim Finden von Kunden und Themen, und stelle eine Infrastruktur vor, welche die Organisation und Durchführung solcher Kurse mit akzeptablen Zeit- und Kostenaufwand ermöglicht.

Software Engineering Curriculum @ Siemens

Peter Zimmerer, Siemens AG, München, Corporate Technology

peter.zimmerer@siemens.com

Zusammenfassung

Dieser Vortrag gibt einen Einblick in ein praxisorientiertes Software-Engineering Curriculum bei Siemens und schildert die Erfahrungen, die wir seit der Einführung im Jahre 2006 damit gemacht haben.

Mit dem Curriculum verfolgt Siemens zwei wesentliche Ziele: Zum einen soll die Qualifikation erfahrener Softwarearchitekten gestärkt und weiter verbessert werden, zum anderen soll durch ein Pre-Assessment und eine Zertifizierung sichergestellt werden, dass diese über die technischen und sozialen Fähigkeiten verfügen, die Softwarearchitekten benötigen, um große, komplexe, innovative, geschäftskritische Software-intensive Systeme nach-haltig zu entwickeln. Dazu kombinieren wir in einem holistischen Ansatz verschiedene Qualifizierungsmaßnahmen zu unterschiedlichen Themengebieten mit praktischen Übungen und konkreter Anwendung innerhalb der Projekte der Teilnehmer. Das Curriculum umfasst mehrere Workshops und Projektarbeitsphasen, die sich über einen Zeitraum von mehreren Monaten erstrecken und wird durch eine Reihe von Assessments der technischen und sozialen Fähigkeiten begleitet.

Inhaltsübersicht

- Hintergrund und Motivation
- Curriculum für Senior Software Architekten (SSWA)
- Curriculum für Software Architekten (SWA)
- Erfahrungen und Ergebnisse
  - o Herausforderungen in Ausbildung und Training
  - o Praxistransfer
  - o Zertifizierung
- Siemens Guiding Principles
- Erfolgsfaktoren
- Zusammenfassung und Ausblick

Literatur

Paulisch, F., Stal, M., Zimmerer, P. (2009): Software-Curriculum: Architektenausbildung bei Siemens. SIGS DATACOM OBJEKTspektrum, Heft Nr. 4, Juli/August 2009.

Paulisch, F., Zimmerer, P. (2010): A role-based qualification and certification program for software architects: An experience report from Siemens. ACM/IEEE 32nd International Conference on Software Engineering (ICSE), Cape Town, South Africa, May 2-8, 2010.

Just-in-Time Teaching

Just-in-Time Teaching für Software-Engineering

Timo Kamph, Peter Salden, Sibylle Schupp und Christian Kautz

Technische Universität Hamburg-Harburg

{timo.kamph,peter.salden,schupp,kautz}@tuhh.de

Zusammenfassung

Ein verbreitetes Problem beim Unterrichten von Software-Engineering ist, dass den Studierenden die Erfahrung mit großen, realen Softwaresystemen fehlt. Dadurch wird die Sinnhaftigkeit der vorgestellten Methoden nicht ausreichend erkannt. Wir versuchen, die fehlende Erfahrung zu simulieren und haben dazu die Methode des Just-in-Time Teaching (JiTT), in Form von Quiz, eingeführt. Die Studierenden, die an den Quiz teilgenommen haben, konnten die Methoden des Software-Engineerings am Ende des Semesters besser einordnen. Die Einführung dieser JiTT-Variante ist mit einem nicht zu unterschätzenden Aufwand verbunden, der sich aus unserer und der Sicht der Studierenden aber lohnt.

Einleitung

An der Technischen Universität Hamburg-Harburg (TUHH) ist die Veranstaltung Software-Engineering eine Pflichtvorlesung in zwei Bachelorprogrammen und eine Wahlpflichtveranstaltung in vier weiteren Studienprogrammen. Die Vorlesung wird von ca. 70 Studierenden im zweiten oder vierten Semester besucht. Die Vorkenntnisse und Erfahrungen der meisten Studierenden beschränken sich im Bereich Softwareentwicklung auf das Bearbeiten kleiner Übungsaufgaben in zwei oder drei einführenden Programmierkursen (Prozedurale Programmierung, Funktionale Programmierung, Objektorientierte Programmierung). Den Studierenden fehlt deshalb der Einblick in große Softwareprojekte und die damit einhergehenden Herausforderungen, z.B. dass viele Entwickler an einem Projekt beteiligt sind, keine Person alle Einzelheiten des Gesamtsystems im Detail kennt und das System auch nach Jahren noch zu warten sein muss. Ohne diesen Einblick fällt es vielen Studierenden schwer, den Sinn der in der Vorlesung behandelten Konzepte und Methoden zu verstehen.

Ein Ansatz, den Studierenden die fehlende Erfahrung zu vermitteln, ist, die Vorlesung um ein begleitendes Softwareprojekt zu ergänzen. Dies erfordert allerdings die Änderung des Curriculums für die betroffenen Studiengänge und war studiengangstechnisch nicht möglich. Wir haben stattdessen versucht, die fehlende Erfahrung zu simulieren, indem wir den Studierenden vor jeder Vorlesung Aufgaben gestellt haben, die die Notwendigkeit des behandelten Themas verdeutlichen und hierzu reale (Open-Source) Softwareprojekte verwenden. Die Methode, den Studierenden bereits vor der Vorlesung den Einstieg in das Thema im Selbststudium zu ermöglichen, wird häufig als „Just-in-Time Teaching“ bezeichnet.

In diesem Artikel beschreiben wir unseren Einsatz von JiTT im Bereich der Veranstaltung Software-Engineering, ein Konzept, das in Zusammenarbeit des Instituts für Softwaresysteme mit dem hochschul- und fachdidaktischen „Zentrum für Lehre und Lernen“ an der TUHH entwickelt wurde. Der Artikel ist wie folgt gegliedert: Abschnitt 2 führt JiTT allgemein ein und erläutert den Hintergrund dieser Lehrmethode. Abschnitt 3 beschreibt unsere konkrete Umsetzung im Rahmen der Software-Engineering Veranstaltung. Die dabei gemachten Erfahrungen werden in Abschnitt 4 besprochen, bevor der Artikel mit unseren Schlussfolgerungen in Abschnitt 5 endet.

„Just-in-Time Teaching“

Just-in-Time Teaching ist eine Lehr-Lern-Strategie, deren Kern eine spezielle Kombination von Präsenzlehre und online-gestütztem Selbststudium ist (grundlegend zu JiTT: Novak u.a. 1999).

Das besondere Merkmal von JiTT ist es, dass den Studierenden stets noch vor Beginn einer Lehrveranstaltung Studienmaterial online zur Verfügung gestellt wird. Dies können Einstiegsfragen in ein neues Themengebiet sein, die bei der Einschätzung helfen, welche Vorstellungen und welches Wissen die Studierenden in diesem Bereich bereits haben. Es können aber auch anspruchsvollere Aufgaben sein, die bereits ein kor-

rektes Grundverständnis erfordern, aber weitergehende Schwierigkeiten aufzeigen. Ergänzend können für die Studierenden offene Fragen angefügt werden, in denen sie Unklarheiten oder Probleme formulieren, die sich bei der Bearbeitung ergeben haben.

Die Aufgaben – deren Umfang gering gehalten wird – bearbeiten die Studierenden vor Beginn der zugehörigen Präsenzveranstaltung. Ihre Antworten übermitteln sie elektronisch an die Lehrperson, idealerweise mittels einer Technik, die die studentischen Lösungen automatisch auswertet bzw. strukturiert. Lehrende haben dann die Möglichkeit, genau rechtzeitig („just-in-time“) einen Überblick über den Wissensstand ihrer Studierenden zu gewinnen und z.B. missverständliche intuitive Auffassungen eines Stoffs (Präkonzepte) zu erkennen. Sie können entsprechend in ihrer Veranstaltung bereits verstandenen Stoff verkürzt behandeln, dafür aber gezielt auf vorliegende Verständnisschwierigkeiten eingehen.

Wie genau dies geschieht, liegt frei in der Hand der Lehrenden. So können sie die aus den Einstiegsaufgaben gewonnenen Informationen lediglich zur Präzisierung ihrer eigenen Vortragsteile nutzen und z.B. kurzfristig in ihre Präsentationen einzelne Folien einbauen, die typische oder besonders interessante studentische Lösungen oder Probleme zeigen. Alternativ können die Informationen aus den Einstiegsaufgaben aber auch zum „roten Faden“ der zugehörigen Veranstaltung gemacht werden, indem beispielsweise gezielt an auffälligen Präkonzepten gearbeitet wird, strittige Fragen per Partnerdiskussion o.ä. in der Vorlesung weiter vertieft oder über Classroom-response-Systeme („Clicker“) nochmals für alle Studierenden zur Disposition gestellt werden (Luo 2008, 166).

Entscheidend ist in jedem Fall, dass die Lehrenden frühzeitig eine Rückmeldung zu den Lernbedürfnissen ihrer Studierenden erhalten bzw. die Studierenden eine Rückmeldung zu ihrem Lernstand. Auf diesem Weg soll die Stoffvermittlung effizienter werden, Interesse und Aktivität der Studierenden sollen steigen.

Didaktischer Hintergrund

Betrachtet man JiTT in einem allgemeindidaktischen Kontext, so liegt sein Charme jenseits des Beitrags zu einer effizienten Lehrgestaltung zunächst in der Aktivierung der Studierenden zum selbstständigen Lernen – denn dass die studentische Eigenaktivität möglichst früh im Lehr-Lernprozess geweckt werden sollte, wird inzwischen gemeinhin als Voraussetzung für erfolgreiches Lernen aufgefasst (Novak u.a. 1999, 9).

Indem JiTT die Studierenden den Einstieg in ein Thema selbst vollziehen lässt, nimmt es dabei auch die grundlegende Idee des didaktischen Konstruktivismus auf, dass jedes Individuum seine Wirklichkeit selbst erzeugt (Siebert 1997, 17). Für das Lernen bedeutet dies: Ein bestimmtes Verständnis lässt sich nicht standardisiert vermitteln, sondern Lernende müssen ihren eigenen Zugang zu einem Thema finden und zusätzliche Informationen in ihr eigenes Wissenskonstrukt (wiederum individuell) integrieren.

An den Hochschulen hatte dieser Ansatz die Proklamation eines „shift from teaching to learning“ zur Folge. Sein Ausdruck ist im Zuge des Bologna-Prozesses u.a. die Berechnung des studentischen Arbeitsaufwands (workload), die für einzelne Veranstaltungen unabhängig von der Präsenzzeit berechnet wird. Empirische Studien zu den neu konzipierten Studiengängen zeigen inzwischen, dass Selbststudium allerdings nur dann funktioniert, wenn es in ein didaktisches Gesamtkonzept eingebunden und in den Präsenzveranstaltungen wieder aufgenommen wird (Metzger 2011, 271 f.). An dieser Stelle setzt JiTT an, indem es die Ergebnisse des Selbststudiums zum Ausgangspunkt der Präsenzlehre macht.

Möglich ist es dabei auch, das Selbststudium durch die Einbindung in das Prüfungskonzept einer Veranstaltung zusätzlich anzureizen. So kann durch JiTT der Forderung nach „formativem“, d.h. das Lernen begleitendem Prüfen nachgekommen werden (Dubs 2006, 2 f.). Die Bearbeitung der Aufgaben kann anteilig auf die Note angerechnet werden, oder bei Bestehen der Abschlussprüfung wird ein Bonus für die Aufgabenbearbeitung gewährt, der die Abschlussnote verbessert. Häufig wird es in diesem Fall von den Lehrenden als ausreichend betrachtet, wenn die Aufgaben überhaupt bearbeitet werden, unabhängig davon, ob die Lösungen richtig sind (Luo 2008, 167; Slunt/Giancarlo 2004, 986).

Verbreitung und Wirksamkeit von JiTT

„Just-in-Time Teaching“ wurde von Physik-Lehrenden der Indiana University-Purdue University at Indianapolis (IUPUI), der United States Air Force Academy (USAFA) und des Davidson College entwickelt (Novak u.a. 1999, 7). Inzwischen hat es sich sowohl über fachliche als auch über geographische Grenzen hinweg weit verbreitet (Poth 2009, 7). Auch in Deutschland ist Just-in-Time Teaching angekommen, allerdings mit noch recht geringer Verbreitung (zu nennen sind insbesondere die TU Hamburg-Harburg und die niedersächsische Ostfalia-Hochschule, siehe aber auch Hagel et al. in diesem Tagungsband). Eingesetzt wird es in mittleren und auch großen Veranstaltungen, so z.B. an der TU Hamburg-Harburg in Vorlesungen mit über 600 Studierenden.

Der vielversprechende Ansatz von JiTT hat auch das Interesse an der Frage geweckt, inwieweit sich ein Erfolg der Methode wissenschaftlich belegen lässt. Hierzu liegt inzwischen eine ganze Reihe kürzerer Studien vor, die überwiegend von Lehrenden mit Datenmaterial aus ihrem eigenen Unterricht durchgeführt worden sind (das in Amerika verbreitete „scholarship of teaching and learning“, vgl. Huber 2011, 118).

Luo (2008) beispielsweise weist nach, dass die regelmäßige Bearbeitung von JiTT-Aufgaben einen deutlich positiven Einfluss auf die Abschlussnoten seiner Studierenden hatte. Slunt und Giancarlo (2004) wiederum vergleichen JiTT mit dem Erfolg sog. „Concept Checks“, d.h. strukturierter Feedbackfragen im laufenden Unterricht, die Vorwissen oder Verständnis von Studierenden prüfen. Gemessen anhand von durchschnittlichen Endnoten fällt das Ergebnis deutlich zugunsten von JiTT aus (Slunt/Giancarlo 2004, 986 f.).

Diese Nachweise der Effektivität von JiTT sind nicht durchgängig auf Zustimmung getroffen. So zweifelt etwa Poth (2009, 2) an der statistischen Belastbarkeit vieler der vorliegenden Studien und kommt in seiner eigenen Analyse von JiTT zu durchaus skeptischen Einschätzungen. Auch er verwirft die Methode aber nicht, sondern fordert genauere Studien, unter welchen Bedingungen sie funktionieren kann (Poth 2009, 230 f.; vgl. Kautz 2006). Dies scheint auch insofern wünschenswert, als dass JiTT bei Evaluationen von den Studierenden regelmäßig als sehr gut und hilfreich bewertet wird.

Im Folgenden soll nun erläutert werden, welche Erfahrungen bei der Anwendung von JiTT im Rahmen der Veranstaltung „Software-Engineering“ an der Technischen Universität Hamburg-Harburg gemacht worden sind.

JiTT für Software-Engineering

Bei den bisherigen Umsetzungen von JiTT in unterschiedlichen Fächern ging es vorrangig darum, den Studierenden Grundwissen vor Beginn der Lehrveranstaltung zu vermitteln und den Lehrenden einen Überblick über den Wissenstand der Studierenden zu verschaffen. Bei unserem Einsatz von JiTT geht es hingegen in erster Linie darum, die Studierenden Erfahrungen mit der Komplexität von realen Softwareprojekten machen zu lassen. Diese Erfahrungen können nur gemacht werden, wenn die Studierenden auch an realen Softwaresystemen arbeiten. Wir haben uns daher entschieden, den Studierenden, im Rahmen eines Online-Quiz, Fragen zu Open-Source-Software zu stellen, deren Beantwortung sie an das Thema der nächsten Vorlesung heranführt und nebenbei den Umfang des betrachteten Systems aufzeigt. Der Einsatz von JiTT besteht aus den folgenden Komponenten: einem wöchentlich wechselnden Softwaresystem, einem zweiteiligen Quiz und einer kurzen Einheit in der folgenden Vorlesung. Die in den jeweiligen Vorlesungen behandelten Themen und die dafür verwendeten Softwaresysteme sind in Tabelle 1 angegeben. Im Folgenden soll nun auf die Quiz im Detail eingegangen werden.

Vorlesungsthema	Softwaresystem
Software Prozesse	Firefox (Bugtracker)
Nichtfunktionale Anforderungen	LibreOffice (Release Notes)
Spezifikation	Aristotle Analysis System
Spezifikation nebenläufiger Systeme	Eclipse (Deadlock)
Design	STL Iteratoren
Architekturen	Diverse Systeme
Objektorientiertes Design	ADT Queue
Testen	Mozilla Test Case Writing Primer
Testüberdeckung	GCC libstdc++ Test Suite
Wartung	Diverse Programme

Tabelle 1: Vorlesungsthemen und verwendete Softwaresysteme

Quiz

Jedes Quiz besteht aus zwei Teilen: der Einleitung, die die ganze Zeit verfügbar ist, und dem Fragenteil, für den es eine begrenze Bearbeitungszeit gibt. In der Einleitung wird das betrachtete System kurz vorgestellt und es wird beschrieben, wo man das benötigte Material findet. Dabei kann es sich um einen bestimmten Eintrag im Bugtracker des Projekts handeln oder um ein Archiv mit dem Quelltext der Software. Darüber hinaus erhalten die Studierenden Hinweise, welche Stelle des Systems relevant ist. So ist sichergestellt, dass den Teilnehmenden beim Fragenteil, der in begrenzter Zeit zu bearbeiten ist, das benötigte Material zur Verfügung steht und sie sich schon etwas mit dem System vertraut machen können. Darüber hinaus werden die Studierenden motiviert, sich Gedanken zu den möglicherweise im Fragenteil gestellten Fragen zu machen und

sich ein wenig im System „umzusehen“.

Das Ziel ist, den Studierenden die Komplexität der Entwicklung realer Softwaresysteme zu verdeutlichen und sie anzuregen, sich eigene Gedanken zum Thema der nächsten Vorlesung zu machen. Dabei kommt es nicht darauf an, die gestellte Frage exakt zu beantworten. Oftmals ist das auch gar nicht möglich, da die Fragen bewusst so gestellt werden, dass es mehrere rationale Antworten gibt, die nur entsprechend begründet werden müssen. Eine solche Frage ist, zum Beispiel, ob bei Kenntnis eines bestimmten Fehlers im Softwaresystem die Veröffentlichung verschoben werden soll. Das Lernziel wird also schon durch das Bearbeiten der Aufgabe erreicht.

Die Aufgaben sind so gestellt, dass das Bearbeiten etwa 30 Minuten dauern sollte. Durch die zeitlich begrenzte Freischaltung der Bearbeitung wird verhindert, dass die Studierenden einen wesentlich höheren Aufwand betreiben.

Durchführung

Es ist wichtig, dass die Studierenden den Zeitpunkt der Bearbeitung flexibel wählen können, um Konflikte mit anderen Veranstaltungen zu vermeiden. Da die Aufgaben von jedem Studierenden individuell zu bearbeiten sind, haben wir als Rahmen nur vorgegeben, dass das Quiz nach dem Ende der vorherigen Vorlesung veröffentlicht wird und bis zum Tag vor der nächsten Vorlesung bearbeitet sein muss. Das lässt dem Dozenten bzw. der Dozentin dann noch Zeit, die Antworten auszuwerten und in der Vorlesung entsprechend darauf zu reagieren.

Im Einzelnen sehen die Schritte wie folgt aus: Am Nachmittag nach der Vorlesung wird der erste Teil, die Einleitung, ins Netz gestellt. Sobald die Teilnehmenden ihre Vorbereitungen abgeschlossen haben und denken, dass sie sich im betroffenen Teilsystem ausreichend gut auskennen, starten sie mit dem zweiten, dem Fragenteil. In dem Moment, in dem sie auf den Start-Knopf klicken, werden ihnen die Fragen gezeigt und die Bearbeitungszeit beginnt zu laufen. Wir haben uns entschlossen, die Bearbeitungszeit auf 60 Minuten zu begrenzen, um zu verhindern, dass zu viel Zeit in die Beantwortung investiert wird. Die Fragen selbst sollten allerdings in unter 30 Minuten zu beantworten sein und wurden von den Studierenden in der Regel auch in dieser Zeit beantwortet.

Um die Studierenden zur Teilnahme anzuregen, haben wir die Quiz in ein Bonussystem integriert, das auch die (erfolgreiche) Teilnahme an den Übungen belohnt. Die durch alle Quiz insgesamt zu erreichenden Bonuspunkte entsprechen allerdings nur 2% der in der Klausur zu erreichenden Punkte. Der Anreiz ist also eher symbolischer Natur, hat aber trotzdem im Durchschnitt mehr als 70% der Vorlesungsteilnehmerinnen und Vorlesungsteilnehmer zum Mitmachen motiviert.

Wir halten die Bewertung einfach: Entweder die Antwort ist ausreichend für einen Teilnahmepunkt oder nicht. Entscheidend für die Bewertung ist hauptsächlich, ob die Antwort den Schluss zulässt, dass die Teilnehmerin oder der Teilnehmer sich mit der Fragestellung beschäftigt hat.

Nachdem die Studierenden die Fragen beantwortet haben, ist der letzte Schritt, dass die Fragen und insbesondere die Antworten in der nächsten Vorlesung aufgegriffen werden. Hierfür hat es sich als sinnvoll erwiesen, bei der internen Auswertung die Antworten in verschiedene Kategorien einzuteilen, z.B. „dafür“ und „dagegen“ oder „das ist immer ein Problem“, „das ist nur unter Voraussetzung A ein Problem“, „das ist nur unter Voraussetzung B ein Problem“ und „das ist nie ein Problem“. Für viele Studierende ist es interessant zu erfahren, wie viele ihrer Kommilitoninnen und Kommilitonen ihre Überlegungen teilen. Aber auch die Lehrenden können daraus ablesen, ob ein bestimmter Aspekt häufig genannt wird und ein anderer selten. Dann kann der selten genannte Aspekt in der Vorlesung nochmals besonders betont werden, falls er wichtig ist. Darüber hinaus werden aus jeder Kategorie besonders gute oder interessante, repräsentative Antworten gezeigt. Die Veröffentlichung erfolgt anonym, da das öffentliche Nennen des Namens von den betroffenen Studierenden oftmals als peinlich empfunden wird. Die Studierenden erkennen allerdings ihre eigene Antwort und erhalten dadurch eine extra Motivation.

Online-Plattform

Zur Durchführung der Quiz verwenden wir die E-Learning Plattform ILIAS (https://www.ilias.de). Die Quiz werden dort als „Test“ eingestellt. Mit ILIAS ist es möglich, die maximale Bearbeitungszeit und den spätesten Abgabetermin automatisch zu setzen. Insbesondere die Bearbeitungszeit lässt sich nur mit einem Online-System genau ermitteln. Die Fragen in den Quiz sind überwiegend Freitextfragen, manchmal kombiniert mit einer Auswahlfrage, die das grobe Einteilen der Antworten in Kategorien automatisiert hat. Bei der oben schon erwähnten Frage nach einem Bug im Firefox-Webbrowser muss man sich beispielsweise zunächst in einer Auswahlfrage für oder gegen das Verschieben des Veröffentlichungstermins entscheiden, bevor man seine Wahl in einer Freitextfrage begründet. Da es bei Freitextfragen nicht möglich ist, die Antworten vom System automatisch bewerten zu lassen, muss die Bewertung vom Lehrenden beim Durchsehen der Antworten erfolgen. Das gleiche

Muster findet sich bei einer Frage zur Aufnahme eines neuen Testfalls in eine Testsuite (siehe Beispiel). In einem anderen Quiz soll ein Vorschlag für eine neue STL Iterator-Hierarchie begründet werden. Dieses Quiz besteht nur aus einer Freitextfrage zur möglichen Begründung.

Beispiel

Nachdem wir nun den allgemeinen Aufbau der Quiz beschrieben haben, wollen wir ein Quiz genauer vorstellen. Das Quiz bereitet die Vorlesung zum Thema Testüberdeckung vor und als Softwaresystem kommt die C++-Standardbibliothek (libstdc++) der Gnu Compiler Collection (GCC) zum Einsatz.

Die Einleitung zum Quiz stellt GCC und libstdc++ kurz vor und leitet die Studierenden dann zur Testsuite des „find"-Algorithmus der libstdc++. Die dazugehörige Frage stellt dann einen neuen Testfall vor und die Studierenden müssen sich in einer Auswahlfrage entscheiden, ob der neue Testfall in die Testsuite aufgenommen werden soll oder nicht. In einer zweiten Frage muss diese Entscheidung dann begründet werden. Um die Entscheidung zu treffen und zu begründen, ist es notwendig, sich zu überlegen, was die Kriterien für die Aufnahme eines neuen Testfalls sind, was von den vorhandenen Testfällen abgedeckt wird und was der neue Testfall testet. Somit machen sich die Studierenden zum einen Gedanken über die Kriterien für eine gute Testsuite und zum anderen erleben sie durch das Navigieren zur Testsuite und das Herausfinden, was genau getestet wird, wie groß und komplex die C++-Standardbibliothek ist.

Die im Durchschnitt zur Beantwortung der Fragen benötigte Zeit betrug bei diesem Quiz 16 Minuten und 20 Sekunden. Der Aufwand zur Vorbereitung kann vom System nicht erfasst werden, wir schätzen jedoch, dass er unter 30 Minuten lag.

Am Tag vor der Vorlesung ist das Quiz beendet und die Antworten müssen ausgewertet werden. Durch die Auswahlfrage erhält man schnell einen Überblick, wie viele Studierende den neuen Testfall aufnehmen wollen und wie viele nicht. In diesem Fall war das Ergebnis unentschieden. Zur Auswertung der Begründung in der Freitextfrage muss jede Antwort einzeln betrachtet werden. Da die Antworten in der Regel recht kurz ausfallen, werden hierfür circa ein bis zwei Minuten pro Antwort benötigt. Die meisten Begründungen, den Testfall aufzunehmen, lassen sich in eine Kategorie „der Testfall testet etwas, das bisher noch nicht getestet wurde" einteilen und die meisten Begründungen ihn abzulehnen in eine Kategorie „der Testfall erhöht die Testabdeckung nicht". Einige Antworten fallen in die Kategorien „jeder neue Testfall ist gut" und „der Test wird immer fehlschlagen". Bei der Kategorisierung der Antworten werden auch drei besonders gute oder interessante Antworten ausgewählt und zusammen mit dem Ergebnis der Auswertung in der Vorlesung präsentiert.

Evaluation

Die Bewertung des Einsatzes von JiTT in der Veranstaltung Software-Engineering erfolgt aus zwei Perspektiven: zuerst aus Sicht der Studierenden und dann aus Sicht der Lehrenden. Grundlage sind die Daten aus dem Sommersemester 2012.

Studentische Evaluation

Die Quiz wurden von den Studierenden gut angenommen. Von insgesamt 70 Teilnehmenden an der Veranstaltung haben durchschnittlich 51 an den Quiz teilgenommen. Das entspricht einer Quote von 72%. Allerdings hat die Beteiligung zum Ende des Vorlesungszeitraums abgenommen, nachdem die Studierenden die maximal erzielbaren Bonuspunkte erreicht hatten. Die durchschnittliche Bearbeitungszeit des Fragenteils lag bei 17:25 Minuten. Dazu kam noch die Zeit für die Vorbereitung, also das Lesen der Einleitung, das Herunterladen des Quelltextes und das erste Umsehen im behandelten System, die nicht vom Onlinesystem erfasst werden kann. Insgesamt erscheint der zusätzliche Zeitaufwand für die Quiz aber vertretbar. Erfreulich ist weiterhin, dass im Schnitt 90% der Teilnehmenden eine Antwort gegeben haben, die zeigt, dass sie sich ernsthaft mit dem Thema auseinandergesetzt haben.

Vom Zentrum für Lehre und Lernen der TUHH (ZLL) wurde am Ende des Semesters ein Bewertungsbogen an die Studierenden verteilt, um ihre Einschätzung des JiTT-Ansatzes zu erfragen. Dabei sollten bestimmte Aspekte mit einer Note zwischen 1 (voll zutreffend) und 5 (nicht zutreffend) bewertet werden. Alle vorgegebenen positiven Äußerungen zu den Quiz wurden im Mittel als „weitgehend zutreffend" bewertet. Erfreulich ist insbesondere die gute Bewertung der Aussagen, dass die Quiz die in der Vorlesung behandelten Konzepte mit realen Systemen verbinden und dass die Quiz geholfen haben, die Wichtigkeit des Vorlesungsinhalts zu verdeutlichen. Dies zeigt, dass das Ziel der Lehrinnovation weitgehend erreicht wurde. Das Ergebnis der studentischen Bewertung ist in Tabelle 2 dargestellt. Die Zeile unter der zu bewertenden Aussage gibt prozentual an wie oft die entsprechende Note von den Studierenden vergeben wurde.

Interne Evaluation

Betrachtet man die Einführung der Quiz aus Sicht

der Lehrenden, so fällt zunächst der große Aufwand zur Vorzubereitung der Quiz auf. Der größte Aufwand besteht darin, ein zum Inhalt der Vorlesung passendes Softwaresystem zu finden. Hat man das System gefunden, so ist noch eine Anleitung für die Studierenden zu schreiben, die sie zum betrachteten Teil des Systems führt. Zum Abschluss der Vorbereitung müssen noch Fragen zum System gestellt werden, die die Studierenden an das in der entsprechenden Vorlesung behandelte Thema heranführen. Dieser Aufwand fällt aber nur bei der Einführung an, da die gefundenen Systeme und Fragen in den folgenden Semestern wiederverwendet werden können.

Aussage					
1	2	3	4	5	k.A.
Der Aufwand für die Lehrinnovation lohnt sich.					
14%	38%	24%	14%	7%	3%
Die Innovation wurde erfolgreich umgesetzt.					
31%	34%	21%	7%	7%	0%
Insgesamt würde ich diesen Kurs weiterempfehlen.					
28%	45%	21%	3%	3%	0%
Die Quiz haben die in der Vorlesung präsentierten Konzepte mit realen Softwaresystemen in Verbindung gebracht.					
38%	48%	7%	7%	0%	0%
Die Quiz haben mir geholfen, die Bedeutung der Vorlesungsinhalte für reale Software-Engineering-Aufgaben zu verstehen.					
21%	45%	24%	3%	7%	0%
Die Auswahl der Softwaresysteme war angemessen.					
24%	55%	10%	7%	0%	0%
Die Aufgaben waren interessant und haben mich angeregt, mir das verfügbare Hintergrundmaterial anzusehen.					
21%	31%	31%	7%	10%	0%

Tabelle 2: Bewertung der Quiz durch die Studierenden

Nachdem die Quiz einmal vorbereitet wurden, müssen sie in die Online-Plattform eingepflegt werden. Gerade beim ersten Einsatz eines Online-Systems sollten der Aufwand und eventuelle technische Probleme nicht unterschätzt werden. Vor jeder Vorlesung müssen die Antworten der Studierenden ausgewertet werden. Durch das Verwenden von Freitextfragen kann die Auswertung nicht automatisiert werden. Bei unserer Veranstaltung (mit durchschnittlich 51 Teilnehmern) hat die Auswertung etwa zwei Stunden in Anspruch genommen. Das Einbinden der Antworten in die Vorlesung ist hingegen kein großer Mehraufwand.

Die Schwierigkeiten liegen hauptsächlich bei der Auswahl der Softwaresysteme und Fragen. Es ist nicht leicht, zu einem bestimmten Thema oder Problem ein Softwaresystem zu finden, das das Problem veranschaulicht und gleichzeitig nicht zu komplex ist, um die Studierenden nicht zu überfordern. Das spiegelt sich auch in der unterschiedlichen Qualität unserer Quiz wieder.

Positiv für die Lehrperson in der Vorlesung ist, dass die Studierenden ein reges Interesse an den Antworten zum Quiz haben. Gerade bei längeren Vorlesungen, wenn die Konzentration der Studierenden nachlässt, hat das Besprechen der Fragen und Antworten einen positiven Effekt auf die Aufmerksamkeit. Des Weiteren war es manchmal möglich, verbreitete, aber falsche Vorstellungen der Studierenden zu erkennen und entsprechend darauf einzugehen.

Schlussfolgerungen

Es war unser Ziel, die Studierenden Erfahrungen mit realen Softwareprojekten machen zu lassen, damit sie die Sinnhaftigkeit der in der Vorlesung vorgestellten Methoden besser verstehen. Dazu haben wir die Methode des JiTT, in Form von Quiz, eingesetzt. Die größte Herausforderung, bei unserem Einsatz von JiTT, ist das Finden von Softwaresystemen, an denen man die in der Vorlesung vorgestellten Konzepte verdeutlichen kann.

Eine systematische empirische Überprüfung, dass die Studierenden mit Hilfe der Methode gegenüber einer rein klassischen Vorlesung bessere Lernergebnisse erzielt haben, steht für den gegebenen Fall aus. Die Aussagen der Studierenden am Ende des Semesters, unsere eigenen Beobachtungen und der Leistungsvergleich mit früheren Jahrgängen legen für uns jedoch nahe, dass wir unser Ziel erreicht haben. Wir sind überzeugt, dass die Quiz das Verständnis nachhaltig verbessert haben und werden JiTT weiterhin in der beschriebenen Weise einsetzen.

Literatur

Dubs, R. (2006): Besser schriftlich prüfen. Prüfun-

gen valide und zuverlässig durchführen. In: Neues Handbuch Hochschullehre, Einzelbeitrag H 5.1.

Huber, L. (2011): Forschen über (eigenes) Lehren und studentisches Lernen – Scholarship of Teaching and Learning (SoTL). In: Das Hochschulwesen, vol. 59, Heft 4, S. 118-124.

Kautz, C. (2006): Aktives Lernen in Großen Vorlesungen: Einsatz von internet-gestützten Vortests und interaktiven Vorlesungsfragen. In: Schlattmann, J. (Hrsg.): Die Bedeutung der Ingenieurpädagogik. Hamburg, S. 1-8.

Luo, W. (2008): Just-in-Time-Teaching (JiTT) improves student's performance in classes – adaptation of JiTT in four geography courses. Journal of Geoscience Education, v. 56, n.2, p. 166-171.

Metzger, C. (2011): Studentisches Selbststudium. In: Schulmeister, R.; Metzger, C. (Hrsg.): Die Workload im Bachelor. Zeitbudget und Studierverhalten – eine empirische Studie. Münster u.a., S. 237-276.

Novak, G.; Patterson, E.; Gavrin, A.; Christian, W. (1999): Just-in-Time Teaching. Blending active learning with web technology. Upper Saddle River, NJ.

Poth, T. (2009): Adressatengerechtes Unterrichten mit dem Just-in-Time Teaching-Verfahren. Diss., Würzburg. http://kola.opus.hbz-nrw.de/volltexte/2009/416/pdf/DissertationPoth.pdf. Zugriff am 23.10.2012.

Riegler, P. (2012): Just in Time Teaching. Wer liest und wer lehrt an der Hochschule? In: Waldherr, F.; Walter, C. (Hrsg.): Wissen, können, verantwortlich handeln. Forum der Lehre 2012. Ansbach, S. 89-95.

Siebert, H. (1997): Didaktisches Handeln in der Erwachsenenbildung. Didaktik aus konstruktivistischer Sicht. 2. Auflage, Neuwied u.a.

Slunt, K., Giancarlo, L. (2004): Student-centered learning: a comparison of two different methods of instruction. Journal of Chemical Education, vol. 81, No. 7, p. 985-988.

Drei Feedback-Zyklen in der Software Engineering-Ausbildung durch erweitertes Just-in-Time-Teaching

Georg Hagel; Jürgen Mottok; Martina Müller-Amthor

{georg.hagel,martina.mueller-amthor}@fh-kempten.de, juergen.mottok@hs-regensburg.de

Zusammenfassung

Das Konzept des Just-in-Time-Teachings existiert seit Ende des 20. Jahrhunderts (Novak, 1998) und wird weltweit in vielen Bereichen erfolgreich praktiziert. Allerdings noch nicht im Bereich Software Engineering. Es basiert auf großen Selbststudiums-Anteilen für die Studierenden und zwei Feedback-Zyklen, durch die die Lehrenden zeitnah, eben „Just-in-Time" auf Probleme der Studierenden eingehen können. Wir setzten diese Lehr- und Lerntechnik erstmals in unseren Software Engineering-Kursen für Master- und Bachelorstudiengänge ein und schildern unsere Erfahrungen mit dieser Technik. Außerdem erweitern wir sie durch einen weiteren Feedback-Zyklus, den wir im Bereich Software Engineering für unabdingbar halten.

Einführung

Die klassische Vorlesung, in der die Lehrenden versuchen die Lehrinhalte durch Vorlesen eines Manuskripts den Studierenden zu vermitteln, gehört, zumindest in der Informatik-Ausbildung, glücklicherweise längst der Vergangenheit an. Aber auch beim heute üblichen Vortrag mit Powerpoint-Unterstützung stellen wir fest, dass mit der Übermittlung des Lernstoffes auf diese Weise nicht das gewünschte Maß des Lernens desselben stattfindet.

Den Lehrenden stehen mit den konstruktivistischen Methodenbaukästen „Methodenpool" (Reich, 2008) und der „Methodensammlung" (Macke, 2009) Ideenquellen zur Ausgestaltung eines aktivierenden Lernprozesses zur Verfügung. Erste positive Versuche im Einsatz dieser Methoden findet man beispielsweise in (Mottok, 2009, 2010, 2012 und Hagel, 2010).

Ein weiterer aktivierender Ansatz, der die klassische und auch die heute häufige Vorlesung quasi auf den Kopf stellt, ist das Just-in-Time-Teaching. Es wird seit Ende des vergangenen Jahrtausends in wechselnden Disziplinen mit Erfolg eingesetzt (Bailey et. al., 2005, Henderson et. al. 2006, Nowak et. al. 1998, 1999, Simkins, 2010). Allerdings finden sich die dort veröffentlichten Einsätze in den Disziplinen Physik, Biologie und Chemie. Im Bereich der Informatik findet sich nur ein Erfahrungsbericht für die Grundlagenausbildung (Bailey et. al., 2005) und einer für Theoretische Informatik (Fleischer, 2004), jedoch nicht für Vorlesungen im Bereich Software Engineering. Das Just-in-Time-Teaching wird in (Prince et. al.,2007) zu den induktiven Lehr- und Lernmethoden gezählt.

Durch die Veranstaltung „Forum der Lehre 2012" inspiriert, bei dem das Just-in-Time-Teaching (JiTT) vorgestellt wurde (Riegler 2012), haben wir uns vorgenommen, diese aktivierende Lehrmethode auch in unseren Software Engineering-Veranstaltungen zu erproben.

Nach einer Einführung in die aktivierende Lehre und das Just-in-Time-Teaching allgemein, beschreiben wir unsere ersten Erfahrungen mit dieser aktivierenden Lehrmethode. Da wir in unseren ersten Versuchen mit dem erhaltenden Feedback noch nicht zufrieden waren, erweiterten wir die Methode um einen weiteren Feedback-Zyklus. Wir beschreiben unsere Erfahrungen mit dem von uns erweiterten Just-in-Time-Teaching, durchgeführt in einer gesamten Lehrveranstaltung. Abschließend wird die andere Herangehensweise an die Vorbereitung der Veranstaltungen aufgezeigt und Vor- und Nachteile der Methode beschrieben.

Aktivierende Lehre

Aktivierende Lehre ist ein wichtiger Bestandteil, um nachhaltiges, kompetenzorientiertes Lernen zu fördern. Aktivierende Lehre und aktives Lernen hängen unmittelbar zusammen (Mottok, 2010). Lehrende geben Studierenden Raum, mitzudenken und mitzuarbeiten. Dieses Miteinander lässt sich sowohl innerhalb als auch außerhalb von Vorlesungen und Übungen gestalten. Dabei vollzieht sich ein Perspektivwechsel von der Inhaltsorientierung in der Lehre hin zur Orientierung an Lernergebnissen bzw. Kompetenzen der Studierenden.

Lernen entspringt aus dem Verrichten der eigentlichen Arbeit mit Feedback (Maier, 2000). Studierende lernen am besten in Kontexten, also situativ. Isoliert Gelerntes ist schwer erinnerbar und schnell entschwunden. Wenn Aktivität von Beginn an in den Lernprozess integriert wird, dann steigt die Wahrscheinlichkeit für gelingendes Lernen im Sinne einer Initiierung neuer Handlungsweisen (Stelzer-Rothe, 2008).

Entdeckung und Motivation spielen eine große Rolle bei aktivierender Lehre. Die Studierenden entdecken ihre eigenen Fähigkeiten und Möglichkeiten, Probleme zu lösen. Es entsteht Handlungskompetenz bei den Studierenden. Diese zeigt sich in der Fähigkeit, Lerninhalte sowohl in Routinesituationen als auch in neuen, nie zuvor erlebten und durchdachten Situationen zielorientiert zu erfassen (Meyer, 2009).

Auch die Lehrenden werden motiviert durch die mobilisierten studentischen Kreativitäts-, Leistungs- und Begeisterungspotenziale.

Der Vorteil aktivierender Lehre ist die aktive Wissenskonstruktion durch die eigene Erfahrung der Lernenden. Dies ist dem klassischen rezeptiven Lernen überlegen.

Aktivierendes Lernen als selbstorganisiertes Lernen soll die Studierenden zu einem höheren Grad an Selbstbestimmung und Selbstorganisation verhelfen. Damit wird aber auch die Übernahme höherer Verantwortung für die individuellen Lernprozesse impliziert (Faulstich, 2012) und die Grundlage für Prozesse des lebenslangen Lernens geschaffen.

Der im Folgenden vorgestellte didaktische Ansatz Just-in-Time-Teaching in Software Engineering trägt die dargestellten Aspekte aktivierender Lehre.

Just-in-Time-Teaching

Oben behaupteten wir, dass durch das Just-in-Time-Teaching die Lehrveranstaltung auf den Kopf gestellt wird. Wie kommen wir dazu? Beim Just-in-Time-Teaching wird die Aufgabe des Lesens der Lehrinhalte nicht mehr durch die Lehrenden vorgenommen (="Vorlesung"), sondern an die Studierenden übergeben. Die Erstvermittlung des Stoffes wird damit an die Studierenden übertragen.

Selbststudium und Aufgaben

Die Lehrenden geben den Studierenden zunächst einen oder mehrere Texte, die zu den Lernzielen der nächsten Lehrveranstaltung passen, zum Selbststudium. Die Aktivierung der Studierenden geschieht durch zusätzlich erstellte webbasierte Aufgaben, die von den Studierenden bis etwa einen Tag vor der Lehrveranstaltung online zu bearbeiten sind. Bei den Aufgaben handelt es sich um allgemeine Verständnisfragen, aber auch inhaltliche Fragen, so dass die Lehrenden merken, ob die Studierenden die Texte tatsächlich gelesen und die entsprechenden Fach-Kompetenzen erworben haben. Die Studierenden können sich die Zeit für die Erarbeitung des Stoffes und die Bearbeitung der Aufgaben frei einteilen. Sie können die Zeiten so wählen, dass sie den Stoff am besten aufnehmen können.

Erstes Feedback der Studierenden

Die Lehrenden erhalten durch die Beantwortung der Aufgaben ein Feedback über die Punkte des Stoffes, die die Studierenden verstanden haben, aber auch über die Themen, die den Studierenden noch Schwierigkeiten bereiten. Die Lehrenden schauen sich die Lösungen der Aufgaben kurz vor der Lehrveranstaltung an und können so „Just-in-Time" die Inhalte der Lehrveranstaltung dem Kenntnisstand der Studierenden anpassen. Damit lehren wir nicht mehr, was wir meinen, lehren zu müssen, sondern unsere Studierenden teilen uns mit, wo sie Lehre benötigen.

Lehrveranstaltung

In der Lehrveranstaltung gehen die Lehrenden auf die Fragen und Probleme der Studierenden ein. Dabei können Sie die Antworten der Studierenden verwenden, so dass diese sich direkt angesprochen und in ihren Problemen ernst genommen fühlen. Weitere Vorteile, Antworten der Studierenden direkt zu verwenden findet man in (Rieger, 2012).

Meist kommen Diskussionen über die Inhalte zustande. Dadurch werden die Studierenden aktiv in die Erarbeitung der Lösungen eingebunden. „Die Lehrveranstaltung dient so nicht mehr primär der Übermittlung des Stoffes, sondern dazu, Studierende bei der Bewältigung ihrer konkreten Schwierigkeiten mit dem Stoff zu unterstützen" (Rieger, 2012).

Zweites Feedback der Studierenden

In der Vorbereitung der Lehrveranstaltung bereiten die Lehrenden neben Antworten auf die Fragen der Studierenden zusätzlich „Just-in-Time" Aufgaben für die Lehrveranstaltung vor, die dort per Klicker oder automatischen Antwortsystemen von den Studierenden beantwortet werden. Hier ist es sinnvoll, ein automatisches Antwortsystem zu verwenden, um allen Studierenden eine individuelle, nicht durch andere Studierende beeinflusste Antwort zu ermöglichen. Dabei können nach einer ersten Abstimmrunde zunächst mittels Peer-Instruction (Mazur, 1997, Simon et. al., 2000) die Studierenden über ihre abgegebenen Antworten und die Gründe dafür diskutieren und sind damit in den Lehr-Lernprozess eingebunden. Nach der Diskussion kann eine erneute Abstimmung stattfinden, die meist wesentlich mehr korrekte Antworten auf-

weist. Durch dieses direkte Feedback erkennen die Lehrenden, ob die Studierenden die Konzepte ausreichend verinnerlicht haben. Falls nicht kann, wiederum Just-in-Time, darauf eingegangen werden. Der positive Effekt auf das Lernen durch den Einsatz von automatischen Antwortsystemen in den Lehrveranstaltungen kann beispielsweise in (Knight et al, 2005) nachgelesen werden.

Erste Erfahrungen mit dem Just-in-Time-Teaching

Wie berichtet durch das „Forum der Lehre“ inspiriert und im Rahmen des Projektes EVELIN zur systematischen Verbesserung des Lernens von Software Engineering (Abke et. al 2012) haben sich die Verantwortlichen für die Software Engineering-Ausbildung der Hochschulen in Kempten und Regensburg zum Ziel gesetzt, das Just-in-Time-Teaching in die Lehre einzuführen. Dazu wählten wir zunächst gewünschte Lernziele aus, die die Studierenden im Rahmen der Veranstaltung erreichen sollten:

Die Studierenden sollen in der Lage sein, die Strukturierte Testfallermittlung durch Grenzwertbetrachtung zu verstehen und anzuwenden.

Dazu erhielten die Studierenden zunächst die Aufgabe, die entsprechenden Kapitel aus den Büchern von Liggesmeyer (Liggesmeyer, 2002) und Spillner (Spillner, 2010) zu studieren. Außerdem wurden die in Abbildung 1 gezeigten allgemeinen Fragen zum Verständnis gestellt.

Die freiwillige Beantwortung der Fragen ergab erwartungsgemäß einen Rücklauf von weniger als 50% bei den Studierenden. Dabei tauchte eine Frage häufig auf:

- „Wie bestimme ich den Repräsentanten, kann ich mir den selber ausdenken oder gibt es da irgendwelche Vorschriften, wie ich auf den komme?“

Darauf konnten wir in der Lehrveranstaltung ausführlich eingehen. Auch die Frage:

- „Gibt es auch einen sog. White-Box-Test?“

leitete am Ende der Lehrveranstaltung sinnvoll auf das nächste Themengebiet über.

Außer den in Abbildung 1 gezeigten Fragen, wurde ein online zu bearbeitendes Beispiel zur strukturierten Testfallermittlung erstellt. In der Vorbereitung der Lehrveranstaltung war die Durchsicht der Antworten zu diesem Beispiel sehr aufwendig, da die verwendete Lehr- und Lernplattform Moodle keine Hilfsmittel anbot, um die Antworten automatisch zu beurteilen. Allerdings konnte durch die Sichtung der Antworten erkannt werden, wo die Studierenden in der Anwendung der strukturierten Testfallermittlung noch Schwierigkeiten hatten.

Die Lehrveranstaltung zur strukturierten Testfallermittlung lief dann vollkommen anders ab, als gewohnt: Zunächst wurde auf die Schwierigkeiten der Studierenden und deren Fragen genau eingegangen. Dann wurde ein weiteres Beispiel gemeinsam in der Lehrveranstaltung erarbeitet.

Das Feedback der Studierenden war in der Mehrzahl positiv. Auch in der Abschlussklausur wurde das Thema überwiegend sehr gut bearbeitet. Daher beschlossen die Lehrenden, das Just-in-Time-Teaching weiter auszubauen.

Was haben wir aus der Testphase gelernt:

- Die Beantwortung der Fragen muss motiviert werden.
- Die Vorbereitung auf die Lehrveranstaltung war sehr aufwendig.
- In der Lehrveranstaltung gab es ausgiebige Diskussionen zu dem dort behandelten Beispiel.

Abb. 1: Allgemeine Verständnisfragen

In Kempten haben wir weiterhin versucht, innerhalb der Vorlesung Softwarearchitektur für den Master-Studiengang „Angewandte Informatik“

einen „Reading-First"-Ansatz zu wählen. Die Studierenden bekamen einen aktuellen Artikel zur Softwarearchitektur über das Thema Messaging und idempotente Nachrichten, den sie bis zur nächsten Lehrveranstaltung lesen sollten (Helland, 2012). Ziel war es, dass die Studierenden das Thema idempotente Nachrichten und ihren Einsatz in der Softwarearchitektur verstehen sollten. In der Lehrveranstaltung haben wir diesen Text ausführlich besprochen und diskutiert. Dadurch, dass die Studierenden jedoch keine Fragen zur online-Beantwortung bekamen, hatte der Lehrende keinen Einblick, wie viel tatsächlich vom Thema verstanden wurde. Durch die lebhafte Diskussion entstand der Eindruck, dass das Thema beherrscht würde. Die Klausur am Ende des Semesters zeigte jedoch, dass lediglich 10% (!) der Studierende noch die Kernaussagen kannten. Daraus schließen wir, dass die online zu beantwortenden Fragen und die Just-in-Time Lehrveranstaltung im Just-in-Time-Teaching die zentrale Rolle spielen.

Eine ganze Lehrveranstaltung durch Just-in Time Teaching

Um das Just-in-Time-Teaching einmal in einer gesamten Lehrveranstaltung anzuwenden, wählte man in Kempten die Lehrveranstaltung „Requirements Engineering und –Management" im Master-Studiengang „Angewandte Informatik", die sich um ein Spezialgebiet des Software Engineerings dreht. Die Veranstaltung wurde durchgehend von 15 Studierenden besucht.

Die Lehrenden überlegten sich von Woche zu Woche, welcher Teil des Lehrbuches, dem Standardwerk von Chris Rupp und den SOPHISTen (Rupp, 2009), bearbeitet werden sollte. Dazu erhielten die Studierenden jede Woche die folgenden drei Standardfragen (Riegler, 2012) über die Lehr- und Lernplattform Moodle:

- „Was fanden Sie schwierig, oder haben Sie nicht verstanden? Es mag sein, dass Sie hier nichts angeben können. Dann antworten Sie bitte mit ‚nichts'."
- „Beschreiben Sie, was Sie am interessantesten oder gewinnbringend fanden."
- „Welche Anknüpfungspunkte sehen Sie zwischen diesem Stoff und dem, was Sie bereits wissen?"

Bei der Beantwortung der ersten Frage kam immer seltener „nichts", je weiter die Lehrveranstaltung fortschritt. Die Fragen der Studierenden waren oft sehr detailliert, so dass wir erkannten, dass sich die Studierenden mit dem Stoff intensiv auseinander gesetzt haben. Beim Kapitel über die Abgrenzung des zu erstellenden Systems kamen z.B. folgende Fragen:

- „Ich verstehe nicht, wie tief man den Systemkontext spezifizieren muss. Was gehört alles dazu?"

Darauf konnte in der folgenden Lehrveranstaltung eingegangen werden. Die dritte Frage (über die Anknüpfungspunkte) zeigte, ob die Studierenden die Sachverhalte des Textes in ihr bisheriges Studium, oder bisherige Erfahrungen einordnen konnten. Damit hingen die gelernten Themen nicht einfach in der Luft, sondern wurden mit schon Gelerntem verknüpft.

Zu den jedes Mal vorhandenen, oben vorgestellten, allgemeinen Fragen kamen spezielle Fragen in der Lehr- und Lernplattform Moodle hinzu, die einfache Sachverhalte abfragten, um zu erkennen, ob die Studierenden den Text überhaupt durchgearbeitet haben. Außerdem sollte das Gelesene an Hand eines Beispiels angewendet werden. Wir verwendeten ein fiktives Projekt „Bewerbermanagementportal für Hochschulen" für das Management von Bewerbern, die bei der Hochschule arbeiten wollen (Berufungsverfahren für Professoren, Einstellung von Labor- und wissenschaftlichen Mitarbeitern, Mitarbeitern in der Verwaltung und Hiwis). Bei der Beantwortung der Fragen ging es nicht nur um Fachwissen, sondern auch um praktische Anwendung des Gelernten. Insgesamt mussten bei jeder Fragerunde 10 Fragen beantwortet werden. Die Studierenden hatten meist 5 Tage Zeit, sich das Thema anzueignen und die Fragen zu beantworten. Einige Studierende bearbeiteten den Stoff gemeinsam und diskutierten über die Inhalte, andere wählten das Selbststudium alleine. Für die Bearbeitung der Aufgaben wurde grundsätzlich ein Stichtag festgesetzt, so dass sichergestellt war, dass alle Antworten kurz vor der Vorlesung vorlagen. Dies war in der Regel etwa 1 Tag vor der Vorlesung. Die Fragen wurden seitens der Studierenden zu einem sehr großen Prozentsatz beantwortet. Das lag daran, dass wir entgegen den letzten Versuchen mit der Methode einen Prozentsatz beantworteter Fragen als Prüfungsvorleistung erbringen ließen. Die Antworten waren meist 2-10 Zeilen lang. Für manche Fragen mussten auch Diagramme erstellt werden, die in Dateiform abgegeben wurden.

Erstes Feedback

Die Lehrenden sichteten die Antworten und erkannten an Hand der Antworten Schwächen der Studierenden. Dieser Teil des Just-in-Time-Teachings war stets sehr aufwändig (4-10 Stunden, manchmal auch mehr), da die Antworten größtenteils nicht automatisch ausgewertet werden konnten. Allerdings bereitete es viel Freude, die nächste Lehrveranstaltung „kundengerecht", also „studie-

rendengerecht“ zu gestalten. Hierbei handelte es sich um die erste Feedback-Schleife, in der die Lehrenden die Probleme der Studierenden mit dem Stoff erkennen konnten. Für die Gestaltung der nächsten Lehreinheit wurden teils Präsentationsfolien erstellt, die Sachverhalte klar stellen sollten, teils aber auch Folien mit anonymen Zitaten der Studierenden, die dann in der Vorlesung diskutiert wurden. Auch weitere Beispiele wurden hinzugefügt.

Zweites Feedback

Parallel dazu überlegten sich die Lehrenden Aufgaben für die Lehrveranstaltung, die mit Hilfe eines automatischen Antwortsystems (siehe Abb. 2) von den Studierenden beantwortet wurden. Der Einsatz dieses Mediums hat den Vorteil, dass die Studierenden, wie oben besprochen, individuell die Fragen beantworten können.

Außerdem hatten die Studierenden viel Spielfreude dabei, was zu einem lockeren Lernklima beitrug. Unser Angebot, die Fragen anonym zu beantworten, wurde bei den Masterstudierenden abgelehnt. Sie hatten Freude am Wettbewerb. Da sich die Studierenden geeinigt hatten, dass wir nach jedem Test die Prozentzahl der richtig beantworteten Fragen gemeinsam anschauen, war die Motivation hoch, sich bei der Beantwortung der Fragen anzustrengen. Die aktive Beteiligung der Studierenden war immens und es bedurfte keiner Maßnahme, die Beantwortung der Fragen zu benoten.

Abb. 2: Automatisches Antwortsystem (SMART)

In der Lehrveranstaltung, die somit Just-in-Time vorbereitet wurde, wurden zunächst die Probleme der Studierenden mit dem Stoff behandelt und danach die erneute Fragerunde mit den Klickern durchgeführt. Hierbei erhielten die Lehrenden das zweite Feedback über den Wissensstand der Studierenden. Danach konnten noch offene Fragen diskutiert, oder weitere Beispiele behandelt werden.

Mit diesen beiden Feedbacks waren die Feedback-Zyklen des reinen Just-in-Time-Teachings erschöpft.

Drittes Feedback

Die Lehrenden wollten darüber hinaus einen passenden Umgebungsrahmen zur Verbesserung weiterer Lernprozesse im Bereich der personalen, sozialen Kompetenzen sowie der Aktivitäts- und Handlungskompetenz schaffen (Heyse 2009). Dazu wurde eine Vertreterin einer realen Unternehmensorganisation als „Kundin“ eingeladen. Wichtige Persönlichkeitseigenschaften der Kundin waren selbstbewusstes und sicheres Auftreten, Offenheit und Bereitschaft zum Praxisdialog, Entscheidungs- und Schlagfertigkeit. Da in der Zusammenarbeit mit den Studierenden Feedback erwartet wurde, waren Kommunikations- und Kooperationsfähigkeit weitere Rahmenparameter. Die gemeinsame Bearbeitung des Projekts „Bewerbermanagementportal“ fand in der Übungsstunde statt. Die Studierenden mussten also zusätzlich zum Selbststudium und zur Just-in-Time-Teaching-Fragerunde keine weiteren Übungsaufgaben bearbeiten. Damit blieb der Aufwand für die Lehrveranstaltung für die Studierenden in einer vergleichbaren Größenordnung wie bei der klassischen Vorlesung mit Übungen.

Die Kundin hat den Studierenden in typischer Kundenpraxis die Anforderungen für das Bewerbermanagementsystem vorgetragen. Die Lehr-Lerngemeinschaft versuchte gemeinsam mit Hilfe der verschiedenen erlernten Requirements Engineering-Techniken, die Anforderungen des Kundensystems zu erheben. Dabei erkannten die Studierenden von Woche zu Woche, wie sie tiefer und tiefer ins Requirements Engineering eintauchten und die gefundenen, sowie dokumentierten Anforderungen immer genauer und aussagekräftiger wurden.

Für die Lehrenden ergab sich so sehr schön eine dritte Feedback-Schleife, da wir sahen, wie die Studierenden im Liveprojekt das Gelernte angewendet haben und Dialogfähigkeit sowie Kundenorientierung sichtbar wurde.

Alle drei Feedback-Zyklen gingen in die Vorbereitung der Lehrveranstaltung der kommenden Woche wieder ein. In dieser Reflexion des „Kundentermins“ fand sowohl der Austausch über kreative, beobachtende und befragende Techniken des

Requirements Engineering (Rupp 2009) statt, als auch eine für die Studierenden überraschende und rege Auseinandersetzung über Kommunikationsstil und Konfliktmanagement sowie über strategische Vertragsverhandlung. Eine Skizze der Schritte unseres Vorgehens haben wir in Abb. 3 dargestellt.

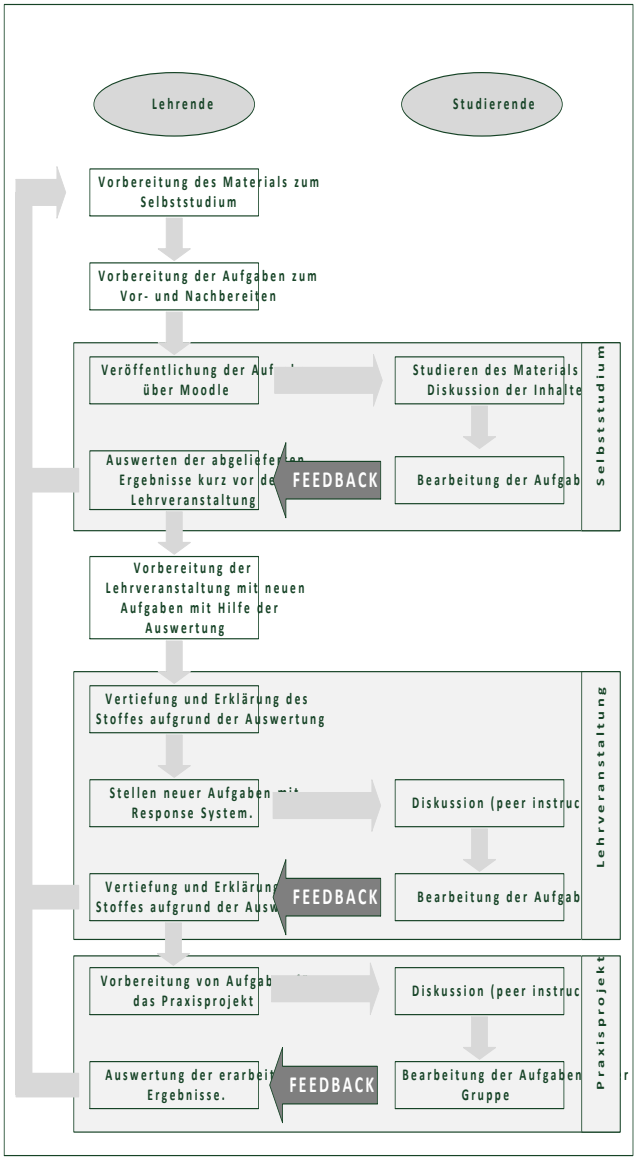


Abb. 3: JiTT mit dreifachem Feedback

Umdenken in der Vorbereitung

Für uns als Lehrende bedeutete diese Methode ein radikales Umdenken in der Vorbereitung der Lehrveranstaltungen (Hesse 2000). Während man früher seine Folien, oder andere Materialien schon frühzeitig fertigstellen konnte und in der direkten Vorbereitung kurz vor der Lehrveranstaltung nochmals darüber schauen musste, was in sehr kurzer Zeit erledigt ist, fordert diese Art der Veranstaltung mehr Flexibilität und einen hohen Aufwand in der Vorbereitung jeder Lehrveranstaltung, der nicht im Voraus erbracht werden kann (vgl. hierzu auch Fleischer, 2004). Durch die Interaktion mit den Studierenden und das direkte Anpassen der Lehrinhalte an deren Vorwissen kann eine Vorbereitung erst in letzter Minute, „Just-in-Time“ eben, erfolgen. Gerade bei Lehrveranstaltungen mit 50 Studierenden und mehr ist der Ansatz ohne eine automatische Auswertung der Fragen nicht realistisch „Just-in-Time“ durchführbar. Peter Riegler weist jedoch in (Riegler 2012) darauf hin, dass die Antworten bei vielen Studierenden auch mit der Unterstützung von Hilfslehrenden gesichtet werden können.

Wird die gleiche Lehrveranstaltung zum wiederholten Male gehalten, geht der Aufwand jedoch zurück: Die Fragen für die Studierenden für den Selbststudiumsanteil können beibehalten werden. Auch wird man mehr und mehr feststellen, dass die Studierenden häufig mit den gleichen Inhalten der Veranstaltung Probleme haben, so dass die Vorbereitung der Lehrveranstaltung zu einem erheblichen Anteil identisch ist. Die „Just-in-Time“-Komponente bleibt mit ihrem Aufwand jedoch erhalten.

Eine weitere Herausforderung für die Lehrenden sind die zu erstellenden Fragen für den online-Teil, aber auch für die Lehrveranstaltung. Es sollen ja nicht nur die Kernaussagen der Bücher abgefragt werden, sondern auch Konzepte verstanden und Methoden gelernt und angewandt werden.

Denkbar wäre auch eine echte Firmenpräsentation mit den erarbeiteten Ergebnissen zu planen und durchzuführen, in den Studierende als konkurrierende Unternehmensberatungen ihr Ergebniss des Requirements Engineerings darstellen. Eine solche Erweiterung sorgt bei den Stakeholdern einer Hochschule für angewandte Wissenschaften, die sich auch als Hochschule der Region versteht, für besondere Profilierung.

Fazit

Die Methode des Just-in-Time-Teachings hat uns aus einer Vielzahl von Gründen motiviert. Unsere Erfahrungen damit sind durchgängig als positiv zu bewerten. Erste Versuche mit einzelnen Lehreinheiten waren für die Einstimmung der Lehrenden auf diese Methode gut geeignet. Allerdings bemerkten wir, dass die Studierenden viel mehr bei der Sache sind, wenn die ganze Lehrveranstaltung nach diesem Prinzip abgehalten wird. Die Vorteile der Methode unter Berücksichtigung der dritten Feedback-Schleife haben wir nach unterschiedlichen Schwerpunkten angeordnet (teils übereinstimmend mit Fleischer, 2004). Unter dem Punkt „Studierende“ haben wir alle Vorteile für die Studierenden aufge-

führt, unter „Feedback“ alle Vorteile das Feedback betreffend, unter „Lehrveranstaltung die Vorteile für unsere Veranstaltungen, wenn wir diese Methode anwenden und unter „Sonstige“ all diejenigen Punkte, die nicht in die anderen Kategorien passen. Dabei sind die Grenzen zwischen den Punkten fließend:

Studierende

- Anstatt lediglich auf die Klausuren zu lernen, lesen die Studierenden regelmäßig im Lehrbuch.
- Aus der Heterogenität der studentischen Vorerfahrungen wird das im Selbststudium Gelesene unterschiedlich verwertet. In der gemeinsamen Erörterung entwickelt sich diese Vielfalt zu einem weiteren aktivierenden, sowie belebenden Element und unterstützt die Motivation der Mitarbeit.
- Die individuelle Beantwortung der Fragen der Studierenden durch die Lehrenden kommt einer individuellen Betreuung der Studierenden nahe. Die Studierenden erhalten jeweils die Information, die sie gerade benötigen.
- Die Studierenden erkennen, dass sie nicht alleine mit ihren Fragen und Problemen zum Lehrstoff sind.
- Die Studierenden besuchen gerne die Lehrveranstaltung, weil sie dort ihre persönlichen Fragen beantwortet bekommen.
- Durch die Erstvermittlung des Lehrstoffes durch die Studierenden bereiten diese sich auf den lebenslangen Lernprozess vor.
- Die Studierenden können die Zeit des Selbststudiums auf Zeiten legen, an denen sie am aufnahmefähigsten sind.
- Die Studierenden üben sich im Schreiben fachlicher Texte durch die schriftliche Beantwortung der Fragen im online-Teil.
- Teamarbeit, wie in Unternehmen wird in der Übung am realen Beispiel situativ geübt und steigert neben den bereits erwähnten Kompetenzen auch die Innovationskraft, da Menschen miteinander aus unterschiedlichen Perspektiven ein Thema erarbeiten.

Feedback

- Die Beantwortung der Fragen in der Selbststudiums-Komponente der Methode gibt den Studierenden Feedback, in wie weit sie den Lehrstoff verstanden haben.
- Die Antworten auf die online gestellten Fragen und auf die Fragen in der Lehreinheit geben direktes Feedback über den Lernfortschritt der Studierenden.
- Die dritte Feedback-Schleife durch die Anwendung der gelernten Techniken an einem größeren, realen Beispiel zeigt, ob die Studierenden die Techniken auch an einem größeren Beispiel anwenden können.

Lehrveranstaltung

- Die Fragerunden in der Lehreinheit motivieren die Studierenden sich auf den Lehrstoff vorzubereiten. Sie bieten auch die Möglichkeit älteres Wissen zu wiederholen.
- Die Studierenden diskutieren Fragestellungen mit, da sie sich auf die Lehreinheit vorbereitet haben.
- Die Studierenden werden in den Lehrprozess durch Peer Instruction eingebunden.
- Die Studierenden machen in der Lehrveranstaltung einen motivierten und interessierteren Eindruck.
- Die Studierenden sind gezwungen aus ihrer reinen Konsumentenhaltung in der Lehrveranstaltung herauszukommen und aktiv am Geschehen teilzunehmen.
- Die Atmosphäre im „Hörsaal“ ist fehlertolerant, d.h. Fragen sind durch die Methode erlaubt.

Sonstige

- Es werden Fehler oder Unstimmigkeiten im Lehrmaterial aufgedeckt.
- Ad hoc gestellte Fragen, die bei dieser Art der Veranstaltung häufiger zu erwarten sind, halten die Veranstaltung für den Lehrenden interessant.

Für Veranstaltungen, in denen unter 50 Studierende erwartet werden, kann das Just-in-Time-Teaching sehr gut durch einen Lehrenden durchgeführt werden. Bei mehr Studierenden wird die Unterstützung von Hilfslehrkräften benötigt. Stellen sich die Lehrenden auf die andere Herangehensweise der Vorbereitung auf eine Lehreinheit um, können sie mit motivierten, aktiven und wissensdurstigen Studierenden rechnen. Durch die Einführung der dritten Feedback-Schleife können ausführlichere Themen des Software Engineerings an größeren Beispielen direkt in der Übung in Form von Teamwork erarbeitet werden.

Ein Rücklauf von ca. 50% in der Beantwortung von Fragen in der der von JiTT genutzten Lernplattform Moodle kann erhöht werden wenn für die studentische Mitarbeit in Just-in-Time-Teaching Anreizsysteme geschaffen werden. So können beispielsweise 10 Bonuspunkte für die Klausur durch kontinuierliche Beiträge der Studierenden in JiTT von den Studierenden gesammelt werden. Hierfür

ist allerdings eine Änderung der Studienpläne nötig. Dies ist in Regensburg erstmals im WS 2012/2013 umgesetzt.

Zusammenfassend lässt sich sagen, dass qualitativ hochwertiges Lernen vor allem dann stattfindet, wenn sich die Lernenden unter sinnvoller Anleitung die Dinge selbst aneignen, selbst Frage- und Problemstellungen entwickeln, diese als relevant erkennen und selbst nach Möglichkeiten und Alternativen der Lösung suchen (Wörner, 2008).

Ausblick

Um die Zeit der Bearbeitung der ersten Feedback-Runde zu verkürzen und diese Methode auch für Lehrveranstaltungen mit mehr als 50 Studierenden allein durchführbar zu machen, bieten sich Systeme für die online-Aufgaben an, die Fragen automatisch auswerten können. Zu diesen gehört LON-CAPA (Kortemeyer et. al., 2008, LON-CAPA), hinter dem ein Computeralgebra System verborgen ist. Damit eignet es sich hervorragend für alle mathematischen und physikalischen Aufgaben. Diese können von dem System sogar automatisch randomisiert werden, so dass die Studierenden je unterschiedliche Fragen bekommen können. Dies ist ein wesentlicher Grund, warum das Just-in-Time-Teaching in Mathematik- und Physik-Lehrveranstaltungen schnell Fuß gefasst hat. Auf dem Gebiet der automatischen Auswertung von Lösungen müssen wir im Software Engineering noch einiges an Arbeit investieren. Existiert außerdem ein Pool an Fragen und Antworten, auf den die Lehrenden zugreifen können, verringert sich auch der Aufwand für die Erstellung der Fragen für die Selbststudiumsphase. Aber schon wenn ein paar Aufgabentypen automatisierbar wären, kann sich das Just-in-Time-Teaching auch in dieser Disziplin durchsetzen.

Das Design einer geeigneten Dokumentationsform für Feedback-Ergebnisse und passender Erklärungsinhalte könnte sich positiv sowohl auf Zeit- und Gestaltungsressourcen in der JITT-Vorbereitung als auch auf die Erfahrungskurve im Umgang auswirken.

Werden die erfassten Feedbacks auf einer Kollaborationsplattform anonym zur Verfügung gestellt, wäre es auch möglich die Studierenden in eine weitere Feedbackschleife zu schicken, in der über die Gesamtergebnisse der zu einer Lehrveranstaltungsstunde gesammelten Aussagen Stellung genommen werden kann.

Der Einsatz von strukturierten Dokumentationsformen zur Erfassung praxisrelevanter Erkenntnisse zum persönlichen Nachlesen und Reflektieren sollten ohne Medienbruch als Hilfe angeboten und jederzeit zugänglich sein. Eine Patternsammlung könnte angelegt und dynamisch erweitert werden (Schmolitzky 2011), welche sowohl für Erfassung, Beschreibung und Auswahl der jeweiligen Lehr-Lernmethode als auch die unterschiedlichen Feedback-Typen zur Verfügung steht (Müller-Amthor 2012).

Weiter zu untersuchen ist, ob diese Art der Lehrveranstaltung zu besseren Leistungen der Studierenden in Klausuren führt. In einer JiTT-basierten Lehrveranstaltung Software Engineering verändern die Studierenden ihr Verhalten von einer passiven hin zu einer aktiven Rolle indem sie im angebotenen Lernarrangement aktiv partizipieren..

Danksagungen

Die vorliegende Arbeit ist gefördert durch das vom BMBF finanzierte Verbundvorhaben "Experimentelle Verbesserung des Lernens von Software Engineering (EVELIN)", Förderkennzeichen 01PL12022{A,B,C,D,E,F}, Projektträger DLR. Das Verbundvorhaben EVELIN ist im Bund-Länder-Programm für bessere Studienbedingungen und mehr Qualität in der Lehre angesiedelt. Die Hochschulen Aschaffenburg, Coburg, Kempten, Landshut, Neu-Ulm und Regensburg sind Verbundpartner, weitere Informationen unter www.las3.de und www.qualitätspakt-lehre.de.

Wir danken auch den Reviewern unseres Artikels, für ihre konstruktiven Anmerkungen.

Literatur

Abke, J., Brune, P., Haupt, W., Hagel, G., Landes, D., Mottok, J. Niemetz, M., Pfeiffer, V., Studt, R., Schroll-Decker, I., Sedlmaier, Y. (2012): EVELIN – ein Forschungsprojekt zur systematischen Verbesserung des Lernens von Software Engineering. Erscheint im Tagungsband des Embedded Software Engineering Kongress' 2012.

Bailey, T., Forbes, J. (2005): Just-in-Time Teaching for CS0. SIGCSE 2005. St. Louis, USA.

Faulstich, P., Zeuner, C. (2010): Erwachsenenbildung, Beltz Verlag, Weinheim.

Fleischer, R. (2004): Just-in-time: Better Teaching in Hong Kong. Proceedings of the Second Teaching and Learning Symposium.

Häfele, H., Maier-Häfele, K., (2010): 101 e-Le@rning Seminarmethoden – Methoden und Strategien für die Online- und Blended-Learning-Seminarpraxis, managerSeminare Verlags GmbH, Bonn.

Hagel, G., Mottok, J., Utesch, M., Landes, D., Studt, R. (2010): Software Engineering Lernen für die berufliche Praxis - Erfahrungen mit dem konstruktivistischen Methodenbaukasten, im Tagungsband des Embedded Software Engineering Kongress' 2010.

Helland, P. (2012): Idempotence Is Not a Medical Condition, Communications of the ACM, vol. 5, no. 5, May 2012.

Henderson, C., Rosenthal, A. (2006): Reading Questions: Encouraging Students to Read the Text Before Coming to Class. Journal of College Science Teaching, July/August 2006.

Hesse, F., W., Mandl, H. (2000): Neue Technik verlangt neue pädagogische Konzepte, in Online – Hochschulentwicklung durch neue Medien, Verlag Bertelsmann Stiftung, Siemens Nixdorf Stiftung (Hrsg.), Gütersloh, S. 31-49.

Heyse, V., Erpenbeck, J. (2009): Kompetenztraining 64 Modulare Informations- und Trainingsprogramme für die betriebliche, pädagogische und psychologische Praxis, Schäffer-Poeschel Verlag, Stuttgart.

Knight, J. K., Wood, W. B. (2005): Teaching More by Lecturing Less. Cell Biology Education. Vol. 4, 298-310.

Kortemeyer, G., Kashy, E., Benenson, W., Bauer, W. (2008): Experiences Using the Open-Source Learning Content Management and Assessment System LON-CAPA in Introductory Physics Courses. The American Journal of Physics, Vol 76, 438-444.

Liggesmeyer, P. (2002): Software-Qualität: Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag

LON-CAPA: The Learning Online Network with CAPA. www.lon-capa.org

Ludewig, J. (2009): Erfahrungen bei der Lehre des Software Engineering, in Jaeger, U. (Hrsg.) und Schneider K. (Hrsg.): Softwareengineering im Unterricht der Hochschulen: SEUH 11, Hannover 2009, dpunkt Verlag.

Macke, G., Hanke, U., Viehmann, P. (2009): Hochschuldidaktik, Lehren, vortragen, prüfen, Beltz Verlag, Weinheim.

Mazur, E. (1997): Peer Instruction. Upper Saddle River. Prentice Hall.

Maier, D. (2000): Accelerated Learning, Handbuch zum schnellen und effektiven Lernen in Gruppen, managerSeminare Verlags GmbH, Bonn.

Meyer, H. (2009): Was ist guter Unterricht. Cornelsen Verlag, Berlin.

Mottok, J., Hagel, G., Utesch, M., Waldherr, F. (2009): Konstruktivistische Didaktik - Ein Rezept für eine bessere Softwareengineering Ausbildung?, im Tagungsband des Embedded Software Engineeing Kongress' 2009, S. 601-610.

Mottok, J., Joas, F. (2010): Aktivierende Lehre in der Erwachsenenbildung - Erfahrungen mit dem konstruktivistischen Methodenbaukasten in der Software Engineering Ausbildung, Jahrestagung der Deutsche Gesellschaft für wissenschaftliche Weiterbildung und Fernstudium e.V. (DGWF) an der Hochschule Regensburg.

Mottok, J., Schroll-Decker, I., Hagel, G., Niemetz, M., Scharfenberg, G. (2012): Internal Conferences as a Constructivism Based Learning Arrangement for Research Master Students in Software Engineering. The 2012 International Conference on Frontiers in Education: Computer Science and Computer Engineering. Las Vegas, USA.

Müller-Amthor, M. (2012): Eingereichter BMBF-Einzelantrag zum Qualitätspakt der Lehre: "Kompetenzorientierte Adaption Innovativer, interdisziplinärer und individueller Lehr-Lernstrategien zur regenerativen Ordination im Studierenden-Life-Cylce (KAI³ROS), Projektträger DLR.

Novak, G. M., Patterson, E. T. (1998): Just-in-Time Teaching: Actice Learner Pedagogy with WWW. IASTED International Conference on Computers and Advanced Technology in Education. Cancun, Mexico.

Novak, G. M., Patterson, E. T., Gavrin, A. D., Christian, W. (1999) : Just-in-Time Teaching: Blending Active Learning with Web Technology. Prentice-Hall.

Prince, M., Felder, R. (2007): The Many Faces of Inductive Teaching and Learning. Journal of College Science Teaching, Vo. 36, Nr. 5.

Reich, K. (2008): Konstruktivistische Didaktik – Lehr- und Studienbuch mit Methodenpool, 4. Auflage, Beltz Verlag, url: http:// methodenpool.uni-koeln.de.

Riegler, P. (2012): Just in Time Teaching: Wer liest und wer lehrt an der Hochschule? Im Tagungsband zum Forum der Lehre an der Hochschule Ansbach 24. Mai 2012, 89-95

Rupp, C., die SOPHISTen (2009): Requirements-Engineering und –Management. 5. Aufl. Carl Hanser Verlag.

Schmolitzky, A. W., Schümer, T. (2011): Towards Pedagogical Patterns on Feedback in Investitagtions of E-Learning Patterns – Context Facotrs, Problems, and Solutions Education. Kohls C., Wedekind, J., (Hrsg.) Information Science Reference, Hershey, New York, S. 181-190.

Simkins, S. Maier, M. H. (Hrsg.) (2010): Just-in-Time Teaching: Across the Disciplines, Across the Academy. New Pedagogies and Practices for Teaching in Higher Education. Stylus Publishing.

Simon, B., Kohanfars, M., Lee, J., Tamayo, K., Cutts, Q. (2010): Experience Report: Peer Instruction in Introductory Computing. SIGCSE 2010, Milwaukee, USA.

SMART: www.smarttech.com Produktseite für das SMART Response XE interactive response system, abgerufen 10.12.2012

Smith, M. K., Trujillo, C. Su T. T. (2011): The Benefits of Using Clickers in Small-Enrollment Seminar-Style Biology Courses. Cell Biology Education - Life Sciences Education, Vol. 10, 14-17.

Spillner, A., Linz, T. (2010): Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard. 4. Aufl. dpunkt Verlag.

Stelzer, R. (2008): Kompetenzen in der Hochschullehre, Rüstzeug für gutes Lehren und Lernen an Hochschulen, Merkur Verlag, Rinteln.

Welbers, U.; Gaus, O. (2009): The Shift from Teaching to Learning, Bertelsmann, Bielefeld.

Wörner, A. (2008): Lehren an der Hochschule, VS Verlag, Wiesbaden.

Software-Engineering Projekte in der Ausbildung an Hochschulen – Konzept, Erfahrungen und Ideen

Dr. Jens Liebehenschel, metio GmbH, Liederbach

Jens.Liebehenschel@metio.de

Überblick

In vielen Studiengängen wurden gegen Ende des Studiums Entwicklungs-Projekte in den Studienplan aufgenommen.

Der Autor beobachtete vor mehreren Jahren zwei Dinge: Für viele Absolvierende von Ingenieur- und Informatik-Studiengängen sind wesentliche Schritte im Entwicklungsprozess und deren Zusammenhänge nicht klar. Trotz oft detaillierter Vorgaben in den Firmen müssen noch immer viele Aspekte der Software-Entwicklung ausgestaltet werden.

Basierend darauf wurde ein Projekt mit starkem Praxisbezug konzipiert und seit 2007 sieben Mal an der FH Frankfurt und der TH Mittelhessen durchgeführt und kontinuierlich verbessert. Die Studierenden sollen die Möglichkeit haben, in einem „geschützten Raum“ eigene Erfahrungen zu machen, da Erfahrung im Rahmen einer Vorlesung schwer weitergegeben werden kann.

Das Ziel des Papers ist eine ausführliche Beschreibung der einzelnen Aspekte des Konzepts und eine Diskussion der Erfahrungen. Des Weiteren wird ein kurzer Ausblick zu angedachten Erweiterungsmöglichkeiten gegeben und am Ende werden die wesentlichen Punkte zusammengefasst.

Beschreibung des Projekts

Das Projekt *Objektorientierte Softwareentwicklung für Steuergeräte* wurde für Studierende in Informatik-Studiengängen am Ende ihrer Hochschulausbildung konzipiert. Eines der wichtigsten Kriterien bei der Konzeption war die Praxisnähe.

Drei Entwicklungsteams mit jeweils vier Personen arbeiten im Projekt an der gleichen Aufgabe, die Benotung erfolgt individuell. In den Teams ist eine Steuerung von Ampeln an einer Kreuzung oder eine Aufzugsteuerung zu entwickeln.

Für das Projekt existieren zwei entsprechende Modelle, außerdem drei unterschiedliche Sätze von Evaluationsboards mit automotive Controllern unterschiedlicher Leistungsklassen. Bilder der Modelle sind unter (metio Webseite, 2012) zu finden.

Das Projekt deckt die typischen Schritte in der Embedded-Software-Entwicklung von der Anforderungsanalyse mit dem Kunden bis hin zur Implementierung in C++ und zum Test ab (Hruschka, 2002; Automotive SPICE, 2007).

Durch Abgaben und Präsentationen sowie die Arbeit in Entwicklungsteams samt Planung der Arbeitspakete wird die Realität von Projekten bereits in der Hochschulausbildung vermittelt. Insbesondere die Praxisnähe macht das Projekt für Studierende interessant.

Alle Teilnehmer müssen mindestens zwei Mal die selbst erarbeiteten Abgaben vor der Gruppe präsentieren.

Jedes Team ist für die rechtzeitige Abgabe mindestens der folgenden Arbeitsprodukte verantwortlich:

- Planung des Projekts
- Anforderungsanalyse
- Architektur und Design
- Dokumentierte Implementierung
- Testkonzept
- Testfälle
- Lessons Learned

Darüber hinaus gibt es optionale Arbeitspakete mit entsprechenden Abgaben, zum Beispiel

- die erneute Anforderungsanalyse und die Überarbeitung von Architektur und Design, wenn eine Anforderungsänderung in das Projekt kam,
- die Erstellung einer Testsuite, mit deren Hilfe der Compiler der zur Verfügung stehenden Entwicklungsumgebung auf die Güte der C++-Unterstützung untersucht wird oder

- die Erstellung einer Simulation der Hardware wegen der Verzögerung ihrer Verfügbarkeit.

Diese zusätzlichen Abgaben erlauben die Variation des Projekts und die Anpassung an unterschiedliche Studiengänge mit unterschiedlichem zeitlichem Umfang für die Studierenden. Neben unterschiedlichen Arbeitspaketen wird die Lehrveranstaltung abgewandelt durch die Verwendung von verschiedenen Controllern oder durch abgewandelte Anforderungen an das System.

Innerhalb der Teams erfolgen zu zwei Zeitpunkten gegenseitige Bewertungen. Diese sind nur dem Dozenten bekannt. Außerdem finden in dem Rahmen kurze Einzelgespräche mit allen Teilnehmern statt. Natürlich ist dies bei Bedarf auch zu anderen Zeitpunkten möglich.

Ziele des Projekts

Das Projekt hat mehrere Ziele, die im Anschluss detaillierter beleuchtet werden:

- Verständnis der Besonderheiten eingebetteter Systeme bei den Studierenden
- Begeisterung für die Welt eingebetteter Systeme
- Fachliche Weiterentwicklung der Studierenden durch einen Schritt von der Hochschulausbildung hin zur Praxis
- Persönliche Weiterentwicklung der Studierenden durch Übernahme von Eigenverantwortung
- Spaß am Lernen und an der Wissensvermittlung

Eingebettete Systeme und deren Entwicklung haben viele Gemeinsamkeiten mit nicht eingebetteten Systemen, aber auch einige Unterschiede. In erster Linie sind es oft harte Echtzeit-Anforderungen, begrenzte Ressourcen und eine starke Nähe zur Hardware (Barr, 1999; Kienzle, 2009). Die Entwicklung orientiert sich meist stark an den Qualitätsmerkmalen Verfügbarkeit, Sicherheit (Betriebssicherheit und Angriffssicherheit), Kosten und Management der Varianten- und Versionenvielfalt. In der Realisierung kommen häufig zeitgesteuerte Systeme zur Anwendung und die erstellte Software läuft nicht auf dem Computer, auf dem sie entwickelt wurde. Diese Unterschiede und deren Auswirkungen für die Entwicklung sind insbesondere Studierenden der Informatik häufig nicht bewusst. Das Projekt schafft dieses Bewusstsein.

In der Praxis werden im Zeitalter einer immer höheren Durchdringung der Welt mit Computern (Mattern, 2012) noch mehr Personen für die Entwicklung eingebetteter Systeme benötigt. Das Projekt zielt darauf ab, bei den Studierenden Begeisterung für dieses interessante Themengebiet zu wecken. Die Begeisterung resultiert auch aus dem zu entwickelnden Endprodukt: Die Ampeln leuchten und lassen sich durch Knöpfe beeinflussen und der Aufzug bewegt sich.

Am Ende der Hochschulausbildung ist es notwendig, den Einstieg der Studierenden in die Praxis zu erleichtern. Dazu gibt es in vielen Informatik-Studiengängen Projekte oder Praktika. Die im Laufe des Studiums erlernten Bausteine werden individuell bei Bedarf vertieft. Insbesondere müssen sie aber kombiniert werden, um die Abhängigkeiten und den Nutzen besser zu verstehen. Dies unterstützt die Einordnung der späteren Tätigkeit in den gesamten Entwicklungsprozess und damit auch das Verständnis für das Zusammenwirken aller notwendigen Schritte in der Produktentwicklung.

Neben der fachlichen Weiterentwicklung wird auch im Rahmen der Möglichkeiten an geeigneten Stellen an der persönlichen Weiterentwicklung der Studierenden gearbeitet (Vigenschow, 2007). Im Projekt treten immer wieder schwierige Situationen wie beispielsweise Konflikte auf. Es werden bei Bedarf erste Lösungsansätze vermittelt und die Umsetzung in der Praxis diskutiert. Außerdem wird den Studierenden das Verständnis Ihrer Eigenverantwortung näher gebracht.

Bei vielen Studierenden konnte eine große Weiterentwicklung der praxisrelevanten Fähigkeiten beobachtet werden – sicherlich unterstützt durch die große Freude an der Projektarbeit. Nicht zuletzt wird das zeitaufwändige Projekt immer wieder durchgeführt, weil die Wissensvermittlung sehr viel Spaß macht.

Konzept und Erfahrungen

In diesem Kapitel werden die im Projekt verwendeten didaktischen Konzepte vorgestellt. Mehrere Punkte entsprechen den Rahmenbedingungen aus (Kleuker, 2011).

Praxisnähe

Ein wesentlicher Aspekt des Konzeptes ist die Fokussierung auf die Praxisnähe. In der Umsetzung wird die zur Verfügung stehende Zeit weniger für ein schwierig zu erstellendes System verwendet. Vielmehr soll ein Verständnis für die Phasen im Projekt, die notwendigen Arbeitsprodukte und Ihre

Abhängigkeiten geschaffen werden. Am Anfang des Studiums liegt der Fokus eher auf einzelnen Bausteinen, am Ende des Studiums sollten sie geeignet zusammengesetzt werden. Das zu lösende Problem und die Implementierung sind wenig komplex, aber doch nicht trivial. Dies zeigt sich sowohl bei der Anforderungsanalyse als auch im weiteren Projektverlauf, wenn im Team festgestellt wird, dass die Anforderungen doch nicht eindeutig oder komplett waren.

Wie schon eingangs beschrieben werden die typischen Schritte in der Embedded-Software-Entwicklung von der Anforderungsanalyse mit dem Kunden bis hin zur Implementierung und zum Test abgedeckt. Realistische Projektsituationen werden partiell durchlebt. Dazu müssen entsprechende Arbeitspakete bearbeitet und vor der Abgabe durch die Studierenden aus dem Team einem Review unterzogen werden. Dies dient dem Verständnis des Inhalts von Arbeitsprodukten und zeigt den Studierenden den Zusammenhang der in Entwicklungsprojekten benötigten Dokumentation.

Die Ergebnisse müssen durch die Ersteller vor allen Gruppen präsentiert werden. Dabei ist nicht unbedingt das Ziel, aus den Abgaben Folien zu erstellen, sondern die Präsentation basierend auf den abgegebenen Arbeitsprodukten zu gestalten. So werden die Anforderungen anhand eines Textdokuments oder die Architektur anhand von Architektursichten in Architektur-Werkzeugen oder in anderen Dokumenten vorgestellt. In der Praxis werden Themen oft anhand der Arbeitsprodukte besprochen, ohne aufwändige Präsentationen zu erstellen. An diesen Stellen werden die Studierenden immer wieder angeleitet, Dokumentation im sinnvollen Umfang anzufertigen. Sie soll der Kommunikation dienen und verständlich sein.

Schließlich ist für die Projekte in der Praxis gegenüber den kleinen Projekten im Studium an den Hochschulen die Teamarbeit sehr viel wichtiger. Dies wird den Studierenden nahe gebracht, in dem einerseits die Aufteilung der Arbeit in parallel zu erstellende Arbeitspakete notwendig ist, andererseits aber die Arbeitspakete auch voneinander abhängen, so dass Abhängigkeiten der verschiedenen Mitglieder in den Teams bestehen und eine gute Zusammenarbeit unerlässlich ist.

Überraschendes

Ein weiterer wichtiger Baustein ist das Erzeugen von Aha-Effekten. Dabei wird das Ziel verfolgt, bei den Studierenden wichtige Aspekte im Gedächtnis zu verankern, weil sie diese nicht nur erzählt bekommen, sondern selbst erleben. Dies geschieht zum Beispiel durch starke Übertreibung, bewusste Fehlinterpretationen oder das „Vorhalten eines Spiegels“.

Bei der Anforderungsanalyse gibt sich der Kunde (der Autor) zunächst bewusst unkooperativ und dreht den Studierenden das Wort im Mund herum oder ignoriert Ihre Fragen. So merken die Studierenden, dass die Ermittlung von Anforderungen schwierig ist. Nachdem dieses Verhalten des Kunden beleuchtet wurde, gibt sich der Kunde kooperativer, damit die Studierenden ihrem Ziel nach eindeutigen und vollständigen Anforderungen näher kommen können.

Wie zuvor angesprochen ist ein weiterer wichtiger Aspekt die Vermittlung von Wissen über den Zusammenhang der Arbeitsprodukte im Software Engineering. Beispielsweise basiert das Design auf Entscheidungen, diese wiederum auf Anforderungen. Diese Zusammenhänge verstehen die Studierenden durch ihre Abgaben. Meistens wird in das Projekt mindestens eine Anforderungsänderung eingebaut. Das Ziel ist es, den Studierenden Konsequenzen dieser in der Praxis üblichen Situation zu zeigen. Es soll das Bewusstsein geschärft werden, dass in solchen Situationen verschiedenste Arbeitsprodukte zu überarbeiten sind.

Oft ist bei Studierenden der genaue Anteil der Implementierung am Gesamtumfang des Projektes nicht bekannt. Die Studierenden werden der Reihe nach gefragt, wie hoch sie den Anteil schätzen. Am Ende nennt der Autor einen Anteil von 10-15%. Natürlich hängt dies stark vom Umfang des Projekts, der Domäne und der Entwicklungsansätze ab. Da meist die Schätzungen deutlich darüber liegen, wird der Aufwand für andere Aufgaben wie beispielsweise Anforderungsanalyse oder Änderungsmanagement erläutert.

Insbesondere für Studierende ohne vorherigen Kontakt mit der Entwicklung eingebetteter Systeme ist die Erfahrung interessant, dass die Arbeit mit Evaluationsboards sehr zeitraubend sein kann. Die Ausführungsumgebung für die Programme ist auf einem vom Entwicklungs-PC getrennten Computer, der in Betrieb genommen werden muss. Außerdem muss die Software geladen werden, und darüber hinaus ist die Dokumentation oft unvollständig oder fehlerhaft.

Rollenspiele

Das Projekt lebt von Rollenspielen. Der Dozent schlüpft wie im vorherigen Abschnitt beschrieben beispielsweise in die Rolle des Kunden, erläutert immer zu geeigneten Zeitpunkten was gerade passiert ist und gibt Lösungsmöglichkeiten vor, die dann im weiteren Verlauf erprobt werden können. Eine wichtige Erfahrung bei den Rollenwechseln ist, den Studierenden jederzeit klar zu machen, in welcher Rolle man sich gerade befindet, da es sonst verwirrend ist.

Gerade in der Rolle eines Kunden kann bewusst eine ganz andere „Sprache“ verwendet werden. Damit kann den Studierenden aufgezeigt werden, dass es oft nicht möglich ist, sich mit Kunden über das System mit den technischen Begriffen der Entwickler zu unterhalten, sondern die Unterhaltung in der Sprache des Kunden geführt werden muss.

Ein Erlebnis des Autors war die Bemerkung einiger Studierender, dass das Verhalten in der Rolle des Kunden während der Anforderungsanalyse absurd sei. Dies wurde vom Autor abgeschwächt – natürlich hatte er aus didaktischen Gründen übertrieben. Erstaunlicherweise berichteten andere Studierende mit bereits vorhandener Praxiserfahrung, dass genau solche Situationen so oder so ähnlich wirklich in der Praxis vorkommen.

In einem Projekt gab es in einem Team sehr große Probleme, so dass diese trotz mehrerer Gespräche mit einzelnen Personen und im Team nicht gelöst werden konnten. In diesem Fall musste der Autor eingreifen, um den Projekterfolg nicht zu gefährden und allen Beteiligten realistische Noten geben zu können. Der Dozent schlüpfte dazu in die Rolle des Managers und unterteilte das Team. Zwei neue Teams mussten nur noch einen Teil der Aufgaben bearbeiten. Die Kommunikation-Schnittstelle wurde auf das absolut notwendige Maß reduziert.

Eigenverantwortung

In vielen Firmen existieren für System- und Software-Entwicklungs-Projekte sehr detaillierte Regelungen und andere Vorgaben, welche Arbeitsprodukte wie aussehen, welche Inhalte vorhanden sein müssen, wie sie ineinandergreifen, usw. Dennoch ist es oft nicht klar, wie genau ein zu erstellendes Arbeitsprodukt aussieht oder wie an die Erstellung eines Arbeitsprodukts herangegangen werden kann. Es ist in der Praxis nicht sinnvoll – wenn nicht sogar unmöglich – das Vorgehen in jedem einzelnen Spezialfall ganz feingranular zu beschreiben. Daher sind die an der Entwicklung beteiligten Personen immer wieder in der Situation, eigenverantwortlich die richtigen Entscheidungen zu treffen. Das Vorgehen muss pragmatisch sein. Einerseits muss im Sinne der Vorgaben gearbeitet werden, andererseits sollte der Aufwand in einem vertretbaren Maß bleiben. Diese Abwägung ist in der Praxis sehr wichtig.

Die Studierenden machen eine Projektplanung, ohne dass zunächst detaillierte Vorgaben zu den erforderlichen Schritten – wie die zu erstellenden und abzugebenden Arbeitsprodukte – vorhanden sind. Auch zu zeitlichen Zusammenhängen im Projekt ist ihnen außer den Präsenzterminen und dem nächsten Abgabetermin nichts bekannt. Nach der ersten Abgabe erhalten sie alle weiteren Abgabetermine mit den Arbeitsprodukten. Die manchmal dadurch notwendige Überarbeitung der Planung wird zusammen mit den anderen Arbeitsprodukten nach der Anforderungsänderung abgegeben.

Durch die geforderte Planung der Arbeitspakete ist eine intensive Beschäftigung mit Inhalt und Dauer der erforderlichen Schritte und deren Abhängigkeiten unerlässlich. Diese Herangehensweise erscheint vielen Studierenden zunächst sehr herausfordernd, weil die Freiheitsgrade in vorherigen Studienabschnitten eher gering sind. Durch diese Herausforderungen haben die Studierenden die Möglichkeit der Weiterentwicklung ihres Verständnisses der Software-Entwicklung, sie werden spezifisch gefördert.

Nicht nur das Management des Projektes mit Aspekten wie Zeitplanung und Verteilung von Arbeitspaketen liegt komplett in der Hand der Studierenden. Auch viele technische Aspekte wie geeignete Ablage der entstehenden Arbeitsprodukte in einer Versionsverwaltung und verwendete Werkzeuge und Inhalte der einzelnen Arbeitsprodukte werden durch die Studierenden eigenverantwortlich gestaltet. Zum Beispiel werden für die Dokumentation der Architektur keine Sprachen wie UML (UML, 2012) oder SysML (SysML, 2012) vorgegeben. Und wenn ein Team sich für eine dieser Modellierungssprachen entscheidet, dann muss es außerdem ein geeignetes Werkzeug auswählen und entscheiden, welche Sichten auf das System vorhanden sind und wie sie ausgestaltet werden (Zörner, 2012).

Erfahrungsgemäß fällt es den Studierenden schwer, ein Testkonzept zu erstellen. Es scheint zunächst schwierig zu sein, die Inhalte eines Testkonzeptes zu erfassen. Nach Diskussion über das Thema verstehen die Studierenden, dass es nichts anderes ist, als das Vorgehen in den Testphasen zu beschreiben und die Art der Tests samt Hintergründen zu erläutern.

Der Weg ist das Ziel

Gerade die genaue Ausgestaltung von Arbeitsprodukten ist auch in der Praxis oft ein Problem, wie am Anfang des letzten Abschnitts angesprochen.

Daher wird im Projekt als Aufgabenstellung ein Ziel vorgegeben, jedoch nicht der Weg dorthin. Dies wird den Studierenden am Anfang der Veranstaltung auch mitgeteilt, damit sie von Anfang an Klarheit über den Ablauf des Projektes haben.

Den Studierenden werden immer wieder – auch angeregt durch Abgaben oder Fragen – Denkanstöße zum Weg der Problemlösung gegeben. Einige Beispiele dazu sind im nächsten Abschnitt zu finden. Durch Hinterfragen der gegangenen Schritte werden die Studierenden an strukturierte Herangehensweisen an Probleme herangeführt. Sie erhalten Anregungen für mögliche Lösungswege. Diese können im Projekt zu einem späteren Zeitpunkt noch genutzt werden, zum Beispiel bei der Überarbeitung der Arbeitsprodukte nach einer Anforderungsänderung. Den Studierenden werden auf diese Weise immer wieder Anregungen zur Selbstreflektion gegeben, um Vorgehen und Verhalten zu analysieren und zu verbessern.

In der Praxis werden Arbeitsergebnisse immer wieder Reviews unterzogen. Auch dies wird im Projekt geübt. Zu jeder Abgabe ist vom gesamten Team ein Review durchzuführen. Neben den Abgaben sind zusätzlich die im Review gefundenen Aspekte mit abzugeben. Damit kann frühzeitig geübt werden, von anderen Personen erstellte Arbeitsprodukte zu verstehen und Fehler aufzudecken. Auch wird klar, dass ein Review auf das Aufdecken von Fehlern zielt und nicht etwa den Ablauf oder Inhalt der Arbeitsprodukte beschreibt.

Bei der Kurzpräsentation der einzelnen abzugebenden Arbeitsprodukte durch den Dozenten ergeben sich bei den Studierenden Fragen. Es werden nicht direkt Antworten gegeben, welche Schritte in welcher Reihenfolge durchgeführt werden müssen. Jedoch merken sie schnell, dass die Antworten sie in die richtige Richtung leiten. Wenn die Studierenden gute Fragen zu stellen, erhalten sie vom Dozenten immer mehr Information. Dies ist auch ein Lernziel, Fragen zu stellen, wenn Dinge nicht komplett verstanden sind.

Vertiefung spezieller Themen bei Bedarf

Bei Bedarf werden im Projekt spezielle Themen vertieft, zum Beispiel bei Fragen der Studierenden oder durch den Dozenten erkannten Unklarheiten. Die Erfahrung zeigt, dass manche Themen in jedem Projekt diskutiert werden müssen. Dies liegt zum Teil an speziellen Eigenschaften von eingebetteten Systemen, aber auch zu allgemeinen Begriffen der Informatik besteht oft Klärungsbedarf.

Beispielsweise werden meistens die Begriffe Architektur und Design (Goll, 2011; Starke, 2008), Safety und Security (Löw, 2010), synchrone und asynchrone Kommunikation (Bengel, 2008), Nebenläufigkeit und Synchronisation (Vogt, 2012) und kritischer Pfad (Wikipedia: Kritischer Pfad, 2012) anhand von Beispielen diskutiert, um ein besseres Verständnis bei allen Studierenden zu erreichen.

Oft werden verschiede Entwicklungsansätze, vom Wasserfallmodell bis hin zu agilen Ansätzen (Summerville, 2007; Vliet, 2008) mit ihren Eigenschaften erläutert und Einsatzbereiche diskutiert.

Meistens besteht bei den Studierenden kein Wissen über unterschiedliche Klassen von Anforderungen (Pohl, 2007). Funktionale Anforderungen sind diejenigen, die vom System komplett erfüllt werden müssen. Qualitätsanforderungen (oder oft auch nicht-funktionale Anforderungen genannt) werden immer zu einem gewissen Grad erfüllt. Wichtig für (Architektur-)Entscheidungen sind die Qualitätsanforderungen (Bass, 2003). Dies kann anhand des folgenden Beispiels veranschaulicht werden.
Es ist ein System zum Sortieren von Daten zu entwickeln (funktionale Anforderung). Dabei ist die mittlere Laufzeit wichtiger als der Speicherplatzbedarf (Qualitätsanforderung). Die Bewertung sieht wie folgt aus:

	Mittlere Laufzeit	Speicher (Stack)
Bubblesort	–	+
Quicksort	+	–

Tabelle 1: Alternativen und Bewertung

Nur mit der Qualitätsanforderung kann eine Entscheidung für eine Alternative getroffen werden:

Quicksort, weil die Laufzeit wichtiger als der Speicherplatzbedarf ist.

Die bei den meisten eingebetteten Systemen vorhandene zeitgesteuerte Ausführung (Kienzle, 2009) ist vielen Studierenden nicht bekannt. Es werden in der Praxis übliche Lösungen diskutiert, die Hintergründe der Ansätze beleuchtet und mit anderen Ausführungsparadigmen verglichen.

Alle diese technischen Diskussionen werden generell mit allen Studierenden gemeinsam geführt. Wenn möglich erklären die Studierenden die Sachverhalte, oder sie werden zumindest angehalten, zur Diskussion beizutragen. Am Ende der Diskussion wird durch Rückfragen geklärt, ob die Inhalte verstanden worden sind.

Die Dauer der Diskussionen richtet sich nach dem Inhalt – von wenigen Minuten bis zu einer Stunde. Natürlich können manche Themen nicht erschöpfend erörtert werden, aber das Verständnis oder Problembewusstsein kann deutlich erhöht werden.

Auch nicht-technische Aspekte werden bei Bedarf aufgegriffen. Wenn beispielsweise bei der Präsentation von Ergebnissen Aspekte verbessert werden können, werden nach Rücksprache mit den Vortragenden diese Themen in den Veranstaltungen angesprochen. Dies können zum Beispiel Hinweise zur Gestaltung der präsentierten Unterlagen oder verwendeten Tools sein. Aber auch Tipps zur Verbesserung der Präsentationsfähigkeiten, zum Beispiel der Rhetorik, werden gegeben.

Oft gehen mehrere Personen eines Teams gemeinsam vor die Gruppe, um Ihre Ergebnisse vorzustellen. Jede Person weiß genau, welchen Teil sie vorstellen wird, jedoch ist nicht klar, wer eine kurze Einführung gibt oder wer beginnt. Das wird vor der Gruppe besprochen. Dieses häufiger beobachtete Phänomen wird abgestellt, nachdem die Studierenden darauf hingewiesen werden. Dadurch werden zukünftige Vorträge – sei es an der Hochschule oder in Unternehmen – professioneller.

Organisatorisches

Am Anfang der Lehrveranstaltung ist es notwendig, den Studierenden das Handwerkszeug aus verschiedenen Themengebieten mitzugeben, um die Aufgaben im Projekt meistern zu können. Dazu findet ein ganztägiger Präsenztermin vor dem Beginn der Vorlesungszeit statt. Jedoch werden schon ab dem zweiten Präsenztermin die Studierenden immer mehr der zur Verfügung stehenden Zeit mit Präsentationen ausfüllen, und spätestens ab dem vierten Präsenztermin reagiert der Dozent nur noch, bringt aber von sich aus keine weiteren Themen ein.

Wie eingangs beschrieben, müssen die Teams regelmäßig Arbeitspakete abgeben. Diese werden nach der Abgabe bewertet und müssen zu einem Präsenztermin, der wenige Tage nach der Abgabe stattfindet, vorgestellt werden. Abgaben und Präsenztermine finden in Abhängigkeit des Aufgabenumfangs alle zwei bis vier Wochen im Umfang von vier Vorlesungsstunden statt.

Als Teamgröße wird vier festgesetzt. Drei Personen pro Team wäre auch möglich. Bei weniger Personen ist der Arbeitsaufwand der Einzelnen zu hoch, bei mehr Personen wird die Aufteilung der Arbeitspakete schwierig. Eine durchgängige Mitarbeit jedes Teammitglieds könnte nicht sichergestellt werden.

Ein weiterer wichtiger Aspekt ist die individuelle Benotung jedes Teilnehmers. Natürlich geht in die Benotung zum kleinen Teil das Teamergebnis mit ein, aber wesentlich sind die Beteiligung und die erstellten Arbeitsergebnisse. Validiert wird die Fairness der individuellen Benotung durch eine gegenseitige Bewertung der Teammitglieder, die zwei Mal durchgeführt wird, in der Mitte und am Ende der Lehrveranstaltung. Diese liefert dem Autor fast immer eine Bestätigung, dass die Sicht im Team auf die Leistungen der einzelnen Mitglieder sich mit seiner deckt.

Der Lernerfolg kann nur dann sichergestellt werden, wenn die Studierenden die Aufgaben selbst bearbeiten und dadurch ihre eigenen Erfahrungen machen. Da oft Arbeitsergebnisse in der Studierendenschaft weitergegeben werden, ist es erforderlich, jedes Semester Änderungen am Projekt zu machen. Diese Änderungen sind zum einen unterschiedliche Themen, aber auch für jedes Thema unterschiedliche Anforderungen und immer wieder andere abzugebende Arbeitsergebnisse. Die Variationsmöglichkeiten wurden eingangs bereits beschrieben.

Das komplette Material für die Veranstaltung erhalten die Studierenden nach dem ersten und zweiten Präsenztermin, das Evaluationsboard gegebenenfalls später – je nach Ausgestaltung des Projektes. Literaturverweise gibt der Autor ausschließlich nach Bedarf aus.

Schließlich soll ein Punkt nicht unerwähnt bleiben. Der Dozent führt die Lehrveranstaltung mit viel Freude durch. Auch bei den Studierenden soll der Spaß an der Arbeit nicht zu kurz kommen. Aber trotz eines guten Arbeitsklimas und einem freundschaftlichem Umgang ist die Wahrung der notwendigen Distanz wichtig – dies ist immer wieder eine Gradwanderung.

Feedback der Studierenden

Letztendlich ist es nicht nur notwendig, dass die richtigen Dinge vermittelt werden, sondern auch eine möglichst große Nachhaltigkeit erreicht wird. Das Projekt soll den Studierenden den Einstieg ins Berufsleben vereinfachen. Dies ist nur möglich, wenn die Studierenden meinen, etwas aus der Veranstaltung „mitzunehmen". Daher wird regelmäßig im Projekt um Feedback gebeten – auch nach der Bekanntgabe der Endnote, um ein möglichst ehrliches Feedback zu erhalten.

Die folgenden Äußerungen von Studierenden werden sinngemäß und verkürzt wiedergegeben. Zunächst eher allgemeine Punkte zum Projekt und im Anschluss ein paar Punkte zu inhaltlichen Aspekten.

- Das Projekt hatte einen hohen Praxisbezug – das gab es sonst im Studium nicht.
- Es war gut, ein komplettes Projekt von Anfang bis Ende zu machen.
- Es war gut, dass keine Prozessanforderungen vorgegeben waren.
- Die Arbeit im Team mit der eigenen Teamorganisation war gut.
- Das Ziel war klar, aber der Weg nicht. Dennoch gab es viel Unterstützung beim Finden des richtigen Wegs.
- Oft wurden nicht alle Informationen gegeben, aber am Ende war doch alles rechtzeitig da.
- Die Praxisnähe war größer als bei allen Veranstaltungen vor dem Projekt.
- Es ist schwierig, vom Kunden gute Anforderungen zu erhalten.
- Die Rollenspiele waren sehr interessant, zum Beispiel die andere Sprache des Kunden.
- Die Einstiegsschwelle in das Thema war niedrig, weil das System einfach ist und keine komplexen Anforderungen besitzt.
- Inhaltlich war das Thema gut: Zunächst einfach und später kamen dann doch noch viele Details und Unklarheiten zu Tage.
- Die Anforderungsänderung war gut, die Auswirkungen auf das Design wurden klar.
- Die Arbeit an der Architektur war gut; ein Mittelweg zwischen zu viel und zu wenig Architektur musste gefunden werden.

Anhand des Feedbacks und vor allem der Diskussionen mit den Studierenden erhält der Dozent den Eindruck, dass der Ansatz zielführend ist. Die Studierenden können die im Studium gelernten Inhalte kombinieren, lernen etwas über deren Abhängigkeiten und erhalten so einen Einblick in die Praxis. Außerdem kann aus dem großen Engagement vieler Studierender geschlossen werden, dass auch die Begeisterung für die Welt eingebetteter Systeme gelungen ist.

Ein letzter Punkt soll nicht unerwähnt bleiben. Das Projekt ist erfahrungsgemäß sehr schnell ausgebucht. Dies gibt dem Autor das gute Gefühl, mit dem Projekt auf dem richtigen Weg zu sein. Die Teilnehmenden entscheiden sich aktiv für das Projekt, um mehr über eingebettete Systeme zu lernen, und bringen daher auch eine hohe Motivation für das Themengebiet mit. Auch diese Tatsache kann ein Grund für das positive Feedback sein.

Einige der ehemaligen Teilnehmer durfte und darf der Autor weiter begleiten. Ein paar Details werden dazu im nächsten Abschnitt vorgestellt.

Ehemalige Projektteilnehmer

Immer wieder haben Studierende aus den Projekten die Möglichkeit, in der Firma des Autors oder bei ihren Kunden Praktika abzuleisten, ihre Abschlussarbeiten anzufertigen, oder als Angestellte in herausfordernden Kundenprojekten zu arbeiten.

In der Zusammenarbeit zeigt sich das erlernte Verständnis für die in der Entwicklung zu erstellenden Arbeitsprodukte. Ein Beispiel ist die Entwicklung von kleinen Werkzeugen. Zunächst wird am Verständnis der Anforderungen gearbeitet, danach werden grundsätzliche Designüberlegungen gemacht. Erst dann wird mit der Umsetzung begonnen.

Auch an einer anderen Stelle zeigt sich eine strukturierte Vorgehensweise. An vielen Stellen müssen Entscheidungen getroffen werden. Dies kann die Auswahl eines Werkzeuges oder eines Frameworks sein, oder auch Designentscheidungen bei der Erstellung von Software. Alternativen werden bewertet und die Entscheidung wird dokumentiert – sie entsteht nicht einfach so und ist nachvollziehbar.

Eine weitere Beobachtung hat der Autor gemacht. Auch wenn die Mitarbeiter sehr selbstständig arbeiten, bitten sie an wichtigen Stellen in den Projekten von sich aus um Reviews, damit sie eine höhere Sicherheit erhalten, sich auf dem richtigen Weg zu befinden.

Ideen für die Zukunft

Das Projekt hat sich über die Jahre immer weiter entwickelt. Drei kombinierbare Ideen für die zukünftige Entwicklung werden kurz aufgelistet:

- Ein Roboter ist anzusteuern, dies erfordert in der Software gegebenenfalls ein Umweltmodell.
- Die Steuerung erfolgt über eine ebenfalls zu erstellende App. Dazu würden neue Controller mit entsprechenden Schnittstellen benötigt.
- Der Teambuilding-Aspekt wird in das Projekt eingebracht, angelehnt an (Schmedding, 2011).

Checkliste als Zusammenfassung

Das Ziel dieser Arbeit ist es, die Konzepte hinter dem mehrfach durchgeführten und verbesserten Projekt offen zu legen und von den Erfahrungen zu berichten. Daher werden als Zusammenfassung abschließend die aus Sicht des Autors wesentlichen Erfolgsfaktoren aufgelistet, um Projekte mit gutem Lernerfolg für die Studierenden durchzuführen.

- ☐ Ist das zu lösende Problem nicht zu komplex?
- ☐ Werden essentielle Schritte der Software-Entwicklung „im Kleinen“ erlebt?
- ☐ Sind die Ziele und nicht die Wege vorgegeben?
- ☐ Werden die Studierenden beim Finden guter Wege hinreichend unterstützt?
- ☐ Präsentieren die Studierenden ihre Ergebnisse?
- ☐ Gibt es eingeplante „Fallstricke“?
- ☐ Werden elementare Aspekte durch Konzepte wie Rollenspiele nachhaltig vermittelt?
- ☐ Werden die Studierenden angeregt, über ihr Vorgehen nachzudenken?
- ☐ Sind die Dozierenden für notwendige Unterstützung gut erreichbar?
- ☐ Besteht ein Team aus drei oder vier Personen?
- ☐ Müssen die Teams das Projekt im Wesentlichen eigenverantwortlich durchführen?
- ☐ Erfolgt die Benotung individuell?
- ☐ Macht die Veranstaltung allen Spaß?

Literatur

Automotive SPICE Prozessassessment (2007), http://www.automotivespice.com/AutomotiveSPICE_PAM_v23_DE.pdf

Barr, M. (1999): Programming Embedded Systems in C and C++

Bass, L. et al. (2003): Software Architecture in Practice, 2nd edition

Bengel, G. et al. (2008): Masterkurs Parallele und Verteilte Systeme

Goll, J. (2011): Methoden und Architekturen der Softwaretechnik

Hruschka, P. et al. (2002): Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML

Kienzle, E. et al. (2009): Programmierung von Echtzeitsystemen

Kleuker, S. et al. (2011): Vier Jahre Software-Engineering-Projekte im Bachelor – ein Statusbericht, SEUH 2011

Löw, P. et al. (2010): Funktionale Sicherheit in der Praxis

Mattern, F., GI Webseite (2012) http://www.gi.de/service/informatiklexikon/detailansicht/article/pervasiveubiquitous-computing.html

metio Webseite mit Bildern der Modelle (2012): Ampel: http://www.metio.de/ger/ampel.html, Aufzug: http://www.metio.de/ger/aufzug.html

Pohl, K. (2007): Requirements Engineering

Schmedding, D. (2011): Teamentwicklung in studentischen Projekten, SEUH 2011

Sommerville, I. (2007): Software Engineering

Starke, G. (2008): Effektive Software-Architekturen

SysML (2012), http://www.omg.org/spec/SysML/

UML (2012), http://www.omg.org/spec/UML/

Vigenschow U. et al. (2007): Soft Skills für Softwareentwickler

Vliet, H. van (2008): Software Engineering

Vogt, C. (2012): Nebenläufige Programmierung

Wikipedia: Kritischer Pfad (2012), http://de.wikipedia.org/wiki/Methode_des_kritischen_Pfades

Zörner, S. (2012): Software-Architekturen Dokumentieren und Kommunizieren

Session 2

Programmierausbildung

Eine softwaretechnische Programmierausbildung?

Axel W. Schmolitzky, Universität Hamburg

schmolit@informatik.uni-hamburg.de

Zusammenfassung

Eine solide Programmierausbildung bildet die Grundlage jeder softwaretechnischen Ausbildung. Art und Umfang der Programmierausbildung können jedoch sehr unterschiedlich sein, je nach Schwerpunkt des jeweiligen Studiengangs. In diesem Artikel werden auf verschiedenen Ebenen Alternativen diskutiert, die für die Gestaltung der einführenden Programmierausbildung bestehen, und mit den Anforderungen abgeglichen, die sich bei einer Schwerpunktsetzung auf die Softwaretechnik ergeben.

1 Einleitung

Programmieren („Codieren“) gehört zum Handwerkszeug jedes Softwaretechnikers und jeder Softwaretechnikerin (Ludewig, 2010). Üblicherweise qualifizieren sich Studierende für softwaretechnische Herausforderungen über ein Studium der Informatik oder eines Faches mit hohem Informatik-Anteil (Typ 1 oder 2 der GI-Typisierung); in einem solchen Studium sollten sie auch das Programmieren erlernen.

1.1 Programmierausbildung…

Dass Programmieren einen hohen Stellenwert in Informatik-Studiengängen haben muss, spiegelt sich exemplarisch in der Weiterentwicklung der GI-Empfehlungen wider. Während in den *Empfehlungen zur Akkreditierung von Studiengängen der Informatik* (GI, 2000) aus dem Jahr 2000 der Begriff *Programmieren* nur sehr knapp auftaucht, werden in den Empfehlungen aus dem Jahre 2005 (GI, 2005) konkrete Programmierveranstaltungen mit konkreten Umfängen vorgeschlagen. Eine ähnliche Entwicklung erwähnt Walker für das Computing Science Curriculum der ACM und IEEE (Walker, 2010).

In diesem Artikel fassen wir alle Bemühungen Lehrender, Studierenden die Kunst des Programmierens zu vermitteln, unter dem Begriff *Programmierausbildung* zusammen. In jeder Programmierausbildung sollte die Fähigkeit trainiert werden, lösbare Probleme mit Hilfe der universellen Maschine Computer lösen zu können („problem solving“). Auf jeden Fall sollte ein Verständnis dafür aufgebaut werden, welche Probleme mit Computern gut gelöst werden können und welche eher nicht. Dies beinhaltet neben theoretischen Grundlagen auch praktische Fähigkeiten im Umgang mit Programmiersprachen.

1.2 …softwaretechnisch konkretisiert

Eine fundierte *softwaretechnische* Programmierausbildung erfordert darüber hinaus, dass die Studierenden Software nicht nur neu entwickeln lernen, sondern auch lernen, bestehende zu warten. Dies schließt mit ein, kompetent *über Software kommunizieren* zu können. Es schließt auch ein, Software selbst so zu gestalten, dass andere sie gut überarbeiten können. Dazu sollten das *automatisierte Testen* von Software und das für die Softwaretechnik so zentrale Konzept der *Schnittstelle* (u.a. für Spezifikationen), aber auch Quelltextkonventionen frühzeitig bei der Programmierung thematisiert werden.

Die Fähigkeit zum Programmieren und zum Kommunizieren über Software ist üblicherweise auch die Voraussetzung für eine erfolgreiche Teilnahme an dedizierten Modulen zum *Software Engineering*, die häufig erst ab dem 3. Semester angeboten werden. Informelles Feedback von Lehrenden solcher Veranstaltungen deutet an, dass bei den Teilnehmern häufig nicht die notwendigen grundlegenden Programmierkenntnisse vorhanden sind. Der Einfluss dieser Dozenten auf die Programmierausbildung ist häufig gering, da selten dieselben Personen sowohl Programmierung als auch Software Engineering lehren.

1.3 Aufbau

Dieser Artikel thematisiert die Gestaltung der *einführenden Programmierausbildung*[1] in Informatikstudiengängen deutscher Hochschulen, indem er auf verschiedenen Ebenen Alternativen ihrer Gestaltung diskutiert. Dabei orientiert er sich an unserer eigenen einführenden Programmierausbil-

[1] Unter „einführend“ sei hier zu verstehen, dass die entsprechenden Veranstaltungen in den frühen Semestern eines Bachelor-Studiums stattfinden und zum Pflichtanteil des jeweiligen Studiengangs gehören.

dung in den Modulen *Softwareentwicklung 1 und 2* (SE1 und SE2) an der Uni Hamburg, die wir seit einigen Jahren möglichst konsequent an softwaretechnischen Anforderungen ausrichten. Deren Gestaltungsprinzipien sollen hier zur Diskussion gestellt werden, quasi als eine mögliche Antwort auf die Frage, wie eine softwaretechnisch ausgerichtete Programmierausbildung aussehen kann.

Der Artikel ist folgendermaßen strukturiert, siehe auch Tabelle 1: In Abschnitt 2 werden Alternativen auf Studiengangsebene diskutiert, die für die Ausrichtung ganzer Bachelor-Studiengänge relevant sind. In Abschnitt 3 folgen Alternativen, die sich bei der Organisation einzelner Module ergeben. Abschnitt 4 diskutiert gezielt einige inhaltliche Alternativen in der Programmierausbildung mit Java. In allen drei Abschnitten werden jeweils bestehende (und häufig übliche) Ansätze vorgestellt und dann dem von uns gewählten gegenüber gestellt, der eher softwaretechnisch motiviert ist. Abschnitt 5 fasst den Artikel zusammen.

Alternativen…		
2	**… auf Studiengangsebene**	
	2.1	Wahl und Reihenfolge der Paradigmen
	2.2	Sprache und Programmierung
	2.3	Algorithmen vs. Interaktive Systeme
3	**… organisatorischer Art**	
	3.1	Vorlesungen: Folien vs. Ausführung
	3.2	Übungen: Heim- vs. Präsenzarbeit
4	**… inhaltlicher Art**	
	4.1	Smarte Sammlungen vor Arrays
	4.2	Interfaces vor Vererbung

Tabelle 1: Diskutierte Alternativen

2 Alternativen auf Studiengangsebene

Im Zuge der Umstellung auf das Bachelor/Master-System an den deutschen Hochschulen sind etliche neue Studiengänge entstanden, die im Kern Informatik-Inhalte vermitteln und darüber hinaus unterschiedliche Ausrichtungen anbieten.

Auf dieser Ebene, auf der strategische Entscheidungen für ganze Studiengänge (nicht nur in Bezug auf Module zur Programmierung) getroffen werden, stellt sich u.a. die Frage, welche Schwerpunkte gesetzt werden sollen: Steht eine berufliche Qualifizierung im Vordergrund oder eine Forschungsorientierung? Bei einer Forschungsorientierung: Welche Schwerpunkte werden betont? Je nach Forschungsschwerpunkt können sich auch unterschiedliche Schwerpunkte im Studienangebot ergeben; für Bildverarbeitung beispielsweise ist eine stärker formal-mathematische Ausbildung notwendig, während für Forschung zum Thema Softwarearchitektur eher praktische und konzeptionelle Inhalte im Vordergrund stehen sollten.

Aus diesen Überlegungen ergibt sich auch der Umfang, welcher der Programmierausbildung im Studienprofil zugestanden wird und der sich in Anzahl und Größe der einführenden Programmiermodule ausdrückt. Häufig sind es drei oder vier Veranstaltungen in den ersten drei bzw. vier Semestern, klassisch orientiert am früheren Grundstudium der Diplomstudiengänge. Typisch sind Veranstaltungen mit zwei oder vier Semesterwochenstunden (SWS) Vorlesung und zwei SWS Übungen dazu.

Ist dieser Rahmen prinzipiell abgesteckt, ergeben sich weitere modulübergreifende Fragen, die in den folgenden Abschnitten dargestellt werden.

2.1 Wahl und Reihenfolge der Paradigmen

In der Programmierung unterscheiden wir hier *imperative* von *deklarativen* Paradigmen. Zu den imperativen Paradigmen zählen wir auch die *objektorientierte Programmierung* (OOP), denn fast alle objektorientierten Sprachen basieren auf einem imperativen Kern. Beim deklarativen Paradigma unterscheiden wir in die Ausprägungen *funktionale Programmierung* und *logische Programmierung*.

Bei der Gestaltung von Studiengängen gibt es je nach thematischem Schwerpunkt die Möglichkeit, eine oder mehrere dieser Ausprägungen auszulassen. An einigen Hochschulen wird beispielsweise die logische Programmierung nicht gelehrt, teilweise wird sich sogar auf die imperative Programmierung beschränkt.

Aus softwaretechnischer Sicht ist die Kenntnis imperativer Programmierung zwingend erforderlich, um die Funktionsweise heutiger Computer verstehen zu können. Objektorientierte Programmierung ist aus softwaretechnischer Sicht vor allem deshalb interessant, weil damit konsequent die Schnittstelle zwischen Klient und Dienstleister und das Abstrahieren von Details eines Dienstleisters thematisiert werden kann. Vertiefte Kenntnisse in funktionaler oder logischer Programmierung hingen sind als Ergänzung zwar begrüßenswert, erscheinen jedoch im Notfall am ehesten entbehrlich, insbesondere falls Programmiermodule im Curriculum eine knappe Ressource sind.

Ein Ansatz: Es erfolgt ein Einstieg in die funktionale Programmierung im ersten Semester, auf den im zweiten oder dritten Semester Veranstaltungen zur imperativen Programmierung folgen.

Dies ist ein häufig praktiziertes Muster an deutschen Hochschulen (vor allem an Universitäten).

Ein oft genanntes Argument für diesen deklarativ orientierten Einstieg ist, dass alle Studierenden dann vom Kenntnisstand gleich seien, da nur die wenigsten Vorerfahrung mit deklarativen Programmierstilen mitbringen. Dieser „Vorteil" relativiert sich dann allerdings spätestens beim Einstieg in die imperative Programmierung.

An der Uni Hamburg wurde bis zum Jahr 2005 innerhalb des ersten Semesters sowohl die funktionale als auch die logische Programmierung gelehrt, im zweiten Semester folgten imperative und objektorientierte Grundlagen. Eine weitere Veranstaltung im dritten Semester ergänzte Konzepte zu Algorithmen und Datenstrukturen.

Alternative: Imperative Grundlagen werden einführend gelehrt, alternative Paradigmen darauf aufbauend später vermittelt.

Bei der Umstellung der Informatik an der Uni Hamburg auf das Bachelor-Master-System wurde 2005 auch die Programmierausbildung umgestellt. In den ersten beiden Semestern aller Bachelor-Studiengänge der Informatik wird die imperative und objektorientierte Programmierung thematisiert. Ab dem dritten Semester haben die Studierenden dann die Möglichkeit, Wahlpflichtmodule zur funktionalen und/oder zur logischen Programmierung zu wählen. Ergänzend wird ein Wahlpflichtmodul zu Algorithmen und Datenstrukturen angeboten.

Konsequenzen: Ein Einstieg im ersten Semester mit imperativer Programmierung führt dazu, dass es große Unterschiede bei den Vorkenntnissen der Teilnehmer gibt. Einige Studierende können bereits recht gut in Java programmieren, weil sie Informatik in der Schule belegt haben und/oder in ihrer Freizeit oder sogar beruflich programmiert haben, während andere Studierende noch keinerlei Vorerfahrung haben. Da die Veranstalter keine Vorkenntnisse voraussetzen dürfen, muss die Erstsemesterveranstaltung inhaltlich bei Null beginnen. Die Herausforderung besteht darin, ein Tempo zu wählen, das einerseits die Anfänger nicht innerhalb kürzester Zeit abhängt und andererseits Teilnehmer mit Vorkenntnissen nicht zu sehr langweilt.

In unserer einführenden Programmierveranstaltung SE1 (derzeit über 400 Teilnehmer aus allen Informatik-Studiengängen) liegt das Verhältnis von Programmieranfängern zu Teilnehmern mit Vorerfahrung seit Jahren bei ca. 30:70[2]. Wir versuchen diese ungleichen Voraussetzungen mit *Zusatzaufgaben* zu kompensieren, die nicht Teil der Scheinbedingungen sind, aber interessantere und anspruchsvollere Inhalte thematisieren. Diese Möglichkeit wird dabei nicht nur von Studierenden mit Vorerfahrung genutzt, sondern durchaus auch von motivierten Programmieranfängern.

Eine weitere Möglichkeit, die ungleichen Vorkenntnisse von einem Nachteil in einen Vorteil umzuwandeln, indem unterschiedlich starke Studierende zusammen arbeiten, diskutieren wir in Abschnitt 3.2.

2.2 Programmiersprache und Programmierung

Im Weiteren gehen wir von einer Umgebung mit einem imperativen Einstieg aus; dieser wird heutzutage oft mit der Programmiersprache *Java* (Gosling et al., 2005) vorgenommen. Als nächstes, ebenfalls modulübergreifend, kann die Frage nach der inhaltlichen Abstimmung zwischen den Programmiermodulen gestellt werden.

Ein Ansatz: Im ersten Semester wird eine Komplettübersicht über Java gegeben, quasi ein *Java-Crash-Kurs*. Im zweiten Semester folgt ein Modul, das klassische Algorithmen und Datenstrukturen mit Java thematisiert.

Das Problem mit diesem Vorgehen ist, dass Java trotz ihres vergleichsweise einfachen Sprachmodells keine leicht zu erlernende Sprache ist. Für Programmieranfänger ist der Bogen von imperativen Ausdrücken und sequentiellen Anweisungen mit typisierten Variablen hin zu Vererbung, Polymorphie, Generizität oder gar Nebenläufigkeit nicht leicht zu schlagen. Die Erwartung, dass die Teilnehmer nach nur einem Semester Java komplett „können", kann üblicherweise kaum erfüllt werden.

Alternative: Im ersten Semester werden die Mechanismen von Java nicht vollständig behandelt, sondern nur eine Teilmenge. Fortgeschrittene Mechanismen werden erst im zweiten Semester thematisiert. Dafür werden schon im ersten Semester elementare Konzepte von Algorithmen und Datenstrukturen eingeführt.

In unseren einführenden Modulen SE1 und SE2 behandeln wir im ersten Semester neben den imperativen Grundlagen (Ausdrücke, Anweisungen, Kontrollstrukturen, Rekursion) nur diejenigen objektorientierten Mechanismen, die *objektbasierte Programmierung* im Sinne von Wegner (Wegner, 1987) ermöglichen: Klassen, Objekte, Typen, (Java-)Referenzen, Schnittstellen. Vererbungskonzepte werden erst im zweiten Semester behandelt, ebenso wie die Mechanismen zur Ausnahmebehandlung.

Den im ersten Semester entstehenden Freiraum nutzen wir in dessen zweiter Hälfte für einen frühen Einstieg in die Grundlagen von Algorithmen und Datenstrukturen (Verkettete Listen, Array-

[2] Zwischenzeitlich lag es durch die Einführung neuer Studiengänge mit teilweise hohem interdisziplinärem Anteil bei 50:50, hat sich aber zuletzt wieder auf den oben genannten Wert eingependelt.

Listen, Bäume, Hash-Verfahren und Sortierverfahren), an denen die objektbasierten Elemente gut geübt werden können, ohne diese mit der Komplexität von Vererbungsmechanismen zu belasten.

Konsequenzen: Eine Verteilung der Sprachkonzepte von Java auf zwei konsekutive Module führt zu einer stärkeren Kopplung zwischen diesen Modulen. Sie sollten deshalb „aus einer Hand" gelehrt werden, um Reibungsverluste zu minimieren.

An der Uni Hamburg haben wir das „Glück", beide einführenden Programmierveranstaltungen gestalten zu dürfen. Dies hat uns die Möglichkeit gegeben, Inhalte anders zu schneiden und zu verteilen. Die Kehrseite ist, dass wir für diese beiden Module aufgrund der seit Jahren sehr großen Teilnehmerzahl regelmäßig einen hohen Betreuungsaufwand leisten müssen.

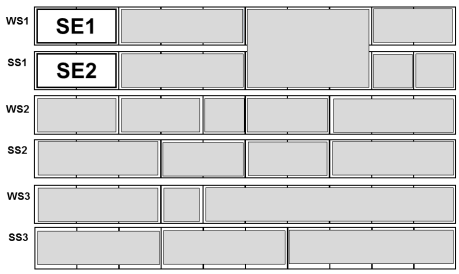


Abb. 1: Anteil der einführenden Pflichtmodule zur Programmierung in den Bachelorstudiengängen der Universität Hamburg mit hohem Informatik-Anteil

Insgesamt stellt sich der Anteil der einführenden Programmierveranstaltungen SE1 und SE2 in unseren Curricula schematisch wie in Abbildung 1 dar. Eine Zelle des unterliegenden Rasters entspricht drei Leistungspunkten, beide Module haben demnach einen Umfang von sechs Leistungspunkten. Formal sind sie jeweils mit 2 SWS Vorlesung und 2 SWS Übung ausgewiesen.

2.3 Algorithmen vs. Interaktive Systeme

Einen imperativen Einstieg und eine Aufteilung der Sprachkonzepte von Java auf mehr als ein Semester vorausgesetzt, gibt es weitere Entwurfsmöglichkeiten, die ebenfalls die strategische Ausrichtung mehrerer Module betreffen.

Ein Ansatz: Die Programmierausbildung legt einen Schwerpunkt auf *Algorithmen und Datenstrukturen* (A&D), die vor allem dazu befähigen soll, beliebig große Datenmengen so effizient wie möglich verarbeiten zu lassen.

Dieser klassische Ansatz setzt einen Schwerpunkt, der vor allem für die Systemprogrammierung nützlich ist, beispielsweise im Umfeld von Datenbanken und Betriebssystemen und bei der Entwicklung von Bibliotheken und Frameworks für Server-Systeme. Aber auch in Anwendungsfeldern wie beispielsweise der Klimasimulation oder der Modellierung und Verarbeitung von Proteinstrukturen fallen große Datenmengen an und erfordern vertieftes Wissen zu A&D.

Alternative: Es wird ein früher Fokus auf *Schnittstellen* gelegt, sowohl auf grafische Benutzungsschnittstellen als auch auf den Entwurf von Klassenschnittstellen, die von Details einer Implementierung abstrahieren (Kapselung, Modularisierung).

Die meisten Anwendungssysteme, mit denen Nicht-Informatiker heutzutage in Kontakt kommen, sind *interaktive Systeme*: Egal, ob Rich-Client auf dem Desktop, Thin-Client für das Web oder mobile Anwendung für ein Smartphone: die Prinzipien reaktiver Systeme sind bei allen grundlegend relevant. Gut entworfene interaktive Systeme haben spezifische Eigenschaften (u.a. Unterscheidung von *Model, View und Controller*, saubere Trennung fachlicher und technischer Klassen, gute Modularisierung), die eine frühe Thematisierung von Schnittstellen in der Programmierausbildung nahe legen.

Wir adressieren in der Pflichtveranstaltung SE2 explizit die Konstruktion interaktiver Anwendungen, während die Studierenden ihre in SE1 erworbenen Kenntnisse zu Algorithmen und Datenstrukturen bei entsprechendem Interesse in einem Wahlpflichtmodul vertiefen können.

Konsequenzen: Ein früher Fokus auf die Programmierung reaktiver Systeme erfordert, dass Entwurfsmuster früh in die Ausbildung einbezogen werden.

In SE2 thematisieren wir explizit auch die Prinzipien von grafischen Benutzungsoberflächen (am Beispiel von Swing). Dazu führen wir auch die Konzepte von Entwurfsmustern ein, indem wir exemplarisch zentrale Muster wie das Beobachtermuster thematisieren und implementieren lassen. Wir haben dabei explizit nicht den Anspruch, den Katalog der *Gang of Four* (Gamma et al., 1995) vollständig abzudecken. Ein schrittweises Einführen einzelner Muster an geeigneten Stellen innerhalb der Programmierausbildung scheint uns erfolgversprechender.

3 Organisatorische Alternativen

Nach den möglichen Alternativen bei der modulübergreifenden Gestaltung von Programmierveranstaltungen werden in diesem und dem folgenden Abschnitt Alternativen diskutiert, die sich innerhalb eines Moduls ergeben: in diesem Abschnitt organisatorische, im folgenden inhaltliche Alternativen.

Aus organisatorischer Sicht sind sowohl bei den *Vorlesungen* als auch den *Übungen* einführender Programmierveranstaltungen alternative Herangehensweisen möglich.

3.1 Vorlesungen: Folienfilme versus Live-Programmierung

Aus didaktischer Sicht sind Vorlesungen eine wenig geeignete Lehrform, unter anderem, weil frontale Wissensvermittlung die Lernenden zu wenig einbezieht. Aus Ressourcensicht hingegen sind sie die kostengünstigste Form der Weitergabe von Wissen, denn die Zahl der Teilnehmer kann fast beliebig skaliert werden (eine entsprechende Hörsaalausstattung vorausgesetzt). Da aus letzterem Grund mit einer Abschaffung der Veranstaltungsform „Vorlesung" in näherer Zukunft nicht zu rechnen ist, soll hier ihr Optimierungspotenzial für *Programmierveranstaltungen* diskutiert werden.

Ein Ansatz: Der Dozent zeigt in einer anderthalbstündigen Vorlesung je nach Folienstil ca. 20 bis 100 Folien, die er mehr oder weniger frei interpretiert vorträgt und gegebenenfalls mit Anekdoten würzt.

Programmieren ist ein komplexes Zusammenspiel von statischen Konzepten zur *Übersetzungszeit* und dynamischen Konzepten zur *Laufzeit*. Da Folien eher statisch sind, sind sie nicht gut geeignet, die dynamischen Aspekte der Programmierung aufzuzeigen. Animierte Folien können diese Schwäche teilweise mildern, sind aber in ihrem Ablauf meist ebenfalls starr festgelegt.

Alternative: Die Dozentin setzt neben ihren Folien auch eine *Entwicklungsumgebung* (hier englisch abgekürzt mit IDE) für die behandelte Programmiersprache ein, die sie live in der Vorlesung benutzt. Idealerweise gibt es keinen Vorlesungstermin, in dem nicht live programmiert wird.

Ein entscheidender Vorteil einer für alle sichtbaren Programmierumgebung ist, dass Fragen aus der Zuhörerschaft direkt mit lauffähigem Code beantwortet werden können. Dies erhöht den möglichen Interaktionsanteil einer Vorlesung erheblich. Wenn die Studierenden häufiger erfahren haben, dass ihre Fragen bei Bedarf direkt per Live-Programmierung beantwortet werden, bekommen sie mehr Mut für Verständnisfragen. Das Konzept des *Show Programming* wird in (Schmolitzky, 2007) als ein *Pedagogical Pattern* beschrieben.

Konsequenzen: Der Dozent sollte die Sprache und die IDE gut beherrschen, muss aber nicht perfekt darin sein. Die IDE muss für den Einsatz in Vorträgen geeignet sein; insbesondere sollte sie sowohl die statischen (Programmtext) als auch die dynamischen Aspekte (Ausführung) der Programmierung visualisieren.

Wir setzen seit über zehn Jahren *BlueJ* (BlueJ) in der Lehre ein, eine IDE, die explizit zum Lernen (und Lehren) objektorientierter Programmierung entwickelt wurde (Kölling et al., 2003). BlueJ bietet eine gute Visualisierung der statischen Konzepte der Programmierung: ein *Klassendiagramm* des aktuellen Paketes wird automatisch generiert (siehe Abbildung 2), der Editor hebt Sichtbarkeitsbereiche im Quelltext farbig hervor. Außerdem ist BlueJ *hochgradig interaktiv*, von jeder beliebigen Klasse in einem Java-System kann interaktiv ein Exemplar erzeugt (siehe die rot dargestellten Objekte in der Objektleiste, unten in Abbildung 2) und seine Methoden interaktiv aufgerufen werden. Der Debugger erlaubt auch die Visualisierung von Programmausführungen, beispielsweise der Stackframes bei Rekursion. All diese Aspekte machen BlueJ auch für den Einsatz in Vorlesungen sehr gut geeignet. In Bezug auf die *Visualisierung von Objektinteraktion* ist BlueJ allerdings eher schwach[3].

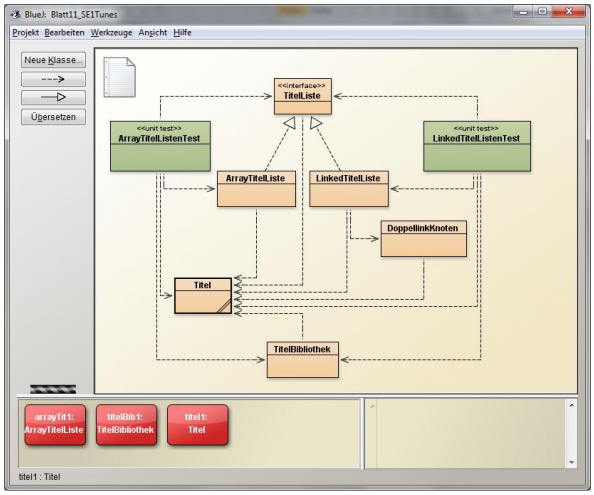


Abb. 2: Die Oberfläche von BlueJ an einem Beispiel

Wir setzen BlueJ in SE1 durchgängig in der Vorlesung und in den Übungen ein, für SE2 wechseln wir jedoch in den Übungen auf *Eclipse* als Entwicklungsumgebung. Aber auch in SE2 kann BlueJ bei der Visualisierung objektorientierter Entwürfe eine gute Hilfe sein und wird deshalb punktuell auch in der Vorlesung eingesetzt.

3.2 Übungen: Heimarbeit versus Präsenz

Übungen sind gerade bei Programmierveranstaltungen wie das Salz in der Suppe: letztere wären ohne erstere fade und langweilig und somit letztlich nicht effektiv. Entsprechend gibt es vermutlich nur sehr wenige Module, die ausschließlich mit Vorlesungen versuchen, den Teilnehmern das Programmieren beizubringen. Allerdings gibt es auch

[3] Vom gleichen Entwicklerteam wurde daher *Greenfoot* entwickelt, ebenfalls eine Java-IDE, die diesen Aspekt deutlich besser adressiert, aber eher auf die Schulinformatik zielt.

bei der Gestaltung von Programmierübungen unterschiedliche Möglichkeiten der Umsetzung.

Ein Ansatz: Jede Woche gibt es ein *Aufgabenblatt*, das *eigenständig* bearbeitet werden soll; dieses Blatt wird in der Ausgabewoche in einem Übungstermin vorab besprochen, um Fragen zu klären und eventuell auch Hinweise auf Lösungsmöglichkeiten zu geben; in der Folgewoche müssen die Studierenden ihre Bearbeitungen des Aufgabenblattes bei ihrem Tutor abgeben, der üblicherweise eine weitere Woche braucht, um diese Bearbeitungen zu korrigieren.

Dieses Muster ist typisch (nicht nur) für Programmierveranstaltungen, es ist aber in vielerlei Hinsicht unbefriedigend, siehe auch (Altenbernd-Giani et al., 2009): Erstens bewirkt es einen sehr großen Abstand von der Aufgabenstellung bis zum Feedback, das bei den Lernenden ankommt (mindestens zwei Wochen). Zweitens ist das Feedback meist nur sehr dünn: im schlechtesten Fall besteht es aus der erzielten Punktzahl, im besten Fall aus einem Text, der nicht nur die Bewertung erläutert, sondern Hinweise auf Verbesserungsmöglichkeiten gibt. Letztere sind insbesondere bei Programmierlösungen häufig sinnvoll oder sogar notwendig. Drittens ist es aus Kapazitätsgründen meist nicht möglich, dass ein Tutor von jedem Studierenden eine Einzelabgabe erhält; deshalb kommt es zu so genannten Arbeitsgruppen von zwei bis vier Personen, die eine gemeinsame Bearbeitung abgeben. Häufig sieht diese leider so aus, dass ein Student die Bearbeitung vornimmt und ein oder zwei weitere Studierende lediglich ihren Namen mit auf das Blatt setzen.

Alternative: Die Studierenden bekommen spezielle *Präsenzaufgaben*, die sie in Laborräumen der Hochschule unter Betreuung in Paaren bearbeiten und auch direkt am Rechner durch Betreuer (wissenschaftliche Mitarbeiter und studentische Hilfskräfte aus höheren Semestern) abnehmen lassen. Die Betreuer können den Studierenden unmittelbares und persönliches Feedback geben und insbesondere bei Problemen sehr viel schneller helfen.

In unseren beiden einführenden Modulen sind die Laboreinheiten drei echte Zeitstunden lang, um genügend Zeit für die Interaktion mit den Studierenden zu ermöglichen. Die Studierenden müssen dabei im Extremfall die Lösungen nicht unbedingt selbst erstellt haben, solange sie bei der Abnahme die Lösung gut darstellen können. Denn in den Abnahmen besteht durch die direkte Kommunikation ausreichend Gelegenheit, das Verständnis der Studierenden für die zu vermittelnden Konzepte und Begriffe zu überprüfen.

Konsequenzen: Ein Präsenzbetrieb bei den Übungen hat mehrere Konsequenzen:

- Die Präsenzaufgaben für einen betreuten Laborbetrieb müssen anders konzipiert werden als Aufgaben für die Heimarbeit.
- Stärkere Studierende (mit Vorerfahrung) können ihren Vorsprung nutzbringend einsetzen, indem sie *im Paar* mit Programmieranfängern zusammenarbeiten.
- Die Betreuer, insbesondere die wissenschaftlichen Mitarbeiter, müssen bereit sein, diese Art des Übungsbetriebs mitzutragen.

Präsenzaufgaben: Mehrere Jahre Erfahrung mit diesem Konzept haben bei uns dazu geführt, dass die Aufgaben eng geführt und kleinschrittig sind. Dies liegt auch daran, dass wir aufgrund der hohen Teilnehmerzahl gezwungen sind, in großem Maße studentische Hilfskräfte für die Betreuung einzusetzen. Diese sind üblicherweise nicht erfahren in der Betreuung von Kommilitonen und brauchen entsprechend detaillierte Hinweise, wie die Aufgaben abzunehmen sind. Kleinschrittige Aufgaben für Präsenzaufgaben sind aber auch deshalb zu bevorzugen, weil die Betreuer dann schneller und häufiger mit den Studierenden in Interaktion treten, und so die Gefahr verringert wird, dass die Studierenden sich in Fragen zu unwichtigen Randdetails verrennen und unnötig Zeit verlieren.

Das *Programmieren im Paar* wurde durch agile Methoden wie das *Extreme Programming* (Beck and Andres, 2004) breiter bekannt. Der Nutzen in der professionellen Softwareentwicklung ist durchaus umstritten, in der Programmierausbildung gibt es jedoch deutliche Hinweise auf seine Wirksamkeit (McDowell et al., 2006). Wir sehen einen weiteren Vorteil in der Möglichkeit, Studierende mit Vorerfahrung bewusst mit Anfängern in den Paaren zu mischen, damit das vorhandene Vorwissen zur Unterstützung von Kommilitonen genutzt werden kann.

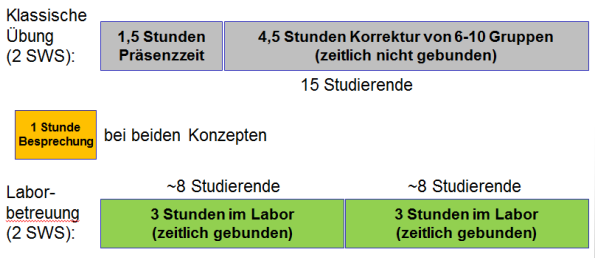


Abb. 3: Betreueraufwand pro Woche im Vergleich

Für die *Betreuer* ergibt sich eine andere Zeiteinteilung für zwei SWS Übung, siehe Abbildung 3: statt 90 Minuten klassische Übung mit einer Gruppe von 15 Teilnehmern (die momentane Übungsgruppengröße in der Informatik an der Uni Hamburg) und der Zeit, die für das Korrigieren der Bearbeitungen anfällt (nach Erfahrungswerten ca. vier bis fünf Stunden pro Woche), verbringt ein Betreuer diese

Zeit in *zwei* Laborterminen von je drei Zeitstunden. In jedem Termin hat er demnach rechnerisch Kontakt mit sieben bis acht Studierenden, also maximal vier Paaren. Insgesamt ergibt dieses Konzept *mehr direkte Kontaktzeit* zwischen Studierenden und Betreuern und ermöglicht somit einen intensiveren Austausch, der insbesondere bei Programmieraufgaben hilfreich ist.

Dieser intensivere Austausch ist allerdings auch kraftraubender; außerdem kann ein Betreuer sich seine Zeit nicht mehr frei einteilen (während die Korrektur von Übungsblättern auch zuhause bei einer Tasse Tee möglich ist). Es gab durchaus wissenschaftliche Mitarbeiter, die das Konzept im Prinzip gut geheißen haben, es aber aus persönlichen Gründen abgelehnt haben.

4 Inhaltliche Alternativen

Wenn wir bis zu diesem Punkt der Diskussion jeweils den vorgeschlagenen Alternativen gefolgt sind, dann betrachten wir nun einen Einstieg in die objektorientierte Programmierung interaktiver Systeme mit Java, der innerhalb von mindestens zwei Semestern vorgenommen wird, die Mechanismen von Java nicht vollständig im ersten Semester vermittelt und bei dem der Dozent viel live in den Vorlesungen programmiert. Die Übungen werden als intensiv betreute Präsenzübungen mit Paararbeit durchgeführt, in denen die Studierenden auf verschiedenen Ebenen (innerhalb des Paares, durch die Betreuer, durch Präsentationen vor der Gruppe) unmittelbares Feedback zu ihrem Lernerfolg erhalten.

Auch bei der inhaltlichen Gestaltung einer solchen zweisemestrigen Veranstaltung gibt es einige interessante Alternativen. Zwei davon sollen hier diskutiert werden: *Sammlungen vor Arrays* und *Interfaces vor Vererbung*.

4.1 Smarte Sammlungen vor Arrays

Sammlungen von Objekten spielen in der Programmierung eine zentrale Rolle. In Java gibt es zwei grundsätzliche Unterstützungen für Sammlungen: den Sprachmechanismus für *Arrays* und die Bibliotheksklassen und Interfaces des *Java Collections Framework* (JCF). Beide sollten in der Ausbildung gründlich berücksichtigt werden; eine mögliche Frage dabei ist: in welcher Reihenfolge?

Ein Ansatz: Arrays werden relativ früh vorgestellt, das JCF deutlich später.

In fast allen Lehrbüchern zu Java wird diese Reihenfolge praktiziert (siehe beispielsweise (Barnes and Kölling, 2008; Schiedermeier, 2010)) oder das JCF sogar ausgelassen, siehe (Bell and Parr, 2003). Arrays sind in Java ein expliziter Sprachmechanismus, der durch eine eigene Syntax unterstützt wird und somit gefühlt zum Kern der Sprache gehört. Das JCF hingegen setzt in seinen Implementationen auf den Arrays von Java auf und erfordert für eine sinnvolle Nutzung Kenntnisse von den *Interfaces* in Java, die üblicherweise ebenfalls spät thematisiert werden. Ein weiteres Argument für eine späte Behandlung könnte sein, dass das JCF umfangreichen Gebrauch von Generizität macht, die ebenfalls eher spät thematisiert werden kann.

Alternative: Das JCF wird deutlich vor den Arrays in Java thematisiert, Arrays werden primär als speichernahe Realisierungsunterstützung für mächtigere Sammlungskonzepte dargestellt.

Die Sammlungsarten des JCF (u.a. *Set*, *List und Map*) sind auf einem höheren Abstraktionsniveau angesiedelt als die (für viele Probleme zu) speichernah konzipierten Arrays. Somit sind sie für viele Aufgabenstellungen die deutlich bessere Wahl und ermöglichen elegantere Lösungen. Entsprechend sollten sie als nützliche Hilfsmittel so prominent in ihrer typischen Nutzung vermittelt werden, dass die Studierenden instinktiv möglichst zuerst zu diesen Sammlungen greifen. Diskutiert wird dieser Ansatz unter anderem von Ventura et al. in (Ventura et al., 2004). Koenig und Moo argumentieren in (Koenig and Moo, 2000) ganz analog für eine Behandlung der STL deutlich vor den Arrays von C++. Wir haben in einer anderen einführenden Programmierveranstaltung mit C++ ebenfalls gute Erfahrungen mit diesem Ansatz gemacht.

Konsequenzen: Eine mögliche Konsequenz für eine Java-Veranstaltung könnte sein, Generizität noch vor dem JCF behandeln zu müssen, siehe obige Anmerkung. Generizität ist vereinfacht gesagt das Parametrisieren von Typen mit Typen, im Falle von Sammlungen eben das Parametrisieren von Sammlungen mit ihren Elementtypen. Tatsächlich ist eine vollständige Behandlung dieses Themas nicht nötig, wenn der benötigte Teilaspekt intuitiv verständlich ist: dass für eine Sammlung von Objekten der Typ der Elemente auch deklariert werden muss.

In SE1 lassen wir die Teilnehmer in der zweiten Hälfte des Semesters kleinere Probleme mit Hilfe des JCF lösen, ohne vorab explizit die Generizität von Java einzuführen. In den bisherigen sieben Durchführungen des Moduls ist es dabei nicht zu Problemen gekommen.

Interfaces hingegen sollten für eine kompetente Nutzung des JCF bereits bekannt sein. Dies leitet über zur nächsten inhaltlichen Alternative.

4.2 Interfaces vor Vererbung

Interfaces sind ein Sprachmechanismus in Java, der es ermöglicht *Schnittstellen* explizit zu beschreiben[4].

Schnittstellen spielen eine zentrale Rolle in der objektorientierten Programmierung und Modellierung, siehe u.a. die zentrale Aussage „*Program to an interface, not an implementation*" im Buch der *Gang of Four* (Gamma et al., 1995). Aber Schnittstellen sind nicht nur für die Objektorientierung zentral, sondern vielmehr ein Schlüsselkonzept bei der Modularisierung großer Softwaresysteme. Weiterhin erlauben sie eine Diskussion darüber, dass eine *Spezifikation* lediglich einen Teil der zu erbringenden Dienstleistungen beschreiben kann[5].

Wenn ein expliziter Sprachmechanismus zur Beschreibung von Schnittstellen zur Verfügung steht, sollte dieser intensiv in einer softwaretechnisch orientierten Programmierausbildung genutzt werden. Aber wann kann dieser Mechanismus eingeführt werden?

Ein Ansatz: Interfaces werden nach *Vererbung* und *abstrakten Klassen* thematisiert.

Diese Reihenfolge, die in praktisch allen Java-Lehrbüchern verfolgt wird, orientiert sich an der historischen Entwicklung von Interfaces: In den vor Java entworfenen objektorientierten Sprachen (wie beispielsweise C++) müssen reine Schnittstellen mit vollständig abstrakten Klassen beschrieben werden.

Die Argumentation lautet verkürzt: Klassen können von anderen Klassen erben; wenn nicht alle Methoden in der Superklasse implementiert sind, wird diese zu einer abstrakten Klasse; wenn alle Methoden nicht implementiert (nur deklariert) sind, wird eine abstrakte Klasse zu einem Interface. Diese Argumentation scheint zwingend zu erfordern, für das Einführen der Interfaces in Java den Umweg über Vererbung und abstrakte Klassen gehen zu müssen.

Alternative: Interfaces werden zur expliziten Modellierung von Schnittstellen sehr früh eingeführt, während das komplexe Thema Vererbung deutlich später thematisiert wird (Schmolitzky, 2004; Schmolitzky, 2006).

Die Interfaces von Java können benutzt werden, um Schnittstellen explizit zu beschreiben. Insbesondere kann demonstriert werden, dass eine Schnittstelle auf verschiedene Weisen implementiert werden kann.

In unserer Programmierausbildung stellen wir Interfaces bereits in der Mitte des ersten Semesters vor, Vererbung (unterschieden in *Typ- und Implementationsvererbung*, siehe (Schmolitzky, 2006)) hingegen erst im zweiten Semester. In der zweiten Hälfte des ersten Semesters wird unter anderem ein (nicht generisches) Interface für eine fachliche Liste vorgestellt, das die Studierenden in den Übungen auf zwei verschiedene Arten implementieren: als verkettete Liste und als Array-Liste. Auf diese Weise kann diskutiert werden, dass Klienten-Code vollständig von der implementierenden Struktur entkoppelt werden kann und beide Implementationsformen das Interface *korrekt* implementieren können. Ihre Unterschiede wirken sich lediglich bei der *Effizienz* bestimmter Benutzungsprofile aus (wie „häufiges Einfügen am Anfang der Liste" etc.), ein Aspekt, der aus Sicht der Korrektheit nachrangig ist. Auf diese Weise kann u.a. verdeutlicht werden, warum es im JCF verschiedene Implementationen des Interfaces *List* gibt.

Konsequenzen: Da Interfaces in Java ein recht abstraktes Konzept sind, sollten möglichst viele Aufgaben nach der Einführung von Interfaces gezielt ihre Vorteile demonstrieren. Dabei muss darauf geachtet werden, dass es sich nicht um Aufgaben handelt, die eher den Einsatz von Implementationsvererbung nahe legen.

5 Diskussion

Eine Motivation für diesen Artikel entstand aus dem Umstand, dass wir unsere eigene Programmierausbildung an verschiedenen Stellen zum Gegenstand von Untersuchungen gemacht haben und immer wieder feststellen mussten, dass wir in etlichen Punkten vom „Mainstream" abweichen. Einzelheiten dieser Abweichungen haben wir an verschiedenen Stellen veröffentlicht, aber das Gesamtbild bisher nicht.

Die beiden Veranstaltungen SE1 und SE2 haben wir nach den hier beschriebenen Konzepten entworfen und führen sie seit dem Wintersemester 2005/06 jedes Jahr durch. In den bisherigen sieben Durchläufen haben wir sehr viel über einführende Programmierveranstaltungen gelernt und diese Erfahrungen auch in ihre Überarbeitung einfließen lassen. Beide Module werden in der studentischen Lehreevaluation regelmäßig sehr positiv bewertet, obwohl sie Pflichtmodule sind. Das Gesamtkonzept von SE1, mitsamt dem hier nicht diskutierten „*Objects First*"-Vorgehen (Barnes and Kölling, 2008), wurde von den Studierenden des Jahrganges 2007/08 für den Hamburger Lehrpreis vorgeschlagen und erreichte den zweiten Platz.

Der Titel dieses Beitrags ist bewusst mit einem Fragezeichen formuliert, das unterschiedlich (wie von einem der Gutachter angemerkt) gedeutet

[4] Im Deutschen können wir das Konzept einer Schnittstelle gut von dem Sprachmechanismus unterscheiden, indem wir den Mechanismus mit dem englischen Begriff benennen, der auch namengebend für das entsprechende Java-Schlüsselwort ist.

[5] Eine Spezifikation, die so vollständig ist, dass eine Implementation maschinell abgeleitet werden kann, verdient ihren Namen nicht.

werden kann: Ist die hier beschriebene Programmierausbildung tatsächlich eine softwaretechnische? Oder: Was heißt eigentlich „softwaretechnische Programmierausbildung“? Oder: Wollen wir überhaupt eine softwaretechnisch geprägte Programmierausbildung? Alle drei Deutungen sind sinnvoll und sollten zukünftig ausführlicher diskutiert werden; dies würde jedoch den Rahmen dieses Beitrags sprengen, der primär als Erfahrungsbericht fungieren soll.

Eine softwaretechnische Programmierausbildung muss vor allem *effektiv* in Hinsicht auf nachfolgende Module sein: Sie muss unter anderem dazu führen, dass die Teilnehmer anschließend kompetent Fragen des Software Engineering diskutieren können.

Als zentrales Mittel, die Effektivität zu erhöhen, sehen wir unsere Präsenzübungen an. Betrachtet man die reine Kontaktzeit zwischen einem Betreuern und seinen Studierenden, führt das Konzept zu ihrer Vervierfachung: 90 Minuten klassischer Übung stehen zweimal drei Stunden im Labor gegenüber. Unabhängig von der vorgeschriebenen Gruppengröße (15, 20 oder 25 Personen, je nach Hochschule) erhöht dies erheblich die Möglichkeit zu *individuellem Feedback* und zur *direkten Kommunikation* über den Stoff, bei *gleichem Ressourceneinsatz* der Hochschule (gerechnet in SWS-Lehrkapazität). Diese Vervielfachung der Kontaktzeit muss jedoch begleitet werden von geeigneten Aufgaben. Wir haben einige Jahre gebraucht, um unsere Aufgaben in dieser Hinsicht zu optimieren.

Auch das *Programmieren Lernen im Paar* hat sich als sehr effektive Möglichkeit erwiesen, die *Kommunikation* über kleinste Entwurfsentscheidungen, die beim Programmieren getroffen werden müssen, zu erhöhen. Die Studierenden der bisherigen Jahrgänge haben auch dieses Konzept mit großer Mehrheit sehr positiv aufgenommen.

6 Zusammenfassung

In diesem Artikel wurden systematisch einige Alternativen diskutiert, die bei der Gestaltung der einführenden Programmierausbildung in Informatik-Studiengängen auftreten können. Dabei wurde über Fragen der strategischen Ausrichtung ganzer Studiengänge der Bogen geschlagen hin zu inhaltlichen Alternativen bei der einführenden Java-Ausbildung.

Wir erhoffen uns, dass dieser Artikel einen Beitrag zur Diskussion der hier behandelten Themen auf der SEUH leisten kann.

Literatur

Altenbernd-Giani, E., et al. (2009): Programmierungsveranstaltung unter der Lupe. Lernen im digitalen Zeitalter, Die 7. eLearning-Fachtagung Informatik (DeLFI), Berlin, Lecture Notes in Informatics (LNI) P-153, S. 55-66.

Barnes, D. and M. Kölling (2008): Objects First with Java - A Practical Introduction Using BlueJ (4th Edition). UK, Pearson Education.

Beck, K. and C. Andres (2004): Extreme Programming Explained - Embrace Change (2nd Ed.), Addison-Wesley.

Bell, D. and M. Parr (2003): Java für Studenten - Grundlagen der Programmierung, Prentice Hall.

BlueJ. "BlueJ - The Interactive Java Environment." http://www.bluej.org.

Gamma, E., et al. (1995): Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, Addison-Wesley.

GI (2000): Standards zur Akkreditierung von Studiengängen der Informatik und interdisziplinären Informatik-Studiengängen an deutschen Hochschulen, Gesellschaft für Informatik e.V.

GI (2005): Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen, Neuauflage der GI-Standards zur Akkreditierung von Informatik-Studiengängen aus dem Jahr 2000, Gesellschaft für Informatik e.V.

Gosling, J., et al. (2005): The Java Language Specification (3rd Ed.). Addison-Wesley Longman, Amsterdam, Addison-Wesley.

Koenig, A. and B. Moo (2000): Rethinking How to Teach C++, Part 1: Goals and Principles. Journal of Object-oriented Programming 13(7), S. 44-47.

Kölling, M., et al. (2003): The BlueJ system and its pedagogy. Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology 13(4), S. 249-268.

Ludewig, J. (2010): Software-Ingenieur werden. Informatik Spektrum 33(3), S. 288-291.

McDowell, C., et al. (2006): Pair programming improves student retention, confidence, and program quality. Commun. ACM 49(8), S. 90-95.

Schiedermeier, R. (2010): Programmieren mit Java, Pearson Studium.

Schmolitzky, A. (2004): "Objects First, Interfaces Next" or Interfaces Before Inheritance. OOPSLA '04 (Companion: Educators' Symposium), Vancouver, BC, Canada, ACM Press.

Schmolitzky, A. (2006): Teaching Inheritance Concepts with Java. Principles and Practices of

Programming in Java (PPPJ), Mannheim, Germany, ACM Press, S. 203-207.

Schmolitzky, A. (2007): Patterns for Teaching Software in Classroom. EuroPLoP 2007, Irsee, Germany, UVK Konstanz.

Ventura, P., et al. (2004): Ancestor worship in {CS1}: on the primacy of arrays. OOPSLA '04 (Companion: Educators' Symposium), Vancouver, BC, Canada, ACM Press.

Walker, H. M. (2010): The role of programming in introductory computing courses. ACM Inroads 1(2), S. 12-15.

Wegner, P. (1987): Dimensions of Object-Based Language Design. OOPSLA '87, Orlando, Florida, ACM SIGPLAN Notices Vol. 22(12).

Programmiergrundausbildung: Erfahrungen von drei Hochschulen

Stephan Kleuker, Hochschule Osnabrück

s.kleuker@hs-osnabrueck.de

Zusammenfassung

Die Programmierausbildung bleibt ein Grundstein jedweder Informatik-Ausbildung und bildet das Fundament der meisten weiterführenden Informatik-Themen. Über die Art und Weise, wie Programmierung am besten gelehrt werden kann, wird wahrscheinlich seit Beginn dieser Lehre gestritten. Gerade Paradigmen-Wechsel haben diese Diskussion befruchtet. In diesem Artikel werden drei Erfahrungen in der Programmierausbildung in unterschiedlichen Programmiersprachen an verschiedenen Fachhochschulen zusammengefasst und mit einigen Hintergründen verknüpft. Für die Programmiersprache Java werden zusätzlich noch Varianten der didaktischen Herangehensweise diskutiert.

Einleitung

Da fast alle Bachelor-Abschlussarbeiten von Fachhochschulen in Unternehmen stattfinden und sich dabei fast immer ein Programmieranteil in dieser Arbeit befindet, ist fachliche Programmierausbildung ein elementares Ziel von Informatik-Studiengängen. Dabei soll es für angehende Absolventen keine Einschränkungen in der Wahl des Themas durch die am Anfang im Studium gewählte Programmiersprache geben. Es wird davon ausgegangen, dass innerhalb des Studiums die Fähigkeit erlangt wird, sich schnell in die Prinzipien einer vorher nicht benutzten Programmiersprache einzuarbeiten. Die eigentliche Nutzung der Sprache ist die Basis für die durchzuführenden Arbeiten und wird nebenbei erlernt.

In der Grundlagenausbildung soll dabei grundsätzlich eine Einführung in die Programmierung und nicht in eine bestimmte Programmiersprache stattfinden. Dies bedeutet, dass man aus didaktischen Gründen lieber auf Sprach-Features verzichtet, um elementare Konzepte wiederholt intensiver einüben zu können. Studierende sollten spätestens ab dem dritten Semester dann in der Lage sein, sich selbständig in neue nicht intensiv geschulte Programmiersprachen einzuarbeiten. Dies wird dadurch gefördert, dass oft in weiteren Lehrveranstaltungen weitere Programmiersprachen eingeführt werden, die für praktische Übungen unerlässlich sind. Beispiele sind JavaScript für Web-Technologien und Sprachen zur Erstellung von Triggern und Stored Procedures, wie PL/SQL und Transact SQL für Datenbanken.

Im Mittelpunkt der nachfolgenden Analyse steht dabei der Erfolg der Masse der Studierenden, also nicht die Anzahl der Studierenden, die zu sehr guten Ergebnissen kommen. Dieser Ansatz ist gerechtfertigt, da sich alle Hochschulen zwangsweise der Forderung der Politik stellen müssen, dass möglichst viele Studienanfänger in relativ kurzer Zeit zum Studienerfolg geführt werden müssen. Eine Forderung, die teilweise zu politisch diktierten Selbstauflagen von Studiengängen oder ganzen Hochschulen führt, dass 90% der Studierenden zum Studienerfolg gebracht werden. Diese Forderung wird hier als Grundlage genommen, wobei sie natürlich hinterfragt werden muss, da sich gerade in der Informatik immer wieder die Frage stellt, wieviel Prozent der Studienanfänger wirklich realistisch für das Themengebiet Informatik geeignet sind. Natürlich darf die Motivation und Förderung guter Studierender nicht vernachlässigt werden, die im konkreten Fall durch Zusatz- oder Alternativaufgaben und die Aufforderung, Studierenden mit aktuell weniger Kenntnissen zu unterstützen, für Lehrveranstaltungen in der Programmierung umgesetzt werden kann.

Motiviert ist dieser Artikel durch verschiedene Beiträge zu früheren SEUH-Konferenzen, in denen die Möglichkeiten zum Einstieg in die Objektorientierung [SZ07] und die Nutzung von Werkzeugen [Bol09] diskutiert wurden. Generell finden einige Untersuchung zum Lernverhalten von Studierenden statt, die häufiger in Magazinen wie „Computer Science Education“, wie z. B. [MFRW11] mit der Untersuchung der Denkmodelle, was bei Zuweisungen passiert, und [KS12] über Erfahrungen bei ersten Programmieraufgaben, veröffentlicht und auch in GI-Workshops, z. B. [Boe07], behandelt werden. Dieser Erfahrungsbericht will diese Forschungsinteressen unterstützen.

Im folgenden Text werden drei Einführungsveranstaltungen in die Programmierung, die mit unterschiedlichen Programmiersprachen durchgeführt wurden, auf ihren Erfolg im Zusammenhang mit der eingesetzten Programmiersprache analysiert. Da der Erfolg ohne den Aufbau eines komplexeren Messsystems nicht vollständig objektiv messbar ist, basieren Ergebnisse auf persönlichen Erfahrungen und Gesprächen mit beteiligten Studierenden und Lehrenden in ähnlichen Situationen. Es wird deshalb fast konsequent in diesem Artikel darauf verzichtet, aus Durchfallquoten in Prüfungen direkt auf den Erfolg eines didaktischen Ansatzes zu schließen. Im Kapitel zu Java werden weiterhin alternative Vorgehen zur Vermittlung und ihre Unterstützung durch verschiedene Werkzeuge betrachtet.

Die Komplexität des Aufbaus eines wirklich aussagekräftigen Messsystems, das auch Vergleiche aushält, wird auch in der Dissertation von Matthew C. Jadud [Jad06] deutlich, in der versucht wird, aus dem Verhalten, wann Studierende was kompilieren, Schlüsse über ihren Lernfortschritt zu ziehen. Die in der genannten Arbeit aufgeworfene Frage, aus welchen Indikatoren, z. B. die Messung von Zeiten, die in einer Entwicklungsumgebung benötigt wurden, die Anzahl von Compiler-Aufrufen oder die Anzahl gescheiterter zur Verfügung stehender Tests, man konkrete Schlüsse ziehen darf, stellt ein offenes spannendes Forschungsgebiet für weitere Untersuchungen dar.

Smalltalk

Smalltalk [Bra08] ist die erste kommerziell eingesetzte Sprache, die konsequent und vollständig auf Objektorientierung gesetzt hat. Alles, also auch einfache Zahlen, sind Objekte, die mit Methoden bearbeitet werden können. Die Geschichte von Smalltalk zeigt deutlich, wie ein hochinnovativer Ansatz historisch einfach völlig zu früh stattgefunden hat. Die Sprache pflegte immer das Image des besonderen und war maßgeblicher Vorreiter in der Entwicklung der Fenster-Technologien und der Maussteuerung. Leider benötigten diese Ansätze in den 1970er und 80er Jahren extrem teure Hardware, so dass Smalltalk zwar in einigen Unternehmen, wie z. B. dem Versicherungsbereich in Deutschland, in der Entwicklung eingesetzt wurde, aber nie einen echten Durchbruch schaffte. Teil dieser Problematik ist die zunächst gewöhnungsbedürftige Syntax, in der z. B. Parameter in Mixfix-Notation in den Methodennamen integriert werden. Die folgenden Befehle erzeugen ein Objekt vom Typ Dictionary, das Paare von Werten verwalten kann, für die dann ein erstes Paar eingetragen wird.

```
d := Dictionary new.
d at: 'Smalltalk' put: 'irgendwie spannend'.
```

Bemerkenswert ist, dass auch diese Schreibweise wesentliche Vorteile hat, da man so genau weiß, wozu die Parameter eingesetzt werden.

Smalltalk wurde und wird in der Programmiereinführung im Studiengang Wirtschaftsinformatik an der privaten Fachhochschule Nordakademie in Elmshorn genutzt. Die Fachhochschule bietet duale Studiengänge an, wobei Unternehmen Studierende aussuchen und deren Studiengebühren bezahlen. Die Studierenden arbeiten zwischen den Vorlesungsblöcken in den Unternehmen. Bemerkenswert ist, dass auch die Unternehmen die Auswahl von Smalltalk als Lernsprache mittragen, was damit zusammenhängt, dass in höheren Semestern konsequent Java eingesetzt wird.

Die Voraussetzungen für die Programmiereinführung unterscheiden sich damit deutlich von denen an öffentlichen Hochschulen. Die Studierenden haben sich bewusst für den dualen Studiengang entschieden, sie wissen, dass ihre Leistungen in der Hochschule auch von ihrem Unternehmen verfolgt werden und oftmals wird die spätere Weiterarbeit im jeweiligen Unternehmen angestrebt.

Fachlich bedeutet dies nach den Beobachtungen des Autors, dass das zumindest anfänglich leistungsschwächere Drittel von Studierenden, das an öffentlichen Hochschulen anfängt, an der Nordakademie kaum vertreten ist. Im oberen Bereich fehlen dafür vereinzelt sehr kreative Köpfe, die ein stark verschultes Ausbildungssystem bewusst umgehen wollen, um experimentelle Freiheiten im Studium zu schaffen.

Die eigentliche Vorlesung wurde mit dem Objects First-Ansatz gehalten, der bereits am Anfang auf die Strukturbegriffe Objekt und Klasse setzt. Generell wurden bei dieser Veranstaltung sehr positive Erfahrungen gemacht, die sich in der auf zwei Semester aufgespaltenen Veranstaltung in großen Lernfortschritten der Studierenden zeigten.

Nachdem anfängliche Schwierigkeiten mit der Syntax und der zunächst kompliziert, dann aber intuitiv nutzbaren Entwicklungsumgebung VisualWorks ausgeräumt waren, fand eine Konzentration auf die eigentliche Programmentwicklung statt.

Smalltalk nutzt den Begriff der Literale, um unveränderliche Objekte (immutable Objects) zu definieren, so kann ein String-Objekt in den meisten Smalltalk-Varianten nicht verändert werden, etwaige Methoden liefern ein neues String-Objekt als Ergebnis. Diese Unterscheidung ist aus Sicht von Anfängern nicht sehr intuitiv und führt zu Problemen. Kritisch aus Anfängersicht ist auch, dass für die Ausgabe eine Klassenmethode genutzt werden muss.

Insgesamt war die Vorlesung erfolgreich, da durch eine schnell angebotene Nachprüfungsmöglichkeit alle Studierenden letztendlich die Veranstaltung bestanden. Weiterhin war der Einstieg in die Objektorientierung erfolgreich, so dass der Übergang zu Java keine besondere Herausforderung darstellte.

Zusammenfassend kann man den Ansatz, Smalltalk als Anfängersprache zu nutzen, als klaren Erfolg ansehen, der allerdings maßgeblich durch die sehr guten Randbedingungen unterstützt wird. Die Möglichkeit, mit Smalltalk gerade an anderen Fachhochschulen anzufangen, schätzt der Autor als sehr gering an, da neben der mangelnden Akzeptanz unter den Kollegen gerade auch Studierende frühzeitig praxisrelevante Lehrstoffe vermittelt bekommen wollen. Dies ist besonders zu beachten, da viele Studierende ihr Studium nebenbei finanzieren müssen und der frühe Einstieg in einfache Programmierjobs in Unternehmen für Studierende und Unternehmen sehr vielversprechend sein kann.

Die Programmiersprache Ruby [Fla08] hat viele der zentralen Ideen übernommen und eignet sich sehr gut zur schnellen Prototyp-Entwicklung für Web-Systeme mit Ruby on Rails [RTH11]. Es wäre deshalb eine Überlegung wert, ob eine Programmiereinführung mit Ruby sinnvoll für Erstsemester sein könnte.

C

Die Programmiersprache C ist in aktuellen Untersuchungen weiterhin eine der zentralen Sprachen bei der Software-Entwicklung in Unternehmen. Da die Sprache sehr hardware-nah ist und gerade der Markt der embedded Systeme kontinuierlich vom Auto bis zum Kühlschrank wächst, wird die Sprache weiterhin mittelfristig ihre Bedeutung behalten.

C wurde 2006 in der Programmierausbildung für Erstsemester in einer Veranstaltung mit fünf Leistungspunkten an der Fachhochschule Wiesbaden (jetzt Hochschule RheinMain) genutzt. Die Rahmenbedingungen waren eine zu dem Zeitpunkt leicht sinkende Anzahl von Erstsemestern, wobei es in dieser Veranstaltung eine besonders hohe Anzahl von Wiederholern gab. C wird als gute Sprache zum Lernen der Programmierung von Befürwortern angesehen, da sie praxisnah ist und mit sehr wenigen Schlüsselwörtern auskommt, was zur Annahme führt, dass die Sprache leicht zu erlernen sei.

In der Durchführung zeigte sich deutlich, dass die notwendige Zeigerprogrammierung allein schon beim Umgang mit Strings, also char-Arrays mit besonderem Terminierungssymbol, und anderen Arrays ohne Möglichkeit, deren Länge implizit zu bestimmen, ein großes Problem darstellt. Ein Verheddern in der Zeigerwelt bei einfach und doppelt verketteten Listen ist für Anfänger oft unausweichlich. Gegenüber Java hat C allerdings den Vorteil, dass man alle Variablen auch per Referenz (`int* wert`) übergeben kann. Da so leider auch Arrays übergeben werden können, geht der intuitive Vorteil schnell verloren.

Weiterhin kann in C auch auf eine Kopiersemantik gesetzt werden, wenn z. B. Arrays in ein `struct` eingebettet sind. Dieser Ansatz wird zwar zunächst als intuitiver von Studierenden angesehen, führt aber schnell zu Problemen, wenn eine Verknüpfung mit der Referenzsemantik notwendig wird. Generell sollte deshalb auf die Kopiersemantik in der Grundausbildung in C verzichtet werden.

Gerade in C ist die Verlockung sehr groß, für Anfänger überflüssige, aus Sicht eines begeisterten C-Nutzers aber sehr interessante Sprachkonzepte wie Funktionszeiger und Makros durchzunehmen, was verpflichtend durchgeführt, für viele Studierende im ersten Semester zu großen Verständnisproblemen führt.

Eine verbreitete besondere Lernumgebung für C existiert nicht. Mit der SDL-Library [@SDL] besteht zwar die Möglichkeit, sehr schnell Spiele zu entwickeln, aber die Lernkurve ist beim Einstieg recht hoch.

Die hier dokumentierte Lehrveranstaltung wurde von vielen verschiedenen Lehrenden hintereinander gehalten, wobei die plumpe Aussage „Beim ersten Mal hält ein Dozent eine Lehrveranstaltung hauptsächlich für sich selbst“ einen Wahrheitswert enthält, da didaktische Erfahrungen aufgebaut werden müssen. Gerade die Beobachtung, dass der Übergang vom Lesen eines Programms zur Erstellung eines Programms selbst mit einfachen if-Anweisungen, ein enormer Schritt sein kann oder dass die didaktische Hürde zwischen einer einfachen Schleife und geschachtelten Schleifen sehr groß ist, muss oft in Selbsterfahrungen erworben werden.

Als Fazit eignet sich C wohl dann als Anfängersprache, wenn viele sprachliche Besonderheiten, wie die Kopiersemantik, Zeiger-Arithmetik und Makros, nicht behandelt werden. Ob solch ein konsequenter Ansatz schon umgesetzt wurde, ist leider unbekannt. Weiterhin zeigen Erfahrungen in höheren Semestern deutlich, dass zumindest bei aktuell nicht sehr leistungsstarken Studierenden die Ideen der Objektorientierung durch eine spätere Einführung nicht verinnerlicht werden.

Die ursprünglich mit C, C++ und Java dreisemestrige Programmierausbildung mit je fünf Leistungspunkten wurde in der Reakkreditierung des Bachelors nach sehr kontroversen Diskussionen durch eine zweisemestrige Programmierausbil-

dung mit jeweils 10 Leistungspunkten mit Schwerpunkt in Java, ergänzt um eine kompakte Einführung in C, ersetzt [HSRM10].

Java

Java verbreitet sich seit seiner Einführung Mitte der 1990er Jahre kontinuierlich in Unternehmen und in der Hochschulausbildung. In diversen Statistiken ist Java die am meisten benutzte Programmiersprache, wenn es um Programme im Business-Bereich, wie ERP geht.

Java ist aus Sicht der Objektorientierung eine hybride Sprache, deren Großteil aus der konsequenten Nutzung von Klassen besteht, die allerdings auf einfachen Basistypen aus C, wie `int` und `char` basieren. Diese Problematik spiegelt sich in verschiedenen didaktischen Ansätzen wider. Der klassische Ansatz besteht darin, zunächst den imperativen Teil der Programmiersprache zu lehren, die auf `if` und `while` fokussieren, und danach zur Objektorientierung überzugehen. Dieser Ansatz wurde lange Zeit auch in imperativen Sprachen, wie Basic, Pascal und C verfolgt, wo der zweite Teil sich auf die Entwicklung komplexerer Datenstrukturen meist zusammen mit ihrer Verlinkung in Listen fokussiert. Didaktisch steht man dann am Anfang vor dem Problem, dass man bei `public static void main(String[] s)` mit „public“, „static“ und „String[]“ drei Sprachanteile vorstellen muss, deren genauer Sinn erst viel später in der Lehrveranstaltung vermittelt werden kann. Gerade dieses „magische Verhalten“ führt aber bei Programmieranfängern oft zur verständlichen Idee, dass diese Magie auch für andere Sprachanteile gilt. Ein Beispiel ist der Konstruktor

```
public Punkt(int x, int y){
}
```

bei dem vermutet wird, dass eine Zuweisung zur Objektvariablen gleichen Namens (`this.x=x`) nicht programmiert werden muss.

Der klassische didaktische Ansatz ist anfänglich durchaus erfolgreich, führt aber beim Übergang zu Klassen oft zu Problemen, da diese dann analog zum `struct` in C als reine Daten-Container ohne eigene Funktionalität angesehen werden. Weiterhin setzt sich bei durchschnittlich begabten Studierenden bei frei gestaltbaren Programmieraufgaben in höheren Semestern eher der Ansatz durch, alles möglichst mit Klassenmethoden lösen zu wollen.

Einen ersten Bruch mit der imperativen Herangehensweise gibt es spätestens, wenn Strings oder Arrays genutzt werden sollen, da es sich hier um Objekte handelt, auf denen Methoden genutzt werden können. Arrays, deren Umsetzung in Java aus didaktischer Sicht sicherlich diskutabel ist, werden dann zum Problem, wenn sie an Funktionen automatisch per Referenz übergeben werden.

Unabhängig vom didaktischen Ansatz bietet Java als hybride Sprache eine weitere Falle für Anfänger mit dem Übergang vom elementaren Datentypen zu seiner kapselnden Klasse, z. B. von `int` zur Klasse `java.lang.Integer`, die z. B. bei der Nutzung von Listen benötigt wird. Zwar findet im Wesentlichen die Umwandlung durch Autoboxing automatisch statt, wird aber durch null-Werte und die Festlegung, dass die Integer-Werte von -128 bis 127 identisch, genauer Singletons, sind, die anderen nicht, bemerkenswert kompliziert. Das folgende Programm zeigt diese Probleme.

```
public static void main(String[] s){
  ArrayList<Integer> l = new ArrayList<>();
  l.add(10);
  l.add(1000);
  Integer vgl1 = 10;
  Integer vgl2 = 1000;
  for(Integer val:l){
    if (val == vgl1){
      System.out.println("val ist 10");
    }
    if (val == vgl2){
      System.out.println("val ist 1000");
    }
  }
  for(int val:l){
    if (val == vgl1){
      System.out.println("val ist 10");
    }
    if (val == vgl2){
      System.out.println("val ist 1000");
    }
  }
  l.add(null);
  Integer valn = l.get(2);
  System.out.println("valn = "+valn);
  int vali = l.get(2);
  System.out.println("vali = "+vali);
}
```

Die Ausgabe liefert

```
val ist 10
val ist 10
val ist 1000
valn = null
Exception in thread "main" java.lang.NullPointerException
```

ein Ergebnis, dass mit reiner Logik ohne zusätzliches Wissen kaum verständlich ist. Es ist möglich, dieses Problem im Wesentlichen zu vermeiden, indem konsequent nur die dazugehörigen Klassen wie `Integer` und `Double` für Variablen genutzt werden. Ein Ansatz, der vom Autor erfolgreich genutzt wurde. Leider kann man nicht vollständig in der Welt der Objekte und Klassen arbeiten, da die Java-

Klassenbibliothek viel mit int-Werten arbeitet. Kritisch ist weiterhin, dass man sehr häufig früh in Programmen viele null-Überprüfungen einbauen muss und es faktisch kein begleitendes Lehrbuch gibt, das diesen Ansatz auch verfolgt.

Java behandelt Zahlen- und String-Objekte wie für Smalltalk bereits andiskutiert als immutable Objects, was für Anfänger nicht unmittelbar verständlich ist.

Objects First

Trotz dieser Mängel wird Java oft und recht erfolgreich in der Programmiergrundausbildung eingesetzt, was auch auf die zweite didaktische Herangehensweise „Objects first" [BK09] zurück zu führen ist, bei der von Anfang an auf die Entwicklung von Objekten und Klassen gesetzt wird.

Die Grundidee besteht darin, sich möglichst früh mit den zentralen Strukturen der Objektorientierung auseinander zu setzen. Der Objektbegriff selbst ist intuitiv und Studierende finden ihn oft in ihrer realen Welt wieder, sei es selbst als Student-Objekt in der Software der Hochschulverwaltung oder als Kundin, die ein Konto-Objekt nutzt. Generell bietet sich hier die Hochschule zur Modellierung an, da Studienanfänger neu in diesem System sind, in das sie sich für drei oder mehr Jahre einarbeiten sollen. So kann z. B. der Modul-Begriff und sein Zusammenhang zur konkreten Vorlesung genauer analysiert werden.

Setzt man den Objekt-Begriff an den Anfang, spielen Begriffe wie Objekt, Objektvariable (auch Exemplarvariable, Instanzvariable oder Attribut genannt), konkrete Werte der Variablen, Objekte, die andere Objekte als Eigenschaft haben und Sammlungen von Objekten zunächst die zentrale Rolle. In Java stößt man dann schnell auf den Typ-Begriff, da jede Variable einen Typen haben muss, was entweder ein elementarer Typ aus der C-Welt oder wieder eine Klasse sein kann. Im nächsten Schritt werden Methoden und Parameter betrachtet, wobei hier der leider inkonsistente Ansatz von Kopien bei elementaren Typen und Referenzen bei Objekten zum ersten Bruch führt. Die eigentlichen Ablaufstrukturen werden dann später in den Methoden eingeführt. Man erhält so einen recht langen Anteil an Vorlesungen, die sich ausschließlich mit Objekten, Methoden und sequentiellen Programmabläufen beschäftigen, wozu man bemerkenswert viele Beispiele konstruieren kann. Dies ist auch ein gutes Beispiel, wie man Vorlesungen verlangsamen kann, da eine aus C bekannte Anweisung

```
x = x+1;
```

für einen Informatiker intuitiv erscheint, für jemand, der noch nicht programmiert hat, aber nach einer nicht erfüllbaren Formel aussieht. Ähnliches gilt für die Frage, warum man nicht `x+1 = x;` schreiben darf, obwohl wenig später `x==y+1` und `y+1==x` semantisch äquivalent sind.

Dieser konsequente Weg des Starts ausschließlich mit Objekten wurde an der Hochschule Osnabrück in zwei einführenden Lehrveranstaltungen des Bachelor-Studiengangs „Informatik - Medieninformatik" genutzt. Der Ansatz wurde im Rahmen einer Reakkreditierung in sehr kontroversen Diskussionen 2010 ausgewählt, in dem nach einer Einführung in Java, in einer Zweitsemesterveranstaltung C und C++ gelehrt werden. Ursprünglich wurde nach einer Einführung in C im zweiten Semester C++ (mit all seinen sprachlichen Besonderheiten) auch in Veranstaltungen zu jeweils zehn Leistungspunkten gelehrt. Die Java-Einführung war ein kleiner Block im Rahmen einer Usability-Veranstaltung.

Die Rahmenbedingungen der ersten Umsetzung des neuen Ansatzes waren relativ schlecht, da es durch einen doppelten Abiturjahrgang und Wiederholer, die diese neue Veranstaltung auch hören mussten, zu einer sehr hohen Teilnehmerzahl mit anfänglich schlechter räumlicher Ausstattung kam.

Statistiken zeigen, dass dieser erste Ansatz trotz schlechter Rahmenbedingungen nicht schlechter abgeschlossen hat, als der vorherige Weg. Weiterhin zeigte die Analyse eine Herausforderung, dass alle an einer Lehrveranstaltung Beteiligten das Konzept der Vorlesung verstehen und aktiv fördern müssen. Dies ist gerade wichtig, da es dauert, bis die Welt aus Objekten, Klassen und Methoden in den Köpfen verankert wird, was den zweiten Teil einer Lehrveranstaltung mit den Ablaufsteuerungen dann aber deutlich vereinfacht, da sich hier die objektorientierten Strukturen festigen. Da im konkreten Fall nicht genügend Zeit zur Schulung der Beteiligten vorlag und diese auch ihre eigenen Vorstellungen vom Programmieren vermitteln wollten, konnte der Weg in den zugehörigen Praktika nicht immer konsequent beschritten werden.

Die Wiederholung der Veranstaltung hatte mit deutlich geringeren Studierendenzahlen bessere Voraussetzungen, wobei sich dann klar die Vorteile des Ansatzes mit einer geringeren Durchfallquote, die von 32,2% auf 20,9% sank, wobei die Quote bei den Erstsemestern von 16,9% auf 12% zurückging, zeigten.

Im Folgeverlauf konnten einige Studierende bereits im zweiten Semester erfolgreich unter Anleitung eigenständige Programmier- und Analyseaufgaben in Forschungsprojekten als studentische Hilfskräfte übernehmen.

Zusammenfassend kann der Ansatz als sehr erfolgsversprechend angesehen werden, wird aber

nicht konsequent weiter betrachtet, da die verantwortlichen Dozenten sich nicht auf diesen Weg einigen konnten.

Die folgenden Unterabschnitte analysieren einige weitere Faktoren, die für den Erfolg einer Programmierveranstaltung relevant sein können, was z. B. die Wahl der Entwicklungsumgebung betrifft.

Eclipse

Eclipse [@Ecl] ist eine mächtige Plattform, die mit vielen Arten von Erweiterungen individuell an Projekte angepasst werden kann und deshalb in Unternehmen in den meisten Java-Projekten gesetzt ist. Neben Java ist die Entwicklungsumgebung auch für weitere Programmiersprachen wie C und C++ effizient nutzbar. Für absolute Programmieranfänger ist aber oft die große Vielzahl an angebotenen Arbeitsschritten sehr verwirrend. Die Flexibilität, die Eclipse bei Entwicklern sonst sehr beliebt ist, macht die Umgebung für Anfänger schwer einschätzbar. Es gibt z. B. mindestens drei Wege, eine Klasse anzulegen, es gibt viele Sichten, die konfiguriert werden können und die Syntaxprüfung weist bereits beim Tippen auf potenzielle Fehler hin. Weiterhin werden für Java viele vereinfachende Schritte, wie die Generierung von get- und set-Methoden sowie `hashCode` und `equals` angeboten. Gerade bei diesen Methoden ist es sinnvoll, dass Anfänger sie für sich durch die eigene Erstellung entdecken, um gerade das sehr gute, aber nicht einfache Konzept der Basisklasse `Object` zu verstehen.

Man kann zwar die Nutzung von Eclipse auf wenige Klicks reduzieren, die für erste Programme wirklich benötigt werden, benutzt allerdings ein Nachbar bei der Programmentwicklung ihm schon bekannte Features, wächst schnell ein innerer Druck, dies auch beherrschen zu wollen.

Ein Einstieg in Eclipse ist zumindest optional sicherlich im Laufe der ersten Programmiererfahrungen sinnvoll und kann zum Selbststudium in der vorlesungsfreien Zeit vorgeschlagen werden.

Eclipse unterstützt nicht vollständig den Objects-First-Ansatz, da zumindest eine `main`-Methode in einer Klasse existieren muss.

BlueJ

BlueJ [@Blu] ist gegenüber Eclipse eine wesentlich einfachere Entwicklungsumgebung mit wesentlich weniger Möglichkeiten. Ein einfaches Googeln nach BlueJ mit der Ergänzung „filetype:pdf“ zeigt, dass BlueJ an vielen Schulen aber auch Hochschulen in der Programmierausbildung genutzt wird.

BlueJ ermöglicht es, einfach Objekte gegebener Klassen zu erzeugen und mit diesen Objekten über deren Methoden zu interagieren. Dazu werden die verfügbaren Klassen in einem Klassendiagramm angezeigt, von denen dann über einen Rechtsklick Objekte erzeugt und Klassenmethoden aufgerufen werden können. Die Objekte liegen unten in einer Objektleiste, so dass über einen Rechtsklick deren Methoden ausführbar sind. Durch einen Doppelklick auf ein Objekt werden alle Objektvariablen mit Namen und Werten angezeigt. Änderungen über Methodenaufrufe werden auch in die angezeigten Objektvariablen übernommen. Hat eine Objektvariable eine Klasse als Typ, kann auch weiter in diese Objekte hineingeschaut werden. BlueJ bietet einen einfachen Debugger, der wie üblich über Break Points die schrittweise Ausführung und Überprüfung von Programmen ermöglicht.

Als weitere Besonderheit bietet BlueJ eine Konsole, das Code Pad, mit dem Java als Skriptsprache genutzt werden kann. Anweisungen werden hier direkt eingetippt und ausgeführt. Weiterhin sind Ausdrücke eingebbar, deren Ergebnisse sofort ausgegeben werden. Handelt es sich hierbei um Objektreferenzen, können diese in die Objektleiste gezogen und weiter analysiert werden.

Der Editor von BlueJ ist recht einfach gehalten, unterstützt aber die sinnvolle automatische Code-Formatierung und die Generierung von Java-Doc. Alle Änderungen werden unmittelbar abgespeichert. Der Compiler gibt nur jeweils die erste Fehlermeldung aus, um nicht mit teilweise unverständlichen Folgefehlern zu verwirren.

Zusammen mit BlueJ werden bereits einige Beispiele geliefert, die den didaktischen Ansatz unterstützen, dass Anfänger zunächst spielerisch lernen, Objekte zu erzeugen und deren Methoden auszuführen. Ein bekanntes Beispiel „shapes“ erlaubt es, verschiedene graphische Elemente auf einer Canvas-Oberfläche zu platzieren und diese zu verschieben und umzufärben. Dieses prinzipiell sehr schöne Beispiel hat allerdings später den Nachteil, dass an zentraler Stelle eine Klassenmethode genutzt wird, ein Konzept, dass man erst spät in einer Einführung in die objektorientierte Programmierung vorstellen sollte, da sonst die zunächst zu erlernende Grenze zwischen einer nicht ausführbaren Klasse und Objekten, mit denen Programmcode ausführbar ist, wieder verschwimmt.

Generell ist die Idee, möglichst einfach graphische Ausgaben oder sogar Interaktionen mit der Oberfläche zu erlauben, sehr sinnvoll, da man als Lernender schnell zu anschaulichen Programmen kommt, was oft auch den Spiel- und den für das Lernen dringend benötigten Probier-Trieb anspricht. Für das gezeigte Beispiel und auch einige Anfangsprogramme in [BK09] gilt leider, dass gegen die konsequente Objektorientierung verstoßen und schnell z. B. zu `System.out.println()` gegriffen wird.

Um auch bei Ein- und Ausgaben konsequent auf Objekte zu setzen, ist es sinnvoll, diese Funktionalität in einer Klasse mit Objektmethoden zu kapseln. Dies löst aber das Problem nicht vollständig, da es typischerweise trotz mehrfacher Konstruktornutzung jeweils nur ein Objekt dieser Klasse gibt.

Greenfoot

Die aus der graphischen Ausgabe gezogene Visualisierung ist auch eine Motivation für eine Alternative zu BlueJ namens Greenfoot [@Gre], ebenfalls einer Entwicklungsumgebung für Anfänger, die allerdings auf BlueJ basiert. Typischerweise werden mit Greenfoot sogenannte Objektwelten definiert, deren Objekte auf einem zweidimensionalen Feld als Objekte platziert werden können. Wie in BlueJ sind dann direkt alle Methoden des Objekts und das Objekt selbst zur Analyse zugreifbar und Methoden können ausgeführt werden.

Das typische Einführungsbeispiel sind Wombat-Objekte, die über das Feld mit Hilfe der Methoden gesteuert werden können und z. B. Blätter suchen und fressen. Generell vermittelt Greenfoot so sehr schön den Objektbegriff, man kann sogar einfache Vererbungen verfolgen, allerdings wird im Beispiel-Code wieder sehr schnell auf Klassenvariablen und Klassenmethoden zugegriffen. Greenfoot nutzt im Wesentlichen dabei eine Objektwelt, um zentral die Nutzung von `if` und `while` zu trainieren. Man muss dabei klar zwischen der reinen Nutzung der Objektwelt, bei der die Objekte nur über Methoden gesteuert werden können, und der Erstellung neuer Welten, Szenarien genannt, die von erfahreneren Entwicklern angelegt werden sollten, trennen. Von Programmieranfängern wird nur die Erweiterung existierender Szenarien erwartet.

Greenfoot eignet sich auch, um einfache interaktive Spiele zu erstellen, wobei hier von der konsequenten Objektorientierung abgewichen wird.

```
if (Greenfoot.mouseClicked(null))
```

Die vorherige Fallunterscheidung dient z. B. dazu, festzustellen, ob die Maus zum Klicken genutzt wurde. Um dann das geklickte Objekt zu finden, muss ein Cast-Operator und ein Klassen-Objekt genutzt werden, wie das folgende Code-Fragment zeigt.

```
Balloon balloon = (Balloon) getOneObjectAtOffset(x , y
                                        , Balloon.class);
```

Verwandt mit Greenfoot sind einige andere Ansätze, wie das Hamster-Modell [Bol08], Kara [@Kara], Karol [@Karo] oder Scratch [@Scr], mit denen eigene Umgebungen, typischerweise in Java, geschaffen werden, um Schüler an die ersten Schritte des Programmierens, also die systematische Hintereinanderausführung von Befehlen, heranzuführen. Dieser Ansatz mag zu schnellen „Erfolgen“ bei den Lernenden führen, was die Motivation hochhalten kann, ob der Übergang dadurch in reale Programmiersprachen einfacher oder durch ein dann fundiertes zu einfaches mentales Entwicklungsmodell erschwert wird, ist bisher in Studien nicht ernsthaft untersucht worden.

Interaktionsbrett

Die Motivation von Lernenden mit graphischen Möglichkeiten hochzuhalten, ist in vielen Ansätzen vertreten. Aus diesem Grund wurde für die hier betrachteten Java-Veranstaltungen in Osnabrück motiviert von der kritischen Analyse von Greenfoot eine eigene Klasse Interaktionsbrett zur Visualisierung erstellt [@Int].

Die zentrale Idee ist es, konsequent objektorientiert eine Klasse nutzbar zu machen, mit der die einfachen graphischen Elemente Punkt, Linie, Kreis und Rechteck gezeichnet und mit der Maus bewegt werden können. Weiterhin wird die Tastatursteuerung unterstützt. Zum Einstieg werden auch hier einfache Befehlsketten genutzt, die ein Bild auf der Oberfläche zeichnen. Danach werden eigene Klassen für graphische Objekte, wie Dreiecke, angelegt, die dann selbst Methoden enthalten, um sich auf ein Interaktionsbrett zu zeichnen.

Das folgende Programm zeigt die Möglichkeit, einen Kreis mit Hilfe der Pfeil-Tasten nach links und rechts zu schieben. Etwas „negative Magie“ wird auch im Interaktionsbrett benötigt, da ein Objekt, das über Tastatureingaben informiert werden möchte, eine Methode der Form `public void tasteGedrueckt(String s)` realisieren muss. Das Interaktionsbrett nutzt Reflektion, um dann diese Methode aufzurufen.

```
public class Steuerung {
  private Interaktionsbrett ib = new Interaktionsbrett();
  private int pos = 0;

  public Steuerung() {
    ib.willTasteninfo(this);
  }

  public void tasteGedrueckt(String s) {
    if (s.equals("Rechts")) {
      pos = pos + 5;
    } else if (s.equals("Links")) {
      pos = pos - 5;
    }
    ib.abwischen();
    ib.neuerKreis(pos, 5, 20);
  }

  public static void main(String[] s){
    new Steuerung();
  }
}
```

Mit Hilfe des Interaktionsbretts können auch leicht etwas kompliziertere Spiele programmiert werden. Abb. 1 zeigt ein Beispiel einer Abschlussaufgabe, in der mehrere Monster, die eine Mauer erklimmen, mit Steinen von einem auf der Burgmauer laufenden Wächter abgewehrt werden müssen.

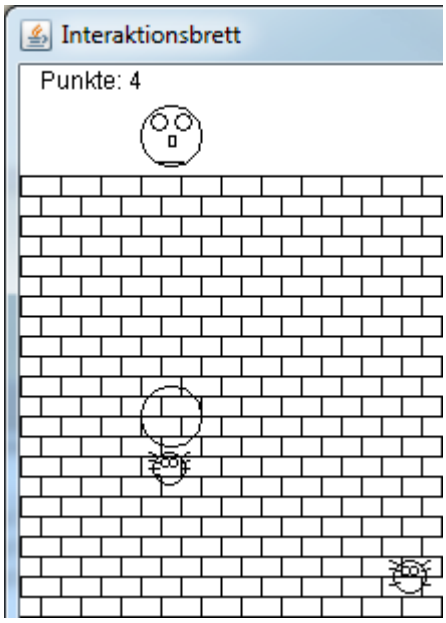


Abb. 1: Interaktives Spiel mit Objekten

Jedes der graphischen Elemente kann mit der Maus bewegt werden, wozu auch etwas Magie notwendig ist. Wieder muss festgelegt werden, welches Objekt informiert werden soll, wenn eine Mausaktion stattfindet. Weiterhin ist es sinnvoll, dass mehrere Objekte über einen Namen als String unterschieden werden können, wenn das gleiche Objekt informiert werden soll. Für einen verschiebbaren Kreis mit x- und y-Koordinate sowie einem Radius, sieht dann eine Methode zum Anmelden beim Interaktionsbrett wie folgt aus.

```
ib.neuerKreis(this,"Ball",ib.zufall(0,300),42,10);
```

Hier wird das Objekt, das diesen Befehl ausführt, auch über die Mausaktion, die in angeklickt, verschoben und losgelassen unterschieden werden, informiert. Dazu muss zumindest eine der drei zu den Aktionen gehörenden Maus-Methoden realisiert werden, die als Parameter den Namen des Objekts und seine neuen Koordinaten übermittelt bekommt.

```
public Boolean mitMausAngeklickt(String name
                          , Integer x, Integer y){
  return !versenkt;
}
```

Der Boolesche Ergebniswert legt fest, ob die Aktion überhaupt erwünscht ist. Will man z. B. einen Knopf realisieren, muss er auf das Klicken mit der Maus reagieren und mit false antworten, da er nicht verschoben, aber informiert werden will.

Generell wurde das Interaktionsbrett von Studierenden in Praktika sehr positiv aufgenommen, da man sehr einfach zeichnen kann. Das Einüben der Maussteuerung ist dabei deutlich komplizierter und gehört in den zweiten Teil der Lehrveranstaltung. Aktuell wird diskutiert, ob die absichtlich gewählte minimale Darstellung von ausschließlich schwarzen Linien und der Verzicht auf einen Hintergrund, durch sinnvolle Erweiterungen ergänzt werden sollen. Die Weiterentwicklung findet dabei durch Studierende statt, die weitere Beispiele erstellen.

Processing

Processing [@Pro] wurde als Sprache und Entwicklungsumgebung entwickelt, um kreativen Leuten einen sehr einfachen Zugang zu den Themen Visualisierung, interaktive Animation und Simulation zu ermöglichen. Die zugehörigen Arbeiten wurden am Massachusetts Institute of Technology von Ben Fry und Casey Reas gestartet. Mittlerweile gibt es im Internet eine große Unterstützung des Ansatzes mit vielen Tutorials und quelloffenen Beispielen. Processing wird in vielen Design-Studiengängen, u. a. auch im Media and Interaction Design-Studiengang an der Hochschule Osnabrück für kleinere und größere Projekte eingesetzt.

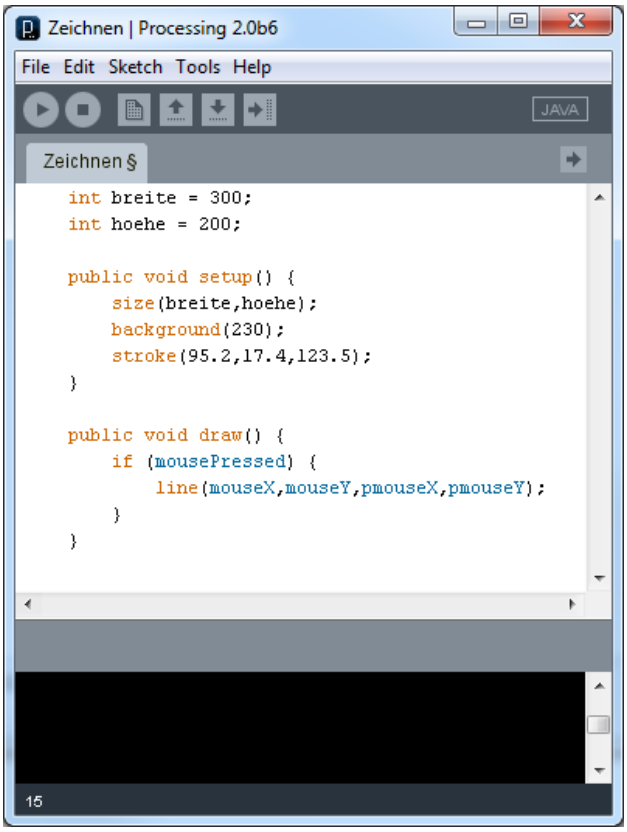


Abb. 2: Malprogramm in Processing

Sehr interessant ist aus didaktischer Sicht, dass viele Studierende, die sich bewusst für einen künstlerischen Studiengang entschieden haben, durch Processing einen schnellen und oft begeisternden Einstieg in die Programmierung finden. Es wird wieder auf die unmittelbare Visualisierung und recht einfache Interaktionsmöglichkeiten gesetzt. Anders als bei „Spielsprachen" findet die Programmierung im Wesentlichen in Java statt, wobei die Entwicklungsumgebung es ermöglicht, den Objektbegriff in den Hintergrund zu stellen. Dies soll mit dem vollständigen Programm in Abb. 2, mit dem ein Nutzer mit gedrückter Maustaste auf einer Fläche zeichnen kann, veranschaulicht werden.

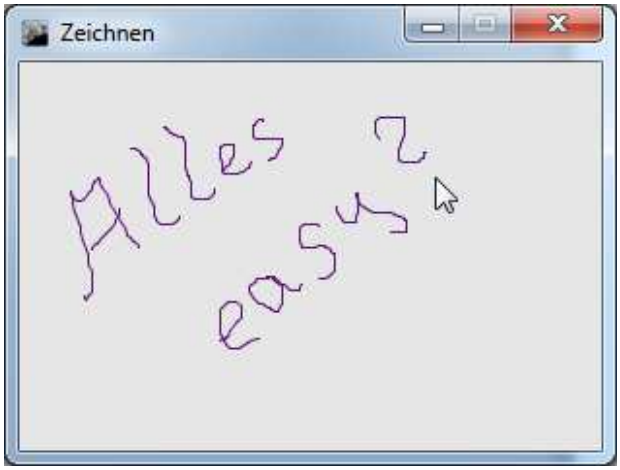


Abb. 3: Malprogramm in Benutzung

Abb. 3 zeigt das entstehende Beispielfenster, das auch leicht als ausführbares Programm, Java-Applet oder als auf Android lauffähiges Programm exportiert werden kann.

Zentral nutzt Processing zwei Funktionen (sogenannt im Processing-Sprachgebrauch), `init()` und `draw()`, wobei `init()` einmal am Programmanfang und `draw()` danach in einer Endlosschleife aufgerufen werden. Zentral in Processing ist die große Menge von Funktionen, die zur Erzeugung und Verarbeitung graphischer Elemente bereits zur einfachen Nutzung vorliegen, wodurch die eigentliche Processing-Sprache definiert wird. Dazu gehören auch viele Varianten, Farbtöne anzugeben und detailliert Fonts auszuwählen. Im obigen Beispiel sieht man, wie die Hintergrundfarbe als eine Möglichkeit mit einem `int`-Wert festgelegt wird. Mit `stroke()` wird die Farbe für die nächsten gemalten Objekte festgelegt. Dabei ist die Grundidee ähnlich wie beim Malen selbst, dass man durch Funktionen festlegt, wie gemalt wird und dies dann für alle folgenden Malaktionen gilt.

Das Beispiel zeigt auch, dass in Processing einfach einige „negative Magie" genutzt wird, um das Programmieren zu erleichtern. Es gibt eine Boolesche Variable `mousePressed`, mit der der Zustand der Maus abgeprüft werden kann. Weiterhin befindet sich die aktuelle Mausposition in den Variablen `mouseX` und `mouseY` sowie die unmittelbar vorherige Position in den Variablen `pmouseX` und `pmouseY`.

Processing ist keine prozedurale Sprache wie C, da die Klassen der Java-Klassenbibliothek genutzt werden und auf deren Objekte mit Methoden zugegriffen wird. Es entsteht so eine Mischung aus Funktionen und Methodenaufrufen. Die Entwicklungsumgebung enthält keinen Debugger. Semantisch programmiert man in Processing eine Erweiterung einer Klasse `processing.core.PApplet`, in die der Code aus dem Editor eingebaut wird. Man kann so auch direkt in Processing neue Klassen selbst entwickeln, die dann zu lokalen Klassen der großen umgebenden Klasse werden und direkt auf die Processing-Funktionen zugreifen können. Alternativ kann man in Processing auch echte neue Klassen schreiben, muss dann aber selbst eine Referenz zur Nutzung der Processing-Funktionen aufbauen.

Da Processing im Wesentlichen in Java geschrieben ist, kann man die zugehörigen Klassenbibliotheken auch in andere Programme einbauen. In [Kle12] wird technisch beschrieben, wie man Processing aus BlueJ heraus nutzen kann, wobei durch die besondere Art der Ausführung der Debugger nicht zur Verfügung steht. Funktional muss der Entwickler für angeklickte Elemente selbst berechnen, welches Element angeklickt wurde, eine Funktionalität, die beim Interaktionsbrett schon gegeben ist. Weiterhin unterstützt Processing keine Eingabe in Konsolenfenstern.

Die vorherigen Abschnitte haben klar gezeigt, welche didaktischen Defizite Processing enthält, weshalb man es mit guten Gründen nicht in einer Programmiereinführung einsetzen sollte. Dem muss man die weite Verbreitung von Processing in kreativen Bereichen gegenüberstellen, in denen es auch gelingt, nicht programmieraffinen Personen die Begeisterung an der Entwicklung beizubringen. Ob dieser Ansatz auch den systematischen späteren Ein- oder Umstieg in die objektorientierte Programmierung erlaubt, müssten längerfristige Untersuchungen zeigen.

Zwischenfazit zu Java

In den letzten Abschnitten wurde der klassische imperative Lehrweg dem konsequenten Ansatz Objects First gegenüber gestellt und aus Sicht des Autors die Vorteile in der durchgeführten Lehre vorgestellt. Diese Sichtweise spiegelt sich in der neu entstehenden Literatur zu Java [HMG11] [Ull11] [Far12] nur selten wieder [Wu10], es wird maximal ein Ansatz „Objects early" versucht, der „schnell" etwas zu Alternativen und Schleifen zei-

gen will, um dann in Objekte und Klassen einzusteigen. Da hier auch am Anfang Programme mit Strukturen geschrieben werden, die konsequent der objektorientierten Vorgehensweise widersprechen, kann aus didaktischer Sicht hier kein Unterschied zum ursprünglichen Vorgehen gesehen werden.

Aus Sicht der doch recht kleinen Gruppe der „Objects first"-Vertreter stellt sich die Frage, ob der Ansatz ein Irrtum ist und man gegen Windmühlen kämpft oder ob man weiterhin bei der Behauptung „die Erde ist rund" bleiben soll. Die weiteren Abschnitte zu Java zeigen eventuell einen anderen Weg, dass man den „Spaß an der Entwicklung" herstellen soll und die Zeit mit den Folgeerfahrungen den Weg zur guten Programmierung ebnet.

Zusammengefasst bietet Java sehr viele Möglichkeiten zu einem spannenden Einstieg in die Programmierung, wobei man die anfänglich genannten sprachlichen Probleme beachten und didaktisch umschiffen muss.

Fazit

Etwas verallgemeinert kann man die dargestellten Erfahrungen in der Form zusammenfassen, dass die Auswahl der ersten Programmiersprache im Studium wahrscheinlich kein wesentlicher Faktor für den Studien- und den Berufserfolg von Studierenden ist. Begeisterte und begeisternde Dozenten mit guter fachlicher Unterstützung und einfach zu nutzenden Werkzeugen können als Erfolgsfaktoren bestimmt werden. Dies bedeutet auch, dass Dozenten zumindest in den ersten Semestern ein gemeinsames Konzept in der Anwendung von Programmiersprachen finden sollten, was häufiger nicht der Fall ist.

Möchte man nach dem ersten Semester eine Programmiergrundlage legen, die recht einfach erweitert werden kann und Studierenden nebenbei recht viele Nebentätigkeiten in Hochschulen oder Unternehmen ermöglicht, sollte die Auswahl auf eine objektorientierte Sprache fallen. Dabei gibt es Indikatoren auf Basis der Literatur und der gemachten Erfahrungen, dass ein konsequenter Einstieg mit dem Erlernen von Objekten und Klassen den Vorteil hat, dass diese Abstraktionsebene sehr früh eingeübt wird.

Eine wirkliche fundierte Untersuchung zeigt sich als schwierig, da hier mehrere Herausforderungen der empirischen Sozialforschung zusammenkommen, um mögliche Hypothesen überprüfen zu können. Oftmals haben Studierende bereits Kenntnisse der Programmierung aus der Schule oder haben sich diese teilweise selbst beigebracht. Dabei zeigt sich immer wieder, dass die Programmierausbildung in den Schulen ein extremst unterschiedliches Niveau hat. Häufiger sind Fälle zu beobachten, dass Lehrer, die natürlich ebenfalls auf der Suche nach dem richtigen Ansatz zur Programmierausbildung sind, zu sehr merkwürdigen Programmierrichtlinien und Programmierstilen führen. Natürlich ist auch das andere Extrem vertreten, bei denen wesentliche Teile von Programmierveranstaltungen an der Hochschule für Studierende dank ihrer Schul- oder evtl. betrieblichen Ausbildung trivial werden. Die Frage, ob ein Informatik-Unterricht mit Inhalten der Studiensemester in der Schule überhaupt sinnvoll ist, sei dabei am Rande gestellt.

Eine Analyse des Erfolgs eines Ansatzes ist noch schwieriger, da er eigentlich nur durch die intensive Analyse der Fähigkeiten von Studierenden ermöglicht wird. Da aber die damit verbundene intensivere Betreuung bereits wesentlichen Einfluss auf den Lernerfolg haben kann, wird eine Beurteilung oder gar ein Vergleich von Ergebnissen sehr schwierig.

Es stellt sich so die Frage, ob statt einer direkten Beobachtung Indikatoren gefunden werden können, die auf den Programmiererfolg schließen lassen. Dies könnte durch die Verbindung von Entwicklungswerkzeugen mit statischen Analysewerkzeugen geschehen, die z. B. messen, wie oft ein Compiler aufgerufen wird und wie oft Fehler z. B. in gleichen Zeilen dabei gefunden werden. Diese Analysewerkzeuge dürfen dabei die eigentliche Nutzung nicht beeinflussen. Vor diesem Ansatz ist aber dann zu klären, ob die Indikatoren wirklich geeignet sind. Der Erfahrungsbericht bietet einige Fragen und Hypothesen, die genauer zu untersuchen sind.

- Da die am meisten angewandten Sprachen objektorientiert sind, stellt sich die Frage, ist ein Start mit C noch sinnvoll?
- Wenn man objektorientiert beginnt, sollte man dann rigoros auf Objects first setzen?
- Ist Java als Einführungssprache geeignet, obwohl sie nicht konsequent objektorientiert ist?
- Welchen Motivationsschub kann aus der Möglichkeit schnell einfache Animationen zu erstellen gezogen werden? Gibt es vergleichbar erfolgreiche Ansätze, wie der Einsatz von kleinen Robotern?
- Welche Bedeutung hat die Auswahl von Entwicklungsumgebungen?
- Wie sieht eine kontinuierliche Integration der Programmierausbildung in die weiteren Lehrveranstaltungen der ersten Semester aus? Ist es z. B. sinnvoll parallel eine weitere Sprache wie JavaScript in einer Einführung zur Web-Technologie zu lehren?

- Welche Bedeutung hat das Engagement der Lehrenden? Ketzerisch formuliert, sind die vorherigen Fragen nachrangig, wenn das echte Bemühen die Programmierung näher zu bringen, deutlich wird?

Zusammengefasst wird die Suche nach dem richtigen Weg in der Programmierausbildung sicherlich wie in der Vergangenheit auch, zu mehrfach neu gefundenen und dann auch wieder konsequent verworfenen Wegen führen.

Der Autor dankt Dr. Cheryl Kleuker und Prof. Dr. Frank Thiesing für die Diskussion und Kommentierung einer Vorversion dieses Artikels.

Literatur

Web-Seiten vom Stand 1.11.2012 betrachtet.

[@Blu] BlueJ, http://www.bluej.org/

[@Ecl] Eclipse, http://www.eclipse.org/

[@Gre] Greenfoot, http://www.greenfoot.org/door

[@Int] Interaktionsbrett, http://home.edvsz.hs-osnab-rueck.de/skleuker/querschnittlich/InteraktionsbrettMID/index.html

[@Kara] Programmieren lernen mit Kara, http://www.swisseduc.ch/informatik/karatojava/index.html

[@Karo] Java Karol, http://www.schule.bayern.de/karol/jdownload.htm

[@Pro] Processing, http://processing.org/

[@Scr] Scratch, http://scratch.mit.edu/

[@SDL] Simple DirectMedia Layer, http://www.libsdl.org/

[BK09] D. J. Barnes , M. Kölling, Java lernen mit BlueJ: Eine Einführung in die objektorientierte Programmierung, 4. Auflage, Pearson Studium, 2009

[Boe07] J. Boerstler, Objektorientiertes Programmieren - Machen wir irgendwas falsch?, Didaktik der Informatik in Theorie und Praxis 2007, http://subs.emis.de/LNI/Proceedings/Proceedings112.html, Seiten 9-20, 2007

[Bol08] D. Boles, Programmieren spielend gelernt mit dem Java-Hamster-Modell, 4. Auflage, Vieweg+Teubner, Wiesbaden, 2008

[Bol09] D. Boles, Threadnocchio – Einsatz von Visualisierungstechniken zum spielerischen Erlernen der parallelen Programmierung mit Java-Threads, in U. Jaeger, K. Schneider (Hrsg.), Software-Engineering im Unterricht der Hochschulen 2009, Seiten 131-144, dpunkt.verlag, Heidelberg, 2009

[Bra08] J. Brauer, Grundkurs Smalltalk - Objektorientierung von Anfang an: Eine Einführung in die Programmierung, 3. Auflage, Vieweg+Teubner, Wiesbaden, 2009

[Far12] J. Farell, Java Programming, 6. Auflage, Course Technology, Cengage Learning, USA, 2012

[Fla08] D. Flanagan, The Ruby Programming Language, O'Reilly, USA, 2008

[HMG11] C. Heinisch, F. Müller-Hofmann, J. Goll, Java als erste Programmiersprache, 6. Auflage, Vieweg+Teubner, Wiesbaden, 2011

[HSRM10] Hochschule RheinMain, Modulhandbuch Angewandte Informatik, 2010

[Jad06] M. C. Jadud, An Exploration of Novice Compilation Behaviour in BlueJ, PhD Thesis University of Kent, 2006 (http://kar.kent.ac.uk/14615/1/An_Exploration_of_Novice_Compilation_Behaviour_in_BlueJ.pdf)

[Kle12] S. Kleuker, Technischer Bericht, http://home.edvsz.hs-osnab-rueck.de/skleuker/querschnittlich/BlueJUserManualMID.pdf

[KS12] P. Kinnunen, B. Simon, My program is ok – am I? Computing freshmen's experiences of doing programming assignments, Computer Science Education, 22:1, Seiten 1-28, 2012

[MFRW11] L. Ma, J. Ferguson, M. Roper, M. Wood, Investigating and improving the models of programming concepts held by novice programmers, Computer Science Education, 21:1, Seiten 57-80, 2011

[RTH11] S. Ruby, D. Thomas, D. Heinemeier Hansson, Agile Web Development with Rails, 4. Auflage, Pragmatic Programmers, USA, 2011

[SZ07] A.Schmolitzky, H. Züllighoven, Einführung in die Softwareentwicklung - Softwaretechnik trotz Objektorientierung? In: A. Zeller and M. Deininger (Hrsg.), Software Engineering im Unterricht der Hochschulen, Seiten 87-100, 2007

[Ull11] C. Ullenboom, Java ist auch eine Insel, 9. Auflage, Galileo Computing, Bonn, 2011

[Wu10] C. T. Wu, An introduction to object-oriented programming with Java, 5. Auflage, Mc Graw Hill, USA, 2010

Analyse von Programmieraufgaben durch Softwareproduktmetriken

Michael Striewe, Michael Goedicke
Universität Duisburg-Essen
{michael.striewe,michael.goedicke}@s3.uni-due.de

Zusammenfassung

Der Einsatz von Softwareproduktmetriken zur Beobachtung und Beurteilung von Softwareprojekten ist ein oft diskutiertes Thema. Es gibt vielfältige Vor- und Nachteile, die beim Einsatz zu berücksichtigen sind. Der vorliegende Artikel diskutiert an Fallbeispielen, wie Metriken zur Lehrunterstützung in einer Lehrveranstaltung zur Programmierung genutzt werden können.

Einleitung

Softwareentwicklung kann über Kennzahlen, die durch den Einsatz von Metriken erhoben werden, beobachtet und bewertet werden. Dabei kann sowohl das entwickelte Softwareprodukt als auch der Entwicklungsprozess Ziel der Beobachtung und Bewertung sein (Conte u. a., 1986). Der Einsatz von Metriken wird vielfach kritisch diskutiert: Auf der einen Seite steht die Aussage, dass ohne Messung keine Kontrolle möglich ist (z.B. (DeMarco, 1986)) und auf der andere Seite die Erkenntnis, dass die Messung das Ergebnis selber beeinflussen kann und viele Metriken gar nicht das messen, was sie zu messen vorgeben (z.B. (Kaner u. Bond, 2004)). Insgesamt kann der Einsatz von Softwaremetriken aber als etabliert betrachtet werden.

Nicht nur in der industriellen Praxis, sondern auch in der Lehre werden Softwareprodukte entwickelt. Es handelt sich hierbei zwar nicht um Produkte im ökonomischen Sinne, aber doch um Artefakte, deren Eigenschaften mit denselben Metriken messbar sind. Zudem ist der Begriff der Kontrolle in Form von Lernfortschrittskontrollen ein zentraler Bestandteil der Lehre. Es erscheint daher sinnvoll, Werkzeuge zur Unterstützung der Lehre zu konzipieren, mit denen Kennzahlen automatisch gewonnen und analysiert werden können. Dies ist insbesondere in E-Learning- und Blended-Learning-Szenarien wichtig, in denen eine (teil-)automatisierte Führung durch den Lernstoff angeboten werden soll und daher nicht immer ein Lehrender zur Verfügung steht, der die Leistungsfähigkeit seiner Studierenden einschätzt und ihnen passende Übungsaufgaben zuweist. Um hier eine Automatisierung zu erreichen sollte untersucht werden, wie Softwaremetriken zur Beobachtung des Lernprozesses und der dabei erzeugten Produkte einsetzbar sind. Der vorliegende Artikel stellt letzteres in den Fokus und befasst sich daher ausschließlich mit Softwareproduktmetriken. Aus praktischen Erwägungen heraus beziehen sich die im Folgenden diskutierten Ansätze und Ergebnisse auf Anfängervorlesungen zur Programmierung. Eine Generalisierung auf fortgeschrittene Lehrveranstaltungen zur Programmierung sowie Software Engineering im Allgemeinen bedarf zweifellos weiterer Forschungsarbeit, die weit über den Rahmen dieses Artikels hinaus geht.

Der Artikel ist wie folgt gegliedert: Zunächst werden einige Szenarien skizziert, in denen Metriken zur Lehrunterstützung angewandt werden können. Im selben Abschnitt werden zudem grundsätzliche Analysemethoden und existierende Ansätze genannt. Anschließend werden Metriken vorgestellt, die bei der Analyse von Programmieraufgaben genutzt werden können. Danach werden mehrere Fallbeispiele diskutiert, in denen eine Auswahl der Metriken auf tatsächliche Lösungen von Programmieraufgaben angewandt wurden. Aus den daraus gewonnenen Erkenntnissen wird am Ende des Beitrags ein Fazit gezogen.

Szenarien, Methoden und Verwandte Arbeiten

Es sind verschiedene Einsatzszenarien denkbar, in denen Softwareproduktmetriken bei der Analyse von Programmieraufgaben und ihren Lösungen zum Einsatz kommen können. Sie unterscheiden sich sowohl nach ihrer Zielsetzung als auch der Analysebasis: Als Zielsetzung gibt es die Rückmeldung an die Studierenden, die Rückmeldung an die Lehrenden und die systeminternen Nutzung zur automatischen Steuerung von Lernprozessen, während bei der Analysebasis zu unterscheiden ist, ob Aussagen über eine individuelle Lösung oder die Gesamtheit der Lösungen einer Aufgabe getroffen werden sollen. Aus diesen Merkmalen lassen sich zahlreiche Kombinationen bilden, z.B.:

- Durch die Analyse einzelner Lösungen und die Aufbereitung der Ergebnisse für die Studierenden können Aussagen zum Verhältnis von studenti-

scher Lösung und Musterlösung getroffen werden. Studierenden kann so zum Beispiel der Hinweis gegeben werden, dass ihre Lösung zwar alle funktionalen Anforderungen erfüllt, jedoch deutlich umfangreicher als die Musterlösung ist. Dies muss nicht mit einer negativen Bewertung der Lösung einhergehen, sondern kann auch als Hinweis verstanden werden, der die Studierenden zur Verbesserung ihrer Lösung anregen soll.

- Durch die Analyse aller Lösungen und die Aufbereitung der Ergebnisse für die Studierenden kann diesen angezeigt werden, ob sie eine typische Lösung gewählt haben, oder ob sie sich in bestimmten Merkmalen deutlich von anderen Lösungen unterscheidet. Auch dies kann den Studierenden einen Hinweis darauf geben, in welche Richtung sie ihre Lösungen verbessern können.

- Lehrende können dieselben Daten nutzen, um Häufungen bei Lösungsmerkmalen zu entdecken und damit einander ähnliche Lösungen zu finden. Handelt es sich dabei um Fehlerschwerpunkte, können diese in Lehrveranstaltungen gezielt aufgegriffen werden. Handelt es sich dagegen um gute Lösungen, die jedoch von der Musterlösung messbar abweichen, kann dies Hinweise auf einen alternativen Lösungsweg geben, der in einer zweiten Musterlösung berücksichtigt werden könnte.

- Ebenfalls durch die Analyse aller Lösungen und den Vergleich dieser Mengen über verschiedene Aufgaben hinweg können Aussagen über generelle Merkmale der jeweiligen Aufgaben getroffen werden. Zum Beispiel lässt sich so der minimale Umfang einer korrekten Lösung ableiten oder Aufgaben können anhand des Mittelwertes verschiedener Kennzahlen miteinander verglichen werden. Letzteres ist insbesondere auch in vollautomatisierten Systemen nutzbar, die auf diese Weise relevante Eigenschaften von Aufgaben lernen können.

Die letzte Kategorie steht im Fokus des vorliegenden Beitrags. Dabei wird davon ausgegangen, dass der Einsatz von Metriken in diesem Fall nur dann zu aussagekräftigen Ergebnissen kommt, wenn eine größere Menge von Aufgabenlösungen analysiert werden kann. Erst dann erscheint eine Verallgemeinerung der durch Kennzahlen gewonnen Aussagen auf die Aufgabe insgesamt sinnvoll, da in einer freien Aufgabenform wie Programmieraufgaben eine zu kleine Stichprobe kaum als repräsentativ angesehen werden kann. Aus der Gesamtheit aller Kennzahlen für eine große Menge von Lösungen sind dagegen Aussagen über die Aufgabe zu erwarten, die beispielsweise auch zur Qualitätssicherung genutzt werden können, wenn Aufgaben zu einem späteren Zeitpunkt erneut eingesetzt werden sollen.

Bei der Analyse von großen Mengen von Lösungen wird durch diese in der Regel ein Lösungsraum aufgespannt, der dann untersucht wird. Die Analyse des Raumes kann dabei wie oben skizziert auf Softwareproduktmetriken basieren, oder aber auch auf Ähnlichkeitsmaßen zwischen Lösungen. Ein möglicher Einsatzbereich ist dabei die Plagiatserkennung, bei der es nicht um die Eigenschaften der Lösungen selber geht, sondern nur um die Nähe der Lösungen zueinander (Leach, 1995). Die Nützlichkeit von Maßvergleichen in diesem Bereich ist jedoch strittig, so dass es auch zahlreiche Ansätze gibt, die Textvergleiche heranziehen.

Der Einsatz von Softwareproduktmetriken zur Qualitätssicherung und Unterstützung der Benotung wurde ebenfalls schon in Ansätzen untersucht (Mengel u. Yerramilli, 1999). Rückt die Kontrolle des Lernfortschritts oder die Generierung von Rückmeldungen in den Fokus, können auch komplexe mathematische Verfahren oder maschinelles Lernen zum Einsatz kommen (Martín u. a., 2009; Gross u. a., 2012). Im vorliegenden Beitrag werden jedoch einfachere Techniken der Datenanalyse eingesetzt.

Einige Metriken

Bei Softwareproduktmetriken wird zwischen Metriken zur Messung von Umfang und Größe, Metriken zur Messung von Struktur und Komplexität sowie Metriken zur Messung von Anwendung und Nutzen unterschieden. Die ersten beiden Kategorien lassen sich dabei den sogenannten internen Attributen von Softwareartefakten zuordnen, während die dritte Kategorie sogenannte externe Attribute berücksichtigt (Conte u. a., 1986; Fenton u. Pfleeger, 1998). Da in letzterer Kategorie die Interaktion eines Artefakts mit der Umwelt (d.h. zum Beispiel die Nutzerfreundlichkeit) gemessen wird, kann diese für die automatisierte Generierung von Kennzahlen nur begrenzt verwendet werden.

Im Folgenden werden einige Softwareproduktmetriken vorgestellt, die für die Anwendung auf Programmcode in objekt-orientierten Programmiersprachen geeignet sind. Dabei wird insbesondere untersucht, ob sie speziellen Anforderungen aus dem Kontext von Lehrveranstaltungen gerecht werden:

- Da in Übungsaufgaben häufig Codevorlagen eingesetzt werden, muss der Einfluss des vorgegebenen Programmcodes auf das Ergebnis der Metrik zuverlässig bestimmt werden können, um eine Verfälschung oder Verwässerung der Ergebnisse zu verhindern.

- Die Metriken müssen zudem geeignet sein, die Verfolgung eines didaktisch sinnvollen Ziels zu unterstützen, d.h. Kennzahlen liefern, die im Rahmen der Lehre überhaupt relevant sind.

Anzahl der Anweisungen

Die Zählung der Anweisungen im Programmcode („statements“ in Abgrenzung zu „expressions“) stellt

eine sehr einfache Umfangsmetrik dar. Im Gegensatz zur Zählung von Codezeilen („lines of code“) hat sie den Vorteil, dass sie resistent gegenüber Formatierungsänderungen am Programmcode ist und die Länge eventueller Code-Kommentare nicht berücksichtigt. Das Ergebnis dieser Metrik wird daher nur durch tatsächliche inhaltliche Änderungen am Programmcode beeinflusst. Allerdings haben nicht alle Änderungen am Programmcode Einfluss auf die Metrik, da Erweiterungen oder Vereinfachungen an Ausdrücken („expressions“) nicht berücksichtigt werden.

Die Metrik kann auch bei Aufgaben mit vorgegebenen Codevorlagen verwendet werden, da die Anzahl der dort vorgegebenen Anweisungen ebenfalls gemessen und gegebenenfalls vom Ergebnis für studentische Lösungen abgezogen werden kann. Ferner können die Kennzahlen didaktisch verwendet werden, da die Vermeidung unnötig aufwändiger und umfangreicher Lösungen zu den Lernzielen einer Veranstaltung gehören kann. Zudem hilft die Metrik zum Beispiel bei der Bestimmung der Schwierigkeit einer Aufgabe, da mehr Anweisungen im Programmcode in der Regel einen erhöhten Zeitaufwand bei der Erstellung der Lösung bedeuten.

Halstead-Metriken

Die nach ihrem Autor benannten Halstead-Metriken (Halstead, 1977) basieren ebenfalls auf der Messung der Länge eines Programms, ziehen aber zusätzlich noch die Menge der verschiedenen verwendeten Operatoren und Operanden in Betracht. Somit kann nicht nur der Umfang des Programms (Summe der Anzahl der Operanden und Operatoren), sondern auch der Umfang des verwendeten sogenannten Vokabulars (Anzahl unterschiedlicher Operanden und Operatoren) bestimmt werden. Beide Werte können verschieden miteinander in Beziehung gesetzt werden.

Bei der Analyse von Programmieraufgaben mit dieser Metrik ergibt sich das Problem, dass der Umfang des Vokabulars durch die Codevorlage nach unten begrenzt ist, auch wenn die Studierenden selber in ihrem Code ein kleineres Vokabular verwenden. Bei der Berechnung der Metrik müsste daher sorgfältig zwischen dem vorgegebenen Code und dem von Studierenden erstellten Code getrennt werden, was je nach Komplexität der Codevorlage nicht immer sauber möglich ist. Zudem ist nicht offensichtlich, ob der Umfang des Vokabulars in direkten Bezug zu einem möglichen Lernziel gesetzt werden kann. Bestenfalls kann noch angenommen werden, dass den Studierenden mit fortschreitenden Programmierkenntnissen ein größeres Vokabular bekannt ist, dass bei schwierigen Aufgaben dann auch zum Einsatz kommt. Da Aufgaben jedoch oft auch auf ein begrenztes Thema (z.B. Implementierung von Vererbungsstrukturen) abzielen, kann nicht einmal vorausgesetzt werden, dass fortgeschrittene Aufgaben tatsächlich ein umfangreiches Vokabular erfordern.

Zyklomatische Komplexität

Die zyklomatische Komplexität (McCabe, 1976) ist eine Metrik zur Komplexität des Kontrollflussgraphen eines Programms. Sie basiert auf der Anzahl der Knoten und Kanten dieses Graphen und damit auf der Entscheidungsstruktur des Programms. Das Ergebnis entspricht der maximalen Anzahl linear unabhängiger Pfade eines Programms und ist damit insbesondere unabhängig vom Hinzufügen oder Entfernen von Anweisungen, die keine Entscheidungsknoten darstellen. Sie gehört damit zu den Metriken, die die logische Struktur eines Programms unabhängig von seinem Umfang beurteilen.

Wie die oben diskutierte Anzahl der Anweisungen kann auch diese Metrik bei Aufgaben mit vorgegebenen Codevorlagen verwendet werden, indem die Komplexität dieser Vorlagen vorab bestimmt wird. Da die Vermeidung unnötig komplexer Lösungen zu den Lernzielen einer Veranstaltung gehören kann, lassen sich die Ergebnisse der Metrik auch direkt verwenden.

Die Metrik lässt sich auch ohne Aufbau des kompletten Kontrollflussgraphen eines Programms berechnen, indem alle `if`-Abfragen, bedingten Ausdrücke, Schleifen, `case`-Anweisungen, `catch`-Anweisungen (für Java) und logische Operatoren gezählt werden.

Objektorientierte Metriken

Es gibt weitere, speziell auf die Messung typischer Eigenschaften objektorientierter Programme ausgerichtete Metriken, z.B. zur Kopplung zwischen Klassen, Kohäsion innerhalb von Klassen oder Sichtbarkeit von vererbten Methoden (e Abreu u. Carapuça, 1994; Chidamber u. Kemerer, 1994). Da einige dieser Metriken so definiert sind, dass sie vom Umfang eines Programms unabhängig sind, lassen sie sich zur Bewertung der Komplexität und logischen Struktur eines Programms verwenden.

Ähnlich wie bei den oben diskutierten Halstead-Metrik besteht die Schwierigkeit, dass die ermittelten Merkmale bei Programmieraufgaben durch Codevorlagen weitgehend vorgegeben sein könnten. Gerade in Anfängervorlesungen ist es üblich, z.B. Felder einer Klasse und Methodensignaturen vorzugeben und nur die Methodenrümpfe programmieren zu lassen. Zudem sind solche Aufgaben in der Regel so begrenzt, dass die zu erstellenden Beziehungen zwischen Klassen überschaubar sind und die Studierenden kaum Wahlmöglichkeiten haben. Erst bei Aufgaben mit deutlich mehr Freiheit für die Studierenden können diese Metriken daher sinnvoll angewandt werden. Dann allerdings lassen sie sich gut in Beziehung zu einzelnen Lernzielen setzen, da zum Beispiel die Erzielung einer hohen Kohäsion ein direktes Thema des objektorientierten Designs sein kann.

Fallbeispiele

Im Folgenden wird an einigen Fallbeispielen diskutiert, wie konkrete Fragestellungen aus dem Aufgabenent-

wurf für eine Lehrveranstaltung durch den Einsatz von Metriken beantwortet werden können. Als Metriken kommen dabei stets die zyklomatische Komplexität und die Anzahl der Anweisungen zum Einsatz, d.h. eine Kombination aus Umfangs- und Komplexitätsbeobachtung. Auf die Verwendung von speziellen objektorientierten Metriken wurde verzichtet, da alle untersuchten Daten aus einer Anfängervorlesung stammen.

Alle Fallbeispiele beziehen sich auf Aufgaben aus der Erstsemestervorlesung „Programmierung“ im Wintersemester 2011/12 . Bei allen Aufgaben wurde die Einreichung von Lösungen in der Programmiersprache Java erwartet und den Studierenden war es erlaubt, beliebig viele Lösungen zu einer Aufgabe einzureichen. Bei einzelnen Aufgaben kamen so bis zu 1400 Lösungen zusammen, wobei im Schnitt 3,6 Lösungen pro Studierendem eingingen. Für die Analysen wurden solche Lösungen ausgefiltert, bei denen die Metriken aufgrund von syntaktischen Fehlern im Programmcode nicht bestimmt werden konnten. Duplikate von Lösungen wurden nicht ausgefiltert, führen aber zwangsläufig zu identischen Ergebnissen in den Metriken. Die Einreichung der Lösungen durch die Studierenden und die Berechnung der Kennzahlen aus den Metriken erfolgte über das Werkzeug JACK (Striewe u. a., 2009). Die Kennzahlen wurden den Studierenden nicht mitgeteilt, sondern nur für die rückblickende Analyse nach dem Semester verwendet.

Freiheitsgrade von Aufgaben

Beim Stellen von Übungs- und Prüfungsaufgaben ist relevant, wie viele Freiheitsgrade die Aufgaben den Studierenden beim Erstellen einer Lösung lassen. Je enger begrenzt die Aufgabenstellungen sind, desto leichter sind sie in der Regel zu korrigieren, da sich die Lehrenden in weniger verschiedene Lösungsansätze hineindenken müssen. Welche didaktischen Vor- und Nachteile sich aus eng begrenzten oder freieren Aufgaben ergeben und in welcher Phase des Lernprozesses welche Aufgaben am besten geeignet sind, soll hier nicht weiter diskutiert werden. Stattdessen soll beleuchtet werden, wie durch die Anwendung von Softwareproduktmetriken für Umfang und Komplexität überhaupt gemessen werden kann, welchen Freiheitsgrad eine Aufgabe hat.

Die Abbildungen 1 bis 3 zeigen drei Verteilungsdiagramme für Lösungen zu drei Übungsaufgaben, die in der zweiten Hälfte des Wintersemesters 2011/12 gestellt wurden und die durch die Studierenden als Hausaufgaben bearbeitet wurden. Aufgetragen sind jeweils Kreise, deren Position sich aus der Anzahl der Anweisungen und der zyklomatischen Komplexität ergibt. Je mehr Lösungen auf dieselbe Position fallen, umso größer ist der dort dargestellte Kreis. Dabei wird unterschieden, ob die Lösung als bestanden gewertet wurde oder nicht. Insbesondere ist zu beachten, dass unterschiedliche Lösungen dieselben Kennzahlen aufweisen können und an derselben Position sowohl bestandene als auch nicht bestandene Lösungen liegen können.

Es ist zu erkennen, dass die erste Übungsaufgabe (Abbildung 1; im Folgenden entsprechend der Benennung in der Lehrveranstaltung als „Miniprojekt 4“ bezeichnet) im Vergleich zu den anderen Aufgaben den geringsten Freiheitsgrad aufweist. Fast alle Lösungen liegen in einem kompakten Bereich zwischen 200 und etwas mehr als 300 Anweisungen und weisen einen Komplexitätswert zwischen knapp 70 und 100 auf. Die untere Grenze des Bereichs ergibt sich dabei automatisch aus Umfang und Komplexität der bereitgestellten Codevorlage.

Einen deutlich höheren Freiheitsgrad weist Miniprojekt 5 (Abbildung 2) auf. Umfang und Komplexität der Codevorlage entsprachen zwar in etwa den Werten aus Miniprojekt 4, aber die Lösungen verteilen sich in einem deutlich größeren Raum. Bis zu einem Umfang von 700 Anweisungen und einem Komplexitätswert von 200 ist der gesamte Lösungsraum abgedeckt, jedoch in einem ähnlich schmalen Band wie bei Miniprojekt 4. Dass es sich bei beiden Aufgaben um echte Freiheitsgrade und nicht nur die notwendige Differenz zwischen unvollständiger Codevorlage und minimaler korrekter Lösung handelt zeigt sich dadurch, dass es in beiden Aufgaben bereits im unteren Drittel der gemessenen Werte bestandene Lösungen gibt.

Dieselbe Beobachtung gilt auch für Miniprojekt 6 (Abbildung 3), auch wenn hier weniger hohe Werte für die Zahl der Anweisungen erreicht werden. Bemerkenswert ist hier jedoch die etwas breitere Streuung des Korridors, in dem die Lösungen liegen. Während in den ersten beiden betrachteten Aufgaben also eine recht starke Korrelation zwischen Umfang und Komplexität galt, lässt diese Aufgabe den Studierenden mehr Freiheiten zu umfangreicheren, aber weniger komplexen Lösungen beziehungsweise kleineren, aber komplexeren Ansätzen.

Bis hierhin kann also festgehalten werden, dass die Messung der Zahl der Anweisungen und der zyklomatischen Komplexität es erlaubt, Aussagen über den Freiheitsgrad einer Aufgabe zu treffen und somit gegebenenfalls gezielt Aufgaben auszuwählen.

Als Nebenbemerkung ist in allen drei Fällen zu beobachten, dass es in der „oberen“ Hälfte der Verteilung (d.h. in der jeweils oberen Hälfte des Wertebereichs der beiden verwendeten Metriken) je einen Punkt mit einer besonders hohen Zahl von Lösungen gibt, der bis zu knapp 20% aller Lösungen repräsentiert. Für diese auffällige Häufung konnten zwei Ursachen ermittelt werden: Zum einen konnten sich Studierende während der Bearbeitungsphase der Aufgabe Ratschläge und Hilfestellung von Tutoren holen, die sich ihrerseits an der Musterlösung einer Aufgabe orientierten. Die Merkmale dieser Musterlösung lassen sich in der Verteilung der studentischen Lösungen entsprechend

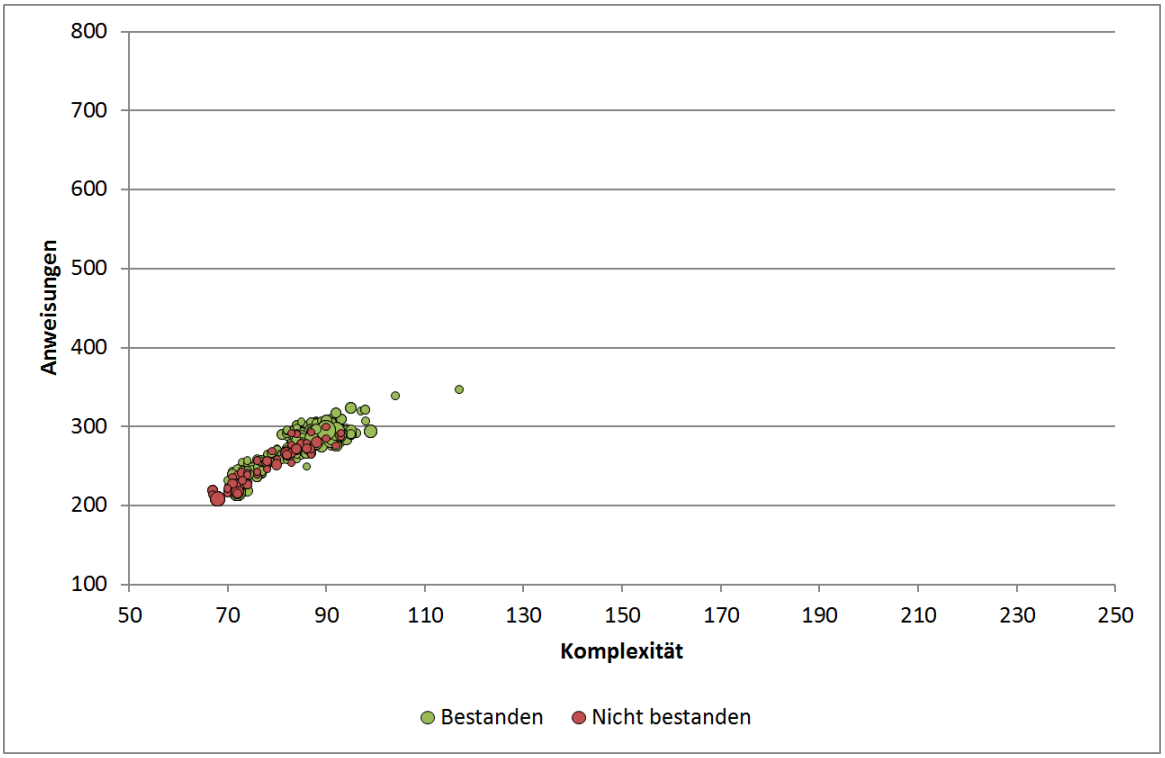


Abbildung 1: Verteilungsdiagramm für 1309 Lösungen zu einer Übungsaufgabe („Miniprojekt 4“). Größere Kreise bedeuten mehr Lösungen mit denselben metrischen Kennzahlen. Es wurden maximal 43 Lösungen mit denselben Kennzahlen eingereicht.

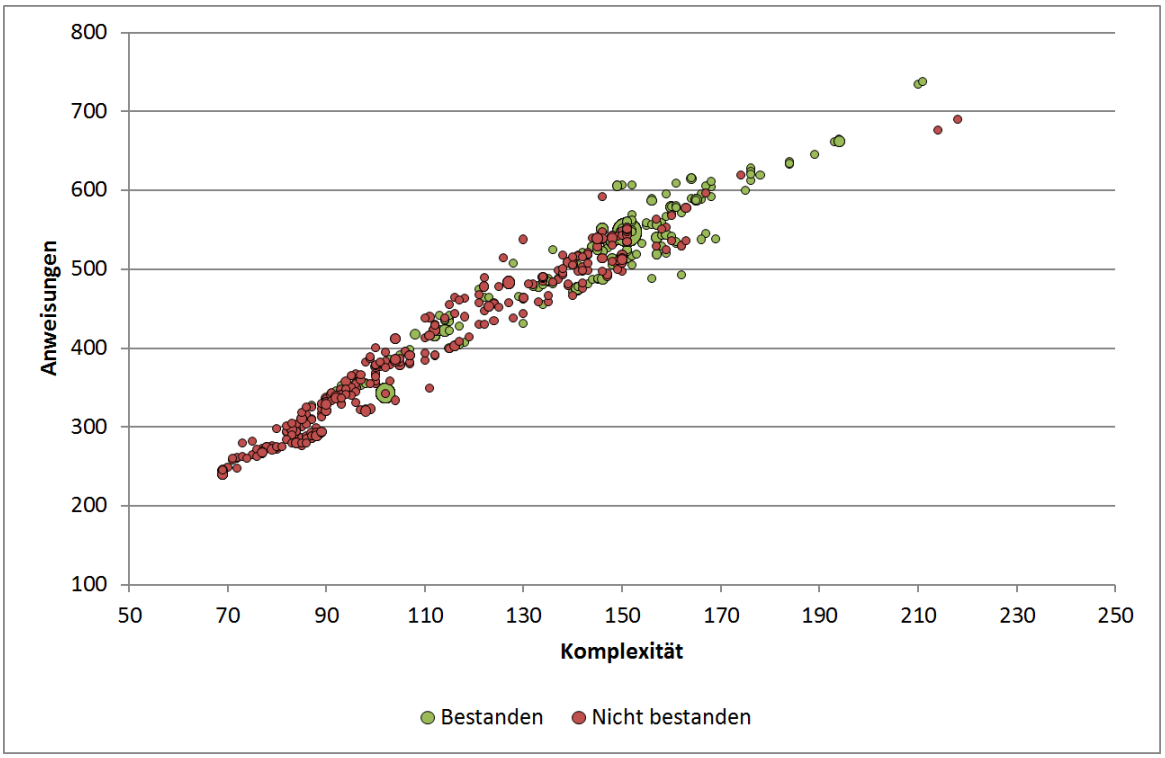


Abbildung 2: Verteilungsdiagramm für 815 Lösungen zu einer Übungsaufgabe („Miniprojekt 5“). Größere Kreise bedeuten mehr Lösungen mit denselben metrischen Kennzahlen. Es wurden maximal 155 Lösungen mit denselben Kennzahlen eingereicht.

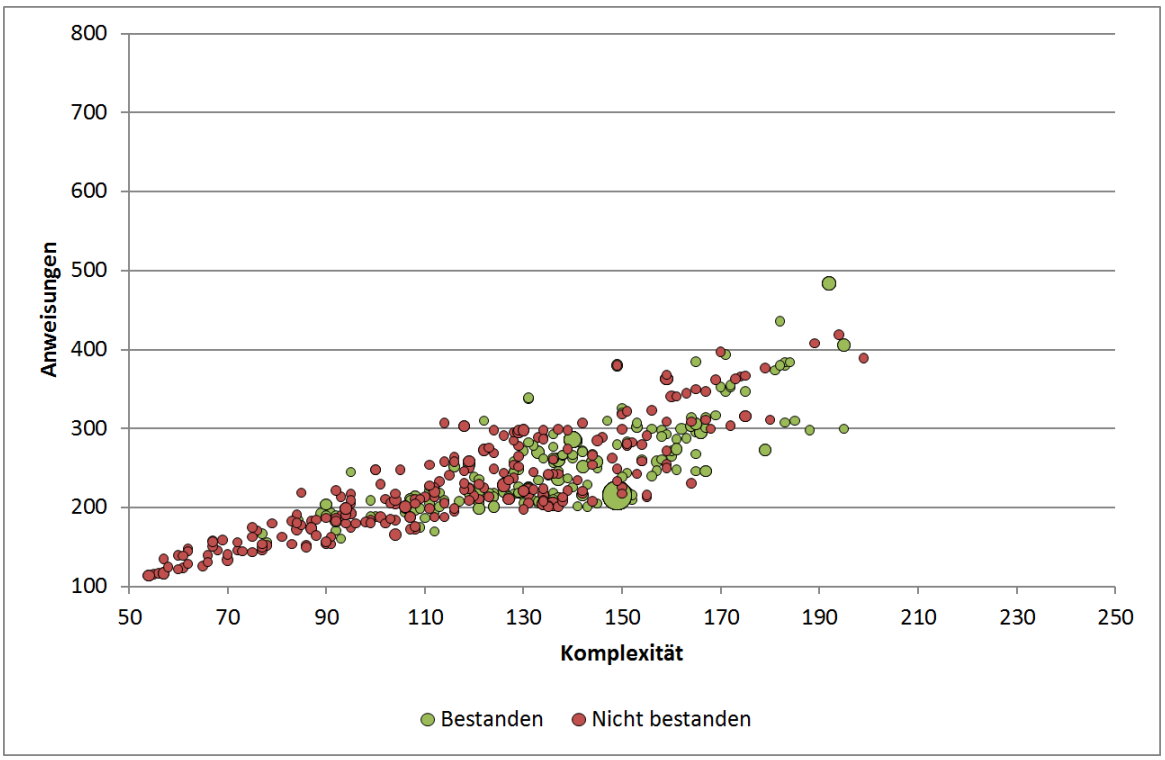


Abbildung 3: Verteilungsdiagramm für 600 Lösungen zu einer Übungsaufgabe („Miniprojekt 6“). Größere Kreise bedeuten mehr Lösungen mit denselben metrischen Kennzahlen. Es wurden maximal 85 Lösungen mit denselben Kennzahlen eingereicht.

wiederfinden. Zum anderen kursierten in den studentischen Internetforen inoffizielle Musterlösungen, an denen sich ebenfalls Studierende orientiert haben. Wie oben beschrieben, wurden die Daten nicht um Plagiate bereinigt. Stichproben zeigten jedoch, dass es sich bei den Anhäufungen nicht ausschließlich um Plagiate einer einzelnen Lösung handelt.

Relatives Aufgabenniveau

Das didaktische Konzept der untersuchten Lehrveranstaltung sah vor, dass zu jedem der oben diskutierten Miniprojekte eine kleinere Prüfungsaufgabe in Form eines Testats gestellt wird. Dieses war jeweils innerhalb von 45 Minuten im PC-Pool der Universität unter Prüfungsbedingungen zu bearbeiten. Da der PC-Pool nicht ausreichend viele Plätze bietet, um alle Studierenden gleichzeitig zu prüfen, mussten die Testate in Gruppen abgenommen werden. Daher mussten stets mehrere Aufgabenvarianten erstellt werden, damit Teilnehmer der ersten Gruppe nicht Details der Aufgabenstellung an die späteren Gruppen weitergeben konnten. Aus diesem Szenario ergeben sich zwei Anforderungen, die die Überprüfung eines relativen Aufgabenniveaus sinnvoll erscheinen lassen: Erstens muss sichergestellt sein, dass durch die verschiedenen Varianten keine Gruppe benachteiligt wird, indem ihre Aufgabe mehr Aufwand erfordert als die Aufgaben anderer Gruppen. Zweitens muss sichergestellt sein, dass die Testataufgaben insgesamt in einem angemessenen Verhältnis zum zugehörigen Miniprojekt stehen und tatsächlich einen Ausschnitt aus diesem bilden.

Um den Studierenden den Einstieg in die Testataufgaben zu erleichtern und zudem grobe Fehler beim Stellen der Testataufgaben zu vermeiden, orientierten sich diese sehr eng an den zugehörigen Miniprojekten. Es kam stets eine sehr ähnliche Codevorlage zum Einsatz und auch der narrative Kontext der Aufgabe war stets identisch mit dem des Miniprojektes. Der eigentliche Kern der Aufgabe war den Studierenden jedoch nicht vorab bekannt, d.h. es musste im Testat immer Funktionalität implementiert werden, die im Miniprojekt nicht diskutiert wurde. Daraus ergeben sich auch die Risiken bei der Aufgabenstellung, die es durch den Einsatz von Metriken abzuschätzen gilt: Zum einen können unterschiedliche Funktionalitäten mehr oder weniger Aufwand erfordern, was einzelnen Gruppen Vor- bzw. Nachteile bringt und zum anderen kann die Funktionalität in Relation zum Miniprojekt eher aufwendig oder eher einfach zu implementieren sein. Die konzeptionelle Arbeit, die von den Studierenden beim Entwurf ihrer Lösung erbracht werden muss, lässt sich freilich über die Metriken des Ergebnisses nicht bestimmen und folglich in den Betrachtungen auch nicht berücksichtigt werden.

Die Abbildungen 4 und 5 zeigen die Verteilungsdiagramme zu zwei Varianten von Testat 4. Die Verteilung der dritten Variante, an der deutlich weniger Studierende teilnahmen, sieht vergleichbar aus und wir aus

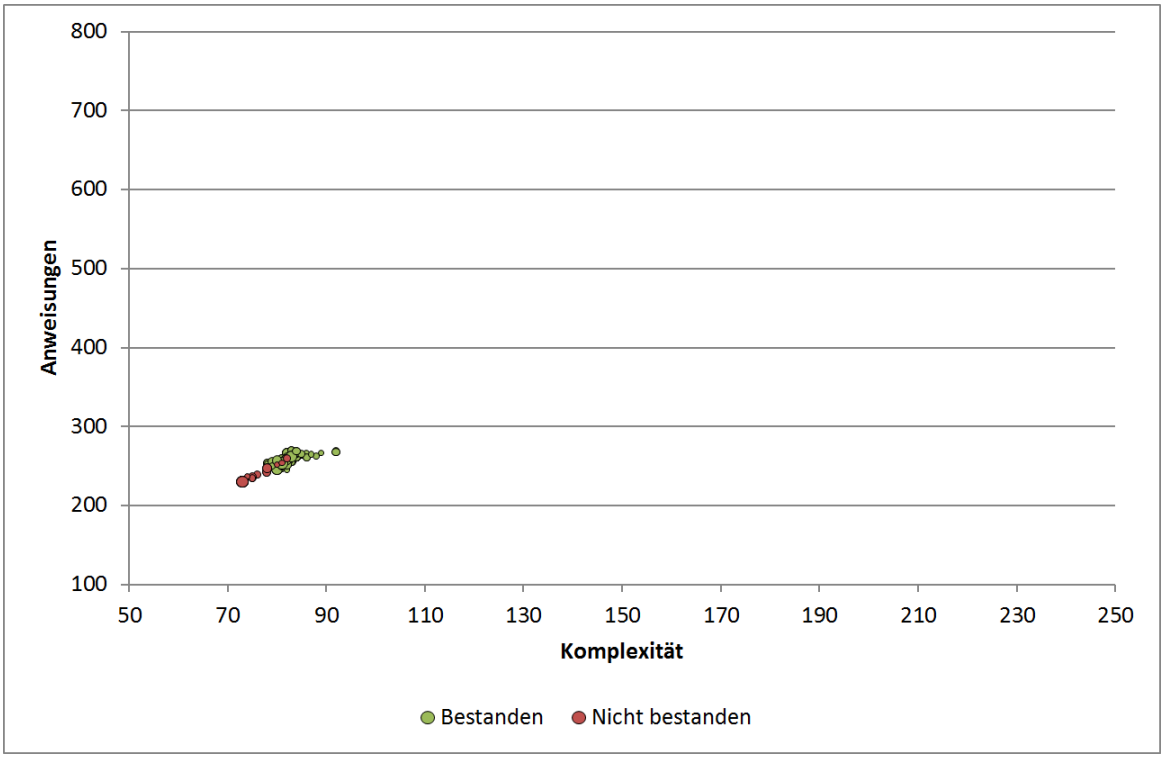


Abbildung 4: Verteilungsdiagramm für 378 Lösungen zu einer Prüfungsaufgabe („Testat 4, Variante 1“). Die Aufgabe griff die Aufgabenstellung aus Miniprojekt 4 (siehe Abbildung 1) auf. Zur Verteilung einer zweiten Aufgabenvariante vgl. Abbildung 5.

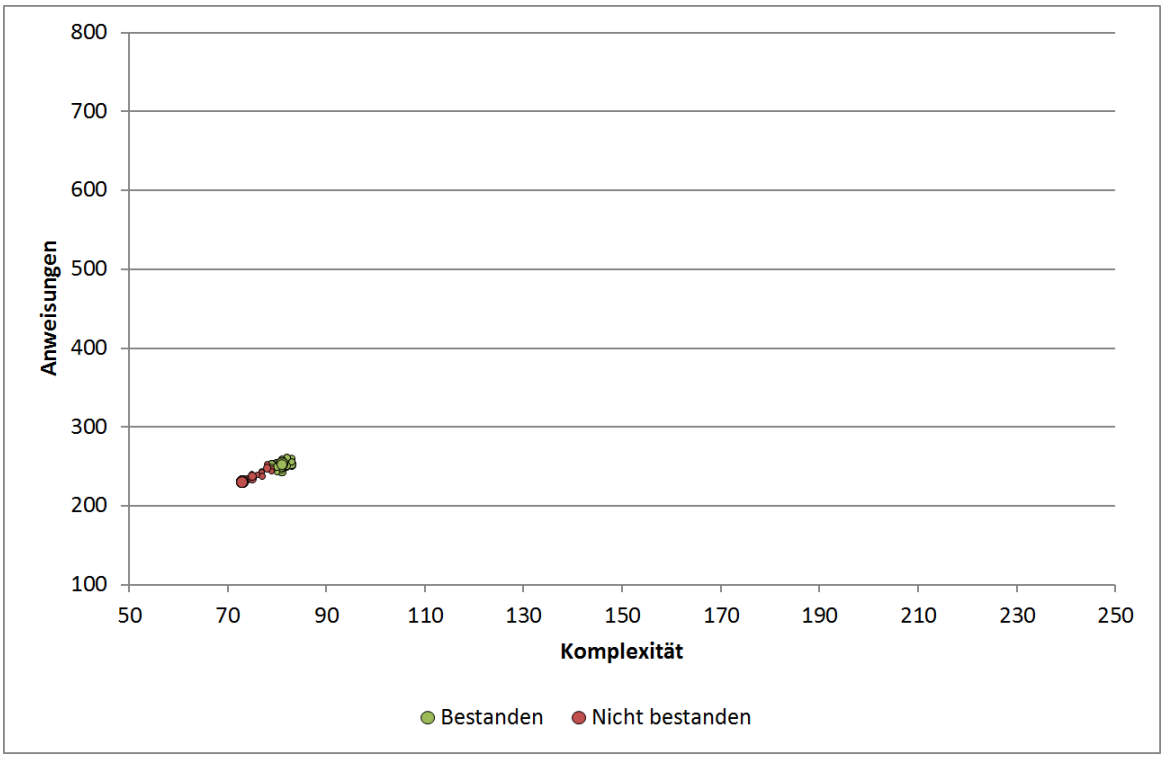


Abbildung 5: Verteilungsdiagramm für 298 Lösungen zu einer Prüfungsaufgabe („Testat 4, Variante 2“). Die Aufgabe griff die Aufgabenstellung aus Miniprojekt 4 (siehe Abbildung 1) auf. Zur Verteilung der ersten Aufgabenvariante vgl. Abbildung 4.

Platzgründen nicht dargestellt. Es ist zu erkennen, dass beide Varianten im Kern dieselbe Verteilung aufweisen. Insbesondere gibt es in beiden Aufgaben einen sehr schmalen Korridor nicht bestandener Lösungen im „unteren“ Bereich und eine größere Anhäufung bestandener Lösungen im „oberen“ Bereich der Verteilung. Es kann daher davon ausgegangen werden, dass die beiden Aufgaben zumindest im Bezug auf den Umfang und die Komplexität der Lösung identische Anforderungen an die Studierenden gestellt haben. Wie oben bereits angesprochen, kann diese Aussage nur dann auf das Aufgabenniveau insgesamt erweitert werden, wenn vorausgesetzt werden kann, dass die Herleitung der Lösung für beide Gruppen gleich schwierig ist. Bei der Interpretation der Verteilungen ist es also insbesondere als deutliches Warnsignal zu werten, wenn bei als gleichwertig vorgesehenen Aufgaben deutliche Unterschiede in den Metriken auftreten. Die Gleichheit von Umfang und Komplexität muss dagegen nicht zwangsläufig auch ein identisches Aufgabenniveau bedeuten.

Die Frage nach dem Verhältnis der Testataufgaben zum Miniprojekt kann durch einen Vergleich mit Abbildung 1 beziehungsweise durch einen Vergleich der Kennzahlen beantwortet werden: Der Komplexitätswert liegt mit 70 bis 90 nahezu mittig in den Grenzen des Miniprojektes und die Zahl der Anweisungen mit Werten zwischen 230 und 270 ebenfalls mittig im Rahmen des Miniprojektes. Es kann daher davon ausgegangen werden, dass die Testataufgaben bezogen auf Umfang und Komplexität sehr genau das Aufgabenniveau des Miniprojektes getroffen haben.

Für das Testat 5 (Abbildung 6; hier betrachtet an der dritten Aufgabenvariante dieses Testats) lässt sich eine leicht abweichende Beobachtung treffen: Der von der Verteilung abgedeckte Bereich liegt zwar im Korridor von Miniprojekt 5 und weißt (bei deutlicher kleiner Größe) eine ähnliche Form auf, liegt aber nicht mittig innerhalb der Werte des Miniprojektes, sondern deutlich nach „oben“ verschoben. Relativ zur Schwierigkeit des Miniprojektes kann hier also ein höheres Aufgabenniveau angenommen werden.

Auch der Vergleich der Testate 4 und 5 untereinander kann unter dem Gesichtspunkt des relativen Aufgabenniveaus angestellt werden. Die Verteilungen beider Testate weisen im Diagramm eine ähnliche Form auf und decken in etwa dieselbe Fläche ab, wobei Testat 5 etwas breiter gestreut ist. Aufgrund des deutlichen Abstands der Cluster kann trotzdem angenommen werden, dass Testat 5 ein höheres Niveau aufweist als Testat 4.

Da die Lehrenden nicht zwangsläufig dasselbe Bewertungsschema für Miniprojekte und Testate anlegten, lassen sich diese Annahmen nicht unmittelbar über die im Mittel erreichten Punktzahlen bestätigen oder widerlegen: Im Miniprojekt 4 wurden im arithmetischen Mittel 71,29 von 100 möglichen Punkten erreicht; im Testat 4 über alle drei Varianten im arithmetischen Mittel 63,38 Punkte. Im Miniprojekt 5 wurden im arithmetischen Mittel 48,57 Punkte erreicht; in allen drei Varianten der Testate im arithmetischen Mittel 45,28 Punkte. Das vermutete höhere relative Aufgabenniveau bei Testat 5 im Vergleich zu Miniprojekt 5 spiegelt sich also nicht in der Veränderung der Punktzahlen wider. Lediglich im Vergleich der beiden Testate untereinander tritt die erwartete Punkteänderung auf.

Die Schwierigkeit der Bestimmung des relativen Aufgabenniveaus unterstreicht auch Abbildung 7, die die Verteilung für die erste Variante von Testat 6 widergibt. Bei den Komplexitätswerten liegt diese in der Mitte des Bereichs für Miniprojekt 6; bei der Anzahl der Anweisungen im unteren Bereich. Im Vergleich mit Testat 4 weist sie insbesondere eine höhere Komplexität, aber einen geringeren Umfang der Lösungen aus. Im Bezug auf Miniprojekt 6 kann also ein leichteres Niveau angenommen werden, während im Bezug auf Testat 4 unklar ist, ob die höhere Komplexität oder der geringere Umfang mehr zum Niveau der Aufgabe beiträgt. In Miniprojekt 6 wurde im arithmetischen Mittel 50,84 Punkte erreicht, in allen Testatvarianten zusammen im arithmetischen Mittel 65,34 Punkte. Dies legt nahe, dass das Testat wie erwartet ein niedrigeres Niveau hatte als das Miniprojekt und im übrigen in etwa gleichwertig zu Testat 4 war.

Aus diesen Überlegungen lässt sich also festhalten, dass die Messung der Zahl der Anweisungen und der zyklomatischen Komplexität es erlaubt, Aussagen über das relative Niveau einer Aufgabe zu machen. Dies könnte in E-Learning-Systemen beispielsweise genutzt werden, um automatisch den Schwierigkeitsgrad von Aufgaben zu bestimmten und Studierenden damit gezielt schwierigere oder leichtere Aufgaben anbieten zu können. Noch einmal sei an dieser Stelle darauf hingewiesen, dass es andere Einflussfaktoren auf das Niveau einer Aufgabe gibt, die sich über die diskutierten Metriken nicht abbilden lassen, so dass die Verwendung von Softwareproduktmetriken trotz ihrer Nützlichkeit nicht als alleiniges Mittel zum Einsatz kommen sollte. Naheliegend ist beispielsweise die Erweiterung des Ansatzes auf Softwaremetriken, die nicht das Produkt, sondern den Erstellungsprozess messen und somit beispielsweise Auskunft darüber geben können, über welchen Zeitraum hinweg oder in welchen Einzelschritten eine Lösung erstellt wurde. Auch aus diesen Metriken kann eine Aussage über das Niveau einer Aufgabe erwartet werden.

Fazit und Ausblick

In diesem Beitrag wurde anhand von Fallbeispielen untersucht, welche Aussagen über Programmieraufgaben aus der Analyse ihrer Lösungen mit Softwareproduktmetriken gewonnen werden können. Es konnte festgestellt werden, dass mithilfe von Metriken zum Umfang und zur Komplexität von Programmen Aussagen über den Freiheitsgrad einer Aufgabe sowie das

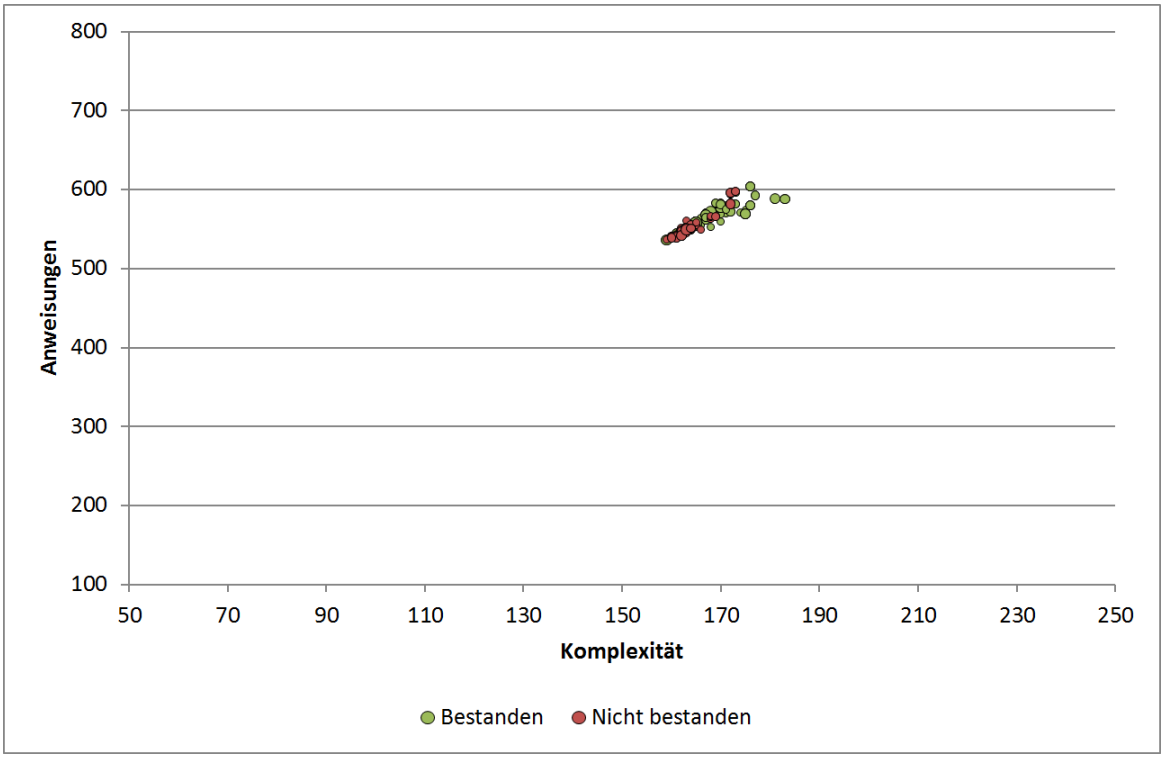


Abbildung 6: Verteilungsdiagramm für 162 Lösungen zu einer Prüfungsaufgabe („Testat 5, Variante 3“). Die Aufgabe griff die Aufgabenstellung aus Miniprojekt 5 (siehe Abbildung 2) auf.

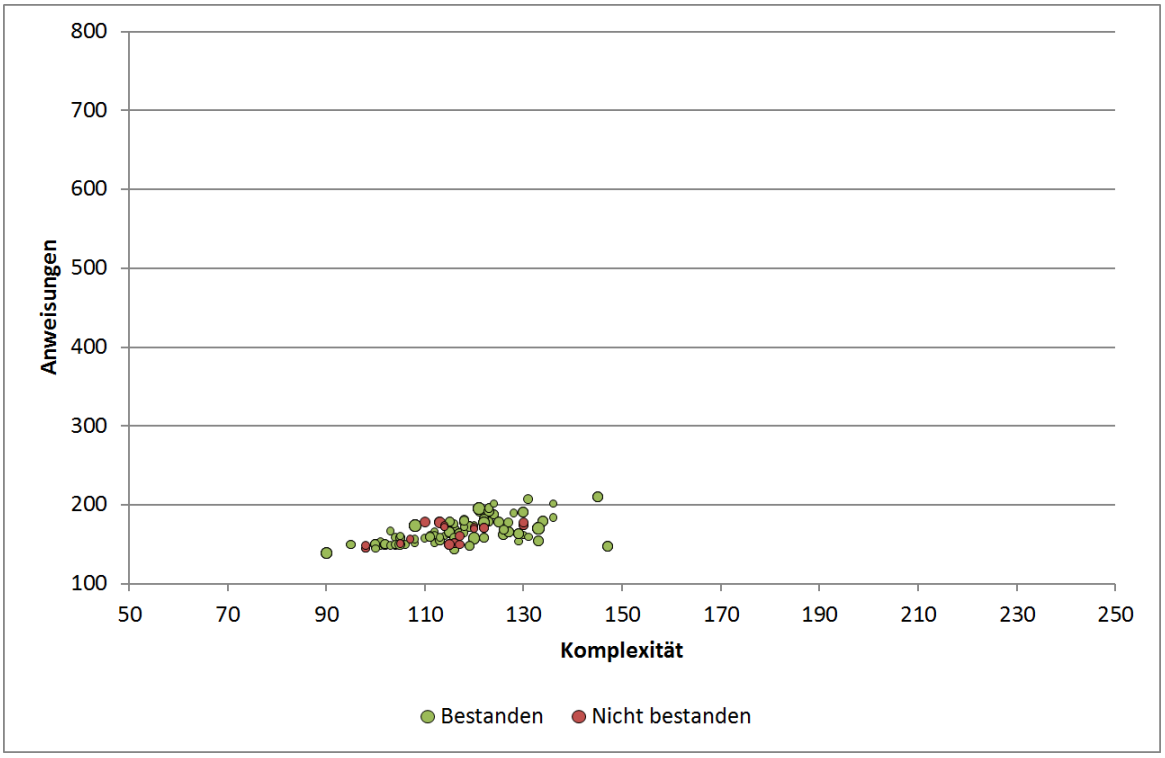


Abbildung 7: Verteilungsdiagramm für 222 Lösungen zu einer Prüfungsaufgabe („Testat 6, Variante 1“). Die Aufgabe griff die Aufgabenstellung aus Miniprojekt 6 (siehe Abbildung 3) auf.

Niveau der Aufgabe im Vergleich zu anderen Aufgaben getroffen werden können.

Daraus ergeben sich unmittelbare Einsatzmöglichkeiten in Werkzeugen zur Lehrunterstützung sowie weitere Forschungsansätze: Die gewonnenen Daten können unmittelbar genutzt werden, um Aufgaben zu klassifizieren und somit in einem E-Learning-System (teil-)automatisiert zur Bearbeitung vorzuschlagen. Im einfachsten Fall könnten Studierende dabei ein System explizit nach leichteren oder schwierigeren Aufgaben fragen. Daraus ergibt sich auch ein unmittelbarer Forschungsansatz, indem die Studierenden anschließend um eine (subjektive) Einschätzung der Schwierigkeit gebeten werden. Kennzahlen, Ergebnissen und subjektive Bewertungen können dann zur weiteren Validierung der Erkenntnisse dieses Artikels miteinander verglichen werden. Auch die oben diskutierten weiteren Einflussfaktoren sollten in diesen Betrachtungen Berücksichtigung finden.

Ferner kann untersucht werden, ob aus der Position einer einzelnen Lösung in Relation zum Verteilungsdiagramm der jeweiligen Aufgabe weitere Erkenntnisse, beispielsweise zur Generierung individueller Rückmeldungen an die Studierenden, gewonnen werden können. Über den Einsatz von Softwareproduktmetriken hinaus können dabei auch weitere Metriken zum Entwicklungsprozess der Lösung zum Einsatz kommen, mit denen beispielsweise eine Folge von mehreren Einreichungen mit inkrementellen Verbesserungen der Lösung untersucht wird. Dies würde auch die Bereiche der Analyse abdecken, die in diesem Artikel nicht besprochen wurden.

Danksagung Die Autoren danken Alexander Jung für seine umfangreiche Recherche zu Softwareproduktmetriken im Rahmen seiner Bachelor-Arbeit.

Literatur

[e Abreu u. Carapuça 1994] ABREU, Fernando B. ; CARAPUÇA, Rogério: *Object-Oriented Software Engineering: Measuring and Controlling the Development Process*. 1994

[Chidamber u. Kemerer 1994] CHIDAMBER, Shyam R. ; KEMERER, Chris F.: A metrics suite for object oriented design. In: *IEEE Transactions on Software Engineering* 20 (1994), jun, Nr. 6, S. 476 –493. http://dx.doi.org/10.1109/32.295895. – DOI 10.1109/32.295895. – ISSN 0098–5589

[Conte u. a. 1986] CONTE, Samuel D. ; DUNSMORE, Hubert E. ; SHEN, Vincent Y.: *Software engineering metrics and models*. Redwood City, CA, USA : Benjamin-Cummings Publishing Co., Inc., 1986. – ISBN 0–8053–2162–4

[DeMarco 1986] DEMARCO, Tom: *Controlling Software Projects: Management, Measurement, and Estimates*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1986. – ISBN 0131717111

[Fenton u. Pfleeger 1998] FENTON, Norman E. ; PFLEEGER, Shari L.: *Software Metrics: A Rigorous and Practical Approach*. 2nd. Boston, MA, USA : PWS Publishing Co., 1998. – ISBN 0534954251

[Gross u. a. 2012] GROSS, Sebastian ; MOKBEL, Bassam ; HAMMER, Barbara ; PINKWART, Niels: Feedback Provision Strategies in Intelligent Tutoring Systems Based on Clustered Solution Spaces. In: DESEL, Jörg (Hrsg.) ; HAAKE, Joerg M. (Hrsg.) ; SPANNAGEL, Christian (Hrsg.): *DeLFI 2012: Die 10. e-Learning Fachtagung Informatik*. Hagen, Germany, 2012. – ISBN 978–3885796015, S. 27–38

[Halstead 1977] HALSTEAD, Maurice H.: *Elements of software science*. Elsevier, 1977

[Kaner u. Bond 2004] KANER, Cem ; BOND, Walter P.: Software Engineering Metrics: What Do They Measure and How Do We Know? In: *In METRICS 2004. IEEE CS*, Press, 2004

[Leach 1995] LEACH, Ronald J.: Using metrics to evaluate student programs. In: *SIGCSE Bull.* 27 (1995), S. 41–43. http://dx.doi.org/10.1145/201998.202010. – DOI 10.1145/201998.202010. – ISSN 0097–8418

[Martín u. a. 2009] MARTÍN, David ; CORCHADO, Emilio ; MARTICORENA, Raúl: A Code-comparison of Student Assignments based on Neural Visualisation Models. In: CORDEIRO, José A. M. (Hrsg.) ; SHISHKOV, Boris (Hrsg.) ; VERBRAECK, Alexander (Hrsg.) ; HELFERT, Markus (Hrsg.) ; INSTICC (Veranst.): *Proceedings of the First International Conference on Computer Supported Eductation (CSEDU), 23 - 26 March 2009, Lisboa, Portugal* Bd. 1 INSTICC, INSTICC Press, 2009. – ISBN 978–989–8111–82–1, S. 47–54

[McCabe 1976] MCCABE, Thomas J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering,* 2 (1976), Nr. 4, S. 308–320. http://dx.doi.org/10.1109/TSE.1976.233837. – DOI 10.1109/TSE.1976.233837. – ISSN 0098–5589

[Mengel u. Yerramilli 1999] MENGEL, Susan A. ; YERRAMILLI, Vinay: A case study of the static analysis of the quality of novice student programs. In: *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*. New York, NY, USA : ACM, 1999 (SIGCSE '99). – ISBN 1–58113–085–6, S. 78–82

[Striewe u. a. 2009] STRIEWE, Michael ; BALZ, Moritz ; GOEDICKE, Michael: A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In: *Proceedings of the First International Conference on Computer Supported Eductation (CSEDU), 23 - 26 March 2009, Lisboa, Portugal* Bd. 2 INSTICC, 2009, S. 54–61

Agile Methoden im Unterricht

Iterativ-inkrementelle Vermittlung von Software-Engineering-Wissen

Veronika Thurner, Hochschule München
thurner@hm.edu

Zusammenfassung

Viele Ansätze der Software Engineering Lehre behandeln in sequenzieller Abfolge die einzelnen Wissensbereiche nacheinander. Sie verlaufen damit analog zum Wasserfallmodell, bei dem ebenfalls einzelne Disziplinen nacheinander erschöpfend abgearbeitet werden, wobei die Sinnhaftigkeit und der Beitrag der dabei erzielten Zwischenergebnisse zum Gesamtergebnis oft erst relativ spät im Projekt erkennbar wird.

In der Praxis komplexer Software Engineering Projekte haben sich heute iterativ-inkrementelle bzw. agile Ansätze gegenüber dem Wasserfallmodell durchgesetzt. Dabei wird in kleinen in sich abgeschlossenen Zyklen, in vielen überschaubaren Schritten und mit kurzen Feedback-Zyklen nach und nach ein System weiterentwickelt.

Der hier vorgestellte Lehr-/Lernansatz greift diese Idee auf und überträgt sie auf die Vermittlung von Software Engineering Wissen. Dabei wird von der ersten Lehreinheit an mit einer inkrementellen Folge von Beispielsystemen gearbeitet, die quasi bei null anfängt und sukzessive immer komplexer wird. Jedes der enthaltenen Systeme spiegelt dabei eine kleine, aber vollständige Iteration durch den Software Life Cycle wider, von der Anforderungsskizze über Entwurf und Implementierung bis hin zum Test.

Motivation

Software Engineering ist heute eine komplexe und sehr vielschichtige Disziplin. Entsprechend hoch sind die Anforderungen, die an ihre Ausführenden gestellt werden. So sind beispielsweise bereits für die Bewältigung kleiner, überschaubarer Projekte Kenntnisse und Kompetenzen in so unterschiedlichen Bereichen wie Analyse oder Implementierung erforderlich, verbunden mit Modellierungssprachen, Architekturprinzipien und aktuellen Frameworks für die konkrete Umsetzung. Neben diesen eher technischen Kerndisziplinen und Wissensbereichen werden darüber hinaus Fähigkeiten in Projektmanagement, Vorgehensmodellen etc. benötigt, sowie diverse grundlegende überfachliche Kompetenzen (Selbst-, Sozial- und Methodenkompetenzen).

Einen guten Überblick über das Lernspektrum für angehende Softwareingenieure vermittelt der Software Engineering Body of Knowledge (Abran u. a., 2005), der nach den 11 identifizierten Wissensbereichen der Version von 2004 in der aktuell entwickelten Version 3 auf 15 Wissensbereiche ausgebaut werden wird (IEEE, 2012). Diese Erweiterung des SWEBOK ist eines von vielen Indizien dafür, dass mit einer gravierenden Vereinfachung des Software Engineerings bis auf Weiteres erst mal nicht zu rechnen ist.

Den Lehrenden stellen der Umfang und die Komplexität der zu vermittelnden Domäne vor zwei zentrale Fragen. Die eine, angesichts der in der Regel streng begrenzten verfügbaren Ausbildungszeit unvermeidliche lauet:

- Was lasse ich weg?

Oder, positiver formuliert:

- Was ist die Essenz, die ich vermitteln muss/will?

Die andere, bei diesem Meer von untereinander abhängigen Wissensbereichen ähnlich drängende Frage ist:

- Wo fange ich an?

Zur ersten Frage existieren bereits diverse allgemeinere und konkretere Überlegungen, welche die zugrunde liegende Problematik zwar vielleicht noch nicht endgültig lösen, aber zumindest Handlungsspielräume aufzeigen (Lehner, 2009).

Systematische Überlegungen zur zweiten Frage nach dem richtigen Einstiegspunkt und einer sinnvollen Vermittlungs- bzw. Lernreihenfolge stehen im Fokus der vorliegenden Arbeit.

Hierfür wird zunächst die aktuelle Verteilung der Lerninhalte auf die einzelnen Fächer im Software-Engineering-Lernpfad dargestellt und beleuchtet, welche Probleme sich in der Lehr-/Lernpraxis aus der derzeit etablierten, am Wasserfallmodell orientierten Reihenfolge der inhaltlichen Vermittlung ergeben. Darauf aufbauend wird kurz die Idee vorgestellt, die Lerninhalte des Software Engineerings nach einem iterativen, inkrementellen Prozess zu vermitteln. Anschließend wird das Beispiel erläutert, anhand dessen die einzelnen Lerninhalte schrittweise eingeführt und von den Studierenden in die Praxis umgesetzt werden. Eine Analyse der ersten mit diesem Lehrkonzept gewonnenen Erfahrungen runden die Arbeit ab.

Problemstellung

Den groben Rahmen der Antwort auf die „Wo fange ich an?“-Frage definieren im Hochschul-Kontext in der Regel die Studienpläne.

Einbettung in das Curriculum

Wir betrachten hier als konkretes Beispiel den Studiengang Bachelor Wirtschaftsinformatik an der Hochschule München, der für den Bereich Software Engineering den folgenden Ausbildungspfad vorsieht:

- 1. Semester: Softwareentwicklung 1
  Einführung in das objektorientierte Programmierparadigma und die Grundzüge der Programmierung am Beispiel der Sprache Java
- 2. Semester: Softwareentwicklung 2
  Vertiefung der Programmierkenntnisse und Einsatz fortgeschrittener objektorientierter Programmierkonzepte am Beispiel der Sprache Java
- 3. Semester: Software Engineering 1
  Objektorientierte Analyse und Entwurf mit UML, Architekturauswahl, Aufwandsschätzung, Qualitäts- und Projektmanagement
- 4. Semester: Software Engineering 2
  Werkzeuge zur Automatisierung der Entwicklung, modellgetriebene Entwicklung, Konfigurationsmanagement, Verifikation und Test, Softwarearchitekturen, Prozessmodelle und agile Vorgehensmodelle

Jede dieser Veranstaltungen ist verpflichtend und umfasst je zwei Semesterwochenstunden seminaristischen Unterricht und zwei Semesterwochenstunden Praktikum. An den Praktikumsgruppen nehmen in der Regel ca. 20 Studierende teil. Der seminaristische Unterricht ist auf ca. 40–50 Studierende ausgelegt, wobei hier bedingt durch die aktuellen starken Jahrgänge zum Teil erheblich nach oben abgewichen werden muss.

Ergänzt werden diese Pflichtfächer durch ein umfangreiches Angebot an Wahlfächern, deren Spektrum von Personalführung bis Webtechniken reicht und die ab dem 4. Semester besucht werden können.

Auftretende Schwierigkeiten

Obiger Studienplan legt also fest, dass nach einer Grundausbildung in objektorientierter Programmierung in den Software-Engineering-Veranstaltungen zunächst im 3. Semester die frühen, eher modellierungsorientierten Disziplinen sowie Management-Themen behandelt werden, während die eher technischen, implementierungsnahen Disziplinen sowie Vorgehensmodelle erst im 4. Semester auf der Tagesordnung stehen.

Diese Aufteilung bewirkt, dass die Studierenden also ein Semster lang Anforderungen erfassen und analysieren, daraus Entwürfe ableiten und all das mit umfassenden UML-Modellen konkretisieren. Wie sich die dabei gefällten Entscheidungen und erstellten Spezifikationen letztendlich auf den Quelltext und das zu entwickelnde System auswirken bleibt dabei zunächst eine rein theoretische Überlegung, da der Round Trip zur technischen Umsetzung erst ein Semester später stattfindet.

Selbst wenn die Software-Engineering-Lehrveranstaltungen durchgängig über zwei Semester hinweg konzipiert und durchgeführt werden bleibt der große zeitliche Abstand zwischen Modellierung und Implementierung problematisch. Letztlich dauert es insgesamt zu lange, bis in den Köpfen der Studierenden ein rundes Bild der gesamten Disziplinen im Software Life Cycle entsteht, sodass Zusammenhänge und Abhängigkeiten zwischen den einzelnen Techniken und Disziplinen oft erst sehr spät erkannt werden. Dadurch sinkt nicht nur die Lernintensität, sondern auch der Spaßfaktor, da der Nutzen vieler Inhalte erst gegen Ende des Entwicklungszyklus deutlich wird.

Falls zwischen dem dritten und vierten Semester dann auch noch der/die Dozierende wechselt und im Praktikum ein anderes Anwendungsbeispiel verwendet kommt der/die Studierende unter Umständen überhaupt nicht an den Punkt, an dem selbsterstellte Spezifikationen zu einem laufenden System umgesetzt werden müssen. Dadurch fehlt ein sehr wesentlicher Erkenntnisschritt im gesamten Lernprozess, da die Studierenden nicht unmittelbar an der eigenen Arbeit erfahren, wie sich ihre bei der Modellierung gefällten Entscheidungen bei der Implementierung auswirken und wieviel Mehraufwand ggf. erforderlich ist, um Modellierungsfehler auszubügeln, die ihren Ursprung in Anforderungsanalyse und Entwurf haben, aber erst bei der Implementierung erkannt werden.

In der aktuellen Lehrpraxis wird daher mitunter bereits im modellierungslastigen Semester zusätzlich zur Erstellung eines komplett ausmodellierten Pflichtenheftes nebst vollständigen Entwurfsmodellen wenigstens eine rudimentäre Umsetzung von Teilen der Systemfunktionalität gefordert, auch wenn ein großer Teil der dafür benötigten Kenntnisse erst ein Semester später systematisch in der Lehrveranstaltung behandelt wird. Insbesondere implementierungsschwächere Studierende sind damit in der Regel völlig überfordert.

Die potenziellen Programmiergurus schaffen es dagegen bis zu einem gewissen Grad, sich durch entsprechende Rechere und Selbststudium die erforderlichen Implementierungstechniken anzueignen. Dafür machen sie aber im Gegenzug zum Teil erhebliche Abstriche bei der Modellierung, welche jedoch das eigentliche „offizielle“ Lernziel der ersten Software-Engineering-Lehrveranstaltung gewesen wäre. Außerdem wird Software Engineering 2 von diesem Personenkreis dann gerne als langweilig empfunden, weil die benötigten Inhalte ja schon zum großen Teil in Software Engineering 1 autodidaktisch erschlossen wurden.

Software Engineering Lehre nach dem Wasserfallmodell

Über ein ganzes Semester hinweg betrachtet sind viele Software Engineering Lehrveranstaltungen so aufgebaut, dass sie zunächst einen sehr groben Überblick über den Stoff des gesamten Kurses vermitteln. Anschließend werden Woche für Woche die zu behandelnden Themen abgearbeitet, in der Regel streng sequenziell hintereinander. Jedes Thema wird dabei umfassend und erschöpfend behandelt, bevor zum nächsten Thema vorangeschritten wird.

Beispielsweise lernen Studierende im Software Engineering auf diese Weise also zunächst alle Facetten der Use-Case-Modellierung kennen. Wenn diese abgeschlossen sind folgen Ablaufbeschreibungen mittels Aktivitätsdiagrammen. Danach werden Klassendiagramme durchgenommen, gefolgt von Sequenzdiagrammen, und so weiter (Abbildung 1). Wenn die ganzen Modellierungsthemen abgehandelt sind folgt im nächsten großen Themenblock dann die Softwarearchitektur.

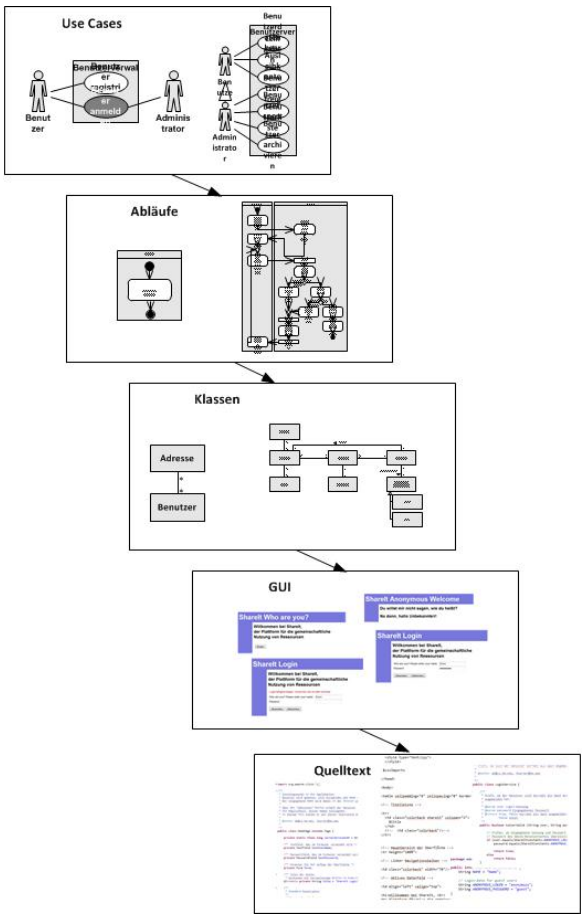


Abbildung 1: Lehre nach dem Wasserfallmodell

Der Vorteil dieses Ansatzes liegt darin, dass die Veranstaltung klar gegliedert ist. Insbesondere wird jedes Thema umfassend und abschließend behandelt. Des Weiteren sind die Themen klar voneinander abgegrenzt. Sorgfältige Mitschriften aus so aufgebauten Veranstaltungen können sich gut als Refernz für die Prüfungsvorbereitung bzw. als Nachschlagewerk eignen, weil alle Informationen, die zu einem Themenbereich gehören, im zugehörigen Kapitel zu finden sind.

Letzten Endes ist dieser didaktische Ansatz analog zum Wasserfallmodell im Software Engineering, mit vergleichbaren Vor- und Nachteilen. So erkauft man sich mit dem Vorteil der klaren Struktur den (gewichtigen) Nachteil, dass Querbeziehungen zwischen Lerninhalten aus frühen und aus späten Lehr-/Lernphasen erst sehr spät aufgedeckt werden. Des Weiteren bleiben bei der übenden bzw. praktischen Anwendung des Gelernten die Auswirkungen von Entscheidungen aus frühen Phasen relativ lange unklar für die Studierenden, da der Round Trip über den gesamten Software Life Cycle sich oft über mehrere Wochen oder Monate hinzieht und die ausführbaren Ergebnisse erst entsprechend spät vorliegen.

Iterativ-inkrementelle Vermittlung von Software Engineering Wissen

Aus diesem Dilemma heraus entstand die Idee, den Grundgedanken der heute in der Praxis vorherrschenden (Rupp u. Joppich, 2009) iterativen, inkrementellen Entwicklung auch auf den Lernprozess für angehende Softwareingenieure zu übertragen.

Zur Umsetzung dieses didaktischen Ansatzes vermittelt der seminaristische Unterricht genau diejenigen theoretischen Grundkenntnisse, die für das jeweilge Inkrement erforderlich sind. Jede Lehr-/Lerneinheit berührt somit die verschiedenen Kerndisziplinen der Entwicklung (mit ggf. unterschiedlicher Gewichtung), also mehrere Entwicklungsschritte, mehrere UML-Diagrammtypen, mehrere Implementierungskonzepte und ggf. mehrere Werkzeuge. Dafür wird in einem Inkrement jeder dieser Lernbereiche jeweils nur um ein kleines Stückchen Information und Wissen erweitert.

Parallel dazu wird im Praktikum ein kleines System iterativ und inkrementell modelliert und entwickelt. Jedes Inkrement deckt eine komplette Iteration durch den Software Life Cycle ab. Die initialen Inkremente sind zunächst sehr klein und einfach und mit viel detaillierter Anleitung hinterfüttert. Je weiter das Projekt fortschreitet, umso anspruchsvoller werden die Inkremente und die dafür benötigten Werkzeuge und Technologien. Essenziell dabei ist, dass am Ende jeder Iteration ein lauffähiges System entsteht, anhand dessen sich die Zusammenhänge zwischen den einzelnen Disziplinen, verwendeten Technologien und angewendeten methodischen Vorgehensweisen unmittelbar erkennen lassen.

Damit die Studierenden ein Verständnis dafür entwickeln, dass sie im Rahmen des Lehrkonzeptes lediglich eines von mehreren möglichen Vorgehensmodellen durchführen wurde entgegen der inhaltlichen Aufteilung in der Modulbeschreibung die Themen Pro-

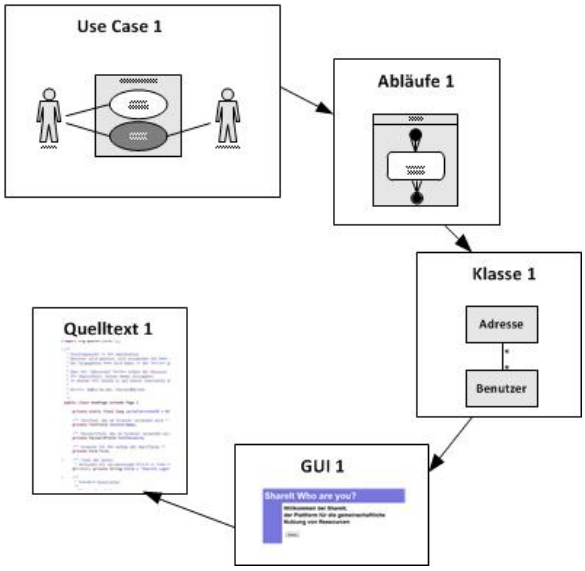


Abbildung 2: Einfaches Inkrement 1

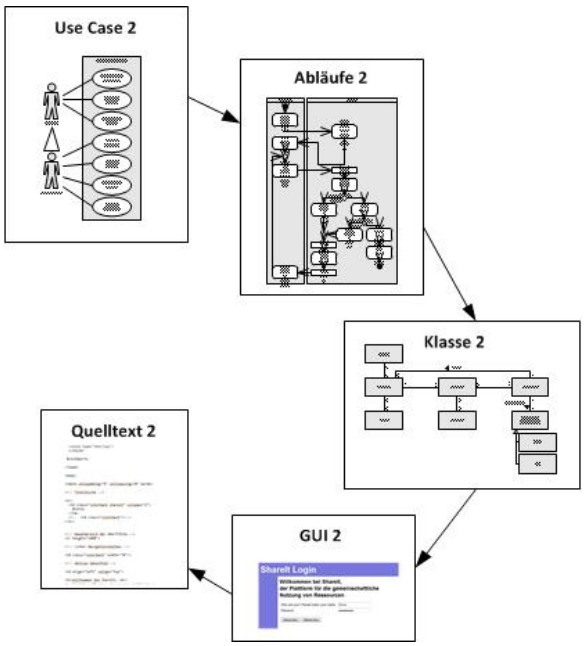


Abbildung 3: Komplexeres Inkrement 2

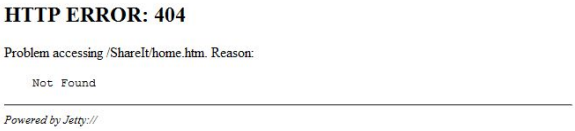


Abbildung 4: JettyServer mit Standardfehlerseite

zessmodelle und agile Vorgehensmodelle aus dem 4. Semester in das 3. Semester vorgezogen. Im Gegenzug wurden die Themen der Aufwandsschätzung und des Projektmanagements vom 3. in das 4. Semester verlagert.

Struktur des zentralen Beispiels

Kern des Lehrkonzeptes ist ein durchgängiges Beispiel, das in Grundzügen vorentwickelt ist und von den Studierenden im Rahmen des Praktikums sukzessive erweitert wird. Eine Urversion davon wurde bereits in (Thurner u. Böttcher, 2010) vorgestellt.

Die folgende Aufzählung skizziert knapp die Funktionalität, die sukzessive in den einzelnen Inkrementen umgesetzt wird. Darauf aufbauend verdeutlicht Tabelle 1, welche Techniken und Wissensbereiche in den einzelnen Inkrementen jeweils zum Einsatz kommen.

1. Der Administrator startet einen Jetty-Server als leichtgewichtigen Application Server. Anschließend greift der Normalbenutzer über den Browser auf den lokalen Port zu, unter dem der Jetty-Server bereit steht. Anhand der angezeigten Standard-Fehlerseite ist erkennbar, dass der Application Server im Hintergrund läuft (siehe Abbildung 4).

2. In der zweiten Aufbaustufe wird die Anfrage an einen Handler weitergeleitet, die als Antwort den minimalistisch formatierten statischen Text „Hello World“ erzeugt.

3. Das nächste Inkrement führt das Zusammenspiel von Java-Klasse und parametrisierter Webseite ein. Dabei ist das titel-Attribut der Java-Klasse an die $titel-Variable der Webseite gebunden. Des Weiteren nutzt die Webseite CSS und HTML zur Formatierung der Inhalte (siehe Abbildung 5).

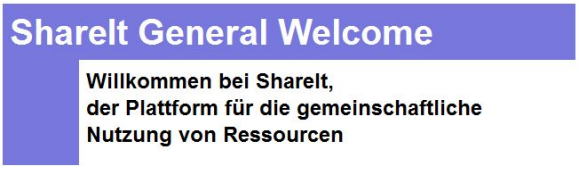


Abbildung 5: Startseite mit variablem Titel

4. Ein erster Button ist das zentrale neue Element der nächsten Ausbaustufe (Abbildung 6). Für dessen Umsetzung wird die Verwendung eines Formulars und eine Listener-Methode eingeführt sowie auf eine Antwortseite weitergeleitet (Abbildung 7).

   Die Modellebene wird ergänzt um ein einfaches Sequenzdiagramm, welches das Zusammenspiel der beteiligten Klassen veranschaulicht. Des Weiteren wird das Aktivitätsdiagramm erweitert zu einem noch recht einfachen Ablauf, bei dem Benutzer und System in mehreren Schritten miteinander interagieren.

5. Darauf aufbauend umfasst das nächste Inkrement ein Textfeld, in das der Benutzer seinen Namen eingeben kann (Abbildung 8). Nach Absenden

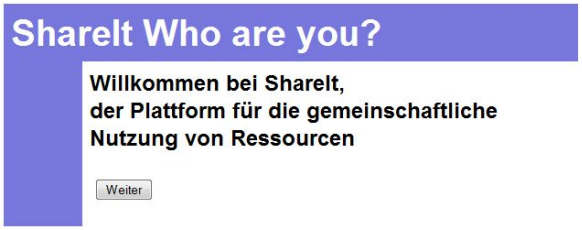


Abbildung 6: Startseite mit Button

Abbildung 7: Statische Antwortseite

der Anfrage über den Button erscheint eine personalisierte Begrüßungsseite (Abbildung 9). Für deren Umsetzung wurde der eingegebene Datenwert in der Session abgelegt und zum Generieren der Antwortseite wieder ausgelesen.

6. Die nächste Ausbaustufe bietet dem Benutzer auf der Startseite zwei Buttons an und erlaubt so eine Auswahlmöglichkeit (Abbildung 10). Entsprechend wird das Aktivitätsdiagramm um die Konzepte der Fallunterscheidung und des Ereignisses erweitert. Auch das Sequenzdiagramm wird um alternative Blöcke ergänzt.

7. Das folgende Inkrement realisiert eine Login-Möglichkeit über eine Gastkennung. Dazu wird das System um eine neue Komponente erweitert, die die Business Services kapselt. Zur Qualitätssicherung der Business Services wird deren Funktionalität mit Hilfe von Unit-Tests abgesichert. Die GUI erhält als neues Element ein Passwort-Feld. Zu den bisher bekannten Notationselementen in Aktivitäts- und Sequenzdiagrammen werden nun Rückkopplungen über join-Knoten bzw. Wiederholungsblöcke hinzugefügt.

8. Weitere Ausbaustufen fokussieren Schritt für Schritt die Realisierung von Entity-Klassen, deren Persistierung in einer Datenbank sowie die Umsetzung von Menüs zur Benutzerführung. Parallel zu den Konzepten der technischen Umsetzung werden auch hier die benötigten Modellierungsnotationen sukzessive weiter ausgebaut.

Erste Erfahrungen

Der hier vorgestellte Lehr-/Lernansatz wird in diesem Semester erstmals auf diese Weise an der Hochschule München erprobt.

Die zentrale Herausforderung in der Vorbereitungsphase bestand darin, in jedem Schritt jeweils nicht zuviel Lernstoff auf einmal anzugehen, sondern statt

Abbildung 8: Startseite mit Texteingabefeld

Abbildung 9: Personalisierte Begrüßungsseite

dessen möglichst simpel anzufangen und die nachfolgenden einzelnen Inkremente angemessen klein zu halten. Dazu wurde das ursprüngliche Beispiel (Thurner u. Böttcher, 2010) solange sukzessive immer weiter abgespeckt und vereinfacht, bis es nur noch die wesentlichen zu zeigenden Inhalte, Techniken und Zusammenhänge umfasst.

Die daraus resultierende minimalisierte Version von Spezifikation und System war für den Einstieg jedoch immer noch viel zu komplex. Daher wurde in einem zweiten Schritt, noch einmal ganz von vorne mit einem leeren Projekt anfangend, eine Folge von Spezifikationen und Systemen erstellt, die von der Komplexität und vom Funktionsumfang her bei null anfängt und in winzigen Einzelschritten sukzessive aufgebaut wird. Am Ende dieser Folge steht die minimalisierte Version, auf die das urspüngliche umfangreiche Beispiel zuvor eingedampft worden war.

Nach den bisher gewonnenen ersten Erfahrungen kommen die Studierenden gut mit dem iterativ-inkrementellen Lernansatz zurecht. Insbesondere werden die Zusammenhänge zwischen den einzelnen Disziplinen und Techniken schneller und deutlicher wahrgenommen als beim eher sequenziell orientierten Aufbau früherer Veranstaltungen. Des Weiteren erscheinen durch die relativ kleinen Schritte von Inkrement zu Inkrement die Studierenden weniger überfordert, als dies beispielsweise beim in der Vergangenheit angewendeten Lehransatz mit Gruppenpuzzle und Lernen durch Lehren der Fall war.

Dadurch, dass beim iterativen Vorgehen bereits behandelte Inhalte und Wissensbereiche immer wieder aufgegriffen und inkrementell weiter ausgebaut werden, werden außerdem die Kerninhalte regelmäßig wiederholt und scheinen sich so besser einzuprägen.

Nachteilig beim iterativ-inkrementellen Lehr-/Lernansatz ist, dass bei dieser Vorgehensweise die Informationen zu den einzelnen behandelten Themenbereichen (wie z. B. einzelne Diagrammtypen der UML) quer über die Lehrmaterialien verteilt sind, da jedes

Wissensbereich	Inkr. 1	Inkr. 2	Inkr. 3	Inkr. 4	Inkr. 5	Inkr. 6	Inkr. 7
Use Case Diagramm	Akteur, Use Case, Assoziation	Systemgrenze, Business / System Use Case					
Use Case Beschreibung		textuelle Kurzbeschreibung, Vorbedingung, Ergebnis	Normalablauf, Fehlerfall			Alternativer Ablauf	
Aktivitätsdiagramm	Start, Stop, Aktion, Kontrollfluss	Schwimmbahn		Folge von Aktionen		Fallunterscheidung, Ereignis	Wiederholung
Komponentendiagramm			Komponente, Beziehung				Schnittstelle
Sequenzdiagramm				Kommunikationspartner, Lebenslinie, Nachricht, Aktionssequenz	Nachricht mit Methodenaufruf, Antwort	Alternative Blöcke	Schleifen-Block
Webtechniken	Webapplikation	HTML, HTTP, Request, Response, Bedeutung Servlet	CSS, Trennung von Darstellung und Logik	Weiterleiten auf Antwortseite	Konzept der Session, Datentransfer über Session		
Architektur	Bedeutung Application Server	Handler	Schichtenarchitektur	Listener-Methode			Business Service
GUI-Framework (Apache Click)			Startseite, Zusammenspiel Klasse und Webseite, @Bindable Attribut, $Variable	Rendering, onInit(), Form, Control, Button	Textfeld		Passwort-Feld, Fehlermeldung
Unit-Test							Unit-Tests für den Business Service

Tabelle 1: Wissenselemente, die in den ersten Inkrementen der Systemfolge behandelt werden

Abbildung 10: Auswahlmöglichkeit durch zwei Buttons

Thema mehrfach und in zunehmenden Detaillierungsstufen aufgegriffen wird.

Abhilfe lässt sich hier schaffen durch Bereitstellung entsprechender Begleitliteratur, sofern diese themenweise aufgebaut ist. Auch die Erstellung eines ergänzenden Skriptes wäre natürlich denkbar, war jedoch aus Zeitgründen vor dieser ersten Umsetzungsphase noch nicht möglich. Statt dessen werden die Studierenden dazu angeleitet, sich als Nachschlagewerk eigene Zusammenfassungen zu den einzelnen Themengebieten zu erstellen und diese semesterbegleitend kontinuierlich auf- und auszubauen. Diese dürfen (auf 5 beidseitig beschriebene A4-Blätter beschränkt) dann als Hilfsmittel in die Prüfung mitgenommen werden.

Die obigen Aussagen spiegeln ausschließlich den Eindruck wider, den die Dozentin in den Lehrveranstaltungen und aus Gesprächen mit den Studierenden gewonnen hat. Prüfungsergebnisse, die als Kennzahl für die Bewertung des Lernerfolgs herangezogen und mit den Ergebnissen anderer Ansätze verglichen werden könnten, liegen zum aktuellen Zeitpunkt jedoch noch nicht vor.

In vergangenen Durchführungen der Lehveranstaltung Software Engineering 1 lag der Fokus überwiegend auf der Modellierung und den frühen Phasen, wie in der Modulbeschreibung vorgegeben. Durch das iterativ-inkrementelle Lehrkonzept sind Aspekte der Implementierung stärker in den Vordergrund gerückt. Dies wird sich in der Gestaltung der Prüfung entsprechend widerspiegeln, um dadurch die von den Studierenden erworbenen Kompetenzen im Bereich des Zusammenspiels von Modellierung und Implementierung zu bewerten.

Bei der initialen Umsetzung des Lehrkonzeptes erwies sich die Granularität der einzelnen Inkremente immer wieder als zentraler Knackpunkt. War die Menge an neuen Konzepten klein genug und deren Anbindung an bereits erarbeitetes Vorwissen ausreichend, so erforderte die anschließende Betreuung der Studierenden in den Praktika in etwa vergleichbaren Aufwand wie in den Vorjahren. Erwies sich die Granularität bzw. Komplexität des Inkrementes dagegen als zu hoch, schnellte der Betreuungsaufwand extrem in die Höhe.

Des Weiteren war zu beobachten, dass die Motivation der implementierungsstarken Studierenden in der Veranstaltung durch die enge Verzahnung zwischen Modellierung und Umsetzung erheblich gestiegen ist, ebenso wie deren Verständnis für die Notwendigkeit und die Aussagekraft der "bunten Bildchen".

Die Teilkohorte der implementierungsschwächeren Studierenden hatte dagegen teilweise erhebliche Probleme mit der Bewältigung der Praktikumsaufgaben. Durch die Notwendigkeit, die erstellten Modelle unmittelbar in ein lauffähiges System umzusetzen, wurden Lücken aus den implementierungsnahen Grundlagenfächern erneut evident. Darüber hinaus erzwang die drohende Umsetzung der Modelle bereits eine hohe Sorgfalt und Präzision in den Diagrammen, welche bei einer reinen Modellierungsaufgabe nicht im gleichen Maße zweifelsfrei offensichtlich geworden wäre. Es war daher aufgrund dieser engen Verzahnung weniger leicht möglich, sich durch die Aufgaben “irgendwie durchzuwursteln", was bei einigen Studierenden zu einer deutlich realistischeren Einschätzung des eigenen Leistungsstandes geführt hat. Das war teilweise sehr heilsam, hat aber bei einigen auch zu nicht unerheblichem Frust geführt.

Zusammenfassung und Ausblick

Der hier vorgestellte iterativ-inkrementelle Lehr-/Lernansatz für Wissensbereiche und Kompetenzen des Software Engineerings stellt eine Möglichkeit dar, die umfangreichen und stark miteinander verzahnten Themengebiete so aufzuschlüsseln, dass die einzelnen Lehr-/Lerneinheiten einerseits nicht überfrachtet sind und andererseits die Zusammenhänge zwischen den Wissensbereichen frühzeitig deutlich erkennbar und für die Studierenden auch selbst praktisch nachvollziehbar werden.

Erste Erfahrungen aus der Umsetzung des Ansatzes verlaufen bisher vielversprechend. Aktuell werden der Lehransatz und die benötigten Materialien kontinuierlich erweitert und ausgebaut. Eine Messung des Lernerfolges über Prüfungsleistungen steht derzeit noch aus.

Interessant ist des Weiteren auch die Frage, ob der systemimmanente hohe Wiederholunganteil des iterativ-inkrementellen Lehr-/Lernansatzes auch die Nachhaltigkeit des Gelernten positiv beeinflusst. Um hier eine Aussage treffen zu können ist jedoch eine Langzeitstudie mit entsprechenden Vergleichsgruppen erforderlich.

Literatur

[Abran u. a. 2005] ABRAN, A. ; MOORE, J. ; DUPUIS, R. ; TRIPP, L.: *Guide to the Software Engineering Body of Knowledge 2004 Version SWEBOK*. IEEE Computer Society Press, 2005

[IEEE 2012] IEEE, Computer-Society: *Software Engineering Body of Knowledge, vorläufiger Stand der Version 3*. www.computer.org/portal/web/swebok, abgerufen am 28.10.2012, 2012

[Lehner 2009] LEHNER, M.: *Viel Stoff – wenig Zeit*. Haupt Verlag, Stuttgart, 2009

[Rupp u. Joppich 2009] RUPP, C. ; JOPPICH, R.: *Dokumentenberge oder Bierdeckel – Requirements Engineering in Zeiten der Agilität*. heise Developer, http://www.heise.de/developer/artikel/Requirements-Engineering-in-Zeiten-der-Agilitaet-804971.html, abgerufen am 28.10.2012, 2009

[Thurner u. Böttcher 2010] THURNER, V. ; BÖTTCHER, A.: Beispielorientiertes Lernen und Lehren im Software Engineering. In: *ESE 2010: Embedded Software Engineering Kongress*, 2010, S. 591–595

Vermittlung von agiler Softwareentwicklung im Unterricht

Martin Kropp, FHNW, martin.kropp@fhnw.ch

Andreas Meier, ZHAW, meea@zhaw.ch

Zusammenfassung

Über den Hype hinaus, der um agile Softwareentwicklung entstanden ist, zeigen verschiedene Umfragen, dass dieses Vorgehen in der Praxis in verschiedener Hinsicht tatsächlich zu Verbesserungen in der Durchführung von Software-Projekten führt. Firmen, die agile Methoden einsetzen, geben an, dass sie seither zufriedener mit ihrem Entwicklungsprozess sind und insbesondere der Umgang mit Anforderungsänderungen sich wesentlich verbessert hat.

Andererseits sind entsprechend ausgebildete Fachleute jedoch Mangelware. In der Praxis sind deshalb Software Ingenieure mit Kompetenzen in agilen Methoden sehr gefragt.

Was bedeutet dies für die Software-Technik Ausbildung an Hochschulen? Was sind die speziellen Herausforderungen? Wie kann agile Software Entwicklung, die neben konkreten Techniken und Praktiken auf der individuellen Ebene, auch auf Team-Ebene und Werte-Ebene spezielle Anforderungen stellt, überhaupt vermittelt werden? Wie kann die Ausbildung von agiler Softwareentwicklung in die Ingenieur-Ausbildung integriert werden?

In diesem Artikel stellen wir unser Konzept zur Ausbildung von agilen Methoden an Hochschulen vor und berichten über unsere Erfahrungen als Dozierende.

Einleitung

Jüngste Umfragen zeigen, dass agile Software Entwicklungsmethoden in verschiedener Hinsicht zu wesentlichen Verbesserungen in der Durchführung von Software-Projekten führt [1,2]. Diese Erfolgsmeldungen tragen wesentlich dazu bei, dass agile Softwareentwicklungsmethoden in der IT-Industrie eine immer grössere Akzeptanz finden.

In der von den Autoren durchgeführten Studie (Swiss Agile Study) über den Einsatz von Software Entwicklungsmethoden in der Schweizer IT-Industrie [1] wurden diese Aussagen bestätigt. Die Studie liefert auch konkrete Zahlen über die Verbreitung, den Nutzen, den Einsatz von konkreten Praktiken, aber auch die Herausforderungen, die mit agilen Methoden einhergehen.

Um die Relevanz der agilen Methoden für die Ausbildung zu ermitteln, war für uns unter anderem die Beantwortung folgender Fragen wichtig:

1. Wie verbreitet ist agile Softwareentwicklung in der Praxis?
2. Haben agile Methoden wirklich Vorteile gegenüber den traditionellen, plan-getriebenen Methoden?
3. Was sind die entscheidenden Erfolgsfaktoren bei der agilen Softwareentwicklung?

Im nächsten Abschnitt werden wir die aus unserer Sicht für die Ausbildung wichtigsten Resultate der Studie vorstellen und danach einige Überlegungen und Konsequenzen für die Ausbildung an (Fach-) Hochschulen näher ausführen.

Nach einer Übersicht über die wesentlichen Elemente der agilen Methoden werden wir im Anschluss unsere Konzepte vorstellen. Wir zeigen auf, wie die erforderlichen Kompetenzen vermittelt bzw. von den Studierenden erlernt werden können.

Anschliessend berichten wir über den konkreten Umsetzungsstand und unsere Erfahrungen, die wir gemacht haben.

Den Abschluss bildet ein Ausblick über die Weiterentwicklung der Konzepte und deren Umsetzung.

Verbreitung und Nutzen von agilen Methoden

In der Swiss Agile Study wurden mehr als 1500 IT-Firmenmitglieder der teilnehmenden ICT-Verbände SwissICT, ICTnet und SWEN befragt, die Rücklaufquote betrug knapp 10%. Dabei wurden sowohl agil als auch nicht-agil arbeitende Unternehmen einbezogen. Von den teilnehmenden Unternehmen gaben 57% an, dass sie mit agilen Methoden entwickeln.

Auf die Frage, wie die agilen Methoden die Entwicklung beeinflusst hat, gaben die Unternehmen

zu allen befragten Aspekten an, dass sich diese verbessert (+) oder sogar sehr stark verbessert (++) haben (siehe Tabelle 1). Dabei sind insbesondere die Aspekte „Ability to manage changing Requirements“ (89%), „Development Process“ (80%) und „Time To Market“ (76%) zu nennen.

Aspect	--[1]	-	*o*	+	++
Time to market	**1**	**2**	**19**	**53**	**23**
Ability to manage changing priorities	**1**	**0**	**9**	**45**	**44**
Productivity	0	2	33	47	15
Software quality	0	2	45	35	16
Alignment between IT & business objectives	0	1	25	46	23
Project visibility	0	2	25	39	28
Risk management	0	5	32	42	17
Development process	**0**	**2**	**17**	**58**	**22**
Software maintainability / extensibility capability	0	7	55	23	12
Team morale	0	4	25	42	24
Development cost	1	12	52	22	7
Engineering discipline	0	4	42	42	9
Management of distributed teams	0	5	42	19	6
Requirements management	0	2	29	51	13

Tabelle 1. ***How has agile software development influenced the following aspects?***[2][3]

Interessant an den Ergebnissen in dieser Tabelle ist, dass mehr als die Hälfte der Unternehmen angeben, dass sich die Wartbarkeit der Software sowie die Kosten nicht verbessert haben. Aus Sicht der Ausbildung ist von Interesse, welche Praktiken und Techniken entscheidend zum Erfolg in agilen Projekten beitragen, da wir auf diese Techniken besonderes Augenmerk legen sollten.

Bei dieser Frage haben wir nach *Engineering Practices* wie z.B. Unit Testing und Continuous Integration sowie *Management Practices* wie Daily Standup, On-Site Customer und Planung unterschieden. Tabelle 2 gibt die detaillierten Resultate für die *Engineering Practices* wieder. Dabei geben die Spaltenwerte die Wichtigkeit für den Erfolg von „ganz unwichtig“ (--) bis „sehr wichtig“ (++) wieder. Als wichtige bzw. sehr wichtige *Engineering Practices* werden dabei insbesondere Kodierrichtlinien (82%), Unit Testing (76%) und Continuous Integration (70%) genannt.

Aspect	--	-	+	++
Behavior Driven Development (BDD)	21	24	17	8
Acceptance Test Driven Development (ATDD)	18	22	20	17
Test Driven Development (TDD)	10	16	29	30
Coding standards	**4**	**9**	**40**	**42**
Collective code ownership	12	12	33	30
Pair programming	20	32	25	18
Unit testing	**2**	**17**	**25**	**51**
Refactoring	3	25	35	28
Automated builds	8	14	26	40
Continuous integration	**4**	**16**	**32**	**38**
Automated acceptance testing	20	26	28	14
Continuous delivery	12	23	38	17

Tabelle 2. How important were the following agile engineering practices for your successful agile projects?[3]

Bei der Interpretation der Daten ist zu beachten, dass gewisse agile Praktiken wie z.B. TDD, BDD, ATDD auch bei agilen Unternehmen noch relativ wenig im Einsatz sind, wie sich bei der Umfrage gezeigt hat.

Bei den *Management Practices* wurden als wichtigste Aspekte Iterationsplanung (89%), User Stories (83%) und Release-Planung (77%) genannt, dicht gefolgt vom Einsatz eines Task Boards (74%).

Aspect	--	-	+	++
Release planning	**3**	**18**	**37**	**40**
Story mapping	4	26	41	21
On-site customer	11	27	33	24
Iteration planning	**0**	**7**	**36**	**53**
User stories	**0**	**10**	**40**	**43**
Daily standup	4	25	32	33
Taskboard	**2**	**20**	**45**	**29**
Burndown charts	11	38	22	24
Retrospective	5	28	34	30
Open work area	12	25	40	11
Kanban Pull System/Limited WIP	27	23	13	4

Tabelle 3. How important were the following agile management and planning practices for your successful agile projects?

Ein wichtiger Aspekt ist, dass die Zufriedenheit der Unternehmen mit den agilen Methoden insgesamt

[1] *Die Bewertungskriterien gehen von „Stark verschlechtert (--) bis „stark verbessert“(++)*

[2] *Sämtliche Angaben in Prozent; die fehlenden Prozent auf 100 gaben an “Weiss nicht”.*

[3] *Die Umfrage wurde auf Englisch durchgeführt, um alle Sprachregionen der Schweiz abdecken zu können, daher sind sowohl die Fragen als auch die Antwort-Optionen auf Englisch.*

deutlich höher ist als mit traditionellen plangetriebenen Vorgehen (84% vs. 62%).

Anforderungen an agile Ausbildung

Im Rahmen der Studie haben wir auch erhoben, ob agile Methoden überhaupt auf universitärer Stufe unterrichtet werden sollten, und wie die Kenntnisse der Bachelor- und Master-Absolventen von der Industrie beurteilt werden. Dazu hatten die Firmenvertreter der teilnehmenden IT-Firmen folgende Aussage zu bewerten:

„*Agile development should be an integral part of the Computer Science curriculum*".

Zur Auswahl standen folgende Antwort-Optionen von "Stimme vollständig zu" bis "Stimme gar nicht zu".

95% der teilnehmenden Firmen stimmen der Aussage zu („Stimme zu": 49%, „Stimme vollständig zu": 46%).

Andererseits geben die Unternehmen auf die Frage, ob die Studienabgänger auf Bachelor- bzw. Masterstufe genügend Kenntnisse in agilen Methoden mitbringen mit klarer Mehrheit an, dass dies nicht der Fall ist (Master-Stufe: 58%, Bachelor-Stufe: 68%).

Somit sollte der Anspruch der Praxis an die Hochschulen ernst genommen werden und die Ausbildung an die neuen Anforderungen angepasst werden. Dabei stellt sich die Frage, was die Hochschulen in der Ausbildung ändern müssen, um den neuen Anforderungen gerecht zu werden.

Dazu betrachten wir Anforderungen, die agile Methoden an den einzelnen aber auch an das Team und die Organisation stellt, auf drei verschiedenen Ebenen:

1. *Individuelle Ebene*: Hier geht es vornehmlich darum, welche agilen *Engineering Practices* ein Software Ingenieur beherrschen sollte: Clean Code, Test Driven Development, Automation, Craftsmanship, um nur einige Beispiele zu nennen.
2. *Team Ebene*: Die Team Ebene stellt vor allem Anforderungen im Bereich *Management Practices:* Selbstorganisierende Teams, Schätzen und Planen, Continuous Integration und Continuous Deployment, oder Pair Programming. Selbstorganisierende Teams wiederum verlangen eine hohe soziale Kompetenz der Teammitglieder, wie zum Beispiel hohe Kommunikationsfähigkeit.
3. *Werte Ebene*: Auf dieser Ebene wird implizit das Wertesystem der agilen Softwareentwicklung betrachtet; wie die agilen Werte wie Respekt, Offenheit, Ehrlichkeit, etc. vermittelt werden können, so dass die Absolventen „in der Wolle gefärbte" Software Ingenieure sind.

Bei der Vermittlung der Anforderungen beziehen wir uns schwerpunktmässig auf *Scrum* [7] und *eXtreme Programming (XP)* [8], da diese in der Schweiz (und auch international [2]) die am weitesten verbreiteten agilen Methoden sind. Ausserdem ergänzen sich diese Methoden aus unserer Sicht sehr gut, da XP eher den Schwerpunkt auf die *Engineering Practices* legt, während Scrum eher auf der Management Ebene anzusiedeln ist.

Erfahrungen mit agiler Entwicklung im Unterricht

Zum Thema der Integration von agilen Methoden in den Unterricht liegen verschiedene Erfahrungsberichte vor [20, 21]. Die Autoren selbst unterrichten seit über fünf Jahren zusammen die Vorlesung *Software Engineering and Architecture.* Diese Vorlesung ist Teil der Schweiz weiten gemeinsamen Masterausbildung (Master of Science in Engineering, MSE) der Schweizerischen Fachhochschulen [3]. Die gemeinsame Masterausbildung betrachten die Autoren als eigentlichen Glücksfall, da dadurch Dozierende von verschiedenen Hochschulen zusammen lehren und ihre Erfahrungen austauschen können.

Wir haben festgestellt, dass die Studierenden noch relativ wenig Wissen über agile Softwareentwicklungsmethoden mitbringen, wobei sich der Wissenstand in den letzten Jahren leicht verbessert hat, aber je nach Fachhochschule sehr unterschiedlich ist. Dafür beobachten wir jedoch, dass das Interesse der Studierenden an dem Thema umso grösser ist.

In unserer gemeinsamen Vorlesung haben wir den Fokus ganz gezielt auf agile Softwareentwicklung gelegt. Unter anderem unterrichten wir agile Software Entwicklungsmethoden wie Scrum, eXtreme Programming oder Kanban. Dabei haben wir die Erfahrung gemacht, dass viele Studierende das „Programmier-Handwerk", welches eine der Voraussetzungen für die erfolgreiche Durchführung agiler Projekte darstellt, nicht beherrschen resp. nie im Bachelor-Studiengang gelernt hatten. Unter „Programmier-Handwerk" verstehen wir das Beherrschen von agilen Praktiken wie zum Beispiel Clean Code, Refactoring, Test-Driven Development oder Continuous Integration.

Weiter halten die Autoren jeweils an ihrer Fachhochschule [4, 5] verschiedene Vorlesungen in Programmieren und Software Engineering auf Bachelor Stufe und haben damit begonnen, auch auf dieser Stufe agile Software Entwicklungsmethoden zu vermitteln.

Agile Software Entwicklung

Für ein besseres Verständnis der im Folgenden vorgestellten Studienkonzepte und Lehr- und Lernmethoden fassen wir hier die aus unserer Sicht wichtigsten Eigenschaften der agilen Softwareentwicklung zusammen.

In erster Linie sind dabei natürlich die Werte und Prinzipien des *Agile Manifesto* [6] zu nennen, die auf der einen Seite zwar noch keine konkreten Praktiken und Techniken vorgeben, aber mit Ihren Prinzipien doch klare Anforderungen an eine agile Vorgehensweise definieren.

Aus diesen Prinzipien hat insbesondere extreme Programming (XP) eine Anzahl ganz konkreter Programmierpraktiken abgeleitet, ohne die aus unserer Sicht eine agile Softwareentwicklung nicht möglich ist.

Auf Managementsicht gehört wohl Scrum zu den Vorreitern der agilen Methoden. Scrum definiert klare Regeln bzgl. Projekt- und Teamorganisation sowie dem Requirements-Management.

Neuere Ansätze wie Lean Software Development [9], die sich ebenfalls an den Prinzipien des *Agile Manifesto* orientieren, zeigen weitere, interessante Aspekte zur Umsetzung der agilen Software Entwicklung auf.

Die folgenden Tabellen fassen die aus unserer Sicht wichtigsten *Engineering* und *Management Practices* zusammen. Diese Einteilung wurde von uns für die Studie entwickelt, um die Verbreitung der genannten Praktiken in der Praxis zu ermitteln und hat sich dabei als sehr zweckmässig erwiesen; wir verwenden sie daher auch in diesem Artikel als Grundlage für die folgende Diskussion über die agile Ausbildung.

Tabelle 4 gibt die Verbreitung der, vor allem von XP definierten, konkreten Programmierpraktiken in den befragten agilen Unternehmen wieder, während Tabelle 5 die Verbreitung der Management Praktiken aufzeigt. Besonders erwähnenswert dabei ist, dass diese entsprechenden Werte bei plangetrieben arbeitenden Unternehmen zwischen 10%-30% tiefer liegen. Dies ist doch umso bemerkenswerter, als es sich bei vielen *Engineering Practices* eigentlich nicht um spezielle agile Praktiken, sondern um allgemeine *Best Practices* handelt. Es lässt sich also sagen, dass die Anwendung von agilen Methoden auch zur verstärkten Anwendung von allgemeingültigen Best Practices führt.

Engineering Practices	Verbreitung
Coding standards	75%
Unit testing	70%
Automated builds	63%
Continuous integration	57%
Refactoring	51%
Test Driven Development (TDD)	44%
Pair programming	31%
Collective code ownership	30%
Continuous delivery	30%
Automated acceptance testing	24%
Acceptance Test Driven Development (ATDD)	18%
Behavior Driven Development (BDD)	15%

Tabelle 4. Engineering Practices in der Praxis[4]

Management Practices	Verbreitung
Release planning	75%
Iteration planning	66%
User stories	65%
Daily standup	53%
Taskboard	46%
Retrospective	41%
Burndown charts	40%
Story mapping	35%
Open work area	27%
On-site customer	23%
Kanban Pull System/Limited WIP	11%

Tabelle 5. Management Practices in der Praxis[5].

Die Werte zeigen deutlich, dass auch bei agil entwickelnden Unternehmen erst relativ wenige der erforderlichen bzw. empfohlenen Praktiken eine breite Anwendung finden.

Studienkonzept für agile Methoden

Die Eigenschaften bzw. Anforderungen an eine agile Software Entwicklung lassen sich nicht alle auf die gleiche Art unterrichten, da sie unterschiedliche Kompetenzen ansprechen. Für die Entwicklung eines geeigneten Studienkonzeptes orientieren wir uns daher an den zuvor eingeführten drei Kompetenzebenen.

---

[4] *Den Firmen, die agile Methoden anwenden, wurde folgende Frage gestellt: „Which of the following engineering practices could be observed by someone visiting your company in the next month"?*

[5] *Den Firmen, die agile Methoden anwenden wurde folgende Frage gestellt: „Which of the following management and planning practices could be observed by someone visiting your company in the next month"?*

Die drei Vermittlungsebenen

Auf der *individuellen Ebene* werden insbesondere die handwerklichen Grundlagen vermittelt. Ohne das Beherrschen dieser fundamentalen Techniken ist eine agile Entwicklung aus unserer Sicht nicht möglich. Aus unserer Erfahrung sind die agilen Techniken der individuellen Ebene am einfachsten zu vermitteln. Dies liegt unserem Ermessen nach daran, dass jeder Studierende einzeln daran arbeiten kann.

Darauf aufbauend können dann die Techniken der agilen Entwicklung auf der *Team-Ebene* vermittelt und erworben werden, die sich vor allem aus den *Management Praktiken* zusammensetzen. Diese Aspekte lassen sich aus unserer Erfahrung heraus sehr gut in Gruppen- und Projektarbeiten vermitteln. Die Konzeption, Organisation und Durchführung solcher Arbeiten ist aufwendig.

Sozusagen die Spitze der Kompetenzen der Agilen Entwicklung bildet die *Werte-Ebene*. Diese agilen Wertehaltungen wie sie im *Agile Manifesto* [6] beschrieben werden, sind naturgemäss am schwierigsten zu vermitteln. Wir versuchen die agilen Werte punktuell immer wieder in den Unterricht einfliessen zu lassen – oder ganz bewusst auch vorzuleben.

Bachelor- und Master-Ausbildung

Aufgrund der sich schon früh abzeichnenden zunehmenden Bedeutung der agilen Entwicklungsmethoden haben wir begonnen die grundlegenden Methoden und Techniken der agilen Software Entwicklung in Form der genannten Management und Engineering Praktiken in das Bachelor-Programm zu integrieren [10,11]. Dies wird im Folgenden noch genauer erläutert.

Auf Master-Stufe behandeln wir in unserem gemeinsamen Kurs „Software Engineering and Architecture“ fortgeschrittene Themen der agilen Software Entwicklung wie Lean Development und Kanban, Incremental Software Design und Software Evolution.

Unterrichtsmethoden

Neben konventionellen Vorlesungen und Programmierübungen verwenden wir als weitere Unterrichtmethoden:

- Case Studies (Fremde / Eigene)
- Gruppenarbeiten
- Gastreferate mit Referenten aus der Software-Industrie
- Simulationen
- Medien: eBooks, Blogs, Internet.

Wichtig ist die Repetition auf allen Ebenen, d.h. regelmässiges Üben in verschiedenen Kontexten. Deshalb kommen die verschiedenen Praktiken und Techniken immer wieder in den verschiedenen Vorlesungen vor. Nicht isoliertes Unterrichten der einzelnen Praktiken steht im Vordergrund, sondern die Verankerung an mehreren Stellen in der Ausbildung. Das unterstützt insbesondere auch die Werte Ebene, in dem die Dozierenden selbst im Rahmen ihrer Möglichkeiten diese agilen Werte vorleben.

Beispiel Bachelor Programm

Obwohl an den beiden Fachhochschulen, auch was das Themengebiet Software Engineering angeht, unterschiedliche Studienpläne existieren, haben sich, unterstützt durch den intensiven Austausch, viele Gemeinsamkeiten in der Vermittlung der agilen Methoden ergeben.

Abbildung 1 zeigt als ein mögliches Beispiel einen

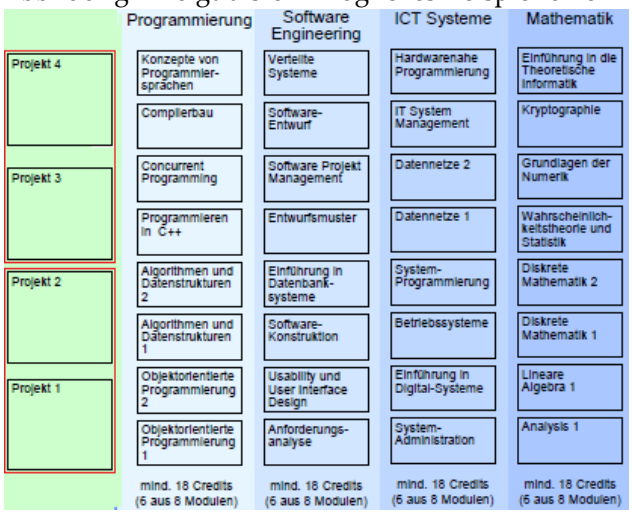


Abbildung 1. Grundstudium Informatik

Ausschnitt des Informatik Bachelor-Programms der Fachhochschule Nordwestschweiz.

Das fachliche Grundstudium im Bachelorstudiengang ist aufgeteilt in die vier Themenbereiche, Modulgruppen genannt, Programmierung, Software Engineering, sowie ICT Systeme und Mathematik. Jede Modulgruppe besteht aus je 8 Modulen, die spezielle Themen aus der jeweiligen Modulgruppe abdecken. Jedes Modul hat einen Umfang von 3 ECTS Punkten.

Begleitet wird die theoretische Ausbildung in den Modulen durch eine vom ersten Semester an beginnende Projektschiene, die sich bis in das 6. Semester fortsetzt. In dieser setzen die Studierenden in grossen Teams (7 bis 8 Studierende) und an realen, d.h. von externen Kunden in Auftrag gegebenen Projekten, die Theorie in die Praxis um. Dabei werden in den beiden Jahresprojekten der ersten 4 Semester unterschiedliche Schwerpunkte gesetzt.

Die explizite Ausbildung in agilen Methoden wird insbesondere in den Modulen Software Konstruktion im 2. Semester und Software Projekt Management im 3. Semester der Modulgruppe Software Engineering vorgenommen. In den Projekten und den sonstigen „programmierlastigen" Modulen wird darauf geachtet, dass auch hier die vermittelten agilen Praktiken zum Einsatz kommen.

In den folgenden Kapiteln beschreiben wir, wie wir an beiden Fachhochschulen die verschiedenen Praktiken und Techniken im Unterricht vermitteln.

Engineering Practices

Bei der Besprechung der Practices orientieren wir uns an der Liste aus Tabelle 4.

Coding Standards

Bereits im ersten Semester führen wir Coding Standards in der Einführungsvorlesung ins Programmieren ein. Wir kombinieren Coding Standards mit *Clean Code* [12], um so bei den Studierenden von Anfang an auch das Bewusstsein für eine „saubere" Programmierung zu wecken. Mit dem Einbezug von Clean Code haben wir sehr gute Erfahrungen gemacht, da dies den konkreten praktischen Nutzen von Coding Standards sichtbarer macht.

Von nicht zu unterschätzender Wichtigkeit ist bei der Einführung der Aspekt des Feedbacks. Das wird wie folgt gehandhabt:

Normalerweise gehört zu jeder Vorlesung eine Programmieraufgabe. Der Dozierende führt mit jedem Studierenden einen Code Review durch und gibt ein ausführliches Feedback. Das Feedback ist im Allgemeinen mündlich, kann aber auch schriftlich erfolgen.

Das Feedbackgeben durch den Dozierenden ist zwar mit einen grossem Aufwand verbunden, ist aus unserer Erfahrung aber entscheidend für den Lernerfolg.

Coding Standards müssen natürlich durch die verwendete integrierte Entwicklungsumgebung (IDE) unterstützt werden. Minimal sollte die IDE auch Unterstützung für einfache *Refactorings* wie *Rename Method* oder *Rename Variable* bieten.

Ebenfalls gute Erfahrungen haben wir mit Werkzeugen wie *Checkstyle* [16] gemacht, welches direkt in der IDE anzeigt, falls die Coding Standards verletzt werden.

Sehr positive Erfahrungen haben wir mit dem frühzeitigen Einsatz dieser Konzepte und Werkzeuge (z.B. im 2. Semester im Rahmen des Modul Software Konstruktion) zusammen mit *Continuous Integration* Umgebungen gemacht. Durch eine frühe Einführung werden die Studierenden animiert, diese Praktiken auch in ihren Projekten einzusetzen, was auch sehr häufig auf freiwilliger Basis geschieht, da sie die Vorteile solcher Umgebungen schon früh erkennen.

Unit Testing

Unit Testing wird im ersten Semester eingeführt. Dabei ist besonders wichtig, dass diese Praktik nicht nur in speziellen Modulen unterrichtet wird, sondern, dass auch sonstige programmierlastige Module Unit Tests einfordern oder vorgeben, um so z.B. die erfolgreiche Implementierung einer Programmieraufgabe zu überprüfen.

So können zum Beispiel im ersten Semester während der Einführung ins Programmieren Unit Tests bei Übungen vorgegeben werden. Die Aufgabe gilt als erfüllt, wenn das zu entwickelnde Programm alle Unit Test besteht. Die Studierenden lernen so spielend die Nützlichkeit von automatischen Tests kennen und erleben nebenbei, wie wertvoll Tests als Dokumentation sind.

Anschliessend wird das automatische Testen mit White-Box/Black-Box Tests und Äquivalenzklassen formal eingeführt. Die Studierenden sind dann in der Lage, einfache Testfälle selbstständig zu schreiben. In den höheren Semestern werden mit zunehmendem Know-how anspruchsvollere Testfälle mit Mock-Objekten entwickelt und verschiedene Testmuster eingeführt. Das automatische Testen sollte sich durch möglichst alle programmier-nahen Vorlesungen durchziehen.

Ein anderer erprobter Weg ist, dass die Studierenden schon vom ersten Semester an in den Programmier-Modulen das Schreiben von einfachen Unit-Tests mittels Unit Testing Frameworks kennenlernen und in den Projekten anwenden. Anschliessend erhalten sie im zweiten Semester im Modul Software Konstruktion eine vertiefte Einführung in das systematische Testen und fortgeschrittene Themen wie Mock Testing kennen, die sie dann wiederum in den Projektarbeiten für das fortgeschrittene Testen einsetzen können.

Refactoring

Test Driven Development (TDD) baut auf automatischen Unit Tests auf. Neben Kenntnissen in Unit Testing brauchen die Studierenden auch Kenntnisse in *Refactoring*. Mit anderen Worten sind gute Kenntnisse in Refactoring eine Voraussetzung für TDD.

Die einfachsten Code-Refactorings werden bereits am Anfang des Studiums zusammen mit den Coding Standards und der IDE Unterstützung eingeführt. Komplexere Design-Refactorings werden in den folgenden Semestern eingeführt. Dazu wird ein Katalog der verschiedenen Refactorings abgegeben und mit den Studierenden durchgearbeitet.

Ergänzt wird dieses Thema durch die Anwendung von komplexeren Refactorings wie Extract Class, Extract Method, Move Method anhand von Case Studies z.B. bei Überschreitungen von Schwellwerten bei Qualitätsmetriken.

Test Driven Development (TDD) und Refactoring

TDD ist eine anspruchsvolle Engineering Practice. Um TDD wirklich zu beherrschen, wäre es am Besten, wenn die Studierenden einen erfahrenen Software-Entwickler als Coach zur Seite hätten. Da dies im Rahmen einer Vorlesung oder Übungsstunde nicht wirklich realisierbar ist, sind alternative Vorgehensweisen nötig:

Bewährt haben sich folgende Alternativen:

- Case Studies
- Programming Katas
- Experten Videos
- Bücher und Fachartikel

Während den Studierenden mittels Experten Videos und Fachliteratur die konzeptionellen Aspekte des TDD vermittelt werden können, sollen Case Studies und Programming Katas den Studierenden dazu dienen, den TDD Ansatz praktisch zu üben und insbesondere dessen Einfluss auf Testbarkeit und Software Design selbst zu erfahren.

Automated Builds

Automated Builds werden mit Hilfe von *Ant-Skripts* und einem *Code Versioning System* (CVS) wie zum Beispiel SVN oder GIT eingeführt. Um den vollen Nutzen der Build-Automatisierung zu erfahren, bauen die Studierenden anhand einer konkreten Case Study über ein ganzes Semester ein Automatisierungsskript kontinuierlich aus, so dass das Kompilieren, Testen, Code Analysieren, als auch das Packaging der Software vollständig automatisiert durchgeführt werden kann.

Automated Builds sind eine Voraussetzung für Continuous Integration und werden im nächsten Kapitel genauer angeschaut.

Continuous Integration (CI)

Automated Builds und Continuous Integration werden mittels einer konkreten Fallstudie in einer Gruppenarbeit vermittelt und ebenfalls schon frühzeitig im Studium eingeführt (z.B. im Rahmen des Moduls Software Konstruktion im 2. Semester). In einem ersten Schritt wird das Beispielprojekt in ein Code Versioning System importiert. Im zweiten Schritt werden die Build-Skripts entwickelt, damit das Beispielprojekt automatisiert gebaut werden kann. Im dritten Schritt Konfigurieren die Studierenden den CI-Server (Jenkins). Im vierten Schritt werden verschiedene Jenkins-Erweiterungen besprochen und ausprobiert. Dabei werden verschiedene Metriken wie Code Coverage, Bindungsstärke von Klassen, Code Complexity, aber auch Lines of Code (LOC) behandelt.

Fortgeschrittene Metriken werden in höheren Semestern (Bewertung einer Software Architektur im 4. Semester Software Entwurf [17]) oder im Master-Studiengang (Technical Debit [18]) behandelt.

Es hat sich als zweckmässig erwiesen, den ganzen Bereich der Software Konstruktion in einer eigenen Vorlesung zusammenzufassen. Da Kenntnisse in Software Konstruktion eine wichtige Voraussetzung für erfolgreiche Projekt- und Bachelorarbeiten sind, sollte die Vorlesung bereits im zweiten oder dritten Semester durchgeführt werden.

Der nachhaltige Erfolg der frühzeitigen Einführung dieser Praktiken zeigt sich unter anderem daran, dass Studierende selbstständig eine solche Infrastruktur für ihre Projektarbeiten anfordern und einsetzen.

Pair Programming

Pair Programming üben wir nicht speziell. Es wird zusammen mit den anderen *Engineering Practices* eingeführt und die Studierenden werden ermuntert, Pair Programming bei Gelegenheit gezielt einzusetzen und für sich zu entscheiden, ob es für sie eine Option darstellt oder nicht.

Wir haben die Beobachtung gemacht, dass die meisten Studierenden ganz automatisch eine Form von Pair Programming einsetzten. Sehr oft setzten sich zwei Studierende zusammen an einen Computer und lösen eine Programmieraufgabe gemeinsam. Es scheint also so etwas wie eine natürliche Vorgehensweise beim Programmieren zu sein.

Automated Acceptance Testing

Auf Automated Acceptance Testing gehen wir ebenfalls im Rahmen der Vorlesung Software Konstruktion ein. Es wird ein Überblick über das Konzept anhand des Test Frameworks Fit [13] vermittelt und Akzeptanztests geschrieben. Diese Tests werden auch in den CI-Build Prozess integriert.

Collective Code Ownership

Ebenso wie Pair Programming wird auch das Thema der Collective Code Ownership nicht explizit unterrichtet. Die Studierenden werden jedoch durch Verwendung von Versionskontrollsystemen, Anwendung von Coding Standards und Wechseln der Verantwortlichkeiten in den Projekten dazu ermuntert, diese Praktik auszuprobieren.

Fortgeschrittene Engineering Practices

Die weiteren *Engineering Practices* wie Continuous delivery, Acceptance Test Driven Development (ATDD) oder Behavior Driven Development (BDD) sprengen unserer Ansicht nach den Rahmen eines Bachelor Studiums. Denkbar, und aus unserer Sicht auch sinnvoll, wäre es, diese weiteren *Engineering Practices* zukünftig im Master Studium zu behandeln, da wir davon ausgehen, dass diese Praktiken in Zukunft an Bedeutung gewinnen werden.

Management Practices

Wie aus Tabelle 5 zu entnehmen ist, sind bei den *Management Practices* diejenigen am weitesten verbreitet, welche sich direkt mit der Projektplanung befassen. Nicht unerwartet wird die Planung vorwiegend unter Verwendung von User Stories gemacht. Überraschenderweise sind Retrospektiven in der Praxis, trotz deren unbestritten hohem Nutzen, nicht sehr weit verbreitet.

Die *Management Practices* betreffen vorwiegend die Team Ebene. Teamarbeiten sind im Gegensatz zu den *Engineering Practices*, welche vorwiegend die Individuelle Ebene betreffen, nicht einfach in den Unterricht zu integrieren. Die meisten *Management Practices* können nur sinnvoll in einem grösseren Team im Rahmen eines „richtigen" Projekts sinnvoll geübt werden. Oft ist es aber nicht zweckmässig oder aus zeitlichen Gründen unmöglich, ein solches Projekt durchzuführen. In diesem Fall kann die Situation mit einem der agilen Games wie zum Beispiel dem XP-Game [14] oder Scrum-City-Game [15] simuliert werden.

Scrum

Als agile Projektmethode verwenden wir Scrum. Die verschiedenen Studien zeigen, dass Scrum am weitesten verbreitet ist. Scrum bietet auch den Vorteil, dass die Anzahl der *roles*, *events* und *artifacts* klein ist.

Wir haben die Erfahrung gemacht, dass die Funktionsweise von Scrum schnell im Unterricht erklärt werden kann. Der schwierige Teil kommt bei der Einführung. Wenn die Studierenden Scrum (oder eine andere Projektmethode) wirklich anwenden sollen. Wie kann im Unterricht ein Klima geschaffen werden, so dass die Studierenden sich wie in einem „echten" Projekt fühlen? Wenn sie das Gefühl haben, dass es sich um ein „Alibi"-Projekt handelt, setzten sie sich nicht ein und profitieren entsprechend wenig.

XP-Game

Das XP-Game [14] simuliert ein agiles Projekt. In drei bis vier Stunden werden drei Iterationen mit Schätzen der Tasks, Planung der Iterationen, Implementation und Abnahme der Tasks sowie einer Retrospektive nach jeder Iteration durchgespielt. Die Planung und Retrospektiven werden „richtig" gemacht, die Implementationen dauern jeweils nur zwei Minuten. In diesen zwei Minuten kann natürlich kein Code entwickelt werden. Anstelle werden kurze, einfache Tasks, wie zum Beispiel ein Kartenhaus bauen oder etwas im Kopf ausrechnen, durchgeführt.

Das XP-Game wurde bis jetzt vier Mal mit je ca. 25 Studierende im Master-Studiengang im Rahmen des Moduls „Software Engineering and Architecture" (3 ECTS) [19] durchgeführt.

Interessant sind dabei folgende Beobachtungen:

- Die erste Iteration ist typischerweise sehr chaotisch. Die Schätzungen sind schlecht und die Selbstorganisation der Teams funktioniert nicht.
- Nach der Retrospektive am Anschluss an die erste Iteration nimmt die Selbstorganisation der Teams merklich zu. Ab der zweiten Iteration funktionieren die Teams, die Schätzungen werden viel besser.
- Die Velocity wird erstaunlich konstant.
- Der Lerneffekt in dieser kurzen Zeit ist sehr gross. Die Studierenden sind aktiv am arbeiten.
- Das XP-Game macht grossen Spass!

Das Feedback der Studierenden ist durchwegs positiv. Die Studierenden schätzen es insbesondere auf eine spielerische Art einen grossen Lerneffekt zu erzielen.

Scrum City Game

Ein alternativer Ansatz um agile Management Praktiken zu vermitteln und, insbesondere, zu erfahren, ist das Scrum Lego City Game [15]. Hier bekommen die einzelnen Teams die Aufgabe gestellt in einem Mini-Projekt während vier 10 minütigen Sprints eine Stadt aus Lego-Bausteinen nach vorgegeben User Stories zu realisieren.

Wir führen das Scrum Lego City Game in Gruppen von 6 bis 7 Studierenden in leicht abgewandelter Form über mehrere Wochen durch:

- An Stelle von (teuren) Lego-Bausätzen wird die Stadt aus Karton und Papier mit Farbstiften Scheren und Klebstoff gebaut.
- Das Schreiben der User Stories ist Teil der Aufgabenstellung. Um zu verhindern, dass die Entwickler ihre eigenen Anforderungen schreiben, bekommt der Product Owner (PO) eines Teams die Entwickler eines anderen Teams als Benutzer zugewiesen und definiert mit diesen die Anforderungen für „sein" Team.

- Auch die Priorisierung der User Stories durch den PO und das Schätzen durch das Entwicklerteam gehören zur Aufgabe.

Danach werden in insgesamt 3-4 Sprints die Aktivitäten wie Sprint-Planung, Task Breakdown, Sprint Review, Retrospektive, Arbeiten am Scrum Board in Mini-Sprints von 10 Minuten Dauer durchlaufen. Aufgrund der kurzen Sprints entfällt der Daily Scrum.

Die dabei gemachten Erfahrungen decken sich durchwegs mit jenen des XP Games. Ein weiterer wichtiger Aspekt ist die Einprägsamkeit der verschiedenen Konzepte durch das konkrete Anwenden in dieser Spielform.

Entwicklung eines Computerspiels mit Scrum

In einem neuen Software Engineering Modul im 5. Semester wird ein anderer Weg beschritten. Die Studierenden sollen die Individuelle-, Team- und Werte-Ebenen während einem Semester möglichst intensiv erfahren und verinnerlichen. Die Vorlesung ist als eigentlicher „Crash"-Kurs für „agile Software Engineering" konzipiert.

In der ersten Hälfte des Semesters liegt der Fokus auf der Individuellen Ebene. Es gibt eine ausführliche Einführung in die *Engineering Practices* von eXtreme Programming: Coding Standards, Unit Testing, Continuous Integration (CI), etc. Die Studierenden werden mit dem automatischen Build Process bekannt gemacht und nach der Hälfte des Semesters haben sie bereits einen kompletten CI-Server in Betrieb genommen.

In der zweiten Hälfte liegt der Fokus auf der Team Ebene. Die Studierenden sollen in Gruppen von sechs bis acht Mitgliedern während den restlichen sieben Wochen ein eigenes 2D-Computerspiel entwickeln. Dazu bekommen sie eine ausführliche Einführung in Scrum sowie in agiler Schätzung und Planung.

Die Studierenden schreiben die User Stories, schätzen die Tasks und planen die Iterationen. Eine Iteration dauert nur eine Woche und der Umfang ist dementsprechend klein. Längere Iterationen sind nicht zweckmässig, da das Projekt mit nur sieben Wochen extrem kurz ist.

Die Dozierenden nehmen bei jeder Gruppe am wöchentlichen Sprint-Meeting teil und unterstützen die einzelnen Teams. Es wird grosser Wert auf die Planung und Retrospektive gelegt.

Die Dozierenden coachen die einzelnen Teams und geben Unterstützung bei auftretenden Schwierigkeiten. Sie versuchen auch, den Teams einen „Spiegel" vorzuhalten, damit sie ihre Entscheidungen reflektieren und dadurch die Konsequenzen der getroffenen (und natürlich auch der nicht getroffenen) Entscheidungen besser verstehen.

Der Wert dieses Scrum-Projekts besteht in der Kombination der *Engineering Practices* mit den *Management Practices* in einer realistischen Projektsituation. Während des Projekts gibt es auch oft Gelegenheit für den Coach, die Werte der agilen Softwareentwicklung einfliessen zu lassen.

Projekt- und Bachelorarbeiten

In den Projekt- und Bachelorarbeiten ist es den einzelnen Dozierenden überlassen, ob sie die Aufgabe mit einer agilen Projektmethode lösen lassen oder nicht.

Vor allem in der sogenannten „Projektwoche", in der die Projektteams einmal pro Semester eine ganze Woche am Stück an ihren Projekten arbeiten können, wird insbesondere das Taskboard für die intensive Teamarbeit genutzt.

Bei externen Projekten ist es eine Herausforderung auch die externen Auftraggeber für das agile Vorgehen zu überzeugen, da dies ein grösseres Engagement des Kunden erfordert. Die Erfahrungen mit solchen Projekten sind jedoch mehrheitlich positiv.

Eine Schwierigkeit, Projektarbeiten nach agilem Vorgehen zu organisieren, ist, dass in diesen Arbeiten die Teams aus Zeitgründen in der Regel nur einmal pro Woche gemeinsam am Projekt arbeiten können, sonst eher getrennt (z.B. Zuhause). Ein „Daily Standup" ist daher in der Regel nicht möglich. Die Studierenden machen daher eher ein Meeting, in dem sie sich gegenseitig auf den neuesten Stand bringen. Statt einer Pinnwand als Scrum Board, werden häufig digitale Boards eingesetzt, die jedoch nicht so einfach zu bedienen sind wie Pinnwände und nicht die gleiche Flexibilität aufweisen. Daher werden diese oft nicht mit der notwendigen Sorgfalt verwaltet.

Erfahrungen

Welche Erfahrungen haben wir nun mit dem Unterrichten von agilen Methoden gemacht, und wie wird dieses Vorgehen von den Studierenden aufgenommen?

Unsere Erfahrungen sind durchwegs positiv, auch wenn noch einige Herausforderungen bestehen. Die Studierenden sind sehr offen gegenüber den agilen Methoden, da sie den Fokus stärker auf die eigentlichen Interessen der Studierenden, die Konstruktion der Software durch Design und Programmierung, legen. Das Feedback der Studierenden ist entsprechend ermutigend.

Gerade auch der Nutzen von Test-getriebener Entwicklung und der Automatisierung mittels CI Umgebungen wird für die Studierenden sehr schnell

ersichtlich und, erfreulicherweise, von diesen dann selbst in Projektarbeiten eingesetzt.

Unterschätzt wird häufig das disziplinierte Vorgehen, welches agile Methoden vom gesamten Projektteam z.B. bei der iterativen Planung und der Software Qualität, einfordern. Hier gilt es sicherlich von der Ausbildungsseite her, ein Augenmerk darauf zu haben.

Die Simulationen von agilem Vorgehen mittels Games (XP, Scrum) hat sich sehr bewährt. In den Games „erfahren" die Studierenden wichtige agile Konzepte wie selbst-organisierte Teams und häufige Auslieferungen in sehr intensiven Mini-Projekten, die sich entsprechend einprägen.

Im Hinblick darauf, dass vor allem die agilen Werte, aber auch deren Praktiken nicht durch einmaliges Vermitteln sondern durch Vorleben und ständige Anwendung erlernt werden müssen, ist es wichtig, dass diese Praktiken auch in den sonstigen Programmiermodulen zum Einsatz kommen.

Die neuen zusätzlichen Anforderungen an die Dozierenden sind dabei nicht zu unterschätzen. Die Dozierenden der entsprechenden Module müssen sich die notwendigen Grundlagen aneignen. Die Vorlesungen müssen entsprechend angepasst, und die agilen Praktiken in die eigenen Module intergiert werden. Der höhere Aufwand der Dozierenden ist sicher eine Herausforderung, welcher aber aus unserer Sicht durch das gute Resultat mehr als gerechtfertigt ist.

Ein weiterer wichtiger Vorteil der Vermittlung der agilen Methoden ist, dass die vor allem von XP propagierten allgemeinen Best Practices viel stärker ins Bewusstsein der Studierenden rücken und auch eine stärkere Akzeptanz erfahren, da mit jeder Iteration eine Qualitätskontrolle stattfindet.

Zusammenfassung und Ausblick

Wie unsere und auch internationale Studien zeigen, sind agile Software Entwicklungsmethoden in der Praxis sehr stark im Kommen. In der Ingenieur-Ausbildung müssen wir uns den geänderten Kompetenzanforderungen stellen und unsere Ausbildungskonzepte entsprechend anpassen und erweitern.

Im vorliegenden Artikel haben wir beschrieben, wie wir die Vermittlung von agilen Softwareentwicklungsmethoden an unseren Fachhochschulen integriert haben. Die Erfahrungen damit sind sehr positiv und die erzielten Erfolge stimmen zuversichtlich. Erste Feedbacks aus der Praxis über die Kompetenzen der Abgänger bestätigen unseren eingeschlagenen Weg.

Eine generelle Herausforderung bleibt, die agilen Konzepte nicht nur in einzelnen Modulen zu unterrichten, sondern diese in möglichst vielen programmier-nahen Modulen anzuwenden und zu vermitteln, was generell ein Wechsel des Unterrichtskonzept der isolierten Fachmodule hin zu einem integrativem Gesamtstudienkonzept bedeutet.

Auf Master-Stufe wollen wir die Vermittlung fortgeschrittenen *Engineering* und *Management Practices* weiter ausbauen, da diese in Zukunft eine zunehmende Bedeutung erlangen werden.

Zum Schluss sei angemerkt, dass auch die agilen Softwareentwicklungsmethoden nicht das Ende der Entwicklung der Softwareprozesse darstellen. Die Informatik ist immer noch eine sehr junge Disziplin, deren Entwicklungsprozessmodelle sich stetig weiterentwickeln werden. Aus Sicht der Ingenieur-Ausbildung ist es daher zentral, sich abzeichnende Entwicklungen in der Ausbildung vorwegzunehmen und damit einen Beitrag dazu zu leisten, dass moderne Entwicklungsmethoden ihren Einzug in die Praxis halten.

Referenzen

[1] Kropp, M., Meier, A. Swiss Agile Study - Einsatz und Nutzen von Agilen Methoden in der Schweiz. www.swissagilestudy.ch, 2012 (Bericht in Bearbeitung).

[2] Version One. State of Agile Development Survey results. http://www.versionone.com/state_of_agile_development_survey/11/, 20.10.2012

[3] Software Engineering and Architecture. http://www.msengineering.ch/fileadmin/user_upload/modulbeschreibungen/de/TSM_SoftwEng_de.pdf, 29.10.2012.

[4] http://www.zhaw.ch/, 29.10.2012

[5] http://www.fhnw.ch/technik, 29.10.2012

[6] Agile Manifesto. http://agilemanifesto.org/, 29.10.2012.

[7] K. Schwaber, M. Beedle. Agile Software Development with Scrum. Prentice Hall, 2001.

[8] K. Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2009

[9] M. und L. Poppendieck. Lean Software Development: An Agile Toolkit. Addison-Wesley, 2003.

[10] S. Hauser, M. Kropp. Software Projekt Management. http://web.fhnw.ch/plattformen/spm/ , 29.10.12

[11] Ch. Denzler, M. Kropp. Software Construction. http://web.fhnw.ch/plattformen/swc. 29.10.2012

[12] R.C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008.

[13] R. Mugridge, W. Cunningham. Fit for Developing Software: Framework for Integrated Tests. Prentice Hall, 2005.

[14] XP Game. XP-Game, http://www.xpgame.org, 29.10.2012

[15] Scrum Lego City Game. http://www.agile42.com/en/training/scrum-lego-city, 29.10.2012

[16] Checkstyle Homepage. http://checkstyle.sourceforge.net/, 31.10.2012

[17] Ch. Denzler. http://web.fhnw.ch/plattformen/swent, 31.10.2012

[18] W. Cunningham. The WyCash Portfolio Management System. OOPLSA 1992.

[19] Modul Software Engineering & Architecture, Master Of Science of Engineering, http://www.msengineering.ch/fileadmin/user_upload/modulbeschreibungen/de/TSM_SoftwEng_de.pdf, 18.12.2012

[20] D.F. Rico; H.H. Sayani. "Use of Agile Methods in Software Engineering Education," Agile Conference, 2009. AGILE '09. , vol., no., pp.174-179, 24-28 Aug. 2009.

[21] A. Shukla; L. Williams. "Adapting extreme programming for a core software engineering course ," Software Engineering Education and Training, 2002. (CSEE&T 2002). Proceedings. 15th Conference on , vol., no., pp.184-191, 2002.

Kanban im Universitätspraktikum Ein Erfahrungsbericht

Jan Nonnen, Paul Imhoff und Daniel Speicher, Universität Bonn
{nonnen, imhoffj, dsp}@cs.uni-bonn.de

Zusammenfassung

Agile Softwareentwicklung praxisnah und erlebbar zu lehren ist eine Herausforderung, die einen deutlichen Einsatz von Zeit und Personal erfordert. Wir hatten die Möglichkeit, viele gute Erfahrungen in unserer Lehre sammeln zu können. Allerdings hatten wir immer noch einige Schwierigkeiten den Studenten bestimmte Probleme während der Entwicklung bewusst und sichtbar zu machen. In dieser Arbeit berichten wir von unseren Erfahrung mit der Anwendung des Kanban-Entwicklungsprozesses in einem agilen Blockpraktikum an der Universität Bonn. Tatsächlich hat dieser einige dieser Schwierigkeiten deutlicher sichtbar machen können. Basierend auf unseren Erfolgen, bewältigten Herausforderungen und neu beobachteten Schwierigkeiten geben wir abschließend Empfehlungen für andere Lehrende.

Einleitung

Agile Softwareentwicklung ist in den letzten Jahren - mehr noch in der Wirtschaft als in der Lehre - beliebt geworden und hat eine große Verbreitung erfahren (Lindvall u. a., 2004). Dem muss auch die universitäre Softwaretechnik-Lehre Rechnung tragen und sei es nur durch die Befähigung zur kritischen Reflexion. Hier stellt sich aber eine besondere Herausforderung: Agilität möchte der oft erfahrenen Wirklichkeit Rechnung tragen, dass sich Softwareentwicklung nur eingeschränkt planen lässt. Diese Erfahrung und wie sie sich durch agile Vorgehensweisen meistern lässt, ist auf theoretischem Weg kaum vermitteln. Daher haben einige Universitäten spätestens seit 2003 begonnen, Praktika zu agiler Softwareentwicklung in ihre Lehrpläne aufzunehmen (Mügge u. a., 2004; Melnik u. Maurer, 2004). In unseren eigenen Praktika haben wir dabei stets als unerlässlich angesehen, "weiche" Faktoren mit einzubeziehen. So unterstützen wir die Teambildung durch regelmäßige Reflexion und ausdrückliche Retrospektiven, um die Studierenden in die Lage zu versetzen, Kommunikation, Kollaboration und den Prozess als Ganzes zu verbessern.

Ein aktueller Trend der letzten Jahre in der agilen Softwareentwicklung ist *Kanban* (Anderson, 2010). In diesem Beitrag erläutern wir, wie wir Kanban in einem universitären Praktikum zur agilen Softwareentwicklung eingesetzt haben. Wir berichten von unseren Erfahrungen, positiv und negativ, und präsentieren anderen Lehrenden Empfehlungen aufgrund dieser Erfahrungen für ihre Anwendung von Kanban.

1. Visualize Workflow *Visualisiere den Fluss der Arbeit*
2. Limit Work-in-Progress *Begrenze die Menge angefangener Arbeit*
3. Measure and Manage Flow *Miss und steure den Fluss*
4. Make Process Policies Explicit *Mache die Regeln für den Prozess explizit*
5. Use Models to Recognize Improvement Opportunities - *Verwende Modelle um Verbesserungsmöglichkeiten zu identifizieren*

Tabelle 1: Die fünf Kanban-Kerneigenschaften nach Anderson (Anderson, 2010, S. 15).

Wir haben unseren Bericht wie folgt strukturiert: Zuerst geben wir eine allgemeine Einführung in Kanban und berichten über den Praktikumsaufbau und die Historie unserer Praktika. Danach beschreiben wir, wie wir unseren Prozess mit Hilfe von Kanban angepasst haben. Anschließend berichten wir, was wir während des Praktikums geändert haben und am Ende ziehen wir ein Fazit und präsentieren unsere Empfehlungen für Lehrende.

Kanban

Kanban ist ein agiler Software-Entwicklungsprozess, dessen Ursprünge im *Toyota Production System* liegen. Entwickelt wurde das Toyota Production System von Taiichi Ōno, der es auch in seinem 1988 auf Englisch erschienen Buch beschrieb (Ōno, 1988). Das japanische Original war bereits 10 Jahre zuvor erschienen.

"kan" bedeutet wörtlich aus dem Japanischen übersetzt "visuell", und "ban" Karte. Toyota nutzte Kanban, um die Lagerbestände zu reduzieren und einen gleichmäßigen Fluss (Flow) in der Automobilfertigung zu realisieren. David Anderson hat diesen Prozess auf Softwareentwicklung übertragen (Anderson, 2010) und durch fünf Kerneigenschaften charakterisiert (siehe Tabelle 1).

Für Kanban ist die sichtbare Tafel oder ein Whiteboard mit einer Übersicht über den Entwicklungszustand ein zentrales und wohl auch das offensichtlichste Instrument.

Zur Illustration gehen wir schon hier kurz unser eigenes Kanban Board ein: Bereits in der agilen Entwicklung und in unseren vorherigen Praktika war und ist es üblich, die sogenannten *User Stories* auf einem Board zu visualisieren. Dieses Board hatte Spalten für die Entwicklungsschritte wie z.B. ”Analyse” oder ”Done”. In Kanban wird diese Praxis weiter verwendet und angereichert durch die in den fünf Kerneigenschaften genannten Imperative. In Abbildung 1 sieht man ein Bild von einer Kanban Tafel aus dem hier vorgestellten Praktikum. Diese Visualisierung wurde mit Stickern, Whiteboard-Stiften und Klebeband realisiert. Die Verwendung dieser Materialien erlaubte uns, flexibel das Board an das Entwicklerteam und andere Einflüsse anzupassen, und so die Entwicklungsschritte für das Team zu individualisieren. Diese Flexibilität ist wichtig, da Kanban auf die Bedürfnisse zugeschnitten werden sollte und sich diese änderen können. Es sollte daher auch mit dem Team diskutiert werden, was die Phasen (bisher Entwicklungsschritte) sind, durch die die Tickets (bisher User Stories) bis zur Fertigstellung wandern.

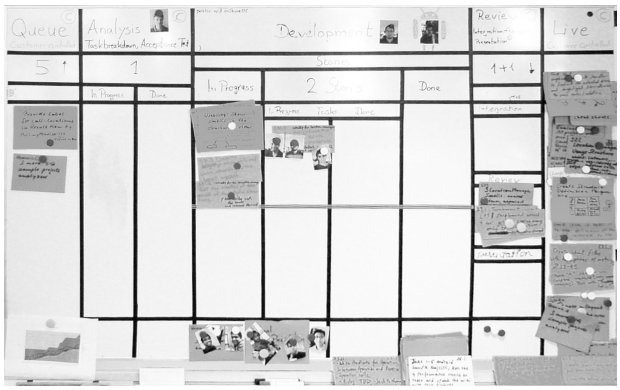


Abbildung 1: Im Praktikum verwendetes Kanban-Board. Das weiter unten referenzierte Video bietet zusätzlich eine dynamische Perspektive auf das Board.

Im Folgenden verwenden wir allgemeine Kanban-Konventionen und orientieren uns an Kanban-Boards welche wir selber eingesetzt und entwickelt haben (siehe Abbildung 2).

Die Stories oder Tickets laufen in der Regel von links nach rechts auf dem Kanban Board und durchlaufen dabei die Entwicklungsphasen bis sie fertig (”done”) sind. Nachdem durch die 2. Kerneigenschaft die Tickets, die in Arbeit sind, begrenzt werden sollten, werden je Spalte Begrenzungen definiert und auf dem Board notiert. Dies bedeutet, dass sobald das Limit in einer Phase erreicht ist, nicht neue Tickets in diese gezogen werden dürfen sondern die Ressourcen genutzt werden müssen um die in Arbeit stehenden Tickets fertig zu stellen.

Kanban basiert auf dem so genannten *Pull-Prinzip*, d.h. dass die Entwickler ein Ticket selbstständig in eine Spalte ziehen, solange die Arbeitslimits in der entsprechenden Phase noch nicht erreicht sind. Sie übernehmen damit die Verantwortung für den der Spalte entsprechenden Schritt in Bezug auf das ”gezogene” Ticket. Des Weiteren werden häufig die einzelnen Phasen in zwei Spalten unterteilt, um zu visualisieren was ”In Arbeit” ist und welche Tickets ”Fertig” sind. Dies erlaubt es in den nachfolgenden Phasen zu sehen, welche Tickets als nächstes gezogen werden dürfen.

Durch die explizite Visualisierung des Ist-Zustandes ist es möglich, den Gesamtprozess zu messen und zu analysieren, wie schnell die einzelnen Tickets durch den Prozess wandern (4. Kerneigenschaft). Ebenso werden Probleme in der Entwicklung sichtbar, wenn man z.B. feststellt, dass die Limits in einer Phase häufig erreicht werden oder die Entwickler in einer Phase regelmäßig auf Arbeit warten müssen.

Diese Informationen können mit genutzt werden, um den Prozess kontinuierlich und gemeinsam zu verbessern (5. Kerneigenschaft). Hier werden häufig auch mathematische Modelle auf Basis der ”Theory of Constraints” (Goldratt u. a., 1984) angewandt um Engpässe (Bottlenecks) in dem Prozess frühzeitig zu erkennen. Diese Modelle geben auch die Möglichkeit, objektiv über den Prozess zu diskutieren. Neben den Limits kann es noch andere Regeln geben die die Bearbeitung von einem Ticket verhindern oder priorisieren (z.B. bestimmte Kartenfarben signalisieren Dringlichkeit).

Damit solche Konventionen nicht zu Fehlern durch Vergessen führen, ist es nützlich, alle Konventionen, die sich im Lauf der Entwicklung ergeben, explizit aufzuschreiben und allen Entwicklern bewusst zu machen (3. Kerneigenschaft). Dieses kann zum Beispiel durch ein Poster mit den Konventionen neben dem Whiteboard geschehen. Dieses fördert analog zu der Verbesserung des Flusses die gemeinsame Diskussion und Verbesserung der Konventionen und Regeln.

Aufbau des Praktikums

Das dieser Arbeit zugrundeliegende Praktikum fand für vier Wochen als Vollzeit-Blockpraktikum im September 2011 während der Vorlesungspause zwischen Sommer- und Winteresemester statt. Blockpraktika dieser Art finden jährlich im Rahmen des *International Program of Excellence in Computer Science (IPEC)* am *Bonn-Aachen International Center for Information Technology (B-IT)* statt, und bieten den Studenten die Möglichkeit Erfahrungen als Softwareentwickler in einem realen Kontext zu sammeln. Seit 2001 werden in den Praktika agile Softwareprozesse gelehrt, angewandt und in Hinblick auf die Lehre angepasst (Mügge u. a., 2004). In der Vergangenheit haben bis zu 16 Studenten in einem Team gearbeitet, welches

dazu von bis zu drei Lehrkräften die ganze Zeit betreut wird.

Dem Praktikum unmittelbar vorgelagert ist ein kleiner Workshop von bis zu drei Tagen. Für diesen vertiefen sich die Studenten in die einzelne Technologien, die im Praktikum genutzt werden, und arbeiten diese in Form von Seminarvorträgen und Ausarbeitungen auf. Neben den Vorträgen wird dieser Workshop vor allem zur Teambildung genutzt. Im Normalfall haben die teilnehmenden Studenten vorher noch nicht zusammengearbeitet. Daher sind teambildende Maßnahmen in der Regel fest in dem Workshop eingeplant. Dieser Workshop hat sich im Laufe der Jahre als nützlich erwiesen, um im Praktikum direkt gemeinsam und produktiv als Team arbeiten zu können.

Der Fokus des Blockpraktikums ist es, einen Einblick in die reale Softwareentwicklung zu geben. Dazu ist das Praktikum als Vollzeit-Präsenzpraktikum ausgelegt, d.h. dass das Team für vier Wochen mit je acht Stunden pro Arbeitstag zusammen arbeitet. Dies hat für die Studenten den Vorteil, das neben dem normalen Arbeitstag keine Überstunden oder Heimarbeiten anfallen. Als eine wichtige Arbeitstechnik hat sich in diesen Praktika das *Pair-Programming* aus dem "eXtreme Programming" (Beck u. Andres, 2004) etabliert. Pair-Programming bedeutet, dass immer zwei Studenten ein Ticket gemeinsam an einem Rechner bearbeiten. Damit sich Wissen zwischen möglichst vielen Teilnehmern gut verteilt, ermutigen wir die Studenten zu regelmäßigem Wechsel des Programmierpartners. Wir halten dafür in einer Matrix fest, wer schon mit wem zusammen entwickelt hat. Dieses Verfahren wurde aus früheren Praktika ohne Änderung übernommen.

Allgemein beinhaltet der tägliche Arbeitstag ein Stand-up Treffen am Morgen und am Abend, sowie ein gemeinsames selbstorganisiertes Teamfrühstück vor Beginn der Arbeit. Wöchentlich gibt es eine Retrospektive (Beck u. Andres, 2004) in der reflektiert wird, was in der Woche erreicht wurde und was man verbessern könnte.

Die drei Lehrkräfte üben während des Praktikums feste aber verschiedene Rollen aus. Wie bereits in (Mügge u. a., 2004) ausgeführt hat sich die Dreiteilung *Kunde*, *Teamleiter* und *technischer Experte* am besten bewährt. Der Kunde legt fest, was er als Produkt entwickelt haben möchte und welche Funktionalitäten er sich wünscht. Das resultierende Produkt kann entweder etwas sein, was in der Forschung genutzt wird oder was von der Industrie an uns herangetragen wird[1]. Der Kunde ist auch in den täglichen Entwicklungsprozess eingebunden wie im Extreme Programming üblich. Das tägliche Projektmanagement des Teams wird von der Rolle des Teamleiters ausgeübt. Dieser stellt die Brücke zwischen dem Entwicklerteam und dem Kunden dar. Die letzte Rolle ist die eines technischen Experten. Dieser ist Ansprechpartner bei technischen Fragen des Teams und hat Wissen in den einzusetzenden Technologien und Projekten.

Anwendung von Kanban

Das Thema des Praktikums in 2011 war die Entwicklung von Analysen für Designprobleme von Java Quellcode für die Android Plattform. Insgesamt nahmen zehn Studenten an dem Praktikum tatsächlich teil, nachdem ursprünglich 16 Plätze vergeben waren. Als Basis gab es schon ein selbstentwickeltes Framework mit dem statische Codeanalysen für Java geschrieben werden konnten. Dieses musste im Rahmen des Praktikums erweitert werden. Ebenso mussten die Studenten erfassen, was hinsichtlich vor allem Performanz und Speicherbedarf problematischer Code auf der Android Plattform ist. Dafür wurden im Laufe des Praktikums z.B. die Entwickler-Richtlinien von Google[2] zu Rate gezogen.

Zur Vorbereitung und Anwendung von Kanban als Entwicklungsprozess wurden die früheren Arbeitsschritte in früheren Praktika analysiert und die Erfahrungen festgehalten. Zu den Bestandteilen früherer Praktika, die wir für erhaltenswert befunden haben gehören (wie bereits beschrieben) das Pair Programming, die Rollenverteilung unter den Mitarbeitern, sowie (wie weiter unten beschrieben) die Differenzierung zwischen Stories und Tasks und ein starker Fokus auf die Selbstorganisation des studentischen Entwicklerteams. Letztere haben wir unter anderem durch die Anleitung zu Stand-Up Treffen und Retrospektiven unterstützt.

Über die Jahre sind wir in den Praktika jedoch auch auf wiederkehrende Schwierigkeiten gestoßen. So ist es immer wieder vorgekommen, dass fast fertige Stories nicht endgültig fertiggestellt wurden. Die letzten integrierenden Arbeitsschritte wurden von den Studenten sowohl als unerfreulich als auch als technisch anspruchsvoll wahrgenommen. Daher wurden andere Arbeiten der Fertigstellung vorgezogen, auch wenn sich das Team bereits dieses Problems bewusst war. Sicherlich hätte sich dieses Problem durch Zuordnung einzelner zu diesen Arbeitsschritten lösen lassen, aber das hätte dann unser Bemühen das Team in seiner Selbstorganisation zu unterstützen deutlich konterkariert. Mit Kanban erhofften wir uns, dass durch die Kombination des Pull-Prinzips mit strikten Limits für die einzelnen Phasen, die Sichtbarkeit des Problems so deutlich würde, dass sich das Team seiner annehmen würde.

Eine weitere Herausforderung, die von den Studenten deutlicher wahrgenommen wurde als von uns, war den Aufwand für Besprechungen - insbesondere zum Anfang einer Iteration - zu reduzieren. In nicht wenigen Praktika verbrachten die Studenten einen Tag oder sogar etwas mehr damit, in die Produktwünsche des Kunden eingeführt zu werden. Bei einer

[1] Ein Beispiel für ein entwickeltes Produkt: `http://www3.uni-bonn.de/Pressemitteilungen/299-2009`

[2] `http://developer.android.com/guide/practices/performance.html`

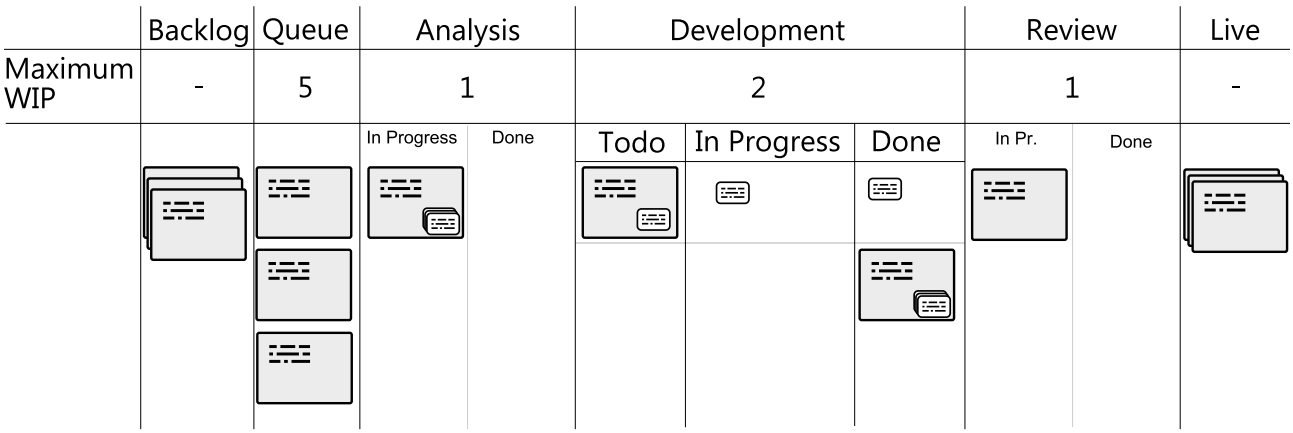


Abbildung 2: Schematische Darstellung des Kanban-Boards des Praktikums mit den einzelnen Phasen und Limits.

Teamgröße von mindestens 10 Studenten führt dies zumindest vorübergehend zu einiger Passivität. Da Kanban weniger Wert auf Iterationen legt bzw. sogar ganz ohne diese auskäme, versprachen wir uns, die initialen Besprechungen etwas entzerren zu können.

Das Ende der Iterationen hingegen stellte den Kunden vor die Herausforderung, Funktionalitäten in einem Umfang überprüfen zu müssen, den er nicht bewältigen konnte. Daher war das Feedback des Kunden über die Qualität des Geleisteten mitunter eher unscharf. Durch einen gleichmäßigeren Fluss fertiggestellter Funktionalität hofften wir dieses Problem entschärfen zu können. Dieses konnten wir schließlich durch vorhergehende Reviews durch Team und Teamleiter beheben.

Basierend auf den Erfahrungen bisheriger Praktika wurde der grundsätzliche Arbeitsfluss in einem initialen Kanban-Board festgehalten. Das initiale Board enthielt die folgenden sechs Phasen: Backlog, Input Queue, Analysis, Development, Review und Live (Abbildung 2).

Das Backlog enthält die Ideen des Kunden, die in noch nicht ausreichend genauer Form definiert sind oder die (noch) nicht realisierbar sind. Dies entspricht dem Backlog des Scrum-Prozesses mit dem Unterschied, dass die Tickets noch nicht von dem Team geschätzt wurden und sie noch in Bearbeitung sind. Die ”Input Queue” Phase wird von dem Kunden kontrolliert und bestimmt, welches das nächste zu entwickelnde Feature ist.

In der Analyse wird mit dem Kunden zusammen festgelegt, was alles zu der Realisierung des Tickets gehört, d.h. es werden Akzeptanzkriterien festgelegt. Außerdem nimmt das Team in der Analyse eine Zerlegung des Tickets in kleinere sogenannte Tasks vor. Diese Unterscheidung von Tickets, die einen durch den Kunden wahrnehmbaren Funktionalitätsfortschritt haben, (User Stories) und den dazu nötigen Arbeitsschritten (Tasks) haben wir aus den vorhergehenden Praktika übernommen. Dadurch wird es möglich einerseits die Teilschritte genau zu beschreiben und auf unterschiedliche Entwicklerpaare zu verteilen und andererseits bleibt der angestrebte Fortschritt sichtbar und verschwindet nicht hinter technischen Details. Die Implementation findet in der Development-Phase statt. Anschließend werden sowohl ein Team-interner Codereview, als auch ein gemeinsamer Review mit dem Kunden durchgeführt. Nach erfolgreichem Abschluss der Reviews ist ein Ticket in der Live-Phase angelangt.

Zur Visualisierung der Phasen auf einem Whiteboard wurde je Phase eine Spalte verwendet (siehe Abbildung 2). Die Phasen Analyse, Development und Review wurden noch weiter unterteilt in die Zustände ”In Progress” sowie ”Done”. Die Arbeitslimits wurden in einer separaten Zeile je Phase dargestellt. Ab der zweiten Praktikumswoche wurde ein Zeitraffervideo des Kanban-Boards erstellt[3]. In diesem sieht man, wie einzelne Tickets von links nach rechts durch die einzelnen Phasen des Prozesses wandern.

Wie erwähnt unterteilt das Team die anfänglichen Tickets des Kunden (die User Stories) in kleinere Tasks, die von den Studenten jeweils in einem Paar in einer kurzen Zeit bearbeitet werden können. In der Entwicklungsphase wandern daher die Task-Karten für ein Ticket von links nach rechts. Ein Ticket ist dann fertig implementiert, wenn alle Tasks des Tickets fertig implementiert wurden. Da es aber Tickets geben kann mit wenigen oder auch welche mit vielen Tasks, gibt es für diese Tasks keine Beschränkung hinsichtlich maximal erlaubter Anzahl an Tasks, die zeitgleich bearbeitet werden dürfen. Hingegen wurde die Anzahl an Tickets - also der User Stories zu den Tasks - in der Development-Phase beschränkt.

Während des Praktikums wurde der aktuelle Zustand zu festen Zeiten von allen Tickets auf dem Kanban Board gemessen und festgehalten. Aus diesen Daten wurde anschließend ein ”Cumulative-Flow” Diagramm (Anderson, 2010) (siehe Abbildung 3) generiert. In dem Diagramm wird über die Zeit aufgetragen, wieviele Tickets sich zu einem Zeitpunkt

[3] http://www.youtube.com/watch?v=0iPaIefQbgM

im System befinden und welchen Zustand diese haben. Dabei zählen fertig bearbeitete Tickets zu der Live-Phase. Eine breite Phase in dem Diagramm kann entweder bedeuteten, dass ein Ticket lange in dieser Phase bearbeitet wird, oder dass die nächste Phase nicht verfügbar ist und die Tickets nicht weiterbearbeitet werden können. Dieses Diagramm wurde im Praktikumsraum neben dem Board aufgehangen und diente als Visualisierung des Fortschritts in täglichen Stand-Up Besprechungen mit dem Team.

In Kanban-Prozessen findet man häufiger spezialisierte Entwicklerrollen (z.B. Architekten oder Tester) die in speziellen Phasen arbeiten. In unseren Praktika legen wir Wert darauf, dass die Studenten alle Entwicklungsphasen erfahren, so dass wir keine spezialisierten Rollen genutzt haben. Als Konsequenz daraus waren die Studenten frei in der Wahl in welchem Entwicklungsschritt sie arbeiten möchten, solange die Kanban-Limits berücksichtigt wurden. Allerdings gab es Priorisierungen von zu bearbeitenden Tickets im System.

Initial war die einzige Regelung, dass Tickets in späteren Phasen bevorzugt weiterbearbeitet werden sollten, damit der Kunde zügiger fertige Features bekommt. Auch wenn diese Regel bereits dem Toyota Production System entstammt, war sie für uns besonders plausibel, da in früheren Praktika - wie oben erwähnt - die Integration von Tasks zum Ticket und der Review unbeliebter waren. In der Konsequenz wurde lieber ein neues Ticket analysiert als ein fast fertiges Ticket fertigzustellen und dem Kunden zu präsentieren.

Anpassungen während des Praktikums

Während des Praktikums wurden einige Anpassungen an den Arbeitsfluss und die Visualisierung des Prozesses vorgenommen. Im Folgenden möchten wir diese genauer beschreiben und vorstellen. Der Hauptgrund für die Anpassungen war ein ”verebben” des Ticketflusses innerhalb des Prozesses. Ebenso wurden Veränderungen aus didaktischen und allgemeinen praktischen Gründen vorgenommen.

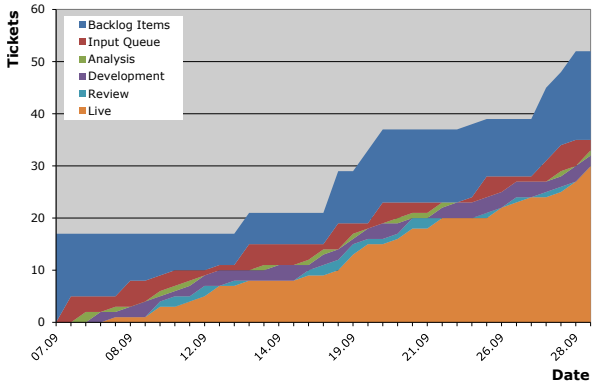


Abbildung 3: ”Cumulative-Flow” Diagramm des Praktikums

Das Verebben des Ticketflusses wurde deutlich anhand der Anzahl der Entwickler die keine Arbeit hatten. Dies wurde teilweise durch blockierende Tickets in späteren Phasen erzeugt. Im Kanban-Prozess, sollten idealerweise die freien Ressourcen mithelfen die Engpässe zu beheben. Dies war aber bei unseren feingranularen Tasks die schon auf ein Entwickler Paar geplant waren selten möglich. Ebenso zeigte sich, dass unsere oben beschriebene anfängliche Priorisierung ungünstig gewählt war. Durch die Engpässe in der Development-Phase, gab es wenige bis keine Tickets für den Review. Nachdem aber auch der Review priorisiert werden sollte, wurde eher der Review gemacht bevor ein neues Ticket in die Analyse-Phase gezogen wurde.

Es stellte sich auch heraus, dass ein Ticket in der Analyse-Phase nicht von mehreren Paaren effektiv bearbeitet werden konnte. Daraus resultierte ein Stocken des Entwicklungsprozesses, in dem keine Tickets in der Analyse oder fertig für die Development-Phase waren. Als Lösungsmöglichkeiten hätten wir die Arbeitslimits in der Development-Phase anpassen können um ein Entwicklerpaar frei zu haben für Analyse oder Review. Wir passten aber unsere unglücklich gewählte Priorisierung an, indem wir die Priorisierung Analyse -> Review -> Development vorschlugen. Dies stellte sich im Laufe des Praktikums als sinnvolle Entscheidung heraus, da es den Engpass entlastete und die Studenten in der Development-Phase Tickets zum Bearbeiten hatten.

Ein weiterer Engpass in dem Prozess war die Rolle des Kunden. Dieser wurde in der Analyse-Phase zur Akzeptierung der genaueren Anforderungen und in dem finalen Schritt der Review-Phase benötigt. Da es nur einen Kunden gab, war dieser eine begrenzte Ressource. Ebenso war es aber weder sinnvoll, die Anzahl an Kunden zu erhöhen noch die Einbindung des Kunden in beide Schritte zu vermeiden, daher wurde versucht, durch den Teamleiter in beiden Schritten einen vorgelagerten Kundenrepräsentanten zu spielen. Dies bewirkte, dass viele Fragen der Studenten in diesen Schritten von dem Teamleiter beantworten werden konnten und der Kunde trotz alledem die fertigen Dokumente präsentiert bekam und selber entscheiden konnte, ob sie fertig sind.

Für die meisten Studenten war es das erste Mal, dass sie in einem größeren Entwicklerteam an einem gemeinsamen Projekt arbeiten mussten. Der Kanban-Prozess war keinem der Studenten vor dem Praktikum bekannt. Eine Einführung in den Kanban-Prozess und das Kanban-Board war zwar durch in den Workshop gegeben, aber diese war nach unserer jetzigen Erfahrung zu kurz. Aus diesen beiden Tatsachen resultierten während des Praktikums Unsicherheiten der Studenten, was zu tun ist oder wo sie helfen konnten. Nachdem die Erfahrung mit dem Prozess und dem

Board merklich zunahm, wurden die Zeiten, in denen die Studenten aufgrund der Limits keine Arbeit am Projekt leisten konnten, für die Studenten demotivierend.

In der Kanban-Literatur (Anderson, 2010) wird dieser nützliche Spielraum ("Slack") genutzt, um einerseits den Prozess zu optimieren und andererseits den Entwicklern die Möglichkeit zu bieten, sich selbstständig fortzubilden. Als Fortbildungsmöglichkeit definierten wir einige Möglichkeiten in Form von Tutorials und Literatur, die sich um die benutzte Technologie oder solche, die später noch genutzt werden sollte, drehten. Diese hatten damit den zusätzlichen Nutzen für den Kunden, dass das Wissen über die Technologien frühzeitig im Team verteilt werden konnte. Die Studenten hatten bei Leerlauf die freie Wahl aus diesen auf Karten beschriebenen Fortbildungen zu wählen.

Neben diesen Karten konnten die Studenten auch jederzeit Vorschläge oder Ideen einbringen, wie man den Gesamtprozess oder das Board verbessern könnte. Ein Vorschlag war zum Beispiel, dass nicht immer klar war welche Studenten an welchen Tickets arbeiteten. Nachdem die Tickets in der Development-Phase in meist mehrere Tasks unterteilt und mit je einem Paar weiterentwickelt wurden, war es wichtig, dass sich diese Studenten über den Zustand austauschen, da es häufig Abhängigkeiten zwischen den Tasks gab. In diesen Schritten ist es daher wichtig, zu wissen, wer an welchen Tasks arbeitet um sich z.B. räumlich nebeneinander zu setzen. Als Lösung des Problems wurde ein Bild von jedem Entwickler und Dozenten gemacht und auf dem Board befestigt. Jeder hatte dann die Aufgabe, sein Bild auf das Ticket oder den Bereich zu verschieben, an dem er arbeitete. Da es diese Bilder auch für Teamleiter und Kunden gab, half diese einfache Idee als Nebeneffekt auch, den Paaren, die in der Analyse oder dem Review arbeiteten, visuell zu erfassen ob der Kunde verfügbar war.

Fazit

Um zu bewerten, wie gut unser Prozess im Praktikum funktionierte, wurde das Cumulative Flow Diagramm (siehe Abbildung 3) herangezogen. Außerdem hatten wir an einem Tag während des Praktikums einen externen Prozess-Reviewer, der Entwickler-Teams in der Industrie bei der Einführung von Kanban professionell berät. Von ihm erhielten wir hilfreiches Feedback und wertvolle Tipps. Schließlich haben wir zudem die Studenten noch mit einem Fragebogen um ihre Einschätzung gebeten.

In unserer Nachbereitung stellten wir aber fest, dass es für uns schwer war zu entscheiden, ob uns das Diagramm eine Aussage über die Prozessqualität machen kann. Dies hängt vor allem daran, dass selten ein festes Paar ein Ticket durch den gesamten Prozess begleitet hat. Daher hängt die Dauer, die ein Ticket in einer Phase verbracht hat, nicht nur von den Eigenschaften des Tickets selber ab, sondern auch von den Entwicklern und der Kommunikation zwischen den Phasen. Durch die kurze Zeit des Praktikums ist es darum schwer aus den wenigen Messpunkten herzuleiten, ob Kanban rein objektiv zur Verbesserung des Gesamtprozesses beigetragen hat.

Als weiterer Kritikpunkt fügt unsere finale Priorisierung von Analyse > Review > Development dem Gesamtprozess noch eine zusätzliche Komplexität hinzu. Diese Entscheidung sollte auf der einen Seite bewirken, dass dem Kunden möglichst zügig Tickets als fertig vorgelegt werden konnten. Auf der anderen Seite sollte es durch die Priorisierung immer fertig analysierte Tickets geben, die direkt in den Development-Schritt gezogen werden konnten. Diese beiden Schritte waren gegeben, trotzdem waren wir mit der resultierenden Priorisierung nicht glücklich, da es eine höhere Aufmerksamkeit auf Seiten der Studenten forderte.

Durch den ungewohnten Entwicklungsprozess und das Umfeld waren die Studenten anfangs eher passiv in dem Prozess. Mit steigender Erfahrung mit Kanban und steigendem Selbstbewusstsein im Team fingen die Studenten an, Verbesserungsvorschläge zu machen. Durch eine bessere und längere Einführung von Kanban in dem Workshop könnte man diese Hemmschwelle vermutlich verringern und frühere Verbesserungsvorschläge erleichtern. Wie so oft in unseren Praktika war das Team in der letzten Woche erst auf voller Produktivität.

Eine Schwierigkeit in der Anpassung des Prozesses während des Praktikums ist der kurze Zeithorizont. Man kann entweder zur Optimierung Arbeitsschritte und Puffer hinzufügen oder entfernen, oder an den Limits in jedem Schritt drehen. Wenn man eines von beiden anpasst, soll man laut Literatur (Anderson, 2010) einige Zeit warten, bis sich der Prozess auf die neuen Rahmenbedingungen eingependelt hat.[4] Ebenso sollte man nur eine Schraube in jedem Schritt verändern. Dies bedeutete aber für unser Praktikum, dass es schwer war den Effekt abzuschätzen. Wir hatten uns daher auch entschieden während des Praktikums keine Arbeitsschritte anzupassen und nur die Limits der Schritte zu justieren. Davon haben wir auch in der ersten Woche Gebrauch gemacht, indem wir die Limits in der Development-Phase von sechs Tasks auf zwei Tickets angepasst haben.

Zur Evaluation des Praktikums haben wir die Studenten am Ende gebeten, einen Fragebogen auszufüllen. Neben acht offenen Fragen enthielt der Fragebogen Fragen, die mit Werten auf einer Skala von 0 bis 6 zu beantworten waren. Unter anderem baten wir die Studenten um ihre subjektive Einschätzung ihrer Kompetenz vor und nach dem Praktikum. Hier bedeutete 0 "keinerlei Wissen" und 6 "exzellentes Wissen". In allen abgefragten Kompetenzen verbesserten sich

[4] Kanban-Entwickler aus der Wirtschaft berichteten uns im persönlichen Gespräch, dass man bis zu einem Monat warten muss, um zu sehen ob eine Änderungen positive Auswirkungen hatte

die Studenten im Mittel um mindestens 1,1 Punkte. Den größten mittleren Zuwachs gaben die Studenten bei den technischen Fertigkeiten an (Verständnis von Android (+1,7), Prolog (+1,9), Verwendung von Subversion (+2,0) und Eclipse (+2,1)). Aber auch bei "weichen" Faktoren war der Zuwachs deutlich (Kommunikation (+1,5), Zusammenarbeit (+2,0)). Auch im Hinblick auf Kanban äußerten sich die Studenten eindeutig positiv. So hat ihrer Meinung nach das Kanban Board die Übersicht verbessert (5,0) und den Arbeitsfluss klar widergespiegelt (5,1). Zudem waren die Work-in-Progress Limits klar (4,7). Nicht ganz so zufrieden waren die Studierenden mit der eigenen Handhabung von "Staus" (3,0).[5]

Empfehlungen für die Lehre

Die Evaluation ergab, dass der Kanban-Entwicklungsprozess für die Studenten nützlich war. Außerdem, hat der Prozess Probleme verringert, die wir in früheren Praktika erlebt haben. Zum Beispiel hatten wir erlebt, dass blockierende Tickets früher nicht immer allen sichtbar und bewusst waren und zusätzlich von dem Prozess nicht explizit behandelt wurden. Zu diesem Zweck hat, das Kanban-Whiteboard in dem Praktikumsraum eine essentielle Rolle als Mittel für Diskussionen geboten. Dieses hat auch den Studenten geholfen den Gesamtüberblick zu behalten und ihnen die Möglichkeit gegeben selbst ihre eigenen Leistungen zu reflektieren. Wir würden daher, trotz digitalen Möglichkeiten, empfehlen, ein physisches Kanban-Board zu nutzen und es zentral in den täglichen Arbeitsalltag zu stellen.

In Scrum (Schwaber u. Beedle, 2001) gibt es feste Iterationen, die im vorhinein geplant werden. Dem Kunden ist es dabei nicht möglich, den Fokus der Entwicklungs dynamisch während einer Iteration anzupassen. Durch die flexible Input-Queue in unserem Praktikum war es dem Kunden jederzeit möglich den aktuellen Fokus anzupassen. Dies ist in einem Forschungsprojekt wichtig, da auf aktuelle Erkenntnisse und Entwicklungen reagiert werden kann.

Die Fortbildungsmöglichkeiten für Studenten während des Lehrlaufes waren unserer Meinung nach sinnvoll, wurden aber nicht gut genug kommuniziert. Die Arbeit an diesen Aufgaben entsprach auch nicht dem allgemeinen Ticketfluss. Wenn man solche Möglichkeiten vorsieht, sollte man diese Aufgaben auch an einem Ort in der Nähe des Whiteboards visualisieren. Der Vorteil davon ist, dass sowohl die Lehrenden als auch die Teilnehmer wissen an welchen Aufgaben die Studenten arbeiten.

Tägliche "feature celebrations" nach der Mittagspause wurden genutzt um im Team kurz vorher fertiggestellte Tickets in einer Livedemonstration zu feiern. Dies hatte den Zweck, den Studenten bewusst zu machen, was sie bereits erreicht haben. Nachdem die Studenten häufig nicht in allen Phasen bei jedem Ticket involviert waren, erzeugte dieses Zelebrieren häufig auch das Gefühl, dass jeder seinen Teil dazu beigetragen hat und der Code vom gesamten Team stammt. Wir haben diese Treffen auch genutzt, um über die Entwicklung des Tickets selber zu reflektieren: Was lief gut und was hätte im Prozess besser laufen können?

Das Praktikum hat drei Probleme aufgezeigt, denen man sich bewusst sein sollte. Erstens, sollte vor einem Praktikum genug Zeit investiert werden um den Prozess und Kanban zu besprechen und zu diskutieren. Zusätzlich sollten die Studenten die Möglichkeit bekommen, den Prozess selber zu erleben, zum Beispiel in Form eines Spieles.

Zweitens waren die unterschiedliche Größen und Komplexitäten der Tickets ein Problem. In Kanban sollten idealerweise alle Tickets ungefähr gleich groß sein und lange brauchen. Durch gemeinsame Schätzungen der Tickets durch das Team konnten im späteren Teil des Praktikums zu große Tickets identifiziert werden. Zu Anfang des Praktikums fehlen im Normalfall die Erfahrungen zu präzisen Schätzungen. Dozenten sollten in dieser Phase verstärkt dem Team mit ihrer Erfahrung helfen und Wert darauf legen, ihr Wissen über Schätzungen an das Team zu vermitteln.

Drittens war die Anpassung der Arbeitsschritte oder der Limits eine Herausforderung. Anpassungen sollten in kleinen Schritten erfolgen und man sollte einige Zeit abwarten, ob die Änderungen den gewünschten Effekt haben. In unseren Praktika haben wir aber nur eine kurze Zeit zur Verfügung, und die Studenten benötigen unserer Erfahrung nach einige Zeit um sich an die Rahmenbedingungen des Teamentwicklung und der Technologien zu gewöhnen. Wenn man den Prozess einsetzen will, sollte man sich als Lehrender dessen bewusst sein und in der Vorplanung bereits berücksichtigen. Ebenso empfehlen wir in so einer kurzen Zeitspanne entweder sich vorher zu überlegen ob die Arbeitsschritte verändert werden dürfen, oder die Limits in den Arbeitsschritten angepasst werden sollen. Wir hatten uns in dem Praktikum entschieden nur die Limits anzupassen und hatten damit gute Erfahrungen gesammelt.

Praktikum 2012

Aufgrund unserer guten Erfahrungen wurde auch im Praktikum im Jahr 2012 der Kanban-Prozess eingesetzt. Als Basis wurde das hier vorgestellte Kanban-Board genutzt. Dieses musste nicht weiter angepasst werden, so dass die Dozenten eine geringere Vorbereitungszeit benötigten. Im Laufe des Praktikums wurden auch nur wenige und erwartete Anpassungen an den Limits der Prozessschritte vorgenommen. Diese Anpassungen wurden nötig, da die problematische Priorisierung des vorherigen Praktikums entfernt wurde. Weiterhin mit Problemen behaftet waren wiederum

[5] Die Fragen und die Verteilung der Antworten können online nachgelesen werden: `http://sewiki.iai.uni-bonn.de/_media/teaching/labs/xp/2011b/agile_lab_2011_evaluation.pdf`

unterschiedlich komplexe Tickets und ihr Einfluss und Aussagekraft auf den Gesamtfluss. Durch die Arbeitslimits bedingt konnten aber auch schwierige Aufgaben fokussierter bearbeitet werden.

Zusammenfassung

In dieser Arbeit haben wir von unseren Erfahrung mit der Anwendung des Kanban-Entwicklungsprozesses in einem Blockpraktikum an der Universität Bonn berichtet. Neben den Vorteilen des Prozesses und den Lösungen früherer Probleme wurden auch neue Probleme aufgezeigt. In einem kürzlich durchgeführten Praktikum konnten diese teilweise auch schon bewältigt werden.

Als wichtigste Erkenntnis würden wir jedem Lehrenden, der Kanban einsetzen möchte, empfehlen, zu Anfang den bisher eingesetzten Prozess und dessen Schwierigkeiten schriftlich festzuhalten. Auf Basis dessen kann man dann überlegen ob Kanban sinnvoll ist und wie man den Prozess anpassen kann. In unserem Fall konnte Kanban die hartnäckigen Schwierigkeiten aufgeschobener Fertigstellung fast fertiger Stories und ungenügender Reviews aufgrund auf das Iterationsende konzentrierter Fertigstellungen aus der Welt schaffen, sowie den Besprechungsaufwand deutlich reduzieren.

Danksagung

Wir möchten uns bei Matthias Bohlen für seinen externen Review unseres Kanban-Prozesses während des Praktikums bedanken. Wir haben wertvolle Tipps erhalten, die unser Verständnis von Kanban vertieft und unseren Prozess verbessert haben.

Literatur

[Anderson 2010] ANDERSON, D.: *Kanban, Successful Evolutionary Change For Your Technology*. Blue Hole Press, 2010

[Beck u. Andres 2004] BECK, K. ; ANDRES, C.: *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004

[Goldratt u. a. 1984] GOLDRATT, E.M. ; COX, J. ; OUTPUT, C.: *The goal: Excellence in manufacturing*. Bd. 262. North River Press New York, 1984

[Lindvall u. a. 2004] LINDVALL, M. ; MUTHIG, D. ; DAGNINO, A. ; WALLIN, C. ; STUPPERICH, M. ; KIEFER, D. ; MAY, J. ; KAHKONEN, T.: Agile software development in large organizations. In: *Computer* 37 (2004), dec., Nr. 12, S. 26 – 34

[Melnik u. Maurer 2004] MELNIK, G. ; MAURER, F.: Introducing agile methods: three years of experience. In: *Euromicro Conference, 2004. Proceedings. 30th*, 2004, S. 334 – 341

[Mügge u. a. 2004] MÜGGE, H. ; SPEICHER, D ; KNIESEL, G.: Extreme Programming in der Informatik-Lehre - Ein Erfahrungsbericht. In: *Informatik 2004 - Beiträge der 34. Jahrestagung der GI, Lecture Notes in Informatics*, 2004

[Ōno 1988] ŌNO, T.: *Toyota production system: beyond large-scale production*. Productivity Pr, 1988

[Schwaber u. Beedle 2001] SCHWABER, K. ; BEEDLE, M.: *Agile software development with Scrum*. Bd. 18. Prentice Hall, 2001

Session 4

Zur Diskussion gestellt

Muster der Softwaretechnik-Lehre

Valentin Dallmeier · Florian Gross · Clemens Hammacher · Matthias Höschele · Konrad Jamrozik · Kevin Streit · Andreas Zeller

Lehrstuhl für Softwaretechnik, Universität des Saarlandes
{dallmeier, fgross, hammacher, hoeschele, jamrozik, streit, zeller}@cs.uni-saarland.de

1 Wissen der Softwaretechnik-Lehre organisieren

Das Organisieren von Softwaretechnik-Projekten ist für neue Dozenten stets eine Herausforderung: Man muss Kunden finden, Projekte definieren, Tutoren schulen, Anforderungen definieren, Terminpläne machen, Regeln aufstellen... und zum Schluss das Projektergebnis bewerten. Eine ungenügende Planung und Umsetzung kann schnell das gesamte Projekt gefährden. Daher empfiehlt es sich, für die häufigsten Fragen und Probleme Lösungen *explizit* aufzuschreiben. Dies können zunächst immer wiederkehrende Dinge sein, wie die Frage, wann welche Vorlesungsinhalte benötigt werden, oder wie die Begutachtung von Praktikums-Dokumenten organisiert werden muss. Andere Fragen mögen zunächst banal erscheinen, machen aber schnell deutlich, warum es sich empfiehlt, Wissen explizit zu dokumentieren:

- Ich möchte eine Grillfeier für 160 Teilnehmer organisieren. Wieviele Getränke und Speisen muss ich pro Person ansetzen? Woher bekomme ich Grills, Würstchen, Besteck, Salate?
- Ich möchte eine „Messe“ organisieren, auf dem meine Praktikums-Teilnehmer ihre Projekte vorstellen können. Was brauche ich, und woher bekomme ich es? Wann muss ich wen verständigen?

Um das Wissen über Softwaretechnik-Lehre zu verwalten, haben wir ein *Wiki* eingerichtet, um das Wissen über Softwaretechnik-Lehre zu organisieren und zu dokumentieren – und zwar in einer Form, die sowohl *lokales* als auch *standortübergreifendes* Wissen dokumentieren kann. Die Artikel nutzen das Format der *Muster* – analog zu Entwurfsmustern als *Lösungsschablonen für immer wiederkehrende Probleme:*

Name. Jedes Muster hat einen *Namen,* damit es unter diesem Namen einfach referenziert und nachgeschlagen werden kann.

Ziel. Jedes Muster löst ein bestimmtes *Problem,* das hier allgemein charakterisiert werden kann.

Motivation. Das Problem sollte *relevant* sein – also häufig oder in verschiedenen Ausprägungen immer und immer wieder auftreten.

Anwendung. Dies ist eine Beschreibung, wie das Muster umgesetzt wird – sowohl in abstrakter (standortunabhängiger) Form als auch illustriert an konkreten Beispielen. Die Anwendung sollte zeigen, wie das Muster das Problem löst.

Zusammenspiel mit anderen Mustern. Voraussetzungen in Form von anderen Mustern sowie Konflikte mit anderen Mustern sollten ebenfalls dokumentiert werden.

Folgen. Was sind die Folgen dieses Musters? Hier sind insbesondere *unerwartete* und *negative* Folgen gemeint, die dem Leser bewusst sein sollten.

Tipps und Tricks. In diesem Abschnitt können Hilfestellungen zur Umsetzung des Musters, sowie gemachte Erfahrungen dokumentiert werden.

Bekannte Einsätze. Dieser Abschnitt verweist auf die Veranstaltungen, in denen das Muster umgesetzt wurde. Die Wiki-Einträge zu den Veranstaltungen wiederum beschreiben die dort eingesetzten Muster, sowie ggf. Kontaktpersonen und Webseiten mit weiterführenden Informationen.

Abbildung 1 zeigt eins der derzeit gut 50 Muster.

2 Ein Wiki für Erfolgsrezepte

Auf `www.sewiki.de` haben wir unser internes (lokales) Wissen in Form von Mustern zu organisieren. Wir möchten damit unser Wissen der Gemeinschaft bereitstellen, es aber auch jedem Interessierten ermöglichen, sie mit eigenen Erfahrungen weiter zu verfeinern, oder eigene Muster hinzuzufügen. Noch spiegeln diese Seiten unsere eigenen „Kochrezepte“ wieder, die zudem stark lokal geprägt sind. Unsere Vision ist aber, dass wir weitere solche Muster sammeln und weiter ausbauen können – und so das Wissen über erfolgreiche Lösungen der Softwaretechnik-Lehre lebendig halten können. In diesem Sinne rufen wir alle auf, die sich mit der Lehre der Softwaretechnik beschäftigen, ihre eigenen Erfolgsrezepte beizusteuern unter

`http://www.sewiki.de/`

Danksagung. Die Idee, ein Wiki zu nutzen, um Konzepte der Softwaretechnik-Lehre auszutauschen, entstand 2008 in Diskussionen mit Kurt Schneider (Hannover), dem wir zu tiefem Dank verpflichtet sind. Die Muster der Saarbrücker Veranstaltungen gehen auf jahrelange Feinarbeit mit zahlreichen Tutoren zurück, für deren Anregungen wir ebenfalls sehr dankbar sind.

Muster: Spiel als Aufgabenstellung

Ziel

Ziel ist es zum einen, die Zeit zu minimieren, die benötigt wird um sich domänenspezifisches Wissen bezüglich der Aufgabe anzueignen. Zum anderen jedoch, und viel wichtiger, zeigen sich die Studierenden bei Verwendung eines Spiels als Implementierungsziel besonders motiviert. Der Sinn der Aufgabenstellung wird kaum kritisiert.

Motivation

Insbesondere in einem →*Vollzeit-Praktikum*, das innerhalb recht kurzer Zeit stattfindet, ist es wichtig, dass nicht zu viel Zeit aufgewendet werden muss, um die Aufgabenstellung und die sich daraus ergebenden Konsequenzen zu verstehen. [...] Mit der Verwendung von Spielen, insbesondere von Brettspielen, als Implementierungsziel wurde die Erfahrung gemacht, dass die Studierenden innerhalb von einem bis maximal zwei Tagen die Aufgabe klar verstanden haben.
Ein weiterer Vorteil von Spielen ist die gesteigerte Motivation und Lernbereitschaft der Studierenden. Die initiale Hürde, die genommen werden muss, um sich qualifiziert mit der Aufgabenstellung auseinanderzusetzen und darüber diskutieren zu können, ist relativ niedrig. Unterschiede in der Vorbildung der Studierenden kommen wenig zum tragen. Selten werden einzelne Gruppen-Mitglieder abgehängt und außen vorgelassen.
Zu guter Letzt sind massenweise geeignete und interessante Spiele, samt deren Anleitungen, verfügbar. Auf diesen Fundus zurückzugreifen spart das komplette Konzipieren einer Aufgabenstellung und erlaubt es dem Veranstalter, sich auf die Umsetzung zu konzentrieren.

Anwendung

Die Umsetzung besteht im Wesentlichen aus folgenden Schritten:

1. Auswahl eines geeigneten Spiels.
   (a) Die Regeln des Spiels müssen einfach zu verstehen, also nicht zu komplex sein.
   (b) Das Spiel sollte Multiplayer-fähig sein. Somit können die Teilnehmer in einem finalen →*Turnier* gegeneinander antreten.
   (c) Kann das Spiel als →*Client/Server Architektur* umgesetzt werden, kann der Veranstalter einen zentralen Spielserver stellen, der von allen Teilnehmern während der Veranstaltungsdauer genutzt werden kann, um gegeneinander anzutreten und zu testen.
   (d) Es sollte relativ einfach möglich sein, für das gewählte Spiel eine einfache künstliche Intelligenz zu schreiben.
   (e) Es sollte möglich sein ein vernünftiges (graphisches) User-Interface zu implementieren, das die Teilnehmer nutzen können, um ihre Implementierung zu testen und zu debuggen.
   (f) Ist das Spiel geeignet um von Menschen gespielt zu werden, kann der Mensch gegen seine eigene KI antreten, was zusätzlich motiviert. [...]
   (g) Die Spielregeln sollten genug Raum für eine kreative Umsetzung geben. Auch die Implementierung einer Strategie in der KI sollte genug Möglichkeiten der Differenzierung zwischen den Teilnehmern bieten.
   (h) Die Implementierung des Spiels sollte einige algorithmische Möglichkeiten bieten.
   (i) Soll die Aufgabenstellung modifiziert werden (→*Modifikation der Aufgabenstellung*), sollten die Regeln des Spiels dies erlauben.
2. Es sollten keine Implementierungen des Spiels frei verfügbar sein, die die Studierenden einfach kopieren können.
3. Ist die Spielidee rechtlich geschützt, sollten die Rechteinhaber kontaktiert werden, um die Bedingungen der Benutzung zu klären. [...]
4. Erstellen der Aufgabenstellung Ist eine spätere →*Modifikation der Aufgabenstellung* vorgesehen, so kann diese Erweiterte Aufgabenstellung bereits mit erdacht werden.
5. Implementieren der vom Veranstalter vorgegebenen Teile [...]
6. Testbarkeit prüfen [...]
7. Anfertigen einer Referenz-Implementierung der von den Studierenden zu implementierenden Teile
8. Infrastruktur aufsetzen

Für die einzelnen Schritte sollte ausreichend Zeit eingeplant werden. Zum Vergleich: Im →*Software-Praktikum an der Universität des Saarlandes* beginnt der erste Schritt (Auswahl eines geeigneten Spiels) etwa fünf Monate vor Praktikumsbeginn.

Zusammenspiel mit anderen Mustern

Ein Spiel als Aufgabenstellung eignet sich sehr gut für ein abschließendes →*Turnier*.

Folgen

...

Abbildung 1: Muster „Spiel als Aufgabenstellung“ (aus www.sewiki.de, gekürzt)

Transparente Bewertung von Softwaretechnik-Projekten in der Hochschullehre

Oliver Hummel, Karlsruher Institut für Technologie (KIT)

hummel@kit.edu

Zusammenfassung

Die Informatik-Curricula an Hochschulen und Universitäten sehen neben Softwaretechnik-Vorlesungen häufig auch praktische Projekte vor, in denen Studierende die Herausforderungen bei der Erstellung eines nicht-trivialen Softwaresystems in einem Team erfahren sollen. Spätestens seit der Bologna-Reform sind auch solche praktischen Studienleistungen zu benoten, was eine nicht unerhebliche Herausforderung für die Betreuer darstellt, da hochkomplexe und entsprechend unterschiedliche Arbeitsergebnisse transparent und nachvollziehbar bewertet werden müssen. Im vorliegenden Beitrag wird aus Kolloquien, einem Programmiertestat und der direkten Bewertung eingereichter Artefakte ein transparentes und verhältnismäßig einfach anwendbares System zur Bewertung von Softwareentwicklungsprojekten an Hochschulen und Universitäten entwickelt und von seiner praktischen Erprobung an der Universität Mannheim berichtet.

Einleitung

Teambasierte Softwareentwicklungsprojekte sind seit Jahren ein fester Bestandteil des Curriculums von Informatikstudiengängen an vielen Hochschulen. Der vorliegende Beitrag zielt primär auf praktische Entwicklungsprojekte im Fach Softwaretechnik (bzw. engl. Software Engineering) an Hochschulen und Universitäten, kann aber, mit entsprechenden Abwandlungen, auch für andere Fachgebiete der Informatik von Nutzen sein. Aus technischer Sicht geht es in entsprechenden Lehrveranstaltungen hauptsächlich um das Erlernen von strukturierten Entwurfs- und Modellierungstechniken für Software sowie deren Umsetzung in lauffähige und getestete Programme. Dazu notwendig ist das Erlernen grundlegender Techniken, Methoden und Werkzeuge zur Entwicklung komplexer Softwaresystemen (vgl. ACM, 2004 oder Ludewig, 1999), so dass Studierende wichtige Kompetenzen zur konstruktiven Mitarbeit in entsprechenden Industrieprojekten erwerben können. Anders ausgedrückt sollen Absolventen möglichst direkt in der Industrie „beschäftigungsfähig" sein (vgl. KMK, 2010 oder auch Coldewey, 2009). Primäres Lernziel der in diesem Artikel diskutierten Veranstaltungen an der Universität Mannheim war die Vermittlung eines strukturierten und artefaktgetriebenen („ingenieurmäßigen") Vorgehens in der Softwareentwicklung und die Einübung dazu erforderlicher Modellierungs- und Implementierungstechniken (UML, Continuous Integration u.ä., vgl. Larman, 2004).

Bekanntlich existiert für die Entwicklung von Softwaresystemen in verschiedensten Domänen kein allgemein gültiges Patentrezept („No Silver Bullet", vgl. Brooks, 1995), so dass erlernte Verfahren von den Ausführenden immer mit Augenmaß an die jeweiligen Umstände angepasst werden müssen (sog. „Tailoring" vgl. z.B. Rausch et al., 2008). Folglich gibt es auch in praktischen Projekten an Universitäten oder Hochschulen nicht eine korrekte Lösung, sondern vielmehr eine Reihe von möglichen und sinnvollen Lösungsstrategien. Darüber hinaus zielen entsprechende Teamprojekte darauf ab, auch sogenannte Softskills, wie Team-, Präsentations- und Kommunikationsfähigkeit der Teilnehmer zu schulen (Böttcher & Thurner, 2011), weshalb sie zumeist in Kleingruppen von etwa drei bis sechs Personen durchgeführt werden. Gleichzeitig sehen einschlägige Prüfungsordnungen aber regelmäßig eine individuelle Benotung aller Teilnehmer vor. Weiter erschwerend kommt bei Softwaretechnik-Projekten hinzu, dass es praktisch unmöglich ist, die Studierenden beständig unter Aufsicht arbeiten zu lassen, wodurch ihre individuellen Beiträge zur Gruppenarbeit nicht ohne weiteres einzuordnen und ggf. auch Täuschungsversuche nur schwer zu entdecken sind (vgl. z.B. Kehrer et al., 2005). Eine objektive Bewertung der studentischen Leistungen ist unter diesen Gegebenheiten offensichtlich nicht trivial (vgl. z.B. Hayes et al., 2007), aber nichtsdestotrotz kritisch, da sie maßgeblich über das Fortkommen der Studierenden entscheidet und letztlich auch ihre Moti-

vation stark beeinflusst. Dabei schränken zahlreiche äußere Zwänge, wie Zeitvorgaben oder Budgetrestriktionen die Möglichkeiten zur Bewertung ein, so dass es nicht verwunderlich ist, wenn in der Lehrpraxis für große Kohorten häufig auf „Ersatzprüfungsverfahren" wie Kolloquien, Lerntagebücher oder gar Klausuren zurückgegriffen wird (vgl. Kehrer et al., 2005), um praktische Studienleistungen zu bewerten. Je nach Ausgestaltung dieser Bewertungsarten ergibt dies einen nicht unerheblichen Mehraufwand für alle Beteiligten, der auf Seiten der Studierenden zudem leicht zu einem Motivationsverlust bei der Entwicklungsarbeit führen kann, da diese überhaupt nicht oder nur in sehr begrenztem Umfang in die Benotung einfließt. Soweit letzteres überhaupt der Fall ist, existiert in der Literatur (und nach Kenntnisstand des Autors auch in der Praxis) bisher selten ein differenziertes und transparentes Bewertungssystem, wie es in diesem Beitrag vorgestellt wird.

Eigene Erfahrungen des Autors, der im Verlauf seines Studiums zwei Mal selbst an einem entsprechenden Projekt teilgenommen und einmal als studentische Hilfskraft an der Betreuung mitgewirkt hat, waren ähnlicher Art: In den ersten beiden Fällen war das Zustandekommen der Note für die Teilnehmer vollkommen intransparent (sie setzte sich „irgendwie" aus der Leistung im Projekt und dem Abschlusskolloquium zusammen); im letztgenannten Fall haben gar die studentischen Hilfskräfte eigene Maßstäbe zur Bewertung von Anforderungs- und Designdokumenten, die sich hauptsächlich auf die Zählung von offensichtlichen Fehlern beschränkten, entwickelt und eingesetzt.

Ziel dieses Beitrags ist es daher, ein nachvollziehbares und dennoch einfach handhabbares Bewertungsverfahren für praktische Softwareentwicklungsprojekte, das eine individuelle und transparente Bewertung der Studierenden ermöglicht, vorzustellen und von Erfahrungen aus seiner praktischen Erprobung an der Universität Mannheim zu berichten. Dazu werden im folgenden Abschnitt zunächst wichtige didaktische und organisatorische Rahmenbedingungen vorgestellt, bevor in aller gebotenen Kürze einige benötigte Hintergründe der Softwaretechnik dargelegt werden. Danach werden gängige Bewertungsverfahren auf ihre Verwendbarkeit in einem Softwaretechnik-Projekt analysiert. In den darauf folgenden Kapiteln werden das in diesem Beitrag thematisierte Bewertungsverfahren im Detail vorgestellt und aus ersten Erprobungen gewonnene Erkenntnisse diskutiert. Ein Vergleich mit anderen in der Literatur veröffentlichten Verfahren und eine Zusammenfassung runden diesen Beitrag schließlich ab.

Grundlagen

Um die wichtigsten Herausforderungen eines Softwaretechnik-Projekts aus verschiedenen Blickwinkeln zu beleuchten, sollen im Folgenden weiterführende Einblicke gegeben werden, die nochmals die Notwendigkeit eines an der Aufgabenstellung orientierten Bewertungssystems unterstreichen sowie wichtige Hintergründe näher erläutern. Anschließend werden zentrale Anforderungen an ein solches Bewertungssystem zusammengetragen.

Didaktische Sicht

Aus didaktischer Sicht springen zunächst die hohen Anforderungen, die sich an die Studierenden in einem Softwaretechnik-Projekt auf Grund der zuvor formulierten Lern- und Kompetenzziele ergeben, ins Auge. Beispielsweise Böttcher et al. (2011) halten eine detailliertere Aufschlüsselung dazu notwendiger Kompetenzanforderungen aus den vier Bereichen Sach-, Methoden-, Selbst- und Sozialkompetenzen bereit. Grundsätzlich bleibt festzuhalten, dass die hohe Komplexität von Softwareprojekten von den Teilnehmern ohne Zweifel ein tiefgehendes Verständnis zahlreicher Abläufe und Techniken verlangt und Prüfer damit regelmäßig vor die Herausforderung stellt, etwas zu prüfen, was sich nicht direkt lehren lässt (vgl. Ludewig, 2011), sondern nur durch praktische Anwendung erfahren werden kann.

In der Softwaretechnik wurde (und wird) für eine solche Lehrform traditionell häufig der Begriff Praktikum verwendet, der damit allerdings nicht korrekt eingesetzt ist. Ein Praktikum an einer Hochschule oder Universität ist zwar ebenfalls eine praktische Unterrichtsform, in dieser setzen sich Studierende allerdings mit vorgegebenen und klar umrissenen Aufgaben, die innerhalb weniger Stunden zu bearbeiten sind, auseinander (vgl. Iller & Wick, 2009). Da diese Lehrform hauptsächlich in klassischen Naturwissenschaften (wie Chemie oder Biologie) Verwendung findet, bedeutet das zumeist, dass Aufgaben nur vor Ort in einem entsprechenden Labor bearbeitet werden können. Als „Laborumgebung" in der Informatik und speziell in der Softwaretechnik ist heute allerdings in der Regel ein handelsüblicher Laptop ausreichend, mit dem bequem an beinahe jedem beliebigen Ort gearbeitet werden kann. Auch bei den Aufgabenstellungen in der Softwaretechnik, die zumeist das selbstständige Durchlaufen des kompletten Entwicklungszyklus einer Software vorsehen, passt obige Begriffsdefinition nicht, vielmehr ist hier aus Sicht der Didaktik von einem praktischen Projekt mit entsprechend höheren Anforderungen an die Eigenverantwortung der Teilnehmer auszugehen, weshalb im Folgenden dieser Begriff Verwendung finden soll.

Studentische Perspektive

Aus studentischer Sicht erfahren Softwareentwicklungsprojekte häufig eine komplett konträre Einschätzung. Zum einen gibt es die leistungsstarken „Vollblut-Informatiker", die gerne, gut und viel programmieren und oft privat bereits eigene kleine Softwareprojekte vorangetrieben oder gar als Praktikant Industrieerfahrung gesammelt haben. Für diese ist ein solches Projekt nicht selten einer der Höhepunkte ihres Studiums, den sie mit großer Motivation und Begeisterung angehen und oft mit beeindruckenden Leistungen abschließen, gerade wenn die Aufgabenstellung eventuell in Kooperation mit einem Industriepartner erfolgt ist. Zum anderen gibt es aber – gerade in interdisziplinären Studiengängen wie der Wirtschaftsinformatik – auch viele Studierende, die Softwareentwicklung und insbesondere Programmieren nur als notwendiges Übel ansehen und daher entsprechende Projekte nur als eine mit einem hohem Zeitaufwand verbundene Pflichtübung betrachten.

Dies führt in der Lehrpraxis häufig dazu, dass studentische Teams sehr unterschiedliche Leistungsniveaus aufweisen und schwächere Studierende sich sehr leicht „abgehängt" fühlen, während die stärkeren diese vor allem als Belastung wahrnehmen (vgl. auch die Erfahrungen von Lindig & Zeller, 2005). Somit überrascht es nicht, wenn in solchen Veranstaltungen häufig auf das Phänomen von „Trittbrettfahrern" zu treffen ist, die sich ohne nennenswerte eigene Beiträge von ihren Kommilitonen durch die Veranstaltung schleppen lassen wollen (vgl. Stoyan & Glinz, 2005, van der Duim et al., 2007 oder Stangl, 2002). Die Leistungsträger erdulden das aus falsch verstandener Solidarität heraus häufig zumindest so lange, wie es die eigene Arbeitskraft nicht überlastet, sobald aber ihre Belastungsgrenze erreicht wird, sind entsprechende Konflikte, bis hin zu auseinanderbrechenden Teams, vorprogrammiert.

Organisatorische Sicht

Die im Folgenden beschriebenen Softwaretechnik-Lehrveranstaltungen an der Universität Mannheim waren nach einem verbreiteten Muster organisiert: ein Professor trug als Prüfer die Hauptverantwortung für die Veranstaltung und vermittelte in Vorlesungen die theoretischen Hintergründe. Wissenschaftliche Mitarbeiter übernahmen die Aufgabe den Studierenden den Lehrstoff an Hand von einzelnen Übungsaufgaben mit direktem Projektbezug auch praktisch näher zu bringen. Die wissenschaftlichen Mitarbeiter waren ferner für die Steuerung des eigentlichen Projekts ebenso zuständig wie für die Klärung von offenen Fragen (z.B. bzgl. der Anforderungen). Die direkte Betreuung der Entwicklungsteams oblag auf Grund der recht hohen Teilnehmerzahlen von anfangs teilweise deutlich über fünfzig Studierenden einer Reihe von studentischen Tutoren. Diese hatten in der Regel bei mindestens wöchentlichen Treffen der Teams direkten Kontakt mit den Studierenden und trafen sich ebenso regelmäßig mit den Mitarbeitern, um über die Fortschritte ihrer Teams zu berichten. Von anderen Universitäten sind ähnliche Konstellationen mit teilweise noch weit höheren Teilnehmerzahlen bekannt (z.B. Kehrer et al., 2005). An Hochschulen sind Kohorten hingegen oft um einiges kleiner, so dass dort auch Projekte möglich sind, in denen Professoren die Studierenden persönlich betreuen und sich einen direkten Eindruck über ihre Leistungen verschaffen können.

Eine weitere Schwierigkeit bei der Koordination und auch Bewertung von Projekten stellt die meist geringe Erfahrung der beteiligten Mitarbeiter und Tutoren dar. Bedingt durch die naturgemäß sehr hohe Fluktuation innerhalb dieser Personengruppen kommt es häufig vor, dass Mitarbeiter oder Tutoren, die eine entsprechende Veranstaltung noch vor wenigen Monaten selbst besucht haben, sich in einer betreuenden Rolle wieder finden, das Wissen aus vorangegangenen Durchläufen aber nicht schriftlich dokumentiert worden ist. Sobald neue Tutoren bzw. Mitarbeiter entsprechende Erfahrungen angesammelt haben, schließen sie ihr Studium oder ihre Promotion ab und wenden sich neuen Aufgaben zu. Dieser Erfahrungsverlust gilt natürlich auch für den Bereich der Bewertung studentischer Abgaben, so dass ein klar definiertes Bewertungsschema fraglos allen Lehrenden die Betreuung erleichtern und dadurch zu einer deutlichen Erhöhung der Lehrqualität führen kann.

Anforderungen an die Benotung eines Softwareprojekts

An dieser Stelle soll zunächst kurz auf die allgemeinen Güteanforderungen an Prüfungen eingegangen werden. Die Literatur (beispielsweise Roloff, 2002 oder Müller & Bayer, 2007) hält dazu die folgenden Testgütekriterien bereit: eine Prüfung bzw. auch eine einzelne Prüfungsleistung sei *objektiv*, also unabhängig, sowohl von der Person des Prüfers, als auch von der des Prüflings bewertbar. Insbesondere gilt dabei, dass beispielsweise Aussehen, oder Herkunft eines Prüflings die Note nicht beeinflussen dürfen. Ferner soll ein Prüfungsverfahren *zuverlässig* die tatsächliche Leistungsfähigkeit eines Prüflings erfassen und somit bei vergleichbaren Prüfungsleistungen eine vergleichbare Benotung ergeben. Eine *valide* Prüfungsform misst genau die Leistung, die sie vorgibt zu messen und genau die, die auch gemessen werden soll. *Nützlich* ist eine Prüfung, sobald sie eine sinnvolle Funktion erfüllt, im Allgemeinen werden in diesem Zusam-

menhang beispielsweise eine Rückmeldefunktion für Dozenten bzw. Studierende oder auch Anreiz- bzw. Selektierungsfunktionen genannt. Eine Prüfung gilt als *ökonomisch*, wenn Nutzen und Aufwand in einem vernünftigen Verhältnis zueinander stehen.

Aus den oben diskutierten Hintergründen und Erfahrungen sowie den eben genannten Testgütekriterien ergeben sich folgende grundlegende Anforderungen an die Benotung einer praktischen Softwaretechnik-Veranstaltung, die weitestgehend auf ähnliche Veranstaltungen in anderen Teilgebieten der praktischen Informatik übertragbar sind:

1. Die Benotung muss sowohl für Lehrende als auch für Studierende nachvollziehbar sein und sollte unabhängig von der Person des Bewerters bzw. des Bewerteten zum gleichen Ergebnis kommen. Auf Grund des relativ großen Aufwands, den Studierende in entsprechende Veranstaltungen investieren, werden idealerweise auch die erstellten Artefakte (ggf. in mehreren Teilschritten) benotet; generell erfasst das Bewertungsschema die in den Lernzielen formulierten Fertigkeiten.
2. Gemäß der Bologna-Kriterien (KMK, 2010) muss die Benotung für jeden Studierenden individuell erfolgen, es kann daher bei Teamarbeiten nicht automatisch eine einheitliche Note für alle Teammitglieder vergeben werden. Ein geeignetes Bewertungsverfahren muss die Zuordnung individueller Leistungen erlauben, dabei aber gleichzeitig Teamarbeit zulassen und fördern bzw. im Idealfall sogar in der Note abbilden. Dabei sollte auf Grund entsprechender Erfahrungen die Möglichkeit bestehen, „Trittbrettfahrer" frühzeitig zu erkennen, um die übrigen Mitglieder eines Teams vor einer Überlastung und deren negativen Folgen für die eigenen Noten zu schützen.

Softwaretechnik-Hintergründe

Zum besseren Verständnis des später erklärten Bewertungsschemas sollen im Folgenden in aller Kürze einige Aspekte der Softwaretechnik und daraus abgeleitete Lernziele hervorgehoben werden. Ein zentraler Punkt ist dabei die Definition einer Software selbst. Laut Brooks (1995) umfasst „Software" nicht nur den ablauffähigen Programmcode, den ein Endbenutzer einsetzt, sondern auch die zu seiner Erstellung notwendigen Anforderungen, daraus abgeleitete Analyse- und Entwurfsmodelle sowie Testfälle und weitere Dokumentation (wie z.B. auch das Benutzerhandbuch). Es ist daher bereits in mittelgroßen Projekten notwendig, eine ganze Reihe von Mitarbeitern mit verschiedenen Rollen miteinander zu koordinieren, um ein System mit möglichst wenigen Reibungsverlusten entwickeln zu können.

Dazu stehen in der Softwaretechnik verschiedene Vorgehensmodelle zur Verfügung (vgl. Bunse & v. Knethen, 2008), von denen das Wasserfallmodell (s. z.B. Sommerville, 2010 oder Larman & Basili, 2003) nach wie vor das bekannteste ist. Modernere, modellbasierte Ansätze, wie das aus dem Unified Process entstandene Agile Modelling (vgl. Larman, 2004), arbeiten zumeist iterativ und schlagen die Erstellung verschiedener, voneinander abgeleiteter (UML-)Modelle für die Systementwicklung vor. Das Erlernen einer solchen modellbasierten und möglichst systematischen Vorgehensweise war das primäre Lernziel der vom Autor an der Universität Mannheim angebotenen Softwaretechnik-Projekte. Der Bewertung der von den Studierenden erstellten Artefakten kommt daher im Folgenden eine zentrale Bedeutung zu.

Mögliche Bewertungsverfahren

Dieses Kapitel geht auf kurz gängige Bewertungsverfahren an Schulen bzw. Hochschulen ein und diskutiert, ebenfalls in aller Kürze, mögliche positive und negative Aspekte bezüglich der Anwendbarkeit innerhalb von praktischen Softwaretechnik-Lehrveranstaltungen. Dazu fasst die folgende Tabelle neben den Bewertungsverfahren auch die jeweils wichtigsten positiven und negativen Aspekte zusammen:

Bewertungsverfahren	**Kurzbeschreibung**	**Stärken**	**Gefahren**
Prakt. Leistungsnachweise (Übungsaufgaben)	Die Studierenden bearbeiten vorgegebene Übungsaufgaben	Kleinere Aufgaben zu einer spez. Fragestellung mit meist überschaubarem Arbeitsaufwand	Meist kein unmittelbarer Bezug zum Projekt; erhöhte Gefahr des Abschreibens gleicher Aufgaben
Vorträge	Die Studierenden präsentieren ihr Vorgehen oder	Stärkung der Präsentationskompetenz; Vertiefung	Evtl. überdeckt der Vortragsstil inhaltliche Stärken

	ihre Ergebnisse	des Fachwissens im entspr. Themenbereich; ggf. Selbstreflektion durch Vergleich mit anderen Studierenden	oder Schwächen; hoher Zeitaufwand; mögl. Spezialisierung auf das Themengebiet
Kolloquien	Die Studierenden werden von den Betreuern zu ihrem System befragt (oft in Verbindung mit Vorträgen durchgef.)	Nachhaken und Klären von Verständnisfragen möglich	Bei großen Gruppen entsprechend hoher Zeitaufwand
Bewertung des Systems	Das abgegebene System und seine Artefakte bzw. auch Zwischenabgaben (Meilensteine) werden bewertet	Direkter Bezug zum Lernziel und damit direktes Feedback zum Lernerfolg	Kriterienkatalog notwendig, hoher Zeitaufwand in der Bewertung, Zuordnung der Leistung nicht immer einfach
Mündliche Prüfung(en) (Testate)	Prüfungsgespräch mit einem oder mehreren Studierenden	Planbarer Zeitaufwand für den Prüfer, etablierte Prüfungsform, individuelle Benotung und Nachfragen möglich	Bewertung spiegelt nicht unbedingt den Beitrag zum Projekt wieder, hoher Zeitaufwand, verwendete Fragen verbreiten sich schnell
Schriftliche Prüfung	Die klassische Klausur	Bekanntes Verfahren, Leistung ist klar zuzuordnen und gut vergleichbar	Erfordert mitunter einen hohen Vorbereitungsaufwand; kein direkter Bezug zur Leistung im Projekt
Programmierprüfung	„Programmierklausur“, idealerweise mit einer Entwicklungsumgebung direkt am Rechner ausgeführt	Gute Überprüfung der Lernziele in Bezug auf Programmierkenntnisse, am Rechner über Testfälle automatisch und damit objektiv bewertbar	Ausreichend große Rechnerräume müssen verfügbar sein; bei autom. Bewertung kein Bewertungsspielraum, minimale Programmierfehler können große Auswirkungen haben
Laborbuch	Studierende dokumentieren Aufgabenverteilung und Vorgehen regelmäßig	Erleichtert Nachvollziehbarkeit des Vorgehens	Overhead für Studierende und Betreuer; Benotung schwierig; Gefahr unreflektierter Aufzählungen groß
Reflektion	Studierende reflektieren über ihre Erfahrungen	Bei korrekter Ausführung relativ hoher Lerneffekt	Weiterer Overhead für die Studierenden bei ohnehin schon starker Belastung; dedizierte Benotung ebenfalls schwierig
Mitarbeitsnoten	Mitarbeit der Studierenden in ihrer Gruppe wird bewertet	Motiviert zu regelmäßiger Mitarbeit, geringer Aufwand	Generell nur schwer objektiv einschätzbar, insbesondere bei Heimarbeit
Benotung der Studierenden untereinander (Peer Assessment)	Studierende geben sich gegenseitig Noten oder verteilen eine begrenzte Anzahl von Punkten an ihre Teamkollegen	Geringer Aufwand für den Betreuer; Studierende reflektieren und lernen durch Bewertung ihrer Kommilitonen	Validität & Legalität sehr fraglich, mögl. negative Auswirkungen auf die Gruppendynamik

Tabelle 1: Mögliche Bewertungsverfahren für ein Softwaretechnik-Projekt.

Alle aufgeführten Bewertungsansätze sind im Kontext eines Softwareprojekts sicherlich sinnvoll anwendbar. Wie beispielsweise von Hayes et al. (2007) vorgeschlagen, erscheint in der Praxis aber

eine Kombination verschiedener Verfahren erforderlich, um alle zuvor genannten Anforderungen an die Benotung abdecken zu können und dabei möglichst wenig Mehraufwand für Studierende und Betreuer zu generieren. So eignet sich beispielsweise die Bewertung der abgegebenen Systemartefakte allein sehr gut als Feedback über die im Projekt erbrachte Leistung, stößt aber an Grenzen, sobald diese in Heimarbeit erstellt werden können und sollte daher durch Präsenzverfahren (wie Kolloquien o.ä.) ergänzt werden. Bewertungsansätze mit hoher Eigenverantwortung (wie Reflektionen oder Lerntagebücher) erzielen bei entsprechend motivierten Studierenden meist gute Ergebnisse, sind jedoch in einem ohnehin stressigen Projekt offensichtlich mit zusätzlichem Mehraufwand verbunden und ferner in Zweifelsfällen nur schwer als nicht ausreichend zu benoten. Sie erfordern daher, wie die meisten anderen Verfahren, ein klares und idealerweise im Voraus kommuniziertes Bewertungsschema, was den Aufwand für die Betreuenden weiter erhöht, insbesondere, wenn die Zusammenstellung der Inhalte individuell gestaltbar ist.

Bewertungsschema

Aus den zuvor angestellten Überlegungen lässt sich das folgende Bewertungsschema ableiten, das sich im Wesentlichen aus fünf einzelnen Bewertungsverfahren zusammensetzt:

1. Einfache Übungsaufgaben mit direktem Projektbezug
2. Ein praktisches Programmiertestat (gegen vorgegebene Testfälle am Rechner)
3. Führen von Stundenzetteln („Timesheets“)
4. Bewertung der abgegebenen Modellierungsartefakte, Quellcodes und Binaries
5. Mehrere Kolloquien

Der zeitliche Ablauf einer entsprechenden organisierten Veranstaltung gestaltet sich folgendermaßen: zeitnah zu Semesterbeginn werden den Studierenden, grobe Anforderungen an das zu erstellende System mitgeteilt und sie erhalten erste Aufgaben bzgl. der Erfassung und Modellierung der Anforderungen mit direktem Projektbezug. Diese sind einzeln oder in Zweiergruppen zu bearbeiten und abzugeben. Somit ist gewährleistet, dass jeder Teilnehmer sich individuell einen grundlegenden Überblick über die Anforderungen an das zu erstellende System erarbeiten muss. Idealerweise werden die Lösungen der Studierenden kontrolliert oder zumindest in einem Tutorium besprochen, aber nicht benotet. Gleichzeitig werden erste (einfache) Programmieraufgaben gestellt, die auch dazu dienen, die Studierenden mit der verwendeten Entwicklungsumgebung (also z.B. Eclipse, s. z.B. Burnette & Staudemeyer, 2009) vertraut zu machen, falls dies nicht bereits in früheren Veranstaltungen geschehen ist.

Programmiertestat & Kolloquien

Nach etwa drei bis vier Wochen wird ein (ebenfalls einfaches) Programmiertestat mit großzügig bemessener Bearbeitungszeit als erster harter Leistungsnachweis zum Einstieg in das Projekt verlangt. Es bietet sich an, dieses als Programmierübung direkt am Rechner (mit Eclipse und den Javadocs, aber ohne Internet-Zugang) durchzuführen und den Studierenden Interfaces und JUnit-Tests vorzugeben, gegen die sie entwickeln müssen (vgl. testgetriebene Entwicklung nach Beck, 1999). Somit wird Beschwerden, dass Programmieren in Papierklausuren weitaus schwieriger als in einer Entwicklungsumgebung (mit „Code-Completion“) sei, von vorne herein der Wind aus den Segeln genommen. Gleichzeitig entfällt das äußerst zeitaufwändige Bewerten von zumeist unübersichtlichem Programmcode für die Betreuer, da nur das automatisierbare Ausführen der JUnit-Tests notwendig wird. Idealerweise werden mehrere in etwa gleichschwere Teilaufgaben gestellt, so dass die Programme der Studierenden nicht alle Testfälle erfüllen müssen, sondern beispielsweise nur drei aus fünf, o.ä. Auf Grund von in Praxis und Literatur bekannten extremen Leistungsunterschieden bei Softwareentwicklern (vgl. Endres & Rombach, 2003), sollte den Studierenden ausreichend Zeit zum Lösen dieser Aufgaben gegeben werden. In der Mannheimer Praxis hat sich etwa die zehnfache (!) Zeit, die wissenschaftliche Mitarbeiter zum „Proberechnen“ benötigen, bewährt (z.B. ca. 18 zu 180 Minuten).

Die Ergebnisse dieses Testats können im Anschluss zur Einteilung der eigentlichen Projektgruppen genutzt werden. Es bietet sich an, die Studierenden gemäß ihrer Leistungen (also z.B. der Zeit, die bis zur Erfüllung der Testkriterien benötigt wurde) zusammen zu gruppieren, um zu große Leistungsgefälle innerhalb der Teams und eine daraus resultierende Demotivation aller Teilnehmer zu vermeiden (vgl. z.B. Stangl, 2002). Gruppen mit einer großen Leistungsheterogenität haben sich in der Erfahrung des Autors nicht bewährt, da sie das Auftreten von „Trittbrettfahrern“ zu fördern scheint. Wer zu diesem Zeitpunkt nicht über minimale Programmierkenntnisse verfügt (also das Testat nicht bestanden hat), muss vom weiteren Verlauf des Projekts ausgeschlossen werden, um die übrigen Studierenden vor einer Überlastung zu schützen. Dies bringt natürlich eine gleichzeitige Reduktion der verlangten Features für den Rest der Gruppe mit sich. Eine Wiederholungsmöglichkeit für das Testat würde den weiteren Ablauf des Projekts

offensichtlich stark beeinträchtigen und ist daher nicht vorgesehen, zumal fehlende Programmierkenntnisse ohnehin nicht binnen weniger Tage aufzuholen sind. Auf Grund der harten Konsequenzen dieses Kriteriums, liegt es aber nahe, das Testat bereits zu oder gar vor Beginn des Semesters als Freiversuch anzubieten, um den Studierenden ein frühes Feedback zu geben und ihnen ggf. eine mehrwöchige Auffrischung ihrer Vorkenntnisse zu ermöglichen.

Nach erfolgter Einteilung der Projektteams sind diese gehalten, sich vertieft in die Projektanforderungen einzuarbeiten und diese entsprechend zu analysieren und zu dokumentieren (durch Use Cases, System-Sequenzdiagramme oder Operation Contracts). Um die abschließende Benotung zu erleichtern, sind alle abzugebenden Artefakte mit dem Namen des Bearbeiters zu versehen und unter dem eigenen Benutzernamen in das verwendete Versionskontrollsystem hochzuladen. Ferner sollten alle Aktivitäten wie das Erstellen von Dokumenten, Modellen oder Quellcode, aber auch Team-Meetings o.ä. in sogenannten Timesheets gelistet werden, ähnlich, wie es auch zu Abrechnungszwecken in Industrieprojekten praktiziert wird.

Nach etwa weiteren zwei Wochen wird ein erstes Kolloquium durchgeführt, um sicherzustellen, dass jedes Teammitglied zum Projektziel beiträgt und alle Anforderungen vollständig verstanden wurden. Um den Studierenden etwaige Berührungsängste zu nehmen, bietet es sich an, dieses Kolloquium nicht zu benoten, sondern nur eine rudimentäre Mindestleistung zum Bestehen zu verlangen, die ggf. in einem Folgegespräch nochmals nachgefordert werden kann. Der Autor hat gute Erfahrungen damit gemacht, bis zum Ende des Projekts noch zwei weitere, benotete Kolloquien durchzuführen, eines nach etwa zwei Dritteln der Dauer, eines in Form einer Systempräsentation bzw. -abnahme ganz am Ende des Projekts.

Systembewertung

Die Benotung des Softwaresystems und der mitgelieferten Artefakte erfolgt auf Basis des im Folgenden vorgestellten Bewertungsrasters. Dabei sind **fett** gedruckte Elemente zwingend zum Bestehen erforderlich (d.h. eine nicht ausreichende Leistung in einem entsprechenden Bereich führt direkt zum Nicht-Bestehen des gesamten Projekts). Ohne Auszeichnung gedruckte Kategorien werden vom Team verantwortet und für das gesamte Team einheitlich benotet, für *kursiv* gedruckte ist jeder Teilnehmer individuell verantwortlich und bekommt eine eigene Bewertung. Eine entsprechende Kennzeichnung der Dokumente ist dafür wie bereits beschrieben obligatorisch.

Im Sinne der agilen Softwareentwicklung (Pichler, 2007 oder Beck, 1999) sollen alle Teilnehmer indviduell Verantwortung für vollständige Features (also meist komplette Use Cases, vgl. Larman, 2004) übernehmen, so dass die Anforderungen vergleichbar bleiben und eine technologieorientierte Aufteilung (z.B. ein Teammitglied programmiert, eines entwickelt die UI, eines testet und eines erstellt nur Diagramme oder Anforderungen) vermieden wird. Details zu den im Folgenden verwendeten Modellen finden sich bei Bedarf bei Larman (2004). Je nach genauer Ausrichtung der Veranstaltung ist natürlich auch eine geänderte Zusammenstellung mit anderen Artefakten (wie z.B. Projektplänen o.ä.) oder veränderten Gewichtungen denkbar:

Kontextmodellierung

1. **Domänenmodell * 2**
2. Geschäftsprozessmodelle * 1
   (bei Wirtschaftsinformatik-Bezug)

Anforderungsmodellierung

3. ***Use Cases* *5**
4. Use Case Diagram *1

Analysemodelle

5. ***System-Sequenzdiagramme* *3**
6. *Contracts der Systemoperationen *3*

Entwurfsmodelle

7. **Schichtenarchitektur und Architekturdiagramm * 2**
8. ***Interaktionsdiagramme (GRASP und Design Patterns)* *5**
9. **Klassendiagramme *3**
10. *UI Storyboard *1*

Spezifikationsdokument

11. Lesbarkeit und Aufbau *1
12. Konsistenz der enthaltenen Modelle *2

Source Code

13. **Ausführbarkeit (lauffähiges JAR-File) *1**
14. *Korrektheit *2*
15. Einhaltung von Coding Guidelines *2
16. Verwendung von Javadoc und Kommentierung *1
17. Konsistenz mit der Spezifikation *2

Testfälle

19. ***Überdeckung* *4**

Benutzerschnittstelle

20. *Ergonomie/Bedienbarkeit *3*
21. ***Reaktivität* *1**

Insgesamt besteht das vorgestellte Bewertungsschema aus zwanzig Einzelleistungen n_1 bis n_{20}, die alle auf einer vierstufigen Skala mit 0 bis 3 Punkten wie folgt bewertet werden:

- 0 Punkte (und ungenügend) werden vergeben, wenn ein Element komplett fehlt oder grob fehlerhaft bzw. unvollständig ist
- 1 Punkt gibt es für Kategorien, die sinnvolle Inhalte haben, die aber noch deutlich fehlerhaft bzw. lückenhaft sind
- 2 Punkte gibt es für weitgehend vollständige und überwiegend korrekte Elemente
- 3 Punkte werden für vollständige, korrekte und sehr sorgfältig erarbeitete Artefakte vergeben

Der Multiplikator am Ende jeder Bewertungskategorie gibt die vorgesehene Gewichtung w_i innerhalb des gesamten Rasters an, so dass eine Gesamtpunktzahl leicht nach folgender Formel berechnet werden kann:

$$g = \sum_{i=1}^{20} n_i \cdot w_i$$

In der vorgestellten Form werden teambasierte Bewertungsanteile 18-fach gewichtet, individuelle Bewertungsanteile 27-fach, so dass sowohl die Teamarbeit gefördert, als auch individuelle Anstrengungen berücksichtigt werden und somit die geforderte individuelle Benotung sichergestellt ist.

Zusammenfassung des Schemas

Da auch bei Verwendung des gerade vorgestellten Bewertungsschemas Täuschungsversuche der Studierenden durch Abschreiben erfahrungsgemäß nicht ausgeschlossen werden können, empfiehlt es sich, das Bestehen des Projekts bzw. seine Gesamtnote nicht alleine von der Artefakt-Bewertung abhängig zu machen. Die vom Autor eingesetzte Kombination mit anderen bereits angesprochenen und vorgestellten Bewertungsverfahren gestaltet sich im Detail wie folgt:

- Übungsaufgaben mit direktem Projektbezug *(unbenotet, nur abzugeben)*
- Programmiertestat am Rechner *(unbenotet, aber zu bestehen)*
- 1. Kolloquium *(unbenotet, nur zu bestehen)*
- 2. Kolloquium *(15% der Endnote)*
- Abschlusspräsentation und -kolloquium *(25% der Endnote)*
- Bewertung des Systems, wie oben gezeigt *(60% der Endnote)*
- Stundenzettel (*unbenotet, glaubhaft und nachvollziehbar geführt abzugeben*)

Durch dieses Bewertungsmodell wird gewährleistet, dass jeder Teilnehmer zumindest über grundlegende Programmierkenntnisse verfügt, die ihn in die Lage versetzen, konstruktiv zur Arbeit seines Teams beizutragen. Durch die Kolloquien wird ferner das Verständnis der geleisteten Arbeit überprüft und eine soziale Kontrollinstanz eingefügt, wodurch letztlich mit großer Wahrscheinlichkeit davon ausgegangen werden kann, dass die abgelieferten und benoteten Artefakte auch von den entsprechenden Studierenden erstellt worden sind. Natürlich sollte auch die Bewertung Kolloquien auf bekannten Richtlinien und Bewertungskriterien, wie z.B. aus Schubert-Henning (2004), aufbauen.

Ein früheres Feedback über die Leistungen der Teilnehmer sollte zur Vervollständigung des Eindrucks ohnehin über regelmäßige Gespräche mit den Tutoren erhoben werden, so dass bei Verdacht auf mangelhafte Leistungen bereits im Projektverlauf eingegriffen werden kann. Auch eine Bewertung einzelner Artefakte aus für den Projektverlauf definierten Meilensteinen ist möglich und gibt den Teilnehmern eine frühe Rückmeldung über ihren Leistungsstand.

Erfahrungsbericht & Diskussion

Unter der Verantwortung des Autors wurde das vorgestellte, kombinierte Bewertungssystem in drei Softwaretechnik-Lehrveranstaltungen (zwei Mal in einer einsemestrigen Bachelorveranstaltung mit je ca. 50 Teilnehmern in Vierergruppen und einmal in einem zweisemestrigen Masterprojekt mit vier Teilnehmern) erfolgreich eingesetzt und aufbauend auf vorherigen Erfahrungen immer wieder leicht verändert. Insbesondere hat sich gezeigt, dass die Verwendung der vollen Notenskala (also 1,0; 1,3; … 4,0 bzw. 5,0) für die Systembewertung nicht zielführend ist, da die entsprechend feinen Unterschiede zwischen den Notenstufen zu zeitaufwändig in der Ermittlung und im Nachhinein praktisch nicht mehr nachvollziehbar sind. Dies gilt insbesondere wenn mehrere Prüfer in den Bewertungsprozess involviert sein sollten. Basierend auf der recht hohen Zahl von ca. 20 Einzelbewertungen hat es sich als ausreichend erwiesen, nur die zuvor genannten vier Bewertungsstufen zu verwenden. Bei deren Verwendung ergeben sich für obiges Raster mit seinen 45 Gewichtungsfaktoren maximal 135 erreichbare Punkte, die mit Hilfe gängiger (teilweise sogar in Prüfungsordnungen festgelegten) Prozentskalen problemlos in entsprechende-Noten umgerechnet werden können.

Ein berechtigter Einwand gegen das vorgeschlagene Bewertungsraster ist sicher sein hoher Detaillierungsgrad, der den Studierenden deutliche Hinweise auf die verlangte Lösung liefern kann (also z.B. welche Modelle werden erwartet?). Sofern die sinnvolle Wahl von Modellen (d.h. das „Tailoring“ der Vorgehensweise und die Auswahl der erstellten Artefakte) ein Lernziel der Veranstaltung dar-

stellt, ist das Bewertungssystem in seiner vorgestellten Form sicherlich nicht anwendbar. Zu diesem Einwand gibt es allerdings zwei wichtige Erwiderungen: zum einen die Feststellung, dass Anfänger im Software Engineering (die erstmals an einer entsprechenden Veranstaltung teilnehmen) häufig noch genug mit dem Erlernen und Einsetzen der vorgestellten Notationen und Techniken gefordert sind und entsprechend mit deren Auswahl noch weit überfordert wären. Konkretere Vorgaben werden somit ohnehin notwendig, um die Verunsicherung der Studierenden zu reduzieren und auch Tutoren und Mitarbeitern eine Orientierung zur Beantwortung entsprechender Rückfragen zu geben. Das vorgestellte Bewertungsraster transportiert somit die Erwartungen des Prüfers an die Studierenden und ermöglicht ihnen, ihre Arbeitsschwerpunkte entsprechend der gegebenen Gewichtungen zu setzen.

Für fortgeschrittenere Studierende, die bereits ein Anfängerprojekt durchlaufen haben, gibt das Raster in der gezeigten Form andererseits tatsächlich zu viele Hinweise preis. Es besteht natürlich die Möglichkeit, beispielsweise nur fünf Hauptkategorien (wie Kontextmodellierung, Anforderungen, Analyse, Entwurf und Sourcecode) und ihre Gewichtungen zu nennen, nicht aber die konkret darin erwarteten Modelle. Ferner könnte noch eine zusätzliche Bewertungskategorie zur Modellauswahl etabliert werden, die die Sinnhaftigkeit der gewählten Modelle bewertet. Die Gewichtung innerhalb dieser Kategorie kann beispielsweise gleichmäßig auf die gelieferten Modelle verteilt werden oder auch von den Studierenden vorgeschlagen werden, wenn sie mit einem entsprechenden Bewertungsschema bereits in einer Grundlagenveranstaltung vertraut gemacht worden sind. Ferner liegt es für länger laufende Projekte ohne feste Funktionalitätsvorgabe nahe, auch einen entsprechend hoch gewichteten Bewertungsfaktor für die Menge der umgesetzten Funktionalität (also für die Produktivität) einzuführen. Weiterhin besteht natürlich die Möglichkeit, auch die Einhaltung des gewählten Vorgehensmodells, die Erstellung und Befolgung eines Projektplans oder auch Softskills, wie z.B. die Teamfähigkeit zu bewerten bzw. letztere evtl. durch Peer Assessments (Race et al., 2005) bewerten zu lassen. Insgesamt bietet das vorgeschlagene Bewertungsraster genügend Flexibilität, um mit wenig Aufwand an die Lernziele vieler Softwaretechnik-Lehrveranstaltungen angepasst werden zu können.

Betrachtung der Testgütekriterien

Die in diesem Beitrag vorgestellte Kombination von Bewertungsverfahren erreicht eine gute Überdeckung der eingangs genannten Testgütekriterien. Da die Bewertung des vorgeschlagenen Programmiertestats automatisiert über die zu durchlaufenden Testfälle erfolgen kann, ist es ein sehr objektives Verfahren. Es misst zudem genau die Leistung, die auch gemessen werden soll, nämlich die Fähigkeit, ein gegebenes Problem innerhalb eines beschränkten Zeitraums in ein passendes Programm umzusetzen. Es gibt sowohl Studierenden als auch Lehrenden ein hilfreiches Feedback bei der Einschätzung der Programmierkenntnisse und ist zu dem noch günstig (da automatisiert) durchführbar. Auch eine gewisse Skalierbarkeit ist gegeben, solange genügend gleichzeitig verfügbare Arbeitsplätze in einem oder mehreren Rechnerpools zur Verfügung stehen.

Da es sich bei Kolloquien im Wesentlichen um Präsentationen und Prüfungsgespräche handelt, ist die Bewertung naturgemäß zu einem gewissen Grad subjektiv (s. z.B. Müller & Bayer, 2007). Ferner erfassen Kolloquien nicht notwendigerweise die tatsächliche Leistungsfähigkeit eines Prüflings, und noch weniger garantieren sie, seine Fähigkeiten in der Softwareentwicklung zu messen, da auch Präsentations- und Gesprächskompetenz eine nicht zu unterschätzende Rolle spielen. Der praktische Nutzen ist nichtsdestotrotz für beide Parteien hoch, da sich alle Beteiligten einen Eindruck über den Wissensstand und das Verständnis eines Studierenden machen und im Zweifelsfall direkt nachfragen können. Für kleine Gruppen sind solche Gespräche auch ein durchaus ökonomisches Prüfungsverfahren, sobald allerdings viele Studierende geprüft werden müssen, steigt die Belastung für den Prüfer sehr stark an.

Das vorgestellte Raster zur Systembewertung ermöglicht eine hohe Objektivität, zumindest dann, wenn, wie oben beschrieben, nicht die volle Notenskala zur Bewertung der Teilleistungen verwendet wird. Die Unterscheidung zwischen beispielsweise einer 1,7 und einer 2,0 für ein Klassendiagramm ist nach Erfahrung des Autors objektiv oft nicht mehr möglich. Mit der vereinfachten, vierstufigen Benotungsvariante hingegen ist das Verfahren sehr zuverlässig und valide, da es wiederholbar genau die Leistung eines Studierenden oder eines Teams bei der Erstellung des Systems bewertet, die bewertet werden soll. Somit ergibt sich auch ein hoher Nutzen für Prüfer und Studierende, da beide ein gutes Feedback über den tatsächlichen Leistungsstand der Studierenden erhalten. Einzig signifikante Schwäche des Verfahrens ist der recht hohe Bewertungsaufwand, der sich bei einem einsemestrigen Projekt mit ca. vier Studierenden pro Team schnell auf vier und mehr Stunden pro Team belaufen kann. Legt man ca. fünf ECTS und damit insgesamt rund 600 Stunden Arbeitslast für eine Viergruppe von Studierenden zugrunde, relativiert

sich diese Zahl allerdings wieder; wie auch im Vergleich zu einer Klausur, die oft einen noch höheren Korrekturaufwand benötigt. Zudem lässt sich durch klare Vorgaben für die geforderten Abgaben (z.B. direkt ausführbares System, Quellcode mit klarer Struktur in einem Versionierungssystem, Dokumente mit einer gewissen Struktur und klare Benennung der Urheber) signifikant Zeit bei der Bewertung einsparen.

Verwandte Arbeiten

In der Literatur ist zwar eine erkleckliche Menge von Berichten über Softwareentwicklungsprojekte an Hochschulen verfügbar, die dort gemachten Aussagen über das verwendete Bewertungsverfahren beschränken sich aber meist auf wenige Sätze. Systematische Vergleiche verschiedener Ansätze finden sich daher leider ebenso wenig, wie auch empirische Studien zur Zufriedenheit der Beteiligten mit den folgenden Bewertungsansätzen bisher nicht verfügbar sind.

Kerer et al. (2005) verwendeten für ihr Entwicklungsprojekt mit ca. 600 Studierenden eine Kombination aus kleineren Übungsaufgaben (sog. Assignments) und einer Klausur mit projektbezogenen Fragen und Programmieraufgaben. Interessant ist bei dieser Veröffentlichung speziell die offenbar hohe, aber nicht näher spezifizierte Zahl von aufgedeckten Plagiatsfällen, sowohl bei den Assignments, als auch bei den abzugebenden Programmen, die nochmals die Notwendigkeit einer individuellen Bewertung unterstreicht.

Ludewig (2011) hingegen kritisiert das bloße Abfragen von Fakten (z.B. mit Hilfe einer Klausur) zur Bewertung einer praktischen Lehrveranstaltung als nicht zielführend und benennt die, aus seiner Sicht recht einfache Bewertung der Gruppenleistung (also des erstellten Systems) als Alternative. Kriterien für deren Bewertung werden aber leider nicht genannt. Von der Prüfungsordnung verlangte individuelle Noten wurden in Ludewigs Veranstaltungen „für Teilnehmer, die [positiv oder negativ] auffielen, durch ein angemessenes Delta“ aus der Gruppennote abgeleitet. Dieses Verfahren wurde nach Kenntnisstand des Autors erstmals von Forbrig (1997) vorgestellt und beispielsweise auch von Stoyan & Glinz (2005) angewendet. Lindig und Zeller (2005) gehen in ihrem Bericht über ein sechswöchiges Vollzeitprojekt ebenfalls nicht näher auf die Benotungskriterien ein, es werden nur Kolloquien und automatisierte Systemtests als Prüfkriterien genannt.

Ein wenig detaillierter berichtet Metzger (2003), der in seinem Projekt zu 50% die Individualleistung und zu 50% die Gruppenleistung bewertet hat. „Zur Gruppenleistung tragen die Abgaben zu den einzelnen Aufgaben und die Abschlusspräsentation bei. In die Individualbewertung fließt die Leistung während der Präsenzzeiten und die Qualität des Einzelvortrags ein.“ Konkrete Bewertungskriterien für die Leistungen der Studierenden werden allerdings wiederum nicht genannt. Predoiu (2010) hat jüngst ein Konzept vorgestellt, in dem Artefakt-Bewertungen durch einen Betreuer mit Peer Assessments der Studierenden kombiniert werden, um eine individuelle Benotung zu ermöglichen. Dazu werden zu Beginn eines Projekts Artefakte von Seiten des Betreuers den Meilensteinen zugeordnet: jeder Studierende muss pro Meilenstein mindestens ein Artefakt bearbeiten und wird zeitnah nach der Abgabe in einer Assessmentsitzung benotet. Vorteile dieses Vorgehens sind sicher die kontinuierliche Bewertung der Teilnehmer und der enge Kontakt zwischen Studierenden und Betreuer; letzterer kann sich aber z.B. hinsichtlich Skalierbarkeit bei vielen Studierenden auch als nachteilig erweisen. Erprobt wurde das System in einer Kleingruppe mit weniger als zehn Teilnehmern. Zudem gibt es in Predoius Vorschlag keine teambasierten Benotungsanteile, so dass allein die individuelle Leistung zählt und nicht die notwendigen Beiträge für das Gesamtsystem. Ungeklärt bleibt auch die Effektivität eines Peer Assessments zur Erkennung bzw. Benennung von Trittbrettfahrern.

Nach dem Kenntnisstand des Autors am nächsten an das in diesem Beitrag vorgestellte Bewertungsraster kommt der Vorschlag von Wikstrand und Börstler (2006) heran. In ihrem Bewertungsansatz erwirbt ein Team gemäß seiner Leistung eine Anzahl von Credits für verschiedenste Aufgaben im Projekt, dazu gehören die Erstellung von Artefakten und Projektplänen ebenso wie gehaltene Präsentationen. Am Ende des Projekts sind die Teammitglieder gefordert, diese Credits eigenständig untereinander zu verteilen, so dass sich gegebenenfalls individuelle Abstufungen in der Benotung ergeben. Für die Betreuer besteht bei offensichtlichen Ungerechtigkeiten eine Eingriffsmöglichkeit. Eine effektive Erkennung von Studierenden, die auf Grund mangelnder Vorkenntnisse nicht hinreichend zum Fortschritt des Teams beitragen konnten, gibt es bei diesem Vorschlag allerdings wiederum nicht, es wird allein auf gegenseitige soziale Kontrolle durch die Studierenden selbst gesetzt.

Verschiedene Empfehlungen zur Verminderung dieses Problems, wie auch zur organisatorischen Durchführung von Softwaretechnik-Projekten insgesamt, geben beispielsweise van der Duim et al. (2007). Sie schlagen vor „Free Riders“, also Trittbrettfahrer, dadurch zu erkennen, dass die Studierende Stundenzettel zu führen und einen vordefinierten Prozess zu verwenden haben; ferner sind wöchentliche Feedback-Gespräche mit den Betreu-

ern vorgesehen, konkrete Nachweise der Programmierfähigkeiten werden aber wiederum nicht verlangt.

ZUSAMMENFASSUNG

Der vorliegende Beitrag beschreibt eine neuartige Zusammenstellung von Bewertungsverfahren zur Benotung von Softwareentwicklungsprojekten an Hochschulen und Universitäten. Durch die Kombination eines automatisierten Programmiertestats mit mehreren Kolloquien und einem gewichteten Bewertungsraster für im Verlauf des Projekts erstellte Artefakte ermöglicht es die Komposition von teambasierten und individuellen Benotungsanteilen sowie eine effektive Erkennung von sogenannten Trittbrettfahrern. Somit wird im Vergleich zu bisherigen Bewertungsansätzen eine wesentlich objektivere Bewertung möglich, zudem wird die Nachvollziehbarkeit der Benotung durch die Studierenden deutlich verbessert.

Das vorgestellte Bewertungsraster für in der Softwareentwicklung erstellte Artefakte ist sicherlich die Kerninnovation dieses Beitrags, die zuvor nur in der Arbeit von Wikstrand und Börstler (2006) in annähernd ähnlicher Form vorgeschlagen wurde. Es bewertet im Gegensatz zu zahlreichen anderen Verfahren wie Klausuren, Übungsaufgaben oder auch Kolloquien direkt die Ergebnisse der Projektarbeit und damit die Erreichung der Lernziele und versucht nicht, eine Benotung über „Umwege“ sicherzustellen. Somit wird die Motivation und Konzentration der Studierenden auf die wesentlichen Ziele des Projekts gerichtet, und sie werden in einem ohnehin schon komplexen Lernumfeld nicht zusätzlich mit weiteren Aufgaben wie der Vorbereitung auf eine Klausur belastet. Durch die Bewertung teambasierter und individueller Leistungsanteile wird sowohl die Zusammenarbeit im Team gefördert, als auch den Ansprüchen von Prüfungsordnungen und Akkreditierungsagenturen nach einer individuellen Benotung genüge getan. Ferner wird durch die Kombination mit einem automatisiert auswertbaren Programmiertestat und zwei Kolloquien sichergestellt, dass Trittbrettfahrer frühzeitig entdeckt werden und die Leistungen ihrer Teamkollegen nicht nachhaltig negativ beeinflussen können. Insgesamt ist die Bewertung der abgelieferten Systemartefakte allerdings mit einem recht hohen Bewertungsaufwand verbunden (nach bisherigen Erfahrungen ca. 4-5 h pro Team), dieser liegt allerdings in der Regel unter einem Prozent des auf Seiten der Studierenden geleisteten Gesamtaufwands und ist somit der Komplexität der Aufgabenstellung angemessen und auch mit dem Aufwand beim Einsatz einer Klausur vergleichbar. Ein weiterer Vorteil ist die hohe Flexibilität des vorgestellten Bewertungsrasters, da abweichende Schwerpunkte in der Lehre durch veränderte Gewichtungen oder eine veränderte Zusammensetzung relativ leicht abgebildet werden können. So sind beispielsweise Aspekte des Projektmanagements, wie ein Projekt- oder Iterationsplan und dessen Einhaltung, in fortgeschrittenen Lehrveranstaltungen problemlos bewertbar. Insgesamt erscheint das vorgestellte Bewertungsschema damit so flexibel, dass es prinzipiell auch für Projekte in anderen Fachgebieten der Informatik anwendbar sein sollte.

Bis dato wurde das Verfahren in jeweils leicht abgewandelter Form in drei Projekten an der Universität Mannheim mit insgesamt gut 100 Studierenden erprobt. Die Benotung und im Extremfall auch ein Durchfallen lassen von Studierenden waren insbesondere durch die angewendete vierstufige Systembewertung gut begründbar und letztlich auch für die Studierenden nachvollziehbar, so dass sich Detaildiskussionen über die Bewertung subjektiv deutlich reduzieren ließen (entsprechende Parameter wurden leider nicht systematisch erfasst). Neben kleineren Verbesserungen, wie einer weiteren Verfeinerung der Abgaberichtlinien oder eventuellen Anpassungen der Gewichtungsfaktoren, soll bei zukünftigen Projekten auch eine systematische Befragung der Projektteilnehmer durchgeführt werden, um Stärken und Schwächen des Verfahrens aus Sicht der Studierenden zu erkennen und gegebenenfalls identifizierte Probleme zu beheben.

DANKSAGUNG

Der Autor möchte Melanie Klinger vom Referat für Hochschuldidaktik sowie Colin Atkinson, Werner Janjic und Marcus Schumacher von der Software-Engineering-Gruppe der Universität Mannheim herzlich für zahlreiche konstruktive Diskussionen und Anregungen im Zusammenhang mit dem vorgestellten Bewertungsschema danken.

Literatur

ACM (2004): Software Engineering Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, in Computing Curricula Series.

Beck, K. (1999): Extreme Programming Explained: Embrace Change. Addison-Wesley.

Böttcher, A. & Thurner, V. (2011): Kompetenorientierte Lehre im Software Engineering, Proceedings des Workshops für Software Engineering im Unterricht der Hochschulen.

Brooks, F. (1995): The Mythical Man-Month. Essays on Software Engineering (2. Aufl.), Addison-Wesley.

Bunse, C. & von Knethen, A. (2008): Vorgehensmodelle kompakt (2. Aufl.), Spektrum Akademischer Verlag.

Burnette, E. & Staudemeyer, J. (2009): Eclipse IDE – kurz & gut (2. Aufl.), O'Reilly.

Coldewey, J. (2009): Schlechte Noten für die Informatik-Ausbildung, in OBJEKTspektrum, Ausgabe 05.

Endres, A. & Rombach, D. (2003): A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories, Addison-Wesley.

Forbrig, P. (1997): Probleme der Themenwahl und der Bewertung bei der Projektarbeit in der Software-Engineering-Ausbildung, Proceedings des Workshops für Software Engineering im Unterricht der Hochschulen.

Hayes, J.H., Lethbridge, T.C. & Port, D. (2007): Evaluating Individual Contribution toward Group Software Engineering Projects, Proceedings of the International Conference on Software Engineering, IEEE.

Iller, C. & Wick, A. (2009): Prüfungen als Evaluation der Kompetenzentwicklung im Studium, Das Hochschulwesen, Vol. 6.

Kerer, C.; Reif, G.; Gschwind, T., Kirda, E.; Kurmanowytsch, R. & Paralic, M.: ShareMe (2005): Running a distributed systems lab for 600 students with three faculty members, IEEE Transactions on Education, Vol. 48, No. 3.

Kultusministerkonferenz: Ländergemeinsame Strukturvorgaben für die Akkreditierung von Bachelor und Masterstudiengängen (2010), verfügbar unter: http://www.kmk.org/fileadmin/veroeffentlichungen_beschluesse/2003/2003_10_10-Laendergemeinsame-Strukturvorgaben.pdf, letzter Zugriff: Juli 2012.

Larman, C. & Basili, V. (2003): Iterative and incremental developments: a brief history, IEEE Computer, Vol. 36, No. 6.

Larman, C. (2004): Applying UML and Pattern (3. ed.), Prentice Hall.

Lindig, C. & Zeller, A. (2005): Ein Softwaretechnik-Praktikum als Sommerkurs, Proceedings des Workshops für Software Engineering im Unterricht der Hochschulen.

Ludewig, J. (1999): Softwaretechnik in Stuttgart – ein konstruktiver Informatik-Studiengang, Softwaretechnik-Spektrum Vol. 22, No. 1.

Ludewig, J. (2011): Erfahrungen bei der Lehre des Software Engineering, Ludewig und Böttcher (Hrsg.): Software Engineering im Unterricht der Hochschulen, CEUR-WS.org.

Metzger, A. (2003): Konzeption und Analyse eines Softwarepraktikums im Grundstudium, Siedersleben & Weber-Wulff (Hrsg.): Software Engin. im Unterricht der Hochschulen, dpunkt.

Müller F. & Bayer C. (2007): Prüfungen: Vorbereitung – Durchführung - Bewertung. In Kröning: Förderung von Kompetenzen in der Hochschullehre, Asanger.

Pichler, R. (2007): Scrum Agiles Projektmanagement erfolgreich einsetzen, dpunkt.

Predoiu, L. (2010): Didaktik und Bewertung in längerfristigen Teamprojekten in der Hochschullehre. Proceedings des 6. Workshops der GI-Fachgruppe "Didaktik der Informatik".

Race, P.; Brown, S. & Smith, B. (2005): 500 Tips on Assessment, Routledge.

Rausch, A.; Hohn, R.; Broy, M.; Bergner, K. & Höppner, S. (2008): Das V-Modell XT: Grundlagen, Methodik und Anwendungen, Springer.

Roloff, S. (2002): Hochschuldidaktisches Seminar: Schriftliche Prüfungen, verfügbar unter: http://www.lehrbeauftragte.net/documents_public/SchriftlPruef_Roloff.pdf, letzter Zugriff: Juli 2012.

Schubert-Henning (2004), S.: Empfehlungen zur Vorbereitung und zum Vortrag eines Referates, Studienwerkstatt der Universität Bremen, verfügbar unter: http://www.philosophie.uni-bremen.de/fileadmin/mediapool/philosophie/Formulare-Informationen/empfehlungen_referate.pdf, letzter Zugriff: Juli 2012.

Sommerville, I. (2010): Software Engineering (9th ed.), Addison-Wesley.

Stangl, W. (2002): Lernen in Gruppen, Werner Stangls Arbeitsblätter, verfügbar unter: http://www.stangl-taller.at /ARBEITSBLAETTER/LERNEN/Gruppenlernen.shtml, letzter Zugriff: Juli 2012.

Stoyan, R. & Glinz, M. (2005): Methoden und Techniken zum Erreichen didaktischer Ziele in Software-Engineering-Praktika, Löhr & Lichter (Hrsg.): Software Engineering im Unterricht der Hochschulen, dpunkt.

van der Duim, L.; Andersson, J. & Sinnema, M. (2007): Good practices for Educational Software Engineering, Proceedings of the International Conference on Software Engineering, IEEE.

Wikstrand, G. & Börstler, J. (2006): Success Factors for Team Project Courses. Proceedings of International Conference on Software Engineering Education and Training.

Forschung mit Master-Studierenden im Software Engineering

Marc Hesenius, Universität Duisburg-Essen
marc.hesenius@paluno.uni-due.de
Dominikus Herzberg, Hochschule Heilbronn
dominikus.herzberg@hs-heilbronn.de

Zusammenfassung

Master-Studierende bringen oft viel Potential mit an die Hochschule. Im Vergleich zu Bachelor-Studierenden haben sie in der Regel völlig andere Sicht- und Herangehensweisen entwickelt, zum Teil aufgrund des höheren Alters, aber maßgeblich wegen der Möglichkeit, bereits einige Jahre außerhalb der Hochschule gearbeitet zu haben. Es stellt sich die Frage, wie dieses Potential für die Forschung an Hochschulen genutzt werden kann. Im Folgenden beschreiben wir unsere Erfahrungen mit einer Grupper Master-Studierender im ersten Semester, denen wir die Aufgabe gestellt haben, aus einem Pool von Forschungsthemen eines auszuwählen und eine Arbeit in Form eines Artikels zu schreiben. Das erklärte Ziel war, publizierbare Ergebnisse zu generieren. Unsere Erwartungen wurden wesentlich übertroffen.

Einleitung

Die Beschäftigung mit aktueller Forschung ist in vielen Studiengängen durchaus Gegenstand der Lehre, jedoch oft nur abgekoppelt von anderen Veranstaltungen im Rahmen von Seminararbeiten und Literaturstudien. Die Einheit von Forschung und Lehre wird vielerorts nicht praktiziert (Euler, 2005). Software Engineering bietet aber weitreichende Fragestellungen, die eine Kombination von Forschung und Lehre interessant machen (Broy u. Pree, 2003). Wie wir versucht haben, das Potential, welches Master-Studierende mit an die Hochschule bringen, für die Forschung zu nutzen, möchten wir im Folgenden anhand unserer Erfahrungen mit einer Gruppe von 19 Studierenden im ersten Semester des Master-Studienganges *Software Engineering and Management* an der Hochschule Heilbronn beschreiben. Bei der Gruppe handelt es sich um internationale Studierende, entsprechend wurden alle Veranstaltungen auf Englisch durchgeführt; des Weiteren kennzeichnet die Gruppe ein breiter Bildungshintergrund mit dem Fokus auf Informatik und IT sowie teils mehrjährige Berufserfahrung.

Im Rahmen der Veranstaltungen *Software Architecture* (SWA) und *Paradigms in Software Development* (PSD) bekamen die Studierenden die Aufgabe, aus einer Zahl von sieben Forschungsthemen eines auszuwählen und dazu eine Arbeit in Artikelform zu verfassen sowie eine Präsentation zu halten. Die Veranstaltungen haben in Summe fünf ECTS bzw. vier SWS und wurden vom verantwortlichen Professor sowie einem Assistenten gemeinsam über die gesamte Zeit begleitet. Die Studierenden hatten die Möglichkeit, in Gruppen zu arbeiten, hiervon machten aber nur wenige Gebrauch.

Dieser Erfahrungsbericht soll unsere Ideen und Vorgehensweise zusammenfassen und zur Wiederholung animieren. Die Ergebnisse sind für Universitäten und Fachhochschulen gleichermaßen interessant. Sie zeigen, dass bei einem Betreuungsaufwand von vier SWS wissenschaftlich publizierbares Material entstehen kann. Doktoranden an Universitäten können so ihre Forschungsarbeiten vorantreiben und in 4-6 Monaten von Studierenden Forschungsaufgaben effizient erarbeiten lassen. An Fachhochschulen, wo der wissenschaftliche Mittelbau fehlt, kann so Freiraum für Forschung geschaffen und gefördert werden.

Themenauswahl

Die Themen orientierten sich am Fokus der Veranstaltungen und waren so gewählt, dass die Studierenden Ergebnisse in der verfügbaren Zeit erreichen konnten. Diese waren (in Klammern die Anzahl der Arbeiten):

(a) Relation of SW Architecture and Organizational Structure (5)

(b) Port Shen to Clojure (1)

(c) Reinvent Programming for iPads/Tabs (4)

(d) Architecture Diagrams Supporting the Functional Paradigm (4)

(e) Program Structure and Run-Time Visualization of Algorithms (1)

(f) Architecture and Scale-Free Networks (0)

(g) Concurrency and Architecture (1)

Die Studierenden waren nicht an diese Auswahl gebunden und konnten ihre Arbeiten feiner auf bestimmte Aspekte ausrichten. Mit den Themen (a) und (b) wurden zwei *Notausgänge* integriert, falls Studierende ihre Stärke oder Schwäche im Programmieren sahen. Thema (a) hatte zwar konkreten Bezug auf Software Architektur, im Fokus lag aber die Beziehung zu Unternehmensstrukturen. Thema (b) war eine anspruchsvolle Programmieraufgabe, sodass hier keine Ausarbeitung geschrieben werden musste. Die Themen (a) und (e) waren bewusst empirisch orientiert und wurden auch entsprechend vorgestellt.

Unsere Erwartung war, aus den eingereichten Arbeiten eine nach eventueller Überarbeitung publizieren zu können, sowie einige andere zu einer weiteren Veröffentlichung zu verbinden.

Struktur und Durchführung

Die Struktur des gesamten Semesters war grob in drei Phasen unterteilt: Einführung (die zeitlich längste Phase), Forschung, Schreiben.

Die Einführungsphase startete mit einem Multiple-Choice-Test über 32 Fragen, die eine breite Themenspanne des Software Engineerings abdeckten. Ziel war, einen Überblick über den Leistungsstand zu bekommen sowie Studierende zu identifizieren, die sich bei der Themenwahl potentiell vergreifen konnten. Es folgten einige Vorlesungseinheiten zu SWA und PSD sowie Einführungen in wissenschaftliches Schreiben, Zitieren sowie die Wertigkeit und korrekte Angabe von Quellen. In diesem Zusammenhang wurde auch das Thema Plagiate behandelt. Des Weiteren bekamen die Studierenden einen Leitfaden, der den groben Aufbau einer wissenschaftlichen Publikation beschrieb und gleichzeit als verpflichtende LaTeX-Vorlage diente sowie eine Leseliste zur Vorbereitung. Im Folgenden wurden die Themen präsentiert und zur Wahl gestellt. Die Studierenden wurden mit kleinen Aufgaben langsam an ihre gewählten Projekte herangeführt: Zuerst sollten Sie ein einseitiges Proposal schreiben, in dem Sie den Kontext sowie das Problem zusammenfassten und die angedachte Lösung präsentierten. Anschließend sollten Sie die wissenschaftliche Community analysieren und wichtige Konferenzen, Personen und Literatur identifizieren. Ihnen war im Zuge dieser Arbeiten einmalig erlaubt, das Thema zu wechseln.

Die folgenden zwei Phasen waren von der Umsetzung der Arbeit sowie der Verschriftung der Ergebnisse geprägt. Aufgrund der vier SWS konnten wir wöchentliche Gespräche mit den Studierenden führen, sie somit kontinuierlich betreuen und viel Feedback geben bzw. bekommen. Die Abgabe war frühzeitig als fixe, nicht verhandelbare Deadline definiert worden.

Ergebnisse und Ausblick

Wir haben einen Versuch beschrieben, Master-Studierende im ersten Semester unter Anleitung eine Forschungsarbeit durchführen zu lassen. Die Ergebnisse haben unsere Erwartungen übertroffen. Aus 16 Arbeiten waren fünf auf sehr gutem Niveau und für eine Publikation geeignet. Alle auf Konferenzen eingereichte Arbeiten wurden vor Einreichung überarbeitet.

Zwei Arbeiten zu den Themen (d) und (g) waren hochwertig, jedoch noch nicht ganz vollständig. Leider entschieden sich die Studierenden, die Themen nicht weiter zu verfolgen. Eine sehr gute empirische Arbeit zum Thema (e) stellt einen Versuch mit Bachelor-Studierenden im ersten Jahr vor, der von zwei Master-Studierenden vorbereitet, durchgeführt und analysiert wurde. Aktuell warten wir auf die Ergebnisse eines Peer-Reviews dieses Artikels. Verschiedene Arbeiten zum Thema (a) haben interessante Zusammenhänge zwischen Architektur und Unternehmensstruktur aufgezeigt, sind jedoch nur schwach empirisch belegbar. Eine diese Arbeiten wurde bei der VARSA 2012[1] eingereicht und vom Peer-Review nicht angenommen, da der Artikel nicht ganz den Fokus der Konferenz traf. Aus den Arbeiten zum Thema (c) stach eine besonders hervor und wurde nach Anreicherung durch weitere Ideen auf der SC 2012[2] durch die Reviewer angenommen und im Juni 2012 unter (Hesenius u. a., 2012) veröffentlicht.

Die Nichtbestehensquote lag bei ca. 36%. Eine auffällige Korrelation mit dem Einführungstest besteht – aus guten Ergebnissen folgten auch gute Arbeiten. Natürlich gibt es erfreuliche wie unerfreuliche statistische Ausreißer: So verhoben sich zwei durchaus begabte Studierende mit guten Ergebnissen im Einführungstest an der Programmieraufgabe (b).

Für die Zukunft ist zu prüfen, wie mehr Lehrinhalte eingewoben werden können, etwa durch Kombination mit dedizierten Veranstaltungen zum wissenschaftlichem Schreiben.

Literatur

[Broy u. Pree 2003] BROY, Manfred ; PREE, Wolfgang: Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme. In: *Informatik-Spektrum* 26 (2003), Januar, Nr. 1, S. 3–7. – ISSN 0170–6012, 1432–122X

[Euler 2005] EULER, Dieter: Forschendes Lernen. In: *W. Wunderlich & S. Spoun (Hrsg.), Universität und Persönlichkeitsentwicklung. Frankfurt, New York: Campus* (2005)

[Hesenius u. a. 2012] HESENIUS, Marc ; OROZCO MEDINA, Carlos D. ; HERZBERG, Dominikus: Touching factor: software development on tablets. In: *Proceedings of the 11th international conference on Software Composition*. Berlin, Heidelberg : Springer-Verlag, 2012. – ISBN 978–3–642–30563–4, S. 148–161

[1] http://sites.google.com/site/varsa2012/
[2] http://wg24.ifip.org/SC2012/

Welche Kompetenzen benötigt ein Software Ingenieur?

Yvonne Sedelmaier, Sascha Claren, Dieter Landes, Hochschule für angewandte Wissenschaften Coburg

{sedelmaier, sascha.claren, landes}@hs-coburg.de

Zusammenfassung

Software ist Teil der modernen Welt und nahezu überall zu finden. Daher gewinnt Software Engineering zunehmend an Bedeutung. Um Software zu entwickeln oder weiterzuentwickeln, sind vielfältige Kompetenzen und Kenntnisse nötig. Das stellt auch die Hochschulausbildung und damit verbunden die didaktische Forschung im Software Engineering vor große Herausforderungen.

In diesem Beitrag wird zunächst das Forschungsinteresse beschrieben und begründet. Dann wird der Kompetenzbegriff diskutiert und erläutert. Nachfolgend werden die methodischen Schritte beschrieben, mit denen das Kompetenzprofil eines Software Ingenieurs erhoben wird. Dabei wird zunächst die Vorgehensweise beschrieben, bevor erste Ergebnisse vorgestellt werden. Zuletzt folgt ein Ausblick, wie die Hochschulausbildung im Software Engineering noch besser auf Kompetenzentwicklung ausgerichtet werden kann.

1. Motivation

Durch die stetig wachsende Komplexität von Software steigen auch die Anforderungen an Informatiker, die mit einer zunehmenden Anzahl von komplexen Rollen, Schnittstellen und Aufgaben konfrontiert werden.

Software wird häufig in großen Teams mit einer Vielzahl fachlicher Rollen entwickelt. Das reicht vom Anforderungsanalysten über den Software Architekten und den Programmierer bis hin zum Software Tester. In jeder dieser Rollen sind zum Teil sehr unterschiedliche Kompetenzen notwendig. Da Software normalerweise nicht für Informatiker, sondern für "fachfremde" Auftraggeber entwickelt wird, muss zum Beispiel derjenige, der Anforderungen erhebt, über ein hohes Maß an interdisziplinärer Kommunikationsfähigkeit verfügen. Ein Software Ingenieur im Bereich Anforderungsanalyse muss in der Lage sein, sich auf für ihn fremde Denkmuster und Denkstrukturen einzulassen, sie zu verstehen und auch zwischen ihnen und der Informatik zu vermitteln. Des Weiteren muss ein Anforderungsanalytiker mit einem gewissen Verständnis für den künftigen Anwendungskontext der zu entwickelnden Software ausgestattet sein, um funktionale und nichtfunktionale Anforderungen möglichst umfassend erheben zu können. Dies setzt ein gewisses Maß an Empathie und Weitblick voraus, sich in die künftige Anwendungssituation und den Anwender hineinzuversetzen. So birgt jede Rolle im Prozess des Software Engineering ihre besonderen Herausforderungen.

Software Ingenieure müssen sich jedoch unabhängig von ihrer Rolle in ein Entwicklungsteam eingliedern können und benötigen neben ihrem fachlichen Wissen auch eine Vielzahl weiterer Kompetenzen. So müssen sie etwa in der Lage sein, Herausforderungen und Probleme rechtzeitig zu erkennen und einzuschätzen, benötigen also ein gewisses Problembewusstsein, um im richtigen Moment angemessen agieren und reagieren zu können. Software Ingenieure müssen ihr erworbenes Wissen ständig auf neue Situationen anpassen und erlernte Methoden selbständig anwenden.

Dieser Artikel konkretisiert und strukturiert den Kompetenzbegriff im Kontext von Software Engineering und stellt ein Modell zur Beschreibung von Kompetenzen vor. Er beleuchtet, wie es entwickelt wurde und welche Anforderungen es erfüllen soll. Er zeigt auf Grundlage des vorgeschlagenen Beschreibungsmodells erste Ergebnisse zu Soll-Kompetenzen eines Software Ingenieurs.

2. Forschungsinteresse

Ein Software Ingenieur benötigt neben einer Vielzahl an fachlichen auch überfachliche Kompetenzen, damit er sein fachliches Wissen in komplexen Situationen überhaupt erfolgreich umsetzen kann. Da es in Lernprozessen keine direkten und klaren Ursache-Wirkungs-Zusammenhänge gibt, können Kompetenzen lediglich trainiert und gefördert, jedoch nicht direkt vermittelt werden. Das stellt die Hochschulausbildung im Software Engineering vor große Herausforderungen. Sie möchte sowohl fachliche als auch überfachliche Kompetenzen im Software Engineering trainieren. Daher starteten sechs bayerische Hochschulen das Projekt EVELIN (Ex-

perimentelle VErbesserung des Lernens von Software EngINeering), um genau hier anzusetzen und das Lehren und Lernen von Software Engineering in seiner gesamten Bandbreite besser verstehen zu können und im Rahmen der Hochschullehre zielgerichtet Rahmenbedingungen zu schaffen, die das Trainieren von Kompetenzen optimal ermöglichen.

Bevor es aber möglich ist, sich darüber Gedanken zu machen, wie man am besten Kompetenzen trainieren kann, stellen sich folgende Fragen:

- Was meint überhaupt "Kompetenz"?
- Wozu dient eine Kompetenzbeschreibung?
- Wie beschreibt man Kompetenzen sinnvoll?
- Was genau kennzeichnet einen guten Software Ingenieur? Welche Kompetenzen soll ein Software Ingenieur haben?

3. Was ist "Kompetenz"?

Der Kompetenzbegriff ist relativ unscharf. Es gibt zahlreiche Definitionen, was unter Kompetenzen verstanden werden kann.

Erpenbeck (2007) schlägt eine Einteilung in personale, aktivitäts- und umsetzungsorientierte, fachlich-methodische sowie sozial-kommunikative Kompetenz vor. Die Definition nach Erpenbeck ermöglicht jedoch keine eindeutige Zuordnung einzelner Kompetenzen zu einer dieser Arten.

Spricht man allgemein von einer Kompetenz, um z.B. eine bestimmte Problemstellung lösen zu können, treten verschiedene Kompetenzarten in Kombination auf. Um etwa Anforderungen erfassen zu können, bedarf es sowohl fachlich-methodischer (z.B. Anforderungen korrekt beschreiben) als auch sozial-kommunikativer Kompetenzen (z.B. Kommunikationsfähigkeit, Teamfähigkeit). Auch Erpenbeck (2007, S. XII) kommt zu dem Schluss, dass Kompetenzen mehr sind als reines Wissen oder Fertigkeiten: "Kompetenzen schließen Fertigkeiten, Wissen und Qualifikationen ein, lassen sich aber nicht darauf reduzieren."

Weinert (2001, S. 27f) gibt eine allgemeinere Definition: "Kompetenzen sind die bei Individuen verfügbaren oder von ihnen erlernbaren kognitiven Fähigkeiten und Fertigkeiten, bestimmte Probleme zu lösen, sowie die damit verbundenen motivationalen (das Motiv betreffend), volitionalen (durch den Willen bestimmt) und sozialen Bereitschaften und Fähigkeiten, die Problemlösungen in variablen Situationen erfolgreich und verantwortungsvoll nutzen zu können." Angelehnt an Weinert unterscheiden wir im vorliegenden Artikel zwei Kompetenzarten:

- Fachkompetenzen entsprechen den kognitiven Fähigkeiten und Fertigkeiten.
- Überfachliche Kompetenzen beinhalten die motivationalen, volitionalen und sozialen Bereitschaften und Fähigkeiten.

Somit beschreiben überfachliche Kompetenzen hier diejenigen Kompetenzen, die gemeinhin nicht direkt als "Fachwissen" bezeichnet werden, sondern vielmehr alle Fähigkeiten, die nötig sind, um Fachwissen situationsbezogen und zielgerichtet in komplexen Situationen anwenden zu können.

Die derzeitige Unterscheidung von lediglich zwei Kompetenzarten ohne weitere Unterteilungen liegt darin begründet, dass aktuell noch keine Aussagen über feinere sinnvolle Differenzierungen getroffen werden können. Das Interesse liegt zunächst auf einer klaren Strukturierung von Forschungsdaten. Erst später ist möglicherweise eine weitere Differenzierung sinnvoll und notwendig.

4. Wozu sollten Kompetenzen beschrieben werden?

Die erforderlichen fachlichen und überfachlichen Kompetenzen im Software Engineering sollen erhoben, analysiert und beschrieben werden, um darauf aufbauend Lehrziele, didaktische Methoden und kompetenzorientierte Prüfungen (weiter) zu entwickeln. Die erhobenen Kompetenzen werden in einem Kompetenzprofil zusammengefasst und stellen so ein umfassendes Bild dar, was ein Software Ingenieur unmittelbar nach der Hochschulausbildung kennen und können soll. Kompetenzprofile sollen sowohl für den Bereich der Kerninformatik als auch für Software Engineering im Nebenfach wie etwa Mechatronik erfasst werden. Darauf aufbauend soll erforscht werden, welche Einflussfaktoren die Kompetenzförderung im Software Engineering begünstigen, so dass in der Hochschulausbildung gezielt kompetenzfördernde Rahmenbedingungen geschaffen werden können.

Die Kompetenzbeschreibung ist somit eine wesentliche Grundlage für die weitere empirisch-experimentelle Didaktikforschung. Kompetenzbeschreibungen werden benötigt, um die Ziele der akademischen Ausbildung klar beschreiben zu können und eine Vergleichsmöglichkeit zu schaffen, inwieweit diese Ziele dann tatsächlich erreicht wurden. Dazu ist eine systematische, vergleichbare Beschreibung von Kompetenzen notwendig. Kompetenzbeschreibungen bilden diese Vergleichsgrundlage, um später in Experimenten zu evaluieren, wie sich die Soll- von den Ist-Kompetenzen Studierender unterscheiden und ob die gezielte Veränderung einzelner Einflussvariablen auf den Lernprozess die Kompetenzentwicklung begünstigt hat. Die Beschreibung von Kompetenzprofilen stellt in diesem Zusammenhang eine Datenbasis zu verschiedenen Zeitpunkten sowie verschiedener Kohorten und Studiengänge dar, die dann miteinander abgeglichen werden können.

Fachliche und überfachliche Kompetenzen unterscheiden sich in ihrer Art, Beschreibung, Förde-

rung und auch Messung stark und werden daher zunächst separat betrachtet. Überfachliche Kompetenzen sind sehr schwer zu fassen, zudem kaum eindeutig definierbar und messbar.

Fachliche Kompetenzen werden mit einem geeigneten Beschreibungsschema erfasst, analysiert und beschrieben. Eine Grundlage dafür bildet SWEBOK (Abran et al. 2004), das die fachlichen Kompetenzen eines Software Ingenieurs mit vier Jahren Berufserfahrung beschreibt. Für überfachliche Kompetenzen im Software Engineering soll in EVELIN ein Schema zur Beschreibung entwickelt und mit konkreten Daten gefüllt werden. Das Ergebnis entspräche dem SWEBOK auf fachlicher Seite. Die erforderlichen überfachlichen Soll-Kompetenzen werden völlig offen und unvoreingenommen ermittelt, um ein möglichst umfassendes Bild zu bekommen. Es geht auch darum zu verstehen, was z.B. "kommunikative Kompetenz" im Kontext von Software Engineering bedeutet.

Die systematische Erfassung der Soll-Kompetenzen ist Voraussetzung dafür, Lernprozesse besser analysieren und verstehen zu können. Zentrale Fragen sind dann: Welche Faktoren wirken überhaupt auf das Lernen? Und wie können diese Einflussfaktoren systematisch erfasst werden, so dass dann zielgerichtet diejenigen angepasst und verändert werden können, die Lernprozesse unterstützen und so die Soll-Kompetenzen gezielt fördern? Hinzu kommt die Schwierigkeit, dass Wechselwirkungen zwischen den Einflussfaktoren bestehen: die Änderung einzelner Faktoren hat immer Auswirkungen auf andere. Es gilt, diese Einflussfaktoren auf Lernprozesse im Software Engineering und ihre Wechselwirkungen untereinander mittels Grounded Theory besser zu verstehen, zumal es speziell für den Bereich Software Engineering noch keine etablierte Theorie gibt.

5. Grounded Theory als Forschungsstrategie

Den methodischen Rahmen für das gesamte Vorhaben bildet die Forschungsstrategie Grounded Theory (Glaser et al. 1998), die häufig verwendet wird, wenn eine auf Daten gestützte Theorie zu einem Thema entwickelt werden soll. Sie zielt darauf ab, Sachverhalte besser zu verstehen und wird in diesem Forschungsprojekt angewandt, um Prozesse umfassend zu verstehen, die dem Lernen von Software Engineering zugrunde liegen. Dieses Verständnis ist Voraussetzung für die kompetenzfördernde Gestaltung der Lernprozesse.

Den Ausgangspunkt der Grounded Theory bilden nicht theoretische Vorannahmen. Vielmehr handelt es sich um einen zirkulären Analyseprozess, der zugleich induktive und deduktive Vorgehenselemente beinhaltet. Ausgangspunkt ist eine erklärende Hypothese, die versucht, von einer Folge auf Vorhergehendes zu schließen (Abduktion). Dann wird ein Typisierungsschema für Hypothesen entwickelt (Deduktion), das mit Forschungsdaten abgeglichen wird (Induktion) (Flick et al. 2000).

Grounded Theory besitzt unter anderem folgende wesentliche Merkmale:

- Eine Theorie wird aus Daten generiert, statt "nur" verifiziert. Der Fokus liegt auf der Entdeckung und Einbeziehung neuer Erkenntnisse, nicht auf der Verifizierung, denn das würde neu auftauchende Perspektiven unterdrücken. Folglich kann eine Theorie nie "fertig" sein, da jederzeit neue Aspekte auftauchen können.
- Ziel ist es, durch das Vergleichen von Gruppen zum einen Kategorien zu bilden und zum anderen bestehende Beziehungen zwischen diesen Kategorien an den Tag zu bringen. Nach Glaser et al. (1998, S. 48f.) "... muss unterstrichen werden, dass die Hypothesen zunächst einen vorläufigen Status haben: Sie beschreiben mutmaßliche, nicht getestete Zusammenhänge zwischen den Kategorien und ihren Eigenschaften - was natürlich nicht heißt, dass man die Hypothesen nicht so gut wie möglich verifizieren sollte. [...] Hypothesen zu generieren heißt, sie im empirischen Material zu verankern - nicht, genug Material anzuhäufen, um einen Beweis führen zu können."
- Grounded Theory fordert eine parallele Datenerhebung und -analyse, und zwar vom Beginn der Forschung an bis zur ihrem Ende.
- Theoretisches Sampling meint, dass die Stichprobe sich während des gesamten Forschungsprozesses weiter entwickelt, so dass bis zum Ende keine endgültigen Aussagen über die erhobenen und einbezogenen Daten getroffen werden können. Die entstehende Theorie steuert, wann im Forschungsprozess welche Daten wo erhoben werden.
- Grundsätzlich gilt ein Mix qualitativer und quantitativer Forschungsmethoden als zielführend.

Daraus resultiert permanente Offenheit für neue Aspekte, die jederzeit in den Forschungsprozess aufgenommen werden.

6. Methodisches Vorgehen

Das Vorgehen der Grounded Theory korrespondiert gut mit dem Anspruch einer angewandten Forschung, da hier sowohl die Forschungsmethodik als auch die Anwendungsorientierung von zu erhebenden Daten ausgehen und diese strukturieren. Zudem soll kein vorgefertigtes Bild verifiziert werden, das es für das Lernen von Software Engineering ohnehin noch nicht gibt. Ein solches Modell könnte nur auf theoretischer Basis entwickelt

werden, hätte dann aber wenig Bezug zur Anwendungsrealität.

Software Engineering ist gekennzeichnet durch ein sehr komplexes Praxisfeld sowie eine Vielzahl zu trainierender Kompetenzen, die zudem von der jeweiligen Rolle abhängen, die ein Software Ingenieur einnimmt. Damit ist der Wissenstransfer von der Theorie auf die praktische Anwendung im Software Engineering eine besondere Herausforderung. Lernen von Software Engineering muss also auf den sehr komplexen Anwendungskontext ausgerichtet sein und wird von zahlreichen Faktoren beeinflusst. Um Anhaltspunkte für Ausgangshypothesen und einen möglichst umfassenden Blick auf diese Einflussfaktoren zu erhalten, ist es sinnvoll, diese Faktoren auf einer detaillierteren Ebene systematisch zu analysieren.

6.1 Überblick über das konkrete Vorgehen

Wird vor diesem Hintergrund die Grounded Theory auf das Lernen von Software Engineering angewandt, ergibt sich folgendes konkretes Vorgehen: Die Kompetenzbeschreibungen ermöglichen einen Vergleich von tatsächlich zu verschiedenen Zeitpunkten bei verschiedenen Studierendengruppen erhobenen Ist- mit den zu erreichenden Soll-Kompetenzen. Auf Grundlage dieser Ergebnisse werden Hypothesen zu den Einflussfaktoren auf das Lernen von Software Engineering gebildet und überprüft. Konkret bedeutet dies, dass zunächst sowohl die Soll-Kompetenzen als auch die Ist-Kompetenzen Studierender erhoben werden.

Dann werden gezielt Einflussvariablen auf den Lernprozess modifiziert und die studentischen Ist-Kompetenzen zu einem späteren Zeitpunkt erneut erhoben. Zuvor definierte Messkriterien zeigen, ob und in welchem Maß sich die Kompetenzen verändert haben, und lassen Rückschlüsse auf die veränderten Einflussvariablen zu. Diese Messkriterien werden über konkrete, auf die Rahmenbedingungen angepasste Lehrziele aus den Soll-Kompetenzen abgeleitet.

Dieser Prozess wird mehrfach durchlaufen, um ein besseres Verständnis über die Einflussfaktoren auf das Lernen von Software Engineering zu erhalten und um so den Lernprozess möglichst kompetenzfördernd gestalten zu können.

Ausgehend von ersten Hypothesen und Daten sollen gezielte Veränderungen und Modifikationen an den Lehr-Lern-Prozessen vorgenommen und dokumentiert werden, so dass Schritt für Schritt bestehende Lehrveranstaltungen weiterentwickelt werden können. Dabei werden - wie bereits beschrieben - Messkriterien festgelegt, anhand derer Kompetenzentwicklung zu verschiedenen Zeitpunkten ablesbar und vergleichbar sein soll. Diese Daten werden dann zu verschiedenen Zeitpunkten erhoben. Aus dem Vergleich der veränderten Einflussvariablen und den Messwerten der Soll- und Ist-Kompetenzen zu verschiedenen Zeitpunkten werden so Rückschlüsse auf mögliche lernfördernde Einflussfaktoren gezogen, die dann gezielt modifiziert werden.

Somit gliedert sich das Vorgehen in zwei wesentliche Schritte: Das Strukturieren und gezielte Verändern von Einflussvariablen auf den Lernprozess und die Kompetenzanalysen zu verschiedenen Zeitpunkten.

Sedelmaier und Landes (2012) beschreiben das Forschungsdesign genauer (vgl. Abb. 4).

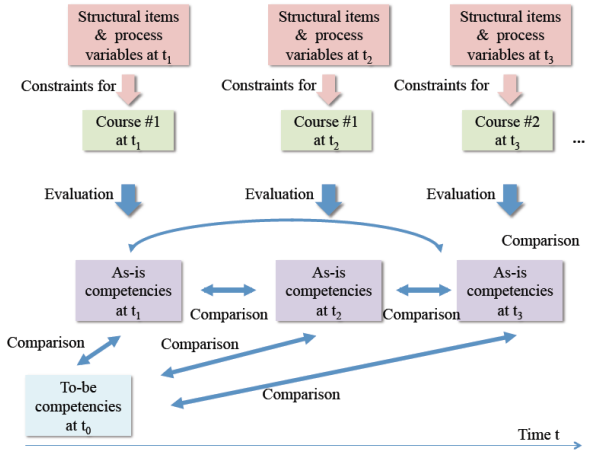


Abb. 4: Überblick über das Forschungsdesign

6.2 Erfassen von Einflussvariablen

Um die Einflussfaktoren auf Lehr-Lern-Prozesse systematisch erfassen zu können, wurden diese zunächst in Struktur-, Prozess- und Ergebnisvariablen (Donabedian 1980) unterteilt. Ergebnisvariablen werden mittels der Kompetenzprofile beschrieben. Strukturvariablen setzen sich aus den didaktischen Aspekten zusammen, die strukturelle Rahmenbedingungen auf den Lernprozess darstellen, wie z.B. Motivationen, Einstellungen, didaktische Kenntnisse der Lehrenden, Räumlichkeiten. Prozessvariablen beeinflussen den tatsächlichen Lernprozess, wie etwa Medien, Lehr- und Lernmethoden.

6.3 Kompetenzanalyse

Um Kompetenzen zu verschiedenen Zeitpunkten erfassen und dokumentieren und Vergleiche anstellen zu können, ist ein einheitliches Beschreibungsschema erforderlich.

Während zu fachlichen Kompetenzen zahlreiche Vorarbeiten zu finden sind, existieren für überfachliche Kompetenzen nur sehr allgemeingültige Einteilungen (vgl. Kap. 3). Daher werden fachliche und überfachliche Kompetenzen separat betrachtet, wodurch sich auch das Vorgehen unterscheidet.

Vorgehen bei überfachlichen Kompetenzen

SWEBOK strukturiert fachliche Inhalte des Software Engineering und bietet somit einen Anhaltspunkt, welche fachlichen Kompetenzen im Software Engineering relevant sein können.

Allerdings vernachlässigt SWEBOK überfachliche Kompetenzen völlig. Zudem ist keine mit SWEBOK vergleichbare "Landkarte" für überfachliche Kompetenzen im Software Engineering bekannt. Auch ist noch unklar, wie trennscharf sich überfachliche Kompetenzen unterscheiden lassen. Lässt sich eine Kompetenz eindeutig beispielsweise den motivationalen oder volitionalen Kompetenzen zuordnen? Diese Frage ist erst nach Erhebung erster konkreter überfachlicher Kompetenzen zu beantworten.

Da im Bereich überfachlicher Kompetenzen für Software Engineering also noch keinerlei Vorerhebungen oder vorstrukturierende Daten vorhanden sind, sind somit die zu erwartenden Inhalte unklar. Daher bietet sich ein iteratives Forschungsdesign an, das bottom-up mit völlig offenem Blick überfachliche Kompetenzen erfasst und dann erst strukturiert. Es wurde ohne ein vorhandenes Schema zur Beschreibung der Kompetenzen begonnen. Dieses entsteht erst parallel mit der Datenerhebung. Die Entwicklung eines Beschreibungsmodells für überfachliche Kompetenzen ist einer ersten Datensammlung nachgelagert.

Überfachliche Soll-Kompetenzen werden aus Sicht unterschiedlicher Stakeholder und zunächst in einem ersten Schritt aus Sicht von Unternehmen für den Bereich Kerninformatik ermittelt. Dazu wurden Interviews, die zu den fachlichen Kompetenzen geführt wurden (siehe unten), nochmals im Hinblick auf überfachliche Kompetenzen ausgewertet. Durch die Nähe zur Datenbasis der Interviews entsteht in diesem bottom-up-Prozess ein Verständnis für die überfachlichen Kompetenzen und ihre Ausprägung. Dieses Vorgehen ermöglicht einen unverstellten Blick auf die tatsächlichen Erfordernisse im Berufsleben und erzeugt schon während des Forschungsprozesses ein Gefühl dafür, was etwa mit "Teamfähigkeit" gemeint ist. Sonst häufig wenig aussagekräftige Worthülsen wie z.B. Teamfähigkeit erfahren durch dieses bottom-up-Vorgehen eine Konkretisierung für den Bereich Software Engineering. Dies ist auch der Grund, warum es in einem ersten Schritt nicht sinnvoll ist, Stakeholder direkt danach zu fragen, welche überfachlichen Kompetenzen benötigt werden. Als Antwort wäre primär eine Aufzählung dieser Worthülsen zu erwarten, es bliebe jedoch offen, was genau im Kontext von Software Engineering darunter zu verstehen ist.

Als erster Schritt zu einem Beschreibungsschema für überfachliche Kompetenzen wurden die vorhandenen Interviews zunächst im Hinblick auf jegliche überfachliche Aussagen codiert. Es entstanden 32 Codes, die unterschiedlich häufig auftraten. Diese Codes deuten auf ein Schema zur Beschreibung überfachlicher Kompetenzen hin, sind jedoch noch weiter zu clustern und zu strukturieren. Gemäß Grounded Theory geschieht dies parallel mit der Erhebung und Auswertung weiterer Daten. In diesem iterativen Prozess entsteht somit neben dem Beschreibungsschema für überfachliche Kompetenzen zeitgleich auch die inhaltliche Beschreibung überfachlicher Kompetenzen, die ein Software Ingenieur benötigt.

Vorgehen bei fachlichen Kompetenzen

Aufgrund anderer Voraussetzungen und Vorkenntnisse wurde bei fachlichen Kompetenzen zuerst top-down ein theoretisches Modell entwickelt und in einem zweiten Schritt mit konkreten Daten, also Kompetenzen, gefüllt und validiert. Das vorgeschlagene Modell zur Beschreibung fachlicher Kompetenzen wurde so in einer Pilotstudie auf seine Verwendbarkeit überprüft. Dazu wurden mittels halbstrukturierter Interviews fachliche Soll-Kompetenzen erhoben und klassifiziert, die Unternehmen von Absolventen im Bereich Kern-Informatik erwarten.

Als Grundlage für mögliche Inhalte und somit für den Interviewleitfaden wurde u.a. SWEBOK herangezogen. Um auch angrenzende Themenkomplexe wie beispielsweise Programmiertechniken und Datenbanken einfließen zu lassen, wurden auch entsprechende fachliche Inhalte vorhandener Lehrveranstaltungen berücksichtigt.

Es wurden Gespräche mit sieben Vertreten von Wirtschaftsunternehmen verschiedener Branchen (vgl. Tabelle 1) im Bereich Software Engineering geführt, aufgezeichnet, transkribiert und ausgewertet. Befragt wurden einerseits Personen mit Entscheidungskompetenz bei der Auswahl neuer Mitarbeiter und Nähe zu den fachlichen Inhalten, also z.B. Geschäftsführer oder Gruppen-/Abteilungsleiter, andererseits Absolventen der eigenen Hochschule mit inzwischen mehrjähriger Berufserfahrung.

Nr	Branche	Entscheider	ehem. Absolvent(in)
1	Marketing, Softwareentwicklung kein Kerngeschäft	X	
2	Softwareentwicklung für Banken und Versicherungen	X	X
3	Bildbearbeitung (Medizin), Embedded, Datenbanklösungen	X	

4	Automobilzulieferer (Embedded)		X
5	Softwareentwicklung und Betrieb einer Marketingplattform	X	
6	Softwareentwicklung und Betrieb einer Marketingplattform		X
7	Großkonzern, Abteilung mit Schwerpunkt Softwareentwicklung	X	

Tabelle 1: Zusammensetzung der Stichprobe

Zwar gewährleistet diese Auswahl noch keine repräsentative Stichprobe, aber immerhin eine relativ gute Abdeckung des fachlichen Spektrums und unterschiedliche Perspektiven. Folgt man der Grounded Theory, genügt dies, um erste Annahmen und Hypothesen zu bilden, die dann den Ausgangspunkt für die weitere Forschung bilden, jedoch noch nicht den Anspruch haben, eine bereits vorhandene These zu validieren.

7. Wie kann man Kompetenzen sinnvoll beschreiben?

Zunächst sind Kompetenzen mittels eines geeigneten Modells zu analysieren und beschreiben. Dazu wurden existierende Klassifikationsschemata auf ihre Eignung im EVELIN-Kontext überprüft.

7.1 Generelle Anforderungen an ein Beschreibungsmodell

Die Untersuchung vorhandener Schemata ergab ein klares Bild der Anforderungen an ein geeignetes Beschreibungsmodell im EVELIN-Kontext. Ein Kompetenzprofil für Software Engineering ist keine rein inhaltliche Sammlung von Stichworten, denn aufgrund der Anwendung in verschiedenen Fachdisziplinen soll es auch ausdrücken können, wie gut Studierende die jeweilige Kompetenz beherrschen. Auch darf ein Kompetenzprofil nicht auf Lehrzielebene formuliert sein, denn hier muss aufgrund der verschiedenen Schwerpunkte einzelner Studiengänge und der Vorstellungen und Priorisierungen der Lehrenden eine gewisse Flexibilität vorhanden sein. Gleichzeitig muss ein Beschreibungssystem auch die Beschreibung, Strukturierung und Einordnung von Lehrzielen unterstützen, so dass es als "Kompass" für die Lehr-Lern-Planung anwendbar ist. Die Lehrziele leiten sich aus dem entsprechenden Kompetenzmodell ab.

Das Modell soll die Beschreibung von Soll- und Ist-Kompetenzen zu verschiedenen Zeitpunkten und deren Vergleich ermöglichen. Da dieses Beschreibungsmodell ein Werkzeug für Planung, Messung und Analyse sein soll, muss es möglichst einfach, aber dennoch ausreichend ausdruckskräftig, verständlich und handhabbar sein.

Wichtig ist auch die Kompatibilität mit anderen Denkansätzen und Beschreibungsmodellen wie z.B. der Taxonomie von Lernzielen im kognitiven Bereich nach Bloom (1972). Es sollte - wenn nötig - zu späteren Zeitpunkten noch erweiterbar sein.

Auch sollte eine Hierarchie aufeinander aufbauender Kompetenzen zwar abbildbar, jedoch nicht zwingend sein.

Klassifikationsschemata für Lernaufgaben und Taxonomien für Software Engineering-Inhalte wurden mit diesen Anforderungen abgeglichen.

7.3 Taxonomien für fachliche Kompetenzen

Die Taxonomie kognitiver Lernziele von Bloom (1972) ist eines der bedeutendsten Klassifizierungssysteme in diesem Bereich. Bloom unterteilt Lernziele in die Hauptklassen Wissen, Verstehen, Anwendung, Analyse, Synthese und Bewertung. Diese sechs Klassen gliedern sich teilweise wiederum in Unterklassen. Insgesamt ergeben sich daraus 24 Klassifikationsmöglichkeiten, was die Einordnung von Kompetenzen erheblich erschwert. Für EVELIN soll jedoch ein Klassifikationsschema eine schnelle und einfache Zuordnung ermöglichen. Trotz der komplexen Unterstruktur ist es schwierig, Kompetenzen verschiedener Bereiche des Software Engineering wie etwa Kerninformatik, Mechatronik oder Wirtschaftsinformatik mit diesem Modell zu unterscheiden. Die Untersuchungen zeigten u.a., dass die Klassifizierung auf der Ebene der Kompetenzen oftmals widersprüchlich ist. Einerseits ist dies auf die strikte kumulative Hierarchie der Klassen zurückzuführen, anderseits sind die Anordnung und Bedeutung der höheren Klassen für diesen Einsatzzweck fraglich (Claren 2012). Da dieses Modell zur Klassifizierung von Lehrzielen entwickelt wurde, eignet es sich nur bedingt für die Beschreibung von Kompetenzen, da diese auf einer abstrakteren Ebene angesiedelt sind.

Anderson und Krathwohl (2001) veröffentlichten eine überarbeitete Fassung der Taxonomie nach Bloom. Gegenüber der Ursprungstaxonomie wurden zunächst die Klassen in aktive Verben umbenannt und die beiden letzten vertauscht, da dies nach Meinung der Autoren der steigenden Komplexität der kognitiven Prozesse besser entspricht. Die wesentliche Änderung ist jedoch die Einführung einer zweiten Dimension, welche die Art des Wissens repräsentiert. Damit ergibt sich das in Abbildung 1 gezeigte zweidimensionale Schema, das zwischen einer kognitiven und einer Wissensdimension unterscheidet.

The Knowledge Dimension	The Cognitive Dimesnion					
	1. Remember	2. Understand	3. Apply	4. Analyze	5. Evaluate	6. Create
A. Factual Knowledge						
B. Conceptual Knowledge						
C. Procedural Knowledge						
D. Metacognitive Knowledge						

Abb. 1: Lernzieltaxonomie nach Anderson und Krathwohl (2001)

Eine weitere Änderung ist der Verzicht auf die kumulativen Eigenschaften der Klassen.

Bei der Anwendung dieser Taxonomie zeigen sich jedoch grundlegende Probleme. Lehrziele bzw. Kompetenzen müssen in diesem Schema immer sowohl einer Wissenskategorie als auch einer Kategorie in der kognitiven Dimension zugeordnet werden. Dabei soll die Formulierung der Lehrziele einen Hinweis für die Einordnung in das Schema geben, was auch die eigentliche Intension des Modells erkennen lässt: Es soll in erster Linie ein Planungs- und Kontrollwerkzeug für den Unterricht sein und geht davon aus, dass bereits formulierte Lehrziele existieren (Anderson und Krathwohl 2001). Im Rahmen von EVELIN sollen Soll-Kompetenzprofile für Software Engineering entstehen, aus denen erst später Lehrziele abgeleitet werden. Generell erhöht die Aufteilung in ein zweidimensionales Schema die Komplexität, da sich für jede zu beschreibende Kompetenz 24 Klassifizierungsmöglichkeiten ergeben. Zu diesem Schluss kommen auch Fuller et al. (2007). Aufgrund der Tatsache, dass höhere Klassen in diesem Modell die darunterliegenden Klassen nicht subsumieren, erlaubt das Modell die Überlappung von Klassen. Ein Lehrziel kann also gleichzeitig in zwei benachbarte Klassen eingeordnet werden. Dadurch fehlt es dem Modell in beiden Dimensionen an der nötigen Trennschärfe, um eine möglichst eindeutige und zielführende Klassifizierung durchzuführen (Claren 2012, Granzer et al. 2008).

Fuller et al. (2007) untersuchten verschiedene Taxonomien zur Beschreibung von Kompetenzen im Bereich der Informatik, darunter auch die Taxonomien nach Bloom sowie nach Anderson und Krathwohl. Ihre Erkenntnisse decken sich weitgehend mit denen der Untersuchung im Rahmen von EVELIN. Darunter fällt etwa die problematische Einordnung in höhere Klassen, die Fragwürdigkeit der hierarchischen Ordnung bei Bloom sowie die zu hohe Komplexität des zweidimensionalen Modells von Anderson (Fuller et al. 2007). Die Autoren schlagen die sogenannte Matrix-Taxonomie (siehe Abbildung 2) vor, die auf der Taxonomie von Anderson basiert.

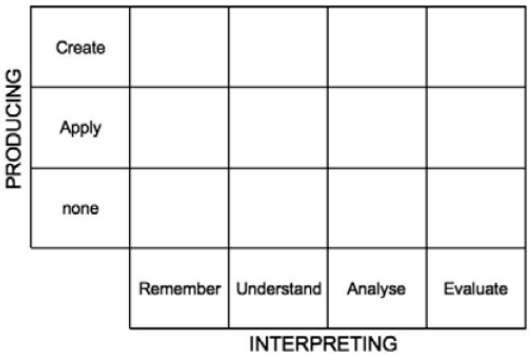


Abb. 2: Matrix-Taxonomie (Fuller et al. 2007)

Die Dimension INTERPRETING umfasst alle Ausprägungen einer Kompetenz, die sich auf Interpretieren und Verstehen beziehen. Die PRODUCING-Ebene dagegen repräsentiert die Fähigkeiten, die nötig sind, um neue Produkte zu entwerfen und entwickeln. Dadurch lassen sich die theoretischen (INTERPRETING) und praktischen (PRODUCING) Aspekte einer Kompetenz beschreiben.

Zunächst wird ein Zielfeld für eine Kompetenz festgelegt, das Studierende über verschiedene Pfade erreichen können. Dabei bewegt sich der Lernprozess vom Startpunkt aus sequenziell entlang der Ebenen zum nächsthöheren Level. Im Lernprozess kann kein Feld übersprungen werden und es ist immer nur eine Bewegung von links nach rechts bzw. von unten nach oben möglich. Im Normalfall beginnt der Lernprozess dabei in REMEBER | NONE (-).

Hier wird deutlich, dass eine andere Zielsetzung als die unseres Projektes vorliegt. Bei Fuller et al. liegt der Fokus auf der Beschreibung, Beobachtung und Planung von Lernpfaden einzelner Studierender oder homogener Studierendengruppen. Während Fuller et al. Prozesse analysieren, beabsichtigt EVELIN, zunächst "Zustände" zu beschreiben.

7.4 Das EVELIN-Modell zur Beschreibung von fachlichen Kompetenzen

Keines der analysierten Modelle eignet sich uneingeschränkt für eine Kompetenzbeschreibung im Rahmen von EVELIN. Für fachliche Kompetenzen wurde daher ein eigenes Modell entwickelt, das bewusst wesentliche Elemente bestehender Taxonomien aufgreift, um sie im Hinblick auf die bereits genannten Anforderungen sinnvoll zu kombinieren und zu erweitern.

Das EVELIN-Modell (vgl. Abbildung 3) besteht aus den Klassen Erinnern, Verstehen, Erklären, Verwenden, Anwenden und (Weiter-)Entwickeln, die sich wie folgt charakterisieren lassen:

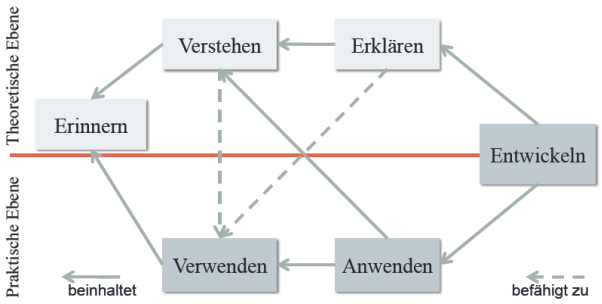


Abb. 3: EVELIN-Modell zur Beschreibung fachlicher Kompetenzen

Erinnern
- sich an Informationen erinnern und sie wortgenau wiedergeben können

Verstehen
- den Sinn / die Bedeutung von Informationen verstehen
- definieren können
- implizites Wissen
- thematisch zugehörige, aber neue Informationen zuordnen können

Erklären
- Zusammenhänge, Abhängigkeiten und Ähnlichkeiten zwischen Informationen erkennen und in eigenen Worten erklären können
- Ursache-Wirkungs-Zusammenhänge theoretisch benennen können
- Informationen strukturieren können
- Vor- und Nachteile erkennen, bewerten und theoretisch analysieren
- Verorten von Informationen in Systemen
- sachliche Analyse
- begründen

Verwenden
- Anwenden in definiertem / eingeschränktem Kontext und / oder unter Anleitung
- Umsetzung in kleinen Schritten oder mittels einer Schablone
- "Kochen nach Rezept"
- Verstehen muss keine Rolle spielen

Anwenden
- selbständiges, eigenverantwortliches Anwenden auch in "schwierigem" Umfeld und / oder Lösungswege situationsgerecht auswählen und umsetzen können
- Verstehen spielt eine Rolle

(Weiter-)Entwickeln
- Nutzen von vorhandenem Wissen zur Entwicklung oder Weiterentwicklung von Lösungen
- Schaffen von neuen Erkenntnissen, Forschung

Dabei sind Erinnern, Verstehen und Erklären auf theoretischer Ebene angesiedelt, während Verwenden, Anwenden und Entwicklung mit praktischer Umsetzung verbunden sind.

8. Welche fachlichen Kompetenzen benötigt ein Software Ingenieur?

Wie in Abschnitt 6.3. beschrieben, wurde eine Befragung von Unternehmensvertretern durchgeführt. Da die Ergebnisse nur einen Baustein in Form einer ersten Datenerhebung zur Bildung von Hypothesen bei der Erstellung der Soll-Kompetenzprofile darstellen, können die Aussagen zu einzelnen Kompetenzen auch noch nicht zu eindeutigen Ausprägungen verdichtet werden, sondern zeigen Tendenzen auf. Dazu trägt auch bei, dass Unternehmen wegen unterschiedlicher Tätigkeitsschwerpunkte unterschiedliche Vorstellungen von Software Engineering haben. Die Entwicklung von Software für Banken und Versicherungen hat etwa auch Auswirkungen auf die erwarteten Kompetenzen im Bereich Softwaretest. Nachfolgend werden wesentliche Ergebnisse der Untersuchung dargestellt, eine detaillierte Diskussion findet sich in Claren (2012).

8.1 Vorgehensmodelle

Bei Vorgehensmodellen gibt es eine Tendenz hin zu umfangreichen theoretischen Kompetenzen. So erwarten die Befragten sowohl bei agilen als auch bei plan-getriebenen Modellen eher Kompetenzen, die sich mit "Erklären" klassifizieren lassen. Eine gute theoretische Basis ist hier wichtig, da die Vorgehensmodelle meist unternehmensspezifisch angepasst sind und dafür bei Neueinsteigern eine Einarbeitung erforderlich ist. Lediglich bei testgetriebener Entwicklung ist die Ausprägung "Anwendung" stärker vertreten.

8.2 Requirements Engineering

Beim Thema Requirements Engineering erwarten die Befragten ein Verständnis dafür, welche Rollen Lasten- bzw. Pflichtenhefte im Anforderungsprozess einnehmen. Auch wenn die Mehrheit der Befragten keine spezielle Technik zur Formulierung von Anforderungen nutzt, erwartet sie dennoch grundlegendes Verständnis dafür, wie sich gute von schlechten Anforderungen unterscheiden. Absolventen sollten Techniken zur Anforderungserhebung sicher und gezielt anwenden können.

8.3 Modellierung

Im Gebiet der Modellierung werden deutlich anwendungsorientierte Kompetenzen erwartet. Absolventen sollten sowohl im Bereich der Systemmodellierung (z.B. mit der UML) als auch der Datenmodellierung (z.B. ER-Modelle) ausgeprägte praktische Erfahrungen im Studium sammeln. Im Gegensatz dazu spielt das Thema Prozessmodellierung bei den Befragten eine eher untergeordnete Rolle.

8.4 Datenbanken

Auch im Themenkomplex Datenbanken dominieren praktisch ausgerichtete Kompetenzen. Vor allem bei Datenbankentwurf und Datenabfrage erwarten die meisten Befragten Kompetenzen auf dem Niveau "Anwenden". Auch bei der Datenbankprogrammierung, wie z. B. dem Erstellen von Triggern und Funktionen, gibt es eine leichte Tendenz in diese Richtung, wobei es manchen Interviewten hier auch genügt, wenn Absolventen "verstehen".

8.5 Design und Implementierung

Hier lassen sich bei zwei Kompetenzen Aussagen über die erwarteten Ausprägungen treffen. Bei der Beherrschung objektorientierter Programmiertechniken sind sich die Befragten überwiegend einig, dass sich ein Absolvent auf Anwendungsniveau befinden sollte. Bei Design-Patterns werden keine praktischen Kompetenzen erwartet. Berufseinsteiger sollten nach Meinung der Befragten begreifen, was hinter diesem Begriff steckt, und die Prinzipien grundlegender Entwurfsmuster verstehen.

8.6 Softwaretest

Bei Kompetenzen im Bereich Softwaretest sind die Meinungen sehr unterschiedlich. Lediglich bei "Unittest" und "Testfallentwurf" können die Mehrzahl der Antworten mit "Anwenden" klassifiziert werden. Außerdem wird erwartet, dass Studienabsolventen die grundlegenden Eigenschaften, Unterschiede und Zusammenhänge verschiedener Testmetriken verstanden haben.

8.7 Konfigurationsmanagement

Die Erwartungen im Bereich Konfigurationsmanagement lassen sich auf rein theoretische Kompetenzen reduzieren. Bei den Themen Change Management, Build Management / Continuous Integration sowie Versionskontrollmechanismen können alle Antworten den theoretischen Klassen "Erinnern", "Verstehen" und "Erklären" zugeordnet werden. Die Erwartungen bei Build Management sind am geringsten, da sich Absolventen nur an grundlegende Aspekte "erinnern" sollen. Bei Change Management oder Versionskontrolle variieren die Antworten stark, jedoch jeweils mit leichter Tendenz zu "Erklären".

8.8 Projektmanagement

Projektmanagement ist überwiegend durch überfachliche Kompetenzen geprägt (z.B. Kommunikations- und Teamfähigkeit). Fachliche Kompetenzen spielen zunächst kaum eine Rolle, da es eher unüblich ist, dass ein Absolvent sofort ein Projekt leitet.

9. Welche überfachlichen Kompetenzen benötigt ein Software Ingenieur?

Wie bereits erläutert kann derzeit noch kein exaktes Schema angegeben werden, das beschreibt, in welcher Ausprägung welche überfachlichen Kompetenzen für Software Engineering konkret erforderlich sind. Es lässt sich jedoch ein erster Eindruck vermitteln, welche Kompetenzen konkret für den Bereich Software Engineering benötigt werden und was darunter zu verstehen ist. Derzeit handelt es sich um die Sichtweise von Software Ingenieuren und Führungskräften. Es geht im Folgenden nicht darum, die genannten Kompetenzen quantitativ aufzuzählen, sondern vielmehr zu verstehen, was zentrale Anforderungen an einen Software Ingenieur sind und was damit gemeint ist.

Dazu wurden in den Interviews insgesamt 306 Stellen codiert, die Rückschlüsse auf überfachliche Kompetenzen zulassen. Insgesamt wurden 32 Codes vergeben, wobei sich über 70 % der codierten Stellen auf die 12 am häufigsten genannten Codes verteilen (vgl. Tabelle 2).

Code	Anzahl Codings	Anteil an allen in %	Dokumente
Problembewusstsein	36	11,43	6
Verstehen komplexer Prozesse	30	9,52	6
Zusammenarbeit	28	8,89	7
Kommunikation mit anderen (auch interdisziplinär)	18	5,71	5
Lernbereitschaft (Gegenteil: Selbstüberschätzung)	18	5,71	5
Problemlösungsfähigkeit/ Kreativität	16	5,08	4
Selbst etwas erarbeiten können	16	5,08	6
Empathie	15	4,76	6
Offenheit für Neues	14	4,44	4
Einfügen in Organisationsstrukturen	13	4,13	5
Praktische Anwendung	12	3,81	3
Abstraktionsvermögen	10	3,17	4

Tabelle 2: Übersicht über die 12 häufigsten Codings für überfachliche Kompetenzen

Unternehmen erwarten also fachlich gut ausgebildete Absolventen, die in der Lage sind, komplexe Prozesse in Unternehmen zu verstehen. Mitarbeiter im Bereich Software Engineering sollen ihre eigene Fachlichkeit realistisch einschätzen. Sie sollen zwar in der Lage sein, sich mit ihrem Wissen im Unternehmen zu verorten, jedoch auch die Bereitschaft mitbringen, sich in Aufbau- bzw. Ablauforganisationen einzufügen. Gleichzeitig wird eine gewisse Aktivität und Selbständigkeit vorausgesetzt. Man erwartet, keine genauen Vorgaben machen zu müssen, wie und wann etwas zu tun ist. Vielmehr soll ein Software Ingenieur sich innerhalb eines Korridors aktiv und eigenverantwortlich bewegen und seine Kompetenzen selbständig einbringen und auch weiterentwickeln. Ebenso notwendig ist die Bereitschaft, sich ständig auf Neues und Unvorhergesehenes einzulassen. Es wird erwartet, dass Absolventen sich selbst und ihre Position im Unternehmen realistisch einschätzen können und so im Team und mit ihren Vorgesetzten konstruktiv zusammenarbeiten und dabei ihren Platz und ihre Rolle finden. Dies ist eng verknüpft mit drei zentralen Anforderungen, die am häufigsten in den Interviews genannt wurden: Problembewusstsein und das Verstehen komplexer Prozesse gepaart mit dem Faktor Zusammenarbeit/ Kommunikation. Zu einem ähnlichen Ergebnis kommen auch Böttcher et al. (2011).

Als zentrale Anforderung wird von einem Software Ingenieur ein ausgeprägtes Problembewusstsein erwartet. Problembewusstsein meint, sich in einem komplexen Prozess zurechtzufinden und zu erkennen, wann welche Kompetenzen wie benötigt werden, und wann und wie sie zusammenhängen. Ein Interviewpartner beschreibt dies folgendermaßen: "... das ist auch so ein Thema, das wurde auch in meinem Studium behandelt, das kann man aber auch erst sozusagen verstehen und adaptieren, wenn man das Problembewusstsein hat, [...] weil man vielleicht schon das eine oder andere Problem wirklich erlebt hat, dass man sagt: o.k., ich habe jetzt eine gewisse Problemstellung und die muss ich irgendwie lösen und dann eben auch dieses Aha-Erlebnis gehabt hat [...]"

Ein anderer Interviewpartner formuliert die Situation so: "Das Problem ist ja, man hört im Studium viele Sachen, die einem erzählt werden und – da heißt es dann: o.k., das macht man so und das macht man so – aber man weiß im Grunde genommen ganz oft nicht, warum macht man das denn eigentlich so. Weil es gibt eigentlich zehn Möglichkeiten, etwas zu machen. Im Studium werden vielleicht zwei Möglichkeiten beleuchtet [...] und dann hat man zwar gehört: Okay, das und das sollte man so machen, aber man weiß gar nicht, warum man das so machen muss und wenn jetzt jemand zu uns kommt, der sich eben über das Studium hinaus mit den Themen beschäftigt hat, dann ist es ganz oft auch der Fall, dass der eben – ja verschiedene Möglichkeiten schon abgeklopft hat und dann auch – ja von sich aus verstehen kann, nachvollziehen kann warum manche Sachen notwendig sind." Auch ein dritter Befragter verlangt Verständnis dafür "...vor allem WARUM muss ich das verwenden, welchen Nutzen ziehe ich denn daraus, wenn ich das mache."

Dabei geht es nicht ausschließlich um das Verstehen und Einordnen von Fach- und Faktenwissen, sondern auch darum, die Tragweite und Zusammenhänge komplexer Probleme bewusst wahrzunehmen. Ein Interviewpartner fasst diese Aspekte so zusammen: "Wenn man dann ins alltägliche Berufsleben hinein geschmissen wird, hat man eher das Problem, dass man von den ganzen… ja… Dingen, die da existierten, erst mal überwältigt ist."

Diese Zitate verdeutlichen, dass der Theorie-Praxis-Transfer von Kompetenzen und Wissen eine zentrale Herausforderung darstellt, die offensichtlich für viele Absolventen nicht leicht zu meistern ist und auch die Unternehmen vor einige Probleme stellt. Umgekehrt ist es für die Hochschulausbildung schwierig, den komplexen Praxisalltag in der Ausbildung abzubilden, so dass die zitierten Aha-Erlebnisse für die Studierenden erfahrbar werden.

Eine weitere wichtige überfachliche Kompetenz, die von fast allen Befragten genannt wurde, ist das Verstehen komplexer Prozesse. Für das Einfinden und angemessene Bewegen in diesen komplexen Abläufen, die dem Software Engineering inhärent sind, ist es absolut notwendig, dass ein Software Ingenieur diese überblickt und versteht. Dies schließt auch "einen Weitblick über seinen eigenen Tellerrand hinaus" mit ein. Ein Befragter beschreibt dies so: "Aber das Prinzip muss er begreifen: wie spielt das alles zusammen."

Ein dritter, sehr wichtiger Komplex, auf den sich überfachliche Kompetenzen beziehen, ist der Bereich der Zusammenarbeit. Einer der Befragten bezeichnet das sehr treffend als "Teamwork-Alltag". Ein anderer Befragter gibt hier einen Einblick in den Arbeitsablauf: "Dieser kommunikative Austausch über Themen nicht nur im Daily Scrum mal ein paar Minuten, sondern eigentlich fachlich und konkret den ganzen Tag über, im Pairing, in immer wiederkehrenden Beratungssituationen. Wir haben auch die Teams grundsätzlich immer zusammensitzen in Räumen oder an Tischinseln von Leuten, die im Team gemeinsam miteinander arbeiten. Dass also auch ein Austausch stattfindet, dass sie nicht nur sich von einem Zimmer ins andere eine E-Mail schreiben, sondern dass sie auch immer diesen mündlichen Kommunikationsweg und dass viele darüber sprechen - das versuchen wir eigentlich in jeder Weise zu befördern." Insgesamt scheint der Faktor Zusammenarbeit zunehmend an Bedeu-

tung zu gewinnen, allerdings verändert sich die Arbeitsrealität in gleichem Maße. Ein Befragter fasst es zusammen: "Teamfähigkeit ist allerdings immer besser ausgeprägt heute, weil eben immer mehr in der Ausbildung – Arbeiten im Team erledigt werden oder erledigt werden müssen. Das hat sich deutlich verbessert im Lauf der 25 oder 30 Jahre, die ich das jetzt betrachte. Am Anfang waren viele, viele, viele Einzelkämpfer da, das hat sich deutlich verbessert und das ist auch gut, die Entwicklung ist gut." Auf der anderen Seite erkennen die Befragten noch immer Handlungsbedarf: "Ich sage mal so, die Leute haben scheinbar eine gewisse technische Kompetenz, aber wie man eine vernünftige Präsentation macht, wie man einen vernünftigen Text schreibt, wie man in einem Team umgeht, wie man Konflikte löst, das sind alles Themen, die hat der gewöhnliche Hochschulabgänger nicht gelernt." Ein weiterer sieht darin einen Ausweg für die Hochschulausbildung, "...dass man Projekte definiert, wo wirklich mal drei, vier, fünf Leute dabei sind. Wo man sich untereinander abstimmen muss. Wo die Aufgaben auch so abgegrenzt sind, dass man notgedrungen Schnittstellen zu anderen hat." Ein weiterer Befragter würde es auch gut finden, wenn den Studierenden auch Projektverantwortung für ein Studienprojekt übergeben würde.

10. Fazit und Ausblick

Insgesamt lässt sich festhalten, dass ein Software Ingenieur eine Vielzahl fachlicher und überfachlicher Kompetenzen benötigt. Nun stellt sich die Frage, wie man diese benötigten Kompetenzen in der Hochschulausbildung am besten fördern kann. Auf diesem Weg ist die Beschreibung der Soll-Kompetenzen mittels eines Schemas, das diese zu verschiedenen Zeitpunkten vergleichbar macht, nur ein erster Schritt.

Die Analyse und Beschreibung der Soll-Kompetenzen für Software Engineering muss noch ausgebaut werden. Dies ist für sowohl für Kerninformatik als auch für alle anderen beteiligten Bereiche wie Mechatronik oder Studiengänge mit nicht primärem Informatikbezug notwendig. Um die Tragfähigkeit der erhobenen Kompetenzprofile weiter zu erhöhen, werden auch die Ergebnisse anderen beteiligten Verbundhochschulen berücksichtigt. Neben der Ausweitung der Datenbasis ist es zudem notwendig, weitere Blickwinkel neben der unternehmerischen Sicht einzubinden. Sinnvoll ist es, beispielsweise auch Absolventen, Studierende oder Personalverantwortliche mit einzubeziehen. Auch kann es sinnvoll sein, die entstandenen Kompetenzprofile für die verschiedenen Bereiche wie Kerninformatik oder Mechatronik miteinander zu vergleichen.

Besonders für überfachliche Kompetenzen gilt es, ein Beschreibungsschema zu entwickeln, das nahezu die gleichen Anforderungen erfüllt wie das für fachliche Kompetenzen. Dieses Modell entsteht parallel zur Analyse der vorhandenen Interviewdaten mit Fokus auf überfachlichen Kompetenzen. Dabei sollen überfachliche Kompetenzen sowohl strukturiert als auch definiert werden. Es gilt, die Frage zu beantworten, was im Kontext von Software Engineering etwa unter Team- oder Kommunikationsfähigkeit verstanden wird. Eine weitere interessante Fragestellung ist, ob überhaupt und ggf. welche Korrelationen zwischen fachlichen und überfachlichen Kompetenzen auftreten. Werden einzelne überfachliche Kompetenzen besonders häufig im Zusammenhang mit einer oder mehreren bestimmten fachlichen Kompetenzen genannt? Für beide Schemata und ihren Inhalt gilt derselbe iterative Prozess, in dem die Daten auch während des gesamten Forschungsprojektes EVELIN immer wieder überprüft und weiterentwickelt werden. Gemäß Grounded Theory fließen ständig neue Erkenntnisse und Aspekte ein, so dass ein möglichst umfassendes Bild zum Lehren und Lernen von Software Engineering entstehen kann.

Parallel dazu findet eine weitere didaktische und inhaltliche Analyse der aktuellen Lehr-Lern-Konzepte für Software Engineering und der Einflussvariablen auf die Lernprozesse statt. Die fachlichen Inhalte von Lehrveranstaltungen werden wiederum in die Weiterentwicklung von Soll-Kompetenzprofilen einfließen. Zudem ist bei der Entwicklung didaktischer Konzepte eine detaillierte Berücksichtigung der zu vermittelnden Inhalte unumgänglich.

Sobald konkretere Soll-Kompetenzprofile vorliegen, ist die Sichtweise der Lehrenden stärker einzubinden. Die Lehrenden nehmen im Lernprozess der Studierenden eine Schlüsselposition ein und prägen besonders auch durch ihre Vorstellungen und "Definitionen gelungenen Lernens" (Bender 2009) das Lehren und Lernen maßgeblich. Daher ist es sinnvoll, diese Sichtweisen zunächst separat mit der erforderlichen Sorgfalt zu berücksichtigen. So sollen nicht nur inhaltlich-fachliche Lehrinhalte mit in die Bildung von Soll-Kompetenzprofilen einfließen, sondern auch Erwartungen, Motive, Werthaltungen, Menschenbild und personenbezogene Merkmale etc. Welche Vorstellungen, Ideale, Ziele und Wertvorstellungen haben die Lehrenden? Darauf aufbauend werden zusammen mit den Lehrenden aus den Soll-Kompetenzprofilen konkrete kompetenzorientierte Lehrziele abgeleitet und Messkriterien für deren Erreichung definiert. Auch dieses Vorgehen ist wiederum ausgerichtet auf die jeweiligen Lehrenden und die Lehrveranstaltungen, denn Lehrziele können nicht allgemeingültig formuliert sein.

Weitere zu erhebende Daten sind die Erwartungen, Motivationen und Vorkenntnisse der Studierenden. In diesem Kontext stellt sich unter anderem die Frage, ob Studierende, die sich für ein bestimmtes Fach entscheiden, signifikante Eigenschaften aufweisen. Auch das sind Faktoren, auf die Lehr-Lern-Konzepte zugeschnitten werden müssen. Können sowohl bei den Lehrenden als auch bei den Studierenden gewisse Systematiken oder Muster erkannt werden, die bei der Entwicklung didaktischer Konzepte berücksichtigt werden müssen und zumindest für Software Engineering oder Teilbereiche davon signifikant auftreten?

So soll versucht werden, die Lücke zwischen den Soll- und Ist-Kompetenzen der Studierenden Schritt für Schritt zu verringern und die Hochschulausbildung im Software Engineering noch weiter zu verbessern.

Danksagung

Wir danken den Gutachtern für hilfreiche Anmerkungen zu einer früheren Version dieses Beitrags.

Das Projekt EVELIN wird gefördert durch das Bundesministerium für Bildung und Forschung unter der Fördernummer 01PL12022A.

Literatur

Abran, A. / Moore, J.W., Hrsg. (2004): Guide to the Software Engineering Book of Knowledge. Los Alamitos, CA: IEEE Computer Society Press. Verfügbar unter: http://www.computer.org/portal/web/swebok/htmlformat

Anderson, L. / Krathwohl, D., Hrsg. (2001): A taxonomy for learning, teaching, and assessing. A revision of Bloom's taxonomy of educational objectives. New York: Longman.

Bender, W.: Wie kann Qualitätsentwicklung aus dem Pädagogischen heraus entwickelt werden? - Pädagogische Reflexivität. In: Dehn, C., Hrsg. (2009): Pädagogische Qualität. Hannover: Expressum-Verlag, 69-77.

Bloom, B. (1972): Taxonomie von Lernzielen im kognitiven Bereich. Weinheim: Beltz.

Böttcher, A., Thurner, V., Müller, G. (2011): Kompetenzorientierte Lehre im Software Engineering. In: Ludewig, J., Böttcher, A., Hrsg. (2011): SEUH 2011, 33-39.

Claren, S. (2012): Erhebung und Beschreibung von Kompetenzen im Software Engineering. Masterarbeit, Hochschule Coburg.

Donabedian, A. (1980): Explorations in Quality Assessment and Monitoring: The definition of quality and approaches to its assessment. Ann. Arbor, MI: Health Administration Press, 1980.

Erpenbeck, J., Hrsg. (2007): Handbuch Kompetenzmessung. Erkennen, verstehen und bewerten von Kompetenzen in der betrieblichen, pädagogischen und psychologischen Praxis. 2. Aufl. Stuttgart: Schäffer-Poeschel.

Flick, U. / von Kardorff, E. / Steinke, I. (2000): Was ist qualitative Forschung? Einleitung und Überblick. In: Qualitative Forschung. Ein Handbuch. Reinbeck: Rowohlt, 13-29.

Fuller, U. et al (2007): Developing a computer sicience-specific Learning Taxonomy. Verfügbar unter http://www.cs.kent.ac.uk/pubs/2007/2798/content.pdf, abgerufen am 24.09.2012.

Granzer, D. / Köller, O. (2008): Kompetenzmodelle und Aufgabenentwicklung für die standardisierte Leistungsmessung im Fach Deutsch. In: Bremerich-Vos, A. / Granzer, D. / Köller, O., Hrsg. (2008): Lernstandsbestimmung im Fach Deutsch. Gute Aufgaben für den Unterricht. Weinheim: Beltz, 10-49.

Glaser, B. / Strauß, A. (1998): Grounded theory - Strategien qualitativer Forschung. Bern: Hans Huber.

Sedelmaier, Y. / Landes, D. (2012): A research agenda for identifying and developing required competencies in software engineering. Proc Int. Conference on Interactive Collaborative Learning 2012.

Weinert, F. (2001): Leistungsmessung in Schulen. 2. Auflage Weinheim: Beltz.

Werkzeuge in und für die Ausbildung

Alles nur Spielerei? Neue Ansätze für digitales spielbasiertes Lernen von Softwareprozessen

Jöran Pieper, Institute for Applied Computer Science, FH Stralsund

Joeran.Pieper@fh-stralsund.de

Zusammenfassung

Softwareprozesse beschreiben Ansätze für die Produktion und Evolution von Software. Sie gehören zu den Wissensgebieten des Software Engineering (SE), die sich weniger gut allein durch klassische Vorlesungen vermitteln lassen. Kursprojekte, die Vorlesungen häufig begleiten, werden durch akademische Rahmenbedingungen begrenzt.

Simulation und Digital Game-Based Learning (DGBL) werden große Potentiale bei der Erweiterung der Lernerfahrung über Vorlesungen und Kursprojekte hinaus zugesprochen. Sie können dazu genutzt werden, Einsicht in die Notwendigkeit von Softwareprozessen zu erzeugen und den Erfahrungshorizont von Studierenden des SE auf virtuelle und effiziente Art und Weise zu erweitern.

Verschiedene Anstrengungen unterschiedlicher Forschungsgruppen weltweit zeigen ermutigende Ergebnisse. Im Forschungsprojekt *Sim4SEEd* werden existierende Erfahrungen gesammelt und neue Ideen entwickelt, um das Potential digitaler Lernspiele in dieser Domäne zu erweitern und zu einer weiter verbreiteten Nutzung in der SE-Ausbildung anzuregen.

Die vorgestellten Ideen dienen als Kernelemente für die Entwicklung einer neuen Spielumgebung, welche das Erlernen von Softwareprozessen effektiv und effizient unterstützt.

Einleitung

Die Entwicklung komplexer Softwaresysteme verlangt nach gut ausgebildeten Softwareingenieuren, welche in der Lage sind, die richtigen Technologien, Werkzeuge und Prozesse auszuwählen, um dynamischen Anforderungen gerecht zu werden.

Zu den großen Herausforderungen heute und in Zukunft gehören dabei die zunehmende Vielfalt und die Forderung nach kürzeren Entwicklungszeiten bei gleichzeitiger Sicherstellung vertrauenswürdiger Qualität (Sommerville, 2010). Die zunehmende Vielfalt umfasst neben Technologien, Plattformen, Werkzeugen und Anwendungsdomänen auch Methoden und Softwareprozesse. Variierende Szenarien erfordern die Berücksichtigung verschiedener Aspekte und das Setzen unterschiedlicher Prioritäten.

Die Ausbildung im Software Engineering (SE) muss dieser Vielfalt Rechnung tragen, um die Softwareingenieure von morgen zu befähigen. Aus Zeitgründen können die Lerninhalte stets nur eine Auswahl aus einem breiten Spektrum sein. Während die Auswahl der Inhalte im Detail variieren wird, ist es breiter Konsens, dass neben allen technologischen Aspekten und Werkzeugen ein fundiertes Wissen über Softwareprozesse essentiell für die erfolgreiche Gestaltung von Softwareprojekten ist.

Motivation

Aufkommende und sich in den Vordergrund drängende agile Prozesse propagieren den Verzicht auf Ballast – die Nicht-Nutzung von Methoden und Werkzeugen, welche wichtige Bausteine traditionellerer Entwicklungsprozesse bilden. Ohne Erfahrung in der Entwicklung komplexerer Softwaresysteme können die Konsequenzen der Nicht-Nutzung von Methoden und Werkzeugen jedoch nur schwerlich qualifiziert eingeschätzt werden. Das Kennenlernen verschiedener Softwareprozesse kann an dieser Stelle helfen, Methoden und Werkzeuge sowie deren Wirkung besser einschätzen zu können.

Durch Lehrende gesteuerte Kursprojekte ergänzen heute gewöhnlich klassische Vorlesungen im Bereich des SE. Akademische Rahmenbedingungen, wie begrenzte Zeit und die Notwendigkeit, individuelle Leistungen der Studierenden zu bewerten, begrenzen Größe, Umfang und Typen solcher Projekte. Um eine positive Lernerfahrung der Studierenden und einen positiven Projektabschluss zu ermöglichen, wird durch Lehrende häufig eine Vorauswahl von geeigneter Technologie, Werkzeugen, Methoden und Prozessen getroffen, der die Studierenden dann folgen sollen. Die Erhö-

hung des Projektumfangs und der Anzahl der Projektmitglieder, die Anhebung des Kommunikationsbedarfs, eine stärkere Teilung von Aufgaben und Verantwortlichkeiten sowie das Beharren auf einem definierten Projektergebnis erhöhen die Realitätsnähe und vermitteln Studierenden idealerweise einen Eindruck von echter Projektarbeit nach dem Studium.

Der Blick auf den Softwareprozess als Ganzes geht in solchen Projekten jedoch allzu schnell verloren, wenn Projektmitglieder unter Zeitdruck damit beschäftigt sind, Projektartefakte zu liefern. Studierende erhalten nicht die Möglichkeit in verschiedene Rollen zu schlüpfen, um bspw. in der Rolle eines Projektmanagers Zielkonflikte zu spüren und richtungweisende Entscheidungen treffen zu müssen. Die begrenzte zur Verfügung stehende Zeit ermöglicht es nicht, alternative Strategien zu verfolgen oder gar andere Softwareprozesse auszuprobieren.

Sommerville betont, „engineering is all about selecting the most appropriate method for a set of circumstances“ (Sommerville, 2010, S. 7). Um dies zu ermöglichen, müssen zukünftige Softwareingenieure ihre Optionen kennen. Sie müssen in der Lage sein, vielfältige Aspekte und Perspektiven zu berücksichtigen. Simulation und Digital Game-Based Learning (DGBL) können neben Kursprojekten einen wichtigen Beitrag dabei leisten, Softwareprozesse kennenzulernen.

Simulation und digitale Spiele in der Software Engineering Ausbildung

Ziel der SE-Ausbildung sollte es sein, ein tiefes Verständnis von Inhalten und deren Aufnahme in den eigenen Wertekanon zu fördern. (Ludewig, 2009). Folgende Aussage zeigt, wie dies gelingen kann: „Jeder Sinn, den ich selbst für mich einsehe, jede Regel, die ich aus Einsicht selbst aufgestellt habe, treibt mich mehr an, überzeugt mich stärker und motiviert mich höher, als von außen gesetzter Sinn, den ich nicht oder kaum durchschaue…“ (Reich, 2008, S. 95). Folgt man diesem idealtypischen Grundsatz konstruktivistischer Didaktik, so besteht die Aufgabe einer geeigneten Lernumgebung darin, die Eigenaktivitäten der Lernenden anzuregen, um die aktive Konstruktion von Wissen bei der Bearbeitung komplexer authentischer Situationen und Probleme zu unterstützen. Die Betrachtung des Lerngegenstands aus verschiedenen Perspektiven, Artikulation und Reflexion im sozialen Austausch tragen grundlegend zum erfolgreichen Lernen bei (Kritzenberger, 2005). Verschiedene Ansätze aus dem konstruktivistischen Methodenbaukasten finden in der SE-Ausbildung bereits Anwendung (Hagel, Mottok, 2011). Simulationen und digitale Spiele bieten in diesem Umfeld besonders geeignete Merkmale.

Simulation eröffnet Wege, relevante Aspekte einer (fast) realen Welt mit den flexiblen Fähigkeiten von Simulationsexperimenten zu verbinden. Zu diesen Fähigkeiten gehört die Möglichkeit,

- Zeit zu komprimieren oder auszudehnen,
- Quellen von Variationen gezielt zu steuern,
- Experimente zu beliebigen Zeitpunkten anzuhalten und zu analysieren,
- Systemzustände (wieder-)herzustellen um Experimente zu wiederholen, sowie
- den Detaillierungsgrad zu definieren.

Ergänzt man Kursprojekte um Lernformen mit diesen Merkmalen, so lässt sich ein hohes Maß an Flexibilität gewinnen. Studierende und Lehrende sind in diesem Fall nicht an das operative Erfordernis gebunden, tatsächlich Software zu produzieren, um den damit verbundenen Prozess lebhaft zu erfahren. Die gezielte Erkundung verschiedener Softwareprozesse in verschiedenen Szenarien wird damit auch in einem begrenzten Zeitfenster möglich. Da keine Ergebnisse in einem echten Kursprojekt gefährdet werden, kann ungezwungenes spielerisches Experimentieren Teil der Lernerfahrung werden. Studierende können auch in (simulierten) Situationen agieren, welche aufgrund begrenzter Ressourcen real nicht denkbar wären.

“Simulations are most engaging when made into games” (Prensky, 2007, S. 225). Spiel und Nachahmung bleiben als natürliche Lernstrategien lebenslang wichtige Akkommodations- und Assimilationsstrategien (Rieber, 1996). Der fesselnde und motivierende Charakter von Spielen fördert die intrinsische Motivation der Lernenden, indem er sie dazu motiviert, die Verantwortung für den eigenen Lernfortschritt zu übernehmen (Akilli, 2007). Jede Verbindung von Ausbildungsinhalt *(educational content)* und digitalen Spielen wird dabei als *Digital Game-Based Learning (DGBL)* bezeichnet (Prensky, 2007). DGBL fördert den Lernprozess, indem durch die Integration attraktiver Spielelemente – wie Interaktivität, Herausforderungen, kontinuierlichem Feedback, Multimedialität, dem Gefühl der Selbstwirksamkeit, Wettbewerb und Belohnungen – eine motivierende und fesselnde Lernumgebung bereitgestellt wird. „Digitale Spiele können demnach […] ein selbstgeleitetes Lernen durch Exploration ermöglichen und befördern.“ (Breuer, 2010, S. 12)

Bisherige Ansätze

Wir finden heute bereits eine Reihe von verfügbaren Anwendungen von Simulation und DGBL für den Bereich des SE. Tabelle 1 fasst Charakteristik und Fähigkeiten verschiedener Ansätze zusammen, in welchen es explizit um SE-Inhalte, Softwarepro-

zesse und deren Elemente geht. Nicht alle in der Tabelle aufgeführten Projekte sind für den direkten Einsatz in der SE-Ausbildung verfügbar. Einige Ansätze werden nicht mehr aktiv entwickelt. Die in den jeweiligen Projekten gewonnenen Erkenntnisse sind jedoch auch in diesen Fällen in Publikationen dokumentiert. In der letzten Spalte der Tabelle werden den bestehenden Ansätzen die Ziele des Projekts *Sim4SEEd* gegenübergestellt.

	SimSE (Navarro, 2006)	SimjavaSP (Shaw, Dermoudy, 2005)	SESAM (Drappa, Ludewig, 2000) /AMEISE (Mittermeir u. a., 2003)	Incredible Manager (Barros u. a., 2006)	OSS (Sharp, Hall, 2000)	MO-SEProcess (En Ye u. a., 2007)	SimVBSE (Jain, Boehm, 2006)	Sim4SEEd* (Pieper, 2012)
Einzel(E)/Mehrspieler(M)-Spiel	E	E	E	E	E	M	E	M
Zusammenarbeit und Wettbewerb unter Spielern	-	-	-	-	-	o	-	+
Dashboards für Team, Kurs und Lehrende	-	-	-	-	o	-	-	+
Spieler in der Rolle eines Projektmanagers	+	+	+	+	-	-	+	+
Spieler mit verschiedenen Aufgaben	-	-	-	-	+	+	-	o
Wiederholbarkeit von Spielabschnitten	+	-	+	-	-	-	-	+
Ermöglicht Abbildung verschiedener Softwareprozesse	+	-	+	-	-	-	-	+
Anzahl verfügbarer Softwareprozesse	6	1	1	1	-	1	1	
Domain Specific Language (DSL) für die Prozessmodellierung	-	-	+	-	-	-	-	o
Regeleditor für die Prozessmodellierung	+	-	-	-	-	-	-	o
Trennung von Softwareprozess und Anwendungsszenarios	-	-	-	-	-	-	-	+
Integration von Industriestandards für die Beschreibung des Softwareprozesses	-	-	-	-	-	-	-	+
Grafische Repräsentation der Nicht-Spieler-Figuren mit welchen interagiert wird	+	o	-	o	+	+	o	+
Nutzung echter Werkzeuge aus der Softwareentwicklung	-	-	-	-	o	-	-	o
In Komponenten für teilweise Wiederverwendung entworfen	-	-	-	-	-	-	-	+
Öffentlich verfügbar	+	-	+	+	-	-	+	+

"+" = ja/enthalten, "-" = nein/nicht enthalten, "o" = teilweise enthalten, "*" = Projektziele

Tabelle 1: Vergleich von DGBL-Anwendungen

Die typische Aufgabe für einen Spieler besteht in diesen Lernspielen i.d.R. darin, als Projektmanager simulierte Softwareentwickler erfolgreich durch ein simuliertes Softwareprojekt zu dirigieren. Dazu werden Personen in das Projektteam aufgenommen oder aus diesem entfernt. Den simulierten Figuren im Team werden dann verschiedene Aufgaben, welche mit dem abgebildeten Softwareprozess korrespondieren, zugewiesen. Dabei entscheidet der Spieler, wer wann welche Aufgabe im Softwareprozess erhält. Die Aufgaben umfassen SE-Tätigkeiten, die Auswirkungen auf den Fortschritt und die Qualität eines Softwareprojekts haben. Das eingesetzte virtuelle Personal wird virtuell entlohnt. Erfolgreich ist, wer sein virtuelles Projekt fristgemäß und ohne Budgetüberschreitung in akzeptabler Qualität abliefert.

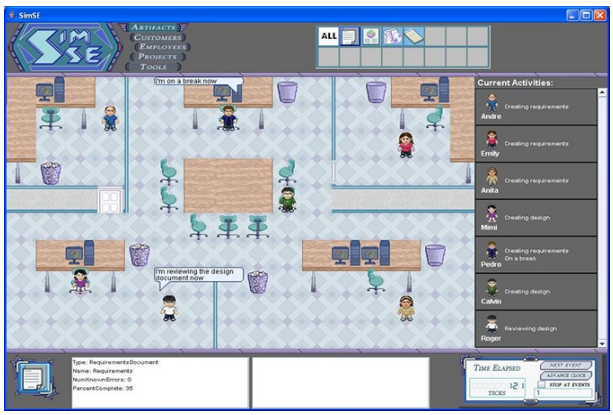


Abb. 1: *SimSE* im Einsatz

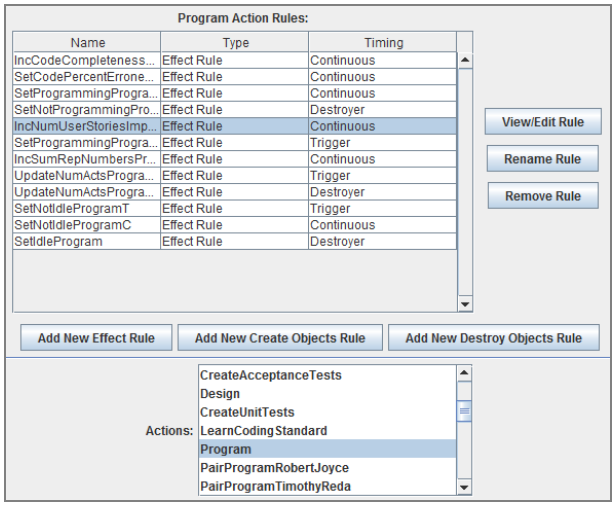


Abb. 2: Ausschnitt aus *SimSE's* Regeleditor

Abbildung 1 zeigt einen Screenshot von *SimSE* (Navarro, 2006), einer an der University of California, Irvine, entwickelten Spielumgebung, welche aktuell die Erkundung der zahlreichsten Softwareprozesse erlaubt. Die simulierten Soft-

wareentwickler mit ihren verschiedenen Eigenschaften, welche durch das simulierte Softwareprojekt gesteuert werden, sind neben bereits erstellten Artefakten zu erkennen. Abbildung 2 stellt den grafischen Regeleditor von *SimSE* dar, über welchen sowohl der Softwareprozess als auch das spezifische Anwendungsszenario modelliert werden. Sichtbar ist hier nur ein kleiner Ausschnitt der modellierten Regeln des Simulationsmodells, welches die Basis des Spiels darstellt.

Ergebnisse bisheriger Forschungsprojekte

Die Auswertung bisheriger Forschungsprojekte zeigt großes Potential aber auch einige Ambivalenzen. Bisherige Ansätze scheinen besser geeignet, bereits erworbenes Wissen zu festigen, als neues Wissen zu vermitteln (Wangenheim, Shull, 2009). Einige Projekte waren nicht in der Lage eine Wirkung von spielbasiertem Lernen nachzuweisen. Andere ermutigende Studien zeigten, dass der Einsatz von Lernspielen geeignet war, auf Seiten der Studierenden Empathie für die Notwendigkeit von Methoden und Prozessen im SE zu entwickeln. Die Lernenden fanden eingesetzte Spiele fesselnd, einfach zu benutzen und zogen sie traditionellen Lehrmethoden vor. Die sorgfältige Planung und Einbettung von Spielaktivitäten in das Curriculum, ausreichende Anleitung, detailliertes Feedback, Diskussion sowie die gründliche Auswertung und Erklärung der Spielresultate werden dabei als essentiell wichtig erachtet (Wangenheim, Shull, 2009).

Die verwendete Methodik der Evaluierungen dieser spielbasierten Ansätze wird durchaus kritisch bewertet (Wangenheim, Shull, 2009), was auch in anderen Anwendungsdomänen von spielbasiertem Lernen der Fall ist (Rieber, 1996; S. Egenfeldt-Nielsen, 2007).

Das Potential ist bei weitem noch nicht ausgeschöpft.

Die meisten verfügbaren digitalen Lernspiele sind ausschließlich für Einzelspieler konzipiert. Als solche fordern Sie von jedem Spieler die Auseinandersetzung mit dem gesamten abgebildeten Softwareprozess, fördern also den Blick auf den Prozess als Ganzes. Jedoch mangelt es bei einer isolierten Spielerfahrung an motivierender Zusammenarbeit und an motivierendem Wettbewerb.

Die Architekturen der Spiele erlauben es Lehrenden nicht, sich einen schnellen Überblick über den bisherigen Fortschritt aller Spieler im Kurs zu erhalten.

Die meisten verfügbaren Ansätze unterstützen nur einen spezifischen Softwareprozess. Angesichts aktueller Entwicklungen und der Vielfalt verfügbarer Prozesse erscheinen die Anpassungsfähigkeit und die Auswahlmöglichkeit verschiedener Prozessmodelle äußerst wichtig.

Die Erzeugung und Anpassung von Simulationsmodellen erfordern einen erheblichen Einarbeitungsaufwand, der außerhalb der jeweiligen Spielwelt kaum direkt von Nutzen ist.

Es mangelt an einer Trennung von Softwareprozessmodell und dem konkreten Anwendungsszenario dieses Prozesses im Spiel, was eine Wiederverwendung erschwert. Visualisierungen der Softwareprozesse werden nicht angeboten. Diese würden die Validierung der schnell komplex werdenden Modelle enorm erleichtern und Transparenz erzeugen. Lehrende könnten leichter überblicken, ob das Modell den Softwareprozess korrekt abbildet und die gewünschten Lerninhalte transportiert. Visualisierung in Kombination mit der Trennung von Softwareprozess und seinem Anwendungsszenario im Spiel wäre als unterstützendes Lernmaterial auch für die Studierenden hilfreich. Die Einbindung von unterstützendem Lernmaterial über den jeweiligen Softwareprozess in die Spielwelt wird in existierenden Ansätzen nicht unterstützt.

Sim4SEEd – Kernelemente einer neuen digitalen Spielumgebung für Softwareprozesse

Nach einer Analyse existierender Ansätze und Anwendungen wurde eine Reihe von Kernelementen identifiziert, welche die Basis einer neuen webbasierten Spielumgebung bilden. Diese wird aktuell im Rahmen des Forschungsprojekts *Sim4SEEd – **Sim**ulation and Digital Game-Based Learning for **S**oftware **E**ngineering (Process) **Ed**ucation* entwickelt. Dabei werden zwei wesentliche Ziele verfolgt:

1. Um Lernerfolg zu maximieren, wird die Spielumgebung die Anforderungen an konstruktivistische Lernumgebungen umfassender als bisherige Ansätze unterstützen. Die Unterstützung sozialer Interaktion, frühes Feedback als Anreiz für Artikulation und Reflexion sowie die Bereitstellung multipler Perspektiven auf den jeweiligen Softwareprozess bilden hierbei Schwerpunkte.
2. Auch gute Lösungen finden nur dann ihren Weg in die Praxis, wenn sich ihre Nutzung für Anwender nicht unnötig kompliziert gestaltet. Lehrende werden unter Ausnutzung von Synergien von der transparenten Erstellung des Simulationsmodells über die Administration des Lernspiels bis hin zu kursweiten Auswertungen unterstützt.

Die nun folgenden Abschnitte beschreiben die Kernelemente, welche zur Erreichung dieser Ziele beitragen werden.

Soziale Interaktion als Kollaboration und Wettbewerb

Die Kombination aus Einzelspiel, in dem alle Entscheidungen selbst getroffen werden, und motivierenden sozialen Elementen ist wünschenswert. Das Hinzufügen eines Teams zum individuellen Spiel jedes Einzelnen erscheint hier erfolgsversprechend. Dabei erkunden und bearbeiten alle Spieler individuell einen gesamten Softwareprozess. Gleichzeitig jedoch agiert jeder Spieler auch in einem Team und sammelt für dieses Punkte. Indem die Performance des Teams – die Kumulation der Einzelspielerergebnisse – als primärer Erfolgsfaktor herangezogen wird, werden Zusammenarbeit und Diskussion innerhalb der Teams gefördert.

Dashboards, welche einerseits Teamrankings und andererseits Rankings einzelner Spieler innerhalb dieser Teams zur Verfügung stellen, bieten dabei Orientierung. Sie zeigen an, wie gut bisher getroffene Entscheidungen in Relation zu anderen Spielern waren. Diese Orientierung dient als Ausgangspunkt für Interaktionen innerhalb der Teams und als Anlass für den ständigen Versuch, Entscheidungen zu optimieren.

nicht geboten werden kann. Lernende erhalten Feedback zu einem viel früheren Zeitpunkt, als dies in einem realen (Kurs-)Projekt der Fall wäre.

Mehr noch – durch die vorgeschlagene Kombination aus Einzelspiel und Zusammenarbeit im Team, kann parallel auch aus Erfahrungen anderer Spieler gelernt und profitiert werden. Das Spiel bietet damit die Basis für einen *„probe, hypothesize, reprobe, rethink cycle“* (Gee, 2007, S. 87 ff.), der zu einer tieferen Auseinandersetzung mit dem simulierten Softwareprozess führt. Die im Spiel gebotene Transparenz geht über die in vielen Projekten vorgefundene Realität hinaus und ist im Idealfall ein Anstoß für das Hinterfragen von Intransparenz in erlebten oder noch folgenden (Kurs-)Projekten der Lernenden.

Abbildung 3 zeigt einen Mockup-Entwurf des webbasierten Spiels mit Kollaborationselementen. Im Kopf der Seite ist erkennbar, dass der Spieler Austin Mitglied des Teams 3 ist. Auf der rechten Seite befinden sich Informationen über das aktuelle Ranking, ein Team-Chat und Nachrichten vom Spielleiter. Das Ranking stellt dar, dass sich das Team 3 momentan an Platz 3 (von 10 Teams) befin-

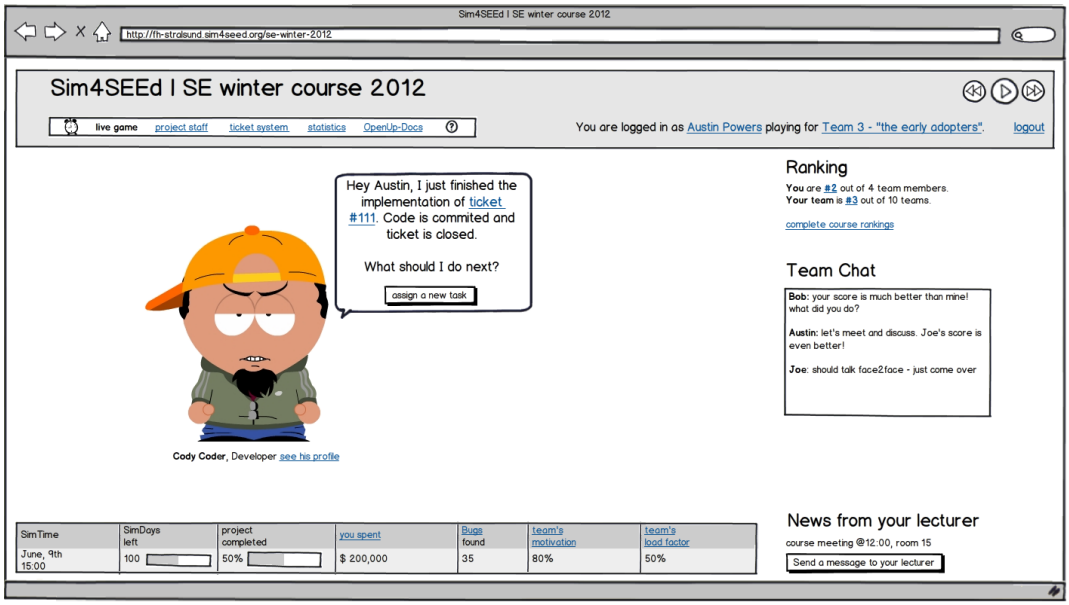


Abb. 3: Mockup mit Kollaborationselementen

Dazu bietet das Spiel die Möglichkeit, einzelne Spielabschnitte zu wiederholen. Wiederholbarkeit in Kombination mit verfügbarer Orientierung unterstützt die Analyse und Reflexion getätigter Schritte und fördert somit die Anwendung des im Spiel erworbenen Wissens. Gleichzeitig stärkt Sie das Gefühl der Selbststeuerung *(control)* der Spielenden und steigert damit deren Motivation.

An dieser Stelle zeigt sich eine Qualität, welche in Kursprojekten in dieser komprimierten Form

det. Auf das Team bezogen drückt es aus, dass Austin innerhalb seines Teams von vier Teammitgliedern am zweitbesten agiert hat. Ein Teammitglied hat bisher also noch bessere Entscheidungen getroffen. Dies ist in diesem Beispiel Anlass für eine kleine Diskussion im Team-Chat. Die Teammitglieder verabreden sich zu einem persönlichen Treffen, in welchem Entscheidungen analysiert und optimiert werden.

Eine Architektur, welche Lehrenden einen schnellen Überblick über den Spielfortschritt des gesamten Kurses bietet, unterstützt die gezielte Interaktion bei Verständnisproblemen sowie die kursweite vergleichende Auswertung und reflektierende Nachbesprechung der Spielergebnisse.

Nutzung von Synergien mit Industriestandards

Der einheitliche Zugang zu Softwareprozessen wird durch deren verschiedene Darstellungsformen erschwert. Die Softwareindustrie hat einigen Aufwand in die vereinheitlichte Dokumentation und Kommunikation von Softwareprozessen investiert. Basierend auf der *Software and Systems Process Engineering Metamodel Specification (SPEM) Version 2.0* (Object Management Group, 2008) und ihrer Vorgängerin *Unified Method Architecture (UMA)* stehen mit dem *Eclipse Process Framework (EPF)* (The Eclipse Foundation, 2012) Werkzeuge bereit, um Softwareprozesse und Methoden zu dokumentieren, zu konfigurieren und zu veröffentlichen.

Inhalte werden mit den EPF-Werkzeugen standardisiert und relativ komfortabel erstellt. Dabei kommt ein gemeinsames einheitliches Vokabular mit definierter Semantik zum Einsatz. Der in der Abbildung 4 dargestellte *EPF Composer* verwendet die vertraute *Unified Modeling Language (UML)* um grafische Visualisierungen der modellierten Softwareprozesse zu erzeugen. Diese Visualisierungen unterstützen die komfortable Erkundung der erzeugten Prozessmodelle.

Der hierbei erbrachte Einarbeitungsaufwand der Lehrenden und Lernenden ist über eine spezifische Simulations- und Spielwelt hinaus von Wert und Nutzen.

Unterstützung der Erkundung neuer Softwareprozesse

Die Evaluation bestehender Ansätze digitaler SE-Lernspiele hat ergeben, dass diese weniger geeignet waren, neue Lerninhalte zu vermitteln (Wangenheim, Shull, 2009). Ein Grund dafür dürfte sein, dass diese Umgebungen die Einbettung von Lern-

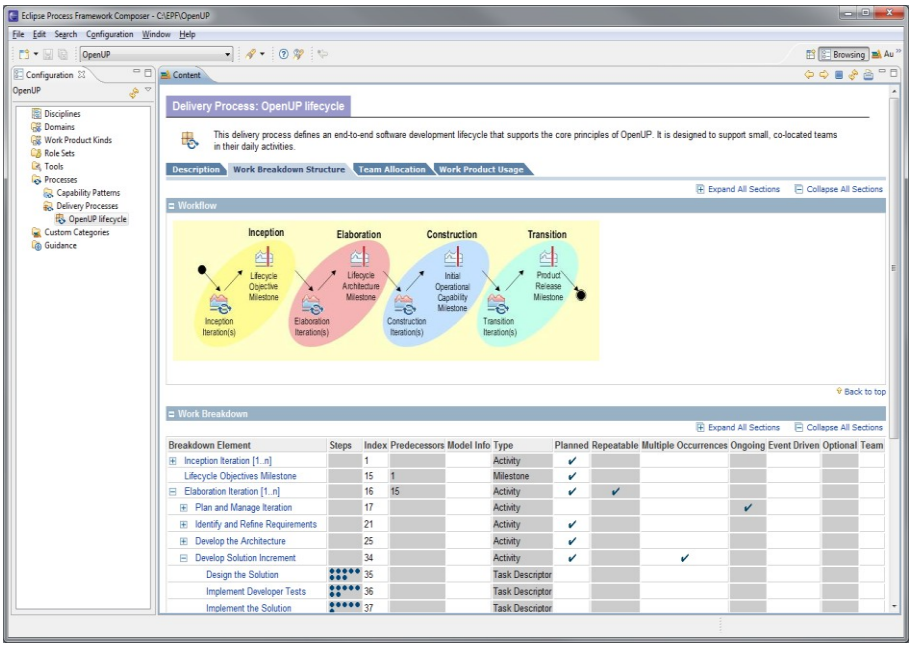


Abb. 4: EPF Composer im Einsatz

EPF stellt über diese Werkzeuge hinaus bereits eine umfangreiche Bibliothek von Softwareprozessinhalten bereit.

Die Nutzung dieser Industriestandards zur Generierung (von Teilen) der Simulationsmodelle als Basis digitaler Spiele bietet eine Reihe von Vorteilen. Rollen, Phasen, Aufgaben, Prozessschritte, Artefakte, Templates und Leitfäden können extrahiert und in die Simulations- und Spielwelt eingebettet werden. Bereits vorhandene Inhalte der EPF Bibliothek werden dabei effizient genutzt. Neue material nicht unterstützen.

Die Verwendung des *Eclipse Process Framework (EPF)* bietet sich auch an, um Lerninhalte eines Softwareprozesses in die Simulations- und Spielwelt zu integrieren. So bietet der *EPF Composer* die Möglichkeit, den dokumentierten Prozess in ein HTML-Format zu exportieren, welches dann, lokal oder auf einem Webserver veröffentlicht, ohne weitere Werkzeuge mit einem Webbrowser erkundet werden kann.

Nach vorheriger Aufbereitung der exportierten Softwareprozessbeschreibung ist es somit möglich, diese für die Erkundung im Stile eines *Adventures* zu verwenden. *Adventures* sind ein Spiele-Genre, in welchem Spieler aktiv ihre Spielwelt erkunden und dabei typischerweise Gegenstände und Informationen sammeln, welche sie im weiteren Spielverlauf anwenden und kombinieren, um gestellte Rätsel und Aufgaben zu lösen. Übertragen auf ein digitales Lernspiel in der SE-Ausbildung bedeutet dies, dass ein Spieler die Prozessbeschreibung erkundet und dabei Informationen und Werkzeuge sammelt, welche anschließend im Spiel verwendet werden können. Eine bestimmte Aufgabe mit einem spezifischen Werkzeug kann erst dann an eine simulierte Software-Entwicklerin übertragen werden, wenn zuvor die zugehörige Dokumentation der Aufgabe in der Prozessbeschreibung besucht und das entsprechende Werkzeug „eingesammelt" wurde.

Da die Spielumgebung die Erkundungsaktivitäten der Spieler verfolgt, erhalten auch Lehrende einen Überblick darüber, wie aktiv Einzelspieler, Teams und der gesamte Kurs im Spiel voranschreiten. So werden Verständnisprobleme frühzeitig erkannt und durch Interaktionen zwischen Lehrenden und Lernenden aufgelöst.

Durch die standardisierte Struktur der Prozessbeschreibungen fällt Studierenden die Erkundung eines zweiten oder dritten Softwareprozesses zunehmend leichter. Sie erhalten spielerisch Zugang zum Kennenlernen verschiedener Softwareprozesse in einem Industriestandard.

Effiziente Erzeugung transparenter Simulationsmodelle

Bisherige Ansätze verwenden eine *Domain Specific Language (DSL)* (Drappa, Ludewig, 2000) oder einen Regeleditor mit grafischer Oberfläche (Navarro, 2006). Die Entwicklung der Modelle erfolgt in den verfügbaren Spielumgebungen von Null an, d.h. jedes einzelne Attribut einer jeden Entität und jedes Element des Softwareprozesses muss aufs Neue modelliert werden. Auch die Steuerung der zugrundeliegenden Simulation muss über Regeln implementiert werden. Ein Konzept der Wiederverwendung existiert hierbei nicht. Erschwerend kommt hinzu, dass in den Modellen nicht zwischen dem eigentlichen Softwareprozess und seiner Anwendung in einem spezifischen Kontext (Spielszenario, Simulationsexperiment) unterschieden wird.

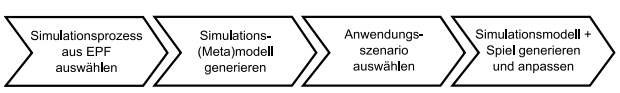


Abb. 5: Erzeugung eines Simulationsmodells

Abbildung 5 stellt einen anderen Entwurfsprozess für derartige Simulationsmodelle dar. In diesem mehrstufigen Modellierungsprozess werden in einem ersten Schritt Inhalte aus der EPF Bibliothek ausgewählt bzw. mit dem *EPF Composer* erzeugt. In einem zweiten Schritt wird daraus ein noch unvollständiges Simulations(meta)modell generiert. Dazu werden Rollen, Aufgaben, Prozessschritte, Artefakte, Leitfäden etc. aus dem EPF-Modell extrahiert. In einem dritten Schritt wird ein passendes Szenario für die Anwendung dieses Softwareprozesses ausgewählt. Dieses Szenario charakterisiert das simulierte Projektumfeld und enthält bspw. Details über verfügbare simulierte Softwareentwickler. Im vierten Schritt wird aus dem gewählten Szenario und dem teilfertigen Simulations(meta)modell ein Simulationsmodell und ein darauf basierendes Spiel generiert, welches anschließend für den konkreten Einsatz konfiguriert und angepasst werden kann. An dieser Stelle werden Parameter eingestellt oder Exkurse in Form von Minispielen in das Spiel integriert. Ziel dieses vorgestellten Entwurfsprozesses ist es, den Aufwand des Lehrenden für die Erstellung eines Simulationsmodells gegenüber bestehenden Ansätzen zu verringern. Die Tätigkeit des Modellerstellers wird sich dabei idealerweise auf das Mapping des gewählten Softwareprozessmodells auf das Anwendungsszenario und die Parametrisierung der Simulation bzw. des Spiels beschränken.

Nutzung von Werkzeugen aus dem echten Softwareentwicklerleben

Aktuelle kommerzielle Spieletitel begeistern Spieler mit fotorealistischen Darstellungen virtueller Welten. Einen Wettbewerb um Realitätsnähe auf dieser Ebene kann ein digitales Lernspiel mit viel geringerem Budget und Ressourceneinsatz nur verlieren. Wie kann ein Lernspiel jenseits solcher Dimensionen dennoch relevant und realistisch wirken?

Die Integration von Werkzeugen aus dem echten Softwareentwicklerleben in das Spielerlebnis trägt an dieser Stelle dazu bei, den realistischen Charakter des Lernspiels zu unterstreichen. Diese Werkzeuge werden durch simulierte Softwareentwickler getrieben. Hierzu ein kleines Beispiel: der simulierte Softwareentwickler Bob hat gerade seine Arbeit am Anforderungsdokument beendet. Er stößt ein Commit in der verwendeten Versionsverwaltung an und schließt anschließend das Ticket im Ticket-Verwaltungssystem – so wie in einem echten Softwareprojekt. Der Spieler sieht diese Aktivitäten anschließend in den Aktivitäten- und Repository-Sichten der jeweiligen Werkzeuge und kann sie nachvollziehen. Aktuelle Werkzeuge, wie bspw. das an der FH Stralsund eingesetzte *Redmine* (Lang, 2012), sind quelloffene Webanwendungen und verfügen über Schnittstellen für die Integrati-

on, so dass sie auch von simulierten Softwareprojekten verwendet werden können.

Studierende, welche bereits mit derartigen Werkzeugen gearbeitet haben, werden diese wiedererkennen und Parallelen herstellen können. Anderen Studierenden werden sie im Laufe ihres Studiums noch begegnen und dann schon etwas vertrauter vorkommen.

Förderung von Analyse und Reflexion

Neben der aktiven Problemlösung, kontinuierlichem Feedback und sozialer Interaktion in Teams tragen verschiedene Perspektiven zu einer konstruktivistischen Lernumgebung bei.

Das aktive Steuern der simulierten Charaktere in der Spielumgebung stellt eine erste Perspektive dar. Die gesammelten Statistiken während des Spiels liefern eine aggregierte zweite Sicht. Integrierte Werkzeuge aus dem echten Softwareentwicklerleben liefern eine dritte realitätsnahe Perspektive. Die verfügbare Beschreibung des Softwareprozesses, welche während des Spiels erkundet wird, stellt eine weitere Perspektive dar, die gleichzeitig vom gespielten Anwendungsszenario abstrahiert.

Da die Spielumgebung mit dem *Eclipse Process Framework (EPF)* auf einem frei verfügbaren Industriestandard basiert, ist es ohne weiteres möglich, Studierende im Anschluss an ein Spiel in dieses einzuführen. Mit dem EPF Composer können bspw. eigene Prozesse modelliert, analysiert und diskutiert werden, was die Reflexion und die Übertragung des erworbenen Wissens auf andere Szenarien und Kontexte fördert.

Bessere Wiederverwendbarkeit und webbasierte Architektur

Die Trennung von Spiel- und Simulationskomponente eröffnet die Möglichkeit, nur einzelne Bestandteile der Umgebung (wieder) zu verwenden. So können Spiele mit einem anderen Spielfluss oder anderen Benutzeroberflächen die gleiche Simulationskomponente nutzen.

Aktuelle Entwicklungen im Bereich der Webtechnologien *(HTML5, CSS3, WebSocket, Server-Sent Events, etc.)* ermöglichen attraktive Spielerfahrungen im Webbrowser – ganz ohne die Nutzung proprietärer Erweiterungen. Die Nutzung des Webbrowsers, in welchem sich heutige Studierende daheim fühlen, vereinfacht gleichzeitig die nahtlose Integration von webbasierten Werkzeugen aus der echten Softwareentwicklung und die Einbindung von Prozessbeschreibungen, welche über das EPF veröffentlicht wurden.

Fazit und Ausblick

Simulation und Digital Game-Based Learning (DGBL) können bei geeigneter Implementierung den Aufbau einer effizienten konstruktivistischen Lernumgebung unterstützen.

Bestehende Lösungen für die Anwendung von Simulation und Digital Game-Based Learning in der SE-Ausbildung haben Pionierarbeit geleistet und zeigen ermutigende Ergebnisse. Sie schöpfen das vorhandene Potential jedoch nicht aus. Anforderungen an die Gestaltung einer konstruktivistischen Lernumgebung sind in Teilen nicht erfüllt. Dies betrifft insbesondere die Unterstützung sozialer Interaktion, frühe Anreize für Artikulation und Reflexion und die Bereitstellung multipler Perspektiven auf den jeweiligen Softwareprozess als Lerngegenstand.

Im Forschungsprojekt *Sim4SEEd* wurden Ideen entwickelt, mit welchen vorhandenes Potential weiter ausgeschöpft wird. Diese Ideen bilden die Kernelemente einer neuen Spielumgebung, welche im Rahmen des Projekts entwickelt wird.

Sim4SEEd (www.sim4seed.org) ist ein laufendes Forschungsprojekt. Der derzeitige Fokus der Projektaktivitäten liegt auf der Konzeption einer passenden Gesamtarchitektur und der Modellierung der Simulationskomponente, welche die Einbindung von Softwareprozessen aus dem *Eclipse Process Framework (EPF)* unterstützt.

Literatur

Akilli, G. (2007): „Games and Simulations: A New Approach in Education?“. In: *Games and Simulations in Online Learning: Research and Development Frameworks*. Information Science Pub., S. 1–20.

Barros, M.O.; Dantas, A.R.; Veronese, G.O. u. a. (2006): „Model-driven game development: experience and model enhancements in software project management education“. In: *Software Process: Improvement and Practice*. 11 (4), S. 411–421.

Breuer, J. (2010): „Spielend lernen? Eine Bestandsaufnahme zum (Digital) Game-Based Learning“. Landesanstalt für Medien Nordrhein-Westfalen (LfM).

Drappa, A.; Ludewig, J. (2000): „Simulation in software engineering training“. In: *Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA: ACM (ICSE '00), S. 199–208, DOI: 10.1145/337180.337203.

En Ye; Chang Liu; Polack-Wahl, J.A. (2007): „Enhancing software engineering education using teaching aids in 3-D online virtual worlds“. In: *Proceedings of the 37th Annual Frontiers In Education Conference , FIE '07*. IEEE, S. T1E–8–T1E–13, DOI: 10.1109/FIE.2007.4417884.

Gee, J.P. (2007): What Video Games Have to Teach Us About Learning and Literacy. Palgrave Macmillan.

Hagel, G.; Mottok, J.; Ludewig, Jochen; Böttcher, Axel (Hrsg.) (2011): „Planspiel und Briefmethode für die Software Engineering Ausbildung-ein Erfahrungsbericht“. In: *SEUH 2011 Software Engineering im Unterricht der Hochschulen*.

Jain, A.; Boehm, B. (2006): „SimVBSE: Developing a game for value-based software engineering“. In: *Proceedings of the 19th Conference on Software Engineering Education and Training*., S. 103–114.

Kritzenberger, H. (2005): Multimediale und interaktive Lernräume. München :: Oldenbourg,.

Lang, J.-P. (2012): „Overview - Redmine“. Abgerufen am 14.11.2012 von http://www.redmine.org/.

Ludewig, J.; Jaeger, Ulrike; Schneider, Kurt (Hrsg.) (2009): „Erfahrungen bei der Lehre des Software Engineering“. In: *Softwareengineering im Unterricht der Hochschulen: SEUH*. 11 .

Mittermeir, R.T.; Hochmüller, E.; Bollin, A. u. a. (2003): „AMEISE – A Media Education Initiative for Software Engineering Concepts, the Environment and Initial Experiences“. In: *Proceedings of the Interactive Computer aided Learning (ICL) 2003 International Workshop*. Villach, Austria.

Navarro, E. (2006): „SimSE: A Software Engineering Simulation Environment for Software Process Education“. Irvine, CA: University of California.

Object Management Group (2008): „OMG - Software & Systems Process Engineering Meta-Model (SPEM), v2.0“. Abgerufen am 26.10.2011 von http://www.omg.org/cgi-bin/doc?formal/2008-04-01.

Pieper, J. (2012): „Learning software engineering processes through playing games“. In: *2012 2nd International Workshop on Games and Software Engineering (GAS)*. Zurich, Switzerland, S. 1 –4, DOI: 10.1109/GAS.2012.6225921.

Prensky, M. (2007): Digital game-based learning. Paragon House.

Reich, K. (2008): Konstruktivistische Didaktik: Lehr- und Studienbuch mit Methodenpool. Beltz.

Rieber, L.P. (1996): „Seriously considering play: Designing interactive learning environments based on the blending of microworlds, simulations, and games“. In: *Educational Technology Research and Development*. 44 (2), S. 43–58, DOI: 10.1007/BF02300540.

S. Egenfeldt-Nielsen (2007): „Third generation educational use of computer games“. In: *Journal of Educational Multimedia and Hypermedia*. 16 (3), S. 263–281.

Sharp, H.; Hall, P. (2000): „An interactive multimedia software house simulation for postgraduate software engineers“. In: *Software Engineering, International Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, S. 688, DOI: http://doi.ieeecomputersociety.org/10.1109/ICSE.2000.10053.

Shaw, K.; Dermoudy, J. (2005): „Engendering an empathy for software engineering“. In: *Proceedings of the 7th Australasian conference on Computing education - Volume 42*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc. (ACE '05), S. 135–144.

Sommerville, I. (2010): Software Engineering. 9th revised edition. Addison-Wesley Longman, Amsterdam.

The Eclipse Foundation (2012): „Eclipse Process Framework Project (EPF) - Home“. Abgerufen am 26.10.2011 von http://www.eclipse.org/epf/index.php.

Wangenheim, C.G. von; Shull, F. (2009): „To Game or Not to Game?“. *IEEE Software*. 26 (2), S. 92–94.

Smarter GQM-Editor mit verringerter Einstiegshürde

Raphael Pham, Kurt Schneider, Leibniz Universität Hannover
{Raphael.Pham, Kurt.Schneider}@inf.uni-hannover.de

Zusammenfassung

Die GQM Methode ist bereits ausreichend definiert. Man muss sie jedoch selbst einmal angewendet haben, um sie durchdringen zu können. Dabei können sogenannte *Abstraction Sheets* unterstützend eingesetzt werden. In der Lehre an der Leibniz Universität Hannover hat sich gezeigt, dass das Ausfüllen dieser Formulare vielen Studenten Probleme bereitet. Sie sind von Abstraction Sheets überfordert oder erkennen den Sinn darin nicht. Als Folge lehnen die Studenten die GQM Methode ab und erlernen sie nicht. Wir stellen einen smarten Editor vor, welcher konkret die Erstellung des Abstraction Sheets und die Entwicklung von zugehörigen Metriken unterstützt. Der Anwender wird mit prozess-gesteuerten Fragestellungen zum korrekten Ausfüllen beider Formulare angeregt. Unerfahrene Anwender können den vollen Unterstützungsumfang des smarten Editors ausschöpfen und so an die GQM Methode herangeführt werden. Erfahrene Benutzer können den Editor verwenden, um Abstraction Sheets, Questions und Metriken zu erstellen und zu verwalten.

Motivation

Ein Fallstrick beim Messen von Software Entwicklungsprozessen oder Produkten ist, dass man das misst, was einfach zu messen ist - und nicht das, was man messen sollte, um aussagekräftige Ergebnisse zu erhalten (Basili, 1992). Man wendet sich an bereits bekannte oder einfach zu erhebende Metriken. Hier verwendet die GQM Methode einen Top-Down Ansatz: Anstatt mittels vorhandener Werkzeuge (z.B. bekannte Metriken) projekt-individuelle Ziele zu messen (Bottom-Up), werden diese Ziele konkretisiert und durch eigens entwickelte Metriken operationalisiert. Dies soll gewährleisten, dass tatsächlich gemessen wird, was auch gemessen werden soll - und auch nicht mehr.

Grundlegende Ideen zu GQM entstanden während einer Untersuchung am Goddard Space Flight Center (Basili u. Weiss, 1984). Später stellt (Basili, 1992) das GQM Paradigma vor und (Van Latum u. a., 1998) erweitern die GQM Methode um sogenannte *Abstraction Sheets*. Seit seiner Einführung konnte die GQM Methode erfolgreich in verschiedenen Industrieprojekten eingesetzt werden. (Van Latum u. a., 1998) und (Birk u. a., 1998) beschreiben den Einsatz bei Schlumberger RPS. (Gantner u. Schneider, 2003) haben den GQM-Prozess in zwei Fällen aus der Automobilindustrie angewendet und beschreiben eingehend den Einsatz von Abstraction Sheets. (Kilpi, 2001) setzt eine angepasste GQM Variante bei Nokia ein. (Van Solingen u. Berghout, 1999) bietet eine eingehende Betrachtung von GQM samt Fallbeispielen.

Dem Erfolg von GQM in der Praxis folgend, wird die GQM Methode auch an der Leibniz Universität Hannover gelehrt. Dem Thema GQM ist ein Kapitel der Vorlesung zur Software-Qualität gewidmet und es wird auch im dazugehörigen Übungsbetrieb behandelt. Erfahrungsgemäß lässt sich ein besserer Lernerfolg erzielen, wenn GQM praktisch angewendet wird. Das Konzept von Abstraction Sheets und deren korrekte Erstellung bereitet den Studierenden jedoch öfter Schwierigkeiten - obwohl der Prozess zum Erstellen eines Abstraction Sheets klar vorgegeben ist. Studierende zeigten wiederholt Schwierigkeiten Abstraction Sheets korrekt auszufüllen. Das mühsame Aufzeichnen der Formulare per Hand könnte eine zusätzlich Hürde darstellen. In dieser Arbeit stellen wir einen Editor für Abstraction Sheets vor, der Studierende mit Hilfsmechanismen aktiv beim Anwenden der GQM Methode unterstützt. Der unsichere Anwender wird mit gezielten Fragestellungen durch den Ausfüllprozess geführt. Studierenden soll so eine mögliche Ablehnung gegenüber der komplex anmutenden GQM Methode genommen werden. Studierende sollen mit diesem Werkzeug Gelegenheit erhalten, das korrekte Ausfüllen von Abstraction Sheets zu üben. (Werner, 2012) hat die in dieser Arbeit vorgestellten Konzepte in einem lauffähigen Webtool realisiert.

Die GQM Methode

Die GQM Methode wurde bereits in zahlreichen Veröffentlichungen behandelt. Im folgenden wird ein kurzer Überblick über die GQM Methode gegeben (in der Form, in welcher diese an der Leibniz Universität Hannover gelehrt wird (Schneider, 2007)), um den Wirkungsbereich unseres GQM-Editors aufzuzeigen.

Der grundlegende Ablauf der GQM Methode sieht die schrittweise Erstellung eines GQM-Modells (auch GQM-Baum genannt) vor: Abstraktere Ziele werden

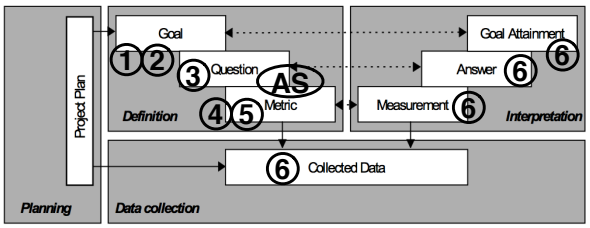


Abbildung 1: GQM Methode an der Leibniz Universität Hannover, basierend auf (Van Solingen u. Berghout, 1999)

in konkrete Subziele (Goal) aufgeteilt. Subziele werden durch beschreibende Fragen (Question) scharf und zweckmäßig umrissen. Zur Beantwortung der charakterisierenden Fragen werden Metriken (Metric) erstellt, die den Erfüllungsgrad der Fragen und somit den Erreichungsgrad des Ziels angeben. Abb. 2 zeigt einen solches GQM-Modell.

Zu Lehrzwecken spalten wir Teile der GQM Methode auf und definieren sechs Schritte:

1. Ziele erheben und verfeinern (Goal)
2. Facettenbeschreibung der Ziele (Goal)
3. Ableitung von charakterisierenden Fragen (Question) zu den Zielen
4. Ableitung von zugehörigen Metriken, (Metric)
5. Messplan für eigentliche Datenerhebung erstellen (Metric)
6. Datenerhebung und Auswertung (Collection, Interpretation)

Die Einordnung dieser Schritte in das umfangreiche Prozessmodell der GQM Methode nach (Van Solingen u. Berghout, 1999) ist in Abb. 1 eingezeichnet.

Schritt 1 produziert einen sogenannten Zielbaum, welcher ein high-level Messziel auf konkretere Subziele verfeinert und in Beziehung miteinander setzt. Aus diesem Zielbaum werden Subziele zur weiteren Bearbeitung ausgewählt. Die Facettendarstellung des Messziels (Schritt 2) forciert die Auseinandersetzung mit verschiedenen Aspekten eines Ziels und regt zur abgewandelten Betrachtung an.

Für die Ableitung von charakterisierenden Fragen zu einem Messziel (nun vorliegend als Zielfacette, Schritt 3) wird ein spezielles Formular eingesetzt, das sogenannte *Abstraction Sheets* (Van Latum u. a., 1998), (Van Solingen u. Berghout, 1999), (Gantner u. Schneider, 2003). Es bietet eine festgelegte Struktur. Mittels vier vordefinierten Fragestellungen baut der Anwender systematisch ein Modell des zu untersuchenden Messziels auf und gelangt zu tiefergreifendem Verständnis. Der Qualitätsaspekt des Betrachtungsgegenstandes (z.B. "Verständlichkeit"von "Quellcode") wird auf konkrete und projektindividuelle Eigenschaften heruntergebrochen. Diese heißen *Qualitätsfaktoren* des Qualitätsaspektes. Ziel ist es, für jeden konkreten Aspekt des Qualitätsaspekts einen beeinflussenden Faktor zu finden (sogenannte *Einflussfaktoren*). Beide gefundenen Faktoren sind über eine Einflusshypothese zu verbinden. Nun ergibt sich eine das Messziel charakterisierende Frage (Question) als Zustandsabfrage von Qualitätsfaktoren.

Die aufgestellte Einflusshypothese wird in Schritt 4 und 5 überprüft. Dabei wird eine Metrik zur Zustandsabfrage des Qualitätsfaktors basierend auf dem Zustand des Einflussfaktoren entwickelt: Es werden verschiedene (mögliche) Ausprägungen für den Einflussfaktoren ermittelt und jeweils der Einfluss auf den Qualitätsfaktor gemessen. Für diesen Vorgang wird ein eigenes Tabellentemplate verwendet.

Die einzelnen Prozessschritte samt Formularen skizziert Abb. 2. Eingehende Betrachtungen der GQM Methode finden sich in (Schneider, 2007) oder auch (Van Solingen u. Berghout, 1999).

Der GQM-Editor nimmt ein Facettenziel entgegen und unterstützt maßgeblich die Erstellung eines zugehörigen Abstraction Sheets und zugehöriger Messpläne (Schritt 3 und Schritt 4). Die Erstellung und Verfeinerung von Messzielen oder deren Darstellung als Facette werden nicht unterstützt (Schritt 1 und Schritt 2). Beide Schritte lassen sich relativ schnell auf Papier bewältigen und erfordern (gemäß Lehrerfahrungen) weniger Unterstützung. Auch die eigentliche Durchführung und anschließende Interpretation werden nicht von unserem GQM-Editor unterstützt.

Das Abstraction Sheet

Das Abstraction Sheet (AS) bietet dem Anwender eine streng definierte Struktur um methodisch und zielsicher charakterisierende Fragen zu einem gegebenen Messziel abzuleiten. Dabei ist die Ausfüllreihenfolge der Quadranten vorgegeben und bildet ein U. Jeder Quadrant verlangt dem Anwender ab, bestimmte Aspekte des Messziels anzugeben. Auf diese Weise wird der abstrakte Qualitätsaspekt des Messziels in Qualitätsfaktoren konkretisiert, beeinflussende Faktoren (Einflussfaktoren) identifiziert und zugehörige Einflusshypothesen aufgestellt. Begonnen wird ein AS mit dem Eintragen des Messziels in Facettenform (oberer Teil von 3), welche aus dem vorigen Prozessschritt 2 übernommen wurde. Der Ablauf im Detail:

1. Oben, links: Der Qualitätsaspekt wird auf konkrete Eigenschaften des Betrachtungsgegenstandes konkretisiert, sogenannte *Qualitätsfaktoren*. Welche Merkmale machen den Qualitätsaspekt in diesem Projekt ganz konkret aus? In Fall von Abb. 3: Verständlichkeit von Quellcode bedeutet für den Ausfüllenden z.B. die Verzweigungstiefe.

2. Unten, links: Zu jedem Qualitätsfaktor wird eine *Ausgangshypothese* aufgestellt, wie es zu Messbe-

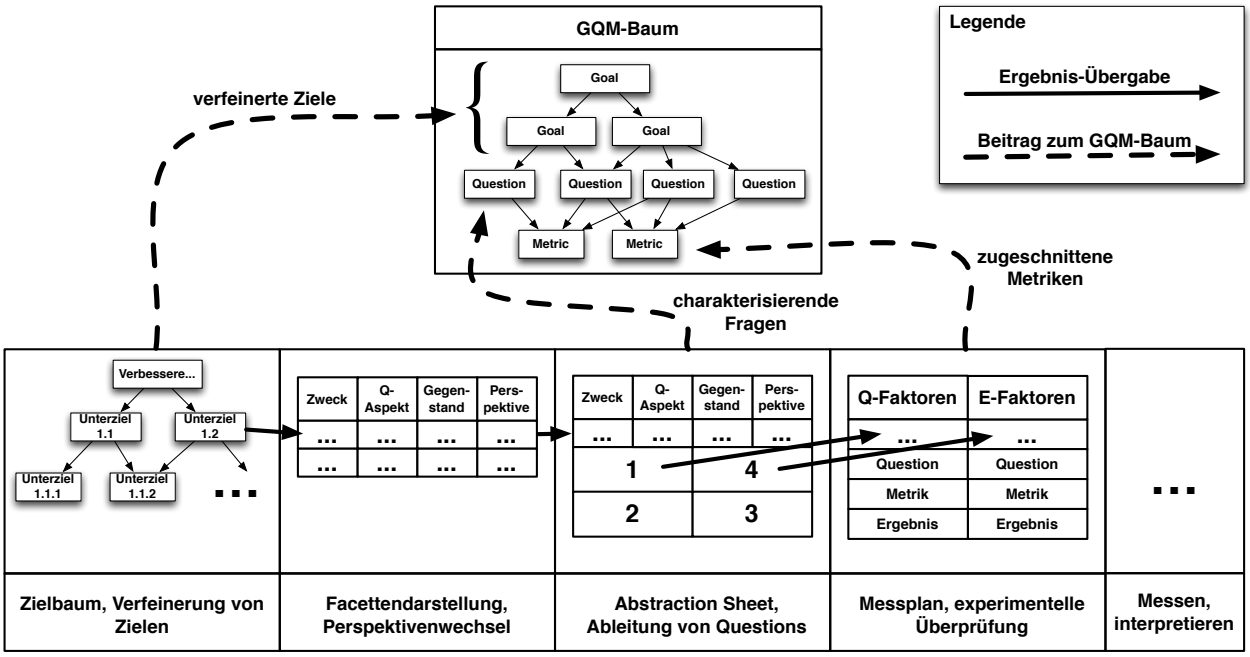


Abbildung 2: Übersicht über die GQM Methode, Einordnung der Prozessschritte und Formulare

ginn darum steht. Alternativ können auch nachgemessene Werte eingetragen werden.

3. Unten, rechts: Der Anwender stellt *Einflusshypothesen* für jeden Qualitätsfaktor auf: Wie könnte der Qualitätsfaktor zu beeinflussen sein?

4. Oben, rechts: Aus den Einflusshypothesen werden *Einflussfaktoren* extrahiert.

Abb. 3 zeigt ein ausgefülltes AS. Obwohl die Anforderungen an den Inhalt eines Quadranten klar definiert sind, bereitet es Studenten öfter Schwierigkeiten, korrekte und brauchbare AS zu produzieren.

Auf einen Studierenden, der das Konzept des Abstraction Sheets noch nicht vollständig durchdrungen hat, kann dieses Formular abschreckend und komplex wirken. Zählt man das Eintragen des Messziels mit, so werden mehr als vier verschiedene Abfragen an den Studierenden gestellt. Hinzu kommt, dass diese Felder konzeptbedingt jeweils eng gefasste Ziele verfolgen und korrekte Antworten für die erfolgreiche Anwendung voraussetzen: Werden z.B. im ersten Quadranten die Qualitätsfaktoren nicht mit konkretisierenden Attributen belegt — sondern bleiben so abstrakt wie das Messziel — so führt das Abstraction Sheet nicht zu dem gewünschten Ergebnis. In Abb. 3 würde der ungeübte Studierende als ersten Qualitätsaspekt möglicherweise einfach ein Synonym für das Messziel eintragen, z.B. Verständnis. Damit ist in diesem Schritt jedoch nichts gewonnen. Der Studierende füllt die nachfolgenden Felder aus — und kommt dennoch nicht zum erwarteten Ergebnis und ist enttäuscht. Damit dieser Fall nicht eintritt, bietet unser Editor dem Studierenden gezielte Hilfestellung. Durch kontextabhängige, konkret formulierte Fragestellungen wird der Anwender durch das Abstraction Sheet geführt.

Auch wenn der Hauptgrund der nicht-erfolgreichen Anwendung des Abstraction Sheets auf Unverständnis des zugrundeliegenden Konzeptes zurückzuführen ist, so kann eine derart geführte (wiederholte) Anwendung zur Durchdringung des Konzeptes führen bzw. beitragen.

Der smarte GQM-Editor

Die Inhalte der Quadranten 1 bis 4 sind abhängig von einander: Inhalte werden von Quadrant zu Quadrant weitergegeben und verarbeitet. Z.B.: Hat der Anwender in Quadrant 1 das Qualitätsziel auf zwei konkrete Faktoren heruntergebrochen, so tauchen diese beiden Faktoren mit einer zusätzlichen Statuseinschätzung (genauer: Ausgangshypothese) im zweiten Quadranten wieder auf. Dieses Beispiel zeigt zwei Ebenen, auf denen der Anwender agieren muss, um das AS auszufüllen: Zum einen muss er einen kreativen Schritt tätigen und das Q-Ziel konkretisieren - dabei muss er genau wissen, was gerade von ihm verlangt wird. Zum anderen muss er die Einträge manuell in den nächsten Schritt übertragen und dort weiterverarbeiten. An diesen Stellen setzt unser GQM-Editor an: Der Anwender wird bei einem kreativen Vorgang durch eine gezielte Fragestellung als dezenter Hinweistext unterstützt (*Grauer Hilfstext*). Weiterhin übernimmt der GQM-Editor manuelle Schritte, überträgt Zwischenergebnisse in Folge-Quadranten und versucht damit die nächsten Einträge teilweise zu generieren (*Automatischer Hilfstext*). Beide Unterstützungsvarianten

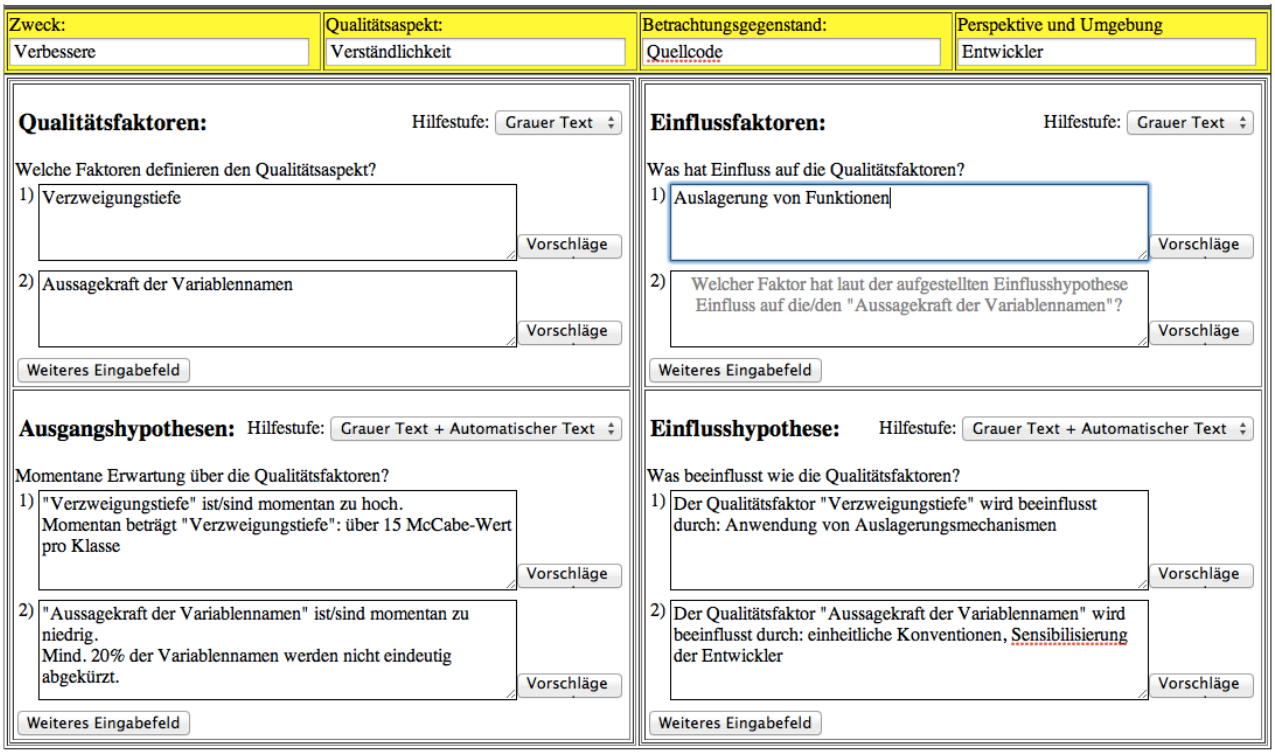


Abbildung 3: Ein ausgefülltes Abstraction Sheet im GQM-Editor

werden dynamisch anhand von vorhergegangenen Eingaben im AS generiert. Auf diese Weise lassen sie eine kontextbezogene Unterstützung zu. Dies ist eine der Stärken des GQM-Editors von denen besonders unerfahrene Anwender profitieren können. Beide Unterstützungstexte werden als vorgeschlagene Antworteinträge in den Quadranten eingeblendet — die der Anwender nach Belieben ändern und anpassen kann.

Für erfahrene Benutzer können die beiden Unterstützungsmechanismen ausgeschaltet werden, sodass der GQM-Editor als reiner Editor fungiert. Alle drei Mechanismen (keine Hilfe, grauer Hilfstext, automatischer Hilfstext) können für jeden Quadranten individuell eingestellt werden (siehe Dropboxen in den Quadranten, Abb. 3).

Die Hilfstexte haben eine weitere Funktion: Sie erscheinen in der Reihenfolge, in der das AS ausgefüllt werden sollte. Jeweils der nächst-auszufüllende Quadrant wird mit grauem oder schwarzem Hilfstext gefüllt. Dies soll den Anwender durch den Ausfüllprozess leiten. Da es sich bei den Hilfstexten nur um Vorschläge handelt, kann der Anwender aber auch gegen diese Reihenfolge verstoßen und z.B. die voreingetragenen Werte entfernen.

Grauer Hilfstext

Das Einblenden von grauem Hilfstext ist die erste Unterstützungsstufe unseres GQM-Editors und für unerfahrene Anwender gedacht. Der Hilfstext soll dem Anwender an geeigneter Stelle ins Gedächtnis rufen, was in dem aktuellen Quadranten einzutragen ist. Er

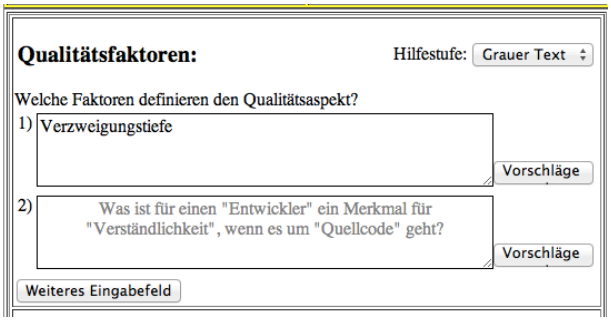


Abbildung 4: Hilfestellung im Quadrant 1

dient als Gedankenstütze, um die Aufgabe klar zu definieren: Der Anwender wird explizit — und angepasst an vorherige Eingaben — nach dem einzutragenden Inhalt gefragt. So erscheint der graue Hilfstext im ersten Quadranten, sobald der Anwender das Messziel als Zielfacette eingetragen hat. Die eingetragenen Werte werden dynamisch im Text verarbeitet und sind kontextualisiert: Anhand der eingegeben Zielfacette in Abb. 3 (gelb unterlegt) generiert der Editor den grauen Hilfstext: “Was ist für einen Entwickler ein Merkmal für Verständlichkeit, wenn es um Quellcode geht?” (siehe Abb. 4).

Sobald der Anwender auf ein Feld mit grauem Hilfstext klickt und einen Eintrag vornehmen will, verschwindet dieser und das Textfeld ist für den Anwender freigegeben. Die graue Farbe wurde bewusst dem Anschein eines flüchtigen Tooltipps nachempfunden.

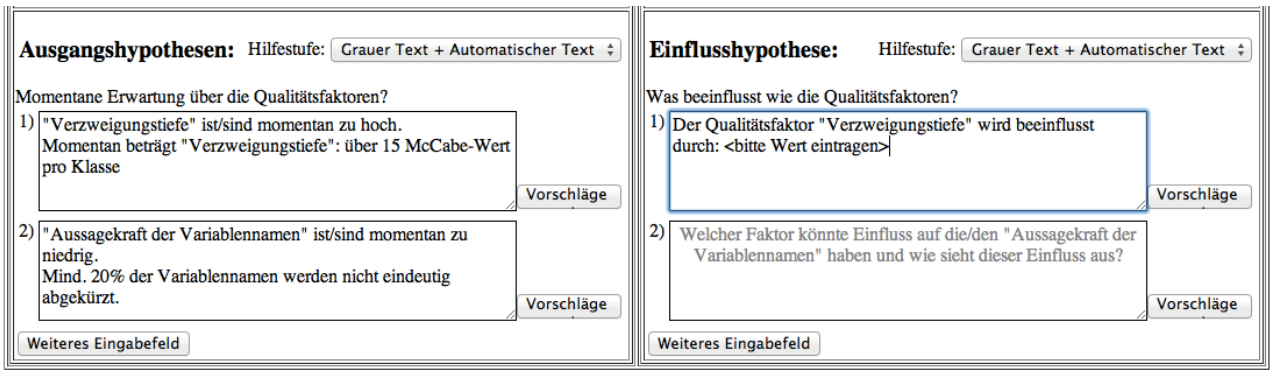


Abbildung 5: Hilfestellung im Quadranten 2 und 3

Automatischer Hilfstext

Die zweite Unterstützungsstufe wendet sich an sehr unerfahrene Anwender: Der GQM-Editor generiert Teile des Eintrags als schwarzen Text vor und lässt gezielt Lücken, die der Anwender ausfüllen soll. Dies lenkt die Gedanken der Anwender in gezielte Bahnen und es werden konkrete Werte verlangt.

Das Ausfüllen von Quadrant 1 bedarf eines kreativen Schrittes, der durch den grauen Hilfstext unterstützt wird. Teile von Antworten zu generieren ist hier nicht möglich. Quadrant 2 (Ausgangshypothese) bietet jedoch die Möglichkeit, die Q-Faktoren automatisch in die rudimentäre Form einer Hypothese zu überführen. Dabei kann der Aussagenkern der Hypothese nicht generiert werden, da dies wieder eines kreativen Schrittes bedarf. Jedoch wird das Eintragen dieses Aussagenkerns durch einen gezielten Lückentext unterstützt:

Bevor der Anwender in das Feld zum Eintragen der ersten Ausgangshypothese klickt, hat der Editor dort in grau eingetragen: “Wie steht es momentan um die/den ’Verzweigungstiefe’? Welche Erfahrungswerte, Vermutungen und/oder Messwerte gibt es zu diesem Qualitätsfaktor?” Dies erinnert den Anwender an die von ihm geforderte Ausgangshypothese. Sobald er in das Feld klickt, erscheint die folgende vorgenerierte Antwort in schwarzem und editierbarem Text: “Verzweigungstiefe ist/sind momentan zu <bitte Eigenschaft eintragen>. Momentan beträgt Verzweigungstiefe: <bitte Wert eintragen>.” Der Benutzer kann diese Vorlage verwenden, um seine Ausgangshypothese zu verfassen oder eine eigene formulieren (s. Abb. 5).

Ähnlich lässt sich mit Quadrant 3 verfahren (siehe Abb. 5), jedoch nicht mit Quadrant 4: Die Einflussfaktoren zuverlässig aus den resultierenden Einflusshypothesen zu extrahieren, ist aufgrund von sprachlichen Abweichungen und Eigenarten schwierig.

Letztendlich lässt sich diese zweite Unterstützungsstufe nur für den zweiten Quadranten (Ausgangshypothese) und dritten Quadranten (Einflusshypothese) einschalten. Quadranten 1 und 4 können lediglich die erste Unterstützungsstufe einsetzen.

Da möglicherweise die generierte Antwort nicht zu dem aktuellen Messprojekt passt, lässt sich diese generierte Antwort wie ein normales Textfeld bearbeiten und auch löschen. In Abb. 5 hat der Anwender die zweite Ausgangshypothese angepasst.

Der Messplan

Nachdem der Anwender mithilfe des Abstraction Sheets Einflussfaktoren der Qualitätsfaktoren und zugehörige Einflusshypothesen aufgestellt hat, folgt der Schritt der experimentellen Überprüfung eines solchen Zusammenhangs. Der sogenannte *Messplan* fordert den Anwender auf, verschiedene Ausprägungen des Einflussfaktors zu ermitteln und dann zu überprüfen, wie sich der Qualitätsfaktor verhält. Auf diese Weise soll die Einflusshypothese validiert bzw. falsche Annahmen aufgedeckt werden.

Die beiden Faktoren werden automatisch aus dem Abstraction Sheet übernommen und bleiben an der selben Stelle stehen. Dies soll Verwirrungen beim unerfahrenen Anwender vermeiden. Bedingt durch die Idee der experimentellen Überprüfung muss der Anwender mit der rechten Seite des Messplans beginnen und zuerst den Einflussfaktor betrachten. Da dies etwas unintuitiv erscheinen mag, wird standardmäßig die linke Seite mit einem Hinweis ausgeblendet (siehe Abb. 6). Persönliche Workflows können auch realisiert werden; der Hinweis lässt sich ausschalten.

Jede Seite des Messplans gliedert sich in drei Teile, die nacheinander ausgefüllt werden.

1. Frage nach den möglichen Ausprägungen des Faktors.
2. Handlungsanweisungen, diese Ausprägungen zu ermitteln.
3. Ausprägungen ermitteln und eintragen.

Die Formulierung der einleitenden Frage nach den Ausprägungen wird mit grauem Hilfstext bzw. schwarzem Lückentext unterstützt (siehe Abb. 6, beide Texte in den Feldern “Frage” sind generiert). Jedoch sind Schritt 2 und 3 stark projektabhängig und individuell. Dementsprechend kann der GQM-Editor diese nicht

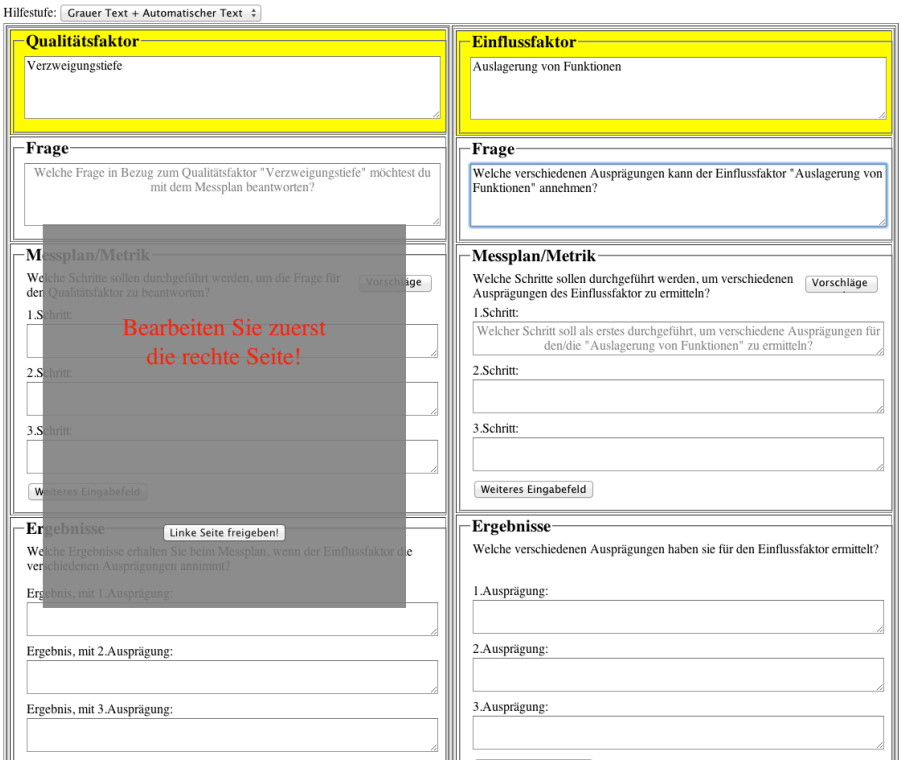


Abbildung 6: Linke Seite des Messplans wird zu Beginn ausgeblendet.

mit automatisch generierten Antworten unterstützen, bietet jedoch hier die erste Hilfestufe an (grauer Text als Gedankenanstoß). Weiterhin gibt der graue Hinweistext dem unerfahrenen Benutzer die Reihenfolge zum Ausfüllen vor. Jede Eingetragene Ausprägung für den Einflussfaktor ist einem Eintragungsfeld für eine Ausprägung gegenübergestellt. Auf diese Weise soll dem Anwender die Gegenüberstellung und der Effekt der verschiedenen Faktor-Ausprägungen vereinfacht werden.

Verwandte Arbeiten

Zur Unterstützung der GQM Methode wurden bereits einige Tools entwickelt. Jedoch ist keines der untersuchten Tools fokussiert auf unerfahrene Benutzer oder bietet Hilfestellung beim eigentlichen Ausfüllen von Abstraction Sheets und Messplänen.

Die Enwicklung von GQMaspect wurde an der Universität Kaiserslautern begonnen und später am Fraunhofer IESE (Institut für experimentelles Software Engineering) weitergeführt (Hoffmann u. a., 1997), (Voigtlaender, 1999). GQMaspect unterstützt die gesamte GQM Methode: Aufnahme und Verwaltung der Goals, Erstellen von Abstraction Sheet mittels Editor bis hin zur Verwaltung von Messplänen. Mit GQMaspect lassen sich Beziehungen zwischen einzelnen Produkten der Arbeitsphasen der GQM Methode herstellen (Ein Goal wird einem AS zugewiesen, etc) und es können Vollständigkeitsabfragen gemacht werden (wurde jedes definierte Goal mit einem AS weiterverfolgt?). Der Ansatz unseres GQM-Editors unterstützt hier lediglich die Erstellung von Abstraction Sheets und Messplänen — Goals werden nicht nennenswert behandelt. Im Gegensatz zu unserem GQM-Editor bietet GQMaspect dem Anwender jedoch wenig Unterstützung, wenn es um das eigentliche Erstellen von neuen GQM-Inhalten geht. Anwendergruppe von GQMaspect sind erfahrene GQM-User, während GQM-Editor für den Lehrgebrauch bzw. für unerfahrene Anwender konstruiert wurde.

GQM–DIVA (GQM–Definition Interpretation Validation) wurde im Rahmen einer Diplomarbeit an der Universität Kaiserslautern entwickelt (van Maris u. a., 1995). Das Tool überprüft die Konsistenz der Einträge eines GQM-Baumes (hier genannt: GQM-Plan), damit keine fehlerhaften Daten weiterverwendet werden. Dieser GQM-Baum kann mit fiktiven Messwerten in Simulation ausgewertet werden. Damit sollen Lücken (fehlende Daten) im GQM-Baum aufgedeckt werden. GQM-Diva benötigt ein fertig ausgefülltes Abstraction Sheet als Eingabe. Dessen Erstellung wird nicht unterstützt. Auch die Erstellung eines Messplans wird von GQM-DIVA nicht abgedeckt.

(Chen u. a., 2003) stellen mit dem Multi-Agent System ISMS (Intelligent Software Measurement System) eine Vision eines intelligenten Wizzards zur Durchführung der GQM Methode vor. Der Anwender gibt ein Unternehmensziel ein und beantwortet interaktive Fragen. Schrittweise werden verfeinerte Messziele, individuelle Metriken, etc. anhand der Anwenderantworten auf intelligente Weise herausgeschält. Dies soll mithilfe von eigenen Agents und einer angeschlos-

senen Knowledge Base samt Reasoning Mechanism geschehen. Am Ende soll ein fertiger und einsatzbereiter Messplan ausgegeben werden. Der Ansatz ist interessant, scheint jedoch noch nicht praktisch umgesetzt zu sein. Auch werden Abstraction Sheets nicht betrachtet.

MetriFlame ist von VTT Electronics entwickelt worden und dient zur Sammlung, Analyse und Präsentation von Messdaten. Die Grundidee ist, dass dieses Tool automatisch Daten für eine anstehende Messung sammelt, indem es sich mit verschiedenen Datenbanken anderer Tools verbinden kann. (Parviainen u. a., 1997) setzte MetriFlame bei der Andererwendung der GQM Methode ein, um passende Metriken zu finden und Daten zu sammeln. MetriFlame erlaubt die Generierung von GQM Bäumen auf Basis von bereits gespeicherten Metriken. Das Aufstellen von Abstraction Sheets oder Messplänen wird nicht unterstützt.

Das GQM-Tool von (Lavazza, 2000) bietet die Möglichkeit GQM Bäume zu erstellen. Deren Struktur wird auf Konsistenz überprüft, indem z.B. kontrolliert wird, ob jedes Messziel zu mindestens einer Frage verfeinert wurde und es zu jeder Frage mindestens eine Metrik gibt. Messziele, Abstraction Sheets, Questions und Metriken werden als Produkte der GQM Methode unterstützt. Einzelne Komponenten von GQM-Plänen können wiederverwendet werden. Das GQM-Tool ist mit einer Datenbank verbunden. Auch dieses Tool bietet keine Hilfestellungen an, die erklären, wie ein Abstraction-Sheet auszufüllen ist.

Diskussion

Der smarte GQM-Editor verwendet Texteingaben des Anwenders, um nächst-auszufüllende Felder mit kontextualisierten Hilfstexten oder sogar vorgenerierten Antworten zu bestücken. Die Konstruktion der grauen und schwarzen Hilfstexte basiert auf Textmustern, welche wir durch wiederholtes Bearbeiten und Lehren von Abstraction Sheets in der Lehre erkannt haben. Grob gesagt hat das begleitete Erklären von Abstraction Sheets anhand solch gearteter Kommentare den höchsten Lernerfolg bei Studenten erzielt.

Der smarte GQM-Editor setzt eine einfache Textersetzung nach diesen Regeln ein. Dieser Mechanismus ist zuverlässiger, je eindeutiger die Eingabetexte sind. Z.B. lässt sich anhand der starren Form der Messziel-Facette des Abstraction Sheets (vier Eingabefelder, jeweils meist nur ein Wort) ein relativ passender und hilfreicher grauer Hinweistext im ersten Quadranten erzeugen (siehe Abb. 4). Je variabler die Eingabetexte werden, desto unzuverlässiger kann der generierte Text im nächsten Feld ausfallen. Dies kann dazu führen, dass der generierte Text unbrauchbar wird. Graue Hilfstexte verschwinden jedoch auch beim Anklicken und vorgenerierte Antworten (schwarzer Text) können einfach anwenderseitig bearbeitet oder gelöscht werden. Werden diese Textstücke generell zu verwirrend, kann der Anwender diese Unterstützung auch insgesamt ausschalten.

Die Feinheiten der deutschen Grammatik (männliche, weibliche Artikel, Plural, Singular) werden nicht gesondert behandelt. Die generierten Texte sind in dieser Hinsicht möglichst allgemein ausgelegt und verwenden mitunter unschöne Doppelbelegungen wie z.B. "die/den". Ein Plugin zur korrekten Lösung solcher Fälle lässt sich jedoch nachträglich einfügen. Generell priorisieren wir den Lerneffekt beim unerfahrenen Anwender gegenüber vollkommen korrekter Sprachkonstrukte. Mögliche sprachliche Fehler oder Ungenauigkeiten werden hingenommen, solange der Anwender versteht, was gemeint ist bzw. was gerade von ihm verlangt wird.

Der smarte GQM-Editor stellt keine Silver-Bullet für die GQM Methode oder das Erstellen von Abstraction Sheets dar. Der Anwender soll jedoch verstehen, was von ihm verlangt wird und wie solche Formulare gehandhabt werden. Der smarte GQM-Editor “hilft dabei auf die Sprünge” — kreative und korrektive Arbeitsschritte seitens des Anwenders werden dabei aber nicht ausgeschlossen.

Die praktische Nützlichkeit der hier vorgestellten Konzepte konnte nicht im universitären Umfeld evaluiert und validiert werden. Eine umfangreiche Evaluation ist bereits in Planung. Da diese Konzepte aber auf praktischer Lehrerfahrung basieren, sind wir zuversichtlich, dass Studierende davon profitieren werden.

Zusammenfassung und Ausblick

Die GQM Methode wurde seit seiner Vorstellung durch (Basili, 1992) erfolgreich in Industrieprojekten eingesetzt und hat die Entwicklung von verschiedenen Werkzeugen angetrieben. Lehrerfahrung an der Leibniz Universität Hannover zeigt jedoch, dass die GQM Methode bei Studierenden nur Akzeptanz findet, wenn sie praktisch ausprobiert und eingesetzt wird. Jedoch schrecken die komplex anmutenden Formulare wie Abstraction Sheet oder Messplan Studierende ab.

In dieser Arbeit wird ein lauffähiger smarter GQM-Editor vorgestellt. Er bietet eine webbasiert Editor-Oberfläche für die beiden Formulare Abstraction Sheet und Messplan und richtet sich in erster Linie an unerfahrene GQM-Anwender (wie z.B. Studierende).

Der GQM-Editor verarbeitet vorhergegangene Anwendereingaben, um kontextualisierte Hinweistexte an passenden Stellen einzublenden. Z.B. wird der Anwender in jedem Quadrant eines Abstraction Sheets explizit und individualisiert nach den einzutragenden Inhalten gefragt. Diese Hinweistexte sollen als Denkanstoß dienen und dem unsicheren Studierendem helfen, diese Formulare zu verstehen.

Zusätzlich kann der smarte GQM-Editor Antwortteile im Abstraction Sheet auf Wunsch anhand vorheriger Eingaben generieren. Antwortteile, die nicht automatisch generiert werden können, werden durch

Lückentexte repräsentiert. Diese sollen den Anwender anregen, passende Antworten einzutragen.

Die generierten Texte erscheinen in der angedachten Ausfüllreihenfolge. Sie können vom Nutzer angepasst, gelöscht bzw. ausgeschaltet werden. Die erstellten Abstraction Sheets und Messpläne können vom Anwender gespeichert und verwaltet werden.

Eine umfangreiche Evaluierung des vorgestellten smarte GQM-Editor in der universitären Lehre wird vorbereitet. Dabei soll die Akzeptanz der GQM Methode und der zugehörige Wissensgewinn bei Studierenden gemessen werden.

Die Konzepte des smarten GQM-Editors sind als Initiative zu verstehen, komplizierte Techniken besser zu erlernen und vermitteln. Tool-basiert kann die SE-Lehre damit unterstützt werden.

Literatur

[Basili 1992] BASILI, Victor R.: *Software modeling and measurement: the goal/question/metric paradigm*. College Park, MD : Maryland Univ., 1992 (Computer science technical report Series)

[Basili u. Weiss 1984] BASILI, Victor R. ; WEISS, David M.: A Methodology for Collecting Valid Software Engineering Data. In: *Software Engineering, IEEE Transactions on* SE-10 (1984), nov., Nr. 6, S. 728 –738. http://dx.doi.org/10.1109/TSE.1984.5010301. – DOI 10.1109/TSE.1984.5010301. – ISSN 0098–5589

[Birk u. a. 1998] BIRK, A. ; SOLINGEN, R. van ; JARVINEN, J.: Business impact, benefit, and cost of applying GQM in industry: an in-depth, long-term investigation at Schlumberger RPS. In: *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, 1998, S. 93 –96

[Chen u. a. 2003] CHEN, T. ; FAR, B.H. ; WANG, Y.: Development of an intelligent agent-based GQM software measurement system. In: *Proceedings of ATS*, 2003, S. 188

[Gantner u. Schneider 2003] GANTNER, Thomas ; SCHNEIDER, Kurt: Zwei Anwendungen von GQM: Ähnlich, aber doch nicht gleich. In: *Metrikon 2003* (2003)

[Hoffmann u. a. 1997] HOFFMANN, M. ; BIRK, A. ; ELS, F. van ; KEMPKENS, R.: *GQMaspect V1. 0: User Manual*. Fraunhofer-IESE, 1997

[Kilpi 2001] KILPI, T.: Implementing a software metrics program at Nokia. In: *Software, IEEE* 18 (2001), nov/dec, Nr. 6, S. 72 –77. http://dx.doi.org/10.1109/52.965808. – DOI 10.1109/52.965808. – ISSN 0740–7459

[Lavazza 2000] LAVAZZA, L.: Providing automated support for the GQM measurement process. In: *Software, IEEE* 17 (2000), may/jun, Nr. 3, S. 56 –62. http://dx.doi.org/10.1109/52.896250. – DOI 10.1109/52.896250. – ISSN 0740–7459

[van Maris u. a. 1995] MARIS, M. van ; DIFFERDING, D.I.C. ; GIESE, D.I.P.: Ein Werkzeug zur Definition, Interpretation und Validation von GQM-Planen. (1995)

[Parviainen u. a. 1997] PARVIAINEN, P. ; JARVINEN, J. ; SANDELIN, T.: Practical experiences of tool support in a GQM-based measurement programme. In: *Software Quality Journal* 6 (1997), Nr. 4, S. 283–294

[Schneider 2007] SCHNEIDER, Kurt: *Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. Dpunkt, 2007

[Van Latum u. a. 1998] VAN LATUM, F. ; VAN SOLINGEN, R. ; OIVO, M. ; HOISL, B. ; ROMBACH, D. ; RUHE, G.: Adopting GQM based measurement in an industrial environment. In: *Software, IEEE* 15 (1998), jan/feb, Nr. 1, S. 78 –86. http://dx.doi.org/10.1109/52.646887. – DOI 10.1109/52.646887. – ISSN 0740–7459

[Van Solingen u. Berghout 1999] VAN SOLINGEN, R. ; BERGHOUT, E.: *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. McGraw-Hill, 1999

[Voigtlaender 1999] VOIGTLAENDER, und Kempkens R. C.: *GQMaspect II. System Documentation*. Fraunhofer-IESE, 1999

[Werner 2012] WERNER, Florian: *Entwurf eines regelbasierten GQM-Editors mit geringer Einstiegshürde*, Leibniz Universität Hannover, Fachgebiet Software Engineering, Bachelor's Thesis, 2012