SEUH 2011

Software Engineering im Unterricht der Hochschulen

Tagungsband 12. Workshop SEUH
München, 24. – 25. Februar 2011

Herausgeber: Jochen Ludewig*, Axel Böttcher**

Redaktion: Holger Röder*

* Universität Stuttgart, Institut für Softwaretechnologie

** Hochschule München, Fakultät für Informatik und Mathematik

Inhaltsverzeichnis

Vorwort 2

Eingeladene Beiträge

Effizientere Software-Entwicklung durch Industrialisierung der Prozesse 3 – 8
Oliver F. Nandico

Praxis lehren und Forschung umsetzen: Wie können Hochschullehrer diese Kluft überbrücken? 9
Heinz Züllighoven

Referierte Beiträge

Planspiel und Briefmethode für die Software Engineering Ausbildung – ein Erfahrungsbericht 10 – 15
Georg Hagel, Jürgen Mottok

Teamentwicklung in studentischen Projekten 16 – 20
Doris Schmedding

Dynamische Klassendiagramme – Nutzung der Metapher vom „Konsumieren und Produzieren“ in BlueJ 21 – 26
Axel Schmolitzky, Chris Stahlhut

Ein Dashboard für Learning-Management-Systeme 27 – 32
Daniel Kulesz

Kompetenzorientierte Lehre im Software Engineering 33 – 39
Axel Böttcher, Veronika Thurner, Gerhard Müller

Vier Jahre Software-Engineering-Projekte im Bachelor – ein Statusbericht 40 – 44
Stephan Kleuker, Frank M. Thiesing

Vorwort

Jochen Ludewig, Universität Stuttgart

ludewig@informatik.uni-stuttgart.de

1992 fand in Stuttgart die SEUH statt; damals war nicht abzusehen, dass daraus eine Institution werden würde. Kurt Schneider, damals Mitarbeiter, heute Kollege, und ich haben freihändig ein Konzept entworfen, das bei den folgenden Tagungen dieser Reihe verfeinert, aber kaum geändert wurde. Heute sind die Invarianten der SEUH:

- Die SEUH findet alle zwei Jahre am letzten Donnerstag und Freitag im Februar der ungeraden Jahre statt.
- Tagungssprache ist Deutsch; wer nicht Deutsch spricht, trägt auf Englisch vor.
- Die Tagung wird durch einen Tagungsband dokumentiert. (Erstmals liegt hier ein virtueller Tagungsband vor!)
- Veranstaltungsort ist abwechselnd eine Universität und eine Fachhochschule. Ist der Ort eine Fachhochschule, dann wird das Programmkomitee von einer Person aus der Universität geleitet, und umgekehrt. Alle Mitglieder im Programmkomitee haben SEUH-Erfahrung.
- Die Teilnahmegebühren sind sehr niedrig.
- Nach dem Vorabendtreffen am Mittwoch steht am Donnerstag das eigentliche „soziale Ereignis“ auf dem Programm.
- Die Diskussion nimmt breiten Raum ein, mindestens die Hälfte der Zeit. Es gibt in der Diskussion keine heiligen Kühe.
- Die Kontinuität sichert ein Lenkungskreis aus (normalerweise) drei Personen.

In Zukunft wird darüber nachzudenken sein, wie Form, Rhythmus und Ausrichtung der Tagung weiterentwickelt werden sollten. Insbesondere ist das Verhältnis zur deutschen Software-Engineering-Konferenz zu klären.

Im Programm dieser zwölften SEUH liegt der Schwerpunkt bei den studentischen Projekten. Denn inzwischen hat sich herumgesprochen, dass wir im traditionellen Unterricht nur Grundlagen vermitteln können, die dann praktisch erprobt, eingeübt und stabilisiert werden müssen. Weitere Beiträge befassen sich mit dem durchgängigen Beispiel in der Lehre, einer Entwicklungsumgebung für die Programmierausbildung und einer Software zur Verwaltung der Lehrveranstaltungen.

Im ersten eingeladenen Vortrag von Oliver F. Nandico (CapGemini) geht es um die Software-Prozesse in der Praxis, im zweiten von Heinz Züllighoven (Universität Hamburg) um die Chancen für die Lehre, die entstehen, wenn der Hochschullehrer und seine Mitarbeiter auch ein Software- und Beratungsunternehmen betreiben.

Das Programm dieser SEUH wurde wie üblich von einem Komitee gestaltet, dessen sämtliche Mitglieder mit dem Thema Software-Engineering-Lehre hauptberuflich befasst sind:

- Axel Böttcher, Hochschule München (lokale Organisation)
- Ralf Bruns, Fachhochschule Hannover
- Marcus Deininger, Hochsch. f. Technik Stuttgart
- Ulrike Jaeger, Hochschule Heilbronn
- Jochen Ludewig, Universität Stuttgart (Vorsitz)
- Barbara Paech, Universität Heidelberg
- Axel Schmolitzky, Universität Hamburg
- Kurt Schneider, Leibniz Universität Hannover
- Silke Seehusen, Fachhochschule Lübeck
- Olaf Zukunft, HfAW Hamburg

Holger Röder, Universität Stuttgart, hat die Vorbereitung und Organisation unterstützt. Für die Verwaltung und Begutachtung der Einreichungen wurde das System ConfISS eingesetzt, das 2008/09 im Rahmen eines Studienprojekts der Softwaretechniker in Stuttgart entwickelt wurde. (ConfISS steht für Conference Information System Stuttgart.)

Allen, die bei Planung und Vorbereitung mitgewirkt haben, ist zu danken, auch all jenen, die – mit oder ohne Erfolg – Manuskripte eingereicht haben. Ganz besonderer Dank gebührt Axel Böttcher, der die Durchführung in München möglich gemacht hat.

2010, im SEUH-freien Jahr, ist die Tagung volljährig geworden. Ich freue mich, dass nach zwei Töchtern im engeren Sinne auch diese dritte attraktiv und erfolgreich ist und in Zukunft des Vaters nicht mehr bedarf. Bei meinem Sohn, dem 1996 entstandenen Studiengang Softwaretechnik, hätte ich noch vor kurzem das Gleiche gesagt. Aber er ist leider in der Pubertät unter schlechten Einfluss geraten (Bologna!). Hoffen wir, dass er diese kritische Phase ohne bleibende Schäden übersteht!

Effizientere Software-Entwicklung durch Industrialisierung der Prozesse

Oliver F. Nandico

oliver.f.nandico@capgemini.com

Zusammenfassung

Es gibt heute keine Branche, kein Unternehmen und keine Geschäftsprozesse, bei denen nicht „ein Stück Software“ involviert ist. Wir können uns eine ganze Reihe alter und vor allem neuer Geschäftsmodelle ohne IT-Unterstützung gar nicht mehr vorstellen.

Trotz, oder gerade aufgrund dieser Alltäglichkeit befindet sich die Entwicklung von Software in einem grundsätzlichen Wandel als Reaktion auf widersprüchliche Anforderungen. Verfallende Marktpreise und steigender Produktivitätsdruck verlangen die Industrialisierung der Entwicklungsprozesse. Das bedeutet Wiederholbarkeit, Standardisierung und Automatisierung. Gleichzeitig fordern die Anwender von Software die schnelle, innovative und ganz spezifische Reaktion des Softwareentwicklers auf ihre besondere Aufgabenstellung und ihr Geschäftsmodell. So erzielen sie ihrerseits einen Vorsprung im Wettbewerb vor der Konkurrenz.

Softwareentwicklung muss also heute Standardisierung und Automatisierung geeignet mit Flexibilität, Reaktionsfähigkeit und Nutzernähe kombinieren.

Diese Veränderung in der Softwareentwicklung wirkt sich konsequenterweise auf die Tätigkeit und das Berufsbild des Software-Ingenieurs aus. Zu diesem Wandel formuliert dieser Artikel sieben Thesen.

These 1: Standardisierter Prozess

These 1: Software-Entwicklung findet heute in einem wiederholbaren, standardisierten und arbeitsteiligen Prozess statt.

Mit „The greatest improvements in the productive powers of labour, and the greater part of the skill, dexterity, and judgment, with which it is anywhere directed, or applied, seem to have been the effects of the division of labour.“ beginnt schon Adam Smith sein Werk „Wealth of Nations“.

Voraussetzung für eine produktivitätssteigernde Arbeitsteilung ist ein klar definierter Prozess, in dem die Aufgaben und die Tätigkeiten für jede Rolle, und Vor- und Nachbedingungen festgelegt sind. Unter dem heute herrschenden Produktivitätsdruck in der Softwareentwicklung ist damit die arbeitsteilige Erstellung von Software in einem standardisierten Vorgehen Pflicht für jede IT-Abteilung und erst recht für jeden IT-Dienstleister.

Die industrialisierte Arbeitsweise hat den genialen Einzelprogrammierer oder das hervorragende Forschungsteam als Schöpfer neuer Programmierverfahren und Algorithmen in der täglichen Arbeit, im „Brot- und Butter“-Geschäft der Informatik abgelöst.

In der Konsequenz ersetzt der Spezialist für bestimmte Tätigkeiten im Entwicklungsprozess oder für bestimmte inhaltliche Fragestellungen den Generalisten. Diesen Verlust ganzheitlicher Handwerkskunst kann man bedauern, aber er ist ein Zeichen für die gewachsene Reife unserer Branche. Nicht zuletzt macht ein standardisierter und arbeitsteiliger Prozess die Softwareentwicklung plan- und kalkulierbarer, damit letztendlich erst erfolgreich.

Handwerklicher Vorgehensweise sind zudem natürliche Grenzen im Hinblick auf Komplexität und Größe eines Vorhabens gesetzt. Gerade die fast vollständige Durchdringung aller Funktionsbereiche von Unternehmen mit Software macht die Erstellung von neuen Programmen heute so schwierig und umfangreich, dass eine arbeitsteilige Vorgehensweise geboten ist.

Der arbeitsteilige und standardisierte Entwicklungsprozess bringt einen weiteren Vorteil: Er ist mess- und quantifizierbar. Erst wenn etwas wirklich messbar ist, kann man es optimieren, klar planen und scharf kalkulieren. Dies wiederum ist unter den verfallenden Marktpreisen für jedes Entwicklungshaus überlebensnotwendig.

Welcher Prozess ist aber der Richtige? Der Rational Unified Process ist sicher weit verbreitet und ein gewisser Standard, nicht zuletzt bei Capgemini.

Im deutschen öffentlichen Dienst ist das V-Modell XT als Standard vorgegeben. COBIT und ITIL sind gute Referenzmodelle für die Softwareentwicklung. Natürlich praktizieren eine Reihe von Entwicklungsteams, bei uns und anderswo, und ganze Firmen agile Vorgehensweisen. Wichtig ist: Der definierte Prozess muss der Komplexität und Aufgabe angemessen, und damit entsprechend konfigurierbar sein. Eine Customer-Relationship-Management-Lösung für ein globales Unternehmen ist anders zu entwickeln als eine Android-Sales-App.

Alle starren, definierten Prozesse sind entsprechend zu verwerfen, andere Anforderungen betrachten wir in der Diskussion der weiteren Thesen.

Was verlangt nun die Einführung standardisierter Prozesse von einem angehenden Softwareingenieur? Zum einen muss er die Diskussion um den Softwareentwicklungsprozess kennen, und gestalten können. Softwaredienstleister wie etwa unser Haus erwarten von ihren angehenden Beschäftigten Wissen um Aktivitäten und Zusammenhänge in der Softwareentwicklung. Der Softwareingenieur muss nach seiner Ausbildung wissen, was die Branche unter bestimmten Arbeitsergebnissen versteht, welche Anforderungen an diese sein Team und der Kunde stellt und über welche Aktivitäten, Methoden und Techniken sie zustande kommen. Kurzum der angehende Softwareingenieur braucht eine Ausbildung in den „industriellen Fertigungsprozessen“ der Softwareentwicklung.

These 2: Verringerte Wertschöpfung

These 2: Die Wertschöpfungstiefe in der Entwicklung verringert sich

“There’s only really one metric to me for future software development, which is — do you write less code to get the same thing done?” sagte Bill Gates 2005 im Hinblick auf die Veränderungen in der Software-Entwicklung.

Weniger Code bedeutet eine Verringerung Wertschöpfungstiefe in der eigentlichen Entwicklung. Diese Tendenz besteht seit Jahrzehnten und hat sich immer weiter verstärkt. Das führte zur Nutzung von ausgefeilteren technischen Produkten wie Transaktionssteuerungen, Datenbankmanagementsystemen und Bibliotheken für fast jeden Anwendungszweck. Schließlich zu immer „höheren“ Programmiersprachen.

Diese Frage der Wertschöpfung diskutieren die Anwender immer wieder unter der dichotomischen Sichtweise von „Standard-Software“, also (Halb-) Fertigsoftware, und Individualsoftware, „make or buy“. Die eine Seite führt immer Preis, Erstellungs- und Wartungsaufwand, die andere Seite die optimale Unterstützung der spezifischen Geschäftsprozesse und den differenzierenden Charakter als wesentliche Entscheidungskriterien an.

Dieser hebt sich aber zunehmend auf: Längst sind Softwarepakete komplexe, konfigurierbare und damit an die jeweiligen spezifischen Anforderungen anpassbare Systeme, denen im Übrigen der Kostenvorteil zunehmend abhandenkommt. Individualentwicklung setzt dagegen immer stärker und nicht nur als Lippenbekenntnis auf Wiederverwendung von bestehenden (Teil-) Lösungen, auf domänenspezifische Sprachen und Programmierung.

Längst gibt es darüber hinausweisend die Angebote „aus der Wolke“. Salesforce.com überzeugt seine Kunden seit mehr als 10 Jahren, dass sie überhaupt nichts mehr mit Softwareentwicklung zu tun haben müssen. Was bedeutet das für die interne IT-Abteilung?

Die Softwareentwicklung ist damit in Zukunft in der überwiegenden Breite mehr das Zusammenfügen von großen und kleinen Bausteinen zu einer dem Nutzer optimal angepassten Lösung. Integration von Software aus ganz unterschiedlichen Quellen bekommt so eine viel wichtigere Rolle als die Programmierung von Funktionen. Es gilt das Grundprinzip der Wiederverwendung: Schon wenige Tage Recherche nach einer angebotenen Lösung können Monate an Entwicklungsarbeit ersetzen.

Das Paradigma für den Umgang mit dieser Entwicklung ist die serviceorientierte Architektur: Der Service, die angebotene, in sich abgeschlossene und elementare fachliche Leistung ist der Baustein, um den sich die Softwareentwicklung dreht. Softwarepakete müssen solche integrierbaren Services anbieten, die Individualentwicklung erstellt sie und Cloud-Betreiber stellen sie über das Web zur Verfügung. Entwicklung wird damit mehr und mehr zur Architekturarbeit, zur richtigen Orchestrierung von Services.

Der Prozess der Softwareentwicklung muss entsprechend dieser Entwicklung serviceorientierte Architektur unterstützen. Das heißt er muss insbesondere Integration und die Schaffung einer Integrationsplattform angemessen berücksichtigen. Im Weiteren unterscheidet dann der Prozess der Softwareentwicklung nicht mehr zwischen individualentwickelten oder fertig bereitgestellten Services.

Bedeutet es nun, dass wir für die Ausbildung in der Softwareentwicklung die intensive Beschäftigung mit den Produkten eines Herstellers erwarten? Sicherlich nicht, denn alles Spezialwissen ist am Ende der Ausbildung mit hoher Wahrscheinlichkeit schon wieder völlig veraltet. Wichtig sind dagegen Fertigkeiten zum Umgang mit Softwarepaketen, zur Integration von Softwarepaketen aus verschiedenen Quellen mit individuell entwickelter Software. Klar formuliert: Nicht der XY-

Modulexperte muss Ziel der Ausbildung sein, sondern der Architekt von Anwendungslandschaften. Dabei gilt natürlich: Dieser Architekt kennt die am Markt angebotenen Produkte, ihre Stärken und Schwächen.

These 3: Wachsende Automatisierung

These 3: Der Automatisierungsgrad in der Softwareentwicklung muss steigen

„Der Einsatz von geeigneten Werkzeugen ist einer der entscheidenden Faktoren, um eine systematische Unterstützung umzusetzen, wie sie von Vorgehensmodellen beschrieben wird. Werkzeuge werden in verschiedenster Form in allen wesentlichen Phasen der Entwicklung und darüber hinaus eingesetzt." So beschreibt der „Leitfaden und die Orientierungshilfe" der BITKOM zum Thema „Industrielle Softwareentwicklung" den Aspekt Automatisierung.

In der Fertigungsindustrie ist eine Vorgabe für die Produktivitätssteigerung und entsprechende Rationalisierung von 10-15% im Jahr völlig normal und allgemein üblich. Will die Softwarebranche diesem Beispiel folgen, ist vor allem eine Automatisierung in der Entwicklung notwendig, neben der Standardisierung des Prozesses und der Verringerung der Wertschöpfungstiefe.

Automatisierung in der Softwareerstellung bedeutet die möglichst formale, konsistente und vollständige Beschreibung der Aufgabenstellung auf möglichst hohem semantischem Niveau, um alles weitere daraus automatisch zu generieren.

Dahinter steht die schon immer verfolgte Absicht, die tatsächliche Programmierarbeit in ihrer Komplexität und Fehleranfälligkeit zu vermeiden. Diese Absicht leitete schon die Entwicklung der ersten Compiler.

Heute bedeutet Automatisierung vor allem Einsatz von modellgetriebenen Techniken. Grundlage ist eine von der technischen Implementierung unabhängige, im Regelfall grafische Spezifikation der Software-Lösung. Entsprechende Werkzeuge transformieren diese in mehreren Schritten zur ablauffähigen Implementierung.

Teil der Automatisierung ist die Automatisierung für den Test von Software. Regressionsfähigkeit von Test und angemessener Abdeckungsgrad lassen sich überhaupt nur über automatisierte Tests erreichen. Hier ist der allgemein anzutreffende Reifegrad bei den Anwenderunternehmen eher niedrig.

Modellgetriebene Softwareentwicklung und Testautomatisierung sind heute in vielen Bereichen der Softwareentwicklung Standard, die Werkzeugwelt fast unüberschaubar. Dennoch zeigen sich noch Schwachstellen: Durchsatzoptimierung verlangt Anpassungen an die konkrete Hardware-Plattform. Daraus resultierende manuelle Eingriffe in den automatisch erzeugten Code zu erhalten, bedeutet immer noch eine Herausforderung für die Werkzeuge.

Modellgetriebene Entwicklung ist heute damit ganz selbstverständlich Teil des Softwareentwicklungsprozesses, Codegenerierung ist Stand der Technik. Diese Entwicklung wird und muss sich fortsetzen, im Sinn einer weiteren Industrialisierung. Das schränkt die manuellen Codierungsarbeiten ein, und verändert so sicher den Beruf des Software-Ingenieurs weiter.

In der Konsequenz muss die Ausbildung für Softwarenentwicklung den Umgang und Einsatz mit Werkzeugen zur Automatisierung in jeder Beziehung lehren. Modellgetriebene Softwareentwicklung ist in unserer Branche ebenso wichtig, wie der Umgang mit Fertigungsrobotern und CNC-Maschinen im Maschinenbau.

These 4: Geografische Verteilung

These 4: Software-Entwicklung erfolgt geografisch verteilt

„It's the combination of flexibility, the right competencies, the blend of onshore and offshore resources and cost savings that make Rightshore® such an attractive proposition." Stefan Fransson, CIO, Mölnlycke Health Care

Es ist kein Geheimnis, dass die Verlagerung von Softwareentwicklungsarbeiten in Regionen mit geringeren Lohnkosten internen IT-Abteilungen erhebliche Kostenvorteile bringen und IT-Dienstleistern niedrigere Preise ermöglichen. Dies ist, trotz der zunehmenden Automatisierung, dem sehr hohen Lohnkostenanteil von 70% (Schweizerische Technische Zeitschrift) in der Softwareentwicklung geschuldet.

Dazu fallen für Software keine im eigentlichen Sinn Logistikkosten an. Natürlich muss in den entsprechenden Ländern die Infrastrukturanbindung gewährleistet sein, also Internetverbindung existieren. Hier leistet etwa in Indien der Staat hohe Vorleistungen. Das gilt analog für die Ausbildung der Entwicklerinnen und Entwickler.

Viele Kunden haben die Erfahrung gemacht, dass eine einfache Verlagerung der Arbeiten in Standorte mit niedrigeren Löhnen nicht zu den gewünschten Ergebnissen führt. Das fachliche Verständnis war häufig nicht gegeben, Kommunikationsprobleme haben die die Fertigstellung der Software verzögert und den Aufwand dafür erhöht.

Geografische Verteilung der Softwareentwicklung verlangt einen sehr angepassten Entwicklungsprozess, in dem Aufgaben und die Interaktion zwischen den Prozessbeteiligten in ihren Rollen sehr sorgfältig definiert sein müssen.

Die Arbeit in einem räumlich, d.h. geografisch verteilten Team verlangt einen wohldefinierten und arbeitsteiligen Prozess in der Entwicklung, der allen Beteiligten hinreichend klar ist.

Von den Beteiligten in der Software-Entwicklung erfordert er zusätzliche soziale Kompetenzen, vor allem interkulturelle Fähigkeiten, die über reine Sprachkenntnisse hinausgehen.

Genau dieser Aspekt ist in der Ausbildung zu verstärken. Das sind dann Sprachkurse, das sind mehr Auslandspraktika. Interkulturelle Fähigkeiten kann man nicht aus dem Buch lernen, man muss sie erleben. Die Erwartungen aus dem Bologna-Prozess harren allerdings leider noch ihrer Erfüllung.

These 5: Mehr Kundennähe

These 5: Die Nähe zum Kunden und die Vertrautheit mit seiner Aufgabenstellung müssen sich erhöhen

„We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value: Individuals and interactions over processes and tools, Working software over comprehensive documentation, Customer collaboration over contract negotiation, Responding to change over following a plan.“ steht im „Agilen Manifest“ von 2001.

Während die bisherigen Thesen alle auf einen wohldefinierten, detaillierten, eben industrialisierten Softwareentwicklungsprozess weisen, zeigt die Diskussion und der Erfolg agilen Vorgehens, dass Kundennähe und die schnelle Reaktion auf veränderte Anforderungen unverzichtbar für den Erfolg eines Softwareprojektes sind.

Vorgehensmodelle, die ein Vorhaben nötigen an einer einmal getroffenen fachlichen Entscheidung oder an dem am Beginn festgelegten funktionalen Umfang festzuhalten, sind zum Scheitern verurteilt. Aus diesem Grund hat niemand Wasserfallmodelle, bei ehrlicher Betrachtung des tatsächlichen Vorgehens, wirklich praktiziert. Lange Spezifikationsphasen und bei Fertigstellung veraltete Software waren und sind verbreitete Abbruchsgründe.

Der Kern für den Erfolg von agilen Methoden ist die frühe und konsequente Einbeziehung des Kunden und Anwenders. Dazu kommt die Konzentration auf lauffähige Software, um ihm die Beurteilung und Steuerung des Projektes zu ermöglichen.

Grundsätzlich bedeutet Agilität nicht wie bisher die Abwehr aller „störenden“ Änderungsanforderungen, sondern die bewusste Ausnutzung der Flexibilität der Softwareentwicklung, um den Nutzen für den Kunden zu maximieren. Im schlechtesten Fall erhält der Kunde am Schluss ein System mit der Hälfte des gewünschten Funktionsumfangs, das aber dennoch lauffähig ist. Das ist besser als eine vollständige Spezifikation, mit der er überhaupt nichts anfangen kann.

Es gibt jedoch Gefahren beim agilen Vorgehen. Das Entwicklungsteam kann Details übersehen. Lokale und kurzfristige Optimierungen können umfassende und nachhaltig tragfähige Gesamtlösungen verhindern. Entscheidungsschwache oder fachlich inkompetente Kundenansprechpartner verhindern den Erfolg.

Mit agilem Vorgehen sind auch Missverständnisse verbunden. Agilität heißt nicht ungeordnetes, auf bloßer direkter Kommunikation basierendes Vorgehen und Verzicht auf jegliche Dokumentation. Vielmehr verlangen agile Prozesse, wie zum Beispiel SCRUM, weitaus mehr Wissen und Erfahrung mit dem Prozess und mehr Disziplin von allen Beteiligten.

Eine Kontroverse besteht, inwieweit agile Vorgehensweisen bezüglich Projektgröße und Projektkomplexität skalieren. Erfahrungen aus unserem Haus deuten in die Richtung, dass wir im Einsatz von agilen Techniken sehr genau nach Projekttyp unterscheiden müssen. Das heißt, dass agile Techniken dort am besten wirken, wo in einem eher kleinen Team in kurzer Zeit hoch innovative Lösungen entstehen sollen. Dessen ungeachtet können agile Vorgehensweise die Effektivität und Effizienz bei großen und umfangreichen Lösungen erhöhen. Hier befinden wir uns noch in einer Phase der Diskussion und Forschung.

Eine andere Frage stellt sich: Interessiert es den Kunden überhaupt, nach welchem Prozess ein Dienstleister für ihn die Software entwickelt? Die Erfahrung bestätigt das. Wenn, wie in unserer Branche ein Übergangsstadium zur Industrialisierung gegeben ist, und gleichzeitig der Anteil von gescheiterten Entwicklungsprojekten nach wie vor sehr hoch ist, dann ist das Vorgehen und damit der Prozess ein wesentliches Argument in der Auftragsvergabe. Zusicherungen in Bezug auf zeitverkürzter Abwicklung und die Anforderung nach intensiver Mitwirkung des Kunden im agilen Prozess verlangen die Darstellung von Prozess und Vorgehen. Folgerichtig gibt es ein sehr vitales Interesse des Kunden in Bezug auf die Umsetzung entsprechend dem zugesicherten Verfahren.

Für die Ausbildung gilt hier eine einfache Anforderung: Kenntnisse und Erfahrung in agilen Techniken. Erfahrung heißt hier, ein Projekt, also etwa ein Praktikum nach einem solchen Vorgehen im Team praktiziert zu haben. Das stellt natürlich entsprechende Anforderungen an die Ausbilder.

These 6: Schneller!

These 6: Umsetzungsgeschwindigkeit ist essentiell

„Während ein Projektleiter vor einiger Zeit noch stolz sein konnte, wenn er nach einem Jahr ein fertiges System abliefern konnte, muss er heute diese Aufgabe in längstens einem halben Jahr erledigen können" (CIO eines Kundenunternehmens, 2007).

Aller Produktion, und das schließt Softwareentwicklung ein, ist gemeinsam, dass die Anwender eine Verringerung der Prozessdurchlaufzeiten fordern. Eine wirtschaftliche Chance hat typischerweise ein begrenztes Zeitfenster, ein Wettbewerbsvorsprung besteht nur in einem engen zeitlichen Rahmen. Wenn Software dafür notwendig ist, und das ist fast immer der Fall, dann muss sie **jetzt** zur Verfügung stehen.

Standardisierung, Automatisierung, selbst die Qualität der Software steht aus Sicht der Anwender hinter der Ausnutzung des Chancenfensters zurück.

Diese grundsätzliche Anforderung hat es schon immer gegeben. Darauf hat das Softwareprojekt mit Stufenkonzepten und Systemdurchstichen reagiert. Die Erwartungshaltung der Anwender hat sich über die Zeit verstärkt: Wo diese Anwender die Entwicklungszeiten für neue Automodelle oder auch neue Flugzeugtypen reduzieren müssen, gibt es kein Verständnis, wenn die Software, die damit einhergeht, nicht diese Zeitvorgabe schafft.

Unter dem gegebenen zeitlichen Druck sind die Kunden häufig bereit, Ansprüche an die Qualität der Software zurückzustellen. Das führt langfristig zu schlechten Lösungen, die weder anpassbar noch betreibbar sind. Hier ist es die Aufgabe der Softwareentwicklung, über die Gestaltung des Prozesses eine Mindestqualität von vornherein zu sichern. Dafür gibt es entsprechende Techniken, die ebenfalls ihre Wurzeln in der Diskussion um agile Techniken haben. Beispielhaft seien hier nightly Builds, frühe Unit Tests und Prototypen genannt.

Trotz der Ausrichtung auf den wohldefinierten Prozess sind Umfang, sowie Art und Weise der Dokumentation einer kritischen Betrachtung zu unterziehen. Dokumente, die keine Wirkung haben und keine Entscheidung beeinflussen, benötigt auch niemand. Für jedes im Prozess zu lieferende Zwischenergebnis muss am Beginn eines Entwicklungsprojektes die Projektleitung Sinn und Zweck bestimmen können. Ergebnisdokumente, deren einziger Zweck die Erfüllung einer Anforderung aus der Prozessdefinition ist, halten die Entwicklung nur unnötig auf.

Zusätzlich zu den Anforderungen an die Ausbildung aus der Diskussion der anderen Thesen ergibt sich hier nur noch einmal hervorzuheben: In der Ausbildung ist nicht nur Lösungs- sondern zukünftig und vor allem Prozesskompetenz gefordert.

These 7: Angemessen!

7: Angepasste Prozesse sind der Schlüssel zum Erfolg

„The danger of standard process is that people will miss chances to take important shortcuts." Dieses Zitat von Tom DeMarco zeigt schon richtig eine der wesentlichen Gefahren eines wohldefinierten Entwicklungsprozesses auf: Er ist nicht an die eigentliche Aufgabe angepasst.

Viele Vorgehensmodelle in der Vergangenheit, wie etwa das V-Modell, und verschiedene Ausprägungen von Vorgehensmodellen bei verschiedenen Firmen haben oft eine Gemeinsamkeit: Sie sind relativ starr, ihre Dokumentation ist komplex und umfangreich und kein betroffener Softwareentwickler hat sie angewendet oder auch nur verstanden.

Das hängt damit zusammen, dass Theoretiker in verschiedenen Definitionen der Prozesse versucht haben, auf der abstrakten Ebene jedes Entwicklungsproblem zu lösen oder zumindest den Lösungsweg über Aktivitäten und Ergebnisdokumente vorzugeben. Das hilft den Entwicklern nicht und macht die Beschreibungen nur unverständlich.

Die Idee, für die ganz unterschiedlichen Aufgabenstellungen in ganz unterschiedlichen Softwareprojekten den gleichen Prozess aufzusetzen, ist zum Scheitern verurteilt. Das Softwareprojekt, das in einem engen vorgegebenen Zeitrahmen ein Preisvergleichsportal erstellt, hat mit einem Projekt für die Entwicklung der Steuerungssoftware für einen Laborautomaten nur wenige Gemeinsamkeiten.

Das Angebot, einen wohldefinierten Standardprozess zu konfigurieren, hilft dann nur bedingt weiter, wenn der Ausgangspunkt zu komplex und abstrakt ist.

Es gibt, wie die Diskussion der Thesen gezeigt hat, verschiedene Aspekte, die auf unterschiedliche Entwicklungsprozesse hindeuten. In unserem Hause sehen wir sechs Dimensionen, die für den angemessenen Entwicklungsprozess relevant sind:

- Priorisierung zwischen Funktionsumfang und Zeit
- Änderungsdynamik bei den Anforderungen
- Kundenkultur
- Einbindung des fachlichen Auftraggebers
- Grad der Abhängigkeiten zwischen einzelnen Teilkomponenten und –funktionen
- Projektgröße

Aus diesen Dimensionen ergeben sich die Aspekte für den jeweils angemessenen Entwicklungsprozess. Im Einzelnen heißt das insbesondere, dass der Grad an agiler Vorgehensweise und der Einsatz von bestimmten agilen Techniken, der Einsatz von

Offshore-Entwicklungskapazitäten und die Einbindung von Softwarepaketen bzw. Cloud-Angeboten daraus abzuleiten ist.

Daraus ergeben sich unterschiedliche Typen von Projekten und Entwicklungsprozessen. Eine nicht abschließende Liste ist:

- Entwicklung relativ kleiner, technisch wie fachlich hoch-innovativer Systeme, die in einem engen Zeitrahmen produktiv werden müssen.
- Entwicklung funktional umfangreicher, fachlich und architektonisch komplexer Systeme, die die differenzierenden Kernleistungen eines Unternehmens unterstützen.
- Integration von verschiedenen schon existierenden oder neu zu entwickelnden Services mit den Angeboten aus Softwarepaketen und Cloud-Angeboten.

Für so zu identifizierende Projekttypen gilt es nun für das Management der Softwareentwicklung, einfache und verständliche Prozessdefinitionen zu erstellen. Auf dem Weg zum industrialisierten Prozess ist nach der Definition dann der zweite Schritt die Herstellung der Automatisierung und Unterstützung der Entwickler in der Auswahl des angemessenen Prozesses, sowie der Konfiguration und Umsetzung dieses so ausgewählten Prozesses.

Die Konsequenzen für Software-Entwickler

Was bedeutet nun die Industrialisierung in der Softwareentwicklung für das Berufsbild des Software-Ingenieurs?

Zunächst einmal heißt es, dass die Reflektion des Entwicklungsprozesses zum selbstverständlichen Kanon in der Ausbildung gehört. Die Grundlage dieser Reflektion sind vertiefte Kenntnisse zu den Standardvorgehensmodellen. Hierzu gehört sicherlich der Rational Unified Process und das V-Modell XT. Dazu kommen die heute ganz standardmäßig vorausgesetzten Referenzprozessmodelle ITIL und COBIT. Eine Wunschvorstellung ist es, wenn Absolventen aus der Hochschulausbildung hier schon entsprechende Zertifizierungen mitbringen.

Genauso, wie statt einer bestimmten Programmiersprache die dahinter stehenden Konzepte wie Objekt- und Aspektorientierung gefragt sind, ist es für Vorgehensmodelle entscheidend, ihren Zweck und beabsichtigten Einsatzrahmen zu kennen und zu wissen, wo und wie sie anzupassen sind.

Die Softwareentwicklung hat einen hohen Bedarf an Prozessingenieuren und –ingenieurinnen. Diese brauchen wir für die Aufnahme, Modellierung und Optimierung von fachlichen Prozessen in den verschiedenen Branchen, nicht zuletzt für die eigene, die Softwareentwicklung.

Heute führen wir für unsere Beschäftigten interne Schulungen zu diesem Thema durch. Dazu leistet eine eigene Gruppe „Produktionssteuerung" die notwendige Unterstützung der Projekte. Diese Unterstützung ist notwendig, weil wir in unserem Hause feste Vorgaben für die Abwicklung und Durchführung von Softwareentwicklungsprojekten machen. Die Weiterentwicklung von Methoden und Vorgehensmodellen ist schließlich in der Kompetenz unserer Entwicklungsabteilung. Insgesamt betreiben wir einen hohen Aufwand in der Umsetzung von definierten und standardisierten Prozessen. Wie schon beschrieben, sind für uns die ständige Verbesserung der Produktivität, und die schnelle Reaktion auf neuere Entwicklungen, die darauf beruhen, essentiell.

Gerade im Rahmen einer weiteren Industrialisierung bleiben für Softwareingenieure die sozialen Kompetenzen wichtig. An vorderster Stelle stehen dabei Teamfähigkeit und Kommunikationsfreude. Trotz aller Standardisierung und Automatisierung bleibt Softwareentwicklung ein kreativer Prozess, bei dem ein hochqualifiziertes Team in der Zusammenarbeit optimale Lösungen sucht und findet. Zunehmend wichtiger ist dabei die globale Ausrichtung, die interkulturelle Kompetenz über bloße Sprachkenntnisse hinaus verlangt.

Literatur

Agile Manifesto (2001) unter: http://agilemanifesto.org

BITKOM 2010, Industrielle Softwareentwicklung, unter: http://www.bitkom.org/files/documents/Industrielle_Softwareentwicklung_web.pdf

DeMarco, T.; Lister, T. (1999): Peopleware: Productive Projects and Teams, Dorset House Publishing Company, 2nd edition, New York N.Y. 1999

Kremer, M. (2010): eee@Quasar: efficient, effective, and economic software development, Vortrag Capgemini Kundenforum Architektur

Smith, A. (2005): An Inquiry into the Nature and Causes of the Wealth of Nations, Pennsylvania State University, Hazelton 2005

Udell, J. (2005): Interview with Bill Gates, Infoworld 25. September 2005

Software made in India (2008) in: Schweizerische Technische Zeitschrift – Swiss Engneering, Ausgabe November 2008, S.10-11

Praxis lehren und Forschung umsetzen: Wie können Hochschullehrer diese Kluft überbrücken?

Heinz Züllighoven, Universität Hamburg und C1 WPS

zuellighoven@informatik.uni-hamburg.de

Zusammenfassung

Die Softwaretechnik an Hochschulen versteht sich meist als anwendungsorientierte (Ingenieur-) Wissenschaft. Was sind aus heutiger Sicht ihre wissenschaftlichen Fundamente für die Lehre, und wie stabil sind sie?

Professionelle Softwareentwicklung sollte sich an wissenschaftlichen Konzepten und Ergebnissen orientieren. Umgekehrt sollte die Praxis auch Impulse für die anwendungsorientierte Wissenschaft geben. Welche Impulse sind dies für die Softwaretechnik?

Der Versuch, anwendungsorientierte Wissenschaft und forschungsorientierte Praxis in Forschung und Lehre zusammenzubringen, läuft in Hamburg seit zwanzig Jahren. Was ist dabei herausgekommen? Was könnte man auf andere Standorte übertragen?

Planspiel und Briefmethode für die Software Engineering Ausbildung - ein Erfahrungsbericht

Georg Hagel, Hochschule Kempten

georg.hagel@fh-kempten.de

Jürgen Mottok, LaS³, Hochschule Regensburg

juergen.mottok@hs-regensburg.de

Zusammenfassung

Die konstruktivistische Perspektive unterstützt einen Paradigmenwechsel der akademischen Lehre hin zu einer Lerner- und Lernprozesszentrierung. Dabei thematisieren Selbstgesteuertes Lernen und aktivierende Lehre das studentische Lernen neu.

Das Zusammenspiel von Lehren und Lernen mit dem Ansatz einer konstruktivistischen Didaktik wird mit Beispielen erprobter Lernkonzepte der Software Engineering Ausbildung der Vortragenden unterlegt: Planspiel und Briefmethode werden exemplarisch diskutiert.

Die gezeigten konstruktivistischen Methoden lassen sich auf Lebenslanges Lernen des Software Engineering im Berufsumfeld übertragen.

Einführung – Lernarrangements für selbstgesteuertes Lernen

Inzwischen ist eine Wende der didaktischen Wahrnehmung erkennbar. Während die curriculumtheoretische Didaktik auf der Überzeugung basierte, dass Lernprozesse Erwachsener zielgerichtet planbar und steuerbar sind, zielt der konstruktivistisch-didaktische Ansatz auf die Ausgestaltung von Lernumgebungen (Siebert, 2009). In diesen Lernumgebungen, auch Lernsettings, genannt, sollen selbstgesteuerte und kreative Lernprozesse angeregt werden. Dabei ist nicht nur das Expertenwissen der Referenten eine Ressource, sondern auch das Vorwissen, die Erfahrungen und die Fragestellungen aller Beteiligten. Dieser methodisch didaktische Ansatz gestaltet einem Paradigmenwechsel „The Shift from Teaching to Learning“ (Welbers, 2005) aus.

Konstruktivismus

Der Konstruktivismus ist eine Theorie über den Erwerb von Wissen, das Lernen und Lehren. Kernaussage ist, dass jeder Mensch durch die Kommunikation mit seiner Umgebung eine eigene persönliche Wirklichkeit erschafft; diese unterscheidet sich von der Wirklichkeit anderer Menschen. Lernen wird als die Konstruktion von Bedeutung und damit als dynamisches Weiterentwickeln der persönlichen Wirklichkeit gesehen. Lernen im didaktisch konstruktivistischen Kontext unterscheidet:

1. Konstruktion („Wir sind Erfinder unserer Wirklichkeit“),
2. Rekonstruktion („Wir sind die Entdecker unserer Wirklichkeit“) und
3. Dekonstruktion („Es könnte auch anders sein! Wir sind die Enttarner unserer Wirklichkeit!“).

Die grundsätzliche Ausrichtung ist: „Selbst erfahren, ausprobieren, untersuchen, experimentieren, immer in eigene Konstruktion ideeller oder materieller Art überführen und in den Bedeutungen für die individuelle Interessen-, Motivations- und Gefühlslage thematisieren.“ (Reich, 2008)

In der Perspektive der Rekonstruktion lautet die Frage: „Wer hat es damals so und wer hat es anders gesehen? Welche Handlungsmöglichkeiten haben Beobachter damals festgestellt und welche fallen uns hierzu ein? Welche unterschiedlichen Experten kommen zu welcher Aussage und wie stehen wir dazu?“ In dieser Perspektive wird gefragt, welche Motive der damalige Beobachter hatte um seine Festlegungen zu treffen. Faktenwissen steht dabei nicht im Vordergrund.

Die Dekonstruktion stellt sich die Frage der selbst vollzogenen Auslassungen, die möglichen anderen

Blickwinkel, die sich im Nachentdecken der Erfindungen anderer oder in der Selbstgefälligkeit der eigenen Erfindung so gerne einstellen. In dieser Perspektive will der Enttarner kritisch gegenüber den eigenen blinden Flecken sein.

Als idealtypischer Grundsatz für die konstruktivistische Didaktik gilt somit (Reich, 2008):

„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 und der nur durch Autorität oder Nicht-Hinterfragen oder äußerlich bleibende Belohnungssysteme gesetzt ist."

Der konstruktivistische Methodenbaukasten in der Software Engineering Ausbildung

Die Software Engineering Studenten müssen von Anfang an nicht nur in Erkenntnistheorie, sondern auch in Problemlösung vertraut werden (Ludewig, 2009). Dies wird durch den Perspektivwechsel von der Input- zur Output-Orientierung unterstützt, wobei durch den Einsatz geeigneter fachdidaktischer Methoden die Lernenden ihren Lernprozess in Selbststeuerung aktiv ausgestalten.

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 und Hagel, 2010). Im Folgenden werden die Erfahrungen mit den Methoden Planspiel und Briefmethode im Fach Software Engineering vorgestellt.

Planspiel

In Planspielen im Fach Software Engineering sollen Studierende durch Simulation einer Praxissituation einen möglichst realistischen und praxisbezogenen Einblick in Probleme und Zusammenhänge der methodischen Softwareentwicklung gewinnen, eigene Entscheidungen treffen und Konsequenzen ihres Handelns erfahren.

Planspiele erfordern zudem eine hohe Partizipation aller Beteiligten. Sie sollten auf eine Erhöhung der Handlungsfähigkeit in dem Sinne zielen, dass sie Konsens und Dissens, Entscheidungsabläufe und Transparenz bei der Bildung von Gruppenentscheidungen aufdecken und diskutierbar werden lassen (Reich, 2008). Die Studierenden werden im Planspiel mittels aktivierender Methoden beteiligt und in ein Software-Projekt involviert. Die Aufgabenstellung eines Softwareprojekts und die einzunehmenden Rollen sind vorgegeben. Das Ergebnis des Planspiels bleibt insofern offen, als dass die Lernenden verschiedene Lösungswege auffinden können.

Im Curriculum des Bachelorstudiengangs Mechatronik der Hochschule Regensburg wird im vierten Semester die Vorlesung Software Engineering im Umfang von 2 SWS/3 ECTS und im fünften Semester das Praktikum/Seminar als Blockveranstaltung Software Engineering im Umfang von 4 SWS/5 ECTS angeboten. Das Praktikum/Seminar Software Engineering verlangt also eine Arbeitszeit von 150 Stunden.

Die Blockveranstaltung Software Engineering (50 Stunden) hat zwei vorgelagerte Vorlesungstermine (je 4 Stunden) zur Vorbesprechung. Danach beginnt schon eine selbstgesteuerte Arbeitsphase der Studierenden. Als Vor- und Nachbereitungszeit der Studierenden verbleiben damit 92 Stunden.

Alle Studierenden haben als Vorkenntnisse die Programmiersprachen C und C++ (10 SWS V+Ü), sowie Mikrokontrollertechnik (6 SWS V+Ü). Sie haben dagegen keine Erfahrung über die bei der Programmierung hinausgehenden Schritte der Software-Entwicklung. Praktika und Projekte sind deshalb für die Lehre von Software Engineering zentral (Ludewig, 2009).

Die Vorlesung Software Engineering bereitet auf eine schriftliche Prüfung vor. Dagegen wird die Blockveranstaltung Praktikum/Seminar Software Engineering mit einem studienbegleitenden Leistungsnachweis abgeschlossen. Der studienbegleitende Leistungsnachweis wird erbracht durch die Vorbereitung eines Fachvortrages, die Vorbereitung eines Posters, die erfolgreiche Mitwirkung am Planspiel und die Erstellung von Arbeitsprodukten, wie qualitätsgesicherter Dokumentationen, sowie funktionierender Software.

Die Blockveranstaltung Praktikum/Seminar Software Engineering an der Hochschule Regensburg findet an fünf Tagen statt. Zwei Lehrende gestalten mit Pair-Teaching als interdisziplinäres Team (Informatiker und Pädagoge/Projekttrainer) diese Veranstaltung. Insbesondere diese Verzahnung im Führungstandem lässt für die Studierenden das Wechselspiel zwischen Konkurrenz versus Zusammenarbeit/Dialog in einem Klima offener Kommunikation sichtbar werden. Eine Übertragung auf die eigene Situation im Planspiel wird möglich.

Während in den ersten beiden Tagen Fachthemen erarbeitet und vermittelt werden, werden in den letzten drei Tagen in einem Planspiel die Kenntnisse und Fertigkeiten in den Fachthemen vertieft und angewendet.

Bereits vier Monate vor der Blockveranstaltung finden zwei Vorbesprechungen im Umfang von jeweils 4 Stunden mit den Studierenden statt. Dabei

werden Fachthemen zur Vorbereitung der Blockveranstaltung an die Studierenden vergeben. Diese Fachthemen werden von den Studierenden eigenständig bearbeitet. Als Ergebnis werden die Fachvorträge als Folienpräsentation und Poster in die Lernplattform moodle abgelegt. Der Lehrende gibt zu diesen Ergebnissen Rückmeldung in moodle.

Den Studierenden werden zusätzlich Literaturhinweise zur Bearbeitung der Aufgaben in der Lernplattform moodle zur Verfügung gestellt. Die Lernplattform ist vor und während des Planspiels im Einsatz.

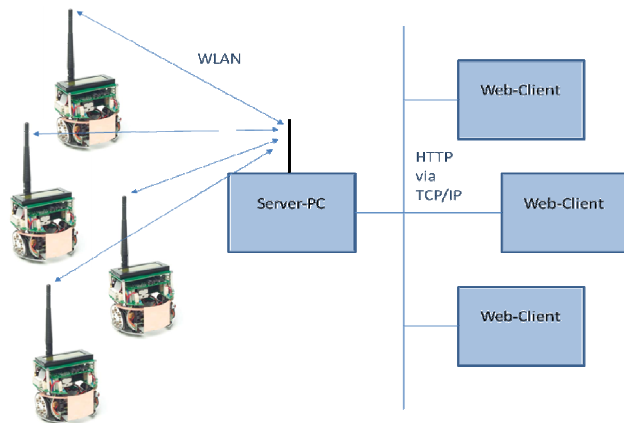


Abbildung 1: Software-Architektur mit c't-Bot, Server-PC und Web-Clienten für das Planspiel

Das durchgeführte Planspiel im Fach Software Engineering behandelt eine Projektaufgabe mit dem Embedded Roboter System c't-Bot. In dieser Aufgabe soll eine Fernsteuerung- und Fernüberwachung des Roboters c't-Bot über einen Server-PC und zusätzlich über Web-Clienten, also Browserapplikationen, erstellt werden (Abbildung 1). Bibliotheken und einfache Beispiele liegen bereits vor.

Zur Durchführung eines Planspiels im Software Engineering müssen folgende Spielmaterialien bereitgestellt werden:

- Eine Fallstudie, in der kurz die vorgegebene Softwareaufgabe skizziert wird und Software-Bibliotheken, sowie bestehende exemplarische Teillösungen vorgegeben werden.
- Eine Arbeitskarte mit Erläuterungen zum Verlauf des Softwareprojekts (Spielverlauf).
- Rollenkarten, durch welche den Teilnehmern spezifische Rollen übertragen werden (Die Softwareentwicklungsprozesse V-Modell 97 und V-Modell XT wurden bereits in der Vorlesung angesprochen.). Die Studierenden nehmen damit die Positionen einer Rolle (Projektleiter, Software Entwickler, Qualitätsmanager, Konfigurationsmanager, …) an.
- Ereigniskarten, die als Impulskarten durch den Spielleiter in die Gruppen gereicht werden können (beispielsweise die Änderungen von Anforderungen).
- Quellen und Literatur

Die einzelnen Phasen des Planspiels bestehen aus:

1. Spieleinführung

Die Semestergruppe der Studierenden wird zu Beginn des Planspiels in mehrere Planspielgruppen mit jeweils ca. 10 Studierenden aufgeteilt. Der Lehrende gibt die Ausgangslage schriftlich vor und klärt Verständnisfragen. Das Spielmaterial wird vorgestellt.

2. Informations- und Lesephase, Rollenverteilung

Die Gruppen erhalten die Rollen- und Arbeitskarten. Das Arbeitsmaterial wird durchgelesen und auftretende Verständnisfragen werden geklärt. Die Teambildung und Rollenverteilung wird von den Lehrenden begleitet.

3. Meinungsbildung und Strategieplanung innerhalb der Gruppe

Die Informationen werden gruppenintern strukturiert und die Projektaufgabe der Softwareentwicklung wird analysiert.

4. Interaktion zwischen den Rollen

In dieser intensivsten Spielphase agieren die Rollen der jeweiligen Planspielgruppe miteinander. Interessenkonflikte zwischen den Rollen treten auf. Diese Interessengegensätze sind typisch bei der Durchführung eines Planspiels. Durch Ereigniskarten kann der Spielleiter nun gezielte Impulse und Veränderungen ins Spiel bringen. Alle Planspielgruppen treffen sich zweimal am Tag zu einem Jour Fix. Die Rolleninhaber müssen dabei unter Zeitdruck Entscheidungen treffen.

5. Vorbereitung eines Plenums / Konferenz

Jeder Rollenträger/Positionsinhaber der jeweiligen Gruppe trägt intern seine Ergebnisse zusammen und verarbeitet und bewertet in dieser Phase die erreichten Ergebnisse.

6. **Durchführung eines Plenums / Konferenz**

Die Ergebnisse des Software-Projektes werden aus der Perspektive des jeweiligen Rollenträgers vorgestellt. Eine Demonstration mit dem realen software-intensiven System eines Roboters ist gewünscht.

7. Spielauswertung

Auswertung des Spielverlaufs mit dem Lehrenden als neutralen Moderators. Diese Reflexion über den

eigenen Lernprozess ist ein weiteres Merkmal eines Planspiels.

Während der Blockveranstaltung Praktikum/ Seminar Software Engineering stehen ausreichend Seminar- und Rechnerräume für die einzelnen Planspielgruppen zur Verfügung.

Der Ergebnispräsentation am Ende des Planspiels folgt eine Reflexion über den Lernprozess.

In der Reflexion wurden folgende Erfahrungen gesammelt und evaluiert:

- Die Lernthemen können von den Studierenden mitbestimmt werden (Rolleninhaber bereiten Themen vor)
- Der Lehrende übt als Spielleiter keine dominante Rolle aus, sondern ist Begleiter des Lernprozesses und berät bei Rückfragen (Aviram, 2000).
- Offene Form des Lernens ermöglicht die einzelnen Aufgaben zu differenzieren und zu individualisieren (Macke, 2009).
- Qualitätssicherung durch Literaturvorgaben, sowie Begleitung und Rückkopplung mit der Lernplattform moodle, - auch schon vor der Blockveranstaltung.
- Eigene Lernprojekte konnten aus der Vorbereitung der Fachthemen eingebracht werden.
- Die Lernorganisation des Planspiels lässt mehrere Lernwege offen, - Anknüpfung an die Lebens- und an die Praktikumserfahrung der Lernenden.
- Förderung der Handhabung verschiedenster Arbeitstechniken.
- Die Lerninhalte sind mit dem Anwendungsfall der Projektarbeit fassbar reduziert.
- Die angebotenen Lerninhalte können selbsttätig erschlossen werden.
- Handlungsbezogene Problemstellungen im Planspiel sind explizit Thema in der Blockveranstaltung.
- Bereichsübergreifendes Denken und Handeln wird gefördert, ebenso wie ein Verständnis für gruppendynamische Prozesse und ihre Auswirkungen.
- Komplexe Themen, wie Projektmanagement, Qualitätssicherung und Konfigurationsmanagement können in der zur Verfügung stehenden Zeit nur in grundlegender Weise vermittelt werden. Eine Vertiefung kann für Mechatronik-Studierende erst im Masterangebot erfolgen.
- Jede Planspielgruppe entwickelt eine andere Kultur.
- Lernen in multiplen Kontexten
- Mit Verlassen des 90-Minutenrythums entsteht Raum, Zeit und Gelassenheit zum Lernen.

Das Planspiel beinhaltet eine große Menge anderer Methoden und Techniken (Methodeninterdependenz), in denen sich der Studierende üben kann. Im Einzelnen sind dies die Arbeitsform der Gruppenarbeit, Strukturierung der Gruppenarbeit durch Moderation, Ideenentwicklung durch Clustering und Concept Learning, sowie Feedback zur Klärung von Gruppenkonflikten.

Der Unterschied zu Lernformen wie der Projektarbeit besteht darin, dass es noch stärker die Entwicklung von Handlungs- und Entscheidungskompetenzen und das Einüben entsprechender Verhaltensweisen betont (Markowitsch, 2004). Die Begründung von Architekturentscheidungen, die Auswahl möglicher Alternativen in Design und Implementierung, die Festlegung eines Testkonzeptes, aber auch die Ausgestaltung qualitätssichernder Review-Sitzungen sind als Beispiele zu nennen. Diese Beispiele finden sich zwar auch in Projektarbeiten wieder, aber im Planspiel wird die soziale Interaktion der Rolleninhaber und die gemeinsame Reflexion über den Lernprozess am Spielende als methodisches Merkmal genannt. Insofern kann die dargestellte Lernform als projektorientiertes Planspiel klassifiziert werden.

An der Blockveranstaltung Software Engineering haben bereits Semestergruppen mit 20 bis 60 Studierenden teilgenommen.

Mithilfe eines standardisierten Fragebogens konnten Werte für die Zufriedenheit der Studierenden mit der Blockveranstaltung Software Engineering ermittelt werden. Insbesondere wurden Fragen zur Veranstaltung selbst, den technischen Lerneinheiten und zur Begleitung durch den Lehrenden gestellt. Bei der Zufriedenheit handelt es sich um Einschätzungen der Beteiligten selbst, d.h. die relativen Werte für die Zufriedenheit sind sehr repräsentativ und spiegeln die Stimmungen angemessen wieder. An der Umfrage nahmen 80% aller Studierenden teil, sodass die Ergebnisse für den ganzen Kurs geltend gemacht werden können. Die Evaluationsergebnisse waren durchweg positiv.

Kritisch zu bewerten ist, dass die Studien- und Prüfungsordnung für Bachelorstudierende der Mechatronik insgesamt nur 2 SWS Vorlesung und 4 SWS Praktikum/Seminar für das Lehrgebiet Software Engineering vorsieht. Deshalb können die Studierenden sich nur grundlegende Kenntnisse in der disziplinierten Software-Entwicklung bei Anwendung von Softwareprozessmodellen aneignen.

Briefmethode

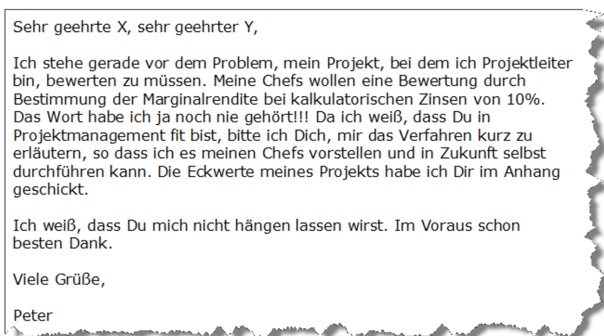
Auch die Briefmethode stammt aus dem Konstruktivistischen Methodenpool (Reich, 2008). Diese Me-

thode wird häufig im Deutschunterricht der Schulen, aber auch in Geschichte und Literatur eingesetzt. Wir wollten untersuchen, ob sich diese Methode auch für den Einsatz in einem technischen Fach, wie Software Engineering an einer Hochschule eignet. Dabei wurden seitens der Dozierenden mehrere Ziele verfolgt: Die Studierenden sollten

- einen Sachverhalt, den sie sich vorher erarbeitet hatten, wiedergeben können und
- lernen, Briefe mit technischer Information zu verfassen.

Das Erstellen technischer Dokumentation kommt aus Sicht der Autoren in den Lehrveranstaltungen zum Software Engineering zu kurz und beschränkt sich meist auf die Erstellung von UML-Diagrammen. Texte mit technischem Inhalt, oder gar Benutzerdokumentation wird sehr selten im Rahmen der Software Engineering-Ausbildung in den Hochschulen verfasst. Ein Versuch, wie eine Ausbildung im Technischen Schreiben aussehen kann, findet man in (Schmidt, 2009). Auch werden an verschiedenen Hochschulen inzwischen explizit Veranstaltungen wie „Software Engineering und Technisches Schreiben" angeboten.

In der Vorbereitung muss zunächst ein hinreichend komplexer Sachverhalt gefunden werden, der sich in Briefform gut vermitteln lässt. In unserem Beispiel bat der Dozierende die Studierenden um ein Antwortschreiben auf einen fiktiven Brief (siehe Abbildung 2). Im Brief bittet ein Freund der Studierenden um Hilfe bei einer betriebswirtschaftlich-mathematischen Aufgabe aus dem Projektmanagement.



Sehr geehrte X, sehr geehrter Y,

Ich stehe gerade vor dem Problem, mein Projekt, bei dem ich Projektleiter bin, bewerten zu müssen. Meine Chefs wollen eine Bewertung durch Bestimmung der Marginalrendite bei kalkulatorischen Zinsen von 10%. Das Wort habe ich ja noch nie gehört!!! Da ich weiß, dass Du in Projektmanagement fit bist, bitte ich Dich, mir das Verfahren kurz zu erläutern, so dass ich es meinen Chefs vorstellen und in Zukunft selbst durchführen kann. Die Eckwerte meines Projekts habe ich Dir im Anhang geschickt.

Ich weiß, dass Du mich nicht hängen lassen wirst. Im Voraus schon besten Dank.

Viele Grüße,

Peter

Abbildung 2: Brief

Die Studierenden, die eine Marginalrenditerechnung schon mehrmals in Übungen gelöst hatten, sollten "ihrem Freund" in Briefform antworten, also durch selbstständiges Handeln ein neues Produkt, nämlich die technische Lösung der Aufgabe als Brief erstellen.

Ergebnis war, dass lediglich 20% der Studierenden einen Antwortbrief verfasst hatten. Auf Nachfrage seitens des Dozierenden stellte sich heraus, dass viele mit einer so gestellten Aufgabe nichts anfangen konnten. Daher ist es speziell in technischen Fächern sinnvoll, den Einsatz der Methode zu motivieren. Ist den Studierenden der Sinn dieser Art Aufgabenstellung transparent, erhöht sich der Rücklauf beträchtlich.

Diejenigen Studierenden, die das Antwortschreiben verfasst hatten, haben sehr gut verständliche technische Dokumente abgeliefert. Dabei wählten sie selbstständig und unabhängig voneinander unterschiedliche Formate für das Antwortschreiben. Die einen antworteten mit einem kommentierten Excel-Sheet, die anderen mit Word-Dokumenten. Der Dozierende hat auf die Antwortbriefe wieder per Brief individuell Feedback gegeben.

In der Reflexion der Methode mit den Studierenden stellte sich heraus, dass sie das Verfassen des Briefes schwierig fanden: Sie mussten

- einen technischen Text sauber formulieren und übersichtlich strukturieren,
- erkennen, wo sie noch Lücken hatten, sich die Terminologie nochmals aneignen,
- ohne direkte Kommunikation einen Sachverhalt schildern und
- Empathie entwickeln für jemanden, der die gestellte Aufgabe nicht eigenständig lösen kann.

Positive Rückmeldungen seitens der Studierenden waren

- Sie verstehen das Thema jetzt, wo sie es jemand anderem erklären mussten wesentlich besser.
- Sie fanden das konstruktive Feedback des Dozierenden sehr hilfreich.

Das durchweg positive Feedback der Studierenden, die diese Aufgabe gelöst hatten ermutigt den Dozierenden, diese Methode zukünftig häufiger einzusetzen, damit die Studierenden mit dieser Art der Aufgabenstellung vertraut werden. Damit wird der Rücklauf bei der Methode sicherlich erhöht.

Eine spezielle Ausprägung der Briefmethode ist ein Online-Forum, das die Studierenden in Eigeninitiative eingerichtet haben. Dort sind Studierende **und** Dozierende angemeldet. Studierende können hier Fragen, Anregungen oder Kritik jederzeit äußern. Speziell zu technischen Problemen kommen häufig Fragen im Forum und werden oft von den Studierenden selbst beantwortet, so dass diese einiges an Erfahrung in technischer Dokumentation aufbauen. Allerdings ist der Schreibstil im Forum nicht immer technisch und formal korrekt, was auch nicht beabsichtigt ist. Auch Anrede und Grußformel fehlen. Allerdings kann durch den Einsatz von Emoticons eine Aussage unterstrichen und die Kommunikation aufgelockert werden. Die-

se Möglichkeit der Hilfe zur Selbsthilfe wird auch seitens der Dozierenden sehr begrüßt und unterstützt: Auf Fragen wird geantwortet und Kritik und Anregungen zukünftig berücksichtigt.

Damit das Forum funktioniert und regelmäßig benutzt wird, ist es notwendig, dass die Dozierenden regelmäßig und zeitnah antworten. Es wird seitens der Studierenden eine fast ständige Verfügbarkeit erwartet, auch wenn das nicht direkt kommuniziert wird. Außerdem muss gewährleistet werden, dass alle Studierenden Zugang zum Forum erhalten. Das muss seitens des Dozierenden überprüft werden, um eine Gleichbehandlung der Studierenden zu gewährleisten, da über das Forum zusätzlich zur Vorlesung Informationen seitens der Dozierenden verteilt werden.

Diese Art Forum läuft schon seit mehreren Jahren und der rege Gebrauch seitens vieler Studierender zeigt, dass sich dieses Medium bewährt hat.

Eine Möglichkeit, die Briefmethode in größerem Rahmen einzusetzen, wäre die Erstellung eines Wiki zusammen mit den Studierenden. In diesem könnten Studierende die Sie interessierenden Themen für alle technisch dokumentieren.

Zusammenfassung und Ausblick

Planspiel und Briefmethode sind zwei konstruktivistische Methoden, die nach Meinung der Autoren sehr gut für die Ausbildung im Softwareengineering geeignet sind. Die Lernenden lassen sich für das Planspiel einfacher motivieren als für die Briefmethode.
Planspiele im Software Engineering konfrontieren möglichst realistisch mit einer Praxissituation. Die Studierenden können dabei zum kreativen, weitgehend autonomen und selbstorganisierten Handeln in Bezug auf konkrete Probleme und deren Lösung motiviert werden und nehmen dabei unterschiedliche Positionen in einem komplexen Softwareentwicklungsprozess ein.

Die Briefmethode führt zu besserem Verfassen technischer Dokumentation und eignet sich sehr gut, um sich ein Thema anzueignen, oder zu wiederholen. Auch der Spezialfall eines Forums für Studierende und Dozierende wird positiv aufgenommen.

Die Reflexion der Studierenden über den eigenen Lernprozess kann zukünftig durch die Führung eines individuellen Lernjournals unterstützt werden.

Die Dozierenden werden beide Methoden zukünftig häufiger einsetzen. Außerdem sind sie durch die gemachten Erfahrungen mit dem konstruktivistischen Methodenpool hoch motiviert, weitere Versuche mit diesen Methoden für die Software Engineering-Ausbildung durchzuführen.

Literatur

Aviram, A. (2000): Beyond Constructivism: Autonomy-Oriented Education. Studies in Philosophy and Education, 19: 465-489., Kluwer Academic Publishers.

Dewey, J. (1910): How we think (deutsch: Wie wir denken, Zürich 1951).

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.

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.

Markowitsch, J., Messerer, K., Prokopp, M. (2004): Handbuch praxisorientierter Hochschulbildung, WUV Universitätsverlag, Wien.

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.

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

Schmidt, G., Hollweg, G. (2009): Ein integrativer interdisziplinärer Lehrversuch: Softwareengineering und Technisches Schreiben, in Jaeger, U. (Hrsg.) und Schneider K. (Hrsg.): Softwareengineering im Unterricht der Hochschulen: SEUH 11, Hannover 2009, dpunkt Verlag.

Service-Stelle Bologna (2004): Hochschulrektorenkonferenz - Texte und Hilfestellungen zur Umsetzung der Ziele des Bologna-Prozesses an deutschen Hochschulen, Beiträge zur Hochschulpolitik.

Siebert, H. (2009): Selbstgesteuertes Lernen und Lernberatung, ZIEL, Augsburg.

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

Teamentwicklung in studentischen Projekten

Doris Schmedding, TU Dortmund

Doris.schmedding@tu-dortmund.de

Zusammenfassung

Teamfähigkeit ist eine der Anforderungen an Software-Entwickler, die sehr häufig in Stellenanzeigen genannt wird. Unter Teamfähigkeit wird bei Wikipedia die Fähigkeit verstanden, mit anderen zusammen sozial zu agieren und sich und sein Können im Sinne einer Gruppenaufgabe optimal einzubringen.

Studentische Software-Entwicklungsprojekte bieten die Möglichkeit durch Learning-by-doing neben den technischen Fähigkeiten auch die Softskills der Studierenden auf dem Gebiet der Teamarbeit zu trainieren. Dieser Artikel stellt in einem Software-Praktikum erfolgreich erprobte Methoden vor, mit denen Lehrende ohne großen Aufwand die Teamentwicklung in studentischen Projekten unterstützen und die Reflexion der Erfahrungen anleiten können, um daraus neue Impulse für eine bessere Zusammenarbeit zu gewinnen.

Einleitung

Das Software-Praktikum an der TU Dortmund (https://sopra.cs.tu-dortmund.de/wiki/) ist eine langjährig etablierte Lehrveranstaltung, die neben der Anwendung von Methoden und Verfahren aus der Software-Technik in Software-Entwicklungsprojekten darauf abzielt, die Teamfähigkeit der Studierenden zu verbessern. In Gruppen zu je 8 Studierenden werden unter Anleitung eines erfahrenen Tutors nacheinander zwei Software-Projekte durchgeführt. Gleichzeitig nehmen 5-12 Gruppen am Praktikum teil. Das Praktikum hat laut Prüfungsordnung einen Umfang von 4 SWS. Konkret heißt das, dass sich eine Gruppe an zwei Terminen pro Woche im Seminarraum trifft, um am Projekt zu arbeiten. Daneben arbeiten die Studieren auch zuhause oder im Rechnerpool an der Universität an dem Projekt.

In einem Projekt sind technische, fachliche, organisatorische und soziale Probleme zu lösen. Projekte scheitern weniger an mangelndem technischen oder fachlichem Wissen, vielmehr stellt die Lösung von organisatorischen und sozialen Problemen eine größere Herausforderung dar (Fleischmann u.a., 2005). Da ich diese Beobachtung aus eigener Erfahrung nur bestätigen kann, möchte ich die Studierenden ebenso, wie wir die Einarbeitung in bis dahin noch unbekannte Werkzeuge wie SVN oder einen GUI-Builder durch speziell vorbereitete Tutorials unterstützen, auch auf die bis dahin noch unbekannte Arbeit in einem Projektteam vorbereiten. Für das Lernen ist es wichtig, die eigenen Erfahrungen mit den neuen Arbeitstechniken zu reflektieren und bei Bedarf die Vorgehensweisen zu verändern. Nur so lassen sich aus den Erfahrungen im ersten Projekt Konsequenzen für das zweite Projekt ziehen. In (Lewerentz u. Rust, 2001) wird ausführlich auf die Bedeutung der Reflexion der Erfahrungen in der Projektarbeit eingegangen. Den Vorschlag der Autoren, neben einem gemeinsamen Reflexionsbericht zusätzlich am Ende jedes Teilprojekts persönliche Reflexionsberichte von allen Studierenden einzufordern, halte ich für wenig praktikabel. Zumindest Dortmunder Informatik-Studierende möchten lieber Java-Code als Selbstreflexionen schreiben.

In diesem Artikel gebe ich zunächst eine Einführung in die Teamentwicklung. Dann stelle ich zwei Beispiele für Übungen zur Teambildung vor, mit denen ohne großen Aufwand die Teambildung unterstützt werden kann. Zu einer derartigen Übung gehört auch die Reflexion der Erfahrungen. Das Thema Reflexion wird vertieft, indem eine Methode vorgestellt wird, die zur Halbzeit des Praktikums eingesetzt wird.

Teamentwicklung

Ein Team unterscheidet sich von einer Gruppe durch die Tatsache, dass die Mitglieder eines Teams gemeinsam eine Aufgabe lösen und/oder ein gemeinsames Ziel verfolgen. In diesem Sinne können wir also bei einem studentischen Software-Entwicklungsprojekt mit mehreren Studierenden von einem Team ausgehen.

Die Entwicklung von einer zufällig zusammengestellten Gruppe von Studierenden mit heterogenem Vorwissen zu einem gut funktionierenden Team läuft in einer Abfolge von Phasen ab (siehe Abb. 1), die in der Literatur in Anlehnung an die Teamuhr von Tuckman (Tuckman, 1965) gerne mit

den eingängigen Begriffen *Forming*, *Storming*, *Norming* und *Performing* bezeichnet werden.

Diese Phasen der Teamentwicklung werden bis auf die Forming-Phase bei neuen Aufgaben oder Zielen, oder wenn ein neues Mitglied zur Gruppe hinzukommt, wieder neu durchlaufen. Anstelle des Forming tritt dann die Reorganisation des Teams, das *Reforming* ein (Vigenschow u.a., 2009).

In (Marks, 2009) wird nicht nur erläutert, was in den einzelnen Phasen unter den Gruppenmitgliedern passiert, sondern es werden auch die Rolle und die Aufgaben der Lehrenden erläutert. Nachfolgend werden die einzelnen Phasen der Teamentwicklung vorgestellt, wie ich sie als Betreuerin von studentischen Software-Entwicklungsgruppen oft beobachten konnte.

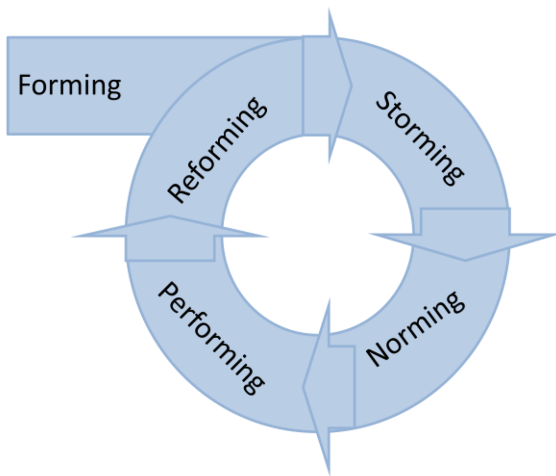


Abb.1: Phasen der Teamentwicklung

Anfangsphase (Forming)

Nach der Gruppenbildung befindet sich die Gruppe zunächst in der Anfangsphase, die von großer Unsicherheit geprägt ist. Die Aufgabenstellung ist noch unbekannt. Die Mitglieder der Gruppe kennen die Persönlichkeiten und Vorkenntnisse der anderen Gruppenmitglieder noch nicht. Der Leitung der Gruppe werden Autorität und soziale Fähigkeiten unterstellt, die diese nutzen kann, um die Gruppe über die schwierige Anfangssituation hinweg zu führen.

Konfliktphase (Storming)

Nach der Phase des Kennenlernens fühlen sich die Gruppenmitglieder sicher. Es folgt eine Phase der Auseinandersetzung um die Einflussmöglichkeiten in der Gruppe. Oberflächlich geht es dabei oft um Sachthemen. Viele Konflikte finden aber auf der Beziehungsebene statt, auf der es um Sympathie und Antipathie, um die eigenen und die fremden Werte und Einstellungen und um das Ansehen in der Gruppe geht, das auf vermuteter fachlicher Kompetenz und Vertrauen beruht. Das Eisbergmodell der Kommunikation (siehe Abb.2) lässt sich auf die Kommunikation in Gruppen übertragen (Virgenschow u.a., 2009) und hilft die Zusammenhänge und die Ursachen für langwierige Diskussionen zu verstehen.

Die Gruppe arbeitet an einer gemeinsamen Zieldefinition und Normen; Verhaltensregeln für die Gruppe werden festgelegt. Als Ergebnis der Konfliktphase entsteht eine informelle Hierarchie, in auch die Lehrenden einbezogen sind. Die Reaktion der Lehrenden, die die Ziele der Lehrveranstaltung vertreten müssen, wird ausgetestet.

Normierungsphase (Norming)

Die Normierungsphase ist geprägt durch die Entwicklung eines Wir-Gefühls im Projektteam. Nachdem ein praktikables Regelwerk zur Lösung von Konflikten erarbeitet worden ist, können Konflikte relativ reibungsfrei beigelegt werden. Wir beobachten, dass die Gruppe sich um die Integration von Außenseitern bemüht und gegenüber Außenstehenden abschließt. Dazu wird oft auch die Gruppenleitung gezählt. Insgesamt wächst die Arbeitsfähigkeit der Gruppe in dieser Phase stark an.

Arbeitsphase (Performing)

In der letzten Phase, der Arbeitsphase (Performing), fließt die gesamte Teamenergie in die Aufgabenbewältigung. Wegen des hohen Gruppenzusammenhalts sind nun auch Spitzenleistungen möglich. Die Arbeit wird innerhalb des Teams nach Effizienzgesichtspunkten verteilt und erledigt. Die Lehrenden benötigen nur noch wenig Aufwand, da sich die Gruppe weitgehend selbst organisiert und kontrolliert. Allerdings müssen Lehrende jetzt darauf achten, dass die Lehr- und Lernziele nicht aus den Augen verloren werden. In meiner Lehrveranstaltung gilt die Regel, dass jeder alle Aktivitäten zumindest ansatzweise einmal ausgeführt hat, was nicht unbedingt effizient im Sinne des Projektteams ist.

Projekt 0 - Übung zur Teambildung

Wenn das Zusammenwachsen einer Gruppe zu einem gut funktionierenden und produktiven Team so schwierig ist, stellt sich die Frage, wie Lehrende eine Gruppe bei der Teamentwicklung unterstützen können. In (Fleischmann, 2005) wird ein recht umfangreiches dreitägiges Teamtraining beschrieben, für das in unserem vierstündigen und einsemestrigen Praktikum leider keine Zeit ist. Auch mit geringerem Aufwand können die Lehrenden durch gezielte Übungen und den Anstoß zur Reflexion Einfluss auf die Teamentwicklung nehmen, wenn sie sich über die Abläufe und ihre eigene Rolle im Entwicklungsprozess der Gruppe bewusst sind.

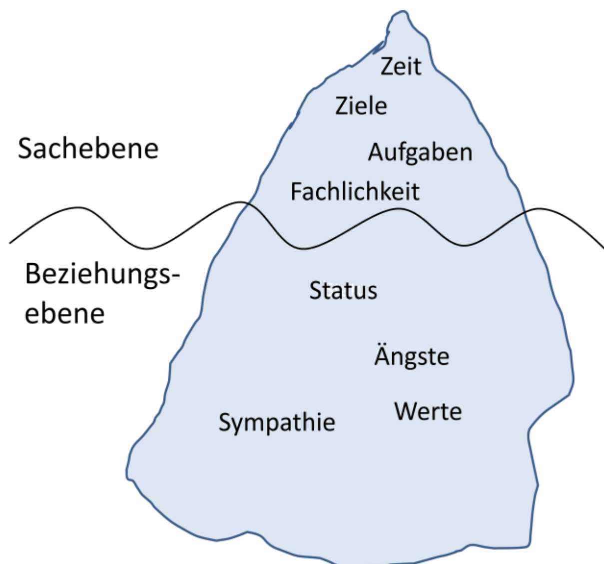


Abb. 2: Eisbergmodell der Kommunikation

In der Anfangsphase können Übungen zur Teambildung helfen, das Kennenlernen zu erleichtern und die Abläufe im Team zu veranschaulichen. Besonders geeignet für Software-Entwicklungsteams sind Übungen, die einen Projektcharakter haben und an denen sich die Analogie zu Software-Projekten gut zeigen lässt. Derartige Übungen bezeichne ich deshalb mit Projekt 0.

Turmbau

Sehr bewährt haben sich in meiner Arbeit im Software-Praktikum Bastel-Projekte wie der Turmbau als Einstieg in die Projektarbeit.

Aufgabenstellung: Aus einer vorgegebenen Anzahl von Blättern Papier soll z.B. ein möglichst hoher Turm gebaut werden. Zwei Studierende aus der Gruppe beobachten das Bauteam und berichten über ihre Beobachtungen. Die restlichen 6 Gruppenmitglieder bilden das Bau-Team. Eine Schere wird als Werkzeug zur Verfügung gestellt. Der Turm soll in 30 Minuten gebaut werden.

Als Beobachter kann man sehr schön die Phasen der Teamentwicklung verfolgen. Anfangs betrachten die Studierenden recht ratlos die Blätter und die Schere. Dann stehen meist konkurrierende Vorschläge von einzelnen Gruppenmitgliedern im Raum. Je nach Sympathie oder Vertrauen in Kompetenz der Vorschlagenden schließen sich einzelne Gruppenmitglieder einer Idee an. Dann wird entweder sehr lange diskutiert oder relativ schnell ein Regelwerk gefunden, um sich auf einen Lösungsvorschlag zu einigen. In der Bauphase sind die meisten aktiv, manche halten sich zurück. Es wird geschnitten, gefaltet, gerollt und zusammengesteckt. Am Ende ist die Gruppe meist mit ihrem Werk sehr zufrieden.

Durch das praktische Tun lockert sich schnell die Atmosphäre in der Gruppe. Ein Einstiegsprojekt bietet die Möglichkeit, die Persönlichkeit der Mitglieder kennen zu lernen. Entscheidungsprozesse können eingeübt werden.

Bevor man die Beobachter berichten lässt, sollten die Mitglieder des Bauteams nach ihren Erfahrungen befragt werden. Folgende Fragen bieten einen guten Einstieg in die Diskussion über den Problemlösungsprozess mit den Studierenden:

- Welche Verhaltensweisen haben der Gruppe bei der Lösung der Aufgabe geholfen?
- Welche Verhaltensweisen haben die Gruppe bei der Lösung der Aufgabe behindert?
- Wer hat sich am meisten beteiligt?
- Wer hat sich am meisten zurückgehalten?
- Wie habt Ihr die Diskussionen und den ganzen Lösungsprozess erlebt?
- Was hat mir/der Gruppe die Übung gebracht?
- Wie ist die Gruppe mit der vorgegebenen Zeit zurechtgekommen?

Die Rolle der Beobachter ist für die Reflexion des Geschehens wichtig. Als nicht unmittelbar am Bau Beteiligte spiegeln sie der Gruppe ihr Verhalten und ihre Vorgehensweise.

Die durchaus gewollte Analogie zum Software-Projekt mit *Planung, Entwurf* und *Realisierung* wird von den Studierenden durchaus gesehen. Man kann über Abweichungen der Realisierung von der Planung und dem Entwurf diskutieren. Auch die Themen *Zeitmanagement* und *knappe Ressourcen* werden in diesem Kurzprojekt sehr gut veranschaulicht.

Die Aufgabenstellung lässt sich durch erweiterte Anforderungen und andere Themengebiete leicht variieren. Das Konzept der *Modularisierung* lässt sich z.B. durch den Bau eine Brücke bestehend aus Pfeilern und einer Auflage aufgreifen. *Schnittstellen* lassen sich z.B. bei einer Kugelbahn gut integrieren, indem man explizit eine Steckverbindung zwischen getrennt zu entwickelnde Komponenten fordert. Die zusammengesetzte Kugelbahn eignet sich auch, um das Thema unabhängiges *Testen von Komponenten* in das Einführungsprojekt mit aufzunehmen.

Am Ende der Analyse der Übung sollte die Frage diskutiert werden, welche Konsequenzen die Erfahrungen aus Projekt 0 für das vor der Gruppe liegende Software-Entwicklungsprojekt haben sollen. Typischen Schlussfolgerungen sind, dass die Diskussion strukturierter erfolgen sollte und dass eine Sitzungsmoderation sinnvoll wäre.

Sin-Obelisk

Eine andere Art von Einstiegsaufgaben stellen gemeinsam von der Gruppe zu lösende Rätsel dar, mit denen man die Bedeutung des Informationsaustausches und die Kooperation üben kann. Ein Beispiel für diesen Typ von Aufgaben ist der sogenannte Sin-Obelisk

(http://www.spielekiste.de/archiv/diverses/komm/komm_004.shtml).

Auf vielen kleinen Kärtchen (>30) stehen jeweils Informationen geschrieben, die die Gruppe zur Beantwortung einer Frage (Lösung einer Aufgabe) benötigt. Eingestreut sind nutzlose Informationen und Fragen. Jedes Gruppenmitglied bekommt eine Reihe von Informationskärtchen. Nur durch Zusammentragen der relevanten Informationen und geschickte Aneinanderreihung gelingt es der Gruppe, die Eingangsfrage zu beantworten. Im Fall des Sin-Obelisken ist es die Frage nach dem Wochentag, an dem er fertig gestellt wird. Eine der benötigten Informationen ist z.B. die Höhe des Obelisken.

Für die gesamte Übung einschließlich Reflexion reicht eine Stunde völlig aus. Nach meiner Erfahrung findet eine Gruppe von Informatikstudierenden in etwa 10-15 Minuten die Lösung.

Wie beim Turmbau werden Beobachter bestimmt, die der Gruppe ihre Beobachtungen vortragen. Mit etwa den gleichen Fragen zum Problemlösungsprozess wie beim Turmbau kann die Diskussion mit den Studierenden initiiert werden. Offene Fragen, die reihum abgefragt werden, helfen, die Studierenden zu aktivieren. Durch die Fokussierung auf bestimmte Punkte kann die Aufmerksamkeit der Beobachter gelenkt werden.

Ein Ziel der Übung ist es, den Umgang mit verstreuter Information im Problemlösungsprozess zu trainieren. Die Analogie zur Situation in einem neu zusammengestellten Team für ein Software-Entwicklungsprojekt besteht z.B. in den unterschiedlichen Vorkenntnissen, die gemeinsam zur Lösung der Aufgabe genutzt werden. Daneben kann man Kooperationsbereitschaft und Führungsverhalten und den Umgang mit Konflikten bei der Problemlösung in der Gruppe studieren. Auch lassen sich sehr schön die Phasen der Teamentwicklung beobachten, von Unsicherheit über Erarbeitung eines Regelwerks zum gut funktionierenden Interagieren.

Konstruktive Reflexion der Erfahrungen

Wenn Lehrende in studentischen Software-Entwicklungsprojekten das Ziel „Stärkung der Teamfähigkeit" ernst nehmen und mehr erreichen wollen als nur eine Lockerung der Atmosphäre und eine Stärkung des Selbstbewusstseins durch eine schöne und schnelle Lösung, muss bei den Studierenden eine eigene Reflexion der Erfahrungen in Gang gesetzt werden. Dazu ist zumindest ein kleiner Einblick in die Theorie der Gruppendynamik notwendig, den ich in der Einführungsveranstaltung zum Software-Praktikum gebe.

Bei der Thematisierung gruppendynamischer Prozesse geht man davon aus, dass neben der Sachebene, die sich mit den Aufgaben und Zielen der Lehrveranstaltung, den Anforderungen und den technischen Fragen im Projekt beschäftigt, auch die Beziehungsebene existiert, die die Sympathie der Gruppenmitglieder untereinander, ihre Werte und Normen und ihre Ängste beinhaltet (vgl. Abb.2). In der Diskussion von Teamprozessen mit Betroffenen setzen wir uns als Lehrende der Software-Technik dem Risiko aus, schwerwiegende Probleme Einzelner und in den Beziehungen untereinander an die Oberfläche zu befördern. Die sorgfältige Behandlung derartiger Probleme braucht sehr viel Zeit und sollte ausgebildeten Trainern überlassen werden (Vigenschow u.a., 2009). Nicht-Psychologen kommen deshalb bei der Beschäftigung mit der Gruppendynamik, der Prozessanalyse und dem Rollenverhalten schnell an ihre Grenzen.

Beim Einholen von Kritik an den Abläufen in der Gruppe und an der Lehrveranstaltung und damit auch an der eigenen Rolle als Lehrende sollte man sich darüber klar sein, dass derartige Rückmeldungen auch für einen selbst unangenehm sein können. Daher sollte zunächst auf gewisse Spielregeln, sogenannte Feed-Back-Regeln, hingewiesen werden (Vigenschow u. Schneider, 2007). So haben wir die Chance, mit Kritik konstruktiv umzugehen.

Da im Software-Praktikum an der TU Dortmund von den Studierenden zwei Software-Entwicklungsprojekte durchgeführt werden, bietet es sich an, nach dem ersten Projekt der Erfahrungen zu reflektieren. Als Leiterin der Lehrveranstaltung besuche ich alle Gruppen, um mit den Studierenden anhand der folgenden Fragen ihre Erfahrungen zu diskutieren:

- Was ist im 1. Projekt gut gelaufen?
- Was muss im 2. Projekt besser werden?

Die Antworten lasse ich auf Karten notieren und clustern. Zugelassen sind sowohl Kritik an der Technik, dem Lehrkonzept als auch an der Arbeit im Team. Die Karten bieten den Vorteil, dass auch die Stillen zu Wort kommen. Aus Häufungen lässt die besondere Relevanz eines Punktes ablesen.

In der Regel wird das Klima in der eigenen Gruppe als kooperativ erlebt und auch in dieser Runde genannt. Die gegenseitige Beteuerung der guten Teamarbeit verstärkt noch einmal das gute Klima. Allerdings werden Schwächen in der Zusammenarbeit durchaus auch gesehen und thematisiert. Daraus können selbst formulierte Regeln für ein geändertes Arbeitsverhalten abgeleitet werden. Beispielsweise soll beim Auftreten von Problemen den anderen früher Bescheid gegeben werden, damit sie schneller unterstützend eingreifen können. Oder beim Einchecken in das Repository soll immer ein Kommentar geschrieben werden, damit die anderen wissen, was geändert wurde. Ein der-

artiges selbst erarbeitetes Regelwerk ist bei der Kooperation im Team äußerst hilfreich.

Auch als Verantwortliche für das Praktikum kann ich aus diesen Sitzungen viel mitnehmen. Das sind Verbesserungsvorschläge für die technische Ausstattung, die eingesetzten Werkzeuge, aber auch für das Konzept der Veranstaltungen, z.B. Anregungen wie die Einarbeitung in Tools besser unterstützt werden kann.

Ein derartiges Reflexionsgespräch mit einer Gruppe dauert etwa eine Stunde.

Fazit

In einem Software-Entwicklungspraktikum stehen naturgemäß die Inhalte im Vordergrund, die sich „direkt" mit der Software-Entwicklung beschäftigen. Auf diesem Gebiet sind wir als Software-Techniker Fachleute und das wollen die Studierenden von uns lernen. Dennoch wissen wir aus langjähriger Berufserfahrung und Beobachtung der studentischen Teams, welche Bedeutung die Zusammenarbeit der Gruppenmitglieder für den Erfolg oder Misserfolg eines Projekts hat.

Neben den bereits zitierten Büchern von Vigenschow und anderen, die sich mit den Softskills einmal aus der Sicht der Entwickler (Vigenschow u. Schneider, 2007) und einmal aus der Sicht eines Projektleiters (Vigenschow u.a., 2009) beschäftigen, verdeutlicht der lesenswerte Ratgeber von Hedwig Kellner (Kellner, 2006), dass neben der fachlichen Kompetenz auch die Teamfähigkeit sehr wichtig für eine Karriere in der Software-Entwicklung ist. Arbeitgeber erwarten von Mitarbeitern neben fachlicher Kompetenz auch soziale Kompetenz. In komplexen Projekten werden Teamworker gebraucht, die ihre guten Einzelleistungen zu Gesamtleistungen kombinieren.

Ganz konkret wird in (Hölzle u. Grünig, 2006) gezeigt, dass für ein erfolgreiches Projektmanagement soziale Sensibilität benötigt wird. Anhand des Eisbergmodells (Abb.2) wird verdeutlicht, dass sich die wirklichen Gründe für Ressourcenprobleme "unter der Wasseroberfläche" einer guten Planung verbergen, die in der Regel nur einen kleinen Anteil am Gesamterfolg eines erfolgreichen Projektmanagements beiträgt.

Wenn also soziale Kompetenz für das Gelingen von Projekten so wichtig ist, sollte ihre Bedeutung auch in den Lehrveranstaltungen thematisiert werden. Für das Themengebiet *Teamarbeit* eignen sich insbesondere studentische Software-Entwicklungsprojekte, da sich hier Theorie und Praxis gut kombinieren lassen. Die Studierenden können sich selbst und die anderen Teammitglieder beobachten und ihr neues Wissen direkt ausprobieren.

In (Fleischmann u.a., 2005) und in (Lewerentz u. Rust, 2001) werden zwei Beispiele vorgestellt, wie im Rahmen von Software-Praktika mit relativ großem zeitlichem und personellem Aufwand erfolgreich daran gearbeitet wird, die Softskills der Studierenden zu verbessern. Ich habe hier gezeigt, dass auch mit weitaus weniger aufwendigen, gut platzierten Maßnahmen die Teamarbeit unterstützt werden kann.

Die vorsichtige Integration einiger Inhalte in eine bestehende und etablierte Veranstaltung, wie ein Kurzvortrag über Teamarbeit, ein Teamentwicklungsprojekt und die angeleitete Reflexion der Erfahrungen, ist einfach zu realisieren, kostet fast keine zusätzliche Zeit, lohnt sich aber auf jeden Fall, denn diese Investition wird durch ein besseres Arbeitsklima belohnt und verbessert die Vorbereitung der Studiereden auf den Beruf.

Literatur

Fleischmann, A., Spies, K., Neumeyer, K. (2005): Teamtraining für Software-Ingenieure. In: Löhr, K.-P., Lichter, H. (Hsg.): SEUH 9, 26-40.

Kellner, H. (2006): Soziale Kompetenz: Für Ingenieure, Informatiker und Naturwissenschaftler. Hanser.

Lewerentz, C., Rust, H. (2001): Die Rolle der Reflektion in Softwarepraktika. In: Lichter, H., Glinz, M. (Hsg.): SEUH 7, 73-86.

Marks, F. (2009): Gruppendynamik und Hochschulunterricht – Gruppendynamische Prozesse im Seminar. Erschienen in Berendt, B., Voss, P., Wildt, J., Tremp, P. (Hrsg.): Neues Handbuch Hochschullehre.

Tuckman, B.W. (1965): Developmental sequences in small groups. Psychological Bulletin, 63, 348-399.

Vigenschow, U., Schneider, B. (2007): Soft Skills für Softwareentwickler. dpunkt.verlag.

Vigenschow, U., Schneider, B., Meyrose, I. (2009): Soft Skills für IT-Führungskräfte und Projektleiter. dpunkt.verlag.

Hölzle, P., Grünig, C., (2006): Projektmanagement. Professionell führen - Erfolge präsentieren. Haufe.

Dynamische Klassendiagramme - Nutzung der Metapher vom „Konsumieren und Produzieren“ in BlueJ

Axel Schmolitzky und Chris Stahlhut, Universität Hamburg
{schmolit, 6stahlhu}@informatik.uni-hamburg.de

Zusammenfassung

Konsumieren und Produzieren sind zwei Seiten derselben Medaille. Wir untersuchen seit einigen Jahren die Metapher vom Konsumieren und Produzieren (kurz: K&P-Metapher) im Umfeld der Lehre objektorientierter Programmierung.

In diesem Artikel stellen wir eine Erweiterung der für die Lehre der objektorientierten Programmierung entwickelten Entwicklungsumgebung BlueJ vor. Diese Erweiterung führt die bereits in BlueJ implizit vorhandene Unterstützung der K&P-Metapher konsequent fort, indem sie auch im BlueJ-Klassendiagramm dynamisch die Unterscheidung zwischen dem Konsumieren und dem Produzieren einer Klasse ermöglicht.

Einleitung

Konsumieren und *Produzieren* sind zwei Seiten derselben Medaille. Ein Buch kann gelesen (konsumiert) und geschrieben (produziert) werden, ein Tisch benutzt oder gebaut, eine Software genutzt oder entwickelt werden.

Damit ein Artefakt (ein Gegenstand, ein Text, ein Konzept) konsumiert werden kann, muss es zuerst produziert werden. Bevor ein Buch gelesen werden kann, muss es geschrieben werden. Bevor ein Tisch für eine Mahlzeit genutzt werden kann, muss er gebaut werden. Bevor Software benutzt werden kann, muss sie programmiert werden. Im Allgemeinen gilt dabei, dass das Produzieren deutlich anspruchsvoller ist als das Konsumieren; eine für sich genommen triviale Erkenntnis.

Wir untersuchen seit einigen Jahren die Metapher vom Konsumieren und Produzieren (kurz: *K&P-Metapher*) im Umfeld der Lehre der objektorientierten Programmierung (Späh u. Schmolitzky, 2007). Bei unserer Didaktik machen wir uns dabei zunutze, dass Konsumieren leichter als Produzieren ist: Häufig vermitteln wir unseren Studierenden ein zu verstehendes Konzept nur in einer „abgespeckten“ Version (lassen sie es nur konsumieren), um es erst zu einem späteren Zeitpunkt ausführlich zu behandeln (sie so damit vertraut zu machen, dass sie es produktiv einsetzen können). Dieser simple Kniff ermöglicht es an etlichen Stellen, die gerade bei der Objektorientierung häufig zirkulären Abhängigkeiten von Grundkonzepten etwas aufzubrechen.

In (Schmolitzky u. Züllighoven, 2007) haben wir dazu bereits einige Beispiele aus dem Bereich der Programmierung angeführt. Unter anderem haben wir *Generizität* in Java als ein Konzept aufgeführt, das für den Umgang mit generischen Sammlungen in Java anfänglich nur konsumiert zu werden braucht (sprich: wie gibt man bei der Deklaration einer Sammlung den Typ ihrer Elemente an), und können so in einer Erstsemesterveranstaltung die sehr nützlichen Sammlungen des *Java Collections Framework* (JCF) relativ früh thematisieren. Erst deutlich später (im darauf folgenden Semester) thematisieren wir Generizität so, dass die Studierenden selbst generische Klassen produzieren können.

Zur Verdeutlichung kann im Kontrast dazu eine andere Sicht dargestellt werden, wie sie beispielsweise in einem Lehrbuch von Liang umgesetzt ist (Liang, 2010). Hier wird das JCF sehr spät (in Kapitel 22) angesprochen; vermutlich deshalb, weil die Autoren der Meinung sind, dass vor dem Umgang mit generischen Sammlungen grundsätzlich geklärt sein muss, was Generizität ist (Kap. 21). Ein weiteres Beispiel für dieses Denken findet sich in (Ashok u. a., 2008). Auf diese Weise wird ein zentrales und nützliches Konzept (Sammlungen) aufgrund einer vermeintlichen formalen Abhängigkeit von einem anderen Konzept (Generizität) aus unserer Sicht unnötig „nach hinten geschoben“.

In diesem Artikel liegt der Fokus nicht auf dem Konsumieren und Produzieren von Konzepten in der Lehre, sondern auf dem Konsumieren und Produzieren von Klassen in der objektorientierten Programmierung (OOP). Während ein Dozent die Entscheidung, welcher Anteil eines Konzeptes konsumierend und

welcher produzierend betrachtet werden soll, mehr oder weniger willkürlich treffen kann, gibt es bei den Klassen der OOP klar definierte Eigenschaften, die für Klienten einer Klasse einerseits und für die Entwickler einer Klasse andererseits relevant sind. Dies macht sie einer automatisierten Darstellung zugänglich.

Im nächsten Abschnitt werden wir zuerst einige Begriffe benennen, die für die nachfolgende Diskussion grundlegend sind. Abschnitt 3 stellt dar, auf welche Weise die K&P-Metapher bisher in der Lehrumgebung BlueJ umgesetzt ist. Abschnitt 4 zeigt, wie dies konsequent fortgeführt werden kann, und stellt eine prototypische Implementation vor. Abschnitt 5 diskutiert, wie die bisherigen Erkenntnisse gewinnbringend eingesetzt werden könnten. Abschnitt 6 fasst die Arbeit zusammen.

K&P in der Objektorientierung

Ein zentraler Gedanke der Objektorientierung ist, dass die Zuständigkeiten für die Aufgaben eines Systems auf verschiedene Objekte verteilt werden und diese Objekte sich gegenseitig benutzen. Ein Objekt, das ein anderes benutzt, bezeichnen wir als einen *Klienten*; ein Objekt, das benutzt wird, als einen *Dienstleister*. Jedes Objekt kann sowohl Klient als auch Dienstleister sein.

Ein weiterer zentraler Gedanke der Objektorientierung (und der Softwaretechnik) ist, dass bei einem Objekt zwischen seiner *Schnittstelle* und seiner *Implementation* unterschieden werden kann. Über seine Schnittstelle bietet ein Objekt seine Dienstleistungen an; in der Implementation wird dieses Angebot umgesetzt. Diese Unterscheidung wird deutlicher, wenn der Fokus von den Objekten auf die sie definierenden Klassen gerichtet wird: In Java wird systematisch zwischen der Schnittstelle einer Klasse und ihrer Implementation unterschieden, indem die mit dem Werkzeug *javadoc* erzeugte Dokumentation einer Klasse üblicherweise ihre Schnittstelle darstellt, während der Quelltext der Klasse ihre vollständige Implementation wiedergibt.

Ein Programmierer eines Klienten, der Exemplare einer anderen Klasse ausschließlich *benutzen* will, konsumiert die andere Klasse lediglich. Er muss idealerweise nur ihre Schnittstelle kennen (die auch explizit in Form eines Java-Interfaces beschrieben sein kann), um qualifiziert als Klient auftreten zu können. Erst wenn ein Programmierer die andere Klasse *warten* muss (Fehler beheben muss, ihre Funktionalität erweitern muss, etc.), muss er ihren Quelltext bearbeiten und kann in Bezug auf ihre Dienstleistungen produzierend aktiv werden.

Die Essenz lautet: eine Klasse konsumieren erfordert nur die Kenntnis ihrer Schnittstelle; eine Klasse produzieren erfordert zwingend Einblick in ihre Implementation. Beide Sichten auf eine Klasse finden sich auch in der Entwicklungsumgebung *BlueJ* wieder, allerdings nur unvollständig.

K&P in BlueJ bisher

BlueJ ist eine für die Lehre des objektorientierten Programmierparadigmas entworfene Entwicklungsumgebung (engl. abgekürzt: IDE) für Java. Sie ist wie Eclipse frei verfügbar, hat aber im Gegensatz zu Eclipse nicht den Anspruch, eine vollwertige IDE für die professionelle Softwareentwicklung darzustellen; BlueJ ist gezielt minimalistisch ausgelegt, um Programmieranfängern die Kernkonzepte der objektorientierten Programmierung zu vermitteln und diese nicht mit umfangreicher Funktionalität zu überwältigen (Kölling u. a., 2003). BlueJ wird weltweit inzwischen von fast 1000 Hochschulen eingesetzt (BlueJ, 2010).

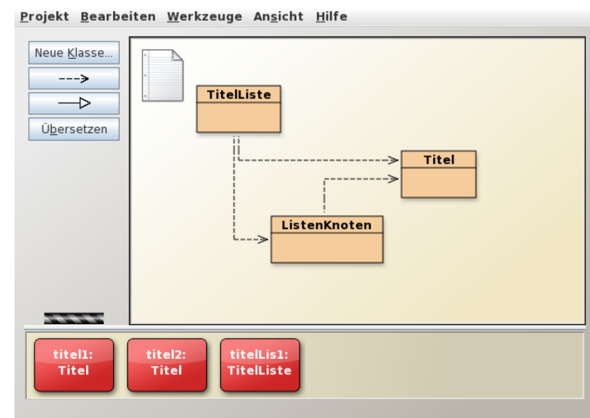


Abbildung 1: Das Hauptfenster von BlueJ

BlueJ stellt die Klassen eines Java-Projektes, anders als übliche IDEs, nicht in Listenform dar, sondern in Form eines einfachen UML-Klassendiagramms (Abbildung 1). BlueJ visualisiert mit diesem Diagramm nicht nur die Struktur des Projektes, sondern bietet auch direkte Interaktionsmöglichkeiten mit den dargestellten Klassen und ihren Exemplaren: Es können interaktiv Exemplare dieser Klassen erzeugt und an diesen Exemplaren dann alle Methoden aufgerufen werden.

Aufgrund der Möglichkeit, Klassen und Objekte interaktiv zu benutzen, ohne Quelltext schreiben zu müssen, kann bei der Benutzung von BlueJ bereits zwischen konsumierenden und produzierenden Personen unterschieden werden. Eine konsumierende Person kann Klassen und Objekte benutzen, ohne wissen zu müssen, wie diese mit Java erstellt werden. Nur eine produzierende Person, also jemand, der Klassen in ihrem Quelltext verfasst, muss mehr über Java wissen (Syntax, Bibliotheksklassen, etc.). Der *Object-First-Ansatz* von Barnes und Kölling (Barnes u. Kölling, 2009) basiert sehr stark auf diesen Möglichkeiten.

Der in BlueJ integrierte Editor zum Bearbeiten von Klassendefinitionen bietet zwei Sichten: eine konsumierende, die die von *javadoc* für die Klasse erstellte (und nicht interaktiv änderbare) Schnittstellenbeschreibung darstellt (*Schnittstellensicht*); und eine

produzierende Sicht, die den vollständigen Quelltext der Klasse darstellt und bearbeitbar macht (*Quelltextsicht*).

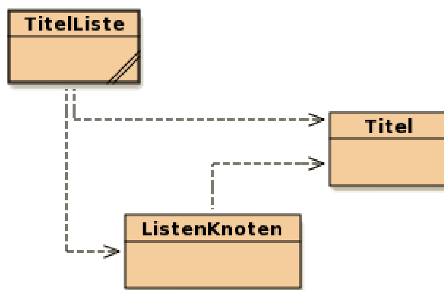


Abbildung 2: Ein Klassendiagramm in BlueJ

Für die nachfolgende Diskussion soll ein Beispiel das Verständnis erleichtern: In einem Lehrprojekt, das fachlich einen MP3-Player modellieren soll, werden neben den abzuspielenden *Musiktiteln* auch *Wiedergabelisten* benötigt. Das BlueJ-Projekt dazu enthält deshalb neben der Klasse `Titel` auch eine Klasse `TitelListe`. Objekte der Klasse `TitelListe` sollen alle Eigenschaften aufweisen, die eine Liste üblicherweise anbietet: Die Reihenfolge der Elemente ist frei manipulierbar und es können beliebig Duplikate eingefügt werden (eine Semantik, die offensichtlich gut zu einer Wiedergabeliste von Musiktiteln passt). Entsprechend bietet ihre Schnittstelle Operationen wie *einfuegenAnPosition*, *entferneAnPosition* etc. an. Die Klasse `TitelListe` sei intern als doppelt verkettete Liste von *Knotenelementen* umgesetzt, die neben den Verkettungsreferenzen jeweils eine Referenz auf einen `Titel` halten sollen (in dem Lehrprojekt geht es um verschiedene Implementationsformen für Listen). Diese Knotenelemente seien in der Klasse `ListenKnoten` realisiert. Abbildung 2 zeigt diese Zusammenhänge in einem BlueJ-Klassendiagramm.

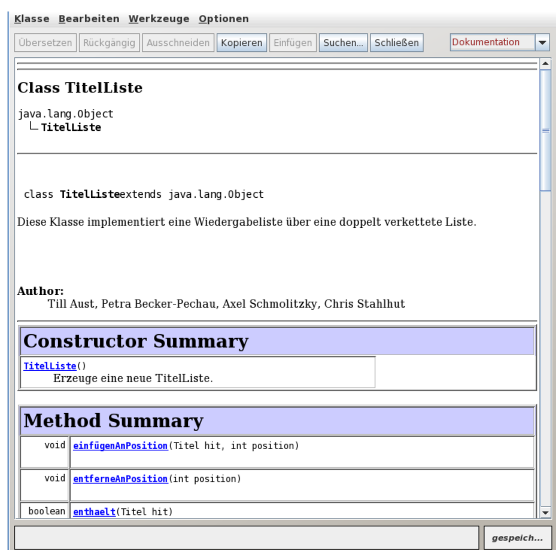


Abbildung 3: Die Schnittstellensicht im BlueJ-Editor

Wenn ein Klient der Klasse `TitelListe` geschrieben werden soll (etwa eine Komponente, die Wiedergabelisten auf dem Bildschirm darstellt), kann der Programmierer diese Klasse im BlueJ-Editor betrachten, um die aufrufbaren Operationen vor sich zu haben. Da den Programmierer somit nur die Schnittstelle interessieren muss, kann dazu die *Schnittstellensicht* des Editors gewählt werden (Abbildung 3).

Diese Sicht verbirgt alle Implementationsdetails, unter anderem auch die Abhängigkeit zu der Klasse `ListenKnoten`; ein Effekt, der im Sinne des Geheimnisprinzips nur erwünscht sein kann.

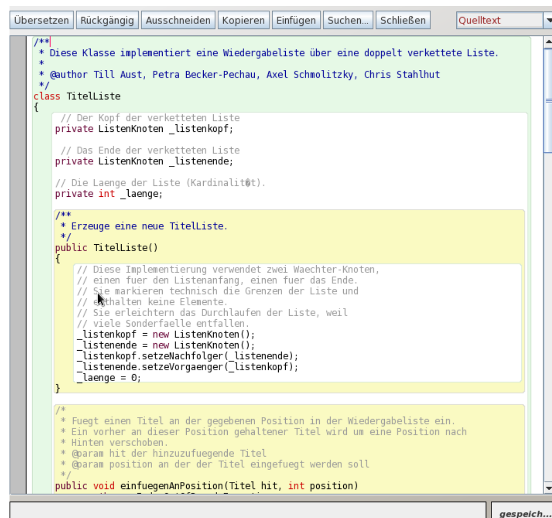


Abbildung 4: Die Quelltextsicht im BlueJ-Editor

Erst wenn die Implementation der Klasse `TitelListe` selbst betrachtet oder bei Wartungsarbeiten verändert werden soll, muss in die *Quelltextsicht* des Editors gewechselt werden. Diese Sicht unterscheidet sich nur geringfügig (primär durch die Blockhervorhebung, Abbildung 4) von der anderer Editoren.

BlueJ unterstützt somit mit den beiden Sichten im Editor direkt die K&P-Metapher: Klienten können eine Klasse nur konsumieren (müssen nur die Informationen sehen, die für eine Benutzung notwendig sind: die Schnittstellensicht), während Wartungsprogrammierer den Zugriff auf den vollen Quelltext benötigen, um produzierend tätig werden zu können.

K&P in BlueJ fortgeführt

Die Klasse `TitelListe` hat eine in ihrer Schnittstelle definierte Benutzt-Beziehung zu der Klasse `Titel`, während die Beziehung zu der Klasse `ListenKnoten` nicht zu ihrer Schnittstelle gehört. Die Klasse `ListenKnoten` benutzt ebenfalls die Klasse `Titel`. Alle drei Abhängigkeiten werden im BlueJ-Klassendiagramm durch Pfeile dargestellt, siehe Abbildung 2.

Da es sich bei der Klasse `Titel` um den Elementtypen der Liste handelt, ist die Abhängigkeit in der

Schnittstelle der Klasse `TitelListe` unumgänglich; erwartungsgemäß muss ein Klient einer `TitelListe` damit rechnen, `Titel` als Parameter zu übergeben (obwohl nicht alle Operationen der Klasse explizit mit Titeln umgehen müssen, siehe oben).

Es ist hingegen für einen Klienten eher nachrangig, wie die Klasse `TitelListe` implementiert ist. Der Umstand, dass dies in einer verketteten Liste passiert, ist für die Benutzung üblicherweise irrelevant. Dies wird auch in den beiden Editor-Darstellungen der Klasse deutlich: In der Schnittstellensicht ist lediglich die Klasse `Titel` aufgeführt, während im Quelltext der Klasse auch die Klasse `ListenKnoten` auftaucht.

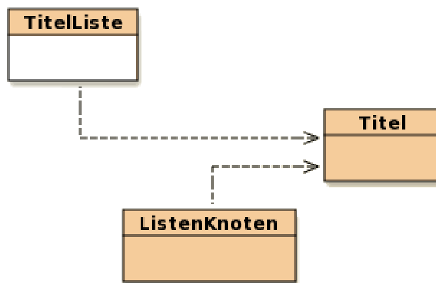


Abbildung 5: Klientensicht der `TitelListe`

Diesen Unterschied könnten wir auch im Klassendiagramm darstellen, indem wir eine *Klientensicht* auf die Klasse `TitelListe` postulieren: in dieser Sicht besteht lediglich eine Abhängigkeit zur Klasse `Titel` (Abbildung 5). Die Darstellung der `TitelListe` in Abbildung 2, in der alle Abhängigkeiten der Klasse gezeigt werden, nennen wir hingegen ihre *Produzentensicht*. Da Produzenten einer Klasse ihren Quelltext erstellen/bearbeiten, sollten für sie alle Abhängigkeiten sichtbar sein.

Die Darstellung in Abbildung 5 stammt aus einer von uns entwickelten Erweiterung von BlueJ, die ein interaktives Umschalten zwischen den beiden Sichten auf *jede Klasse* im Klassendiagramm ermöglicht. Wir nennen diese Erweiterung im Folgenden kurz *BlueJ-CP*.

Die grundsätzliche Möglichkeit zweier Sichten auf jede Klasse führt in BlueJ-CP dazu, dass es nicht nur ein statisches, sondern eine Vielzahl von möglichen Klassendiagrammen für dasselbe Projekt gibt. Durch die interaktive Möglichkeit des Wechsels der Sichten wird das statische Klassendiagramm zu einem dynamischen Klassendiagramm. Bei einem Projekt mit n Klassen ergeben sich bis zu 2^n mögliche Klassendiagramme. Das zu einem bestimmten Zeitpunkt angezeigte Klassendiagramm bezeichnen wir als das *momentane Klassendiagramm*.

Um die Unterscheidung visuell stärker zu verdeutlichen, haben wir für die Darstellung der Klientensicht auch das Klassensymbol leicht verändert: der untere Teil erscheint weiß. Für die Produzentensicht haben wir nichts am Symbol verändert, weil dies der bisherigen Darstellung in BlueJ entspricht.

Ausblenden von Klassen

Da im Kontext unseres Beispiels allein die Klasse `TitelListe` die Klasse `ListenKnoten` benutzt, kann letztere als ein ausgelagertes Implementationsdetail betrachtet werden. Aus Sicht eines Klienten der `TitelListe` ist nicht nur die Benutzt-Beziehung zwischen den beiden Klassen irrelevant, sondern die komplette Klasse `ListenKnoten`; sie muss in der Konsumentensicht der Klasse `TitelListe` nicht angezeigt werden. Abbildung 6 zeigt eine in dieser Hinsicht konsequente Darstellung des Projekts mit der Konsumentensicht der `TitelListe`. Dieses Diagramm ist deutlich einfacher, ein von uns erwünschter Effekt; durch das Umschalten von Klassen in ihre Klientensicht soll die Komplexität des Diagramms reduziert werden können.

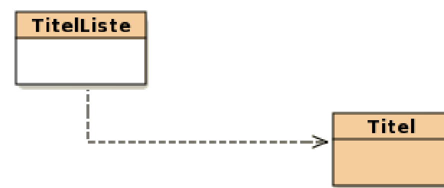


Abbildung 6: Die konsequente Klientensicht

Für unsere Erweiterung ergibt sich somit ein rein formales Kriterium, nach dem Klassen verborgen werden können: Zeigt das momentane Klassendiagramm eine Abhängigkeit zu einer Klasse `A`, so ist diese Klasse anzuzeigen. Dies ist von der Art der Abhängigkeit (basierend auf der Schnittstelle oder nicht) unabhängig. Benutzen zwei Klassen `B` und `C` dieselbe Klasse `A`, so ist `A` anzuzeigen, wenn min. eine eingehende Abhängigkeit angezeigt wird. Eine Klasse wird also nur verborgen, wenn sie im momentanen Klassendiagramm keine eingehenden Abhängigkeiten besitzt.

Wird ein Projekt erstmalig in BlueJ-CP geöffnet, könnte für alle Klassen die Klientensicht gewählt werden, in der Hoffnung, dass dies einen ersten „high-level" Überblick vermittelt, weil transitiv alle Implementationsdetails ausgeblendet sind. Bei der Wahl der anfangs anzuzeigenden Klassen sind jedoch einige Dinge zu beachten.

Einstiegspunkte für Java klassisch

Einen offensichtlichen Einstiegspunkt in eine allgemeine Java-Anwendung bietet die `main`-Methode. Klassen, die eine Methode mit einer speziellen Signatur anbieten (es kann mehrere in einem Projekt geben), bilden somit einen guten Ausgangspunkt. Im Idealfall verfügt genau eine Klasse über eine `main`-Methode. Diese Klasse wird in der Klientensicht angezeigt und ist somit möglicherweise sogar die einzige Klasse, die im momentanen Klassendiagramm angezeigt werden muss. Erst durch das Umschalten bei dieser Klasse auf die Produzentensicht werden die Klassen sichtbar, die im Rumpf der `main`-Methode benutzt werden. Auf

diese Weise kann ein Klassendiagramm nach und nach „entfaltet“ werden.

Java-Pakete verfügen ebenfalls über eine Schnittstelle, sie besteht aus allen *öffentlichen* (mit dem Modifikator `public` deklarierten) *Klassen*. Nur die auf diese Weise „exportierten“ Klassen können aus anderen Paketen benutzt werden, alle anderen Klassen gehören zur Implementation eines Paketes.

Die öffentlichen Klassen eines Pakets (insbesondere der Pakete, die keine `main`-Methode enthalten) stellen somit weitere sinnvolle Einstiegspunkte dar, die initial bei der Darstellung eines Paketes sichtbar sein sollten. Auch sie sollten anfangs in der Klientensicht dargestellt werden (inklusiv ihre transitiven Schnittstellen-Beziehungen) und können bei Bedarf aufgefaltet werden.

Eine Ausnahme stellen JUnit-Testklassen dar. Diese Klassen müssen nur deshalb öffentlich sein, damit sie vom JUnit-Rahmenwerk gefunden und ausgeführt werden können, und gehören somit nicht zur Schnittstelle eines Pakets. Zur Reduktion der Diagramm-Komplexität können sie gesondert ausgeblendet werden.

Einstiegspunkte für BlueJ

Das Problem mit beiden Einstiegsheuristiken (`main`- und öffentliche Klassen) ist, dass sie nicht gut für echte BlueJ-Projekte funktionieren: Da in BlueJ Objekte interaktiv erzeugt werden und die Benutzer direkt an diesen Objekten Methoden aufrufen können, werden `main`-Methoden nur in Ausnahmefällen implementiert; siehe dazu unter anderem (Kölling u. Rosenberg, 2001). Und die Lehre-Projekte in BlueJ bestehen auch nur sehr selten aus mehreren Paketen (obwohl BlueJ diese auch darstellen kann), sondern meist nur aus einem, typischerweise dem unbenannten Paket.

Für echte BlueJ-Projekte ergibt sich deshalb die auf den ersten Blick paradoxe Schlussfolgerung, dass genau die Klassen initial dargestellt werden sollten, die nicht von anderen Klassen benutzt werden. Denn bei diesen liegt die Vermutung nahe, dass sie für den interaktiven Gebrauch entwickelt wurden und somit gute Einstiegspunkte sind. Auf diese Weise ist auch sichergestellt, dass eine gerade neu erstellte, aber noch unbenutzte Klasse angezeigt wird.

Einsatzmöglichkeiten

Die Erkenntnisse aus unserem anfänglich rein gedanklichen Experiment mit BlueJ und der realen Umsetzung in BlueJ-CP weisen uns zwei interessante Wege: Zum einen sollte untersucht werden, inwieweit BlueJ-CP sinnvoll in der Programmierausbildung eingesetzt werden kann, um die Konzepte Schnittstelle und Implementation anschaulicher zu vermitteln. Zum anderen scheint uns das Konzept der verschiedenen Sichten auf Klassendiagramme übertragbar auf andere Werkzeuge.

In der Programmierausbildung

Idealerweise würden die Teilnehmer an einer einführenden Programmierveranstaltung gar nicht merken, dass sie mit einem veränderten BlueJ arbeiten, und an passender Stelle würden die zusätzlichen Möglichkeiten von BlueJ-CP zum Einsatz kommen. Wir haben in dieser Hinsicht bisher noch keine belastbaren Erfahrungen sammeln können.

Immerhin haben wir mit 23 freiwilligen Teilnehmern einer Erstsemesterveranstaltung eine kleine Studie durchgeführt, in der sie eine Vorversion von BlueJ-CP alternativ zu BlueJ bei der Lösung einer Übungsaufgabe einsetzen konnten (Stahlhut, 2010). Ein Ergebnis dieser Studie war, dass 21 der 23 Probanden der Meinung waren, dass BlueJ-CP beim Verständnis der Konzepte Schnittstelle und Implementation unterstützt, während 2 Probanden es nicht als unterstützend empfunden haben. 18 von 23 empfanden es als angemessen, Klassen benutzungsabhängig verbergen zu können (bei 5 Enthaltungen).

Die Erkenntnisse aus dieser Studie sind primär in die Weiterentwicklung von BlueJ-CP eingeflossen; aber das Feedback aus den geführten Interviews lässt deutlich darauf schließen, dass die interaktive Möglichkeit zweier Sichten auf eine Klasse, intelligent eingesetzt, beim Verständnis von Kapselung und Schnittstellen hilfreich sein kann.

In der Visualisierung von Software

Seit wir BlueJ-CP zur Visualisierung von Java-Projekten zur Verfügung haben, erscheinen uns die aus Quelltexten generierten Klassendiagramme anderer UML-Werkzeuge merkwürdig „flach“: Sie zeigen immer alle Abhängigkeiten, unabhängig von ihrer Einordnung als Schnittstellen- oder als Implementations-Abhängigkeit. Die Kriterien für diese Einordnung sind rein formal und können ohne weiteres auch in anderen Werkzeugen als in BlueJ dafür genutzt werden, automatisiert verschiedene Abstraktionen desselben Quelltextes anzubieten.

Ein interessantes Vorhaben könnte somit sein, die mit BlueJ-CP gesammelten Erkenntnisse auch in anderen Werkzeugen umzusetzen; zumindest in solchen, die quelloffen zur Verfügung stehen. Wir sehen dabei insbesondere die Möglichkeit im Vordergrund, ein bestehendes System mit Hilfe der beschriebenen Mittel nach und nach interaktiv aufzufalten und somit einen objektorientierten Entwurf zu explorieren. Inwieweit dies wirklich beim Einlesen und Verstehen hilfreich ist, gälte es zu untersuchen.

Ein Werkzeug, das explizit zur Untersuchung der K&P-Metapher entwickelt wurde, ist *jMango* (Späh, 2008). Es bietet neben einer filtergestützten Visualisierung von Java-Systemen auch eine Unterstützung für die Modellierung von Konzept-Abhängigkeiten. Die in einer Erstsemesterveranstaltung vermittelten Konzepte wurden mit Hilfe von jMango systematisch modelliert (Albers, 2009).

Da jMango als eine Eclipse-RCP-Anwendung konstruiert ist, wäre es denkbar, gezielt die Visualisierungsmöglichkeiten für allgemeine Java-Projekte als Plugin für Eclipse anzubieten. Auf diese Weise könnte das dynamische Falten von Klassendiagrammen auf breiter Ebene für Java-Eclipse-Projekte zur Verfügung gestellt werden. Bisher fehlt in jMango allerdings noch ein guter Algorithmus für das automatische Layouten von Klassendiagrammen (zumindest als Ausgangspunkt), so dass ein gebrauchsfertiges Plugin noch Forschungs- und Entwicklungsaufwand erfordert.

Zusammenfassung

Das Konsumieren einer Klasse von ihrem Produzieren zu unterscheiden ist ein Kernkonzept der Objektorientierung, dem – softwaretechnisch breiter betrachtet – die Trennung von Schnittstelle und Implementation zugrunde liegt. Die IDE BlueJ macht diese Unterscheidung in ihrem Editor deutlich, indem sie zwei Sichten auf eine Klasse anbietet.

In diesem Artikel haben wir dargestellt, wie BlueJ erweitert werden kann, um diesen Unterschied auch im BlueJ-Klassendiagramm zu visualisieren. Konzeptuell ergab sich daraus, dass es für ein Klassendiagramm mit n Klassen 2^n verschiedene Darstellungen geben kann, die ein System auf mehreren Abstraktionsebenen darstellen können. Diesen Umstand gilt es weiter zu untersuchen, sowohl in seinen Einsatzmöglichkeiten in der Lehre als auch in der Visualisierung von Softwaresystemen.

Literatur

[Albers 2009] ALBERS, T.: *Evaluation eines prototypischen Werkzeuges zur Visualisierung von Konzeptabhängigkeiten unter Berücksichtigung der Metapher „Konsumieren vor Produzieren“*, Universität Hamburg, Bachelorarbeit, 2009

[Ashok u. a. 2008] ASHOK, S ; KIONG, D ; POO, D: *Object-Oriented Programming and Jav*. Springer Verlag, 2008. – 2. Auflage

[Barnes u. Kölling 2009] BARNES, D. J. ; KÖLLING, M.: *Objects First with Java — A Practical Introduction using BlueJ*. Harlow, United Kingdom : Prentice Hall / Pearson Education, 2009. – 4. Auflage

[BlueJ 2010] BLUEJ: *BlueJ — The interactive Java environment*. Version: 2010. `http://www.bluej.org`. – zuletzt besucht: 2010-12-16

[Kölling u. a. 2003] KÖLLING, M. ; QUIG, B. ; PATTERSON, A. ; ROSENBERG, J.: The BlueJ system and its pedagogy. In: *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology* (2003)

[Kölling u. Rosenberg 2001] KÖLLING, M. ; ROSENBERG, J.: Guidelines for Teaching Object Orientation with Java. In: *Information Technology in Computer Science Education (ITiCSE 2001)*, 2001

[Liang 2010] LIANG, Y. D.: *Introduction to Java Programming*. Harlow, United Kingdom : Prentice Hall / Pearson Education, 2010. – 8. Auflage

[Schmolitzky u. Züllighoven 2007] SCHMOLITZKY, A. ; ZÜLLIGHOVEN, H.: Einführung in die Softwareentwicklung - Softwaretechnik trotz Objektorientierung? In: *A. Zeller and M. Deininger (Hrsg.), Software Engineering im Unterricht der Hochschulen (SEUH)*, 2007

[Späh 2008] SPÄH, C.: *Konsumieren und Produzieren als nützliche Metaphern in der Softwaretechnikausbildung und der Exploration von Klassenstrukturen*, Universität Hamburg, Diplomarbeit, 2008

[Späh u. Schmolitzky 2007] SPÄH, C. ; SCHMOLITZKY, A.: „Consuming before Producing“ as a Helpful Metaphor in Teaching Object-Oriented Concepts. In: *Eleventh Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts*, 2007

[Stahlhut 2010] STAHLHUT, C.: *Die Metapher „Konsumieren und Produzieren“ in BlueJ*, Universität Hamburg, Bachelorarbeit, 2010

Ein Dashboard für Learning-Management-Systeme

Daniel Kulesz
Institut für Softwaretechnologie, Universität Stuttgart
daniel.kulesz@informatik.uni-stuttgart.de

Zusammenfassung

In der Softwaretechnik-Lehre werden üblicherweise Lehrveranstaltungen durchgeführt, in denen studentische Teams gemeinsam Übungsaufgaben bearbeiten. Dabei werden auch studentische Hilfskräfte als Tutoren für die Korrektur der Übungsaufgaben eingesetzt. Doch gerade beim Einsatz vieler Tutoren entsteht auf Seiten der Lehrverantwortlichen erheblicher Aufwand für Kommunikation und Qualitätssicherung.

In den letzten Jahren hielten Learning-Management-Systeme wie Moodle oder ILIAS zunehmend Einzug in den Hochschulbetrieb. Viele Lehrverantwortliche erhoffen sich vom Einsatz dieser Systeme auch eine Steigerung der Effizienz bei der Koordination ihrer studentischen Tutoren und der Zusammenarbeit mit ihnen. In der täglichen Anwendungspraxis haben wir jedoch die Erfahrung gemacht, dass diese Anforderungen nicht ausreichend erfüllt werden.

Dieser Beitrag beschreibt einen Verbesserungsansatz, der vor zwei Jahren am Institut für Softwaretechnologie der Universität Stuttgart entwickelt wurde und seitdem mit großem Erfolg eingesetzt wird. Den Kern bildet dabei ein simples, als Online-Spreadsheet realisiertes Dashboard, das auf einem bereits existierenden Learning-Management-System aufsetzt.

Übungen und Korrekturen

In der Softwaretechnik-Lehre werden üblicherweise Lehrveranstaltungen durchgeführt, in denen studentische Teams gemeinsam Übungsaufgaben bearbeiten. Die Lösungen der Studierenden müssen natürlich korrigiert und bewertet werden, außerdem soll den Studierenden eine Rückmeldung zu ihrer Lösung gegeben werden, um ihren Lernerfolg zu sichern.

An solchen Lehrveranstaltungen, typisch zu Beginn des Studiums, nehmen sehr viele Studierende teil; häufig ist die Lehrveranstaltung sogar Pflicht. Ein insbesondere an den Universitäten häufig beschrittener Weg zur Bewältigung dieses enormen Andranges ist der Einsatz studentischer Hilfskräfte als Tutoren für Übungen.

Obwohl durch dieses System in vielen Fällen ein betreutes Übungsangebot überhaupt erst möglich wird, erhöht sich durch die Delegation von Lehraufgaben auch der Kommunikationsbedarf. Zudem besteht, wie bei jeder Delegation, immer das Risiko, dass die Aufgaben nicht den Vorstellungen des Verantwortlichen entsprechend erledigt werden: Es kann zu verschiedenen Qualitätseinbußen kommen, beispielsweise bei der Einheitlichkeit der Bewertungen oder der Ausführlichkeit und Termintreue der Rückmeldungen.

Um diese Problematik in den Griff zu bekommen, sollten Lehrverantwortliche geeignete Qualitätssicherungsmaßnahmen durchführen. Diese sollten bezüglich des notwendigen Aufwandes und der erzielbaren Qualitätssteigerung möglichst effizient sein.

E-Learning

Laut (Steffens u. Reiss, 2009) stellt die klassische Präsenzlehre in der Informatik und den Wirtschaftswissenschaften immer noch die führende Lehrform an Deutschlands Hochschulen dar. Über 70 Prozent der 200 befragten Lehrenden nutzen der Studie zufolge auch einzelne Techniken des E-Learnings wie beispielsweise Diskussionsforen, Chats, Umfragen, kommentierte Linksammlungen, Vorlesungsaufzeichnungen oder kleinere Online-Tests. Die didaktisch sinnvolle Kombination der Präsenzlehre mit E-Learning-Elementen wird auch als 'Blended Learning' oder 'Integriertes Lernen' bezeichnet.

Die Studie macht jedoch deutlich, dass die Form des Präsenzunterrichts klar überwiegt. Denn obwohl über 60 Prozent der befragten Lehrenden die räumliche und zeitliche Flexibilität der E-Learning-Techniken als sehr vorteilhaft bewerten, nimmt der E-Learning-Anteil rein quantitativ betrachtet bei über 85 Prozent der Befragten nur höchstens 30 Prozent des gesamten Rahmens einer Lehrveranstaltung ein.

Learning-Management-Systeme

Mittlerweile stellen nahezu alle Rechenzentren der deutschen Hochschulen für ihre Mitglieder auch sogenannte Learning-Management-Systeme (LMS) bereit. Es handelt sich dabei um komplexe Software-

systeme zur Bereitstellung von Lerninhalten und Organisation von Lernvorgängen. Laut einer aktuellen Online-Umfrage (Oevel u. Lange) des Arbeitskreises 'ZKI-AK elearning' der Universität Paderborn werden bevorzugt die beiden Open-Source-LMS ILIAS (Ili) und Moodle (Moo) bereitgestellt, die zusammen auf einen 'Marktanteil' von über 55 Prozent kommen.

Besonders bemerkenswert ist dabei, dass offenbar jede der befragten Hochschulen mindestens ein LMS vorhält. 35 Prozent der befragten Hochschulen stellen sogar zwei oder mehr unterschiedliche LMS bereit.

Die LMS werben zwar mit Merkmalsdimensionen wie Interaktivität und Kollaboration, viele Lehrende sind jedoch skeptisch, ob die gebotene Unterstützung wirklich praxistauglich ist. Es gibt zwar einige Untersuchungen in denen unterschiedliche LMS miteinander verglichen werden (Graf u. List, 2005)(Colace u. a., 2003), doch für die meisten Lehrenden bieten sie keine praktische Hilfe, da die Studien keine konkreten Lernszenarien aufführen. Zudem steht offenbar an 65 Prozent der Hochschulen ohnehin nur ein einziges LMS zur Verfügung.

In der Abteilung Software Engineering des Instituts für Softwaretechnologie der Universität Stuttgart nutzen wir seit nunmehr über zwei Jahren ILIAS für die Abwicklung von Gruppenübungen, die mit Hilfe studentischer Tutoren durchgeführt werden. Deshalb möchten wir über die Eignung von ILIAS für solche Lehrveranstaltungen berichten. Dazu möchten wir zunächst ein typisches Lernszenario aus unserem täglichen Lehrbetrieb vorstellen.

Lernszenario

Wir bieten im Studiengang Softwaretechnik insgesamt sechs Lehrveranstaltungen an, bei denen die Studierenden Aufgaben in Dreier-Teams bearbeiten. Dabei erstellen die Studierenden zahlreiche aufeinander aufbauende Artefakte wie Projektpläne, Kostenschätzungen, Spezifikationen, Review-Protokolle, Testpläne, Entwurfsdokumente und natürlich auch kleinere Programme.

Die von den studentischen Dreier-Teams abgelieferten Artefakte werden individuell begutachtet und bewertet. Der durchschnittliche Aufwand für die rein inhaltliche Prüfung und Korrektur eines Artefaktes beträgt etwa 20 Minuten. Pro Semester und Lehrveranstaltung geben im Durchschnitt 20 Dreier-Teams je 8 Artefakte ab. Dadurch ergibt sich pro Semester und Lehrveranstaltung ein Aufwand von über 50 Stunden allein für die inhaltliche Prüfung der Artefakte.

Die Übungen werden in der Regel von ein bis zwei wissenschaftlichen Mitarbeitern betreut. Daneben werden auch - je nach Veranstaltung - zwei bis vier studentische Tutoren für die Korrekturen eingesetzt.

Rollen und Anforderungen

Aus dem vorgestellten Lernszenario können für die Gruppenübungen insgesamt vier relevante Rollen abgeleitet werden: Dozent, Übungsleiter, Tutor und Student. Einige Rollen (beispielsweise Dozent und Übungsleiter) können dabei auch in Personalunion wahrgenommen werden. Zudem kann auch innerhalb der einzelnen Rollen eine hierarchische Ordnung vorhanden sein. Beispielsweise könnte ein langjähriger Tutor die Rolle eines 'Chef-Tutors' bekleiden. Ebenso könnte es im Dreier-Team der Studierenden einen Teamleiter geben.

Für jede der vorgestellten Rollen ergeben sich nun im Kontext des Lernszenarios unterschiedliche Wünsche und Anforderungen:

Dozent

- Abstimmung der Übungen auf die Lernziele der Vorlesung

Übungsleiter

- Bereitstellen der zu bearbeitenden Aufgabe
- Frühes Feedback zur Verständlichkeit und Angemessenheit der Aufgabe
- Vorgabe von Abgabe- und Korrekturfristen
- Archivierung der Abgaben
- Transparenz bezüglich Status der Korrektur und Qualität der studentischen Abgaben
- Überprüfung der Korrekturleistung der Tutoren mit Stornierungsmöglichkeit bei fehlerhaften Korrekturen
- Ausgleich von zeitweiligen Ausfällen einzelner Tutoren
- Überblick über die bisherige Qualität eines ausgewählten Teams, speziell bei 'Problemkindern'

Tutor

- Klare Festlegung der Übungsgruppen, deren Abgaben korrigiert werden sollen
- Einfache und schnelle Möglichkeit, an die studentischen Abgaben heranzukommen
- Verständliche, leicht zugängliche Korrekturanweisungen
- Vergleich der eigenen Korrektur mit den Korrekturen anderer Tutoren oder früherer Jahrgänge
- Zeiterfassung

Student

- Einfache und schnelle Möglichkeit, an die Aufgaben heranzukommen
- Einfache und schnelle Möglichkeit, die geleistete Ausarbeitung abzugeben
- Schnelle Antwort auf Fragen zur Aufgabenstellung

- Sicherheit, dass die Abgabe auch angekommen ist
- Zeitnahe und nachvollziehbare konstruktive Kritik und Bewertung

Alle beteiligten Rollen

- Konzentration auf die inhaltliche Beschäftigung mit den Aufgaben bei möglichst wenig zusätzlichem Overhead
- Räumliche und zeitliche Flexibilität

Gute Absicht

Ein Learning-Management-System ist keinesfalls Voraussetzung für die Durchführung von Gruppenübungen, die durch studentische Tutoren unterstützt werden. Ein LMS bietet jedoch den großen Vorteil, dass die Ausgabe der Aufgabenstellung, die Abgabe der Lösungen, die Verteilung der Korrekturaufgaben und die Rückmeldung an die Studenten auf einer einheitlichen Plattform konsolidiert werden können. Durch die enge Vernetzung und die damit verbundene Automatisierbarkeit lassen sich Synergieeffekte nutzen. So können beispielsweise die Studierenden durch ein Subskriptionsmodell automatisch über eine erfolgte Korrektur benachrichtigt werden, ohne dass hierfür eine zusätzliche Aktion des Korrektors notwendig wäre.

Ein LMS verfolgt damit die richtige Intention. Es versucht, das Lehren und Lernen in den Vordergrund zu stellen und alle beteiligten Rollen bei den administrativen Belangen der Durchführung zu entlasten.

Der Einsatz eines LMS ist aber kein Garant für das Erreichen einer solchen Entlastung. Bei unangemessenem Gebrauch eines LMS kann es sogar zu einer 'Virtualisierung der Ausbildung' (Schulmeister, 2005) und damit zu einer Gefährdung des Lernerfolgs kommen. Deshalb sollten im Vorfeld des Einsatzes eines LMS mögliche Blended-Learning-Szenarien identifiziert und auf ihre Passgenauigkeit zum didaktischen Gesamtkonzept der Lehrveranstaltung untersucht werden. (Budden, 2009) empfiehlt zudem, während der Durchführung sowohl den Lernfortschritt der einzelnen Individuen als auch der Lerngruppen als Ganzes kontinuierlich zu überwachen.

Unzureichende Wirkung

Sowohl ILIAS als auch Moodle bieten auf den ersten Blick sehr viele verschiedene E-Learning-Funktionen. Dabei ist es die Aufgabe des Lehrenden, eine sinnvolle Kombination dieser Werkzeuge zur Unterstützung seiner Lehrveranstaltung zusammenzustellen. Die Werkzeuge stehen dabei in Form von Modulen zur Verfügung. Obwohl dieser Ansatz den Lehrenden eine gewisse Flexibilität verspricht, mussten wir feststellen, dass für viele der geschilderten Anforderungen weder ein einziges Werkzeug noch eine Werkzeugkombination geeignet ist. Wir stützen uns in der folgenden Beschreibung zwar auf ILIAS, das Problem besteht bei Moodle jedoch gleichermaßen.

So bietet ILIAS zwar ein Modul zur Einreichung und Bewertung von Übungen, das Modul ist jedoch auf Einzelabgaben beschränkt. Eine Unterstützung für die im Lernszenario geschilderten Gruppenübungen ist nicht vorgesehen.

Dabei wird das Modul auch vielen Anforderungen nicht gerecht, die nicht spezifisch für Gruppenübungen sind. So ermöglicht es das Modul zwar dem Übungsleiter, eine Aufgabe mit einer festgelegten Frist auszugeben und die Abgaben komfortabel einzusammeln. Auch kann die Korrektur und Bewertung über das in Abbildung 1 dargestellte Formular durch die Tutoren erfolgen. Es bietet jedoch keine ausreichende Unterstützung für die Interaktion zwischen Übungsleitern, Tutoren und Studenten.

So kann der Übungsleiter in ILIAS nicht festlegen, welche Tutoren für welche Abgaben zuständig sind. Ferner können weder die Studierenden noch der Übungsleiter automatisch bei erfolgten Korrekturen benachrichtigt werden. Zudem fehlt eine Unterscheidung zwischen Übungsleiter und Tutor. Dadurch könnte ein studentischer Tutor die Bewertung eines Übungsleiters ändern, ohne dass dieser benachrichtigt wird. Da diese Änderung auch nicht versioniert würde, könnte der Vorgang später nicht mehr nachvollzogen werden. Darüber hinaus können sich weder Übungsleiter noch Tutor einen Eindruck über die bisherige Qualität der Leistungen eines Studierenden verschaffen, da Bewertungen nur auf der Granularität von einzelnen Aufgaben sichtbar sind.

Durch die geschilderten Probleme werden die Lehrenden genötigt, viele Interaktionen außerhalb der Lernplattformen vorzunehmen. Dieser Effekt wird besonders durch die fehlende Unterstützung für Gruppenübungen verstärkt.

Unser Ansatz

Wie an den meisten deutschen Universitäten, Fachhochschulen und Berufsakademien stellt auch das Rechenzentrum der Universität Stuttgart eine zentrale LMS-Installation bereit. Dadurch können die Abteilungen der Institute aller Fakultäten die angebotenen E-Learning-Funktionen nutzen, ohne sich selbst um Auswahl, Betrieb und Wartung eines solchen Systems kümmern zu müssen. An der Universität Stuttgart wird dabei als LMS ausschließlich ILIAS angeboten.

Da ILIAS unter den beschriebenen Defiziten leidet, haben wir einen Verbesserungsansatz entwickelt, mit dem sich ILIAS ohne Modifikation der Installation effektiv für Gruppenübungen einsetzen lässt. Als Lehrende im Software Engineering war es uns dabei wichtig, eine möglichst einfache, aber dennoch effiziente Lösung zu realisieren. Zudem wollten wir den Aufwand so gering wie möglich halten und das Projekt innerhalb weniger Tage realisieren. Die Entwicklung eines entsprechenden Gruppenübungsmoduls, das sich als Plug-In für ILIAS entwickeln und integrieren ließe, wurde deshalb ausgeschlossen.

Einreichung
Letzte Übertragung: Heute, 11:19
Abgegebene Dateien: 1 Download-Dateien
Notiz:
Kommentar für Lerner:
Bewertung
Bestanden
Note:
Letzte Änderung:
Heute, 11:19
Mail
Nachricht versenden

Abbildung 1: Ausschnitt des Bewertungsformular für Einzelabgaben in ILIAS

Der von uns entwickelte Ansatz baut dennoch auf den in ILIAS vorhandenen Modulen ‘Gruppe‘ und ‘Diskussionsforum‘ auf. Das Modul Diskussionsforum wird dabei aber stark zweckentfremdet. Darüber hinaus verwenden wir außerhalb von ILIAS ein Online-Spreadsheet, durch das ein Controlling-Dashboard realisiert wird.

Organisation und Gruppenbildung

Vor Beginn einer Lehrveranstaltung legen wir in ILIAS einen ‘Kurs‘ an. Es handelt sich dabei um einen Container, der alle für die Lehrveranstaltung relevanten Inhalte kapselt. Dort stellen wir Material wie Vorlesungsfolien und Übungsblätter sowie ein Diskussionsforum bereit.

Als ersten Schritt zur Realisierung der Übungsgruppen drucken wir zunächst Handzettel aus, auf denen jeweils die Gruppennummer, ein individuelles Gruppenpasswort sowie der Name des für die Gruppe zuständigen studentischen Tutors vermerkt sind. Am Ende der ersten Lehrveranstaltung holen sich jeweils drei Studierende einen solchen Handzettel ab.

Für die technische Abbildung der Übungsgruppen erstellen wir im ILIAS-Kurs zudem einen Ordner namens ‘Übungsgruppen‘. Dort legen wir mit Hilfe des Moduls ‘Gruppe‘ einzelne, passwortgeschützte Unterbereiche fest. Das Passwort für den Zugriff auf jeden dieser Bereiche entspricht dabei dem jeweiligen Gruppenpasswort. Zudem begrenzen wir die Anzahl der Gruppenmitglieder auf drei.

Da die Benutzerkonten der Studierenden für ILIAS an das universitätsweite Authentifizierungssystem gekoppelt sind, ist für die Nutzung des Systems keine Registrierung notwendig. Da ILIAS zudem einen Export aller Kursteilnehmer erlaubt, müssen wir diese Gesamtliste lediglich um die Gruppenzugehörigkeit ergänzen. Das verursacht allerdings etwas Aufwand, da ILIAS nur den CSV-Export aller Kursteilnehmer ungeachtet der Gruppenzugehörigkeit vorsieht. Vergleicht man das aber mit anderen Verfahren wie etwa dem Herumreichen einer Papierliste während der Vorlesung, so ist das beschriebene Verfahren dennoch viel schneller, effizienter und genauer. Außerdem kommt der Dozent auch unabhängig von einer zusätzlichen Aktion der Studierenden zu dieser Liste.

Abgabe, Korrektur und Feedback

Durch die Mitgliedschaft in ihren jeweiligen Übungsgruppen in ILIAS haben die Studierenden die Möglichkeit, Dateien in ihrem Gruppenverzeichnis abzulegen. Leider sieht ILIAS keine automatische Benachrichtigung des Übungsleiters oder der Tutoren bei Nutzung dieser Ablage vor.

Um trotzdem eine Benachrichtigung bei der Einreichung von Abgaben und Korrekturen zu realisieren, erstellen wir in jeder Übungsgruppe ein Diskussionsforum. Dieses dient jedoch nicht der Diskussion allgemeiner Themen, sondern nur der Einreichung von Abgaben durch die Studierenden sowie dem Feedback durch die Tutoren. Da für neue Beiträge im Diskussionsforum eine Benachrichtigungsfunktion zuschaltbar ist, werden bei Aktionen wie Abgabe oder Feedback sofort die anderen Mitglieder des Dreier-Teams, der zuständige studentische Tutor und auch der Übungsleiter per E-Mail informiert. Missverständnisse zwischen Übungsleiter und Tutoren können dadurch schnell identifiziert und entsprechende Schadensbegrenzungsmaßnahmen eingeleitet werden. Dadurch, dass die Benachrichtigungen jederzeit dynamisch ein- und ausschaltbar sind, lassen sich bei Ausfällen durch Urlaub oder Krankheit auch Vertretungen unter den Tutoren organisieren.

Da die Studierenden durch den Passwortschutz nur ihrer eigenen Übungsgruppe beitreten können, sehen sie die Abgaben der anderen Gruppen nicht. Die studentischen Tutoren können jedoch alle Abgaben und auch alle Korrekturen einsehen. Das hat den Vorteil, dass sie sich notfalls auch asynchron und ohne Intervention des Übungsleiters gegenseitig helfen können.

Dashboard

Die beschriebenen Abgabenforen mit ihren einzelnen E-Mail-Benachrichtigungen allein reichen einem Übungsleiter jedoch nicht für die Einschätzung der Gesamtqualität der Abgaben und den aktuellen Stand der Korrekturen aus. Deshalb wurde zusätzlich außerhalb von ILIAS das in Abbildung 2 ausschnittsweise dargestellte Dashboard in Form eines Online-Spreadsheets realisiert. Darin werden nur einfache Funktionen zur Eingabe von Daten sowie bedingte Formatierungen eingesetzt. Bei benoteten Abgaben wäre aber auch

Gruppe/ Ü-Leiter/ Tutor			Meilenstein	M2				M3		M
			Abgabe	Analysenot.		Projektplan		Spezifikation		k
			Datum	29.10.	05.11.	29.10.	05.11.	16.11.	23.11.	3
			Versuch	I.	II.	I.	II.	I.	II.	I.
1	D	A		MO		MO		MO		
3	D	M		AO		AO		AO		
5	D	A		AO+		AO+		AR	AO	
7	D	M		AO		AO		AO+		
9	D	A		AO+		AO+		AO-		
11	D	M		DR	AO	DR	AO	AR	AO-	
13	D	A		AO+		AO+		AO		
15	D	M		MO		MO		I1		
17	D	A		MO-		MO		I3		
19	D	M		MO+		MO+		I2		
21	D	A		DR	DO	DR	ME1	DO		
2	V	J		JR	JO	JR	JO	VO		
4	V	S		SR	SO	SR	SO	SL		
6	V	J		JR	JE	JR	JE	JR		
8	V	S		SR	SO	SR	SO	SO		
10	V	J		SO+		SO+		I2		
12	V	S		SR	SO	SO		SO-		
14	V	J		JO+		JO+		JR	I4	
16	V	S		JR	DO-	JR	JO	I3		
18	V	J		SR	SO+	SO		SO-		
20	V	S		JO		JO		JO+		

Abbildung 2: Ausschnitt des Controlling-Dashboards

der Einsatz von einfachen Formeln denkbar. Sowohl die Übungsleiter als auch die studentischen Tutoren haben Schreibzugriff auf das Dashboard. Für die Studenten ist es dagegen nicht zugänglich.

In dem in Abbildung 2 dargestellten Dashboard wird ein Lernszenario abgebildet, bei dem 21 Übungsgruppen von zwei Übungsleitern und vier studentischen Tutoren betreut werden. Die Zuordnung zwischen den Übungsgruppen, Übungsleitern und studentischen Tutoren wird dabei durch die ersten drei Spalten geregelt. In jeder Zeile werden die Abgaben einer Übungsgruppe visualisiert. Pro Abgabe werden zwei Spalten verwendet, da den Studierenden bei einer unbefriedigenden Leistung eine Nacharbeit gestattet wird.

Der erste Buchstabe in jeder Zelle kennzeichnet den Bearbeiter einer Abgabe, der zweite Buchstabe steht für ihren Status. Eine Ausnahme bildet dabei der Buchstabe 'I', der zum Kennzeichnen der Eingangsreihenfolge ('Incoming') der Abgaben verwendet wird. Eine Abgabe kann dabei zwischen den folgenden Status wechseln:

- I (Incoming): Die Übungsgruppe hat eine neue Abgabe eingereicht.
- L (Locked): Die Abgabe wird gerade korrigiert.
- H (Help): Der Tutor ist sich unsicher und braucht Hilfe bei der Bewertung der Abgabe.
- O (Okay): Die Abgabe wurde akzeptiert. Ein Plus oder Minus kennzeichnet dabei besonders gute und eher grenzwertige Leistungen.
- R (Rejected): Die Abgabe wurde abgelehnt, eine Nacharbeit ist erforderlich.
- E (Escalated): Es sind größere Probleme aufgetreten (z. B. Abgabetermin wurde nicht eingehalten), die den Einsatz des Übungsleiters erfordern.

Die einzelnen Zellen werden dabei durch Nutzung bedingter Formatierungen auch farblich gekennzeichnet. So sieht der Übungsleiter auf den ersten Blick, dass es bei der in Abbildung 2 dargestellten Gruppe 6 größere Probleme gibt. Durch die Realisierung des Dashboards als reines Online-Spreadsheet ist die Übersicht dabei stets aktuell, da sämtliche Eingaben der Übungsleiter und Tutoren sofort sichtbar sind. Zudem erfolgt die Versionierung des Dokumentes automatisch, was die Korrektur von erst spät bemerkten Fehleingaben erleichtert.

Durch die fehlende Kopplung des Dashboards an ILIAS hat der Lehrende auch die Flexibilität, den zugrundeliegenden Geschäftsprozess jederzeit anzupassen. Dasselbe Dashboard könnte zudem auch in Verbindung mit einem anderen LMS realisiert werden. Gerade für Hochschulen, an denen unterschiedliche LMS für unterschiedliche Lehrveranstaltungen genutzt werden, könnte dies einen Vorteil darstellen.

Im Grunde genommen könnte das Dashboard auch ganz ohne ein LMS eingesetzt werden, beispielsweise mit einfachen Abgaben per E-Mail. Viele der beschriebenen Vorteile des kombinierten Ansatzes würden dabei jedoch verloren gehen.

Einschränkungen

Der vorgestellte Ansatz wurde erstmals zum Wintersemester 2008/2009 für Lehrveranstaltungen unserer Abteilung genutzt und seitdem schrittweise verfeinert. In unserer täglichen Lehrpraxis hat sich der Ansatz sehr bewährt. Dennoch gibt es einige kleinere Schwächen, die vor allem auf die fehlende Integration des Dashboards in das LMS und die Zweckentfremdung der in ILIAS bereitgestellten Funktionen des Diskussionsforums zurückzuführen sind.

Da das Dashboard technisch nicht an ILIAS gekoppelt ist, müssen die Daten manuell zwischen den beiden Systemen synchronisiert werden. Es wäre daher wünschenswert, wenn neue Abgaben oder erfolgte Korrekturen in Ilias auch automatisch im Dashboard erscheinen würden.

ILIAS bietet zwar die Möglichkeit, ganze Kurse zu archivieren. Leider werden dabei aber die Beiträge in Diskussionsforen nicht archiviert, wodurch die studentischen Abgaben manuell oder mit einem externen Skript archiviert werden müssen.

Auch besteht in ILIAS die Möglichkeit, einen Kurs ins nächste Jahr zu kopieren. Hier ist das Verhalten von ILIAS, dass die Beiträge im Diskussionsforum nicht mitkopiert werden, hingegen vorteilhaft. Da bei dieser Aktion auch die Gruppenpasswörter kopiert werden, müssen sie neu gesetzt werden, was manuellen Aufwand erfordert. Die Zuteilung der Tutoren zu den Übungsgruppen erfolgt leider ebenfalls nicht automatisch, ebenso wie die Ergänzung der Mitgliederliste um die Gruppenzugehörigkeit der Studierenden. Hier ist jeweils das manuelle 'Abgrasen' der einzelnen Übungsgruppen notwendig.

Es gibt bei Nutzung des vorgestellten Ansatzes auch keine technische Möglichkeit, die Korrekturen der studentischen Tutoren erst nach einer Prüfung durch den Übungsleiter freizuschalten. Zwar könnte man ein 'moderiertes' Forum verwenden, dies hätte jedoch den Nachteil, dass auch die Abgaben der Studenten erst freigeschaltet werden müssten, was wiederum das Feedback und die Transparenz innerhalb der Dreier-Teams reduzieren würde.

Soll ILIAS als LMS zum Einsatz kommen, können wir den vorgestellten Ansatz für die Durchführung von Gruppenübungen, deren Korrektur durch studentische Tutoren unterstützt werden soll, dennoch empfehlen. Die zuvor beschriebenen positiven Effekte überwiegen deutlich die geschilderten Nachteile und Probleme.

Fazit und Ausblick

Obgleich er noch einige Integrationsdefizite aufweist, demonstriert der beschriebene Ansatz, welche Verbesserungen damit direkt erzielbar sind. Die bisherigen Erfahrungen mit dem von uns als Online-Spreadsheet realisierten Pilotsystem lassen erkennen, dass der Ansatz auch in der täglichen Lehrpraxis gut funktioniert. Um das gesamte Leistungsvermögen einer größeren Allgemeinheit zur Verfügung zu stellen und die beschriebenen Defizite vollständig zu beseitigen, müsste der Ansatz noch enger in ein bestehendes Learning-Management-System integriert werden.

Sowohl ILIAS als auch Moodle sind sehr modular aufgebaut. Technisch gesehen sollte es also möglich sein, eine Erweiterung zu entwickeln, die das beschriebene Gruppenabgabeverfahren umsetzt. Allein aus der Perspektive unserer Abteilung wäre eine solche Implementierung jedoch nicht wirtschaftlich. Wir würden uns daher sehr freuen, wenn die Entwickler eines der bestehenden LMS den beschriebenen Ansatz aufgreifen und zur Produktreife führen würden.

Literatur

[Ili] *ILIAS*. `http://www.ilias.de`

[Moo] *Moodle*. `http://www.moodle.org`

[Budden 2009] BUDDEN, Dallas: Online Collaboration. In: BASTIAENS, Theo (Hrsg.) ; DRON, Jon (Hrsg.) ; XIN, Cindy (Hrsg.): *Proceedings of World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2009*. Vancouver, Canada : AACE, October 2009, 2398–2401

[Colace u. a. 2003] COLACE, F. ; DE SANTO, M. ; VENTO, M.: Evaluating on-line learning platforms: a case study. In: *Proceedings of the 36th Annual Hawaii International Conference on Systems Sciences*, 2003, S. 9 pp.

[Graf u. List 2005] GRAF, S. ; LIST, B.: An evaluation of open source e-learning platforms stressing adaptation issues. In: *ICALT 2005. Fifth IEEE International Conference on Advanced Learning Technologies*, 2005, S. 163 – 165

[Oevel u. Lange] OEVEL, Gudrun ; LANGE, Gerald: *ZKI LMS-Umfrage*. `http://www.doodle.com/uyvcg2wz6s4bwv6v`

[Schulmeister 2005] SCHULMEISTER, Rolf: *Lernplattformen für das virtuelle Lernen. Evaluation und Didaktik*. 2. Auflage. Oldenbourg, 2005

[Steffens u. Reiss 2009] STEFFENS, Dirk ; REISS, Michael: Blended Learning in der Hochschullehre. Vom Nebeneinander der Präsenzlehre und des E-Learning zum integrierten Blended-Learning-Konzept. In: *HSW: Das Hochschulwesen* (2009), Nr. 4, S. 115–123

Kompetenzorientierte Lehre im Software Engineering

Axel Böttcher und Veronika Thurner, Hochschule München
ab | thurner@cs.hm.edu
Gerhard Müller, TNG Technology Consulting GmbH
gerhard.mueller@tngtech.com

Zusammenfassung

Software Engineering ist eine komplexe Aufgabe, die von den Beteiligten ausgeprägte Kompetenzen in vielen verschiedenen Bereichen fordert. Die Vermittlung dieser Kompetenzen über das reine Fachwissen hinweg ist eine zentrale Aufgabe, die eine Software-Engineering-Ausbildung leisten muss, um Absolventen für den Einsatz in der Praxis zu qualifizieren. Ausgehend von den zu vermittelnden Kompetenzen zeigen wir, wie wir in unserer Lehrpraxis mit Hilfe eines möglichst einfach gehaltenen, aber so komplex wie nötig gestalteten durchgängigen Beispiels sowie durch innovative Lehrmethoden praxisnah eine fundierte Einführung in die komplexen Tätigkeiten des Software Engineerings vermitteln.

Motivation

Anforderungen an Softwaresysteme werden heute zunehmend umfangreicher und komplexer. Analog dazu entwickeln sich auch die Technologien und Werkzeuge für Software Engineering stetig weiter und werden immer mächtiger – und damit selbst komplexer. Da sich diese Technologien und Werkzeuge laufend weiter entwickeln, ist die Halbwertzeit der dazu erworbenen Kompetenzen entsprechend kurz.

Aufgrund der Menge und Komplexität des erforderlichen Wissens und wegen der Größe der zu erstellenden Systeme wird Software in der heutigen Praxis nahezu immer im Team entwickelt. Neben den rein technisch-fachlichen Kompetenzen werden damit in zunehmendem Maße auch Fähigkeiten wichtig, die sich mit der Positionierung einer Einzelperson im Team sowie mit der Integration und Interaktion aller Beteiligten untereinander befassen.

Eine Software-Engineering-Ausbildung, die nicht nur theoretisch fundiert, sondern auch praxisnah und berufsbefähigend sein soll, muss diesen Entwicklungen Rechnung tragen. Hier stößt die Lehre sehr schnell auf das heute bereits klassische Dilemma, dass die verfügbare Lernzeit in der initialen Ausbildungsphase eines Menschenlebens seit Jahren in etwa konstant bleibt, die Menge des verfügbaren und als wichtig empfundenen Wissens aber exponentiell ansteigt (Spitzer, 2006). Da dieser Entwicklung alleine mit erhöhter Packungsdichte in der Wissensvermittlung nicht mehr beizukommen ist, ist zwangsweise Selektion erforderlich. Eine „gute“ Selektion wird somit zu einer der wesentlichen Herausforderungen bei der Konzeption der Software-Engineering-Ausbildung.

Im Folgenden definieren wir zunächst kompetenzorientierte Lernziele für die Software-Engineering-Ausbildung. Diese basieren im Wesentlichen auf drei Quellen:

- klassischer Software-Engineering-Literatur,
- einer umfangreichen Befragung von Absolventen unserer Fakultät aus den letzten fünf Jahrgängen sowie
- Aussagen von Spezialisten aus der beruflichen Praxis über die Eigenschaften und Fähigkeiten, die sie von potenziellen neuen Mitarbeitern/innen erwarten.

Im Anschluss daran beschreiben wir ein Beispielprojekt und die zugehörigen Lehrkonzepte, die wir in den Bachelor-Studiengängen Informatik sowie Wirtschaftsinformatik einsetzen. Abschließend setzen wir die mit unserem Lehrkonzept gemachten Erfahrungen in Beziehung zu den zuvor definierten Kompetenzzielen.

Definition von Lernzielen

Grundlage einer adäquaten Selektion von Ausbildungsinhalten und -methoden ist die Definition der zu erreichenden Lernziele. In der Regel stellt die Definition dieser Lernziele bereits selbst eine Auswahl dar, da in der meist begrenzten verfügbaren Zeit realistischer Weise nicht immer alle eigentlich gewünschten Ziele erreicht werden können (Lehner, 2009).

Ausbildung und Lernziele müssen sich dabei zum einen an Kompetenzen auf unterschiedlichen Wissensgebieten orientieren. Zum anderen sollen sie die Studierenden auf die Übernahme verschiedener Rollen vorbereiten. Analog zu Schott und Ghanbari (Schott u. Ghanbari, 2009) betrachten wir Kompetenzen als diejenigen Eigenschaften und Fähigkeiten, die man besitzen muss, um eine bestimmte Menge von Aufgaben sinnvoll ausführen zu können.

Softwarekompetenzen (SWEBOK)	Inf	WI
Requirements	++	++
Design	++	++
Construction	++	+
Testing	++	+
Maintenance	–	–
Configuration Management	o	o
Engineering Management	o	+
Engineering Process	+	++
Engineering Tools and Methods	+	+
Quality	+	+

Tabelle 1: Unsere Prioritäten für die Vermittlung der Kompetenzen (– –, –, o, +, ++) in den Bachelor-Studiengängen Informatik (Inf) und Wirtschaftsinformatik (WI)

Als Grundlage für die Definition von zu vermittelnden Kompetenzen und daraus abzuleitenden Lernzielen, empfiehlt sich zunächst der „Software Engineering Body of Knowledge“ (Abran u. a., 2005). Die dort genannten Kompetenzfelder haben wir für die Durchführung der Lehre in den Bachelor-Studiengängen Informatik und Wirtschaftsinformatik priorisiert und in Tabelle 1 zusammengefasst. Der Themenbereich „Software Maintenance“ wird aktuell nur am Rande adressiert. Das liegt unter anderem daran, dass Software Maintenance zum einen sehr technologiespezifisch ist und zum anderen stark auf anderen Kompetenzen aufbaut. Diese müssen somit zuerst vermittelt werden, bevor diese Thematik auf technisch anspruchsvollem Niveau behandelt werden kann.

Als ergänzende Informationsquelle haben wir eine Umfrage herangezogen, die unter unseren Absolventen der Informatik und Wirtschaftsinformatik der letzten Jahre durchgeführt wurde – also unter Personen, die sich bereits mit den bei uns erworbenen Fähigkeiten in der Industrie bewähren müssen. Darüber hinaus befragten wir erfahrene Praktiker aus der Industrie, welche konkreten Anforderungen die Praxis an unsere Absolventen stellt. Die auf diese Weise identifizierten Kompetenzbedarfe gliedern wir im Folgenden nach den vier Bereichen Sach-, Methoden-, Selbst- und Sozialkompetenz (vergleiche (Schaeper u. Briedis, 2004)). Dabei ordnen wir Kompetenzen, die mehr als einen dieser Bereiche bedienen, demjenigen Kompetenzbereich zu, zu dem sie nach unserer Einschätzung den stärksten Beitrag leisten.

Sachkompetenz

Der Bereich der Sachkompetenz fokussiert das erworbene Fachwissen und dessen zielgerichtete Anwendung. Hierzu zählen die in Tabelle 1 aufgelisteten Kompetenzbereiche aus dem SWEBOK. Diese wurden aus der Praxis durch die folgenden Kompetenzforderungen ergänzt.

- *Modellierung und Entwurf*
  Um eine „ordentliche“, d.h. professionelle und nachhaltige Softwareentwicklung rein technisch-handwerklich überhaupt zu ermöglichken müssen Software-Ingenieure/innen modellieren und designen können. Dazu gehören unter anderem auch das Verständnis für bzw. der Einsatz von Design Patterns, SOLID-Prinzipien und ggf. Domain Driven Development (Evans, 2003).

- *Beschreibung von Architekturen*
  Software-Ingenieure/innen müssen in der Lage sein, technische und fachliche Architekturen zu konzipieren und so darzustellen, dass sie als klar verständliche Grundlage für die Kommunikation aller Projektbeteiligten dienen können.

- *Einarbeitung in große, fremde Projekte*
  Softwaresysteme entstehen heute häufig im Kontext bereits bestehender Systemlandschaften. Eine wichtige Fähigkeit von Software-Ingenieuren/innen ist somit die Einarbeitung in umfangreiche fremde Projekte. Typische Aufgabenstellung ist das Bug-Fixing in fremdem Code, aber auch das Einbauen neuer Features in ein bestehendes System.

- *Gesamtsicht auf Zusammenspiel von Frameworks*
  Moderne Softwaresysteme sind gekennzeichnet durch das Zusammenwirken vieler Bibliotheken und Frameworks. Jeder dieser Bausteine ist für sich genommen nicht schwierig zu verstehen. Die Integration dieser Frameworks in einem Produkt erhöht die Komplexität jedoch sprungartig. Entsprechend müssen diese Bausteine in der Lehre nicht nur einzeln, sondern auch in ihrer Gesamtsicht vermittelt werden.

- *Test Driven Development*
  Test Driven Development ist in der heutigen Praxis eine Basistechnik der professionellen, nachhaltigen Softwareentwicklung. Damit einher gehen das Erlernen eines sauberen Designs, Continuous Integration, Refactoring und Techniken wie das Pair Programming.

- *Verständnis für Lebensdauer von Code*
  Die Praxis zeigt, dass der Quelltext geschäftsrelevanter Softwaresysteme oft über Jahrzehnte im Einsatz bleibt. Entsprechend müssen Software-Ingenieure/innen ein Verständnis für die Lebensdauer von Code entwickeln die Wichtigkeit von Informationen erkennen, die in „Clean Code“ beschrieben sind (Martin, 2008). Daarüber hinaus müssen Sie in der Lage sein, selbst sauberen Code zu schreiben. Ebenfalls erforderlich ist ein Veständnis für die Kosten von Zeit.

- *Größenordnungen verstehen*
  In der Praxis bestimmen nicht-funktionale Requirements oft zu ca. 80% die Systemarchitek-

tur, während die funktionalen Anforderungen im Vergleich dazu eine eher untergeordnete Rolle spielen. Um auf systematische Weise adäquat performante Systemarchitekturen konzipieren zu können müssen Software-Ingenieure/innen ein Gefühl für Größenordnungen entwickeln, beispielsweise bzgl. algorithmischer Komplexität, Anzahl der Transaktionen pro Sekunde, Speicherbedarf pro Session, IO-Geschwindigkeit, CPU-Geschwindigkeit, Bandbreite, Ressourcenbedarf pro Benutzer oder in Summe, ... Darüber hinaus ist ein Verständnig für die Auswirkungen dieser Werte erforderlich sowie die Fähigkeit, ggf. passende Konsequenzen daraus herzuleiten. Ohne dieses Verständnis werden die erarbeiteten Lösungen nicht auf angemessene Weise skalierbar.

- *Aufwandsschätzung*
  Aufwandsschätzungen bilden nicht nur die Grundlage für viele planerischen und organisatorischen Tätigkeiten im Software Engineering, sondern beispielsweise auch für die Entscheidung zwischen verschiedenen Realisierungsalternativen. Um verlässliche Aussagen zu Machbarkeit, Zeit- und Ressourcenbedarf treffen zu können müssen Software-Ingenieure/innen Aufwände auf realistisch treffende Weise schätzen können.

- *Vorgehensmodelle*
  Der in der Theorie gelegentlich verfolgte Grundgedanke eines universell einsetzbaren Vorgehensmodells („One size fits all") funktioniert in der Praxis in der Regel nicht. Entsprechend müssen Software-Ingenieure/innen verschiedene Vorgehensmodelle wie z.B. Scrum, XP, Crystal-Familie, RUP und V-Modell-XT kennen und deren Vor- bzw. Nachteile und Einsatzgebiete einschätzen können. Insbesondere sind Kompetenzen sowohl zu klassisch ingenieurwissenschaftlichen als auch zu aktuellen agilen Vorgehensmodellen von Nöten.

- *Beherrschung grundlegender Werkzeuge*
  Damit alle Beteiligten auf technischer Ebene reibungsfrei zusammen arbeiten können gehört die Beherrschung grundlegender Werkzeuge zum elementaren Handwerkszeug von Software-Ingenieuren/innen, wie beispielsweise die verwendete IDE oder die Versionsverwaltung incl. Kenntnissen zu geeigneten Branching-Strategien.

Methodenkompetenz

Im Bereich der Methodenkompetenz sind grundlegende Arbeitstechniken zusammengefasst, sowie Fähigkeiten und Verfahren, die ein effektives, ergebnisorientiertes Arbeiten ermöglichen.

- *Denken in Systemen*
  Wer nicht in übergeordneten Strukturen denkt, beschränkt sich auf lokale Optimierungen und tendiert dazu, „echte" Probleme nicht anzugehen. Strukturprobleme anzugehen erfordert auch Mut.

- *„Faulheit", DRY-Prinzip*
  Das Prinzip des „Don't Repeat Yourself" bedeutet, im Normalfall keine eigenen Frameworks bauen wollen, Erkennen der Notwendigkeit Automatisierung auf allen Ebenen in möglichst großem Umfang eu erreichen. Software lebt von der Wiederverwendung bewährter Konzepte und Lösungen. Die so genannten „Not Invented Here"-Lösungen bedeuten eine Verschwendung von Ressourcen.

- *Vernünftiges Maß an Pragmatismus*
  In der beruflichen Praxis müssen ständig Entscheidungen getroffen werden, ob etwa die Menge der Tests ausreichend ist, ob ein noch abstrakteres Framework einzusetzen ist, oder ob an einer Stelle noch mehr Optimierungsaufwand getrieben werden soll etc. Um in solchen für ein Projekt kritischen Fragen vernünftig abwägen zu können, ist viel Erfahrung erforderlich.

- *Prägnantes Dokumentieren von Anforderungen*
  Die in Projekten auftretenden Probleme sind oftmals weniger technischer Natur als vielmehr organisatorischer Art. Software-Ingenieure/innen müssen oft zuerst herausfinden, welches Problem der Kunde eigentlich gelöst haben will. Hier ist es insbesondere wichtig, Requirements kurz und prägnant aufschreiben zu können.

- *Ausdrucks- und Schreibfähigkeit*
  Software-Ingenieure/innen müssen in der Lage sein, aussagekräftige, klar verständliche Dokumente und Dokumentation zu schreiben. „Ordentliche" Dokumente sind in der Praxis eine zentrale Voraussetzung dafür, dass Lösungen bzw. Lösungsideen von den Beteiligten verstanden und von den Nutzern akzeptiert und eingesetzt werden und dass Lösungen weiterentwickelbar sind, ggf auch über einen längeren Zeitraum und mehrere Entwicklergenerationen hinweg.

Selbstkompetenz

Zum Bereich der Selbstkompetenz zählen Fähigkeiten, die eigene Situation wahrzunehmen, eigene Bedüfnisse zu erkennen und zu artikulieren, eigenverantwortlich zu handeln und über all diese Punkte selbstkritisch und konstruktiv zu reflektieren.

- *Reflexions- und Kritikfähigkeit*
  Reflexions- und Kritikfähigkeit sind Grundvoraussetzungen für eine erfolgreiche Berufstätigkeit, nicht nur im Umfeld des Software Engineerings. Ziel der Reflektion ist es, sich selbst und die (Projekt-)Umgebung immer wieder zu hinterfragen, um so sicherzustellen, dass „das Richtige" getan wird. Kritikfähigkeit ermöglicht es, das ggf. kritische Feedback Anderer nicht als persönlichen Angriff zu werten, sondern als Verbesserungspotenzial anzuerkennen und daraus zu lernen. Auch

das Geben von konstruktiv verwertbarem Feedback ist in diesem Kontext eine wesentliche Schlüsselkompetenz.

- *Dinge „richtig“ machen*
  Die Informatik allgemein, und damit auch das Software Engineering, erfordern nicht nur eine präzise Denkweise und ein hohes Maß an Wissen, sondern auch eine ständige Aktualisierung dieses Wissens, um mit den ständig neuen Entwicklungen Schritt halten zu können. Der Wunsch, Dinge „richtig“ zu machen, gepaart mit dem Streben nach lebenslangem Lernen und Verbessern, sollte in Software-Ingenieuren/innen intrinsisch motiviert sein. *Ohne* diese Eigenschaft werden die vielen Innovationen und die steigende Komplexität eher als Belastung empfunden und nicht als Anreiz. In einer solchen Konstellation *kann* die Informatik mit ihren vielen Änderungen auf Dauer keinen Spaß machen – und dann werden auch keine vernünftigen Ergebnisse erzielt.

- *Ethisches Handeln*
  Softwaresysteme durchziehen heute nahezu alle Bereiche des beruflichen und privaten Lebens. Damit beeinflusst die Informatik in hohem Maße, wie andere Menschen leben und arbeiten. Software-Ingenieure/innen gestalten somit mehr oder weniger direkt die Zukunft. Dies bringt nicht nur eine kaum absehbare Vielzahl an (Geschäfts-)Möglichkeiten mit sich, sondern auch ein sehr hohes Maß an Verantwortung für die Rolle der Informatik in der Welt. Eine ganzheitliche Software-Engineering-Ausbildung sollte in angehenden Software-Ingenieuren/innen zumindest das Bewusstsein für diese Verantwortung wecken, sowie ein gewisses Maß an kritischem Weitblick aufbauen und fördern.

Sozialkompetenz (Soft Skills)

Der Bereich der Sozialkompetenz fokussiert schließlich die Fähigkeit, die Bedürfnisse und Interessen anderer Menschen wahrzunehmen, sich mit diesen auseinander zu setzen und konstruktiv und erfolgreich mit anderen Menschen zusammenzuarbeiten.

- *Verständnis für Andere*
  Software entsteht nicht in einem luftleeren Raum, sondern durch die Zusammenarbeit verschiedenster Gruppen, wie beispielsweise Management, Benutzer, Fachexperten, QA, Operations und vielen anderen. Schon die Zielfindung in einem Entwicklungsprojekt erfordert intensive Kommunikation zwischen den Beteiligten sowie eine offene Wahrnehmung der Belange und Interessen aller Beteiligten.

- *Teamfähigkeit*
  In der Praxis ist Software heutzutage fast immer das Produkt eines Teams. Entsprechend kommt der Fähigkeit, sinnvoll, effektiv und gerne (!) in einem Team zusammenzuarbeiten eine fundamentale Bedeutung zu: Wer das nicht kann, kann nichts von dem, was er/sie kann, sinnvoll im komplexeren Umgebungen einbringen. Zu den zentralen Teilfähigkeiten gehören unter anderem eher pragmatische Punkte wie ein freundlicher Umgangston, nett sein und zuhören können. Gerade in kritischen Situationen gewinnt darüber hinaus theoretisch fundiertes Wissen über Team-Regeln an Bedeutung, wie beispielsweise das Tucker-Modell, das Vier-Ohren-Modell von Schulz von Thun. Ebenfalls nützlich sind Change Management, wie z.B. (Rising u. Manns, 2004), sowie ein Grundstock an psychologischem Grundwissen, wie beispielsweise in (Vigenschow u. Schneider, 2007) beschrieben.

Nicht alle dieser eigentlich wünschenswerten Kompetenzen können in der zur Verfügung stehenden Lehr- und Lernzeit im vollen Umfang vermittelt werden. In den Kompetenzbereichen, die wir nicht vollständig bedienen können, soll unsere Ausbildung zumindest das Bewusstsein für die Notwendigkeit dieser Kompetenzen wecken, indem sie deutlich macht, für welche Aufgaben, Disziplinen und Rollen diese Kompetenzen erforderlich sind.

Lehrkonzept

Um dem Curriculum der von uns bedienten Bachelor-Studiengänge „Informatik“ und „Wirtschaftsinformatik“ mit nur einem Beispiel gerecht zu werden konzipierten wir dieses als eine moderne Webapplikation. Bislang kommt dieses Beispiel in folgenden Modulen zum Einsatz, deren Umfang jeweils bei zwei SWS seminaristischem Unterricht und zwei SWS Praktikum liegt und mit fünf ECTS-Punkten gewichtet ist:

- Modul „Software Engineering I“ im 3. Semester des Bachelorstudiengangs Wirtschaftsinformatik
- Modul „Software Engineering II“ im 4. Semester des Bachelorstudiengangs Wirtschaftsinformatik
- Modul „Software-Architektur“ im 4. Semester des Bachelorstudiengangs Informatik

Die zu Größe der zu unterrichtenden Gruppen liegt bei etwa 45 Studierenden. Für das Praktikum werden die Gruppen jeweils noch einmal geteilt.

Thematisch bildet das Beispiel ein Verleihsystem namens *ShareIt* ab, über das ein Freundes- und Bekanntenkreis dingliche Ressourcen unterschiedlicher Art (wie z.B. Bücher, DVDs, Werkzeug, etc.) untereinander verleihen bzw. gemeinschaftlich nutzen kann. Technisch ist die Beispielapplikation als klassische drei-Schicht-Architektur realisiert. Fachlich gliedert sich das Beispiel bisher in die Hauptbereiche „Benutzerverwaltung“, „Verwaltung der Leihobjekte“ und „Ausleihe“ (siehe Abbildung 1).

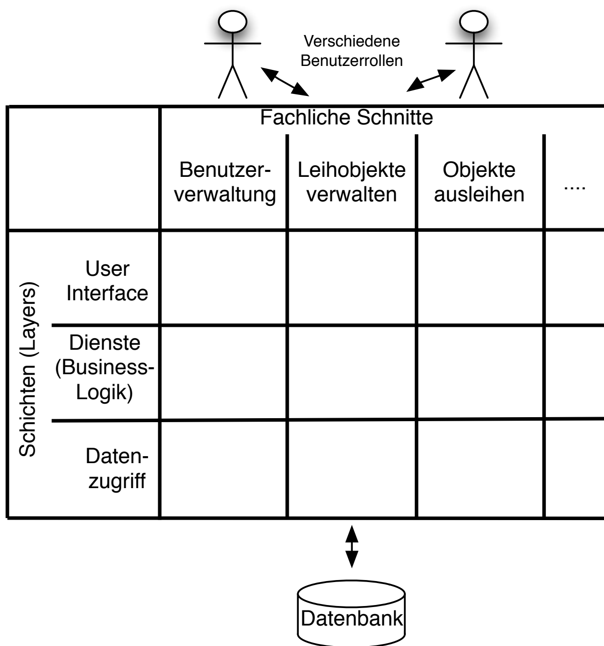


Abbildung 1: Struktur des durchgängigen Beispiels

Für den ersten fachlichen Schnitt, die Benutzerverwaltung, werden Planung, Analyse- und Entwurfsdokumente, Implementierung und umfangreiche Tests den Studierenden als Lernmaterialien zur Verfügung gestellt. Diese dienen sowohl als veranschaulichendes Beispiel als auch als Vorlage für die von den Studierenden durchzuführenden Projektarbeiten.

Der zweite fachliche Schnitt „Verwaltung der Leihobjekte“ ist ebenfalls ausgearbeitet, je nach Fokus der Lehrveranstaltung aber nur teilweise oder gar nicht für die Studierenden verfügbar. In der Lehrveranstaltung „Softwarearchitektur“ (Bachelor Informatik) erhalten die Studierenden die vorgefertigten Analyse- und Entwurfsdokumente, die sie anschließend in eine funktionsfähige Implementierung umsetzen. In „Software Engineering“ (Bachelor Informatik und Wirtschaftsinformatik) dagegen erstellen die Studierenden selbst die benötigten Analyse- und Entwurfsdokumente und setzen anschließend ihre eigene Spezifikation um. Bei Bedarf ist das Beispiel um zusätzliche fachliche Schnitte erweiterbar.

Wir setzen das Lehrmaterial in den einzelnen Veranstaltungen wie folgt ein:

Software Engineering I: Hier stehen die Kompetenzfelder Requirements, Design sowie Engineering Management und Process im Vordergrund. Vorgabe für das Praktikum ist daher lediglich eine ausgearbeitete Spezifikation für den ersten fachlichen Schnitt „Benutzerverwaltung“. Die Studierenden müssen mindestens einen weiteren fachlichen Schnitt selbstständig spezifizieren.

Software Engineering II: Dieses Modul deckt den Kompetenzbereich Construction ab. Hier müssen die Studierenden den im ersten Teil spezifizierten fachlichen Schnitt implementieren. Als Vorgabe erhalten sie eine referenzimplementierung des ersten Schnittes.

Software-Architektur: Lernziel ist hier die Fähigkeit, moderne Architekturen für komplexe Software-Systeme zu bewerten, zu entwerfen, zu realisieren und zu betreiben (kompetenzbereiche design. Construction, Testing). Die Studierenden bekommen für den ersten fachlichen Schnitt sowohl eine Spezifikation als auch deren Implementierung als Vorgabe. Sie müssen den zweiten fachlichen Schnitt implementieren und bekommen dazu die Spezifikation als Vorgabe.

Als Lehrmethode im seminaristischen Unterricht wurde eine Mischung aus interaktiver Vorlesung, Lernen durch Lehren, Gruppenpuzzle und Projektarbeit in wechselnden Teams eingesetzt.

Abgleich Lehrkonzept vs. Lernziele

Nach den bisherigen Erfahrungen adressiert unser Lehrkonzept eine Vielzahl der Schlüsselkompetenzen, die aus der Praxis geforderten wurden.

Sachkompetenz

Die Studierenden durchlaufen im Rahmen der Projektarbeit mindestens einen kompletten Softwareentwicklungszyklus, vom Skizzieren erster Anforderungen bis hin zum Integrationstest. Das vorgefertigte, durchgängige Beispiel hilft dabei, neu eingeführte Techniken und deren zielgerichtete Verwendung zu veranschaulichen. Des Weiteren dient es als Anhaltspunkt für die Lösungsbestandteile, welche die Studierenden im Rahmen ihrer Projektarbeit nach und nach selbst entwickeln.

Das Beispiel ist aktuell auf der Basis von Servlets implementiert. Als Frameworks kommen Hibernate, Google Guice (Dependency Injection), Apache Click, Log4j und Selenium zum Einsatz. Die zugehörigen Technologien und Frameworks werden in der Lehrveranstaltung sukzessive vermittelt und von den Studierenden bei der Erstellung ihrer eigenen Lösungsanteile entsprechend weiterverwendet.

Für die Modellierung stehen verschiedene Modellierungswerkzeuge zur Verfügung, beispielsweise der IBM Rational Software Modeler oder die UML2-Modellierungsumgebung der aktuellen Eclipse-Version. Entwickelt wird mit dem JEE-Plug-in-Set von Eclipse.

Die Software des vorgefertigten Beispiels wird den Studierenden in einem Subversion-Repository zur Verfügung gestellt. Über dieses Repository koordinieren die Studierenden den Austausch der im Team erstellten Artefakte. Des Weiteren geben sie darüber ihre Ergebnisse ab.

Im Rahmen der Projektarbeit experimentieren die Studierenden mit verschiedenen Vorgehensmodellen. Neben einem klassischen, iterativ-inkrementell orien-

tierten ingenieurmäßigen Vorgehen kommt als Repräsentant der agilen Ansätze auch Scrum zum Einsatz.

Methodenkompetenz

Um die Studierenden nicht bereits am Anfang mit einer großen Vielzahl verwendeter Technologien zu überfordern, dient ein minimalistisches Exzerpt dieses Beispiels als niederschwelliger Einstieg für den Lernprozess. So wird an einem zunächst noch bewusst sehr einfach gehaltenen System zunächst ein Überblick über die wesentlichen Zusammenhänge zwischen Modellen und Implementierung, sowie innerhalb der Implementierung über das Zusammenspiel der einzelnen Technologien erarbeitet. Darauf aufbauend wird dieses System sukzessive weiter ausgebaut und nach und nach um entsprechende Komplexitäten erweitert, nicht nur bzgl. der realisierten Funktionalität, sondern auch bzgl. der verwendeten Technologien.

Dadurch, dass das Beispiel hinreichend komplex ist und die Aufgabenstellung nicht alle Entscheidungen vorgibt, müssen die Studierenden an verschiedenen Stellen abwägen und entscheiden, wie weit sie bestimmte Kerntechniken anwenden (beispielsweise hinsichtlich des Detallierungsgrades bei der Modellierung, oder bzgl. der Testabdeckung). Da auch innerhalb des studentischen Projektes Zeit eine knappe Ressource darstellt, wird die Notwendigkeit der Abwägung zwischen Perfektionismus und pragmatischer Einschränkung des investierten Aufwandes erlebbar.

Die Ergebnisse, die die Studierenden im Rahmen der praktischen Projektarbeit erstellen, umfassen neben dem eigentlichen Quelltext auch weitere Artefakte des Software Engineerings, beispielsweise Projektpläne, Anforderungsdokumente, Architekturmodelle und Ähnliches. Da diese Dokumente in die Gesamtbewertung mit einfließen, spendieren die Studierenden zwar meist ein gewisses Maß an Sorgfalt in deren Erstellung, sehen diese Dokumente zunächst aber oft als ein notwendiges Übel an. Das Verständnis für die fachliche Notwendigkeit dieser Dokumente (und damit eine intrinsiche Motivation, diese Dokumente sorgfältig zu erstellen und weiter zu pflegen) erwächst im Laufe der Lehrveranstaltung aus der Dauer und dem Umfang der Projektarbeit. Typischerweise stellen die Studierenden nach wenigen Wochen fest, dass ihr Projekt ohne klare Dokumentation ineffektiv wird, weil Anforderungen unklar und doppeldeutig sind, bereits getroffene Entscheidungen nicht mehr einheitlich erinnert werden oder weil Uneinigkeit darüber besteht, wer wann welche Schritte als nächstes durchzuführen hat.

Selbstkompetenz

Im Rahmen der verschiedenen Phasen der Projekt- und Teamarbeit erhalten die Studierenden Feedback von Kommilitonen und Dozenten, sowohl über ihren Arbeitsprozess als auch über die erzielten Ergebnisse. Beispielsweise stellen einzelne Projektteams Teile ihrer Arbeiten den anderen Gruppen vor. Ziel dabei ist das Erlernen des konstruktiven fachlichen Austausches, sowohl in der Rolle des Präsentierenden als auch in der Rolle des kritisch Hinterfragenden. Beide Rollen fallen erstaunlich vielen Studierenden zunächst schwer. Eine wichtige Voraussetzung für die Vermittlung von Reflexions- und Kritikfähigkeit besteht darin, in der Lehrveranstaltung eine Atmosphäre der vertrauensvollen Zusammenarbeit zu schaffen, in der alle Beteiligten auf Augenhöhe wertschätzend miteinander umgehen.

Bzgl. der Vermittlung ethischen Handelns stoßen wir mit der Ausbildung in dieser Form schnell an Grenzen. Als Dozenten können wir hier im Wesentlichen sensibilisieren, den Blickwinkel auf grundlegende Fragestellungen richten und natürlich versuchen, als gutes Beispiel voranzugehen. Die hinter dem ethischen Handeln liegenden Wertesysteme lassen sich „mal eben so nebenbei" aber nur ansatzweise vermitteln – und auch nur dann, wenn die der Hochschule vorgelagerte Erziehung und Bildung den Grundstock dafür gelegt hat.

Sozialkompetenz

Durch den relativ hohen Anteil an Teamarbeit kommt dem Bereich der Sozialkompetenz quasi als „Hintergrundprozess" eine fortlaufend wichtige Rolle zu. Bereits beim Bilden der Teams und bei der Aufteilung der zu erledigenden Arbeiten treffen unterschiedliche Interessen und Vorlieben aufeinander. Die Dozenten beobachten aufmerksam, aber unaufdringlich das Zusammenspiel innerhalb der einzelnen Teams. Bei Bedarf greifen sie (möglichst minimalistisch) in das Geschehen ein, beispielsweise durch das Setzen von gruppendynamischen Impulsen. Des Weiteren moderieren sie zu ausgewählten Meilensteinen der Projektarbeit den Prozess der Gruppenreflexion über Ergebnisse und Ablauf der Teamarbeit.

Ergänzend wird theoretisch fundiertes Grundlagenwissen zu Sozialkompetenzen in spezialisierten Veranstaltungen vermittelt und gezielt praktisch eingeübt.

Zusammenfassung und Ausblick

Mit dem hier vorgestellten Beispiel lassen sich viele zentrale Aspekte des Software Engineerings von der Machbarkeitsanalyse bis hin zum Abnahmetest zeigen. Der besondere Fokus liegt dabei auf der Integration der einzelnen Disziplinen. Dadurch werden für die Lernenden die Zusammenhänge zwischen einzelnen Arbeitsschritten und Artefakten klar ersichtlich und nachvollziehbar. Insbesondere werden dadurch die Auswirkungen einzelner Entwicklungsentscheidungen greifbar verdeutlicht. Das Beispiel und die zugehörigen Lernmaterialien ermöglichen so einen ganzheitlichen Einstieg in die komplexe Materie des Software Engineerings.

Im Rahmen einer qualitativen Evaluierung haben unsere Studierenden die folgenden repräsentativen Aussagen gemacht. Als positiv wurde die Praxisnähe

des Beispiels empfunden, sowie die Tatsache, dass in den Lehrveranstaltungen ein Beispiel von realistischer Komplexität behandelt wird. Diese Komplexität wurde jedoch gleichzeitig auch als negativ empfunden, da zu Beginn der Veranstaltung sehr viele Dinge gleichzeitig zu lernen sind, beispielsweise verschiedene Frameworks. Des Weiteren wurde die Größe der Aufgabe kritisiert.

Der beschrittene Weg, für ein signifikant großes Projekt einzelne Teile in einer ausgearbeiteten Form vorzugeben, die auch kritischen Blicken aus der Praxis Stand hält, hat sich also im Ansatz bewährt. An manchen Stellen haben wir einen Teil unserer Studierenden mit der Komplexität des Beispiels aber offenbar überfordert.

In folgenden Punkten haben wir konkret Verbesserungspotenzial sowohl für das Beispiel als auch für die eingesetzten didaktischen Methoden identifiziert:

- *Kleinere Wissenseinheiten vermitteln*
  Die Darstellung von Technologien und Basistechniken müssen wir gezielt in einem überschaubaren Kontext aufbereiten.

- *Referenzimplementierung inkrementell einführen*
  Wir werden Referenzimplementierungen zukünftig in kleineren Zwischenschritten vorgeben. Zum Einstieg haben wir ein minimalistisches Beispiel gebaut, das sich auf einen Server (Tomcat, Jetty) „deployen“ lässt und einen Login für einen hart codierten Benutzer mit entsprechendem Service, aber ohne eine Datenbank, realisiert.

- *Mehr Diskussion individueller Lösungen*
  Wir wollen mit den Studierenden stärker deren Lösungen diskutieren. Hier stoßen wir allerdings von der Betreuungsrelation und unserem Zeitbudget an Grenzen.

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

[Evans 2003] EVANS, Eric: *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Longman, Amsterdam, 2003

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

[Martin 2008] MARTIN, Robert C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall International, 2008

[Rising u. Manns 2004] RISING, Linda ; MANNS, Mary L.: *Fearless Change: Patterns for Introducing New Ideas*. Addison-Wesley, 2004

[Schaeper u. Briedis 2004] SCHAEPER, H. ; BRIEDIS, K.: *Kompetenzen von Hochschulabsolventinnen und Hochschulabsolventen, berufliche Anforderungen und Folgerungen für die Hochschulreform*. HIS-Kurzinformation, 2004

[Schott u. Ghanbari 2009] SCHOTT, F. ; GHANBARI, S. A.: Modellierung, Vermittlung und Diagnostik der Kompetenz kompetenzorientiert zu unterrichten – wissenschaftliche Herausforderung und ein praktischer Lösungsversuch. In: *Lehrerbildung auf dem Prüfstand* 2 (2009), Nr. 1, S. 10–27

[Spitzer 2006] SPITZER, Manfred: *Lernen: Gehirnforschung und die Schule des Lebens*. Spektrum Akademischer Verlag, 2006

[Vigenschow u. Schneider 2007] VIGENSCHOW, Uwe ; SCHNEIDER, Börn: *Soft Skills für Softwareentwickler – Fragetechniken, Konfliktmanagement, Kommunikationstypen und -modelle*. dpunkt.verlag, 2007

Vier Jahre Software-Engineering-Projekte im Bachelor – ein Statusbericht

Stephan Kleuker, Hochschule Osnabrück
Frank M. Thiesing, Hochschule Osnabrück
{s.kleuker,f.thiesing}@hs-osnabrueck.de

Zusammenfassung

Software-Engineering-Projekte wurden als berufspraktische Ausbildungsinhalte als Lehrveranstaltung in viele Informatik-Bachelor-Studiengänge aufgenommen.

Der Artikel fasst die Erfahrungen der Autoren bei der Organisation, der Durchführung und mit den Ergebnissen der Projekte aus den Anfangsjahren der Bachelor-Einführung an den Hochschulen RheinMain und Osnabrück zusammen. Es werden Erfolgsfaktoren und mögliche Verbesserungspotenziale herausgearbeitet.

Ausgangssituation

Mit der Umstellung der Informatik-Studiengänge auf Bachelor- und Masterabschlüsse wurden auch die zugehörigen Studieninhalte neu strukturiert. Generell enthalten alle Bachelor-Informatikstudiengänge einen wesentlichen Kernanteil aus dem Bereich Software-Engineering, beginnend mit der Programmierausbildung, meist weitergeführt mit objektorientierter Modellierung und Design. In einige Studiengänge wurden neben Vorlesungen zum Software-Engineering auch Software-Engineering-Projekte mit in den Studienplan aufgenommen. Dies war u. a. an der Hochschule RheinMain und der Hochschule Osnabrück der Fall, die jeweils eine Veranstaltung mit 10 Kreditpunkten (KP), aufgeteilt in eine begleitende Vorlesung (3 KP) und ein von 6-10 Studierenden zu bearbeitendes Projekt in ihren Lehrplänen verankerten. Die ersten drei Veranstaltungen wurden jeweils von den Autoren organisiert und durchgeführt. Durch den Wechsel des ersten Autors besteht somit die Möglichkeit, die Organisation, die Erfolge und Verbesserungsmöglichkeiten der Veranstaltungen aus einem inneren Blickwinkel heraus zu vergleichen.

Historie

Softwarepraktika dienen in vielen Informatikstudiengängen schon seit einiger Zeit zur Einübung von Softwaretechniken. So liegen Erfahrungen in der Anwendung objektorientierter Ideen in der Grundausbildung Softwaretechnologie vor, die in praxisnahen Projektsituationen erworben werden (Demuth, Hußmann, Zschaler, Schmitz, 1999). Die Bedeutung der Teamarbeit bei der Einbeziehung von Projekten in die Ausbildung beschreiben auch (Claus, Reissenberger, 1997). Weil die spezifischen Techniken in der Softwaretechnik schnell veralten, ist es wichtig, dass die Studierenden lernen, ihre eigene Arbeitsweise immer wieder zu reflektieren und in Praxisprojekten zu erproben (Lewerentz, Rust, 2001). Dabei dient der Softwareentwicklungsprozess als Lerngegenstand. Der Einsatz neuer Medien in der Lehre ermöglicht derartige Lehrformen, die stärker dem entdeckenden Lernen und der Wissenskreierung verpflichtet sind als traditionelle Lehrveranstaltungen (Kopka, Alfert, 2002).

Hochschulabgängern von Ingenieurstudiengängen fehlte häufig Projekterfahrung sowie Erfahrung mit Teamarbeit. (Göhner, Bitsch, Mubarak, 2005) beschreiben, wie Studenten in kleinen Teams im Rahmen eines Fachpraktikums Softwaretechnik lernen wie kleine Firmen einen Auftrag zur Softwareentwicklung termingerecht zu bearbeiten. (Stoyan, Glinz, 2005) weisen darauf hin, dass gerade die wirksame Ausbildungsform Praktikum eine besondere didaktische Herausforderung darstellt. Zu den didaktischen Zielen gehören eine hohe Motivation der Studierenden, eine wirksame und individuelle Leistungskontrolle, Praxisbezug sowie erfolgreicher Umgang mit unterschiedlichen Vorkenntnissen.

Rahmenbedingungen

Die Software-Engineering-Projektveranstaltung findet an beiden Standorten jeweils nach Abschluss der Programmierausbildung und Vorlesungen zum Thema objektorientierte Analyse und Design im vierten (HS RheinMain) und fünften (HS Osnabrück) Semester statt. Innerhalb der begleitenden Vorlesung werden ausgewählte Themen des Software Engineering (u. a. Qualitätssicherung, Versionsmanagement), eine Einführung in das Projektmanagement und Grundlagen von Soft-Skills behandelt. Der Vorlesungsanteil wird mündlich geprüft (30%), das Projekt davon getrennt bewertet (70%).

Während die Vorlesungen an beiden Hochschulen für Lehrende gleich abgerechnet werden, können an der HS RheinMain vier Stunden pro Praktikumsgruppe, die typischerweise zwei Projektgruppen umfasst, die das gleiche Thema bearbeiten, und an der HS Osnabrück 0.25 Stunden pro Studierenden im Projekt abgerechnet werden. An der HS RheinMain wird die Stundenanzahl auf zwei gesenkt.

Die Findung von Projektthemen findet durch wiederholte Aufrufe per E-Mail und persönlicher Ansprache durch die Veranstalter der Vorlesung statt. Während es an der HS RheinMain schwierig war, mehrere Kollegen zu motivieren, herrscht an der HS Osnabrück ein Überangebot, was auf der sehr lebendigen Forschungslandschaft der Hochschule beruht.

Die Projekte selbst werden auf zwei- bis vierseitigen Dokumenten beschrieben. Die den Studierenden zugängliche Beschreibung wird meist durch eine kurze Projektvorstellung innerhalb der Vorlesung ergänzt. Danach können sich die Studierenden ab einem bestimmten Zeitpunkt für Projekte eintragen. Nach Abschluss der Eintragung werden einvernehmlich für alle Beteiligten Lösungen für Projekte gesucht, die zu wenig Teilnehmer (unter sechs, in Ausnahmen unter vier) haben.

Die eigentlichen Projekte können von beliebigen Veranstaltern, typischerweise Hochschuldozenten, durchgeführt werden und müssen nur als Schwerpunkt eine Software-Entwicklungsaufgabe haben.

Das Projekt und die Vorlesung werden formal möglichst voneinander getrennt, damit möglichst viele Projektvorschläge eingehen, auch wenn die Veranstalter die Vorlesung nicht kennen. Damit zwischen den Projekten eine Vergleichbarkeit entsteht, wurden für die Veranstalter Rahmenbedingungen definiert, die auch für die Studierenden einsehbar sind. Die Rahmenbedingungen (Kleuker, Thiesing, 2010) enthalten im Wesentlichen folgende Forderungen für den Projektveranstalter (PV):

- Der PV muss sich seiner nicht einfach zu verknüpfenden Rollen als Kunde, Berater, Mediator und Bewerter im Klaren sein.
- Der Veranstalter muss für die Studierenden als Ansprechpartner kurzfristig zur Verfügung stehen.
- Die interne Projektorganisation soll den Studierenden im Wesentlichen selbst überlassen werden.
- Die Rahmentermine für mögliche Zwischenabnahmen und die Endabnahme sollen frühzeitig vereinbart werden.
- Die Endabnahme erfolgt mit einer hochschulöffentlichen Präsentation, die um eine Fragerunde und Vorführungen ergänzt werden.
- Die typischen Ergebnisse eines Entwicklungsprojekts, die Anforderungsanalyse, die Systemspezifikation, das Programm und die Systemtestspezifikation sollen am Ende vorliegen.
- Der PV sollte sich im engen Zeitrahmen über den Projektstand informieren und die Entwicklung der einzelnen Studierenden im Projekt kritisch beobachten.
- Die Hochschule stellt einen Server mit Versionsmanagement (Subversion) zur Verfügung, die Einrichtung und Nutzung weiterer Werkzeuge muss die Projektgruppe planen. Der PV hat vor der Veranstaltung die Durchführbarkeit, z. B. das Vorhandensein benötigter funktionierender Hardware, zu gewährleisten.
- Persönliche Probleme sollen zunächst projektintern gelöst werden, dann steht der PV als Mediator und erst zuletzt als Verantwortlicher der Lehrveranstaltung zur Verfügung
- Die Studierenden haben Stundenzettel zu führen, auf denen die Dauer und das aktuell bearbeitete Thema erfasst werden.
- Bei der Notenfindung wird die individuelle Leistung unter Berücksichtigung des Gesamtprojektergebnisses berücksichtigt. Der Dozierende, der die begleitende Vorlesung veranstaltet, kann beratend bei der Notenfindung hinzugezogen werden.

Zusätzlich wird an der HS Osnabrück als Abschluss der Veranstaltung eine Projektmesse durchgeführt, auf der alle Projekte kurz über ihre Arbeiten vortragen, die Ergebnisse vorführen und auf einem Poster zusammenfassen. Diese Veranstaltung wird auch für jüngere Semester zur Information durchgeführt, da teilweise die Möglichkeit von Anschlussprojekten besteht.

Durchgeführte Projekte

An der HS Osnabrück wurde ein relativ breites Spektrum von Projekten angeboten und durchgeführt. Es reichte von klassischen Datenverwaltungsaufgabenstellungen mit Java und Access zur

Verwaltung von Mathematikaufgaben in Word, über ein Thema zur komponentenbasierten Software-Entwicklung in .NET, das in Zusammenarbeit mit einem Kooperationsunternehmen durchgeführt und von einer Exkursion flankiert wurde, weiter über das Redesign eines Unternehmensplanspiels mit MySQL, das von einem Kollegen der Wirtschaftsinformatik betreut wurde, bis hin zur Entwicklung eines autonom fahrenden Fahrzeugs zur Teilnahme am Carolo-Cup, der von der TU Braunschweig veranstaltet wird. Letzteres Thema wurde im darauffolgenden Semester mit einem neuen Team fortgesetzt.

Einige Themen kamen speziell aus dem Bereich der Medieninformatik, die in Osnabrück vom größten Teil der Studenten studiert wird. Wieder andere Themen griffen den Bereich der Mobilkommunikation auf, wie zum Bespiel der "Android Context Twitter", eine Anwendung auf Smart-Phones, mit deren Hilfe ein Nutzer digitale virtuelle Notizen oder ein digitales Graffiti hinterlegen kann.

An der HS RheinMain wurden hauptsächlich Projekte durchgeführt, die in Java verteilte Systeme realisierten, da durch eine Client-Server-Aufteilung sich bereits sinnvolle Projektstrukturen andeuten. Neben einer Spielesammlung wurden graphische Editoren für ER-Diagramme und Aktivitätsdiagramme entwickelt, an denen koordiniert gleichzeitig mehrere Personen an Diagrammen arbeiten. Ein Editor wurde nach Projektabschluss noch weiter gepflegt und steht im Internet (ShUND, 2008) zur Verfügung.

Im Rahmen forschungsnaher Projekte wurden u. a. ein Liederverwaltungssystem für einen iPod entwickelt und eine Reimplementierung einer numerischen Bibliothek durchgeführt.

Teilnehmer

Als Einleitung zu Aspekten des Umgangs der Projektteilnehmer miteinander wurde an beiden Hochschulen eine Persönlichkeitsanalyse angelehnt an Meredith Belbin (Belbin, 2010) durchgeführt. Die Studierenden werden aufgefordert zu bestimmten Rahmenbedingungen wie „Wie verhalte ich mich mit anderen Leuten im Projekt?“ aus acht Reaktionen, z. B. „Ich kann andere Leute beeinflussen ohne Druck auszuüben.“ oder „Man kann sich auf meine originellen Ideen verlassen.“, die auszuwählen, die auf sie selbst zutreffen. Insgesamt sind zehn Punkte auf die jeweils acht Reaktionen zu verteilen, wobei möglichst eine klare Schwerpunktbildung (z. B. 6-3-1) erfolgen soll. Jede der acht Reaktionen steht für eine bestimmte Art von Charaktertypen, die alle positive wie negative Eigenschaften haben, so dass die Studierenden eine Selbsteinschätzung erhalten.

Verkürzt dargestellt werden folgende Typen zur Auswertung genutzt:

CW Company Worker: diszipliniert, zuverlässig, setzt vorgegebene Aufgaben um; neuen Ideen gegenüber kritisch

CH Chairman: dominant, extrovertiert, will externe Ziele erreichen, bringt Teams zum Arbeiten; eher nicht sehr kreativ

SH Shaper: dynamisch, vorantreibend, fordernd; ungeduldig, reizbar

PL Plant: kreativ, unorthodox, löst komplexe Probleme; introvertiert, wenig kritikfähig

RI Resource Investigator: extrovertiert, kommunikativ, vermittelt Kontakte; lässt Ideen schnell fallen, benötigt Impulse von außen

ME Monitor Evaluator: strategisch, wägt alle Optinionen ab; braucht lange für Entschlüsse, manchmal taktlos

TB Team Builder: kooperativ, diplomatisch, kann zuhören; entscheidet ungern

CF Completer Finisher: ehrgeizig, sucht und erkennt schnell Fehler, sehr zuverlässig; delegiert ungern

Durch die Punkteverteilung sehen die Studierenden, wo der Schwerpunkt ihres Typs liegt. Die Ergebnisse durften freiwillig für eine statistische Auswertung abgegeben werden, wobei für jeden Studierenden zwei oder drei maßgebliche Typen als Schwerpunkt herausgerechnet wurden. Der Rücklauf lag zwischen 40% und 90%. Tabelle 1 zeigt die erhaltenen Ergebnisse, im Wintersemester (W) 2009 wurde die Veranstaltung an der Hochschule Osnabrück (OS) getrennt für die Studiengänge Medieninformatik und Technische Informatik angeboten.

Sem./Ort	Summe der Merkmale	Company Worker	Chairman	Shaper	Plant	Resource Investigator	Monitor Evaluator	Team Builder	Completer Finisher
S 09 RM	34	11	2	3	2	1	2	8	5
W 09 OS	28	6	2	5	2	2	4	5	2
W 09 OS	47	12	2	7	1	3	6	9	7
S 10 OS	41	8	3	8	5	3	1	8	5
W 10 OS	65	18	3	13	5	2	6	9	9
Summe	215	55	12	36	15	11	19	39	28

Tabelle 1: Auswertung der Belbin-Befragung

Das Ergebnis zeigt, dass es sich bei Studierenden in der Selbsteinschätzung um typische Mitarbeiter eines Teams (CW) handelt, die ein Team auch zusammenhalten (TB) können. Typische Führungspersonen (CH) sind seltener, Vorantreiber (SH) durchaus vorhanden. Personen mit Tendenz zu Marketing und Vertrieb (RI) sind durchgehend selten.

In Nachbesprechungen wurde herausgearbeitet, dass meist Projekte sehr gut funktionieren, wenn unterschiedliche Typen zusammenkommen, die nach persönlichen Fähigkeiten eingesetzt werden. Weiterhin wurde erkannt, dass das Zusammentreffen extrovertierter, dominanter Typen besonderes Konfliktpotenzial beinhaltet. In Diskussionen wurde die berechtigte Erwartung formuliert, dass mit der Zeit, durch persönliche und Projekterfahrungen, aber auch Soft-Skills-Schulungen Veränderungen der Typen möglich sind.

Die Befragungen werden jeweils am Anfang der Projektarbeiten erstellt und geben den Studierenden die Möglichkeit das erhaltene Ergebnis kritisch zu evaluieren und ggfls. in ersten Runden der Teambildung zu nutzen. Weiterhin wurden im Rahmen der Vorlesung die Chancen für Unternehmen diskutiert, einfacher effiziente Projektteams zu bilden, aber auch die Gefahren, wenn Mitarbeiter einfach in Schachteln mit festen möglichen Karrierewegen gesteckt werden, angesprochen.

Generell zeigt die Tabelle gewisse Tendenzen auf, die sich bei Informatikstudierenden über die Jahrgänge wiederholen. Die Evaluation der Werte kann weitere soziologische Untersuchungen ermöglichen, mit denen man den typischen Durchschnittsstudierenden charakterisieren könnte, bzw. die zumindest zur Entwicklung neuer zu evaluierender Thesen führen können.

Kritische Analyse

An der HS Osnabrück zeigte sich insbesondere bei den klassischen Projektaufgaben zur Entwicklung einer vollständigen Applikation nach dem 3-Schichten-Modell die konstruktive Zusammenarbeit von Studierenden der Technischen und der Medien-Informatik, die ihre jeweiligen Schwerpunkte gewinnbringend in den Rollen im Backend bzw. Frontend einbringen konnten. Die aufgrund der Projektgröße von ca. einem halben Personenjahr notwendige Aufgabenteilung mit der Zuteilung verschiedener Rollen im SWE-Prozess führte für viele Beteiligte zu dem gewünschten Lerneffekt, die Notwendigkeit von Softskills praktisch zu erfahren. Kommunikation, Organisation, Präsentation und Führung wurden eingeübt. Die positiven Evaluationen der Software-Engineering-Projekt-Veranstaltung, wobei die Studierenden besonders positiv bewerten hier erstmals Software-Entwicklung "im Großen" erfahren zu haben, sowie die Rückmeldungen der Firmen, die im anschließenden 6. Semester die Bachelorarbeitsthemen vergeben und bereits "projekterfahrene" Praktikanten bekommen, bestätigen die Ausrichtung der Veranstaltung in Richtung auf handlungsorientiertes Lernen (Gudjons, 2008), (Meyer, 1987).

Ähnliche positive Erfahrungen wurden an der HS RheinMain gemacht, als Besonderheit wurden einige Projekte doppelt durchgeführt, was den Vorteil der Vergleichbarkeit bringt. Neben dem meist positiven Effekt, den eine nicht triviale von Studierenden umgesetzte Software typischerweise hat, kann man so auch Umsetzungsvarianten oder die Auswahl optionaler Features kritischer beurteilen. Da Projekte aber auch den ersten Kontakt von Studierenden mit aktiven Forschungsarbeiten bringen können und Präsentationen ähnlicher Arbeiten Langeweile erzeugen, ist eine Projektvielfalt zu bevorzugen.

Kritisch wurde von Studierenden und Lehrenden angemerkt, dass die eigentliche Zeit zur Umsetzung der Projekte relativ knapp ist. Oftmals wurden größere Anteile der vorlesungsfreien Zeit für Projektaufgaben benötigt. Dies kann gerade am Ende des Wintersemesters Probleme machen, wenn das Sommersemester sich ohne Pause anschließt. Die Veranstaltung macht damit deutlich, dass die Frage nach dem realen Arbeitsaufwand hinter einem KP weiter kritisch diskutiert werden muss.

Im Rahmen der Reakkreditierungen wurden an beiden Hochschulen die Projektveranstaltungen in identischer Form übernommen. Ein konsequenterer Schritt zu weiteren größeren Projekten im Studium zu anderen Themenschwerpunkten in Richtung Projektstudium (Kjaerulff, 2009) wurde nicht gemacht/gewagt, Projekte können aber als optionale Form in Wahlpflichtveranstaltungen durchgeführt werden. Dass in der begleitenden Lehrveranstaltung Inhalte wie Projektplanung vermittelt werden, die eigentlich zum Start des Projekts zum Einsatz kommen, wurde zwar kontrovers diskutiert, aber als kleines Problem gesehen. Ansätze, das Projekt über zwei Semester zu verteilen und im ersten Semester die Vorlesung und nur eine Analysephase zu machen, so dass im zweiten Semester die Umsetzung, ggfls. sogar in anderen Teams erfolgen kann, wurden verworfen.

Zusammenfassung

Die Aufnahme von Software-Engineering-Projekten in Lehrpläne von Bachelor-Informatik-Studiengängen ist und bleibt ein wichtiger Beitrag das „reale

Leben zu simulieren“ und Studierende Erfahrungen in der Teamarbeit sammeln zu lassen.

Um möglichst viele Dozenten zur Veranstaltung von Projekten zu bewegen, ist ein großer Freiheitsgrad notwendig, der durch elementare Rahmenbedingungen eingegrenzt werden muss.

Projekte geben Studierenden auch die Möglichkeit mit aktuellen Forschungsthemen in Kontakt zu kommen und bereiten so das Abschluss-Semester mit der Bachelorarbeit vor. Eine weitere Projektorientierung der Studiengänge in folgenden Reakkreditierungen könnte die Qualität der Studiengänge weiter erhöhen.

Literatur

Belbin, M. (2010), Management Teams, 3. Auflage, Elsevier, Oxford, UK

Claus, V., Reissenberger, W. (1997), Teamarbeit an der Universität - Einbeziehung von Projekten in die Ausbildung, in: Handbuch Hochschullehre, 13. Ergänzungslieferung, Raabe-Verlag

Demuth, B., Hußmann, H., Zschaler, S., Schmitz, L. (1999), Erfahrungen mit einem frameworkbasierten Softwarepraktikum, in: Tagungsband des 6. Workshops Software-Engineering im Unterricht der Hochschulen, Teubner-Verlag

Göhner, P., Bitsch, F., Mubarak, H. (2005), Softwaretechnik live – im Praktikum zur Projekterfahrung, 9. Workshop SEUH 2005

Gudjons, H. (2008), Handlungsorientiert lehren und lernen: Schüleraktivierung, Selbsttätigkeit, Projektarbeit, 7. Auflage, Klinkhardt Verlag

Kjaerulff, U. (2009), Problem-Oriented and Project-Based Learning (POPBL) in Software Engineering. In: Jaeger U., Schneider K. (Hrsg.): Software Engineering im Unterricht der Hochschulen: SEUH 11 - Hannover 2009, Seite 1-2, dpunkt-Verlag

Kleuker, S. und Thiesing, F. (2010), Empfehlungen zur Durchführung eines Projekts im Rahmen der Veranstaltung Software-Engineering-Projekt http://home.edvsz.hs-osnabrueck.de/skleuker/querschnittlich/SWProjektEmpfehlungen.pdf

Kopka, C., Alfert, K. (2002), Der Softwareentwicklungsprozess als Lerngegenstand oder Von einem, der auszog, das reflektierte Handeln zu lehren, in: Sigrid E. Schubert, Bernd Reusch, Norbert Jesse (eds.) Informatik bewegt: Informatik 2002 - 32. Jahrestagung der Gesellschaft für Informatik e.v. (GI), 30. September - 3. Oktober 2002 in Dortmund, Lecture Notes in Informatics, Seiten 401-407, GI, Bonn

Lewerentz, C., Rust, H. (2001), Die Rolle der Reflexion in Softwarepraktika, 7. Workshop SEUH, Zürich 2001

Meyer, H. (1987), Unterrichts-Methoden, Cornelsen Verlag

ShUND (2008), Shared Universal Network Diagram Editor, http://www.shund.de/ (am 1.11.2010 aufgerufen)

Stoyan, R., Glinz, M. (2005). Methoden und Techniken zum Erreichen didaktischer Ziele in Software-Engineering-Praktika. In: K.-P. Löhr, H. Lichter (Hrsg.): Software Engineering im Unterricht der Hochschulen, SEUH-9 2005. Heidelberg, dpunkt-Verlag.