Selbstbestimmung oder Anleitung: Erfahrungen mit einem Softwaretechnikpraktikum im Bereich Qualitätssicherung

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

Zusammenfassung

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

Im Vergleich der beiden Durchführungen ergab sich eine höhere fachliche Erfolgsquote in der zweiten Version, aber auch eine größere Unzufriedenheit mit dem Ergebnis bei den Studierenden. Wichtiger als dies, deckte die zweite Durchführung Schwächen, aber auch Lernfortschritte bei einem unerwarteten und wichtigen Thema auf: Der Arbeitsmethodik der Studierenden.

1 Einführung

Die Softwaretechnikausbildung an der Freien Universität Berlin ist dreistufig gegliedert. Im dritten Semester findet ein Blockpraktikum über drei Wochen statt, in dem unter Anleitung von Tutoren ein kleines Softwareprojekt in Gruppenarbeit realisiert wird. Im fünften Semester folgt „Softwaretechnik“ in Form einer klassischen Pflichtvorlesung mit Übungsblättern und Klausur. Von Master-Studenten der Informatik und Diplom-Studenten kann schließlich das „Praktikum Softwaretechnik“ als Wahlpflichtveranstaltung gewählt werden. Die Veranstaltung hat 8 ECTS-Punkte, was einem Zeitaufwand von einem Tag pro Woche während der Vorlesungszeit entspricht. Etwa zwanzig Studierende nehmen pro Jahr an dem Praktikum teil.

Im Folgenden werden zwei Durchführungen des Praktikums Softwaretechnik vergleichend beschrieben. Das Praktikum behandelte in beiden Fällen nur einen kleinen Teil der Softwaretechnik, nämlich die analytische Qualitätssicherung, d.h. Maßnahmen

zur Einschätzung und Erhöhung der Qualität eines schon erstellten Softwareprodukts oder Vorgängerdokumentes.

Im nachfolgenden Abschnitt werden wir begründen, warum wir diese Ausrichtung auf das Thema Qualitätssicherung gewählt haben und wie wir sie realisierten. Die beiden Durchführungen der Jahre 2004 und 2005 unterschieden sich kaum in den Lernzielen, auch nicht in den fachlichen Inhalten, sondern lediglich in der Vorgehensweise: Während das Praktikum 2004 einen deutlichen Vorlesungscharakter hatte mit praktischen, aufeinander bauenden Übungsaufgaben – wir erläutern dies im dritten Abschnitt –, fehlten Übungsblätter und Anweisungen in 2005 beinahe gänzlich. Stattdessen wurden lediglich die Erfolgskriterien für die ansonsten eigenständig zu planende Arbeit klar definiert. Wir erörtern die Gründe für diese Änderung und deren Folgen im vierten Abschnitt.

Im sechsten Abschnitt versuchen wir eine Gegenüberstellung der beiden Durchführungen hinsichtlich einiger uns wichtigen Kriterien. Dieser Vergleich und insbesondere die Erfahrungen aus dem Jahr 2005 deckten unter anderem Schwächen der Studierenden im Bereich der Arbeitsmethodik, also der Planung der eigenen Arbeit, der Selbstmotivation, der Zielorientierung und der Erfolgs- und Ergebnismessung auf. Diese Schwächen erscheinen uns nicht weniger wichtig als das Beherrschen der Praktiken des Software Engineering im Speziellen. Im siebten Abschnitt schließlich präsentieren wir die Lehren, die wir aus dem Vergleich und unseren Erfahrungen gezogen haben.

2 Softwaretechnikpraktikum mit Schwerpunkt Qualitätssicherung

Ein Softwaretechnikpraktikum sollte aus unserer Sicht folgende Kriterien erfüllen (siehe auch [Prechelt93], [Shaw00]):

1. Theoretisch vermittelte Techniken und Methoden sollen angewendet und die Konsequenzen einer Nichtanwendung im Vergleich zu einem unreflektierten Ad-Hoc-Vorgehen erkannt werden.
2. Das Anwendungsbeispiel sollte eine realistische Größe haben, mit der „schmutzigen Wirklichkeit“ bekannt machen und nicht lediglich akademisches „Spielzeug“ sein.
3. Die Aufgabe soll nicht von einer Person alleine lösbar sein, sondern aufgrund der geringen zur Verfügung stehenden Zeit nur durch eine Gruppe von Teilnehmern.
4. Zeitmanagement, Kommunikation, Konfliktlösung und Entscheidungsfindung sollen geübt werden.

Der geringe Stundenumfang von acht Stunden pro Woche für das Praktikum erlaubt es nicht, ein komplettes Softwareprojekt von der Erhebung der Anforderungen bis zur

Auslieferung der getesteten Version zu realisieren, da hier der Implementierungsanteil andere Lerninhalte verdrängen würde. Da viele Methoden und Verfahren Mittel zur Erreichung und Sicherstellung einer ausreichenden Qualität der Software sind und entsprechende Aspekt der Softwareentwicklung in der Ausbildung oft zu kurz kommen [Lethbridge00, Shepard01], haben wir uns entschieden Themen der Qualitätssicherung anhand existierender Software zu vertiefen.

Mit dieser Entscheidung öffnete sich uns eine weitere Möglichkeit: Wir konnten ein real existierendes Projekt untersuchen, ohne allzu sehr in den Entwicklungsprozess eingebunden werden zu müssen. Unsere Anforderungen waren nun lediglich, Zugriff auf die Fehlerdatenbank und einen relevanten Teil des Quellcodes zu haben. An letzterem scheiterten Versuche kommerzielle Anbieter als Projektpartner zu gewinnen, so dass wir das Praktikum anhand eines Open-Source-Projektes verwirklichten.

Wir entschieden uns für OpenCms[1], einem quelloffenen, datenbankgestütztem Web Content-Management-System auf Java-Servlet/JSP-Basis. Die Software entstand 1999 aus dem Projektgeschäft und wird inzwischen vor allem von hauptamtlichen Programmierern der Firma Alkacon erstellt und erweitert. Sie umfasst 1160 Klassen und 360 kLOCs[2]. Während beider Praktikumsdurchführungen fand ein Major-Release mit den verbundenen Implikationen für die Qualitätssicherung statt.

Probleme hatten wir mit der unzureichenden Dokumentation, sowohl für die Benutzer (z.B. Erläuterung der neuen Funktionalitäten) als auch für die Entwickler (z.B. Entwurfsdokumente). Insbesondere war es oft unklar, was die Software im Detail leisten soll, was also ihre Reaktion auf Benutzereingaben und Konfigurationen hätte sein sollen. Ohne diese Information ist es natürlich schwer möglich, ein Versagen der Software zu erkennen und somit einem Defekt nachzugehen. Nur in einigen Fällen konnten uns die OpenCms-Entwickler oder eine Antwort auf einer Mailingliste aushelfen. Auch waren vermutete Entwurfsprobleme ohne Entwurfsdokumente kaum zu begründen; offensichtliche Anforderungslücken endeten aus dem gleichen Grund stets nur als ein Änderungswunsch und nicht als ein Defekt.

Einige Lerninhalte scheinen uns in dieser Konstellation besonders gut vermittelbar:

- Bedeutung der Dokumentation und Spezifikation zur Kommunikation zwischen Entwicklern und anderen technischen Abteilungen wie der Qualitätssicherung
- Wichtigkeit, zur Beurteilung der Software und ihrer Güte die Anforderungen, die Szenarien und Geschäftsprozesse verstehen zu müssen.
- Betrachtung von (auch „geschenkter“) Software als etwas, was von Kunden eingesetzt wird und was ihnen Probleme bereitete: Rückwärtskompatibilität, Stabilität, Installierbarkeit, Bedienbarkeit, etc.

[1] http://www.opencms.org
[2] CVS vom 16.09.2005

- Erkenntnis, dass analytische Qualitätssicherung in erster Linie Defekte und Schwächen aufdecken soll, statt Verbesserungs- und Erweiterungsvorschläge zu machen.
- Kritische Betrachtung eines *vorhandenen* Entwicklungsprozesses und einer *vorhandenen* Software statt immer nur selbst etwas Neues zu erstellen.

Beide Versionen des Praktikums enthielten die folgenden Inhalte:

- Nach einer Einführung in allgemeine Anforderungen an Content-Management-Systeme galt es zunächst sich in die Software einzuarbeiten. Dazu wurde eine kleine Beispielwebsite erstellt und in beiden Praktika zwei Übungsblätter bearbeitet.
- Selbst auszuwählende Teile der Software wurden anhand verschiedener Qualitätskriterien untersucht (näheres dazu später).
- Wurden funktionale Schwächen entdeckt, so waren sie in die für uns zugängliche Fehlerdatenbank[3] einzutragen. Die Entwickler von OpenCms haben die neuen Einträge daraufhin kommentiert, bewertet und in ihre Planung übernommen oder zurückgewiesen.
- Zum Abschluss des Praktikums wurde von den Studierenden ein Paket an automatisierten Testfällen, Analysen und gemeinsam erstellte Ratschläge an die Entwickler von OpenCms übergeben.

Es wurde in Gruppen zu drei bis vier Mitgliedern gearbeitet, d.h. die Studierenden wurden angehalten, die Aufgaben in ihren Gruppen gemeinsam oder arbeitsteilig zu erledigen und die Ergebnisse auch als Gruppe vorzustellen. Es gab rotierend einen Gruppenleiter, der Ansprechpartner für uns war, um über den Stand der Dinge zu berichten. Jeder Teilnehmer wurde als Gruppenleiter einmal zu einem Einzelgespräch eingeladen, um über die Zusammenarbeit und eventuelle Probleme in der Gruppe zu diskutieren. Wir nutzten dieses Gespräch auch als eine der Grundlagen für die persönliche Note, die für den Teilnahmeschein vergeben wurde.

3 Erste, angeleitete Version

Die erste Durchführung des Praktikums hatte Vorlesungscharakter:

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

[3] http://www.opencms.org/bugzilla/

- In der Praxisstunde wurden die Ergebnisse der vergangenen Übungsblätter vorgestellt und diskutiert.

Nach einer umfangreichen Einführung in die Software gab es drei große Themenblöcke: Unittests für einen kleinen Bereich der Software, Durchsichten bzw. Codeanalyse für einen anderen Teil und Abnahme einer speziellen Funktionalität der Software[4].

Letztlich blieben die Ergebnisse des Praktikums hinter den Erwartungen zurück. Es wurden nur wenige Defekte gefunden, die erstellten Testfälle waren trivial, die Durchsichten schien eher dem Ziel zu folgen, möglichst einfache Perspektiven einzunehmen und bei der funktionalen Abnahme äußerten sich Fehldeutungen, was die Software leisten müsste und in den Abnahmekriterien auftauchen sollte.

Nach einer Analyse der Probleme machten wir für die schwachen Ergebnisse vor allem den Charakter des Praktikums verantwortlich:

- Vordergründiges Ziel der Studierenden schien es gewesen zu sein, das Übungsblatt zwar rechtzeitig, aber ohne viel Aufwand zu lösen. Während die expliziten Anforderungen eines Übungsblattes erfüllt wurden, also z.B. die Erstellung einer Menge von Testfällen, blieben jedoch die Ergebnisse bzgl. der impliziten Zielsetzung, die Qualität der Software durch Aufdecken von Defekten und Versagen zu erhöhen, hinter den Möglichkeiten zurück.
- Es wurde versucht, das Übungsblatt mit den geringsten Mitteln zu lösen. So wurde z.B. fast ausnahmslos die leicht zu prüfenden Kodierungsrichtlinien[5] (coding guidelines) und deren Einhaltung als Durchsichtsperspektive gewählt. Eine Reflektion darüber, wie weit dies dazu dient, Defekte aufzudecken, blieb auch bei den leistungsfähigen Studierenden meist aus.
- Bei der Besprechung schließlich wurde das schwache Ergebnis (im Beispiel: Anzahl und Relevanz der aufgedeckten Defekte) entweder mit der schlecht strukturierten und dokumentierten Software begründet oder mit den ungenauen Vorgaben in den Übungsblättern.

Dies geschah, obwohl wir regelmäßig die Kriterien an ein gutes Ergebnis wiederholten . Das Praktikum war grundsätzlich so motiviert worden, dass wir dem OpenCms-Projekt helfen wollten. Diese Motivation ging jedoch allmählich verloren, und so passierte es, dass die meisten Studierenden in einen passiven Arbeitsstil verfielen und lediglich Anweisungen befolgten.

---

[4] Wir gehen im Folgenden nicht weiter auf die Inhalte ein. Genaueres kann man unter der Veranstaltungswebsite http://projects.mi.fu-berlin.de/w/bin/view/SWTprak/ nachlesen.

[5] Für Java zum Beispiel http://java.sun.com/docs/codeconv/

4 Zweite, selbstbestimmte Version

Aufgrund der gewonnenen Erfahrung versuchten wir, die zweite Durchführung ein Jahr später nach dem zu erreichenden Ergebnis zu orientieren. Die ersten Wochen waren wieder für eine Einführung in das Softwaresystem und der Anwendung von Content-Management-Systemen vorgesehen. Als einzige inhaltliche Änderung wurde die Lehreinheit zur funktionalen Abnahme durch ein Übungsblatt zum Thema Benutzbarkeit ersetzt und nach vorne gestellt. Wir vollzogen den Wechsel, da das Schreiben eines Pflichtenheftes durch die Qualitätssicherungsabteilung eher ungewöhnlich ist. Wir haben dadurch zudem dem in der Lehre wenig beachteten Benutzbarkeitsaspekt mehr Beachtung geschenkt, während die Studenten die Software gleichzeitig durch eine gründliche Evaluation besser kennen lernen konnten.

Die restlichen zehn Wochen bestanden aus zwei Arbeitsaufträgen:

1. Es sollten so viele relevante Versagensfälle der Software provoziert werden wie möglich. Die Relevanz errechnete sich hierbei vor allem aus der Einstufung (severity) des Versagensberichts durch die Entwickler und Projektleiter von OpenCms. Das Versagen sollte so gut wie möglich dokumentiert werden, idealerweise in Form eines Testfalls, was allerdings nicht immer möglich oder angemessen war. Hinweise zur Behebung des zugrunde liegenden Defekts waren nicht gefordert.

2. Jede Gruppe musste wöchentlich auf der Webseite der Veranstaltung einen Bericht hinterlegen, der den beschrittenen Weg, die zugrunde liegende Überlegung und die erzielten Ergebnisse und zukünftige Arbeitsschritte beschreibt. Das Vorgehen musste ob der begrenzten Zeit in erster Linie ökonomisch, aber auch fachlich begründet werden.

Wie man die Versagensfälle findet, blieb den Gruppen überlassen. Da dies einen Zufallsaspekt in die Aufgabe einbringt (z.B. weil man „in der falschen Ecke“ gesucht hat und nichts fand), wurde der schriftlichen Retrospektive in Punkt 2 besondere Aufmerksamkeit geschenkt: Es gab in der großen Runde regelmäßig Rückmeldung und Diskussionen zu angemessenen und gewählten Vorgehensstrategien. Für die Endnote war eine gut begründete Vorgehensweise genauso wichtig wie ihr Erfolg. Die Diskussion ermöglichte es dabei, von der speziellen Situation in diesem Praktikum zu abstrahieren und allgemein anwendbare Vorgehensweisen und Heuristiken vorzustellen.

Ein Angebot von uns war es, wöchentlich einen Vortrag über ein Spezialthema der Softwaretechnik oder Qualitätssicherung – sowohl technischer, methodischer als auch empirischer Art – zu halten. Die Anfrage für einen solchen Vortrag musste allerdings von den Studierenden selbst kommen. Da wir anfangs noch Orientierungsprobleme vermuteten, waren zwei Vorträge zu den Themen funktionales Testen und Durchsichten fest eingeplant. Darüber hinaus wurden wir leider nur zweimal gebeten, ein Thema zu referieren: Über Techniken der Inspektion und über die JUnit-Einbindung in OpenCms.

Eine weitere Änderung gegenüber der ersten Durchführung ergab sich automatisch: Da nicht mehr wir Veranstalter die Arbeit verteilten, standen die Gruppen im Wettbewerb zueinander, denn natürlich ist ein gefundenes Softwareversagen nur gewertet worden, wenn die Gruppe es als *neuen* (nicht duplizierten) Eintrag in der Fehlerdatenbank veröffentlicht. Zudem wussten wir Veranstalter nicht, wie defekt die Software überhaupt ist, so dass der Durchschnitt der Gruppenleistung das Richtmaß definierte. Wir erwarteten durch den Wettbewerb einen weiteren Schub, ergebnisorientiert vorzugehen. Durch die Veröffentlichung der Vorgehensweisen der einzelnen Gruppen ergab sich die Möglichkeit einer Kooperation und Koordination der Gruppen.

Der Wettbewerbscharakter schien allerdings nur wenige Studierende zu beflügeln. Im Großen und Ganzen gab es eine Abwehrhaltung dagegen. Eine größere Kooperation zwischen den Gruppen entstand nur in Form von nicht verbindlichen Aufgabenverteilungen, denen aber kaum koordinierende Aktivitäten folgten. Dies verhinderten vermutlich der damit verbundene Zeitaufwand und ein fehlender Gesamtprojektleiter.

Alle Gruppen haben zunächst versucht, ein Modul oder eine Funktionalität der Software zu finden, die vermutlich „buggy“ ist. Es gab dafür sehr viele kreative Heuristiken. So wurden die Versionen und ihre Änderungen detailliert betrachtet, die Software mit freien Werkzeugen[6] statisch analysiert, die Entwickler-Maillingliste durchforstet u.v.m.. Nach dieser Phase wurde meist in zwei oder drei Modulen näher gesucht. Dabei wurde nach ersten funktionalen Testversuchen zum Schluss fast ausschließlich der Quellcode direkt nach Defekten durchsucht. Dies lag vor allem an dem Aufwand, systematisch Testfälle zu entwickeln und anzuwenden. Durch die umfangreichen Durchsichten mutierte das Praktikum allerdings fast zu einem „Java für Fortgeschrittene“.

5 Vergleich der Versionen

Zum Vergleich interessant sind

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

[6] Eingesetzt wurden z.B. http://metrics.sourceforge.net/, http://emma.sourceforge.net/ und http://findbugs.sourceforge.net/

Version die Stimmen stark positiv und negativ aus. Wir sahen also sowohl mehr Spitzenleistung als auch mehr demotivierte Studenten in der zweiten Version.

6 Inhalte, Ergebnisse und Leistung

Unsere Enttäuschung hinsichtlich der Leistungen und Ergebnisse nach der ersten Durchführung hat sich bei der zweiten Version nicht wiederholt. Die Studierenden haben sowohl im Umfang (Anzahl gefundener Versagensfälle und Menge der zumindest durchdachten Techniken) als auch in der Güte der Ergebnisse (Relevanz der Versagensfälle gemessen an der vergebenen „severity“ im Defektverfolgungssystem und deren Dokumentation, z.B. in Form von automatisierten Testfällen) eine bessere Leistung gezeigt. In einigen Gruppen haben allerdings vermutlich nur wenige Mitglieder die meiste Arbeit geleistet, und es gab deutliche Unterschiede zwischen den Gruppen. Die Kommunikation innerhalb einer Gruppe und die Bereitschaft, Initiative zu übernehmen waren wesentliche Erfolgsfaktoren, denn während die erste Durchführung individuelle Arbeit wegen vorgeschlagener Arbeitsaufteilung leicht ermöglichte, waren gute Ergebnisse beim zweiten Mal nur durch funktionierende Koordination innerhalb der Gruppe möglich.

Die Lernergebnisse beider Durchführungen unterschieden sich deutlich, obwohl vordergründig die gleichen Lerninhalte vorlagen. Die Quelltextdurchsicht und die damit verbundenen technischen Fragen haben im zweiten Fall z.B. einen deutlich größeren Umfang eingenommen. Dadurch fiel aber das systematische Erstellen von Testfällen eher kurz aus, denn die Studierenden haben dieser Technik nicht zugetraut, innerhalb der kurzen Zeit ein angemessenes Ergebnis zu erreichen.

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

6.1 Softwaretechniktheorie in der Praxis

Durch den direkten Bezug zur Praxis der Softwareentwicklung und Qualitätssicherung sind uns einige Lücken im Kanon des Software Engineering aufgefallen:

- Es fehlt Hilfestellung im Bereich des Programmverstehens. Die meisten Gruppen setzten Debugger zur schrittweisen Durchdringung der Softwarefunktionalität ein und konnten nicht auf ein strukturiertes Vorgehen zum Verständnis von fremder Software zurückgreifen.
- Eine bessere empirische Grundlage zum Nutzen von Metriken für Fehlervorhersage oder der Mächtigkeit statischer Codeprüfungen ist wünschenswert, da diese Multiplikationsfaktoren für die erfolgreiche Qualitätssicherung sind. So gelang es den Studierenden leider kaum, durch

den Einsatz von typischen Metriken oder statischer Codeprüfung den untersuchenswerten Bereich der Software einzuschränken.
- Wir vermissen konkrete Hinweise für die Durchführung von Durchsichten und Inspektionen. Vor allem die Verteiltheit von Belangen und unzureichende Spezifikation erschwerte es, die Vorlagen aus den konsultierten Lehrbüchern der Softwaretechnik [Balzert98, Sommerville04] einzusetzen. Um eine einzelne Zeile zu begutachten, mussten oftmals eine Stunde lang Aufrufhierachien und Klassenbeziehungen nachverfolgt werden, was selten zu Erfolg führte. Darüberhinaus scheinen Checklisten-basierte Durchsichten durch die Evolution von C++ zu Java einen Großteil ihrer Wichtigkeit verloren zu haben, da in Java eine Vielzahl der „gemeinen" Fallgruben aus C++ nicht mehr existieren.
- Da der ökonomische Aspekt im Sinne von Zeit und Arbeitseinsatz eine wichtige Rolle in diesem Praktikum spielte, fiel auch das Fehlen diesbezüglicher Informationen aus der Literatur sofort ins Auge. Wie lassen sich qualitätssichernde Maßnahmen gegeneinander im Bezug auf Qualitätsverbesserung pro Kosten bewerten?
- Zuletzt hat sich der Einsatz von Werkzeugen aus Theorie und Praxis als problematisch gezeigt. Die von den Studenten verwendeten Werkzeuge (z.B. Hammurabi[7], Pmd[8] u.a.) scheiterten ausnahmslos an der Größe des Quellcodes und stürzten meist ohne Ergebnisse ab.

6.2 Mangel in der Arbeitsmethodik

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

Viele Studierende gingen wenig souverän mit diesen Bedingungen um. Folgende Verhaltensweisen waren häufig zu beobachten:

- Langes theoretisches Nachdenken über den einzuschlagenden Weg, obwohl a-priori-Urteile kaum möglich waren.
- Langes Festhalten an vermeintlich wichtigen Vorarbeiten (z.B. Installieren noch eines Sourcecode-Analyse-Werkzeugs), obwohl allmählich der Endtermin nahe rückte.
- Fehlendes Durchhaltevermögen, einem gut begründeten Weg über längere Zeit zu folgen, auch wenn dieser nicht sofort Erfolg bewies.
- Fehlende Akzeptanz der Komplexität der Aufgabe und Software und daraus resultierende geringe Bereitschaft und Phantasie, sich aktiv um ein tieferes

[7] http://wiki.hammurapi.biz/index.php?title=Hammurapi
[8] http://pmd.sourceforge.net/

Verständnis zu kümmern. Bis zuletzt hielt sich z.B. bei einigen Studenten der Glaube, durch die Wahl des richtigen Werkzeuges die Aufgabe ohne große Arbeitsinvestition gelöst zu bekommen.

- Unnötig skeptische Beurteilung eigener Überlegungen und Überzeugungen mit übersteigertem Glauben, Erfolg sei nur Glücksache.
- Wenig Risikobereitschaft, noch nicht (von anderen Gruppen) gegangene Wege zu nehmen. Evtl. herrschte die Befürchtung, dass auch erfolglose (aber gut begründete) Wege nicht honoriert werden.
- Fehlendes Gespür für zeitliche Entwicklungen, z.B. dem Bedenken, wie viel Zeit noch bleibt. Auch fehlende zeitliche Selbstorganisation, z.B. regelmäßige wöchentliche Termine einzurichten.

Die natürlicherweise fehlende Urteilsfähigkeit, wann welche softwaretechnische Theorie in der Praxis einsetzbar ist und wann nicht, fällt dagegen zurück und war weniger bedeutend, als wir zunächst vermutet hatten.

Darüber hinaus offenbarte sich die Gruppenarbeit als schwierig. Ein verbreiteter Irrtum war, dass Gruppenarbeit gemeinsame gleichzeitige Arbeit an einem Problem bedeutet. Die Aufgaben in diesem Praktikum waren aber so angelegt, dass die Kunst der effizienten Gruppenarbeit darin bestand, die Arbeit möglichst gut aufzuteilen, Termine und Verantwortlichkeiten einzuhalten, doppelte Arbeit zu verhindern und nur gelegentlich Zweier- oder Dreierarbeit einzulegen, wenn das Problem komplex oder schwierig genug ist. So gelang es einer Gruppe, die Defektsuche so ungeschickt in zeitlich hintereinander liegende Verantwortungen mit unterschiedlichen Bearbeitern zu zerlegen, dass mehrere originelle und neue Defekte erst Tage später für einen Eintrag in die Fehlerdatenbank aufbereitet waren. Andere Gruppen und externe Nutzer der Software waren zu diesem Zeitpunkt jedoch bereits tätig gewesen.

Rückblickend erscheint zunehmend, dass diesen *arbeitsmethodischen* Fähigkeiten in der universitären Ausbildung, insbesondere bei so Ingenieurs-nahen Fächern wie der Softwaretechnik, deutlich breiteren Raum gegeben werden muss.

7 Schlussfolgerungen

Wir haben ein Softwaretechnik-Praktikum, bei dem anhand eines Open-Source-Projektes qualitätssichernde Maßnahmen eingeübt wurden, in zweierlei Durchführungen vorgestellt: einer angeleiteten und einer selbst gesteuerten Variante. Den Nachteilen der verschulten ersten Version wurde unter einigen Opfern in der zweiten Version begegnet.

Die ungewohnte Unklarheit in der Aufgabenstellung, der Wettbewerbscharakter, die geringe Hilfe, die wir Veranstalter geben wollten, die teilweise schwierige Zusammenarbeit innerhalb der Gruppen, das Gefühl, Glück sei der entscheidende Erfolgsfaktor und schließlich das subjektive Empfinden, dass das erzielte Ergebnis in keinem guten Verhältnis zum eingesetzten Aufwand lag – all dies ist vermutlich Grund

für die geringere Zufriedenheit der Studierenden mit dem Praktikum in seiner zweiten, selbst gesteuerten Form. Dies war erkennbar an der nachfolgenden, fachbereichsweiten Evaluation der Lehrveranstaltung.

Wir Veranstalter sind hingegen zufriedener mir dem (auch objektiv besseren) Ergebnis, denn wir sind uns sicher, in dieser freien Form praxisrelevantere Lernziele erreicht zu haben als durch die Orientierung an Übungsblätter. Abschließend wollen wir diskutieren, wie wir versuchen werden, die verbleibenden Schwachstellen des Praktikums zu beseitigen:

Uns erscheint es notwendig eine Balance zwischen dem bequemen Wochentakt der angeleiteten Version und dem elfwöchigen Auf-sich-gestellt-sein des zweiten Durchlaufs zu erreichen. Wir wollen Zwischenstopps einführen, bei denen man Teilziele zu erreichen hat, die für sich einen Wert haben. Wir hoffen die Wettbewerbssituation zu entschärfen, indem wir jeder Gruppe eine oder zwei eigene, mit den anderen aber koordinierte Aufgabe zuteilen. Diese Meilensteine werden idealerweise gemeinsam erarbeitet, aber letztlich von den Veranstaltern vergeben und kontrolliert. Kandidaten hierfür sind:

- Einsatz und Evaluation von Werkzeugen zur Ermittlung von gängigen Softwaremaßen und deren Einsatzmöglichkeit im Praktikum
- Einsatz von Werkzeugen zur statischen Codeanalyse
- Durchforstung der Versionsdaten zur Ermittlung der Änderungshäufigkeit von Modulen
- Ermittlung typischer Fehlermuster aus den Fehlerverfolgungsdaten
- Re-Engineering des Entwurfs und der Modularisierung, die nicht ausreichend dokumentiert sind
- Ermittlung und Beurteilung der Testabdeckung aufgrund der schon implementierten Regressionstests

Am Ende eines solchen zeitlich klar begrenzten Blocks bleibt dann die Zusammenfassung der einzelnen Ergebnisse, die jede Gruppe für ihre eigene Weiterarbeit nach belieben verwenden kann. Diese Arbeitseinheiten sind weiterhin Mittel zum Zweck, nämlich der Aufdeckung von Schwächen der Software. Wir wollen sowohl den freien Charakter (dass die Gruppen sich für einen eigenen Weg entscheiden und diese Entscheidung begründen müssen) als auch den Wettbewerbscharakter (dass alle Gruppen die gleiche Ausgangsbasis und das gleiche Ziel haben) beibehalten, d.h. das Praktikum liefe danach wie in der zweiten Version. Wenn notwendig und sinnvoll, kann es noch ein zweites Arbeitspaket pro Gruppe geben – mehr gibt aber der zeitliche Rahmen von 14 Wochen nicht her.

Eine alternative Idee für die Verbesserung entlehnt sich der Idee der Story Cards [Beck05]. Die Studenten sollen in diesem Ansatz vor dem Beginn einer Arbeitseinheit eine Aufgabenbeschreibung mit erwarteten Kosten und Nutzen an die Veranstalter richten. Diese bewerten die Karte dann mit einer Punktzahl, die bei Erreichen des Ziels an die Gruppe verteilt wird und steuern so den Fortgang des Praktikums indirekt. Es

entsteht durch diesen Ansatz ein besserer Überblick für Teilnehmer und Veranstalter über erreichte und nicht erreichte Ziele. Auch sollte es so möglich sein, den Übungsblattcharakter der angeleiteten Version zu minimieren, da die Aufgaben maßgeblich von den Studenten selbst bestimmt werden.

Wir werden zudem die Gruppen und ihr inneres Funktionieren genauer beobachten und in Zweifelsfällen deeskalierend eingreifen müssen. Dass sich die Gruppenarbeit meist zu einer Einzelarbeit, vielleicht Paararbeit entwickelt, erscheint uns hingegen eine im gesetzten Rahmen sinnvolle, ja sogar notwendige Organisation zu sein.

Letztlich werden wir auch an der ursprünglichen Motivation der Studierenden arbeiten müssen: Die Lust daran, eine Software so „auseinander zu nehmen“, dass ihr Versagen klar zum Vorschein tritt. Wir glauben, dass eine solche Mentalität – richtig ausgelebt – einige Härten und Orientierungslosigkeit vermeidet, die eine Orientierung an fremde Techniken und Methoden eher fördert.

Dank

Wir möchten uns herzlich bei der Firma Alkacon bedanken, ohne deren Hilfe das Softwarepraktikum nicht möglich gewesen wäre.

Literaturverzeichnis

[Balzert98] Helmut Balzert: "Lehrbuch der Software-Technik II: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung", Spektrum Akademischer Verlag Heidelberg, 1998.

[Beck05] Kent Beck: Extreme Programming Explained, Embrace Change – Second Edition (2005)

[Lethbridge00] Timothy Lethbridge: What Knowledge Is Important to a Software Professional?, IEEE Computer, May 2000

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

[Shaw00] M. Shaw. Software engineering education: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (Limerick, Ireland, June 04 - 11, 2000). ICSE '00. ACM Press, New York, NY, pp. 371-380

[Sommerville04] Ian Sommerville: "Software Engineering", 7th edition, Pearson Education, 2004

[Shepard01] T. Shepard, M. Lamb, D. Kelly: More testing should be taught. Comm. ACM, Vol. 44, No. 6. June 2001, pp. 103-108.