Ulrike Jaeger, Kurt Schneider (Hrsg.) (2009): Software Engineering im Unterricht der Hochschulen,
SEUH 11, Hannover, dpunkt.verlag, Heidelberg

Irrwege und Wegweiser im Praktikum der frühen Softwareentwicklungsphasen

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

Zusammenfassung

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

1 Einleitung

1.1 Didaktisches Ziel

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

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

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

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

1.2 Lehrveranstaltung

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

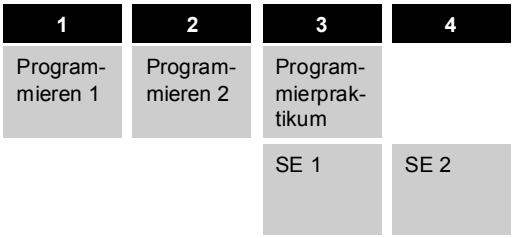

Abb. 1 *Ausschnitt aus dem Curriculum (1.-4. Semester, 5 ECTS-Punkte je Modul)*

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

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

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

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

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

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

1.3 Überblick über diesen Beitrag

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

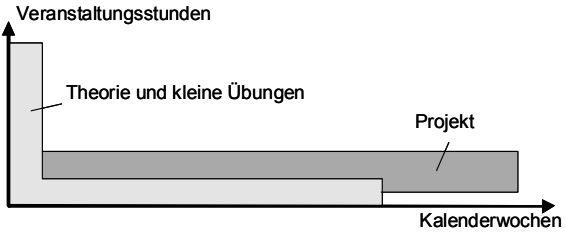

Abb. 2 *L-förmige Verzahnung von Theorie und Praktikum*

2 L-förmige Verzahnung von Theorie und Praxis

2.1 Das Problem

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

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

2.2 Lösungsvorschlag

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

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

2.3 Erfahrungen

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

1 Eine entsprechende Anpassung des offiziellen Curriculums ist geplant.

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

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

3 Softwareentwicklung als Arbeit im Steinbruch

3.1 Das Problem

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

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

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

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

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

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

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

3.2 Lösungsvorschlag

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

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

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

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

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

Verwandter Ansatz

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

3.3 Erfahrungen

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

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

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

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

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

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

4 Code first

4.1 Das Problem

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

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

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

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

4.2 Lösungsvorschlag

»Code first« im Kleinen

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

»Model first« in der Breite

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

Zeitnahes Feedback

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

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

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

Verwandter Ansatz

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

4.3 Erfahrungen

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

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

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

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

Ulrike Jaeger, Kurt Schneider (Hrsg.) (2009): Software Engineering im Unterricht der Hochschulen,
SEUH 11, Hannover, dpunkt.verlag, Heidelberg

5 Brücke von Lernenden zu Praktikern

5.1 Das Problem

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

5.2 Lösungsvorschlag

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

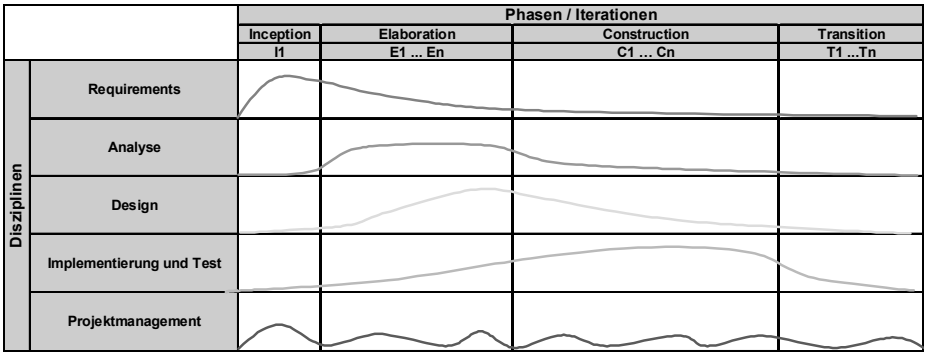

Abb. 3 *Zuschnitt des Unified Process für die Praxis. Die Kurven deuten den Arbeitsaufwand je Disziplin an.*

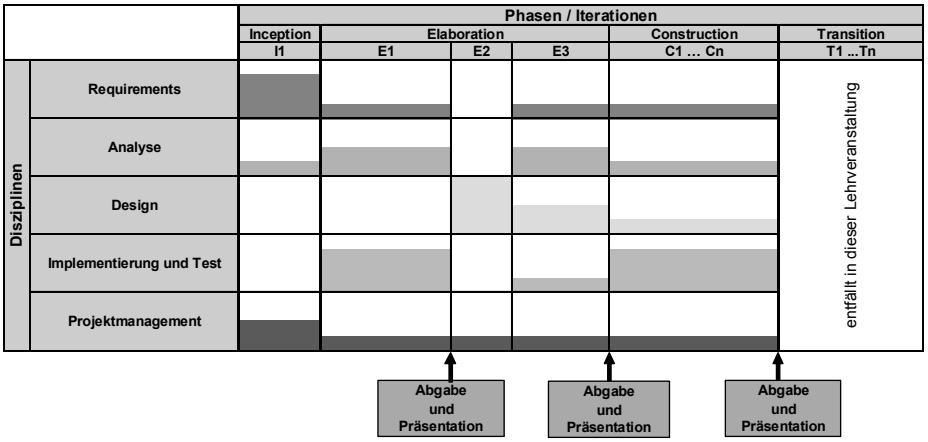

Abb. 4 *Zuschnitt des UP für die Lehrveranstaltung. Die Höhe der Balken deutet den Arbeitsaufwand je Disziplin an.*

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

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

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

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

5.3 Erfahrungen

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

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

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

6 Diskussion

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

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

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

Danksagung

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

Literatur

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

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

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

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

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

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

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

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