Auf dem Weg zu einer robusten Programmierausbildung

Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler[1], Marcus Crestani, Herbert Klaeren, Eric Knauel[2] und Michael Sperber

Albert-Ludwigs-Universität Freiburg
{bieniusa,degen,heidegger,thiemann,wehr}@informatik.uni-freiburg.de
Zühlke Engineering AG, martin.gasbichler@zuehlke.com
Eberhard-Karls-Universität Tübingen,
{crestani,klaeren}g@informatik.uni-tuebingen.de
T-Systems Enterprise Services GmbH, eric.knauel@t-systems.com
DEINPROGRAMM, sperber@deinprogramm.de

Zusammenfassung: Die gelungene Durchführung einer Vorlesung „Informatik I – Einführung in die Programmierung“ ist schwierig, trotz einer Vielfalt existierender Materialien und erprobter didaktischer Methoden. Gerade aufgrund dieser vielfältigen Auswahl hat sich bisher noch kein robustes Konzept durchgesetzt, das unabhängig von den Durchführenden eine hohe Erfolgsquote garantiert. An den Universitäten Tübingen und Freiburg wurde die Informatik I aus den gleichen Lehrmaterialien und unter ähnlichen Bedingungen durchgeführt, um das verwendete Konzept auf Robustheit zu überprüfen. Die Grundlage der Vorlesung bildet ein systematischer Ansatz zum Erlernen des Programmierens, der von der PLT-Gruppe in USA entwickelt worden ist. Hinzu kommen neue Ansätze zur Betreuung, insbesondere das Betreute Programmieren, bei dem die Studierenden eine solide Basis für ihre Programmierfähigkeiten entwickeln. Der vorliegende Bericht beschreibt hierbei gesammelte Erfahrungen, erläutert die Entwicklung der Unterrichtsmethodik und der Inhaltsauswahl im Vergleich zu vorangegangenen Vorlesungen und präsentiert Daten zum Erfolg der Vorlesung.

In seinem Beitrag zur HDI 2006 beanstandet Volker Claus [3], dass Erkenntnisse über Lehr- und Lernmethoden der Informatik in zu geringem Maße gesammelt und veröffentlicht werden. So fordert er, dass Lehr- und Lernmethoden *robust* sein sollen. Darunter versteht er, dass der Erfolg der Methoden von der Person des Dozenten nachweisbar unabhängig ist. Diese Unabhängigkeit ist wichtig, weil besonders bei den Einführungsvorlesungen oft viele unterschiedliche Dozenten zum Einsatz kommen. Dennoch soll aber ein gleichbleibend hohes Ausbildungsniveau garantiert werden. Die Erforschung der Prinzipien und Methoden für diesen gewünschten Dauerzustand erfordert „flächendekkende Untersuchungen und Auswertungen auf der Grundlage von vergleichbaren Messgrößen“ [3].

Die Entwicklung und Durchführung einer Grundvorlesung ist ein aufwendiger (und damit auch teurer) Prozess, der trotzdem häufig enttäuschende Ergebnisse produziert. Deshalb tut jeder verantwortliche Dozent gut daran, die eigene Veranstaltung kritisch zu beobachten, vor allen Dingen aber von den Beobachtungen und Erfahrungen anderer zu

[1] teilweise unterstützt durch die Universität Tübingen
[2] unterstützt durch die Universität Tübingen

profitieren. Trotz Volker Claus' Aufsatz sind viele Grundvorlesungen aber immer noch „Inseln", die nur unzureichend gesicherte didaktische Erkenntnisse einbeziehen. Leidtragend sind dabei in erster Linie die Studierenden, die letztlich gehalten sind, sich Stoff und geforderte Kompetenzen selbst anzueignen, anstatt von der Grundvorlesung selbst aktive Unterstützung zu erhalten.

Dieser Aufsatz beschreibt Gestaltung und Prozess einer Reihe von Grundvorlesungen in Informatik an den Universitäten Tübingen und Freiburg. Diese Reihe begann 1999 mit dem Ziel, die Lernerfolge in der Grundausbildung gegenüber vorangegangenen Jahrgängen entscheidend zu verbessern. Die Tübinger und Freiburger Durchführenden an den Lehrstühlen Klaeren respektive Thiemann taten sich ursprünglich zusammen, um ein gemeinsames Lehrbuch zu nutzen; inzwischen findet auch ein weitreichender Abgleich bei der Durchführung statt. Aus dieser Kooperation ist ein robustes Konzept für eine Anfängerausbildung im Programmieren und in einigen weiteren ausgewählten Grundlagenthemen der Informatik entstanden. Das Konzept ist inzwischen von fünf Dozenten (Klaeren, Sperber, Gasbichler, Schilling in Tübingen, Thiemann in Freiburg) in insgesamt sieben Durchgängen erfolgreich übernommen und weiterentwickelt worden; zwei weitere Durchgänge stehen im Wintersemester 2008/2009 an. Kontinuierliche Beobachtung und Messung des Lernerfolgs der Veranstaltungen belegt und sichert die erzielten Erfolge, von denen wir die wichtigsten in diesem Aufsatz zusammenfassen.

Bei der Verbesserung des Lernerfolgs spielt die inhaltliche Gestaltung die zentrale Rolle: die konsequente Unterrichtung in systematischer Programmierung und die Verwendung einer funktionalen Programmiersprache mit hohem Abstraktionsgrad liefern die Grundlage; verschiedene organisatorische Maßnahmen sichern den dadurch ermöglichten Lernerfolg. Aus diesem Grund dokumentiert dieser Aufsatz nicht nur die von Claus angefragten organisatorischen Rahmenbedingungen (Abschnitt 1), sondern auch die inhaltliche Entwicklung, die unmittelbar dabei gemachten Beobachtungen und die daraus gezogenen Konsequenzen (Abschnitt 2). In Abschnitt 3 werden einige Messergebnisse zum Erfolg unserer Methoden präsentiert; Abschnitt 4 ist die Zusammenfassung.

1 Klassifizierung der Veranstaltung

In Anlehnung an die Kriterien von Claus kategorisieren wir im Folgenden unsere Veranstaltungen:
Umfang 4V + 2Ü + 2BP (Betreutes Programmieren, s.u.)
Ausrichtung funktionales Paradigma
Abstraktion zumeist Formalisierungen und Abstraktion anhand von Beispielimplementierungen
Programmieranteil 60%-70% in der Vorlesung
Material Foliensätze, Tafelanschriebe, Quellcodedateien, Lehrbuch „Die Macht der Abstraktion" [13] („DMdA"), Vorlesungsaufzeichnungen (Freiburg)
Verzahnung von Vorlesung und Übungen(en) Vertiefung der Vorlesung
Intension der Übungen Aufarbeitung der formal-technischen Inhalte (80%) und Programmieraufgaben (20%) in den Übungen, Intensivierung der Programmierfähigkeiten im BP
Größe der Gruppen 10-12 Studierende pro Übung, max. 22 Studierende pro BP

Betreuung der Übungen ein studentischer Tutor pro Übung, 2 Tutoren und ein Mitarbeiter pro BP
Durchführung der Übung Vorstellung der Lösung durch Studierende und Tutoren in der Übung, individuelle Bearbeitung der Aufgaben mit Unterstützung durch die Tutoren im BP
Abgabemodus wöchentliche Abgabe der Übungsaufgaben in Teams, individuelle Abgabe nach Ende der Bearbeitungszeit im BP
Bereitstellen von Lösungen Musterlösungen der Übungsaufgaben online verfügbar, keine Musterlösung für BP
Testklausuren Tübingen: zwei Testatübungsblätter, nach ca. 1/3 und ca. 2/3 des Semesters; Freiburg: eine Übungsklausur in der Mitte des Semesters
Scheinkriterien 50% bzw. 60% der Punkte in den Übungen und BPs jeweils vor und nach Weihnachten, Tübingen: Bestehen der Testate, Freiburg: Bestehen der Übungs- und der Abschlussklausur

Erläuterungen und Ergänzungen zur Klassifizierung

Inhalt der Vorlesung Im Mittelpunkt der Veranstaltung stehen die Konzepte des systematischen Programmierens und der Abstraktion. In der Vorlesung werden aus Beispielen Mittel zur Problemlösung abstrahiert, und konkrete Probleme mit diesen Mitteln gelöst. Dabei werden bei Bedarf Verfeinerungen und Vertiefungen der bekannten Techniken eingeführt. In vielen Sitzungen wird Code „on-the-fly“ entsprechend der unterrichteten Methodik entwickelt. Viele der eingeführten Programmiertechniken werden in Form von Konstruktionsanleitungen explizit verbalisiert und schriftlich fixiert. (Mehr dazu im nächsten Abschnitt.) Eine detailliertere Diskussion der Vorlesungsinhalte bietet Bieniusa et al. [2].

Sprachen Die Veranstaltung verwendet eine Abfolge von Sprachebenen, die eingeschränkte Dialekte von Scheme sind. Der Sprachumfang ist dabei immer an den aktuellen Kenntnisstand der Studierenden (bzw. den aktuellen Stand der Vorlesung) angepasst. Die Einschränkungen verhindern eine Reihe von Fehlern, die durch (beabsichtigte oder unbeabsichtigte) Verwendung von Sprach-Features entstehen können, die noch nicht in der Vorlesung behandelt wurden und zwingen die Studierenden, das Material der Vorlesung anzuwenden.

Aufzeichnungen Die Freiburger Aufzeichnungen enthalten die Folien inklusive interaktive Hervorhebungen und Änderungen durch den Dozenten, Fotos des Tafelaufschriebs sowie den Vortrag des Dozenten. Die Tübinger Vorlesung stellte die Folien des Dozenten zur Verfügung, und wird ab dem WS 2008/09 Videoaufzeichnungen anfertigen und ins Netz stellen.

Betreuung Jede Übungsgruppe wird durch jeweils einen studentischen Tutor betreut. Um individuelle Hilfestellung geben zu können, wird jedes Betreute Programmieren durch mindestens zwei studentische Tutoren betreut, oft ist noch ein Assistent dabei. Die Lösungen der wöchentlichen Übungsaufgaben, der Testataufgaben und der Aufgaben des BP korrigiert der Übungsgruppen-Tutor.

Durchführung der Übungen In den Übungsgruppenwiederholen die Tutoren im Frontalunterricht (Tafel, Folien, Beamer) besonders den Teil des Vorlesungsstoffs, der für die aktuellen Übungsaufgaben relevant ist und beantworten Fragen; außerdem sind Studierende gehalten, ausgewählte Aufgaben vorzurechnen. Die Aufgabenblätter können die Studierenden in Zweierteams bearbeiten. Zwei ausgewählte Aufgabenblätter sind Testat-Aufgabenblätter, die die Studierenden jeweils allein bearbeiten und die Lösungen im Einzelgespräch ihrem Tutor vorstellen müssen.
Im Betreuten Programmieren werden Übungsaufgaben unter Aufsicht bearbeitet: Betreuer stehen für Fragen und Hilfestellungen zur Verfügung. Außerdem gehen sie aktiv auf die Studierenden zu, schauen über die Schulter und geben Lösungstipps.

Art der Aufgaben Die Bandbreite der praktischen Aufgaben reicht vom Durchführen einer Konstruktionsanleitung mit minimalem Einfügen von Operatoren über das selbständige Erkennen und Definieren von Hilfsoperationen bis hin zu Aufgaben, bei denen Teile der Spezifikation sinnvoll ergänzt und ausgearbeitet werden müssen. Die Aufgaben des Betreuten Programmierens sind ähnlich, allerdings mit eingeschränktem Umfang, damit sie in der zur Verfügung stehenden Zeit bearbeitet werden können. Entscheidend ist jeweils, dass die Aufgaben mit den Techniken des systematischen Programmierens aus der Vorlesung zu lösen sind. Dies wird durch konsequente Entwicklung von Musterlösungen vorab sichergestellt. Entsprechend wird auch bewertet, in welchem Maß sich die Studierenden an die Konstruktionsanleitungen gehalten haben.
Die theoretischen Aufgaben umfassen das Nachrechnen von Beispielen, das Abprüfen von Definitionen, das Anwenden von Ergebnissen auf bekannte Situationen, sowie das Übertragen von Ergebnissen auf neue Aufgabenstellungen.

Schulung der Tutoren Die Tutoren werden nach Erfahrung, Programmierkompetenz und ihrer Bereitschaft, sich auf etwas Neues einzulassen, ausgesucht. Kenntnisse in Scheme sowie einen Einblick in die Konzeption der Vorlesung erhalten sie durch eine mehrtägige Einführung. Hier werden die wesentlichen Punkte der Vorlesung und der Übungen vorab präsentiert, diskutiert und entschieden. In weiteren Sitzungen sowie in der wöchentlichen Tutorenbesprechung erhalten die Tutoren weitere Anweisungen auf Gebieten, wo ein einheitliches Vorgehen wichtig ist: Korrekturrichtlinien für die Aufgaben, Verhaltensregeln im Betreuten Programmieren, Verhalten bei Täuschung usw.

Räume und Ausstattung Die Vorlesungen finden in großen Hörsälen mit Multimedia-Ausstattung statt, die Tutorate in kleineren Seminarräumen. Das Betreute Programmieren wird in einem Poolraum durchgeführt, der mit 22 Rechnern für die Studierenden und einem Rechner für die Betreuer ausgestattet ist. Auf den Poolrechnern wird eine Umgebung installiert, in der Studierenden lediglich die Entwicklungsumgebung verwenden, auf Vorlesungsmaterialien zugreifen und die Lösungen der Aufgaben einreichen können.

Werkzeugunterstützung Die Programmieraufgaben werden in der Entwicklungsumgebung DrScheme [4] bearbeitet, die speziell auf die Anfängerausbildung ausgerichtet ist und die daher die bereits erwähnten unterschiedlichen Sprachebenen unterstützt. Die Korrektur der Aufgaben wird durch ein halbautomatisches Werkzeug erleichtert, das die Abgaben auch auf Plagiate untersucht.

Plagiat Im Falle eines Plagiats werden alle beteiligten Studierenden durch Bewertung des Blattes mit Null Punkten bestraft. Im Wiederholungsfall wird die Zulassung zur Klausur nicht erteilt, die Vorlesung ist nicht bestanden.

2 Entwicklung und Verbesserung

> „Wir sind froh, dass die Absolventen schon Java können. Programmieren müssen wir denen halt noch beibringen."

Dieses Zitat eines mittelständischen IT-Unternehmers im Schwarzwald ist typisch für die Erfahrungen vieler: Die Erwartungen des Arbeitsmarkts an Hochschulabsolventen im Fach Informatik sind vielfältig; dennoch traut kaum ein erfahrener Arbeitgeber den Absolventen eine ausgebildete Programmierkompetenz zu.

Bereits an der Hochschule beobachten viele Dozenten, dass viele Studierende in Veranstaltungen des Hauptstudiums, die Programmierfähigkeiten erfordern, große Schwierigkeiten selbst bei einfachen Aufgaben haben. Eigentlich ist dies nicht überraschend: Die Ausbildung in vielen komplexen Tätigkeiten benötigt ca. zehn Jahre – die Erwartung, dass eine vollständige Ausbildung zum Software-Entwickler in wenigen Semestern stattfinden kann, ist unrealistisch [14]. Hinzu kommt, dass die formale Pflicht- Programmierausbildung in vielen Hochschulcurricula nur in zwei bis vier Semestern überhaupt vertreten ist.

Trotzdem sind Lehrende von Anfängerveranstaltungen häufig von den Resultaten frustriert – fast jeder Informatiker, der versucht hat, einem Anfänger das Programmieren beizubringen, kann von Enttäuschungen berichten[3]. Dies war bei unseren eigenen Veranstaltungen nicht anders.

Am Anfang der Gestaltung der Informatik I-Veranstaltungen in Tübingen und Freiburg in den letzten Jahren stand die Überzeugung, dass es möglich sein muss, den Lernerfolg der Studierenden gegenüber früheren Veranstaltungen deutlich zu verbessern. Der daraus resultierende Verbesserungsprozess lässt sich in zwei Phasen aufteilen: Erhöhung der Stoffdichte (1999–2001) und Verbesserung des Lernerfolgs (2004–).

Die folgenden Abschnitte geben einen kurzen Abriss dieser Entwicklung. Eine ausführlichere Darstellung der Abfolge von Beobachtungen und Verbesserungen findet sich bei Bieniusa et al. [2].

2.1 Erhöhung der Stoffdichte

Ein zentrales Problem der traditionellen Programmierausbildung der 80er und 90er Jahre war die Verwendung von Programmiersprachen, die Programmen nur einen vergleichs-

[3] Bei einer informellen Umfrage beim Ehemaligentreffen des Bundeswettbewerbs Informatik 2007 hatten 100% der ca. 80 Anwesenden dies schon einmal versucht, und dieselben 100% waren von den Resultaten oft enttäuscht.

weise geringen Abstraktionsgrad gestatten (z.B. Pascal, C), und dabei gleichzeitig viel sprachspezifischen Ballast in Form von idiosynkratischen Konstrukten und komplexer Syntax mitbrachten. Dies führte dazu, dass selbst einfache Programmbeispiele viel Platz und Zeit in der Vorlesung bzw. in den Übungen beanspruchten.

Programmiersprachen mit höherem Abstraktionsgrad hingegen versprachen kürzere Programme und weniger spezifischen und syntaktischen Ballast. Die Verwendung der funktionalen Sprache Scheme in der Anfängerausbildung wurde in den 80er Jahren vom MIT ausgehend an vielen Hochschulen populär. Scheme erlaubt als funktionale Sprache insbesondere durch die automatische Speicherverwaltung sowie die Behandlung von Prozeduren als Objekte erste Klasse die Bildung mächtigerer Abstraktionen als traditionellere Sprachen wie Pascal, C oder Java [10]. Außerdem sind funktionale Programme deutlich kürzer [9] und der für die Vorlesung relevante Sprachstandard [11] ist dramatisch kleiner. Da das funktionale Programmieren erlaubt, weitgehend ohne Seiteneffekte aus zukommen, kann in der Vorlesung auf die Querbeziehungen zur Schulalgebra abgehoben werden, und die entstehenden Programme sind leichter zu entwickeln und zu verstehen [6].

Das am MIT entstandene Buch zur Anfängerausbildung [1] gilt noch immer als herausragendes Werk mit vielen kompakten und spannenden Beispielen zur Programmierung und zur Abstraktion. Mittlerweile gibt es eine Reihe von Lehrbüchern, die Scheme verwenden [8, 15].
Dementsprechend stellten Klaeren und Sperber 1999 die Informatik I an der Universität Tübingen von Pascal auf Scheme um, in der Hoffnung, dass durch die Erhöhung der Stoffmenge die Studierenden mehr lernen würden. Daraus ergaben sich sofort tief greifende Konsequenzen für die Veranstaltung:

- Der Zeitanteil der Behandlung der Programmiersprache selbst in der Vorlesung schrumpfte von ca. 50% auf ca. 15%. Dies schuf mehr Zeit für andere Themen.
- Die gegenüber Pascal stark verbesserten Abstraktionsfähigkeiten erlaubten die Behandlung anspruchsvollerer Programmiertechniken wie symbolische Programmierung, Higher-Order-Programmierung, objektorientierte Programmierung, metazirkuläre Interpretation etc.

Damit wurden signifikante Fortschritte in Richtung der von Claus ebenfalls geforderten Kompaktifzierung gemacht: Unsere Verwendung einer abstraktionsstärkeren Programmiersprache entspricht Claus' Idee einer kompakten Notation von Inhalten.

Die Erhöhung der Stoffdichte in der Vorlesung gestattete uns gleichzeitig, die Stoffmenge der zugehörigen Vordiplomsklausur deutlich zu erhöhen. Trotzdem sank die Durchfallquote bei gleich bleibender Klausurlänge und gleich bleibendem Notenspiegel um 5–10% gegenüber Vorgängerjahrgängen. Dies war also ein erster Erfolg unserer Verbesserungsbemühungen. Die inhaltlichen Veränderungen flossen in eine umfassende Revision des Lehrbuchs ein [12].

Die Entscheidung für eine Programmiersprache, die in der Industrie nicht so verbreitet war, wie (damals) C++, brachte uns einige Kritik von Kollegen ein, die insbesondere eine eingeschränkte "Verwendbarkeit" der Studierenden in der Industrie und geringe Anfängerzahlen wegen des befürchteten Attraktivitätsverlusts der Informatik I befürchteten. Beides ist nicht eingetreten. Insbesondere haben die Erfahrungen bei der Durchführung von Folgeveranstaltungen gezeigt, dass die Studierenden, die unsere Informatik I erfolgreich absolviert hatten, kaum Schwierigkeiten hatten, sich in andere Programmierspra-

chen einzuarbeiten. (In Tübingen und Freiburg folgt im Anschluss an die Info I eine Einführung in die objektorientierte Programmierung und Java.)

2.2 Verbesserung des Lernerfolgs

Nach zwei Durchläufen der Veranstaltung in Tübingen und einem Durchlauf in Freiburg nahmen wir Kontakt mit dem TeachScheme!-Projekt der PLT-Gruppe [7] auf, die schon deutlich früher und mit größerem Aufwand angefangen hatte, die Anfängerausbildung durch systematische Beobachtung zu verbessern. Das TeachScheme!-Projekt hatte – deutlich intensiver als wir – das Lernverhalten der Studierenden untersucht und dabei folgende wichtige Beobachtung gemacht:
Die bis dahin generell an Hochschulen (und anderswo) praktizierte Programmierausbildung funktionierte primär durch die Präsentation von Beispielen, in der Erwartung, dass die Studierenden die in den Beispielen demonstrierten Techniken selbst auf neue Probleme übertragen können. Während manche Studierende diese Transferleistung erbringen können, gelingt es vielen nicht. Schlimmer noch: die Studierenden, die im Rahmen eines solchen Lehransatzes tatsächlich lernen, funktionsfähige Programme zu schreiben, produzieren häufig chaotischen Code, der im wesentlichen durch planloses „Basteln" und Ausprobieren entsteht. Dies passt nicht zum gemeinhin propagierten Bild der Informatik als Ingenieurswissenschaft. Außerdem ist es enttäuschend im Hinblick auf die Erfahrung professioneller Software-Entwickler, die systematisch entwickelte Programme unsystematischem „Spaghetti-Code" vorziehen, dieser aber trotzdem in vielen industriellen Projekten grassiert.

Das TeachScheme!-Projekt beobachtete allerdings, dass trotz Verwendung von Scheme obiges Grundproblem der Anfängerausbildung bestehen blieb, wenn primär mit Beispielen gearbeitet wurde. (Die Verwendung von Scheme konnte das Problem immerhin dahingehend lindern, dass die Beispiele nicht mehr primär spezielle Features der verwendeten Programmiersprache demonstrierten, sondern generelle Programmiertechniken.) Diese Beobachtung bestätigte sich bei uns bei einer genaueren Inspektion des Codes, den unsere Studierenden produziert hatten: Viele Programme waren zwar funktionsfähig, aber unsystematisch konstruiert worden. Die übliche Metrik in der Bewertung von Übungs- und Klausuraufgaben hatte dieses Problem nicht registriert. Unsere Analyse zeigte, dass wir zwar in unseren Beispielen systematische Techniken benutzt hatten, diese den Studierenden aber nicht explizit vermittelt hatten.

Das TeachScheme!-Projekt setzt gegen dieses Defizit das Konzept der *design recipes* [5]– eine Sammlung von expliziten systematischen Anleitungen zur Konstruktion von Programmen, das wir unter dem Namen *Konstruktionsanleitungen* übernahmen[4].

Die Konstruktionsanleitungen geben zunächst eine immer gleiche Schrittfolge für die Konstruktion von Prozeduren vor. Diese führen durch eine Analyse der in einer Aufgabe vorkommenden Daten schließlich zu einer so genannten *Schablone*, welche die Form der

[4] Das TeachScheme!-Projekt identifizierte eine anfängergerechte Programmierumgebung als weiteren für den Lernerfolg entscheidenden Faktor.Wir setzen die vom TeachScheme!-Projekt entwickelte DrScheme-Entwicklungsumgebung [4] ein, für die wir spezielle Sprachebenen entwickelten. Hierauf gehen wir allerdings hier aus Platzgründen nicht näher ein.

zu schreibenden Prozedur vorgibt. Kurzdokumentation und testgetriebene Entwicklung spielen ebenfalls entscheidende Rollen. Ein substantieller Anteil der Entwicklung einer Prozedur wird damit mechanisiert. Die vorgegebene Schrittfolge erlaubt den Studierenden, Fortschritte zu machen, ohne schon den gesamten Lösungsweg von vornherein zu sehen. Der verbleibende „kreative" Teil der Aufgabenlösung erscheint ihnen häufig einfach. Die Konstruktionsanleitungen lösen mehrere Probleme:

- Die Studierenden bauen Schwierigkeiten und Ängste am Anfang schnell ab.
- Die Studierenden üben von Anfang an systematisches Vorgehen.
- Auch schwierige Probleme werden durch die Verwendung der Konstruktionsanleitungen zugänglich.

Insbesondere verschwinden beispielsweise die viel beklagten Schwierigkeiten mit der Behandlung von Rekursion durch die Verwendung von Konstruktionsanleitungen völlig. Außerdem sind die Konstruktionsanleitungen die Voraussetzung dafür, dass die Studierende größere zusammenhängende, offene Aufgabenstellungen lösen können: Z.B. gibt es in der Regel ein Übungsblatt, bei dem die Studierenden ein Videospiel selbständig programmieren und eigene Erweiterungen entwerfen und implementieren.

Die geeignete Programmiersprache ist Voraussetzung für die Formulierung und Umsetzung der Konstruktionsanleitungen in der Grundvorlesung: Scheme erlaubt, die Schablonen nachvollziehbar an der Struktur der Daten zu orientieren, ohne substantiellen sprachlichen Ballast. (Die Anleitungen können später auf andere Sprachen wie Java übertragen werden, benötigen dort aber deutlich mehr Raum und idiosynkratische Sprachelemente.)

Im Jahrgang 2004/2005 behandelten wir in Tübingen die Konstruktionsanleitungen erstmals explizit; gleichzeitig begannen wir eine umfassende Überarbeitung des Lehrbuchs [13]. Genauere Beobachtungen ergaben, dass die Konstruktionsanleitungen zwar den Studierenden halfen, wenn sie diese aktiv verwendeten, häufig aber gar nicht zur Anwendung kamen. Zu den Gründen gehörten:

- Viele Studierende hatten zunächst eine aversive Reaktion auf die systematische Vorgehensweise und programmierten „lieber selbst".
- Viele Übungsaufgaben-Lösungen waren von anderen abgeschrieben.

Beides rächte sich für die betroffenen Studierenden bei der abschließenden Klausur, bei der sie die gestellten Aufgaben oft nicht eigenständig lösen konnten. Die Ergebnisse waren somit gegenüber den Vorgängerjahrgängen kaum verbessert.

Wir beschlossen für den Jahrgang 2006/2007 aktive Maßnahmen gegen diese Probleme. In Abschnitt 1 wurden bereits die organisatorischen Aspekte dieser Maßnahmen beschrieben. Das Betreute Programmieren erschwerte Plagiate und ermöglichte uns, direkt auf das Lernverhalten der Studierenden Einfluss zu nehmen. Außerdem gestattete es uns, die Studierenden kontinuierlich beim Lösen von Aufgaben zu beobachten, und somit unsere Erwartungen mit der Realität zu vergleichen und ggf. unmittelbar Korrekturen bei Vorlesung oder Übungen zu veranlassen. Wie in Abschnitt 3 dargelegt, war das Betreute Programmieren außerordentlich erfolgreich und erlaubte uns, das Potential des entwickelten Materials besser zu realisieren.

Insbesondere konnten wir durch das Betreute Programmieren die oben genannten aversive Reaktionen korrigieren, indem wir direkt demonstrierten, dass systematisches Vorgehen deutlich schneller und effektiver zum Ziel führt. Zum Zeitpunkt der Vorlesungs-

umfragen waren keinerlei nennenswerte Unterschiede in der Motivation zu vorhergehenden Veranstaltungen zu beobachten.

Mit der Einführung des Betreuten Programmierens ist das didaktische Instrumentarium für unsere Veranstaltungen vorläufig komplett: Die Vorlesungen 2007 in Tübingen und Freiburg stützten sich auf dasselbe Konzept, mit ähnlichem Erfolg.

3 Beobachtungen und Analyse

In diesem Abschnitt präsentieren wir eine Auswahl der Beobachtungen, die wir gesammelt haben, um den Effekt unserer Lehrmethoden zu evaluieren. Insbesondere versuchen wir zu messen, welchen Effekt das Betreute Programmieren auf die Programmierkenntnisse der Studierenden hat. Außerdem untersuchen wir die Klausurergebnisse, um festzustellen, welche Vorlesungsthemen bereits gut funktionieren und bei welchen Themen wir noch Verbesserungen vornehmen müssen. Die Ergebnisse haben nur bedingt statistische Aussagekraft, liefern aber dennoch nützliche Informationen.

Betreuung Die Tutoren des Betreuten Programmierens hatten von Anfang an den Auftrag, den Teilnehmern aktiv zu helfen, d.h. nicht nur ihre Fragen zu beantworten, sondern kontinuierlich den Arbeitsverlauf zu beobachten und ggf. einzugreifen. Viele Studierenden zeigten sich positiv überrascht, wenn ihnen Hilfe angeboten wurde und stufen die Bedeutung des Betreuten Programmierens entsprechend positiv ein.

Systematisches Vorgehen Viele Studierende mit Programmiererfahrung glauben zunächst nicht, dass systematisches Vorgehen ihnen tatsächlich beim Lösen der Aufgaben helfen kann. Dies hält sie davon ab, mit den ersten Schritten der Konstruktionsanleitungen zu beginnen, obwohl diese ihnen bereits erste Punkte einbringen würden. Stattdessen starren einige buchstäblich den Bildschirm an in der Hoffnung, dass die vollständige Lösung ihnen noch einfällt, oder programmieren erst einmal „drauflos“, was aber in der knapp bemessenen Zeit meist nicht zum Ziel führt. Die Betreuer können in solchen Situationen korrigierend eingreifen.

Plagiate Ein substantieller Anteil der Studierende greift reflexartig zum Plagiat, sobald Schwierigkeiten bei der Lösung von Aufgaben auftreten. Das führte so weit, dass Studierende für das Betreute Programmieren die (zu Hause reproduzierten) Lösungen anderer auswendig lernen, obwohl der Aufwand hierfür den Aufwand für eine fachliche Vorbereitung oft weit übersteigen dürfte. Auch dies deutet auf Defizite in der schulischen Ausbildung hin.

Aufgabenstellungen und Vorwissen Wir waren erstaunt darüber, wie viele Aufgaben die Grenzen des mathematischen Vorwissens vieler Studierender sprengten, z.B. „Teilen mit Rest“. Manche dieser Aufgaben waren schon oft in Vorgängerjahren gestellt worden – die Defizite waren dort aber wegen der Plagiatsproblematik nicht aufgefallen.

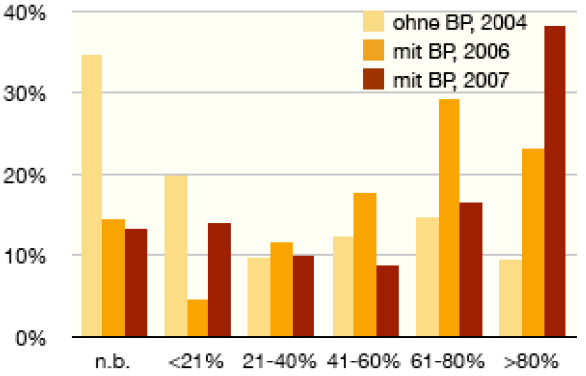
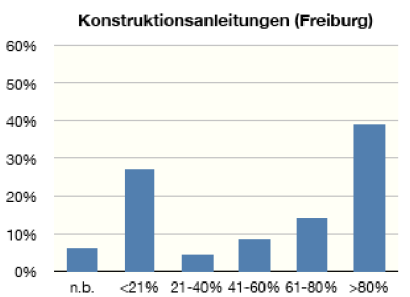
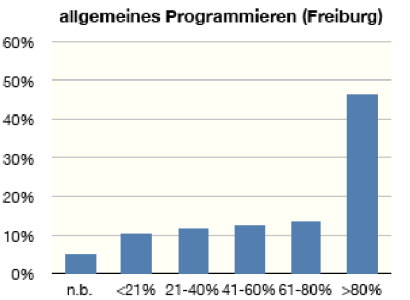


Abb. 1: Klausurergebnisse von Vorlesungen mit und ohne Betreutem Programmieren

Die Klausurergebnisse bestärkten diesen Eindruck: Abb. 1 vergleicht die Ergebnisse der Studierenden in den Programmieraufgaben der Klausuren des Tübinger Jahrgangs 2004/2005 ohne Präsenzübungen mit denen des Tübinger Jahrgänge 2006/2007 und 2007/2008 mit Präsenzübungen. Das Diagramm zeigt, wie viele Studierende (y-Achse) welche Punktzahl (x-Achse) erreicht haben. In der erste Spalte „nicht bearbeitet“ („n.b.“) sind alle Studierenden zusammengefasst, die nicht einmal versucht haben, die Programmieraufgaben in der Klausur zu lösen. Es ist offensichtlich, dass die Ergebnisse der Studierenden, die vom Betreuten Programmieren profitieren konnten, deutlich besser sind als die Ergebnisse der Studierenden des vorherigen Durchgangs ohne Betreutes Programmieren. Dementsprechend ist das Betreute Programmieren fortan fester Baustein unserer Grundausbildung.





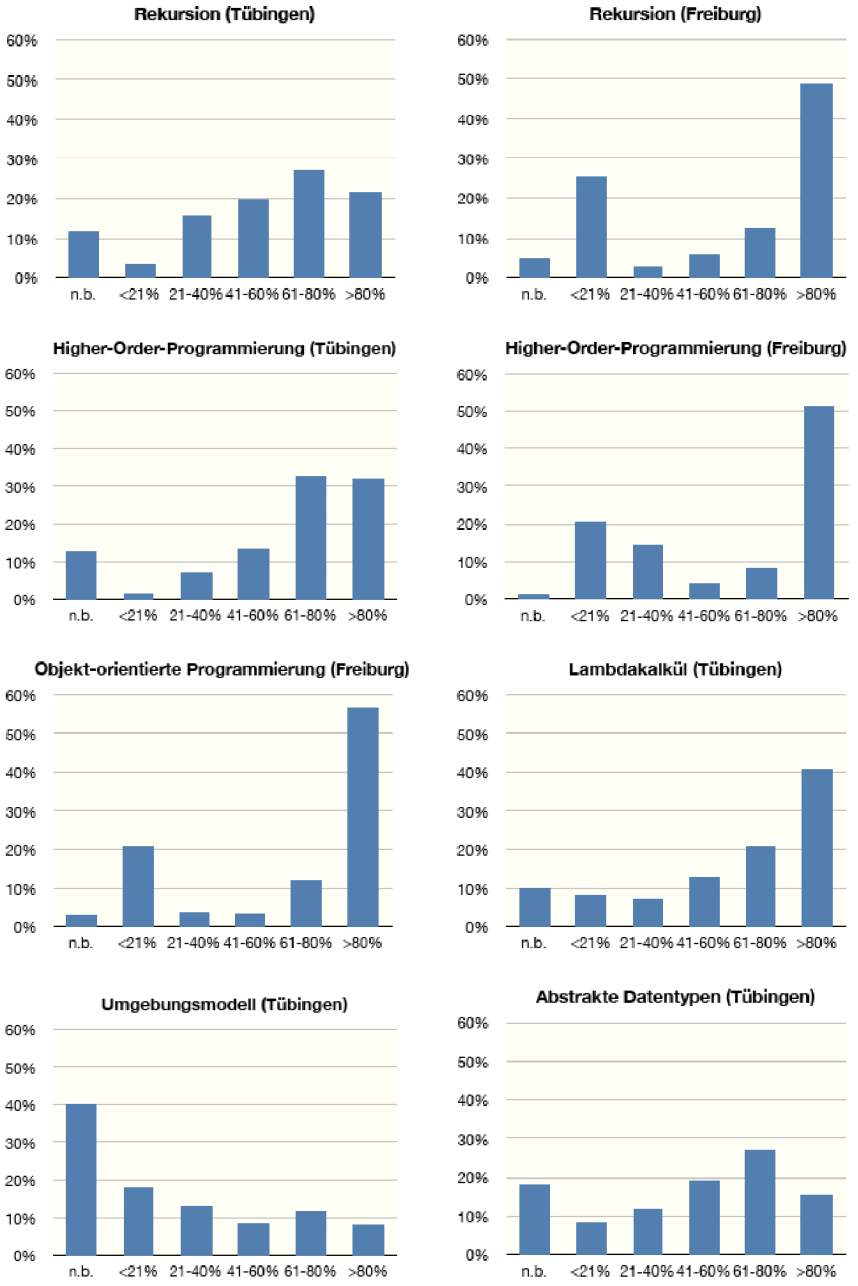


Abb. 2: Ergebnisse bei ausgewählten Klausuraufgaben

Abb. 2 zeigt einen Auszug der Ergebnisse der Studierenden in den Übungs- und Abschlussklausuren bei der Bearbeitung einzelner ausgewählter Aufgaben. (Eine vollständigere Auflistung findet sich in Bieniusa et al. [2].) Die Ergebnisse aus Freiburg stammen aus dem Durchgang 2007/2008, die Ergebnisse aus Tübingen stammen aus dem Durchgang 2006/2007. Die Klausurergebnisse erlauben u.a. folgende Schlussfolgerungen:

- Die Studierenden lösten erfolgreich Programmieraufgaben, die mit dem Benutzen der Konstruktionsanleitungen zu lösen waren. Insbesondere beherrschten die Studierenden, die generell die Programmieraufgaben erfolgreich bearbeiteten, auch die Aufgaben zum Thema Rekursion erfolgreich.
- Auch in einigen theoretischen Themen (exemplarisch hier der Lambda-Kalkül) erzielten die Studierenden gute Ergebnisse.
- Einige Themenbereiche sind noch verbesserungsfähig: Die Ergebnisse der Tübinger Aufgaben zum Umgebungsmodell zeigen, dass das Umgebungsmodell (trotz grafischer Visualisierung) nicht für die Erklärung von Zustand und Mutation geeignet ist. Ebenso ist die Behandlung von abstrakten Datentypen – spezifisch die Unterscheidung zwischen Syntax und Semantik sowie zwischen Spezifikation und Implementierung – noch nicht zufrieden stellend.

Generell ermitteln wir durch die Aufschlüsselung der Klausurergebnisse die Themenbereiche, bei denen sich eine ähnliche Behandlung in Folgejahrgängen lohnt, und solche, die noch zu verbessern sind: Bei den angesprochenen Themen „Umgebungsmodell“ und „Abstrakte Datentypen“ werden wir im kommenden Semester alternative Ansätze erproben.

4 Zusammenfassung und Ausblick

Das in Tübingen und Freiburg entwickelte Konzept zur inhaltlichen Gestaltung und Durchführung einer Einführungsvorlesung in der Informatik erfüllt wichtige Anforderungen, die an die Grundausbildung gestellt werden müssen:

- hohe Stoffdichte und Abdeckung der zentralen Programmierkompetenzen,
- hohe Lernerfolgsquote und
- robuste Ergebnisse, unabhängig von der Person des Dozenten.

Während Claus in den Mittelpunkt seiner Klassifikation von Grundvorlesungen organisatorische Aspekte der Durchführung gerückt hat, liefern unsere Erfahrungen wichtige Indizien dafür, dass die inhaltliche Gestaltung die entscheidenden Voraussetzungen für einen guten Lernerfolg liefert, der dann durch organisatorische Maßnahmen gesteigert werden kann. Es sind eine Reihe von Voraussetzungen nötig, die den guten Lernerfolg bei unserer Grundausbildung im Programmieren gewährleisten:

1. Ein didaktisches Konzept, das konsequent auf systematische Programmierung setzt,
2. eine Programmiersprache, die erlaubt, das didaktische Konzept direkt umzusetzen, ohne viel Zeit bei der Präsentation zu beanspruchen,
3. eine Programmierumgebung, die auf die Bedürfnisse von Programmieranfängern zugeschnitten ist, und
4. genaue Beobachtung des Lernverhaltens über die Dauer der Veranstaltung und Steuerung des Lernverhaltens durch Betreutes Programmieren.

Keiner dieser Aspekte ist dozentenabhängig; eine möglichst gute Grundausbildung erfordert primär die systematische Beobachtung und die Umsetzung der sich daraus erge-

benden Konsequenzen. Unsere Beobachtungen weisen darauf hin, dass der von uns erreichte Lernerfolg eine substantielle Verbesserung gegenüber der „traditionellen“ Grundvorlesung ist. Wir mutmaßen, dass ähnliche Lernerfolge in vergleichbarem zeitlichen Rahmen mit einer traditionellen inhaltlichen Gestaltung (objektorientiert, mit industrieller Programmiersprache) nicht zu erreichen sind.

Der Verbesserungsprozess ist auch bei uns nicht abgeschlossen: die anstehenden Durchläufe in Tübingen und Freiburg werden alternative Ansätze zu den Einheiten erproben, die in der Vergangenheit nicht zufrieden stellend abgeschnitten haben.

Literatur

[1] H. Abelson, G. J. Sussman, J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 2. Edition, 1996.

[2] A. Bieniusa, M. Degen, P. Heidegger, P. Thiemann, S. Wehr, M. Gasbichler, M. Crestani, H. Klaeren, E. Knauel, M. Sperber. HtDP and DMdA in the battlefield. In F. Huch, A. Parkin (Ed.) *Functional and Declarative Programming in Education*, Victoria, BC, Canada, September 2008.

[3] V. Claus. 2b v :2b - Maßnahmen zur Förderung der Hochschuldidaktik Informatik. In P. Forbrig, G. Siegel, M. Schneider (Ed.), *HDI 2006: Hochschuldidaktik der Informatik, Organisation, Curricula, Erfahrungen, 2. GI-Fachtagung*, vol. 100 der *LNI*, S. 11–22, München, Dezember 2006. Springer-Verlag.

[4] M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi. The DrScheme project: An overview. *SIGPLAN Notices*, 33(6): 17–23, Juni 1998.

[5] M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[6] M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi. The structure and interpretation of the computer science curriculum. In *Functional and Declarative Programming in Education (FDPE)*, S. 21–26, 2002.

[7] M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, März 2004.

[8] D. P. Friedman, M. Wand, C. T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 2. Edition, 2001.

[9] P. Hudak, M. P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. :, an experiment in software prototyping productivity. Technical report, Yale University, Department of Computer Science, New Haven, CT 06518, Juli 1994.

[10] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2): 98–107, 1989.

[11] R. Kelsey, W. Clinger, J. Rees. Rev. report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1): 7–105, 1998.

[12] H. Klaeren, M. Sperber. *Vom Problem zum Programm*. Teubner, Stuttgart, 3. (überarbeitete) Auflage, 2001.

[13] H. Klaeren, M. Sperber. *Die Macht der Abstraktion*. Teubner Verlag, 1. Auflage, 2007.

[14] P. Norvig. Teach yourself programming in ten years. http://norvig.com/21-days.html, 2001.

[15] C. Wagenknecht. *Programmierparadigmen*. Teubner, 2004.