Eine softwaretechnische Programmierausbildung?

Axel W. Schmolitzky, Universität Hamburg

schmolit@informatik.uni-hamburg.de

Zusammenfassung

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

1 Einleitung

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

1.1 Programmierausbildung…

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

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

1.2 …softwaretechnisch konkretisiert

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

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

1.3 Aufbau

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

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

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

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

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

Tabelle 1: Diskutierte Alternativen

2 Alternativen auf Studiengangsebene

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

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

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

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

2.1 Wahl und Reihenfolge der Paradigmen

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Programmiersprache und Programmierung

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

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

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

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

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

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

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

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

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

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

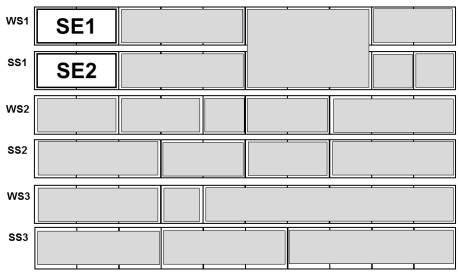

Abb. 1: Anteil der einführenden Pflichtmodule zur Programmierung in den Bachelorstudiengängen der Universität Hamburg mit hohem Informatik-Anteil

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

2.3 Algorithmen vs. Interaktive Systeme

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

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

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

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

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

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

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

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

3 Organisatorische Alternativen

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

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

3.1 Vorlesungen: Folienfilme versus Live-Programmierung

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

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

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

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

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

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

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

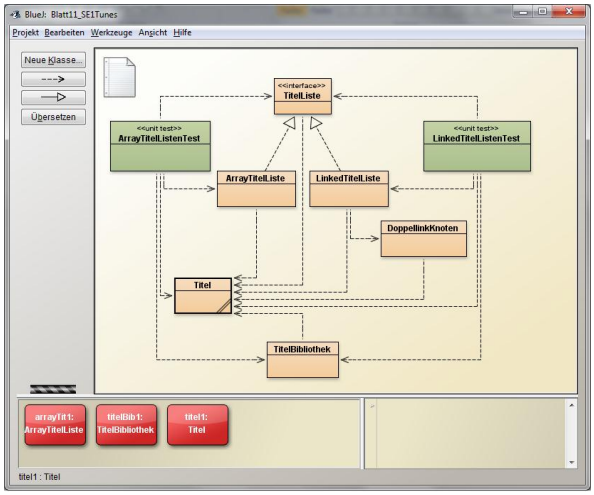

Abb. 2: Die Oberfläche von BlueJ an einem Beispiel

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

3.2 Übungen: Heimarbeit versus Präsenz

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

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

bei der Gestaltung von Programmierübungen unterschiedliche Möglichkeiten der Umsetzung.

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

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

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

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

Konsequenzen: Ein Präsenzbetrieb bei den Übungen hat mehrere Konsequenzen:

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

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

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

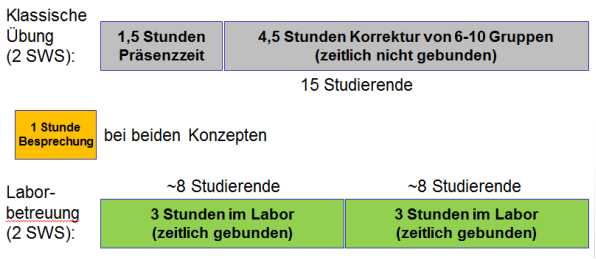

Abb. 3: Betreueraufwand pro Woche im Vergleich

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

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

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

4 Inhaltliche Alternativen

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

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

4.1 Smarte Sammlungen vor Arrays

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

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

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

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

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

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

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

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

4.2 Interfaces vor Vererbung

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

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

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

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

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

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

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

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

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

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

5 Diskussion

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

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

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

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

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

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

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

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

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

6 Zusammenfassung

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

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

Literatur

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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