Andreas Zeller · Marcus Deininger (Hrsg.) (2007): Software Engineering im Unterricht der Hochschulen
SEUH 10, Stuttgart, dpunkt.verlag, Heidelberg

Einführung in die Softwareentwicklung – Softwaretechnik trotz Objektorientierung?

Axel Schmolitzky · Heinz Züllighoven

Arbeitsbereich Softwaretechnik, Department Informatik, Universität Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg
{schmolitzky|zuellighoven}@informatik.uni-hamburg.de

Zusammenfassung

Kaum ein Thema ist so umstritten wie der richtige Weg bei der grundständigen Programmierausbildung. Dieser Beitrag beschreibt einen neuen Ansatz zur Einführung in die Softwareentwicklung, der in den letzten Jahren im Arbeitsbereich Softwaretechnik an der Universität Hamburg entwickelt wurde. Er orientiert sich an einem Objects First-Ansatz, macht aber gleichzeitig einen Einstieg in klassische Themen wie Algorithmen und Datenstrukturen früher als allgemein üblich und geht einen ungewöhnlichen Weg bei der Vermittlung von Interfaces und Vererbungskonzepten.

1 Einführung

Programmierausbildung im Informatikstudium: Zuerst funktional oder zuerst imperativ? *Objects First* oder *Algorithms First*? *Modeling First* oder *Interaction First*? Was ist mit *Test First*? Kaum ein Thema ist so umstritten wie der richtige Weg bei der grundständigen Programmierausbildung. Einigkeit besteht wohl nur bei der Aussage, dass Programmieren als das Handwerkszeug der Softwareentwicklung nach wie vor eine zentrale Rolle in der Ausbildung spielen sollte.

Zumindest aus Sicht der Softwaretechnik lassen sich einige der eingangs gestellten Fragen leichter beantworten. Da sich in der praktischen Softwareentwicklung das objektorientierte Paradigma in den letzten Jahren durchgesetzt hat, steht außer Zweifel, dass gute Kenntnisse objektorientierter Programmierung notwendig sind. Die funktionale Programmierung hingegen verliert zunehmend an Stellenwert, was einige bedauern; sie wird aus softwaretechnischer Sicht eher als eine hilfreiche Ergänzung denn als elementar angesehen.

Andreas Zeller · Marcus Deininger (Hrsg.) (2007): Software Engineering im Unterricht der Hochschulen
SEUH 10, Stuttgart, dpunkt.verlag, Heidelberg

Auch bei eher konservativen Studiengängen, die in den letzten beiden Jahrzehnten beim Einstieg in die Programmierung nicht vom imperativen auf das funktionale Paradigma umgestellt haben, wird jetzt häufiger gefragt: Klassisch imperativ, beginnend mit Kontrollstrukturen und Prozeduren (in der Tradition imperativer Sprachen wie Pascal, Modula-2 und C) oder konsequent objektorientiert mit Objekten vom ersten Tag an, unter Nutzung der Möglichkeiten von Sprachen wie Java und C#?

Und schließlich: Bedeutet »konsequent objektorientiert« nicht insbesondere eine frühe Vermittlung des für die Objektorientierung laut Wegner [Weg87] so zentralen Konzeptes der Vererbung? Dann stellt sich aus Sicht der Softwaretechnik schon die Frage, ob dieses leicht zu missbrauchende Instrument (Systeme mit großen Vererbungshierarchien gelten als sehr schwierig zu warten) nicht zu früh in die Hände noch unerfahrener Programmierer gegeben wird; nicht zu unrecht wurde Vererbung auch schon als das GoTo der 90er Jahre bezeichnet.

Unser Eindruck ist, dass statt polarisierender Alternativen eine geeignete Kombination gewählt werden kann. Dieser Beitrag beschreibt einen Ansatz, den wir in den letzten Jahren im Arbeitsbereich Softwaretechnik an der Universität Hamburg entwickelt haben. Darin sind viele Erfahrungen eingeflossen, die wir in etlichen Jahren der praktischen Programmierausbildung an der Hochschule und im industriellen Umfeld gesammelt haben.

Der Text gliedert sich wie folgt: In Abschnitt 2 wird kurz der Hintergrund dargestellt, vor dem der neue Ansatz entstanden ist. Abschnitt 3 diskutiert verschiedene Entwurfsprinzipien, die bei der Gestaltung einer Programmiereinführung berücksichtigt werden können. Die Abschnitte 4, 5, 6 und 7 bilden den Kern des Beitrags, in dem der inhaltliche und didaktische Aufbau zweier konsekutiver Veranstaltungen zur Einführung in die Softwareentwicklung vorgestellt wird, samt einer ersten Bewertung. Abschnitt 8 fasst den Beitrag zusammen.

2 Hintergrund

Seit dem Wintersemester 2005/06 löst der *Bachelor of Science in Informatik* das Informatik-Diplom an der Universität Hamburg ab. Im Zuge der Umstellung wurde die Chance genutzt, die einführenden Veranstaltungen substanziell neu zu strukturieren. Die in [BBSZ03] beschriebenen P-Veranstaltungen (einführende Veranstaltungen zur Programmierung) mit funktionaler Programmierung als Einstieg wurden abgelöst durch eine Modulfolge zur Softwareentwicklung (SE), die mit der imperativen Programmierung beginnt und erst im dritten Semester (SE3) wahlweise die funktionale oder die logische Programmierung anbietet. Sie wird ergänzt durch das Modul »Algorithmen und Datenstrukturen« im dritten Semester. Der Arbeitsbereich Softwaretechnik ist derzeit verantwortlich für die ersten beiden Module, Softwareentwicklung I (SE1) und Softwareentwicklung II (SE2). SE1 besteht aus Vorlesung und Übungen (jeweils zweistündig) und soll die impe-

rative und objektorientierte Programmierung vermitteln. SE2 bietet zwei parallele zweistündige Vorlesungen mit einer gemeinsamen Übung (4+2); eine Vorlesung behandelt objektorientierte Programmierung und Modellierung (OOPM), die andere thematisiert vor allem Softwareentwicklungsprozesse unter dem Titel »Softwaretechnik und -Ergonomie«.

Im Übungsbetrieb beider Module haben wir bewährte Konzepte aus der alten P-Veranstaltung P2 übernommen, die wir in verschiedenen Kontexten bereits beschrieben haben [Sch06a]: ein betreuter Laborbetrieb, Arbeiten in Paaren, möglichst viel direktes Feedback. Auf diese eher prozessbezogenen Aspekte wollen wir hier nicht näher eingehen. In diesem Beitrag konzentrieren wir uns auf die inhaltliche Gestaltung von SE1 und dem darauf unmittelbar aufbauenden OOPM-Anteil von SE2.

3 Gestaltungsmuster

Bevor wir unseren neuen Ansatz ausführlich beschreiben, stellen wir in diesem Abschnitt einige grundsätzliche Möglichkeiten dar, den Einstieg in die Programmierausbildung zu gestalten. Teilweise haben wir diese Ansätze in Lehrbüchern vorgefunden, teilweise in den Beschreibungen von Informatik-Studiengängen. Einige sind bereits in der wissenschaftlichen Gemeinde diskutiert worden [Bru04]. Die hier vorgestellte Auswahl erhebt dabei keinen Anspruch auf Vollständigkeit.

Einige der beschriebenen Muster tragen ein »First« in ihrem Namen. Sie folgen damit dem didaktischen Muster des »frühen Vogels« (siehe das *Early Bird Pattern* im Pedagogical Pattern Projekt [PPP]), das besagt, dass zentrale Konzepte früh vermittelt werden sollten, damit sie länger wirken und auch häufiger wiederholt werden können als spät vermittelte. Offensichtlich führt ein Befolgen aller First-Ansätze, falls vereinbar, zu einer unrealistischen Stoffdichte in den ersten Wochen.

3.1 Algorithms First

Die meisten Einführungen beruhen auf der Annahme, dass Algorithmen im Kern der Informatik stehen: ohne ein solides Verständnis von grundlegender Algorithmik keine soliden Programmierkenntnisse. Für einen »sanften« Einstieg kann dafür das klassische Paar »Algorithmen und Datenstrukturen« zuerst ohne Datenstrukturen betrachtet werden: Prozeduren oder Funktionen mit festen Ein- und Ausgabegrößen berechnen Lösungen für kleinere und mittlere Probleme. Wir bezeichnen diesen Ansatz hier verkürzt als *Algorithms First*-Ansatz.

In der imperativen Welt wird so das Prozedur-Konzept früh eingeführt. Beim Einstieg mit einer funktionalen Sprache wird die reine Algorithmik noch stärker betont. So lässt sich beispielsweise ein Quicksort in ML deutlich einfacher (in einer Zeile) formulieren als in jeder imperativen Umsetzung.

3.2 Objects First

Der Objects First-Ansatz basiert auf einer simplen Grundannahme: Moderne Softwaresysteme bestehen aus einer großen Anzahl von Objekten, die miteinander auf vielfältige Weise interagieren; wenn Objekte die elementaren Bausteine solcher Systeme sind, dann sollten diese Bausteine von Beginn an behandelt werden. So eingängig die Grundidee ist, so schwierig ist ihre Umsetzung. In einer Diskussion auf einer SIGCSE-Mailingliste wurde beklagt, dass der Ansatz ohne Werkzeugunterstützung kaum gangbar ist [Bru04]. Entsprechend existieren zahlreiche Unterstützungen, von Mikrowelten über didaktisch motivierte Werkzeuge hin zu Bibliotheken einfacher grafischer Objekte.

Eines der Werkzeuge mit didaktischem Anspruch ist BlueJ [KQPR03], eine Entwicklungsumgebung für Java, die explizit für die Vermittlung objektorientierter Programmierung entwickelt wurde. Hier können auf einer Objektleiste einzelne Objekte interaktiv erzeugt und untersucht werden; die Klassenstruktur des aktuellen Projektes wird in einem interaktiven UML-Diagramm visualisiert und ermöglicht das direkte Erzeugen von Exemplaren dieser Klassen.

Da ein Werkzeug ohne begleitende Kursstruktur zu leicht »missbraucht« werden kann, habe die BlueJ-Designer ein Lehrbuch [BaK06] geschrieben, das BlueJ mit einem konsequenten Objects First-Ansatz nutzt. Die dritte Auflage dieses Buches basiert auf Java 1.5 und ist derzeit eines der erfolgreichsten Lehrbücher zu diesem Ansatz (es wurde u.a. bereits in fünf Sprachen übersetzt).

3.3 Modeling/Design First

Der Modeling First-Ansatz folgt dem Grundsatz, dass Programmieren primär dem Problemlösen dient. Um ein Problem zu lösen, muss zuerst der Problembereich geeignet modelliert werden. Hier spielt eine wichtige Rolle, dass die objektorientierte Programmierung auch deshalb so erfolgreich wurde, weil sie ohne Strukturbruch einen Ausschnitt der Welt objektorientiert als ein System interagierender Exemplare von Klassen repräsentieren kann.

Eine Einführung in diesen Ansatz zielt also darauf ab, Gegenstände der natürlichen Welt objektorientiert zu modellieren und idealerweise auch zu simulieren. Klassisch ist das Restaurant-Beispiel aus dem COOL-Projekt (u.a. von Kristen Nygaard) [COOL], in dem ein Restaurant mitsamt Einrichtung, Gästen und Kellnern als ein System interagierender Objekte modelliert werden soll. Bewusst wird zum Einstieg kein simples Beispiel gewählt, sondern ein ausreichend komplexes Beispiel, mit dem alle Facetten der objektorientierten Programmierung ausgelotet werden können. Erste Erfahrungen mit diesem Ansatz sind in [BeC04] beschrieben.

3.4 Interactive Programming

Interactive Programming ist ein Ansatz für das erste Semester der Programmierausbildung, der von Lynn Andrea Stein seit über zehn Jahren vertreten und seit mehreren Jahren praktiziert wird, anfangs am MIT, inzwischen am Franklin W. Olin College of Engineering [RCS101]. Ihr Ansatz geht von interagierenden Objekten aus, jedoch nicht nach einem sequenziellen Programmiermodell, sondern als nebenläufige Einheiten. Stein betont den Bezug zur realen Welt mit ihren parallel agierenden Einheiten und erwartet so einen intuitiven Zugang für Programmieranfänger. Sie wendet sich bewusst gegen das Paradigma *Computation as Calculation* und betont die Interaktion zwischen Programmeinheiten innerhalb fortlaufender Prozesse. Einen positiven Erfahrungsbericht liefert Weber-Wulff in [Web00].

3.5 Programmiersprache zuerst, Konzepte danach

Ein häufig anzutreffender Ansatz in vielen Studiengängen ist, eine Programmiersprache als Handwerkszeug zu sehen und sie vollständig im ersten Semester zu vermitteln; im zweiten Semester folgt dann häufig eine Programmierveranstaltung mit Algorithmen und Datenstrukturen als explizitem Thema. Dabei werden die Sprachkonstrukte des ersten Semesters auf klassische algorithmische Probleme der Informatik angewendet.

Problematisch an diesem Ansatz ist, dass eine hohe »Konzeptdichte« im ersten Semester entsteht: Java ist zwar einfacher als vergleichbare Sprachen (C++, C#), umfasst aber seit der Java-Version 1.5 anspruchsvolle Konzepte wie Generizität, Vererbung und Nebenläufigkeit; aus unserer Sicht ein zu hartes Themenpaket für Programmieranfänger.

3.6 Der historischen Entwicklung folgen

Eine weitere häufige Vorgehensweise könnte so formuliert werden: Programmiersprachen haben sich über mehrere Jahrzehnte entwickelt; vermittle die Sprachkonzepte entlang ihrer historischen Entwicklung.

Zwei Beispiele sollen dies verdeutlichen:

- C++ wurde als Nachfolger der Programmiersprache C entworfen, indem vor allem objektorientierte Konzepte hinzugefügt wurden. Das Klassenkonzept ist auf diese Weise in eine imperative Sprache hineingewachsen, so dass neben den Verbünden mit dem Schlüsselwort `struct` auch Klassen mit dem Schlüsselwort `class` definiert werden konnten. Die Ähnlichkeiten sind groß, und Klassen können auf diese Weise als eine Erweiterung des Verbund-Konzeptes angesehen werden. Für den Weg von C zu C++ ist diese Entwicklung relevant, für die Vermittlung der Idee von Klassen und Objekten scheint sie uns eher

Andreas Zeller · Marcus Deininger (Hrsg.) (2007): Software Engineering im Unterricht der Hochschulen
SEUH 10, Stuttgart, dpunkt.verlag, Heidelberg

hinderlich. Ein Beispiel für eine aktuelle Anwendung dieses Prinzips für Java ist in [Mös05] zu finden.

- Das Sprachkonstrukt *Interface*, das durch Java in den Mainstream der Programmiersprachen eingeführt wurde, ist historisch gesehen eine Weiterentwicklung der *abstrakten Klasse*. Eine Ausprägung abstrakter Klassen sind *vollständig abstrakte Klassen*, die als reine Spezifikation der zu implementierenden Funktionalität dienen; Interfaces sind letztendlich genau dies. Interfaces können aber auch als eigenständiges Konzept angesehen werden, das unabhängig von Fragen der Implementationsvererbung ist. In Lehrbüchern zu Java werden aber üblicherweise Interfaces nach Vererbung und abstrakten Klassen eingeführt [Sch06a].

Diese Herangehensweise ist meist motiviert durch profundes (technisches) Hintergrundwissen der Dozenten über Programmiersprachen, das den meisten Programmieranfängern jedoch üblicherweise fehlt.

3.7 Erst Konsumieren, dann Produzieren

Eines unserer eigenen Entwurfsprinzipien beim Kursaufbau lautet: *Konsumieren* ist einfacher als *Produzieren*, vermittle deshalb das Konsumieren eines Konzeptes, ehe es produktiv eingesetzt wird. Nehmen wir z.B. das Konzept »geschriebener Text«: Das Lesen eines Textes (das Konsumieren) ist zwar eine anspruchsvolle Tätigkeit, die eine solide Ausbildung erfordert; das Schreiben eines Textes (das Produzieren) ist jedoch noch anspruchsvoller und wird üblicherweise später vermittelt. Um einen Text zu lesen müssen darüber hinaus nicht alle Details eines geschriebenen Textes (Grammatik, Stil, Argumentationslinien etc.) bekannt sein; man kann Texte auch mit partiellen Kenntnissen dieser Details lesen.

Angewendet auf Programmiersprachkonzepte lässt sich das Prinzip an zwei Beispielen veranschaulichen:

- Ein *konsumierendes* Verständnis von Paketen in Java (packages) bedeutet, dass bereits implementierte Dienstleistungen von Klassen in benannten Paketen liegen und von dort importiert werden können; dieser Umstand kann relativ früh vermittelt werden. Pakete zu produzieren schließt die Fähigkeit ein, größere Systeme in Subsysteme zu zerlegen; dazu sollten alle Konzepte der Paketsichtbarkeit (für Klassen und Methoden) verstanden sein, die aber erst beim eigenen Umgang mit vergleichsweise großen Systemen nachvollziehbar werden und somit später vermittelt werden sollten.
- Das *Konsumieren* von Generizität in Java setzt voraus, dass beim Umgang mit Sammlungen des Java Collection Frameworks der Typ der Elemente deklariert werden muss; dies kann sehr früh in der Ausbildung vermittelt werden. Das produzierende Definieren einer generischen Klasse erfordert etliches mehr: das Wissen über formale und aktuelle Typparameter, über Einschränkungen bei der Benutzung formaler Typparameter, über Type Erasure u.Ä.

Andreas Zeller · Marcus Deininger (Hrsg.) (2007): Software Engineering im Unterricht der Hochschulen
SEUH 10, Stuttgart, dpunkt.verlag, Heidelberg

Für Lehrende bietet dieses Prinzip den Vorteil, dass vermeintlich komplizierte Themen früh angesprochen werden können, wenn die Voraussetzungen für ein konsumierendes Verständnis geeignet ausgewählt werden. Aus didaktischer Sicht ist dies eine Anwendung des *Spiralmusters* aus dem Pedagogical Pattern Project [PPP]; dieses besagt, dass nicht alles zu einem Thema auf einmal gesagt werden muss, sondern dass nur ein Teil vermittelt und das Thema später wieder aufgegriffen und vertieft werden kann.

4 Unsere Einführung in die Softwareentwicklung

Für unsere zweisemestrige Einführung in die objektorientierte Softwaretechnik haben wir nach dem gerade genannten Prinzip einige Themen ausgewählt, deren Spannungsbogen von den Objekten (zur Laufzeit) über die Klassen (als grundlegende Konstruktionseinheiten) zu den Softwaretechnikkonzepten hinter dem Entwurf reicht. Dabei machen wir deutlich, dass Softwaretechniker beim Entwurf üblicherweise nicht die ganze Welt modellieren, sondern bewusst das Systemmodell (Was bauen wir in Software?) vom Anwendungsmodell (In welchem Kontext läuft die Software?) trennen sollten. Eine Übersicht der beiden Semester lässt sich so charakterisieren:

- Semester 1: Objects First und Interactive Programming über (nicht grafische) Schnittstellen von Objekten mit BlueJ, frühes Testen über Schnittstellen mit Interfaces, früher Einstieg in Algorithmen und Datenstrukturen am Beispiel von Sammlungsimplementationen.
- Semester 2: Abstraktionen über die Programmierung mit abstrakten Datentypen, dem Vertragsmodell und verschiedenen Formen der Polymorphie (Subtyping, Generizität); Implementationsvererbung und objektorientierte Modellierung von Werten und Objekten; Grundlagen grafischer Benutzungsschnittstellen, von Entwurfsmustern und Refactorings.

Im ersten Semester beschränken wir uns bewusst auf die handwerklichen Aspekte der Programmierung und bleiben im objektbasierten Sprachmodell einer Sprache (Java). Erst auf Basis dieser Programmiererfahrung abstrahieren wir im zweiten Semester stärker von der verwendeten Sprache und stellen weitergehende Konzepte vor (Pass-by-Reference, benutzerdefinierte Wert- und Referenzsemantik etc.).

Was haben wir bewusst ausgelassen?

Zunächst erläutern wir hier, welche der in Abschnitt 3 genannten Grundsätze wir bewusst ausgelassen oder weniger stark berücksichtigt haben.

Wie bereits in [BBSZ03] beschrieben, bringen wir in unseren Programmierveranstaltungen unseren Industriehintergrund mit ein: Beim »Training on the Job« hat sich eine enge Verbindung von Konzeptvorstellung und praktischer Um-

setzung bewährt; häufig lassen wir in Schulungen erst praktische Erfahrung sammeln, um dann die Theorie aufzuarbeiten. Dabei zeigt sich immer wieder: Erfahrungsgrundiertes Konzeptwissen benötigt Zeit. Eine zu hohe Verdichtung oder mangelnde praktische Erfahrung verhindert ein nachhaltiges Verständnis. Eine vollständige Behandlung von Java in nur einem Semester (siehe 3.5) kam für uns deshalb nicht in Frage.

Zum Thema *Modeling First* haben wir bereits in früheren P2-Veranstaltungen Erfahrungen gesammelt, die nicht ermutigend waren: Bereits sehr früh sollte dort, ausgehend von Szenarios der realen Welt, ein System für ein Taxiunternehmen modelliert werden. Da die Studierenden noch kein Gefühl dafür hatten, was von einem objektorientierten System geleistet werden kann (und was nicht), waren die Modelle entsprechend haarsträubend. Wir sind deshalb inzwischen der Meinung, dass vor den ersten Modellierungsversuchen eine handwerkliche Vermittlung der grundlegenden Bausteine (Objekte und Klassen) mit ihren technischen Eigenschaften stehen sollte, um die Zielwelt einer objektorientierten Modellierung zu verdeutlichen.

Im Entwurf der ersten beiden Semester haben wir auch bewusst die nebenläufige Programmierung ausgelassen (sie wird in unserem Bachelor-Lehrplan frühestens im dritten Semester in den Modulen zu Datenbanken und zu Systemsoftware thematisiert). Während wir, ähnlich wie Wegner in [Weg97], beim Umgang mit Computern einen Trend zu interaktiven Systemen erkennen, der die Bedeutung reiner Berechnungen durch Algorithmen in den Hintergrund treten lässt, und somit auch das Interactive Programming als einen lohnenswerten Ansatz sehen, schätzen wir die explizite Berücksichtigung nebenläufiger Prozesse als zu ambitioniert für die ersten Semester ein.

5 Semester 1: Grundlagen der objektorientierten Programmierung

Der Anspruch im ersten Semester ist, *Programmieranfänger* an das systematische Programmieren heranzuführen. Wir nehmen dabei bewusst in Kauf, dass einige Studierende bereits vor dem Studium Erfahrungen mit imperativer Programmierung gesammelt haben und bei Teilen der Veranstaltung unterfordert sind. Wir versuchen jedoch, durch das Laborkonzept diese Vorkenntnisse zu nutzen, indem die erfahreneren Studierenden ihr Wissen in einem Programmierpaar an Anfänger weitergeben sollen. Unsere Erfahrungen damit sind jedoch gemischt, denn nicht jedem Studierenden mit Vorerfahrung liegt es, sein Wissen in didaktisch geeigneter Form weiterzugeben.

Wir haben, insbesondere für das erste Semester, sogenannte *Komplexitätsstufen* für die Lehre entworfen, um kleine Objektsysteme einzuordnen. Jede Stufe definiert dabei Konzepte (samt reservierter Wörter), die auf den Konzepten niedrigerer Stufen aufbauen. SE1 strukturieren wir in vier Komplexitätsstufen. Wir

wollen auf diese Stufen im Rahmen dieses Beitrags nicht näher eingehen, erwähnen sie jedoch an einigen Stellen im nachfolgenden Text. Ein erster Ausblick wurde für einen Workshop der ECOOP 2005 gegeben [Sch05].

5.1 Objects First mit BlueJ

Beim Einstieg mit BlueJ werden Klassen vorgegeben, die anfangs so konsumiert werden, dass interaktiv Exemplare erstellt und manipuliert werden. Wir nutzen dabei eine Besonderheit von BlueJ: Beim Doppelklick auf eine Klasse lassen wir nicht, wie üblich, den Editor in der Quelltextansicht erscheinen, sondern in der von javadoc erzeugten Schnittstellenansicht der Klasse. Die Studierenden haben so früh Kontakt mit der Idee einer *Schnittstelle*, die die benutzbaren Operationen beschreibt, ohne Details der Realisierung preiszugeben.

5.2 Java Level 1: Vereinfachte Syntaxbeschreibung

Ein wichtiges Detail in der Programmierausbildung sind formale Syntaxbeschreibungen. Hier sollen die Studierenden praktisch erfahren, dass solche Beschreibungen für den Umgang mit einer Programmiersprache nützlich sind. Wir hoffen, dass die Studierenden anschließend in der theoretischen Informatik der Behandlung formaler Sprachen leichter folgen können. Problematisch ist dabei, dass aufgrund der neuen Spracheigenschaften die vollständige Syntaxbeschreibung von Java 1.5 so kompliziert geworden ist, dass sie Anfängern kaum noch vermittelbar ist. Wir haben deshalb für unsere erste Komplexitätsstufe eine abgespeckte Syntaxbeschreibung erstellt, die aufgrund der wenigen Konzepte (u.a. noch keine Schleifen) auf einer A4-Seite Platz findet.

5.3 Interfaces zur Modellierung von Schnittstellen

Während die zweite Komplexitätsstufe Referenztypen einführt und sich mit Iteration und Rekursion beschäftigt, stellt die dritte Stufe die Interfaces von Java als eine Möglichkeit vor, die Schnittstelle einer Klasse explizit zu beschreiben. Motiviert wird dies über den Anspruch, in einer Testklasse Black-Box-Tests zu implementieren; wenn die Testklasse nur ein Interface benutzt, wird die Black-Box-Eigenschaft eher garantiert als in einem direkten Test der Klasse. In einer weiteren Woche benutzen die Studierenden die Interfaces `Set` und `List` des Java Collection Frameworks, um kleine Probleme mit Hilfe von Sammlungen zu lösen. Sie lernen konsumierend, dass es für `List` zwei Implementationen geben kann (`LinkedList` und `ArrayList`), die beliebig austauschbar sind.

Wir zeigen, dass ein Interface/Typ auf verschiedene Weise implementiert werden kann, und unterscheiden dabei den statischen und dynamischen Typ einer Variablen. So stellen wir ein erstes Beispiel für Polymorphie vor, ohne auf die

volle Komplexität von Implementationsvererbung eingehen zu müssen (Konsumieren von Vererbung, aber noch kein Produzieren mit Vererbung). Diesen Ansatz haben wir bereits an anderer Stelle beschrieben [Sch06b].

5.4 Sammlungen: Mengen und Listen implementieren

Auf der vierten Stufe stellen wir nur noch wenige neue Sprachelemente vor; stattdessen vertiefen wir in den letzten fünf Wochen die auf den ersten drei Stufen vermittelten Konzepte anhand eines zentralen Themas: Implementierungen von Sammlungen. Hier werden dynamische Datenstrukturen (verkettete Listen, wachsende Arrays, Bäume, einfache Hash-Verfahren) erklärt und teilweise mit den objektbasierten Mitteln von Java implementiert. Suchen und Sortieren auf diesen Strukturen werden in ihrer Zeitkomplexität für die Studierenden unmittelbar erfahrbar. Wir erläutern dann passend die O-Notation. Abschließend werden anhand einfacher Graph-Probleme die Vorteile von rekursiven Algorithmen verdeutlicht.

6 Semester 2: Abstraktion – der Weg zu objektorientierter Modellierung

Während wir uns im ersten Semester knapp oberhalb der »Grasnarbe« der Programmierung bewegen, stellen wir im zweiten Semester deutlich höhere Ansprüche an die Abstraktionsfähigkeiten der Studierenden. Wir diskutieren zu Anfang die Korrektheit von Software, untersuchen Strategien zur Fehlerbehandlung mit Exceptions und stellen das Vertragsmodell vor. Anschließend folgen zwei getrennte Themenblöcke: Polymorphie und Implementationsvererbung.

Polymorphie: Subtyping und Generizität

Wir stellen die Taxonomie von Polymorphie nach Cardelli/Wegner [CaW85] vor und diskutieren Merkmale der objektorientierten Typisierung: Subtyping durch Typhierarchien, Probleme mit Ko- und Kontravarianz, Generizität in Java. Hier zahlt sich aus, dass wir den Typbegriff anhand der Interfaces von Java ausführlich in SE1 eingeübt haben. Der Schritt zu Hierarchien von Interfaces fällt so leicht.

Implementationsvererbung

Bewusst spät (ganz im Sinne der Umkehrung des »frühen Vogels«) behandeln wir die Implementationsvererbung als ein Konzept zur inkrementellen Modifikation von Implementationen. Hier erklären wir die bisher nur konsumierte Methodenauswertung beim dynamischen Binden, abstrakte Methoden, das Redefinieren von Methoden, die Konstruktoren-Verkettung in Java und das Konzept der Erbenschnittstelle. All diese Themen sollten Softwaretechniker beherrschen; aber

wir legen großen Wert darauf, dass die Unterscheidung von Subtyping und Implementationsvererbung deutlich wird. Letzere sollte überwiegend konsumiert und nur mit Vorsicht aktiv produziert werden.

Auf der Basis der bisher vorgestellten technischen Grundlagen können objektorientierte Rahmenwerke kompetent genutzt werden. Dies wird am Beispiel der Swing-Bibliothek von Java geübt. Die zweite Hälfte der Lehrveranstaltung behandelt Analyse, Modellierung und Entwurf sowie Refactorings in Form eines kleinen Projekts.

7 Feedback der Studierenden

Jedes Lehrkonzept muss sich an seinem Erfolg messen lassen. Bisher, nach nur einem Durchlauf, haben wir keine belastbaren Zahlen zur Effektivität des neuen Ansatzes. Bisher liegen nur die Ergebnisse der allgemeinen anonymen Studierendenbefragung nach SE1 (regelmäßig, einheitlich und für alle Pflichtveranstaltungen durchgeführt von der studentischen Fragebogen AG) und die Antworten auf unsere gezielte Umfrage nach SE2 vor. Die Ergebnisse der Fragebogen AG für SE2 lagen uns beim Verfassen dieses Beitrags noch nicht vor.

Das Feedback zu SE1 über die Fragebogen AG war sehr positiv. Das Modul hat exzellente Noten bekommen, insbesondere für seine klare Strukturierung (über 80% wählten »sehr gut« oder »gut«). Den Studierenden war bewusst, dass wir für Interfaces einen ungewöhnlichen Weg gewählt haben. Es gab keine Beschwerden über die fehlende Behandlung von Vererbung, und für keinen Studierenden war unsere Verwendung von Interfaces problematisch.

Nach SE2 haben wir per E-Mail eine Umfrage zu unserer Vermittlung von Vererbungskonzepten durchgeführt. Die Umfrage bestand aus den folgenden Aussagen mit den jeweiligen Antworten; die Prozentzahlen geben die Verteilung der gewählten Antworten wieder:

- »Vererbung ist ein wichtiges Thema, das für das volle Verständnis objektorientierter Programmierung zentral ist«:

  Ja, absolut (57%) Ja (33) eher ja (5%) eher nein (5%) gar nicht ( )

- »Mir ist der Unterschied zwischen Subtyping und Implementationsvererbung klar geworden«:

  Ja (71%) ein bisschen (29%) eher nicht ( ) gar nicht ( )

- »Die Unterscheidung hat mir geholfen, das komplizierte Thema Vererbung zu strukturieren«:

  Ja (33%) **eher ja (57%)** eher nicht (5%) nein, gar nicht (5%)

- »Ich finde die Unterscheidung relevant, sie sollte gelehrt werden«:

  Ja (62%) eher ja (38%) eher nicht ( ) nein, gar nicht ( )

- »Ich finde die Reihenfolge der Vermittlung (erst Subtyping, dann Implementationsvererbung) gut«:

  Ja, gut nachvollziehbar (86%) nein, besser andersrum (14%)

- »Ich habe auch an SE1 teilgenommen und halte es für sinnvoll, Interfaces dort zu thematisieren, bevor Vererbung vollständig erklärt wurde, weil Interfaces auch ohne ausführliche Behandlung von Vererbung verstanden werden können:«

  Ja (75%) eher ja (15%) eher nicht (10%) nein, gar nicht ( )

8 Zusammenfassung

In diesem Beitrag haben wir unsere Überlegungen und Entwurfsprinzipien zur Gestaltung des Einstiegs in die Softwareentwicklung im neuen Bachelor-Studiengang Informatik an der Universität Hamburg dargestellt. Das Ergebnis ist eine Kursfolge, in der Objekte vom ersten Tag an behandelt und Schnittstellen und Interfaces früh angesprochen werden. Vererbung wird bewusst erst vollständig im zweiten Semester behandelt, um im ersten Semester grundlegende Algorithmen auf Basis objektbasierter Sprachkonzepte zu verdeutlichen. Die Bewertung des neuen Ansatzes durch die Studierenden ist ermutigend.

Danksagungen

Wir danken allen Kolleginnen und Kollegen im Arbeitsbereich Softwaretechnik an der Universität Hamburg für die exzellente Zusammenarbeit der letzten Jahre, die diesen Ansatz erst ermöglicht hat.

Literatur

[BaK06] Barnes, D., Kölling, M.: *Objects First with Java – A Practical Introduction Using BlueJ (3rd Edition)*, Pearson Education, UK, 2006.

[BBSZ03] Becker-Pechau, P., Bleek, W.-G., Schmolitzky, A., Züllighoven, H.: »Integration Agiler Prozesse in die Softwaretechnik-Ausbildung im Informatik-Grundstudium«, in Siedersleben, J. and Weber-Wulff, D. (Eds.), *Software Engineering im Unterricht der Hochschulen (SEUH),* dpunkt.verlag, S. 8 – 21, 2003.

[BeC04] Bennedsen, J., Caspersen, M. E.: »Programming in context: a model-first approach to CS1«, *Proc. 35th SIGCSE technical symposium on Computer Science Education*, Norfolk, Virginia, USA; ACM Press, S. 477 – 481, 2004.

[Bru04] Bruce, K. B.: »Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list«, *Inroads (newsletter of SIGCSE),* 36:4, S. 29 – 34, 2004.

[CaW85] Cardelli, L., Wegner, P.: »On Understanding Types, Data Abstraction and Polymorphism«, *ACM Computing Surveys,* 17:4, S. 471 – 522, 1985.

[COOL] Nygaard, K.: *COOL – Comprehensive Object-Oriented Learning*, *http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_COOL1.html* (zuletzt besucht am 15.10.2006)

[KQPR03] Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: »The BlueJ system and its pedagogy«, *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology,* 13:4, S. 249 – 268, 2003.

[Mös05] Mössenböck, H.-P.: *Sprechen Sie Java? Eine Einführung in das systematische Programmieren (3. überarb. Auflage)*, dpunkt.verlag, 2005.

[PPP] Bergin, J.: *The Pedagogical Patterns Project*, *http://www.pedagogicalpatterns.org* (zuletzt besucht am 15.10.2006)

[RCS101] Stein, L. A.: *The Rethinking CS101 Project*, *http://www.cs101.org* (zuletzt besucht am 15.10.2006)

[Sch05] Schmolitzky, A.: »Towards Complexity Levels of Object Systems Used in Software Engineering Education (position paper)«, *Proc. Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, ECOOP 2005*, Glasgow, UK, 2005.

[Sch06a] Schmolitzky, A.: »Hochschullehre im Umbruch – Neue Lehrmethoden im softwaretechnischen Anteil des Informatikstudiums«, *LOG IN,* Heft 138/139, S. 48 – 54, 2006.

[Sch06b] Schmolitzky, A.: »Teaching Inheritance Concepts with Java«, *Proc. Principles and Practices of Programming in Java (PPPJ)*, Mannheim, Germany; ACM Press, 2006.

[Web00] Weber-Wulff, D.: »Combating the code warrior: a different sort of programming instruction«, *Proc. 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, Helsinki, Finland; ACM Press, S. 85 – 88, 2000.

[Weg87] Wegner, P.: »Dimensions of Object-Based Language Design«, *Proc. OOPSLA '87*, Orlando, Florida; ACM SIGPLAN Notices, Vol. 22, 12, 1987.

[Weg97] Wegner, P.: »Why Interaction is More Powerful than Algorithms«, *Communications of the ACM,* Vol. 40:5, S. 80 – 91, 1997.