Handout 2 zur informatik-didaktischen Fortbildung von Lehrenden

Dr. Nicole Weicker
Universität Stuttgart
weicker@informatik.uni-stuttgart.de

13. Mai 2005

1 Vergleich von Programmiersprachen

Die folgende Übersicht beruht auf Henning and Vogelsang (2004) und eigenen Erfahrungen. Sie dient einer Orientierung über mögliche Programmiersprachen im Schulunterricht.

1.1 Pascal

Pascal ist in seiner Urform eine prozedurale Programmiersprache wie z.B. C, die jedoch durch die von der Sprachspezifikation erzwungenen Struktur sehr gut lesbar ist. Damit ergibt sich ein leichter Zugang zur Programmierung. Programmierkonzepte werden durch die Strukturierung leicht verständlich. Insgesamt ergibt sich ein niederschwelliger Einstieg in die Programmierung.

Nachteilig ist beim reinen Pascal, dass keine Modularisierung oder Objektorientierung möglich ist.

1.2 Delphi

Delphi ist eine objektorientierte Weiterentwicklung von Pascal und ist im prozeduralen sehr ähnlich zu Pascal. Ein Vorteil von Delphi ist, dass in manchen Entwicklungsumgebungen gerade für den Einstieg in die Objektorientierung durch Fenstergestaltungen schnell sichtbare Ergebnisse erzielt werden können. Damit ist ein leichter Einstieg möglich.

Nachteilig ist, dass die objektorientierte Umsetzung viel anfangs undurchsichtige Syntax benötigt. Diese wird von den Entwicklungsumgebungen so gut wie möglich verborgen. Denoch ist dies eine Fehlerquelle, die für Anfänger schwierig werden kann.

Insgesamt ist die objektorientierte Umsetzung etwas unübersichtlich.

1.3 C/C++

C ist eine Programmiersprache mit einem relativ kleinen SPrachkern. Sie wurde ursprünglich zur Entwicklung harwarenaher Aufgaben konzipiert, hat aber inzwischen in nahezu alle Anwendungsbereiche gefunden.

C hat die folgenden Eigenschaften:

- Compiliert: Der Quelltext muss erst mit Hilfe eines Compilers und Linkers in einen maschinenunabhängigen Code übersetzt werden.
- Prozedural: Zur Strukturierung des Quelltextes können mehrfach genutzte Codefragmente in Funktionen gekapselt werden.
- Typisiert: C arbeitet mit Datentypen, die der Entwickler aber relativ frei in andere Typen konvertieren kann.
- Statisch gebunden: Eine aufzurufende Funktion wird stets zum Zeitpunkt des Linkens ermittelt.
- Unsicher: C kennt direkte Speicherzugriffe, die nicht überwacht werden. Somit ist es leicht möglich, den Speicher zu korrumpieren.

C++ basiert auf einem modifizierten C und wurde um objektorientierte Eigenschaften erweitert. Das Einsatzgebiet ist extrem weit ausgedehnt.

C++ hat die folgenden Eigenschaften:

- Compiliert: s.o.
- Hybrider Ansatz: Es werden neben prozeduralen überwiegend objektorientierte Elemente wie Klassen und deren Methoden zur Strukturierung des Quelltextes eingesetzt.
- Typisiert: s.o.
- Statisch und dynamisch gebunden: Methodenaufrufe werden sowohl zum Zeitpunkt des Linkens als auch zur Laufzeit aufgelöst.
- Unsicher: s.o.

Insgesamt ist C/C++ relativ schwer zugänglich. Es handelt sich um eine „Hackersprache“, mit der viele unsaubere Programmiertechniken möglich sind. Gerade die Zeigerarithmetrik verleitet zu Fehlern. C/C++ ist weit schwieriger zu lesen als Pascal. Statt `begin` und `end` werden { und } verwendet.

1.4 Java

Java ist eine objektorientierte Sprache, die komplett auf prozedurale Elemente verzichtet. Java besitzt eine umfangreiche Klassenbibliothek.

Java hat die folgenden Eigenschaften:

- Interpretiert und compiliert: Der Quelltext muss mit Hilfe eines Compilers n einen plattformunabhängigen Zwischencode übersetzt werden. Dieser wird in einer vierutellen Maschine interpretiert. Darin eingebaut ist ein so genannter Hotspot-Compiler, der kritische Programmteile während der Programmausführung in plattformabhängingen Code übersetzt.
- Objektorientierter Ansatz: Alle Elemente außer primitiven Datentypen sind Objekte. Selbst Klassen sind wiederum Objekte.
- Streng typisiert: Java arbeitet mit Datentypen, die der Entwickler relativ eingeschränkt in andere Typen konvertierten kann. Die Umwandlung wird zur Übersetzungs- und Laufzeit überwacht.
- Statisch und dynamisch gebunden: Methodenaufrufe werden sowohl zum Zeitpunkt des Übersetzens als auch zur Laufzeit aufgelöst.
- Sicher: Java kennt keine direkten Speicher- und Hardwarezugriffe.

Java ist zur Zeit „in“. Frage ist, wie man Java vermitteln will. Es ist möglich, Java anfangs rein prozedural zu betrachten, in dem alles in einer Klasse behandelt wird. Dann ist es allerdings schwierig, später die Objektorientierung als Konzept noch aufzusetzen. Die Syntax in Java ist schnörkellos, allerdings schwieriger zu lesen als in Pascal. Anders als in C++ gibt es keine Zeiger sondern nur Referenzen. Damit wird die unschöne Zeigerarithmetrik unmöglich gemacht.

Es gibt eine Reihe von Entwicklungsumgebungen (z.B. Jbuilder (kommerziell), Eclipse (frei verfügbar)).

2 Programmierlehre

2.1 Didaktische Aspekte der Programmierlehre

Die Fähigkeit, in einer Programmiersprache sicher und schnell programmieren zu können, gehört ebenso wie die Fähigkeit, neue Programmiersprachen schnell erlernen zu können, zum Handwerkzeug eines Informatikers. Im Gegensatz zu vielen anderen Inhalten des Informatikstudiums genügt es hierbei nicht, die grundlegenden Prinzipien verstanden zu haben. Vielmehr können Studierende die pragmatische Fertigkeit guten Programmierens nur durch individuelle, aktive und wiederholte Umsetzung erwerben.

2.1.1 Richtziele der Programmierlehre

Die Ziele der Lehre in diesem Kontext sind vielfältig und anspruchsvoll. Die Studierenden sollen zu einem ▷guten und adäquaten Programmierstil◁ (Z1) erzogen werden, womit sowohl die formale Einhaltung von Programmierrichtlinien als auch eine inhaltliche, algorithmische Geradlinigkeit gemeint ist. Eng verknüpft ist damit auch die Fähigkeit, ▷fremden Code lesen, verstehen und verwenden◁ (Z2) zu können.

Die ersten beiden Richtzielen verdeutlichen wie auch die Ausführungen in Abschnitt 2.1.2, dass die Lernziele nur zu erreichen sind, wenn tatsächlich alle Studierenden ▷selbst programmieren◁ (Z3).

Desweiteren sollen die Studierenden zu einer ▷kritischen Reflexion ihrer eigenen Lösungsansätze◁ (Z4) hingeführt werden, die unter anderem auch zur Aufwandsabschätzung und entsprechender algorithmischer Optimierung zu führen hat. Dieses Hinterfragen der eigenen Arbeit sowie die Bereitschaft, Aufwandsabschätzungen durchzuführen, sollte den Studierenden selbstverständlich werden. Wichtig ist hierbei, den Studierenden zu vermitteln, dass Fehler unvermeidbar sind. Durch eine kritische Analyse der eigenen Arbeit und der Bereitschaft, aus den Fehlern zu lernen, besteht jedoch die Möglichkeit zu einem ▷konstruktiven Umgang sowohl bei der Fehlervermeidung als auch bei der Fehlerbehebung◁ (Z5). Dazu gehört auch die Erziehung zur ▷eigenen Testkultur◁ (Z6), die mit dem Programmieren einhergehen sollte. Die Studierenden sollten lernen, die Verantwortung für die Qualität des eigenen Codes zu übernehmen.

Auch wenn es widersprüchlich erscheint, sollen Studierenden neben der Kompetenz, sich ▷exakt an Vorgaben und verabredete Regeln halten◁ (Z7) zu können, auch zur ▷Kreativität◁ (Z8) angehalten werden. Gerade im Bezug auf algorithmische Optimierungen aber auch für die Füllung von nicht spezifizierten Details ist die kreative Seite gefragt.

Ein Fundament der Programmierlehre ist die Kenntnis von grundlegenden Datenstrukturen wie Bäumen, verketteten Listen, Graphen etc. und dazu passenden Algorithmen sowie der Umgang mit den maßgeblichen Entwurfsstrategien wie Greedy, dynamisches Programmieren, Backtracking etc. Die Studierenden sollen sich mit diesen ▷Grundlagen aktiv auseinander setzen◁ (Z9), um den jeweiligen Nutzen, die Anwendbarkeit und die Vor- bzw. Nachteile aus eigener Erfahrung einschätzen zu können.

Die bisher angeführten Aspekte der Programmierlehre umfassen im Wesentlichen die Punkte, die für ein isoliertes Programmieren im Kleinen notwendig sind. Tatsächlich sind die Aufgaben und Probleme, denen sich ein Informatiker später zu stellen hat, in der Regel anderer Natur. Die ▷Entwicklung und Implementierung von Software im Team◁ (Z10) stellt völlig neue Herausforderungen an die Studierenden und damit auch an die Programmierlehre.

Weitere Lernziele der Programmierlehre bestehen darin, die Lernenden dazu zu erziehen, benutzerfreundliche, sprich selbst erklärende Produkte zu erzeugen (Z11) und Probleme ebenso wie Programme durch Modularisierung und Strukturierung sinnvoll zu gliedern (Z12).

2.1.2 Lerntheoretischer Hintergrund

Aus lerntheoretischer Sicht setzt sich das Programmierenlernen aus kognitiven und pragmatischen Teilen zusammen. Kognitiv ist einerseits die Entwicklung eines Verständnisses für algorithmische Denkweisen und andererseits das Verständnis des Zusammenhangs zwischen der Syntax und der Semantik der zu erlernenden Programmiersprache. Die pragmatischen Aspekte des Programmierenlernens haben mit dem eigenen Codieren zu tun. Dabei ist es wichtig, zunächst für einfache Programme Vorbilder imitieren zu können. Im nächsten Schritt sollten die Studierenden die Möglichkeit bekommen, diese imitierten Programme durch einfache Veränderungen zu manipulieren, um die Auswirkungen ihrer Aktionen durch eigene Erfahrungen kennen zu lernen. Erst viel später werden die Studierenden in der Lage sein, Programme tatsächlich durch Präzisierungen algorithmisch und programmiertechnisch selbst zu verbessern. Den pragmatischen Aspekt kann man sich leicht verdeutlichen, wenn man das Erlernen der Muttersprache als Vergleich heranzieht, welches ebenfalls nicht auf der Basis exakter Grammatikregeln sondern über Imitation geschieht.

Da das pragmatische Lernen ausschließlich über eigene Aktivitäten erfolgt, ist es zwingend notwendig, dass sich jeder Studierende allein mit den Fallstricken und Problemen der zu vermittelnden Programmiersprache auseinandersetzt. Die Vorstellung, in dieser Phase der Programmierlehre durch Gruppenarbeit die Lernziele erreichen zu können, lässt sich vergleichen mit der Annahme, man könne dadurch, dass man einem Tischler zuschaut, erlernen, wie man einen Tisch hobelt.

Bei der Einschätzung, welche Anteile jeweils der pragmatische und der kognitive Aspekt des Programmierenlernens besitzen, ist es wichtig zu unterscheiden, ob die erste oder eine weitere Programmiersprache erlernt werden soll. Tatsächlich werden beim Lernen einer ersten Programmiersprache viele Dinge, wie beispielsweise der Aufbau oder die Formulierung bestimmter Kontrukte (z.B. `for`-Schleife) als Paradigma akzeptiert. Die Entwicklung eines Verständnisses für die Syntax und Semantik der ersten Programmiersprache sowie deren Verinnerlichung kompensieren die kognitiven Kapazitäten der Studierenden.

Die kognitive Auseinandersetzung, die zu einem tieferen Verständnis von dahinterliegenden Programmierkonzepte führt, erfolgt in aller Regel erst beim Erlernen einer weiteren Programmiersprache.

Da die Kompetenz, schnell eine neue Programmiersprache erlernen zu können, ein wesentliches Ziel der Programmierlehre ist, sollten die Studierenden bereits im Studium mit mindestens zwei unterschiedliche Programmiersprachen arbeiten.

2.2 Didaktische Methoden der Programmierlehre

Nach dem Aufzeigen der unterschiedlichen Lernziele innerhalb der Programmierlehre stellt sich die Frage, mit welchen didaktischen Methoden, diese Lernziele angestrebt werden können.

Feedback (M1) Optimalerweise sollte jeder Studierende auf seine Abgaben ein individuelles Feedback erhalten. Das Feedback sollte zeitnah zur Abgabe der Lösung gegeben werden, damit die Studierenden einen direkten Zusammenhang zwischen ihrer eigenen aktiven Arbeit und dem Feedback herstellen können.

Entscheidend für die Motivierung der Studierenden ist, dass in dem Feedback neben den Hinweisen auf Fehler die Punkte gelobt werden, die gut gelungen sind Heckhausen (1974). Beispielsweise kann der Prozentsatz für jeden überprüften Aspekt, inwieweit dieser Aspekt erfüllt wurde, als Zahl oder auch als Diagramm dargestellt werden. Erstrebenswert ist ein Feedback, dass auf den persönlichen Lernfortschritt jedes Einzelnen eingeht. Auch kann für jede Aufgabe ein Ranking in der Form „Sie haben 8 von 10 Punkte erreicht; der Durchschnitt liegt bei 6.3 Punkten.“ mitgeteilt werden.

Nach den Anforderungen an die Form eines Feedbacks stellt sich die Frage, was die Inhalte eines detaillierten Feedbacks auf Programmieraufgaben sein können. Zum einen sollte kommuniziert werden, welche funktionalen Tests der abgegebene Code bestanden hat und welche nicht. So kann gewährleistet werden, dass Grundwissen als Inhalt der Vorlesung tatsächlich verstanden und angewandt werden kann (Z9). Ferner werden Studierende durch das detaillierte Feedback ermuntert, frühzeitig ihre Lösungen selbst mit mehreren Testfällen zu überprüfen – sie können bei weiteren Aufgaben die Arten der Testfälle nachbilden und werden so zur eigenen Testkultur (Z6) erzogen. Ein Nebeneffekt des funktionalen Feedbacks ist zudem die Motivation zum exakten Arbeiten (Z7).

Neben dem Feedback zur Funktionanlität einer abgegebenen Programmierlösung kann es für die Studierenden hilfreich sein, wenn passende Kommentare zu den in dieser Abgabe verwendeten Strukturen gegeben werden. Ein Hinweis darauf, dass die Lösung der Aufgabe am besten mit einer doppelt verketteten Liste gelöst werden kann, die abgegebene Lösung jedoch mit einem Array arbeitet, bringt im Zweifel einen größeren Lernerfolg. Diese Form des Feedbacks zielt auf die Entwicklung eines guten inhaltlichen Programmierstils (Z1) hin.

Um die Studierenden auch zu einem guten formalen Programmierstil (Z1) und zum exakten Einhalten (Z7) der vereinbarten Regeln zu erziehen, sollten im Feedback darüberhinaus Informationen enthalten sein, inwieweit sich der abgegebene Code an vereinbarte Styleguide-Regeln hält. Gut ist es, wenn es möglich ist, mit den Studierenden gemeinsam in einer Diskussion zu erarbeiten, was ein guter Stil ist und welche Regeln für die Programmieraufgaben hinsichtlich des Stils gelten sollen.

Neben all diesen Punkten des individuellen Feedbacks ist es wichtig, den Studierenden deutlich zu machen, dass sie tatsächlich selbst programmieren (Z3) sollen.

Die beiden Ziele (Z11) und (12) werden über ein Feedback auch unterstützt.

Herantasten erlauben (M2) Eine didaktische Methode in der Programmierlehre ist es, den Studierenden durch ein sofortiges Feedback zu ermöglichen, ihre Lösung bis zur endgültigen Abgabe stückweise zu verbessern (vgl. Krinke et al. (2002)). Dadurch werden die Studierenden unterstützt, in der Anwendung von Grundwissen (Z9) Erfahrungen zu sammeln. Sowohl im Hinblick auf das Ziel, die Studierenden zu

einem konstruktiven Umgang mit Fehlern (Z5) zu verhelfen als auch sie zu einem guten formalen Programmierstil (Z1) erziehen zu wollen, kann das Herantasten an eine akzeptierte Lösung hilfreich sein. Die Benutzerfreundlichkeit eines Produktes (Z11) kann über die Möglichkeit des Herantastens auch angenähert werden.

Andererseits kann diese Methode bei den Studierenden die Haltung fördern, Lösungen nicht mehr komplett zu durchdenken sondern das Programm evolutionär bis zur Akzeptanz zu variieren. Diesem Verhalten wird im Praktomat durch die Existenz von geheimen Tests begegnet, die dazu motivieren sollen, nicht nur das offensichtlich durch das gegebene Feedback geforderte Minimum abzugeben.

Korrektur einfordern (M3) Eine kontrollierte Variante, den Studierenden die Möglichkeit und die Motivation zu geben, ihre Abgaben anhand von gegebenem Feedback zu verbessern, ist, den Studierenden das Feedback erst mit Abgabetermin zu geben. Dabei wird dieses Feedback mit der Aufforderung verbunden, bis zu einem nächsten offiziellen Abgabetermin eine korrigierte Version ihrer Lösung abzugeben, um eventuell verfehlte Punkte nachträglich bekommen zu können. Wichtig bei einem solchen Vorgehen ist, dass überprüft wird, dass die verbesserte Lösung tatsächlich von der ursprünglich abgegebenen Lösung abstammt und nicht einfach von dritter Seite übernommen wurde. Neben den Lernzielen (Z1, Z6, Z9 und Z11), die auch beim Herantasten an eine Lösung unterstützt werden, motiviert diese Methode die Studierenden eine eigene Testkultur (Z6) zu entwickeln und fördert das exakte Arbeiten (Z7). Anders als beim schrittweisen Herantasten an eine Lösung werden die Studierenden zu einer kritischen Reflexion (Z4) der eigenen Lösungen animiert. Auch das Erlernen einer geeigneten Modularisierung und Strukturierung (Z12) wird durch eine eingeforderte Korrektur unterstützt.

Wettbewerb (M4) Eine andere Form der Motivierung besteht darin, Wettbewerbe zwischen den abgegebenen Lösung durchzuführen Heckhausen (1974). Insbesondere wenn es um die Optimierung von Algorithmen geht, können die schnellsten Lösungen prämiert oder ein Teil der Punktevergabe anhand eines Schnelligkeitsranking bestimmt werden. Falls die Aufgabe aus der Programmierung eines Spielstrategie (z.B. prisoner's dilemma Fogel (1998)) besteht, können die abgegebenen Lösungen in Form eines Tuniers gegeneinander antreten und so bewertet werden.

Wettbewerbssituationen fordern von den Studierenden Kreativität (Z8), können die Teamarbeit (Z10) positiv beeinflussen Göhner et al. (2005); Lindig and Zeller (2005) und insbesondere auch die Entwicklung einer eigenen Testkultur (Z6) unterstützen. Je nachdem was die Wettbewerbsanforderungen sind, kann auch die Benutzerfreundlichkeit (Z11) gefördert werden.

Unvollständige Anforderungen (M5) Spätestens wenn die Studierenden in der Diplomarbeitsphase auf sich selbst gestellt eine Software entwerfen und realisieren sollen, benötigen sie die Kompetenz, kreativ (Z8) mit unvollständigen Anforderungen umgehen zu können. Aus diesem Grunde ist eine didaktische Methode, sie mit unvollständigen Anforderungen in Programmieraufgaben zu konfrontieren. Beispiels-

weise kann die Ein- und Ausgabe oder die zu verwendende Datenstruktur offen gelassen werden. Möglich ist auch, Optionen zur Präzisierung der Lösung einer Aufgabe zu lassen. Bei allen diesen Punkten ist es wichtig, dass die Studierenden dazu angehalten werden, über die bestehenden Freiheitsgrade kritisch zu reflektieren (Z4), um diese bewusst nutzen zu können. Im Zusammenhang mit Softwarepraktika kann diese Methode auch der Förderung der Teamarbeit (Z10) dienen, da sich die Gruppe zu einigen hat, wie sie die unvollständigen Anforderungen füllen will. Diese Art der Aufgabenstellung fordert eine eigene Modularisierung und Strukturierung seitens der Lernenden (Z12).

Aufeinander aufbauende Aufgaben (M6) Für fortgeschrittene Studierende ist es denkbar, die unterschiedlichen Programmieraufgaben aufeinander aufbauend zu gestalten. Beispielsweise kann in einer ersten Aufgabe eine Klasse in Java zu schreiben sein, die entweder gemäß des Feedbacks korrigiert oder auch nicht korrigiert und damit fehlerhaft in einer späteren Aufgabe ergänzt und mit weiteren Klassen in Zusammenhang gebracht werden soll. Eine didaktische Methode bei einem solchen Vorgehen besteht darin, die Lösungen der ersten Aufgabe unter den Studierenden so zu verteilen, dass jeder die Lösung eines anderen weiterzuentwickeln hat. Gerade wenn fehlerhafter Code weiterzuentwickeln ist, wird die Wichtigkeit einer eigenen Testkultur (Z6), eines guter formaler wie inhaltlicher Programmierstil (Z1) sowie der Exaktheit (Z7) im Arbeiten besonders deutlich. Zusätzlich hilft ein solches Vorgehen, die Studierenden zu einem konstruktiven Umgang mit Fehlern (Z5) zu erziehen. Weitere Lernziele, die mit dieser Methode unterstützt werden, ist die Fähigkeit, fremden Code zu nutzen (Z2) sowie eigene Lösungen kritisch zu reflektieren (Z4). Bei größeren Aufgaben können auch Teams benutzt werden (Z10). Durch die Notwendigkeit mit der Lösung eines anderen weiterzuarbeiten, wird eine Bewusstsein auf die Auswirkungen von guter bzw. weniger geschickteren Modularisierung und Strukturierung verdeutlicht, da diese Auswirkungen erfahrbar werden (Z12) .

Gegenseitig begutachten lassen (Reviews) (M7) Der geschriebene Code wird von anderen Schülern/Studierenden gegengelesen und begutachtet. Wichtig dabei ist, Richtlinien für die Gutachten aufzustellen, die die Punkte enthalten, auf die die Reviewer achten sollen. Dabei können die folgenden Lernziele unterstützt werden: (Z1), (Z2), (Z3), (Z4), (Z5), (Z6) und (Z7).

Spielerische Anwendung (M8) Über den spielerischen Zugang zur Programmierung wird eine besondere Form der Motivation erreicht. Zusätzlich können die folgenden Lernziele unterstützt werden: (Z3), (Z4), (Z6), (Z8) und (Z11).

Nachprogrammieren (imitieren) lassen (M9) Gerade zum Beginn der Programmierlehre ist es wichtig, den Lernenden Vorbilder bereitzustellen, die sie imitieren können. Die folgenden Lernziele werden dadurch angesprochen: (Z1), (Z2), (Z5) und (Z7).

Gegenseitige Schulung (M10) Die Idee der gegenseitigen Vermittlung kann einerseits umgesetzt werden, damit die Schüler/Studierenden, die bereits programmieren können, denen helfen, die es erst lernen. Andererseits ist es in späteren Phasen der Programmierlehre möglich, Experten zu verschiedenen Aufgaben zu bestimmen, die sich in ihr Thema einarbeiten und später die anderen darin unterrichten (vgl. Gruppenpuzzle). Die Lernziele, die durch die Methode gefördert werden, sind: (Z1), (Z2), (Z3), (Z4), (Z5) und (Z7).

Handbuch zu einer Programmiersprache verfassen lassen (M11) Die Idee hinter diese Methode ist die, dass man in aller Regel eine neue Sache erst dann wirklich verstanden hat, wenn man in der Lage ist, sie jemand anderem zu erklären. Die Aufgabe besteht darin, die wichtigsten Eigenschaften und Besonderheiten einer Programmiersprache so zu erarbeiten, dass sie schriftlich festgehalten einem Dritten beim Erlernen der Sprache hilfreich sein können. Insbesondere die Lernziele (Z7), (Z8), (Z9), (Z10), (Z11) und (Z12) werden dabei erreicht.

Dokumentation des erstellten Programms (M12) Über die Anforderung eine Dokumentation zu einem selbst geschriebenen Programm zu erstellen, lernen die Schüler/Studierenden die kritische Reflexion der eigenen Arbeit (Z4) und sind aufgefordert, tatsächlich selbst zu programmieren (Z3). Das exakte Arbeiten wird dabei gefördert (Z7) und sie lernen ihre Programme sinnvoll zu strukturieren, damit sie sie in ihrer Dokumentation beschreiben können (Z12). Schließlich hilft die Dokumentation sich über die Benutzerfreundlichkeit des eigenen Produkts Gedanken zu machen (Z11).

Fertigen fremden Code kommentieren lassen (M13) Zu Beginn der Programmierlehre ist es wichtig, dass die Lernenden sich mit dem Codebild der zu lernenden Programmiersprache auseinander setzen. Wenn sie fertigen fremden Code kommentieren sollen, haben sie die Strukturen zu erkennen und geeignet zuzuordnen (Z12). Sie lernen dabei, selbst zu kommentieren, und lernen durch die Vorbildfunktion einen guten Programmierstil kennen (sofern der vorgesetzte Code in einem guten Stil verfasst ist) (Z1). Sie lernen fremden Code verstehen (Z2).

Literatur

Rüdiger Baumann. *Didaktik der Informatik*. Klett, Stuttgart, 1996.

Franz Eberle. *Didaktik der Informatik bzw. einer informationstechnologischen und kommunikationstechnologischen Bildung auf der Sekunda*. Sauerländer GmbH Verlag, Aarau, 1996.

David B. Fogel. The iterated prisoner's dilemma. In David B. Fogel, editor, *Evolutionary Computation: the fossil record*, pages 541–543. IEEE Press, Piscataway, NJ, 1998.

↓ Ziele Methoden →	Feedback (M1)	Herantasten erlaubt (M2)	Korrektur einfordern (M3)	Wettbewerb (M4)	Unvollständige Anford. (M5)	Aufeinander aufbauend (M6)	Reviews (M7)	Spielerisches (M8)	Imitieren (M9)	Gegenseitige Schulung (M10)	Handbuch zur Sprache (M11)	Dokumentation (M12)	Kommentieren (M13)
Programmierstil (Z1)	X	X	X			X	X		X	X			X
Reuse von Code (Z2)						X	X		X	X			X
Selbst programmieren (Z3)	X						X	X		X		X	
Kritische Reflektion (Z4)			X		X	X	X	X		X		X	
Umgang mit Fehlern (Z5)		X	X			X	X		X	X			
Testkultur (Z6)	X		X	X		X	X	X					
Exaktes Arbeiten (Z7)	X		X			X	X		X		X	X	
Kreativität (Z8)				X	X			X			X		
Grundwissen anwenden (Z9)	X	X	X								X		
Teamarbeit (Z10)				X	X	X				X	X		
Benutzerfreundlichkeit (Z11)	X	X	X	X				X			X	X	
Strukturierung (Z12)	X		X	X	X	X					X	X	X

Peter Göhner, Friedemann Bitsch, and Hisham Mubarak. Softwarerechnik live – im Praktikum zur Projekterfahrung. In Klaus-Peter Löhr and Horst Lichter, editors, *Software Engineering im Unterricht an Hochschulen*, pages 41–55, Heidelberg, 2005. dpunkt.verlag. SEUH 9.

H. Heckhausen. Motive und ihre Entstehung (Kap. 3). Einflußfaktoren der Motiventwicklung (Kap. 4). Bessere Lernmotivation und neue Lernziele (Kap. 18). *Funkkolleg Pädagogische Psychologie*, 1, 1974.

Peter A. Henning and Holger Vogelsang. *Taschenbuch Programmiersprachen.* Fach-

buchverlag Leipzig, Leipzig, 2004.

Peter Hubwieser. *Didaktik der Informatik: Grundlagen, Konzepte, Beispiele*. Springer, Berlin, 2000.

Jens Krinke, Maximilian Störzer, and Andreas Zeller. Web-basierte Programmierpraktika mit Praktomat. *Softwaretechnik-Trends*, 22(3):51–53, 2002.

Christian Lindig and Andreas Zeller. Ein Softwaretechnik-Praktikum als Sommerkurs. In Klaus-Peter Löhr and Horst Lichter, editors, *Software Engineering im Unterricht an Hochschulen*, pages 68–80, Heidelberg, 2005. dpunkt.verlag. SEUH 9.

Sigrid Schubert and Andreas Schwill. *Didaktik der Informatik*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2004.