Didaktische Anmerkungen zur Unterstützung der Programmierlehre durch E-Learning

Nicole Weicker[1] Karsten Weicker[2]

[1] Institut für Formale Methoden der Informatik, Universität Stuttgart, weicker@fmi.uni-stuttgart.de

[2] Fachbereich Informatik, Mathematik und Naturwissenschaften, HTWK Leipzig, weicker@imn.htwk-leipzig.de

Abstract: Diese Arbeit verknüpft Lernziele, didaktische Methoden und Techniken zur Bearbeitung und Bewertung von Programmieraufgaben in E-Learning-Plattformen. Das Ziel ist dabei, sowohl eine Bewertungsgrundlage für den Einsatz einer Plattform für beliebige Lehrveranstaltungen der Programmierlehre zu schaffen als auch ein Gesamtkonzept für eine idealisierte E-Learning-Anwendung zu präsentieren. Dabei steht bewusst die Kompetenzbildung der Studierenden im Zentrum der Überlegungen – die dafür benötigte technische Unterstützung wird aus den didaktischen Methoden zur Vermittlung der Kompetenzen abgeleitet.

1 Motivation

Programmieren im weitesten Sinne ist ein wichtiger Bestandteil der Informatikausbildung an Hochschulen, da dies letztendlich das Bindeglied zwischen den zumeist abstrakten Informatikinhalten und deren konkreter Durchführung auf Datenverarbeitungsmaschinen ist. Daher gibt es auch seit den Anfängen des E-Learning Bestrebungen, das neue Medium positiv für die Programmierlehre zu nutzen. Beispiele sind Plattformen wie Praktomat [Ze00, KSZ02], Webassign [BKW03] oder EClaus [BEW04], aber auch die Integration von Programmierübungen in E-Learning-Angebote [EFGW03, BD03].

Gründe für den E-Learning-Einsatz sind das Ausloten der technischen Machbarkeiten, die Reaktion auf veränderte Randbedingungen wie steigende Studierendenzahlen oder weniger verfügbare finanzielle Mittel, die Verbesserung von organisatorische Umgestaltung von Abläufen etwa in der Fernlehre oder die Verbesserung der Ausbildung der Studierenden.

Um beurteilen zu können, inwieweit die E-Learning-Plattformen dem letzten Punkt gerecht werden, bietet der vorliegende Artikel eine umfassende Analyse der didaktischen Ziele und Methoden in der Programmierlehre. Über die möglichen Techniken zur Umsetzung der didaktischen Methoden in einem E-Learning-System werden ferner idealisierte Prototypen zur Unterstützung der verschiedenen Veranstaltungsformen skizziert.

2 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 [Ho98].

2.1 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 [EFGW03], 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 [EFGW03, Ze00].

Die ersten beiden Richtziele verdeutlichen wie auch die Ausführungen in Abschnitt 2.2, 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) [BD03] 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) [Ho98]. Dazu gehört auch die Erziehung zur ▷eigenen Testkultur◁ (Z6), die mit dem Programmieren einhergehen sollte [Ze99]. 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) [Ze99] zu können, auch zur ▷Kreativität◁ (Z8) angehalten werden (siehe z.B. im Studienplan [WW02]). 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) [SG05, FSN05] stellt völlig neue Herausforderungen an die Studierenden und damit auch an die Programmierlehre.

2.2 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 [Me91], ist es zwingend notwendig, dass sich jeder Studierende selbst mit den Fallstricken und Problemen der zu vermittelnden Programmiersprache auseinandersetzt. Eine arbeitsteilende Herangehensweise in einer Gruppe kann dazu führen, dass bestimmte Fertigkeiten nicht gelernt werden – wie auch etwa ein Tischler alle Arbeitsschritte selbst üben muss.

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 Konstrukte (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.

3 Konkrete Lernziele verschiedener Veranstaltungsformen

Programmierlehre umfasst, wie in Abschnitt 2.1 beschrieben, weit mehr als das reine Erlernen der Syntax und Semantik einer Programmiersprache. Aus diesem Grunde ist die Programmierlehre in verschiedenen Lehrveranstaltungen verankert. Während sich Programmierkurse und Programmier- oder Softwarepraktika im Wesentlichen den pragmatischen Aspekten der Programmierlehre widmen, dienen auch die einführende Veranstaltungen in den ersten beiden Semestern eines Studiums („Grundlagen der Informatik“ im ersten und „Algorithmen und Datenstrukturen“ im zweiten Semester oder „Einführung in die Informatik I und II“ oder „Praktische Informatik I und II“) den kognitiven und durch begleitende Übungen den pragmatischen Aspekten der Programmierlehre. Prototypisch wurden hier einige klassische Veranstaltungen gewählt; bei einer anderen curricularen Gestaltung können die Lernziele in anderen Zusammensetzungen auftreten.

Programmierkurse Um eigene Erfahrungen mit der Programmierung zu sammeln (Z3, Z9) setzen die Studierenden im praxisbetonten Programmierkurs das Grundwissen handelnd um, das sie in der begleitenden Vorlesung kognitiv erworben haben. Besonders wichtig ist dabei, dass die Studierenden sich mit den Details der Syntax und Semantik der Programmiersprache in Anwendungen der Grundkonstrukte und Kontrollstrukturen auseinandersetzen. In dieser Phase der Programmierlehre sollen zusätzlich Grundlagen für die Entwicklung eines guten Programmierstils (Z1), den konstruktiven Umgang mit Fehlern (Z5) sowie ein exaktes Arbeiten (Z7) gelegt werden. Ebenso wichtig ist es, die Studierenden bereits zu diesem Zeitpunkt zur Entwicklung einer eigenen Testkultur zu erziehen (Z6). Am Ende des Programmierkurses sollen die Studierenden in der Lage sein, isoliertes Programmieren im Kleinen in der entsprechenden Programmiersprache bewältigen zu können.

„Einführung in die Informatik I“ In der Vorlesung „Einführung in die Informatik I“ werden die kognitiven Grundlagen für ein Verständnis der Programmierlehre gelegt werden, indem Programmierparadigmen gelehrt, die im praktischen Teil des Programmierkurses konkret umgesetzt werden.

Die Studierenden sollen lernen, wie ein Algorithmus aufgebaut ist, welche Grundkonstrukte und Kontrollstrukturen es in der zu lernenden Programmiersprache gibt, und wie Syntax und Semantik in dieser Sprache zusammenhängen. D.h. die Studierenden sollen die spezielle algorithmische Denkweise der zu lernenden Programmiersprache erfassen. Am Ende dieser Veranstaltung sollen sie in der Lage sein, einfache Laufzeitabschätzungen vorzunehmen, um so ein Hilfsmittel an der Hand zu haben, mit dem sie ihren eigenen Code beurteilen können (Z4).

„Einführung in die Informatik II“ Die Studierenden sollen in dieser Vorlesung die grundlegenden Datenstrukturen (z.B. Bäume, Listen, Keller, Graphen etc.) und Standardalgorithmen (z.B. Such- und Sortieralgorithmen etc.) ebenso wie die wesentlichen algorithmischen Entwurfsstrategien kennen und verstehen lernen. Durch begleitende Programmieraufgaben soll dieses Basiswissen der Informatik auch praktisch umgesetzt werden (Z9). Ein Lernziel dieser Programmieraufgaben ist, dass die Studierenden in der Lage

sind, eine vorgestellte Idee (z.B. eines Standardalgorithmus) in einer Programmiersprache zu implementieren (Z3). Zu diesem Zweck ist die grundsätzliche Idee am Beispiel nachzuvollziehen, es sind geeignete Datenstrukturen auszuwählen und bei der Implementierung einer möglichst effizienten Lösung sind zusätzlich Randfälle und Ausnahmen zu beachten (Z4, Z8). Um diese Aufgaben bewältigen zu können, ist es notwendig, dass die Studierenden Abstraktionsvermögen und ein tiefergehendes Verständnis für die Programmierung entwickeln. Spätestens am Ende der Vorlesung sollen die Studierenden ein Gefühl dafür bekommen haben, wann welche Datenstruktur und welcher algorithmische Ansatz angebracht sein kann.

„Software- oder Programmierpraktikum“ Im dritten oder vierten Semester findet im Rahmen der Programmierlehre in der Regel ein Praktikum statt, in dem Teams von 3 bis 6 Studierenden mittelgroße und damit noch überschaubare Aufgaben lösen (Z10). Ein entscheidendes Lernziel dieser Praktika ist es, durch die Zusammenarbeit und den Austausch mit anderen Studierenden zu einem guten Programmierstil zu finden (Z1), der von anderen leicht gelesen und verstanden werden kann. Umgekehrt sollen die Studierenden lernen, fremden Code zu lesen und zu nutzen (Z2). Im Rahmen des Softwarepraktikums sollen die Studierenden erste praktische Erfahrungen mit einem eigenen Projektmanagement sammeln und das Konzept der Modularisierung, eventuell der objektorientierten Analyse und des objektorientierten Entwurfs anwenden.

4 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 [SG05].

Entscheidend für die Motivierung der Studierenden ist, dass in dem Feedback neben den Hinweisen auf Fehler die Punkte gelobt werden, die gut gelungen sind [He74]. 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 Funktionalität 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.

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. [KSZ02]). 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.

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 des Herantastens und des Lernens aus Fehlern besteht darin, bei fehlerhaften Abgaben eine Korrektur einzufordern. Den Studierenden wird das Feedback erst nach Ablauf einer Abgabefrist gegeben und sie haben die Pflicht, eine korrigierte Version ihrer Lösung einzureichen. 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), 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.

Wettbewerb (M4) Eine andere Form der Motivierung besteht darin, Wettbewerbe zwischen den abgegebenen Lösung durchzuführen [He74, SG05]. Insbesondere wenn es um die Optimierung von Algorithmen geht, können die schnellsten Lösungen prämiert oder ein Teil der Punktevergabe anhand eines Schnelligkeitsranking bzw. Software-Metriken (*lines of code*) bestimmt werden. Falls die Aufgabe aus der Programmierung eines Spielstrategie (z.B. prisoner's dilemma [Ax87]) besteht, können die abgegebenen Lösungen in Form eines Turniers gegeneinander antreten und so bewertet werden.

Wettbewerbssituationen fordern von den Studierenden Kreativität (Z8), können die Teamarbeit (Z10) positiv beeinflussen [GBM05, LZ05] und insbesondere auch die Entwicklung einer eigenen Testkultur (Z6) unterstützen.

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. Beispielsweise 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. Ein Beispiel wäre die Entwicklung eines Spiels, bei dem zwar grobe Vorgaben vorliegen, aber Feinheiten selbst geklärt werden müssen. 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.

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 weiterentwickeln muss. Gerade bei fehlerhaftem Code wird die Wichtigkeit einer eigenen Testkultur (Z6), eines guten formalen wie inhaltlichen Programmierstils (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).

5 E-Learning Techniken zur Unterstützung der Programmierlehre

Da die Programmierlehre in aller Regel in den ersten vier Semestern des Grundstudiums angesiedelt ist und in diesen Semestern durch hohe Anfängerzahlen sowie den Exportcharakter der Grundstudiumsveranstaltungen sehr viele Studierenden zu betreuen sind, ist eine didaktisch fundierte Betreuung jedes Einzelnen nur durch einen ausgefeilten Einsatz von E-Learning-Komponenten möglich.

Blackbox-Tests (T1) Die Funktionalität von Programmabgaben anhand von Testfällen zu überprüfen, gehört zum Standard in allen gängigen Plattformen. Die Technik kommt überall dort zum Einsatz, wo eine objektive oder automatisiert feststellbare Korrektheit benötigt wird. Allerdings müssen die Aufgaben hierfür sehr detailliert beschrieben und auch von den Studierenden präzise bearbeitet werden. So wird entweder die Ein- und Ausgabe genau spezifiziert oder eine Schnittstelle für eine Komponente/Klasse wird vorgegeben.

Glassbox-Tests (T2) Kontrollflussanalysen können benutzt werden, um automatisiert genauere Informationen über die innere Struktur eines Programms zu extrahieren. Grundsätzlich ist ein solcher Ansatz in [JHS02] vorgestellt. Für den Einsatz im E-Learning kann dadurch beispielsweise in allen Testfällen ungenutzter Programmcode detektiert werden, was Rückschlüsse auf den inhaltlichen Programmierstil erlaubt oder ein Hinweis auf ein vertuschtes Plagiat sein kann. Für die Durchführung dieser Tests gilt dieselbe Einschränkung wie für die Blackbox-Tests. Sie sind auch nur bei kleinen Aufgaben anwendbar.

Performance-Tests (T3) Falls die Qualität einer Implementation von Interesse ist, kann ein Laufzeittest ein möglicher Indikator sein [Ro05]. Dies ist einerseits bei Wettbewerbssituation eine mögliche Realisierung, kann andererseits jedoch auch auch eine Prüfungsmöglichkeit hinsichtlich der sinnvollen Anwendung von Algorithmen sein.

Styleguide-Kontrolle (T4) Die Kontrolle von Programmierrichtlinien ist beispielsweise für Java leicht realisierbar [KSZ02]. Wünschenswert ist dabei jedoch eine leichte Konfigurierbarkeit sowie die Unterstützung mehrerer Programmiersprachen. Meist werden auch nur Fehler angezeigt, wo auch ein positives Feedback erwünschenswert ist.

Plagiatstest (T5) Auch für die Detektion von Plagiaten stehen viele Werkzeuge zur Verfügung, die auch leicht in E-Learning-Plattformen integrierbar sind [KSZ02]. Die Anwendungsbereiche der Technik sind mannigfaltig. Über das Ähnlichkeitsmaß des Plagiatstests kann sowohl die Ähnlichkeit als auch die Verschiedenheit von Abgaben festgestellt werden. Neben der Abschreckung gegen „Abschreiben“ sind auch die folgenden Anwendungsmöglichkeiten denkbar. Ersteres kann beispielsweise auch dafür benutzt werden, um sicherzustellen, dass tatsächlich ein bestehendes Programm verbessert wurde. Mit Zweiterem können den Studierenden möglichst andersartige Programme ihrer Kommilitonen zum Review oder zur Verwendung/Überarbeitung zugewiesen werden.

↓ Ziele Methoden →	Feedback (M1)	Herantasten erlaubt (M2)	Korrektur einfordern (M3)	Wettbewerb (M4)	Unvollständige Anford. (M5)	Aufeinander aufbauend (M6)
Programmierstil (Z1)	T2 T4 T8	T4	T2 T4 T5			T8 T10
Reuse von Code (Z2)						T1 T5 T10
Selbst programmieren (Z3)	T5 T9					
Kritische Reflektion (Z4)			T1 T2 T3		T8	T8 T10
Umgang mit Fehlern (Z5)		T1 T2 T4	T1 T2 T4			T10
Testkultur (Z6)	T1		T1	T6 T7		T1 T5 T10
Exaktes Arbeiten (Z7)	T1 T4		T1 T4			T10
Kreativität (Z8)				T3 T6	T8	
Grundwissen anwenden (Z9)	T1 T2 T3	T1 T2 T3	T1 T2 T3 T5			
Teamarbeit (Z10)				T3 T6	T8 T11	T8 T10

Legende:
T1 Blackbox-Test
T2 Glassbox-Test
T3 Performance-Test
T4 Styleguide-Kontrolle
T5 Plagiatstest
T6 Wettbewerbsfunktionalität
T7 Ranking-Funktionalität
T8 Review-Funktionalität
T9 Code-Fragmente
T10 Basis-CSCW-Funktionalität
T11 Kommunikations-Funktionalität

Tabelle 1: Übersicht über die Verzahnung der Richtziele mit den didaktischen Methoden sowie die mögliche technische Unterstützung im E-Learning.

Wettbewerbsfunktionalität (T6) Für eine höhere Motivation der Studierenden kann ein Wettbewerbscharakter hilfreich sein. Für einen spielerischen Wettbewerb müssen innerhalb der Software Möglichkeiten zur Verfügung stehen, dass Programmabgaben von verschiedenen Studenten gegeneinander antreten und so miteinander interagieren. Dabei sind aus den Ergebnissen solcher Wettkämpfe Bewertungen der einzelnen Studierenden abzuleiten, z.B. indem die einzelnen Wettkämpfe als Turnier angeordnet werden.

Ranking-Funktionalität (T7) Auch die Ranking-Funktionalität setzt Programmabgaben der Studenten zueinander in Beziehung, nutzt jedoch keinen direkten interaktiven Vergleich. Stattdessen wird eine Rangliste aufgrund von Performance-, Blackbox-Tests oder Software-Metriken erstellt und daraus evtl. eine Punktevergabe abgeleitet.

Basis-CSCW-Funktionalität (T10) Im klassischen Szenario der Programmierlehre werden Programmabgaben nur abgegeben und vom System bewertet. Sollen jedoch auch andere Datenflüsse möglich sein, ist eine gewisse Basis-CSCW-Funktionalität bereitzustellen. Ein mögliches Szenario ist die Weiterleitung einer Abgabe an andere Studierende zur weiteren Bearbeitung, Benutzung oder Begutachtung. Im letzten Fall ist zusätzlich der Rückfluss der Abgabe und des Gutachtens an den ursprünglichen Autoren zu ermöglichen [Ze00]. Ein anderes Szenario ist die tatsächliche Teamarbeit, bei der mehrere Studierende einen gemeinsamen Abgaberaum haben.

Review-Funktionalität (T8) Die allgemeinste Form, Begutachtungen durch andere Studierende im System zu verwalten, besteht darin, Texte oder Dokumente an Programmierabgaben anheften zu können, die dann abhängig vom Szenario für andere sichtbar sind. Dies kann einerseits einem Peer-Review dienen, aber auch essentiell für die Teamarbeit sein [Ze00].

Code-Fragemente (T9) Gerade beim ersten Erlernen einer Programmiersprache ist es für Studierende oft schwierig, sich auf die Funktionsweise der eigentlich zu lernenden Kontrollstrukturen (z.B. `while`, `for`, `if` etc.) zu konzentrieren, wenn wie in Java zusätzlich Klassen, allgemeine Deklarationen und einzuladende Pakete beachtet werden wollen. In [BD03] wurde bereits ein System vorgestellt, mit dem Codefragmente ausführbar sind, so dass die Studierenden in der Lage sind, sich bei dem ersten Erlernen des Programmierens auf die wesentlichen Inhalte zu konzentrieren.

Kommunikations-Funktionalität (T11) Ist die Zusammenarbeit von mehreren Personen an einem Projekt in der E-Learning-Plattform gefragt, sind zusätzliche Kommunikationsmöglichkeiten gefragt wie Foren, Chat oder der Versand von E-Mails. Gerade für die Teamarbeit kann man sich auch spezielle Funktionen vorstellen, die bestimmte Teamprozesse unterstützen, wie etwa die Erstellung eines Teamprofils (siehe [FSN05]), was insbesondere bei einer großen räumlichen Distanz zwischen den einzelnen Studierenden nützlich ist.

Bei all denen Techniken, die ein Feedback an die Studierenden ermöglichen, ist es wichtig, dass dieses Feedback leicht für den Lernenden zugreifbar ist und in jedem Fall ein Hinweis auf diese Rückmeldung an prominenter Stelle erfolgt (eventuell eine kurze Zusammenfassung direkt nach dem Einloggen). Ferner sollte insbesondere bei automatisch erzeugtem Feedback auf eine leichte Lesbarkeit (evtl. über vorgefertigte Texte) geachtet werden. Auch wäre bei konkreten Details in der Programmierlösung ein leicht nachvollziehbarer Verweis auf oder ein direktes Markieren im Programmtext wünschenswert.

6 Ausblick

Der Beitrag dieses Artikels besteht in erster Linie in der Verknüpfung der möglichen Lernziele der Programmierausbildung mit den didaktischen Methoden und möglichen Techniken für eine entsprechende Unterstützung im Rahmen einer E-Learning-Plattform. Dies soll einerseits einer schärferen Abgrenzung der unterschiedlichen Plattformen dienen, andererseits aber auch zu Programmen führen, die tatsächlich besser an die jeweiligen Bedürfnisse der Lehre angepasst sind.

Derzeit wird die Plattform EClaus der Universität Stuttgart um die automatische Korrektur von Programmieraufgaben erweitert. Dabei wurde das obige Konzept als Leitbild herangezogen. Zunächst beschränken wir uns zunächst auf die Standardtechniken Plagiatsuche [Kö05], Blackbox-Tests und Überprüfung von Programmierrichtlinien [BE05], wobei alle diese Funktionen weitestgehend programmiersprachenunabhängig umgesetzt und direkt in EClaus integriert werden. Davon versprechen wir uns, dass beispielsweise die Plagiatsuche auch tatsächlich einfach und ohne großen Overhead im Weiteren etwa für die Unterstützung/Überprüfung des Code-Reuse genutzt werden kann.

Die praktischen Anwendung obiger Ergebnisse hat für die Autoren umso mehr unterstrichen, dass der Abgleich zwischen den Lernzielen und der didaktischen Methoden mit den technischen Möglichkeiten sehr schwierig ist und in der Regel unterbleibt. Diese Arbeit hat uns jedoch ein Mittel in die Hand gegeben, mit dem wir bestehende Plattformen für bestimmte Lehrveranstaltungen und unsere individuelle Lehrziele bewerten bzw. die eigene Plattform gezielt ausbauen können.

Literatur

[Ax87] Axelrod, R.: The evolution of strategies in the iterated prisoner's dilemma. In: Davis, L. (Hrsg.), *Genetic Algorithms in Simulated Annealing*. S. 32–41. Pitman. London. 1987.

[BD03] Bieg, C. und Diehl, S.: Entdeckendes Lernen mit einem interaktiven Online-Tutorium zur Programmierung in Java. In: Bode, A., Desel, J., Rathmayer, S., und Wessner, M. (Hrsg.), *DeLFI 2003: Die 1. e-Learning Fachtagung Informatik*. S. 154–162. Bonn. 2003. Gesellschaft für Informatik.

[BE05] Behringer, F. und Engeldinger, D.: Automatische Überprüfung von Programmierabgaben im universitären Übungsbetrieb. Diplomarbeit. FMI, Universität Stuttgart. 2005.

[BEW04] Behringer, F., Engeldinger, D., und Weicker, K.: Web-basierte Administration des Übungsbetriebs mit EClaus. In: Engels, G. und Seehusen, S. (Hrsg.), *DeLFI 2004: Die 2. e-Learning Fachtagung Informatik*. S. 79–90. Bonn. 2004. Gesellschaft für Informatik.

[BKW03] Beierle, C., Kulaš, M., und Widera, M.: Automatic analysis of programming assignments. In: Bode, A., Desel, J., Rathmayer, S., und Wessner, M. (Hrsg.), *DeLFI 2003: Die 1. e-Learning Fachtagung Informatik*. S. 144–153. Bonn. 2003. Gesellschaft für Informatik.

[EFGW03] Eichelberger, H., Fischer, G., Grupp, F., und Wolff von Gudenberg, J.: Programmierausbildung online. In: Bode, A., Desel, J., Rathmayer, S., und Wessner, M. (Hrsg.), *DeLFI 2003: Die 1. e-Learning Fachtagung Informatik*. S. 134–143. Bonn. 2003. Gesellschaft für Informatik.

[FSN05] Fleischmann, A., Spies, K., und Neumeyer, K.: Teamtraining für Software-Ingenieure. In: Löhr, K.-P. und Lichter, H. (Hrsg.), *Software Engineering im Unterricht an Hochschulen*. S. 26–40. Heidelberg. 2005. dpunkt.verlag. SEUH 9.

[GBM05] Göhner, P., Bitsch, F., und Mubarak, H.: Softwarerechnik live – im Praktikum zur Projekterfahrung. In: Löhr, K.-P. und Lichter, H. (Hrsg.), *Software Engineering im Unterricht an Hochschulen*. S. 41–55. Heidelberg. 2005. dpunkt.verlag. SEUH 9.

[He74] Heckhausen, H.: Motive und ihre Entstehung (Kap. 3). Einflußfaktoren der Motiventwicklung (Kap. 4). Bessere Lernmotivation und neue Lernziele (Kap. 18). *Funkkolleg Pädagogische Psychologie*. 1. 1974.

[Ho98] Hornecker, E.: Programmieren als Handwerkszeug im ersten Semester. In: Claus, V. (Hrsg.), *Informatik und Ausbildung*. S. 43–51. Berlin. 1998. GI. Springer.

[JHS02] Jones, J. A., Harrold, M. J., und Stasko, J.: Visualization of test information to assist fault localization. In: *Proc. of the 24th International Conference on Software Engineering*. S. 467–477. 2002.

[Kö05] König, T.: Erstellung einer in das System eClaus integrierten Plagiaterkennung für abgegebene Programme. Diplomarbeit. FMI, Universität Stuttgart. 2005.

[KSZ02] Krinke, J., Störzer, M., und Zeller, A.: Web-basierte Programmierpraktika mit Praktomat. *Softwaretechnik-Trends*. 22(3):51–53. 2002.

[LZ05] Lindig, C. und Zeller, A.: Ein Softwaretechnik-Praktikum als Sommerkurs. In: Löhr, K.-P. und Lichter, H. (Hrsg.), *Software Engineering im Unterricht an Hochschulen.* S. 68–80. Heidelberg. 2005. dpunkt.verlag. SEUH 9.

[Me91] Meyer, H. L.: *Trainingsprogramm zur Lernzielanalyse*. Beltz. Weinheim. 1991.

[Ro05] Rost, S.: Entwurf eines verteilten Software-Test-Frameworks. Diplomarbeit. IMN, HTWK Leipzig. 2005.

[SG05] Stoyan, R. und Glinz, M.: Methoden und Techniken zum Erreichen didaktischer Ziele in Software-Engineering-Praktika. In: Löhr, K.-P. und Lichter, H. (Hrsg.), *Software Engineering im Unterricht der Hochschulen (SEUH9).* S. 2–15. Heidelberg. 2005. dpunkt.verlag.

[WW02] Weidemann, B. und Werner, H. Studienordnung für den Diplomstudiengang Informatik an der Universität Kassel. http://www.uni-kassel.de/eecs/dekanat/studium/pdf/SO-D-Informatik.pdf. 2002.

[Ze99] Zeller, A.: Funktionell und verständlich programmieren – so lernen es die Passauer. *Softwaretechnik-Trends*. 19(3):29–34. 1999.

[Ze00] Zeller, A.: Making students read and review code. In: Tarhio, J., Fincher, S., und Joyce, D. (Hrsg.), *5th ACM SIGCSE/SIGCUE Annual Conference on Innovation and Technology in Computer Science Education*. S. 89–92. New York. 2000. ACM Press.