aus

Oliver J. Bott, Peter Fricke, Uta Priss, Michael Striewe (Hrsg.)

Automatisierte Bewertung in der Programmierausbildung

Digitale Medien in der Hochschullehre Band 6
2017, 420 Seiten, br., 42,90 €, ISBN 978-3-8309-3606-0

Waxmann Verlag GmbH
www.waxmann.com info@waxmann.com

2 Automatisierte Bewertung in der Programmierausbildung – Eine Übersicht

Sven Strickroth und Niels Pinkwart

Zusammenfassung
Dieses Kapitel gibt eine Übersicht über verschiedene Sichtweisen und Klassifikationsansätze für Programmierlernsysteme beziehungsweise automatische Bewertungssysteme. Dabei werden besondere Ansätze innerhalb dieser Kategorisierungen mit Beispielen hervorgehoben sowie Trends identifiziert und diskutiert.

2.1 Einleitung

An vielen Schulen, Hochschulen und in der Weiterbildung nimmt die Relevanz des Themas Informatik gegenwärtig stark zu – so werden zum Beispiel Rufe nach einem Pflichtfach Informatik in Schulen lauter, zahlreiche Universitäten führen Einführungskurse zu IT-Themen in die Curricula von auch nicht informatischen Studiengängen ein, und Selbstlernangebote wie MOOCs werden insbesondere zu IT-Themen stark frequentiert. In diesen Lern- und Ausbildungskontexten spielt Programmierung typischerweise eine zentrale Rolle. Nicht allen fällt es jedoch leicht, Programmierfähigkeiten zu erwerben: Oft stehen Lernende beim Erstellen, Testen und Debuggen von Programmen vor schier unüberwindlichen Hindernissen. Lehrpersonen können hier natürlich unterstützen, haben aber nicht immer die Zeit, auf die Bedürfnisse aller Kursteilnehmer ausführlich einzugehen.

Computer sind in der Lage, Lernerlösungen zu Programmieraufgaben automatisiert zu analysieren. Die Ergebnisse der automatischen Bewertung können zum Beispiel als Feedback an Lerner zurückgespiegelt werden, um diese bei der individuellen Lösungsfindung zu unterstützen. In Lernszenarien, an denen viele Lernende teilnehmen, aber nur begrenzte Lehrpersonenzeit zur Verfügung steht, wie etwa in E-Learning-Kursen oder großen universitären Veranstaltungen, ist diese

partielle Ersetzung von Lehrpersonal bei der Aufgabe der Prüfung von Lösungsversuchen praktisch hochrelevant [KJH16] und wird von Lernenden auch nicht grundsätzlich abgelehnt [LMM98; Mat94]. Nach Auffassung von Kay und Kollegen [Kay+94] sei auch ganz grundsätzlich die manuelle Bewertung von Lernerlösungen hinsichtlich Konsistenz und Genauigkeit als problematisch anzusehen. Eine andere Nutzungsvariante der automatischen Programmanalyse besteht darin, Lehrpersonen bei der Bewertung und Feedbackgabe zu unterstützen [SG14] – so kann eine semiautomatische Bewertung insbesondere Vorteile hinsichtlich einer ganzheitlichen Bewertung bieten, da so auch Punkte bei kleineren Fehlern vergeben werden können (vgl. [AM05]).

Unabhängig von der Zielperson, welcher die Ergebnisse der automatischen Programmanalyse präsentiert werden (Dozent oder Lerner), müssen Analyseverfahren für Lernerlösungen im Bereich der Programmierung in der Lage sein, auch mit fehlerhaften und unvollständigen Lösungen umzugehen, die zusätzlich noch bei komplexeren Aufgabenstellungen auf verschiedenen und eventuell fehlerbehafteten Lösungsstrategien beruhen können [LLP13]. Diese Aspekte sind eine Herausforderung für automatisierte Analyseverfahren wie auch für den Entwurf von Feedbackmechanismen auf Basis der Analyseergebnisse. Professionelle Programmierentwicklungsumgebungen (IDEs – Integrated-Development-Environments) sind diesen Herausforderungen mit ihren Analyseverfahren nicht gewachsen: Sie haben Funktionen, die für Novizen zu schwierig sein können. Außerdem sind die in IDEs eingebauten Typen von Codeanalyse und Rückmeldungen nicht auf Lerner zugeschnitten und können diese daher oft eher verwirren als ihnen helfen [Pea+07].

Aufgrund der praktischen Relevanz und der weder eindeutigen noch einfachen Konstruierbarkeit von Systemen zur automatisierten Bewertung von Lernerlösungen zu Programmieraufgaben ist es wenig überraschend, dass Forschung und Entwicklung zu Unterstützungssystemen für die Programmierausbildung mittlerweile eine beachtliche Tradition in der Informatik haben. In der Forschung reicht diese mit dem LISP-Tutor mindestens bis in die 1980er Jahre zurück [AFS84]. Forschungsergebnisse zu mehreren hundert weiteren Systemen wurden in der Zwischenzeit publiziert – so beinhalteten 99 von den insgesamt 444 zwischen 1984 und 2003 auf dem SIGCSE Technical Symposium veröffentlichten Forschungsartikeln ein Softwareunterstützungssystem [Val04]. Die Zahl nicht publizierter Systeme, die sich in der universitären Praxis in der Anwendung befinden, ist sicherlich noch deutlich größer, da wahrscheinlich jedes Informatikinstitut in irgendeiner Weise ein eigenes System zur Unterstützung des Lehrbetriebs implementiert hat. Viele Systeme weisen sehr ähnliche Features auf und wären grundsätzlich auch für ein Assessment in anderen Szenarien einsetzbar [Iha+10], was jedoch

nicht praktiziert wird. Die folgenden Gründe für die Vielfalt an Systemen wurden von [Pea+07] und [Iha+10] identifiziert:

- Ein System wurde für einen speziellen Kurs entwickelt. Dabei wird ein Tool beziehungsweise Analyseverfahren genau für diesen Kurs angepasst und ist dadurch in der Regel nicht ohne weiteres in anderen Kontexten erneut einsetzbar. Die Adaptierung an andere Kontexte wäre oft zwar möglich – aber es gibt niemanden, der dies tut.

- Ein System wurde im Rahmen einer Abschlussarbeit oder eines Forschungsprojekts entwickelt. Dabei steht die Erstellung und Evaluation eines neuen Ansatzes im Mittelpunkt, weniger die Praxistauglichkeit. Bei so erstellten Systemen handelt es sich meist um einen Prototypen, der nach dem Abschluss des Projekts beziehungsweise der Abschlussarbeit keinen Support mehr erhält.

Diese Gründe, wie auch die teils nicht erfolgte Publikation von Systemen, trugen sicherlich dazu bei, dass immer wieder ähnliche Systeme entwickelt und lokal eingesetzt wurden, dass sich aber bislang kaum Systeme breit durchgesetzt haben [Iha+10]. Neben diesen eher praktisch-organisatorischen Argumenten ist es jedoch natürlich auch so, dass auf der Ebene der Forschung zu Programmierlernsystemen in den letzten 4 Jahrzehnten etliche Weiterentwicklungen stattgefunden haben. Folglich gibt es auch eine Reihe von Review- und Survey-Artikeln [DE88; DM98; LMM98; KP05a; Guz04; GA05; DLO05; AM05; Pea+07; Iha+10; CR13; SG14; Le16; KJH16], von denen im folgenden Abschnitt die acht aktuellsten genauer betrachtet und vorgestellt werden. Diese zeigen auf, welche verschiedenen Sichtweisen und Klassifikationsansätze für die Menge an Programmierlernsystemen in der jüngeren Vergangenheit gewählt worden sind.

2.2 Übersicht über verschiedene Reviews

In diesem Abschnitt werden verschiedene Sichtweisen und Klassifikationsansätze für Programmierlernsysteme und automatische Bewertungssysteme vorgestellt. Tabelle 2.1 gibt dabei einen Überblick über die Fokusse der einzelnen betrachteten Reviews. Aufgrund des Themas dieses Buches wurden Reviews zu Visualisierungsansätzen (z. B. [SKM13]), Programmierumgebungen (z. B. [Guz04; KP05a]) und intelligenten Tutoring-Systemen (ITS, z. B. [Pil03]) nicht genauer betrachtet.

Review	Fokus/Klassifizierung
Douce, Livingstone und Orwell [DLO05]	historische Übersicht; identifiziert auch allgemeine Schwierigkeiten und Probleme von automatischem Assessment
Ala-Mutka [AM05]	automatische Bewertungsmethoden; diskutiert auch semiautomatisches vs. automatisches sowie formatives vs. summatives Assessment
Pears u. a. [Pea+07]	Lehrmethoden; geht auch auf Curricula, Pädagogik und Sprachauswahl ein
Ihantola u. a. [Iha+10]	Systemfeatures; geht dabei auch auf Aspekte wie Sicherheit, Lizenzierung und Verfügbarkeit ein
Caiza und Ramiro [CR13]	Bewertungsmetriken; grobe Klassifikation in „mature“ und „recently developed tools“
Striewe und Goedicke [SG14]	statische Analysetechniken; stellt auch Vor- und Nachteile dieser Analysetechniken vor
Le [Le16]	adaptive Feedbackansätze; stellt auch aktuelle Forschungslücken dar
Keuning, Jeuring und Heeren [KJH16]	formative Feedbackansätze; analysiert zudem Anpassbarkeit und Qualität von Evaluationen

Tabelle 2.1: Schwerpunkte der betrachteten Review- und Survey-Artikel

2.2.1 Historische Übersicht und Klassifizierung nach Generationen

Das Review von [DLO05] betrachtet die Geschichte von automatischen testbasierten Bewertungstools ausgehend von ersten Publikationen in den 1960er Jahren bis 2005. [DLO05] klassifiziert die Tools anhand des Alters in drei Generationen:

Erste Generation – frühe Bewertungssysteme Die erste Generation beinhaltet erste Ansätze von Entwicklungen automatisierter Bewertungssysteme für Programmiersprachen auf digitalen Systemen im weitesten Sinne.

Zweite Generation – Toolorientierte Systeme Die zweite Generation beschreibt verschiedene Ansätze und Tools, die von Dozenten genutzt und entwickelt wurden, um die automatische Bewertung für sie zu vereinfachen (z. B. durch skriptbasierte Ansätze zum Testen, oftmals kommandozeilenbasiert), teilweise schon mehrere Kriterien auf einmal zu überprüfen und daraus Reports zu generieren. Fortgeschrittenere Systeme dieser Genera-

tion konnten auch von Lernenden genutzt werden, um Vorlesungsnotizen sowie Aufgaben abzurufen (z. B. Ceilidh, [Hig+03]).

Dritte Generation – Weborientierte Systeme Einige Tools der zweiten Generation wurden um Webtechnologien erweitert und richten sich sowohl an Lernende als auch an Lehrende, um sie bei Schwierigkeiten zu unterstützen und einen Überblick über den Fortschritt aller Lernenden bereitzustellen (z. B. CourseMarker, [Hig+03], oder BOSS2, [JGB05]). Zentrale Erweiterungen bestehen in der Integration von Plagiatserkennungen und Eingriffsmöglichkeiten von Lehrenden in die automatische Bewertung.

Darüber hinaus wurden drei Schwierigkeiten als allgemeine Probleme von automatischen Bewertungssystemen identifiziert: 1) Die Spezifikation von Aufgaben muss sehr sorgfältig durchgeführt werden, um Interpretationsschwierigkeiten zu vermeiden. 2) Das Erstellen von grafischen Programmen ist für Lernende attraktiver, jedoch sind diese sehr schwer automatisiert zu überprüfen. Zum Beispiel sind sehr konkrete Events oder Methoden vorzugeben oder es ergeben sich allgemeine Probleme: „Zeichen Sie die britische Flagge in einem Applet“. 3) Eine Lösung kann nach Meinung von Douce zwar funktional korrekt, aber pathologisch konstruiert sein – daher könne nur durch die Verbindung verschiedener Kriterien eine automatische Bewertung erfolgen.

2.2.2 Übersicht und Kategorisierung automatischer Bewertungsmethoden

[AM05] fokussiert in ihrem Review auf Merkmale, die automatisch bewertet werden können. Sie unterscheidet grundsätzlich zwischen dynamischen Bewertungen, die eine Ausführung der Lernerlösung erfordern, und statischen Bewertungen, die allein anhand des Quellcodes durchgeführt werden können:

2.2.2.1 Dynamisches Assessment

Charakteristisch für dynamisches Assessment ist das Ausführen der Lernerlösungen. [AM05] identifiziert dabei die folgenden Aspekte, die betrachtet werden können. Grundsätzlich wird für die Ausführung eine besonders gesicherte Umgebung benötigt.

Funktionalität Die Funktionalität wird durch Überprüfung von Ausgaben gegen vorher definierte Testeingaben geprüft. Dies kann über festgelegte er-

wartete Ausgaben des gesamten Programms (Blackbox-Test, exakt oder zum Beispiel mit regulären Ausdrücken) oder auf der Basis von einzelnen Methoden (Whitebox-Test) erfolgen, wobei die Ausgaben auch dynamisch mit denen einer Musterlösung verglichen werden können. Im Fall von Whitebox-Tests können Lernenden dabei konkrete zu implementierende Schnittstellen vorgegeben werden (die Mehrzahl der Systeme geht so vor) oder Lernende lösen sogenannte Fill-in-the-Gap-Aufgaben, in denen der studentische Quellcode in eine Vorlage eingefügt wird (z. B. ELP, [Tru07]).

Effizienz Effizienz kann zum Beispiel durch das Messen der Ausführungszeit mit verschiedenen Eingabedaten bewertet und mit einer Musterlösung verglichen werden. Ceilidth und Assyst verwenden zum Beispiel dynamisches Profiling für die Messung der Effizienz, wobei gezählt wird, wie oft bestimmte Blöcke ausgeführt werden [AM05].

Testfähigkeiten Lernende sollen lernen, Tests für ihre Programme zu entwickeln und selbstständig vor der Abgabe zu überprüfen. Dabei gibt es verschiedene Ansätze, wobei Lernende sowohl ihre Lösung als auch ihre Tests einreichen: [Che04] überprüft die Tests der Lernenden durch fehlerhafte „Musterlösungen“. Als ein weiteres Beispiel wird Web-CAT [Edw04] genannt; ein System, das ausschließlich eine Aufgabenstellung sowie eine durch den Dozenten erstellte Musterlösung benötigt. Zur Bestimmung einer Bewertung werden schließlich die von den Lernenden erstellten Tests gegen die Musterlösung sowie deren eigene Lösung ausgeführt und mit der Testabdeckung verrechnet.

Spezielle Features [Rin99] überprüft spezifische Aspekte der Programmiersprachen, indem das normale Memorymanagement mit einem eigenen überschrieben wird, um Missmanagement, wie zum Beispiel nicht freigegebenen Speicher, zu entdecken.

2.2.2.2 Statisches Assessment

Statisches Assessment erfolgt ausschließlich auf der Quellcodeebene. Gemeinsame Voraussetzung für die durch [AM05] identifizierten Ansätze sind syntaktisch korrekte Lösungen.

Coding-Style Neben der Überprüfung, ob ein Programm syntaktisch korrekt ist, können Abgaben zum Beispiel auf ungenutzte Variablen (wofür viele

Interpreter und Kompiler Funktionalität mitbringen) oder Einhaltung von Style-Regeln geprüft werden (z. B. Checkstyle).

Programmierfehler Unabhängig von dynamischen Funktionstests, die von den meisten Tools im Review genutzt werden, können statische Analysen zum Einsatz kommen, um häufige Fehler zu erkennen (z. B. Zählvariable in einer Schleife wird nicht aktualisiert, redundante Zuweisungen oder idempotente Operationen).

Softwaremetriken Softwaremetriken können genutzt werden, um allgemeine Indikatoren einer Software zu bestimmen (z. B. Anzahl der Attribute, Anzahl von Operationen und Operanden, Codezeilen oder die zyklomatische Komplexität).

Design Zum Design zählen Überprüfungen, ob eine Lösung eine vorgegebene Schnittstelle einhält oder wie sich die Struktur einer Lösung zu Musterlösungen verhält. Daneben gibt es Ansätze, die aus den Lösungen UML-Diagramme generieren, Software-Patterns erkennen und dies zur Bewertung nutzen.

Spezielle Features Hierzu zählen zum einen Überprüfungen, ob eine Lösung bestimmte Strukturen der Sprache oder Bibliotheksfunktionen nutzt, die zum Beispiel für eine Aufgabe ausgeschlossen sind. Zum anderen spielen hier auch Plagiatserkennungsmethoden eine Rolle.

Neben der Kategorisierung diskutiert [AM05] auch allgemein die Nutzung von automatischen Bewertungsmethoden in der Ausbildung: Es darf keine Uneindeutigkeiten in der Spezifikation geben und selbst kleinste Fehler in den Tests können zu großen Problemen führen, wobei die Auswahl der eingesetzten Bewertungsmethoden didaktisch begründet werden sollte. Zudem werden Systeme oftmals als reine Bewertungssysteme angesehen. Sie können darüber hinaus auch zur „automatisierten Verwaltung“ von zum Beispiel Peer-Reviews oder komplexeren Settings, wie im Quiver-System [EFK04] genutzt werden: Dort erstellen fortgeschrittene Lernende Schnittstellen und Tests, die danach von Anfängern in Einführungsveranstaltungen implementiert werden. Weiterhin werden semiautomatische und vollautomatische Bewertung sowie formatives und summatives Feedback diskutiert. [AM05] weist darauf hin, dass nicht alle Kriterien „guten Programmierens“ automatisch bewertet werden können: Menschen bewerten Lösungen ganzheitlich und vergeben eventuell auch bei kleineren Fehlern (z. B. Syntaxfehlern) die volle Punktzahl. Dies kann durch semiautomatische Bewertung ebenfalls erreicht werden, wobei Tests vollautomatisch durchgeführt und die Ergebnisse den Bewertern

präsentiert werden. Einige Systeme sind auf ausschließlich summative Bewertungen ausgelegt, andere geben Feedback und erlauben erneute Einreichungen, wobei sehr viele Ansätze, obwohl sie iterative Entwicklungen hervorheben, immer eine vollständige Version der Lösung erwarten. [AM05] identifiziert innerhalb der formativen Ansätze Tutoring-Systeme als weitere Unterkategorie, die hauptsächlich zur Unterstützung von Lernenden entwickelt wurden und dadurch im Vergleich mehr (adaptives) Feedback bereitstellen als reine Assessment-Systeme.

2.2.3 Übersicht über Lehrmethoden zur Einführung in die Programmierung

Das Literatur-Survey von [Pea+07] beschäftigt sich mit der Beantwortung der Frage: „Welche Forschungsliteratur kann Lehrende unterstützen neue Einführungsprogrammierkurse zu entwerfen?“. Dazu beschäftigt sich das Survey sowohl mit Curriculadesign, pädagogischen Aspekten, Auswahl einer passenden Programmiersprache und schließlich auch mit Tools, die sich speziell an Programmieranfänger richten und in diesem Kontext eingesetzt werden können. [Pea+07] klassifizieren Systeme in die folgenden vier Kategorien. Insgesamt spielen Systeme zur automatischen Programmbewertung dabei nur eine untergeordnete Rolle.

Visualisierungstools Da Menschen gut beim Verarbeiten von grafischen Informationen sind, liegt es nahe, eher abstrakte Algorithmen und Konzepte zu visualisieren. Es kann grundsätzlich zwischen Tools unterschieden werden, die Strukturen und die Ausführung von Code veranschaulichen (z. B. visuelle Debugger, Animationen oder Simulationen).

Automatische Bewertungstools Tools zur automatischen Programmbewertung dienen meist sowohl der Erleichterung von Lehrenden als auch der Bereitstellung von formativem Feedback für Lernende in angemessener Zeit. Viele Tools bieten dabei Unterstützung, um die Korrektheit von Abgaben zu prüfen (meist Blackbox-Testing). Es gibt aber auch Ansätze interne Datenstrukturen oder Programme detaillierter zu überprüfen. Die Bewertung erfolgt in den Tools oftmals semiautomatisch, aber zum Teil auch vollautomatisch. Dies kann zum Beispiel zu einer Verringerung der Subjektivität bei Bewertungen genutzt werden. Grundsätzlich sind diese Tools nicht auf Programmcode beschränkt, sondern es gibt auch Systeme, die Ablaufdiagramme (CourseMarker, [Hig+05]) oder Algorithmen in Simulationsumgebungen (TRAKALA2, [Nik+04]) bewerten.

Programmierumgebungen Umgebungen stellen Programmierern aller Erfahrungsstufen Möglichkeiten und Unterstützungen bereit, um Programme zu erstellen und auszuführen. Eine Unterteilung erfolgt in „Programming support tools“ (z. B. reduzierte IDEs wie BlueJ, [Kö+03]) und „Microworlds“ (Umgebungen basierend auf physischen Metaphern, z. B. Karel oder Robocode). Eine weitere Übersicht über Programmierumgebungen findet sich bei [KP05a].

Andere Tools In diese Kategorie fallen Tools zur Plagiatserkennung oder intelligente Tutoring-Systeme (z. B. LISP Tutor, [AS86]).

2.2.4 Features automatischer Bewertungstools

Ziel des Reviews von [Iha+10] ist die Darstellung der Features von automatischen Bewertungstools für Programmierübungen aus den Jahren 2006–2010 basierend auf einem systematischen Literaturreview. Zentrale Ergebnisse des Reviews sind:

Unterstützte Programmiersprachen Die Mehrheit der Systeme zielen auf Java oder unterstützen diese Sprache. Weiterhin werden C/C++, Python und Pascal, aber auch Assembler, Shell Scripts oder VHDL von einigen Systemen angeboten. Es gibt aber auch Systeme, die unabhängig der Sprache zum Beispiel für Blackbox-Tests (darunter auch zum Testen von Webapplikationen, z. B. AWAT, [SQF08]) eingesetzt werden können.

Integration in Learning-Management-Systeme Es zeigt sich ein Trend zur Integration von Bewertungssystemen in LMS, da dadurch zum Beispiel keine Kursverwaltungsfunktionen erneut implementiert werden müssen.

Definition der Tests Tests werden entweder mit Tools aus der Industrie (XUnit, Akzeptanztest-Frameworks sowie Web-Testing-Frameworks wie Watir) oder spezialisierten Lösungen (Blackbox-Testing über Ausgabenvergleiche, Scripting sowie neuen Ansätzen aus der Forschung) durchgeführt.

Erneute Einreichung von Lösungen Erneute Einreichungen werden partiell sehr unterschiedlich gehandhabt: Bei den meisten Systemen kann die Anzahl der Abgaben limitiert werden. Andere Ansätze bestehen im Einschränken des Feedbacks, Zeitstrafen für fehlgeschlagene Tests, Nutzung parametrisierter Aufgaben oder Durchführung von sogenannten Programmierkontesten (Lösung von Aufgaben gegen die Zeit und andere Teilnehmer) und Kombinationen daraus.

Möglichkeiten einer manuellen Bewertung Im Review wurden drei Level für den Umfang manueller Bewertungen identifiziert: keine Eingriffsmöglichkeiten, reine manuelle Bewertung sowie Kombinationen aus manuellen und automatischen Bewertungen, bei denen Lehrende das automatische Feedback durch weitere Anmerkungen anreichern. Für kein System des Reviews ist es explizit möglich, die automatische Bewertung zu überschreiben.

Sandboxing (Sicherheit) Die Bewertung von eingereichten Lösungen erfordert es oftmals, dass diese ausgeführt werden sollen. Lösungsansätze beinhalten die Ausführung in einer geschützten Umgebung, Entfernen von möglichem schädlichen Code mittels statischer Analyse und Durchführung der Bewertung direkt auf dem Client. Teilweise werden Lösungen auch auf gesonderten Servern ausgeführt.

Verbreitung und Verfügbarkeit Im Review zeigte sich, dass nur sehr wenige Systeme unter einer Open-Source-Lizenz veröffentlich wurden beziehungsweise frei verfügbar sind. Viele der publizierten Prototypen konnten nicht im Internet gefunden werden.

Spezielle/einmalige Funktionen Als Besonderheiten stellten sich Systeme für die automatische Bewertung von grafischen Benutzeroberflächen, SQL-Tutoring, nebenläufige Programmierung, Webprogrammierung und Ansätzen heraus, in denen Lernende gegenseitig die Qualitätssicherung vornehmen.

2.2.5 Kategorisierung in ausgewachsene und neue Tools sowie Übersicht über Bewertungsmetriken

[CR13] unterteilen die Tools in ihrem Review in „mature tools“ und „recently developed tools“ und beschreiben unter anderem „key features“, unterstützte Programmiersprachen und Bewertungsmetriken. Auf welcher Grundlage die Systeme in diese beiden Kategorien eingeordnet werden, wird nicht genauer erläutert, außer allgemein Feature-Reichtum und Verbreitung zu nennen. Die meisten der „mature tools“ sind unter einer Open-Source-Lizenz veröffentlicht und unterstützen Java, wobei insbesondere bei den neuen Tools ein Trend zu LMS-Extensions (Moodle-Plugins) erkennbar ist.

Ein weiterer Beitrag des Reviews besteht in der Diskussion von Bewertungsmetriken. Es wird hervorgehoben, dass jede Institution beziehungsweise jeder Dozent eigene Metriken verwendet, wobei diese nach Meinung der Autoren jedoch möglicherweise nicht vollständig automatisch zur Bestimmung eines Wer-

tes angewendet werden können. Jedes Tool des Reviews enthält eigene Bewertungsmethoden und -metriken, um eine Note für studentische Abgaben zu bestimmen. Teilweise sind diese im System fest einprogrammiert, können aber zum Teil auch durch Plug-Ins erweitert werden. Insgesamt wurden oftmals für gleiche beziehungsweise sehr ähnliche Metriken verschiedene Namen gewählt. Tabelle 2.2 zeigt die identifizierte Klassifikation von Metriken. Letztlich wird die Lücke eines fehlenden konfigurierbaren Bewertungsschemas identifiziert, um beliebige Metriken abzubilden.

Kategorisierung		**Metrik**
Ausführung		Kompilierung (z. B. Compiler)
		Ausführung (z. B. Interpreter)
Funktionstests		Funktionalität (System- oder Methodenlevel, z. B. Unit-Tests)
Nicht-funktionale Tests	Spezielle Anforderungen	Spezielle Anforderungen an eine Aufgabe
	Wartbarkeit	Design
		Stil (z. B. Style-Checker)
		Komplexität
	Effizienz	Nutzung physischer Ressourcen
		Ausführungszeiten
		Anzahl von Prozessen
		Datei- oder Quellcodegröße

Tabelle 2.2: Klassifikation von Bewertungsmetriken nach [CR13]

2.2.6 Statische Analysetechniken für die automatische Programmbewertung

Das Review von [SG14] fokussiert auf didaktische Vor- und Nachteile von statischen Analysetechniken, die zur automatischen Programmbewertung eingesetzt werden können. Das Review beschränkt sich dabei auf Ansätze für die objektorientierte Programmierung in Java sowie Technologien, die in serverbasierten Systemen eingesetzt werden können. Folglich werden Analyse- und Feedbackmechanismen aus IDEs ausgeklammert. Eine weitere Annahme besteht darin, dass Lernende in der Lage sind, lokal auf ihren Computern (z. B. mithilfe von IDEs) syntaktisch korrekte Programme zu erzeugen.

Die folgenden zentralen Anforderungen für statische Analysen in diesem Kontext werden genannt:

- Statische Analysen sind in der Lage Misskonzeptionen aufzudecken, die auch in syntaktisch korrekten Lösungen auftreten können.
- Feedback muss auch für Teile der studentischen Abgabe gegeben werden, die keinen relevanten Beitrag zur Lösung liefern.
- Statische Analysen können den Quellcode nach „verbotenen“ oder „erforderlichen“ Konstrukten durchsuchen (z. B. soll Rekursion und keine Schleifen benutzt werden).
- In Tutoring-Szenarien sollten Lernende nicht nur über Fehler, sondern auch über mögliche Fehlerbehebungen oder nächste Schritte informiert werden.
- Überprüfungen auf Plagiate.

Neben den Anforderungen und technischen Lösungen werden auch die Integration von Tools in existierende Systeme sowie das Erstellen von Tests und Feedbackregeln diskutiert: Die Tools des Reviews unterstützen hauptsächlich die Anwendung von regelbasierten Überprüfungen auf Quell- oder Bytecodeebene. Zum Einsatz kommen vor allem Analysetools aus der professionellen Softwareentwicklung, wie zum Beispiel Checkstyle, PMD und FindBugs, oder Analysen von abstrakten Syntaxbäumen beziehungsweise Graphen. Eine wichtige Voraussetzung zum Einsatz in der Lehre sind konfigurierbare Tests, die auch von Lehrpersonen angepasst werden können. Zum Teil sind diese fest einprogrammiert (z. B. FindBugs), erfordern das Erstellen von Regeln in vorgegebenen Anfragesprachen (z. B. XPath oder GReQL, [BE08]) oder die Nutzung von komplexen Analyseprogrammen, die in das Analyseframework integriert werden müssen. Um zu einer Bewertung zu gelangen, kann bei existierenden professionellen Tools zur Gewichtung auf vorgegebene Schweregrade zurückgegriffen werden (z. B. bei PMD, Checkstyle oder FindBugs).

2.2.7 Klassifikation von adaptiven Feedbackansätzen

Das Review von [Le16] fokussiert auf eine Klassifikation von adaptiven Feedbackansätzen. Dabei werden die Feedbackansätze wie folgt unterteilt:

Ja/Nein-Feedback Bei dem Ja/Nein-Feedback handelt es sich um die kürzeste Variante, die lediglich angibt, ob eine Lösung korrekt ist oder nicht. Dazu zählen zum Beispiel auch „Es liegen Syntaxfehler vor“, „Es gibt keine Syntaxfehler, aber die Berechnung ist falsch“ oder „Lösung ist korrekt“.

Handelt es sich um abgefragte Berechnungen zum Beispiel von Variablenbelegungen, wird den Lernenden teilweise auch eine richtige Lösung präsentiert. In der Regel werden Lösungen mit vorgegebenen Werten verglichen.

Syntaxfeedback Dieses Feedback basiert auf den Ausgaben eines Compilers, die von den meisten Systemen unverändert an die Lernenden durchgereicht werden. Es gibt aber auch Ansätze (z. B. bei VC Prolog, [Pey+00]), in denen eine detaillierte syntaktische Fehlerbeschreibung generiert wird.

Semantisches Feedback Semantisches Feedback bezieht sich auf Hinweise auf Fehler in der Erfüllung der Anforderungen. Dabei kann das Feedback einerseits zweistufig („intention- and code-based“) oder andererseits ausschließlich „code-based“ generiert werden. Im ersten Fall wird versucht die Intention beziehungsweise Lösungsstrategie der Lernenden zu erkennen und basierend darauf Hinweise zu geben. Ansätze finden sich hier vor allem für Prolog und funktionale Sprachen (z. B. Hong's Prolog, [Hon04], und Ask-Elle, [Ger+16]), aber zum Beispiel mit Java-Bugs [SS08], einer Bibliothek von häufigen Programmierfehlern beziehungsweise Misskonzepten, auch für objektorientierte Sprachen. Im zweiten Fall beschränkt sich die Analyse auf das Finden von fehlerhaftem Code. Zum Beispiel untersucht JACK (vgl. Kapitel 9) Programmtraces von Lernenden und zeigt Ausschnitte, in denen von der korrekten Lösung abgewichen wird. Einen datengetriebenen Ansatz verwenden zum Beispiel FIT Java Tutor [GP15] und ITAP [RK15], die eine Lösung mit existierenden Lösungen aus einem Lösungsraum vergleichen und diese zur Hilfe (eventuell in Auszügen) bereitstellen oder auf konkrete Unterschiede hinweisen.

Layout-Feedback Dieses Feedback soll Lernenden helfen, Programme zu erstellen, die gewissen Coding-Conventions entsprechen. Ziel dabei ist, dass die Lernenden den Quellcode leichter verstehen und so zum Beispiel auch Fehler einfacher finden können. Dies kann u. a. durch Style-Checker, Codemetriken oder auch durch Layoutvergleiche mit einer Musterlösung (VC Prolog, [Pey+00]) erfolgen, wird aber insgesamt nur von wenigen Systemen angeboten.

Qualitätsfeedback Zur Messung der Qualität werden Metriken des Software-Engineering eingesetzt. Gemessen wird dabei nicht, ob ein Algorithmus korrekt ist, sondern wie effizient er hinsichtlich Zeit beziehungsweise Speicher implementiert wurde. Dazu zählt ebenfalls ELP [Tru07], wo Hinweise für schlechte Programmierpraktiken (wie „if (a==true)“ anstatt von „if (a)“)

gegeben werden, oder JACK (vgl. Kapitel 9), wo Java-Programmtraces mit Musterlösungen verglichen werden, um unnötig komplexe Programme zu identifizieren.

2.2.8 Betrachtung von formativen Feedbackansätzen

Das Review von [KJH16] fokussiert auf die systematische Betrachtung von formativen Feedbackansätzen und klassifiziert Learning- als auch Assessment-Tools hinsichtlich unterstützter Feedbacktypen. Die Feedbacktypen orientieren sich an von [Nar08] aufgestellten Kategorien.

- Kenntnis des Erfolgs für eine Menge von Aufgaben (z. B. summatives Feedback: 85 % korrekt)
- Kenntnis des Ergebnisses bzw. der Antworten (z. B. erfüllt alle Testfälle/Constraints, Ausgabe entspricht der Musterlösung)
- Kenntnis eines korrekten Ergebnisses (z. B. Darstellung einer Musterlösung)
- Kenntnis über Aufgabenbeschränkungen (z. B. Hinweise zu Anforderungen: for-Schleife muss genutzt werden)
- Kenntnis über Konzepte (z. B. weitergehende Erklärungen oder Beispiele zur Thematik)
- Kenntnis über Fehler (z. B. Testfehlschläge, Syntaxfehler, Laufzeit- bzw. logische Fehler, Styleverletzungen, Performanzprobleme). Oftmals werden hierfür professionelle Testtools eingesetzt. Feedback kann sehr einfach (z. B. Anzahl von Fehlern, Note, Prozentsatz, einfacher Hinweis) oder ausführlicher mit Beschreibungen der Fehler erfolgen.
- Kenntnis darüber, wie fortzufahren ist (z. B. Hinweise, wie ein Fehler behoben werden kann oder wie die aktuelle Lösung ergänzt werden sollte). Feedback kann in Form von Hinweisen (z. B. Vorschlag, Fragen, Beispiel), einer Lösung, die direkt anzeigt, was noch zu erledigen ist, oder als Kombinationen davon erfolgen.
- Kenntnis über Metakognition (z. B. Überprüfung, ob ein Lernender weiß, warum eine Lösung korrekt ist)

In etwas mehr als der Hälfte (60 %) der Systeme wird nur ein einzelner der oben genannten Feedbacktypen angeboten. [KJH16] stellen fest, dass das bereitgestellte Feedback im Allgemeinen nicht sehr divers ist, sondern hauptsächlich auf die Identifikation von Fehlern abzielt. Feedback des Typs „Kenntnis über Fehler“ findet sich in allen bis auf einem System innerhalb des Reviews: Hinweise auf Testfehlschläge bieten 75 %, Laufzeit- beziehungsweise logische Fehler 42 %, Syntaxprüfungen 36 %, Style- 17 % und Performanzanalysen 9 % der Systeme des Reviews. Feedback hinsichtlich „Kenntnis darüber, wie fortzufahren ist“ beinhalten 32 % der Tools, wobei 26 % Hinweise zur Fehlerbehebung und 14 % zum weiteren Vorgehen zur Lösung der Aufgabe geben. Feedback der Typen „Kenntnis über Aufgabenbeschränkungen“ wird durch 16 %, „Kenntnis über Konzepte“ durch 12 % und „Kenntnis über Meta-Kognition“ durch lediglich 1 % der Systeme ausgegeben. Als mögliche Erklärung geben [KJH16] an, dass viele Systeme als reine Bewertungssysteme für eine große Anzahl von Lernenden konzipiert und entwickelt wurden und somit der Fokus auf der automatischen Bewertung und nicht auf detailliertem Feedback liege. Trotzdem haben viele Autoren angegeben, dass ein Ziel der Systeme die Verbesserung des Lernens sei, wozu wiederum einfache Listen mit Fehlern nach Meinung der Autoren des Reviews nicht ausreichend seien.

Bei den unterstützten Programmiersprachen ist ein starker Fokus auf imperative beziehungsweise objektorientierte Sprachen festzustellen, wobei neuere Tools vor allem Support für Java, C und C++ bereitstellen.

Neben diesen Feedbackkategorien werden die Systeme hinsichtlich ihrer „ill-definedness“ der höchsten möglichen Aufgabentypen gruppiert. Dabei wurde auf die Klassifikation von [LP14] zurückgegriffen. Aufgaben der Klasse 1 haben eine einzige korrekte Lösung und sind oft Multiple-Choice Fragen oder Programmvorlagen, in die Codeteile eingefügt werden müssen. Aufgaben der Klasse 2 können über verschiedene Implementierungen gelöst werden – wobei oftmals eine grobe Vorlage (z. B. Schnittstelle) oder eine Beschreibung, die auf die Lösungsstrategie hindeutet, gegeben wird. Die dritte Klasse ist die höchste von [KJH16] betrachtete und beinhaltet Aufgaben, die sowohl durch unterschiedliche Strategien als auch durch Algorithmen gelöst werden können. 20 % der Systeme unterstützten maximal Aufgaben der Klasse 2 und die restlichen 80 % maximal Aufgaben der Klasse 3.

Daneben werden sowohl Techniken für die Generierung von Feedback, Möglichkeiten für Lehrende die Tools für ihre Bedürfnisse anzupassen (auch das Einstellen von Übungsaufgaben zählt dazu) als auch die Qualität und Effektivität der Tools dargestellt. Bemerkenswert ist dabei, dass durch viele Tools dynamische und statische Analysetechniken eingesetzt werden. Teilweise werden auch kom-

plexere Techniken eingesetzt, wie zum Beispiel Model-Tracing, constraint-based modelling, dynamisches Testen mit Reflections, statische Codeanalysen oder Intentionserkennung. Jedoch erschweren diese komplexeren Techniken das Hinzufügen neuer Aufgaben und das Vornehmen von Anpassungen am System – publizierte Angaben für die Dauer zur Erstellung einer Aufgabe variieren zwischen einigen Stunden und einer Personenwoche.

2.3 Zusammenfassung und Diskussion

In diesem Kapitel wurden acht verschiedene Sichtweisen und Klassifikationsansätze für Programmierlernsysteme beziehungsweise automatische Bewertungssysteme vorgestellt. Dabei wurden besondere Ansätze innerhalb dieser Kategorisierungen mit Beispielen hervorgehoben. Insgesamt ist auffällig, dass viele verschiedene Klassifikationsansätze mit teilweise sehr ähnlichen Kategorien vorgenommen wurden, es jedoch (noch) keinen Konsens für die Benennungen gibt (vgl. [KJH16]). Zudem scheint kein Review oder Survey eine umfassende Darstellung der existierenden Systeme und Tools vorzunehmen. Dies äußert sich vor allem darin, dass oftmals die Kriterien für die Aufnahme eines Systems oder Tools in das Review nicht eindeutig nachvollziehbar sind (vgl. [KJH16]) – eine besondere Schwierigkeit besteht sicherlich auch darin, dass einige Systeme nicht auf Englisch publiziert wurden, sondern „lediglich“ auf lokalen Workshops oder Konferenzen, wie zum Beispiel im deutschsprachigen Raum der DeLFI-Tagung.

Generell wird bei den vorgestellten Reviews ausführlich auf grundsätzliche Analyseansätze eingegangen. Speziell widmen sich die neueren Reviews sehr detailliert verschiedenen Feedbackarten. Dennoch gibt es keine umfassenden Reviews oder Übersichten, die sich genauer mit den Features der Systeme beschäftigen. Insbesondere könnten solche eine gute Grundlage für die Auswahl eines passenden existierenden Systems für Praktiker bieten, die ein Unterstützungssystem neu einführen möchten. Dabei sollte darüber hinaus auch die Verfügbarkeit und Lizenzierung der Systeme betrachtet werden. Zudem gibt es keine umfassenden Reviews mit Fokus auf technische Aspekte, wie zum Beispiel Architekturen und konkrete technische Lösungsansätze, die für Entwickler Aufschluss über bewährte Ansätze beziehungsweise Möglichkeiten zur Wiederverwendung existierender Systeme geben könnten. Ein Grund dafür ist wahrscheinlich, dass in vielen Publikationen technische Details oftmals nur relativ knapp am Rande oder ausschließlich innovative Features der Systeme erwähnt werden (vgl. [Iha+10]). Dennoch scheint sich ein Trend zu webbasierten Systemen zu entwickeln, die mindestens eine Anbindung an ein LMS bieten. Aber auch bei Lösungen zur sicheren Ausfüh-

rung von Lernerlösungen finden sich nur selten genauere Angaben. Insbesondere bei älteren Systemen oder Prototypen, die später in den Produktivbetrieb gewechselt sind, wurde oftmals auf Sicherheit kein großer Wert gelegt (vgl. [Iha+10]). [For06] gibt einen guten Einstieg in grundsätzliche Probleme und Lösungsmöglichkeiten.

Die unterstützten Programmiersprachen sind sehr divers, wobei von den meisten Systemen imperative oder objektorientierte Sprachen wie C/C++ oder Java angeboten werden. Hierbei ist besonders erwähnenswert, dass es hinsichtlich der Feedbacktypen und Analysetechniken eine starke Abhängigkeit von speziellen Sprachen zu geben scheint (vgl. [Le16]). Beispielsweise findet sich sehr detailliertes Feedback mit Intentionserkennung hauptsächlich für logische oder funktionale Programmiersprachen.

Die Mehrzahl der Systeme scheint sich auf das Bewerten von klassischen Programmieraufgaben zu fokussieren. Das automatische Bewerten von Aufgaben mit grafischen Oberflächen, „webbasierten Sprachen“ oder Android Apps [Hei+15] sind Ausnahmen. Speziell für die Bewertung von Webapplikationen reicht es in der Regel nicht, einen solchen Interpreter lokal auszuführen, sondern mit einer auf einem Webserver bereitgestellten Version zu interagieren. Ebenso sind Systeme sehr selten, die Kollaboration über Peer-Reviews hinaus unterstützen oder fördern. Eine nennenswerte Ausnahme ist das Quiver-System [EFK04], in dem verschiedene Kurse auf unterschiedlichen Erfahrungsstufen zusammenarbeiten.

Hinsichtlich automatisierter Bewertung wird von [AM05] und [Iha+10] hervorgehoben, dass sich nicht alle Programmieraufgaben vollständig automatisch bewerten lassen: [AM05] spricht dabei vom Vorteil einer ganzheitlichen Bewertung von Aufgaben durch Lehrende – so dass zum Beispiel teilweise Punkte vergeben werden können, obwohl einzelne Tests eventuell aufgrund von Kleinigkeiten fehlgeschlagen sind (vgl. [SOP11]). In diesem Zusammenhang beklagen [Iha+10], dass nur selten berichtet wird, wie mit solchen Aufgaben umgegangen wird beziehungsweise werden sollte.

Bereits bei [AM05] wurde festgestellt, dass viele Systeme unabhängig voneinander entwickelt wurden und es keine gemeinsamen Standards oder Schnittstellen gibt. Ein Grund dafür ist sicherlich, dass es sich um ein aktuelles Forschungsfeld handelt und immer wieder innovative, neue Ansätze entwickelt werden. Grundsätzlich gibt es daran auch nichts auszusetzen, dass verschiedene Systeme mit unterschiedlichen Schwerpunkten und Ansätzen existieren – einige Systeme haben auch eine größere Verbreitung gefunden. Aber unter Berücksichtigung der oftmals zeitaufwändigen Entwicklung guter Aufgaben und Analysetechniken wird hier auch viel Potenzial verschenkt: Gute, bewährte Aufgaben, welche nicht nur eine Beschreibung beinhalten, sondern speziell auch anspruchsvolle Feedback-

verfahren, können nicht über Systemgrenzen hinweg ausgetauscht oder eingesetzt werden. Auf der einen Seite gibt es hierfür technische Gründe: unterschiedliche Funktionalitäten von Systemen, die schwer zu überbrücken sind. Auf der anderen Seite gibt es aber auch kein etabliertes gemeinsames Austauschformat für Programmieraufgaben – in Kapitel 24 wird eine Spezifikation vorgeschlagen, die versucht, dieses Problem zu lösen.

Literatur für dieses Kapitel

[AFS84] John R. Anderson, Robert Farrell und Ron Sauers. „Learning to program in LISP“. In: *Cognitive Science* 8.2 (1984), S. 87–129.

[AM05] Kirsti M. Ala-Mutka. „A Survey of Automated Assessment Approaches for Programming Assignments“. In: *Computer Science Education* 15.2 (2005), S. 83–102. DOI: 10.1080/08993400500150747.

[AS86] J. R. Anderson und E. Skwarecki. „The Automated Tutoring of Introductory Computer Programming“. In: *Commun. ACM* 29.9 (Sep. 1986), S. 842–849. DOI: 10.1145/6592.6593.

[BE08] Daniel Bildhauer und Jürgen Ebert. „Querying software abstraction graphs“. In: *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2005), collocated with ICPC 2008* (2008).

[Che04] P. M. Chen. „An automated feedback system for computer organization projects“. In: *IEEE Transactions on Education* 47.2 (Mai 2004), S. 232–240. DOI: 10.1109/TE.2004.825220.

[CR13] Julio C. Caiza und José María del Álamo Ramiro. „Programming assignments automatic grading: review of tools and implementations“. In: *7th International Technology, Education and Development Conference (INTED2013)*. 2013, S. 5691–5700.

[DE88] M. Ducassé und A.-M. Emde. „A Review of Automated Debugging Systems: Knowledge, Strategies and Techniques“. In: *Proceedings of the 10th International Conference on Software Engineering*. ICSE ’88. IEEE Computer Society Press, 1988, S. 162–171.

[DLO05] Christopher Douce, David Livingstone und James Orwell. „Automatic Test-based Assessment of Programming: A Review“. In: *ACM Journal of Educational Resources in Computing* 5.3 (2005).

[DM98] Fadi P. Deek und James A. McHugh. „A Survey and Critical Analysis of Tools for Learning Programming“. In: *Computer Science Education* 8.2 (1998), S. 130–178. DOI: 10.1076/csed.8.2.130.3820.

[Edw04] Stephen H. Edwards. „Using Software Testing to Move Students from Trial-and-error to Reflection-in-action“. In: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '04. ACM, 2004, S. 26–30. DOI: 10.1145/971300.971312.

[EFK04] Christopher C. Ellsworth, James B. Fenwick Jr. und Barry L. Kurtz. „The Quiver system“. In: *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. SIGCSE '04. ACM, 2004, S. 205–209. DOI: 10.1145/971300.971374.

[For06] Michal Forišek. „Security of programming contest systems“. In: *Informatics in Secondary Schools, Evolution and Perspectives* (2006).

[GA05] Mercedes Gómez-Albarrán. „The Teaching and Learning of Programming: A Survey of Supporting Software Tools“. In: *The Computer Journal* 48.2 (2005), S. 130–144. DOI: 10.1093/comjnl/bxh080.

[Ger+16] Alex Gerdes u. a. „Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback“. In: *International Journal of Artificial Intelligence in Education* (2016), S. 1–36. DOI: 10.1007/s40593-015-0080-x.

[GP15] Sebastian Gross und Niels Pinkwart. „Towards an Integrative Learning Environment for Java Programming“. In: *IEEE 15th International Conference on Advanced Learning Technologies (ICALT), 2015*. Los Alamitos, CA: IEEE Computer Society Press, Juli 2015, S. 24–28. DOI: 10.1109/ICALT.2015.75.

[Guz04] Mark Guzdial. „Programming environments for novices“. In: *Computer science education research* 2004 (2004), S. 127–154.

[Hei+15] Mathis Heimann u. a. „Automatische Bewertung von Android-Apps“. In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2015)*. Bd. 1496. CEUR Workshop Proceedings. 2015.

[Hig+03] Colin Higgins u. a. „The CourseMarker CBA System: Improvements over Ceilidh“. In: *Education and Information Technologies* 8 (3, 2003), S. 287–304. DOI: 10.1023/A:1026364126982.

[Hig+05] Colin A. Higgins u. a. „Automated Assessment and Experiences of Teaching Programming“. In: *J. Educ. Resour. Comput.* 5.3 (Sep. 2005). DOI: 10.1145/1163405.1163410.

[Hon04] Jun Hong. „Guided programming and automated error analysis in an intelligent Prolog tutor“. In: *International Journal of Human-Computer Studies* 61.4 (2004), S. 505–534.

[Iha+10] Petri Ihantola u. a. „Review of Recent Systems for Automatic Assessment of Programming Assignments“. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling '10. ACM, 2010, S. 86–93. DOI: 10.1145/1930464.1930480.

[JGB05] Mike Joy, Nathan Griffiths und Russell Boyatt. „The boss online submission and assessment system“. In: *J. Educ. Resour. Comput.* 5.3 (Sep. 2005). DOI: 10.1145/1163405.1163407.

[Kay+94] David G. Kay u. a. „Automated grading assistance for student programs“. In: *Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education, 1994, Phoenix, Arizona, USA, March 10-12, 1994*. 1994, S. 381–382. DOI: 10.1145/191029.191184.

[KJH16] Hieke Keuning, Johan Jeuring und Bastiaan Heeren. *Towards a Systematic Review of Automated Feedback Generation for Programming Exercises – Extended Version*. Techn. Ber. UU-CS-2016-001. Department of Information und Computing Sciences, Utrecht University, März 2016. URL: http://www.cs.uu.nl/research/techreps/repo/CS-2016/2016-001.pdf.

[KP05a] Caitlin Kelleher und Randy Pausch. „Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers“. In: *ACM Computing Surveys (CSUR)* 37.2 (2005), S. 83–137.

[Kö+03] Michael Kölling u. a. „The BlueJ System and its Pedagogy“. In: *Computer Science Education* 13.4 (2003), S. 249–268. DOI: 10.1076/csed.13.4.249.17496.

[Le16] Nguyen-Thinh Le. „A Classification of Adaptive Feedback in Educational Systems for Programming“. In: *Systems* 4.2 (2016). DOI: 10.3390/systems4020022.

[LLP13] Nguyen-Tinh Le, Frank Loll und Niels Pinkwart. „Operationalizing the Continuum between Well-Defined and Ill-Defined Problems for Educational Technology“. In: *IEEE Transactions on Learning Technologies* 6.3 (2013), S. 258–270. DOI: 10.1109/TLT.2013.16.

[LMM98] José Paulo Leal, Nelma Moreira und Leal Nelma Moreira. *Automatic Grading of Programming Exercises*. Techn. Ber. Kurnia, A.; Cheang, B., Lim, A „Online Judge“, Computers und Education, 1998.

[LP14] Nguyen-Thinh Le und Niels Pinkwart. „Towards a classification for programming exercises“. In: *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science*. 2014.

[Mat94] Urs von Matt. „Kassandra: The Automatic Grading System“. In: *SIGCUE Outlook* 22.1 (Jan. 1994), S. 26–40. DOI: 10.1145/182107.182101.

[Nar08] Susanne Narciss. „Feedback strategies for interactive learning tasks“. In: *Handbook of research on educational communications and technology*. Hrsg. von David Jonassen u. a. Bd. 3. Routledge, 2008, S. 125–144.

[Nik+04] Jussi Nikander u. a. „Visual algorithm simulation exercise system with automatic assessment: TRAKLA2“. In: *Informatics in Education – An International Journal* Vol. 3_2 (2004), S. 267–288.

[Pea+07] Arnold Pears u. a. „A Survey of Literature on the Teaching of Introductory Programming“. In: *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '07. ACM, 2007, S. 204–223. DOI: 10.1145/1345443.1345441.

[Pey+00] Christoph Peylo u. a. „A Web-based intelligent educational system for PROLOG“. In: *Proceedings of the International Workshop on Adaptive and Intelligent Web-Based Education Systems held in conjunction with ITS 2000 Montreal, Canada.* 2000, S. 85–96.

[Pil03] Nelishia Pillay. „Developing Intelligent Programming Tutors for Novice Programmers“. In: *ACM SIGCSE Bulletin* 35.2 (Juni 2003), S. 78–82. DOI: 10.1145/782941.782986.

[Rin99] M. Rintala. *Tutnew memory management library*. Englisch. 1999. URL: https://www.cs.tut.fi/~bitti/tutnew-www/english/.

[RK15] Kelly Rivers und Kenneth R. Koedinger. „Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor“. In: *International Journal of Artificial Intelligence in Education* (2015), S. 1–28. DOI: 10.1007/s40593-015-0070-z.

[SG14] Michael Striewe und Michael Goedicke. „A Review of Static Analysis Approaches for Programming Exercises“. In: *Computer Assisted Assessment. Research into EAssessment: International Conference, CAA 2014, Zeist, The Netherlands, June 30 – July 1, 2014. Proceedings*. Springer International Publishing, 2014, S. 100–113. DOI: 10.1007/978-3-319-08657-6_10.

[SKM13] Juha Sorva, Ville Karavirta und Lauri Malmi. „A Review of Generic Program Visualization Systems for Introductory Programming Education“. In: *Trans. Comput. Educ.* 13.4 (Nov. 2013), 15:1–15:64. DOI: 10.1145/2490822.

[SOP11] Sven Strickroth, Hannes Olivier und Niels Pinkwart. „Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben?“ In: *DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik*. Bd. 188. LNI. GI, 2011, S. 115–126.

[SQF08] Mate' Sztipanovits, Kai Qian und Xiang Fu. „The Automated Web Application Testing (AWAT) System“. In: *Proceedings of the 46th Annual Southeast Regional Conference on XX*. ACM-SE 46. ACM, 2008, S. 88–93. DOI: 10.1145/1593105.1593128.

[SS08] Merlin Suarez und Raymund Sison. „Automatic Construction of a Bug Library for Object-Oriented Novice Java Programmer Errors“. In: *Intelligent Tutoring Systems: 9th International Conference, ITS 2008, Montreal, Canada, June 23-27, 2008 Proceedings*. Hrsg. von Beverley P. Woolf u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 184–193. DOI: 10.1007/978-3-540-69132-7_23.

[Tru07] Nghi Truong. *A Web-Based Programming Environment for Novice Programmers*. 2007. URL: http://eprints.qut.edu.au/16471/1/Nghi_Truong_Thesis.pdf.

[Val04] David W. Valentine. „CS educational research: a meta-analysis of SIGCSE technical symposium proceedings“. In: *ACM SIGCSE Bulletin* 36.1 (2004), S. 255–259.