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

6 Automatisierte Bewertung in der UML-Modellierung

Marianus Ifland und Frank Puppe

Zusammenfassung
Die Modellierung von UML-Klassen- und Aktivitätsdiagrammen ist ein wichtiges Thema im Informatikstudium und verwandten Studiengängen. Da deren Bewertung zeitaufwändig ist, andererseits das Stellen möglichst vieler Übungsaufgaben den Lernerfolg steigert, ist eine automatisierte Bewertung erstrebenswert. Während bei Klassendiagrammen ein textueller Vergleich mit einer Musterlösung im Vordergrund steht, kann man Aktivitätsdiagramme darüber hinaus in Programmcode übersetzen und ausführen. Wir präsentieren Übungssysteme zur Modellierung von Klassen- und Aktivitätsdiagrammen für Studierende mit generiertem Feedback auf der Basis von Musterlösungen sowie deren Evaluation im Rahmen von Übungen zur Vorlesung Softwaretechnik der Universität Würzburg.

6.1 Einleitung

Die Programmierausbildung ist ein zentrales Element im Informatikstudium und informatiknaher Studiengänge. Dabei soll auch ein Denken über Programme auf höheren Abstraktionsebenen vermittelt werden. Dafür eignet sich die UML-Notation ([RJB10], [Bal05]). Mit Klassendiagrammen wird die Datenstruktur übersichtlich dargestellt, mit Aktivitätsdiagrammen kann die Programmlogik sowohl auf abstrakter als auch auf konkreter, ausführbarer Ebene modelliert werden. Die visuelle Darstellung erleichtert dabei das Verständnis und abstrahiert von syntaktischen Details der Programmiersprache. Der erste Schritt bei der Modellierung ist meist die Definition der Datenstrukturen mit Klassendiagrammen, die auch einfach auf Entity-Relationship-Diagramme für den Datenbankentwurf abgebildet werden können. Dabei werden neben Klassen und deren Attributen und Relationen (Assoziationen und Vererbungsbeziehungen) auch die wichtigsten Operationen eingetragen, die die Grundlage für die Definition der Aktivitätsdiagramme

darstellen. Letztere sind ein gutes didaktisches Mittel, um das Denken über große, aber auch kleine Programme, wie sie in Anfängerlehrveranstaltungen genutzt werden, zu fördern.

Wir präsentieren Übungsprogramme für die Bewertung von Klassen- und Aktivitätsdiagrammen und deren Evaluation in Kap. 6.2 und 6.3 (eine ausführliche Darstellung findet sich in [Ifl14]). Eine besondere Schwierigkeit ist die Bewertung von Folgefehlern. Daher basiert die Bewertung von Aktivitätsdiagrammen auf vordefinierten Klassendiagrammen und noch nicht auf einer Kopplung beider Bewertungen, was auch unserer Praxis beim Stellen von Übungsaufgaben entspricht.

6.2 Das Einsatzszenario „Automatische Bewertung von Klassendiagrammen“

6.2.1 Stand der Forschung

Einen Ansatz zur Bewertung von Klassendiagrammen, der auf dem Vergleich zu einer Musterlösung basiert, stellen [ASI07] mit dem UML Class Diagram Assessor (UCDA) vor. Dabei werden die Klassendiagramme von Dozenten und Studierenden mit dem kommerziellen UML-Werkzeug (Rational Rose) erstellt. Die von UCDA zur Überprüfung der Lernerlösungen anhand der Musterlösung verwendete Softwarekomponente besteht aus drei Modulen, die sequenziell ausgeführt werden, wobei jedes Modul den Studierenden im Falle von Fehlern Feedback gibt. Dabei wird das zweite (bzw. dritte) Modul nur dann ausgeführt, wenn das erste (bzw. zweite) keine Fehler liefert. Studierende können dabei ihr Diagramm iterativ verbessern. Zuerst überprüft das Class Structure Analysis Module die Lernerlösung beispielsweise auf die korrekte Anzahl von Klassen, die korrekte Anzahl von Attributen innerhalb von Klassen oder die Korrektheit von Parametertypen. Das Verification Process Module generiert Fehlermeldungen bezüglich der Relationen zwischen den Klassen, also ob die Anzahl der Relationen insgesamt korrekt ist, ob die Anzahl der Relationen bestimmten Typs (Assoziation, Generalisierung, Aggregation, Komposition) korrekt ist und ob konkrete Relationen zwischen Klassen vorhanden sind. Das Language Checking Module überprüft, ob bei der Benennung von Klassen und Attributen Substantive und bei der Benennung von Operationen Verben verwendet wurden. Eine Möglichkeit, wie mehrere Musterlösungen verwendet werden können, wird nicht gezeigt. Von einer Evaluation wird nicht berichtet.

[Sol+10] stellen einen Ansatz zur Klassendiagrammbewertung vor, der in das webbasierte ACME-DB Framework integriert ist. Dabei handelt es sich um eine E-Learning-Plattform der Universität Girona (Spanien), welche in Kursen zu Datenbanken eingesetzt wird. Es können mehrere Musterlösungen pro Aufgabe hinterlegt werden, wobei diese direkt in einer XML-ähnlichen Struktur eingegeben werden. In einem webbasierten Editor können Lernende dann zu den gestellten Aufgaben Klassendiagramme eingeben und prüfen lassen. Nach einer Prüfung durch das Korrekturmodul werden den Lernenden entsprechende Fehler in ihrem Klassendiagramm in Textform mitgeteilt. Fehler sind beispielsweise eine unkorrekte Anzahl an Klassen, falsch benannte Klassen oder falsche Werte bei Kardinalitäten. Nach einer Prüfung kann die eigene Lösung weiter verbessert werden, so dass sich Studierende in einem iterativen Prozess der Musterlösung annähern. Es bleibt unklar, wie das Korrekturmodul den Vergleich von Lernerlösung zu Musterlösungen durchführt, also beispielsweise wie erkannt wird, ob eine bestimmte Klasse aus der Musterlösung auch in der Lernerlösung vorkommt. Ebenfalls unklar bleibt, wie entschieden wird, welche Musterlösung bei einer Prüfung als Referenzlösung verwendet wird. Das System wurde in einem Parallelgruppenvergleich mit 48 Studierenden evaluiert. Eine Gruppe bereitete sich mit 4 Aufgaben aus der ACME-Lernplattform auf eine Prüfung vor, während die andere Gruppe ermutigt wurde, die Aufgaben per Hand zu lösen und bei Fragen das Büro des Dozenten aufzusuchen. Die Prüfung beinhaltete dabei eine Aufgabe, die ähnlich zu den zuvor gestellten Aufgaben war. Die Gruppe, welche die Lernplattform benutzen sollte, erzielte dabei im Durchschnitt leicht bessere Ergebnisse als die zweite Gruppe, doch waren die Unterschiede nicht signifikant. Die Eindrücke der Studierenden wurden insgesamt als positiv beschrieben.

Einen Ansatz ohne explizite Musterlösungen verfolgen [SG11a]. Klassendiagramme werden hier als formale Graphen interpretiert, welche mit einer geeigneten Anfragesprache, einer sogenannten Graph Query Language, untersucht werden können. Als Anfragesprache wird GReQL verwendet, mit welcher Anfragen gestellt werden können, die über die Existenz oder Nichtexistenz von Elementen anhand deren Typs oder deren Eigenschaften Auskunft geben. Um eine Aufgabe zu stellen, muss nun seitens des Aufgabenstellers neben der Domänenbeschreibung eine Reihe von Regeln mittels solcher Abfragen definiert werden. Dabei muss bei jeder Abfrage markiert werden, ob es sich um ein gewünschtes oder nicht gewünschtes Element handelt, damit bei der automatischen Korrektur entsprechende Fehlermeldungen gegeben werden können. Die Regeln können dabei logisch kombiniert werden, so dass beispielsweise die Akzeptanz alternativer Schreibweisen abgebildet werden kann. Es ist möglich, mit diesen Regeln auch stilistische Eigenschaften der Lernerlösung zu prüfen, beispielsweise ob die

Namen aller vorkommenden Klassen mit Großbuchstaben beginnen. Da Dozenten, die Aufgaben zu UML-Klassendiagrammen stellen, GReQL im Allgemeinen wohl nicht beherrschen, müssen diese Regeln also von entsprechenden Experten eingegeben werden. Es wird hier nur die Gleichheit der entsprechenden Namen geprüft. Alternative Schreibweisen, die beispielsweise durch Flexion oder Tippfehler entstehen, müssten in den Regeln explizit angegeben werden. Für eine Beispielaufgabe mussten 17 solche Regeln definiert werden. 5 davon sind allerdings allgemeine Regeln, die auch in anderen Aufgaben weiter verwendet werden können, so dass speziell für diese Aufgabe 12 Regeln definiert werden mussten. Für die Auswertung, also die Berechnung der Punktzahl für eine Lernerlösung, gibt es nun zwei Strategien. Entweder wird von 0 Punkten ausgegangen und Studierende erhalten Punkte für jede Regel, die nach einem gewünschten Element sucht und dieses auch findet (pessimistischer Ansatz). Oder es wird von der maximalen Punktzahl ausgegangen und den Studierenden werden für jede Regel Punkte abgezogen, die nach einem gewünschten Element sucht und dieses nicht findet (optimistischer Ansatz). Der Punktwert muss bei der Definition der entsprechenden Regel hinterlegt werden. In einem Experiment, bei welchem der pessimistische Ansatz verwendet wurde, wurden sechs Gruppen von Studierenden eine Aufgabe vorgelegt, die diese in Teamarbeit lösten. Die automatischen Bewertungen der Lernerlösungen reichten von 67 bis 91 von 100 Punkten. Diese Bewertungen lagen im gleichen Bereich der Bewertungen, die ein menschlicher Korrektor vorgenommen hatte, dem die Regeln zur automatischen Bewertung nicht bekannt waren. Über die Korrelation der Bewertungen der beiden Methoden wird allerdings keine Aussage gemacht. Es zeigte sich außerdem, dass die definierten Regeln nicht alle Aspekte abdecken konnten, die bei einer manuellen Bewertung betrachtet würden. Dies wird nicht als Problem der Technik beschrieben, sondern als Problem der Tatsache, dass die Regeln geschrieben werden, ohne die Lernerlösungen zu kennen. Es ist also nötig, die Regeln in einem iterativen Prozess nach der Bewertung einiger Lernerlösungen anzupassen. Unter der Voraussetzung, dass eine praxiserprobte Menge an allgemeinen Regeln besteht und dass Dozenten die Fähigkeit besitzen, in akzeptabler Zeit gute Regeln zu erstellen, verspricht dieser Ansatz gute Ergebnisse. Positiv hervorzuheben ist, dass anhand der Anfrageergebnisse ein eindeutiges Feedback, beispielsweise in natürlichsprachiger Form, generiert werden kann, in welchem mitgeteilt wird, welche Regeln verletzt wurden. Das Fehlen einer Musterlösung im Feedback ist allerdings durchaus als Nachteil zu sehen. Ebenfalls bleibt die Problematik verschiedener Schreibweisen von Elementen ohne deren explizite Angabe ungelöst, wobei die Autoren andeuten, dass es eine Möglichkeit gibt, GReQL um entsprechende Funktionen zu erweitern. Als problematisch anzusehen ist die komplex erscheinende Eingabe der Regeln, die

eine Hürde für Dozenten darstellen kann, ein entsprechendes Trainingssystem einzusetzen.

6.2.2 Eigener Ansatz

Wir verwenden ein Überdeckungsmaß zum Vergleich von Lerner- und Musterlösung. Dabei ist ein Klassendiagramm D ein 6-Tupel $(K_D, V_D, A_D, M_D, T_D, O_D)$ mit

- K_D: Menge aller in D enthaltenen Klassen,
- V_D: Menge aller in D enthaltenen Vererbungsbeziehungen,
- A_D: Menge aller in D enthaltenen Assoziationen,
- M_D: Menge aller in D enthaltenen Multiplizitäten ($|M_D| = |A_D|$),
- T_D: Menge aller in D enthaltenen Attribute,
- O_D: Menge aller in D enthaltenen Operationen.

Beim Vergleich eines vom Lerner erstellten Klassendiagramms mit einer Musterlösung wird zunächst ein Zuordnungsproblem gelöst, dass möglichst jedem Element des Lernerklassendiagramms einem Element der Musterlösung zuordnet. Anschließend werden die Zuordnungen mit einem Bewertungsmaß B für jedes Tupelelement mit einer Zahl zwischen 0 und 1 bewertet:

- $B_K : K \times K \mapsto [0..1]$,
- $B_V : V \times V \mapsto [0..1]$,
- $B_A : A \times A \mapsto [0..1]$,
- $B_M : M \times M \mapsto [0..1]$,
- $B_T : T \times T \mapsto [0..1]$,
- $B_O : O \times O \mapsto [0..1]$.

Die einzelnen Bewertungstypen werden für die Tupelelemente gewichtet. Der Gesamtscore kann als Prozentsatz erzielter Punkte zu maximal möglichen Punkten berechnet werden. Alternativ kann er auch als Abzugsmodell („optimistischer Ansatz“) von der Maximalpunktzahl umgesetzt werden, dass für jeden Fehler Punkte

abgezogen werden. Ein Fehler bedeutet hier, dass für ein Element aus der Musterlösung kein passendes Element der Lernerlösung gefunden wurde. Die Bewertungsfunktionen für Assoziationen und Vererbungsbeziehungen zwischen Klassen und für die Multiplizitäten sind leicht zu berechnen, falls die Klassen richtig erkannt wurden. Dagegen ist die Bewertung von Klassen-, Attribut- und Operatornamen schwieriger, da Studierende bei der Namensgebung nicht eingeschränkt sind. Um das Problem zu umgehen, werden die Aufgaben so gestellt, dass die jeweiligen Namen im Aufgabentext explizit genannt werden und die Studierenden darauf hingewiesen werden, wenn möglich Namen aus der Aufgabenstellung zu übernehmen. Trotzdem gibt es in Lösungen von Übungsaufgaben viele abweichende Schreibweisen. Um diese zuordnen zu können, wird die Ähnlichkeit von Zeichenketten zu den Texten der Musterlösung heuristisch berechnet, zu der auch Synonyme hinterlegt sein können, indem Standardtechniken wie Ignorieren von Groß-/Kleinschreibung, Stemming und String-Ähnlichkeit auf Basis der Levenshtein-Distanz benutzt werden. Falls Datentypen angegeben sind, werden Grundtypen wie Boolean, String, Zahl, Enumeration, usw. betrachtet. Die Ähnlichkeit von vordefinierten Varianten innerhalb dieser Grundtypen (z. B. für Zahlen: integer, int, double, long, BigDecimal, ...) wird wesentlich höher bewertet als zwischen verschiedenen Grundtypen. Zusätzlich wird bei der Bewertung der Ähnlichkeit von Attribut- und Operatornamen beachtet, ob diese in der gleichen Klasse vorkommen. Abb. 6.1 zeigt zwei Klassendiagramme und Abb. 6.2 die be-

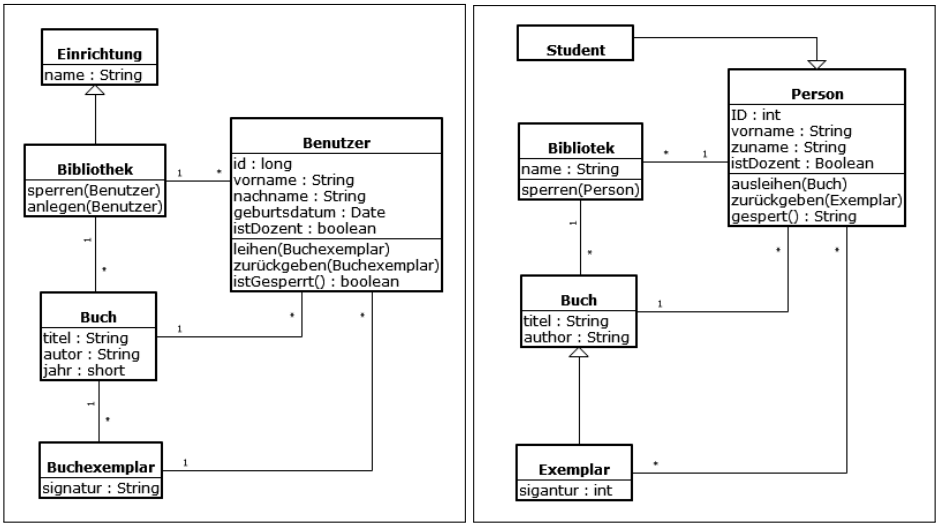

Abbildung 6.1: UML-Klassendiagramm: Beispiel einer Musterlösung (links) und Lernerlösung (rechts)

	A	B	C	D	E	F
1		**Musterlösung**	**Ihre Lösung**	**max. Punkte**	**Bewertung**	**Ihre Punkte**
2	**Klassen**			**10**	**80%**	**8,0**
3	Einrichtung		-	2	0%	0
4	Bibliothek		Bibliotek	2	100%	2
5	Benutzer		Person	2	100%	2
6	Buch		Buch	2	100%	2
7	Buchexemplar		Exemplar	2	100%	2
8						
9	**Generalisierungen:**			**1**	**0%**	**0,0**
10	Bibliothek -> Einrichtung		-	1	0%	0
11						
12	**Assoziationen:**			**10**	**80%**	**8,0**
13	Bibliothek <-> Benutzer		Bibliotek <-> Person	2	100%	2
14	Bibliothek <-> Buch		Bibliotek <-> Buch	2	100%	2
15	Buch <-> Benutzer		Buch <-> Person	2	100%	2
16	Buch <-> Buchexemplar		-	2	0%	0
17	Buchexemplar <-> Benutzer		Exemplar <-> Person	2	100%	2
18						
19	**Kardinalitäten:**			**5**	**50%**	**2,5**
20	Bibliothek <-> Benutzer	1:*	*:1	1	0%	0
21	Bibliothek <-> Buch	1:*	1:*	1	100%	1
22	Buch <-> Benutzer	1:*	1:*	1	100%	1
23	Buch <-> Buchexemplar	1:*	-	1	0%	0
24	Buchexemplar <-> Benutzer	1:*	*:*	1	50%	0,5
25						
26	**Attribute:**			**10**	**64%**	**6,4**
27	Buch	titel : String	titel : String	1	100%	1,0
28	Buch	autor : String	author : String	1	100%	1,0
29	Buch	jahr : short	-	1	0%	0,0
30	Einrichtung	name : String	-	1	0%	0,0
31	Buchexemplar	signatur : String	sigantur : int	1	50%	0,5
32	Benutzer	id : long	ID : int	1	88%	0,9
33	Benutzer	vorname : String	vorname : String	1	100%	1,0
34	Benutzer	nachname : String	zuname : String	1	100%	1,0
35	Benutzer	geburtsdatum : Date	-	1	0%	0,0
36	Benutzer	istDozent : boolean	istDozent : Boolean	1	100%	1,0
37						
38	**Methoden:**			**5**	**70%**	**3,5**
39	Bibliothek	sperren(Benutzer)	sperren(Person)	1	100%	1,0
40	Bibliothek	anlegen(Benutzer)	-	1	0%	0,0
41	Benutzer	leihen(Buchexemplar)	ausleihen(Buch)	1	75%	0,8
42	Benutzer	zurückgeben(Buchexemplar)	zurückgeben(Exemplar)	1	100%	1,0
43	Benutzer	istGesperrt() : boolean	gespert() : String	1	75%	0,8
44						
45						
46	**Gesamt:**			**41**	**69%**	**28,4**

Abbildung 6.2: Beispiel eines Feedbackdokuments für Muster und Lernerlösung aus Abbildung 6.1

rechneten Ähnlichkeiten. Als Autorenwerkzeug wurde ein spezieller Editor innerhalb des an der Universität eingesetzten fallbasierten Trainingssystems CaseTrain [Hö+09] entwickelt (s. Abb. 6.3).

6.2.3 Evaluation

Das System wurde beim Einsatz in der Vorlesung Softwaretechnik der Universität Würzburg im Sommersemester 2013 und 2014 evaluiert. Dabei zeigen wir zunächst den Effekt des komplexen Zuordnungsmaßes im Vergleich zu einem ein-

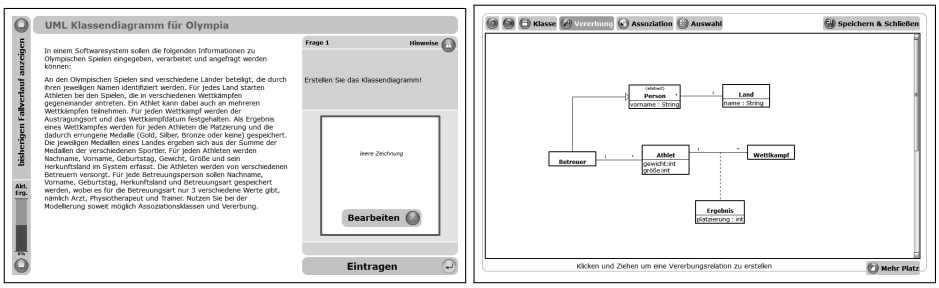

Abbildung 6.3: Editor zum freien Zeichen von UML-Klassendiagrammen in CaseTrain (links: Aufgabenstellung, rechts: Editor, der nach Klicken auf den Button „Bearbeiten“ im linken Fenster erscheint).

	Anz.	**RP**	**FP**	**FN**	**Precision**	**Recall**	**F_1-Maß**	**Fehlerr.**
SoSe13 trivial	2849	1699	0	291	100 %	85,4 %	92,6 %	10,2 %
SoSe13 komplex	2849	1969	16	21	99,2 %	98,9 %	99,1 %	1,3 %
SoSe14 trivial	1976	1282	0	278	100 %	82,2 %	90,2 %	14,1 %
SoSe14 komplex	1976	1540	24	20	98,5 %	98,7 %	98,6 %	2,2 %
gesamt trivial	4825	2981	0	569	100 %	84,0 %	**91,3 %**	**11,8 %**
gesamt komplex	4825	3509	40	41	98,9 %	98,8 %	**98,9 %**	**1,7 %**

Tabelle 6.1: Ergebnisse der manuellen Evaluation der Zuordnungen von 2849 Klassen im Sommersemester 2013 und 1976 Klassen im Sommersemester 2014 (RP=Richtig Positiv; FP=Falsch Positiv; FN=Falsch Negativ).

fachen (trivialen) Zuordnungsmaß (Tab. 6.1), das eine exakte Übereinstimmung der Namen von Klassen, Attributen und Methoden verlangt, wie sie in der Aufgabenstellung explizit gefordert wurden. Dabei sind Assoziationen, Vererbungen und Kardinalitäten wie oben erwähnt einfach zu bewerten und werden hier nicht betrachtet. Als Goldstandard diente eine manuelle Zuordnung von Dozenten. Das triviale Maß hat eine Precision von 100% (wenn zwei Namen identisch sind, ist die Zuordnung korrekt), aber nur einen Recall von 84%, d. h. es werden viele Zuordnungen nicht erkannt. Mit dem komplexen Maß, das Namenszuordnungen aufgrund der oben erwähnten Heuristiken berechnet, steigt der Recall um 14,4 Prozentpunkte auf 98,8% an, wobei die Precision nur um 1.1 Prozentpunkte auf 98,9% fällt.

Obwohl das Programm nicht perfekt arbeitet, wurde es von den Studierenden akzeptiert. In einer Umfrage im Sommersemester 2013, die zwei Wochen nach

Aspekt	Ø Schulnote (1-6)
Konzept des Tools, unabhängig von der Implementierung	2,25
Umsetzung des Tools, Gesamtnote	2,61
Aspekt Bedienbarkeit (Einarbeitung, intuitive Bedienbarkeit)	2,47
Aspekt Technik (Verfügbarkeit, Robustheit, Effizienz)	3,57
Aspekt Inhalt der Übungsaufgaben	2,18
Aspekt Fairness der Bewertung der Übungsaufgaben	2,59
Aspekt hilfreiche Erklärungen als Text oder nützliches Feedback	2,95

Tabelle 6.2: Umfrageergebnisse in Schulnoten zu verschiedenen Aspekten des freien Zeichnens von Klassendiagrammen bei Studierenden im Sommersemester 2013 (n=240)

Bearbeitung der Aufgabe durchgeführt wurde, wurde das Programm mit Schulnoten zwischen gut und befriedigend hinsichtlich der verschiedenen Aspekte bewertet; lediglich die Technik wurde um eine Notenstufe schlechter bewertet, da das Programm noch nicht robust genug war (Tab. 6.2). Die Umfrage 2014 war breiter angelegt und daher nicht direkt mit der von 2013 vergleichbar, hatte aber ähnliche Bewertungen (bis auf die Technik, die um eine Notenstufe besser ausfiel). 90% befürworteten einen weiteren Einsatz des Trainingssystems für UML-Klassendiagramme.

6.2.4 Diskussion und Verbesserungen

Die Evaluationen haben gezeigt, dass es noch viel Potenzial für Verbesserungen gibt. Dazu gehört der regelbasierte Ansatz von [SG11a], der spezifische Schwächen des Überdeckungsansatzes ausgleichen kann:

- Regeln, die Konventionen in UML bewerten, zum Beispiel dass Klassennamen Substantive im Singular sein sollen.

- Regeln, die zu viele Angaben wie zum Beispiel mehr Attribute oder Methoden, als in der Musterlösung angegeben, und die im Allgemeinen toleriert werden, in speziellen Fällen negativ bewerten (z. B. überflüssige Assoziationen zwischen Klassen).

- Regeln, die für typische Fehler in der Aufgabe ein angemessenes Feedback geben, zum Beispiel wenn in Lernerlösungen Vererbungsbeziehungen fehlen (die Überdeckungsmetrik liefert da oft zu schlechte Bewertungen).

- Spezielle Regeln für Besonderheiten der Aufgabenstellung, dass man zum Beispiel zwar Buchexemplare, aber keine Bücher ausleihen kann, wenn beide Klassen im Klassendiagramm vorkommen.

Weitere Verbesserungen beinhalten verfeinerte Verfahren zur Berechnung der Levenshtein-Distanz zwischen Namen, das Erkennen von Teilwörtern in Komposita, das zum Beispiel auch für die Angabe von Synonymen für Teilwörtern genutzt werden kann, oder die variable Zugehörigkeit von Operationen zu verschiedenen Klassen, um die aufwändige Angabe verschiedener Musterlösungen zu verringern.

6.3 Das Einsatzszenario „Automatische Bewertung von Aktivitätsdiagrammen“

Das Ziel der Modellierung von Aktivitätsdiagrammen ist, die wesentliche Funktionalität des Programms übersichtlich darzustellen. Bei großen Programmen ist eine starke Abstraktion erforderlich. Kleine Programme können dagegen auch in vollem Umfang exakt abgebildet werden. Beides erlaubt die Syntax der Aktivitätsdiagramme in UML. Typische Lernziele beim Einüben der Erstellung von Aktivitätsdiagrammen sind:

- Erstellen von syntaktisch korrekten Aktivitätsdiagrammen,
- Verwendung von Bedingungen und entsprechenden Verzweigungen,
- Verwendung von Schleifen,
- Übergabe und Rückgabe von Parametern,
- Kapselung häufig verwendeter Subroutinen,
- Zugriff auf Objektattribute,
- Verständnis des Zusammenhangs von Diagrammen und Programmcode,
- algorithmisches Problemlösen.

Durch die Verwendung von Subroutinen, deren konkretes Verhalten vordefiniert ist, lassen sich Aktivitätsdiagramme bei geeigneter Syntax direkt in lauffähigen Programmcode übersetzen.

6.3.1 Stand der Forschung

Für die einzelnen Aspekte gibt es viel Literatur. So sind in der Programmierausbildung Systeme, die eingegebenen Programmcode überprüfen und entsprechendes Feedback generieren, bereits etabliert. Ähnliches gilt für die automatische Codegenerierung aus UML. [UN09] stellen ein Werkzeug namens „UJECTOR“ vor, das UML-Klassen-, Sequenz-, und Aktivitätsdiagramme in Java-Code übersetzen kann. Das Werkzeug wird mit bestehenden kommerziellen und quelloffenen Werkzeugen verglichen. Es wird belegt, dass der generierte Programmcode funktionsfähig und verständlich ist. Ein weiteres Werkzeug, welches UML-Diagramme in Java-Code übersetzt, ist „Fujaba“ [NNZ00]. Der Name ist ein Akronym für „From UML to Java and back again“. Auch eine Rückübersetzung des modifizierten Codes in Diagramme ist damit möglich. [GR11] stellen ebenfalls einen Ansatz zur Generierung von Programmcode aus Aktivitätsdiagrammen vor. Programmieren mit Aktivitätsdiagrammen kann als visuelle Programmierung aufgefasst werden. Dabei werden Programme in zwei- oder mehrdimensionaler Weise spezifiziert [Mye90]. Bekannte Systeme zur Programmierausbildung mit visueller Programmierung und Programmvisualisierung sind „Alice“, „Greenfoot“ und „Scratch“. Alice ist eine um das Jahr 2000 erschienene Programmiersprache mit zugehöriger Entwicklungsumgebung, mit der das Erscheinungsbild und das Verhalten von Personen und Objekten in einer virtuellen dreidimensionalen Welt modifiziert werden kann [CDP00]. Ein Drag-and-Drop-Konzept verhindert hier die Erstellung syntaktisch unkorrekter Programme. Alice wird an „hunderten Hoch- und Sekundarschulen“ eingesetzt [Fin+10]. Einige Jahre später wurde Greenfoot [Kö10] veröffentlicht, eine Entwicklungsumgebung mit didaktischem Fokus, die darauf spezialisiert ist, interaktive, grafische Anwendungen zu erstellen. Die Programme werden hier direkt mit Java-Code entwickelt, die Ausführung wird visualisiert, wobei parallel die Vererbungsstruktur der beteiligten Klassen angezeigt wird. Im Jahr 2007 erschien Scratch [Mal+10], eine visuelle Programmierumgebung für den schulischen Einsatz, deren Zielgruppe bei Kindern und Jugendlichen von 8 bis 16 Jahren liegt. Der Fokus liegt dabei auf dem Umgang mit Medien, also mit Bildern, Geräuschen, Musik und Videos. Scratch ist mittlerweile weit verbreitet; laut Angaben auf der offiziellen Webseite waren im August 2016 fast 13 Millionen Nutzer registriert[1]. Auch für die Programmierung von Anwendungen für mobile Endgeräte („Apps“) wurde mit dem Google „App Inventor“ bereits eine visuelle Programmierumgebung in der Lehre eingesetzt [WS11].

1 http://scratch.mit.edu/statistics/ (August 2016)

6.3.2 Eigener Ansatz

Im hier vorgestellten Trainingssystem „WARP“ (Würzburger Aktivitätsdiagramm Roboter Programmierung) [Ifl+14a] werden Aktivitätsdiagramme zur Definition des Verhaltens eines virtuellen Roboters erstellt und direkt in Java-Code übersetzt. Dieser Java-Code wird in einer vordefinierten Simulationsumgebung ausgeführt, wobei der Roboter eine bestimmte Aufgabe erfüllen soll. Dabei wird den Lernenden Feedback in vier Ebenen angeboten:

- Erste Feedbackebene: Falls das Diagramm syntaktisch nicht valide ist, werden Fehler direkt im Diagramm mit einem textuellen Kommentar angezeigt (Abb. 6.4, oben).
- Zweite Feedbackebene: Falls der aus dem Diagramm automatisch erzeugte Java-Code syntaktische Fehler hat, werden entsprechende Hinweise generiert (Abb. 6.4, unten).
- Dritte Feedbackebene: Der Java-Code wird in der Robotersimulationsumgebung RoSE [Her13] ausgeführt und eine Animation erzeugt, welche die Studierenden schrittweise abspielen können (Abb. 6.5).
- Vierte Feedbackebene: Die Studierenden erhalten eine Rückmeldung, ob die Aufgabe korrekt gelöst wurde. Zudem werden Metriken über die primitiven Roboteraktionen angezeigt, zum Beispiel wie viele Schritte der Roboter gebraucht hat.

WARP wurde als Webanwendung implementiert, die sich leicht an Lernmanagementsysteme wie Moodle koppeln lässt. Der Editor besteht aus vier Teilen. Zunächst gibt der Benutzer in einer grafischen Oberfläche ähnlich zu dem UML-Klassendiagrammeditor (s. Abb. 6.3) sein Aktivitätsdiagramm ein. Dabei werden folgende Knoten unterstützt:

- Startknoten und Endknoten,
- Aktionsknoten (mit konkretem Java-Code),
- Verzweigungsknoten und Verbindungsknoten,
- strukturierte If-Then-Knoten,
- strukturierte Schleifenknoten: For-Do-Knoten, Do-While-Knoten, While-Do-Knoten.

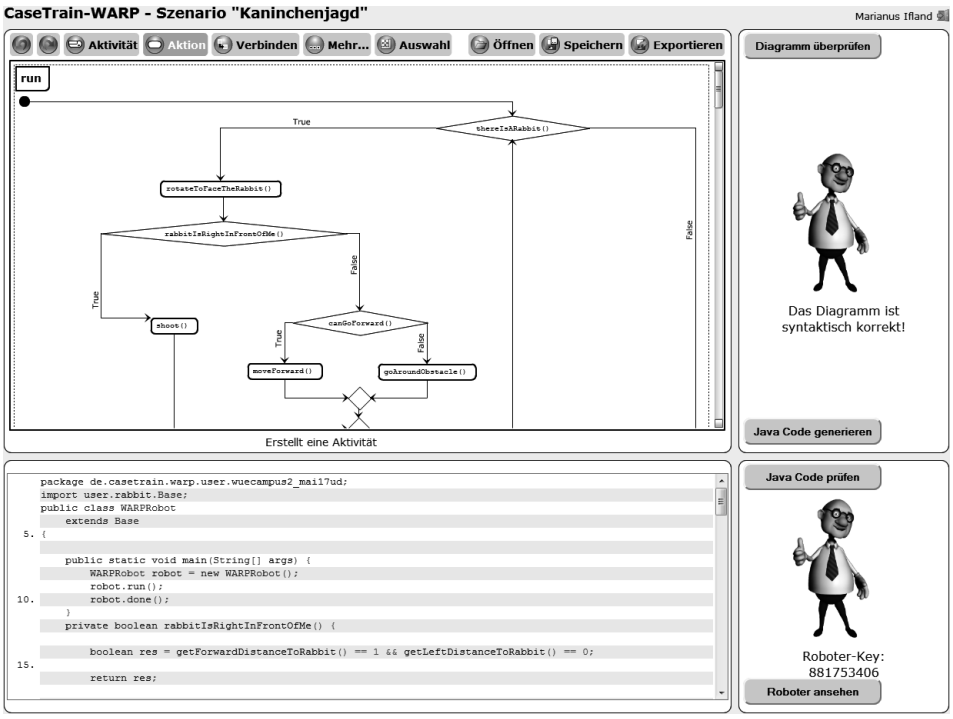

Abbildung 6.4: CaseTrain-WARP Editor: Vier Bereiche des zentralen Dialogs, die zur Erstellung der Aktivitätsdiagramme dienen. Links oben: grafischer Editor, rechts oben: Feedback zur Syntax des Diagramms, links unten: automatisch generierter Java-Code, rechts unten: Feedback zur Syntax des Java-Codes. Eventuelle Fehler im grafischen Editor oder im generierten Code würden links rot markiert und rechts kommentiert. Wenn der Java-Code korrekt ist, wird ein entsprechendes Roboterprogramm für eine „Arena“ erstellt, die sich der Benutzer anschauen kann (s. Abb. 6.5 mit komplexerer Aufgabe der Kaninchenjagd).

Das Programm überprüft in der ersten Feedbackebene nur die syntaktische Korrektheit des gezeichneten Diagramms, zum Beispiel ob alle Knoten die passende Anzahl von eingehenden und ausgehenden Kanten haben. Wenn das Programm syntaktisch korrekt ist, kann es der Benutzer in Java-Code übersetzen lassen. Dabei wird der erzeugte Java-Code auf syntaktische Korrektheit überprüft und gegebenenfalls Fehler markiert und kommentiert. Wenn der Java-Code korrekt ist, kann sich der Benutzer einen Roboter (d. h. ein ausführbares Programm) erzeugen, den er in einer Arena (Abb. 6.5) ausführen kann. Die Ausführung kann schrittwei-

se gesteuert werden, damit der Benutzer sich das Verhalten an kritischen Stellen genauer anschauen kann, um eventuelle Laufzeitfehler zu finden. Schließlich werden als vierte Feedbackebene noch Metriken erzeugt, die Aussagen über die Programmqualität machen (z. B. wie viele Schritte zur Erledigung der Aufgabe benötigt wurden). Als eine spezielle Art von Metrik gibt es auch Wettbewerbsspiele, bei denen zwei Roboter von zwei Benutzern im Wettbewerb gegeneinander antreten (in Abb. 6.5 sollen sie ein sich zufällig bewegendes Kaninchen jagen).

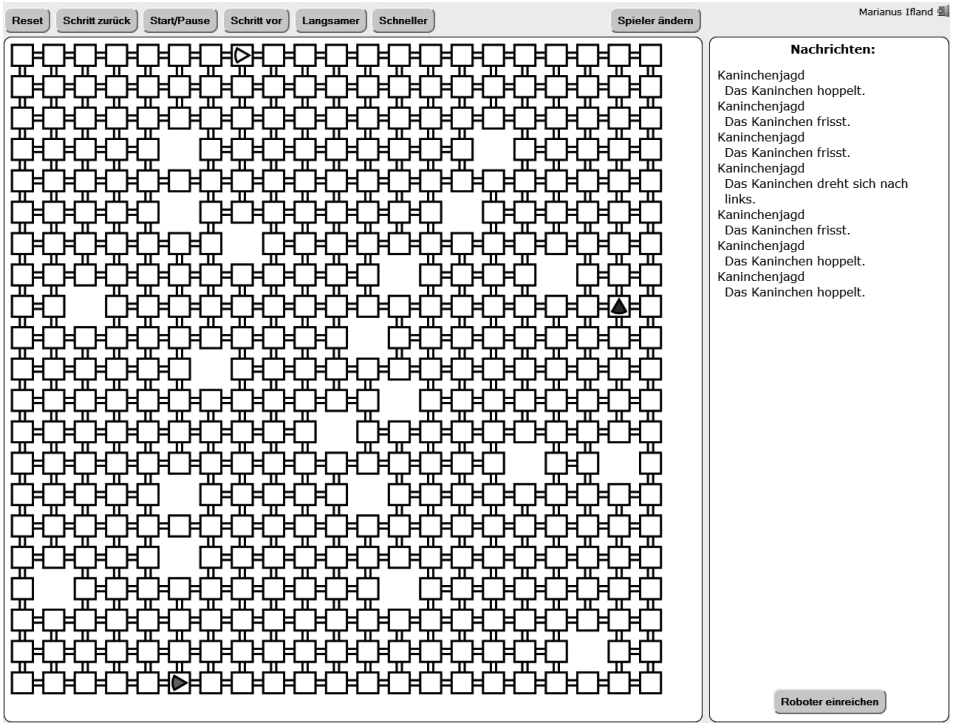

Abbildung 6.5: CaseTrain-WARP Arena: Die Abbildung zeigt einen Zustand der Umgebung eines Multiplayer-Szenarios, in welchem die Aufgabe für die beiden Spieler (in der Arena mit unterem rotem und oberen gelben Pfeil markiert) darin besteht, sich zu einem sich zufällig bewegenden Kaninchen (blauer Pfeil, Mitte rechts) zu bewegen und es dann zu erlegen. Auf der rechten Seite werden Statusmeldungen angezeigt.

6.3.3 Evaluation

Im Sommersemester 2013 wurden in der Vorlesung Softwaretechnik an der Universität Würzburg Aufgaben zu 3 Szenarien angeboten: Zeichnen eines Rechtecks als Einstiegsaufgabe, Markieren jedes begehbaren Feldes einer Arena mit nichtbegehbaren Feldern („Staubsauger“), Finden eines Weges zu einem vorgegebenen Ziel („Labyrinth“). Im Sommersemester 2014 kamen noch zwei anspruchsvollere Aufgaben dazu: die Staubsauger-Aufgabe wurde um stochastische Elemente erweitert. Die Labyrinth-Aufgabe wurde so erweitert, dass ein bewegliches Ziel („ein Kaninchen“) gejagt werden muss, dessen Position immer abgefragt werden kann. Letztere Aufgabe wurde auch im Wettbewerb zweiter Roboter angeboten.

Die Studierenden konnten Aufgaben sooft abgeben, wie sie wollten. 2013 wurden für ca. 250 verschiedene Studierende für die drei Aufgaben mehr als 12.000 Aktivitätsdiagramme geprüft. Insgesamt konnten die einfachen Aufgaben (Rechteck, Staubsauger, Labyrinth) von ca. 90% der Studierenden 2013 und 98% 2014 erfolgreich gelöst werden. Die schwereren Aufgaben in 2014 (Staubsauger mit Zufallselementen, Kaninchenjagd) wurden von weniger Studierenden bearbeitet, aber auch noch mit knapp 90% erfolgreich gelöst (s. Tab. 6.3).
Die subjektiven Bewertungen der Studierenden aus Umfrageergebnissen zeigen ein gemischtes Bild (s. Tab. 6.4). Während 2013 die Bewertungen ähnlich wie bei den UML-Klassendiagrammen zwischen den Schulnoten gut und befriedi-

2013	**RE**	**ST**	**LAB**	**gesamt**
Anzahl Einreichungen	244	228	206	678
Anteil bestandener Einreichungen	91,8 %	91,7 %	87,3 %	90,4 %
Ø Anzahl Aktivitäten	2,9	3,0	2,4	2,8
Ø Anzahl Knoten	12,8	20,3	13,1	15,4
Ø Anzahl Kanten	11,6	16,3	11,0	13,0

2014	**RE**	**ST**	**STZ**	**LAB**	**KAN**	**gesamt**
Anzahl Einreichungen	256	243	209	221	144	1073
Anteil bestandener Einreichungen	98,0 %	98,8 %	89,5 %	98,2 %	88,2 %	95,3 %
Ø Anzahl Aktivitäten	2,5	3,0	3,8	2,3	3,4	2,9
Ø Anzahl Knoten	17,2	21,5	39,3	12,9	34,5	23,2
Ø Anzahl Kanten	15,0	17,9	30,9	9,9	18,5	18,2

Tabelle 6.3: Erfolgsquoten der Bearbeitungen der drei Aktivitätsdiagrammaufgaben im Sommersemester 2013 (oben) und 2014 (unten) mit Anzahl der Aktivitäten (die Studierenden sollten große Aktivitätsdiagramme in mehrere kleine aufteilen) sowie der Knoten und Kanten pro Aufgabe. (RE = Rechteck, ST = Staubsauger, STZ = Staubsauger mit Zufallselementen, LAB = Labyrinth, KAN = Kaninchen)

Aspekt	2013	2014
Konzept des Tools, unabhängig von der Implementierung	2,28	1,91
Umsetzung des Tools, Gesamtnote	2,79	3,24
Bedienbarkeit (Einarbeitung, intuitive Bedienbarkeit)	2,76	3,59
Technik (Verfügbarkeit, Robustheit, Effizienz)	3,81	3,87
Inhalt der Übungsaufgaben	2,19	2,66

Tabelle 6.4: Ergebnisse der Benutzerumfrage; Mittelwerte von Schulnoten (1-6)

gend liegen (mit Ausnahme wiederum der Technik mit der Note 3,81, die auf mangelnde Robustheit des Programms zurückzuführen ist), haben sich 2014 die Noten insgesamt verschlechtert. Ein möglicher Grund sind die anspruchsvolleren Aufgaben, die auch zu mehr Problemen bei der Technik geführt haben. Das wurde insofern anerkannt, dass das Konzept im SoSe 2014 mit einer besseren Note (1,91) bewertet wurde. Ähnlich wie bei dem UML-Tool hat sich die große Mehrheit für die weitere Nutzung elektronischer Korrekturtools ausgesprochen (2014: 78%). Allerdings wurden auch zahlreiche Verbesserungen angesprochen.

6.3.4 Diskussion und Verbesserungen

Da die Syntax von Aktivitätsdiagrammen weitgehend vorgegeben ist, beziehen sich die Verbesserungen aufgrund der bisherigen Erfahrungen mit dem Einsatz nicht auf die Vereinfachung der Sprache, sondern hauptsächlich auf pragmatische Aspekte des Editors:

- Variables Platzangebot im Editor: Es sollte für den Benutzer möglich sein, die Zeichenfläche nicht nur in der Tiefe, sondern auch in der Breite zu vergrößern. Wie beim Programmieren wachsen auch Aktivitätsdiagramme oft inkrementell in alle Richtungen.
- Verschieben von Elementen in Knoten-Container: Beim herkömmlichen Programmieren kann man mit Copy-and-Paste Code für ein Teilproblem an eine andere Stelle beliebig verschieben. Derzeit wird beim grafischen Programmieren das Verschieben von grafischen Strukturen direkt auf der Zeichenfläche unterstützt, aber nicht in andere Container und insbesondere in strukturierte Knoten hinein, deren Begrenzungen sich dann vergrößern müssten.
- Alternative Eingabe von Name und Ein-/Rückgabe-Parametern einer Aktivität: Momentan werden Name und Ein-/Rückgabe-Parametern einer Akti-

vität mittels der Eingabe einer Methodensignatur in Java-Syntax definiert. Da das Lernziel hauptsächlich in der Erstellung von Aktivitätsdiagrammen und nicht in der Erstellung von Programmcode liegt, sollte die Eingabe der Signaturen vereinfacht werden.

- Einbinden von vorgegebenen Befehlen: Die Eingabe vorgegebener aktorischer und sensorischer Befehle (z. B. Abfrage der Farbe eines Feldes) wird aktuell nur durch die Funktion des Auto-Vervollständigens unterstützt. Die Analyse der aufgetretenen syntaktischen Fehler im generierten Quellcode ergab, dass sich ein großer Teil (etwa 50 %) der meist durch Schreib- oder Tippfehler verursachten Fehler des Typs „Methode existiert nicht“ auf die vorgegebenen Befehle bezog. Beispielsweise wurde statt rotateRight() fälschlicherweise rotateright() oder turnRight() eingegeben. Diesen Fehlern kann vorgebeugt werden, indem für den Editor eine Funktion implementiert wird, mit der Anwender vorgegebene Befehle beispielsweise wie in Alice per Drag-and-Drop [CDP00] eingeben können.

6.4 Fazit und Ausblick

Die praktischen Erfahrungen im Einsatz zeigen, dass die Idee, den Studierenden Werkzeuge bereitzustellen, mit denen sie zu von ihnen erstellten UML-Klassen- und Aktivitätsdiagrammen automatisch generiertes Feedback bekommen, gut ankommt und die weitaus meisten eine Fortsetzung des Angebotes wünschen. Das galt trotz Problemen bei der Robustheit der Programme, was sich in den im Vergleich zu den übrigen Bewertungen um ca. eine Schulnote schlechteren Bewertungen zur Technik niederschlug. Sehr viele Verbesserungsvorschläge beziehen sich auf die Gestaltung des Editors. Bezüglich des automatischen Feedbacks ist die Übersetzung von Aktivitätsdiagrammen in ausführbaren Code, dessen Anwendung in einer Arena visualisiert wird, wesentlich attraktiver als der Vergleich von Klassendiagrammen mit Musterlösungen gemäß einer Überdeckungsmetrik. Diese sollte durch regelbasierte Bewertungen ergänzt werden, die eine höhere Flexibilität ermöglichen. Für Dozenten wäre eine Unterstützung bei der Korrektur von Prüfungsaufgaben sehr attraktiv, was allerdings erst möglich ist, wenn die Prüfungsaufgaben direkt am Computer geschrieben werden.

Literatur für dieses Kapitel

[ASI07] Noraida Haji Ali, Zarina Shukur und Sufian Idris. „A design of an assessment system for UML class diagram“. In: *Proceedings – The 2007 International Conference on Computational Science and its Applications, ICCSA 2007*. 2007, S. 539–544.

[Bal05] Heide Balzert. *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. Heidelberg: Spektrum, 2005.

[CDP00] Stephen Cooper, Wanda Dann und Randy Pausch. „Alice: a 3-D tool for introductory programming concepts“. In: *Journal of Computing Sciences in Colleges* 15 (2000), S. 107–116. DOI: 10.1145/1953163.1953243.

[Fin+10] Sally Fincher u. a. „Comparing Alice, Greenfoot & Scratch“. In: *41st ACM technical symposium on Computer science education*. 2010, S. 192–193. DOI: 10.1145/1734263.1734327.

[GR11] Dominik Gessenharter und Martin Rauscher. „Code Generation for UML 2 Activity Diagrams Towards a Comprehensive Model-Driven Development Approach“. In: *ECMFA'11 Proceedings of the 7th European conference on Modelling foundations and applications*. 2011, S. 205–220.

[Her13] Felix Hermann. *Erweiterung der Robotersimulationsumgebung RoSE auf Multiagentensysteme mit didaktischem Fokus*. Bachelorarbeit. 2013.

[Hö+09] Alexander Hörnlein u. a. „Konzeption und Evaluation eines fallbasierten Trainingssystems im universitätsweiten Einsatz (CaseTrain)“. In: *GMS Medizinische Informatik, Biometrie und Epidemiologie* 5.1 (2009). DOI: 10.3205/mibe000086.

[Ifl+14a] Marianus Ifland u. a. „WARP – ein Trainingssystem für UML-Aktivitätsdiagramme mit mehrschichtigem Feedback“. In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014.

[Ifl14] Marianus Ifland. *Feedback-Generierung für offene, strukturierte Aufgaben in E-Learning-Systemen*. Dissertation. 2014.

[Kö10] Michael Kölling. „The Greenfoot Programming Environment“. In: *ACM Transactions on Computing Education (TOCE)* 10.4 (2010).

[Mal+10] John Maloney u. a. „The Scratch Programming Language and Environment“. In: *ACM Transactions on Computing Education (TOCE)* 10 (2010), 16:1–16:15. DOI: 10.1145/1868358.1868363.

[Mye90] Brad A. Myers. *Taxonomies of visual programming and program visualization*. 1990. DOI: 10.1016/S1045-926X(05)80036-9.

[NNZ00] Ulrich Nickel, Jörg Niere und Albert Zündorf. „The FUJABA environment“. In: *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium* (2000). DOI: 10.1109/ICSE.2000.870485.

[RJB10] James Rumbaugh, Ivar Jacobson und Grady Booch. *The Unified Modeling Language Reference Manual (Paperback)*. The Addison-Wesley object technology series. ADDISON WESLEY Publishing Company Incorporated, 2010.

[SG11a] Michael Striewe und Michael Goedicke. „Automated checks on UML diagrams“. In: *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education – ITiCSE '11*. New York, USA: ACM Press, 2011, S. 38. DOI: 10.1145/1999747.1999761.

[Sol+10] Josep Soler u. a. „A web-based e-learning tool for UML class diagrams“. In: *IEEE EDUCON 2010 Conference*. Ieee, 2010, S. 973–979. DOI: 10.1109/EDUCON.2010.5492473.

[UN09] Muhammad Usman und Aamer Nadeem. „Automatic Generation of Java Code from UML Diagrams using UJECTOR“. In: *International Journal of Software Engineering and Its Applications* 3.2 (2009), S. 21–38.

[WS11] David Wolber und Fulton Street. „App Inventor and Real-World Motivation“. In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011, S. 601–606. DOI: 10.1145/1953163.1953329.