Ein Format für Bewertungsvorschriften in automatisiert bewertbaren Programmieraufgaben

Robert Garmann[1]

Abstract: Automatisiert bewertbare Programmieraufgaben definieren Tests, die auf Einreichungen angewendet werden. Da Testergebnisse nicht mit Bewertungsergebnissen gleichzusetzen sind, schlagen wir ein Beschreibungsformat vor, das Testergebnisse auf Bewertungsergebnisse abbildet. Lehrkräfte können die Abbildungsvorschrift an ihren Lehrkontext anpassen. Der Vorschlag ist unabhängig von den beteiligten Autobewertern, von den eingesetzten Benutzungsschnittstellen und von der zu lernenden Programmiersprache einsetzbar. Das Format basiert auf verschachtelten Bewertungskategorien, welche um ein Nullifikationen-Konzept ergänzt werden.

Keywords: Automatisierte Bewertung, Programmieraufgaben, Grader, Autobewerter, E-Assessment, Bewertungsschema, ProFormA, Softwaretest

1 Einleitung

Automatisiert bewertbare Programmieraufgaben werden häufig im formativen E-Assessment von Informatik- und verwandten Lehrveranstaltungen mit großen Teilnehmerzahlen eingesetzt, um das regelmäßige Programmieren zu fördern. Abb. 1 gibt ein Beispiel.

Aufgabe: Schreiben Sie eine Java-Klasse `Circle` mit zwei statischen Methoden `calcArea` und `calcRadius`. Die erste Methode soll für einen gegebenen Radius die Kreisfläche berechnen, die zweite Methode soll die Umkehrfunktion berechnen.

Syntax (10%)			✓	10%
Funktion (60%)				18%
	calcArea (×30%)		✓	
	calcRadius (×70%)		✗	
Wartbarkeit (30%)				0%
	Stil	Anweisungen/Zeile (×30%)	(✓)	
		Bezeichner	✗	
	Kommentare (×70%)		✗	
Summe				28%

Abb. 1: Beispielaufgabe (links) und Bewertungsergebnis (rechts)

Um Anforderungen an ein Format für Bewertungsvorschriften zu motivieren, stellen wir in Tab. 1 ein mögliches Bewertungsschema vor, das von menschlichen Bewertern eingesetzt werden könnte. Die in Abb. 1 dargestellte, für eine konkrete Einreichung vorgenommene Bewertung könnte in ähnlicher Weise automatisiert durch einen sog. Autobewerter (Grader) erfolgen. Solche Bewertungsschemata haben, wenn sie vorab veröffentlicht werden, im summativen Assessment das Potenzial, möglichst transparent und möglichst exakt diejenigen Lernanreize zu setzen, die von der Lehrkraft intendiert werden [Bi99]. Zudem gibt es Hinweise darauf, dass transparente Bewertungskriterien auch im formativen Assessment die Lernleistung von Studierenden verbessern können [PJ13].

[1] Hochschule Hannover, Fakultät IV – Wirtschaft und Informatik, Ricklinger Stadtweg 120, 30459 Hannover, robert.garmann@hs-hannover.de

doi:10.18420/delfi2019_73

Aufwändig entwickelte Aufgaben werden möglichst häufig von verschiedenen Lehrkräften wiederverwendet. Das ProFormA-Aufgabenformat [St15] vereinfacht den Austausch zwischen Lehrenden und die Wiederverwendung von Aufgaben über System- und Institutsgrenzen hinweg. Eine automatisiert bewertbare Aufgabe besteht u. a. aus dem Aufgabentext, den sog. *Tests* zur automatischen Prüfung einer Einreichung sowie der *Bewertungsvorschrift* (*grading-hints*), die aus Testergebnissen ein Bewertungsergebnis berechnet. Während Tests in der Regel sehr detaillierte Angaben enthalten, sind Bewertungsvorschriften häufig (und so auch in Version 1.1 des ProFormA-Formats) knappgehalten, wenn sie nicht gar vollständig fehlen. Bei diesem Mangel können Aufgabenautoren und Lehrkräfte keine Angaben in der Aufgabe hinterlegen, wie die einzelnen zu bewertenden Aspekte zu gewichten sind, so dass ein Autobewerter die Testergebnisse eins-zu-eins als Bewertungsergebnisse ausgibt. Dieser Beitrag befasst sich mit der Frage, wie diese Lücke zu schließen ist. Da es viele verschiedene Auffassungen gibt, welcher Aspekt wie zu gewichten ist, soll diesbezüglich eine gewisse Offenheit des Formats bestehen.

Dim.	**1,0**	**0,7**	**0,3**	**0,0**
Syntax, 10%	Fehlerfrei			Fehlerhaft. Oder: Trivialabgabe[a].
Funktion, 60%	Zwei Methoden funktionieren	Nur calcRadius funktioniert.	Nur calcArea funktioniert.	Nichts funktioniert
Wartbarkeit, 30%	Kommentare *vorhanden* und Code ist stil*konform*[b]	Kommentare *vorhanden,* aber Code *verletzt* Stilregeln[b]	Kommentare *fehlen*, aber Code ist stil*konform*[b]	Kommentare *fehlen* und Code *verletzt* Stilregeln[b]. Oder: Trivialabgabe[a].

[a] eine Trivialabgabe (bspw. Klasse ohne Methoden) gilt als Versagen in Syntax und Wartbarkeit (und in Funktion, aber das muss ja nicht extra erwähnt werden).
[b] Stilkonformität: max. eine Anweisung pro Zeile *und* konventionsgemäße Bezeichner

Tab. 1: Bewertungsschema für die Beispielaufgabe

2 Methode

Unser Forschungsansatz ist dem Design-Based Research zuzuordnen. Die auf Vorarbeiten in [Ga15, Ga16] basierenden, zunächst im kleinen Rahmen für ausgesuchte Lehrveranstaltungen entwickelten Werkzeuge und Methoden wurden von mir und weiteren Lehrenden selbst im Lehreinsatz erprobt und anschließend im Rahmen regelmäßiger Interventionen evaluiert und überarbeitet. Neben dem Ziel, praxistaugliche Systeme und Methoden für konkrete Lehre vorgegebener Programmiersprachen zu erreichen, werden Erkenntnisse von größerer Allgemeingültigkeit angestrebt, die sich auf weitere zur Autobewertung eingesetzte Systemlandschaften und Programmiersprachen übertragen lassen. Forschungsziel ist, die in Abschnitt 1 erwähnte Lücke für automatisch bewertete Programmieraufgaben zu schließen. Manuelle Bewertungen oder Bewertungen anderer Aufgabentypen stehen nicht im Fokus.

Um sich dem Ziel der Verallgemeinerbarkeit zu nähern, wurde zum einen die vergleichsweise etablierte Spezifikation IMS QTI zum Austausch von Tests in der aktuellen Version 2.2 konsultiert[2], ob die darin spezifizierten Datenmodelle verwendet werden können. Des Weiteren wurden die Bewertungsmechanismen bestehender Grader und LMS untersucht. Die Auswahl umfasste diejenigen Systeme, die von Lehrenden und Forschenden der ProFormA-Arbeitsgruppe des eCult+-Projekts[3] eingesetzt werden, deren Angehörige allesamt intensiv an der Entwicklung und Nutzung von Systemen und Methoden für die automatisierte Programmbewertung arbeiten. Untersucht wurden die LMS Moodle und LON-CAPA sowie die Grader Praktomat, JACK, Graja, aSQLg, GATE, ASB, PABS und VEA, die allesamt bspw. in [Bo17] beschrieben sind. Die aus den Interventionen entstandenen Weiterentwicklungen wurden kontinuierlich in der Arbeitsgruppe vorgestellt und diskutiert. In den Diskussionen entstanden Anforderungen an und Lösungsvorschläge für ein Format für Bewertungsvorschriften, welches Anspruch auf größere Allgemeingültigkeit erheben kann und Teil des ProFormA-Formats werden kann.

Den vorliegenden Beitrag beginnen wir mit einem eingehenderen Blick auf die Eigenarten von Tests und einer Einordnung des Qualitätsbegriffs von Bewertungsschemata. Abschnitt 4 beschreibt die im o. g. Entwicklungsprozess entstandenen Anforderungen, vor deren Hintergrund wir ab Abschnitt 5 die entstandenen Lösungsvorschläge herleiten und argumentativ bewerten. Das Beispiel aus Abb. 1 und weitere, aus der konkreten Lehrpraxis erwachsende Normalfälle und Sonderfälle dienen dabei zur Illustration und Evaluation. Quantitative Analysen z. B. durch Umfragen unter Studierenden und Lehrenden wurden nicht durchgeführt. Schließlich beschreibt Abschnitt 7 eine weitergehende Evaluation durch eine konkrete Implementierung. Weitere Beispiele und Untersuchungen sind in einem Forschungsbericht dokumentiert [Ga19].

3 Tests und Bewertungen

Typische Tests zur automatischen Prüfung einer Einreichung reichen von der einfachen Compilierung über dynamische, in einem Unittest-Framework verfasste Tests bis hin zu statischen Codeanalysen und Effizienzprüfungen. Weitere Tests sind etwa Plagiat-Checks, aber auch einfache Prüfungen auf Überschreitungen von Abgabeversuchen und -terminen. Es hat sich für die Ableitung von Bewertungsvorschriften als nützlich herausgestellt, zwei Klassen von Tests zu bilden (s. Tab. 2). Unter robuste Tests (R-Tests) fällt die dynamische Funktionsprüfung, die sich bspw. als Unittest oder als Prüfung der Ergebnisse von SQL-Abfragen manifestiert, aber auch die statische Analyse des Quelltextes auf Anwesenheit erwünschter Konstrukte. R-Tests prüfen Eigenschaften, welche nur durch konstruktive Maßnahmen eines Studierenden erreichbar sind, der die zu prüfenden Kompetenzen besitzt[4]. Dagegen können fragile Tests (F-Tests) für sich genommen z. B. durch funktionslose Trivialabgaben bestanden werden, ohne dass der Studie-

[2] http://www.imsglobal.org/question
[3] http://ecult.me
[4] Ob dies der einreichende Studierende ist oder eine Plagiatquelle, ist dabei unerheblich.

rende alle zu prüfenden Kompetenzen besitzt. Zu den F-Tests zählen wir Compilierung, Regelverletzungen prüfende statische Codeanalysen, Plagiatstests, Effizienzprüfer, etc. Alle Tests bewerten eine Einreichung mit einem Ergebnis in [0,1], wobei 1 einen vollständigen Erfolg benennt und 0 einen vollständigen Misserfolg.

(R)obust	Prüft Merkmale, die nur durch Kompetenz erreichbar sind.
(F)ragile	Prüft Merkmale, die isoliert gesehen mit Trivialabgaben erreichbar sind.

Tab. 2: Testtypen

Offenbar ist es sinnvoll, Einreichungen immer mit beiden Testarten zu testen. Ein Autobewerter, der nur auf F-Tests setzt, ist leicht durch eine Trivialabgabe wie einen funktionslosen Programmrahmen auszuhebeln. Häufig reicht es auch nicht, ausschließlich R-Tests einzusetzen, wenn von Studierenden die Einhaltung von Randbedingungen gefordert wird (Effizienz, Wartbarkeit, akademischer Ehrenkodex, etc.), deren Prüfung mit R-Tests schwierig ist. Automatisiert bewertete Aufgaben enthalten daher üblicherweise mehrere Tests, deren Ergebnisse zu Bewertungsergebnissen aggregiert werden.

Häufig nutzen Menschen sog. „rubrics“ (Bewertungsschemata), um den Bewertungsprozess zu strukturieren (vgl. Tab. 1). Diese können – wie im Beispiel dargestellt – aufgabenübergreifende und aufgabenspezifische Kriterien enthalten. Aus den je Zeile ermittelten Einzelbewertungen wird dann mit Hilfe einer sog. *Aggregierungsstrategie* eine Gesamtbewertung errechnet. Häufig lassen sich die Bewertungsdimensionen nicht vollkommen unabhängig voneinander gestalten [Sa89]. Wenn sich Dimensionen überlappen oder Teile von gröberen Dimensionen sind, können Leistungen in der einen Dimension ggf. nur bei gleichzeitigen Leistungen in einer anderen Dimension erbracht werden. Eine Gesamtbewertung einfach als Summe von Teilergebnissen zu berechnen wird der Abhängigkeit der Dimensionen ggf. nicht gerecht.

Die Auswahl der Bewertungsdimensionen, die Gestaltung der Bewertungsstufen sowie die Konstruktion der Gesamtformel haben Auswirkungen auf die Qualität eines Bewertungsschemas, d. h. darauf, dass alle relevanten und keine irrelevanten Bewertungsaspekte Beachtung finden und dass es keine unerwünschten Variationen zwischen Bewertungen eines Bewerters (intra-rater) und zwischen Bewertungen verschiedener Bewerter (inter-rater) gibt [JS07]. Im automatisierten E-Assessment sind intra-rater-Variationen nicht zu erwarten, wenn wir davon ausgehen, dass der Autobewerter ausschließlich deterministische Testmethoden oder erwartungskonforme probabilistische Testmethoden nutzt. Unter inter-rater-Variationen verstehen wir im E-Assessment hauptsächlich die Diskrepanz zwischen dem Ergebnis des Autobewerters und dem gleichzeitig für dieselbe Einreichung erstellten Ergebnis eines menschlichen Bewerters.

In diesem Beitrag schlagen wir weder ein konkretes Bewertungsschema vor noch bewerten wir dessen inter-rater-Qualität. Ziel dieses Beitrags ist auf einer Meta-Ebene ein Format zur Beschreibung von Bewertungsschemata, das durch seine Betonung der Nachvollziehbarkeit gute inter-rater-Qualitäten erwarten lassen kann.

4 Anforderungen

Tab. 3 listet die Anforderungen an ein Format für Bewertungsvorschriften auf, die in dem in Abschnitt 2 beschriebenen Prozess enstanden sind. R1 nennt den Status quo, dass viele LMS zu einer Aufgabe ein metrisch oder ordinal skaliertes Zahlergebnis speichern. Die dazu notwendige Aggregierung mehrerer, dem Intervall [0,1] entstammender Teilergebnisse setzt aus testtheoretischer Sicht viel Wissen über die Natur der Teilergebnisse voraus. Es besteht jedoch explizit keine Anforderung an die Berücksichtigung der Testtheorie. Wenn ein Aufgabenautor eine gewichtete Summe für nicht intervallskalierte Teilergebnisse bilden will, dann soll er das dürfen.

R1	Die Bewertungsvorschrift soll die Errechnung eines skalaren Gesamtergebnisses für eine Aufgabe ermöglichen.
R2	Besonderheiten einzelner Systeme und Programmiersprachen sollen nicht im Format implementiert sein.
R3	Die Bewertungsvorschriften sind für eine Lehrkraft leicht verständlich und an den Lehrkontext anpassbar. Eine Skalierung mit einem Faktor ist möglich.
R4	Die Bewertungsvorschrift muss aus Sicht der Lehrkraft ermöglichen, voneinander unabhängige Bewertungsaspekte getrennt „zu würdigen“.
R5	Häufig wiederkehrende Teilaspekte der Bewertung und deren Verknüpfung sollen durch die Lehrkraft und durch etwaige maschinelle Unterstützung des LMS einfach auf mehrere Aufgaben einer Lehrveranstaltung „ausgerollt“ werden können.
R6	Das maschinell erstellte Bewertungsergebnis soll für Studierende verständlich und nachvollziehbar sein.
R7	Die Bewertungsvorschrift soll Eigenheiten der von Testwerkzeugen generierten Testergebnisse berücksichtigen.

Tab. 3: Anforderungen

Anforderungen R2 und R3 sollen die Wiederverwendung von Aufgaben fördern. Der in der o. g. Arbeitsgruppe häufig von Lehrkräften geäußerte Wunsch, die maximal erreichbare Punktzahl je Aufgabe anpassen zu können, fordert eine einfache Skalierung.

Im formativen E-Assessment von Programmieraufgaben werden häufig beliebig viele Abgabeversuche bis zum Abgabetermin gewährt, die unmittelbar mit automatisiertem Feedback beantwortet werden. Manche Lehrende beobachten das unerwünschte Studierendenverhalten, Abgaben in hoher Frequenz einzureichen und dabei immer nur leicht zu verändern, bis (zufällig?) eine Punktzahl erreicht ist, die einigermaßen zufrieden stellt. Dem entgegenwirkend lassen transparente, jeden einzelnen Aspekt würdigenden Bewertungskriterien (Anforderung R4) erwarten, dass Studierende vor einer Einreichung anhand des Bewertungsschemas prüfen, ob sie alle Bewertungsaspekte berücksichtigt haben [PJ13]. Um den Aufwand bei vielen Aufgaben beherrschbar für die Lehrkraft zu gestalten, wurde R5 aufgenommen. Man soll etwa zentral für alle Aufgaben eines Semesters festlegen können, mit welchen Prozentsätzen Syntax, Funktion und Wartbarkeit in das Gesamtergebnis eingehen. Werden allerdings neben klassischen Programmieraufga-

ben, bei denen Studierende selbst programmieren müssen, auch sog. code execution exercises gestellt, bei denen Studierende die Ausgabe eines gegebenen Programms nennen müssen, kann R5 nur bedingt erfüllt werden, da der letztgenannte Aufgabentyp keine Bewertungsaspekte für Syntax und Wartbarkeit kennt. R5 fordert letztendlich, dass aufgabenübergreifende Bewertungsaspekte für LMS identifizierbar vorliegen müssen.

Im E-Assessment überbringt eine Maschine Feedback. Dabei gewinnt die in [PJ13] genannte, durch nachvollziehbare Bewertungskriterien geförderte Selbstregulation des Lernens durch Angstreduktion an Bedeutung. Anforderung R6 will bei Studierenden kein lernbehinderndes Gefühl aufkommen lassen, Opfer von willkürlicher Bewertung zu sein.

Testwerkzeuge wurden für die Softwarenentwicklung und nicht für die Benotung von Lernleistungen entwickelt (vgl. Abschnitt 3). Anforderung R7 wurzelt in der Erkenntnis, dass Testergebnisse nicht mit Bewertungsergebnissen verwechselt werden dürfen.

5 Aggregierungsstrategien

Es lassen sich bei den in Abschnitt 2 genannten Standards, Gradern und LMS verschiedene Aggregierungsstrategien beobachten (s. Tab. 4). In diesem Abschnitt wollen wir hieraus die den Anforderungen genügenden Strategien auswählen.

Verbreitet werden gewichtete Summen eingesetzt, bspw. bei voneinander weitgehend unabhängigen R-Tests, die je einen Funktionsaspekt prüfen. Die Min.-Strategie kommt z. B. bei zwei Stilregeln (F-Tests) zum Einsatz, wobei je nach dem vom Studenten gewählten Lösungsansatz mal die eine und mal die andere Stilregel verletzt sein kann, aber nicht beide. Eine Aufgabe mit einer Max.-Strategie fordert bspw. entweder den Einsatz der for-Schleife oder der while-Schleife und verknüpft dabei zwei zugehörige R-Tests. Vor dem Hintergrund von R1, R3, R6 und R7 sind alle drei Strategien (Summe, Min., Max.) sinnvoll und notwendig. Zu R3 sei bemerkt, dass Studierende häufig fragen, welchen Einfluss ein bestimmter Bewertungsaspekt (bspw. der Aspekt *calcRadius* in Abb. 1) auf die Gesamtpunktzahl einer Aufgabe hat. Mit den Strategien Summe, Min. und Max. lässt sich die Antwort (hier *Funktion*□*calcRadius*=60%□70%=42%, vgl. Abb. 1) leicht auf eine von der Lehrkraft vorgegebene neue Gesamtpunktzahl skalieren.

Für die Conditional-Aggregation beschreiben wir ein Beispiel, an dem das Ergebnis F eines robusten Funktionstests und die Ergebnisse P_1 und P_2 zweier fragiler Plagiatstests beteiligt sind. Die Plagiatstests liefern unabhängig voneinander je einen Erfolgswert zwischen 0 (sicheres Plagiat) und 1 (sicher kein Plagiat). Wenn mindestens eines der beiden Plagiatstestergebnisse 1 lautet, soll im Zweifel für den Studenten angenommen werden, dass kein Plagiat vorliegt. Das kombinierte Ergebnis ist dann gleich F. Ansonsten soll vollständiger Misserfolg wegen Vorliegen eines Plagiats angenommen werden:

$$\text{Gesamterfolg} = \begin{cases} \text{F, falls } P_1 = 1 \text{ oder } P_2 = 1 \\ 0, \text{ sonst} \end{cases}$$

Unter der Voraussetzung, dass P_1 und P_2 Ergebnisse aus {0,1} liefern, ließe sich der Gesamterfolg auch als eine Kombination von Min. und Max. darstellen [Ga19], vermutlich jedoch zu Lasten der intuitiven Verständlichkeit (R6).

Summe	Je mehr Einzeltests erfolgreich verlaufen, desto höher wird der Gesamterfolg gewertet (Summe der Einzelerfolge)
Min. (worst)	Der Gesamterfolg ist der geringste Erfolg aller Einzeltests
Max. (best)	Der Gesamterfolg ist der größte Erfolg aller Einzeltests
Conditional	Bedingungsgesteuerte Verknüpfung von Einzelerfolgen zu einem Gesamterfolg
Assoziativ-speicher	Mit menschlichen Bewertungen trainierter Assoziativspeicher, dessen aus den Einzelerfolgen berechneter Gesamterfolg möglichst gut mit Ergebnissen menschlicher Bewerter übereinstimmt (inter-rater-Variation).
Spreadsheet	Flexible Kombination von Einzelerfolgen durch eine Tabelle, deren Zellen Funktionen einer Tabellenkalkulation enthalten.
Median	Der Gesamterfolg ist der mittlere Erfolg aller Einzeltests
Vektor	Der Gesamterfolg ist ein Tupel der Erfolge der Einzeltests

Tab. 4: Aggregation von Testergebnissen

Zur Eignung von Assoziativspeichern sei als Gegenbeispiel das Anwendungsgebiet des *Automatic essay scoring (AES)* genannt, das dem automatisierten Bewerten von Programmen, die als frei innerhalb syntaktischer Regeln zu formulierende Aufsätze verstanden werden können, ähnlich ist. AES hat eine lange Tradition in der Definition hunderter Bewertungskriterien, die teilweise in kommerziellen Werkzeugen als Geschäftsgeheimnis implementiert sind. Aus diesen Kriterien wird mit Hilfe von Trainingsdaten und mit statistischen Methoden bzw. mit Methoden der Künstlichen Intelligenz eine Aggregierungsstrategie berechnet. Die daraus errechneten Bewertungsergebnisse entbehren häufig jeder Nachvollziehbarkeit (R6). Dies wird bspw. in [AB06] zum Anlass genommen, die Bewertung im AES-System e-rater auf ein vollständig transparentes Bewertungsschema mit wenigen, nachvollziehbaren Bewertungskriterien umzustellen.

Ein Spreadsheet bietet zwar volle Transparenz über den Bewertungsmechanismus. Allerdings kann ein Spreadsheet sehr komplexe Berechnungen vornehmen. Bspw. könnte ein Testteilergebnis komplexen mathematischen Funktionen unterworfen werden, bevor es mit anderen Testteilergebnissen akkumuliert wird. Es ist dann zumindest mühsam, zu erkennen, welchen Einfluss ein einzelnes Testteilergebnis auf das Gesamtergebnis hat. Die Anforderungen R3 und R6 sind nur bedingt erfüllt.

Der Median ist zwar das testtheoretisch geeignete Mittel, um oridinalskalierte Teilergebnisse zu aggregieren. Vor dem Hintergrund von R6 entscheiden wir uns gegen die Einbindung des Medians in der Annahme, dass dessen Kenntnis unter Studierenden nur gering verbreitet ist.

Schließlich bemerken wir, dass ein Vektor als Gesamtergebnis R1 verletzt. Vorteil einer Lösung, in der der Grader alle Einzelergebnisse an das LMS in Form eines Vektors lie-

fert, wäre, dass das LMS die Teilergebnisse auf sehr flexible und aufgabenübergreifende Weise zu einer Kursnote verknüpfen könnte. Nachteil einer solchen Lösung wäre, dass bei weitem nicht jedes LMS solch weitreichende Bewertungsfunktionen unterstützt (R2).

Zusammenfassend sollen gewichtete Varianten von Summe, Min. und Max. sowie die Conditional-Strategie in den Formatvorschlag aufgenommen werden.

IMS QTI kann prinzipiell durch individuelle, automatisiert bewertbare Programmieraufgaben mittels *custom response processing* erweitert werden. Für solch individuelle Aufgaben sieht QTI jedoch keine standardisierten Bewertungsvorschriften vor. QTI definiert auf der nächsthöheren Ebene in sog. *Assessment*s mit der *ExpressionGroup* ein großes Arsenal von Verrechnungsregeln zur Verknüpfung mehrerer Aufgaben. Hierunter sind auch die im vorstehenden Absatz selektierten Aggregierungsstrategien. Hauptvorteil bei der Verwendung eines Standards oder eines etablierten Datenformats ist, dass es bereits LMS gibt, die das Format unterstützen[5]. Der naheliegende Ansatz, eine Teilmenge des QTI-Datenmodells zu nutzen, um Bewertungsvorschriften innerhalb einer ProFormA-Aufgabe zu definieren, wurde verworfen, weil bestehende LMS das Datenformat eben nicht für eine einzelne Aufgabe unterstützen, sondern für mehrere Aufgaben, die als Teile eines größeren Quizzes oder Assessments verknüpft werden.

6 Formatspezifikation und Illustration am Beispiel

Um Ergebnisse fragiler und robuster Tests separat aggregieren zu können (R7), reicht eine einstufige Hierarchie nicht aus. Mehrstufige Hierarchien ineinander verschachtelter Bewertungskategorien werden in verschiedenen LMS und Gradern bereits genutzt, so dass wir diese Struktur in das Format übernehmen wollen (vgl. Abb. 2 links). Die Hierarchie*node*s erhalten *id*s und Aggregierungs*function*s[6]. Anforderung R5 wird gut erfüllt, wenn Hierarchieknoten auf oberen Stufen *id*s besitzen, die sich alle Aufgaben einer Lehrveranstaltung teilen. Dann kann das LMS die gemeinsamen Teile der Bewertungsvorschriften aller Aufgaben einer Lehrveranstaltung automatisiert harmonisieren.

Eine Baumstruktur hat den Vorteil einer vorgegebenen Lesestruktur, die durch in den Baumknoten enthaltene *title*s inhaltlich strukturiert wird. Studierende können einen als baumartig verschachtelte Tabelle dargestellten Bewertungsbaum top-down lesen und kennen diese Darstellung vermutlich bereits aus anderen Kontexten (R6). Beispielsweise werden Verzeichnisbäume häufig mit tabellarischen Darstellungen verknüpft. Aggregierte Ergebnisse, die zu inneren Knoten gehören, werden gemeinsam mit Einzelergebnissen der Blätter in einer neben dem Baum dargestellten Tabelle präsentiert. Solche Darstellungen können sowohl in webbasierten LMS als auch in Fat Clients durch in GUI-Bibliotheken vielfach verfügbare, sog. TreeTable-Widgets leicht realisiert werden (R2). Die

[5] Uns sind keine Autobewerter für Programmieraufgaben bekannt, die QTI unterstützen.
[6] Kursiv gesetzte Wortteile sind Elemente des Domänenmodells der Bewertungsvorschrift.

beiden *description*s enthalten auf Nutzeranforderung (bspw. als hover popup) anzeigbare Zusatzinformationen für Studierende bzw. Lehrkräfte.

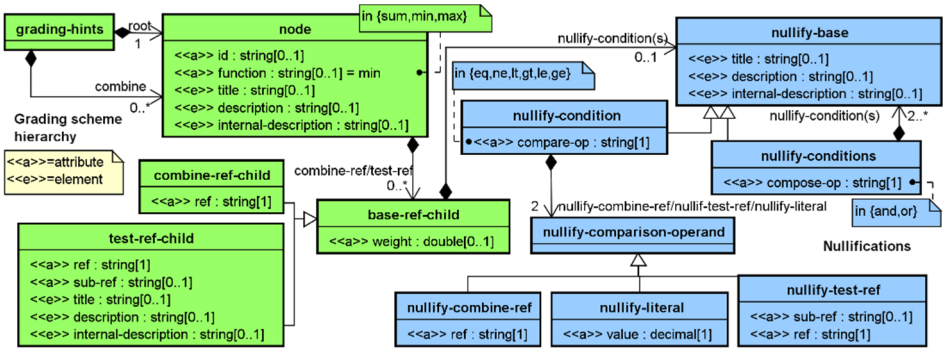

Abb. 2: UML-Domänenmodell des vorgeschlagenen XML-Formats. Links ist die Hierarchie ineinander verschachtelter Bewertungskategorien dargestellt, rechts die Elemente des Nullifikationen-Konzepts. Details des XML-Formats sind im ProFormA-Format-Repository[7] spezifiziert.

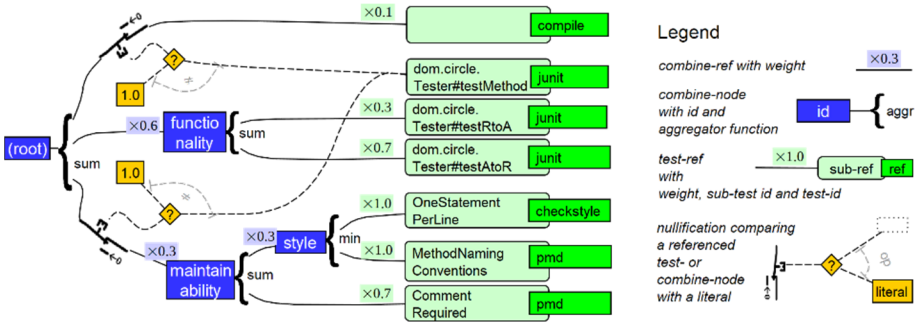

Abb. 3: Graphische Darstellung einer Bewertungsvorschrift im vorgeschlagenen Format. Die combine-ref-Kanten bilden einen Baum. Zusätzliche test-ref- und nullification-Verweise reichern den Baum zu einem Graphen an.

Abb. 3 zeigt das Bewertungsschema aus Tab. 1, wie es im vorgeschlagenen Format repräsentiert[8] wird. Testergebnisse sind rechts jeweils mit einer *ref*erence auf die Id des Tests (bspw. pmd) und der *sub-ref*erence auf die Id des Teiltests (bspw. CommentRequired) repräsentiert. Teiltests sind identifizierbare Teilprozesse eines Testwerkzeug-Durchlaufs. Ein Stilprüfer definiert üblicherweise die Namen einzelner Stilregeln als Teiltest-Ids; ein Unittest wird in einzelne Testfälle unterteilt, die z. B. im Falle von JUnit durch den Namen der Testmethode identifiziert werden. Der Compiler wird in unserem Fall nicht in Teiltests unterteilt. Die Testergebnisse werden an inneren, mit *id*s versehenen *combine*-Knoten über die ge*weight*eten Kanten aggregiert. Blattkanten referenzieren Tests und heißen deshalb *test-ref*s; Kanten zwischen inneren Knoten heißen *combine-*

[7] https://github.com/ProFormA/proformaxml
[8] Die entsprechende XML-Repräsentation findet sich in [Ga19].

*ref*s. Tab. 1 fordert Stilkonformität bzgl. Bezeichnern **und** der Anzahl der Anweisungen pro Zeile. Daher aggregiert der style-Knoten mit der min-Funktion.

Der in Tab. 1 erwähnte Sonderfall einer Trivialabgabe lässt sich durch sog. Nullifikationen abbilden, welche eine Ausprägung der Conditional-Aggregierung sind (vgl. Abschnitt 5). *Nullify-condition(s)* werden als Teil einer *test-ref*- oder *combine-ref*-Kante spezifiziert und bewirken, wenn die formulierte Bedingung erfüllt ist, die Auslöschung des von der Kante referenzierten Teilergebnisses, bevor es aggregiert wird. Ein *nullify-conditions*-Element definiert eine Auslöschungsbedingung, die aus mehreren *and*- oder *or*-verknüpften Teilbedingungen besteht. Einfache Vergleichsbedingungen besitzen im *nullify-conditon*-Element (beachten Sie das fehlende „s“ am Ende) einen *compare-op*erator, der per *nullify-combine-ref* oder *nullify-test-ref* referenzierte Teilergebnisse mit Zahlwerten (*nullify-literal*) vergleicht. Abb. 3 nutzt eine dedizierte JUnit-testMethod, um deren Ergebnis über gestrichelte Linien mit zwei *nullify-condition*s zu vernetzen, die das compile-Testergebnis bzw. das maintainability-Ergebnis dann auslöschen, wenn das Ergebnis von testMethod ≠ 1.0 ist. Die Java-Umsetzung von testMethod meldet einen Testerfolg (1.0), wenn keine Trivialabgabe vorliegt.

7 Evaluation

Zu den meisten der Anforderungen haben wir im obigen Text bereits argumentativ evaluiert, dass diese als erfüllt gelten können. Bzgl. R3 und R6 demonstriert Abb. 4, wie eine für Studierende nachvollziehbare LMS-Darstellung aussehen könnte, die zudem zuvor von einer Lehrkraft auf die Maximalpunktzahl 8 skaliert wurde. Die interaktiv navigierbare Darstellung hat der Student im Syntax- und im Style-Knoten „aufgeklappt“, um eine Erläuterung zur dort jeweils maximal erreichbaren Punktzahl von 0.80 bzw. 0.72 zu erhalten. Die Anordnung der Gewichte in Abb. 3 ist top-down-orientiert und damit günstig, wenn eine die Aufgabe wiederverwendende Lehrkraft leicht abwiechende Gewichtungen für Syntax, Funktion und Wartbarkeit definieren will (R3). Studierendenfreundlich (R6) wird es, wenn man die Gewichte wie in Abb. 4 automatisch zu Darstellungs- und Skalierungszwecken so zu den Blättern hin „propagiert“, dass die studentische Frage nach dem Einfluss der Einzelaspekte auf das Gesamtergebnis direkt ablesbar ist.

Die im vorliegenden Beitrag vorgeschlagene Erweiterung wurde und wird im Studienjahr 2018/19 in zwei konsekutiven Veranstaltungen zur Java-Programmierung mit je ca. 80 Studierenden im Bachelorstudiengang Angewandte Informatik der Hochschule Hannover real eingesetzt. Dabei wurden ca. 50 via Moodle zugängliche Graja[9]-Aufgaben auf das neue Format angepasst. Einige Aufgaben wurden dabei als randomisierbare Aufgaben umgesetzt, so dass hier bereits eine Übertragbarkeit der Ergebnisse auf neue Kontexte gezeigt werden konnte. Die Einsatzeffekte in Bezug auf Lernerfolge wurden nicht systematisch untersucht. Maßgebliches Ziel bei der Umstellung war, die Bewertung des Programmierstils fairer und damit lernförderlicher zu gestalten.

[9] http://graja.hs-hannover.de

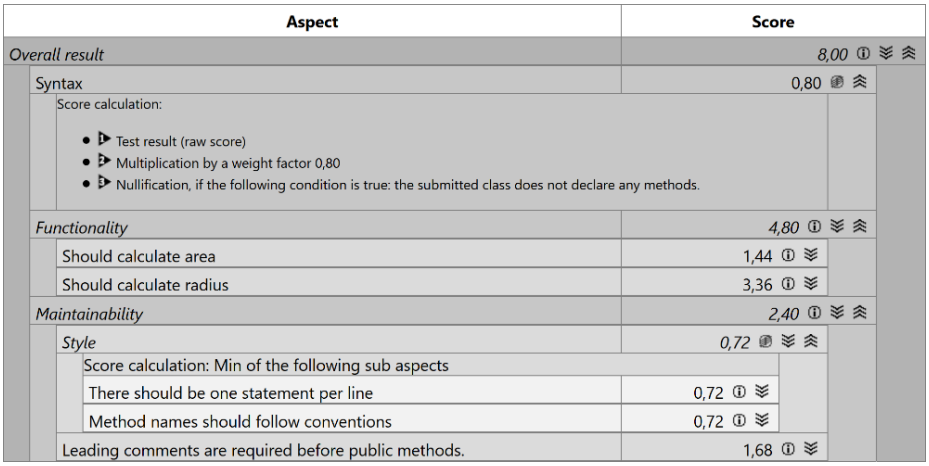

Abb. 4: Evaluation durch Realisierung einer Darstellungsmöglichkeit

Bzgl. des Forschungsziels der Übertragbarkeit des vorgeschlagenen Formats auf andere Bewertungssysteme und Programmiersprachen ist festzustellen, dass die Implementierungsarbeiten für weitere Systeme der o. g. Arbeitsgruppe noch nicht abgeschlossen sind. Die bei Durchstichen gewonnenen Erkenntnisse sind bereits in den Vorschlag eingeflossen, der inzwischen Teil der Version 2.0 des ProFormA-Formats ist und sich somit nahtlos in ein übergreifend nutzbares Aufgabenformat einfügt.

8 Ausblick

Will man auf der Grundlage unserer Ergebnisse nun ein gutes Bewertungsschema entwerfen, so kann man bspw. ein iteratives Forschungsdesign nutzen, wie es etwa in [St16] zur Entwicklung eines Bewertungsschemas für Codequalität beschrieben wird. Um zu einer bestimmten Bewertungsdimension wie der „Syntax“ in Tab. 1 eine umfassende Liste von Deskriptoren zu erhalten, die womöglich mit Exemplaren angereichert sind, kann man dem Beispiel von [JM15] folgen und viele ehemalige Klausuren durchforsten.

Grader und LMS müssen i. d. R. zur Unterstützung des vorgeschlagenen Bewertungsschemaformats angepasst werden. Indem wir die Eigenheiten vieler existierender Systeme berücksichtigt haben, gehen wir von überschaubaren Aufwänden zur Integration des neuen Formats in bestehende Systeme aus, welche idealerweise entlang des von der ProFormA-Arbeitsgruppe spezifizierten Formats für Bewertungsaufträge (submission) und des Antwortformats (response) erfolgt. Beide Formate[7] basieren auf dem hier vorgeschlagenen Bewertungsformat. Das submission-Format unterstützt Anforderung R3 dadurch, dass die in einer Aufgabe (task) festgelegte Bewertungsvorschrift durch eine eigene Vorschrift der Lehrkraft ersetzt werden kann.

Für die wertvollen Diskussions- und Entwicklungsbeiträge der Mitglieder der o. g. Arbeitsgruppe danke ich wie folgt namentlich und herzlich: Karin Borm, Peter Fricke, Elmar Ludwig, Oliver Müller, Uta Priss, Paul Reiser, Oliver Rod.

Literaturverzeichnis

[AB06] Attali, Y.; Burstein, J.: Automated Essay Scoring With e-rater® V.2. Journal of Technology, Learning, and Assessment, 4(3), 2006.

[Bi99] Biggs, J.: Enhancing teaching through constructive alignment. Higher Education, 32(3), 347-364, 1999.

[Bo17] Bott, O. J.; Fricke, P.; Priss, U.; Striewe, M.: Automatisierte Bewertung in der Programmierausbildung. Digitale Medien in der Hochschullehre, ELAN eV, Waxmann, 2017.

[Ga15] Garmann, R.; Heine, F.; Werner, P.: Grappa - die Spinne im Netz der Autobewerter und Lernmanagementsysteme. In: DeLFI 2015. LNI 247, GI, 169-181, 2015.

[Ga16] Garmann, R.; Fricke, P.; Bott, O. J.: Bewertungsaspekte und Tests in Java-Programmieraufgaben für Graja im ProFormA-Aufgabenformat. In: DeLFI 2016. LNI 262, GI, 215-220, 2016.

[Ga19] Garmann, R.: Ein Format für Bewertungsvorschriften in automatisiert bewertbaren Programmieraufgaben., Forschungsbericht, Hochschule Hannover, urn:nbn:de:bsz:960-opus4-13432, 2019.

[JM15] Jara, N.; Madrid, M. M.: Bewertungsschema für eine abgestufte Bewertung von Programmieraufgaben in E-Klausuren. In: DeLFI 2015. LNI 247, GI, 233-239, 2015.

[JS07] Jonsson, A.; Svingby, G.: The use of scoring rubrics: Reliability, validity and educational consequences. Educational research review, 2(2), 130-144, 2007.

[PJ13] Panadero, E.; Jonsson, A.: The use of scoring rubrics for formative assessment purposes revisited: A review. Educational Research Review, 9, 129-144, 2013.

[Sa89] Sadler, D. R.: Formative assessment and the design of instructional systems. Instructional science, 18(2), 119-144, 1989.

[St15] Strickroth, S.; Striewe, M.; Müller, O.; Priss, U.; Becker, S.; Rod, O.; Garmann, R.; Bott, O.; Pinkwart, N.: ProFormA: An XML-based exchange format for programming tasks. eleed e-learning & education, 11(1), 2015.

[St16] Stegeman, M.; Barendsen, E.; Smetsers, S.: Designing a rubric for feedback on code quality in programming courses. In: Koli Calling, ACM, 160-164, 2016.