Repository of Hochschule Hannover

HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS

Ein Format für Bewertungsvorschriften in automatisiert bewertbaren Programmieraufgaben

Robert Garmann

Suggested citation:

Garmann, Robert. 2019. "Ein Format für Bewertungsvorschriften in automatisiert bewertbaren Programmieraufgaben." Hannover: Hochschule Hannover. https://doi.org/10.25968/opus-1343.

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. Letzteres sucht einen Ausgleich im Spannungsfeld zwischen einem für Studierende einfach verständlichen Bewertungsergebnis und den Eigenarten der eigentlich nicht für Bewertungszwecke erfundenen, nichtsdestotrotz regelmäßig und sinnvollerweise für Bewertungszwecke eingesetzten Softwarewerkzeuge.

Terms of use **CC BY-NC-SA 4.0**

This document is made available under these conditions:
Creative Commons - CC BY-NC-SA - Namensnennung - Nicht kommerziell - Weitergabe unter gleichen Bedingungen 4.0 International
For more information see:
https://creativecommons.org/licenses/by-nc-sa/4.0/deed.de

Ein Format für Bewertungsvorschriften in automatisiert bewertbaren Programmieraufgaben

Robert Garmann[1]

HOCHSCHULE HANNOVER
UNIVERSITY OF APPLIED SCIENCES AND ARTS
Fakultät IV Wirtschaft und Informatik

Bericht

31. Mai 2019

Hochschule Hannover
Fakultät IV – Wirtschaft und Informatik
Ricklinger Stadtweg 120
30459 Hannover

[1] E-Mail: robert.garmann@hs-hannover.de

Zusammenfassung

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. Letzteres sucht einen Ausgleich im Spannungsfeld zwischen einem für Studierende einfach verständlichen Bewertungsergebnis und den Eigenarten der eigentlich nicht für Bewertungszwecke erfundenen, nichtsdestotrotz regelmäßig und sinnvollerweise für Bewertungszwecke eingesetzten Softwarewerkzeuge.

Schlagwörter

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

DDC Klassifikation

004 Datenverarbeitung; Informatik

GND-Schlagwörter

Programmierung, E-Learning, Computerunterstütztes Lernen, XML, Übung <Hochschule>, Lernaufgabe

ACM CCS (2012)

• **Social and professional topics~Computer science education** • **Social and professional topics~Student assessment** • *Applied computing~Computer-assisted instruction* • *Applied computing~E-learning*

Inhalt

1 Einleitung ... 5
2 Test vs. Bewertung ... 7
2.1 Tests ... 7
2.2 Testwerkzeuge produzieren Testergebnisse ... 10
2.3 Arten von Tests ... 11
2.4 Bewertungsschemata ... 12
2.5 Identifikation von Teiltestergebnissen ... 13
3 Methode ... 15
4 Anforderungen ... 16
4.1 Unterstützung des Status quo ... 16
4.2 Interoperabilität und Wiederverwendbarkeit ... 16
4.3 Würdigung einzelner Leistungsaspekte ... 16
4.4 Studierendensicht ... 17
4.5 Technische Sicht ... 18
5 Konzept ... 19
5.1 IMS QTI ... 19
5.2 Aggregierungsstrategien ... 19
5.2.1 Vektor ... 20
5.2.2 Summe ... 20
5.2.3 Minimum (worst) ... 20
5.2.4 Maximum (best) ... 22
5.2.5 Conditional ... 22
5.2.6 Assoziativspeicher ... 23
5.2.7 Spreadsheet ... 23
5.2.8 Weitere Strategien ... 23
5.2.9 Fazit ... 24
5.3 Verschachtelte Kategorien ... 24
5.3.1 Motivation ... 24
5.3.2 Prüfung der Anforderung [R7] ... 24
5.3.3 Prüfung der Anforderungen [R1] [R3] [R4] [R6] ... 24
5.3.4 Prüfung der Anforderung [R2] ... 25
5.3.5 Prüfung der Anforderung [R5] ... 25
5.3.6 Testprotokolle ... 25
5.3.7 Beispiel ... 26
5.4 Nullifikationen ... 26
5.5 Validität vs. Score ... 27
6 Ergebnis ... 29
6.1 Beschreibung des XML-Datenformats der verschachtelten Kategorien ... 29

6.2 Ein Beispiel eines nachzubildenden Bewertungsschemas 30
6.3 XML-Fragment zum Beispiel 31
6.4 Darstellung für Studierende 32
6.5 Beschreibung des XML-Datenformats der Nullifikationen 32
6.6 „Rohe“ vs. gewichtete Ergebnisse in Nullifikation-Vergleichsbedingungen – Definition vs. Darstellung 33
6.7 Ein Beispiel eines mit einer einfachen Nullifikation nachzubildenden Bewertungsschemas 34
6.8 XML-Fragment zum Beispiel 34
6.9 Darstellung für Studierende 35
6.10 Ein Beispiel eines mit einer zusammengesetzten Nullifikation nachzubildenden Bewertungsschemas 36
6.11 XML-Fragment zum Beispiel 37
6.12 Vorbereitungen zur Darstellung für Studierende 37
6.13 Bedingungen für eine einfache Darstellbarkeit einer Nullifikations-Vergleichsbedingung 38
6.14 Algorithmus 39
6.15 Darstellung des Beispiels für Studierende 40
6.16 Das Beispiel vom Anfang 40
6.17 XML-Fragment zum Beispiel 41
6.18 Darstellung des Beispiels für Studierende und Lehrkräfte 42
6.19 Darstellung eines Bewertungsergebnisses 43
7 Zusammenfassung und weiterführende Arbeiten 45
7.1 Zusammenfassung 45
7.2 Bewertung der Ergebnisse 45
7.3 Weiterführende Arbeiten 46
8 Quellen 47

1 Einleitung

Automatisiert bewertbare Programmieraufgaben werden häufig im formativen Assessment von Informatik- und verwandten Lehrveranstaltungen mit größeren Teilnehmerzahlen eingesetzt, um trotz beschränkter Betreuungsressourcen das regelmäßige Programmieren zu fördern. Abbildung 1 zeigt eine einfache Beispielaufgabe zusammen mit einer denkbaren studentischen Einreichung. Die in der Einreichung vorgenommenen handschriftlichen Markierungen und die auf der rechten Seite dargestellte Bewertung wurden von einem Menschen erstellt, hätten jedoch in ähnlicher Weise automatisiert erfolgen können. Da die Entwicklung automatisiert bewertbarer Aufgaben aufwändig ist, werden einmalig von einem Aufgabenautor erstellte Aufgaben möglichst häufig und idealerweise auch von verschiedenen Lehrkräften wiederverwendet. Das ProFormA-Aufgabenformat [1] ist ein Austauschformat, das den Austausch zwischen Lehrenden und damit die Wiederverwendung von Aufgaben über System- und Institutsgrenzen hinweg vereinfachen soll.

Aufgabe: *Schreiben Sie eine Java-Klasse* `geom.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.*

Beispieleinreichung:

```
package geom;
public class Circle {
  /** @return πr² */
  public static double calcArea(double r) {
    return Math.PI * Square(r);
  }
  /** @return v² */
  private static double Square(double v) {
    return v * v;
  }
  /* return √a / π */
  public static double calcRadius(double a) {
    return Math.sqrt(a) / Math.PI;
  }
}
```

Wartb. Wartb. Fkt.

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

Abbildung 1: Beispielaufgabe mit Beispieleinreichung und einer von Menschen erstellten Beispielbewertung.

Automatische Werkzeuge zur Bewertung eines zu einer Aufgabe eingereichten studentischen Programms heißen Autobewerter (Grader). Es gibt Grader mit eigenen Funktionen zur Administration einer Lehrveranstaltung. Andere Grader sind reine Dienstprogramme und werden an ein herkömmliches Lernmanagementsystem (LMS) angebunden. In [1] und [2] werden mehrere Grader genannt und teilweise ausführlich beschrieben. Wenn wir im Folgenden von einem Grader sprechen, meinen wir damit den Teil, der sich ausschließlich mit der Prüfung einer Einreichung befasst. Mit dem LMS meinen wir immer den Teil, der die Benutzungsschnittstelle für Studierende und Lehrkräfte anbietet. Manchmal wird die in der Regel asynchrone Kommunikation zwischen LMS und Grader und die eventuell erforderliche Stapelverarbeitung in einer speziellen, zwischen LMS und Grader geschalteten Middleware erledigt [3]. Das ProFormA-Format strebt an, den Aufgabenaustausch zwischen verschiedenen LMS, Middlewares und Gradern möglichst verlustfrei zu ermöglichen.

Eine automatisiert bewertbare Aufgabe besteht neben dem Aufgabentext aus weiteren Komponenten, durch die die automatische Prüfung einer Einreichung erst ermöglicht wird. Am Beispiel des ProFormA-Formats listen wir einige der zu einer Programmieraufgabe gehörenden Artefakte auf:

- Eine *Aufgabenbeschreibung* erläutert die Aufgabe der Studentin oder dem Studenten.

- Sog. *Tests* beschreiben, welchen automatischen Prüfungen eine Einreichung zu unterziehen ist.
- *Testkonfigurationen* beschreiben jeweils die detaillierten Einstellungen bei der Durchführung jedes Tests.
- *Dateien* können als Anlage Teil einer Aufgabe sein. Es gibt Dateien, die für Studierende gedacht sind – bspw. eine Programmschablone, die von Studierenden zu vervollständigen ist. Weitere Dateien können unsichtbar für Studierende sein. Solche Dateien dienen i. d. R. der Parametrierung eines Tests und sind daher Teil der Testkonfiguration. Beispiele sind Unittest-Skripte, Konfigurationsdateien für Stilprüfungswerkzeuge, etc.
- *Bewertungsvorschriften* beschreiben, wie aus den Testergebnissen ein Bewertungsergebnis berechnet wird.

Während die ersten vier Spiegelpunkte in vielen graderspezifischen Aufgabenformaten (und so auch im ProFormA-Aufgabenformat) sehr detaillierte Angaben enthalten, sind Bewertungsvorschriften häufig (und so auch in Version 1.1 des ProFormA-Formats) sehr knapp gehalten, wenn sie nicht gar vollständig fehlen. Bei diesem Mangel kann ein Aufgabenautor keine Angaben in der Aufgabe hinterlegen, wie die einzelnen zu bewertenden Aspekte zu gewichten sind. Manche Grader nehmen grundsätzlich an, dass eine Aufgabe nur dann als vom Studenten erfolgreich bearbeitet angesehen werden sollte, wenn alle Tests erfolgreich waren. Andere melden einen Prozentsatz erfolgreicher Tests als „Bewertung“. In vielen Fällen kann der Aufgabenautor oder die Lehrkraft keinen Einfluss auf den Algorithmus nehmen, nach dem aus Testergebnissen eine Bewertungszahl berechnet wird.

In diesem Beitrag wird ein Vorschlag zur Spezifikation von Bewertungsvorschriften beschrieben. Wir formulieren Anforderungen an einen solchen Vorschlag und untersuchen einige Lösungsmöglichkeiten. Da (wie oben erwähnt) gegenwärtig nur wenige Autobewerter existieren, die detaillierte Bewertungsvorschriften in ihrem jeweiligen Aufgabenformat vorsehen, untersuchen wir zusätzlich Strategien einiger LMS, nach denen diese mehrere Teilleistungen eines Studenten zu einer Gesamtbewertung zusammenführen. Dieses Problem weist nämlich Ähnlichkeiten zu dem Problem auf, aus mehreren Testergebnissen eine Gesamtbewertung zu errechnen. Im Gegensatz zu Autobewertern besitzen LMS häufig sehr ausgereifte Bewertungsmechanismen, in die Lehrende in der Regel nach Belieben steuernd eingreifen können. Wir werden die Entwurfsentscheidungen, die bei der Entwicklung unseres Spezifikationsvorschlags getroffen wurden, begründen. Dieser Vorschlag hat zwischenzeitlich Einzug in Version 2.0 des ProFormA-Formats gehalten[2].

Wir fahren im folgenden Abschnitt 2 zunächst mit einem einführenden Blick auf die Unterschiede zwischen Tests und Bewertungen fort. In Abschnitt 3 erläutern wir kurz unser Forschungsprinzip um dann in Abschnitt 4 die Anforderungen an das zu entwickelnde Datenformat für Bewertungsvorschriften zu formulieren. In Abschnitt 5 untersuchen wir das Problem genauer. Wir schauen uns bei verschiedenen Autobewertern und LMS Lösungsmöglichkeiten ab. Wir diskutieren diese vor dem Hintergrund der Anforderungen. Abschnitt 6 spezifiziert schließlich den versprochenen Vorschlag eines Formats für Bewertungsvorschriften und illustriert diesen an Beispielen.

Die Entwicklung des hier beschriebenen Formats ist das Ergebnis eines langen Diskussions- und Entwicklungsprozesses in der ProFormA-Arbeitsgruppe des eCult+-Projekts[3], in den Sichtweisen und Expertenwissen aus verschiedenen System- und Lehrkontexten eingeflossen sind. Namentlich sei folgenden Personen für ihre wertvollen Beiträge gedankt: Karin Borm, Peter Fricke, Elmar Ludwig, Oliver Müller, Uta Priss, Paul Reiser, Oliver Rod.

[2] https://github.com/ProFormA/proformaxml
[3] http://ecult.me

2 Test vs. Bewertung

2.1 Tests

ProFormA-Aufgaben definieren mindestens einen, meistens jedoch mehrere Tests. Von jedem Test in einer ProFormA-Aufgabe wird erwartet, dass er ein Testergebnis generiert. Die Testergebnisse werden von automatisch ausgeführten Testwerkzeugen erzeugt. Jedes Testergebnis besteht je nach Werkzeug aus einem oder mehreren der folgenden Artefakte

- ein boolesches Ergebnis *bestanden* oder *nicht bestanden,*
- eine Punktzahl,
- ein Testprotokoll.

Eine Einreichung zu unserer Beispielaufgabe (s. Abbildung 1) werde den folgenden Tests unterzogen:

- Compiler: es wird versucht, die Einreichung zu übersetzen.
- JUnit[4]-Test: es werden einige Testfälle dynamisch getestet.
- PMD[5]-Analyse: das Programm wird einigen Prüfregeln in einer statischen Codeanalyse unterzogen.
- Checkstyle[6]-Analyse: wie PMD-Analyse, nur ein weiteres Werkzeug mit anderen Prüfmöglichkeiten.

Im ProFormA-Format sähe die Spezifikation der Tests etwa so aus, wie es in Abbildung 2 dargestellt ist. Wie ersichtlich ist, erhält jeder Test eine id, einen Testtyp und eine i. w. auf Dateien basierende Testkonfiguration. Die Inhalte der Dateien sind in Abbildung 3, Abbildung 4 und Abbildung 5 ersichtlich. Das XML-Fragment aus Abbildung 2 ist Teil einer Datei `task.xml`, die wiederum zusammen mit den referenzierten Dateien in ein Zip-Archiv eingebettet ist.[7]

Abbildung 6 zeigt die zu den einzelnen Tests gehörenden Ausführungskommandos und die produzierten Ausgaben, wenn sie auf die Beispieleinreichung angewendet werden. Der Compiler und Checkstyle haben im Gegensatz zu JUnit und PMD keine Fehler gefunden. Zwischen diesen Ausgaben und der von einem Menschen erstellten Bewertung (vgl. Abbildung 1) besteht eine Diskrepanz, die durch die in diesem Beitrag vorgeschlagene Bewertungsvorschrift überbrückt werden soll.

[4] http://junit.org
[5] https://pmd.github.io
[6] http://checkstyle.sourceforge.net
[7] Alternativ können die referenzierten Dateien auch direkt in die `task.xml` eingebettet werden. In diesem Fall würde das umhüllende Zip-Archiv entfallen.

```
<p:files>
  <p:file id="file1" used-by-grader="true" visible="no">
    <p:attached-bin-file>Test.jar</p:attached-bin-file>
  </p:file>
  <p:file id="file2" used-by-grader="true" visible="no">
    <p:attached-txt-file encoding="UTF-8">rules.xml</p:attached-txt-file>
  </p:file>
  <p:file id="file3" used-by-grader="true" visible="no">
    <p:attached-txt-file encoding="UTF-8">checks.xml</p:attached-txt-file>
  </p:file>
</p:files>

<p:tests>
  <p:test id="compile">
    <p:title>Compilation</p:title>
    <p:test-type>java-compilation</p:test-type>
    <p:test-configuration/>
  </p:test>

  <p:test id="junit">
    <p:title>JUnit dynamic analysis</p:title>
    <p:test-type>unittest</p:test-type>
    <p:test-configuration>
      <p:filerefs>
        <p:fileref refid="file1"/>
      </p:filerefs>
      <u:unittest framework="junit" version="4">
        <u:entry-point>dom.circle.Tester</u:entry-point>
      </u:unittest>
    </p:test-configuration>
  </p:test>

  <p:test id="pmd">
    <p:title>PMD static analysis</p:title>
    <p:test-type>java-pmd</p:test-type>
    <p:test-configuration>
      <p:filerefs>
        <p:fileref refid="file2"/>
      </p:filerefs>
    </p:test-configuration>
  </p:test>

  <p:test id="checkstyle">
    <p:title>Checkstyle static analysis</p:title>
    <p:test-type>java-checkstyle</p:test-type>
    <p:test-configuration>
      <p:filerefs>
        <p:fileref refid="file3"/>
      </p:filerefs>
    </p:test-configuration>
  </p:test>
</p:tests>
```

Abbildung 2: Dateien und Tests zur Beispielaufgabe

```
package dom.circle;
import java.util.List;
import java.util.function.UnaryOperator;
import org.junit.Test;
import org.junit.BeforeClass;
import java.util.stream.Collectors;
import java.util.stream.DoubleStream;
import static org.junit.Assert.assertTrue;

public class Tester {
  private static double callSubmission(String method, double arg) {
    try { return (Double)Class.forName("geom.Circle")
              .getDeclaredMethod(method, double.class).invoke(null, arg); }
    catch (Exception e) { throw new AssertionError(e); }
  }
  private static void impl(String method, UnaryOperator<Double> expected, double ... args) {
    List<String> msgs= DoubleStream.of(args)
      .filter(arg -> Math.abs(expected.apply(arg) - callSubmission(method, arg)) > 1E-5)
      .mapToObj(arg -> String.format("Circle.%s(%.1f) should return %.4f but returned %.4f",
                            method, arg, expected.apply(arg), callSubmission(method, arg)))
      .collect(Collectors.toList());
    double score= 1.0 - (double)msgs.size() / args.length;
    assertTrue(String.format("[test-score=%.2f]%n%s", score, String.join(".\n", msgs)),
               msgs.isEmpty());
  }
  @Test public void testRtoA() {
    impl("calcArea", r->Math.PI*r*r, 0, 1, 3, 10, 30);
  }
  @Test public void testAtoR() {
    impl("calcRadius", a->Math.sqrt(a/Math.PI), 0, 1, 3, 10, 30);
  }
  @org.junit.Test public void testMethod() throws Exception {
    assertTrue("should implement methods",
                Class.forName("geom.Circle").getDeclaredMethods().length > 0);
  }
}
```

Abbildung 3: Der Quellcode der in `Test.jar` enthaltenen Testklasse

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration 1.3//EN"
"http://checkstyle.sourceforge.net/dtds/configuration_1_3.dtd">
<module name="Checker">
  <property name="charset" value="UTF-8"/>
  <module name="TreeWalker">
    <module name="OneStatementPerLine"/>
  </module>
</module>
```

Abbildung 4: Die Datei `checks.xml`

```
<?xml version="1.0"?>
<ruleset xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation
    ="http://pmd.sourceforge.net/ruleset/2.0.0 http://pmd.sourceforge.net/ruleset_2_0_0.xsd">
  <rule ref="rulesets/java/comments.xml/CommentRequired">
    <description>Leading comments are required before public methods.</description>
    <properties>
      <property name="fieldCommentRequirement" value="Ignored"/>
      <property name="protectedMethodCommentRequirement" value="Ignored"/>
      <property name="publicMethodCommentRequirement" value="Required"/>
      <property name="headerCommentRequirement" value="Ignored"/>
    </properties>
  </rule>
  <rule ref="rulesets/java/naming.xml/MethodNamingConventions"/>
</ruleset>
```

Abbildung 5: Die Datei `rules.xml`

Compiler

```
javac geom/Circle.java
```

```
(no output)
```

JUnit

```
java -cp .:Test.jar:junit.jar:hamcrest-core.jar org.junit.runner.JUnitCore
    dom.circle.Tester
```

```
JUnit version 4.11
.E..
Time: 0.069
There was 1 failure:
1) testAtoR(dom.circle.Tester)
java.lang.AssertionError: [test-score=0.20]
Circle.calcRadius(1.0) should return 0.5642 but returned 0.3183.
Circle.calcRadius(3.0) should return 0.9772 but returned 0.5513.
Circle.calcRadius(10.0) should return 1.7841 but returned 1.0066.
Circle.calcRadius(30.0) should return 3.0902 but returned 1.7435
[…]
FAILURES!!!
Tests run: 3,  Failures: 1
```

PMD

```
run.sh pmd -d geom -f text -R rules.xml -version 1.8 -language java -shortnames
```

```
Circle.java:8:  Method names should not start with capital letters
Circle.java:12: publicMethodCommentRequirement Required
```

Checkstyle

```
java -cp checkstyle-all.jar com.puppycrawl.tools.checkstyle.Main -c checks.xml
    geom/Circle.java
```

```
Starting audit...
Audit done.
```

Abbildung 6: Testausführungen für die Beispieleinreichung aus Abbildung 1 mit teilweise gekürzten Ausgaben

2.2 Testwerkzeuge produzieren Testergebnisse

Die hier genannten Testwerkzeuge wurden nicht entworfen, um Punktzahlen zu berechnen. Bspw. wird JUnit in der professionellen Softwareentwicklung genutzt, um automatisiert Hinweise auf Fehler im Programm zu finden, die sich ggf. durch Codeänderungen ergeben haben. In der Regel beobachten Entwicklungsverantwortliche den Prozentsatz erfolgreicher JUnit-Tests, der von einem nächtlichen oder kontinuierlich durchgeführten Testdurchlauf berichtet wird. Der Prozentsatz wird unter der Annahme gebildet, dass jeder Testfall einen gleichen Beitrag zum Gesamterfolg des Tests leistet. Auch Werkzeuge wie PMD und Checkstyle können so gesteuert werden, dass im Anschluss der Prozentsatz der verletzten Prüfregeln als Testergebnis vorliegt. Wir halten fest, dass Testwerkzeuge so gesteuert werden können, dass sie ein Testergebnis im Intervall [0,1] liefern.

Der Compiler gilt in der professionellen Softwareentwicklung nicht in erster Linie als Testwerkzeug. Übersetzbarkeit ist eine Grundvoraussetzung, so dass in der Regel kein detaillierter Prozentsatz berechnet wird, zumal die maximal mögliche Anzahl der Compilerfehler ja auch nicht bekannt ist. Der Compiler meldet einfach „Erfolg“ oder „Misserfolg“, was bei einem Einsatz als Test in einer automatisiert bewerteten Programmieraufgabe als 1 oder 0 codiert wird.

Es wird deutlich, dass Testwerkzeuge keine Bewertungsergebnisse liefern, sondern das, wofür die Werkzeuge gebaut wurden, nämlich Testergebnisse. Bewertungsergebnisse bewerten die Leistung oder die Kompetenz eines Studierenden und sind nicht notwendigerweise proportional zu den von Testwerkzeugen erzeugten Testergebnissen.

2.3 Arten von Tests

Die bisher genannten Testwerkzeuge und die hiervon durchgeführten Tests sind Beispiele. Abhängig von der in der Programmieraufgabe geforderten Programmiersprache mögen einzelne der genannten Tests entfallen und andere hinzukommen. Wir versuchen eine Klassifizierung in robuste und fragile Tests (vgl. Abbildung 7). Manche Testwerkzeuge sind relativ klar der Klasse der R-Tests zuzuordnen, weil sie Eigenschaften der eingereichten Lösung prüfen, die nur konstruktiv durch Programmierung einer Lösung erreicht werden können. Andere Testwerkzeuge wie die o. g. Stilprüfer PMD und Checkstyle sind klare F-Werkzeuge. Und wieder andere Werkzeuge können sowohl zur Definition von R-Tests als auch zur Definition von F-Tests genutzt werden.

Art	**Beschreibung**	**Beispiele**	**Interpretation eines Ergebnisses in [0,1]**
(R)obust	Prüfung der Einhaltung von Pflichten, d. h. der Anwesenheit von erwünschten Positiv-Eigenschaften, welche **nur durch konstruktive Maßnahmen eines Studierenden erreichbar** sind, der die zu prüfenden Kompetenzen besitzt. R-Tests prüfen nicht, wer die Kompetenz besitzt, d. h. Plagiate bestehen einen R-Test, wenn der Originalautor die zu prüfende Kompetenz besitzt.	• Testfallbasierte dynamische Funktionsprüfung, bspw. durch Unittests oder durch Vergleich der Ergebnisse von SQL-Abfragen. • graphbasierte Prüfung des Quelltextes auf Anwesenheit erwünschter Konstrukte [13].	• 0 = vollständiger Misserfolg (keine durch den Test geprüfte Eigenschaft ist anwesend) • 1 = vollständiger Erfolg (alle durch den Test geprüften Eigenschaften sind anwesend)
(F)ragil	Prüfung der Einhaltung von Verboten, d. h. der Abwesenheit von unerwünschten Negativ-Eigenschaften, welche für sich genommen **vermeidbar sind, ohne dass der Studierende die zu prüfenden Kompetenzen besitzt**, z. B. durch Trivialabgaben.	• Compiler • Regelverletzungen testende Stilprüfer • graphbasierte Prüfung des Quelltextes auf Abwesenheit unerwünschter Konstrukte [13] • Plagiatprüfer • Prüfung, dass der späteste Abgabetermin nicht überschritten wurde • Prüfung, dass die maximale Anzahl von Einreichungsversuchen nicht überschritten wurde • Effizienzprüfer, der bspw. die verbrauchten CPU-Zyklen misst oder der eine Kostenprüfung einer SQL-Anweisung vornimmt [14].	• 0 = vollständiger Misserfolg (alle durch den Test geprüft Eigenschaften sind anwesend) • 1 = vollständiger Erfolg (keine der durch den Test geprüften Eigenschaften ist anwesend)

Abbildung 7: Arten von Tests

Offensichtlich ist es sinnvoll, eingereichte Lösungen immer mit beiden Testarten zu testen. Ein Autobewerter, der ausschließlich auf F-Tests setzt, ist leicht durch eine Trivialabgabe wie einen leeren Programmrahmen ohne jede Funktion auszuhebeln. Häufig reicht es auch nicht, ausschließlich R-Tests einzusetzen, wenn von Studierenden die Einhaltung von Rand-

bedingungen gefordert wird (Effizienz, Wartbarkeit, akademischer Ehrenkodex, etc.), deren Prüfung mit R-Tests schwierig ist.

2.4 Bewertungsschemata

Der Prozess der Bewertung einer studentischen Einreichung ist nicht trivial. Menschliche Bewerter tun sich nicht selten schwer, eine Einreichung als Ganze auf einer vorgegebenen Skala zu bewerten. Häufig nutzen Menschen daher sog. „rubrics" (Bewertungsschemata), um den Bewertungsprozess zu strukturieren. Unterschieden werden ganzheitliche Schemata („holistic rubrics"), in denen keine Unterteilung in unterschiedliche Beurteilungsdimensionen erfolgt, und analytische Schemata („analytic rubrics"), die ausgewählte Beurteilungsdimensionen umfassen. Abbildung 1 zeigt eine Beispielbewertung, die auf einem analytischen Schema basiert. Das Schema beschreibt in Form von Deskriptoren für jede Dimension, welche Eigenschaften die Einreichung besitzen muss, um einen bestimmten Bewertungswert zu rechtfertigen. In einem analytischen Schema wird aus den Einzelbewertungen dann mit Hilfe einer sog. *Aggregierungsstrategie* eine Gesamtbewertung errechnet.

Bewertungsschemata werden aufgabenspezifisch oder aufgabenübergreifend erstellt. Bspw. können in einem aufgabenübergreifenden Bewertungsschema für Englischaufsätze die Bewertungsdimensionen Satzstruktur, Zeichensetzung, Wortschatz, Stil und Handschrift vorkommen. In einem aufgabenspezifischen Bewertungsschema können konkretere Dimensionen enthalten sein, die z. B. die inhaltliche Substanz bzgl. des vorgegebenen Aufsatzthemas in den Blick nehmen. Eine Schwierigkeit bei der Erstellung eines Bewertungsschemas ist, dass sich die Bewertungsdimensionen häufig nicht vollkommen unabhängig voneinander gestalten lassen [4]. Manche Dimensionen überlappen sich, manche sind Teile von gröberen Dimensionen. Es kann vorkommen, dass Leistungen in der einen Dimension nur dann erbracht werden können, wenn gleichzeitig Leistungen in einer anderen erbracht werden, etc.

Wenn wir für einen Moment die in einer automatisiert bewertbaren Aufgabe definierten Tests als Bewertungsdimensionen auffassen, können wir Ähnliches beobachten. Eine erfolgreiche Übersetzung etwa ist Voraussetzung, um ein Programm automatisiert auf korrekte Funktionalität zu überprüfen. Diese Abhängigkeit besteht jedoch in dieser Absolutheit nur für maschinelle Autobewerter. Ein menschlicher Bewerter könnte ggf. trotz falscher Syntax dennoch funktionale Aspekte bewerten.

	Bewertung			
	Stufe A	**Stufe B**	**Stufe C**	**Stufe D**
Dimension	**1**	**0,7**	**0,3**	**0**
Syntax (10%)	Fehlerfrei	Bis zu *zwei vernachlässigbare* Flüchtigkeitsfehler (bspw. Semikolon vergessen)	Mindestens *ein gröberer* Fehler, der die Programmstruktur betrifft, bspw. unausgeglichene Klammerung	Mindestens *zwei gröbere* Fehler der Stufe C. Oder: Trivialabgabe*.
Funktionalität (60%)	Beide Methoden funktionieren	Nur die Methode calcRadius funktioniert.	Nur die Methode calcArea funktioniert.	Nichts funktioniert
Wartbarkeit (30%)	Kommentare *vorhanden* und Code ist stil*konform* (Anweisungen pro Zeile, Bezeichner).	Kommentare *vorhanden, aber* Code *verletzt* Stilregeln (Anweisungen pro Zeile, Bezeichner)	Kommentare *fehlen*, aber Code ist stil*konform* (Anweisungen pro Zeile, Bezeichner).	Kommentare *fehlen* und Code *verletzt* Stilregeln (Anweisungen pro Zeile, Bezeichner). Oder: Trivialabgabe*.
* eine Trivialabgabe (bspw. Klasse ohne jegliche Methoden) gilt als Versagen in Syntax und Wartbarkeit (und in Funktionalität, aber das muss ja nicht extra erwähnt werden).				

Abbildung 8: Ein Beispiel eines Bewertungsschemas für menschliche Bewerter.

Abbildung 8 zeigt ein denkbares Bewertungsschema, das sowohl aufgabenübergreifende Elemente als auch aufgabenspezifische Elemente für die Beispielaufgabe der Abbildung 1 enthält.

Bei voneinander abhängigen Dimensionen stellt sich die weitere Schwierigkeit, aus einzelnen Bewertungswerten eine Gesamtbewertung zu berechnen [4]. Eine einfache Summenformel wird der Abhängigkeit der Dimensionen möglicherweise nicht gerecht.

Die Auswahl der Bewertungsdimensionen, die Gestaltung der Bewertungsstufen sowie die Konstruktion der Gesamtformel haben Auswirkungen auf die Qualität eines Bewertungsschemas. Üblicherweise versteht man hierunter die Eigenschaft, 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 [5]. 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 unterbreiten wir keinen konkreten Vorschlag für gute Bewertungsdimensionen. Konsequenterweise treffen wir in diesem Beitrag auch keine Aussagen zur Qualität von Bewertungsschemata. Ziel dieses Beitrags ist, auf einer Meta-Ebene ein Format zu schaffen, das die Beschreibung von Bewertungsschemata ermöglicht, von denen anzunehmen ist, dass sie eine gute Qualität im Sinne des vorstehenden Absatzes besitzen.

Für die Bewertungsstufen schlagen wir vor, diese aus dem Intervall [0,1] zu entnehmen. Für die Formel zur Berechnung einer Gesamtbewertung schlagen wir vor, gewichtete Summen, Maximum- oder Minimumbildung sowie eine spezielle Nullifikationsfunktion einzusetzen. Diese Vorschläge spannen einen Rahmen auf, innerhalb dessen sich unseres Erachtens sowohl gute als auch schlechte Bewertungsschemata entwickeln lassen[8].

Wie erwähnt unterbreiten wir keinen konkreten Vorschlag für gute Bewertungsdimensionen. Will man auf der Grundlage unserer Ergebnisse nun im Anschluss ein gutes aufgabenübergreifendes oder aufgabenspezifisches Bewertungsschema entwerfen, das in den aufgespannten Rahmen passt, so kann man bspw. ein iteratives Forschungsdesign einsetzen, wie es bspw. in [6] zur Entwicklung eines Bewertungsschemas für Codequalität beschrieben wird. Um zu einer bestimmten Bewertungsdimension wie der „Syntax“ in Abbildung 8 eine möglichst ausführliche und umfassende Liste von Deskriptoren zu erhalten, die womöglich mit Exemplaren angereichert sind, kann man dem Beispiel von [7] folgen und viele ehemalige Klausuren durchforsten.

2.5 Identifikation von Teiltestergebnissen

Die oben genannten Testergebnisse (wie wir festgestellt hatten liegen diese alle im Intervall [0,1]) müssen zu einem Bewertungsergebnis verknüpft werden. Eine einfache Möglichkeit wäre, den Durchschnitt der Testergebnisse zu ermitteln und die Testprotokolle (vgl. Abbildung 6) aller durchgeführten Tests zu verketten. Im ProFormA-Format 1.1 war die Definition solcher Bewertungsvorschriften i. w. Sache und Domäne des Autobewerters. Tatsächlich haben viele Autobewerter ihre eigenen, häufig „hart im Code des Autobewerters verdrahteten“ Regeln, nach denen aus Testergebnissen Bewertungsergebnisse werden. Mit dem vorliegenden Beitrag soll stattdessen eine Bewertungsvorschrift als Teil der Programmieraufgabe vom Autor der Programmieraufgabe spezifiziert werden.

Um Bewertungsvorschriften spezifizieren zu können, müssen wir uns die von Tests erzeugten Testergebnisse genauer ansehen. Viele Test-Werkzeuge liefern zwar ein skalares Ergebnis in

[8] Das größte Gefahrenpotential, dass unsere Vorschläge notwendigerweise zu qualitativ schlechten Bewertungsschemata führen, birgt u. E. die Beschränkung der Gesamtformel auf Summe, Minimum und Maximum. Wir treten in diesem Beitrag keinen Beweis an, führen jedoch für verschiedene Normal- und Sonderfälle Argumente dafür an, dass diese Beschränkung in der Praxis des formativen und summativen Assessments von Programmieraufgaben nicht ins Gewicht fällt.

[0,1]. Dieses wird jedoch vom Testwerkzeug meist aus mehreren Einzelergebnissen akkumuliert. Bspw. führt ein Unittest-Werkzeug in der Regel mehrere Testfälle für eine bestimmte Einreichung aus, wobei jeder Testfall ein individuelles Ergebnis meldet. In diesem Beitrag wird ein Datenformat für eine Bewertungsvorschrift vorgeschlagen, die jeden einzelnen Testfall separat bewerten kann. Dazu muss eine Bewertungsvorschrift die sog. *Teiltestergebnisse* individuell referenzieren können.

Im o. g. Beispiel lassen sich Teiltests bspw. wie folgt referenzieren:

- Das JUnit-Werkzeug führt jede Methode der JUnit Testklasse `dom.circle.Tester` einzeln durch. Jedes Einzelergebnis kann individuell anhand des Methodennamens identifiziert werden.
- Die PMD- und Checkstyle-Werkzeuge definieren sog. *rules* und *checks* in Konfigurationsdateien. Jede einzelne Verletzung einer rule bzw. eines checks kann individuell anhand des Namens der rule / des checks identifiziert werden.

3 Methode

Die in diesem Beitrag vorgestellten Ergebnisse basieren auf Vorarbeiten in [3] und [8]. Begleitend zu diesen und weiteren Veröffentlichungen waren IT-technische Systeme und Lernmaterialien (Aufgaben) entstanden, die regelmäßig eingesetzt und sukzessive verbessert wurden. Unser Forschungsansatz lässt sich am ehesten dem Design-Based Research zuordnen [9].

Im vorliegenden Beitrag werden die Ergebnisse der Phase seit Mitte 2017 dargestellt, in der es zunächst um konkrete Verbesserungen der Bewertungsmechanismen real eingesetzter Java-Aufgaben in zwei Programmieren-Grundlagenlehrveranstaltungen des ersten und zweiten Semesters des Bachelor-Studiengangs der Hochschule Hannover ging. Die zunächst nur im kleinen Rahmen entwickelten Werkzeuge und Methoden wurden von mir und weiteren beteiligten Lehrenden sowohl selbst im praktischen Lehreinsatz erprobt als auch anschließend im Rahmen regelmäßiger Interventionen evaluiert und einem Re-Design unterzogen. Neben dem Ziel, praxistaugliche Systeme und Methoden für konkret durchzuführende Lehrveranstaltungen für fest vorgegebene Programmiersprachen zu erreichen, wurde und wird angestrebt, Erkenntnisse von größerer Allgemeingültigkeit zu erlangen. Verallgemeinerungen beziehen sich vor allem auf die Übertragbarkeit der Ergebnisse auf weitere zur Autobewertung eingesetzte Systemlandschaften und auf weitere Programmiersprachen, was sich besonders prominent in der Entwicklung des Grader- und Programmiersprachenunabhängigen ProFormA-Formats manifestiert.

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[9], ob die darin spezifizierten Datenmodelle verwendet werden können. Des Weiteren wurden Bewertungsmechanismen bestehender Grader und LMS untersucht. Die Auswahl der untersuchten Systeme orientierte sich hauptsächlich daran, welche Systeme von Lehrenden und Forschenden der ProFormA-Arbeitsgruppe des eCult+-Projekts[10] 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, VEA. Die aus den Interventionen entstandenen Weiterentwicklungen wurden kontinuierlich in der Arbeitsgruppe vorgestellt und diskutiert. In den Diskussionen entstanden sukzessive Anforderungen an und Lösungsvorschläge für ein Beschreibungsformat für Bewertungsvorschriften, welches einen Anspruch auf größere Allgemeingültigkeit erheben kann und damit Teil des ProFormA-Formats werden kann. Zudem wurden Parallelaktivitäten der Hochschule Hannover, die sich mit der Entwicklung von Methoden für randomisierbare Programmieraufgaben befassten, stets in die Weiterentwicklung des Bewertungsformats einbezogen, so dass eine wechselseitige Kompatibilität der Ergebnisse angestrebt wurde.

Im vorliegenden Beitrag bewerten wir die entstandenen Lösungsvorschläge argumentativ vor dem Hintergrund der entstandenen Anforderungen. Dabei werden aus der konkreten Lehrpraxis erwachsende Normalfälle und Sonderfälle beschrieben und demonstriert, wie diese Fälle durch die vorgeschlagene Lösung abgedeckt werden. Es werden keine quantitativen Analysen z. B. durch Umfragen unter Studierenden und Lehrenden bzgl. der Verständlichkeit der Bewertungsvorschriften durchgeführt.

[9] http://www.imsglobal.org/question
[10] http://ecult.me

4 Anforderungen

Wie lässt sich aus einer Ansammlung von Test- und Teiltestergebnissen ein Bewertungsergebnis berechnen. Und welche Struktur soll ein Bewertungsergebnis überhaupt besitzen? Wir formulieren im Folgenden Anforderungen an ein Grader- und Programmiersprachenunabhängig einsetzbares Format für Bewertungsvorschriften.

4.1 Unterstützung des Status quo

Viele Autobewerter berechnen ein metrisch oder ordinal skaliertes Bewertungsergebnis. Die Verschmelzung verschiedener, voneinander unabhängiger Bewertungsaspekte in einer „Gesamtnote“ kann zwar kritisch gesehen werden, entspricht jedoch nach unserer persönlichen Wahrnehmung dem Wunsch vieler Lernender und Lehrender. Wir formulieren daher die folgende Anforderung:

[R1] Die Bewertungsvorschrift soll die Errechnung eines skalaren Gesamtergebnisses für eine Aufgabe ermöglichen.

Aus testtheoretischer Sicht ist die Berechnung eines Gesamtergebnisses aus einzelnen Teilergebnissen eine Aufgabe, welche viel Wissen über die Natur der Teilergebnisse voraussetzt. Wenn Teilergebnisse aus dem Intervall [0,1] entstammen, darf bspw. nicht automatisch angenommen werden, dass es sich um metrisch skalierte Werte handelt. Aus Sicht der Testtheorie sind abhängig vom Skalenniveau nur ganz bestimmte Aggregierungsstrategien sinnvoll. Wir wollen das Format für Bewertungsvorschriften in dieser Hinsicht jedoch ganz bewusst nicht überfrachten. Es besteht somit ganz explizit keine Anforderung daran, dass die Berechnung des skalaren Gesamtergebnisses korrekt aus Sicht der Testtheorie sein muss. Wir wollen stattdessen ganz bewusst die übliche Praxis unterstützen, dass eine Bewertungsvorschrift auf Geheiß des Aufgabenautors oder Lehrkraft auch dann gewichtete Summen bilden kann, wenn die Einzelergebnisse nicht intervallskaliert sind.

4.2 Interoperabilität und Wiederverwendbarkeit

Da das Datenformat den Anspruch erhebt, von verschiedenen LMS und von verschiedenen Autobewertern für Programmieraufgaben verschiedener Programmiersprachen eingesetzt werden zu können, besteht selbstverständlich die Anforderung:

[R2] Besonderheiten einzelner Systeme und Programmiersprachen sollen nicht im Format implementiert sein.

Um den Austausch und die Wiederverwendung von Programmieraufgaben zu fördern, sollen die Bewertungsvorschriften für eine Lehrkraft, die eine von einem Aufgabenautor erstellte Programmieraufgabe einsetzt, leicht verständlich sein. Zudem soll eine Lehrkraft die Bewertungsvorschriften ohne großen Aufwand auf den jeweiligen Lehrkontext anpassen können.

Eine sehr einfache, aber häufige Anforderung der Lehrkraft ist, die maximal in einer Aufgabe erreichbare Punktzahl zu verändern, um darüber das Gewicht der Aufgabe innerhalb der jeweiligen Lehrveranstaltung zu steuern. An die Bewertungsvorschrift besteht die Anforderung, die Skalierung durch Multiplikation mit einem Faktor so zu ermöglichen, dass die resultierende Bewertungsvorschrift weiterhin intuitiv verständlich ist.

[R3] Die Bewertungsvorschriften sind für eine Lehrkraft leicht verständlich und an den Lehrkontext anpassbar. Eine Skalierung mit einem Faktor ist möglich.

4.3 Würdigung einzelner Leistungsaspekte

[R4] Die Bewertungsvorschrift muss aus Sicht der Lehrkraft ermöglichen, voneinander unabhängige Bewertungsaspekte getrennt „zu würdigen“.

Zum einen ist dieser Aspekt im summativen Assessment wichtig, wenn die Prüfungsanforderungen möglichst transparent und möglichst exakt diejenigen Lernanreize setzen sollen, die von der Lehrkraft intendiert werden [10].

Zudem gibt es deutliche Hinweise darauf, dass transparent bewertete einzelne Bewertungskriterien in Form von Bewertungsschemata auch im formativen Assessment das Potential haben, die Lernleistung von Studierenden zu verbessern [11].

Um einen von Panadero und Jonsson genannten Aspekt herauszugreifen, kann erwartet werden, dass Studierende vor einer Einreichung anhand des Bewertungsschemas prüfen, ob sie alle Bewertungsaspekte berücksichtigt haben. Im formativen Assessment von Programmieraufgaben werden häufig beliebig viele Abgabeversuche bis zum Abgabetermin gewährt, die stets unmittelbar mit automatisiertem Feedback beantwortet werden. Hier kann man durch transparente Bewertungskriterien dem unerwünschten „trial-and-error" Effekt entgegenwirken, dass Studierende Abgaben in hoher Frequenz einreichen und dabei immer nur leicht verändern, bis sie (zufällig?) eine Punktzahl erreichen, die sie einigermaßen zufrieden stellt.

[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.

Diese Anforderung berührt sowohl die Wiederverwendbarkeit als auch die Würdigung einzelner Leistungsaspekte. Es soll beispielsweise möglich sein, für alle Aufgaben einer Lehrveranstaltung leicht und zentral festzulegen, dass Bewertungsaspekte, die die Syntax und die Programmfunktion betreffen, mit 60% in das Gesamtergebnis eingehen sollen, während Programmierstil-Aspekte nur mit 40% eingehen sollen.

Diese Anforderung ist geknüpft an die Voraussetzung, dass alle Aufgaben der Lehrveranstaltung tatsächlich gemeinsame Bewertungsaspekte besitzen. Ein Gegenbeispiel wäre eine Lehrveranstaltung, die sowohl eine Aufgabe wie die oben genannte Beispielaufgabe stellt, als auch eine weitere Aufgabe, die für einen gegebenen Programmtext und eine gegebene Eingabe nach der zu erwartenden Ausgabe fragt. Die letztgenannte Aufgabe kann bzgl. Syntax und Programmierstil nicht sinnvoll bewertet werden, da ja gar nicht programmiert wird. Die Bewertungsstruktur der beiden Aufgaben ist demnach nicht vergleichbar.

Anforderung [R5] fordert nicht mehr und nicht weniger, als dass das gesuchte Datenformat aufgabenspezifische und aufgabenübergreifende Bewertungsdimensionen (vgl. Abschnitt 2.4) identifizierbar beherbergen soll.

4.4 Studierendensicht

[R6] Das maschinell erstellte Bewertungsergebnis soll für Studierende verständlich und nachvollziehbar sein.

Es soll einerseits gewährleistet sein, dass Studierende auf einen Blick etwaige Programmierfehler erkennen können. Andererseits sollen Studierende nachvollziehen können, wie aus Einzelfehlern ein Gesamt-Bewertungsergebnis errechnet wurde.

Die Nachvollziehbarkeit ist wichtig, um kein lernbehinderndes Gefühl aufkommen zu lassen, Opfer einer willkürlichen maschinellen Bewertung zu sein. Erneut führen wir [11] an, wo bspw. Effekte wie eine verbesserte Selbstregulierung des Lernens durch Angstreduktion diskutiert werden. Dieser Aspekt gewinnt bei automatisiert bewerteten Programmieraufgaben, bei denen das Feedback nicht von einem Menschen, sondern von einer Maschine überbracht wird, noch an Bedeutung.

Zur Illustration dieser Anforderung 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. Dies wird bspw. in [12] zum Anlass genommen, die Bewertung im AES-System e-rater auf ein vollständig transparentes Bewertungsschema mit wenigen, nachvollziehbaren Bewertungskriterien umzustellen.

4.5 Technische Sicht

[R7] Die Bewertungsvorschrift soll Eigenheiten der von Testwerkzeugen generierten Testergebnisse berücksichtigen.

Testwerkzeuge sind häufig Werkzeuge der professionellen Softwareentwicklung und nicht für die Benotung von Lernleistungen entwickelt worden. Insofern muss eine Bewertungsvorschrift ein Testergebnis als das behandeln was es ist, nämlich ein Testergebnis und kein Bewertungsergebnis.

Illustrierend nehmen wir an, dass ein von einem Studenten eingereichter, leerer Programmrahmen durch das PMD-Werkzeug analysiert wird. Da keine Stilfehler vorliegen (es wurde ja auch nicht wirklich etwas programmiert), erzeugt das PMD-Werkzeug das Testergebnis 1. Dies ist aber nicht gleichzusetzen mit der Bewertung, der Student zeige einen sehr guten Programmierstil.

5 Konzept

Ausgehend von einigen in LMS und Gradern implementierten Bewertungsmechanismen diskutieren wir in diesem Abschnitt verschiedene Lösungsmöglichkeiten. Wir streben keine Auflistung aller Möglichkeiten an (die wäre ja auch nicht endlich), sondern wir diskutieren naheliegende Varianten vor dem Hintergrund der zuvor formulierten Anforderungen.

5.1 IMS QTI

IMS QTI deckt verschiedene Aufgabentypen ab (Auswahl, Ordnung, Zuordnung, etc.) und 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. *Assessments* mit der *ExpressionGroup* und dem *TestFeedback* ein großes Arsenal von Verrechnungs- und Feedbackregeln zur Verknüpfung mehrerer Aufgaben.

Hauptvorteil bei der Verwendung eines Standards oder eines etablierten Datenformats ist, dass es bereits Systeme gibt, die das Format unterstützen. Als Systeme kommen in unserem Fall LMS, Middleware und Grader in Betracht. Uns sind keine Grader und keine für die Vermittlung von Bewertungsaufträgen für Programmieraufgaben konzipierte Middleware-Systeme bekannt, die QTI unterstützen. Einige existierende LMS unterstützen QTI. 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.

5.2 Aggregierungsstrategien

In automatisiert bewertbaren Programmieraufgaben kommen mehrere Tests bzw. Teiltests zum Einsatz. Die Testergebnisse können grundsätzlich auf unterschiedliche Arten verknüpft werden. Abbildung 9 stellt mehrere grundlegende Verknüpfungsarten dar, die wir in den folgenden Abschnitten genauer beleuchten wollen.

Art	**Beschreibung**
Summe	Je mehr Einzeltests erfolgreich verlaufen, desto höher wird der Gesamterfolg gewertet (Summe der Einzelerfolge)
Minimum (worst)	Als Gesamterfolg wird der geringste Erfolg aller Einzeltests angenommen
Maximum (best)	Als Gesamterfolg wird der größte Erfolg aller Einzeltests angenommen
Conditional	Hier kann man sich weitere Konstrukte vorstellen, die Testerfolge bedingt zu einem Gesamterfolg verknüpfen.
Assoziativspeicher	Mit menschlichen Bewertungen trainierter Assoziativspeicher, dessen als Ergebnis aus den Einzelerfolgen berechneter Gesamterfolg möglichst gut mit Ergebnissen menschlicher Bewerter übereinstimmt (inter-rater-Variation).
Spreadsheet	Flexible Kombination von Teilerfolgs-Ergebnissen durch eine Tabelle, deren Zellen Funktionen einer Tabellenkalkulation enthalten.
Median	Als Gesamterfolg wird der mittlere Erfolg aller Einzeltests angenommen
Vektor	Der Gesamterfolg wird als Tupel der Erfolge der Einzeltests angenommen

Abbildung 9: Arten der Verknüpfung von Tests

5.2.1 Vektor

Diese Verknüpfung ist hauptsächlich sinnvoll, wenn die Verknüpfung der Einzelergebnisse an einen weiteren, häufig von Menschen durchgeführten Prozessschritt delegiert werden soll.

Der Praktomat [15] erzeugt sog. CheckerResults, in denen ein flag das Bestehen des jeweiligen Tests signalisiert. Die Einzelergebnisse werden danach in der Regel durch Tutoren zu einem Gesamtbewertungsergebnis zusammengeführt. Insofern führen wir diesen Grader als ein Beispiel an, bei dem das Bewertungsergebnis ein Vektor von Testergebnissen ist. Die einzelnen Testergebnisse werden automatisiert nicht zu einem skalaren Gesamtergebnis verknüpft. Ganz entsprechend agiert der Grader PABS [16], der Grader ASB [17] und das GATE-System in [18].

Vorteil einer Lösung, in der der Grader alle Einzelergebnisse an das LMS in Form eines Vektors liefert, wäre, dass das LMS die Teilergebnisse automatisiert und 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]. Nach unserer Wahrnehmung gehen viele LMS bei Aufgaben davon aus, dass sie einen skalaren Wert als Ergebnis besitzen, der metrisch oder ordinal skaliert ist. Dass Grader die Aggregierung zu einem Skalar grundsätzlich an das LMS delegieren, wird bereits durch Anforderung [R1] ausgeschlossen.

Die oben genannten Grader, die keine Aggregierung der einzelnen Testergebnisse vornehmen, setzen, falls das LMS oder Menschen keine Weiterverarbeitung der Testergebnisse vornehmen, faktisch die folgende Aggregierungsstrategie ein: Eine Einreichung wird nur dann automatisch als korrekt erkannt, wenn alle Test- und Teiltestergebnisse auf *bestanden* lauten. Eine solche einfache Bewertungsvorschrift widerspricht dem in Anforderung [R4] formulierten Wunsch, einzelne Bewertungsaspekte angemessen würdigen zu können. Dies birgt die Gefahr eines demotivierenden Effekts [R6], denn bereits kleinste Fehler würden hier mit größter Härte (0 Punkte) bestraft.

5.2.2 Summe

Als Beispiel sei JACK [13] angeführt. Hier kann eine Bewertungsvorschrift als einfache Formel angegeben werden, die die Testergebnisse anhand ihrer Id referenziert und gewichtet summiert. Der Grader aSQLg setzt ebenfalls auf eine gewichtete Summe von Teiltestergebnissen [14].

Die Aggregatorfunktion „gewichtete Summe“ wird sinnvollerweise dann eingesetzt, wenn die Tests Aufgabenaspekte bewerten, die mehr oder weniger unabhängig voneinander gelöst werden können bzw. die aus anderen Gründen proportional zum Erfüllungsgrad in das Gesamtergebnis eingehen sollen. Beispielsweise kann es sinnvoll sein, mehrere voneinander weitgehend unabhängige R-Tests, die je einen kleinen Funktionsaspekt prüfen, durch eine Summe zu aggregieren.

Mit hohen Gewichten wird man üblicherweise die schwierigen Teilaspekte gewichten. Wenn die Summe der Gewichte 1 ergibt und wenn alle Teilergebnisse in [0,1] sind, dann ist auch das Bewertungsergebnis, das sich aus der gewichteten Summe ergibt, in [0,1]. Im Bewertungsschema der Abbildung 8 kann die gewichtete Summe genutzt werden, um bspw. die funktionale Korrektheit der beiden im Aufgabentext geforderten Methoden einzeln zu würdigen. Die Umsetzung der Methode `calcArea` wird als weniger wertvoll erachtet (Gewicht 0,3), da hier keine Wurzelberechnung erforderlich ist. Eine korrekt umgesetzte Methode `calcRadius` geht hingegen mit dem höheren Gewicht 0,7 in die Funktionalität ein.

5.2.3 Minimum (worst)

Es stellt sich die Frage, ob neben der gewichteten Summenbildung weitere Aggregierungsstrategien benötigt werden, um die Anforderungen zu erfüllen.

Beispielsweise kann es sinnvoll sein, das Ergebnis eines R-Tests, der eine gewünschte Funktionseigenschaft prüft, nur dann als Erfolg zu werten, wenn das eingereichte Programm gleichzeitig eine Plagiatsprüfung (F-Test) besteht. Ein weiteres Beispiel: Es sollen zwei Stilregeln (F-Test) auf eine Einreichung angewendet werden, wobei je nach dem vom Studenten gewählten Lösungsansatz mal die eine und mal die andere Stilregel verletzt sein kann, aber in der Regel

nicht beide zusammen. Würden die beiden Tests Summen-kombiniert, würde jeder Student ohne eigenes Zutun einen „halben“ Erfolg attestiert bekommen.

Abbildung 1 zeigt, dass in dem dargestellten konkreten Fall die vom Studenten eingereichte Lösung eine zusätzliche Hilfsmethode enthält. Die Benennung der Hilfsmethoden liegt im Ermessen des Studenten und ist in diesem konkreten Fall nicht konventionsgemäß erfolgt. Die vom Testwerkzeug eingesetzte Stilregel, die Punkte für konventionsgemäß gewählte Methodennamen vergibt, würde allerdings nicht nur dann Punkte vergüten, wenn die Hilfsmethode korrekt benannt wäre, sondern auch dann, wenn der Student überhaupt keine Hilfsmethode verwendet hätte. Das Testwerkzeug stellt in diesem Fall ja lediglich fest, dass kein Verstoß gegen die Stilregel vorliegt. Würden die so vergüteten Punkte summativ in ein Gesamtergebnis eingehen, wäre das Ergebnis verfälscht. Die Ursache liegt darin, dass Anforderung [R7] verletzt ist. Bei Verwendung des Summen-Aggregators können wir nicht gut auf die Eigenheit des Stilprüfungswerkzeugs eingehen, dass dieses nur erkennen kann, dass keine Regelverletzung vorliegt, dass dieses aber nicht ohne weitere Hilfe erkennen kann, dass dies nichts mit den Kompetenzen des Studierenden zu tun hat.

Bei dem Aspekt „Anweisungen pro Zeile“ hat unser Student Glück gehabt. Er ist in keiner seiner Methoden überhaupt in Verlegenheit gekommen, eine zweite Anweisung in dieselbe Zeile zu schreiben. Hier würde er also Punkte vergütet bekommen, für eine Leistung, die der Student aufgrund seiner Programmstruktur gar nicht erbringen konnte.

Um solchen ungerechtfertigten Punktevergütungen entgegenzuwirken, wurde im Bewertungsschema in Abbildung 8 die Bewertung der beiden Stilaspekte „Anweisungen/Zeile“ und „Bezeichner“ ganz bewusst als Paket („Code ist stilkonform“) mit 30% der Wartbarkeitspunkte gewichtet. Dahinter steht die Annahme, dass ein Student nur dann Punkte für Stileinhaltung erhalten sollte, wenn er alle Regeln einhält. In echten Aufgaben würden wir daher nicht nur zwei Stilregeln, sondern alle einzuhaltenden Stilregeln (das kann eine zweistellige Anzahl sein) auflisten. Punkte gibt es nur dann, wenn alle Regeln eingehalten wurden. Oder anders formuliert: das aggregierte Bewertungsergebnis aller durch einzelne Stilregeln gelieferten Teiltestergebnisse ist das Minimum dieser Teiltestergebnisse.

Wichtig in dem Zusammenhang ist, dass die Stilregeln u. U. von verschiedenen Testwerkzeugen geprüft werden und dass die Testwerkzeuge außer Stilregeln auch noch weitere Aspekte (Kommentierung, etc.) testen. Aus den Konfigurationsdateien in Abbildung 4 und Abbildung 5 erkennen wir in Kombination mit Abbildung 8, dass wir, wenn wir das Bewertungsschema exakt nachbilden wollen, nicht einfach für „Wartbarkeit“ volle Punkte geben können, wenn sowohl Checkstyle als auch PMD fehlerfrei durchlaufen. Das Werkzeug PMD liefert ein Testergebnis für das Bewertungskriterium „Kommentare“, das mit einem anderen Gewicht als das von PMD gelieferte Testergebnis für das Bewertungskriterium „Bezeichner“ eingeht. Es reicht also nicht, die Aggregation der Form „alle Stilregeln müssen erfüllt sein“ einfach an das Testwerkzeug zu delegieren. Wir müssen uns im Bewertungsschema selbst um eine geeignete Aggregierung kümmern. Für diesen Zweck ist die Aggregierungsstrategie der Minimum-Bildung das geeignete Mittel.

Zusammenfassend: wenn wir Teiltestergebnisse verschiedener Stilprüfungswerkzeuge, welche auf der Erkennung von Regelverletzungen basieren, an verschiedenen Stellen im Bewertungsschema mit verschiedenen Gewichten „einbauen“ wollen (das ist [R4]), dann brauchen wir den Minimum-Aggregator um gleichzeitig Anforderung [R7] erfüllen zu können.

Die Minimum-Aggregierungsstrategie ist generell nützlich in einer Bewertungssituation, in der vom Studierenden gefordert wird, alle Teilaspekte zu lösen. Der am schlechtesten gelöste Teilaspekt bestimmt das berechnete akkumulierte Ergebnis. Gewichte lassen sich auch hier verwenden. Man wird üblicherweise dem Kind, welches den am einfachsten zu lösenden Teilaspekt bezeichnet, das Gewicht 1 zuordnen. Weitere Teilaspekte können Gewichte größer als 1 bekommen. So lange das Minimum der Gewichte 1 ist und wenn alle Kind-Bewertungsergebnisse in [0,1] sind, dann ist auch das Minimum der gewichteten Kindergebnisse in [0,1].

5.2.4 Maximum (best)

Nachdem wir Minimum und Summe als Aggregierungsstrategie als notwendig zur Erfüllung der Anforderungen erkannt haben, liegt es nahe, auch noch die Maximumstrategie zu berücksichtigen. Wir führen einige Beispiele an:

- Eine Aufgabe fordert den Einsatz der for-Schleife (R-Test) oder der while-Schleife (R-Test). Es müssen nicht beide Schleifentypen genutzt werden.
- Eine Aufgabe fordert die Implementierung eines beliebigen effizienten Sortieralgorithmus. Es wird je ein R-Test für jeden erlaubten Algorithmus ausgeführt.
- In der Aufgabe wird freigestellt, den Einrückungsstil 1 oder 2 zu nutzen. Jeder Stil wird separat durch einen F-Test geprüft. Der Gesamterfolg ist dann der bessere der beiden Einzelerfolge.

Eine Maximumstrategie wird in Situationen genutzt, in denen es reicht, wenn Studierende wenigstens einen der Teilaspekte lösen. Dies kann sinnvoll sein, wenn es bspw. für eine Aufgabe mehrere Lösungsstrategien gibt, die jeweils von unterschiedlichen Tests geprüft werden. Eine Einreichung, die wenigstens einen der gleichgewichteten Tests vollständig besteht, erhält bei einer Maximum-Bildung volle Punktzahl. Auch hier ziehen wir daher Anforderung [R4] zur Begründung heran, um die Notwendigkeit oder wenigstens Nützlichkeit der Maximumstrategie festzustellen.

Mit Gewichten kann wieder die Schwierigkeit der Teilaspekte bewertet werden. Wenn eine der Lösungsstrategien besonders schwer ist, dann kann man dies mit dem Gewicht 1 honorieren. Für leichtere Lösungsstrategien erhalten die entsprechenden Teilaspekte entsprechend Gewichte kleiner als 1. So lange das Maximum der Gewichte 1 ist und wenn alle Teil-Bewertungsergebnisse in [0,1] sind, dann ist auch das Maximum der gewichteten Teilergebnisse in [0,1].

5.2.5 Conditional

Das in Abschnitt 5.2.3 genannte Beispiel kann komplexer werden, wenn wir mehrere Plagiatsprüfer einsetzen. Es seien drei Tests beteiligt: ein robuster Funktionstest F und zwei fragile Plagiatstests P_1 und P_2. 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, und das kombinierte Ergebnis ist gleich dem Funktionstestergebnis. Ansonsten soll vollständiger Misserfolg wegen Vorliegen eines Plagiats angenommen werden.

Das Beispiel kann statt durch eine Bedingung auch mit den Minimum und Maximum Aggregatoren geschrieben werden. Die Berechnung

$$\text{Gesamterfolg} = \min(\max(P_1, P_2), F)$$

ist gleichbedeutend mit

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

wenn wir davon ausgehen, dass P_1 und P_2 Ergebnisse aus {0,1} liefern, d. h. keine gebrochenen Werte. Die Formulierung mit der Bedingung halten wir für verständlicher als diejenige unter Einsatz der Minimum- und Maximum-Funktionen. Außerdem kann in der Bedingungsschreibweise problemlos auch der Fall abgedeckt werden, dass die Tests P_1 und P_2 gebrochene Ergebnisse aus [0,1] produzieren.

Die Verknüpfung von Teilerfolgen durch Bedingungen, wobei Teilerfolge sowohl als Teil der Bedingung als auch als Teil eines Ergebnisausdrucks auftauchen können, ist ein mächtiges Werkzeug, kann jedoch, wenn es für sinnvolle Zwecke wie das im vorigen Absatz genannten Beispiel eingesetzt wird, vermutlich leicht durch Studierende verstanden werden (Anforderung [R6]).

5.2.6 Assoziativspeicher

Statt explizit einige Aggregierungsfunktionen wie Summe, Minimum, Maximum argumentativ als notwendig oder sinnvoll herzuleiten, gibt es eine alternative Option. Man könnte die Aggregierungsstrategie als Assoziativspeicher, z. B. als neuronales Netz, mit *n* Eingangsvariablen und einem Ausgangswert modellieren, wobei die *n* Variablen den jeweiligen Teilergebnissen entsprechen. Im Sinne einer guten Qualität des Bewertungsschemas (vgl. Abschnitt 2.4) soll die Ausgabe des Assoziativspeichers möglichst gut mit Ergebnissen menschlicher Bewerter in Übereinstimmung gebracht werden. Dazu muss der Assoziativspeicher geeignet trainiert werden, indem Einreichungen zu neu erstellten Aufgaben zuerst in einem gewissen Umfang von Menschen bewertet werden. Nachteil einer solchen Aggregierungsstrategie ist die mangelnde Transparenz der automatisiert berechneten Bewertungsergebnisse, was Anforderung [R6] verletzt. Zudem wäre bei dieser Lösung das Vorliegen mehrerer durch Menschen erstellter Bewertungsergebnisse Voraussetzung für einen Ersteinsatz einer Aufgabe, während man unter dem Einsatz explizit eingesetzter Aggregierungsstrategien allein durch Argumente und logisches Schlussfolgern zu einer vermutlich brauchbaren Bewertungsvorschrift gelangt.

5.2.7 Spreadsheet

Es gibt LMS, die sehr ausgefeilte Bewertungsmechanismen besitzen. Beispielsweise erlaubt LON-CAPA[11] die Definition von *Spreadsheets*, in denen das Ergebnis einer Aufgabe als Vektor dargestellt wird. Die Elemente des Vektors können herkömmliche Punktzahlen sein. Es können jedoch auch weitere Aspekte einer Aufgabeneinreichung als Ergebnis enthalten sein, wie z. B. die Versuchsanzahl, die Überschreitung des Abgabetermins, etc. Die Elemente des Vektors werden hier aus einzelnen Parametern ähnlich wie in einer Tabellenkalkulation berechnet. In die Berechnung können Parameter der Aufgabe, Einzelheiten eines Testergebnisses, aber auch Parameter des Einreichungs- und Testdurchführungsprozesses eingehen.

Die Bewertungskriterien einer automatisiert bewertbaren Programmieraufgabe sind im Prinzip frei definierbar. Es ist denkbar, dass ein Aufgabenautor Tests in die Aufgabe aufnimmt, welche als Testergebnis die Anzahl verbrauchter Versuche, die Überschreitung des Abgabetermins o. ä. liefern. Im Fokus stehen jedoch Tests, die Eigenschaften der eingereichten Lösung untersuchen.

Ein Spreadsheet könnte als Aggregator genutzt werden, um aus den Teiltestergebnissen auf flexible Weise einen skalaren Gesamtwert zu berechnen. Das komplette Spreadsheet mit all seinen Berechnungen wäre dann die als Teil einer Aufgabe gespeicherte Aggregierungsstrategie. Das daraus berechnet skalare Gesamtergebnis kann dann von einem LMS angezeigt werden [R2].

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 [R6] und [R3] sind nur bedingt gegeben. Die in Anforderung [R3] angesprochene Skalierung um einen Faktor ist ohne eine Beschränkung der im Spreadsheet erlaubten Operationen nur auf die eine Weise möglich, dass das zuvor auf das Intervall [0,1] beschränkte Gesamtergebnis nachträglich um den Faktor skaliert wird. Studierende wollen jedoch z. B. wissen: „Wie viele Punkte gibt es für den Teilaspekt „Kommentare“? Die Antwort ist bei einem komplexen Spreadsheet nicht direkt ablesbar, sondern muss mühsam selbst berechnet werden.

5.2.8 Weitere Strategien

Als eine naheliegende weitere Option kann der Median in Betracht gezogen werden. Dieser ist tatsächlich für ordinal skalierte Werte das testtheoretisch richtige Aggregierungsmittel. Aus Gründen der Anforderung [R6] haben wir uns gegen die Aufnahme des Median in die Bewertungsvorschrift entschieden, da die Kenntnis dieses Maßes nach unserer Wahrnehmung

[11] https://lon-capa.org

unter Studierenden nur gering verbreitet ist. Abschnitt 4.1 schließt zudem explizit die Notwendigkeit testtheoretischer Korrektheit aus.

5.2.9 Fazit

Als sinnvolle Lösungselemente haben sich die gewichtete Summe, sowie das gewichtete Minimum und das gewichtete Maximum herauskristallisiert. Weiterhin ist zur Formulierung komplexerer Abhängigkeiten eines Gesamterfolgs von Teilerfolgen die Formulierung mit Bedingungen (Conditional) die im Sinne der Verständlichkeit für Studierende vielversprechendste Möglichkeit.

5.3 Verschachtelte Kategorien

5.3.1 Motivation

Die Minimum-Bildung und deren Begründung in Abschnitt 5.2.3 ist nur für die Teiltestergebnisse von Belang, die Regelverstöße des Programmierstils betreffen. Da wir in einem Gesamtbewertungsschema aber auch summative Aggregierungen für z. B. Teilfunktionstests vornehmen wollen, bietet sich die hierarchische Strukturierung in mehreren Ebenen an. In einem Stil-Teilbaum wird minimiert, in einem anderen Funktions-Teilbaum wird summiert. Und an der Wurzel wird dann vielleicht ebenfalls summiert. Mit einer solchen hierarchischen Strukturierung wollen wir uns nun befassen.

Das LMS Moodle[12] verwendet baumartig verschachtelte *Bewertungskategorien*. Jedes Einzelergebnis wird einer Elternkategorie zugeordnet, die wiederum rekursiv weiteren Elternkategorien zugeordnet wird. Auf diese Weise entsteht ein Baum mit Kategorien an inneren Knoten und Einzelergebnissen an den Blättern. An inneren Knoten sind in Moodle verschiedene *Aggregierungsstrategien* einsetzbar (gewichteter Mittelwert, Median, Minimum, Maximum, etc.).

Einen ähnlichen Bewertungsansatz nutzt das als Middleware zwischen LMS und Grader agierende Grappa [3], wobei dort derzeit lediglich Punktsummen unterstützt werden. Ganz ähnlich werden in Graja [8] Bewertungsvorschriften als Hierarchie von Teilaspekten aufgebaut.

Während Moodle einen solchen Bewertungsbaum auf Kursebene zur Aggregation von Ergebnissen mehrerer Einzelleistungen nutzt, kann auch hier dasselbe Prinzip „eine Ebene tiefer“ zur Bewertung von Einzelaspekten einer einzelnen Aufgabe dienen. Blätter des Baumes sind Test- oder Teiltestergebnisse wie etwa das Ergebnis einer JUnit-Testmethode oder einer PMD rule. Jeder innere Knoten repräsentiert einen Skalar als Teil des Bewertungsergebnisses, der aus mehreren Kind-Ergebnissen mit den üblichen Aggregierungsstrategien berechnet wird. Die Referenzen auf Kindknoten erhalten dazu je ein skalares Gewicht. Die Wurzel repräsentiert das Gesamtbewertungsergebnis der Aufgabe.

5.3.2 Prüfung der Anforderung [R7]

Der Einsatz der Minimum-Aggregierungsstrategie wurde bereits in Abschnitt 5.2.3 als Argument für die Erfüllung der Anforderung [R7] ins Feld geführt. In einem Autobewerter haben wir es mit verschiedenartigen Testwerkzeugen zu tun. Ein Bewertungsbaum erlaubt, in verschiedenen Teilbäumen unterschiedliche, an das jeweilige Testwerkzeug angepasste Aggregierungsstrategien einzusetzen

5.3.3 Prüfung der Anforderungen [R1] [R3] [R4] [R6]

Eine Baumstruktur hat den Vorteil einer vorgegebenen Lesestruktur. Studierende können einen als baumartig verschachtelte Tabelle dargestellten Bewertungsbaum top-down lesen. Viele Studierende kennen diese Darstellung vermutlich bereits aus anderen Kontexten. Beispielsweise werden Verzeichnisbäume häufig mit tabellarischen Darstellungen verknüpft. Daten, die zu inneren Knoten gehören, werden gemeinsam mit Blattdaten in einer neben dem Baum dargestellten Tabelle präsentiert (vgl. Abbildung 10). Die Anforderungen [R1] [R4] [R6] und [R3] sind erfüllt. Da die an inneren Knoten einsetzbaren Aggregierungsstrategien beschränkt sind,

[12] https://moodle.org

kann die mit [R3] geforderte Skalierung einer Bewertungsvorschrift mit einem Faktor leicht dadurch erreicht werden, dass die Gewichte sämtlicher Blattknoten um diesen Faktor skaliert werden. Die Antwort auf die oben bereits erwähnte studentische Frage nach der maximal erreichbaren Punktzahl im Aspekt „Kommentare“ lässt sich somit direkt aus der skalierten Bewertungsvorschrift ablesen.

	Größe	Belegt	Dateien	Verzeic...	Prozent (Bele...	Letzte Änderung
0 Bytes Pending Pings	0 Bytes	0 Bytes	0	0	0,0 %	18.08.2018
393,6 MB Profiles	390,9 MB	393,6 MB	2.116	1.872	99,7 %	22.01.2019
393,6 MB ypc5j5sg.default	390,9 MB	393,6 MB	2.116	1.871	100,0 %	22.01.2019
1,0 MB bookmarkbackups	1,0 MB	1,0 MB	15	0	0,3 %	22.01.2019
1,4 MB browser-extension-data	1,4 MB	1,4 MB	7	9	0,4 %	22.01.2019
8,0 KB chrome	1,7 KB	8,0 KB	4	0	0,0 %	14.07.2015
20,0 KB crashes	16,2 KB	20,0 KB	1	1	0,0 %	22.01.2019
1,7 MB datareporting	1,5 MB	1,7 MB	130	3	0,4 %	22.01.2019
1,6 MB archived	1,4 MB	1,6 MB	127	2	96,8 %	22.01.2019
928,0 KB 2018-12	819,3 KB	928,0 KB	75	0	55,2 %	31.12.2018
752,0 KB 2019-01	649,7 KB	752,0 KB	52	0	44,8 %	22.01.2019
56,0 KB [3 Dateien]	55,9 KB	56,0 KB	3	0	3,2 %	22.01.2019
56,0 KB aborted-session-ping	55,7 KB	56,0 KB	1	0	100,0 %	22.01.2019
0 Bytes session-state.json	164 Bytes	0 Bytes	1	0	0,0 %	22.01.2019
0 Bytes state.json	51 Bytes	0 Bytes	1	0	0,0 %	13.12.2014
6,9 MB extensions	6,9 MB	6,9 MB	19	0	1,8 %	08.10.2018
52,0 KB features	48,7 KB	52,0 KB	1	1	0,0 %	13.01.2019
48,0 KB firebug	44,8 KB	48,0 KB	2	0	0,0 %	31.10.2017
44,0 KB annotations.json	43,3 KB	44,0 KB	1	0	91,7 %	31.10.2017
4,0 KB breakpoints.json	1,5 KB	4,0 KB	1	0	8,3 %	05.02.2016
0 Bytes gmp	86 Bytes	0 Bytes	3	5	0,0 %	25.12.2018

Abbildung 10: Ausschnitt einer baumartigen Darstellung eines Verzeichnisbaums mit aggregierten Daten an inneren Knoten und Einzeldaten an Blättern (erstellt mit TreeSize Free)

5.3.4 Prüfung der Anforderung [R2]

Eine Festlegung auf Besonderheiten einer bestimmten, in der Programmieraufgabe geforderten Programmiersprache sind bei Verwendung verschachtelter Kategorien nicht zu befürchten. Anforderung [R2] kann für LMS erfüllt werden, allerdings vermutlich nicht ohne Entwicklungsaufwand im LMS. Gleiches gilt für Autobewerter, die einen Bewertungsbaum als Ergebnis produzieren wollen. Im Gegensatz zu mächtigeren Ansätzen wie dem Spreadsheets-Ansatz gehen wir hier jedoch von geringeren Entwicklungsaufwänden aus. Die Darstellung von sog. *TreeTables* oder *TreeGrids* wird im Web und in Entwicklungsplattformen für Fat Clients mit grafischer Benutzungsoberfläche durch verschiedentliche Bibliotheken erleichtert.

Die Tatsache, dass ein Bewertungsbaum stets eine Wurzel hat und dass an dieser Wurzel das Ergebnis der Gesamtbewertung steht, kann von LMS genutzt werden, die keinen Baum darstellen können, sondern lediglich das quantitative Gesamtergebnis. So ist wenigstens eine rudimentäre Unterstützung der Bewertungsvorschrift möglich.

5.3.5 Prüfung der Anforderung [R5]

Eine baumartige Bewertungsstruktur ist ideal für die Erfüllung von [R5]. Die unmittelbaren Kindknoten unterhalb der Wurzel repräsentieren die groben Kategorien, die in jeder Aufgabe einer Lehrveranstaltung vorkommen (bspw. „Syntax“, „Funktion“, „Stil“). Wenn diese Knoten mit Identifizierern versehen werden, die allen Aufgaben der Lehrveranstaltung gemein sind, kann das LMS die an diesen Knoten platzierten Gewichte automatisch mit von der Lehrkraft einmalig vorzugebenen Werten überschreiben. In unserer konkreten Beispielaufgabe würden wir alle Teiltestergebnisse der Werkzeuge PMD und Checkstyle unter „Stil“ einhängen, alle aus JUnit-Testmethoden generierten Teiltestergebnisse unter „Funktion“ und das Compilierungsergebnis unter „Syntax“[13].

5.3.6 Testprotokolle

Das von Testwerkzeugen erzeugte Testprotokoll (vgl. Abschnitt 2) kann häufig durch geeignete Adapter in Teilprotokolle je Teiltest separiert werden. Dann ist es sinnvoll, dass das LMS die

[13] Wohl wissend, dass ein Compiler nicht nur die Syntax, sondern in Teilen bereits die Semantik eines Programms prüft.

Teilprotokolle jeweils im Kontext eines Blatts des Bewertungsbaumes anzeigt (ggf. ein-/ausblendbar). Die in Form verschachtelter Kategorien vorliegende, baumartige Bewertungsvorschrift kann vom LMS also genutzt werden, um das Feedback in genau dieser Baumstruktur zu präsentieren.

5.3.7 Beispiel

Wir zeigen in Abbildung 11, wie das in Abbildung 8 dargestellte Bewertungsschema durch hierarchisch verschachtelte Kategorien dargestellt werden kann. Die Blätter dieses Baums verweisen auf Test- oder Teiltestergebnisse. Eines der Testergebnisse wird zwei Mal referenziert (`testMethod`), so dass es sich eigentlich um keinen Baum, sondern um einen gerichteten azyklischen Graphen handelt. Durch die Verwendung der Minimum-Aggregierung wird an zwei Stellen erreicht, dass eine Trivialabgabe keine Syntaxpunkte und keine Wartbarkeitspunkte erhält. Der Minimum-Aggregator ist gewissermaßen ein Workaround dafür, dass die Testwerkzeuge Compiler und PMD/Checkstyle nicht in der Lage sind, den Fall einer Trivialabgabe als Testfehlschlag zu melden (Anforderung [R7]). Das letzte, am tiefsten verschachtelte Minimum über die beiden Regeln OneStatementPerLine und MethodNamingConventions ist eine Illustration der Ausführungen des Abschnitts 5.2.3.

```
Summe über {
  10% x Syntax, wobei Syntax = Minimum über {
    Teiltestergebnis der JUnit-Testmethode testMethod (Trivialabgabe?)
    Testergebnis des Compilers
  },
  60% x Funktionalität, wobei Funktionalität = Summe über {
    70% x Teiltestergebnis der JUnit-Testmethode testRtoA,
    30% x Teiltestergebnis der JUnit-Testmethode testAtoR
  },
  30% x Wartbarkeit, wobei Wartbarkeit = Minimum über {
    Teiltestergebnis der JUnit-Testmethode testMethod (Trivialabgabe?),
    Summe über {
      30% x Minimum über {
        Teiltestergebnis der Checkstyle-Regel OneStatementPerLine,
        Teiltestergebnis der PMD-Regel MethodNamingConventions
      },
      70% x Teiltestergebnis der PMD-Regel CommentRequired
    }
  }
}
```

Abbildung 11: Überführung des Bewertungsschemas aus Abbildung 8 in hierarchisch verschachtelte Kategorien

Der Bewertungsbaum in Abbildung 11 illustriert außerdem die in Abschnitt 4.5 erwähnte Entkopplung des Bewertungsergebnisses von den Ergebnissen der Testwerkzeuge. Die Zuordnung des Testergebnisses der JUnit-Testmethode `testMethod` erfolgt flexibel sowohl zur Syntax als auch zur Wartbarkeit. Die inneren Knoten der Bewertungshierarchie gruppieren Teiltestergebnisse aus didaktischer Sicht zu sinnvollen Bewertungskategorien, während die Tests des ProFormA-Aufgabenformats dazu dienen, Teiltests aus pragmatischer Sicht zu sinnvollen Testausführungen zu gruppieren.

5.4 Nullifikationen

Das Beispiel in Abschnitt 5.3.7 demonstriert, dass die Erfüllung der Anforderung [R7] zu mehreren, etwas künstlich wirkenden Zwischenstufen in der Bewertung führt. Wir wollen statt

```
Summe über {
  10% x Syntax, wobei Syntax = Minimum über {
    Teiltestergebnis der JUnit-Testmethode testMethod (Trivialabgabe?)
    Testergebnis des Compilers
  },
```

```
  60% x Funktionalität, …
  …
}
```

lieber formulieren:

```
Summe über {
  falls testMethod fehlschlägt, dann 0, sonst 10% x Syntax, wobei Syntax = Testergebnis des Compilers,
  60% x Funktionalität, …
  …
}
```

Die bedingte Auswahl des festen Ergebnisses 0 unter einer bestimmten Bedingung nennen wir *Nullifikation*. Eine Nullifikation nimmt Bezug auf ein Test- oder Teiltestergebnis und vergleicht dieses mit einem bestimmten Ergebniswert. Wenn die Bedingung erfüllt ist, geht der entsprechende Operand (hier der erste Operand der am Wurzelknoten definierten Summe) mit dem Wert 0 in die übergeordnete Aggregation ein. Wir formulieren hier also eine Variante der Conditional-Verknüpfung von Teilerfolgen (vgl. Abschnitt 5.2.5).

Mit Nullifikationen sind auch Verweise auf innere Knoten möglich. Denkbar wäre, dass wir für die Wartbarkeit (die „Kür“) nur dann Punkte vergeben wollen, wenn die erreichte Funktionalität (die „Pflicht“) mindestens p% beträgt. Hierzu würden wir

```
Summe über {
  …,
  30% x Wartbarkeit, wobei Wartbarkeit = Minimum über {
    Teiltestergebnis der JUnit-Testmethode testMethod (Trivialabgabe?),
    Summe über {
      30% x Minimum über {
        Teiltestergebnis der Checkstyle-Regel OneStatementPerLine,
        Teiltestergebnis der PMD-Regel MethodNamingConventions
      },
      70% x Teiltestergebnis der PMD-Regel CommentRequired
    }
  }
}
```

durch das hier ersetzen:

```
Summe über {
  …,
  falls Funktionalität < p%, dann 0, sonst 30% x Wartbarkeit, wobei Wartbarkeit = Summe über {
    30% x Minimum über {
      Teiltestergebnis der Checkstyle-Regel OneStatementPerLine,
      Teiltestergebnis der PMD-Regel MethodNamingConventions
    },
    70% x Teiltestergebnis der PMD-Regel CommentRequired
  }
}
```

So wäre sichergestellt, dass die von PMD und Checkstyle generierten Testergebnisse nur dann in das Bewertungsergebnis eingehen, wenn die Einreichung ein Mindestmaß an Funktion aufweist. Ursprüngliche Ideen, die Nullifikation weicher durch Multiplikation mit einer stetigen Sigmoidfunktionen zu gestalten, haben wir verworfen, weil diese zu schwerer durchschaubaren Zahlergebnissen führen und demnach der Anforderung [R6] zuwider laufen.

Durch Nullifikation sparen wir eine Hierarchiestufe ein und können dennoch die Anforderung [R7] erfüllen. Als Vorteil der Nullifikationen im Gegensatz zu der Variante mit den eingeschobenen Minimum-Zwischenknoten sehen wir die größere Nähe der Bewertungsvorschrift zu dem originalen Bewertungsschema (Abbildung 8). Wir versprechen uns davon u. a. eine bessere Verständlichkeit für Studierende [R6].

5.5 Validität vs. Score

Der Grader VEA [19] setzt einen heuristischen Algorithmus ein, um neben der Richtigkeit der Einreichung (Score) auch noch die Zuversicht des Autobewerters zu berechnen, dass er mit seiner Einschätzung richtig liegt (Validität). Die Algorithmen beruhen teilweise auf

Textvergleichen von Programmtexten und auf der Fähigkeit, mehrere Musterlösungen mit unterschiedlicher Güte angeben zu können. Diese Funktionen sind sicher nicht in vielen Gradern anzutreffen [R2]. Zudem werden gute Erfahrungen insbesondere mit Prolog berichtet – einer Programmiersprache die zumindest derzeit nicht im Fokus von Massenveranstaltungen der Informatik steht.

Die im Algorithmus verwendeten Formeln sind durchaus komplex. Die für eine Einreichung berechneten Score- und Validität-Werte können vermutlich durch Studierende nicht einfach nachvollzogen werden [R6].

Da uns außer VEA keine Ansätze bekannt sind, die Validität von Bewertungsergebnissen zu berechnen, und da Validität auch in LMS bestenfalls eine Randerscheinung ist, werden wir Validität nicht in den Vorschlag aufnehmen.

6 Ergebnis

In diesem Abschnitt beschreiben wir unseren Vorschlag eines Datenformats für Bewertungsvorschriften, welcher auf verschachtelten Kategorien basiert. Das vorgeschlagene Format ist Teil der ProFormA-Formatversion 2.0. Eine grafische UML-Repräsentation, die dem XML-Format nahe kommt, ist in Abbildung 12 dargestellt.

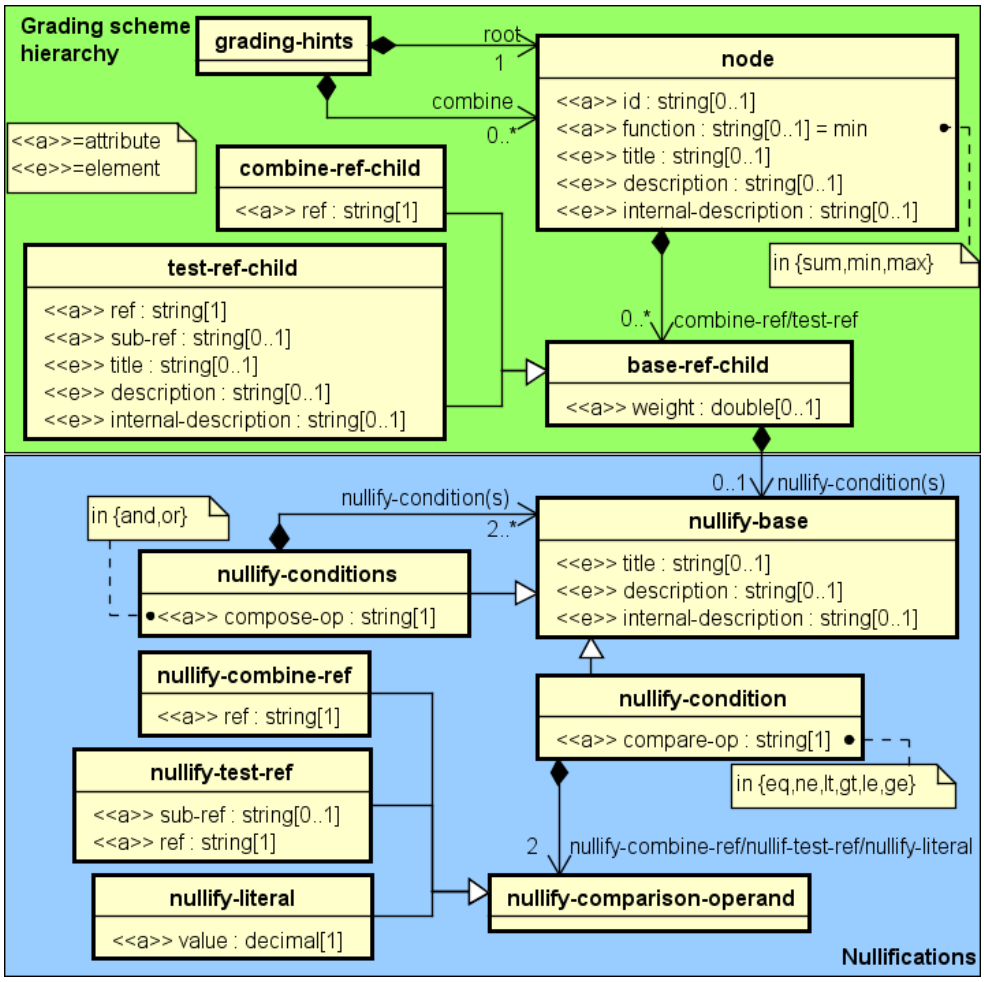

Abbildung 12: UML-Repräsentation

6.1 Beschreibung des XML-Datenformats der verschachtelten Kategorien

Jeder innere Knoten der Hierarchie wird als separates XML-Element realisiert. Der Wurzelknoten erhält den besonderen Elementnamen `root`, die weiteren inneren Knoten heißen `combine`. Beide Elemente sind jedoch vom selben XML-Schema-Datentyp und besitzen folgende Daten:

- Ein `id`-Attribut, das nur für die Wurzel optional ist.
- Ein `function`-Attribut, das die Aggregierungsstrategie des Knotens festlegt. Erlaubte Werte sind `sum`, `min` und `max`. Der Standardwert (bei Fehlen) ist `min`.
- Kindelemente `title`, `description` und `internal-description`, die den Knoten mit einer Überschrift und Beschreibungen für Studierende und Lehrkraft ausstatten.
- Und schließlich eine Sequenz von Referenzen auf Kindknoten.

Es gibt zwei Typen von Kind-Referenzen: Referenzen auf andere `combine`-Knoten und Referenzen auf Testergebnisse. Eine Referenz besitzt folgende Daten:

- beiden Typen von Kind-Referenzen gemein ist ein optionales `weight`-Attribut. Fehlt dieses, wird als Gewicht 1 angenommen.
- Referenzen auf `combine`-Knoten werden durch den Elementnamen `combine-ref` spezifiziert und besitzen ein Attribut `ref`, dessen Wert mit der `id` des referenzierten `combine`-Knotens übereinstimmen muss.
- Referenzen auf Testergebnisse werden durch den Elementnamen `test-ref` spezifiziert und besitzen ebenfalls ein Attribut `ref`, dessen Wert mit der `id` des referenzierten Tests übereinstimmen muss.
- Zusätzlich können `test-ref`-Elemente weitere Daten definieren:
    - Eine optionale Identifizierung des referenzierten Teiltestergebnisses durch ein Attribut namens `sub-ref`. Fehlt dieses, bedeutet die Testreferenz, dass das gesamte Testergebnis referenziert wird. Wie ein Gesamttestergebnis aus Teiltestergebnissen berechnet wird, liegt dann in der Verantwortung des Testwerkzeugs.
    - Optionale Kindelemente `title`, `description` und `internal-description`, die die Referenz mit einer Überschrift und Beschreibungen für Studierende und Lehrkraft ausstatten
- Schließlich können beide Typen von Kind-Referenzen ein zusätzliches Kindelement zur Definition von Nullifikationen besitzen. Nullifikation beschreiben wir in Abschnitt 6.5.

Die Entwurfsentscheidung, dass Gewichte an Kanten und nicht an Knoten des Baumes gespeichert werden, ermöglicht, Teiltestergebnisse an verschiedenen Stellen der Bewertungshierarchie mehrfach mit unterschiedlichen Gewichten einzubinden. Der Baum wird dadurch zum gerichteten, azyklischen Graphen. Dies ist ein Feature, das wir in Abschnitt 5.3.7 genutzt hatten. Wir geben ein weiteres Beispiel zur Nutzung eines azyklischen Graphen: Dazu denken wir uns eine Bewertungsvorschrift mit inneren Knoten für die Bewertungskategorien Effizienz und Wartbarkeit. Zudem sei ein Teiltest vorhanden, der prüft, ob toter Code in der Einreichung vorliegt. Toter Code ist in erster Linie ein Wartbarkeitsproblem. In zweiter Linie, da toter Code die Codegröße unnötig erhöht, liegt auch ein Effizienzverstoß vor. Man kann nun das Teiltestergebnis mit einem vergleichsweise hohen Gewicht in die Wartbarkeitsbewertung und mit einem geringeren Gewicht in die Effizienzbewertung einbinden.

Auch wenn der Baum nun zum azyklischen Graphen geworden ist, soll für alle `combine`-Knoten weiterhin gelten, dass sie einen eindeutigen Vorgänger besitzen. Lediglich Blattknoten – das sind die durch `test-ref`-Kanten referenzierten Knoten – besitzen u. U. einen fan-in größer als 1.

6.2 Ein Beispiel eines nachzubildenden Bewertungsschemas

Da wir in diesem Unterabschnitt noch keine Nullifikationen einsetzen und ein Bewertungsschema wie das in Abbildung 8 dargestellte ohne Nullifikationen sehr komplex würde, nutzen wir hier zunächst ein einfacheres Bewertungsschema (s. Abbildung 13).

	Bewertung			
	Stufe A	**Stufe B**	**Stufe C**	**Stufe D**
Dimension	**1**	**0,7**	**0,3**	**0**
Syntax (10%)	Fehlerfrei			Fehlerhaft
Funktionalität (60%)	Beide Methoden funktionieren	Nur die Methode calcRadius funktioniert.	Nur die Methode calcArea funktioniert.	Nichts funktioniert
Wartbarkeit (30%)	Alle Wartbarkeitsregeln wurden eingehalten			Mind. eine Wartbarkeitsregel ist verletzt

Abbildung 13: Ein einfaches Bewertungsschema

6.3 XML-Fragment zum Beispiel

Wir zeigen in Abbildung 14 die Umsetzung als Bewertungsvorschrift. Diese ist in Abbildung 15 illustriert. Der Wurzelknoten referenziert drei Kindknoten, wovon der erste eine Testreferenz ist. Die Testreferenz auf den `compile`-Test bezeichnet den gesamten Compilierungstest, d. h. bei der Compilierung berücksichtigen wir keine Teiltestergebnisse, sondern nur das Gesamtergebnis der Compilierung. Anders bei den Kindreferenzen unterhalb der Knoten mit den `id`-Werten `functionality` und `maintainability`. Einzelne JUnit-Testergebnisse referenzieren wir durch die `sub-ref`-Angabe der Testmethode[14], einzelne PMD rules bzw. einzelne Checkstyle checks referenzieren wir durch die `sub-ref`-Angabe des Namens der rule bzw. des checks. Alle Testreferenzen haben wir mit einem Titel ausgestattet, der eine für Studierende und Lehrkräfte verständliche Anzeige des Bewertungsschemas durch das LMS ermöglicht.

Im Bewertungsschema der Abbildung 13 wird von Studierenden gefordert, alle Wartbarkeitsregeln einzuhalten. Die drei Testreferenzen auf den Checkstyle check `OneStatementPerLine` sowie auf die PMD rules `MethodNamingConventions` und `CommentRequired` repräsentieren je ein Wartbarkeitsprüfungsergebnis. Jedes Wartbarkeitsprüfungsergebnis ist entweder 0 oder 1, je nachdem, ob PMD bzw. Checkstyle eine Regelverletzung entdeckt haben. Die Werte werden mit einer Minimum-Funktion verknüpft, d. h. wenn mindestens eine Regel verletzt ist, erhält der Student am Knoten `maintainability` 0 Punkte.

```
<p:grading-hints>
  <p:root function="sum">
    <p:test-ref ref="compile" weight="0.1">
      <p:title>Syntax</p:title>
    </p:test-ref>
    <p:combine-ref ref="functionality" weight="0.6"/>
    <p:combine-ref ref="maintainability" weight="0.3"/>
  </p:root>
  <p:combine id="functionality" function="sum">
    <p:title>Functionality</p:title>
    <p:test-ref ref="junit" sub-ref="dom.circle.Tester#testRtoA" weight="0.3">
      <p:title>Should calculate area</p:title>
    </p:test-ref>
    <p:test-ref ref="junit" sub-ref="dom.circle.Tester#testAtoR" weight="0.7">
      <p:title>Should calculate radius</p:title>
    </p:test-ref>
  </p:combine>
  <p:combine id="maintainability" function="min">
    <p:title>Style</p:title>
    <p:test-ref ref="checkstyle" sub-ref="OneStatementPerLine">
      <p:title>There should be one statement per line</p:title>
    </p:test-ref>
    <p:test-ref ref="pmd" sub-ref="MethodNamingConventions">
      <p:title>Method names should follow conventions</p:title>
    </p:test-ref>
    <p:test-ref ref="pmd" sub-ref="CommentRequired">
      <p:title>Leading comments are required before public methods.</p:title>
    </p:test-ref>
  </p:combine>
</p:grading-hints>
```

Abbildung 14: XML-Definition einer Bewertungsvorschrift

[14] Das Format des `sub-ref`-Attributs ist abhängig vom Testwerkzeug. Bei JUnit verwenden wir den voll qualifizierten Namen der JUnit-Testklasse, gefolgt von einer Raute, gefolgt vom Namen der Testmethode.

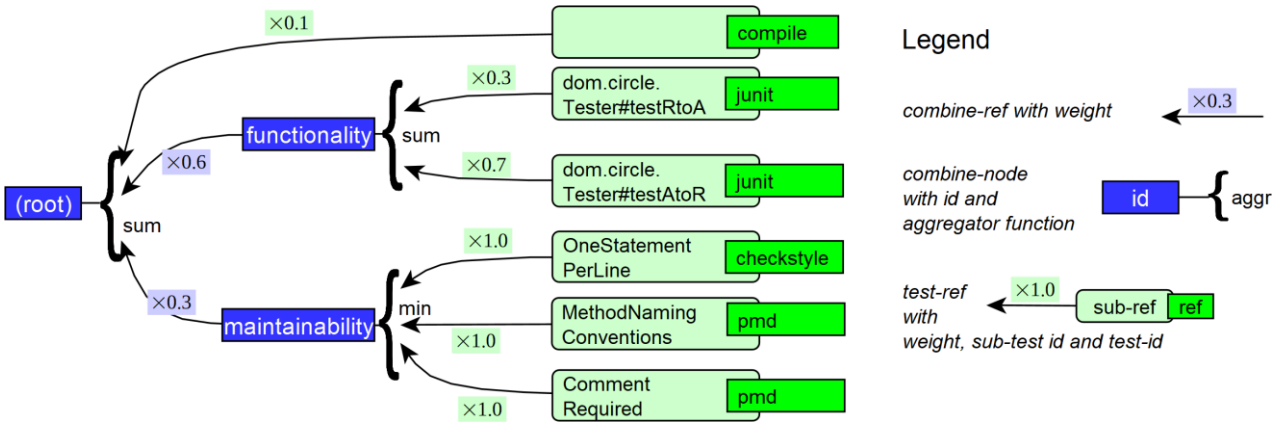

Abbildung 15: Illustration einer Bewertungsvorschrift. Die Pfeile sind hier von den Blättern des Baumes in Richtung der Wurzel notiert, weil diese Richtung beschreibt, wie aus Teilergebnissen sukzessive durch mehrere Rechenoperationen ein Gesamtergebnis wird.

6.4 Darstellung für Studierende

Beispielhaft zeigt Abbildung 16 eine mögliche Darstellung der Bewertungsvorschrift durch ein LMS. Hier wurde eine Gewicht-Darstellung gewählt, die nach unserer Erfahrung studentischen Wünschen näherkommt als die Definition in Abbildung 14. Die Gewichte wurden entlang aller Pfade von der Wurzel bis zu den Blättern „gepushed“ und dabei aufmultipliziert[15]. Im Ergebnis sind die Kanten zu Blättern des Baumes die einzigen Knoten, die von 1 abweichende Gewichte besitzen.[16]

Auf diese Weise kann jedem einzelnen Blatt-Aspekt direkt angesehen werden, welchen Einfluss dieser auf das Gesamtergebnis ausübt [R6]. Eine Skalierung dieser Darstellung auf eine andere von der Lehrkraft gewünschte Gesamtpunktzahl ist entsprechend einfach umsetzbar (vgl. auch [R3]).

Aspect	Score
Overall result	*1,00*
Syntax	0,10
Functionality	*0,60*
Should calculate area	0,18
Should calculate radius	0,42
Maintainability	*0,30*
Score calculation: • Min of the following sub aspects (raw score) • Multiplication by a weight factor 1,00	
There should be one statement per line	0,30
Method names should follow conventions	0,30
Leading comments are required before public methods.	0,30

Abbildung 16: Mögliche Darstellung der Bewertungsvorschrift im LMS. Diese Darstellung ist nicht Teil des ProFormA-Formats, sondern eine mögliche Umsetzung durch LMS oder Grader. Hier hat der Student bereits Zusatzinformationen zur Berechnungsvorschrift der Bewertungskategorie Maintainability einblenden lassen, indem er das kleine Informations-Icon betätigt hat. Die Darstellung ist interaktiv navigierbar. Bspw. kann der Student alle Kindelemente der Kategorie Functionality mit dem ⩔-Icon ausblenden.

6.5 Beschreibung des XML-Datenformats der Nullifikationen

In Abschnitt 5.4 wurden Nullifikationen als Bedingungen eingeführt, die einige Test- und Bewertungsergebnisse prüfen, um abhängig vom Ergebnis an anderer Stelle ein

[15] Da alle `combine`-Knoten einen eindeutigen Vorgänger haben, ist dies eindeutig möglich.
[16] Den Sonderfall des Gewichts 0 werden wir in Abschnitt 6.13 beleuchten.

Bewertungsergebnis auf 0 zu setzen. Abschnitt 6.1 definiert, dass Nullifikationen als Teil einer Kindreferenz spezifiziert werden.

Eine Nullifikation ist eines der beiden folgenden Objekte (vgl. auch Abbildung 12):

- Eine Vergleichsbedigung als `nullify-condition`-Element. Dieses definiert eine boolesche Bedingung mit einem als `compare-op`-Attribut spezifizierten Vergleichsoperator aus der Menge { `eq`, `ne`, `lt`, `le`, `gt`, `ge` } (in den naheliegenden Bedeutungen $=$, $\neq$, $<$, $\leq$, $>$, $\geq$). Als Operanden einer solchen Nullifikation werden zwei Kindelemente erwartet, die unabhängig voneinander aus den folgenden drei Typen gewählt werden können:
  - ein Zahlliteral: ein `nullify-literal`-Element mit `value`-Attribut
  - eine Referenz auf einen `combine`-Knoten, realisiert durch ein `nullify-combine-ref` Element mit `ref`-Attribut,
  - eine Referenz auf ein Testergebnis oder ein Teiltestergebnis, realisiert durch ein `nullify-test-ref` Element, welches ein `ref`-Attribut und ein optionales `sub-ref`-Attribut besitzt.
- Eine zusammengesetzte Bedingung als `nullify-conditions`-Element (beachten Sie das `s` am Ende). Dieses definiert eine boolesche Bedingung mit einem als `compose-op`-Attribut realisierten Verknüpfungsoperator aus der Menge { `and`, `or` }. Als Operanden einer solchen Nullifikation werden 2 oder mehr Nullifikationen als Kindelemente erwartet, also rekursiv wiederum Vergleichsbedingungen oder zusammengesetzte Bedingungen.

Beide Elemente (`nullify-condition` und `nullify-conditions`) erlauben zusätzlich die Angabe von Beschreibungsinformationen durch Kindelemente `title`, `description`, `internal-description`.

6.6 „Rohe“ vs. gewichtete Ergebnisse in Nullifikation-Vergleichsbedingungen – Definition vs. Darstellung

Die durch ein `nullify-combine-ref` Element bzw. durch ein `nullify-test-ref` Element referenzierten Ergebnisse meinen immer den ungewichteten Wert des referenzierten Knotens. Dies ist eine bewusste Entwurfsentscheidung, die dem Aufgabenautor die Formulierung von Bedingungen erleichtert. Er kann Nullifikationsbedingungen immer auf der „rohen“ Ergebnisskala [0,1] formulieren. Auch die Anpassung einer Bewertungsvorschrift durch Lehrende auf den jeweiligen Lehrkontext ist durch diese Entwurfsentscheidung einfacher [R3], weil bei einer Anpassung der Gewichte die Nullifikationsbedingungen i. d. R. nicht mitangepasst werden müssen. Die Darstellung für Studierende wird durch diese Entwurfsentscheidung etwas komplizierter, was wir im folgenden Absatz beleuchten wollen.

Wie Abschnitt 6.4 demonstriert, ist die Darstellung der Kantengewichte für Studierende in einer Form vorteilhaft, dass sie als integraler Bestandteil des von der Kante referenzierten Knoten verstanden werden können. Studenten stellen ja – wie erwähnt – die Frage „Wie viele Punkte gibt es für den Aspekt ‚Kommentierung‘?“ und nicht die Frage „Mit welchem Gewicht wird das dem Intervall [0,1] entstammende Ergebnis des Aspekts ‚Kommentierung‘ multipliziert?“. Die in Abschnitt 6.4 vorgeschlagene Darstellung hat Konsequenzen für die Darstellung von Nullifikationsbedingungen, deren Operanden sich ja – aus den im vorigen Absatz erläuterten Gründen – auf die „rohen“ Ergebnisse beziehen sollen. Für Darstellungszwecke müssen die rohen Ergebnisse der Vergleichsoperanden so umgerechnet werden, dass sie für Studierende aussehen, als bezögen sich die Nullifikationsbedingungen auf gewichtete Werte. Einen entsprechenden Algorithmus werden wir in Abschnitt 6.14 formulieren.

Für den Moment halten wir fest:

- Zur Erfüllung der Anforderung [R6] werden (Teil-)testergebnisse und (Teil-)bewertungsergebnisse dem Studenten stets in gewichteter Form angezeigt. Ungewichtete, „rohe“ Ergebnisse sind, wenn überhaupt, nur in Detailansichten zu sehen.

- Wir halten ein auf diese Weise gewichtet dargestelltes (Teil-)ergebnis fest und nennen den betreffenden Knoten n und die[17] den Knoten n referenzierende Kante e=e(n). Die Darstellung r(n) besteht aus einem Titel[18] t (entnommen aus n, falls e eine `combine-ref`-Kante ist, oder aus e, falls e eine `test-ref`-Kante ist) und einer maximal erreichbaren Punktzahl p, welche im Zuge der oben erwähnten „Push"-Operation aus den Gewichten an e und ihren Vorgängerkanten berechnet wurde: r=r(n)=(t,p).
- Eine Nullifikation-Vergleichsbedingung bezieht sich auf einen Knoten wie n. Im Datenformat wird dieser Bezug als ungewichtetes Ergebnis interpretiert (wegen Anforderung [R3]). Im Sinne einer konsistenten Darstellung für Studierende (Anforderung [R6]) muss der Bezug auf das gewichtete Ergebnis umgerechnet werden. Die Darstellung der Nullifikation-Vergleichsbedingung soll dazu den Titel t enthalten. Dargestellte Literal-Operanden der Vergleichsbedingung sollen auf die Punktzahl p umgerechnet dargestellt werden.
- Da nach dem vorigen Spiegelpunkt die Darstellung eines Nullifikation-Vergleichsbedingungs-Bezugs direkt aus r=(t,p) abgeleitet wird, erübrigt sich die Speicherung eines Titels und eines Gewichts in der Vergleichsbedingung. Wir entscheiden uns daher bewusst dafür, in den Elementen der Typen `nullify-combine-ref` und `nullify-test-ref` keine Titel, keine Gewichte und auch weder eine Beschreibung noch eine interne Beschreibung zu speichern.
- Im Gegensatz dazu entscheiden wir uns, in Elementen des Typs `nullify-condition` sowohl Titel als auch Beschreibungen zu speichern. Der Hintergrund dieser Entscheidung wird in Abschnitt 6.12 deutlich, wenn wir den Fall betrachten, dass e(n) nicht existiert oder nicht eindeutig ist.

6.7 Ein Beispiel eines mit einer einfachen Nullifikation nachzubildenden Bewertungsschemas

Wir ändern das Bewertungsschema aus Abschnitt 6.2 ab. Wartbarkeitspunkte (die „Kür") sollen annulliert werden, wenn die Funktionalität (die „Pflicht") nicht gut genug bewertet wurde. Abbildung 17 zeigt das neue Schema, welches sich nur in der unteren rechten Zelle von Abbildung 13 unterscheidet.

	Bewertung			
	Stufe A	**Stufe B**	**Stufe C**	**Stufe D**
Dimension	**1**	**0,7**	**0,3**	**0**
Syntax (10%)	Fehlerfrei			Fehlerhaft
Funktionalität (60%)	Beide Methoden funktionieren	Nur die Methode calcRadius funktioniert.	Nur die Methode calcArea funktioniert.	Nichts funktioniert
Wartbarkeit (30%)	Alle Wartbarkeitsregeln wurden eingehalten			Mind. eine Wartbarkeitsregel ist verletzt. Oder: Funktionalität wurde mit Stufe C oder schlechter bewertet.

Abbildung 17: Bewertungsschema für einfache Nullifikation

6.8 XML-Fragment zum Beispiel

Wir ändern das Beispiel aus Abbildung 14 ab. Aus Platzgründen und aus Gründen der Übersichtlichkeit lassen wir die Titel weg. Für die Kindreferenz vom Wurzelknoten in Richtung des

[17] Die Fälle keiner oder mehrerer eingehender Kanten werden in Abschnitt 6.12 betrachtet.
[18] Tatsächlich gehört zum Titel auch noch eine Beschreibung und eine interne Beschreibung. Zur Vereinfachung der Darstellung an dieser Stelle ignorieren wir zunächst Beschreibung und interne Beschreibung und holen dies in Abschnitt 6.14 nach.

`maintainability`-Knotens fügen wir eine Nullifikation ein, die das `maintainability`-Bewertungsergebnis dann auf 0 reduziert, wenn der Grad der funktionalen Korrektheit unter 40% liegt.

Abbildung 18 zeigt die XML-Definition, welche in Abbildung 19 illustriert ist.

```
<p:grading-hints>
  <p:root function="sum">
    <p:test-ref ref="compile" weight="0.1"/>
    <p:combine-ref ref="functionality" weight="0.6"/>
    <p:combine-ref ref="maintainability" weight="0.3">
      <p:nullify-condition compare-op="lt">
        <p:nullify-combine-ref ref="functionality"/>
        <p:nullify-literal value="0.4"/>
      </p:nullify-condition>
    </p:combine-ref>
  </p:root>
  <p:combine id="functionality" function="sum">
    <p:test-ref ref="junit" sub-ref="dom.circle.Tester#testRtoA" weight="0.3"/>
    <p:test-ref ref="junit" sub-ref="dom.circle.Tester#testAtoR" weight="0.7"/>
  </p:combine>
  <p:combine id="maintainability" function="min">
    <p:test-ref ref="checkstyle" sub-ref="OneStatementPerLine"/>
    <p:test-ref ref="pmd" sub-ref="MethodNamingConventions"/>
    <p:test-ref ref="pmd" sub-ref="CommentRequired"/>
  </p:combine>
</p:grading-hints>
```

Abbildung 18: XML-Definition einer Bewertungsvorschrift mit einfacher Nullifikation (fett hervorgehoben)

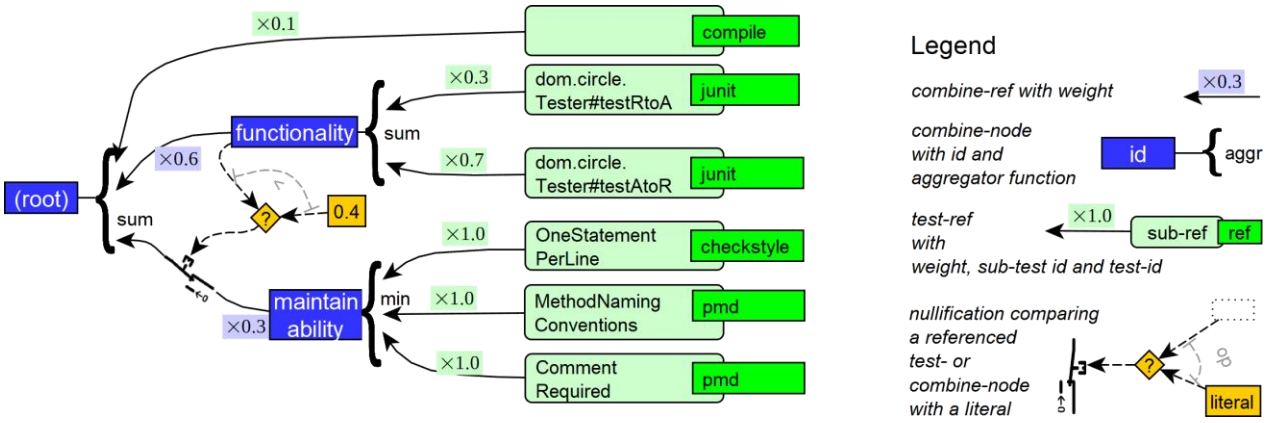

Abbildung 19: Illustration der XML-Definition aus Abbildung 18

Die Illustration in Abbildung 19 macht deutlich, dass sich die Nullifikationsbedingung auf das ungewichtete `functionality`-Bewertungsergebnis bezieht. Der gestrichelte Pfeil, der vom blau dargestellten `functionality`-Knoten ausgeht, zweigt ganz bewusst noch vor der Multiplikation mit dem Wert 0.6 ab. Falls die Vergleichsbedingung zutrifft, fließt in diesem Fall vom `maintainability`-Knoten kein Wert mehr an den Wurzelknoten, sondern stattdessen der Wert 0, was durch das Taster-Symbol illustriert wird.

6.9 Darstellung für Studierende

Abbildung 20 zeigt eine Präsentation im LMS. Studierende können erkennen, dass die Berechnungsregel für die Wartbarkeits-Kategorie eine Nullifikation enthält, die dann wirksam wird, wenn in der Kategorie Functionality weniger als 0,24 Punkte erreicht werden. Hier wurde das Vergleichs-Literal entsprechend der Anpassung aller anderen präsentierten Gewichte mitskaliert (40% von 0,6 sind 0,24). Das Beispiel demonstriert die Erfüllung der Anforderung [R6]. Die Erfüllung geht auf die Berücksichtigung der Ausführungen des Abschnitts 6.6 zurück. Der Student kann den Text „Nullification, if […] weighted *Functionality* score < 0,24“ im Kontext der Gesamtdarstellung verstehen. Er versteht, dass er von den maximal 0,60 möglichen

Functionality-Punkten (Zeile 4 in Abbildung 20) mind. 0,24 Punkte erreichen muss, um keine Nullifikation der Maintainability-Punkte zu erleiden.

Aspect	Score
Overall result	*1,00*
Syntax	0,10
Functionality	*0,60*
Should calculate area	0,18
Should calculate radius	0,42
Maintainability	*0,30*
Score calculation: • Min of the following sub aspects (raw score) • Multiplication by a weight factor 1,00 • Nullification, if the following condition is true: True, if weighted *Functionality* score < 0,24	
There should be one statement per line	0,30
Method names should follow conventions	0,30
Leading comments are required before public methods.	0,30

Abbildung 20: Mögliche Darstellung der Bewertungsvorschrift aus Abbildung 18.

6.10 Ein Beispiel eines mit einer zusammengesetzten Nullifikation nachzubildenden Bewertungsschemas

Manchmal möchte man Nullifikationen aus mehreren Bedingungen zusammensetzen. In unserem Beispiel erkennen wir in Abbildung 3, dass die beiden JUnit-Testmethoden jeweils mehrere Testdaten an die studentische Einreichung senden und die beobachteten Ausgaben prüfen. Wir sehen, dass jede der beiden Testmethoden einen Wert aus [0,1] als Ergebnis liefert, der jeweils den Prozentsatz der erfolgreich getesteten Testdaten benennt.

Die Syntax-Bewertung soll nun auf 0 reduziert werden, wenn nur sehr wenige der Testdaten erfolgreich geprüft wurden. Die Idee sei dabei, dass ein syntaktisch korrektes Programm, das nicht einmal einfachste Funktionalität aufweist, überhaupt nicht punktewürdig ist. Dabei erwarten wir im Falle der ersten Testmethode, die die vermeintlich einfacherer Aufgabe darstellt, dass mehr als 20% der Testdaten korrekt verarbeitet werden. Im Falle der zweiten Testmethode für den vermeintlich schwereren Aufgabenteil begnügen wir uns mit einem Prozentsatz, der von 0% abweicht. Anders formuliert: die Bewertung der Syntax wird auf 0 reduziert, wenn das Bewertungsergebnis für „Should calculate area“ kleiner oder gleich 20% ist UND das Bewertungsergebnis für „Should calculate radius“ gleich 0% ist.

Abbildung 21 zeigt das im Vergleich zu Abbildung 17 nur in der oberen rechten Zelle veränderte Bewertungsschema, wobei wir die Prozentangaben 20% und 0% auf die je Testmethode bearbeiteten fünf Testfälle umgerechnet haben.

	Bewertung			
	Stufe A	**Stufe B**	**Stufe C**	**Stufe D**
Dimension	**1**	**0,7**	**0,3**	**0**
Syntax (10%)	Fehlerfrei			Fehlerhaft. Oder: calcRadius besteht max. einen Testfall und calcArea keinen Testfall.
Funk-tiona-lität (60%)	Beide Methoden funktionieren	Nur die Methode calcRadius funk-tioniert.	Nur die Methode calcArea funk-tioniert.	Nichts funktioniert
Wart-barkeit (30%)	Alle Wartbarkeits-regeln wurden eingehalten			Mind. eine Wartbarkeitsregel ist verletzt. Oder: Funktionalität wurde mit Stufe C oder schlechter bewertet.

Abbildung 21: Bewertungsschema für zusammengesetzte Nullifikation

6.11 XML-Fragment zum Beispiel

Das entsprechende XML-Element ist in Abbildung 22 zu sehen, welches wir wie gewohnt in Abbildung 23 visualisieren.

```
<p:grading-hints>
  <p:root function="sum">
    <p:test-ref ref="compile" weight="0.1">
      <p:nullify-conditions compose-op="and">
        <p:nullify-condition compare-op="le">
          <p:nullify-test-ref ref="junit" sub-ref="dom.circle.Tester#testRtoA"/>
          <p:nullify-literal value="0.2"/>
        </p:nullify-condition>
        <p:nullify-condition compare-op="eq">
          <p:nullify-literal value="0.0"/>
          <p:nullify-test-ref ref="junit" sub-ref="dom.circle.Tester#testAtoR"/>
        </p:nullify-condition>
      </p:nullify-conditions>
    </p:test-ref>
    <p:combine-ref ref="functionality" weight="0.6"/>
    <p:combine-ref ref="maintainability" weight="0.3">
      <p:nullify-condition compare-op="lt">
        <p:nullify-combine-ref ref="functionality"/>
        <p:nullify-literal value="0.4"/>
      </p:nullify-condition>
    </p:combine-ref>
  </p:root>
  <p:combine id="functionality" function="sum">
    <p:test-ref ref="junit" sub-ref="dom.circle.Tester#testRtoA" weight="0.3"/>
    <p:test-ref ref="junit" sub-ref="dom.circle.Tester#testAtoR" weight="0.7"/>
  </p:combine>
  <p:combine id="maintainability" function="min">
    <p:test-ref ref="checkstyle" sub-ref="OneStatementPerLine"/>
    <p:test-ref ref="pmd" sub-ref="MethodNamingConventions"/>
    <p:test-ref ref="pmd" sub-ref="CommentRequired"/>
  </p:combine>
</p:grading-hints>
```

Abbildung 22: XML-Definition einer Bewertungsvorschrift mit zusammengesetzter Nullifikation (fett hervorgehoben)

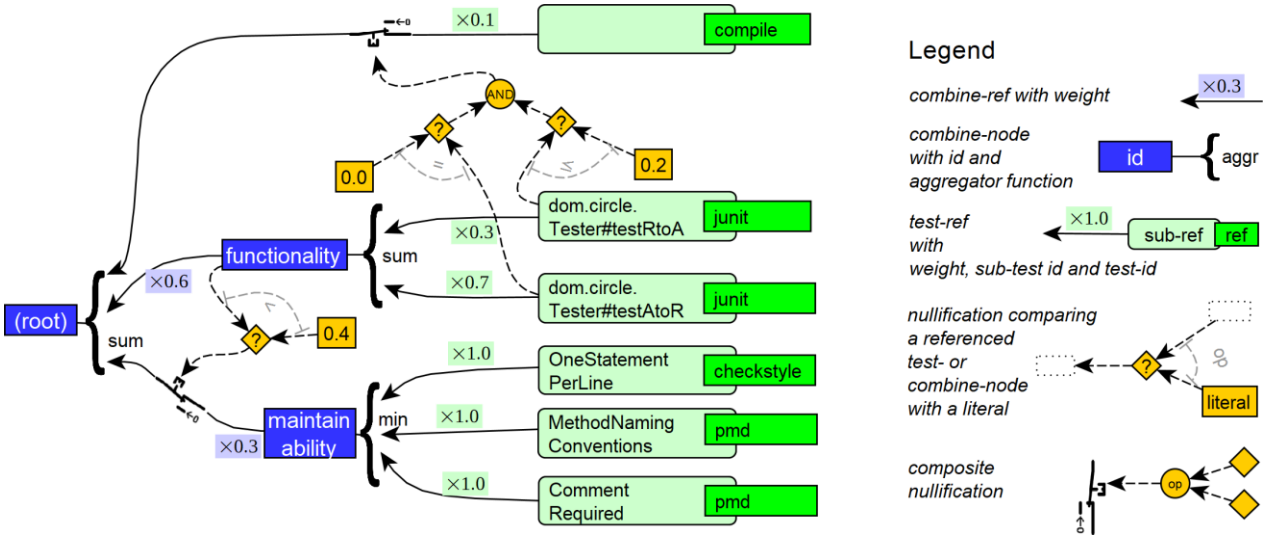

Abbildung 23: Illustration der XML-Definition aus Abbildung 22

6.12 Vorbereitungen zur Darstellung für Studierende

Bei der Darstellung im LMS ist zu beachten, dass sich in diesem Beispiel die Vergleichsoperanden nicht auf innere Knoten, sondern auf Blätter beziehen. Wenn wir wie in Abschnitt 6.4 beschrieben die Gewichte zu den Blättern „pushen“, dann sind die Kanten zu den

Blättern die einzigen Kanten, deren Gewichte von 1 abweichen (es sei denn, die Ursprungsgewichte sind 0, aber dazu kommen wir gleich). Wir stellen in Abbildung 24 den Baum dar, der durch die in Abschnitt 6.4 beschriebene Push-Operation entsteht.

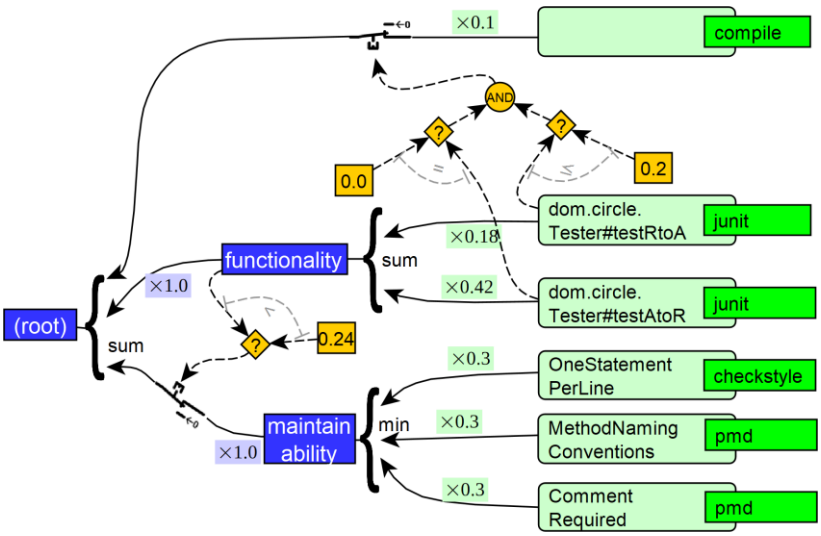

Abbildung 24: Bewertungsbaum zu Abbildung 22, der durch „Pushen“ der Gewichte zu den Blättern entsteht.

In Abbildung 24 ist zu erkennen, dass die gestrichelten Pfeile, die von den Teiltestergebnissen `dom.circle.Tester#testRtoA` und `dom.circle.Tester#testAtoR` ausgehen, so gezeichnet wurden, dass sie noch vor der Multiplikation mit dem Gewicht 0.18 bzw. 0.42 abzweigen. Das ist kein Zufall, sondern Ergebnis der in Abschnitt 6.5 beschriebenen Entwurfsentscheidung. Auch der gestrichelte Pfeil, der vom `functionality`-Knoten abzweigt, transportiert den ungewichteten Wert. Da hier das Gewicht aber sowieso 1 beträgt, kommt der Unterschied bei inneren Knoten nicht zum Tragen.

6.13 Bedingungen für eine einfache Darstellbarkeit einer Nullifikations-Vergleichsbedingung

In dem `test-ref`-Element, das den o. g. Teiltest referenziert, kann man ein Gewicht, einen Titel und Beschreibungen angeben. Dies ist jedoch nicht vorgesehen für `nullify-test-ref`-Elemente (vgl. Abschnitt 6.6). D. h. in der Darstellung für Studierende gibt es für die gestrichelten Pfeile, die von den Teiltestergebnissen `dom.circle.Tester#testRtoA` und `dom.circle.Tester#testAtoR` ausgehen, keine Titel-Daten und keine Beschreibungsdaten. Um Studierenden etwas Besseres als „dom.circle.Tester#testRtoA“ anzeigen zu können, bietet es sich an, die beiden Titel und Gewichte wiederzuverwenden, die an den normalen `test-ref`-Kanten gespeichert sind, die in Abbildung 24 die Gewichte 0.18 und 0.42 tragen (vgl. Abschnitt 6.6). Dies geht für jede der beiden `nullify-test-ref`-Elemente allerdings nur unter ganz bestimmten Bedingungen, die wir hier beispielhaft für eines der beiden Elemente (`dom.circle.Tester#testRtoA`) beschreiben:

- Es gibt mindestens eine eingehende `test-ref`-Kante am Knoten `dom.circle.Tester#testRtoA`. Dies ist die Existenz der in Abschnitt 6.6 genannten Kante e.
- Das Gewicht an der eingehenden Kante ist von 0 verschieden. Wäre es 0, könnte man die Nullifikationsbedingung nicht verlustfrei für Darstellungszwecke skalieren (so wie es in Abbildung 20 geschah).
- Falls es mehrere eingehende Kanten am Knoten `dom.circle.Tester#testRtoA` gibt, sind die an diesen Kanten notierten Gewichte, Titel und Beschreibungen konsistent.

So lange diese drei Bedingungen alle erfüllt sind, kann folgender Algorithmus zur Darstellung der Nullifikationsbedingung die Anforderung [R6] erfüllen.

6.14 Algorithmus

Gegeben sei ein `nullification-condition`-Element, das genau einen Operanden vom Typ `nullify-literal` besitzt. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass dies der zweite Operand ist. Der erste Operand ist dann vom Typ `nullify-combine-ref` oder `nullify-test-ref`. Da der Algorithmus in beiden Fällen nahezu identisch verläuft, nehmen wir zur Vereinfachung der Beschreibung an, es handele sich um ein `nullify-combine-ref`-Element. D. h. wir gehen von folgendem `nullification-condition`-Element aus und wollen dieses für Studierende darstellen:

```
<p:nullify-condition compare-op="o">
  <p:nullify-combine-ref ref="x"/>
  <p:nullify-literal value="v"/>
  <p:title>t</p:title>
  <p:description>d</p:description>
  <p:internal-description>i</p:internal-description>
</p:nullify-condition>
```

Die Nullifikations-Vergleichsbedingung ist also durch die sechs Variablen (o, x, v, t, d, i) beschrieben. Einige Kind-Elemente können fehlen und auch das `compare-op`-Attribut kann fehlen. Dann greifen jeweils die folgenden Standardwerte `o`=min, `t=d=i`=leere Zeichenkette.

a)

Wir setzen zunächst voraus, dass das `nullify-combine-ref`-Element die in Abschnitt 6.13 formulierten Bedingungen erfüllt, d. h. es gibt in der gesamten Bewertungsvorschrift genau ein `combine-ref`-Element, dessen `ref`-Attribut mit x übereinstimmt:

```
<p:combine-ref ref="x" weight="w">
  <p:title>t'</p:title>
  <p:description>…</p:description>
  <p:internal-description>…</p:internal-description>
</p:combine-ref>
```

Einige oder sogar alle Kind-Elemente dieses `combine-ref`-Elements können fehlen und auch das `weight`-Attribut kann fehlen. Dann greifen jeweils die Standardwerte `w`=1, `t'`=leere Zeichenkette. Das `combine-ref`-Element ist also durch die drei Variablen (x, w, t‘) beschrieben.

Eine Darstellung für Studierende ist nun wie folgt umsetzbar:

> *„Nullification, if the following condition is true: “ <t>*
> *<d und/oder i>*
> *„True, if weighted “ <t’> „ score “ <flip(o)> <w×v>*

wobei die in Anführungszeichen gesetzten Teile buchstäblich angezeigt werden und die in spitzen Klammern gesetzten Teile aus den oben beschrieben Variablen gefüttert werden.

- Mit *<w×v>* ist das Produkt der Zahlwerte w und v gemeint.
- Die Funktion flip(o) kehrt den Vergleichsoperator o um, wenn w negativ[19] ist. Damit bilden wir die mathematische Operation nach, die beim Multiplizieren einer Ungleichung mit einem negativen Faktor vorzunehmen ist: flip(lt) = gt, flip (le) = ge, flip(eq) = eq, etc.

b)

Wir nehmen nun an, dass mindestens eine der in Abschnitt 6.13 formulierten Bedingungen nicht erfüllt ist. In diesem Fall sind t‘ und w nicht existent, nicht eindeutig bestimmt oder w besitzt den Wert 0. In diesem Fall soll die Darstellung für Studierende wie folgt lauten:

> *„Nullification, if the following condition is true: “ <t>*
> *<d und/oder i>*

D. h. wir lassen die letzte Zeile einfach weg.

[19] Mit negativen Gewichten liegen seitens des Autors noch keine Erfahrungen vor. Bei allen bisher auf die neue Bewertungsvorschrift umgestellten Aufgaben haben sich ausschließlich positive Gewichte oder Gewichte mit dem Wert 0 als nützlich erwiesen.

Unangenehm wird es, wenn die Nullifikation-Vergleichsbedingung keine Werte für t, d und i definiert. Dieser Fall ist eigentlich als Versäumnis des Aufgabenautors zu klassifizieren. Dennoch können wir in diesem Fall die – für Studierende nicht mehr verständliche – Ausgabe produzieren:

„Nullification, if the following condition is true: “
„True, if raw “ <x> „ score “ <o> <v>

Diese Ausgabe lässt immerhin noch Raum für Interpretationen, wenn der Identifier x vielleicht (ähnlich wie die Identifier `functionality` oder `maintainability` im Beispiel in Abbildung 22) bedeutungstragend gewählt wurde.

Zusammenfassend kann der angegebene Algorithmus belegen, dass – solange kein Versäumnis des Aufgabenautors vorliegt – eine für Studierende verständliche Ausgabe der Nullifikations-bedingung automatisiert erstellt werden kann (Anforderung [R6]).

Ein entsprechender Algorithmus lässt sich leicht für ein `nullification-condition`-Element entwickeln, dessen beide Operanden vom Typ `nullify-combine-ref` oder `nullify-test-ref` sind.

6.15 Darstellung des Beispiels für Studierende

Abbildung 25 zeigt, wie das `nullify-literal` mit dem Originalwert 0.2 in den Wert 0.04 als Teil der Ausgabe „True, if weighted *Should calculate area* score ≤ 0,04“ transformiert wurde. Der Wert 0,04 ergibt sich aus der Multiplikation des Originalwerts 0,2 mit dem Gewicht 0,18, welches an der Kante zum Blatt „Should calculate are“ durch die „Push“-Operation berechnet wurde, mit anschließender Rundung auf zwei Nachkommastellen.

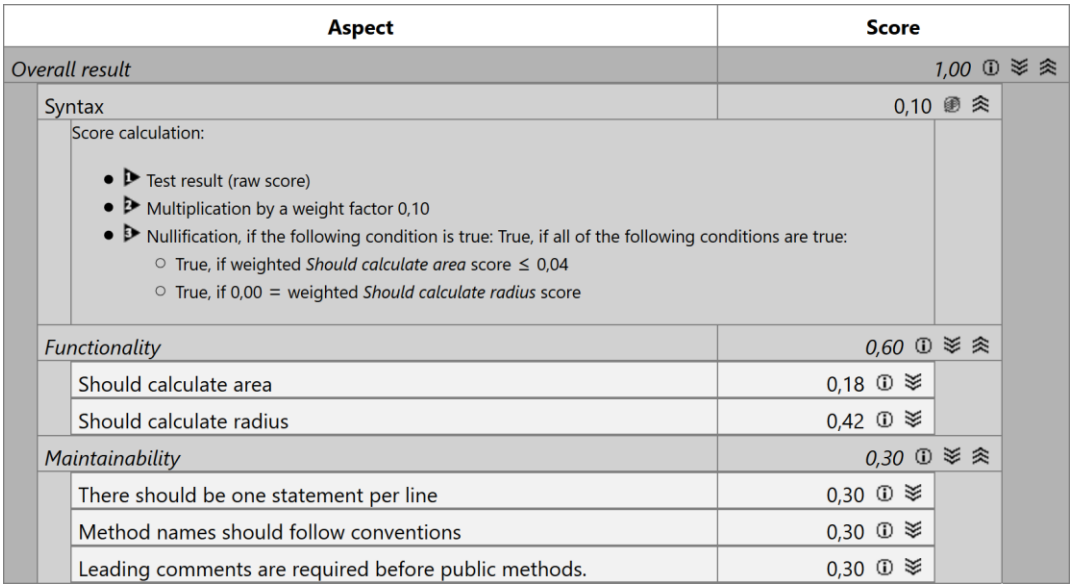

Abbildung 25: Mögliche Darstellung der Bewertungsvorschrift aus Abbildung 22

6.16 Das Beispiel vom Anfang

Wir nehmen das Beispiel-Bewertungsschema aus Abbildung 8 wieder auf. Leicht verändert in der Syntax-Zeile[20] zeigt es Abbildung 26. Hiermit greifen wir die Diskussion in Abschnitt 5.3.7 und 5.4 auf und führen die dort nur abstrakt und schematisch vorgestellten Ideen zu einer fertig implementierten Lösung.

[20] Für die gestufte Bewertung von Syntaxfehlern bietet der hier genutzte Grader Graja keine Testmöglichkeiten, weswegen wir das Bewertungsschema in der Syntax-Zeile vereinfacht haben.

Dimension	Bewertung			
	Stufe A	**Stufe B**	**Stufe C**	**Stufe D**
	1	**0,7**	**0,3**	**0**
Syntax (10%)	Fehlerfrei			Fehlerhaft. Oder: Trivialabgabe*.
Funktionalität (60%)	Beide Methoden funktionieren	Nur die Methode calcRadius funktioniert.	Nur die Methode calcArea funktioniert.	Nichts funktioniert
Wartbarkeit (30%)	Kommentare *korrekt* und Code ist stil*konform* (Anweisungen pro Zeile, Bezeichner).	Kommentare *korrekt, aber* Code *verletzt* Stilregeln (Anweisungen pro Zeile, Bezeichner)	Kommentare *falsch oder fehlen*, aber Code ist stil*konform* (Anweisungen pro Zeile, Bezeichner).	Kommentare *falsch oder fehlen* und Code *verletzt* Stilregeln (Anweisungen pro Zeile, Bezeichner). Oder: Trivialabgabe*.
* eine Trivialabgabe (bspw. Klasse ohne Methoden) gilt als Versagen in Syntax und Programmierstil (und in Funktionalität, aber das muss ja nicht extra erwähnt werden).				

Abbildung 26: Ein komplexes Bewertungsschema

6.17 XML-Fragment zum Beispiel

In Abbildung 28 zeigen wir die Umsetzung als XML-Fragment, welches in Abbildung 27 illustriert ist. Die Besonderheit bei diesem Beispiel ist, dass das JUnit-Teilergebnis `dom.circle.Tester#testMethod` ausschließlich für die Nullifikationen zur Feststellung einer Trivialabgabe eingesetzt wird und ansonsten nicht in das Gesamtergebnis eingeht. Wenn das Teilergebnis ungleich 1 ist, werden Syntax- und Wartbarkeitsergebnisses annulliert.

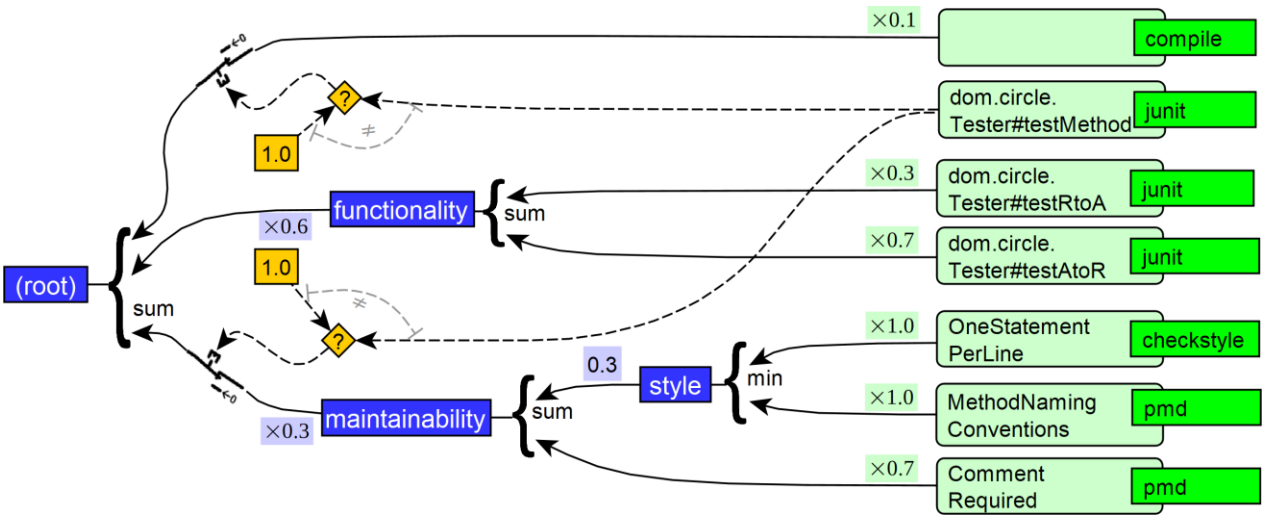

Abbildung 27: Illustration des XML-Fragments der Abbildung 28

```
<p:grading-hints>
  <p:root function="sum">
    <p:test-ref ref="compile" weight="0.1">
      <p:title>Syntax</p:title>
      <p:nullify-condition compare-op="ne">
        <p:title>the submitted class does not declare any methods</p:title>
        <p:nullify-test-ref ref="junit" sub-ref="dom.circle.Tester#testMethod"/>
        <p:nullify-literal value="1.0"/>
      </p:nullify-condition>
    </p:test-ref>
    <p:combine-ref ref="functionality" weight="0.6"/>
    <p:combine-ref ref="maintainability" weight="0.3">
      <p:nullify-condition compare-op="ne">
        <p:title>the submitted class does not declare any methods</p:title>
        <p:nullify-test-ref ref="junit" sub-ref="dom.circle.Tester#testMethod"/>
        <p:nullify-literal value="1.0"/>
      </p:nullify-condition>
    </p:combine-ref>
  </p:root>
  <p:combine id="functionality" function="sum">
    <p:title>Functionality</p:title>
    <p:test-ref ref="junit" sub-ref="dom.circle.Tester#testRtoA" weight="0.3">
      <p:title>Should calculate area</p:title>
    </p:test-ref>
    <p:test-ref ref="junit" sub-ref="dom.circle.Tester#testAtoR" weight="0.7">
      <p:title>Should calculate radius</p:title>
    </p:test-ref>
  </p:combine>
  <p:combine id="maintainability" function="sum">
    <p:title>Maintainability</p:title>
    <p:combine-ref ref="style" weight="0.3"/>
    <p:test-ref ref="pmd" sub-ref="CommentRequired" weight="0.7">
      <p:title>Leading comments are required before public methods.</p:title>
    </p:test-ref>
  </p:combine>
  <p:combine id="style" function="min">
    <p:title>Style</p:title>
    <p:test-ref ref="checkstyle" sub-ref="OneStatementPerLine">
      <p:title>There should be one statement per line</p:title>
    </p:test-ref>
    <p:test-ref ref="pmd" sub-ref="MethodNamingConventions">
      <p:title>Method names should follow conventions</p:title>
    </p:test-ref>
  </p:combine>
</p:grading-hints>
```

Abbildung 28: XML-Fragment zum Bewertungsschema der Abbildung 26

6.18 Darstellung des Beispiels für Studierende und Lehrkräfte

Um die Darstellung für Studierende zu unterstützen, wurde die Nullifikationsbedingung in Abbildung 28 mit einem erläuternden Titel versehen: „the submitted class does not declare any methods“. Dieser Titel taucht in Abbildung 29 an entsprechender Stelle wieder auf. Die Ausgabe eines für Studierende unverständlichen Identifiers, wie es am Ende des Abschnitts 6.14 diskutiert wurde, kann dann entfallen. Für Lehrkräfte, die Zugriff auf die Testdefinitionen haben, kann der Identifier enthalten sein, um eine eventuelle Fehlersuche zu erleichtern (s. Abbildung 30).

Aspect	Score
Overall result	*1,00*
Syntax	0,10
Score calculation: • Test result (raw score) • Multiplication by a weight factor 0,10 • Nullification, if the following condition is true: the submitted class does not declare any methods.	
Functionality	*0,60*
Should calculate area	0,18
Should calculate radius	0,42
Maintainability	*0,30*
Score calculation: • Sum of the following sub aspects (raw score) • Multiplication by a weight factor 1,00 • Nullification, if the following condition is true: the submitted class does not declare any methods.	
Style	*0,09*
There should be one statement per line	0,09
Method names should follow conventions	0,09
Leading comments are required before public methods.	0,21

Abbildung 29: Darstellung des Bewertungsschemas der Abbildung 28 für Studierende

Aspect	Score
Overall result	*1,00*
Syntax	0,10
Score calculation: • Test result (raw score) • Multiplication by a weight factor 0,10 • Nullification, if the following condition is true: the submitted class does not declare any methods. True, if Raw *junit/dom.circle.grader.Tester#testMethod* score ≠ 1,00	
Functionality	*0,60*
Should calculate area	0,18
Should calculate radius	0,42
Maintainability	*0,30*
Score calculation: • Sum of the following sub aspects (raw score) • Multiplication by a weight factor 1,00 • Nullification, if the following condition is true: the submitted class does not declare any methods. True, if Raw *junit/dom.circle.grader.Tester#testMethod* score ≠ 1,00	
Style	*0,09*
There should be one statement per line	0,09
Method names should follow conventions	0,09
Leading comments are required before public methods.	0,21

Abbildung 30: Darstellung des Bewertungsschemas der Abbildung 28 für Lehrkräfte.

6.19 Darstellung eines Bewertungsergebnisses

Bisher haben wir Bewertungsschemata dargestellt. Die Darstellung eines Bewertungs*ergebnisses* für eine konkrete Einreichung muss und sollte sich unserer Ansicht nach nicht grundlegend von der Struktur des Schemas unterscheiden. Abbildung 31 zeigt ein Ergebnis für eine Trivialabgabe, welches vom Studierenden vollständig aufgeklappt wurde. Der Berechnungsweg der Ergebnisse

für *Syntax* und *Maintainability* wird als „breadcrumb“ nachvollziehbar dargestellt. Am Beispiel der *Syntax* nennt der erste Schritt das „rohe“ Testergebnis. Der zweite Schritt bezeichnet den nach Gewichtung entstandenen Wert. Der letzte, mit 3▶ bezeichnete Schritt erläutert dem Studierenden den Grund für die Reduzierung der Punkte auf den Wert 0.

Innere Knoten erläutern die Aggregierungsstrategie, die in diesem Fall entweder die Summe oder das Minimum ist. Unnötige Berechnungsschritte sind in diesem Darstellungsvorschlag ausgeblendet worden. Bspw. wird für *Functionality* einfach die Summe der beiden Teilergebnisse dargestellt (0.00). Eine nachfolgende, tatsächlich stattgefundene Gewichtung ändert das Ergebnis nicht mehr und wird demnach auch nicht dargestellt.

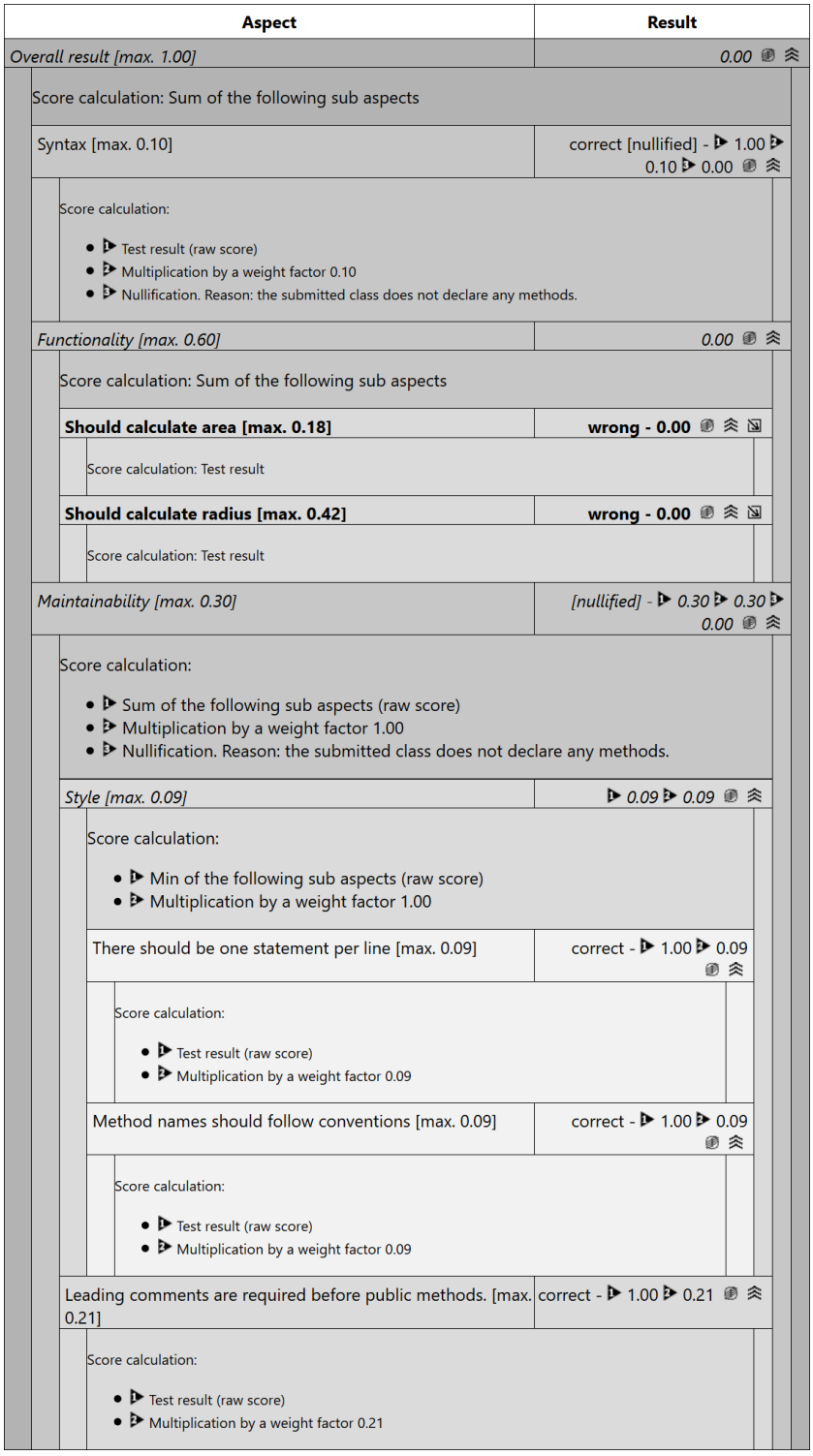

Abbildung 31: Darstellung eines Bewertungsergebnisses für eine Trivialabgabe

7 Zusammenfassung und weiterführende Arbeiten

7.1 Zusammenfassung

Zusammenfassend liegt ein Vorschlag zur Spezifikation von Bewertungsvorschriften für automatisiert bewertete Programmieraufgaben vor,

- der es Lehrenden ermöglicht, einzelne Bewertungsaspekte getrennt zu würdigen und gleichzeitig ein Gesamtergebnis berechnen zu lassen,
- der Studierende leicht den Einfluss gemachter Fehler auf das Gesamtbewertungsergebnis erkennen lässt,
- der die Besonderheiten von auf Regelverletzungen basierenden Testwerkzeugen berücksichtigt,
- der durch seine Struktur der verschachtelten Kategorien leicht auf neue Lehrkontexte anpassbar ist,
- und der in dieser Struktur einfach in verschiedensten LMS und Gradern umzusetzen ist und daher gut zu den Zielen des ProFormA-Formats passt.

Der vorgestellte Vorschlag basiert auf den Vorarbeiten in [3] und [8], in denen eine Hierarchie von Punktsummen um das Konzept der Referenzierung von Teiltests erweitert wurde. Im vorliegenden Vorschlag schließlich werden statt Summen auch andere Aggregationsstrategien unterstützt. Mit dem Konzept der Nullifikationen schließlich wird der Natur einiger Testwerkzeuge, Regelverletzungen zu zählen, Rechnung getragen.

7.2 Bewertung der Ergebnisse

Die ursprünglichen, in [3] und [8] vorgestellten Bewertungsvorschriften, werden an der Hochschule Hannover bereits seit mehreren Jahren im realen Lehrbetrieb in der Java- und SQL-Ausbildung erfolgreich eingesetzt.

Die im vorliegenden Beitrag vorgeschlagene Erweiterung wurde 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 Aufgaben auf das neue Format angepasst. Einige dieser Aufgaben wurden dabei als randomisierbare Aufgaben umgesetzt, so dass hier bereits eine Übertragbarkeit der Ergebnisse auf neue Kontexte gezeigt werden konnte.

Die Effekte des Einsatzes in Bezug auf Lernerfolge wurden nicht systematisch untersucht und es ist derzeit auch nicht geplant, dies zu tun. Maßgebliches Ziel bei der Umstellung der 50 Aufgaben war, die Bewertung des Programmierstils fairer und damit vermutlich lernförderlicher zu gestalten. Dies ist aus meiner persönlichen Sicht, der ich die genannte Lehrveranstaltung durchführe und die Nutzung der überarbeiteten Aufgaben im Rahmen von Rechnerraumübungen beobachte, geglückt.

Bzgl. des Forschungsziels der Übertragbarkeit des vorgeschlagenen Formats auf andere Bewertungssysteme und Programmiersprachen liegen noch keine konkreten Einsatzerfahrungen vor. Allerdings ist das Format inzwischen Teil der Version 2.0 des ProFormA-Formats und fügt sich somit nahtlos in ein übergreifend nutzbares Aufgabenformat ein. Die in Abschnitt 3 erwähnte Arbeitsgruppe hat seit Oktober 2017 in zweiwöchentlich abgehaltenen Telefonkonferenzen neue Entwicklungsideen bzgl. des hier vorgestellten Bewertungsformats diskutiert und diese Ideen in den dazwischenliegenden Zeiten regelmäßig im Rahmen von technischen Durchstichen und konzeptionellen Überlegungen mit den Erfordernissen der jeweils eigenen Infrastruktur abgeglichen. Die dabei gewonnenen Erkenntnisse flossen stetig in die Weiterentwicklung des Formats ein, dessen vorläufiger Endstand in diesem Beitrag präsentiert wurde.

7.3 Weiterführende Arbeiten

Das Format einer Bewertung steuert maßgeblich, wie eine Ausgabe eines Graders auszusehen hat. Das vom Grader generierte Feedback orientiert sich idealerweise an der Struktur des vorgegebenen Bewertungsschemas. Die Spezifikation eines diesbezüglichen Antwortformats (*response*) wurde in der ProFormA-Arbeitsgruppe aufbauend auf dem hier vorgeschlagenen Bewertungsformat vorgenommen[2]. Zudem wurde ebenfalls aufbauend auf dem Bewertungsformat ein Format für vom LMS an Grader erteilte Bewertungsaufträge (*submission*) definiert. Letzteres unterstützt die in diesem Beitrag aufgestellte Anforderung, dass Lehrkräfte die Bewertung an ihren Lehrkontext anpassen können, dadurch, dass die in einer Aufgabe (*task*) festgelegte Bewertungsvorschrift durch eine eigene Vorschrift der Lehrkraft ersetzt werden kann.

8 Quellen

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

[2] Bott, O. J., Fricke, P., Priss, U., Striewe, M. (Hrsg.): Automatisierte Bewertung in der Programmierausbildung, Waxmann, 2017

[3] Garmann, R., Heine, F., Werner, P. (2015): Grappa - die Spinne im Netz der Autobewerter und Lernmanagementsysteme, Tagungsband der 13. e-Learning Fachtagung Informatik (DeLFI), GI, Bonn.

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

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

[6] Stegeman, M., Barendsen, E., Smetsers, S. (2016): Designing a rubric for feedback on code quality in programming courses. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research (pp. 160-164). ACM.

[7] Jara, N., Madrid, M. M. (2015): Bewertungsschema für eine abgestufte Bewertung von Programmieraufgaben in E-Klausuren. DeLFI 2015--Die 13. E-Learning Fachtagung Informatik.

[8] Garmann, R., Fricke, P., Bott, O. J. (2016): Bewertungsaspekte und Tests in Java-Programmieraufgaben für Graja im ProFormA-Aufgabenformat, DeLFI 2016: Die 14. e-Learning Fachtagung Informatik, 11.-14. September 2016, Potsdam.

[9] Reinmann, G. (2019): Die Selbstbezüglichkeit der hochschuldidaktischen Forschung und ihre Folgen für die Möglichkeiten des Erkennens. In Hochschulbildungsforschung (pp. 125-148). Springer VS, Wiesbaden.

[10] Biggs, J. (1996). Enhancing teaching through constructive alignment. Higher Education, 32(3), 347–364.

[11] Panadero, E., Jonsson, A. (2013): The use of scoring rubrics for formative assessment purposes revisited: A review. Educational Research Review, 9 (0), 129-144.

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

[13] Striewe, M. (2017): Der Grader JACK. In: [2].

[14] Heine, F., Kleiner, C. (2017): Der Grader aSQLg. In: [2].

[15] Breitner, J., Hecker, M., Snelting, G. (2017): Der Grader Praktomat. In: [2].

[16] Iffländer, L., Dallmann, A. (1027): Der Grader PABS. In: [2].

[17] Herres, B., Oechsle, R., Schuster, D. (2017): Der Grader ASB. In: [2].

[18] Müller, O., Strickroth, S. (2017): Der Grader GATE. In: [2].

[19] Gust, H. (2017): Der Grader VEA. In: [2].