Ein Datenformat für variable Programmieraufgaben

Robert Garmann[1]

Bericht

9. August 2019

HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS

Fakultät IV
Wirtschaft und
Informatik

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

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

Zusammenfassung

Automatisiert bewertbare Programmieraufgaben dienen Studierenden zum Einüben von Programmierfertigkeiten. Die Verfügbarkeit von mehreren verschiedenen Aufgaben, die denselben Stoff abdecken, ist für verschiedene Zwecke hilfreich. Eine Programmieraufgabe lässt sich durch Einführung von Variationspunkten variabel gestalten. Die hierbei entstehende Aufgabenschablone ist Ausgangsbasis der sog. Materialisierung, der automatischen Generierung konkreter Aufgaben. Der vorliegende Beitrag stellt ein Datenmodell mit dem Ziel vor, sowohl die Auswahl von Variationspunktwerten als auch die automatische Materialisierung auf verschiedenen Systemen in verschiedenen Programmiersprachen zu unterstützen. Das vorgeschlagene Datenformat ermöglicht Lernmanagementsystemen die Unterstützung variabler Programmieraufgaben bei gleichzeitiger Unkenntnis des eingesetzten Autobewerters.

Schlagwörter

Individuelle Programmieraufgaben, Grader, Autobewerter, E-Assessment, Variabilität, ProFormA, automatisierte Bewertung

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* • *Software and its engineering~Software product lines*

Inhalt

1 Einleitung

Automatisiert bewertbare Programmieraufgaben dienen Studierenden zum Einüben von Programmierfertigkeiten. Die Verfügbarkeit mehrerer verschiedener Aufgaben, die denselben Stoff abdecken, ist für verschiedene Zwecke hilfreich und kann durch automatische Generierung mehrerer Aufgabenvarianten aus einer Aufgabenschablone erreicht werden. Einerseits kann durch individuelle Zuteilung von verschiedenen Aufgaben an Studierende die Gefahr von Plagiaten reduziert werden. Andererseits wünschen sich manche Studierende zum intensiven Üben mehrere verschiedene Aufgaben zum selben Stoff. Darüber hinaus kann es hilfreich sein, Aufgabenvarianten unterschiedlichen Schwierigkeitsgrades an verschiedene Studierende auf der Basis vergangener Lernleistungen zu vergeben. Weitere Anwendungsfälle und zu berücksichtigende didaktische Aspekte werden etwa in [5] diskutiert.

Programmieraufgaben lassen sich durch Einführung von sog. Variationspunkten (*vp*) variabel gestalten [2]. Beispiele für Variationspunkte reichen von einfachen Bezeichnern über konkrete Datenwerte bis hin zu Konstrukten der verwendeten Programmiersprache und Bewertungsaspekten. Die folgende Tabelle stellt für eine objektorientierte Programmiersprache und für eine Abfragesprache Beispiele für Variationspunkte dar.

	Java	**SQL**
Bezeichner	Programmierung einer Funktion zur Berechnung des Kreisflächeninhalts **vp:** Name der Funktion	Selektion einer Tabellenspalte einer gegebenen Tabelle **vp**: Name der Tabellenspalte
Funktion / Algorithmus	Berechnung des Flächeninhalts einer geometrischen Form **vp:** Vorgegebene Form (Kreis, …)	Aggregation einer Tabellenspalte **vp**: Aggregatfunktion
Datenwerte	Programmierung einer Funktion zur Berechnung des Kreisflächeninhalts **vp**: Vorgabe, ob und wie ungültige Kreisradien zu behandeln sind	Selektion der Datensätze mit vorgegebenem Attributwert **vp**: Der vorgegebene Attributwert
Sprach-konstrukt	Programmierung einer Schleife **vp**: Vorgabe, ob eine for- oder while-Schleife programmiert werden soll	Verknüpfung zweier Tabellen **vp**: Implementierung durch JOINs oder durch Unterabfragen
Bewertungs-aspekt	**vp**: Bewertungsgewicht des Programmierstils (nicht bewertet, bewertet mit Gewicht w, …)	
Schwierigkeit	**vp:** Schwierigkeit der Aufgabenvariante (ableitbar aus allen anderen Variationspunkten)	

Tabelle 1: Beispiele für Variationspunkte (vp)

Stattet man eine Programmieraufgabe mit Variationspunkten aus, so entsteht eine Aufgaben-schablone. Aus der Schablone lassen sich konkrete Aufgaben generieren. Den Prozess der Generierung konkreter Aufgaben nennt man auch *Materialisierung* [4]. Im vorliegenden Beitrag wird ein Datenformat[2] für die Spezifikation des Materialisierungsprozesses von Aufgabenschablonen vorgeschlagen. Dieses Format nutzt das ProFormA-Aufgabenformat [8] sowie ein zuvor entwickeltes Format zur Beschreibung von Variationspunkten [2].

[2] Das Datenformat in Gestalt eines XML-Schemas ist unter https://github.com/ProFormA/varproformaxml verfügbar.

2 Grundlegendes

2.1 Das ProFormA-Aufgabenformat

Das ProFormA-Aufgabenformat[3] beschreibt eine automatisiert bewertbare Programmieraufgabe (*task*) als Ansammlung verschiedener Artefakte, von denen einige in Abbildung 1 dargestellt sind. Neben dem Text der Aufgabe (*description*) werden verschiedene, automatisch durchführbare *Tests* definiert, die u. a. durch verschiedene Dateien (*files*) konfiguriert werden. Musterlösungen (*model-solutions*) können ebenfalls als Teil der automatischen Testdurchführung angegeben werden. Ein Bewertungsschema (*grading-hints*) beschreibt, wie aus Testergebnissen ein Bewertungsergebnis zusammengestellt wird [3].

Die genannten Dateien werden entweder direkt in eine XML-Datei eingebettet oder es wird ein Zip-Archiv erstellt, welches die `task.xml`-Datei und weitere Dateien enthält.

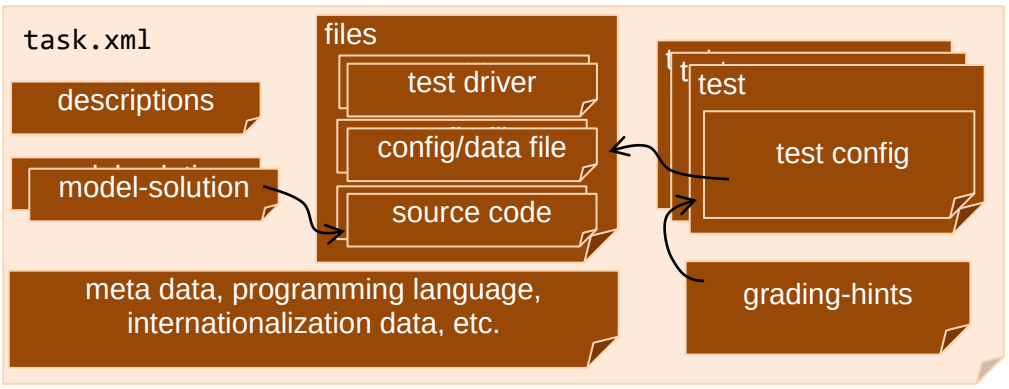

Abbildung 1: Elemente einer ProFormA-Aufgabe

2.2 Systeme

Typischerweise kommunizieren Lehrkräfte und Studierende über ein Lernmanagementsystem (*LMS*) mit dem Autobewerter (*Grader*). Die Lehrkraft legt eine Aufgabe an (*task*). Studierende reichen Lösungsversuche ein (*submission*) und erhalten Feedback (*response*). Meist vermittelt eine spezielle *Middleware* zwischen LMS und Grader (vgl. Abbildung 2).

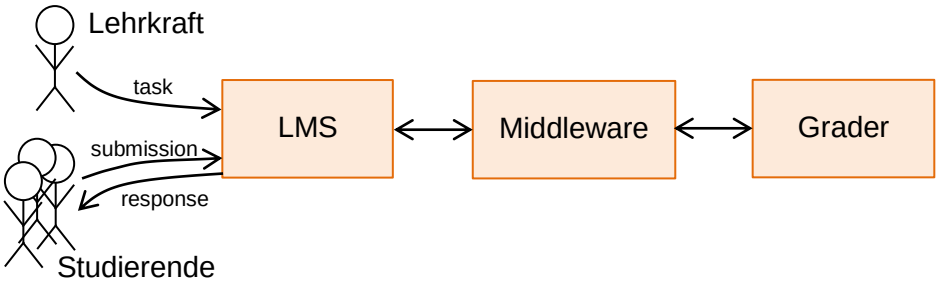

Abbildung 2: Systeme

2.3 Anforderungen

Um Interoperabilität der beteiligten Systeme für variable Aufgaben zu erreichen, ist ein allgemeines, von den Systemen unabhängig einsetzbares Format hilfreich. Mit dem ProFormA-

[3] https://github.com/ProFormA/proformaxml

Format existiert bereits ein solches Format für konkrete Aufgaben. Für variable Aufgaben soll der vorliegende Beitrag einen Lösungsvorschlag unterbreiten.

Das vorgeschlagene Format soll die gezielte sowie eine zufällige Auswahl einer Aufgabenvariante ermöglichen. Mit Hilfe der gewählten Werte der Variationspunkte soll aus einer Aufgabenschablone durch eine Materialisierung eine konkrete Aufgabe entstehen. Sowohl die Auswahl der Werte der Variationspunkte als auch die Materialisierung soll weitgehend von jedem der beteiligten Systeme erledigt werden können. Insb. soll es dem LMS möglich sein, eine konkrete Aufgabe ohne Beteiligung des Graders zu erzeugen. Wenn in Sonderfällen die Beteiligung des Graders notwendig ist, soll das Format für diese Fälle spezielle durchzuführende Materialisierungsschritte separat ausweisen, so dass LMS und Grader die Materialisierung kooperativ durchführen können.

Die in einer Aufgabenschablone zu implementierende Variabilität kann auf verschiedene Weise umgesetzt werden. Artefakte können bspw. mit Platzhaltern ausgestattet werden oder mehrfach in unterschiedlicher Ausprägung in der Schablone enthalten sein. Das zu entwickelnde Datenformat für Materialisierungen muss verschiedene Schablonentypen unterstützen.

Mit dem vorgeschlagenen Datenformat soll Interoperabilität für variable Programmieraufgaben entlang der Aufrufkette LMS – Middleware – Grader erreicht werden. Bspw. soll ein LMS, das das vorgeschlagene Format unterstützt, durch ein anderes LMS austauschbar sein, ohne dass die Nutzbarkeit der variablen Aufgabe wesentlich beeinträchtigt ist. Für Grader kann die Anforderung der Austauschbarkeit jedoch nur zurückhaltend aufgestellt werden, denn uns bekannte Grader habe durchweg einen sehr begrenzten Funktionsumfang für eine oder einige wenige Programmiersprachen. Das vorgeschlagene Format wird bspw. nicht geeignet sein, eine für die Programmiersprache SQL gestellte variable Aufgabe mit einem Java-Grader zu bewerten.

Üblicherweise werden variable Programmieraufgaben für eine bestimmte, von Studierenden zu nutzende Programmiersprache entworfen. Abweichend ist vorstellbar, dass eine Aufgabe die Programmiersprache selbst variabel vorsieht – bspw. wahlweise Java und C++. Wenn auch derzeit kein uns bekannter Grader in der Lage ist, derart variabel gestaltete Aufgaben automatisch zu bewerten, soll es aus Gründen der Zukunftsorientierung dennoch möglich sein, auch die Programmiersprache variabel zu halten. Es liegt dann in der Verantwortung des Aufgabenautors, ausschließlich sinnvolle[4] und vom Grader unterstützte Ausprägungen der Programmiersprache vorzusehen.

2.4 Vorgehen

Um dem beschriebenen Ziel näher zu kommen, wurden zuerst konkrete Aufgaben verschiedener Grader untersucht und Variationspunkte identifiziert. Auf der Basis der in Tabelle 1 dargestellten Beispiele wurden dann mehrere Aufgaben für den Grader Graja[5] variabel gestaltet. Auf dieser Grundlage wurde ein abstraktes Datenformat geschaffen, welches sowohl die in den Aufgaben vorhandene Variabilität als auch die zur Herstellung einer konkreten Aufgabe notwendigen Materialisierungsschritte beschreibt. Die Grader-Unabhängigkeit des entwickelten Datenformats wurde argumentativ gestützt. Es wurde eine von Graja unabhängige Java-Bibliothek geschaffen, die das Datenformat, die Auswahl von Werten für Variationspunkte und die Materialisierung implementiert. Der gewählte Forschungsansatz lässt sich am ehesten dem Design-Based Research zuordnen [7]. Einige der entstandenen variablen Graja-Aufgaben wurden im Wintersemester 2018/19 und im Sommersemester 2019 im formativen, semesterbegleitenden Assessment zweier aufeinander aufbauender Grundlagen-Lehrveranstaltungen zur Java-Programmierung an der Hochschule Hannover in dem Sinne eingesetzt, dass verschiedene der ca. 80 teilnehmenden Studierenden zufallsbasiert verschiedene Aufgabenvarianten erhielten. Der Einsatz wurde nicht bzgl. didaktischer Effekte evaluiert, sondern es ging hierbei i. w. darum, die technische Machbarkeit eines Einsatzes in dem Grader-unabhängigen LMS Moodle unter Beweis

[4] Vermutlich wäre eine Aufgabe zur Kreisflächenberechnung für Programmiersprachen wie Java oder C++ sinnvoll zu stellen, nicht jedoch für die Programmiersprache SQL.
[5] http://graja.hs-hannover.de

zu stellen. Eine Anwendung der bisherigen Ergebnisse auf Aufgaben weiterer Grader wurde noch nicht durchgeführt.

2.5 Verwandte Arbeiten

In der Produktlinienentwicklung (PLE) werden Variabilitätsmodelle genutzt, um Varianten und deren Constraints darzustellen. Beispielhaft seien die Common Variability Language (CVL, [4]) und das Orthogonal variability model [6] genannt. Die PLE hat mit variablen Programmieraufgaben gemeinsam, dass Variablen, Wertausprägungen und teilweise komplexe Randbedingungen formuliert werden müssen und dass die Variablen und ihre Werte mit den einzelnen Artefakten des Produkts in Verbindung gebracht werden müssen. Die Möglichkeiten solcher Ansätze gehen weit über die Erfordernisse einer variablen Programmieraufgabe hinaus. Das verwundert nicht, da in der PLE Modelle mit hunderten von Variablen keine Seltenheit sind [6], während wir in variablen Programmieraufgaben mit kaum mehr als einer kleinen zweistelligen Anzahl von Variablen rechnen. Außerdem kann in variablen Programmieraufgaben die Materialisierung einer Schablone (PLE-Begriffe resolution und materialization [4] bzw. binding und realization [6]) sehr spezifisch auf das spezielle ProFormA-Aufgabenformat zugeschnitten werden, wohingegen im allgemeinen PLE-Fall ein weites Spektrum von Artefakten (Anforderungen, Entwürfe, Implementierungen, Tests) berücksichtigt werden muss.

3 Datenmodell der Variabilität

Ein LMS- und Grader-übergreifend einsetzbares Datenmodell der in einer Aufgabenschablone angelegten Variabilität wird in [2] vorgestellt. Dieses spezifiziert den Wertebereich der Variationspunkte mit den Grundoperationen kartesisches Produkt, Vereinigungsmenge und Ableitungsfunktion. Dieses – zwischenzeitlich um ein Tabellenkonzept erweiterte und kleinen Umbenennungen unterworfene – Datenmodell soll in diesem Abschnitt erläutert werden.

3.1 Variationspunkt

Ein grundlegendes Element des Datenmodells ist der Variationspunkt (vgl. Abbildung 3):

- Ein Variationspunkt (*vp*) besitzt einen Namen (*key*) und einen Typ (*vpt*).
- Grundlegende Variationspunkt-Typen sind ganze Zahlen (*integer*), boolesche Werte (*boolean*), Zeichenketten (*string*), etc. Der Variationspunkt-Typ *table* bezeichnet einen Tabellentyp, deren Spalten rekursiv als Variationspunkte definiert sind. Der Variationspunkt-Typ *double* definiert zusätzlich eine Genauigkeit (*accuracy*) als kleinsten von 0 unterscheidbaren Wert.
- Eine Aufgabenschablone definiert in der Regel mehrere Variationspunkte, die sich in einem *composite variation point* (*cvp*) zusammenfassen lassen.

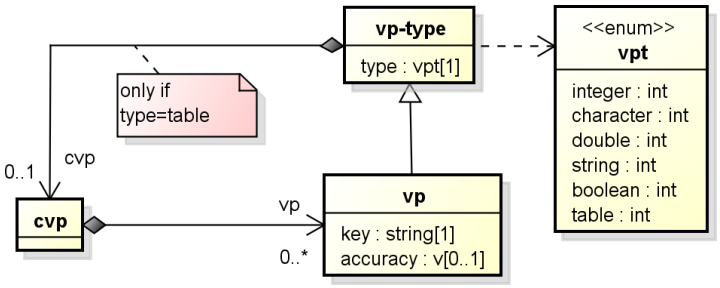

Abbildung 3: Datenmodell Variationspunkte

Die in Tabelle 1 genannten Bezeichner sind ein Beispiel eines Variationspunkts vom Typ string. Die Vorgabe, ob eine for- oder while-Schleife zu realisieren ist, könnte als boolean-Variationspunkt definiert werden. Die Berücksichtigung des Bewertungsaspekts Programmierstil kann durch einen boolean-Variationspunkt realisiert werden, der die Ausführung oder Nicht-Ausführung eines *test*s steuert. Alternativ können Bewertungsgewichte in den *grading-hints* als double-Variationspunkte formuliert werden. Die in Tabelle 1 genannten Datenwerte werden ebenfalls als Variationspunkte eines numerischen Typs realisiert. Wenn mehrere ungültige Kreisradien vorgegeben werden sollen, kann der Typ table genutzt werden, indem dieser rekursiv durch eine Spalte zur Aufnahme des jeweiligen ungültigen Datenwerts und durch weitere Spalten zur Beschreibung der vorgegebenen, vom studentischen Programm zu implementierenden Routinen definiert wird.

3.2 Variante

Variationspunkte können Werte annehmen (sog. *Varianten*). Eine Variante wird durch das Domänenobjekt *v* repräsentiert (vgl. Abbildung 4), das je Variationspunkt-Typ durch einen Subtyp spezialisiert wird. Bspw. beschreibt *vs* einen konkreten Bezeichner[6], den die vom Studenten programmierte Funktion besitzen soll. Die konkrete Ausprägung der Werte aller Variationspunkte einer Aufgabenschablone wird als *composite variant* (*cv*) modelliert. Die Ausprägung eines Variationspunktes vom Typ table ist konsequenterweise als Liste von cv-Objekten zu beschreiben (*cv-list*), wobei jedes Element der Liste einer Tabellenzeile entspricht.

[6] Dass der Wert (*value*) als optionales Attribut realisiert ist, hängt damit zusammen, dass Variationspunkte je nach Wahl anderer Variationspunktwerte obsolet werden können. Für Details s. [2].

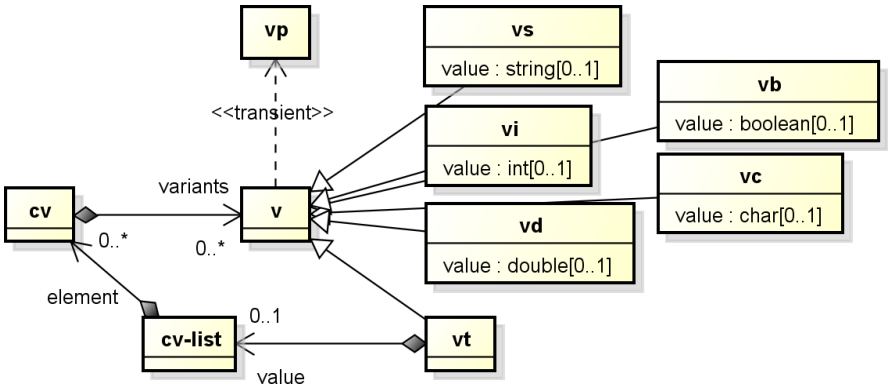

Abbildung 4: Datenmodell Varianten

3.3 Spezifikation aller gültigen Varianten aller Variationspunkte (var-spec)

Die Menge aller gültigen Varianten aller Variationspunkte einer Aufgabenschablone kann durch Kombinationseffekte so groß werden, dass es nicht vernünftig erscheint, alle Kombinationen aufzulisten. Das in [2] beschriebene Datenmodell spezifiziert die Menge aller gültigen Varianten (*var-spec*) stattdessen als Baum der Grundoperationen kartesisches Produkt, Vereinigungsmenge und Ableitungsfunktion (vgl. Abbildung 5).

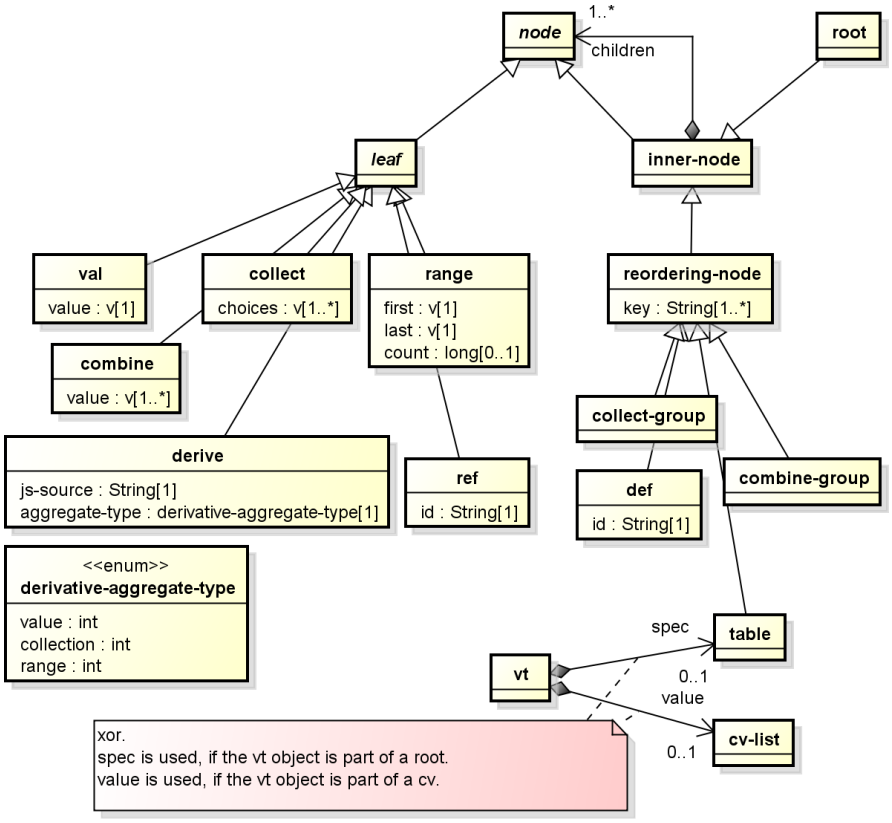

Abbildung 5: Datenmodell Variabilität

Die Blätter des Baumes spezifizieren in der Regel Wertebereiche einzelner Variationspunkte (*val, collect, range*). Um diese zu einer Gesamtspezifikation aller Variationspunkte zu kombinieren, werden in ihrer Werteauswahl unabhängige Variationspunkte in der Regel durch eine *combine-group* als kartesisches Produkt spezifiziert. Das *root*-Objekt definiert in diesem Fall als einziges Kindelement dieses kartesische Produkt. Abweichend von dem zuvor beschriebenen Standardfall können die gültigen Varianten umgekehrt als Vereinigungsmenge (*collect-group*) an der Wurzel spezifiziert werden. In diesem Fall kommen an den Blättern auch Tupel als Werte mehrere Variationspunkte vor (*combine*). Durch die baumartige Verschachtelung können beliebige

Kombinationen von Teilen des Raums aller gültigen Varianten spezifiziert werden, wobei in *reordering-node.key* die Namen der den jeweiligen Teilraum aufspannenden Variationspunkte benannt werden. An den Blättern werden konkrete Werte als *v*-Objekte spezifiziert. Im Falle von Variationspunkten einfacher Datentypen stehen hier konkrete Werte. Im Falle eines table-Variationspunkts steht hier ein *vt*-Objekt, welches mit *table* rekursiv eine Baumspezifikation enthält. Schließlich erlaubt das Datenmodell, die gültigen Varianten eines Variationspunktes als in Javascript implementierte Ableitungsfunktion (*derive*) anderer Variationspunkte zu spezifizieren. Ein Beispiel für die Verwendung dieser Möglichkeit sowie weitere Details zur Möglichkeit der Wiederverwendung von Teilbäumen (*def, ref*) sind in [2] beschrieben.

4 Materialisierung

Wir schlagen vor, eine Aufgabenschablone so weit wie möglich an die Struktur einer konkreten Aufgabe anzulehnen. D. h. eine Aufgabenschablone besteht zunächst aus den gleichen Artefakten wie eine konkrete Aufgabe (vgl. Abschnitt 2.1). Die Artefakte der Aufgabenschablone werden fallweise um Zusatzinformationen erweitert, die unter Rückgriff auf die Werte der Variationspunkte beschreiben, wie aus dem Schablonenartefakt ein Artefakt einer konkreten Aufgabe generiert werden kann. In [6] nennt man diese Zusatzinformationen *artefact dependency*.

Wir betrachten Schablonenartefakte als Gegenstände der *Materialisierung*. Eine *Materialisierung* ist die Anwendung einer Materialisierungs-*Methode* auf ein zu materialisierendes *Artefakt*. Nach Durchführung aller Materialisierungen entsteht aus einer Schablone eine konkrete Aufgabe. Die Zusatzinformation im Sinne des vorherigen Absatzes lässt sich schreiben als eine Menge { m : m=(ma, mm) } von Materialisierungen m, wobei jede Materialisierung m ein Paar eines Materialisierungsartefakts ma und einer Materialisierungsmethode mm ist. Diese grundlegende Idee wurde in [1] erstmals entwickelt und anschließend verfeinert.

4.1 Materialisierung von Schablonenartefakten

In diesem Abschnitt besprechen wir an einer einfachen Java-Programmieraufgabe verschiedene von Materialisierungen betroffene Artefakte.

4.1.1 Platzhalter in task.xml und weiteren Dateien

Im einfachsten Fall enthält das Schablonenartefakt Platzhalter für Variationspunkte, die durch konkrete Varianten ersetzt werden. Die folgende Tabelle zeigt ein Beispiel eines Aufgabentextes mit einem Platzhalter für den Variationspunkt namens *shape*. Die konkreten Texte wurden durch Einsatz des mustache[7]-Frameworks generiert:

Schablone	*Given an interface `Shape` with an `area` method, write a subclass `§(shape)§` accepting `Double[]` as a constructor parameter.*
Konkrete Aufgabe	*Given an interface `Shape` with an `area` method, write a subclass `Circle` accepting `Double[]` as a constructor parameter.*
	Given an interface `Shape` with an `area` method, write a subclass `Rectangle` accepting `Double[]` as a constructor parameter.

Das vorstehende Beispiel wendet die Materialisierungsmethode mm="mustache mit Prefix §(und Suffix)§" auf das Materialisierungsartefakt ma="task.xml" an, da der Aufgabentext Teil der task.xml-Datei ist.

Nicht nur Texte, sondern auch Quelltexte von z. B. Testtreibern können durch Platzhalter variabel gestaltet werden, wie die folgende Tabelle beispielhaft vorführt. Der Quelltext `Testdriver.java` der Schablone ist somit nicht mehr syntaktisch korrekt. Er wird es erst wieder durch die Materialisierung:

Schablone	`Testdriver.java`
	`Double[] data= { §(data)§ };`
	`Shape submission= new §(shape)§(data);`

Konkrete Aufgabe	`Double[] data= { 5.0 };` `Shape submission= new Circle(data);`
	`Double[] data= { 5.0, 3.0 };` `Shape submission= new Rectangle(data);`

Das vorstehende Beispiel ist durch mm="mustache mit Prefix §(und Suffix)§" und ma="`Testdriver.java`" beschreibbar.

[7] https://mustache.github.io/

In komplexeren Fällen wird ein Variationspunkt durch ein separates Artefakt repräsentiert (bspw. eine Schnittstellendeklaration), welches durch je ein weiteres Artefakt je Variante realisiert wird (bspw. eine konkrete Implementierung der Schnittstelle). Im folgenden Beispiel wird der Variationspunkt *shape* durch die Variable `expected` vom Typ `DoubleSupplier` repräsentiert. Die konkreten Implementierungen folgen direkt im Anschluss innerhalb einer Fallunterscheidung, die auf den Wert des Variationspunktes *shape* abstellt.

Schablone	

	`DoubleSupplier expected;`
	`String s= "§(shape)§";`
	`switch (s) {`
	`  case "Circle":  expected= () -> Math.PI * data[0]*data[0]; break;`
	`  case "Rectangle": expected= () -> data[0]*data[1]; break;`
	`}`

4.1.2 Existenz von Artefakten

Das vorstehende Beispiel kann noch innerhalb einer Quelltextdatei realisiert werden. Es wird jedoch deutlich, dass Konkretisierungen genauso gut in je einer zusätzlichen Quelltextdatei stattfinden können. Das folgende Beispiel zeigt zwei konkrete Musterlösungen:

Schablone	`Shape.java:`
	`interface Shape {`
	`  double area();`
	`}`

Konkrete Aufgabe	`Circle.java:`
	`class Circle implements Shape {`
	`  double r;`
	`  Circle(Double[] d) { r= d[0]; }`
	`  double area() { return Math.PI * r*r; }`
	`}`

	`Rectangle.java:`
	`class Rectangle implements Shape {`
	`  double w, h;`
	`  Rectangle(Double[] d) { w= d[0]; h= d[1]; }`
	`  double area() { return w*h; }`
	`}`

In Fällen wie diesem wollen wir annehmen, dass die Aufgabenschablone alle Konkretisierungen des Artefakts als Dateien enthält. Zur Erzeugung einer konkreten Aufgabe muss dann aus diesen Dateien die eine passende ausgewählt werden bzw. es muss die unpassende Datei entfernt werden. Dazu führen wir zusätzliche Variationspunkte ein, die vom Variationspunkt *shape* abgeleitet werden (derive):

- *is_circle* = true, falls *shape*="Circle", sonst false
- *is_rectangle* = true, falls *shape*="Rectangle", sonst false

Hiermit charakterisieren wir die Materialisierung durch (ma="Existenz der Datei Circle.java", mm="Setze Wert von *is_circle*") und (ma="Existenz der Datei Rectangle.java", mm="Setze Wert von *is_rectangle*"). D. h., wenn *shape*≠Circle ist, dann ist *is_circle*=false und dann wird die Existenz der Datei `Circle.java` verneint, d. h. `Circle.java` aus der Aufgabenschablone entfernt. Dieses Beispiel zeigt, dass wir als Artefakte nicht nur Dateiinhalte, sondern auch Existenzen von Elementen einer Aufgabenschablone vorsehen.

4.1.3 Dateinamen

Wollen wir Studierenden eine Ausfüllvorlage zur Verfügung stellen, muss diese je nach Konkretisierung einen anderen Dateinamen besitzen. Statt in der Aufgabenschablone mehrere Ausfüllvorlagen vorzuhalten, kann es bequemer sein, genau eine Datei `__shape__.java` in der Aufgabenschablone abzulegen. In der konkreten Aufgabe soll diese Datei umbenannt werden und darüber hinaus sollen Platzhalter im Inhalt ersetzt werden:

Schablone	__shape__.java:
	`class §(shape)§ implements Shape {`
	`  // TODO: complete this template`
	`  double area() { return 0.0; }`
	`}`

Konkrete Aufgabe	Circle.java:
	`class Circle implements Shape {`
	`  // TODO: complete this template`
	`  double area() { return 0.0; }`
	`}`

	Rectangle.java:
	`class Rectangle implements Shape {`
	`  // TODO: complete this template`
	`  double area() { return 0.0; }`
	`}`

Das vorstehende Beispiel beschreibt die zweischrittige Materialisierung { (ma="Inhalt von `__shape__.java`", mm="mustache mit Prefix §(und Suffix)§"), (ma="Dateiname von `__shape__java`", mm="mustache mit Prefix __ und Suffix __") }.

4.1.4 Inhaltsabschnitte in task.xml und weiteren Dateien

Wir wollen einen in der XML-Datei definierten *test* nur für einige Aufgabenvarianten ausführen. Hier bietet es sich an, die entsprechende Passage der task.xml-Datei mit Platzhaltern eines Template-Frameworks zu markieren. So können Aufgabenvarianten generiert werden, in denen bestimmte Aspekte einer Einreichung nur dann geprüft werden, wenn der dies steuernde boolean-Variationspunkt den Wert true annimmt. Beispielhaft zeigen wir einen Ausschnitt einer task.xml-Datei, die eine sog. mustache *section* einsetzt:

Schablone	`§(#do_checkstyle)§`
	`<p:test id="checkstyle">`
	`  <p:title>…</p:title>`
	`  <p:test-type>java-checkstyle</p:test-type>`
	`  <p:test-configuration>…</p:test-configuration>`
	`</p:test>`
	`§(/do_checkstyle)§`

Konkrete Aufgabe	Falls `do_checkstyle`=true, dann ist in der konkreten task.xml das `test`-Element enthalten. Sonst nicht.

Mustache sections können darüber hinaus genutzt werden, um table-Variationspunkte schleifenartig zu durchlaufen und für jede Tabellenzeile einen bestimmten Inhaltsabschnitt zu generieren.

4.1.5 Arithmetische Manipulation

Um das Beispiel abzuwandeln, soll der Checkstyle-Test grundsätzlich in allen Aufgabenvarianten ausgeführt werden. Die Gewichtung des Testergebnisses soll jedoch variieren. Hierzu kann man bspw. einen Variationspunkt *offset* einführen, dessen Wert auf die Bewertungsgewichte addiert bzw. von diesen subtrahiert wird: { (ma="Gewicht des Tests ‚junit'", mm="addiere den Wert *offset*"), (ma="Gewicht des Tests ‚checkstyle'", mm="subtrahiere den Wert *offset*") }. Die Schablone enthält hier keine Platzhalter, sondern einfach eine gültige Bewertungskonfiguration. Durch Anwendung der beiden vorgenannten Materialisierungen entsteht eine neue Bewertungskonfiguration mit entsprechend erhöhten bzw. verringerten Gewichten:

Schablone	`<p:grading-hints><p:root function="sum">`
	`  <p:test-ref ref="junit" weight="0.7"/>`
	`  <p:test-ref ref="checkstyle" weight="0.3"/>`
	`</p:root></p:grading-hints>`

Konkrete Aufgabe für *offset*=0.05	`<p:grading-hints><p:root function="sum">`
	`  <p:test-ref ref="junit" weight="0.75"/>`
	`  <p:test-ref ref="checkstyle" weight="0.25"/>`
	`</p:root></p:grading-hints>`

Konkrete Aufgabe für *offset*=0.3	`<p:grading-hints><p:root function="sum">`
	`  <p:test-ref ref="junit" weight="1.0"/>`
	`  <p:test-ref ref="checkstyle" weight="0.0"/>`
	`</p:root></p:grading-hints>`

4.2 Aufgabenschablone

In einigen der vorstehenden Beispiele besitzt die Schablone eine task.xml-Datei, die nicht valide ist. Daher kann die task.xml-Datei in Aufgabenschablonen nicht genutzt werden, um Informationen zu den Variationspunkten, Wertebereichen und Materialisierungen zu speichern. Diese Information muss separat abgelegt werden. Eine Aufgabenschablone besitzt daher die folgende Struktur:

- Zip-Archiv
- Darin die Datei `task.xml` einer normalen ProFormA-Aufgabe zzgl. etwaiger weiterer, von task.xml referenzierter Artefakte
- Und (neu!) eine Datei `tpl.xml`.

Die Datei `tpl.xml` (*tpl* ist eine Abkürzung für *template*) enthält die in Abbildung 6 dargestellten Daten: *var-spec* spezifiziert die Variationspunkte und die Menge aller gültigen Wertkombinationen (vgl. Abschnitt 2.3), *default-value* spezifiziert eine gültige Wertkombination als Standardbelegung, *mat-spec* spezifiziert die Materialisierungen, d. h. wie aus einer gültigen Wertkombination eine konkrete Aufgabenvariante generiert werden kann. Im verbleibenden Teil dieses Abschnitts soll das Objekt *mat-spec*, welches die Materialisierung der Aufgabenschablone spezifiziert, genauer beschrieben werden.

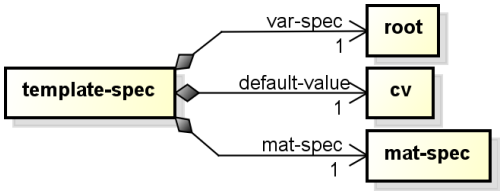

Abbildung 6: Inhalt von tpl.xml

4.3 Datenmodell der Materialisierung

Wie oben erwähnt ist eine Materialisierung die Anwendung einer Materialisierungsmethode auf ein zu materialisierendes Artefakt. Die gedankliche Trennung von Methode und Artefakt hat den Vorteil, dass häufig anzuwendende Methoden wie das Template-Framework mustache nur

einmal spezifiziert werden müssen, aber dennoch auf viele verschiedene Artefakte angewendet werden können. Abbildung 7 zeigt, dass eine *materialization* tatsächlich als Verknüpfung mehrerer *artifact-ids* und mehrerer *method-ids* modelliert wurde, um die gleichförmige Anwendung einer Menge mehrerer Methoden auf eine Menge mehrerer Artefakte möglichst redundanzarm beschreiben zu können.

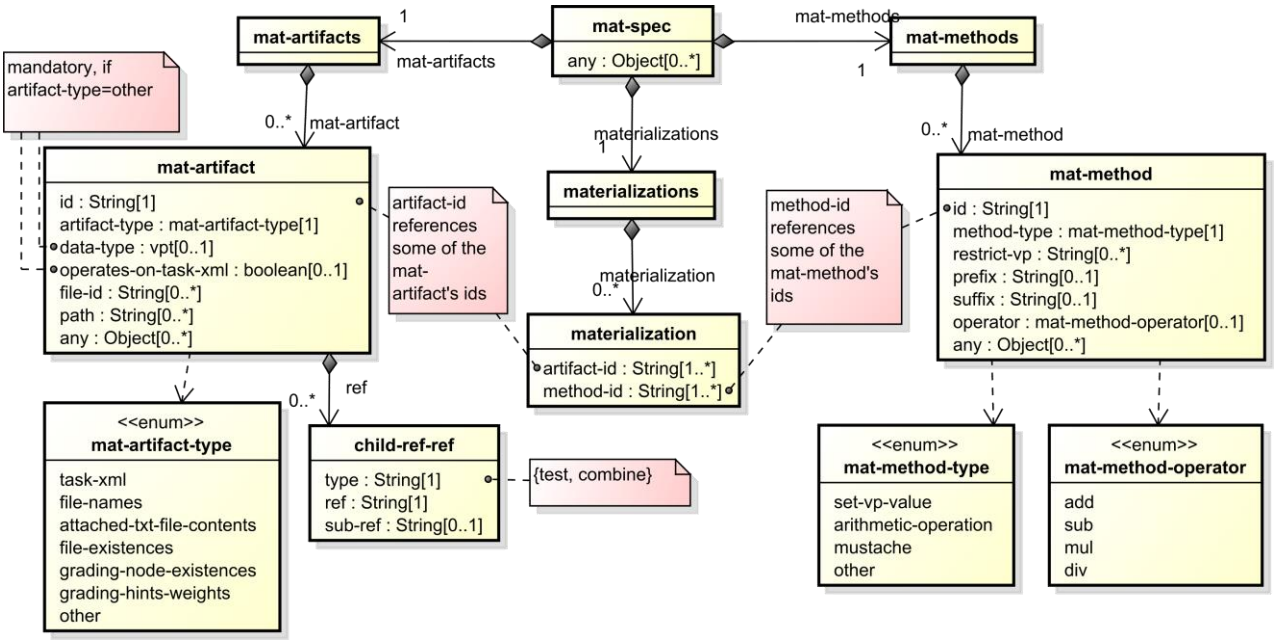

Abbildung 7: Datenmodell Materialisierung

Als Artefakte-Typen (mat-artifact-type) haben wir die aus dem obigen Beispiel gefundenen Szenarien abgebildet. Ein sicherlich in fast jeder Aufgabenschablone verwendetes Artefakt ist:

```
<v:mat-artifact id="a1" artifact-type="task-xml"/>
```

welches den Inhalt der Datei task.xml bezeichnet. Einige der Artefakttypen müssen mit zusätzlichen Parametern spezifiziert werden. Bspw. spezifiziert

```
<v:mat-artifact id="a2" artifact-type="file-existences">
  <v:path>samplesolutions/Circle.java</v:path>
</v:mat-artifact>
```

die Existenz der Musterlösung `Circle.java`. Und das folgende Fragment spezifiziert das Bewertungsgewicht des JUnit-Tests:

```
<v:mat-artifact id="a3" artifact-type="grading-hints-weights">
  <v:ref ref="junit" type="test"/>
</v:mat-artifact>
```

Auch die Methoden wurden auf der Grundlage der obigen Szenarien entwickelt. Eine Methode operiert auf einem Artefakt. Die Methode nutzt sowohl die konkrete Wertebelegung der Variationspunkte als auch das Artefakt als Eingabe und produziert als Ausgabe ein ggf. verändertes Artefakt.

Die vom mustache-Framework realisierte Suchen-und-Ersetzen-Operation wird durch

```
<v:mat-method id="m1" method-type="mustache" prefix="§(" suffix=")§"/>
```

spezifiziert. Wir haben uns dafür entschieden, das mustache-Framework als Standard-Methode aufzunehmen, weil dieses in einer sehr großen Anzahl von Programmiersprachen und Plattformen verfügbar ist, so dass jedes LMS, jede Middleware und jeder Grader in der Lage sein sollte, diese Materialisierungsmethode auszuführen.

Weitere Methoden müssen durch Angabe eines Variationspunktes genauer spezifiziert werden. Die folgende Methode addiert auf den durch ein Artefakt spezifizierten numerischen Wert den

Wert des Variationspunkts *offset* auf und liefert das Ergebnis als neues Artefakt. Diese Methode kann gut mit dem oben definierten Artefakt mit der id=a3 zu einer Materialisierung verknüpft werden:

```
<v:mat-method id="m2" method-type="arithmetic-operation" operator="add">
  <v:restrict-vp>offset</v:restrict-vp>
</v:mat-method>
```

Als weiteres Beispiel zeigen wir die folgende Methode, die einfach den Wert des Variationspunkts *is_circle* ausliest und als neues Artefakt liefert. Durch Verknüpfung dieser Methode mit dem Artefakt a2 deklariert man die Entfernung unbenötigter Musterlösungen als Materialisierung:

```
<v:mat-method id="m3" method-type="set-vp-value">
  <v:restrict-vp>is_circle</v:restrict-vp>
</v:mat-method>
```

4.4 Grader-spezifische Erweiterungen

Die im vorstehenden Abschnitt spezifizierten Materialisierungsmethoden und -artefakte decken bereits eine Vielzahl von typischen Szenarien ab. Manche Grader oder Programmiersprachen mögen jedoch zusätzliche Materialisierungsmechanismen benötigen.

Ein typisches Beispiel ist die Materialisierung von Dateien, die in einem proprietären Binärformat in der Aufgabenschablone vorliegen und die für verschiedene Aufgabenvarianten jeweils durch einen Grader-spezifischen Prozess, bspw. eine Compilierung, neu generiert werden müssen. Um solche Grader-spezifischen Materialisierungsmethoden abzubilden, wird *mat-method-type.other* genutzt. In *mat-method.any* können Grader-spezifische Zusatzinformationen deklariert werden (bspw. Compiler-Flags), die den Grader-spezifischen Materialisierungsprozess steuern.

Auch auf der Seite der Artefakte ist es möglich, mit *mat-artifact-type.other* ein Grader-spezifisches Artefakt zu deklarieren. Denkbar wären hier Angaben, die einen ganz bestimmten Abschnitt einer in der Aufgabenschablone enthaltenen Quelltextdatei spezifizieren, wobei spezielle, von der Programmiersprache abhängige, in *mat-artifact.any* transportierte Vokabeln zur Spezifikation des Abschnitts genutzt werden. Illustrierend kann etwa eine ganz bestimmte Java-Methode innerhalb der Quelltextdatei unter Angabe ihres Namens und ihrer Parametertypen als Artefakt deklariert werden.

Alle Methoden- und Artefakttypen wurden in der in Abschnitt 2.3 genannten Bibliothek realisiert. Durch die Trennung von Methode und Artefakt war es möglich, Grader-spezifische Methoden so zu kapseln, dass sie an bereits implementierte Standard-Artefakte softwaretechnisch „angedockt“ werden konnten. Und umgekehrt können Implementierungen Grader-spezifischer Artefakte so gekapselt werden, dass sie an bereits implementierte Standard-Methoden „angedockt“ werden können.

Da wir letztlich jedoch annehmen müssen, dass es in einzelnen Gradern keine Möglichkeit gibt, die bereits implementierten Standardmethoden und –artefakte durch Aufruf zu verwenden, sehen wir schließlich eine zusätzliche Möglichkeit vor, in *mat-spec.any* beliebige Daten für eine gänzlich Grader-spezifische Materialisierung abzulegen.

4.5 Weitere Methoden und Artefakte

Die in Abbildung 7 dargestellten Materialisierungsmethodentypen und -artefakttypen stellen eine grundlegende Menge dar, mit der alle bisher betrachteten Aufgaben variabel gestaltet werden konnten. Im Zuge der variablen Gestaltung weiterer Aufgaben ist es wahrscheinlich, dass die Liste dieser Typen fallbezogen erweitert wird, wenn neue Methoden und Artefakte entdeckt werden, die potentiell in verschiedenen Programmiersprachen und Gradern nutzbringend eingesetzt werden können.

Um ein Beispiel zu nennen: die Entfernung von unbenötigten Musterlösungen wird derzeit recht aufwändig durch viele zusätzliche boolesche Variationspunkte realisiert, die dann im Rahmen der

Materialisierungsmethode *set-vp-value* auf Materialisierungsartefakte vom Typ *file-existences* angewendet werden. Um die zusätzlichen Variationspunkte einzusparen, bietet es sich an, stattdessen einen speziellen Materialisierungsartefakttyp *model-solution-existences* zu erfinden, der mit den Pfaden oder Ids aller Musterlösungen parametriert wird, und auf diesen Artefakttyp eine Methode eines neu zu erfindenden Materialisierungsmethodentyps *javascript* anzuwenden, der in einer Javascript-Funktion aus den gegebenen Pfaden oder Ids die zu den Variationspunktwerten passenden Pfade oder Ids selektiert und als neues Artefakt liefert.

Spezielle Methoden und Artefakte, die nur einzelne Systeme nutzen, sollen hingegen wie in Abschnitt 4.4 abgebildet werden. Dazu zählen wir ausdrücklich den Einsatz anderer, programmiersprachenabhängiger, Template-Frameworks, die mächtigere Mechanismen als das weit verbreitete Framework mustache bieten.

4.6 Materialisierungen zur Laufzeit des Graders

Bei der Realisierung verschiedener Graja-Aufgaben hat es sich als nützlich erwiesen, in einer generierten Aufgabenvariante nicht nur die materialisierten Artefakte abzulegen, sondern zusätzlich die selektierten Werte der Variationspunkte. Diese Werte können optional für späte, erst zur Laufzeit eines Autobewerters durchgeführte, Grader-spezifische Materialisierungsschritte genutzt werden. Das ProFormA-Element *meta-data* der task.xml bietet sich hier als Container für diese Werte an, die dort als *cvvp*-Element abgelegt werden (vgl. Abbildung 8).

Abbildung 8: Datenmodell der selektierten Werte

Ein Grader, der diese Daten aus einer konkreten Aufgabenvariante ausliest und einem Testtreiber zur Verfügung stellt, ermöglicht Aufgabenschablonen wie diese:

Schablone	`DoubleSupplier expected;`
	`switch (cvvp.getString("shape")) {`
	`  case "Circle":    expected= () -> Math.PI * data[0]*data[0]; break;`
	`  case "Rectangle": expected= () -> data[0]*data[1]; break;`
	`}`

Der Vorteil gegenüber der am Ende von Abschnitt 4.1.1 gelisteten Schablone ist, dass der vorverarbeitende Materialisierungsschritt entfällt. So lässt sich der Testtreiber während der Entwicklung der Aufgabe leichter in einer integrierten Entwicklungsumgebung (IDE) testen, da die IDE in der Regel keinen vorverarbeitenden Materialisierungsschritt unterstützt.

5 Zusammenfassung und Ausblick

Der vorliegende Beitrag beschreibt ein Datenmodell zur Beschreibung der Variabilität automatisiert bewertbarer Programmieraufgaben. Das Datenmodell enthält keine Festlegungen auf bestimmte Grader oder Programmiersprachen und hat somit das Potential, das bereits existierende Grader-unabhängig einsetzbare ProFormA-Format um den Aspekt der Variabilität zu erweitern. Die Variationspunkte einer variablen Aufgabe werden mit ihren Wertebereichen durch kaskadierend verschachtelte Operationen (Vereinigung, kartesisches Produkt, Ableitungsfunktion) spezifiziert. Die Auswirkungen von für die Variationspunkte ausgewählten Werten auf die verschiedenen Artefakte einer Programmieraufgabe (Aufgabentext, Dateien, Bewertungsschema, etc.) werden durch Paare von Materialisierungsartefakten und hierauf anzuwendenden Materialisierungsmethoden spezifiziert. Das Datenformat ist offen für Grader-spezifische Erweiterungen und liegt als XML-Schemadatei[2] vor.

Zukünftige Arbeiten werden sich mit der Abbildung des Formats auf weitere Programmieraufgaben und Grader sowie mit einer geeigneten Übertragung des Internationalisierungskonzepts von konkreten ProFormA-Aufgaben auf variable Aufgaben befassen. Eine weitere interessante Forschungsfrage ist die Standardisierung der Skala von Schwierigkeiten der aus einer Aufgabenschablone generierten konkreten Varianten. Schließlich wollen wir die derzeitige Einbindung in Moodle hinsichtlich der Unterstützung der Lehrperson verbessern, um den Einsatz variabler Aufgaben praktikabel zu gestalten.

6 Quellen

[1] Drangmeister, R. (2018): Entwurf und Implementierung eines Instanziierungsservices für variable Programmieraufgaben. Bachelorarbeit, Hochschule Hannover, urn:nbn:de:bsz:960-opus4-12207.

[2] Garmann, R. (2018): Ein Schnittstellen-Datenmodell der Variabilität in automatisch bewerteten Programmieraufgaben. Proceedings „SEELS – Software Engineering für E-Learning-Systeme“, Software Engineering Workshops 2018, CEUR, urn:nbn:de:0074-2066-4.

[3] Garmann, R. (2019): Ein Format für Bewertungsvorschriften in automatisiert bewertbaren Programmieraufgaben. In: Pinkwart, N., Konert, J.: „Die 17. Fachtagung Bildungstechnologien (DeLFI)“.

[4] Haugen, Ø. (2012), Common Variability Language (CVL) – OMG Revised Submission. OMG document ad/2012-08-05.

[5] Otto, B., Goedicke, M. (2017): Auf dem Weg zu variablen Programmieraufgaben: Requirements Engineering anhand didaktischer Aspekte. In: Proc. of the ABP 2017. CEUR, urn:nbn:de:0074-2015-4.

[6] Pohl, K., Böckle, G., van der Linden, F. (2005), Software Product Line Engineering: Foundations, Principles, and Techniques, Springer.

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

[8] 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).