E-Assessment mit Graja – ein Vergleich zu Anforderungen an Softwaretestwerkzeuge

Robert Garmann[1]

Abstract: Automatisiert bewertbare Programmieraufgaben zu erstellen ist so aufwändig, dass man die Aufgaben häufig wiederverwenden und idealerweise auch zwischen verschiedenen Autobewertern (Gradern) portieren will. Da die automatisierte Bewertung studentischer Einreichungen Parallelen zum automatisierten Test in der professionellen Softwareentwicklung aufweist, setzen Autobewerter häufig de-facto-Standardwerkzeuge des Softwaretests wie JUnit ein. Dennoch sind viele Programmieraufgaben heute schlecht oder gar nicht auf andere Autobewerter portierbar. In diesem Beitrag zeigen wir zunächst Parallelen und grundsätzliche Unterschiede zwischen e-Assessment und Softwaretest auf. Wir identifizieren anschließend beispielhaft einige Anforderungen an Autobewerter, die nicht an Softwaretestwerkzeuge gestellt werden. Am Beispiel des an der Hochschule Hannover entwickelten und seit mehreren Jahren im Einsatz befindlichen Autobewerters „Graja“ zeigen wir, dass diese Anforderungen aufgrund fehlender, Autobewerter-übergreifender Standards zu Abhängigkeiten zwischen Aufgabe und Autobewerter führen. Eine Schlussfolgerung ist, dass Standard-Schnittstellen für diejenigen Anforderungen, die das e-Assessment vom Softwaretest unterscheiden, erst noch entwickelt werden müssen.

Keywords: e-Assessment, computer based assessment, CBA, Grader, Bewertung, Java, Programmierung, Programmieraufgabe, Programmieranfänger, Feedback, formatives Assessment, Aufgabenkonfiguration, Anpassung und Wartung von Programmieraufgaben, Testautomation, Autobewerter

1 Einleitung

1.1 Parallelen zwischen dem formativen Assessment von Programmieraufgaben und dem Software-Test

Formatives Assessment in Programmieren-Lehrveranstaltungen wird häufig durch studentische Hilfskräfte durchgeführt, welche die von Lernenden eingereichten Programme bewerten und kommentieren. Dabei fallen hohe menschliche Aufwände für den „Test“ des eingereichten Programms auf Übersetzbarkeit, funktionale Korrektheit und Qualität (Wartbarkeit, Effizienz, etc.) an. Einem hohen Lernerfolg steht entgegen, dass menschliches Feedback verzögert und in unterschiedlicher Güte zu den Lernenden gelangt.

Tests werden in der professionellen Softwareentwicklung üblicherweise automatisiert, um Tests mit geringen Kosten wiederholen zu können. Im e-Assessment ist Testautoma-

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

tisierung sinnvoll, weil viele gleichartige studentische Einreichungen getestet werden müssen. In der professionellen Softwareentwicklung kommen neben den auch dort genutzten menschlichen Reviews verschiedene Testmethoden und -werkzeuge zum Einsatz [BM11], wie etwa xUnit-Testframeworks [Me07], aber auch Werkzeuge der statischen Analyse (z. B. PMD[2], Checkstyle[3]) und der dynamischen Analyse (z. B. VisualVM[4]). Die Werkzeuge der beiden letztgenannten Werkzeugklassen befassen sich hauptsächlich mit nicht-funktionalen Qualitätseigenschaften. xUnit-Testframeworks können sowohl für Funktionstests als auch für den Test einiger Qualitätseigenschaften eingesetzt werden. Viele Werkzeuge (z. B. JUnit, PMD, Checkstyle) sind als de-facto-Standards weit verbreitet im Einsatz.

Seit mehreren Jahren wurden und werden Autobewerter für verschiedene Programmiersprachen für verschiedene Einsatzszenarien entwickelt (s. z. B. [SSM+15] und [IAK+10] für Überblicksartikel jüngeren Datums). Programmieraufgaben können mit Hilfe eines Autobewerters so formuliert werden, dass studentische Einreichungen ganz oder teilweise automatisch bewertet und mit Feedback versehen werden. Dazu setzen viele Autobewerter eines oder mehrere der oben als de-facto-Standards erkannten Softwaretestwerkzeuge ein. In diesem Beitrag betrachten wir den Autobewerter „Graja“ als Stellvertreter solcher Autobewerter, die Testmethoden und -werkzeuge aus der professionellen Softwareentwicklung einsetzen[5].

1.2 Hindernisse bei der Wiederverwendung von Programmieraufgaben

Die Erstellung automatisiert bewertbarer Aufgaben ist so aufwändig, dass der Wiederverwendung in vielen Lehrszenarien durch möglichst viele Lehrkräfte eine hohe Bedeutung zukommt. Die Vielzahl verschiedener, im Einsatz befindlicher Autobewerter fordert zudem eine möglichst einfache Portierung einer Programmieraufgabe von einem zum anderen Autobewerter. Die Portierbarkeit scheint auf den ersten Blick gegeben, da Autobewerter ja auf weit verbreiteten, de facto standardisierten Softwaretestwerkzeugen aufsetzen. Jedoch führen spezielle, einen lernförderlichen Einsatz des Autobewerters anstrebende Anforderungen, von denen wir in diesem Beitrag einige nennen und die nicht an professionelle Testautomatisierungswerkzeuge gestellt werden, in der Regel zu Abhängigkeiten, die die Portierung einer Programmieraufgabe von einem Autobewerter zu einem anderen erheblich erschweren.

1.3 Überblick über diesen Beitrag

Kapitel 2 beleuchtet die Stakeholdergruppen, die einerseits beim professionellen

[2] http://pmd.sourceforge.net/
[3] http://checkstyle.sourceforge.net/
[4] https://visualvm.java.net/
[5] Andere Autobewerteransätze, die spezialisierte, nicht im Softwaretest eingesetzte Verfahren umsetzen (z. B. den graphbasierten Vergleich der Einreichung mit Musterlösungen), werden hier nicht weiter betrachtet.

Softwaretest und andererseits bei der Autobewertung beteiligt sind. In Kapitel 3 formulieren wir auf der Basis der Ziele der einzelnen Stakeholdergruppen ausgewählte Anforderungen an den Autobewerter, die nicht oder nicht in gleicher Weise an Softwaretestwerkzeuge gestellt werden. Wir erläutern eine mögliche Umsetzung der Anforderungen, wie sie im Autobewerter „Graja“ realisiert wurde, den wir gleich im Anschluss in Abschnitt 1.4 einführen. Den Abschluss des Beitrages bildet eine Zusammenfassung in Kapitel 4.

1.4 Graja – Grader for java programs

Graja bewertet studentische Java-Programme unter Einsatz von JUnit 4 sowie einer mitgelieferten Klassenbibliothek. Eine für Graja einsetzbare Programmieraufgabe besteht aus einem JUnit-Testtreiber, der durch seine Abhängigkeit von Graja-Klassen und -funktionen nur in Graja und nicht in einer klassischen JUnit-Laufzeitumgebung gestartet werden kann. Graja erwartet als studentische Einreichung eine Zip-Datei, entpackt die Datei, übersetzt Quelltexte, lädt den resultierenden Bytecode und führt schließlich den Testtreiber aus. Als Rückgabe sieht Graja u. a. die erreichten Punkte und Hinweistexte vor. Graja ist konzipiert für die Beobachtung des studentischen Programmverhaltens an den Schnittstellen Console, Datei-Ein-/Ausgabe sowie an internen Programmschnittstellen. Derzeit nicht Gegenstand ist die Nutzung von Mauseingabe und GUI-Ausgabe[6]. Eine prototypische Erweiterung von Graja erweitert das Feedback um eine Bewertung des Programmierstils unter Einsatz von PMD[2].

Graja entstand aufgrund von Vorerfahrungen mit dem System Web-CAT[7], welches auf testgetriebene Entwicklung setzt. Graja nutzt und erweitert die „student“-Bibliothek von Web-CAT, die den Dozenten bei der Erstellung von Testtreibern unterstützt. Maßgeblichen Einfluss auf die Neuentwicklung von Graja hatte der Wunsch, den Grader in andere Lernmanagementsysteme (LMS) einbinden zu können, was Web-CAT nicht unterstützt. Graja wird seit Oktober 2012 regelmäßig in Programmieren-Lehrveranstaltungen verschiedener Studiengänge der Hochschule Hannover eingesetzt.

2 Stakeholdergruppen bei Softwaretest und Autobewertung

Beim automatisierten Test professionell entwickelter Software werden in der Regel drei Rollen unterschieden [Gä12]: Fachexperten definieren die fachlichen Anforderungen und die fachlichen Testfälle, Entwickler implementieren die Anforderungen in Programmcode, Tester implementieren die Testfälle in ausführbare Testroutinen.

Eine naive Abbildung dieser drei Rollen auf die Akteure im e-Assessment könnte wie folgt aussehen (vgl. Tab. 1).

[6] Ausnahme: Programmieraufgaben für einfache Java 2D-Ausgaben wurden bereits realisiert.
[7] http://web-cat.org/

Rolle im e-Assessment	Rolle im Softwaretest	Bemerkung
Lehrkraft	Fachexperte	Definiert die vom einzureichenden Programm P_{subm} zu realisierenden Anforderungen.
Studentin	Entwickler	Schreibt das geforderte Programm P_{subm}.
Aufgabenautor	Tester	Schreibt Testroutinen unter Einsatz von Softwaretestwerkzeugen.

Tab. 1: Rollen in e-Assessment und Softwaretest im Hinblick auf das Programm P_{subm}

Diese Zuordnung ignoriert, dass wir es im e-Assessment mit zwei zu entwickelnden Anwendungen zu tun haben. Zum einen entwickeln Studierende ein von der Aufgabe gefordertes Programm P_{subm}. Zum anderen entwickelt der Aufgabenautor ein weiteres Programm P_{task}, das sich wie ein Plugin in die vom Autobewerter angebotenen Programmierschnittstellen einfügt.

Bzgl. des Programms P_{task} bekleidet der Aufgabenautor neben der Entwickler-Rolle häufig[8] auch die des Testers und des Fachexperten (vgl. Tab. 2). In der Rolle des Fachexperten definiert der Aufgabenautor die Anforderungen an P_{task}, d. h. er definiert das Lernziel der Programmieraufgabe, legt das Bewertungsschema fest und überlegt sich mögliche Feedbacktexte für einreichende Studierende. Als Tester von P_{task} erstellt er mögliche studentische Einreichungen (z. B. Standard- und Grenzfälle) und testet diese entweder einzeln manuell oder automatisch.

Rolle im e-Assessment	Rolle im Softwaretest	Bemerkung
Aufgabenautor	Fachexperte, Entwickler, Tester	Erfindet, entwickelt und testet die Aufgabe.
Lehrkraft	Administrator	Passt eine Aufgabe für seine Lehrveranstaltung an und verwendet sie.
Student/-in	Benutzer	Konsumiert die Aufgabe als Lernobjekt

Tab. 2: Rollen in e-Assessment und Softwaretest im Hinblick auf das Programm P_{task}

Aufwändig entwickelte Programmieraufgaben will man wiederverwenden. Wiederverwendende Lehrkräfte passen einzelne Aspekte der Aufgabe an ihre Lehrsituation an, wollen i. d. R. aber nicht in die softwaretechnisch anspruchsvolle (Weiter-)Entwicklung der Aufgabe einsteigen. Sie sind somit am ehesten mit der Benutzerrolle des „Administrators“ in Geschäfts- oder technischen Anwendungen zu vergleichen. Studierende sind Benutzer des Programms P_{task}. Sie konsumieren eine Aufgabe als Lernobjekt.

Eine Analogie zum Musikgeschäft halten wir für hilfreich: Der Student ist in dieser Analogie der Hörer eines Werks, die Lehrkraft ist Interpret, der Aufgabenautor ist Komponist. So wie softwaretechnisches Know-How in dieser Reihung aufsteigend vorhanden

[8] Zumindest werden Programmieraufgaben heutzutage in der Regel nicht arbeitsteilig mit verteilten Rollen spezifiziert, entwickelt und getestet.

sein muss, muss Musik-Know-How aufsteigend in den analogen Rollen vorhanden sein.

3 Was unterscheidet den Softwaretest von der Autobewertung?

In diesem Abschnitt formulieren wir beispielhaft drei Anforderungen an einen Grader (vgl. Tab. 3), die im klassischen Softwaretest nicht vorkommen. Viele weitere derartige Anforderungen, die in diesem kurzen Beitrag keinen Platz finden, werden in einem technischen Bericht detailliert besprochen [Ga15]. Wir argumentieren im Folgenden jeweils, warum die Anforderung im klassischen Softwaretest nicht vorkommt und demnach nicht durch das eingesetzte Softwaretestwerkzeug standardisiert ist. Wir skizzieren die Realisierung der Anforderung im Autobewerter Graja und zeigen auf, welche Abhängigkeiten sich zwischen der Programmieraufgabe und dem Autobewerter ergeben.

Nr.	Anforderung	Stake-holder[9]
(a)	Indirekter Aufruf des „system under test“	A
(b)	Funktionen für intelligente Vergleiche und für die Darstellung von Synopsen	A, S
(c)	Anpassung von Aufgaben durch Lehrkräfte	L

Tab. 3: Beispielanforderungen an Autobewerter, die nicht an Softwaretestwerkzeuge gestellt werden. Viele weitere derartige Anforderungen sind in einem technischen Bericht [Ga15] beschrieben.

Anforderung (a) besagt, dass das zu testende System (das *system under test* = SUT) indirekt aufgerufen werden muss, um Bindungsfehler abfangen zu können. Im professionellen Softwaretest rufen Testroutinen das SUT direkt auf. Fehlen Teile des SUT, bspw. aufgrund einer fehlerhaften Installation des SUT auf dem Testrechner, darf der Test mit einer kurzen Fehlermeldung abbrechen. Im e-Assessment programmiert der Aufgabenautor Aufrufe des studentischen Codes aufwändig und indirekt durch Reflexion[10], da die studentische Lösung im Zeitpunkt der Aufgabenerstellung noch nicht existiert und weil während der Autobewertung auftretende Bindungsfehler abgefangen und lernförderlich aufbereitet werden sollen.

Graja enthält eine Bibliothek für indirekte Aufrufe via Java Reflection. Zudem ermöglicht Graja die automatisch kommentierte Abweisung von Einreichungen mit unerlaubten Packages oder Klassen. Das folgende reduzierte Beispiel zeigt den indirekten Aufruf des Konstruktors der eingereichten Klasse myp.Foo (Zeile 07-08) und der Methode getP (Zeile 09f). Die Annotation der Zeile 02 führt zu einer verständlichen Fehlermeldung, falls die Einreichung Klassen außerhalb des gewünschten Package myp enthält[11]. Zeile

[9] A=Aufgabenautor, L=Lehrkraft, S=Studierende. Letztendlich stehen hinter jeder Anforderung Studierende mit ihren Lernzielen. Mittelbar sind auch Lehrkraft und Aufgabenautor Anforderungsquellen.

[10] https://de.wikipedia.org/w/index.php?title=Reflexion_%28Programmierung%29&oldid=141731957

[11] Text der Fehlermeldung z. B.: Error (class level). cannot find class myp.Foo. The grader expects a class 'Foo' in package 'myp'. Did you put your classes into the right package? (Cause: Found submitted class 'Foo' in the default package, but the only allowed package is myp).

01 ist dafür verantwortlich, dass eine eventuell miteingereichte Klasse myp.Point nicht übersetzt und geladen wird. Die Zeilen 06 und 11f sind in ähnlicher Form auch in normalen JUnit-Testmethoden professionell entwickelter Software zu finden.

```
01 @SkipSubmittedClasses({"myp.Point"})
02 @RestrictSubmissionToPackages({"myp"})
03 public class Grader extends AssignmentGrader {
04  @Test @ScoringWeight(12.5)
05  public void getterShouldReturnValuePassedToConstr() {
06   Point p= new Point(1,2);
07   Class<?> subm= getPublicClassForName("myp.Foo");
08   Object inst= ReflectionSupport.create(subm, p);
09   Point pObs= ReflectionSupport.invoke(
10                            inst, Point.class, "getP");
11   Assert.assertEquals("getP should return the Point"+
12       " that was passed to the constructor", p, pObs);
13  }
14 }
```

Das Beispiel zeigt, dass das e-Assessment-Artefakt P_{task} Abhängigkeiten zur Graja-Bibliothek besitzt, die eine Portierung auf andere auf JUnit basierende Autobewerter zumindest erschwert. Ursächlich ist das Fehlen von Autobewerter-übergreifenden Standardschnittstellen zur Deklaration von Punktzahlen, erlaubten und ignorierten Klassen und Packages sowie indirekten Aufrufen.

Expected and observed behaviours differ.

In the following diff view the observed output was normalized by the following steps:

- remove trailing newlines
- remove trailing spaces per line
- remove uniform indentation

	Expected		Observed	
1	_____¶		_____¶	1
2	/ \¶		/ \¶	2
3	/ \¶	<D>	/ ¬· \¶	3
4	-"-'-"-'-"-¶	<D>	-'-'-'-'-'-¶	4
5	\ /¶		\ /¶	5
6	_____/		_____/	6

Legend: <D>=difference, ¶=line feed, ¬·=tabulator

Abb. 1: Beispiel einer Synopse

Wir wenden uns der Anforderung (b) in Tab. 3 zu: Vergleiche von Ist- und Soll-Ausgaben des studentischen Programms müssen intelligent erfolgen. Manchmal ist im Gegensatz zum klassischen Softwaretest keine exakte Übereinstimmung erforderlich. Im Softwaretest reichen bei fehlschlagenden Assertions die einfachen Soll-Ist-Ausgaben des xUnit-Frameworks aus. Im e-Assessment muss der Aufgabenautor fehlschlagende Assertions abfangen und als lesbare Synopse (Soll-Ist-Vergleich) aufbereiten. Dafür

wünscht er sich Unterstützungsfunktionen im Autobewerter.

Graja bietet Funktionen an, die Ist- und Soll-Ausgaben nach zuvor durchgeführten Normalisierungen vergleichend darstellen. Abb. 1 zeigt eine Beispielausgabe von Graja, die auf Unterschiede zwischen der beobachteten Ausgabe des eingereichten Programms und der erwarteten Ausgabe hinweist. Die abweichenden Zeilen sind in der Mittelspalte der Tabelle markiert. Informationen über durchgeführte Normalisierungen sind darüber aufgelistet. Auch dieses Beispiel zeigt, dass P_{task} Graja-spezifische Abhängigkeiten besitzt, die sich entweder durch redundante Implementierung der Synpose-Funktion in jeder Aufgabe oder durch Auslagerung dieser Funktion in eine Grader-übergreifend einsetzbare (Standard-)Bibliothek lösen ließen.

Schließlich diskutieren wir die Anforderung (c) der Tab. 3. In der professionellen Softwareentwicklung wird ein Testtreiber i. d. R. genau für ein SUT mit möglichst genau spezifizierten Anforderungen an das SUT geschrieben. Ändern sich die Anforderungen an das SUT, wird neben dem Code des SUT auch der Testtreiber gewartet. Die Wartung wird regelmäßig von Personen mit softwaretechnischer Expertise durchgeführt, so dass keine Anforderungen an die Anpassung des Testtreibers durch Laien bestehen. Im Gegensatz dazu fordern Lehrkräfte von einem Autobewerter, dass sich eine Aufgabe auch mit wenig Entwicklungs-Expertise und -Aufwand an ihren jeweiligen Einsatzzweck anpassen lässt.

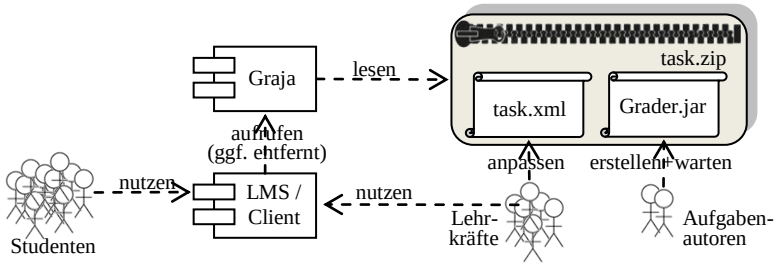

Abb. 2: Stakeholder und Nutzungsarten

Graja erwartet Aufgaben in Gestalt einer *task.zip*-Datei (vgl. Abb. 2) mit genau definiertem Aufbau. Darin sind zwei verschiedene Gruppen von Dateien enthalten, von denen in Abb. 2 je ein Vertreter skizziert ist. Eine *task.xml*-Datei enthält u. a. den Aufgabentitel und die maximal erreichbare Punktzahl. *Grader.jar* enthält den Bytecode von P_{task} mit vielen weiteren Festlegungen (Gewichtung der einzelnen Bewertungsaspekte, Texte, etc.). Abb. 2 prägt die Begriffe *Wartung* für die Änderung einer Aufgabe durch einen Aufgabenautor (mit JUnit-Fachwissen) und *Anpassung* für die Änderung durch eine Lehrkraft (ohne JUnit-Fachwissen). Eine Lehrkraft, die eine XML-Datei bearbeiten kann, kann eine Aufgabe in dem Maße an eigene Zwecke anpassen, wie es Parameter in der XML-Datei gibt[12].

[12] Es ist geplant, vermehrt Parameter aus dem Wartungsbereich (JUnit-Testtreiber) in den Anpassungsbereich

Auch dieses Beispiel zeigt, dass die zu einer Programmieraufgabe gehörenden Artefakte in hohem Maße von dem verwendeten Autograder abhängen, weil die sich aus Wartung und Anpassung ergebenden Anforderungen nicht standardisiert in JUnit umgesetzt sind.

4 Zusammenfassung

Im vorliegenden Beitrag haben wir Rollenanalogien zwischen dem Softwaretest und dem e-Assessment aufgezeigt. Aus Sicht der verschiedenen Rollen haben wir einige spezielle Anforderungen identifiziert, die das e-Assessment vom Softwaretest abgrenzen. Die speziellen Anforderungen an Autobewerter führen in der Regel zu Abhängigkeiten, die die Portierung einer Programmieraufgabe von einem Autobewerter zu einem anderen erheblich erschweren. Wir haben dies am Beispiel des Autobewerters „Graja“ illustriert; ähnliche Abhängigkeiten werden jedoch auch von anderen Autobewertern definiert. Ein interessantes zukünftiges Forschungsfeld ist, die Portabilität von Programmieraufgaben zu verbessern, indem häufig von Aufgabenautoren genutzte Funktionen Autobewerter-übergreifend standardisiert und in frei verfügbaren Bibliotheken implementiert werden.

Literaturverzeichnis

[BM11] Bath, G.; McKay, J.: Praxiswissen Softwaretest, 2. Aufl., dpunkt, 2011.

[Me07] Meszaros, G.: xUnit Test Patterns: Refactoring Test Code, Addison Wesley, 2007.

[SSM+15] Strickroth, S.; Striewe, M.; Müller, O.; Priss, U.; Becker, S.; Rod, O.; Garmann, R.; Bott, O.; Pinkwart, N.: ProFormA: An XML-based exchange format for programming tasks. eleed e-learning & education, 11(1), 2015.

[IAK+10] Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O.: Review of recent systems for automatic assessment of programming assignments. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research, S. 86-93, 2010.

[Gä12] Gärtner, M.: ATDD by Example, Addison Wesley, 2012.

[Ga15] Garmann, R.: E-Assessment mit Graja – ein Vergleich zu Anforderungen an Softwaretestwerkzeuge, Forschungsbericht, http://serwiss.bib.hs-hannover.de/frontdoor/index/index/docId/618, 2015.

(XML-Datei) zu verlagern, um die Anzahl möglicher Einsatzszenarien einer Graja-Aufgabe durch leichte Anpassung der Parameter zu vergrößern.