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

Robert Garmann[1]

HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS

*Fakultät IV*
*Wirtschaft und*
*Informatik*

Forschungsbericht

24. Juni 2015

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

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

### Zusammenfassung

Die automatisierte Bewertung studentischer Übungsabgaben in Programmieren-Lehrveranstaltungen weist Parallelen zum automatisierten Test in der professionellen Softwareentwicklung auf. Allerdings muss ein Autobewerter (Grader), um lernförderlich zu sein, andere Zielsetzungen erfüllen als üblicherweise im professionellen Softwaretest eingesetzte Analyse- und Testwerkzeuge. Dieser Beitrag identifiziert wesentliche Unterschiede und beschreibt, wie sich diese Unterschiede in dem an der Hochschule Hannover entwickelten und seit mehreren Jahren im Einsatz befindlichen Autobewerter „Graja“ niederschlagen.



### Schlagworte

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

### DDC Klassifikation

004 Datenverarbeitung; Informatik

### GND-Schlagworte

Programmierung, Softwaretest, E-Learning, Computerunterstütztes Lernen, Java <Programmiersprache>, Konfiguration <Informatik>, Softwarewartung, Übung <Hochschule>, Lernaufgabe, Softwarewerkzeug, JUnit

### 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 testing and debugging*


# 1 Einleitung

## 1.1 Nutzen und Herausforderungen des formativen Assessments von Programmieraufgaben

Programmieren-Lehrveranstaltungen in Informatik- und Informatik-Mischstudiengängen beinhalten in der Regel immer einen großen Anteil praktischer Übungselemente, weil die praktische Erprobung gelernter Programmierkonzepte nach einhelliger Meinung entscheidend für einen nachhaltigen Lernerfolg ist. Die praktische Übung erfolgt in den ersten Studiensemestern häufig durch Bearbeitung vieler kleiner Programmieraufgaben, in denen Studierende ein Programm nach einer vorgegebenen Spezifikation erstellen müssen. Um den Lernerfolg abzusichern, ist regelmäßiges Feedback zu den von Studierenden erarbeiteten Lösungen notwendig („formatives Assessment“).

Um adäquates Feedback geben zu können, sind häufig studentische Hilfskräfte mit der Sichtung, Bewertung und Kommentierung von Einreichungen befasst. Dabei fallen hohe menschliche Aufwände für den „Test“ des studentischen Programms auf Übersetzbarkeit und funktionale Korrektheit an. Bei vielen zu begutachtenden Einreichungen gerät der Test

zu einer ermüdenden und dadurch häufig von menschlichen Gutachtern nicht in gleichbleibender Güte erledigten Aufgabe. Zudem liegen zwischen Einreichung einer Lösung und Erhalt des Feedbacks in der Regel ein oder mehrere Tage. Unmittelbares Feedback, welches schon nach wenigen Sekunden zur Verfügung stünde, könnte den Lernprozess frühzeitig in vorteilhafter Weise steuern [1].

Bei Programmier-Leistungen kommt es nicht nur auf die funktionale Korrektheit des Programms an, sondern auch auf Qualitätseigenschaften wie Wartbarkeit, Effizienz, Benutzbarkeit, etc. (vgl. [2]). Die Qualitätseigenschaften werden von studentischen Hilfskräften erfahrungsgemäß in sehr unterschiedlicher Güte befeedbackt. Die Aufwände für Sichtung, Bewertung und Feedback sind so hoch, dass sie von qualifizierterem Personal nicht in vertretbarer Weise in einem für den Lernerfolg idealen Umfang geleistet werden könnten.

## 1.2 Parallelen zum Software-Test

Tests werden in der professionellen Softwareentwicklung üblicherweise automatisiert, um Tests mit geringen Kosten wiederholen zu können. Im Assessment-Kontext ist Testautomatisierung 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 [3]:

- xUnit-Testframeworks [4]
- Werkzeuge der statischen Analyse (z. B. PMD[2], Checkstyle[3])
- Werkzeuge 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. Die obigen Auflistung der im Softwaretest eingesetzten Werkzeugklassen ist keineswegs vollständig sondern beschränkt sich auf diejenigen, die ebenfalls häufig im e-Assessment eingesetzt werden.

Seit mehreren Jahren wurden und werden Autobewerter für verschiedene Programmiersprachen für verschiedene Einsatzszenarien entwickelt (s. z. B. [5] [6] [7] [8] für einen Überblick). In diesem Beitrag betrachten wir solche Autobewerter, die Testmethoden und -werkzeuge aus der professionellen Softwareentwicklung einsetzen[5]. Eine Programmieraufgabe kann hier etwa aus einem JUnit-Testtreiber bestehen, der einen funktionalen Test der studentischen Einreichung durchführt, und dazu aus einer PMD-Konfigurationsdatei, die unerwünschte Verstöße gegen Programmierkonventionen definiert (s. etwa Web-CAT[6]). Trotz der offensichtlichen Parallelen bestehen Unterschiede in der Zielsetzung professioneller Testautomatisierungswerkzeuge und der Zielsetzung der Generierung eines automatischen, lernförderlichen Feedbacks durch einen Autobewerter.

## 1.3 Überblick über diesen Beitrag

Kapitel 2 beleuchtet als wesentlichen Unterschied die Stakeholdergruppen, die einerseits beim professionellen Softwaretest und andererseits bei der Autobewertung beteiligt sind. In den darauf folgenden Abschnitten des Kapitels 3 formulieren wir auf der Basis der Ziele der einzelnen Stakeholdergruppen jeweils eine Anforderung an den Autobewerter, die nicht oder nicht in gleicher Weise an Softwaretestwerkzeuge gestellt wird. In jedem Abschnitt erläutern

[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 Vergleich der Einreichung mit guten und schlechten Musterlösungen, vgl. [1]), werden hier nicht weiter betrachtet.
[6] http://web-cat.org/

wir jeweils eine mögliche Umsetzung der Anforderung, 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[7]. 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[6], 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 [9]: Fachexperten definieren die fachlichen Anforderungen und die fachlichen Testfälle, Entwickler implementieren die Anforderungen in Programmcode, Tester implementieren die Testfälle in ausführbare Testroutinen. Mehrere Rollen werden mitunter von einer Person bekleidet. So ist die Entwicklung von Modultests mit xUnit-Frameworks häufig integraler Bestandteil der Entwicklung und wird von den Entwicklern erledigt. Fachexperten sind hingegen und in aller Regel nie in die Erzeugung von Programmcode oder Testroutinen involviert. Über die drei genannten Rollen hinaus gibt es weitere Rollen, insbesondere die der Benutzer, die die getestete Anwendung für ihre Benutzerziele einsetzen.

Eine naive Abbildung der drei Rollen Fachexperte, Entwickler und Tester auf die Akteure im e-Assessment könnte wie folgt aussehen (vgl. Tabelle 1).

| Rolle im e-Assessment | Rolle im Softwaretest | Bemerkung |
|---|---|---|
| Lehrkraft | Fachexperte | Definiert die vom studentischen Programm zu realisierenden Anforderungen. |
| Studentin | Entwickler | Schreibt das geforderte Programm und reicht es zur Bewertung ein. |
| Aufgabenautor | Tester | Schreibt Testroutinen unter Einsatz von Softwaretestwerkzeugen. |

Tabelle 1: Naive Zuordnung der Rollen in e-Assessment und Softwaretest

[7] Ausnahme: Programmieraufgaben für einfache Java 2D-Ausgaben wurden bereits realisiert.

Tatsächlich ignoriert diese Zuordnung einige Besonderheiten des e-Assessments. Im e-Assessment haben wir es mit zwei zu entwickelnden Anwendungen zu tun. Zum einen entwickeln Studierende ein von der Aufgabe gefordertes Programm $P_{subm}$, welches die Grundlage der oben dargestellten Zuordnung ist. Zum anderen entwickelt der Aufgabenautor ein weiteres Programm $P_{task}$, das sich in die vom Autobewerter angebotenen Programmierschnittstellen einfügt. Im einfachsten Fall nutzt $P_{task}$ die Programmierschnittstelle eines Frameworks wie JUnit oder die Konfigurationsschnittstelle eines Analysewerkzeuges wie PMD. In komplexeren Fällen kann der Aufgabenautor umfangreiche, vom Autobewerter mitgelieferte Bibliotheken nutzen. Eine solche Entwicklungstätigkeit ist vielleicht am besten mit der Entwicklung eines Plugins vergleichbar. In der Rolle des $P_{task}$-Entwicklers benötigt der Aufgabenautor erhebliche softwaretechnische Expertise. In der Rolle des Testers prüft der Aufgabenautor nicht nur die studentische Einreichung $P_{subm}$ sondern auch sein Programm $P_{task}$, indem er mögliche studentische Einreichungen (z. B. Standard- und Grenzfälle) erstellt und entweder einzeln manuell oder automatisch überprüfen lässt.

Der Aufgabenautor bekleidet neben den Rollen Tester und Entwickler häufig auch noch die des Fachexperten. Zumindest werden Programmieraufgaben heutzutage in der Regel nicht arbeitsteilig mit verteilten Rollen entwickelt und getestet. 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.

Aufwändig entwickelte Programmieraufgaben will man vielfach wiederverwenden. Lehrkräfte werden somit nicht notwendigerweise gleichzeitig Aufgabenautor sein. Die wiederverwendende Lehrkraft wird die Aufgabe in der Regel nicht identisch sondern leicht variiert einsetzen. Die Lehrkraft ist somit ein Benutzer der Programmieraufgabe mit ganz besonderen Benutzungsanforderungen. Am ehesten ist die Lehrkraft mit der Benutzerrolle „Administrator“ in Geschäfts- oder technischen Anwendungen zu vergleichen. Lehrkräfte besitzen besondere Anforderungen an die Konfiguration der Programmieraufgabe. Dafür kann man ein gewisses softwaretechnisches Know-How der Lehrkraft erwarten, das jedoch in aller Regel nicht so umfassend wie das des Aufgabenautors sein wird.

Studierende sind nicht nur Entwickler, sondern auch und in erster Linie Benutzer. Sie konsumieren eine Programmieraufgabe als Lernobjekt. Sie haben im Vergleich zu Aufgabenautoren und Lehrkräften vermutlich das geringste softwaretechnische Wissen. Nach diesen Ausführungen stellt sich die Zuordnung der Rollen wie in Tabelle 2 dar.

| **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 |

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

Eine Analogie zum Musikgeschäft halten wir für hilfreich: Der Student ist in dieser Analogie der Hörer eines Werks, die Lehrkraft ist Interpret des Werks, der Aufgabenautor ist der Komponist. So wie softwaretechnisches Know-How in dieser Reihung aufsteigend vorhanden 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 Anforderungen an einen Autobewerter, wie sie von einer oder mehreren Stakeholdergruppen gestellt werden (vgl. Tabelle 3). Wir argumentieren,

warum diese Anforderungen im klassischen Softwaretest nicht vorkommen. Anschließend skizzieren wir jeweils die Realisierung der Anforderung im Autobewerter Graja bzw. legen dar, wie die noch nicht realisierte Anforderung in Graja umgesetzt werden kann.

| Nr. | Anforderung | Stakeholder[8] | in Graja umgesetzt |
|---|---|---|---|
| (a) | Ressourcenbegrenzung und sichere Ausführung | Lehrkraft | ja |
| (b) | LMS-Interoperabilität und prompte Ergebnisanzeige | Studierende | ja |
| (c) | Einfache Benutzungsschnittstelle und lernförderliche Ergebnisaufbereitung | Studierende | ja |
| (d) | Offline-Nutzbarkeit | Studierende | nein, geplant |
| (e) | Indirekter Aufruf des „system under test“ | Aufgabenautor | ja |
| (f) | Funktionen für intelligente Vergleiche und für die Darstellung von Synopsen | Aufgabenautor | ja |
| (g) | Gewichtung von Testroutinen | Aufgabenautor, Lehrkraft | ja |
| (h) | Verschiedenes Feedback für Lehrkräfte und Studierende | Lehrkraft | ja |
| (i) | Lokalisierung einer Aufgabe | Aufgabenautor | ja |
| (j) | Internationalisierung einer Aufgabe | Aufgabenautor, Lehrkraft, Studierende | nein, geplant |
| (k) | Entwicklung und Auslieferung von Aufgaben unterstützen | Aufgabenautor | ja |
| (l) | Anpassung von Aufgaben durch Lehrkräfte | Lehrkraft | gering, Ausbau geplant |
| (m) | Kontextabhängige Testläufe, Bestrafung und Belohnung | Lehrkraft | nein, geplant |
| (n) | Variantengenerierung | Lehrkraft | nein, geplant |

Tabelle 3: Anforderungen an Autobewerter, die nicht an Softwaretestwerkzeuge gestellt werden

## 3.1 Ressourcenbegrenzung und sichere Ausführung

Anforderung: Die studentische Einreichung $P_{subm}$ muss während der Ausführung in ihrem Ressourcenzugriff (flüchtiger und nichtflüchtiger Speicherplatz, Rechenzeit, weitere Betriebsmittel) beschränkt werden. Es muss eine sichere Ausführung ohne unerwünschte Seiteneffekte sichergestellt werden. Stakeholder der Anforderung: Lehrkraft (ggf. in Vertretung für den Betreiber des Autobewerters).

Im klassischen Softwaretest vertraut der Testcode der getesteten Anwendung. Im e-Assessment hingegen muss der Autobewerter mit versehentlichen oder böswilligen Angriffen der getesteten Einreichung rechnen.

Graja startet für eine Programmieraufgabe für jede Einreichung eine eigene JVM (java virtual machine). Die Kommandozeilenparameter beim JVM-Start verschaffen der in der Aufgabe festgelegten Sicherheitspolicy (Syntax angelehnt an die „default policy file implementation“[9]) und der in der Aufgabe festgelegten Hauptspeicherbegrenzung Wirkung [10]. In einer Aufgabe muss die maximale Rechenzeit festgelegt werden (wall-clock time). Nach Ablauf dieser Zeit beendet Graja die Ausführung. Um den für die studentische Einreichung

[8] Letztendlich stehen hinter jeder Anforderung Studierende mit ihren Lernzielen. Mittelbar treten allerdings auch Lehrkräfte und Aufgabenautoren als Stakeholder auf. Wir haben diejenigen Stakeholder aufgeführt, die unmittelbar betroffen sind. Aufgabenautoren etwa sind unmittelbar von der durch den Autobewerter angebotenen Programmierschnittstelle betroffen, Lehrkräfte sind unmittelbar von Konfigurationsschnittstellen der Programmieraufgabe betroffen, Studierende sind unmittelbar von der Benutzungsschnittstelle des Autobewerters betroffen.

[9] http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html

verfügbaren Plattenplatz zu begrenzen, legt Graja eine Datei geeigneter Größe an und verwendet diese als Loopback Device[10].

## 3.2 LMS-Interoperabilität und prompte Ergebnisanzeige

Anforderung: der Autobewerter muss über Schnittstellen verfügen, die den Einsatz in dem für den Kurs eingesetzten LMS ermöglichen. Beim formativen Assessment müssen Bewertungsergebnisse ohne erhebliche Verzögerung (z. B. höchstens 30 Sekunden) nach dem Einreichen eines Programms einsehbar sein. Stakeholder der Anforderung: In erster Linie Studierende, in zweiter Linie auch Lehrkräfte.

Im klassischen Softwaretest ist der automatisierte Start von Testsuiten durch z. B. Shell-Skripte erforderlich und i. d. R. ausreichend. Nicht gefordert wird der Start des Testprozesses über eine Webanwendung. Verzögerungen im Sekunden- oder Minutenbereich bei der Anzeige des Testergebnisses sind in einem verteilten Continuous-Integration-Szenario häufig vernachlässigbar, da umfangreiche Testsuiten mitunter mehrere Stunden laufen.

Graja kann über die Kommandozeile in einer eigenen JVM gestartet werden. Außerdem kann Graja über ein Java API direkt aus einer laufenden JVM gestartet werden. In beiden Fällen müssen mehrere Informationen angegeben werden: Aufgabenstruktur, Versionsinformationen, Maximalpunktzahl, Angaben zur Ressourcenbegrenzung, Dateipfade zu jar-Archiven, zur Sicherheitspolicy und zur Einreichung. Auf der Kommandozeile übergibt man dazu eine diese Informationen enthaltende Zip-Datei, am Java API stattdessen ein Java-Objekt. Durch diese beiden Schnittstellen kann Graja direkt auf dem LMS-Server ausgeführt werden, wenn dort ein JDK installiert ist[11]. Eine Webservice-Schnittstelle für einen entfernten Aufruf gehört nicht zum Lieferumfang von Graja. Hierfür wurde an der Hochschule Hannover das Produkt Grappa [11] entwickelt, welches als Webservice-Schicht für verschiedenste Grader konzipiert ist. Im Verbund mit Grappa und einem speziell dafür entwickelten GrajaPlugin kann Graja über einen REST-konformen Webservice aus einem LMS heraus aufgerufen werden (vgl. Abb. 1). Die Lehrkraft lädt für jede Aufgabe eine *task.zip*-Datei (Format: [5]) im LMS hoch. Ein Student lädt eine *submission.zip*-Datei in einem Graja-spezifischen Format hoch. Das LMS überträgt diese Dateien im Hintergrund an Grappa/Graja und zeigt anschließend das im HTML-Format zurückerhaltene Bewertungsergebnis an.

Eine Besonderheit von Graja ist die Unterstützung von sog. Multi-Requests. Wenn ein Student ein Übungsblatt mit mehreren Aufgaben bearbeitet, kann Graja die Aufgabenlösungen nicht nur einzeln bewerten, sondern auch alle in einem Zuge. Dadurch kann die Wartezeit für die einreichenden Studierenden bei einem entfernten Graja-Aufruf deutlich reduziert werden. Das LMS muss nur einen Graja-Aufruf für die gesamte Einreichung tätigen, statt vieler Einzelaufrufe für jede Aufgabe des Übungsblattes.

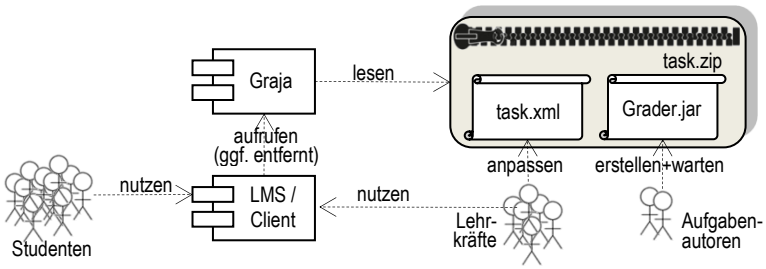



Abb. 1: Stakeholder und Nutzungsarten

[10] https://de.wikipedia.org/w/index.php?title=Loop_device&oldid=139906126
[11] Ein Java runtime environment (JRE) reicht nicht, weil Graja studentischen Code compilieren muss.

## 3.3 Einfache Benutzungsschnittstelle und lernförderliche Ergebnisaufbereitung für Studierende

Anforderung: Die Benutzungsschnittstelle des Autobewerters muss einfach zu bedienen sein, das Feedback muss lernförderlich aufbereitet sein. Stakeholder: Studierende.

Im professionellen Softwaretest werden Profi-Werkzeuge wie xUnit-Frameworks oder Analyse-Werkzeuge von Softwareprofis genutzt. Die Steuerung der Werkzeuge und die Lektüre der Detail-Ausgabe setzt erhebliche softwaretechnische Expertise voraus. Umfangreiche Konfigurationsmöglichkeiten der Werkzeuge sind unerlässlich. Es bestehen jedoch keine Anforderungen an Verständlichkeit für Lernende. Projektleiter haben manchmal die Anforderung an eine aggregierte Darstellung der Testergebnisse in Gestalt von Dashboards, wie es etwa das Werkzeug SonarQube[12] anbietet. Diese Anforderung wiederum ist im formativen Assessment weniger wichtig, weil qualitatives Feedback mit Tipps für Verbesserungspotential der eingereichten Programme den Lernerfolg besser unterstützt als z. B. eine Gesamtzahl entdeckter Fehler.

Graja verbirgt die Natur des eingesetzten JUnit-Werkzeuges fast vollständig vor dem studentischen Benutzer. Die Studierenden bekommen keine Testklassen zu Gesicht, Lehrkräfte bekommen lediglich eine *task.zip*-Datei als Aufgabenartefakt in die Hand (vgl. Abb. 1). Übersetzungs- und Ausführungspfade, Konfigurationsdateien, etc., müssen weder von der Lehrkraft noch von Studierenden angegeben werden. Prototypisch wurde zudem das Werkzeug PMD für eine statische Stilprüfung so in Graja eingebaut, dass Studierende und Lehrkräfte mit der Steuerung von PMD nichts zu tun haben.

Graja legt besonderen Wert auf eine gut strukturierte Ausgabe des Bewertungsergebnisses. Zu jedem Bewertungsaspekt vergibt Graja Punkte und ggf. einen Hinweis, wenn der Bewertungsaspekt nicht erfüllt wurde. Punkte und Text-Feedback stehen im Ergebnis direkt nebeneinander, damit Studierende leicht den Zusammenhang erkennen können. Es ist möglich, Bilder in das Feedback einzubetten.

Graja nutzt HTML zur Aufbereitung des Feedbacks und kann bei Bedarf „plain text“ generieren. Consolen-Ausgaben eines Compiler- oder JUnit-Durchlaufs werden nicht eins-zu-eins ins Feedback kopiert, weil dies die Suche nach dem Fehler für Studierende erschweren würde. Stacktraces werden von Graja auf den für Studierende interessanten oberen Stapelbereich gekürzt. Auch das Feedback des prototypisch eingebauten PMD-Werkzeuges wird von Graja lernförderlich aufbereitet, indem die entdeckten Regelverletzungen beschrieben und mit dem entsprechenden Code-Ausschnitt illustriert werden. All diese Maßnahmen sind in Graja einfach möglich, weil Graja auf das Java-API von Compiler, JUnit und PMD zugreift, welches einen komfortablen und wahlfreien Zugriff auf Einzelaspekte des Testergebnisses erlaubt. Auch können so Erkenntnisse des Compilers und Erkenntnisse von PMD über ein und dieselbe Quelltextdatei zusammenhängend dargestellt werden. In einem Autobewerter, der Übersetzungsvorgang, JUnit-Durchlauf und Stilprüfung als drei voneinander getrennte Prozesse betrachtet, wäre eine solch hoch integrierte Darstellung des Feedbacks nicht einfach möglich.

## 3.4 Offline-Nutzbarkeit

Anforderung: Der Autobewerter muss von Studierenden zumindest phasenweise offline, d. h. ohne Vermittlung des LMS, genutzt werden können, um auch ohne Internetverbindung Bewertungsfeedback zu Zwischenlösungen erhalten zu können. Stakeholder: Studierende.

Softwaretestwerkzeuge werden in der Regel immer auf einem lokalen Rechenknoten ohne besondere Anforderungen an eine Online-Schnittstelle betrieben. Die „Offline“- Anforderung ist im professionellen Softwaretest also eigentlich immer erfüllt. Beispielsweise wird eine xUnit-Testsuite lokal durch ein Shellskript o. ä. gestartet; das Ergebnis wird in eine Logdatei protokolliert. Wird das Shellskript automatisiert (z. B. durch einen nightly build) oder

[12] http://www.sonarqube.org/

ereignisgesteuert (z. B. durch ein Einchecken in die Versionsverwaltung im Rahmen einer kontinuierlichen Integration) gestartet, werden Protokolldateien häufig am Ende an entfernte Rechenknoten übertragen.

Graja besitzt zur Zeit lediglich eine Kommandozeilenschnittstelle zur Benutzung ohne LMS, die Studierenden so nicht zuzumuten ist. Es ist geplant, einen graphischen Client zu entwickeln, der die Benutzung vereinfacht. Einen ähnlichen Client gibt es bereits für von Aufgabenautoren durchgeführte Tests neu entwickelter Aufgaben (vgl. auch Abschnitt 3.11). Über die Entwicklung des Clients hinaus müssen u. a. folgende Probleme gelöst werden:

- Unabhängigkeit von der Laufzeitumgebung des Studierendenrechners und von den dort zur Verfügung stehenden Bibliotheken.
- Möglichst installationsfreie Nutzung auf dem Studierendenrechner.
- Einbindung von ausschließlich offen lizensierter Drittsoftware, um die kostenfreie Nutzung auf dem Studierendenrechner zu ermöglichen.
- Einfache Einbindung einer neuen Aufgabe (*task.zip*, vgl. Abb. 1) in die Graja-Instanz auf dem Studierendenrechner.
- Verhinderung der Dechiffrierung der manchmal in *Grader.jar* (vgl. Abb. 1) enthaltenen Musterlösung der Aufgabe.

## 3.5 Indirekter Aufruf des „system under test“

Anforderung: Das zu testende System muss indirekt aufgerufen werden, um Bindungsfehler abfangen zu können. Stakeholder: Aufgabenautoren.

Im professionellen Softwaretest rufen Testroutinen das zu testenden Systems (das *system under test* = SUT) direkt auf. Fehlen Teile des SUT, bspw. aufgrund einer fehlerhaften Installation des SUT auf dem Testrechner, darf der Test mit einer Fehlermeldung abbrechen. Fehlermeldungen müssen möglichst genau den Ort der Fehlerentstehung benennen. Der Text der Fehlermeldung kann kurz sein und ist an Profis adressiert, die den Fehler dann normalerweise eingehender analysieren müssen. Diese Anforderungen führen normalerweise zu kurzen Testroutinen, die sich auf die zu testenden Eigenschaften des SUT konzentrieren. Im Lernszenario programmiert der Aufgabenautor Aufrufe des studentischen Codes aufwändig und indirekt durch Reflexion [12], 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. Weiterhin 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 de.hsh.exmpl.TheClass (Zeile 07-08) und der Methode getP (Zeile 09). Die Annotation der Zeile 02 führt zu einer verständlichen Fehlermeldung, falls die Einreichung Klassen außerhalb des gewünschten Package de.hsh.exmpl enthält[13]. Zeile 01 ist dafür verantwortlich, dass eine eventuell miteingereichte Klasse de.hsh.exmpl.Point nicht übersetzt und geladen wird. Die Zeilen 06 und 10f sind in ähnlicher Form auch in normalen JUnit-Testmethoden professionell entwickelter Software zu finden.

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

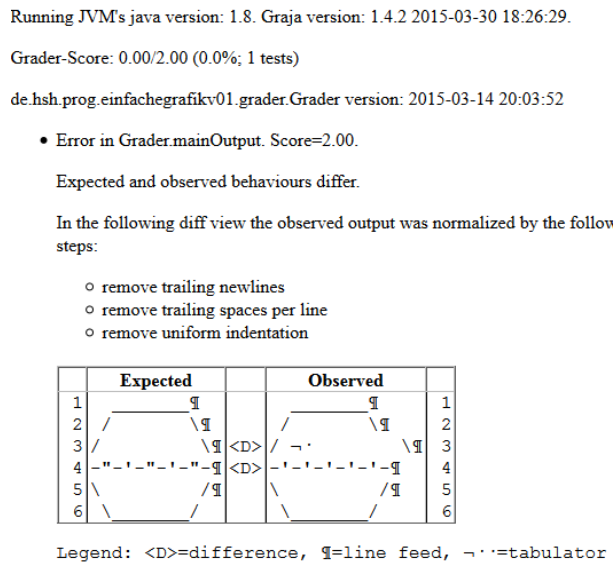
```
01 @SkipSubmittedClasses({"de.hsh.exmpl.Point"})
02 @RestrictSubmissionToPackages({"de.hsh.exmpl"})
03 public class Grader extends AssignmentGrader {
04   @Test @ScoringWeight(12.5)
05   public void getterShouldReturnValuePassedToConstr() {
06     Point p= new Point(1,2);
07     Class<?> subm= getPublicClassForName("de.hsh.exmpl.TheClass");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

## 3.6 Funktionen für intelligente Vergleiche und für die Darstellung von Synopsen

Anforderung: Vergleiche von Ist- und Soll-Ausgaben des studentischen Programms müssen intelligent erfolgen. Manchmal ist keine exakte Übereinstimmung erforderlich. Bei Fehlern müssen Ist- und Soll-Werte in einer für Programmieranfänger geeigneten Weise aufbereitet werden. Stakeholder: Aufgabenautoren.

Im professionellen Softwaretest reichen bei fehlschlagenden Assertions die einfachen Soll-Ist-Ausgaben des xUnit-Frameworks aus. Auch reichen einfache Vergleiche von Ist- und Sollwert aus. Im Lernszenario müssen Aufgabenautoren die Testroutinen mit lernförderlichen Fehlermeldungen ausstatten. Fehlschlagende Assertions des xUnit-Frameworks werden vom Aufgabenautor abgefangen und als lesbare Synopse (Soll-Ist-Vergleich) aufbereitet. Weiterhin muss ein Aufgabenautor verschiedene Lösungsmöglichkeiten einer Aufgabe abdecken, indem er intelligente Vergleiche von Ist- und Sollergebnis des studentischen Programms durchführt.

Graja enthält eine Bibliothek für intelligente Ergebnisvergleiche. Ein Vergleich von Ausgaben erfolgt dabei erst nach einer zuvor durchgeführten Normalisierung der Ist- und Soll-Ergebnisse. Für die Erstellung formatierter Kommentare und Synopsen bietet Graja mehrere Funktionen an, die auch die vorher durchgeführten Normalisierungen berücksichtigen. Abb. 2 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.



Running JVM's java version: 1.8. Graja version: 1.4.2 2015-03-30 18:26:29.

Grader-Score: 0.00/2.00 (0.0%; 1 tests)

de.hsh.prog.einfachegrafikv01.grader.Grader version: 2015-03-14 20:03:52

- Error in Grader.mainOutput. Score=2.00.

  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. 2: Beispiel einer Synopse

## 3.7 Gewichtung von Testroutinen

Anforderung: Fehler des studentischen Programms müssen gewichtet werden können. Stakeholder: Aufgabenautoren und Lehrkräfte.

Im professionellen Softwaretest wird zunächst jeder fehlschlagende Testaspekt gleich behandelt. Bei einer Sichtung des Testergebnisses entscheidet der Tester oder der Entwickler, wie schwer einzelne Fehler einzuschätzen sind.

Im Lernszenario müssen Aufgabenautoren die Testroutinen mit Fehlergewichten ausstatten, um auf „halb richtige“ Lösungen Teilpunkte nach fachdidaktischen Kriterien vergeben zu können.

In Graja kann jede JUnit-Testmethode durch eine Java-Annotation mit einem Gewicht (*ScoringWeight*) versehen werden, welches von Graja bei der Berechnung der erreichten Gesamtpunktzahl berücksichtigt wird (vgl. Codebeispiel in Abschnitt 3.5). Es ist möglich, Testmethoden durch eine separate Annotation als Platzhalter für Bewertungen durch menschliche Gutachter vorzusehen. Graja erzeugt dann einen entsprechenden Hinweis im Bewertungstext[14].

## 3.8 Verschiedenes Feedback für Lehrkräfte und Studierende

Anforderung: Der Autobewerter muss zusätzliches Feedback für die Lehrkraft generieren, das nicht für den einreichenden Studenten bestimmt ist. Die Lehrkraft kann anhand des zusätzlichen Feedbacks besser nachvollziehen, was die Ursache eines von Graja gemeldeten Fehlers in der studentischen Einreichung ist. Außerdem kann das Lehrkraftfeedback Musterlösungen o. ä. enthalten. Stakeholder: Lehrkraft.

Im professionellen Softwaretest werden Ausgaben von Testwerkzeugen von Testern und Entwicklern gelesen. Es müssen keine Informationen vor einer Lesergruppe verborgen werden. Im Szenario der Autobewertung sind die Adressaten des Feedbacks Studierende und Lehrkräfte, die am besten mit einfachen Benutzern und Administrator-Benutzern vergleichbar sind (s. Kapitel 2). Aufgrund der unterschiedlichen Rechte und Fähigkeiten der beiden Lesergruppen besteht hier die Anforderung eines zielgruppenspezifischen Feedbacks.

Graja bietet Aufgabenautoren Funktionen für die Erzeugung von Hinweistexten mit einer anzugebenden Zielgruppe an. Der folgende Codeausschnitt zeigt beispielhaft die Behandlung einer Ausnahmesituation in einer Aufgabe, deren Bewertung aufgrund eines internen Fehlers abgebrochen werden muss. Die Meldung an die Lehrkraft enthält Detailinformationen, die vor dem Studenten verborgen werden.

```
if (/* cannot create file */) {
  throw new CommentAssertionError(
    InlineComment.format(Level.SEVERE, Audience.TEACHER,
                         "Cannot create file %s", fname),
    InlineComment.format(Level.SEVERE, Audience.STUDENT,
                         "Internal error. Please ask the teacher for details."));
}
```

## 3.9 Lokalisierung

Anforderung: Der Autobewerter muss Lokalisierungseinstellungen der Aufgabe berücksichtigen. Stakeholder: Aufgabenautoren.

Im professionellen Softwaretest lassen sich Werkzeuge üblicherweise lokalisieren (Einstellung von Codierungen, Zahlenformaten, etc.). Diese Einstellungen ändern sich in der

[14] Text z. B.: Note: HumanScoreProxies.shouldBeMaintainableAndAdhereToAllRequirements. Score=2.00. This score will be awarded by humans. Humans will manually assign scores for your implementation. If this is the only message and you have written maintainable code and implemented all requirements of the assignment, consider your submission as successfully graded by Graja.

Regel nicht von Testlauf zu Testlauf, d. h. die Lokalisierungsanforderungen des SUT bleiben konstant. Im Lernszenario kann jede Aufgabe eine andere Lokalisierung der Laufzeitumgebung erfordern. Der Autobewerter muss sich im Gegensatz zu einem klassischen Testwerkzeug auf variierende Lokalisierungsanforderungen des gerade ausgeführten Programms $P_{task}$ einstellen können.

Graja ermöglicht die Einstellung der Standard-Lokalisierung und der Zeichensatzcodierung deklarativ über Java-Klassenannotationen in den Testklassen. Der folgende Codeausschnitt zeigt beispielhaft, wie die Einstellung vorgenommen werden kann:

```
@DemandCharset("UTF-8")
@DemandLocale(language="en", country="US")
public class Grader extends AssignmentGrader { …
```

## 3.10 Internationalisierung

Anforderung: Der Autobewerter muss eine Einreichung in jeder von der Aufgabe vorgesehenen Lokalisierung bewerten können. Ergebniskommentare müssen in jeder von der Aufgabe vorgesehenen Sprache / Regionseinstellung produzierbar sein. Stakeholder: Aufgabenautoren, Lehrkräfte, Studierende.

Im professionellen Softwaretest ist Internationalisierung des SUT ein Testgegenstand an sich. Es bestehen keine Anforderungen an mehrsprachige Fehlermeldungen des Testwerkzeugs. Hier im Lernszenario soll nicht das SUT internationalisiert sein, sondern das Testwerkzeug.

Graja ist derzeit noch nicht für internationalisierte Aufgaben vorbereitet. Es müssen u. a. folgende Probleme gelöst werden:

- Standardisierung von Schnittstellen einer Aufgabe bzgl. mehrsprachiger Texte (Aufgabentext, Eingabe-/Ausgabedaten, Bewertungskommentare).
- Standardisierung von Schnittstellen einer Aufgabe bzgl. Datums-, Zeit- und Zahlenformaten (Eingabe-/Ausgabedaten, Bewertungskommentare).

Internationalisierung ist ein weites Feld und reicht von der Verwendung von Schrift und Schreibrichtung bis hin zur Sortierung und zur Normalisierung äquivalenter Texte [13]. Für einen vernünftigen Einsatz des Autobewerters in einem internationalen Umfeld müssen vermutlich nicht alle üblicherweise im Rahmen der Internationalisierung zu lösenden Probleme auf einmal angegangen werden.

## 3.11 Entwicklung und Auslieferung von Aufgaben unterstützen

Anforderung: Die Entwicklung sowie die auslieferungsfertige Bündelung aller für eine Aufgabe benötigten Ressourcen muss unterstützt werden. Stakeholder: Aufgabenautoren.

Der Aufbau einer Entwicklungsumgebung für Softwaretests in der professionellen Softwareentwicklung wird von Spezialisten durchgeführt und ist häufig ein manueller Prozess. Es bestehen weder Anforderungen an einen leichten Austausch des SUT noch an einen leichten Austausch des Testtreibers. Im Lernszenario hingegen muss die Entwicklungsumgebung des Aufgabenautors viele SUT unterstützen (eines pro Einreichung). Für jede Programmieraufgabe muss die Entwicklungsumgebung zudem einen Arbeitsbereich für einen eigenen Testtreiber unterstützen.

Professionelle Softwaretests werden häufig auf Entwicklerrechnern und/oder auf einem Integrationsrechner ausgeführt. Die Ausführungsumgebung wird von Profis in der Regel manuell installiert und konfiguriert. Es bestehen häufig keine Anforderungen an eine einfache Installation der Entwicklungsumgebung[15] oder der Betriebsumgebung[16] durch Laien auf

[15] Ausnahme: Projekte mit vielen Entwicklern bereiten normalerweise eine Standard-Entwicklungsumgebung für jeden Entwickler vor, die dann vollautomatisch auf einem neuen Entwicklerrechner installiert werden kann.
[16] Ausnahme: Falls Testläufe sehr lange dauern, werden für Parallelläufe manchmal mehrere Rechner mit der gleichen

vielen Rechnern. Im Autobewertungsszenario muss der Aufgabenautor die Ergebnisse seiner Arbeit so bündeln, dass der eigentliche Bewertungsdurchlauf auf einer beliebigen Autobewerter-Installation stattfinden kann. Der Aufgabenautor kennt die Betriebsumgebung häufig nicht einmal, da die Aufgabe von einer Lehrkraft eingesetzt wird, die häufig nicht gleichzeitig Aufgabenautor ist (vgl. Kapitel 2). Die auslieferungsreife Bündelung aller vom Aufgabenautor erstellter und verwendeter Artefakte zu einer installierbaren Aufgaben-Datei (z. B. einer Zip-Datei) ist somit eine Anforderung an einen Autobewerter, nicht jedoch an ein professionelles Softwaretestwerkzeug.

Zu Graja gehört ein vorgefertigtes Projekt für die Softwareentwicklungsumgebung eclipse[17], welches zur Entwicklung von Aufgaben genutzt werden kann. Teil dieses Projekts sind Ant-Buildskripte zur Erstellung auslieferungsfähiger Aufgaben in Gestalt einer *task.zip*-Datei (vgl. Abb. 1). Im selben Projekt kann der Aufgabenautor Beispieleinreichungen programmieren und daraus automatisiert mehrere *submission.zip*-Dateien schnüren, die für einen anschließenden Test der Aufgabe herangezogen werden können. Weiterhin gehört ein graphischer Client zu Graja, der für Aufgabenautoren den Test neu entwickelter Aufgaben vereinfacht.

## 3.12 Anpassung von Aufgaben durch Lehrkräfte

Anforderung: Eine einmal erstellte Aufgabe wird u. U. in vielen verschiedenen Kontexten von verschiedenen Lehrkräften eingesetzt. Dazu muss die Aufgabe bzgl. mehrerer Aspekte anpassbar sein: Maximalpunktzahl, Gewichtung von Bewertungsaspekten, Benennung von Packages, Klassen und Methoden im studentischen Programm, Formulierung von Aufgabentext und Bewertungskommentaren, Format von Ein-/Ausgabedaten. Stakeholder: Lehrkräfte.

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 erfolgt mit den gleichen Mitteln wie die Ersterstellung des Testtreibers und ist nichts anderes als eine Softwareentwicklungstätigkeit. 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.

Graja trennt eine Aufgabe in zwei verschiedene Gruppen von Dateien auf, von denen in Abb. 1 je ein Vertreter skizziert ist: eine *task.xml*-Datei enthält u. a. den Aufgabentitel und die maximal erreichbare Punktzahl der Aufgabe; im JUnit-Testtreiber *Grader.jar* werden viele weitere Festlegungen codiert (Gewichtung der einzelnen Bewertungsaspekte, Texte, etc.). Abb. 1 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. Es ist geplant, vermehrt Parameter aus dem Wartungsbereich (JUnit-Testtreiber) in den Anpassungsbereich (XML-Datei) zu verlagern, um die Anzahl möglicher Einsatzszenarien einer Graja-Aufgabe durch leichte Anpassung der Parameter zu vergrößern.

## 3.13 Kontextabhängige Testläufe, Bestrafung und Belohnung

Anforderung: Reicht ein Student sein Programm zu einer Aufgabe mehrfach ein, muss der Autobewerter abhängig von der Zahl der Einreichungen, vom Einreichungstermin, von der

Betriebsumgebung ausgestattet. In Projekten, in denen dies häufig und mit variierendem Parallelisierungsgrad geschieht, wird die Betriebsumgebung sinnvollerweise vollautomatisch auf dynamisch in einer Cloud erzeugte virtuelle Maschinen installiert.

[17] http://www.eclipse.org/

Anzahl wiederholt gemachter Fehler, etc. Bestrafungen oder Belohnungen in die Bewertung aufnehmen. Stakeholder: Aufgabenautor und Lehrkraft.

Im professionellen Softwaretest ist es wichtig, dass jeder Testlauf unter gleichen Voraussetzungen, d. h. möglichst frei von Kontextabhängigkeiten durchgeführt wird, um Effekte von zwischen den Testläufen durchgeführten Maßnahmen zuverlässig bewerten zu können. Manchmal geht dies so weit, dass sogar die Systemzeit des den Testtreiber ausführenden Rechners auf eine vorher definierte Zeit zurück gestellt wird. Im Lernszenario kann es hingegen sinnvoll sein, Kontextinformationen wie zuvor durchgeführte Testläufe, in das Bewertungsergebnis aufzunehmen, um etwa unerwünschtes häufiges „Bewertenlassen ohne nachzudenken“ zu unterbinden [6].

Graja besitzt zur Zeit keine vorbereitete Möglichkeit, zuvor durchgeführte Bewertungsläufe in das Bewertungsergebnis einfließen zu lassen. Zu lösende Probleme sind u. a.:

- Standardisierung von Schnittstellen einer Aufgabe bzgl. vorgegebener Abgabefristen und zugehöriger Belohnungs- und Bestrafungsregeln,
- Standardisierung von Schnittstellen einer Aufgabe bzgl. der Belohnung bzw. Bestrafung abhängig von der Anzahl der bisherigen Einreichungen derselben Person.

### 3.14 Variantengenerierung

Anforderung: Aus einer Aufgabe müssen mit dem Ziel, Plagiate zu verhindern, Varianten generiert werden können. Verschiedene Studierende erhalten je eine Variante der Aufgabe, wobei jede Variante aus einer Aufgabenschablone generiert wurde und von vergleichbarer Schwierigkeit ist. Stakeholder: Lehrkraft.

Auch im professionellen Softwaretest hat man es manchmal mit Varianten des SUT z. B. für verschiedene Zielplattformen (Internetbrowser, mobile Geräte, Betriebssysteme) zu tun. Keine Anforderungen bestehen hierbei i. d. R. an die Generierbarkeit von Varianten und der zugehörigen Testtreiber.

In [14] ist ein Ansatz beschrieben, wie aus einer parametrisierten Programmschablone für Studierende jeweils neue Programminstanzen erstellt werden. Die Aufgabe für die Studierenden besteht darin, das Programmverhalten vorherzusagen und z. B. den Wert einer Variablen am Programmende zu ermitteln („code-execution question“). Dieser Aufgabentyp ist in der Bloom’schen Taxonomy auf der niedrigen Stufe „comprehension“ einzuordnen [15]. Wir zielen hingegen auf eine parametrisierte Spezifikationsschablone, aus der für Studierende jeweils neue Spezifikationsinstanzen erstellt werden. Studierende haben dann die in der Bloom’schen Taxonomie weiter oben unter „application“, „analysis“ oder „synthesis“ einzuordnende Aufgabe, das zur Spezifikationsinstanz passende Programm zu schreiben.

Graja ist zur Zeit nicht für Aufgabenschablonen mit Variantengenerierung vorbereitet. Zu lösende Probleme sind u. a.:

- Standardisierung der für die Variantenbildung geeigneten „Stellen“ einer Aufgabe. Denkbar sind Ein-/Ausgabedaten, Klassen- und Methodennamen.
- Erweiterung von Graja um einen Variantengenerator.

## 4 Zusammenfassung

Im vorliegenden Beitrag haben wir Rollenanalogien zwischen dem Softwaretest und dem e-Assessment aufgezeigt. Es zeigt sich, dass die Anforderungen an Werkzeuge, die sowohl im Softwaretest als auch im e-Assessment genutzt werden, teilweise deutlich voneinander abweichen. Ein Autobewerter, der die Belange von Aufgabenautoren, Lehrkräften und Studierenden berücksichtigt, muss mehr sein, als eine einfache Aneinanderreihung diverser Werkzeuge, wie Compiler, xUnit-Testtreiber und Stilprüfer. Für alle identifizierten Anfor-

derungen haben wir Realisierungsmöglichkeiten am Beispiel des Autobewerters Graja erläutert.

## Literaturverzeichnis


[1] Vujošević-Janičić, M.; Nikolić, M.; Tošić, D.; Kuncak, V.: Software verification and graph similarity for automated evaluation of students' assignments. Information and Software Technology, 55(6), S. 1004-1016, 2013.

[2] ISO/IEC 25010:2011: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.

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

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

[5] 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.

[6] Douce, C.; Livingstone, D.; Orwell, J.: Automatic test-based assessment of programming: A review. Journal on Educational Resources in Computing (JERIC), 5(3), 4, 2005.

[7] Ala-Mutka, K. M.: A survey of automated assessment approaches for programming assignments. Computer science education, 15(2), 83-102, 2005.

[8] 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.

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

[10] Garmann, R.: Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito. In: Priss, U., Striewe, M. (Hrsg.), Proceedings of the First Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2013), Hannover, Germany, CEUR-WS.org, 2013.

[11] Garmann, R.; Heine, F.; Werner, P.: Grappa – die Spinne im Netz der Autobewerter und Lernmanagementsysteme. In: DeLFI, 2015.

[12] Wikipedia: Reflexion (Programmierung), https://de.wikipedia.org/w/index.php?title=Reflexion_%28Programmierung%29&oldid=141731957, Stand: 23.06.2015

[13] Wikipedia: Internationalisierung (Softwareentwicklung), https://de.wikipedia.org/w/index.php?title=Internationalisierung_%28Softwareentwicklung%29&oldid=138367081, Stand: 23.06.2015

[14] Brusilovsky, P.; Sosnovsky, S.: Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK. Journal on Educational Resources in Computing (JERIC), 5(3), 6 S., 2005.

[15] Losada, I. H.; Flores, C. P.; Iturbide, J. É. V.: Pedagogical use of automatic graders. In: Rosson, M. B. (Hrsg.), Advances in Learning Processes, InTech, 2010.