Repository of Hochschule Hannover

Graja - Autobewerter für Java-Programme

Robert Garmann

Suggested citation:

Garmann, Robert. 2016. “Graja - Autobewerter für Java-Programme.” Hannover: Hochschule Hannover. https://doi.org/10.25968/opus-941.

Abstract

In diesem Bericht wird der Autobewerter Graja für Java-Programme vorgestellt. Wir geben einen Überblick über die unterstützten Bewertungsmethoden sowie die beteiligten Nutzerrollen. Wir gehen auf technische Einzelheiten und Randbedingungen der in Graja eingesetzten Bewertungsmethoden ein und zeigen die Einbindung von Graja in eine technische Gesamtarchitektur. An einem durchgehenden Beispiel stellen wir die Struktur einer Programmieraufgabe sowie die von Graja unterstützten Feedback-Möglichkeiten dar. Informationen zum bisherigen Einsatz des Graders runden den Bericht ab.

Terms of use

CC BY-NC-SA 4.0

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

Graja – Autobewerter für Java-Programme

Robert Garmann[1]

Bericht

31. März 2016

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

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

Zusammenfassung

In diesem Bericht wird der Autobewerter Graja für Java-Programme vorgestellt. Wir geben einen Überblick über die unterstützten Bewertungsmethoden sowie die beteiligten Nutzerrollen. Wir gehen auf technische Einzelheiten und Randbedingungen der in Graja eingesetzten Bewertungsmethoden ein und zeigen die Einbindung von Graja in eine technische Gesamtarchitektur. An einem durchgehenden Beispiel stellen wir die Struktur einer Programmieraufgabe sowie die von Graja unterstützten Feedback-Möglichkeiten dar. Informationen zum bisherigen Einsatz des Graders runden den Bericht ab.

Schlagworte

e-Assessment, computer based assessment, CBA, Grader, Bewertung, Java, Programmierung, Programmieraufgabe, Programmieranfänger, Feedback, ProFormA-Aufgabenformat, formatives Assessment, Testautomation, Autobewerter, Graja

DDC Klassifikation

005 Computerprogrammierung, Programme, Daten

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*

Inhalt

1 Einleitung ... 4
1.1 Was ist „Graja“? ... 4
1.2 Nutzen automatisierter Programmbewertung ... 4
1.3 Überblick über diesen Bericht ... 4
2 Nutzerperspektive ... 5
2.1 Bewertungsmethoden ... 5
2.2 Rollen ... 5
2.3 Feedback-Möglichkeiten ... 5
3 Technische Funktion ... 6
3.1 Besondere Funktionen des dynamischen Softwaretests ... 6
3.2 Besondere Funktionen der statischen Analyse ... 7
3.3 Weitere technische Funktionen ... 8
4 Technische Architektur ... 9
5 Beispielaufgabe und -einreichung ... 10
6 Aufgabenstruktur ... 13
6.1 Aufgabenstruktur aus Sicht des Aufgabenautors ... 13
6.2 Aufgabenstruktur aus Sicht der Lehrperson ... 17
7 Bisherige Einsätze ... 18
7.1 Umfang ... 18
7.2 Evaluierung ... 18
8 Ausblick ... 19
Literaturverzeichnis ... 20

1 Einleitung

1.1 Was ist „Graja“?

Graja ist ein automatischer Bewerter für Java-Programme. Graja ist eine Abkürzung für „*Gra*der for *ja*va programs“. Wir setzen Graja seit einigen Jahren in Einführungs-Lehrveranstaltungen zur Java-Programmierung als ein das Übungsangebot erweiterndes und bereicherndes Element des formativen Assessments ein. Die Ausführungen dieses Berichts basieren auf der Graja-Version 1.5 (März 2016).

Graja ist derzeit auf Anfrage für interessierte Lehrende verfügbar, die über die Graja-Webseite[2] Kontakt mit dem Graja-Projekt aufnehmen können.

1.2 Nutzen automatisierter Programmbewertung

Dieser Bericht konzentriert sich auf die Darstellung der Funktionen des Autobewerters und seine technischen Besonderheiten. Informationen zum Einsatzzweck und didaktische Hintergründe sind z. B. in [7] zu finden. Auf eine Studie zur Evaluierung des Graja-Einsatzes gehen wir in Abschnitt 7.2 ein.

Um das grundsätzliche, unabhängig von der Umsetzung in Graja bestehende Potenzial der automatischen Programmbewertung als Spezialfall des „computer aided assessment“ (CAA) zu belegen, soll hier beispielhaft eine Studie zitiert werden, an der Informatik-Lehrende aus verschiedenen Ländern teilnahmen [2]. Es ging in der umfragebasierten Studie nicht explizit um das Fach Programmieren oder eine spezifische zu erlernende Programmiersprache, sondern um Informatiklehre im Allgemeinen. Die Befragten stimmten in großer Zahl und erwartungsgemäß zu, dass durch CAA der Aufwand manueller Bewertungen reduziert wird und dass durch CAA Feedback ohne Zeitverzögerung zu den Lernenden gelangen kann. Ebenfalls überwiegend Zustimmung erfuhren die Aussagen, dass CAA zu objektiveren Bewertungsergebnissen führt und dass CAA es Lernenden erlaubt, flexibel und in ihrer eigenen Geschwindigkeit zu lernen. Leicht ablehnend standen die Befragten den Aussagen gegenüber, dass CAA zu geringeren Sicherheitsrisiken führt, dass CAA gegenüber manueller Bewertung zu höheren Aufwänden führt, dass die Feedbackqualität durch CAA erhöht wird und dass Studierende mit Behinderungen durch CAA benachteiligt sind. Überwiegend neutral eingestellt waren die Befragten bzgl. der Aussagen, dass CAA Studierende ängstlicher macht bzw. dass CAA geeignet ist, um höherwertige Lernziele im Sinne einer Lernzieltaxonomie (z. B. [1]) zu prüfen. Auffällig ist bzgl. der letzten Aussage, dass Lehrende mit CAA-Erfahrung die Möglichkeiten von CAA, höherwertige Lernziele zu prüfen, für größer erachteten als Befragte mit geringer CAA-Erfahrung.

Viele Einflussfaktoren bei der automatisierten Bewertung von Java-Programmen sind vergleichbar mit den die gesamte Informatik-Lehre betreffenden Erfahrungen und Meinungen des vorstehenden Absatzes.

1.3 Überblick über diesen Bericht

Wir beginnen in Abschnitt 2 mit der Darstellung der Graja-Funktionen aus der Perspektive des Benutzers, wobei wir drei verschiedene Benutzerrollen unterscheiden. Abschnitt 3 widmet sich verschiedenen funktionalen und technischen Details der von Graja angebotenen Bewertungsfunktionen. Die Einbindung von Graja in eine Gesamtarchitektur ist Gegenstand des Abschnitts 4. Anhand einer in Abschnitt 5 vorgestellten Beispielaufgabe zeigen wir in Abschnitt 6, aus welchen Artefakten eine automatisch bewertbare Programmieraufgabe zusammengesetzt ist und wie sich diese an ein konkretes Lehrszenario anpassen lässt. Dabei unterscheiden wir die Sicht des Aufgabenautors von der Sicht der die Aufgabe einsetzenden Lehrperson. Abschnitt 7 geht auf bisherige Graja-Einsätze in der Lehrpraxis

[2] `http://graja.hs-hannover.de`

ein und nennt Einsatzumfang und Erfahrungen. Abschließende Betrachtungen stellen wir in Abschnitt 8 an.

2 Nutzerperspektive

2.1 Bewertungsmethoden

Graja stützt sich zurzeit i. w. auf den Java Compiler, auf den dynamischen Softwaretest durch das Werkzeug JUnit4[3] und auf die statische Analyse des Quellcodes durch das Werkzeug PMD[4]. Dadurch besitzt Graja in erster Linie Stärken bei der Prüfung der funktionalen Korrektheit einer Einreichung sowie bei der Prüfung von Wartbarkeitseigenschaften des studentischen Programms. Jedoch sind die eingesetzten Werkzeuge nicht auf diese Anwendungsbereiche beschränkt. Auch Aspekte der Effizienz, Sicherheit, Zuverlässigkeit und weiterer Qualitätseigenschaften können berücksichtigt werden. Nicht zuletzt kann Graja zu verschiedenen Bewertungsaspekten ein später durchzuführendes menschliches Feedback in das Gesamtfeedback integrieren.

Graja fokussiert ausschließlich auf den Bewertungsprozess und bietet keine Funktionen für üblicherweise in einem Lernmanagementsystem (LMS) implementierte Funktionen wie Kurs- oder Benutzerverwaltung. Auch Plagiatsprüfungen sind nicht in Graja integriert.

2.2 Rollen

Graja betrachtet drei Nutzerrollen (vgl. Tabelle 1): *Aufgabenautoren* erschaffen eine Aufgabe und müssen dazu mit den in Graja eingesetzten Werkzeugen gut umgehen können. Die Erstellung einer Aufgabe aus Sicht des Aufgabenautors ist ein kleines Softwareprojekt mit definierten Anforderungen und deren Umsetzung. *Lehrpersonen* nutzen Aufgaben in ihrem individuellen Lehrkontext. Dazu passen sie Aufgaben an die Lehrsituation an, indem sie von der Aufgabe angebotene Stellschrauben geeignet justieren. Graja bietet standardmäßig Stellschrauben für zu vergebende Punkte und einige Ausgabetexte an, wodurch bereits eine gute Wiederverwendbarkeit einer Aufgabe erreicht wird. *Studierende* schließlich blicken auf eine Aufgabe als einen Dienst, der ihnen Feedback zu einer eingereichten Lösung liefern kann.

Rolle	**Verhalten**
Aufgabenautor	Entwickelt einen Aufgabentext, Bewertungskriterien, JUnit-Testmethoden und PMD-Regeln; entwickelt Musterlösungen (richtige und falsche)
Lehrperson	Setzt eine Aufgabe in einer Lehrveranstaltung ein; adaptiert ggf. einige Parameter wie Aufgabentext oder Bewertungsschema; führt evtl. nachträglich manuelle Bewertung zu Teilaspekten durch
Student	Verwendet eine Aufgabe als Lernobjekt

Tabelle 1: Rollen im Prozess der Aufgabenentwicklung und –nutzung

2.3 Feedback-Möglichkeiten

Das von Graja erzeugte Feedback besteht aus einer erreichten Punktzahl zusammen mit einem Kommentar. Mit Kommentar bezeichnet Graja eine detaillierte Aufschlüsselung und Erläuterung von Teilergebnissen in strukturierter Textform. Überschriften, Tabellen, Auflistungen, Codeausschnitte und Grafiken können, wenn dies für die betrachtete

[3] `http://junit.org`
[4] `http://pmd.github.io`

Programmieraufgabe sinnvoll ist, integriert werden. Sämtliche Bewertungsaspekte können gewichtet werden [4].

Die Compiler-, Test- und Analyseergebnisse werden von Graja aufbereitet, um von Programmieranfängern besser verstanden zu werden. Es wird sowohl eine überblicksartige Zusammenfassung der Ergebnisse in tabellarischer Form generiert als auch Detail-Feedback zu jedem bewerteten Aspekt. Der Client – in der Regel ein LMS – erhält das Feedback wahlweise im HTML-Format oder als einfachen Text[5]. Das LMS kann die gewünschte Detailtiefe des Kommentars in mehreren Stufen vorgeben.

Graja bereitet Kommentare für zwei Zielgruppen auf: Kommentare für einreichende Studierende, kurz S(tudent)-Feedback, und Kommentare für Lehrpersonen, kurz T(eacher)-Feedback. Die beiden Kommentare weichen je nach Aufgabe erheblich voneinander ab. Graja nimmt an, dass ein LMS der Lehrperson Einblick in beide Feedbacks gewährt, während es dem Studenten den Einblick in das T-Feedback verwehrt. Das S-Feedback fokussiert auf die Erläuterung der entdeckten Mängel und vermeidet lange Stacktraces oder Fehlerprotokolle, die teilweise sicherheitssensitive Informationen enthalten könnten. Im T-Feedback hingegen können je nach Programmieraufgabe Musterlösungen enthalten sein, Hinweise für Tutoren zur manuellen Nachbearbeitung, Ablaufprotokolle der eingesetzten Werkzeuge, detaillierte Stacktraces zur Fehleranalyse, etc.

3 Technische Funktion

Graja erwartet grundsätzlich Quellcode als Einreichung. Evtl. zusätzlich eingereichter Bytecode wird ignoriert und stattdessen selbst von Graja durch Aufruf des Compilers generiert.

3.1 Besondere Funktionen des dynamischen Softwaretests

Das Spektrum der von Graja bewertbaren Programme reicht von kleinen Anfängerprogrammen des „Hello world"-Typs bis hin zu komplexen nebenläufigen Anwendungen. Studentische Programme können an den folgenden Schnittstellen kontrolliert und beobachtet werden (vgl. Tabelle 2): Console, Datei, Umgebungsinformation der Laufzeitumgebung, Java-Methode und -Attribut, grafische Ein-/Ausgabe (z. B. Java 2D), ereignisgesteuerte graphische Benutzerschnittstelle (z. B. Swing). Ein X in der Tabelle bedeutet, dass die Schnittstelle im praktischen Lehreinsatz genutzt wird. Das (X) bedeutet, dass die Nutzung einer graphischen Benutzeroberfläche unter Einsatz von Drittbibliotheken wie UISpec4J[6] denkbar ist, bisher jedoch noch nicht erprobt wurde.

Schnittstelle	**unterstützt**
Console (`System.in`, `System.out`)	X
Dateien	X
System properties	X
Default-Lokalisierung, Default-Zeichensatz	X
Java-Methoden-Parameter / -Rückgabewert	X
Attribute / Klassenkonstanten einer studentischen Klasse	X
Java 2D (`Graphics`, `BufferedImage`)	X
Ereignisgesteuerte GUI (z. B. Swing)	(X)

Tabelle 2: Schnittstellen, an denen ein studentisches Programm kontrolliert und beobachtet werden kann.

[5] Derzeit können Bilder und Tabellen nicht in einfachen Text integriert werden. In HTML-Feedback einzufügende, dynamisch vom Grader erzeugte oder statisch vorliegende Bilder werden als Data-URL realisiert.

[6] `https://github.com/UISpec4J`

In der Graja-Bibliothek angebotene Funktionen erleichtern es einem Aufgabenautor, ein studentisches Programm durch Aufruf von dessen Methoden bzw. durch Setzen und Auslesen von dessen Attributen zu steuern und zu beobachten. Da sich die Graja-Bibliothek auf Java reflection abstützt, kann ein Aufgabenautor Bewertungsroutinen spezifizieren ohne eine Musterlösung erstellen zu müssen. Selbstverständlich ist es ebenfalls möglich, Prüfungen durch Vergleiche von Ausgaben der studentischen Einreichung mit Ausgaben einer vom Aufgabenautor erstellten Musterlösung durchzuführen. Graja überlässt hierbei dem Aufgabenautor die didaktische Entscheidung zwischen der Forderung nach exakter Übereinstimmung und der Tolerierung von bestimmten Abweichungen. Für Ungefähr-Vergleiche stellt Graja verschiedene Routinen[7] zur Verfügung, die erwartete und beobachtete Textausgaben vor dem Vergleich bspw. bzgl. Leerraumzeichen und Interpunktion normieren.

Teil der Einreichung	**unterstützt**
Ganzer Quellcode	X
Quellcodefragment	X
Teile des eingereichten Quellcodes isoliert vom Rest des eingereichten Quellcodes	X

Tabelle 3: Granularität, in der eine studentische Einreichung in ihrem Verhalten untersucht werden kann.

Eine studentische Einreichung kann im einfachsten Fall als ganzes Programm inkl. `main`-Methode übersetzt, zur Ausführung gebracht und dabei in seinem Verhalten beobachtet werden (vgl. Tabelle 3). In diesem Modus bietet Graja dem Aufgabenautor viele vorgefertigte Bibliotheksfunktionen, die den Aufruf der `main`-Methode, die Beschickung des laufenden Programms mit Benutzereingaben sowie die anschließende Prüfung der Consolenausgabe erledigen. Das Hauptaugenmerk des Aufgabenautors kann bei diesem Aufgabentyp auf der Erstellung einer Musterlösung liegen und ist daher als erster Aufgabentyp für Aufgabenautoren empfehlenswert, die erstmals Graja-Programmieraufgaben erarbeiten wollen.

Weiterhin gibt es praktische Lehreinsätze von Graja, in denen ein eingereichtes Codefragment, z. B. ein Ausdruck mit bestimmten gewünschten funktionalen Eigenschaften, automatisch in einen vom Aufgabenautor erstellten Coderahmen eingesetzt und zur Ausführung gebracht wird. Darüber hinaus gibt es Einsatzbeispiele, in denen durch Verwendung von Mocking-Bibliotheken Teile der studentischen Einreichung durch Musterlösungen ersetzt werden, um den verbleibenden Teil der Einreichung isoliert zu prüfen und auf diese Weise eine irreführende Kette von Folgefehlermeldungen zu vermeiden. Wir nutzen hierfür eine speziell angepasste Version der Bibliothek mockito[8] [3], der Aufgabenautor kann jedoch auch eine andere, von ihm favorisierte Mocking-Bibliothek einsetzen.

3.2 Besondere Funktionen der statischen Analyse

Graja kann eine beliebige Menge der verfügbaren PMD-Regeln unverändert oder in mit PMD-Bordmitteln parametrierter Form einsetzen, um eine Einreichung zu analysieren. Jede PMD-Regel kann separat gewichtet werden. Auch selbst implementierte PMD-Regeln sind denkbar, wurden bisher jedoch nicht eingesetzt. Die Darstellung aller verfügbarer Regeln und deren Einsetzbarkeit würde den Rahmen der hier beabsichtigten Darstellung sprengen. Beispielhaft seien einige Regeln in Tabelle 4 zusammen mit dem prüfbaren

[7] Diese und einige weitere Routinen hat Graja von Web-CAT (`http://web-cat.org`) geerbt, dem System das an der Hochschule Hannover erstmals für die automatisierte Bewertung von Java-Programmen erprobt wurde. Die Erfahrungen mit Web-CAT waren hinsichtlich der Bewertungsergebnisse gut, jedoch wurde Web-CAT als ungeeignet eingeschätzt, um an hiesige Lernmanagementsysteme angebunden zu werden. Dies führte letztlich zur Graja-Neuentwicklung.

[8] `http://mockito.org`

Bewertungsaspekt genannt. Für detaillierte Beschreibungen der Regeln verweisen wir auf die PMD-Projektwebseite[9].

Noch nicht zum Funktionsumfang von Graja gehört die Möglichkeit, bestimmte Teile des studentischen Codes von der statischen Analyse bzw. von einzelnen Regelprüfungen auszunehmen. Letzteres könnte sinnvoll sein, wenn die studentische Einreichung eine Weiterentwicklung eines von der Lehrperson zur Verfügung gestellten Rumpfprogramms ist.

Aspekt	**Einige PMD-Regeln**
Unnötig aufgeblähter Code	CollapsibleIfStatements, SimplifyBooleanReturns, LogicInversion, AvoidDeeplyNestedIfStmts
Code conventions	VariableNamingConventions, MethodNamingConventions, ClassNamingConventions, FieldDeclarationsShouldBeAtStartOfClass, BooleanGetMethodName
Lesbarer Code	ShortClassName, CommentRequired
Wartbarkeitsmetriken	CouplingBetweenObjects, NPathComplexity
Einhaltung von Schnittstellenverträgen	OverrideBothEqualsAndHashcode
Hinweise auf Funktionsfehler oder mangelnde Robustheit	ReturnEmptyArrayRatherThanNull, EmptyCatchBlock, UseEqualsToCompareStrings, SwitchStmtsShouldHaveDefault, ConstructorCallsOverridableMethod
Effizienzfehler	CloseResource

Tabelle 4: Beispiele der durch PMD-Regeln prüfbaren Aspekte.

3.3 Weitere technische Funktionen

3.3.1 Default-Zeichenkodierung und Default-Lokalisierung zur Laufzeit des studentischen Programms

Graja erlaubt einem Aufgabenautor die Steuerung der Default-Lokalisierung und der Default-Zeichenkodierung der die studentische Einreichung ausführenden Laufzeitumgebung. So können Aufgaben gestellt und bewertet werden, bei denen die bewusste Berücksichtigung von Lokalisierung und Zeichenkodierung ein Bewertungskriterium darstellt.

3.3.2 Bündelung mehrerer Aufgaben und Lösungen

Im Normalfall bewertet Graja eine Einreichung zu genau einer Programmieraufgabe. Im Graja-Jargon heißt eine solche Bewertungsanforderung *single request*. Graja kann andererseits eine Einreichung bewerten, die Lösungen zu mehreren Übungsaufgaben enthält. Im Übungsbetrieb einer Präsenzlehrveranstaltung ist es nicht unüblich, von Woche zu Woche Aufgabenblätter auszugeben, wobei jedes Aufgabenblatt mehrere kleine Programmieraufgaben stellt. Graja ermöglicht die Bewertung einer mehrere Übungsaufgaben abdeckenden Einreichung in einem Durchgang. Ein diesbezüglicher Bewertungsauftrag an Graja heißt *multi request*.

Als studentische Einreichung wird sowohl im Falle eines single request als auch im Falle eines multi request ein Zip-Archiv erwartet. Die interne Struktur des Archivs ist vorgegeben (vgl. Tabelle 5). Studentische Klassen können je nach Aufgabenstellung im unbenannten default Package oder in benannten Packages eingereicht werden.

[9] `https://pmd.github.io/pmd-5.4.1/pmd-java/rules/index.html`

Request	Format	Erwarteter Inhalt
single	Zip	Studentische Quellcodedateien auf der Wurzelebene des Archivs bzw. in direkt darunter angelegten Package-Ordnern.
multi	Zip	Ein einziger Wurzelordner, darunter je Aufgabe ein Unterordner, darunter wie single request. Die Namen der Ordner werden von der Lehrperson vorgegeben.

Tabelle 5: Format einer Einreichung

3.3.3 Transformation und Prüfung des eingereichten Quellcodes

Von Studierenden eingereichte Quelltextdateien müssen nicht in einer ganz bestimmten Zeichenkodierung vorliegen. Durch Einsatz der ausgereiften Bibliothek icu4j[10] detektiert Graja die Zeichenkodierung mit hoher Zuverlässigkeit. Auf der sicheren Seite sind Studierende mit Einreichungen in der Zeichenkodierung UTF-8.

Ein Aufgabenautor kann Einreichungen auf bestimmte Packages beschränken. Nicht konforme Einreichungen weist Graja mit einer Erläuterung ab. Zudem kann Graja gezielt einige eingereichte Klassen ignorieren. Dies wird z. B. genutzt, um an Studierende ausgegebenen Code nicht in der ausgegebenen Form, sondern während des Bewertungsvorgangs in einer speziell für die Bewertung abgewandelten Version einzusetzen.

4 Technische Architektur

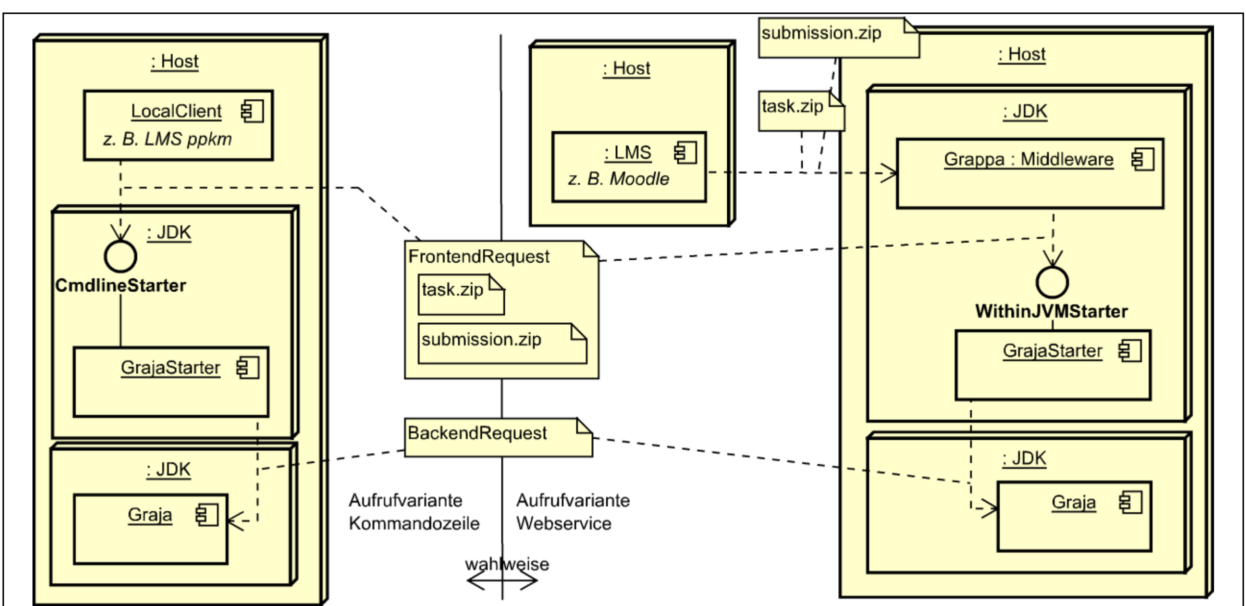

Abbildung 1: Ein LMS kann Graja wahlweise als LocalClient über die Kommandozeile (links) oder über einen von der Middleware Grappa angebotenen Webservice starten (rechts).

Graja ist in Java implementiert und wurde bisher auf verschiedenen Windows- und Linuxsystemen genutzt.

Graja ist als eigenständiges Programm konzipiert, das über klar definierte Schnittstellen an verschiedene LMS angebunden werden kann. Zur Ausführung wird ein Java development kit (JDK[11]) benötigt (vgl. Abbildung 1). Zunächst kann Graja einfach auf der Kommandozeile einer Kommandoshell mit Dateieingaben gestartet werden (Schnittstelle *CmdlineStarter*). Eine Dateieingabe besteht aus einer sog. *FrontendRequest*-Datei, welche die

[10] `http://site.icu-project.org`
[11] Java SE 7 oder höher

Aufgabendatei[12] *task.zip* und die Einreichung *submission.zip* enthält. Die Bewertungsergebnisse stellt Graja nach Abschluss der Bewertung als Ausgabedateien bereit. Auf der Basis der Schnittstelle *CmdlineStarter* wurde Graja in die LMS-Eigenentwicklung ppkm der Hochschule Hannover eingebunden. Außerdem besitzt Graja eine Java-API *WithinJVMStarter*, die zur Einbindung in die Middleware Grappa [5] genutzt wurde, welche wiederum als Webservice in das LMS Moodle[13] integriert wurde [8].

Graja unterstützt Interoperabilität durch die zur Wahl stehenden Schnittstellenformate XML, JSON oder Java-Objekte. XML oder wahlweise JSON werden an der Schnittstelle *CmdlineStarter* erwartet, Java-Objekte an der Schnittstelle *WithinJVMStarter*.

Als Bewertungsergebnis steht ein Aufgabenergebnis[14] zur Verfügung, welches u. a. eine erreichte Punktzahl und ein oder mehrere Base64-kodierte Kommentare im HTML-Format oder in einem einfachen Textformat enthält.

Graja ist intern in zwei Module aufgeteilt. Ein Starter-Modul sorgt dafür, dass alle benötigten Dateien ausgepackt vorliegen, und startet dann das Hauptmodul. Das Hauptmodul wird in einem separaten Prozess gestartet, der dann unter den vom Aufgabenautor vorgegebenen Ressourcen- und Klassenpfadbedingungen ausgeführt wird. Beim Aufruf des Hauptmoduls trifft Graja besondere Vorkehrungen für die Sicherheit des ausführenden Systems und unterbindet unerlaubte Aktionen des studentischen Programmcodes. Dabei stützt sich Graja hauptsächlich auf die Java Sicherheitsarchitektur [3].

Graja überwacht bzw. begrenzt die vom studentischen Programm genutzten Speicherresourcen (flüchtiger und nichtflüchtiger Speicher). Nichtflüchtiger Speicher im Dateisystem wird nur dann begrenzt, wenn das Betriebssystem das Mounten von loop devices[15] unterstützt. Der Bewertungsprozess kann nach einer vom Aufgabenautor oder der Lehrperson vorzugebenden maximalen Zahl von Systemzeit-Sekunden abgebrochen werden, um etwaige Verklemmungen und Endlosschleifen im studentischen Programm aufzulösen.

5 Beispielaufgabe und -einreichung

Description: Write a class `Student` in the package `de.hsh` that stores a name and a matriculation number. Provide a constructor and a `toString` method. Your class should reject illegal, i. e. negative matriculation numbers. Test your class using the following client code:

```
Student s1= new Student("Smith", 68930);
System.out.println(s1); // prints Smith (68930)
Student s2= new Student("Smith", -1); // throws IllegalArgumentException
```

Grading criteria: The program is working functionally correct in the normal case (10 p), it is maintainable (15 p; graded aspects are encapsulation, code conventions, comments, readability) and it proves to be robust against illegal parameters (5 p).

Abbildung 2: Beispielaufgabe mit Bewertungskriterien

Um die Bewertung einer Aufgabe illustrieren zu können, betrachten wir eine kleine Beispielaufgabe (s. Abbildung 2). Die Aufgabe ist ganz bewusst sehr einfach gehalten, bietet aber dennoch Einblick in einige der Bewertungsmöglichkeiten von Graja. Neben dem Aufgabentext zeigt die Abbildung die Bewertungskriterien, die der Aufgabenautor auf Einreichungen anwenden will.

[12] Im Falle eines multi request (vgl. Abschnitt 3.3) sind mehrere Aufgabendateien in einem *FrontendRequest* enthalten. Die Darstellung hier beschränkt sich auf single requests.
[13] `https://moodle.org`
[14] Im Falle eines multi request (vgl. Abschnitt 3.3) stehen mehrere Aufgabenergebnisse zur Verfügung
[15] `http://man7.org/linux/man-pages/man8/mount.8.html\#THE_LOOP\%20DEVICE`

```
package de.hsh;
/**
 * This class can store a student's data.
 */
public class Student {
  String name;
  int matrn;
  /**
   * Creates a student.
   * @param name name of the student
   * @param matrn matriculation number (must not be negative)
   * @exception Exception on invalid parameter
   */
  public Student(String Name, int Matrn) throws Exception {
    if (Matrn < 0) throw new Exception();
    name= Name;
    matrn= Matrn;
  }
  /**
   * @return a string representation
   */
  @Override public String toString() {
    return name+" "+matrn;
  }
}
```

Abbildung 3: Mängelbehaftete Einreichung

Ein Student reiche die in Abbildung 3 dargestellte Datei als Teil einer Zip-Datei ein. Die Abbildung 4 zeigt S(tudent)-Feedback hoher Detailtiefe zu dieser Einreichung. Es ist zu erkennen, dass JUnit, PMD und Mensch als Quelle von Bewertungen auftauchen. Abbildung 5 zeigt S-Feedback zur selben Einreichung in geringer Detailtiefe. Die letztgenannte Darstellung dient dem einreichenden Studenten vor allem zur ersten Orientierung, bevor er sich die Detail-Kommentare ansieht.

Category	Aspect	Source	Result	Achieved	Max.
Functional correctness	Should have a functionally correct constructor and toString method	JUnit	**wrong**	**0.00**	**10.00**
				0.00	*10.00*
Maintainability	Attributes in class Student should be private	JUnit	**wrong**	**0.00**	**4.00**
	Variable naming conventions	PMD	**wrong**	**0.00**	**2.00**
	Fields (attributes) should be at the start of the class	PMD		2.00	2.00
	Comments needed in front of methods and classes	PMD		2.00	2.00
	Code should be readable	Non-automated activity	delayed	0.00	5.00
				4.00	*15.00*
Reliability	Should reject illegal matriculation number	JUnit	**wrong**	**0.00**	**5.00**
				0.00	*5.00*
Total scores				**4.00**	**30.00**

Result details

- Functional correctness. Score: 0.00/10.00
  - *Wrong. Should have a functionally correct constructor and toString method. Score: 0.00/10.00*

    Output of new Student("John", 79205).toString();

    Expected and observed behaviours differ.

	Expected		Observed	
1	John (79205)	<D>	John 79205	1

    ```
    Legend: <D>=difference
    ```
- Maintainability. Score: 4.00/15.00
  - *Wrong. Attributes in class Student should be private. Score: 0.00/4.00*

    There are at least 2 non-private attributes in de.hsh.Student.
  - *Wrong. Variable naming conventions. Score: 0.00/2.00*

    Variables should be named conventionally.
    - Variables should start with a lowercase character, 'Matrn' starts with uppercase character.

      ```
      Student.java
      14 : public Student(String Name, int Matrn) throws Exception {
      ```
    - Variables should start with a lowercase character, 'Name' starts with uppercase character.

      ```
      Student.java
      14 : public Student(String Name, int Matrn) throws Exception {
      ```
  - *Delayed - Non-automated activity. Code should be readable. Score: 0.00/5.00*

    A grading assistant will manually assign scores for readability of your code. - The evaluation and grading of this aspect is a human activity.
- Reliability. Score: 0.00/5.00
  - *Wrong. Should reject illegal matriculation number. Score: 0.00/5.00*

    Your constructor should reject an illegal matriculation number with an IllegalArgumentException (observed: Exception).

Abbildung 4: Beispielfeedback von hoher Detailtiefe für Studierende

Category	Aspect	Result
Functional correctness	Should have a functionally correct constructor and toString method	**wrong**
Maintainability	Attributes in class Student should be private	**wrong**
	Variable naming conventions	**wrong**
	Code should be readable	delayed
Reliability	Should reject illegal matriculation number	**wrong**

Abbildung 5: Beispielfeedback von geringer Detailtiefe für Studierende

6 Aufgabenstruktur

Eine Aufgabe besteht aus Sicht der einzelnen Rollen aus verschiedenen Elementen (vgl. Tabelle 6). In diesem Abschnitt betrachten wie die Sichten des Aufgabenautors und der Lehrperson. Die Sicht der Studierenden ist hauptsächlich durch das LMS vorgegeben, dessen Gestaltung nicht im Einflussbereich von Graja liegt.

Rolle	Charakterisierung
Aufgabenautor (vgl. Abschnitt 6.1)	**Sicht auf eine Aufgabe**: JUnit-Testmethoden, PMD-Regelauswahl, *descriptor.xml*
	Werkzeugeinsatz: Vorkonfiguriertes eclipse-Projekt oder andere Entwicklungsumgebung
Lehrperson (vgl. Abschnitt 6.2)	**Sicht auf eine Aufgabe:** Verteilbare Komponente im ProFormA-Aufgabenformat [6]
	Werkzeugeinsatz: Unzipper, Editor für XML-Dateien, Weboberfläche des LMS zur Konfiguration einer Aufgabe
Student	**Sicht auf eine Aufgabe:** Vom LMS ausgegebener oder auf anderem Wege kommunizierter Aufgabentext; ggf. zugehörige Artefakte wie ein vorgegebener Programmrumpf, sonstige Dateien oder Bibliotheken
	Werkzeugeinsatz: Entwicklungsumgebung zur Erstellung einer Lösung, Weboberfläche des LMS zur Einreichung einer Aufgabenlösung

Tabelle 6: Aufgabenstruktur und Werkzeugeinsatz aus Sicht der verschiedenen Rollen

6.1 Aufgabenstruktur aus Sicht des Aufgabenautors

Graja bietet einem Aufgabenautor ein vorkonfiguriertes eclipse[16]-Projekt an, in dem dieser in der Regel mehrere JUnit-Testmethoden programmiert bzw. PMD-Regeln konfiguriert. Graja bietet dem Aufgabenautor die Möglichkeit, automatisch mit einem gegebenen Buildscript[17] eine verteilbare Komponente im ProFormA-Aufgabenformat [6] zu generieren, die anschließend z. B. über die Weboberfläche eines LMS hochgeladen und dann direkt genutzt werden kann. Graja bietet dem Aufgabenautor alle Freiheiten von JUnit und PMD an, so dass dieser den Prozess der Feedbackerzeugung auf fast unbegrenzte Weise auf die jeweilige Lernsituation einstellen kann. Im Rahmen der bisherigen Einsätze (vgl. Abschnitt 7) wurden sowohl kleine Aufgaben mit einer kleinen einstelligen Anzahl von Bewertungsaspekten realisiert als auch Aufgaben mit über 20 verschiedenen, in Form von JUnit-Testmethoden umgesetzten Bewertungsaspekten. Der Aufgabenautor muss JUnit-Testmethoden in der Regel aufgabenspezifisch implementieren. Die statische Codeanalyse mit PMD-Regeln hingegen kann in der Regel aufgabenübergreifend in wiederverwendbarer

[16] `https://eclipse.org`
[17] Derzeit wird ant (`http://ant.apache.org`) genutzt; in zukünftigen Graja-Versionen wird gradle (`http://gradle.org`) als Buildwerkzeug genutzt werden.

Form spezifiziert werden, indem aus der fast unüberschaubaren Menge existierender Regeln die geeignetsten ausgewählt werden.

In Abbildung 6 ist die oben beschriebene Beispielaufgabe als Softwareentwicklungsprojekt in eclipse dargestellt. Neben einer JUnit-Testklasse *Grader.java* erstellt der Aufgabenautor eine PMD-Regeldatei *ruleset1.xml* sowie optional diverse falsche und korrekte Musterlösungen unterhalb des Ordners *test*. Eine Klammer um all diese Artefakte bildet die Datei *descriptor.xml*.

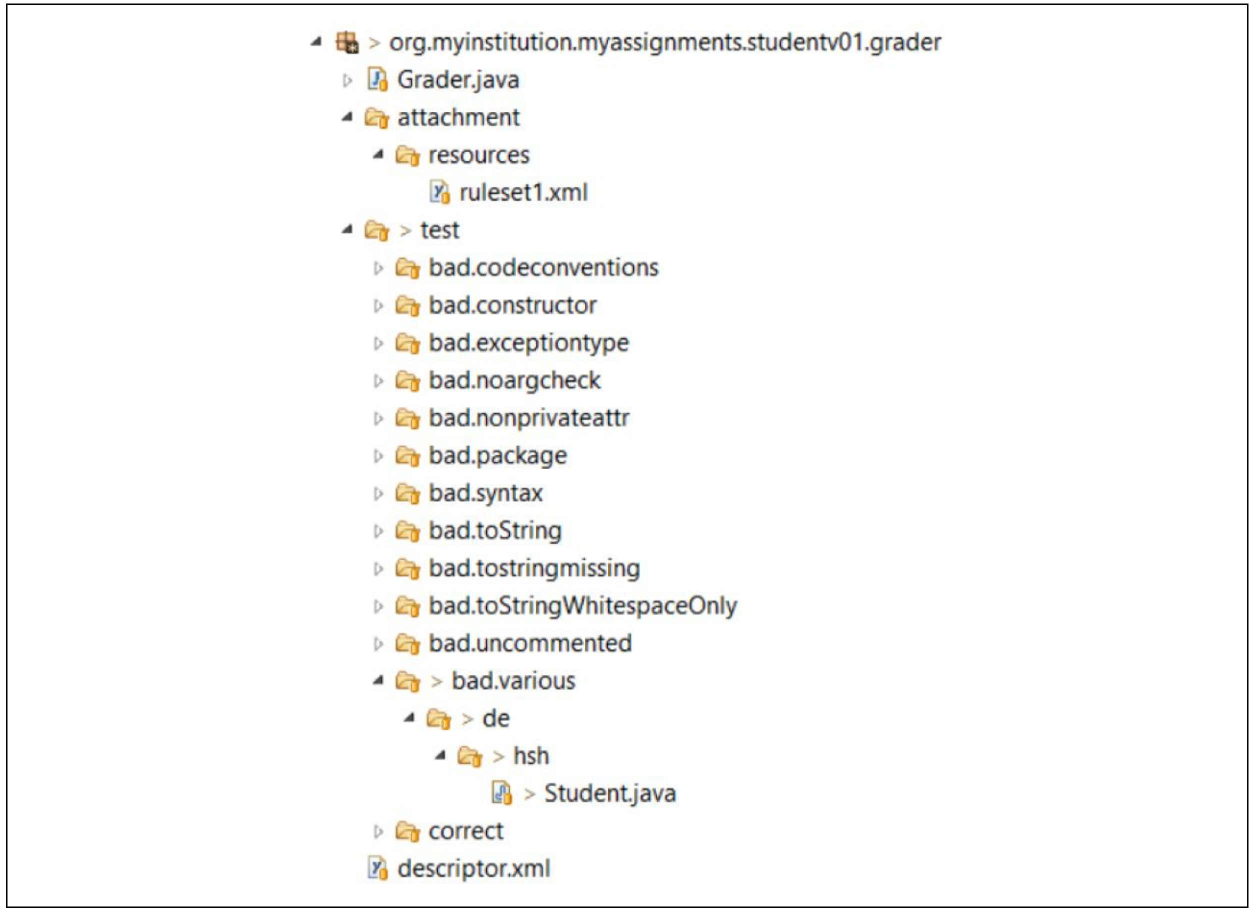

Abbildung 6: Ansicht einer Aufgabe für den Aufgabenautor

Die Datei *descriptor.xml* für unsere Beispielaufgabe ist in Abbildung 7 gekürzt abgedruckt. Wir gehen auf einige Punkte der Datei ein. Mit dem XML-Element *policy* spezifiziert der Aufgabenautor Erlaubnisanforderungen an den Securitymanager der die Bewertung durchführenden Laufzeitumgebung. Im Fall dieser Beispielaufgabe bestehen keine besonderen Erlaubnisanforderungen. Unter *attachments* können weitere Artefakte angegeben werden wie genutzte Drittbibliotheken, Eingabedaten, etc. Die Ausführungszeit und den Ressourcenbedarf beschränkt der Aufgabenautor im *computationResources*-Element. Unter *scores* schließlich wird das Bewertungsschema spezifiziert. Die Datei *descriptor.xml* enthält zu jedem technisch umzusetzenden Bewertungsaspekt Informationen zum Bewertungsgewicht *scoreMax* und zur Beschriftung des Aspekts (*title*). Kategorie-Überschriften (*scoreCategory*) bündeln ein oder mehrere Teilaspekte. Die technische Umsetzung von Bewertungsaspekten lässt sich derzeit via JUnit, PMD und Mensch konfigurieren. Je nach Modul werden unterschiedliche Konfigurationsparameter spezifiziert. Bei JUnit z. B. die betreffende Test-Methode[18], bei PMD der Name der betreffenden Regel. Die Namen der JUnit-Testmethoden bzw. der PMD-Regeln sind Bezeichner, die in den Dateien *Grader.java* und *ruleset1.xml* wieder aufgegriffen und mit einer technischen Umsetzung verknüpft werden.

[18] Die Packageangabe haben wir gekürzt wiedergegeben.

```
<?xml version="1.0" encoding="UTF-8"?>
<descriptor xmlns="...">
  <version>1.0</version>
  <javaVersion>7</javaVersion>
  <name>Student</name>
  <description><![CDATA[
<p>Write a class <code>Student</code> in the package ...]]>
</description>
  <policy>grantgrader {};grantsubmission {};</policy>
  <targetGroup>Computer science, entry level</targetGroup>
  <computationResources>
   <maxRuntimeSecondsWallclockTime>20</maxRuntimeSecondsWallclockTime>
   <maxDiscQuotaKib>500</maxDiscQuotaKib>
   <maxMemMib>64</maxMemMib>
  </computationResources>
  <maxSubmissionSizeBytes>10000</maxSubmissionSizeBytes>
  <attachments>
    <attachment file="attachment/resources/ruleset1.xml"
                type="library" audience="grader"/>
  </attachments>
  <scores>
    <scoreCategory title="Functional correctness">
      <jUnitModule title="JUnit dynamic analysis"
                   graderClassFqn="org.myins....studentv01.grader.Grader">
        <aspect
         title="Should have a functionally correct constructor and toString method"
         testMethod="org.myins....studentv01.grader.Grader#shouldReturnString"
          scoreMax="10.0"/>
      </jUnitModule>
    </scoreCategory>
    <scoreCategory title="Maintainability">
     <pmdModule title="PMD static analysis" rulesetRef="ruleset1.xml">
     <aspect title="Variable naming conventions"
            rule="VariableNamingConventions"
            scoreMax="2"/>
     <aspect .../>
    </pmdModule>
      <jUnitModule title="JUnit dynamic analysis"
                   graderClassFqn="org.myins....studentv01.grader.Grader">
        <aspect title="Attributes in class Student should be private"
                testMethod=.../>
      </jUnitModule>
      <humanModule title="Manually assigned scores">
        <aspect title="Code should be readable" scoreMax="5">
          A grading assistant will manually assign
          scores for readability of your code.
        </aspect>
      </humanModule>
    </scoreCategory>
    <scoreCategory title="Reliability">
      <jUnitModule ...  /jUnitModule>
    </scoreCategory>
  </scores>
</descriptor>
```

Abbildung 7: Gekürzter Inhalt von *descriptor.xml*

Die Dateien *Grader.java* und *ruleset1.xml* enthalten die technische Umsetzung der Bewertungsaspekte. Abbildung 8 definiert drei mit *@Test* annotierte Testmethoden. Alle Methoden setzen Routinen der Graja-Bibliothek ein. Die Methode *setupClass* lädt durch Aufruf von *getPublicClassForName* die studentische Klasse. Sollte diese nicht existieren, weil sie vielleicht vom Studenten falsch benannt wurde, erzeugt *getPublicClassForName* den notwendigen JUnit-Abbruch mit einem Feedback für den Studenten. Die Testmethode *shouldReturnString* demonstriert die Instanziierung der studentischen Klasse, den Aufruf einer Methode sowie den Vergleich von erwartetem und beobachtetem Wert. Es wird ein ungefährer Vergleich spezifiziert, der Unterschiede in Leerraumzeichen ignoriert.

Schließlich werfen wir noch einen Blick auf die Datei *ruleset1.xml* (vgl. gekürzte Darstellung in Abbildung 9). Das Format dieser Datei ist von PMD vorbestimmt. In diesem Fall definiert und parametriert der Aufgabenautor drei Regeln. Der Aufgabenautor kann wie hier genau die genutzten Regeln oder auch eine Obermenge der Regeln definieren, die er in einer konkreten Aufgabe benutzen will. Im letzteren Fall muss die Regeldatei nicht für jede Aufgabe separat erstellt und gepflegt werden.

```
package org.myins....studentv01.grader;  ...

@RestrictSubmissionToPackages("de.hsh")
public class Grader extends AssignmentGrader {

  private static final String VALID_NAME= "John";
  private static final int VALID_MATRN= 79205;
  private static Class<?> submission;

  @BeforeClass public static void setupClass() {
    submission= getPublicClassForName("de.hsh.Student");
  }
  @Test public void attributesShouldBePrivate() {
    Support.assertAllAttributesArePrivateOrClassConstants(submission);
  }
  private String toString(Object student) {
    return ReflectionSupport.invoke(student, String.class, "toString");
  }
  @Test public void shouldRejectIllegalMatrn() {
    final int illegalMatrn= -55;
    try {
      ReflectionSupport.createEx(submission, VALID_NAME, illegalMatrn);
      org.junit.Assert.fail("Your constructor should reject "+illegalMatrn+
                            " as matriculation number");
    } catch (IllegalArgumentException ex) {
    } catch (Exception ex) {
      org.junit.Assert.fail("Your constructor should reject an illegal "+
        "matriculation number with an IllegalArgumentException (observed: "+
        ex.getClass().getSimpleName()+")");
    }
  }
  @Test public void shouldReturnString() {
    Object student= ReflectionSupport.create(submission, VALID_NAME, VALID_MATRN);
    DiffHelper
      .compareStrings(VALID_NAME+" ("+VALID_MATRN+")", toString(student))
      .normalizeOutputExcludedFromDiffSynopsis(
        new StringNormalizer(StandardRule.OPT_IGNORE_ALL_WHITESPACE))
      .includeExplanationInDiffSynopsis(new ParagraphComment(Level.INFO,
        Audience.STUDENT,
        "Output of new Student(\""+VALID_NAME+"\", "+VALID_MATRN+").toString();"))
      .start();
  }
}
```

Abbildung 8: Gekürzter Inhalt von *Grader.java*

```
<?xml version="1.0"?>
<ruleset name="Simple rules" ...>
  <description>A few simple rules</description>
  <rule ref="rulesets/java/naming.xml/VariableNamingConventions">
    <description>Variables should be named conventionally.</description>
  </rule>
  <rule ref="rulesets/java/design.xml/FieldDeclarationsShouldBeAtStartOfClass"/>
  <rule ref="rulesets/java/comments.xml/CommentRequired">
    <properties>
      <property name="fieldCommentRequirement" value="Ignored"/>
      <property name="protectedMethodCommentRequirement" value="Required"/>
      <property name="publicMethodCommentRequirement" value="Required"/>
      <property name="headerCommentRequirement" value="Required"/>
    </properties>
    <description>Leading comments are required before a class and ...</description>
  </rule>
</ruleset>
```

Abbildung 9: Gekürzter Inhalt von *ruleset1.xml*

6.2 Aufgabenstruktur aus Sicht der Lehrperson

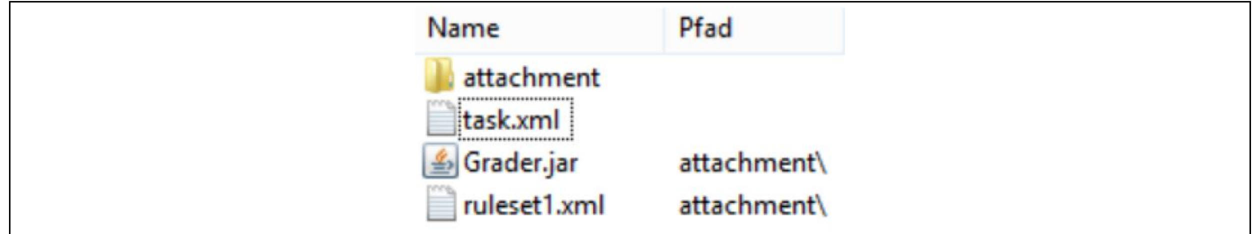

Abbildung 10: Ansicht einer Aufgabe als Zip-Archiv für die Lehrperson

Eine Aufgabe wie die oben beschriebene wird als Zip-Archivdatei im ProFormA-Aufgabenformat verteilt (vgl. Abbildung 10). Eine im Archiv enthaltene Datei *task.xml* enthält die zentralen Einstellungen für die Aufgabe. Im Prinzip sind in *task.xml* die gleichen Informationen enthalten wie in der Datei *descriptor.xml*. Die Struktur der Daten ist lediglich an das Standard-ProFormA-Format angepasst worden. Da es sich um eine XML-Datei handelt, kann eine Lehrperson den Inhalt der Datei mit Standardwerkzeugen einsehen und verändern. So lassen sich z. B. Punktgewichte und Ausgabetexte problemlos anpassen. Neben der Datei *task.xml* enthält das Zip-Archiv eine Datei *Grader.jar*, welche aus dem Quelltext *Grader.java* entstanden ist.

Da das ProFormA-Aufgabenformat ein universelles Format ist, welches für beliebige Autobewerter einsetzbar sein soll, sind in der Datei *task.xml* einige Referenzen und Indirektionen enthalten, die im Original *descriptor.xml* nicht enthalten waren. Dadurch wird die Datei *task.xml* deutlich länger. Da sie im Informationsgehalt kaum von *descriptor.xml* abweicht, verzichten wir auf einen Abdruck des Inhalts der Datei *task.xml*.

Die Sicht auf eine Aufgabe ist für eine Lehrperson stark von dem Funktionsangebot des LMS abhängig. Erlaubt das LMS, mehrere Aufgaben eines Übungsblattes gebündelt an einen angeschlossenen Autobewerter zu versenden? Welche diesbezüglichen Konfigurationsmöglichkeiten stehen der Lehrperson zur Verfügung? Kann die Lehrperson eine Aufgabe direkt im LMS bearbeiten oder editiert sie die *task.xml* mit separaten Werkzeugen? Auch der Blick auf die Bewertungsergebnisse ist stark vom LMS abhängig und soll aus diesem Grunde hier nicht weiter vertieft werden.

7 Bisherige Einsätze

7.1 Umfang

Graja wird seit 2012 fakultätsübergreifend an der Hochschule Hannover in Programmieren-Lehrveranstaltungen in Informatikstudiengängen der Hochschule Hannover eingesetzt. Der Einsatz wurde aus didaktischer und teilweise aus technischer Sicht evaluiert [7].

In den folgenden Lehrveranstaltungen wurde Graja bisher eingesetzt:

- Studiengang Angewandte Informatik, Hochschule Hannover, 1. und 2. Semester, Lehrveranstaltungen „Programmieren 1“ und „Programmieren 2“, nahezu durchgehend seit September 2012 bis heute (März 2016), in jedem Semester zwischen 80 und 120 Studierende. Genutztes LMS: ppkm[19]. Umfang der Nutzung: wöchentliche, über das Semester verteilte Übungsaufgaben; je Semester ca. 32 Aufgaben (1. Semester) bzw. ca. 17 Aufgaben (2. Semester)
- Studiengang Medizinisches Informationsmanagement, Hochschule Hannover, 1. Semester, Lehrveranstaltung „Einführung in die Informatik“, durchgehend einmal jährlich seit September 2013 bis heute (März 2016), in jedem Semester ca. 60-70 Studierende. Genutztes LMS: ppkm[19] (bis 2014), Moodle (seit 2015). Umfang der Nutzung: wöchentliche, über das Semester verteilte Übungsaufgaben; 8 Übungen mit 16 Aufgaben.

In den bisherigen Einsätzen wurden ca. 50 verschiedene Programmieraufgaben genutzt. Um einen Eindruck von der durch die Aufgaben abgedeckten thematischen Breite zu vermitteln, nennen wir einige Aufgabeninhalte: einfache Consolenaus- und -eingabe, Methoden-Parameter und -Rückgabewerte, Kontrollstrukturen, boolesche Logik, mathematische Problemstellungen, Arrays, Collections, Datei-Ein/-Ausgabe, Rekursion, Klassen, Vererbung, Interfaces, abstrakte Klassen, nebenläufige Programme.

Insgesamt wurde Graja im Zeitraum September 2012 bis März 2016 von ca. 700 an der jeweiligen Lehrveranstaltung teilnehmenden Studierenden mit der Bewertung von ungefähr 18.000 Einreichungen beauftragt. Typische Einreichungen umfassten je 1 bis 3 Aufgaben. Einige Einreichungen wichen deutlich nach oben mit bis zu 13 gebündelt eingereichten Lösungen ab. Geht man von durchschnittlich 2 Aufgabenlösungen je Einreichung aus, wurden insgesamt etwa 36.000 Aufgabenlösungen automatisch bewertet.

7.2 Evaluierung

Im Wintersemester 2012/13 wurde eine Umfrage unter 56 Studierenden eines 1. Semesters im Studiengang „Angewandte Informatik“ der Hochschule Hannover durchgeführt. In einem Blended-Learning-Einsatzszenario mussten die Studierenden 60% aller in Übungsaufgaben erreichbaren Punkte als verpflichtenden Teil der Prüfungsleistung erlangen. Zusammenfassend (vgl. Tabelle 7) beurteilen 78% der Studierenden den Autobewerter Graja als hilfreich. Verglichen mit menschlichen Tutoren wird Graja als schneller, vergleichbar gut beim Entdecken von Fehlern, jedoch nicht als gerechter eingeschätzt. Dem Einsatz in einer Klausur ohne flankierende, durch menschliche Bewerter durchgeführte Überprüfungen, stehen die Befragten dieser Studie mit 73% mehrheitlich ablehnend gegenüber.

Tabelle 8 zeigt Ergebnisse von Kurzevaluationen in weiteren Lehrveranstaltungen. Wenn auch auf dieser Datenbasis nicht gesichert abzuleiten, deutet sich an, dass Studierende von “Bindestrich-Studiengängen” etwas weniger stark vom Feedback des Autobewerters profitieren können.

[19] Eigenentwicklung der Hochschule Hannover

Fragestellung	Ergebnis
Die Meldungen von Graja mussten mir durch andere Personen erklärt werden	Gar nicht (41%), Selten (38%)
Graja hat Fehler angezeigt, die keine waren	Gar nicht (43%), Selten (32%)
Graja hat Lösungen akzeptiert, die falsch waren	Gar nicht (73%), Selten (13%)
Die Überprüfung mittels Graja hat mir beim Verständnis der Aufgaben geholfen	Trifft völlig (7%) bzw. überwiegend (41%) zu
Die Überprüfung mittels Graja hat mir bei der Lösung der Aufgaben geholfen	Trifft völlig (16%) bzw. überwiegend (41%) zu
Die Überprüfung mittels Graja hat mich dazu animiert, meine Fehler genauer zu analysieren	Trifft völlig (25%) bzw. überwiegend (54%) zu
Die Überprüfung mittels Graja hat mich in die Irre geführt	Trifft völlig (2%) bzw. überwiegend (18%) zu
Graja ist beim Auffinden von Fehlern mindestens genauso gut wie ein Mensch	Trifft völlig (5%) bzw. überwiegend (38%) zu
Die schnelle Prüfung von Programmieraufgaben mit Graja ist ein großer Vorteil gegenüber menschlich überprüften Lösungen	Trifft völlig (29%) bzw. überwiegend (50%) zu
Die automatische Überprüfung von Graja ist gerechter als menschliche Überprüfungen	Trifft völlig (4%) bzw. überwiegend (21%) zu
Die Unterstützung durch Graja bei Programmierübungen ist insgesamt hilfreich	Trifft völlig (30%) bzw. überwiegend (48%) zu
Ich kann mir vorstellen, auch eine Klausur mit Hilfe von Graja bewerten zu lassen	Trifft völlig (4%) bzw. überwiegend (23%) zu

Tabelle 7: Evaluierung der Leistungsfähigkeit des Autobewerters Graja im Wintersemester 2012/13 im Vergleich zu den Bewertungsmöglichkeiten eines menschlichen Tutors. Die Ergebnisse sind als Prozentwerte aller Befragten dargestellt und wurden auf einer fünfstufigen (immer, oft, manchmal, selten, gar nicht) bzw. vierstufigen Antwortskala ermittelt (trifft völlig / überwiegend / weniger / überhaupt nicht zu).

Fragestellung	Ergebnisse		
	LV1	LV2	LV3
Wie hilfreich war für Sie das automatisierte Feedback zu Ihren Aufgabenlösungen (1=sehr hilfreich, 5=nicht hilfreich)?	2,0	2,1	2,3
Führte der Einsatz des Programms dazu, dass Sie Ihre Fehler im Quellcode Ihrer Java-Lösung leichter erkennen konnten? (1=ja, 5=nein)?	1,8	2,0	2,7

Tabelle 8: Kurz-Evaluierung des Autobewerters Graja seit 2014 in drei Erstsemester-Lehrveranstaltungen der Hochschule Hannover. LV1: Angewandte Informatik (WS 2014/15, N=72), LV2: Angewandte Informatik (WS 2015/16, N=63), LV3: Medizinisches Informationsmanagement (WS 2015/16, N=56). Ergebnisse sind Mittelwerte der von Studierenden gegebenen Antworten.

8 Ausblick

Die automatisierte Bewertung von Java-Programmen mit Graja ist Gegenstand intensiver Forschungsbemühungen. In näherer Zukunft ist geplant, die Bewertungsmöglichkeiten von Graja u. a. auf folgende Bereiche auszudehnen:

- Offline-Bewertung ohne Zugang zum Internet,
- Internationalisierung von Aufgaben,
- parametrierte Aufgaben als Grundlage einer zufallsbasierten Erzeugung gleichwertiger aber im Detail unterschiedlicher Aufgaben,
- Erweiterung der didaktischen Handlungsspielräume, wie Strafpunkte für mehrfache Einreichungen, Hinzubuchung zusätzlicher Tests gegen Punktabzug, etc.

Literaturverzeichnis

[1] Lorin W. Anderson, David R. Krathwohl, Benjamin Samuel Bloom. A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives. Allyn & Bacon, 2001.

[2] Janet Carter, Kirsti Ala-Mutka, Ursula Fuller, Martin Dick, John English, William Fone, Judy Sheard. "How shall we assess this?" In: ACM SIGCSE Bulletin 35.4 (2003), S. 107–123.

[3] Robert Garmann. "Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito." In: Proceedings of the First Workshop „Automatische Bewertung von Programmieraufgaben" (ABP 2013), Hannover, Germany, October 28, 2013. URL: http://ceurws.org/Vol-1067/abp2013_submission_1.pdf.

[4] Robert Garmann, Peter Fricke und Oliver J. Bott. "Bewertungsaspekte und Tests in Java-Programmieraufgaben für Graja im ProFormA-Aufgabenformat". In: DeLFI – Die 14. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. 2016, akzeptierter, für die Veröffentlichung vorgesehener Beitrag.

[5] Robert Garmann, Felix Heine und Peter Werner. "Grappa – die Spinne im Netz der Autobewerter und Lernmanagementsysteme". In: DeLFI – Die 13. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. LNI 247. GI, 2015, S. 169–181.

[6] Sven Strickroth, Michael Striewe, Oliver Müller, Uta Priss, Sebastian Becker, Oliver Rod, Robert Garmann, Oliver J. Bott, Niels Pinkwart. "ProFormA: An XML-based exchange format for programming tasks". In: eleed 11.1 (2015). ISSN: 1860-7470. URL: http://nbn-resolving.de/urn:nbn:de:0009-5-41389.

[7] Andreas Stöcker, Sebastian Becker, Robert Garmann, Felix Heine, Carsten Kleiner, Oliver J. Bott. "Evaluation automatisierter Programmbewertung bei der Vermittlung der Sprachen Java und SQL mit den Gradern „aSQLg" und „Graja" aus studentischer Perspektive." In: DeLFI – Die 11. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. LNI 218. GI, 2013, S. 233–238.

[8] Andreas Stöcker, Sebastian Becker, Robert Garmann, Felix Heine, Carsten Kleiner, Peter Werner, Sören Grzanna, Oliver J. Bott. "Die Evaluation generischer Einbettung automatisierter Programmbewertung am Beispiel von Moodle und aSQLg." In: DeLFI – Die 12. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. LNI 233. GI, 2014, S. 301–304.