aus

Oliver J. Bott, Peter Fricke, Uta Priss, Michael Striewe (Hrsg.)

Automatisierte Bewertung in der Programmierausbildung

Digitale Medien in der Hochschullehre Band 6
2017, 420 Seiten, br., 42,90 €, ISBN 978-3-8309-3606-0

Waxmann Verlag GmbH
www.waxmann.com info@waxmann.com

11 Der Grader Graja

Robert Garmann

Zusammenfassung
Graja ist ein Autobewerter für Java-Programmieraufgaben. Dieses Kapitel beschreibt die in Graja verfügbaren Funktionen, die technische Architektur, die Aufgabenstruktur sowie Einsatzerfahrungen mit Graja.

11.1 Was ist Graja?

Graja – *Gra*der for *ja*va programs – ist ein automatischer Bewerter für Java-Programme. Die Ausführungen dieses Kapitels basieren auf der Graja-Version 1.5 (März 2016). Graja ist derzeit auf Anfrage für interessierte Lehrende verfügbar, die über die Graja-Webseite[1] Kontakt mit dem Graja-Projekt aufnehmen können.

11.2 Nutzerperspektive

In diesem Abschnitt werden die den Nutzern angebotenen Bewertungs- und Feedbackfunktionen sowie die verschiedenen Nutzerrollen dargestellt.

11.2.1 Bewertungsmethoden

Graja stützt sich zurzeit im Wesentlichen auf den Java Compiler, auf den dynamischen Softwaretest durch das Werkzeug JUnit4[2] und auf die statische Analyse des Quellcodes durch das Werkzeug PMD[3]. 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

1 http://graja.hs-hannover.de
2 http://junit.org
3 http://pmd.github.io

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.

11.2.2 Rollen

Wir betrachten drei Nutzerrollen (vgl. Tabelle 11.1): Aufgabenautoren erschaffen eine Aufgabe und müssen sich dazu mit den in Graja eingesetzten Werkzeugen gut auskennen. Die Erstellung einer Aufgabe ist ein kleines Softwareprojekt mit definierten Anforderungen und deren Umsetzung. Lehrpersonen nutzen Aufgaben

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

Tabelle 11.1: Rollen im Prozess der Aufgabenentwicklung und -nutzung

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.

11.2.3 Feedbackmöglichkeiten

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 Programmieraufgabe sinnvoll ist, integriert werden. Sämtliche Bewertungsaspekte können gewichtet werden [GFB16].

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 Detailfeedback zu jedem bewerteten Aspekt. Der Client – in der Regel ein LMS – erhält das Feedback wahlweise im HTML-Format oder als einfachen Text[4]. 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.

11.3 Technische Funktion

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

4 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.

11.3.1 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: Console, Datei, Umgebungsinformation der Laufzeitumgebung, Java-Methode und -Attribut, grafische Ein-/Ausgabe (z. B. Java 2D). Die vorgenannten Schnittstellen wurden im praktischen Lehreinsatz genutzt. Noch nicht genutzt, aber prinzipiell möglich ist die Kontrolle und Beobachtung eines studentischen Programms an einer ereignisgesteuerten graphischen Benutzerschnittstelle (z. B. Swing). Hierzu wäre der Einsatz von Drittbibliotheken wie UISpec4J[5] erforderlich.

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 Graja auf Java Reflection stützt, kann ein Aufgabenautor Bewertungsroutinen spezifizieren, ohne eine Musterlösung erstellen zu müssen. Selbstverständlich ist es auch 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[6] zur Verfügung, die erwartete und beobachtete Textausgaben vor dem Vergleich beispielsweise bezüglich Leerraumzeichen und Interpunktion normieren.

Eine studentische Einreichung kann im einfachsten Fall als ganzes Programm inklusive `main`-Methode übersetzt, zur Ausführung gebracht und dabei in seinem Verhalten beobachtet werden. Graja unterstützt den Aufgabenautor durch vorgefertigte Lösungen für den `main`-Methodenaufruf, für die Simulation von Benutzereingaben und für das Abfangen und Prüfen der Consolenausgabe. 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

5 https://github.com/UISpec4J

6 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 Autobewertung 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.

und zur Ausführung gebracht wird. Darüber hinaus können 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.

11.3.2 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 11.2 zusammen mit dem prüfbaren Bewertungsaspekt genannt. Für detaillierte Beschreibungen der Regeln verweisen wir auf die PMD-Projektwebseite[7].

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

11.3.3 Weitere technische Funktionen

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

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 üblich, wöchentlich Aufgabenblätter auszugeben, wobei jedes Blatt mehrere kleine Programmieraufgaben stellt. Graja ermöglicht die Bewertung einer mehre-

7 https://pmd.github.io/pmd-5.4.1/pmd-java/rules/index.html

Aspekt	Einige PMD-Regeln
Unnötig aufgeblähter Code	CollapsibleIfStatements, SimplifyBooleanReturns, LogicInversion, AvoidDeeplyNestedIfStmts
Code conventions	MethodNamingConventions, FieldDeclarationsShouldBeAtStartOfClass
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 11.2: Beispiele der durch PMD-Regeln prüfbaren Aspekte.

re 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 (Details: [Gar16]). Studentische Klassen können je nach Aufgabenstellung im unbenannten default Package oder in benannten Packages eingereicht werden.

Von Studierenden eingereichte Quelldateien müssen nicht in einer ganz bestimmten Zeichenkodierung vorliegen. Durch Einsatz der Bibliothek icu4j[8] 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.

8 http://site.icu-project.org

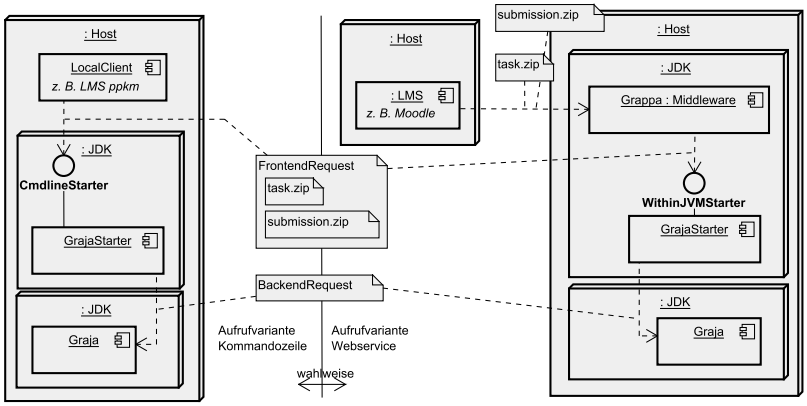

Abbildung 11.1: Ein LMS kann Graja wahlweise als LocalClient über die Kommandozeile (links) oder über einen von der Middleware Grappa angebotenen Webservice starten (rechts).

11.4 Technische Architektur

Graja ist in Java implementiert und wurde bisher auf verschiedenen Windows- und Linuxsystemen genutzt. Graja ist als eigenständiges Programm konzipiert, das über Schnittstellen an verschiedene LMS angebunden werden kann. Zur Ausführung wird ein Java Development Kit (JDK; Java SE 7 oder höher) benötigt (vgl. Abbildung 11.1). Zunächst kann Graja einfach auf der Kommandozeile mit Dateieingaben gestartet werden (Schnittstelle *CmdlineStarter*). Eine Dateieingabe besteht aus einer sogenannten *FrontendRequest*-Datei, welche die Aufgabendatei[9] *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 (vgl. Kapitel 23 und [GHW15]) genutzt wurde, welche wiederum als Webservice beispielhaft in das LMS Moodle integriert wurde (vgl. Kapitel 18 und [Stö+14]). Graja ist interoperabel durch die zur Wahl stehenden Schnittstellenformate XML, JSON oder Java-Objekte. XML bzw. JSON werden an der Schnittstelle *CmdlineStarter* erwartet, Java-Objekte an der Schnittstelle *WithinJVMStarter*.

9 Im Falle eines multi request sind mehrere Aufgabendateien in einem *FrontendRequest* enthalten. Die Darstellung hier beschränkt sich auf single requests.

Als Bewertungsergebnis steht ein Aufgabenergebnis 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 Startermodul sorgt dafür, dass alle benötigten Dateien ausgepackt vorliegen, und startet dann das Hauptmodul. Das Hauptmodul wird in einem separaten Prozess unter den vom Aufgabenautor vorgegebenen Ressourcen- und Klassenpfadbedingungen ausgeführt. 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 vornehmlich auf die Java Sicherheitsarchitektur [Gar13].

Graja 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[10] unterstützt. Der Bewertungsprozess kann nach einer vom Aufgabenautor oder der Lehrperson vorzugebenden maximalen Zahl von Systemzeitsekunden abgebrochen werden, um etwaige Verklemmungen und Endlosschleifen im studentischen Programm aufzulösen.

11.5 Beispielaufgabe und -einreichung

Um die Bewertung einer Aufgabe illustrieren zu können, betrachten wir eine kleine Beispielaufgabe mit einigen Bewertungskriterien. Die Aufgabe ist ganz bewusst sehr einfach gehalten, bietet aber dennoch Einblick in einige der Bewertungsmöglichkeiten von Graja:

> ***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
```

10 http://man7.org/linux/man-pages/man8/mount.8.html#THE_LOOP%20DEVICE

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

Ein Student reiche die folgende Datei als Teil einer Zip-Datei ein:

```
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 no. (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 11.2 zeigt S(tudent)-Feedback in geringer Detailtiefe zu dieser Einreichung. S(tudent)-Feedback hoher Detailtiefe zeigt Abbildung 11.3. Die erstgenannte Darstellung dient den einreichenden Studierenden vor allem zur ersten Orientierung, bevor sie sich die Detail-Kommentare ansehen. Es ist zu erkennen, dass JUnit, PMD und Mensch als Quelle von Bewertungen auftauchen.

11.6 Aufgabenstruktur

Eine Aufgabe besteht aus Sicht der einzelnen Rollen aus verschiedenen Elementen (vgl. Tabelle 11.3). In diesem Abschnitt betrachten wir die Sichtweisen des Aufgabenautors und der Lehrperson. Die Sicht der Studierenden ist hauptsächlich durch das LMS vorgegeben, dessen Gestaltung nicht im Einflussbereich von Graja liegt.

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**

Abbildung 11.2: Beispielfeedback von geringer Detailtiefe für Studierende

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
    10 : public Student(String Name, int Matrn) throws Exception {
    ```

  - Variables should start with a lowercase character, 'Name' starts with uppercase character.

    ```
    Student.java
    10 : 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 11.3: Beispielfeedback von hoher Detailtiefe für Studierende

11.6.1 Aufgabenstruktur aus Sicht des Aufgabenautors

Graja bietet einem Aufgabenautor ein vorkonfiguriertes Eclipse[11]-Projekt an, in dem dieser in der Regel mehrere JUnit-Testmethoden programmiert bzw. PMD-Regeln konfiguriert. Die oben beschriebene Beispielaufgabe enthält eine JUnit-Testklasse *Grader.java*, eine PMD-Regeldatei *ruleset1.xml* sowie optional diverse falsche und korrekte Musterlösungen. Eine Klammer um all diese Artefakte bildet eine Datei *descriptor.xml*. Graja bietet dem Aufgabenautor die Möglichkeit, automatisch mit einem gegebenen Buildscript[12] eine verteilbare Komponente im ProFormA-Aufgabenformat (vgl. Kapitel 24 und [Str+15]) 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 11.7) wurden sowohl kleine Aufgaben mit einer kleinen einstelligen Anzahl von Bewertungsaspekten realisiert als auch Auf-

Rolle	Charakterisierung
Aufgabenautor	**Sicht auf eine Aufgabe:** JUnit-Testmethoden, PMD-Regelauswahl, *descriptor.xml* (vgl. Abschnitt 11.6.1)
	Werkzeugeinsatz: Vorkonfiguriertes eclipse-Projekt oder andere Entwicklungsumgebung
Lehrperson	**Sicht auf eine Aufgabe:** Verteilbare Komponente im ProFormA-Aufgabenformat [Str+15]
	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 11.3: Aufgabenstruktur und Werkzeugeinsatz aus Sicht der verschiedenen Rollen

11 https://eclipse.org
12 Derzeit wird ant (http://ant.apache.org) genutzt; in zukünftigen Graja-Versionen wird gradle (http://gradle.org) als Buildwerkzeug genutzt werden.

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

In der Datei *descriptor.xml* spezifiziert der Aufgabenautor diverse Einstellungen: Erlaubnisanforderungen an den Securitymanager der die Bewertung durchführenden Laufzeitumgebung, Dateianhänge wie genutzte Drittbibliotheken oder Eingabedaten, Beschränkungen der Ausführungszeit und des Ressourcenbedarfs, und schließlich das Bewertungsschema. Zu jedem technisch umzusetzenden Bewertungsaspekt gibt der Aufgabenautor ein Bewertungsgewicht und einen Titel an und gruppiert diese nach frei einzustellenden Kategorien. Außerdem kann man Platzhalter für nachträglich von Menschen durchzuführende Bewertungen in der Datei *descriptor.xml* vorsehen. Mit vom LMS angebotenen Mitteln kann eine Lehrperson später im Feedback enthaltene Platzhalter durch aktuelle Kommentare ersetzen.

Die Dateien *Grader.java* und *ruleset1.xml* enthalten die technische Umsetzung der Bewertungsaspekte. Abbildung 11.4 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 11.5). 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 11.4: 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 11.5: Gekürzter Inhalt von *ruleset1.xml*

11.6.2 Aufgabenstruktur aus Sicht der Lehrperson

Eine Aufgabe wie die oben beschriebene wird als Zip-Archivdatei im ProFormA-Aufgabenformat verteilt. Eine im Archiv enthaltene Datei *task.xml* (ohne Abb.) 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 weitere Dateien, u. a. eine Datei *Grader.jar*, welche aus dem Quelltext *Grader.java* entstanden ist, sowie die Datei *ruleset1.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.

11.7 Bisherige Einsätze

Graja wird seit 2012 fakultätsübergreifend an der Hochschule Hannover in Programmierlehrveranstaltungen in Informatikstudiengängen eingesetzt. Der Einsatz wurde aus didaktischer und teilweise aus technischer Sicht evaluiert [Stö+13].

Graja wurde bisher überwiegend im Studiengang Angewandte Informatik der Hochschule Hannover im 1. und 2. Semester eingesetzt, und zwar nahezu durchgehend seit September 2012 bis heute (März 2016). In jedem Semester nahmen zwischen 80 und 120 Studierende teil. Darüber hinaus wurde Graja im Studiengang Medizinisches Informationsmanagement der Hochschule Hannover im 1. Semester durchgehend jährlich seit September 2013 bis heute eingesetzt. In jedem Semester nahmen hier ca. 60 bis 70 Studierende teil.

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, Methodenparameter 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.

Eine im Wintersemester 2012/13 unter 56 Erstsemestern durchgeführte Umfrage [Stö+13] belegt durch die in Tabelle 11.4 dargestellten Antworten, dass Studierende das Bewertungssystem als hilfreich erachten.

Fragestellung	**++/+**
Die Meldungen von Graja mussten mir durch andere Personen erklärt werden	79
Graja hat Fehler angezeigt, die keine waren	75
Graja hat Lösungen akzeptiert, die falsch waren	86
Die Überprüfung mittels Graja hat mir beim Verständnis der Aufgaben geholfen	48
Die Überprüfung mittels Graja hat mir bei der Lösung der Aufgaben geholfen	57
Die Überprüfung mittels Graja hat mich dazu animiert, meine Fehler genauer zu analysieren	79
Graja ist beim Auffinden von Fehlern mindestens genauso gut wie ein Mensch	43
Die schnelle Prüfung von Programmieraufgaben mit Graja ist ein großer Vorteil gegenüber menschlich überprüften Lösungen	79
Die automatische Überprüfung von Graja ist gerechter als menschliche Überprüfungen	25
Die Unterstützung durch Graja bei Programmierübungen ist insgesamt hilfreich	79
Ich kann mir vorstellen, auch eine Klausur mit Hilfe von Graja bewerten zu lassen	27

Tabelle 11.4: Evaluierung der Leistungsfähigkeit des Autobewerters Graja (Prozent aller Befragten). Fragen 1 bis 3: fünfstufige Antwortskala immer, oft, manchmal, selten (+), gar nicht (++). Fragen 4 ff.: vierstufige Skala trifft völlig (++) / überwiegend (+) / weniger / überhaupt nicht zu.

Tabelle 11.5 zeigt Ergebnisse von Kurzevaluationen im Einsatzszenario „Übung und Tutorium“. Wenn auch auf dieser Datenbasis nicht gesichert abzuleiten, deu-

tet sich an, dass Studierende von „Bindestrich-Studiengängen“ etwas weniger stark vom Feedback des Autobewerters profitieren können.

Fragestellung	**Ergebnisse**		
	L1	L2	L3
Wie hilfreich war für Sie das automatisierte Feedback zu Ihren Aufgabenlösungen (1=sehr / 5=nicht hilfreich)?	2,0	2,1	2,3
Führte der Graja-Einsatz dazu, dass Sie Fehler im Code Ihrer Lösung leichter erkennen konnten? (1=ja, 5=nein)?	1,8	2,0	2,7

Tabelle 11.5: Kurzevaluierung des Autobewerters Graja seit 2014 in drei Erstsemesterlehrveranstaltungen der Hochschule Hannover. L1: Angewandte Informatik (WS 2014/15, N=72), L2: Angewandte Informatik (WS 2015/16, N=63), L3: Med. Informationsmanagement (WS 2015/16, N=56). Ergebnisse sind Mittelwerte der von Studierenden gegebenen Antworten.

11.8 Zusammenfassung

Verglichen mit anderen Autobewertern für Java-Programme ist Graja noch relativ jung, bietet jedoch innovative Ansätze zur Feedbackgestaltung und ist durch seine Schnittstellen sehr gut für die Einbindung in fremde Kursverwaltungswerkzeuge vorbereitet. Da Graja weit verbreitete Softwareentwicklungswerkzeuge wie JUnit und PMD einsetzt, ist der Einarbeitungsaufwand für mit diesen Werkzeugen vertraute Aufgabenautoren relativ klein.

Die automatisierte Bewertung von Java-Programmen mit Graja ist Gegenstand intensiver Forschungsbemühungen. In Zukunft ist geplant, die Bewertungsmöglichkeiten von Graja auszudehnen. So sollen Bewertungen ohne Internetzugang, internationalisierte Aufgaben und parametrierte Aufgaben als Grundlage einer zufallsbasierten Erzeugung gleichwertiger aber im Detail unterschiedlicher Aufgaben ermöglicht werden. Zudem ist geplant, die didaktischen Handlungsspielräume durch Maßnahmen wie Strafpunkte für mehrfache Einreichungen, Hinzubuchung zusätzlicher Tests gegen Punktabzug, etc. zu erweitern.

Literatur für dieses Kapitel

[Gar13] Robert Garmann. „Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito.“ In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. 2013.

[Gar16] Robert Garmann. *Graja – Autobewerter für Java-Programme*. Bericht (SerWisS) 941. Hochschule Hannover, 2016. URL: http://serwiss.bib.hs-hannover.de/frontdoor/index/index/docId/941.

[GFB16] Robert Garmann, Peter Fricke und Oliver Bott. „Bewertungsaspekte und Tests in Java-Programmieraufgaben für Graja im ProFormA-Aufgabenformat“. In: *DeLFI 2016 – Die 14. E-Learning Fachtagung Informatik*. Bd. 262. LNI. GI, 2016, S. 215–220.

[GHW15] Robert Garmann, Felix Heine und Peter Werner. „Grappa – die Spinne im Netz der Autobewerter und Lernmanagementsysteme“. In: *DeLFI 2015 – Die 13. E-Learning Fachtagung Informatik*. Bd. 247. LNI. GI, 2015, S. 169–181.

[Str+15] Sven Strickroth u. a. „ProFormA: An XML-based exchange format for programming tasks“. In: *eleed* 11.1 (2015). URL: https://eleed.campussource.de/archive/11/4138.

[Stö+13] Andreas Stöcker u. a. „Evaluation automatisierter Programmbewertung bei der Vermittlung der Sprachen Java und SQL mit den Gradern *aSQLg* und *Graja* aus studentischer Perspektive.“ In: *DeLFI 2013 – Die 11. E-Learning Fachtagung Informatik*. Bd. 218. LNI. GI, 2013, S. 233–238.

[Stö+14] Andreas Stöcker u. a. „Die Evaluation generischer Einbettung automatisierter Programmbewertung am Beispiel von Moodle und aSQLg.“ In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014, S. 301–304.