Sven Strickroth, Michael Striewe, Oliver Rod (Hrsg.)

Proceedings of the Fourth Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2019)

8. und 9. Oktober 2019
Essen

In Zusammenarbeit mit der Fachgruppe Bildungstechnologien der GI e.V.

Vorwort

Das Lernen von Programmiersprachen ist heute in den Studienordnungen vieler Disziplinen zu finden. Dementsprechend ist die Frage nach der Unterstützung sowohl der Studierenden als auch der Lehrenden noch immer ein aktuelles und spannendes Thema. Die in diesem Bereich aktive Community ist bereits zu drei erfolgreichen Workshops zusammengekommen und hat sich über neue sowie bewährte Ansätze in der Technik, Organisation und der Didaktik ausgetauscht. Speziell die zeitnahe Unterstützung von Studierenden durch eine automatische Bewertung bzw. individuelle Feedbackgabe steht dabei im Mittelpunkt.

Die vom Programmkomitee ausgewählten Beiträge werden in bewährter Art in drei thematisch gruppierten Sessions präsentiert. Ergänzt wird das Programm durch eine Keynote von Prof. Dr. Peter Hubwieser, der aus Sicht eines Fachdidaktikers über Kompetenzen und Kompetenzmodelle zum objektorientierten Programmieren spricht und aus einem aktuellen DFG-Projekt zur automatisierten Messung von Programmier-kompetenzen berichtet. Erstmals enthält das Programm des Workshops in diesem Jahr eine Poster-Session, in der spannende Praxiserfahrungen und neue Ideen präsentiert werden und unter den Teilnehmern besonders intensiv diskutiert werden können.

Abschließend möchten wir uns herzlich bei allen Einreichenden sowie bei den Mitgliedern des Programmkomitees für die Durchführung der Reviews und dem zum Teil sehr ausführlichen Feedback zu den eingereichten Beiträgen bedanken.

Sven Strickroth, Michael Striewe, Oliver Rod
Oktober 2019

Organisation

Oliver Rod (TU Braunschweig)
Sven Strickroth (Universität Potsdam)
Michael Striewe (Universität Duisburg-Essen)

Programmkommitee

Oliver Bott (Hochschule Hannover)
Torsten Brinda (Universität Duisburg-Essen)
Ralf Dörner (RheinMain University of Applied Sciences)
Robert Garmann (Hochschule Hannover)
Rainer Oechsle (Hochschule Trier)
Sven Eric Panitz (RheinMain University of Applied Sciences)
Niels Pinkwart (Humboldt-Universität zu Berlin)
Uta Priss (Ostfalia Hochschule für angewandte Wissenschaften)
Andreas Schwill (Universität Potsdam)
Michael Striewe (Universität Duisburg-Essen)

Inhaltsverzeichnis

Kurzbeiträge

Kompetenzen und Kompetenzmodelle zum objektorientierten Programmieren

Peter Hubwieser[1]

Abstract: Ein wichtiges Verdienst der bahnbrechenden Large-Scale-Untersuchungen TIMMS und PISA war die Entwicklung und Durchsetzung einer neuartigen Forschungsmethodik für Lernprozesse, in deren Mittelpunkt der Kompetenzbegriff (im Sinne von Weinert) und die Messung von Kompetenzen mit Hilfe der Item Response Theory rückte. Ein Nebeneffekt war allerdings dabei die Popularisierung des Begriffs „Kompetenz“, der inzwischen in der Alltagssprache zu einer vagen Bezeichnung für „etwas können“ verkommen ist. International wird die Fachgemeinde der Informatikdidaktik (DDI) vor allem von den US-Verbänden ACM und IEEE dominiert, bei denen der Kompetenzbegriff bei weitem noch nicht die Aufmerksamkeit und Bedeutung erlangt hat wie in Europa. Daher gibt es auf internationaler Ebene bisher nur sehr wenige solide Forschungsergebnisse zu empirisch fundierten Kompetenzen oder -modellen aus der Informatik, insbesondere zum Programmieren. Für unser neues DFG-Projekt AKoFOOP haben wir uns daher zusammen mit der Essener Paluno-Gruppe vorgenommen, einige ausgewählte Kompetenzen aus dem Bereich der objektorientierten Programmierung empirisch zu identifizieren, Instrumente zu deren Messung zu entwickeln und auf der Grundlage dieser Vorarbeiten das Feedback bei der automatischen Programmanalyse durch das JACK System zu verbessern. Die Ergebnisse sollen dann auch in unserem MOOC zur objektorientierten Programmierung (LOOP) genutzt werden. In diesem Vortrag will ich die Grundlagen der Beschreibung und Messung von Kompetenzen vorstellen und die Anwendung dieser Grundlagen im Rahmen unseres Projekts darstellen. Dabei kommt vor allem der Entwicklung von Kompetenzstruktur- und Kompetenzniveaumodellen eine besondere Bedeutung zu.

[1]TU München, peter.hubwieser@tum.de

Automatisierte Bewertung von Java-Programmieraufgaben im Rahmen einer Moodle E-Learning-Plattform

Niels Gandraß[1], Axel Schmolitzky[1]

Abstract: Die Programmiersprache Java wird an zahlreichen Hochschulen gelehrt, um Studierende mit grundlegenden Programmierkonzepten vertraut zu machen. Zur Integration von Online-Java-Programmieraufgaben in ein Moodle LMS wurde ein Fragetyp entwickelt, welcher die parallele Ausführung sowie die automatisierte Bewertung von Quellcode auf Basis von JUnit-Tests ermöglicht. Studierende erhalten hierbei ein sofortiges und individuelles Feedback, welches dynamisch schon während der Bearbeitung einer Aufgabe erzeugt wird. In diesem Beitrag werden sowohl die technischen Details des entwickelten Fragetyps als auch erste Erfahrungen mit seinem Einsatz in der Programmierlehre an der Hochschule für Angewandte Wissenschaften Hamburg thematisiert.

Keywords: automated code evaluation; feedback generation; grading; Java; Java class isolation; JUnit; learning management system; Moodle; signature validation; web service

1 Einleitung

Die an der Hochschule für Angewandte Wissenschaften (HAW) Hamburg entwickelte Lernplattform viaMINT [La18], welche auf dem Learning Management System (LMS) Moodle[2] basiert, ermöglicht sowohl angehenden Studierenden als auch solchen in den ersten Semestern ihr MINT-Wissen aufzufrischen und zu erweitern. Neben interaktivem Lernmaterial aus der Mathematik, Physik und Chemie erhalten Inhalte der Informatik vermehrt Einzug. Den Studierenden werden hierbei unter anderem Programmieraufgaben im Rahmen freiwilliger, studienbegleitender Selbsttests angeboten. Da in der Programmierlehre an vielen Hochschulen die objektorientierte Programmiersprache Java zum Einsatz kommt [Sc17], liegt hier derzeit der Fokus bei der Entwicklung neuer Fragen und Inhalte.

Aufgaben zur Schulung der *Lesekompetenz für Quellcode* sind mit den von Moodle bereitgestellten Fragetypen (u. a. Multiple-Choice, Lückentext, Kurzantwort) bereits gut umsetzbar, indem Quellcodefragmente vorgegeben und Verständnisfragen zu diesen gestellt werden. Neben der Beantwortung solcher als auch theoretischer Fragen zu Programmierkonzepten ist deren praktische Anwendung für den Lernerfolg der Studierenden ebenfalls von großer Bedeutung. Die Fähigkeit, zu einer gegebenen Anforderung passende Java-Quellcodefragmente zu formulieren, bezeichnen wir als *Schreibkompetenz für Quellcode*. Hierbei ist unter anderem

[1] Hochschule für Angewandte Wissenschaften Hamburg, Department Informatik, Berliner Tor 7, 20099 Hamburg, {Niels.Gandrass, Axel.Schmolitzky}@haw-hamburg.de
[2] Moodle LMS Website: https://moodle.org/ (Stand: 03.06.2019)

doi:10.18420/abp2019-1

eine Bewertung der Lösung auf logischer Ebene, welche über den rein textuellen Vergleich hinaus geht, essentiell [Ei03]. Da dies mit den klassischen Moodle-Fragetypen (siehe oben) bisher nicht möglich ist, wurde ein neuer Fragetyp entwickelt. Dieser ermöglicht eine effektive Ausführung, Analyse sowie Bewertung von Java-Quellcodefragmenten. Hierbei erhalten die Studierenden ein sofortiges und interaktives Feedback zu ihrem aktuellen Lösungsansatz und können dies somit noch während der Bearbeitung der Aufgabe direkt mit einbeziehen.

Neben einer Vorstellung des entwickelten Fragetyps wird auf dessen Nutzung auf der viaMINT-Plattform und an der Hochschule für Angewandte Wissenschaften Hamburg, im Rahmen der Programmierlehre für Studierende der ersten zwei Semester, eingegangen.

2 Online-Java-Programmieraufgaben im Moodle LMS

Zur Umsetzung interaktiver Java-Programmieraufgaben wurde ein Moodle-Fragetyp entwickelt, der die Korrektheit einer Lösung mithilfe von Java-Unittests bewertet. Dieser ist anfangs als eine Erweiterung des Plugins `qtype_javaunittest` [Be16] entstanden, wurde jedoch später durch eine komplette Neuentwicklung ersetzt. Nichtsdestotrotz finden einige Konzepte des an der Technischen Universität Berlin entwickelten ursprünglichen Fragetyps hier weiterhin Verwendung.

2.1 Bisherige Softwarelösungen

Neben einer Vielzahl eigenständiger Grader wie beispielsweise dem Praktomat [KSZ02], AuDoscore [OKP17], Graja [Ga16] und auch JACK[3] [GS17] existieren bereits Plugins für das Moodle LMS, welche die Ausführung und Bewertung von Java-Programmen erlauben. Diese können entweder zur Beurteilung eingereichter Komplettlösungen [Ba13; Lü17] oder zur dynamischen Auswertung einzelner Java-Aufgaben [Be16; Öz08] eingesetzt werden. Im Rahmen des hier beschriebenen Szenarios sind die Plugins des letzteren Typs relevant.

Bisherige Entwicklungen in diesem Bereich ermöglichen keine vollständige Realisation der benötigten Funktionalitäten und bieten lediglich eingeschränkte Feedbackmöglichkeiten (siehe Abschnitt 2.2). Ferner werden die bereits vorhandenen Plugins zur Zeit nicht mehr aktiv gepflegt, was zu Kompatibilitätsproblemen mit neuen Moodle-Versionen führt.

2.2 Anforderungen und Zielsetzung

Die bisher verfügbaren Fragetypen bieten bereits eine gute Basisfunktionalität, die weiterhin erhalten bleiben soll. Nichtsdestotrotz werden für den Einsatz auf der viaMINT-Plattform zusätzliche Funktionalitäten benötigt. Diese beinhalten unter anderem:

[3] JACK erlaubt eine Einbindung in Moodle als sogenannte ”Externes Tool”-Aktivität.

- Unterstützung mehrerer virtueller Dateien / Eingabepuffer
- Statische Prüfung der Code-Signatur inklusive generischer Typparameter
- Detaillierteres, auch für Programmieranfänger leicht verständliches Feedback
- Sperren einzelner Quellcode-Abschnitte (engl. *read only*)
- Verbergen einzelner Quellcode-Abschnitte, um Studierende gegen Schnittstellendefinitionen (z. B. generiertes Javadoc[4]) programmieren zu lassen (engl. *black box*)
- Reduktion der Auswertungszeit sowie parallele Bewertung mehrerer Aufgaben
- Vollständige Isolation der ausgeführten Aufgaben
- Zugriff auf Aufgaben-Metadaten (u. a. Quellcode) aus der Testklasse heraus

2.3 Aufbau des Fragetyps

Der entwickelte Fragetyp besteht aus einem Moodle-Plugin (`qtype_junittest`) sowie einem zusätzlichen Webservice (`JUnitQuestionServer`), welcher die Bewertung eines eingereichten Lösungsvorschlags mithilfe von JUnit[5] durchführt. Ersteres stellt lediglich erweiterte Eingabemöglichkeiten bereit und übernimmt die Erzeugung des Feedbacks. Hierbei kommt CodeMirror[6] als Editor zum Einsatz. Dieser erlaubt die komfortable Eingabe von Quellcode und bietet eine umfangreiche Schnittstelle für Erweiterungen, welche die Realisation der benötigten Funktionalitäten (siehe Abschnitt 2.2) erlaubt.

Studierende können, analog zu den im Praktikum eingesetzten Entwicklungsumgebungen, mithilfe einer Tab-Leiste zwischen mehreren virtuellen Dateien beziehungsweise Eingabepuffern wechseln. Dies ermöglicht die Einhaltung der Java-Konvention, je Datei maximal eine öffentliche Klasse zu definieren, und erhöht zudem die Übersichtlichkeit von Aufgaben, für deren Lösung mehr als eine Klasse benötigt wird. Ferner kann jeder dieser Tabs sowohl Quellcode-Abschnitte (jeweils durch den Studierenden bearbeitbar oder gesperrt) als auch beliebige HTML-Elemente beinhalten. Dies kann beispielsweise dazu genutzt werden, Studierende gegen eine gegebene und mithilfe von Javadoc dokumentierte Schnittstelle programmieren zu lassen, ohne ihnen Einblick in die genauen Implementationsdetails oder sogar den vollständigen Quellcode zu geben.

Die Komponenten des entwickelten Fragetyps sowie der Ablauf einer einzelnen Bewertung sind in Abbildung 1 schematisch dargestellt. Hierbei sendet die Moodle-Instanz über das Plugin eine Anfrage, bestehend aus dem zu testenden Java-Quellcode sowie Metadaten (u. a. die gewünschte maximale Ausführungszeit), an den Webservice, welcher die Bewertung der Lösung durchführt und ein Testergebnis zurückliefert. Basierend auf diesem Ergebnis generiert das Moodle-Plugin anschließend ein Feedback, welches dem Studierenden

[4] Siehe: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html (Stand: 03.06.2019)
[5] Java testing framework JUnit, Website: https://junit.org/ (Stand: 03.06.2019)
[6] CodeMirror Text Editor, Website: https://codemirror.net/ (Stand: 04.06.2019)

präsentiert wird. Die einzelnen Teilschritte sowie die bewerteten Aspekte der eingereichten Java-Programme werden detailliert in Abschnitt 2.4 beschrieben.

Abb. 1: Kommunikation der Komponenten zur Bewertung einer Lösung

Die Auskopplung der Ausführung des Java-Programms ermöglicht einige Effizienzoptimierungen sowie eine einfache horizontale Skalierbarkeit des Systems. Für kleinere Szenarien ist es jedoch ebenfalls möglich, den `JUnitQuestionServer` auf demselben Server zu betreiben, auf welchem sich auch die Moodle-Instanz befindet. Ferner wird es explizit unterstützt, den Webservice, inklusive aller für den Betrieb benötigten Bibliotheken, als eigenständigen Docker[7]-Container zu betreiben.

2.4 Bewertung einer eingereichten Lösung

Die gesamte Analyse sowie Bewertung der eingereichten Lösungen findet im `JUnitQuestionServer` statt. Hierbei handelt es sich um einen Java-Webservice, welcher eine JSON-basierte API für das Moodle-Plugin bereitstellt. Der schematische Aufbau des Servers ist in Abbildung 2 grafisch dargestellt. Eintreffende Anfragen werden in eine Warteschlange eingereiht und von einem verfügbaren Worker-Thread bearbeitet. Die Ausführungszeit einer jeden Anfrage ist begrenzt und überfällige Threads werden konsequent terminiert.

Jeder Thread des Worker-Threadpools bearbeitet genau eine der Anfragen und besitzt jeweils einen eigenen `ClassLoader`[8], in den die kompilierten Klassen geladen werden. Dieser wird stets nach Bearbeitung einer Anfrage geleert. Somit wird eine Isolation der einzelnen parallel bearbeiteten Anfragen erreicht. Hierbei führt jeder Worker folgende Schritte aus:

2.1-2 Kompilieren des Quellcodes der Lösung
2.3 Statische Validierung der Code-Signatur (optional)
2.4-5 Kompilieren der JUnit-Testklasse
2.6 Ausführen des JUnit-Runners in gesicherter Ausführumgebung

[7] Docker Container Platform, Website: https://www.docker.com/ (Stand: 04.06.2019)
[8] Siehe: https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html (Stand: 03.06.2019)

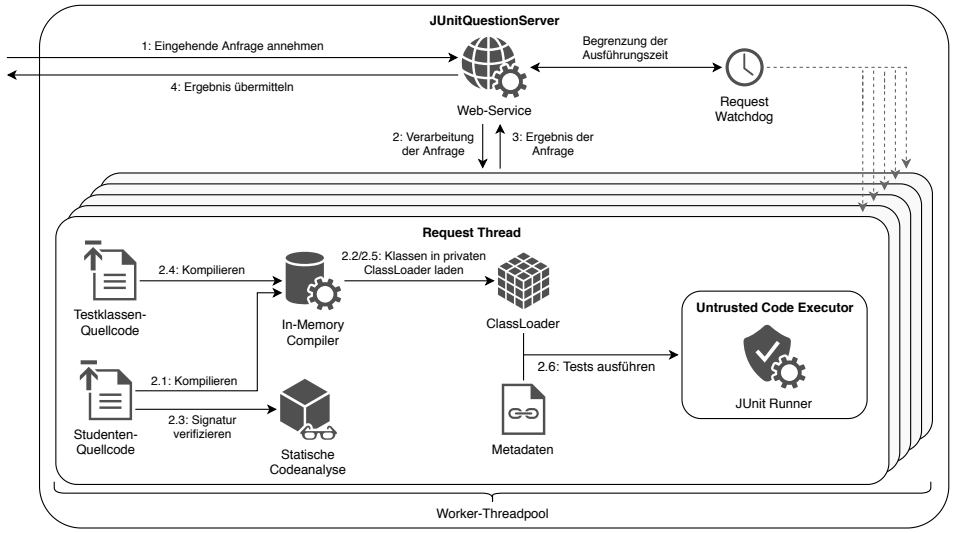

Abb. 2: Architekturdetails des entwickelten JUnitQuestionServers

Im ersten Schritt wird der Inhalt aller Quellcode-Eingabepuffer kompiliert und die hierbei entstandenen Klassen werden geladen. Zur Vermeidung zusätzlicher I/O-Operationen werden alle Quellcode-Teile direkt aus dem Arbeitsspeicher kompiliert, ohne zuvor auf die Festplatte geschrieben zu werden. Kommt es beim Übersetzen zu einem Fehler, so wird dem Studierenden eine Fehlermeldung inklusive Problembeschreibung des Compilers angezeigt.

Um sicherzustellen, dass der eingereichte Quellcode der von der Testklasse erwarteten Schnittstellenspezifikation entspricht, kann optional eine statische Signaturprüfung durchgeführt werden. Hierbei wird geprüft, ob die eingereichte Lösung allen vom Aufgabenersteller spezifizierten Eigenschaften genügt. Die erwartete Signatur entspricht der vom Java Disassembler `javap`[9] generierten Signaturbeschreibung bzw. einer Untermenge dieser und kann somit automatisiert aus der Musterlösung erzeugt werden. Zur Verifikation der Signatur wird mithilfe von ANTLR[10] und einer passenden Java-Grammatik ein abstrakter Syntaxbaum (AST) aus dem Code sowie der erwarteten Signatur erzeugt und verglichen. Dies ermöglicht es, komplexen Fehlermeldungen beim Kompilieren der Testklasse vorzubeugen und stattdessen eine für Programmieranfänger leicht verständliche Fehlermeldung zu generieren. Außerdem erlaubt dieses Vorgehen das Verifizieren generischer Typparameter, welche durch die *Type Erasure*[11] zur Laufzeit nicht mehr geprüft werden können.

Im Anschluss wird die Testklasse kompiliert und ebenfalls in den `ClassLoader` des Threads geladen. Zusammen mit einer Menge an Metadaten (u. a. kompletter Quellcode, Kompilierzeit, Timeouts) werden die Klassen an den JUnit-Runner übergeben und die zugehörige Testklasse

[9] Siehe: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html (Stand: 04.06.2019)
[10] **AN**other **T**ool for **L**anguage **R**ecognition, Website: https://www.antlr.org/ (Stand: 03.06.2019)
[11] Siehe: https://docs.oracle.com/javase/tutorial/java/generics/erasure.html (Stand: 03.06.2019)

ausgeführt. Zur Absicherung des Systems geschieht dies in einem durch den Java-eigenen *Security Manager*[12] abgesicherten Kontext. Hierdurch wird beispielsweise der Zugriff auf System- sowie Netzwerkressourcen unterbunden und ebenfalls die Menge der verfügbaren Programmbibliotheken eingeschränkt. Diese Sicherheitsrichtlinien können vom Serveradministrator nach Bedarf individuell angepasst, sollten jedoch so restriktiv wie möglich gewählt werden.

Ist die Durchführung aller definierten Testfälle abgeschlossen, wird ein Testbericht generiert und von dem Webservice zurück an das Moodle-Plugin übergeben. Hierbei wird zu jedem der Testfälle eine individuelle Fehlermeldung erzeugt, die dem Studierenden anschließend angezeigt wird. Diese kann sowohl eine interne Java-Fehlermeldung inklusive Stacktrace als auch eine extra aufbereitete Meldung mit zusätzlichen Hinweisen sein. Dies erlaubt es dem Aufgabenersteller, eine dem Zielpublikum angemessene Fehlermeldung zu spezifizieren. Gerade Programmieranfängern kann somit ein einfach verständliches Feedback gegeben werden, welches ohne komplexe Java-Interna auf den Grund des Fehlschlagens eines Tests hinweisen kann. Kommt es ferner während der Ausführung einer Testklasse zu einem Timeout, so wird die Ausführung abgebrochen und ein entsprechendes Feedback mit dem Hinweis auf eventuelle Endlosschleifen oder Deadlocks erzeugt.

2.5 Integrationsmöglichkeiten des Webservices

Die bereitgestellte JSON-Schnittstelle des `JUnitQuestionServers` ist agnostisch gegenüber dem eingesetzten Lernmanagementsystem und kann somit potenziell von einer Vielzahl weiterer Softwarelösungen genutzt werden. Für eine Auswertungs-Anfrage an den Webservice werden lediglich die folgenden Daten benötigt: 1) Lösungs-Quellcodefragmente 2) Erwartete Signatur 3) Testklassen-Quellcode 4) Vom Client gewünschter Timeout[13]

Durch die Aufteilung in Plugin- und Serverkomponente können somit Erweiterungen für alternative LMS entwickelt werden. Ferner kann der Webservice so auch zur automatisierten Bewertung von abgegebenen Lösungen im Rahmen von Prüfungsvorleistungen oder Klausuren eingesetzt werden.

3 Einsatz an der HAW Hamburg

An der HAW Hamburg werden bereits seit mehreren Semestern Java-Programmieraufgaben über die Onlineplattform viaMINT angeboten. Die Bearbeitung ist für die Studierenden freiwillig und kann zeitlich unabhängig von laufenden Lehrveranstaltungen erfolgen. Die Aufgaben richten sich an Programmieranfänger der unterschiedlichen Bachelorstudiengänge und thematisieren grundlegende Konzepte der Programmierlehre. Das zur Verfügung

[12] Siehe: https://docs.oracle.com/javase/tutorial/essential/environment/security.html (Stand: 03.06.2019)

[13] Der durch den Client spezifizierte Timeout kann aus Sicherheitsgründen höchstens der im Server hinterlegten maximalen Ausführungszeit entsprechen.

gestellte Angebot beinhaltet sowohl theoretische und *Lesekompetenz*-Fragen als auch praktische Programmieraufgaben zur Schulung der *Schreibkompetenz von Quellcode*.

Letztere behandeln im ersten Semester der zweisemestrigen Programmiergrundausbildung lediglich *Objekt-basierte* [We87] Aspekte von Java. Der Umfang einzelner Aufgaben variiert zwischen einfachen Einzeiler-Lösungen und grundlegenden Interaktionen zweier Objekte beziehungsweise Klassen. Hierbei ermöglicht die vorgelagerte statische Signaturprüfung es, den Studierenden, anstelle komplexer interner Compilerfehler der für sie nicht einsehbaren Testklasse, leicht verständliche Fehlermeldungen zu präsentieren. Besonders Programmieranfänger werden somit vor einigen Stolpersteinen bewahrt, was gerade bei der individuellen Bearbeitung der Aufgaben dem Erhalt des Lernflusses dienlich ist.

Im zweiten Semester der Programmiergrundausbildung werden fortgeschrittene Themen der *Objekt-orientierten* [We87] Programmierung behandelt. Dies beinhaltet unter anderem die Trennung von Typ- und Implementationsvererbung sowie Generizität. Einzelne Aufgaben umfassen nun mindestens eine, meist jedoch mehrere Klassen sowie die sich hieraus ergebenden komplexeren Klassengeflechte inklusive Vererbungsstrukturen. Die Unterstützung mehrerer virtueller Dateien beziehungsweise Eingabepuffer ist für diese Art der Aufgaben unabdingbar. Die optionale Signaturprüfung dient nun weniger dem Abfangen komplexer Fehlermeldungen als der Ermöglichung von Korrektheitstests der generischen Typparameter, welche zur Laufzeit aufgrund der *Type Erasure* nicht zu prüfen sind.

Obwohl die angebotenen Inhalte von den Studierenden freiwillig und außerhalb der Präsenzzeiten bearbeitet werden, konnte eine hohe Beteiligung sowie eine, wie auch schon bei vergleichbaren Angeboten [St14] aufgetretene, überaus positive Resonanz auf das Zusatzangebot festgestellt werden. Gerade auch die vorlesungs- sowie praktikumsbegleitende Nutzung des Onlineangebots wurde von den Studierenden im Rahmen der semesterweise durchgeführten Lehrevaluationen häufig als positiv hervorgehoben.

Momentan beschäftigt sich eine Arbeitsgruppe an der HAW Hamburg mit der Entwicklung weiterführender Programmieraufgaben. Hierbei werden erstmals auch fortgeschrittene Themen der Java-Programmierung wie beispielsweise Generizität oder Nebenläufigkeit im Rahmen von Online-Programmieraufgaben behandelt. Ferner ist ein Einsatz des entwickelten Fragetyps im Kontext von Prüfungsvorleistungen oder E-Klausuren in der Zukunft denkbar.

4 Zusammenfassung und Ausblick

Der vorgestellte Moodle-Fragetyp ist erfolgreich seit mehreren Semestern an der HAW Hamburg im Einsatz und wird seither erweitert. Das entwickelte Plugin erlaubt unter anderem die Umsetzung von Java-Programmieraufgaben im Moodle LMS, welche auch fortgeschrittene Programmierkonzepte thematisieren. Studierende erhalten die Möglichkeit, ein sofortiges und interaktives Feedback auf Basis ihrer aktuellen Lösung einzuholen, noch bevor die Aufgabe abgegeben wird. Hierbei wurden die zur Verfügung stehenden Feedbackmöglichkeiten erweitert, wovon sowohl Programmieranfänger als auch Studierende der

höheren Semester profitieren. Ferner konnte das Abbilden von Klassengeflechten, was gerade bei der objektorientierten Programmierung unabdingbar ist, durch die Einführung mehrerer virtueller Dateien beziehungsweise Eingabepuffer ermöglicht werden. Schlussendlich war es durch die Entwicklung eines optimierten Java-Webservices zur Bewertung der eingereichten Lösungen möglich, die benötigte Auswertungszeit deutlich zu reduzieren.

Potenzielle Erweiterungen des entwickelten Fragetyps beinhalten unter anderem die Integration eines Frameworks zur statischen Codeanalyse. Somit können sowohl weitere Eigenschaften der eingereichten Lösung mit in die Bewertung einfließen als auch den Studierenden ein Feedback zur Qualität des eingereichten Quellcodes gegeben werden. Ferner ist die Integration von Testmöglichkeiten für GUI-Anwendungen ein Themengebiet, welches in späteren Versionen Einzug erhalten kann.

Dieser Beitrag wurde im Rahmen der hochschulübergreifenden MINTFIT E-Assessment Initiative durch die Behörde für Wissenschaft, Forschung und Gleichstellung (BWFG) Hamburg gefördert.

Literatur

[Ba13] Becker, S.; et al.: Prototypische Integration automatisierter Programmbewertung in das LMS Moodle. In: 1. Workshop Autom. Bewertung von Programmieraufgaben. 2013.

[Be16] Bertalan, G.; Rumler, M.: Moodle Fragetyp: javaunittest (`qtype_javaunittest`), Version 2.03, Technischen Universität Berlin, März 2016, URL: `https://moodle.org/plugins/view.php?plugin=qtype_javaunittest`, Stand: 31. 05. 2019.

[Ei03] Eichelberger, H.; Fischer, G.; Grupp, F.; von Gudenberg, J. W.: Programmierausbildung Online. In: DeLFI. 2003.

[Ga16] Garmann, R.: Graja - Autobewerter für Java-Programme, Techn. Ber., 2016, S. 20.

[GS17] Goedicke, M.; Striewe, M.: 10 Jahre automatische Bewertung von Programmieraufgaben mit JACK. In: INFORMATIK 2017. Gesellschaft für Informatik, Bonn, S. 279–283, 2017.

[KSZ02] Krinke, J.; Störzer, M.; Zeller, A.: Web-basierte Programmierpraktika mit Praktomat. Softwaretechnik-Trends 22/3, 2002.

[La18] Landenfeld, K.; Göbbels, M.; Hintze, A.; Priebe, J.: A Customized Learning Environment and Individual Learning in Mathematical Preparation Courses. In: Distance Learning, E-Learning and Blended Learning in Mathematics Education: International Trends in Research and Development. Springer International Publishing, Cham, S. 93–111, 2018.

[Lü17] Lückemeyer, G.: Moodle Plugin: JUnit Exercise Corrector (`assignsubmission_mojec`), Version 1.0, Hochschule für Technik Stuttgart, Jan. 2017, URL: `https://moodle.org/plugins/assignsubmission_mojec`, Stand: 31. 05. 2019.

[OKP17] Oster, N.; Kamp, M.; Philippsen, M.: AuDoscore: Automatic Grading of Java or Scala Homework. In: 3. Workshop Automatische Bewertung von Programmieraufgaben. 2017.

[Öz08] Özcan, S.: Moodle Fragetyp: sojunit (`qtype_sojunit`), Sep. 2008, URL: `https://moodle.org/mod/forum/discuss.php?d=102690`, Stand: 31. 05. 2019.

[Sc17] Schmolitzky, A.: Zahlen, Beobachtungen und Fragen zur Programmierlehre. In: Tagungsband 15. Workshop "Software Engineering im Unterricht der Hochschulen". 2017.

[St14] Stöcker, A.; Becker, S.; Garmann, R.; Heine, F.; Kleiner, C.; Werner, P.; Grzanna, S.; Bott, O. J.: Die Evaluation generischer Einbettung automatisierter Programmbewertung am Beispiel von Moodle und aSQLg. In: DeLFI. 2014.

[We87] Wegner, P.: Dimensions of Object-based Language Design. In: Conference Proceedings. OOPSLA '87, ACM, Orlando, Florida, USA, S. 168–182, 1987.

Trace-Vergleich zur Feedback-Erzeugung im automatisierten E-Assessment-System JACK

Christoph Olbricht[1]

Abstract: An der Universität Duisburg-Essen wird zur automatischen Korrektur von Programmieraufgaben seit mehr als zehn Jahren erfolgreich das E-Assessment-System JACK verwendet. Bisher war das Feedback von JACK auf vordefinierte Hinweise beschränkt, welche in den statischen und dynamischen Tests formuliert wurden. Für die Prüfung der Einsatztauglichkeit von Trace-Vergleichen zur Feedback-Erzeugung in JACK wurden Traces studentischer Lösungen mit Musterlösungen der Aufgaben automatisch verglichen. Hierzu musste zunächst mittels statischer Checks eine passende Musterlösung gewählt werden. Ziel war die Erzeugung eines Feedback-Markers zur Fehlererkennung, um den Studierenden erweitertes Feedback zu bieten. Es wurde festgestellt, dass bestimmte Anforderungen an Länge und Struktur der Aufgaben erfüllt sein müssen, damit der Algorithmus Feedback-Marker erzeugt.

Keywords: E-Assessment, automatische Bewertung, Trace-Vergleich, JACK

1 Einleitung

An der Universität Duisburg-Essen wird zur automatischen Korrektur von Programmieraufgaben seit mehr als zehn Jahren [GS17] erfolgreich das E-Assessment-System JACK verwendet. Die JACK-Architektur, detailliert in [St16] beschrieben, stellt asynchrone Checker-Komponenten bereit. Mittels der asynchronen Checker werden Programmier- und Modellierungsaufgaben ausgewertet. Dem Lehrenden ist es möglich Aufgaben zu erstellen und diese mit statischen und dynamischen Checkern zu versehen. Eingereichte Lösungen werden mittels dieser Checker auf Korrektheit überprüft. Ein statischer Checker überprüft mittels der Anfragensprache GReQL [Gr19] den Quellcode auf vorhandene oder abwesende Programmstrukturen. Es kann sowohl überprüft werden, ob die bestehenden Konventionen der Programmierung eingehalten wurden, als auch, ob unerwünschte Programmkonstrukte verwendet wurden. Ein dynamischer Checker führt die eingereichte Lösung mit mehreren Testfällen aus und prüft das Programm so auf funktionale Korrektheit. Übliche Programmierfehler können mit gezielten Testfällen erkannt werden, so dass dem Studierenden spezifisches Feedback gegeben werden kann. Tritt ein Fehler auf, wird dem Studierenden zudem ein Programm-Trace der Ausführung präsentiert.

Das Feedback dieser Checker ist auf vordefinierte Hinweise beschränkt. Wird ein Fehler durch keinen für diesen Fehlertyp erstellten Testfall abgedeckt, sondern lediglich durch

[1] University of Duisburg-Essen, paluno - The Ruhr Institute for Software Technology, Gerlingstraße 16, 45127 Essen, christoph.olbricht@paluno.uni-due.de

doi:10.18420/abp2019-2

ein falsches Ergebnis erkannt, kann folglich auch nur eine allgemeine Fehlermeldung als Feedback gegeben werden. Dieser Hinweis hilft dem Studierenden nicht bei der Fehlerfindung. Bisher gab es für diese Fälle keine weiteren Hilfestellungen.

Mit Hilfe des Trace-Vergleichs einer studentischen Lösung mit einer passenden Musterlösung des Lehrenden kann erweitertes Feedback geliefert werden. Der Feedback-Marker zeigt die Codezeile auf, ab der die Traces immer stärker voneinander abweichen, da sich in dieser Zeile mit hoher Wahrscheinlichkeit ein Programmierfehler befindet. Somit kann dem Studierenden eine Hilfestellung gegeben werden, um den Fehler zu identifizieren. Eine solche Hervorhebung möglicher Fehler wurde in [Gr17] laut Experten als nützlich angesehen.

Im Rahmen einer Abschlussarbeit sollte die Einsatztauglichkeit der Feedback-Erzeugung dieser Trace-Vergleiche im System JACK geprüft werden. Die Ergebnisse dieser Arbeit werden hier vorgestellt. Es konnten Erkenntnisse über den Aufwand zur Abdeckung des Lösungsraums gewonnen werden. Zudem wurden Anforderungen an die Aufgaben erarbeitet, die erfüllt sein müssen, damit die Feedback-Marker zuverlässig erzeugt werden können.

2 Trace-Vergleich

Traces können vielseitig Verwendung finden. In [ZY18] wurden aus dynamischen Testfällen gewonnene Traces genutzt, um Regeln der Programmierung zu gewinnen. In [Vi03] wurden zum „runtime model checking“ in Java Traces automatisch generiert und analysiert. Zur Verbesserung der Lehre wird in [GH17] die manuelle Erstellung von Traces von Studierenden als Übung zum besseren Verständnis von Programmieraufgaben besprochen. Eine werkzeugunterstützte automatische Korrektur von Programmierfehlern mittels Trace-Vergleich für erweitertes Feedback wird in [Su17] vorgestellt.

Beim Trace-Vergleich zwischen der studentischen Lösung und der Musterlösung des Lehrenden muss zunächst sichergestellt werden, dass beide Lösungen denselben Lösungsansatz besitzen. Weichen die Lösungen zu sehr in ihrem Vorgehen voneinander ab, kann kein sinnvoller Trace-Vergleich stattfinden. Um dieses Problem zu lösen, wurden mehrere Musterlösungen für die vorhandenen Aufgaben erstellt. Mittels statischer Codeprüfung konnte der studentischen Lösung eine ähnliche Musterlösung zugeordnet werden. Eine automatische Zuordnung passender Musterlösungen zu studentischen Lösungen und die daraus resultierenden Herausforderungen wird in [Gr17] besprochen. Die Möglichkeit mittels Syntaxanalyse und Programmtransformation von einzelnen Musterlösungen viele Lösungsvarianten zu erzeugen und diese mit normalisierten studentischen Lösungen zu vergleichen wird in [GJH12] besprochen.

Der vorhandene Algorithmus von Ukkonen [Uk85] vergleicht alle Variablen beider Lösungen miteinander, um eine korrekte Zuordnung der Variablen zu erreichen. Besitzt eine Lösung mehr Variablen, wird die kleinere Variablenmenge betrachtet. Die

Kombination aller Variablen mit der höchsten Übereinstimmung in Variablen-typ und –wert wird für die folgenden Schritte verwendet. Es folgt eine Bewertung jedes einzelnen Schritts der Traces. Sind die gleichen (zuvor zugeordneten) Variablen mit identischem Wert im Schritt beider Traces vorhanden, führt dies zu einer Erhöhung des *Alignment*-Werts. Wird in einem Schritt eine Diskrepanz festgestellt, wird der *Alignment*-Wert reduziert. Eine ausführliche, technische Beschreibung des Verfahrens ist in [St15] nachzulesen.

Ein Schritt, in dem das *Alignment* reduziert wird, wird als *candidate step* vorgemerkt. Sinkt der Wert des *Alignments* für einen Schritt und steigt anschließend wieder, wird von einer weniger performanten Lösung ausgegangen und der *candidate step* verworfen. Sinkt der Wert dauerhaft, muss von einem Fehler in der studentischen Lösung ausgegangen werden und der *candidate step* wird mit einem Feedback-Marker versehen. Das Verhältnis von *candidate steps* zur Länge des Trace wird als *average match* bezeichnet. Der *average match* dient als Kenngröße, ob studentische Lösung und Muster ähnlich zueinander sind. Er kann Werte von 0 bis 1 annehmen und bietet somit eine prozentuale Bewertung der Ähnlichkeit beider Lösungen zueinander. Liegt der Wert unter 0.8, muss von unterschiedlichen Lösungsansätzen ausgegangen werden, womit die Erzeugung eines Feedback-Markers als zufällig anzusehen ist. Bei einem Wert von 1 wurde dagegen kein *candidate step* gefunden, womit auch kein Marker erzeugt werden konnte. Dies lässt jedoch keine Aussage über die Korrektheit des Trace zu, da auch die Möglichkeit besteht, dass keine verarbeitenden Variablen einander zugeordnet wurden und somit ein Fehler nicht erkannt werden konnte.

Da ein *candidate step* im nächsten Schritt bestätigt oder widerlegt werden muss, können bestimmte Fehler nicht durch den Trace-Vergleich aufgezeigt werden. Hierzu gehören Exceptions und Fehler, welche im letzten Schritt des Trace auftreten. Außerdem können Aufgaben, die lediglich logische Abfragen verlangen nicht ausgewertet werden, da der Algorithmus Variablen und ihre Belegung vergleicht. Folglich werden verarbeitende Variablen in den Aufgaben benötigt.

3 Vorgehen

Durch die langjährige Nutzung von JACK standen die anonymisierten Lösungen der Übungsaufgaben aller Studierenden aus den letzten Semestern zur Verfügung. Nach ersten Tests fand eine Bewertung der vorhandenen Aufgaben statt, so dass im Rahmen der Abschlussarbeit vier Aufgaben im Detail ausgewertet werden konnten. Alle weiteren Aufgaben stellten sich leider als unbrauchbar heraus, da sie bestimmte Anforderungen des Algorithmus zur Feedback-Erzeugung nicht erfüllten. Von jedem Studierenden mit fehlerhaften Lösungen wurde eine dieser Lösungen, welche wenigstens ein Drittel der Maximalpunktzahl erreicht hatte zufällig ausgewählt. Lösungen mit weniger Punkten waren für den Trace-Vergleich unbrauchbar, da sie keine ernsthaften Lösungsversuche darstellten und somit keine vergleichbaren Musterlösungen existieren konnten.

Insgesamt wurden 160 studentische Lösungen im Detail ausgewertet. Es lagen keine Musterlösungen zu den bereitgestellten Aufgaben vor, daher wurden diese aus den korrekten Lösungen der Studenten ausgewählt. Mittels der Anfragensprache GReQL [Gr19] wurden Auswahlregeln aufgestellt, um den studentischen Lösungen Musterlösungen mit gleichem Ansatz zuzuordnen.

Für jede Lösung wurde der Wert des *Alignments*, sowie der *average match* notiert und geprüft, ob ein Feedback-Marker erzeugt wurde. Anschließend wurde geprüft, ob für die Lösung ein passendes Muster vorlag, oder für einen sinnvollen Trace-Vergleich ein weiteres Muster mit passendem Ansatz notwendig war. Lag ein erzeugter Marker vor wurde überprüft, ob der Marker die Zeile des Fehlers hervorhob. War dies der Fall wurde der Marker als „korrekt“ erfasst. Abschließend wurden die Fehler der Lösung ausgewertet und notiert.

In der ersten Auswertung wurden alle Lösungen einer Aufgabe mit einer Musterlösung verglichen. Anschließend wurde überprüft, ob mehr als 90% der Lösungen einen *average match* von mindestens 0.8 erreicht hatten und somit der Lösungsraum abgedeckt war. War dies nicht der Fall wurde eine zweite Musterlösung hinzugenommen und eine zweite Auswertung vorgenommen. Dieses Vorgehen sollte fortgesetzt werden, bis der Lösungsraum abgedeckt war, oder durch weitere Musterlösungen keine Verbesserung eintrat. Eine hundertprozentige Abdeckung des Lösungsraums ist aufgrund ungewöhnlicher Fehler und Lösungsansätze mittels Musterlösungen nicht zu erreichen. [RK14]

4 Resultate

Die vier Aufgaben waren in ihrem Aufbau sehr ähnlich. Sie besaßen zwei Methoden und ein Array mit bereitgestellten Daten. Es konnten 182 Fehler verzeichnet werden. Diese Fehler wurden zur besseren Übersicht manuell in Fehlertypen gruppiert. Die Anzahl an unterschiedlichen Fehlertypen reichte von fünf bis zwölf vertretenen Fehlertypen in den einzelnen Aufgaben. Die sieben häufigsten Fehlertypen machten 52 % aller Fehler aus. In 7,5 % der Fälle wurde ein Marker erzeugt.

4.1 Voraussetzungen für Feedback-Marker

Die niedrige Rate, mit der der Algorithmus einen Feedback-Marker erzeugt hat, ließ sich bei der Auswertung der Daten auf bestimmte Anforderungen an die Aufgaben zurückführen. Um einen Feedback-Marker zu erzeugen, müssen wenigstens zwei verarbeitende Variablen im Programmcode vorhanden sein. Dies konnte an zwei Aufgaben nachgewiesen werden. Die Aufgabenstellung forderte einen minimalen bzw. maximalen Wert aus einem Array herauszusuchen. In einer der Aufgaben wurde eine weitere Variable benötigt, um die Position der Zahl im Array festzuhalten. Bei dieser Aufgabe wurden verlässlich Feedback-Marker erzeugt, während die Aufgabe mit nur einer verarbeitenden Variable keine Marker erzeugte.

Weiterhin ist bei einfachen Variablen eine Mindestlänge des Trace von sechs Schritten notwendig, damit ein *Alignment* stattfinden kann. Werden die Daten mittels eines Arrays verarbeitet sind mehr Schritte notwendig, um ein *Alignment* zu erreichen. Der exakte Grenzwert ist davon abhängig, wie häufig und in welcher Art die Daten des Arrays verändert werden. Der Grenzwert ist nicht trivial bestimmbar und konnte im Rahmen der Abschlussarbeit nicht erfasst werden. In einer Aufgabe reichten bereits elf Schritte, während bei einer anderen Aufgabe 42 Schritte notwendig waren.

Letztlich müssen die Testfälle der Aufgaben einzeln erstellt werden, damit sie durch den Trace-Vergleich sinnvoll verarbeitet werden können und für den Studierenden einen Mehrwert darstellen. Werden die Testfälle durch eine Schleife aneinandergereiht und in einem Stück verarbeitet, wird die Ausgabe für den Studierenden unübersichtlich und der Trace-Vergleich liefert keine sinnvollen Ergebnisse.

In einer Aufgabe, die diese Anforderungen erfüllt wurde in 25,8 % der Lösungen ein Marker erzeugt. Wird nur die Methode betrachtet, welche die Anforderungen erfüllte, liefert der Algorithmus in 42 % der Lösungen einen Marker. Zudem waren 100 % der Marker korrekt, haben also die Zeile des vorhandenen Fehlers markiert.

4.2 Lösungsraum der Aufgaben

Die Abdeckung des Lösungsraums einer Aufgabe konnte mit maximal drei Musterlösungen erreicht werden (vgl. Tab. 1). Bei der Aufgabe „Aufgaben mit int Arrays“ wurde keine bessere Abdeckung erreicht, da die Lösungen die notwendige minimale Trace-Länge unterschritten.

Aufgabe	**1 Muster**	**2 Muster**	**3 Muster**
Aufgaben mit int Arrays	62 %	72 %	76 %
Elemente eines Arrays vertauschen	56 %	77 %	93 %
Testat Gruppe 1 WS18	98 %	/	/
Arrays und Schleifen	90 %	/	/

Tab. 1: Lösungsraumabdeckung

Warum bereits drei Muster zur Abdeckung des Lösungsraums ausreichen lässt sich nur vermuten. Zum einen sind die Aufgaben mit dem Ziel erstellt worden, dass sie durch JACK auswertbar und mit wenigen Testfällen überprüfbar sind. Dies führt zu kurzen Methoden, die weniger Raum für starke Abweichungen bieten. Zum andern stammen alle Lösungen von Studierenden, welche dieselbe Vorlesung besucht haben und demnach einen ähnlichen Programmierstil erlernen. In [RK14] wird dies ebenfalls als Grund genannt, warum der Lösungsraum nicht unendlich groß ist. Es ist davon auszugehen, dass diese beiden Faktoren den Lösungsraum eingeschränkt haben.

Ein größerer Lösungsraum führt zu weiterer Arbeit des Lehrenden bei der Erstellung von Aufgaben, da er mehr Musterlösungen mit verschiedenen Ansätzen für den Trace-Vergleich bereitstellen muss. Es erscheint schwierig, dass ein Lehrender den gesamten kreativen Lösungsraum seiner Studenten abdecken kann. Insbesondere, wenn auch fehlerhafte Ansätze bedacht werden müssen.

Dies kann auf zwei Arten abgemildert werden. Zum einen bietet JACK die Möglichkeit mittels statischer Checks den Lösungsraum der Studierenden einzuschränken. Hierbei geht es nicht darum die Kreativität der Studierenden zu beschneiden, sondern unnötige oder sogar kontraproduktive Programmkonstrukte auszuschließen. Somit wird der Lösungsraum eingegrenzt und gleichzeitig den Studierenden eine Hilfestellung gegeben, damit sie sich nicht in falschen Lösungsansätzen verrennen.

Zum anderen kann der Lehrende die Ergebnisse einer Übungsaufgabe in JACK einsehen und Lösungen, für die kein passendes Muster vorlag, auswerten. Ihm stehen drei Möglichkeiten zur Verfügung. Erstens kann die Lösung in die Menge der Musterlösungen aufgenommen werden, womit für alle folgenden Semester ein passendes Muster für diesen Ansatz vorliegt. Zweitens kann ein Testfall für diesen Fehlertypen erstellt werden, damit dem Studierenden eine Hilfestellung gegeben wird. Drittens kann der Lösungsraum wie oben beschrieben eingegrenzt werden, falls sich der Ansatz als unbrauchbar herausstellt.

5 Fazit

Die Einsatztauglichkeit von Trace-Vergleichen zur Feedback-Erzeugung in JACK konnte im Rahmen der Abschlussarbeit nur teilweise bewertet werden. Da die vorhandenen Daten zum Großteil aus Aufgaben stammten, welche für den Trace-Vergleich eine schlechte oder unbrauchbare Struktur aufwiesen, musste eine Differenzierung in der Bewertung vorgenommen werden. Mit den aktuellen Aufgaben lag die Erfolgsrate einen Feedback-Marker zu erzeugen bei 7,5 %. Dies ist zwar ein sehr geringer Prozentsatz, allerdings kann mit wenig Aufwand dafür gesorgt werden, dass der Algorithmus wesentlich bessere Ergebnisse liefert.

Werden die Aufgaben an die Anforderungen für den Trace-Vergleich angepasst und Musterlösungen zur Abdeckung des Lösungsraums erstellt, kann der Trace-Vergleich, obwohl bestimmte Fehlertypen aus technischen Gründen nicht erkannt werden können, eine Erfolgsquote von 42 % vorweisen und damit zur Feedback-Erzeugung sinnvoll eingesetzt werden. Somit wird den Studierenden ohne viel Aufwand eine weitere Hilfestellung zur Fehlererkennung geboten. Weiterhin können die Testfälle angepasst werden. Spezielle kurze Testfälle für spezifische Fehler mit fehlerbezogenen Hinweisen kombiniert mit einem ausführlichen Testfall, um nicht abgedeckte Fehler mittels Trace-Vergleich abzufangen, würden den Studierenden in allen Fällen gute Hinweise auf ihre Fehler liefern.

Mit der Information, dass sieben Fehlertypen für 52 % aller Fehler in den Aufgaben verantwortlich waren, lässt sich darauf schließen, dass bestimmte Programmierkonzepte für die Studierenden schwer zu verstehen sind. Lehrende könnten speziell auf diese Fehler eingehen und dafür Sorge tragen, dass diese Fehler nicht mehr so häufig auftreten.

Hierzu muss JACK um die Möglichkeit erweitert werden, automatisch fehlerhafte Lösungen nach Fehlertypen zu gruppieren. Da gleiche Fehler meistens zu einem gleichen *Alignment*-Wert führen, kann dieser als Anhaltspunkt genutzt werden. Diese gruppierten Fehler können jeweils mit einer Bezeichnung versehen werden. Damit wäre es für den Lehrenden in jedem Semester möglich sich die häufigsten Fehlertypen der Übungsaufgaben anzeigen zu lassen und diese nochmals in der Vorlesung zu behandeln.

Literatur

[GJH12] Gerdes, A.; Jeuring, J.; Heeren, B.: An Interactive Functional Programming Tutor. Annual Conference on Innovation and Technology in Computer Science Education, 2012.

[GH17] Goltermann, R.; Höppner, F.: Internalizing a Viable Mental Model of Program Execution in First Year Programming Courses. ABP, 2017.

[Gr17] Gross, S.; Mokbel, B.; Hammer, B.; Pinkwart, N.: Feedback provision strategies in intelligent tutoring systems based on clustered solution spaces. DeLFI 2012: Die 10. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V.. Gesellschaft für Informatik, Bonn, S. 27-38, 2012.

[Gr19] Graph Repository Query Language (GReQL), https://www.uni-koblenz-landau.de/de/koblenz/fb4/ist/rgebert/research/graph-technology/GReQL, besucht: 2019-06-10.

[GS17] Goedicke, M.; Striewe, M.: 10 Jahre automatische Bewertung von Programmier-aufgaben mit JACK - Rückblick und Ausblick. INFORMATIK 2017. Gesellschaft für Informatik, Bonn, S. 279–283, 2017.

[RK14] Rivers, K.; Koedinger, K.R.: Automating Hint Generation with Solution Space Path Construction. In Proceedings of the 12th International Conference on Intelligent Tutoring Systems, S. 329-339, 2014.

[St15] Striewe, M.: Automated analysis of software artefacts - a use case in e-assessment. Dissertation, 2015.

[St16] Striewe, M.: An architecture for modular grading and feedback generation for complex exercises. Science of Computer Programming 129. Special issue on eLearning Software Architectures, S. 35–47, 2016.

[Su17] Suzuki, R. et.al.: TraceDiff: Debugging unexpected code behavior using trace divergences. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) S. 107-115, 2017.

[Uk85] Ukkonen, E.: Finding approximate patterns in strings. Journal of Algorithms 6.1 S. 132–137, 1985.

[Vi03] Visser, W. et.al.: Model Checking Programs. Automated Software Engineering 10, S. 203-232, 2003.

[ZY18] Zaman, T.S.; Yu, T.: Extracting Implicit Programming Rules: Comparing Static and Dynamic Approaches. In Proceedings of the 7th International Workshop on Software Mining, 2018.

Automatische Bewertung von Haskell-Programmieraufgaben

Marcellus Siegburg[1], Janis Voigtländer[1], Oliver Westphal[1]

Abstract: Wir beschreiben unsere Vorgehensweise bei der Durchführung von Online-Übungen zur Programmierung in Haskell. Der Fokus liegt insbesondere auf dem Zusammenspiel der verwendeten Sprachmittel, Programmbibliotheken und Tools, die es uns durch ihre Kombination erlauben, verschiedene Aspekte des Übungsbetriebs zu automatisieren bzw. zu erleichtern. Unser Ansatz erlaubt uns das automatische Bewerten von Einreichungen zu typischen Programmieraufgaben. Darüber hinaus sind wir in der Lage, Studierende durch geeignet gestaltete Aufgabenstellungen in Richtung bestimmter Lösungen zu führen, und währenddessen Hilfestellung durch entsprechendes Feedback zu geben.

1 Einführung

Wir führen an der Universität Duisburg-Essen eine Lehrveranstaltung „Programmierparadigmen“ durch, hauptsächlich als Pflichtveranstaltung in einem Kerninformatik-Bachelorstudiengang. Schwerpunkt ist funktionale Programmierung mit Haskell, zu einem geringeren Teil logische Programmierung mit Prolog. Wir möchten konkrete Programmierfertigkeiten und Anwendung der in der Vorlesung behandelten Programmiersprachenkonzepte vermitteln, daher haben die begleitenden Übungen einen hohen Anteil an Programmieraufgaben. Zur Unterstützung, der Studierenden wie auch der Tutoren, setzen wir Automatisierung ein. Dabei geht es sowohl um Bewertung der Korrektheit von finalen Einreichungen als auch um Hinweise während der Bearbeitung, zu Fehlern und Verbesserungsmöglichkeiten.

Unser Vorgehen ist gekennzeichnet durch ineinandergreifenden Einsatz verschiedener Systeme und Werkzeuge, die bei geeignet orchestriertem Zusammenspiel unsere Ziele realisieren, und uns gleichzeitig mit relativ wenig Neuentwicklung auf der technischen (statt inhaltlichen) Seite auskommen lassen. Den Studierenden wiederum wird eine Integration präsentiert, damit sie möglichst wenig Brüche wahrnehmen. Tatsächlich werden ihnen von den einzelnen Bestandteilen wohl nur die ersten beiden als eigenständige Entitäten bewusst:

- CodeWorld – eine Online-Lernumgebung zum Erzeugen von vorwiegend Bildern und Animationen, durch mathematische Ausdrücke und Funktionen
- Autotool – ein E-Learning-System zum Erzeugen, Stellen, Einreichen und Bewerten diverser Arten von Übungsaufgaben und Lösungen
- Typ- und Modulsystem von Haskell: Sprachfeatures, die bei Aufgabengestaltung sowie insbesondere Lenkung von Lösungsversuchen in bestimmte Bahnen hilfreich sind

[1] Universität Duisburg-Essen, {marcellus.siegburg, janis.voigtlaender, oliver.westphal}@uni-due.de

doi:10.18420/abp2019-3

- GHC – Haskell-Compiler, mit zusätzlichen Möglichkeiten der Überprüfung von Code über dessen Erfüllen des Sprachstandards hinaus
- diverse Haskell-Bibliotheken – statische Prüfung zum Beispiel eigener Syntaxbedingungen, sowie dynamische Ausführung von Code in abgesicherter Weise
- HLint – Linter für Haskell, mit reicher Regelsprache, konfigurier- und erweiterbar
- QuickCheck – deklaratives Testframework für pure Funktionen, eigenschaftsbasiert
- IOSpec – Möglichkeit des Testens auch effektbehafteten Codes innerhalb der Sprache, durch Reifikation als normale Datenstrukturen

2 Umgebungen und Ausgangspunkte

CodeWorld [Co] ist zuvorderst ein edukatives Projekt zur Unterstützung der Mathematikausbildung im Kontext nordamerikanischer Mittelschulkurrikula, insbesondere zum Thema Algebra. Im Webbrowser können mittels mathematischer Ausdrücke und Funktionen eigene Bilder, Animationen, Interaktionen erstellt bzw. programmiert werden. Den Unterbau bildet dabei eine Haskell-Bibliothek für geometrische Objekte, und tatsächlich gibt es neben der an das Schulpublikum gerichteten Version, etwa mit stärker an mathematische Schreibweise statt Haskell-Programmierung angelehnter Syntax, auch eine Version der Plattform für „vanilla“ Haskell (`https://code.world/haskell#`). Wir haben diese sowie eine Vorgängerbibliothek bereits seit mehreren Jahren, auch an der Universität Bonn in einer ähnlichen Lehrveranstaltung, eingesetzt, um vor allem zu Beginn des Semesters den Paradigmenwechsel von anweisungsbasiertem zu ausdrucksbasiertem Programmieren möglichst intuitiv zu gestalten. Während die CodeWorld API auch offline als normale Bibliothek genutzt werden kann, scheinen die Studierenden zusätzliche Möglichkeiten der CodeWorld-Plattform zu schätzen, etwa die Online-IDE und interaktive Features wie Inspektion/Zerlegung eines Bildes mit direkter Rückkopplung zum erzeugenden Code.

Über Autotool wurde durch dessen Urheber bereits auf dem dritten ABP-Workshop berichtet [Wa17], wobei der Schwerpunkt auf programmiersprachenunabhängigen, algorithmischen Aufgaben, deren automatischer Überprüfung, und auf der Erstellung/Konfiguration/Randomisierung solcher Aufgaben lag. Autotool ist in Haskell implementiert, und so gab es naheliegenderweise schon früh auch einen Autotool-Aufgabentyp zur Haskell-Programmierung. Dieser wurde ebenso bereits an der Universität Bonn eingesetzt, und um diverse Aspekte erweitert, die wir auch nun weiter nutzen und im Folgenden mit beschreiben.

3 Eigenschaftsbasiertes Testen

Da anders als bei üblichen Autotool-Aufgaben zu formalsprachlichen Themen, Algorithmen oder Datenstrukturen eine Überprüfung von Einreichungen nicht durch Entscheidungsprozeduren geleistet werden kann, arbeiten wir mit Tests. Dabei wird statt Unit-Testing der QuickCheck-Ansatz verwendet [CH00], im Wesentlichen bestehend aus eingebetteten DSLs

für einerseits datentypgetriebene Erzeugung zufälliger Eingaben, andererseits deklarativ spezifizierte Eigenschaften von Funktionen. Für eine Aufgabe zur Ermittlung der Anzahl von Vorkommen eines Elementes in einer Liste, also das Schreiben einer Funktion mit der Signatur `countElem :: Int -> [Int] -> Int`, werden etwa folgende Tests verwendet:

```
main :: IO ()
main = do putStrLn "Singleton list [x] contains x once:"
          quickCheck $ \x -> countElem x [x] == 1
          putStrLn "Singleton list [y] does not contain x if x /= y:"
          quickCheck $ \x y -> x /= y ==> (countElem x [y] == 0)
          putStrLn "Occurrences in a combined list sum up:"
          quickCheck $ \x ys zs -> countElem x (ys ++ zs)
                                   == countElem x ys + countElem x zs
```

Hier werden durch die `quickCheck`-Aufrufe automatisch passende Zahlen(paare) und Listen von Zahlen erzeugt, die angegebenen Gleichungen überprüft, und gefundene Gegenbeispiele (gegebenenfalls nach per ebenfalls datentypgetriebenen Strategien versuchter automatischer Minimierung) rückgemeldet. Mithilfe einer solchen Testsuite kann nun eine studentische Lösung entweder als korrekt, das heißt, dem gewünschten Verhalten entsprechend, oder inkorrekt klassifiziert werden. Obwohl es prinzipiell möglich wäre, auf Basis der einzelnen, in der Testsuite kodierten Eigenschaften zwischen verschiedenen Stufen von Korrektheit zu unterscheiden, verwenden wir aktuell lediglich eine binäre Klassifizierung.

In der obigen Version ist die Testsuite für die Studierenden direkt in der Konsole ausführbar, auf Autotool-Seite erfolgt die Einbettung etwas anders, da zum einen mehr Kontrolle gewünscht ist (etwa eine Staffelung von Tests, oder explizite Timeouts), zum anderen Maßnahmen getroffen werden müssen, um Ausführung von problematischem, eventuell sogar malignem Code auf dem Server zu unterbinden. Exzessives Sandboxing etwa durch Auslagerung in separate Prozesse oder Virtualisierung ist dabei nicht nötig, da einerseits Haskells Semantik und Typsystem diverse potentiell schädliche Seiteneffekte bereits unter Kontrolle halten, andererseits es im Haskell-Ökosystem etablierte Verfahren zur sicheren eingebetteten Ausführung gibt.[2]

Dass die obige Testsuite den Studierenden mit der Aufgabenstellung zur Verfügung gestellt wird, ist ein wesentlicher Aspekt. Sie ist aussagekräftiger als Unit-Tests, stellt hier sogar eine vollständige formale Spezifikation dar, und erlaubt den Studierenden auch lokal eine gründliche Prüfung[3], ohne etwa zum Vergleich bereits Zugriff auf eine Musterlösung haben zu müssen.

Bei anderen Aufgabenstellungen wäre es sogar unmöglich, Korrektheit durch Übereinstimmung mit einer (versteckten) Musterlösung zu definieren, da es kein eindeutiges Lösungsverhalten gibt. Zum Beispiel stellen wir Aufgaben, bei denen Serialisierung und

[2] `http://hackage.haskell.org/package/mueval`, `http://hackage.haskell.org/package/hint`

[3] Die den Studierenden zur Verfügung gestellte Testsuite kann je nach Aufgabe auch lediglich eine Untermenge der tatsächlich durchgeführten Tests sein.

Deserialisierung programmiert werden sollen. Etwa für einen endlichen Aufzählungstyp `data Color = Red | Blue | Yellow` bestünde die Aufgabe dann im Schreiben zweier Funktionen (die zweite per `Maybe` im Ergebnistyp explizit eine partielle Funktion):

```
encode :: [Color] -> [Bit]
decode :: [Bit] -> Maybe [Color]
```

welche in geeignetem Sinne invers zueinander sein müssen. Da es verschiedene Kodierungsstrategien gibt, kann eine Überprüfung einer studentischen Einreichung hier nicht im Vergleich der für `Color`-Listen erzeugten `Bit`-Listen mit den für dieselben `Color`-Listen durch eine vom Aufgabensteller hinterlegte Musterlösung erzeugten `Bit`-Listen bestehen. Vielmehr liegt der Schlüssel im direkten Ausdruck der (für funktionale Korrektheit einzig) relevanten Roundtrip-Eigenschaften, mit bei nicht korrekten Lösungsversuchen automatischem Finden, durch QuickCheck, von diese Eigenschaften verletzenden Gegenbeispielen:

```
main :: IO ()
main = do quickCheck $ \v -> decode (encode v) == Just v
          quickCheck $ \c -> let mv = decode c in
                             isJust mv ==> encode (fromJust mv) == c
```

Der Ansatz wie oben beschrieben ist direkt nur auf pure Funktionen (wie `countElem`, `encode`, `decode`) anwendbar, also Funktionen, die man zwar im Interpreter aufrufen kann, die aber keine vollständigen Programme sind. Ein Programm, das zum Beispiel Eingaben liest, diese verarbeitet, und Ergebnisse ausdruckt, hat in Haskell immer einen `IO`-Typ, verwendet Primitiven wie `putStrLn`, und hat gewöhnlich eine gegenüber purem Code andere Codestruktur (etwa oben in den `main`-Definitionen zu sehen an der imperativ programmiert anmutenden Sequenzialisierung von Anweisungen nach jeweils dem `do`-Schlüsselwort). Auch das Schreiben solcher Programme wollen wir in der Lehrveranstaltung üben, können für die Überprüfung entsprechender Aufgaben aber nicht einfach Eigenschaften der Art „Aussagen über Ergebniswerte für Funktionsaufrufe mit bestimmten Argumentwerten/-mustern“ verwenden, da es nun für die Korrektheit mindestens genauso sehr auf die `IO`-Effekte des Programms ankommt. Auf eine gesteuerte/überwachte Ausführung in einer Konsole mit tatsächlicher Auslösung der Effekte (und dann externe Beobachtung dieser, etwa Einlesen und Parsen der Programmausgabe zur Analyse in Tests) können wir jedoch verzichten dank eines Mechanismus zur Reifikation von Effekten als Datenstrukturen innerhalb Haskell selbst [SA07]. Dieser Ansatz, realisiert in der Bibliothek IOSpec, erlaubt uns zum Beispiel für eine Aufgabe wie „Lies eine Zahl n ein, dann weitere n Zahlen, und drucke deren Summe aus“ die Studierenden ein Programm wie

```
main :: IO ()
main = do putStr "Anzahl der Summanden: "
          n <- readLn
          while ...
            ...
          print ...
```

schreiben zu lassen, welches sie lokal kompiliert in einer Konsole ausführen und mit tatsächlichen Ein- und Ausgabeeffekten walten lassen könnten, für welches wir auf Autotool-Seite jedoch via stiller Ersetzung des „echten“ `IO`-Typs durch eine IOSpec-Typannotation (aber ohne irgendeine weitere Änderung des Programmcodes) eine als pure Funktion testbare Version des Programms erhalten. Diese Version löst keine Effekte mehr aus, sondern berechnet einfach aus Werten einer Eingabestrom-Datenstruktur einen wiederum als Datenstruktur weiterverarbeitbaren Strom von Ausgabewerten, samt Informationen zum Interleaving der Ein- und Ausgaben. Überprüfung der Korrektheit von Einreichungen, und Erzeugung von Feedback etwa in Form von Gegenbeispielen, erfolgt dann konzeptionell nicht anders als für von vornherein pure Funktionen wie `countElem`, `encode`, `decode` oben erläutert. Statt QuickCheck etwa Zahlen, Listen von Zahlen, oder `Color`- bzw. `Bit`-Listen erzeugen zu lassen, verwenden wir nun einen Generator für (zur Aufgabenstellung passende) Eingabeströme, und die überprüften Gleichungen/Eigenschaften nehmen zusätzlich auf den jeweils erhaltenen Ausgabestrom Bezug.

Bezüglich CodeWorld-Aufgaben ist zu sagen, dass wir aktuell keine automatischen Funktionstests durchführen. Es wäre bereits realisierbar, bestimmte dynamische Aspekte zu prüfen, etwa ob Bilder erzeugende Funktionen für relevante Eingaben zumindest in vorgegebener Zeit terminieren. Prinzipiell wäre auch ein QuickCheck-Zugang zur Überprüfung der erzeugten Bilder oder Animationen, auf Passung zur Aufgabenstellung, möglich, da CodeWorld intern mit einer expliziten Repräsentation von Vektorgrafiken als Datenstruktur arbeitet, auf der man Eigenschaften formulieren könnte. Bisher wird die Testsuite für CodeWorld-Aufgaben in Autotool aber einfach leer gelassen. Das Einreichenlassen auch dieser Aufgaben per Autotool (statt etwa Moodle) lohnt dennoch, da damit alle in den folgenden beiden Abschnitten beschriebenen Möglichkeiten zum Tragen kommen.

4 Vorgaben und Einschränkungen

Neben im einfachsten Fall „nur Vorgabe von Typsignaturen“ nutzen wir diverse weitere Mittel, um Aufgabenstellungen näher zu umreißen, Lösungsversuche in eine gewünschte Richtung zu bringen, oder aus verschiedenen Gründen nicht intendierte Wege auch auszuschließen. Ein vorhandenes Sprachmittel sind selektive Imports. Wollen wir etwa in der `countElem`-Aufgabe darauf hinwirken, dass ein Lösungsweg gewählt wird, der möglichst deklarativ die gewünschte Berechnung auf der Eingabeliste als Ganzes ausdrückt (etwa durch eine Comprehension und/oder durch Komposition existierender Transformations- und Aggregierungsfunktionen auf Listen), statt kleinteilig in Anlehnung an imperative Schleifenprogrammierung und mit Indexzugriff auf einzelne Elemente vorzugehen, dann können wir die Standardbibliothek genau minus den Indexzugriffsoperator importieren:

```
import Prelude hiding ((!!))
```

Wollen wir zusätzlich ausschließen, dass Lösungen ergoogelt und eventuell unverstanden übernommen werden, können wir auch noch alle in der zur Aufgabe passenden Antwort auf StackOverflow vorkommenden Higher-Order Funktionen verbieten:

```
import Prelude hiding ((!!), filter, map, fmap, foldl, foldr)
```

Wollen wir umgekehrt, an einem späteren Punkt in der Lehrveranstaltung, gerade den Einsatz einer solchen Funktion üben, also etwa erzwingen, dass eine Lösung per `foldr` geschrieben wird, dann können wir nicht auf ein vorhandenes Sprachmittel zurückgreifen. Es dürfte keine Programmiersprache geben, in der man rein durch Typ- oder Modulkonzepte die Verwendung einer bestimmten Funktion in der Implementierung einer anderen Funktion erzwingen kann. Stattdessen muss konkret die Syntax des eingereichten Programms geprüft werden. Wir verwenden für solche Zwecke eine in der Haskell-Community als Standard geltende Syntax-Bibliothek[4]. Für die meisten Aufgabenstellungen verlangen wir mindestens, dass die Einreichungen einen „Mustervergleich“ gegenüber einem vorgegebenen Programmtemplate erfüllen. Dieses dient sozusagen als Lückentext, der zu füllen/ergänzen ist. Die Syntax von Haskell als ausdrucksbasierter Sprache, in der fast alle bedeutungstragenden Teile kompositionell aufgebaute Ausdrücke sind (statt getrennter syntaktischer Konzepte für Ausdrücke, Anweisungen, Kontrollstrukturen), ist solch einer Formulierung besonders zugänglich. Als „Lücke“ dient der vordefinierte Ausdruck `undefined`. Eine Aufgabenstellung, die etwa `countElem` einfach nur irgendwie implementieren lassen will, würde als Template

```
countElem :: Int -> [Int] -> Int
countElem = undefined
```

vorgeben, eine spezifischer auf `foldr` abstellende Aufgabe stattdessen folgendes Template:

```
countElem :: Int -> [Int] -> Int
countElem value = foldr undefined undefined
```

Sind über einen Lückentext/Mustervergleich hinausgehende Forderungen oder Einschränkungen angezeigt, können wir speziellere Syntaxprädikate verwenden. Auch mittels haskell-src-exts implementiert haben wir so etwa die Möglichkeit der Überprüfung, ob eine eingereichte Funktion ein bestimmtes Fallunterscheidungsmittel nutzt oder nicht, rekursiv definiert ist oder nicht, eine bestimmte andere Funktion aufruft (ggfs. auch tiefer geschachtelt als oben mittels des `foldr undefined undefined` ausdrückbar) oder nicht, etc.

5 Statische Analyse und Hinweise

Neben syntaktischer und Typkorrektheit kann der Haskell-Compiler GHC weitere statische Analysen von Code durchführen. So lassen sich etwa Warnungen ausgeben, wenn bestimmte Arten von Fallunterscheidungen nicht vollständig sind oder sich darin Fälle überlappen, wenn Variablenbindungen zu Name Shadowing führen, wenn für eine Definition keine Typsignatur angegeben ist, wenn toter Code existiert, etc. Unser Haskell-Aufgabentyp in Autotool erlaubt selektives, einzelnes An- und Ausschalten dieser Warnungen, und lässt uns außerdem konfigurieren, ob sie nur als zusätzliches Feedback an die Studierenden

[4] `http://hackage.haskell.org/package/haskell-src-exts`

gegeben werden, oder ob Einhalten der Warnungen, bzw. Entfernen ihrer Ursachen im Code, erzwungen wird, bevor eine Einreichung als korrekt akzeptiert werden kann. So können wir einerseits gezielt typische Fehlerursachen abfangen, andererseits bestimmte Richtlinien durchsetzen.

Darüber hinaus setzen wir mit HLint [HL] ein Tool ein, das spezifische Vorschläge zur Vereinfachung und Verbesserung von Haskell-Code macht. Es weist zum Beispiel auf die Möglichkeit der Verwendung von Standardfunktionen hin, merkt überflüssige Klammerung oder anderweitig ungeschickte Syntaxverwendung an, erfasst aber auch tiefergehende Potentiale für Umstrukturierung oder Zusammenfassung von Ausdrücken, bis hin zu Fällen, in denen eine Datenstruktur unnötigerweise mehrfach traversiert wird (analog zu Loop Fusion). Nicht alle möglichen Hinweise sind für den Code Studierender sinnvoll, gegebenenfalls auch abhängig vom Fortschritt in der Lehrveranstaltung, etwa weil durch HLint vorgeschlagene Higher-Order Funktionen noch gar nicht behandelt wurden. Günstigerweise ist auch HLint feinkörnig konfigurierbar, so dass wir auch hier für jede Aufgabenstellung gezielt entscheiden können, welcher Fundus von Regeln in Anschlag gebracht werden soll. Ebenso behalten wir uns wieder vor, Hinweise nur als Feedback zu geben oder ihre Einhaltung zu erzwingen.

Ein weiterer nützlicher Aspekt von HLint ist, dass es relativ einfach erweiterbar ist. Davon haben wir einerseits bereits Gebrauch gemacht, indem wir basierend auf vergangenen Einreichungen Studierender zusätzliche Regeln abgeleitet haben, die nun Eingang in HLint gefunden haben und in Zukunft automatisch mit geprüft werden. Zum anderen sind auch in einer lokalen Konfiguration von HLint zusätzliche domänenspezifische Regeln angebbar, etwa für CodeWorld-Ausdrücke.

6 Diskussion und Ausblick

Die Bewertung von Programmieraufgaben automatisch durchführen zu lassen und dafür verschiedene Tools zur Bewertung zu verwenden, erscheint naheliegend. Ein zu unserem Vorgehen sehr ähnliches gibt es für OCaml [HP19], jedoch ohne Compilerwarnungen als Teil der Rückmeldung. Ein weiterer Fokus liegt dort zusätzlich darauf, die Studierenden zu motivieren, Unit-Tests für ihre Funktionen zu schreiben. Auch zu Lehrveranstaltungen mit Haskell-Programmierung gibt es bereits entsprechende Erfahrungen, wie zum Beispiel von Blanchette et al. [Bl14] berichtet. Sie setzen weniger Automatisierung ein als wir, im Wesentlichen beschränken sie sich dahingehend auf die Ausführung von Testfällen. Es gibt zahlreiche weitere Ansätze für automatische Bewertung von Programmieraufgaben mit unterschiedlichen Graden an Interaktivität und Rückmeldung, die auf Grund der hohen Anzahl hier nicht alle betrachtet werden können. Keuning et al. [KJH18] geben einen zusammenfassenden Überblick über entsprechende Veröffentlichungen.

Wir haben die Verwendung etablierter Tools für Haskell aufeinander abgestimmt, dabei haben wir insbesondere eine Reihenfolge des Ablaufs festgelegt, so dass die Rückmeldung

bei der schrittweisen Behebung von Problemen hilft und nicht mit zu vielen unterschiedlichen Meldungen überfordert. Die Konfiguration der Tools erfolgt für jede gestellte Aufgabe individuell, dies ermöglicht gezieltere Vorgaben für die Bearbeitung und eine bessere Feinabstimmung der Rückmeldungen. Durch unterschiedliche Vorgaben können die Einreichungen automatisch hinsichtlich verschiedener Kriterien überprüft und bewertet werden. Das ermöglicht es, Richtlinien ohne erhöhten Korrekturaufwand durchzusetzen, außerdem erfolgt die Rückmeldung vom System umgehend.

Es gibt jedoch auch Erweiterungsmöglichkeiten für das entwickelte Vorgehen. Beispielsweise ermöglicht es HLint zwar, strukturelle Vorgaben zu forcieren, allerdings sind diese Mittel eingeschränkt. Auch über unseren Mustervergleich sind bisher nur bestimmte strukturelle Überprüfungen möglich, er unterstützt keine aufgabenspezifischen Anforderungen. Individuelle strukturelle Vorgaben, wie zum Beispiel Überprüfung der Verwendung von Tail-Rekursion, könnten durch erweiterte Analysen des Syntaxbaums der Einreichung forciert werden. Solche Möglichkeiten gibt es zum Beispiel bereits in Learn-OCaml [HP19]. Als gänzliche Neuerung entwickeln wir gerade eine DSL zur intentionellen Beschreibung von (im Moment speziell `IO`-)Aufgaben [WV19], um dann benötigte QuickCheck-Artefakte wie Generatoren, zu testende Eigenschaften und andere Funktionalität automatisch zu erzeugen.

Literatur

[Bl14] Blanchette, J. C.; Hupel, L.; Nipkow, T.; Noschinski, L.; Traytel, D.: Experience Report: The Next 1100 Haskell Programmers. In: Proc. Haskell. S. 25–30, 2014.

[CH00] Claessen, K.; Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: Proc. ICFP. ACM, S. 268–279, 2000.

[Co] CodeWorld, `https://github.com/google/codeworld`, Accessed: Sept. 2019.

[HL] HLint, `https://github.com/ndmitchell/hlint`, Accessed: Sept. 2019.

[HP19] Hameer, A.; Pientka, B.: Teaching the Art of Functional Programming Using Automated Grading (Experience Report). Proc. ACM Program. Lang. 3/ICFP, Article 115, 2019.

[KJH18] Keuning, H.; Jeuring, J.; Heeren, B.: A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. ACM Trans. Comput. Educ. 19/1, Article 3, 2018.

[SA07] Swierstra, W.; Altenkirch, T.: Beauty in the beast: A Functional Semantics for the Awkward Squad. In: Proc. Haskell. ACM, S. 25–36, 2007.

[Wa17] Waldmann, J.: Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen. In: Proc. ABP, CEUR WS vol. 2015. 2017.

[WV19] Westphal, O.; Voigtländer, J., Describing Textual I/O Behavior for Testing Student Submissions in Haskell. Submitted: Post-Proc. TFPIE, 2019.

Relevanz der Codequalität in einem Praktikum mit automatisch getesteten Programmierabgaben

Andre Greubel[1], Tim Hegemann[1], Marianus Ifland[1], Martin Hennecke[1]

Abstract: Systeme zur automatisierten Bewertung von Programmieraufgaben bewerten üblicherweise nicht die Codequalität. In dieser Arbeit wird an einem Praktikum mit 212 Studierenden empirisch untersucht, welche Relevanz diese der Codequalität beimessen und welche Relevanz verschiedene Qualitätsmetriken auf das Bestehen des Praktikums haben. Des Weiteren wird analysiert, ob diese Metriken von externen Faktoren wie dem Studiengang oder der Praktikumswiederholung beeinflusst werden.

Keywords: Codequalität, Softwaremetriken, Java, Grader, Automatische Bewertung, Programmieraufgaben

1 Einleitung

Der Studiengang Informatik erfreut sich aufgrund guter Jobperspektiven immer größerer Beliebtheit, was sich durch immer größere Kurse bemerkbar macht. Systeme zur automatischen Bewertung von Programmieraufgaben sind vielerorts elementar, um den wachsenden Korrekturaufwand zu stemmen.

Ein Nachteil dieser Automatisierung ist, dass das Feedback häufig nicht sehr differenziert ausfällt. Meist wird nur eine Rückmeldung über die semantische Korrektheit des Programms oder seiner Teile gegeben. Insbesondere fehlt in vielen Systemen ein Feedback zur Codequalität, mit dem Studierende die Qualität ihrer Lösungen verbessern könnten. In dieser Arbeit wollen wir analysieren, welche Codequalität typische studentische Bearbeitungen aufweisen.

Ausgangspunkt ist die Erfahrung, dass Studierende der Codequalität bei automatisierter Bewertung zumeist nur wenig Bedeutung beimessen. Das ist kritisch zu betrachten, da die Codequalität in der Praxis aufgrund der hohen Relevanz von Wartungsaufgaben elementar ist und daher auch in der Programmierausbildung einen entsprechenden Stellenwert haben sollte.

[1] Institut für Informatik, Julius-Maximilians-Universität, 97074 Würzburg, {andre.greubel, marianus.ifland, martin.hennecke}@uni-wuerzburg.de, tim.hegemann@stud-mail.uni-wuerzburg.de

doi:10.18420/abp2019-4

In dieser Arbeit wird empirisch überprüft, welche Relevanz Studierende der Codequalität einräumen, welche Aspekte der Codequalität mit dem Bestehen in einem Programmierpraktikum zusammenhängen und ob diese Metriken von externen Faktoren wie dem Studiengang oder der Praktikumswiederholung beeinflusst werden.

2 Organisatorische Rahmenbedingungen

Als Datensatz dienen Abgaben von Studierenden des Java-Programmierpraktikums, das an der Universität Würzburg im Frühjahr 2019 durchgeführt wurde. Die größten Aufgaben in diesem Praktikum haben im Regelfall bei vorgegebener Architektur einen Umfang von etwa 2000 Zeilen Quellcode (Lines of Code).

Das Praktikum wird in organisatorischer Einheit als Programmierpraktikum (PP) für Informatik (INFO) und Einführendes Programmierpraktikum (EPP) für Mensch-Computer-Systeme (MCS) und Wirtschaftsinformatik (WINF) angeboten [1]. Dabei besitzt INFO mit 95 ECTS Informatik im Pflichtbereich den größten Informatikschwerpunkt, gefolgt von MCS mit 50 ECTS und WINF mit 30 ECTS. Die Studierenden haben im Vorfeld des Praktikums üblicherweise 20–25 ECTS in Kursen mit Programmierinhalt.

Der Arbeitsaufwand für die Aufgaben selbst soll in Summe in beiden Modulen jeweils 240 Stunden betragen. Die Abgabe erfolgt über das Würzburger Programmieraufgaben-Bewertungssystem (PABS) [ID17]. Die Einreichungen werden mit serverseitig in PABS ausgeführten, gewichteten JUnit-Tests bewertet und gelten als bestanden, wenn pro Aufgabe mindestens 90 % der Punkte erreicht wurden. Das Bestehen jeder Aufgabe ist notwendig, es dürfen dabei beliebig viele Versuche abgegeben und getestet werden.

Das Praktikum selbst ist in drei Phasen gliedert: (1) Die Einführungsaufgabe dient als Überprüfung, ob die für das Praktikum notwendigen Kenntnisse vorhanden sind. Die Bearbeitungszeit beträgt ca. sechs Tage. (2) Drei weitere (im EPP zwei weitere) Aufgaben müssen innerhalb von sechs Wochen nach Praktikumsstart bearbeitet und eingereicht werden. Die Aufgaben der Phase 2 haben gegenüber der Einführungsaufgabe größeren Umfang und höhere Komplexität. (3) Zur Überprüfung, ob die Aufgaben eigenständig bearbeitet wurden, muss zusätzlich eine etwa 90-minütige Abschlussklausur am PC bestanden werden.

Im analysierten Durchlauf haben nach insgesamt 60 930 Abgabeversuchen $n = 212$ Studierende (95 INFO, 46 MCS, 44 WINF) ausreichende Lösungen zur Einführungsaufgabe eingereicht. Davon haben 111 Studierende (52.4 %) auch alle Aufgaben der Phase 2 bestanden. Die Bestehensquote lag dabei in MCS deutlich höher (73.9 %) als in INFO (54.7 %) und WINF (47.7 %). Das Praktikum wird in jedem Semester angeboten und kann (außer in MCS) bei Misserfolg beliebig oft wiederholt werden. Eine numerische Note wird nicht vergeben.

[1] Das Praktikum kann noch in in neun weiteren Studiengängen belegt werden. Die Teilnehmerzahlen sind mit insgesamt 27 Absolventen allerdings jeweils geringfügig.

3 Studentische Einschätzung zur Codequalität

Im ersten Schritt erfolgt eine Bestandsaufnahme, welche Relevanz Studierende der Codequalität einräumen. Hierfür wurde nach Abschluss des Praktikums im Rahmen einer am Institut üblichen, freiwilligen anonymen Veranstaltungsevaluation eine Umfrage mit elf Fragen bezüglich der Einstellung zur Codequalität durchgeführt. Es liegen 88 ausgefüllte Evaluationsbögen vor. Nach eigener Aussage der Befragten wurden in 59 Fällen alle Aufgaben der Phase 2 erfolgreich bearbeitet. Der Anteil der Antworten nach Studiengang entspricht weitgehend dem Anteil des Studiengangs an der Grundgesamtheit [2]. Sofern nicht anders angegeben wird eine Likert-Skala (1: „Trifft voll und ganz zu“ ⇔ 4: „Trifft gar nicht zu“) verwendet.

Alle Antworten sind in Tabelle 1 dargstellt. Die erste Fragengruppe betrifft die Einstellung zur Codequalität, welche im Praktikum (Fragen 1-4) insgesamt als eher relevant und im Allgemeinen (Frage 5) sogar als sehr relevant eingestuft wird. Arbeitsweisen, die die Codequalität unterstützen, werden allerdings nur gelegentlich genutzt (Fragen 6 und 7, Skala hier abweichend: 1: „nie“ ⇔ 4: „oft“). Auch im Praktikum vorhandene Unterstützungsangebote, wie Angaben zum typischen Codeumfang in Klassendiagrammen der Angabe, werden als eher nicht hilfreich eingestuft (Fragen 8 und 9). Konsequenzen für das Praktikum werden kontrovers beurteilt (Fragen 10 und 11). Mehr Feedback zur Codequalität wird eher befürwortet, eine Aufnahme als Bewertungskriterium jedoch sehr kontrovers bewertet. Alle Antworten weisen allerdings eine hohe Varianz auf.

Ein Erfahrungswert bei der Betreuung des Praktikums ist, dass Rückmeldung zur Codequalität häufig interessierter angenommen wird, wenn die betroffene Person einen Studiengang mit hohem Informatikanteil studiert. Um zu testen, ob sich das in den gegebenen Antworten bemerkbar macht, wurden alle Antworten für jeden Studiengang in zwei Gruppen aufgeteilt (Teil des Studienganges vs. nicht Teil des Studienganges). Anschließend wurde mittels eines Welch-t-Tests untersucht, ob sich die Mittelwerte der Gruppen unterscheiden. Deutlich erkennbare Abweichungen ($p < 20\,\%$) sind ebenfalls in der Ergebnistabelle angegeben.

Derartige Abweichungen haben sich jedoch nur bei der Relevanz der Codequalität im Praktikum (Fragen 1-4) und der Einbindung der Codequalität in die Praktikumsbewertung (Frage 11) gezeigt. Die Richtung der Effekte entsprach den Erwartungen: Befragte aus der Gruppe INFO (WINF) messen der Codequalität im Praktikum tendenziell eine höhere (niedrigere) Relevanz bei als solche aus den übrigen Gruppen. MCS Studierende wichen hier nicht deutlich erkennbar von anderen Studiengängen ab. Eine Aufnahme der Codequalität als Bewertungskriterium wurde von Teilnehmenenden des PP stark befürwortet, von denen des EPP eher abgelehnt.

Insgesamt werten wir die Ergebnisse als Indikator, dass die Codequalität allgemein auch unter Studierenden als relevant erkannt wird. Im Kontext des Praktikums ist das vor allem im Bachelorstudiengang Informatik der Fall.

[2] Relative Abweichung der Zusammensetzung: INFO: −7.3 %, MCS: −0.2 %, WINFO: +5.4 %, Sonstige: +2.1 %

Frage	Ergebnis avg	stdev	Abweichung Studiengang: avg (p-Wert)
1: Die Hauptsache bei der Bearbeitung ist es, die PABS-Test grün zu bekommen.	1.80	0.90	WINF: 1.57 (14.3%)
2: Mir ist es wichtig, dass mein Code über das Bestehen der PABS-Test hinaus inhaltlich richtig ist.	1.94	1.02	WINF: 2.34 (5.1%) INFO: 1.73 (10.6%)
3: Mir ist es wichtig, dass ich über das Bestehen des Praktikums hinaus lerne gut zu programmieren.	1.62	0.96	WINF: 2.04 (3.3%) INFO: 1.36 (4.4%)
4: Mir ist es wichtig, dass mein Code nicht nur inhaltlich richtig, sondern auch gut lesbar ist.	1.94	0.96	WINF: 2.22 (14.0%) INFO: 1.73 (9.8%)
5: Programmieren ist einfacher, wenn man konsequent darauf achtet, guten Code zu schreiben	1.47	0.87	
6: Ich habe meinen eigenen Code refactored (= bereits inhaltlich richtigen Code überarbeitet, um ihn lesbarer zu machen oder besser zu strukturieren)	2.66	1.09	
7: Ich habe mir Codemetriken ausrechnen lassen, damit ich weiß, wie gut mein Code ist	1.26	0.86	MCS: 1.00 (1.6%)
8: Die LoC-Angaben in den Klassendiagrammen der Graphen-Aufgabe haben mir bei der Bearbeitung geholfen	2.85	1.27	WINF: 2.48 (15.8%)
9: In freieren Aufgaben meine eigene Struktur definieren zu können hilft mir, besseren Code zu schreiben	2.44	1.20	INFO: 2.16 (9.8%)
10: Ich würde mir mehr Rückmeldung zu meiner Codequalität wünschen	2.02	1.23	
11: Solange das Praktikum dadurch nicht schwieriger wird sollte auch die Qualität des Codes in die Bewertung mit einbezogen werden (statt nur inhaltliche Korrektheit zu bewerten).	2.54	1.26	WINF: 2.96 (3.7%) INFO: 1.21 (0.1%) MCS: 2.95 (12.7%)

Tab. 1: Ergebnisse der Umfrage zur Codequalität.

4 Codequalität der Einführungsaufgabe

Im nächsten Schritt folgt eine Analyse, ob besserer Code das Bestehen des Praktikums begünstigt. Hierfür wurden für alle ausreichenden Lösungen der Einführungsaufgabe insgesamt neun Qualitätsmetriken erhoben, die sich in drei Kategorien einteilen lassen.

Folgende drei Metriken wurden gewählt, da sie unserer Meinung nach übliche Metriken für die Qualität der Implementierung darstellen: (1) *Gesamtkomplexität (WMC)* definiert als Weighted Methods per Class nach [CK94]. Wir definieren eine Abgabe als besser, wenn sie die gleiche Funktionalität mit einer niedrigeren Gesamtkomplexität erbringt. (2) *Maximale Schachtelungstiefe (Nesting Depth)* von Blöcken unter allen Methoden der Abgabe. Eine Methode, die außer ihrem Rumpf keinen Block enthält, hat hier einen Wert von 0. Da geschachtelte Anweisungen schwieriger zu verstehen sind, ist ein niedrigerer Wert besser. (3) *Maximale Komplexität einer Methode* berechnet als das Maximum der zyklomatischen Komplexität nach McCabe [Mc76] aller Methoden.

Daneben wurden folgende Metriken erhoben, die als Indikatoren für die Strukturiertheit gelten können: (4) *Lines of Code (LoC)* der Abgabe, ohne Kommentare und Leerzeilen.

(5) *Maximaler Umfang einer Methode* unter allen Methoden in Codezeilen (vgl. LoC). (6) *Anzahl Methoden* in der Abgabe. Wir erwarten, dass sinnvoll strukturierte Abgaben viele, kurze Methoden benutzen und Code wiederverwenden, was zu weniger LoC führt.

Zusätzlich wurden folgende Metriken erhoben, die als Good bzw. Bad Smells im Code angesehen werden können [FB18]: (7) *Anzahl Literale* in der Abgabe. Viele Literale können auf fehlerträchtige Indextransformationen oder Magic Numbers im Code hinweisen. Wir gehen daher davon aus, dass bessere Abgaben weniger Literale beinhalten. (8) *Anzahl Typecasts* in der Abgabe. Die Aufgaben im Praktikum sind so gestellt, dass explizite Typecasts ausschließlich bei equals-Methoden notwendig sind. Viele Typecasts werten wir als Indikator, dass hier Lücken im Umgang mit generischer Programmierung und Vererbung vorhanden sind. (9) *Anzahl Lambda-Ausdrücke* in der Abgabe. Da funktionale Programmierung kein fester Teil des Curriculums ist, gehen wir davon aus, dass wer sich diese Programmiertechniken im Selbststudium beigebracht hat, besser abschneidet.

Metrik		Referenz-	Stud+		Stud−		Signifikanz
Nr.	Kürzel	lösung	avg	stdev	avg	stdev	(p-Wert)
1	WMC	66	127.41	12.73	133.15	17.79	0.86%
2	Nesting Depth	2	2.87	0.80	3.11	0.98	5.31%
3	Most Complex M.	5	13.83	3.53	14.50	3.94	9.06%
4	LoC	353	496.64	61.80	505.56	67.68	10.84%
5	Longest Method	12	38.98	15.05	37.99	9.64	43.41%
6	Methods	52	54.22	2.94	54.83	4.83	35.14%
7	Literals	41	87.80	25.20	93.37	22.58	0.72%
8	Typecasts	1	1.86	2.20	1.97	2.44	44.40%
9	Lamdas	14	1.37	3.30	1.30	3.05	48.67%

Tab. 2: Absolute Werte der Metriken und Einfluss auf den Praktikumserfolg.

Für die Erhebung wurden die Abgaben in zwei Gruppen aufgeteilt: Die Gruppe (Stud−) aller Studierenden, die zwar die Einführungsaufgabe bestanden haben, jedoch nicht alle folgenden Aufgaben ($n = 102$) sowie die Gruppe (Stud+) derjenigen, die alle Aufgaben bestanden haben ($n = 111$). Zusätzlich wurde eine bereits vorhandene Referenzlösung untersucht, die unserer Meinung nach die Anforderungen an eine qualitativ hochwertige Lösung erfüllt.

Die Ergebnisse sind in Tabelle 2 dargestellt. Erkennbar (und erwartbar) ist, dass die Referenzlösung in jeder Kategorie deutlich besser abschneidet als der Durchschnitt der Studierenden. Der Abstand fällt mit teilweise mehreren Standardabweichungen dennoch größer aus als von uns erwartet. Insbesondere bestehen Studierendenlösungen, unabhängig vom Praktikumserfolg, aus etwa einem Drittel mehr Codezeilen als die Referenzlösung und haben beinahe die doppelte Komplexität.

Nach Erhebung der Metriken wurde anschließend für jede Metrik analysiert, ob sie das Bestehen im Praktikum beeinflusst. Da Codemetriken üblicherweise nicht normalverteilt sind [Co07] und ein Shapiro-Wilk-Test für alle erhobenen Metriken eine Normalverteilung

Metrik		INFO		MCS		WINF	
Nr.	Kürzel	avg	p-Wert	avg	p-Wert	avg	p-Wert
1	WMC	127.62	1.12%	132.20	15.76%	133.48	4.82%
2	Nesting Depth	2.88	4.28%	2.89	34.44%	3.27	0.76%
3	Most Complex M.	13.56	0.10%	14.57	5.25%	15.57	2.26%
4	LoC	494.96	19.61%	495.22	19.92%	516.93	3.11%
5	Longest Method	37.49	8.29%	38.98	28.37%	41.86	14.75%
6	Methods	54.52	16.56%	53.80	44.00%	53.89	16.16%
7	Literals	87.82	20.02%	90.35	44.35%	95.45	12.83%
8	Typecasts	1.64	7.99%	2.07	34.44%	2.27	2.20%
9	Lamdas	2.03	0.10%	0.48	4.14%	0.48	3.14%

Tab. 3: Durchschnittliche Werte nach Studiengang und Signifikanz der Abweichungen.

widerlegte (jeweils $p < 1\,\%$) wurde der Wilcoxen-Rangsummentest (zum Niveau $p = 10\,\%$) verwendet. Die zugrundeliegende Nullhypothese besagt, dass die Werte der entsprechenden Metrik in den Gruppen gleich verteilt sind. Die Ergebnisse mit p-Werten sind ebenfalls in Tabelle 2 aufgeführt. Die drei klassischen Komplexitätsmetriken haben sich dabei als Indikator für den Praktikumserfolg bestätigt. Insbesondere die Hypothese, dass die Gesamtkomplexität der Abgabe unabhängig vom Praktikumserfolg ist, konnte verworfen werden ($p = 0.86\,\%$). Bei den anderen Metriken war das, mit Ausnahme der Literalanzahl, nicht der Fall.

Anschließend wurde analysiert, ob die Codequalität abhängig vom Studiengang ist. Hierfür wurde erneut mittels Wilcoxen-Test untersucht, ob sich die Verteilung der Codemetriken eines Studiengangs von denen anderer Studierender unterscheidet. Die Ergebnisse sind in Tabelle 3 dargestellt. Dabei erreichten INFO Studierende in allen Metriken bessere Ergebnisse. Diese Abweichung war in sechs Fällen signifikant. WINF Studierende hingegen erreichten in allen Metriken schlechtere Ergebnisse. Diese Abweichung war ebenfalls in sechs Fällen signifikant, insbesondere auch in den drei klassischen Komplexitätsmetriken. MCS Studierende zeigten bei mittelmäßigen Ergebnisen nur zwei signifikante Abweichungen.

Interessanterweise war die individuelle Bearbeitungszeit zwischen erstem Test und Lösungsabgabe ebenfalls stark vom Studiengang abhängig. So benötigen WINF Studierende mit etwa 85.25 Stunden durchschnittlich fast doppelt so viel Zeit für ihre Abgabe wie INFO (45.12) und MCS (46.94) Studierende.

Als weiterer Einflussfaktor wurde die Praktikumswiederholung analysiert. Dabei wurden 80 Personen mit durchschnittlich 2.75 (stdev 1.04) Praktikumsdurchläufen identifiziert, die mit exakt 50% auch eine leicht niedrigere Bestehensquote hatten als Studierende im Erstversuch. Erneut wurde getestet, ob sich die Verteilungen der Codemetriken tendenziell unterscheiden. Dies war jedoch nur bei zwei Metriken der Fall: Die längste Methode war im Durchschnitt um $11.23\,\%$ kürzer ($p = 0.37\,\%$) und es wurden durchschnittlich 0.54 Lambdas weniger pro Abgabe ($p = 1.58\,\%$) genutzt als in der Referenzgruppe. Die

durchschnittliche Bearbeitungszeit lag mit 60.52 Stunden jedoch signifikant ($p = 3.09\%$) höher, als die der Teilnehmenden im Erstversuch (48.43).

Weiterhin wurde analysiert, ob sich die Codequalität der Gruppe mit wiederholter Teilnahme im Vergleich zum letzten Versuch tendenziell verbessert hat. Da die absoluten Werte von der Aufgabe abhängen und nicht direkt vergleichbar sind, wurden für diese Analyse stattdessen die Verteilung der relativen Position ausgewertet. Mit Ausnahme der Literal- und Lambdaanzahl konnte eine leichte Verbesserung in allen Metriken festgestellt werden. Signifikant ist diese Änderung bei der Schachtelungstiefe ($p = 3.51\%$), der Komplexität der komplexesten Funktion ($p = 5.80\%$), Anzahl der Methoden (3.34%) und Typecasts ($p = 2.23\%$); allerdings auch bei der Anzahl der Lambdas ($p = 0.13\%$).

Zusammenfassend lässt sich sagen, dass gutes Abschneiden hinsichtlich der Metriken 1–3 tendenziell das Bestehen im Praktikum begünstigt. Die Referenzlösung war wesentlich besser als die der Studierenden. In Studiengängen mit höherer Affinität zur Informatik und umfangreicherer vorangegangener Programmierausbildung war die Codequalität tendenziell besser. Bei Praktikumswiederholung wird der Code im darauf folgenden Durchgang etwas besser und ist dann absolut kaum vom Durchschnitt zu unterscheiden. Die Bestehensquote liegt hier dennoch leicht unter dem Durchschnitt.

5 Verwandte Arbeiten

Einen aktuellen Überblick über den Stand automatischer Bewertungssysteme haben u.a. Ihantola et al. 2010 [Ih10] zusammengetragen. Dabei wurden in mehreren Arbeiten und mit unterschiedlichen Zielsetzungen auch Codemetriken berücksichtigt. Cardell-Oliver beschrieb 2011 [Ca11], wie sich Codemetriken nutzen lassen um das Lernverhalten von Studierenden zu analysieren und zu verbessern. Auch Pettit et al. 2015 [Pe15] betrachten dies mit Fokus auf die Veränderung in den Codemetriken zwischen verschiedenen Abgabeversuchen. Der Einfluss verschiedener Abgabemodelle wurde bereits 2005 von Malmi et al. [Ma05] untersucht.

6 Zusammenfassung und Ausblick

Die Auswertung des aktuellen Semesters hat gezeigt, dass auch Studierende eine hohe Codequalität als erstrebenswert ansehen, im Vergleich dennoch signifikant schlechtere Ergebnisse zeigen. Außerdem legen unsere Ergebnisse nahe, dass die Codequalität Einfluss auf den Praktikumserfolg hat. Ein erster Vergleich mit vorangegangenen Praktika hat zwar bereits eine Verallgemeinerbarkeit, aber auch den Einfluss von Organisationsstruktur und Aufgabe auf die Relevanz unterschiedlicher Metriken angedeutet. In unserer weiteren Arbeit wollen wir genauer analysieren, wie sich allgemeingültige Bestehensindikatoren ableiten lassen und welche Ziele diesbezüglich in der Programmierausbildung angesetzt

werden können. Auch soll untersucht werden, inwiefern unsere Politik der unbegrenzten Abgabeversuche Einfluss auf die Codequalität hat, weil sie evtl. Eigeninitiative hemmt, aber auch mehr Raum für Refactorings lässt.

Des Weiteren wollen wir mögliche Maßnahmen zur Verbesserung der studentischen Codequalität diskutieren. Dies könnte sowohl über zusätzliche Hilfsmittel wie Codereviews während der Bearbeitung oder einer Einbindung der Codequalität in die Bewertungskriterien erreicht werden. Auch dabei soll analysiert werden, welche Ansätze zur Umsetzung den gewünschten Effekt am besten erreichen.

Literatur

[Ca11] Cardell-Oliver, R.: How Can Software Metrics Help Novice Programmers? In: Proc. of the 13th Australasian Computing Education Conference. Bd. 114, Perth, Australia, S. 55–62, 2011.

[CK94] Chidamber, S. R.; Kemerer, C. F.: A metrics suite for object oriented design. IEEE Trans. on Software Engineering 20/6, S. 476–493, 1994.

[Co07] Concas, G.; Marchesi, M.; Pinna, S.; Serra, N.: Power-Laws in a Large Object-Oriented Software System. IEEE Trans. on Software Engineering 33/10, S. 687–708, Okt. 2007.

[FB18] Fowler, M.; Beck, K.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 2018, ISBN: 978-0-13-475759-9.

[ID17] Iffländer, L.; Dallmann, A.: Der Grader PABS. In: Automatisierte Bewertung in der Programmierausbildung. Bd. 6, Digitale Medien in der Hochschullehre, Waxmann Verlag, Kap. 15, S. 241–254, 2017.

[Ih10] Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O.: Review of Recent Systems for Automatic Assessment of Programming Assignments. In: Proc. of the 10th Koli Calling Int. Conference on Computing Education Research. Koli, Finland, S. 86–93, 2010.

[Ma05] Malmi, L.; Karavirta, V.; Korhonen, A.; Nikander, J.: Experiences on Automatically Assessed Algorithm Simulation Exercises with Different Resubmission Policies. ACM Journal on Educational Resources in Computing 5/7, Sep. 2005.

[Mc76] McCabe, T. J.: A Complexity Measure. IEEE Trans. on Software Engineering SE-2/4, S. 308–320, 1976.

[Pe15] Pettit, R.; Homer, J.; Gee, R.; Mengel, S.; Starbuck, A.: An Empirical Study of Iterative Improvement in Programming Assignments. In: Proc. of the 46th ACM Technical Symposium on Computer Science Education. Kansas City, Missouri, USA, S. 410–415, 2015.

Security Considerations for Java Graders

Sven Strickroth[1]

Abstract: Dynamic testing of student submitted solutions in evaluation systems requires the automatic compilation and execution of untrusted code. Since the code is usually written by beginners it can contain potentially harmful programming mistakes. However, the code can also be deliberately malicious in order to cheat or even cause damage to the grader. Therefore, it is necessary to run it in a secured environment. This article analyzes possible threats for graders which process Java code and points out Java specific aspects to consider when processing untrusted code.

Keywords: grader security, grader testing, security testing, automatic assessment, automatic grading

1 Introduction

Automatic graders are widely used in a variety of different systems and contexts such as assessment systems for programming courses, programming contest systems, and intelligent tutors [Ih10]. Robustness and trust are two central requirements on such systems (based on [Fo06]): Robustness means that a system does not crash randomly or when used in a way not intended by the developers. Trust includes two layers: (1) the system acts in a reliable way and cannot be cheated (the system behaves as expected at all times, it makes sure test results are gathered in a secure way and that the integrity of stored results is protected) and (2) sensitive data handed to the system remains private (e. g., by restricting access).

Nearly all of these systems require students to submit their code which is then automatically compiled and executed for grading or generating feedback on a server. In order to ensure trust and robustness considering security is important as untrusted code and data of students is processed. On the one hand the code is often from inexperienced programmers who might inadvertently code an endless loop or even harmful code. On the other hand more experienced students might try to contest a system to explore its boundaries (e. g. for cheating). Therefore, it is necessary to secure the environment on which the code is evaluated.

The main goal of this paper is to sensitize to security aspects as security is often not discussed in publications [Ih10] and to give an overview on possible threats. This should help developers of Java graders to test and optimize their systems. This paper focuses on Java as this is one of the most used language for programming support systems [St15].

In the following, the related research and the Java security architecture are presented. Then, key security aspects grouped into six sections are discussed. Finally, a conclusion is drawn.

[1] Universität Potsdam, Institut für Informatik & Computational Science, August-Bebel-Straße 89, 14482 Potsdam
sven.strickroth@uni-potsdam.de

doi:10.18420/abp2019-5

2 Related research

Security aspects and advises for graders are discussed in several publications: In [Fo06] a classification of types of attack vectors for programming contest systems (denial of service, privilege escalation, destructive attack, and covert channel) which can occur at different times of the contest (compilation-time, execution-time, and contest-time) is provided. Considered attacks include forcing a high compilation time, consuming resources at compilation time, accessing restricted information, misusing the network, modifying or harming the testing environment, circumventing the time measurement, exploiting covert channels, misusing additional services, and exploiting bugs in the OS – as well as possible preventions. Another list of possible attacks is provided in [TB10] with a quite similar categorization: attacks during compilation (e. g., excessive submit size, referencing forbidden files, and denial of service), attacks during sandboxing (e. g., read/write files/directories, open sockets/access network, spawn multiple threads/processes, raising privileges resp. breaking the sandbox, vulnerabilities in the grader, and denial of service), and exploits during checking. Sims [Si12] categorizes attacks into denial of service (e.g., using a fork bomb, infinite loop, memory leak or excessive output), corruption of test harness, and leaking test data and discusses different security models (user-level restrictions, process-level restrictions, mandatory access control (e. g., selinux), and virtual machines). Problematic system call groups (e. g., access to files, allocation of memory, creation of processes and threads, inter-process communication, executing other programs, and flushing buffers to disk) and an evaluation of the performance of different sandboxing approaches can be found in [MB12]. However, neither of the articles mentioned above is dedicated to Java and provides an overview of relevant security aspects.

3 The Java Virtual Machine and the Java Security Architecture

Java provides its own sandbox – the Java Virtual Machine (JVM) – in which the bytecode is executed. It is not a full virtual machine, but more like an additional layer of abstraction of the host operating system. Since version 1.2 of the JDK Java includes an enhanced security architecture which allows to restrict possible harmful actions programs can perform at run time. Java uses a stack-based permission approach.[2] When an action requires a special permission the `SecurityManager` (which delegates to `AccessController` since Java 1.3) is asked before executing the action which then checks whether all code blocks on the stack have the proper permissions. If and only if all code blocks on the call-stack have a specific permission, the permission is granted. Otherwise a runtime `java.security.AccessControlException` is thrown and the action is denied. There is, however, a special `PrivilegedAction` block which changes this logic in order to allow trusted code take responsibility for executing privileged actions on the behalf of otherwise untrusted code: If the call stack includes a `PrivilegedAction` block and the calling code has the proper permissions, then the action is permitted – the rest of the call stack is not checked.

[2] `https://www.oracle.com/technetwork/java/seccodeguide-139067.html#9`, `https://docs.oracle.com/en/java/javase/12/security/java-se-platform-security-architecture.html`

Therefore, it is important that `PrivilegedAction` blocks are as short as possible and used with care in order to not open additional attack vectors (also check loaded libraries).[3] Such an `PrivilegedAction` block (maybe in a trusted wrapper class) would be required for testing/mocking frameworks in order to generate and load test doubles on the fly.

The Java security manager can be enabled by passing `java -Djava.security.manager` to the JVM on the command line and a policy file can be specified using `-Djava.security.policy=FILENAME` (cf. List. 1).[4] An easy way to check whether a security manager is active is to check whether `System.getSecurityManager()` is `null` or not. Using `System.out.println(System.getSecurityManager().getClass().getName());` the active security manager can be detected (default is `java.lang.SecurityManager`).

```
grant codeBase "file:/FULLPATH/TESTCASE.jar" {
  permission java.lang.RuntimePermission "setIO";
  permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
};
grant {
  permission java.io.FilePermission "-", "read,␣write,␣delete";
  permission java.lang.RuntimePermission "accessDeclaredMembers";
};
```

List. 1: An example of a typical policy file which grants more permission to the TESTCASE than to the student code; e.g., `ReflectPermission ``accessDeclaredMembers''` is required for JUnit

Concluding, the Java sandbox together with the security manager allow to restrict access of untrusted student code to the operating system. Permissions need to be explicitly granted (with care, i. e. least privileges at the smallest scope possible). Not (fully) restricted by the `SecurityManager` are, however, the execution duration as well as memory and disk usage.

4 Memory Restrictions (Heap Size)

Allocating as much memory as possible can be used as a denial of service attack. It is possible to limit the memory usage by the host system (e. g., using POSIX `setrlimit`, cf. [Si12]), however, the Java Virtual Machine already automatically limits the maximum usable memory (usually to 25 % of available physical memory, depends on the host system architecture etc.; in order to see your Oracle/OpenJDK JVM details issue `java "-XX:+PrintFlagsFinal"` on the CLI and look for `MaxHeapSize`)[5] and throws an `OutOfMemoryError` exception in case an object runs out of memory and no more memory could be made available by the garbage

[3] see guidelines on `https://www.oracle.com/technetwork/java/seccodeguide-139067.html#9` also see `https://www.exploit-db.com/papers/45517`

[4] `https://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html`, certain placeholders are possible; `https://docs.oracle.com/en/java/javase/12/security/permissions-jdk1.html`

[5] `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html#default_heap_size`

collector. If a JVM is exclusively used for evaluating a single student solution, the JVM dies cleanly with an exit code ! = 0. However, if the student code is executed in a shared VM with other system code, this might be a more complex issue on how to react on the error (the Java documentation explicitly says "a reasonable application should not try to catch" an `Error` exception)[6]. A different maximum can be set by passing `-Xmx1G` to the JVM.[7] Generally, the maximum heap size should be chosen with care (how many processes might run in parallel, can the host go into an out of memory condition or start swapping). A lower limit could also be considered for the Java compiler (cf. end of Sect. 5).

5 Time Restrictions

Limiting the maximum time a student written program runs is very important so that lingering around processes can not occupy limited evaluation slots. An endless loop, a dead lock or infinite wait (e. g., attempts to acquire a mutex twice) can easily occur as a mistake. Generally, restricting time is considered more complex [Ma07]. Special attention must be paid, as a time restriction (e. g., based on POSIX `setrlimit`, cf. [Ma07]) does not limit the real execution time, but only the used CPU seconds. It is important that the real execution time (wall clock time) gets limited, as a waiting process might run for several hours (more than 3 hours on a test system) with a CPU time limitation of only 5 seconds (`ulimit -t 5`).

```
class A {{
  int a;
  try { a=0; } finally { try { a=0; } finally { try { a=0; } finally {
  try { a=0; } finally { try { a=0; } finally { try { a=0; } finally {
  try { a=0; } finally { try { a=0; } finally { try { a=0; } finally {
  try { a=0; } finally { try { a=0; } finally { try { a=0; } finally {
   a=0;
  }}}}}}}}}}}}
}}
```

List. 2: Java "Compiler Bomb" with 12 nested try-finally-blocks; `https://habr.com/en/post/245333/`

Time restrictions should not just be enforced for the run time of the student code, but also for the compilation of the code. Take the code listed in List. 2, it shows an example of a so-called compiler bomb (i. e. (relatively) small files which produce enormous output or consume a lot of resources), which is outputs an approx. 6 MiB class-file and takes about 3 seconds to compile (with OpenJDK 1.8.0_222). By just inserting 10 additional `try {a=0;} finally {` lines and corresponding closing braces, the compile time grows to about 5 minutes (but finally fails with `java.lang.OutOfMemoryError: Java heap space`). Additionally, there already were compiler bugs which caused infinite loops [Si12]. Such attacks can be used for a denial of service regarding compile time and potentially disk space (cf. Sect. 6).

[6] `https://docs.oracle.com/javase/9/docs/api/java/lang/Error.html`
[7] `https://docs.oracle.com/javase/9/tools/java.htm`

6 Filesystem access

File system access is an attack vector which is considered in all related research articles. It is important that student code can neither access restricted material (e. g., correct outputs for the test data, author's solution), modify or harm the testing environment (e. g., delete all files, replace evaluators, store state information between runs, overwrite test results), fill the hard disk (disk space and/or inodes; can also happen indirectly by outputting a huge amount of data to `stderr` or `stdout`), nor causing a high IO load by performing a lot of small writes.

Using `FilePermission` fine grained permissions can be granted, however, it is important to know that "code always automatically has permission to read files from its same (URL) location, and sub directories of that location; it does not need explicit permission to do so"[8] (as long as the user running Java has the appropriate rights; [Si12] provides a user/group model) and that code "can also obtain the pathname of the directory it is executed from, and this pathname may contain sensitive information"[8]. Indeed, in tests it was possible to access files and directories using a priori known names in and below the mentioned directory, however, it was not possible to list all files/directories using `File.listFiles()`. Yet, files a student should not access must be placed somewhere else as *security by obscurity* does not work (cf. [Fo06]). Access to other files requires explicit permissions in the policy file.

For some tasks it is required, however, that students can create and write to files. Particularly in these cases it is important that a clean(ed) environment is used for every evaluation of a student solution (called test "idempotency" resp. "hygiene" in [Si12]). In any case, it should be ensured that no arbitrary files can be placed into a directory which is on the class path. This is especially true for precompiled Java bytecode (`.class`) files. Loading these classes could be used to circumvent static analysis for detecting forbidden code/calls (ineffectivity is discussed in [Fo06]), package-private access to test code, or to "override" existing library classes (e. g., by creating a directory structure such as `org/junit` and dropping a prepared `.class` file there). This is not an issue for classes in a package whose name starts with `java.` as the default class loader refuses to load these from the user defined class path. Generally, the student code should be at the end of the class path after the test code for mitigation reasons. At best, use different class loader instances to separate classes.[9] There also exists a related problem: If custom security managers are used which rely solely on the absolute name of Java packages/classes for granting permissions (instead of the code location as the default security manager does) those can be tricked by using the approach described above.

A limitation of the explicit used disk space (and inodes) can be enforced e. g. by using quota features, by using distinct (logical) partitions or fixed-sized image files (those could be set up in advance and maybe pooled in order to not require root permissions). A limitation should also be enforced for the compiler (cf. Sect. 5). Generally, granted `FilePermission`s need to be tested carefully. There is no way in the standard JVM to limit a high IO and overwhelming output to `stderr` or `stdout` – here, other approaches such as Linux containers (cf. [MB12]) and trimming the data before storing in databases are necessary.

[8] `https://docs.oracle.com/en/java/javase/12/security/permissions-jdk1.html`, at least since Java 1.6

[9] `https://www.oracle.com/technetwork/java/seccodeguide-139067.html#4`; also prevents package-priv. access

7 Reflection

A quite special feature of Java is reflection, which has to be considered. Using reflection is always possible with Java regardless of any granted permissions. This way access to any classes on the class path is generally possible, e. g. directly calling the model solution or loading injected `.class` files. However, for listing as well as accessing protected/private methods/fields special permissions are required (`RuntimePermission "accessDeclaredMembers"` resp. `ReflectPermission "suppressAccessChecks"`; additional checks for packages of the JDK which are not considered public API such as `com.sun.proxy` exist and require further permissions (`accessClassInPackage.{package name}`). Whether or not reflection can be exploited needs to be carefully considered on a case-by-case basis (with the stack-based security model and class path in mind, cf. Sect. 9 for a case). There is no general solution.

8 Serialization and Deserialization

Deserialization of untrusted data can be used for denial of service (e. g., resource exhaustion, deserialization bombs) or also for (remote) code execution [Sv16]. This is especially an issue if serialized data is unserialized in code which has more privileges as this could be used for privilege escalation: That could be grader code which unserializes a student-created object or also any existing `PrivilegedAction` block within the JDK or any used library. Serialized data should be checked without deserializing it (e. g., compare the byte stream).

Serialization could also be used to access/modify otherwise protected/private data (using the serialized byte stream) or to instantiate classes with a private constructor. In principle, all classes on the class path that implement the `Serializable` interface are affected. Note, that no file system access is required for serialization (e. g., by using `ByteArrayOutputStream`). Therefore, it should be ensured that potential critical classes do not implement the `Serializable` interface.

9 Grader-Result Interface

How is the result of the student code resp. the grader passed back to a calling system? Possible ways include using `stdout`, a specific results file, calling an API of the surrounding system by the grader and/or checking the exit code of the JVM process. In all cases it is important to make sure that the untrusted student code cannot mimik (e. g., by printing a fake output similar to the one of the text based JUnit runner or calling the very same API) or overwrite the results (file) of the grader (e. g., by writing to and write protecting it or by using a JVM shutdown hook, the latter requires the `RuntimePermission "shutdownHooks"`). It might even be possible the fake the exit code and/or terminating the JVM by using `System.exit(0);` (after faking a positive result or just as a denial of service if the student code runs in a shared JVM), because the “"exitVM.*" permission is automatically granted

to all code loaded from the application class path"[10] by default. In order to prevent this an extended security manager is required (cf. List. 3). No generic satisfying solution is known to the author.

```
public class NoExitSecurityManager extends SecurityManager {
  public void checkExit(int status) {
    super.checkPermission(new RuntimePermission("exitTheVM." + status));
  }
}
```

List. 3: An example for an extended `SecurityManager` which checks `System.exit(int)` and introduces a new permission `exitTheVM.*` which might need to be granted to the grader code

10 Discussion and Conclusions

Denial of service vulnerabilities may seem less critical, however, there are often peaks which often occur before deadlines in assessments [SG11]. A lot of issues can be mitigated by just safely and strictly enforcing a maximum run time, usable hard disk space as well as a proper queuing of evaluations of student solutions (especially important of the evaluation takes a long time).

Despite from errors or attacks in the student code, graders and testing code can also contain security vulnerabilities or programming errors (cf. [TB10]) and should, therefore, also run with least privileges possible (who writes the test code and how trustful is that person?). A general advise is to keep the grader and the policy file as simple as possible in order to allow easy review for security and correctness. Sample testing runs should be conducted in order to test whether the permissions are correctly applied and actually do work. Some Java source code which addresses/triggers the mentioned attacks for testing purposes can be found on `https://gitlab.com/javagradersec/examples`.

Not considered in this paper are exploiting bugs in the operating system, misusing additional services, spawning processes/threads, sending signals, network access, covert channels and the security of surrounding systems such as the web interface where the grader results are presented. The JVM comes with a sophisticated security architecture which requires permissions to be explicitly granted if the security manager is enabled (exceptions see Sect. 6 and Sect. 9 as well as default read permissions for some system properties). Spawning threads inside the JVM is always possible. Permissions should be granted with care (cf. Sect. 3) and possible new attack vectors need to be anticipated. Network access for example could be abused for starting a denial of service attack on third party systems or exfiltrate data. The Secure Coding Guidelines for Java SE state: "Utilizing lower level isolation mechanisms available from operating systems or containers is also recommended."[11]

[10] `https://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html`, explicitly stated until Java 8

[11] `https://www.oracle.com/technetwork/java/seccodeguide-139067.html#9`

JVM and SecurityManager:
- ☐ Is the Java `SecurityManager` active?
- ☐ Are permissions granted and `PrivilegedAction` blocks used as restrictive as possible (how trustful are the tests and used/loaded libraries)?
- ☐ Is the student code tested in a separate JVM?

Memory:
- ☐ Is the maximum usable (heap) memory for compilation and execution reasonably restricted according to the host hardware (e. g., multiple tests running in parallel)?

Time:
- ☐ Are the program compilation and execution reasonably wall-clock time limited?

Reflection (Reflection is always possible):
- ☐ Are critical classes reachable inside the JVM (e. g., model solution code)?

Serialization:
- ☐ Untrusted objects must not be unserialized.

Filesystem:
- ☐ Is the test environment reset after each test run?
- ☐ Are model solutions or other critical files inaccessible from student code?
- ☐ Is the class path secured against injection of (.class) files (via upload or student code)?
- ☐ Are code permissions granted based on code path (instead of package name)?
- ☐ Are reasonable filesystem quotas enforced (for compilation and execution)?
- ☐ Is IO load limited?
- ☐ Is the output of the compiler/student program limited before saving (to a database)?

Grader-Result Interface:
- ☐ Is the interface safeguarded against skipping of tests or faking the test outputs?
- ☐ Is the JVM secured against prematurely exit?

Fig. 1: Java grader security checklist

Fig. 1 sums up all aspects of this paper. Some discussed aspects might seem to be academic, but still need to be considered. Of course this paper cannot be complete or guarantee perfect security. Hopefully, it sensitized readers to optimize their own systems for less known cases.

References

[Fo06] Forišek, M.: Security of programming contest systems. Information Technologies at School/, pp. 553–563, 2006.

[Ih10] Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O.: Review of recent systems for automatic assessment of programming assignments. In: Proc. Int. Conf. on Computing Education Research. ACM, New York, NY, USA, pp. 86–93, 2010.

[Ma07] Mareš, M.: Perspectives on grading systems. Olympiads in Informatics 1/, pp. 124–130, 2007.

[MB12] Mareš, M.; Blackham, B.: A New Contest Sandbox. Olympiads in Informatics 6/, pp. 100–109, 2012.

[SG11] Striewe, M.; Goedicke, M.: Studentische Interaktion mit automatischen Prüfungssystemen. In: Proc. DeLFI 2011. GI, pp. 209–220, 2011.

[Si12] Sims, R. W.: Secure Execution of Student Code, tech. rep., University of Maryland, Department of Computer Science, 2012.

[St15] Strickroth, S.; Striewe, M.; Müller, O.; Priss, U.; Becker, S.; Rod, O.; Garmann, R.; Bott, J. O.; Pinkwart, N.: ProFormA: An XML-based exchange format for programming tasks. eleed 11/1, 2015, ISSN: 1860-7470.

[Sv16] Svoboda, D.: Exploiting Java Deserialization for Fun and Profit, 2016.

[TB10] Tochev, T.; Bogdanov, T.: Validating the Security and Stability of the Grader for a Programming Contest System. Olympiads in Informatics 4/, pp. 113–119, 2010.

ProFormA 2.0 – ein Austauschformat für automatisiert bewertete Programmieraufgaben und für deren Einreichungen und Feedback

Paul Reiser[1], Karin Borm[2], Dominik Feldschnieders[3], Robert Garmann[4], Elmar Ludwig[3], Oliver Müller[5], Uta Priss[2]

Abstract: Das seit 2011 entwickelte ProFormA-Format soll den Austausch von automatisiert bewertbaren Programmieraufgaben zwischen verschiedenen Lernmanagementsystemen und Gradern befördern. Dieser Beitrag stellt die neue Version ProFormA 2.0 vor, die neben der eigentlichen Aufgabe jetzt auch das Format der studentischen Einreichung, des Bewertungsschemas und der Grader-Antwort standardisiert.

Keywords: automatisch bewertete Programmieraufgaben, XML Austauschformat, Interoperabilität, E-Assessment, LMS, Grader

1 Einleitung und Motivation

Das ProFormA-Format wird seit 2011 im Rahmen des eCULT-Projekts entwickelt [St15, St17]. In der ersten Version war es auf die detaillierte Beschreibung von Programmieraufgaben fokussiert einschließlich der Aufgabenbeschreibung, zumindest einer Musterlösung und Tests zur Bewertung der Aufgabe. Dieses Format soll die Kommunikation von Lernmanagementsystemen (LMS), die Aufgaben verwalten und von Studierenden und Lehrenden bedient werden, mit Bewertungssystemen (Gradern) und eventuell Vermittlungssystemen (Middleware) vereinheitlichen. Grader laufen im Hintergrund und werden zur Bewertung eingesetzt, zum Beispiel zur Ausführung von Kompilations- und Unittests und Stilprüfungen. Eine Middleware kann dazu dienen, mehrere verschiedene Grader an ein LMS anzuschließen. Durch die Vereinheitlichung der Kommunikation können unterschiedlichste Systeme (LMS, Grader und Hybrid-Systeme, siehe Abschnitt 4) kombiniert werden, sofern sie das Format unterstützen. Programmieraufgaben, die für ein System geschrieben werden, funktionieren auch auf den anderen Systemen und können mit anderen Lehrenden ausgetauscht und wiederverwendet werden.

[1] ZLB – E-Learning Center, Hochschule Hannover, Expo Plaza 4, 30539 Hannover, paul.reiser@hs-hannover.de
[2] ZeLL, Ostfalia Hochschule, Salzdahlumer Straße 46/48, 38302 Wolfenbüttel, {k.borm,u.priss}@ostfalia.de
[3] Zentrum für Digitale Lehre, Campus-Management und Hochschuldidaktik, Universität Osnabrück, Heger-Tor-Wall 12, 49074 Osnabrück, {dofeldsc,elmar.ludwig}@uos.de
[4] Fakultät IV, Hochschule Hannover, Ricklinger Stadtweg 120, 30459 Hannover, robert.garmann@hs-hannover.de
[5] Institut für Informatik, TU Clausthal, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, oliver.mueller@tu-clausthal.de

doi:10.18420/abp2019-6

Einen Überblick über ProFormA 1.1 und dessen Einsatz in diversen Systemen bietet [Bo17]. ProFormA 2.0, ein Whitepaper und Beispiele sind auf Github[6] verfügbar.

Neben einem überarbeiteten Aufgabenformat wurde in der zweiten, in diesem Artikel vorgestellten Version, auch das Format, in dem das LMS die studentische Lösung an den Grader schickt und die Antwort, die der Grader an das LMS schickt, standardisiert. Dieser Beitrag bietet einen Überblick über ProFormA 2.0.

2 Neue Anforderungen an das Format

Die in [St15] veröffentlichte Erstversion des ProFormA-Formats deckt eine Obermenge der Anforderungen jedes Graders ab. Sie fokussiert auf Aspekte der Aufgabe (*task*): Aufgabentext, Ausfüllvorlagen (Templates), Code-Bibliotheken, Testcode, Konfigurationsdateien und Musterlösungen. Auch ProFormA 2.0 soll möglichst viele Grader sowie zusätzlich LMS und Middlewares unterstützen. Die daraus erwachsenden zusätzlichen Anforderungen werden nun skizziert und nummeriert. Alle Anforderungen adressieren den Dateninhalt. Die Anforderung eines zwischen den Systemen abgestimmten, technischen Kommunikationsprotokolls (REST, SOAP, etc.) ist kein Gegenstand dieses Beitrags.

Es müssen Formate für die Kommunikation zwischen LMS-Frontend und Grader-Backend formuliert werden. Möglichst jedes LMS soll eine studentische Einreichung (*submission*) an jeden ProFormA-kompatiblen Grader senden können **(R1)**. Ein LMS soll das als Antwort (*response*) automatisch generierte Feedback entgegennehmen und Zielgruppenorientiert anzeigen können **(R2)**. Feedback liegt gewöhnlich quantitativ (z. B. als Punktzahl) und qualitativ vor (z. B. Prosa, Bilder, XML-Datei). Da dieses meist in Teilen den ausgeführten Testschritten zugeordnet wird, soll die Zuordnung vom neuen Format unterstützt werden **(R3)**. Besteht der Bewertungsprozess einer Aufgabe aus mehreren Testschritten, so werden die einzelnen Testergebnisse i. d. R. gewichtet und aggregiert. Während viele der in der Erstversion des Formats spezifizierten Teile einer Aufgabe bei einer Wiederverwendung nicht geändert werden müssen, hängt die Gewichtung einzelner Testergebnisse häufig vom Lehrkontext ab. Lehrende sollen die in einer Aufgabe angelegten, durch Tests bewerteten Aspekte individuell gewichten können **(R4)**. Es wurde ein Format für ein Bewertungsschema (*grading-hints*) gesucht, das ein LMS der Lehrkraft editierbar anzeigen kann, und das die Systeme LMS, Middleware oder Grader zur Berechnung des Gesamtergebnisses zu einer Einreichung nutzen können. Die Wiederverwendung von Aufgaben führt zu der Anforderung, das in der Aufgabe spezifizierte Bewertungsschema für jede Einreichung austauschbar zu gestalten **(R5)**. Als letzte und weitere wichtige Anforderung sollen Aufgaben internationalisiert und mit gezielten Zusatzinformationen spezifiziert werden können, um eine Wiederverwendung durch Menschen (insb. Lehrkräfte) und Maschinen (insb. LMS) zu unterstützen **(R6)**.

[6] https://github.com/ProFormA/proformaxml, https://github.com/ProFormA/examples

3 ProFormA Format 2.0

Abb. 1 stellt die Elemente von ProFormA 2.0 dar. Die Neuerungen gegenüber Version 1.1 werden in den folgenden Abschnitten erläutert.

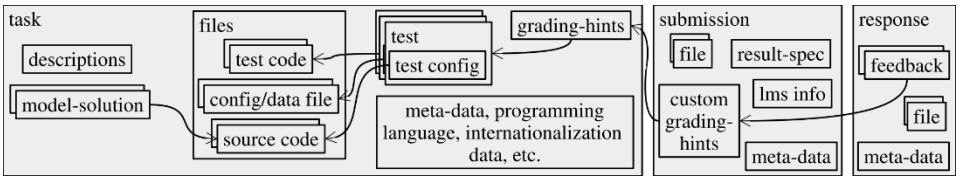

Abb. 1: Elemente einer ProFormA 2.0 Aufgabe, Einreichung und Feedback. Einige Verweise zwischen Elementen sind als Pfeile dargestellt.

3.1 Internationalisierung und allgemeine Datentypen

Bereits in ProFormA 1.1 lässt sich festlegen, ob zu einer Aufgabe gehörende Dateien (Elemente *files/file*, vgl. Abb. 1) bzw. deren jeweiliger Inhalt in einer task.xml eingebettet oder als Teil einer Archivdatei angehängt werden. Für *file*-Elemente, die nun wie in Abb. 1 gezeigt auch innerhalb von *submission*- und *response*-Elementen verwendet werden können, wird in ProFormA 2.0 zusätzlich zwischen Textdateien (Elemente *embedded-txt-file, attached-txt-file*) und Binärdateien (*embedded-bin-file, attached-bin-file*) unterschieden. Während eine eingebettete Textdatei stets UTF-8 kodiert ist und eine eingebettete Binärdatei einer XML base64-kodiert hinzugefügt wird, gibt es die Möglichkeit, für eine angehängte Textdatei über ein Attribut ein zutreffendes Encoding anzugeben. Durch die beschriebenen Änderungen bei *file*-Elementen soll ein LMS beim Anzeigen und ein Grader beim Verarbeiten von Dateien besser unterstützt werden (R1 und R6).

ProFormA 2.0 unterstützt zudem die Möglichkeit für bestimmte Elemente in *task* oder *response* nicht nur eine, sondern beliebig viele natürliche Sprachen zu verwenden (R6, Abb. 1 *internationalization*). Die Hauptsprache einer Aufgabe wird über das optionale *task*-Attribut *lang* angegeben, sofern die Aufgabe in einer einzigen Sprache vorliegt oder nur ein Teil (z. B. die sich an Studierende richtende Aufgabenstellung) in andere Sprachen übersetzt ist. Sind Übersetzungen für *title, description, internal-description* und/oder *content*-Elemente vorhanden, so wird dies im entsprechenden Element gekennzeichnet. Die Kennzeichnung erfolgt durch Angabe der Zeichen @@@ (z. B. <title>@@@title2@@@</title>). Die konkreten Übersetzungen selbst stehen bezugnehmend auf die oben genannten Kennzeichnungen in einer UTF-8 kodierten *strings.txt*-Datei, wobei für jede natürliche Sprache eine separate Datei unter Verwendung entsprechender Ländercodes angelegt werden muss. Dateien lassen sich ebenfalls in mehreren natürlichen Sprachen anbieten und in dem Fall ausschließlich als Anhang beifügen. Im jeweils hierfür vorgesehenen *attached-txt-file*- bzw. *attached-bin-file*-Element wird der anzugebende Dateiname wie oben bereits dargestellt gekennzeichnet (z. B. <attached-bin-file>@@@path-to-pdf-file@@@</attached-bin-file>).

3.2 Bewertungsschema (das *grading-hints*-Element)

Die Gestaltung des Bewertungsschemas (R4) wurde detailliert in [Ga19] beschrieben und kann hier aus Platzgründen nicht ausführlich wiedergegeben werden. Wesentliches Gestaltungsprinzip ist ein Baum, dessen Blätter Verweise auf einzelne Testergebnisse sind (vgl. Abb. 1). An inneren Knoten können verschiedene Aggregationen (Summe, Maximum, Minimum) definiert werden.

3.3 Aufgabe (das *task*-Element)

Das *task*-Element aus Abb. 1 wird aus Platzgründen nicht vollständig beschrieben. Die hier beschriebenen Änderungen gegenüber ProFormA 1.1 ergaben sich – über die in den Abschnitten 3.1 und 3.2 beschriebenen Aspekte hinaus – vor allem aus Anforderung R6.

Das *description*-Element, das die Aufgabenstellung enthält, erlaubt nun flexibel alle gültigen HTML Elemente anstatt einer Teilmenge. Zusätzlich gibt es ein neues *internal-description*-Element, das z. B. die Elemente *model-solution* und *file* mit näheren, nur für Lehrende sichtbaren Informationen ausstattet. Durch das *submission-restrictions*-Element lassen sich Restriktionen für von Studierenden einzureichende Dateien definieren (z. B. Vorgabe von Dateinamen). Diese können unter anderem in Form eines regulären Ausdrucks angegeben werden, für den in ProFormA 2.0 POSIX ERE als Standard gilt.

In ProFormA 1.1 wurde für jede Datei eine Dateiklasse zur Verdeutlichung des Verwendungszwecks angegeben (z. B. *template*, *library*). Da sich die Dateiklassen hierfür zunehmend als unzureichend erwiesen, werden in ProFormA 2.0 stattdessen drei neue Attribute verwendet. Diese geben für jede Datei an, ob sie für die Nutzung durch einen Grader bestimmt ist (*used-by-grader: yes, no*), wie ein LMS mit der Datei umgehen soll (*usage-by-lms: edit, display, download*) und ob sie für Studierende sofort, gar nicht oder erst zu einem späteren Zeitpunkt zur Verfügung gestellt werden soll (*visible: yes, no, delayed*). Im letztgenannten Fall muss eine Lehrkraft den Zeitpunkt (z. B. unmittelbar nach Einreichungsfrist einer Lösung zu einer Aufgabe) über ihr LMS festlegen, das dann zu diesem Zeitpunkt die jeweilige Datei (z. B. eine Musterlösung) den Studierenden bereitstellt. Durch die genannten Attribute lässt sich z. B. je nach Bedarf eine Datei für Studierende als zu nutzende Code-Bibliothek klassifizieren (*used-by-grader*: *yes*, *visible*: *yes*, *usage-by-lms*: *download*), die über ein LMS als Download zur Verfügung steht, oder als Ausfüllvorlage, die Studierende im optimalen Fall direkt im LMS editieren (*used-by-grader*: *no*, *visible*: *yes*, *usage-by-lms*: *edit*).

3.4 Einreichung (das *submission*-Element)

Eine Einreichung (*submission*) setzt sich aus einer studentischen Aufgabenlösung in Form von Dateien sowie aus weiteren, für Grader nützlichen Informationen zusammen. Da Grader eine Lösungsabgabe gegen eine konkrete Aufgabe bewerten, besitzt die Einreichung einen Verweis auf die Aufgabe, der in unterschiedlichen Formen angegeben werden

kann, um den Anforderungen unterschiedlicher Grader zu genügen (R1). Eine Aufgabe kann damit entweder als externe, für einen Grader lokalisierbare Ressource (bspw. in einem Repository im Internet), oder als Bestandteil der Einreichung angegeben werden. Letzteres ermöglicht wahlweise das Einbetten der Aufgabe als XML-Element oder als *file*-Bestandteil der Einreichung gemäß Abschnitt 3.1. Da Aufgaben über eine eindeutige, globale ID verfügen, kann eine Einreichung alternativ nur die Aufgaben-ID referenzieren, falls Grader ein internes Caching von Aufgaben über die ID vornehmen.

Eine Einreichung enthält abgabespezifische Informationen wie Datum und Uhrzeit der Abgabe, optionale Nutzerkennungen der einreichenden Studierenden (bei Einzel- und Gruppenabgaben) sowie LMS-spezifische Daten wie LMS-URL und Kurs-ID. Grader können diese Angaben bspw. dazu nutzen, um Verstöße bei Plagiatsprüfungen auf betroffene Nutzer zurückzuführen. Optional können noch weitere, beliebige Meta-Informationen zu der Einreichung bereitgestellt werden (R6).

Es kann aus technischen Gründen notwendig sein, einem Grader mitzuteilen, in welcher Form ein LMS das Feedback einer Antwort (*response*) verarbeiten und darstellen kann (R2). Das *result-spec*-Element steuert den Detailgrad und die Struktur des erwarteten Feedbacks. Des Weiteren muss das vom LMS verarbeitbare Format der Antwort (als in einer ZIP verpackten oder unverpackten XML-Datei) spezifiziert werden. Anschließend kann die von Studierenden bevorzugte, natürliche Sprache für Feedback-Texte festgelegt werden (Abschnitt 3.1).

Ein weiterer Bestandteil der Einreichung ist das anpassbare Bewertungsschema (*grading-hints*, Abschnitt 3.2). Aufgabenautoren veröffentlichen Aufgaben mit einem vordefinierten Bewertungsschema, das nicht zwangsläufig den Vorstellungen der die Aufgaben einsetzenden Lehrenden entsprechen muss. Das *submission*-Element erlaubt Lehrenden, das voreingestellte Bewertungsschema in einem LMS wunschgemäß anzupassen (R5, Abb. 1). Durch die Anpassung des Bewertungsschemas bildet sich automatisch eine neue Aufgabenkonfiguration, die im Rahmen des jeweiligen LMS-Kurses aktiv wird (R4). Die originale Aufgabe mit dem ursprünglichen vom Aufgabenautor oder von der Aufgabenautorin intendierten Bewertungsschema bleibt unbeeinflusst.

3.5 Antwort (das *response*-Element)

Eine Antwort (*response*) ist ein gradergeneriertes Bewertungsergebnis zu einer studentischen Lösungseinreichung (*submission*). Sie kann die resultierende Gesamtpunktzahl, die Punktzahl für jedes Testergebnis, das textuelle Feedback und Dateien, die für das Verständnis des Bewertungsergebnisses relevant sein können (vgl. Abb. 1) umfassen. In welcher Struktur ein Grader das Feedback bereitstellt, spezifiziert das LMS mit der Einreichung (Abschnitt 3.4, Abb. 1 *submission/result-spec*). Feedback kann damit als vorformatiertes HTML-Fragment strukturiert werden, das bereits alle Informationen (Texte, Punkte und eingebettete Dateien) umfasst und von einem technisch limitierten LMS dargestellt wird. Alternativ wird das Feedback als eine Sammlung von mehreren, einzelnen Testschritten zugeordneten Feedback-Einträgen strukturiert (vgl. R3 und Abb. 1), bei der

die Aufbereitung und Darstellung dem LMS überlassen bleibt. Ein Feedback-Eintrag setzt sich aus einem Titel, dem Text (formatiert als Klartext oder HTML-Fragment) und den für das Feedback relevanten Dateien zusammen, die das LMS als Download-Links darstellen kann. Darüber hinaus gliedern Grader das Feedback nach hierarchischen Log-Leveln (*Debug, Info, Warning, Error*). Das Format sieht vor, dass mehrere Feedback-Einträge einem einzelnen Testschritt zugeordnet werden können, weshalb sich Feedback-Einträge für einen Testschritt mit unterschiedlichen Log-Leveln nicht ausschließen müssen. Sollte ein Testschritt fehlschlagen (bspw. wegen eines in der Einreichung enthaltenen Syntaxfehlers im Quellcode), können diesem Testschritt mehrere Feedback-Einträge zugewiesen werden, bspw. ein *Info*-Eintrag mit der Meldung, dass der Testschritt gescheitert ist, und ein *Error*-Eintrag, der den detaillierten Kompilierfehler aufführt. Lehrende können durch eine entsprechende Konfiguration im LMS steuern, welche Log-Level des Feedbacks Studierende einsehen können (R2). Erkennt ein Grader oder eine Middleware einen Fehler, der auf das Gradingsystem statt auf die studentische Lösungseinreichung zurückzuführen ist, setzen sie ein internes Flag in der Antwort, damit das LMS den jeweiligen Einreichungsversuch invalidiert und betroffene Studierende keinen Einreichungsversuch verlieren.

Feedback für Lehrende und Studierende kann inhaltlich variieren, wenn Grader Feedback mit zusätzlichen, für Lehrende nützlichen Informationen anreichern, die für Studierende keinen didaktischen Nutzen aufweisen. Grader stellen daher das gesamte Feedback einer Antwort in zweifacher Ausführung bereit, einmal für Studierende und einmal für Lehrende (R2). Grader können die beiden Feedback-Teile bspw. inhaltlich disjunkt gestalten. Oder sie erstellen das Lehrendenfeedback als Obermenge des Studierendenfeedbacks. Andere Gestaltungen sind denkbar. Das LMS präsentiert Studierenden nur das studierendenspezifische Feedback, während Lehrende sowohl ihr eigenes, als auch das studierendenspezifische Feedback einsehen können.

4 ProFormA-Kompatibilität

Das ProFormA-Format wird seit mehreren Jahren an verschiedenen Hochschulen eingesetzt. ProFormA 2.0 wird durch die folgenden Software-Projekte zumindest prototypisch unterstützt: Graja, GATE, ein Moodle-Plugin mit Praktomat-Grader an der Ostfalia und Stud.IP. GATE ist ein hybrides System, das sowohl LMS- als auch Grader-Funktionalität beinhaltet und an der TU Clausthal entwickelt wird [Mü17]. Die anderen Systeme verwenden entweder Moodle (ein international bekanntes LMS) oder Stud.IP (ein an deutschen Hochschulen verbreitetes LMS). Graja wird an der Hochschule Hannover (HsH) entwickelt und über die spezielle Middleware Grappa an Moodle angebunden [Ga17]. An der Ostfalia-Hochschule[7] wurde der Praktomat[8] um eine ProFormA-Schnittstelle erweitert, so dass er als reiner Grader nutzbar ist [Ro17]. Eine LMS-seitige Anbindung existiert für Moodle (ProFormA-Moodle-Plugin) und durch das Stud.IP-Aufgaben-

[7] https://github.com/elearning-ostfalia
[8] https://github.com/KITPraktomatTeam/Praktomat

Tool „Vips“ für Stud.IP. Vips bietet Funktionen zur Erstellung und Verwaltung von Programmieraufgaben und war ursprünglich ein eigenständiges Werkzeug, das jetzt aber Auswertungen von ProFormA-kompatiblen Gradern verarbeiten kann. An der Ostfalia wird außerdem ein Editor entwickelt, mit dem man ProFormA-Dateien unabhängig von einem LMS oder Grader bearbeiten kann [Pr17].

Von den verschiedenen Projekten werden zum Teil unterschiedliche Aspekte von ProFormA 2.0 unterstützt. Alle Systeme unterstützen Java-Programmieraufgaben (Kompilierung und JUnit-Tests), bei denen die Test-Dateien in die *task* eingebettet werden, die studentische Einreichung mitgesendet wird und das Feedback in die einzelnen Tests untergliedert als *response* zurückgesendet wird. Alle Systeme außer GATE unterstützen auch Java-Checkstyle. Andere Sprachen (z. B. Python) und Testtypen (z. B. Regexp-Test, Java-PMD) werden nur von einzelnen Systemen bereitgestellt. Ob die Dateien als XML oder als gezipptes XML erstellt und gelesen werden, wird recht unterschiedlich gehandhabt. Auch bezüglich der Darstellung von LMS-Bedingungen (z. B. die maximale Dateigröße der studentischen Einreichung) gibt es Unterschiede. Einige Systeme speichern diese Informationen im ProFormA-Format, andere betrachten dies als nur zum LMS gehörig und bilden sie nicht im Format ab. Auch die Art und Weise, auf die die studentische Einreichung in die *submission* eingebettet wird, ist noch uneinheitlich. Es fällt auch auf, dass die Schwerpunkte in den einzelnen Systemen unterschiedlich gewählt werden. An der Ostfalia wird beispielsweise nur eine sehr rudimentäre Fassung des Bewertungsschemas umgesetzt während an der HsH der volle Umfang implementiert ist. Dagegen fehlt das Bewertungsschema in anderen Systemen komplett.

Da ProFormA 1.1 als Obermenge der Funktionalitäten der verschiedenen Systeme entstanden ist, erlaubt ProFormA 2.0 in einigen Bereichen mehrere sich gegenseitig ausschließende Varianten, aus denen nur mindestens eine implementiert werden muss. Zum Beispiel erwarten einige Systeme, dass Dateien in das XML-Format eingebettet sind, andere verarbeiten sie als Anhänge in einer ZIP-Datei. Bisher müssen Aufgaben im ProFormA-Format daher eventuell überarbeitet werden, nachdem sie von einem System exportiert und bevor sie in ein anderes System importiert werden sollen. Es gibt verschiedene Möglichkeiten, dieses Problem zu lösen, die hier in absteigender Komplexität aufgeführt werden: 1) alle Systeme müssen alle ProFormA 2.0-Aspekte lesen können, auch wenn sie selber nur eine Variante erzeugen. 2) Systeme kommunizieren über eine Middleware, die alle ProFormA 2.0 Aspekte unterstützt und die die XML-Dateien entsprechend konvertiert. 3) eine verpflichtende Untermenge wird definiert, die ProFormA 2.0.-kompatible Systeme unterstützen müssen. 4) Systeme veröffentlichen ein ProFormA-Profil, das beschreibt, was sie lesen und produzieren können. Ob zwei Systeme Aufgaben austauschen können, lässt sich dann anhand der Profile im Voraus bestimmen.

5 Zusammenfassung und Ausblick

In diesem Beitrag wurden Änderungen und Erweiterungen des ProFormA-Aufgabenaustauschformats zur Vereinfachung und zur besseren Unterstützung vor allem von LMS und Middleware vorgestellt, die aufgrund damit einhergehender neuer Anforderungen notwendig waren. Das Format definiert zu diesem Zweck nun Standards für Einreichungen (*submission*) und Antworten/Feedback (*response*) zu studentischen Abgaben, bietet Support für die Nutzung mehrerer Sprachen und enthält die Definition eines komplexen Bewertungsschemas. Zudem wurden Änderungen an Elementen, die eine Datei repräsentieren, vorgenommen. Zum Schluss wurde diskutiert, welche Bestandteile des Formats von einem System unterstützt werden müssen, um kompatibel zum ProFormA-Format zu sein.

Zukünftig soll der Fokus auf dem verstärkten Praxiseinsatz liegen. Zu diesem Zweck muss insbesondere die Anbindung an ein Repository vorangetrieben werden, über das Programmieraufgaben im ProFormA-Format bereitgestellt und ausgetauscht werden können. Hierzu ist es u. a. wichtig festzulegen, welche Metadaten (Abb. 1 *meta-data*) zu einer Aufgabe offiziell vom Format unterstützt werden. In diesem Zusammenhang wird die Nutzung einer Teilmenge des Learning Objects Metadata (LOM) Standards in Betracht gezogen. Zusätzlich muss abschließend definiert werden, ab wann ein System als ProFormA-kompatibel eingestuft werden kann.

Literatur

[Bo17] Bott, O.; Fricke, P.; Priss, U.; Striewe, M. (Hrsg.): Automatisierte Bewertung in der Programmierausbildung. Digitale Medien in der Hochschullehre, ELAN e.V., Waxmann, 2017.

[Ga17] Garmann, R.: Der Grader Graja. In [Bo17].

[Ga19] Garmann, R.: Ein Format für Bewertungsvorschriften in automatisiert bewertbaren Programmieraufgaben. In: Pinkwart, N.; Konert, J. (Hrsg.): DeLFI 2019 - Die 17. Fachtagung Bildungstechnologien, 2019.

[Mü17] Müller, O.; Strickroth, S.: Der Grader GATE. In [Bo17].

[Pr17] Priss, U.; Borm, K.: An Editor for the ProFormA Format for Exchanging Programming Exercises. In: Dritter ABP Workshop, Potsdam 2017.

[Ro17] Rod, O.: Integration mithilfe der Middleware ProFormA-Server. In [Bo17]

[St15] Strickroth, S. et al..: ProFormA: An XML-based exchange format for programming tasks. eleed, 11, 2015. https://eleed.campussource.de/archive/11/4138

[St17] Strickroth, S.; Müller, O.; Priss, U.: Ein XML-Austauschformat für Programmieraufgaben. In [Bo17].

Ein Datenformat zur Materialisierung variabler Programmieraufgaben

Robert Garmann[1]

Abstract: Automatisiert bewertbare Programmieraufgaben dienen Studierenden zum Üben von Programmierfertigkeiten. Um mehrere verschiedene, denselben Stoff abdeckende Aufgaben zu gewinnen, lässt sich eine Aufgabe durch Einführung von Variationspunkten variabel gestalten. Die entstehende Aufgabenschablone ist Ausgangsbasis der sog. Materialisierung, der automatischen Generierung konkreter Aufgaben. Dieser Beitrag stellt ein Datenformat vor, das die automatische Materialisierung auf verschiedenen Systemen in verschiedenen Programmiersprachen ermöglichen soll. Das Datenformat gestattet Lernmanagementsystemen, variable Programmieraufgaben bei gleichzeitiger Unkenntnis des eingesetzten Autobewerters zu unterstützen.

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

1 Einleitung

Die Verfügbarkeit mehrerer verschiedener, denselben Stoff abdeckender Programmieraufgaben, kann durch Generierung mehrerer Aufgabenvarianten aus einer Aufgabenschablone erreicht werden. Anwendungsfälle wie Plagiatbekämpfung, Zusatzübungen, Adaptivität sowie didaktische Aspekte werden etwa in [OG17] diskutiert.

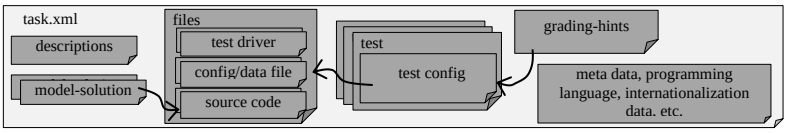

Abb. 1: Elemente einer ProFormA-Aufgabe

ProFormA[2] beschreibt eine automatisiert bewertbare, nicht variable Programmieraufgabe (*task*) als Ansammlung verschiedener Artefakte (s. Abb. 1 und [St15]). Neben dem Text der Aufgabe (*description*) werden automatisch durchführbare *Tests* definiert, die u. a. durch Dateien (*files*) konfiguriert werden. Musterlösungen (*model-solutions*) können ebenfalls angegeben werden. Ein Bewertungsschema (*grading-hints*) beschreibt, wie aus Testergebnissen ein Bewertungsergebnis errechnet wird [Ga19a]. Die *files* werden entweder direkt in eine XML-Datei eingebettet oder es wird ein Zip-Archiv erstellt, welches die task.xml-Datei und weitere Dateien enthält.

[1] Hochschule Hannover, Fakultät IV Wirtschaft und Informatik, Ricklinger Stadtweg 120, 30459 Hannover, robert.garmann@hs-hannover.de
[2] https://github.com/ProFormA/proformaxml

doi:10.18420/abp2019-7

Programmieraufgaben lassen sich durch Einführung von *Variationspunkten* (*vp*) zu einer Aufgabenschablone umgestalten [Ga18]. Beispiele für Variationspunkte reichen von einfachen Bezeichnern über konkrete Datenwerte bis hin zu Konstrukten der verwendeten Programmiersprache und Bewertungsaspekten (vgl. Tab. 1). Daneben kann die Aufgabenschwierigkeit als weiterer Variationspunkt vorgesehen werden.

	Java	**SQL**
Bezeichner	Programmierung einer Funktion zur Berechnung des Kreisflächeninhalts **vp:** Name der Funktion	Selektion einer Tabellenspalte einer gegebenen Tabelle **vp**: Name der Tabellenspalte
Funktion / Algorithmus	Berechnung des Flächeninhalts einer geometrischen Form **vp:** Vorgegebene Form (Kreis, …)	Aggregation einer Tabellenspalte **vp**: Aggregatfunktion
Datenwerte	Programmierung einer Funktion zur Berechnung des Kreisflächeninhalts **vp**: Vorgabe, ob und wie ungültige Kreisradien zu behandeln sind	Selektion der Datensätze mit vorgegebenem Attributwert **vp**: Der vorgegebene Attributwert
Sprachkonstrukt	Programmierung einer Schleife **vp**: Vorgabe des Schleifentyps (for- oder while-Schleife)	Verknüpfung zweier Tabellen **vp**: Implementierung durch JOINs oder durch Unterabfragen
Bewertungsaspekt	**vp**: Bewertungsgewicht des Programmierstils (nicht bewertet, bewertet mit Gewicht w, …)	

Tab. 1: Beispiele für Variationspunkte (vp)

Die Generierung konkreter Aufgaben aus einer Schablone nennt man auch Materialisierung [Ha12]. Dieser Beitrag schlägt ein Datenformat[3] für die Spezifikation des Materialisierungsprozesses von Aufgabenschablonen vor und nutzt dazu das ProFormA-Format sowie ein zuvor entwickeltes Format zur Beschreibung von Variationspunkten [Ga18].

2 Grundlegendes

2.1 Anforderungen und Vorgehen

Typischerweise kommunizieren Lehrkräfte und Studierende über ein Lernmanagementsystem (*LMS*) mit dem Autobewerter (*Grader*). Die Lehrkraft legt eine Aufgabe an. Studierende reichen Lösungsversuche ein und erhalten Feedback. Meist vermittelt eine spezielle Middleware zwischen LMS und Grader. Interoperabilität der beteiligten Systeme bzgl. *normaler* Programmieraufgaben stellt das ProFormA-Format her. Mit Hilfe zufällig oder gezielt gewählter Werte der Variationspunkte soll aus einer Aufgabenschablone durch Materialisierung eine konkrete Aufgabe entstehen. Sowohl die Auswahl der Werte der Variationspunkte als auch die Materialisierung soll weitgehend von jedem der beteiligten Systeme erledigt werden können. Insb. soll das LMS eine konkrete Aufgabe ohne Beteiligung des Graders erzeugen können. Wenn in Sonderfällen der Grader beteiligt werden muss, soll das gesuchte Datenformat hierfür spezielle durchzuführende Materialisierungsschritte separat ausweisen, so dass LMS und Grader die Materialisierung kooperativ durchführen können. Das Format soll Interoperabilität von LMS, Middleware

[3] Ein XML-Schema nebst Whitepaper ist unter https://github.com/ProFormA/varproformaxml verfügbar.

und Grader für variable Aufgaben unterstützen, so dass bspw. ein System der Aufrufkette ausgetauscht[4] werden kann, ohne die Nutzung der Aufgaben wesentlich zu beeinträchtigen. Verschiedene Spezifikationsarten variabler Artefakte sind dabei zu unterstützen. Sowohl mit Platzhaltern oder Fallunterscheidungen versehene Artefakte als auch mehrfach in unterschiedlicher Ausprägung vorliegende Artefakte sind denkbar.

Im Vorgehen identifizierten wir zuerst Variationspunkte in konkreten Aufgaben verschiedener Grader. Anhand der in Tab. 1 dargestellten Beispiele wurden dann mehrere Aufgaben für den Grader Graja[5] variabel gestaltet. Auf dieser Grundlage wurde ein abstraktes Datenformat geschaffen, welches sowohl die in den Aufgaben vorhandene Variabilität als auch die zur Herstellung einer konkreten Aufgabe notwendigen Materialisierungsschritte beschreibt. Die Grader-Unabhängigkeit des entwickelten Datenformats wurde argumentativ gestützt. Es wurde eine von Graja unabhängige Java-Bibliothek geschaffen, die das Datenformat, die Auswahl von Werten für Variationspunkte und die Materialisierung implementiert. Der gewählte Forschungsansatz lässt sich dem Design-Based Research zuordnen. Einige der entstandenen variablen Graja-Aufgaben wurden im Wintersemester 2018/19 und im Sommersemester 2019 zufallsbasiert im formativen, semesterbegleitenden Assessment an ca. 80 Studierende zweier aufeinander aufbauender Grundlagen-Lehrveranstaltungen zur Java-Programmierung an der Hochschule Hannover ausgegeben. Der Einsatz wurde nicht bzgl. didaktischer Effekte evaluiert, sondern es ging hierbei i. w. darum, die technische Machbarkeit eines Einsatzes in dem Grader-unabhängigen LMS Moodle unter Beweis zu stellen. Die bisherigen Ergebnisse wurden noch nicht auf Aufgaben weiterer Grader angewendet.

2.2 Verwandte Arbeiten

In der Produktlinienentwicklung (PLE) werden Variabilitätsmodelle genutzt, um Varianten und deren Constraints darzustellen. Beispielhaft seien die Common Variability Language (CVL, [Ha12]) und das Orthogonal variability model [PBL05] genannt. Die PLE hat mit variablen Programmieraufgaben gemeinsam, dass Variablen, Wertausprägungen und teilweise komplexe Randbedingungen formuliert werden müssen und dass die Variablen und ihre Werte mit den einzelnen Artefakten des Produkts in Verbindung gebracht werden müssen. Die Möglichkeiten solcher Ansätze gehen über die Erfordernisse einer variablen Programmieraufgabe hinaus. Zum einen weisen variable Programmieraufgaben i. d. R. deutlich weniger Variationspunkte als komplexe Produkte der PLE auf. Zum anderen kann in variablen Programmieraufgaben die Materialisierung einer Schablone (PLE-Begriffe resolution und materialization [Ha12] bzw. binding und realization [PBL05]) sehr spezifisch auf das spezielle ProFormA-Aufgabenformat zugeschnitten werden, während im PLE-Fall ein weites Spektrum von Artefakten (Anforderungen, Entwürfe, Implementierungen, Tests) berücksichtigt werden muss.

[4] Die Austauschbarkeit des Graders ist unabhängig vom Aspekt der Variabilität durch die Kompatibilität des Graders zu den in der Aufgabe definierten Bewertungsmethoden und Programmiersprachen limitiert.
[5] http://graja.hs-hannover.de

2.3 Platzhalter und Template-Frameworks

Eine Schablone enthält meist mit Platzhaltern versehene Artefakte. Tab. 2 zeigt einen Aufgabentext mit dem Variationspunkt-Platzhalter *shape*. Die konkreten Texte wurden durch das mustache[6]-Framework generiert. Es können weitere Artefakte (Testtreiber, Musterlösungen) Platzhalter besitzen. Nicht nur der Inhalt, sondern auch der Dateiname des an Studierende verteilten Coderahmens enthält einen Platzhalter. Außer einer einfachen Wertersetzung sind zudem Schleifen-Konstrukte üblich, die ein Inhaltsfragment mehrfach mit verschiedenen, aus einer Liste gespeisten Wertbelegungen erzeugen. Im Falle einer leeren Liste sind auch wertabhängige Löschungen von Fragmenten möglich.

	Aufgabentext	**Coderahmen**
Schablone	Given an interface Shape with an area method, write a subclass §(shape)§ accepting Double[] as a constructor parameter.	class §(shape)§ implements Shape { double area() { return 0.0; } /* TODO: complete this template */ } *file §(shape)§.java*
Konkrete Aufgabe	Given an interface Shape with an area method, write a subclass Circle accepting Double[] as a constructor parameter.	class Circle implements Shape { double area() { return 0.0; } /* TODO: complete this template */ } *file Circle.java*
	Given an interface Shape with an area method, write a subclass Rectangle accepting Double[] as a constructor parameter.	class Rectangle implements Shape { double area() { return 0.0; } /* TODO: complete this template */ } *file Rectangle.java*

Tab. 2: Schablone mit materialisierten Versionen

2.4 Datenmodell der Variabilität

Ein LMS- und Grader-übergreifend einsetzbares Datenmodell der in einer Schablone angelegten Variabilität stellt [Ga18] vor. Dieses – derweil um ein Tabellenkonzept erweiterte und kleinen Umbenennungen unterworfene – Datenmodell kann hier aus Platzgründen nicht erschöpfend wiederholend beschrieben werden. Für Details verweisen wir auf [Ga19b]. Hauptelemente des Datenmodells sind *vp* (Variationspunkte), *v* (Varianten = Werte von Variationspunkten), *cvp* und *cv* (Tupel von Variationspunkten bzw. Varianten, c=composite). Der Raum aller gültigen Wertbelegungen (*cv*) der Variationspunkte (*cvp*) einer Schablone wird durch das *var-spec*-Element mit den Grundoperationen kartesisches Produkt, Vereinigungsmenge und Ableitungsfunktion spezifiziert.

3 Materialisierung

Dieser Abschnitt beschreibt eine Spezifikation der Materialisierung einer Aufgabenschablone, welche danach in Abschnitt 4 diskutiert wird. Die Artefakte der Schablone werden um Zusatzdaten ergänzt, die für gegebene Variationspunktwerte definieren, wie aus dem Schablonenartefakt ein Artefakt einer konkreten Aufgabe generiert wird. [PBL05] nennt diese Zusatzdaten *artefact dependency*. Wir betrachten Schablonenartefakte als Gegenstände der Materialisierung. Eine Materialisierung ist die Anwendung einer Mate-

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

rialisierungs*methode* auf ein zu materialisierendes *Artefakt*. Nach Anwendung aller Materialisierungen entsteht aus der Schablone eine konkrete Aufgabe. Die o. g. Zusatzdaten lassen sich als eine Menge { m : m=(ma, mm) } von Materialisierungen m schreiben, wobei jedes m ein Paar eines Materialisierungsartefakts ma und einer Materialisierungsmethode mm ist. Diese Grundidee wurde in [Dr18] entwickelt und danach verfeinert.

3.1 Die Datei tpl.xml

Platzhalter in der task.xml-Datei können das task.xml-Artefakt der Schablone in eine ungültige XML-Datei verwandeln. Variationspunkte, Werte und Materialisierungen werden daher separat abgelegt. Eine Schablone sei ein Zip-Archiv, das neben der üblichen task.xml-Datei und etwaiger weiterer, von task.xml referenzierter Artefakte dazu eine tpl.xml-Datei mit dem in Abb. 2 gezeigten Inhalt umfasst (tpl=template). *var-spec* und *default-value* spezifiziert die Variabilität und eine Standard-Wertkombination (vgl. Abschnitt 2.4), *mat-spec* die Materialisierungen. Im Folgenden soll das Objekt *mat-spec*, welches die Materialisierung der Schablone spezifiziert, genauer beschrieben werden.

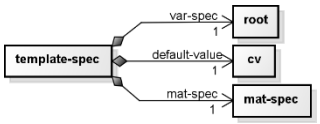

Abb. 2: Inhalt von tpl.xml

3.2 Datenmodell der Materialisierung

Die Trennung einer Materialisierung in Methode und Artefakt gestattet, häufig anzuwendende Methoden wie das Framework mustache nur einmal zu spezifizieren und dennoch auf verschiedene Artefakte anzuwenden. Abb. 3 verknüpft sogar mehrere *artifact-ids* und mehrere *method-ids* in einer *materialization*, um die gleichförmige Anwendung vieler Methoden auf viele Artefakte redundanzarm zu beschreiben.

Ein in sicher jeder Schablone genutzter Artefakttyp ist *task-xml*, welcher den task.xml-Dateiinhalt bezeichnet. Einige Artefakttypen werden mit zusätzlichen Parametern spezifiziert. Bspw. definiert *attached-txt-file-contents* zusammen mit *path* die Inhalte der angegebenen Dateien. Ähnlich definiert *file-names* die Dateinamen. Statt *path* kann alternativ *file-id* auf Dateien verweisen, die task.xml als *file*-Elemente definiert.

Eine Methode nutzt die konkreten Werte der Variationspunkte und das Artefakt als Eingabe und erzeugt als Ausgabe ein ggf. verändertes Artefakt. Die von mustache realisierte Suchen-und-Ersetzen-Operation wird durch den Typ *mustache* spezifiziert, dessen Platzhalter-*prefix* und -*suffix* als Parameter genannt werden, wenn sie vom mustache-Default {{ bzw. }} abweichen. Wir haben das mustache-Framework als Standard-Methode auf-

genommen, weil dieses in vielen Programmiersprachen und Plattformen verfügbar ist, so dass alle LMS und Grader diese Methode ausführen können sollten.

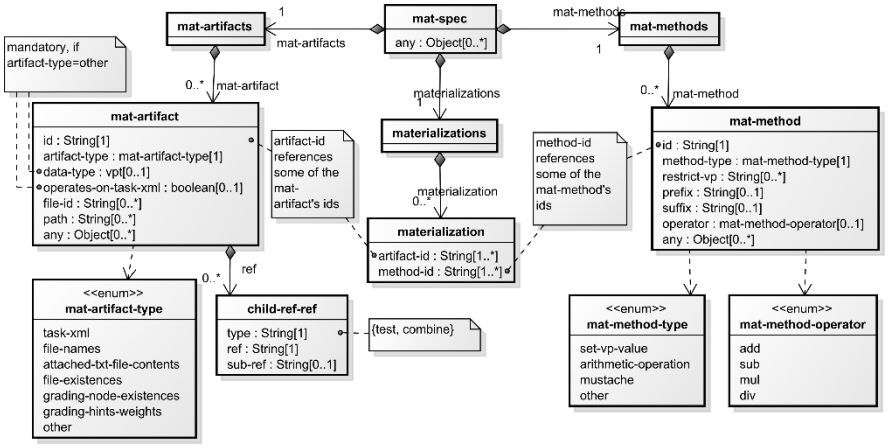

Abb. 3: Datenmodell der Materialisierung

Mit den genannten Artefakttypen und dem Methodentyp *mustache* lassen sich bereits viele Situationen bei der Definition einer Aufgabenschablone meistern. Einige zusätzlich benötigte Artefakttypen und -methoden motivieren wir nun durch Beispiele. Im ersten Beispiel soll eine in der Schablone stehende Datei via Materialisierung entfernt werden. Dazu verknüpft man den per *path* oder *file-id* parametrierten Artefakttyp *file-existences* mit dem Methodentyp *set-vp-value*, der mittels *restrict-vp* auf einen Variationspunkt booleschen Typs bezogen wird. Hat der Variationspunkt den Wert false, so wird die Existenz der betreffenden Dateien durch Anwendung der Methode auf „nicht existent" gesetzt – die Dateien werden gelöscht. Im zweiten Beispiel soll das Bewertungsgewicht eines Tests abhängig vom Wert eines Variationspunktes erhöht oder verringert werden. Dazu verknüpft man die mit einem *operator* und mit *restrict-vp* parametrierte Methode *arithmetic-operation* mit dem mit *child-ref-ref*-Parametern versehenen Artefakt *grading-hints-weights*. Im Zuge der Materialisierung wird dann das der arithmetischen Operation unterworfene Bewertungsgewicht in die generierte Aufgabenvariante geschrieben, wobei der zweite Operand vom Wert des bezeichneten Variationspunktes gespeist wird.

Manche Grader oder Programmiersprachen mögen über die oben spezifizierten Methoden und Artefakte hinaus zusätzliche Materialisierungsmechanismen benötigen. Ein typisches Beispiel ist die Materialisierung von Binärdateien, die für verschiedene Aufgabenvarianten jeweils durch einen Grader-spezifischen Prozess, bspw. eine Compilierung, neu generiert werden müssen. Um solche Grader-spezifischen Methoden abzubilden, nehmen wir den Erweiterungspunkt *mat-method-type.other* auf. In *mat-method.any* können Grader-spezifische Zusatzinformationen deklariert werden (bspw. Compiler-Flags), die die Grader-spezifische Materialisierung steuern. Auch für Artefakte ist es möglich, mit dem Erweiterungspunkt *mat-artifact-type.other* ein Grader-spezifisches Artefakt zu

deklarieren. Denkbar sind hier Angaben, die einen Abschnitt einer in der Schablone enthaltenen Quelltextdatei spezifizieren, wobei programmiersprachenspezifische, in *mat-artifact.any* transportierte Vokabeln zur Spezifikation des Abschnitts genutzt werden. Illustrierend kann etwa eine bestimmte Java-Methode innerhalb der Quelltextdatei unter Angabe ihres Namens und ihrer Parametertypen als Artefakt deklariert werden.

4 Diskussion und Ausblick

Das beschriebene Datenformat der Materialisierung variabler, auto-bewerteter Programmieraufgaben enthält keine Festlegungen auf bestimmte Grader oder Programmiersprachen und kann das existierende Grader-unabhängig einsetzbare ProFormA-Format um den Variabiliätsaspekt erweitern. Wertebereiche der Variationspunkte werden durch kaskadierend verschachtelte Operationen (Vereinigung, kartesisches Produkt, Ableitungsfunktion) spezifiziert. Effekte selektierter Variationspunktwerte auf Aufgabenartefakte (Aufgabentext, Dateien, Bewertungsschema, etc.) werden durch Paare von Materialisierungsartefakten und darauf angewendete Materialisierungsmethoden spezifiziert. Das Format ist Grader-spezifisch erweiterbar und liegt als XML-Schemadatei[3] vor.

Alle Methoden- und Artefakttypen sind in der in Abschnitt 2.1 genannten Bibliothek realisiert. Die Trennung von Methode und Artefakt gestattet, Grader-spezifische Methoden zu kapseln und an bereits implementierte Standard-Artefakte softwaretechnisch „anzudocken“. Umgekehrt können Implementierungen Grader-spezifischer Artefakte gekapselt und an bereits implementierte Standard-Methoden „angedockt“ werden. Beispielhaft haben wir die als Variationspunkt gegebene, maximal erreichbare Punktzahl über die Methode *set-vp-value* in ein neu implementiertes Artefakt gespeist, das die in den *grading-hints* der Aufgabenschablone definierten Bewertungsgewichte geeignet skaliert.

Falls einzelne Grader die bereits implementierten Standardmethoden und -artefakte nicht durch Aufruf verwenden können, werden in *mat-spec.any* beliebige Daten für eine gänzlich Grader-spezifische Materialisierung abgelegt. Beispielhaft haben wir einen Graja-spezifischen Materialisierungsschritt implementiert, der ein zusätzliches Artefakt aus allen Variationspunktwerten generiert und dessen Inhalt über in *mat-spec.any* transportierte Parameter gesteuert werden kann. Die weiter oben geforderte Eigenschaft des Datenformats, dass ein LMS die Materialisierung in Unkenntnis des Graders durchführen kann, wird nur insofern eingeschränkt, dass bei Nutzung der Erweiterungsmöglichkeiten für jeden Grader eine entsprechende Softwarekomponente zur Verfügung stehen muss, die sich um die Grader-spezifischen Teile der Materialisierung kümmert. Das Datenformat definiert hierbei eine Grader-unabhängige Komponentenschnittstelle.

Als Erweiterbarkeitsbeleg des Formats führen wir folgenden Gedanken an. Derzeit entfernen wir unbenötigte, in der Schablone vielfach ausgeprägte Musterlösungen aufwändig durch viele zusätzliche boolesche Variationspunkte, die wir mit *set-vp-value* auf *file-existences*-Artefakte anwenden. Um die Extra-Variationspunkte einzusparen, könnte man stattdessen einen speziellen Artefakttyp *model-solution-existences* erfinden, der mit

den Ids aller Musterlösungen parametriert wird, und auf diesen eine neu zu erfindende Methode *javascript* anwenden, die in einer Javascript-Funktion aus den gegebenen Ids die zu den Variationspunktwerten passenden Ids selektiert und als neues Artefakt liefert.

Ein offener Punkt ist derzeit, ob die generierte Aufgabenvariante die Variationspunkte und deren selektierte Werte enthalten soll. Als Ablageort bietet sich das ProFormA-Element *meta-data* an. Hierdurch würden späte Materialisierungen zur Laufzeit des Graders möglich. Statt Testtreiber durch Platzhalter syntaktisch von der Programmiersprache zu entfremden kann man dann alternativ Platzhalternamen exklusiv in syntaktisch vom Unittestframework erlaubten Literalen unterbringen. Unklar ist hierbei noch eine Graderunabhängige Spezifikation des zur Grader-Laufzeit erfolgenden Zugriffs. In [Ga19b] wird eine mögliche Umsetzung für Graja demonstriert.

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

Literatur

[Dr18] Drangmeister, R.: Entwurf und Implementierung eines Instanziierungsservices für variable Programmieraufgaben. Bachelorarbeit, Hochschule Hannover, urn:nbn:de:bsz:960-opus4-12207, 2018.

[Ga18] Garmann, R.: Ein Schnittstellen-Datenmodell der Variabilität in automatisch bewerteten Programmieraufgaben. Proceedings „SEELS – Software Engineering für E-Learning-Systeme“, Software Engineering Workshops 2018, CEUR, 2018.

[Ga19a] Garmann, R.: Ein Format für Bewertungsvorschriften in automatisiert bewertbaren Programmieraufgaben. In: Proc. DeLFI 2019.

[Ga19b] Garmann, R.: Ein Datenformat für variable Programmieraufgaben, Bericht, Hochschule Hannover, urn:nbn:de:bsz:960-opus4-13972, 2019.

[Ha12] Haugen, Ø.: Common Variability Language (CVL) – OMG Revised Submission. OMG document ad/2012-08-05, 2012.

[OG17] Otto, B.; Goedicke, M.: Auf dem Weg zu variablen Programmieraufgaben: Requirements Engineering anhand didaktischer Aspekte. In: Proc. of the ABP 2017. CEUR, urn:nbn:de:0074-2015-4, 2017.

[PBL05] Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, 2005.

[St15] Strickroth, S. et al.: ProFormA: An XML-based exchange format for programming tasks. eleed e-learning & education, 11(1), 2015.

Einsatz einer Online-Programmierplattform in der Präsenzlehre – ein Erfahrungsbericht

Torge Hinrichs[1] und Axel Schmolitzky[2]

Abstract: Online-Programmierplattformen bieten umfangreiche Möglichkeiten, die individuellen Programmierfähigkeiten zu verbessern. Durch die Verwendung einer solchen Plattform in einer Präsenzveranstaltung konnte die Motivation der Teilnehmer stark erhöht werden. Es wurden aber auch etliche Defizite deutlich, die in zukünftigen Plattformen, die für die Hochschullehre zugeschnitten sind, adressiert werden sollten.

Keywords: Programmierwettbewerb, automatische Bewertung von Aufgaben, Softwaretechnik, Code-Qualität

1 Einleitung

In den Bachelor-Studiengängen der Informatik an der HAW Hamburg hat die Programmierlehre hat seit jeher einen besonderen Stellenwert. Trotz dieser Betonung im gesamten Curriculum und obwohl nur wenige Module ohne Aufgaben auskommen, die Programmierkenntnisse erfordern, fällt es vielen Studierenden auch in höheren Semestern noch schwer, Lösungen für komplexe oder abstrakte Problemstellungen geeignet in Quellcode zu modellieren. Im fünften Semester sehen alle Bachelor-Studiengänge der Informatik ein unbenotetes *Projekt* vor, in dem die Studierenden weitgehend eigenständig eine umfangreichere Problemstellung bearbeiten sollen, idealerweise im Team. Den fachlichen Kontext bestimmen die Lehrenden der Informatik, indem sie mehrere Projekte anbieten, die von den Studierenden gewählt werden können.

In einem dieser Projekte haben wir die Teilnehmer angeleitet, an verschiedenen Trainingsaufgaben und Wettbewerben einer Online-Programmierplattform teilzunehmen. Neben der individuellen Verbesserung der Programmierfähigkeiten sowie der Fähigkeit, abstrakte Problemstellungen mit Software zu lösen, war ein weiteres Ziel, die Teilnehmer wiederholt zu fordern, die von ihnen gewählten Lösungsansätze einander seminaristisch vorzustellen. Durch diese regelmäßigen Präsenztermine sollte zusätzlich die Fähigkeit trainiert werden, über Entwürfe mit Kommilitonen zu diskutieren und von guten Designs anderer zu profitieren. Ein Kerngedanke war somit, dass komplexe Sachverhalte spielerisch thematisiert und als weniger schwerfällig wahrgenommen werden.

In diesem Erfahrungsbericht stellen wir unser Konzept für den Einsatz einer Online-Programmierplattform in einer Lehrveranstaltung sowie erste Erfahrungen damit vor.

[1] HAW Hamburg, Dep. Informatik, Berliner Tor 7, 20099 Hamburg, torge.hinrichs@haw-hamburg.de
[2] HAW Hamburg, Dep. Informatik, Berliner Tor 7, 20099 Hamburg, axel.schmolitzky@haw-hamburg.de

doi:10.18420/abp2019-8

2 CodinGame: eine Online-Programmierplattform

Da Programmierfähigkeiten durch die zunehmende Digitalisierung unserer Gesellschaft eine Schlüsselkompetenz für immer mehr Stellenbeschreibungen darstellen, erfreuen sich *Online-Programmierplattformen* immer größerer Beliebtheit. Die Nutzer dieser Plattformen können zwischen verschiedenen Modi wählen, um ihre Programmierfertigkeiten auf unterschiedlichen Niveaus zu trainieren. Dazu gibt es eine Fülle an Übungsaufgaben, die einfache Schleifen bis hin zu anspruchsvollen Algorithmen erfordern. Die Aufgaben werden in der Regel in einem beschreibenden Text formuliert und üblicherweise mit einem oder mehreren Beispielen genauer erläutert.

Für die Lösung ist die Wahl der Programmiersprache unerheblich, da die Eingaben der notwendigen Größen, wie zum Beispiel der aktuelle Zustand der Umgebung in einer Simulation, über die *Standard-Eingabe* eingelesen und die Ausgaben der Lösungen auf der *Standard-Ausgabe* ausgegeben werden müssen (*Standard-IO*). Die Plattform vergleich die Ausgaben mit erwarteten Ergebnissen. Die Aufgaben werden häufig als Minispiele aufgebaut, die rundenbasiert durchlaufen werden. Der Spielablauf wird dabei in einer *zweidimensionalen Darstellung* der Spielwelt visualisiert.

CodinGame[3] verfügt zusätzlich über verschiedene Wettkampfmodi, in denen Nutzer direkt gegeneinander antreten können. Im *Arena-Modus* programmiert der Nutzer das Verhalten eines Bots, der dann in einer Arena gegen die Implementierungen von anderen Nutzern antritt. Aufgrund der rundenbasierten Spielweise mit Visualisierung können Nutzer jede Runde eines jeden Spiels genau nachvollziehen. Die Nutzer werden für jedes Spiel in Ligen mit aufsteigendem Anspruch einsortiert. Dabei werden die Aufgaben teils erweitert und es können neue Funktionalitäten hinzukommen, die neue und komplexere Strategien ermöglichen bzw. erfordern.

CodinGame bietet Einzelpersonen umfangreiche und individuelle Unterstützung bei der Verbesserung ihrer Programmierfähigkeiten. Die eingesetzten Wettkampfkonzepte fördern für viele Personen den Spaß am Programmieren, die spielerischen Inhalte wirken durch die interaktive Visualisierung eines Spielverlaufs in Duell-Form stark motivierend.

Aus softwaretechnischer Sicht bestehen aber durchaus Defizite:

- Der „EVA-Flaschenhals“ (Standard-Eingabe, Verarbeitung, Standard-Ausgabe) bietet den Vorteil der Sprachunabhängigkeit, bewirkt aber einen Fokus auf algorithmisch geprägte Aufgaben; die Architektur der implementierten Lösungen ist irrelevant.
- Die Code-Qualität wird vernachlässigt; das Einhalten von Quellcode-Konventionen oder von Regeln des guten Entwurfs spielt keine Rolle.
- Die Plattformen ist auf Einzelpersonen ausgelegt: durch die integrierte Online-IDE, die die Eingabe in nur einem Textfenster gestattet, wird es erschwert, Lösungen im Team zu erarbeiten, zu modularisieren und zusammenzuführen.

[3] https://www.codingame.com/, letzter Abruf Juni 2019

Diese Defizite schmälern keinesfalls den Erfolg der Plattform, sind jedoch relevant für den Einsatz in einer Projekt-Lehrveranstaltung eines Informatik-Studiums.

3 Einsatz in einem Lehre-Projekt

Für die Pilot-Projektveranstaltung, in der wir den Einsatz einer Online-Programmierplattform testen wollten, hatten wir uns als Lernziel neben der Verbesserung der Programmierfähigkeiten gesetzt, dass die Studierenden besser darin werden sollen, sich über Softwareentwürfe gewinnbringend auszutauschen. Für das letztere Ziel schien uns CodinGame aufgrund der Visualisierung des Spielverlaufs besonders geeignet, weil diese den Studierenden die Analyse und das Verständnis unterschiedlicher Lösungen erleichtert. Der Umfang des Moduls *Projekt* im fünften Semester unserer Bachelor-Studiengänge beträgt 9 Credit Points, den studentischen Aufwand kalkulieren wir deshalb mit etwa 12 Stunden pro Woche. Für die Dozenten werden lediglich 3 LVS angerechnet, so dass die Kontaktzeit nur einen kleinen Anteil des studentischen Gesamtaufwandes ausmacht.

3.1 Projektablauf

Unser Projekt begann mit einer zweiwöchigen Einarbeitungsphase in CodinGame. Dazu haben die Studierenden in Einzelarbeit von uns ausgewählte Einstiegsaufgaben mit aufsteigender Komplexität gelöst, um sich mit den Besonderheiten der Plattform vertraut zu machen. Im nächsten Schritt wurde die Einzel- durch Teamarbeit ersetzt, indem Teams mit jeweils vier Personen gebildet wurden. Die Teamkomposition erfolgte zufällig; es gab lediglich die Maßgabe, dass in jedem Team verschiedene Studiengänge vertreten sind, weil die Programmierlehre in den ersten Semestern teilweise sehr unterschiedlich ist.

Die Teams mussten an einem von uns vorgegebenen Arenaspiel teilnehmen, mit dem Ziel, möglichst weit in den Ligen aufzusteigen. Da Teamarbeit durch die Plattform nicht vorgesehen ist, mussten sich die Teams selbst organisieren. Damit wir einen Überblick über den Fortschritt erhielten, haben wir wöchentliche StandUp-Meetings mit den einzelnen Gruppen durchgeführt, in denen diese den aktuellen Stand der Bearbeitung sowie den Anteil der jeweiligen Mitglieder am Fortschritt vorstellten. Den „Erfolg“ eines Teams konnten wir zusätzlich durch seinen automatisch ermittelten Liga-Rang bemessen; eine höhere Platzierung korrelierte systembedingt stark mit einer besseren Lösung.

Nach zwei Wochen wurde ein Plenumstermin durchgeführt, in dem die Gruppen ihre Strategien und verwendeten Algorithmen und Datenstrukturen in Vorträgen (30 Minuten) vorstellten. Anschließend erhielten die Gruppen erneut zwei Wochen Zeit (mit einem StandUp in der Mitte), um die beim Austausch gelieferten Anregungen in die eigene Lösung zu integrieren und idealerweise in die höchste Liga aufzusteigen. Abschließend gab es erneut einen Plenumstermin, diesmal zum Austausch in 15-minütigen Vorträgen über die jeweiligen Fortschritte.

Dieser Prozess wurde im Lauf des Semesters dreimal mit verschiedenen Arenaspielen durchlaufen. Des Weiteren haben die Studierenden nach Ende der Bearbeitung eines Spiels ihren Quellcode an eine andere Gruppe zum Code-Review weitergeben. Die Ergebnisse wurden auch im Plenum diskutiert.

3.2 Beobachtungen

Die Visualisierung des Spielverlaufs (bzw. der Aufgabenstellung bei den Übungsaufgaben) von CodinGame ermöglichte den Studierenden einen leichten Einstieg in die Plattform. Die implementierte Lösung kann im Detail untersucht und verstanden werden. Die automatische Prüfung der Aufgabe konnte dazu genutzt werden, direktes Feedback zu erhalten. Nach Bearbeitung einer Aufgabe haben sich die Studierenden häufig die Lösungen anderer Nutzer angesehen und diskutiert.

Über den Verlauf des Projekts konnten wir eine Änderung der Denkweise bzw. der Bearbeitungsweise der Studierenden beobachten. So arbeiteten sie beim ersten Spiel überwiegend „drauf los“ und hatten Probleme sich im Team zu koordinieren. Unübersichtlicher Quellcode und ein langsamer Aufstieg in den Ligen waren die Folge. Des Weiteren wurden Probleme in der Abstraktion der Problemstellung deutlich. Der erste Plenumstermin zwang die Studierenden, ihre Ansätze zu ordnen und in einen Vortrag zu bringen. Bei den folgenden Arenaaufgaben war ein deutlich strukturierterer Ansatz mit überlegten Architekturen erkennbar. Die Teams waren in der Lage, die Aufgabenstellung in Basisbestandteile zu zerlegen und Diskussionen über das Verhalten auf einer abstrakteren Ebene zu führen. Außerdem waren der Austausch bzw. das Review der Quellcodes anderer Gruppen nach Aussage der Studierenden besonders hilfreich. Das Konfrontieren mit anderen Denkweisen brachte die Studierenden dazu, stärker über ihre eigene Lösung zu reflektieren.

4 Fazit und Ausblick

Der Einsatz einer Online-Programmierplattform in einer Präsenzveranstaltung war aus unserer Sicht ein Erfolg. Die von uns verwendete Plattform CodinGame bietet eine attraktive Visualisierung der Lösungen und einen Arena-Modus, dessen „Programmieren gegen Andere“ stark motivierend wirkte. Der Wettbewerb und insbesondere das Präsentieren der eigenen Lösungsansätze brachten die Studierenden dazu, strukturierter über ihre Lösungen zu reflektieren. Ergänzt durch das systematische Peer-Feedback über den produzierten Quellcode konnte eine effektive Lernerfahrung geboten werden. Die automatische Überprüfung der Lösungen war ebenfalls für uns als Dozenten hilfreich, weil weniger Zeit für händische Prüfungen anfiel und der Schwerpunkt der Diskussion auf die Kern-Abstraktionen der Lösungen gelegt werden konnte. Das entwickelte Konzept wird als Blaupause für weitere Projekte dieser Art dienen.

Ansatz zur automatischen Generierung von Java-OOP-Aufgaben inkl. Bewertungsschemen

Ulf Döring[1]

Abstract: Im Fach „Algorithmen und Programmierung für Ingenieure“ werden an der TU Ilmenau noch papierbasierte Klausuren mit einem sehr geringen Multiple-Choice-Anteil geschrieben. Entsprechend hoch ist der Aufwand der händischen Korrektur. Zudem müssen selbst zum Zeitpunkt der Prüfung die meisten Studierenden noch als Programmieranfänger gesehen werden. Hierdurch führen Ansätze, welche compilierbaren Quellcode voraussetzen, sowohl beim Üben als auch bei Klausuren regelmäßig nicht zu angemessenen Bewertungen. Dieser Artikel beschreibt die Entwicklung eines Aufgabengenerators für einen bestimmten Aufgabentyp im Kontext von Java und OOP. Die automatische Generierung der Aufgaben soll in Bezug auf die zielgerichtete Vorbereitung auf die Prüfung den Studierenden Übungsmöglichkeiten bieten. Hinsichtlich der Klausurkontrolle sollen zudem auch Bewertungsschemen zur Anwendung bei der händischen Korrektur erzeugt werden.

Keywords: Ingenieurstudium; Java; Klassendefinition; Aufgabengenerator; Anfängerquellcode

1 Einführung

An der TU Ilmenau beginnen jährlich ca. 400 Studierende ihre Ausbildung in einem Ingenieurstudiengang, in dessen Rahmen sie die Lehrveranstaltung „Algorithmen und Programmierung für Ingenieure“ absolvieren. Ein grundsätzliches Problem gegenüber informatikbezogenen Studiengängen besteht in der oft fehlenden Motivation sowie einem durchschnittlich geringerem Vermögen, eine Programmiersprache zu erlernen und Algorithmen zu verstehen. Dementsprechend ist ein langsamerer Lernfortschritt typisch. Bezüglich des Erlernens einer Programmiersprache (für die Lehrveranstaltung wurde dazu Java ausgewählt) ist die Berücksichtigung der Probleme von Programmieranfängern mit den bereitgestellten Tools (insbesondere Eclipse) jedoch noch unzureichend. Von Seiten der Lehrenden wird hier ein entsprechend hoher Aufwand betrieben, um dieses Manko in Seminaren und Tutorien durch persönliche Hilfestellungen beim Finden und Beheben von Fehlern auszugleichen, siehe auch [SH15] zu Erfahrungen im Umgang mit Programmieranfängern. Um den Betreuungsaufwand zu senken, wurden bereits Angebote zum Üben typischer Verarbeitungsmuster für 1d-Felder ([DA19; DF17]) sowie zur Erstellung einfacher Java-Klassen entwickelt. Hilfestellungen werden bei diesen insbesondere durch das Aufzeigen der Beziehungen zwischen Aufgabenstellung und Beispiellösung sowie durch die Möglichkeit zur Simulation der Beispiellösung im Webbrowser gegeben. Der Generator für Aufgaben zur Erstellung von Java-Klassen soll nun erweitert werden. Ziel ist

[1] TU Ilmenau, IA/GDV, Helmholtzplatz 5, 98693 Ilmenau, ulf.doering@tu-ilmenau.de

doi:10.18420/abp2019-9

es dabei, sowohl die Studierenden beim Üben besser zu unterstützen, als auch die Qualität der Klausurkorrektur für derartige Aufgaben zu erhöhen.

2 Grundsätzliche Arbeitsweise des Aufgabengenerators

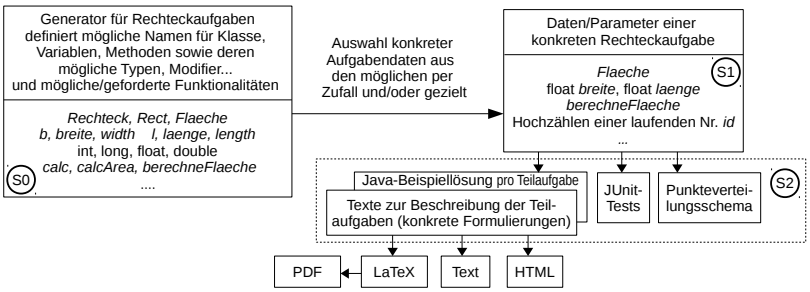

Abb. 1: Generierungsablauf am Beispiel eines Generators für Rechteckaufgaben

- Der Aufgabengenerator basiert auf spezifischeren Generatoren, welche jeweils eine grundsätzliche Aufgabenstruktur definieren (Stufe S0). Abb. 1 zeigt ein Beispiel.
- Eine solche Aufgabenstruktur wird in einer ersten Stufe (S1) durch die Vergabe konkreter Namen für die zu entwickelnde Klasse, für die Objekt- und Klassenvariablen und für die Methoden spezifischer festgelegt („verpackt“)[2]. Hier erfolgt auch die Auswahl der zu implementierenden Methoden (z. B. zufällig im sinnvollen Rahmen).
- In einer zweiten Stufe (S2) erfolgt dann die Generierung der passenden Java-Beispiellösung, der JUnit-Tests, des Bewertungsschemas[3] sowie die Formulierung der konkreten Aufgabentexte. Auch die Auswahl der konkreten Formulierung ist zufallsgesteuert auf Basis der Menge der jeweils für die Teilaufgaben zur Verfügung stehenden parametrisierten Satzbausteine (siehe auch Fußnote auf nächster Seite).
- Die generierte Aufgabenstellung kann online wahlweise als HTML-Seite und/oder als PDF gelesen werden. Lösungen sind bei Bedarf anzeigbar. Erläuterungen zum Zusammenhang zwischen bestimmten Teilen der Aufgabentexte und den Beispiellösungen wie in [DF17] sind wieder angestrebt.
- Um den oft geringen Programmierkenntnissen und -fähigkeiten der Übenden Rechnung zu tragen, wird ein Eingabeformular angeboten, welches die einzugebende Lösung auf mehrere Textfelder verteilt, siehe Abb. 2.

[2] Die Festlegungen lassen sich dabei jeweils gezielt händisch oder per Zufall aus vorgegebenen Mengen treffen. Aufgabenstrukturen, die momentan erarbeitet werden, beziehen sich z. B. auf die Modellierung einer Fläche (mit Ausprägungen wie Rasenfläche, Tafelfläche oder Chipfläche) oder eines Volumens (mit Ausprägungen wie Wasserbecken, Container oder Verpackungsbox).

[3] Für die Papierklausur wird bisher eine per Hand erzeugte Liste mit Hinweisen an die Korrektoren zur Punktevergabe bereitgestellt. Diese soll durch eine generierte Version ersetzt werden, welche der Prüfer selbstverständlich anpassen kann. Ziel ist es dann jedoch, dass die der Änderung zugrunde liegende Idee wieder in den Generator einfließt und das entsprechende Bewertungswissen erhalten bleibt.

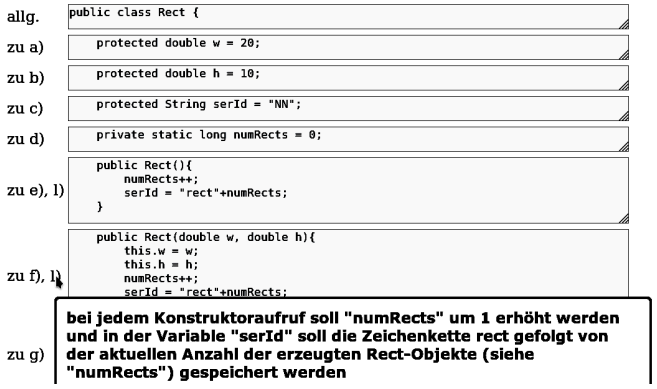

Abb. 2: Ansicht der Eingabemasken für die Lösungen mit Aufgabentext als Tooltip

3 Aktueller Stand der Umsetzung und Umsetzungsdetails

Die Implementierung des Aufgabengenerators erfolgt in Java. Die Java-Klassen lassen sich lokal verwenden, z. B. zur Erzeugung von Aufgaben auf dem Rechner des Lehrenden oder Prüfenden. Auf einem Aufgabenserver wird der Generator via Tomcat (Java Server Pages) in die Übungsoberfläche eingebunden.

Bei der Erzeugung der Aufgabentexte werden Einleitung und einzelne Teilaufgaben auf Basis der abstrakten Aufgabenbeschreibung durch unterschiedliche Satzbausteine wiedergegeben[4]. Dies soll helfen, das Verständnis von „Informatik“-Deutsch zu verbessern. Um verschiedene Ausgabeformate (HTML, Text, LaTeX) angemessen zu unterstützen, werden die Hervorhebungen semantisch wichtiger Texte (z. B. Variablennamen) oder textstrukturierende Elemente (z. B. Aufzählungen) pro Ausgabeformat anders erzeugt. Ein gekürztes Beispiel für einen vollständig generierten Aufgabentext könnte wie folgt aussehen:

Erstellen Sie eine öffentliche Klasse namens `Rect`*, welche Folgendes enthalten soll:*

a) eine geschützte `double`*-Objektvariable mit dem Namen* `w`*, wobei* `w` *am Anfang stets den Wert 20 haben soll und eine Breite speichert,*

c) eine geschützte `String`*-Variable, welche* `serId` *heißen soll und pro Objekt separat zu speichern ist, wobei* `serId` *am Anfang stets den Wert NN enthalten soll,*

d) eine private `long`*-Klassenvariable namens* `numRects`*, wobei* `numRects` *am Anfang stets den Wert 0 haben soll und die aktuelle Anzahl der erzeugten Objekte speichert,*

f) einen öffentlichen Konstruktor mit einer Parameterliste, welche zur Initialisierung von `w` *und* `h` *dienen soll (die Reihenfolge der Parameter genau so wie eben aufgezählt),*

[4] `public static String[] coD_V = { "einen ${deM1}1 Defaultkonstruktor"  // gekürztes Beispiel`
`, "einen Konstruktor, welcher parameterlos und ${deX0}1 ist"`
`, "einen ${deM1}1 Konstruktor, welcher keine Parameter hat" };`
Hier ist andeutungsweise zu sehen, dass auch grammatikalische Aspekte bei der Parametersubstitution Berücksichtigung finden können (z. B. wird aus `public` durch `{deX0}` „öffentlich“ und durch `{deM1}` „öffentlichen“)

l) bei jedem Konstruktoraufruf soll `numRects` *um 1 erhöht werden und in der Variable* `serId` *soll die Zeichenkette rect gefolgt von der aktuellen Anzahl der erzeugten* `Rect`*-Objekte (siehe* `numRects`*) gespeichert werden.*

Die Aufteilung der Teillösungen auf einzelne Felder erhöht die Chance für besseres Feedback erheblich („teile und herrsche“). Bei einer derartigen Eingabeform wirkt sich z. B. eine vergessene schließende geschweifte Klammer am Methodenende nur auf die entsprechende Teilaufgabe aus. Da die Aufgabe (inkl. den einzelnen Teilaufgaben und -lösungen) dem Generator bekannt ist, lassen sich einzelne Tests grundsätzlich auch bei nicht komplett compilierbarem Quellcode generieren und aufrufen. Der bisherige Ansatz einer JUnit-Testklasse, welche nur compiliert, wenn alle zu testenden Methoden und Variablen in der Klasse zugreifbar sind, wird momentan durch eine wesentlich flexiblere reflection-basierte Lösung ersetzt. Pro Teilaufgabe hängt die Art der Bewertung davon ab, ob der jeweilige Code compiliert (ggfs. unter Einbeziehung der Lösungsbsp. anderer Teilaufg.). Falls ja, werden die zugehörigen generierten funktionalen Tests gestartet. Falls nein, sollen Pattern bekannter/typischer Fehler eingesetzt werden, um Lösungshinweise zu geben. Diese Pattern dienen auch als Basis für die Erstellung des zur Aufgabe gehörenden Korrekturschemas.

4 Zusammenfassung

Der Beitrag beschreibt kurz den Stand der Arbeiten am Aufgabengenerator im August 2019. Der Generator sollen eine gerechtere automatische Bewertung von nicht-compilierfähigem Quellcode (typischerweise Anfängerquellcode) unterstützen - aber durchaus auch in Verbindung mit dynamischen Tests der compilierfähigen Lösungsteile.

Literatur

[DA19] Döring, U.; Artelt, B.: Web-Based Tools for Interactive Training in Implementing Java Methods. In: Proceedings of INTED 2019, 13th annual International Technology, Education and Development Conference. Valencia, Spain, S. 3974–3979, März 2019.

[DF17] Döring, U.; Fincke, S.: Interaktive Ansätze zur Vermittlung von Programmierfähigkeiten im Rahmen des Ingenieurstudiums. In: Tagungsband der 12. Ingenieurpädagogischen Regionaltagung 2017. Ilmenau, Germany, S. 171–176, Mai 2017, ISBN: 978-3-9818728-1-1.

[SH15] Schulten, B.; Höppner, F.: Zur Einschätzung von Programmierfähigkeiten - Jedem Programmieranfänger über die Schultern schauen. In (Priss, U.; Striewe, M., Hrsg.): 2nd Workshop on „Automatische Bewertung von Programmieraufgaben“ (ABP 2015), Wolfenbüttel, Germany 6. Nov. 2015. CEUR Workshop Proceedings 2015, Wolfenbüttel, Germany, 2015, URL: http://ceur-ws.org/Vol-1496/#paper7.