Einsatz von Musterlösungen für automatische Regressionstests von Autobewertern für Programmieraufgaben

HOCHSCHULE HANNOVER
UNIVERSITY OF APPLIED SCIENCES AND ARTS
Fakultät IV Wirtschaft und Informatik

Robert Garmann[1]
Marc Herschel[2]

Bericht

15. Oktober 2021

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

[1] E-Mail: robert.garmann@hs-hannover.de
[2] E-Mail: marc@herschel.io

Zusammenfassung

Automatische Bewertungssysteme für Programmieraufgaben (Grader) sind komplexe Softwaresysteme. Automatisch ausführbare Regressionstests können kostengünstig potenzielle Fehler im Grader aufdecken. Im vorliegenden Beitrag soll beschrieben werden, wie Musterlösungen als Eingabedaten für automatische Regressionstests fungieren können. Es geht also um die Vorstellung einer Lösung eines Software Engineering Problems für E-Learning-Systeme. Wir betrachten, welche Eigenschaften des ProFormA-Aufgabenformats für automatische Regressionstests genutzt werden können und schlagen Erweiterungen des Formats vor. Der Beitrag beschreibt die beispielhafte Implementierung eines automatischen Black Box Systemtests für den Autobewerter Graja und geht dabei u. a. auf die Gestaltung eines Record-Playback-Vorgehens, auf einen unscharfen Soll-Ist-Vergleich sowie auf die Frage der Lokalisierbarkeit von entdeckten Regressionen ein.

Schlagwörter

automatisch bewertete Programmieraufgaben, Austauschformat, ProFormA, E-Assessment, Grader, Softwaretest

DDC Klassifikation

004 Datenverarbeitung; Informatik

GND-Schlagwörter

Programmierung, E-Learning, Computerunterstütztes Lernen, Softwaretest, Übung <Hochschule>, Lernaufgabe

ACM CCS (2012)

• Social and professional topics~Professional topics~Computing education~Computing education programs~Computer science education

• Software and its engineering~Software creation and management~Software verification and validation~Software defect analysis~Software testing and debugging

• Social and professional topics~Professional topics~Computing education~Student assessment

• Applied computing~Education~E-learning

Inhalt

1 Einleitung

Automatische Bewertungssysteme für Programmieraufgaben bestehen häufig aus zwei Teilen. Das Framework F (s. Abbildung 1) implementiert aufgabenunabhängige Routinen. Die Aufgabe T implementiert Konfigurationsdateien, Testtreiber, Bewertungsvorgaben, Musterlösungen und den Aufgabentext einer ganz speziellen Programmieraufgabe. Beide Teile sind mehr oder weniger komplexe Softwaresysteme, wobei F komplexer als T ist. Beide Systeme können Fehler aufweisen, welche i. d. R. durch Tests entdeckt werden können. Von besonderem Interesse sind automatisch durchführbare Tests, da sie auch nach geringfügigen Änderungen an F oder T kostengünstig durchgeführt werden können.

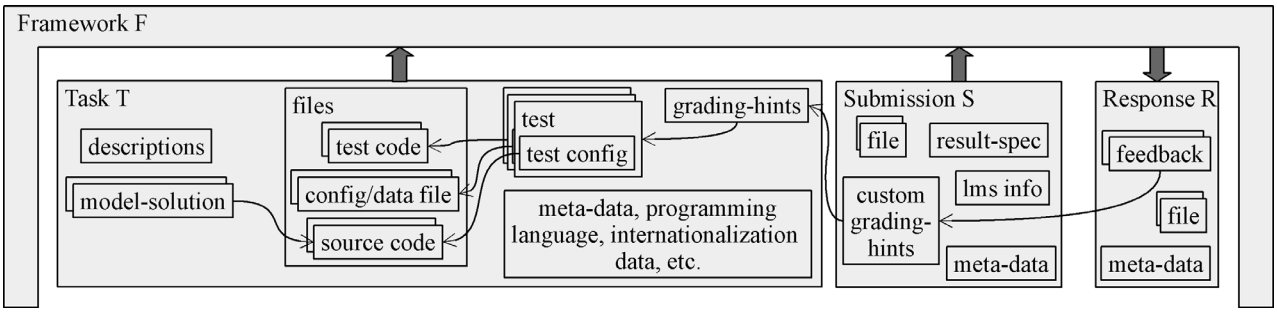

Abbildung 1: Framework, Aufgabe, Einreichung und Ergebnis

Im vorliegenden Beitrag soll beschrieben werden, wie die in T verfügbaren Musterlösungen als Submission S an F gegeben werden können und wie die resultierende Response R genutzt werden kann, Regressionen zu entdecken. Abschnitt 2 führt in das ProFormA-Format ein und skizziert eine einfache Beispielaufgabe mit zwei Musterlösungen. Abschnitt 3 formuliert Anforderungen und Lösungsansätze für einen automatischen Regressionstest. Anhand einer prototypischen Implementierung für den Autobewerter Graja wird in Abschnitt 4 demonstriert, wie Regressionen in F und T entdeckt werden.

2 Das ProFormA-Format

Das ProFormA-Austauschformat[3] (vgl. Abbildung 1) definiert den Aufbau der drei Artefakte Task, Submission und Response. In der Task wird die Aufgabe beschrieben. Für diese werden die Aufgabenstellung sowie weitere Metadaten, die Konfiguration zugehöriger Tests, Files und Musterlösungen sowie ein baumartig aufgebautes, die einzelnen Tests referenzierendes Bewertungsschema bereitgestellt. Die Submission definiert einen Standard für die Beschreibung studentischer Abgaben. Hier können kursspezifische Anpassungen des Bewertungsschemas erfolgen. Durch die Response wird eine standardisierte Aufbereitung des vom Grader automatisiert erzeugten Feedbacks ermöglicht. Die Response bezieht sich in Teilen auf das oben angesprochene Bewertungsschema, so dass ein LMS das Feedback strukturiert darstellen kann.

Wir betrachten eine einfache Beispiel-Task (vgl. Abbildung 2), deren Bewertungsschema aus zwei Tests besteht: die Übersetzbarkeit wird mit 1 Punkt honoriert, die korrekte Funktion mit 4 Punkten. Abbildung 2 zeigt rechts zwei Musterlösungen, die sowohl Teil der Task sind als auch als Submission fungieren sollen. Die vom Grader Graja [1] erstellten Responses sind in Abbildung 3 dargestellt.

Schreiben Sie eine statische Methode `Kreis.flaeche` zur Berechnung der Kreisfläche des als Parameter gegebenen Radius.

Aspect	Score
Overall result	*5,00*
Should successfully compile	1,00
Functional correctness	4,00

```java
public class Kreis {
  public static double flaeche(double r) {
    return Math.PI * r * r;
  }
}
```

```java
public class Kreis {
  public static double flaeche(double r) {
    return 2.0 * Math.PI * r;
  }
}
```

Abbildung 2: Beispielaufgabe mit Bewertungsschema und mit je einer korrekten (oben) und falschen (unten) Musterlösung

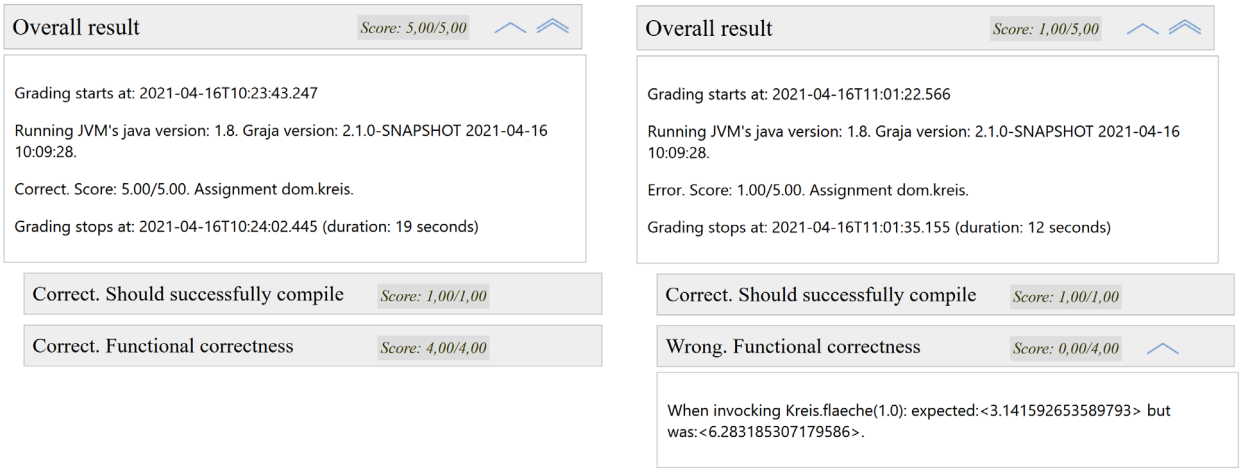

Abbildung 3: Bewertungsergebnisse der korrekten (links) und falschen (rechts) Musterlösung

[3] https://github.com/ProFormA/proformaxml

3 Anforderungen und Lösungen für einen Regressionstest

Es sollen Anforderungen und Lösungsansätze für einen automatischen Regressionstest formuliert werden. Der Grader soll unter Einsatz der in der Task vorhandenen Musterlösungen als Black Box ausgeführt werden. Es handelt sich also um einen Systemtest, bei dem die beobachtete Response mit einer erwarteten Response verglichen wird.

[R1] Der Regressionstest führt den Grader als Black Box aus, indem er Musterlösungen als Submission nutzt und die beobachtete Response prüft.

3.1 Rauschunterdrückung

In [6] wird als Vorbedingung für die Vergleichbarkeit von Testläufen die *controlled regression testing assumption* formuliert. Nur wenn Rahmenbedingungen und Eingabedaten des Tests exakt übereinstimmen, können exakt gleiche Ergebnisse erwartet werden. Solche Bedingungen finden wir in der Realität selten vor. Wir wollen in unserem Ansatz ganz bewusst von dieser Annahme abweichen. Dennoch soll der Test falsch positive und falsch negative Ergebnisse möglichst gut vermeiden. Dazu wollen wir den Vergleich relevanter von dem irrelevanter Bewertungsergebnisse unterscheiden.

Einige Teile der Response wie das quantitative Ergebnis (die erreichte Punktzahl oder Prozentzahl) sind relevant für den Regressionstest. Ein unerwartetes quantitatives Element einer beobachteten Response soll zu einem Testfehlschlag führen. Einige andere Bestandteile sind nicht relevant und würden bei einem exakten Vergleich der Response mit zuvor aufgezeichneten Ergebnissen zu falsch positiven Alarmen führen. Irrelevant ist bspw. die im Textfeedback enthaltene Uhrzeit des Graderlaufs, aber auch andere Teile der textuellen Bewertung wie temporär auf dem Bewertungssystem gültige Dateipfade können irrelevant sein. In Abbildung 3 hat sich bspw. ein Tippfehler im Feedback eingeschlichen (invocking). Wenn dieser nun vom Aufgabenautor korrigiert wird, sollte der Regressionstest so parametriert werden können, dass er nicht fehlschlägt. Wir sprechen von einer Unterdrückung eines gewissen Rauschens in der beobachteten Response.

[R2] Quantitative Elemente der Response (die erreichte Punktzahl oder Prozentzahl) sollen exakt den erwarteten Werten gleichen.

[R3] Qualitative Elemente der Response müssen, da sie gewollten Änderungen unterworfen sind, nicht exakt mit den erwarteten Elementen übereinstimmen. Der Regressionstest soll Rauschen in beobachteten Responses unterdrücken können.

Bei der Rauschunterdrückung denken wir zunächst an eine Normalisierung des textuellen Feedbacks der erwarteten und der beobachteten Responses durch Ersetzungen für gewisse reguläre Ausdrücke. Weiterhin können Ungefährvergleiche mit einem aufgabenspezifischen Schwellwert für die Levenshtein-Distanz hilfreich sein.

3.2 Umgang mit Pseudozufall und Nichtdeterminismus

Es gibt automatisch bewertete Aufgaben, deren textuelles Bewertungsergebnis von Pseudozufällen abhängt, die im Framework F oder in der Aufgabe T eingebaut sind. Das quantitative Ergebnis ist in der Regel nicht vom Pseudozufall abhängig, wohl aber das textuelle Feedback, wenn es bspw. Testdaten auflistet, die der Grader pseudozufällig erzeugt hat. Für solche Fälle können mehrere Maßnahmen sinnvoll sein.

- Zum einen kann der o. g. Ungefährvergleich helfen.
- Zudem sollten Pseudozufallszahlenfolgen während des Regressionstests stets mit demselben seed erzeugt werden.
- Als dritte Möglichkeit soll gefordert werden, die beobachtete Response nicht nur mit einer, sondern mit mehreren potenziell erwartbaren Responses zu vergleichen. Dieser Ansatz ist

insb. geeignet, wenn es nur wenige verschiedene Zufallsergebnisse gibt, die sich bspw. aus einer nichtdeterministischen Abarbeitungsreihenfolge in nebenläufig gestalteten Musterlösungen ergeben.

[R4] Die beobachtete Response muss u. U. mit mehreren potenziell erwartbaren Responses verglichen werden.

3.3 Vorhalten des relevanten Teils der erwarteten Response

Um beobachtete mit erwarteten Responses vergleichen zu können, müssen die erwarteten Responses zur Verfügung stehen. Es reicht, die für den Regressionstest relevanten Teile der erwarteten Response vorzuhalten. Wir nennen diese relevanten Teile *regression test expected behavior (rteb)*.

[R5] Zu jeder Musterlösung soll ein regression test expected behavior (rteb) zur Verfügung stehen. Im rteb sind quantitative Ergebnisse und relevante qualitative Ergebnisse enthalten.

Im Prototyp (vgl. Abschnitt 4) zeichnen wir die rteb automatisch auf. Denkbar ist aber auch eine manuelle Erstellung. Die quantitativen Ergebnisse werden sinnvollerweise in einer die Baumstruktur der *grading-hints* spiegelnden separaten Baumstruktur im rteb abgelegt. Die qualitativen Ergebnisse können aus textuellen Kommentaren bestehen, deren für den Regressionstest irrelevante Bestandteile zuvor ausgefiltert wurden.

Das zu einer Musterlösung gehörende rteb kann ungültig werden, wenn eine Aufgabe T oder das Framework F verändert werden. Bei den Änderungen können wir Korrekturen von Anpassungen unterscheiden [4]. Bei Korrekturen in T und F soll das rteb möglichst weiterhin gültig bleiben dürfen (*corrective regression testing* [5]). Dies wird durch die oben angesprochene Rauschunterdrückung angestrebt. Bei Anpassungen wird das rteb erwartbar ungültig (*progressive regression testing*). Wenn etwa das Bewertungsschema so angepasst wird, dass fortan 2 Punkte für die Übersetzbarkeit und 3 Punkte für die korrekte Funktion gewährt werden, muss das rteb entsprechend angepasst werden. Für diesen Zweck ist es hilfreich, das rteb in einem sowohl für Menschen als auch Maschinen lesbaren Format mit der Aufgabe zu bündeln.

[R6] Zu jeder Musterlösung soll das rteb als Teil der Aufgabe im XML-Format gespeichert werden.

Das bestehende ProFormA-Format erlaubt in der aktuellen Version 2.1 keine zusätzlichen Angaben zu einer Musterlösung. Es ist jedoch möglich, *file*-Elemente hinzuzufügen. Über Namenskonventionen der vergebenen Ids oder Dateipfade lässt sich eine Zuordnung von rteb zu Musterlösungen erreichen.

[R7] Zu jeder Musterlösung soll das rteb als *file*-Element der Task hinzugefügt werden, dessen id oder Dateipfad eine Zuordnung zur Musterlösung erlaubt.

In zukünftigen Versionen des ProFormA-Formats sollte überlegt werden, die Aufnahme von rteb-Objekten explizit vorzusehen.

3.4 Lokalisierung bei Testfehlschlägen

Bei entdeckten Regressionen muss die verursachende Softwarekomponente identifiziert werden können.

[R8] Die beobachtete Response einer Musterlösung soll genauso wie das rteb Informationen enthalten, die die Lokalisierung der einen Testfehlschlag verursachenden Softwarekomponente ermöglichen.

Die Lokalisierung von ausschließlich quantitativ abweichenden Ergebnissen ist häufig leicht auf den betreffenden, in der Aufgabe definierten Test zurückzuführen. Abweichendes textuelles Feedback wird in manchen Gradern an vielen Code-Stellen produziert. Um bspw. Stacktraces als Meta-Information an Fragmente des Textfeedbacks anzuhängen, wäre eine Erweiterung des ProFormA-Response-Formates erforderlich. Im Prototyp (vgl. Abschnitt 4) erzeugt das Frame-

work F neben der üblichen Response R zusätzlich für jedes in R enthaltene Textfragment je einen Stacktrace in einer separaten Ausgabedatei. Diese wird vom Regressionstestwerkzeug gelesen und zur Generierung von Lokalisierungsinformation im Testbericht genutzt.

3.5 Konsequenzen für das ProFormA-Format

Um den automatischen Regressionstest unter Einsatz von Musterlösungen zu erleichtern, machen wir die folgenden Vorschläge zur Weiterentwicklung des ProFormA-Formates.

Zu jeder Musterlösung sollen relevante Teile der erwarteten Response in der Task abgelegt werden können. Wir nennen diese Teile *regression test expected behavior (rteb)*.

Zu jedem Fragment eines in einer Response enthaltenen *feedback*-Elements sollen zusätzlich Lokalisierungsinformationen abgelegt werden können. Denkbar sind hier Stacktraces oder andere Grader-spezifische Daten, die entweder einem kompletten *feedback*-Element oder einem Teil davon unter Angabe von Start- und Endposition zugeordnet werden.

3.6 Weitere Aspekte

Variable Programmieraufgaben [2] sind Aufgabenschablonen, aus denen nach Festlegung von Parameterwerten Aufgabeninstanzen entstehen. Hier ist ein pragmatischer Weg, den Regressionstest für eine aus Standard-Parameterwerten entstandene Aufgabeninstanz durchzuführen. Abhängig von der Art der variablen Aufgabe T und ihrer Instanziierung könnten auf diese Weise aber große Teile der Softwarekomponente T ungetestet bleiben. Ggf. können explizit für den Regressionstest mehrere Parametersätze für zu testende Aufgabeninstanzen konfiguriert werden

Das Regressionstestwerkzeug muss etliche Musterlösungen durch den Grader bewerten lassen. Die Laufzeit dieses Prozesses kann erheblich sein. Unser Prototyp benötigte für 57 Aufgaben mit insg. 437 Musterlösungen fast 35 Minuten für einen vollständigen Regressionstest. Um die Effizienz zu verbessern, ist es sinnvoll, im Regressionstestwerkzeug Parallelverarbeitung vorzusehen. Da die Laufzeit des Regressionstests von Aufgabe zu Aufgabe stark variieren kann, sollte eine Priorisierung der Testfälle nach Effizienz überlegt werden. Andere Priorisierungen – bspw. nach Codeabdeckung oder nach in vorherigen Tests beobachtetem Fehleraufdeckungspotenzial – sind denkbar [8].

4 Ergebnisse einer prototypischen Implementierung

In diesem Abschnitt demonstrieren und erläutern wir einige Eigenschaften der Implementierung eines Regressionstests für Graja. Mehr Details sind in [3] nachlesbar.

Graja besitzt für Aufgabenautoren einen Build-Mechanismus zur Erstellung einer ProFormA-Aufgabe aus mehreren Einzeldateien und Quelltexten. Dieser Mechanismus wurde nun ergänzt, so dass XML-Dateien des rteb zur ProFormA-Aufgabe als *file*s hinzugefügt werden. Die entsprechenden XML-Dateien können zuvor automatisch durch einen Aufzeichnungsmechanismus für alle vorliegenden Musterlösungen erzeugt werden.

Nach Korrektur des Tippfehlers im Feedback der Aufgabe (invocking → invoking) und der Definition eines aufgabenspezifischen Schwellwerts von 1 für die Levenshtein-Distanz werden keine Regressionen entdeckt. Mit dem Schwellwert 0 meldet die Implementierung für die falsche Musterlösung eine in diesem Fall als unerwünscht angenommene Regression wie sie in Abbildung 4 dargestellt ist. Für die richtige Musterlösung wird keine Regression festgestellt.

```
=== Expected ===

 When invocking Kreis.flaeche(1.0):  expected:<3.141592653589793> but was:<6.283185307179586>.

=== Observed ===

 When invoking Kreis.flaeche(1.0):  expected:<3.141592653589793> but was:<6.283185307179586>.
```

Abbildung 4: Unerwünschte Regression bei Levenshtein-Schwellwert 0

Nach Änderung des Bewertungsschemas (2 + 3 Punkte statt 1 + 4 Punkte) wird im Prototyp intentionsgemäß keine Regression für die richtige Musterlösung entdeckt, da für jeden Bewertungsaspekt lediglich die beobachteten mit den erwarteten erreichten prozentualen Erfolgsraten (nicht absoluten Punktzahlen) verglichen werden. Für die falsche Musterlösung wird zutreffenderweise eine Regression festgestellt (vgl. Abbildung 5).

Violation(s) occured within a node of the grading schema.

Path in grading schema:

`root`

	Grades	Success
Expected	`CORRECT, WRONG`	`0,200`
Observed	`CORRECT, WRONG`	`0,400`

Abbildung 5: Erwünschte Regression nach Änderung des Bewertungsschemas

Als letztes Beispiel einer Regression nehmen wir an, dass ein Entwickler des Graja-Frameworks F versehentlich einen temporär intendierten Texteinschub nicht wieder entfernt hat. Hier werden bei beiden Musterlösungen Regressionen entdeckt. Abbildung 6 zeigt das Ergebnis für die richtige Musterlösung. Dort ist auch zwecks besserer Lokalisierbarkeit der Ursache der Regression ein Soll-Ist-Vergleich der internen DOM-Struktur des Bewertungstextes visualisiert. Andere Autobewerter werden vermutlich andere Informationen zur Lokalisierung vorsehen. Im konkreten Fall wurde in Zeile 102 der Quelltextdatei CompileModuleRunner.java eine Ausgabeanweisung irrtümlich nicht entfernt.

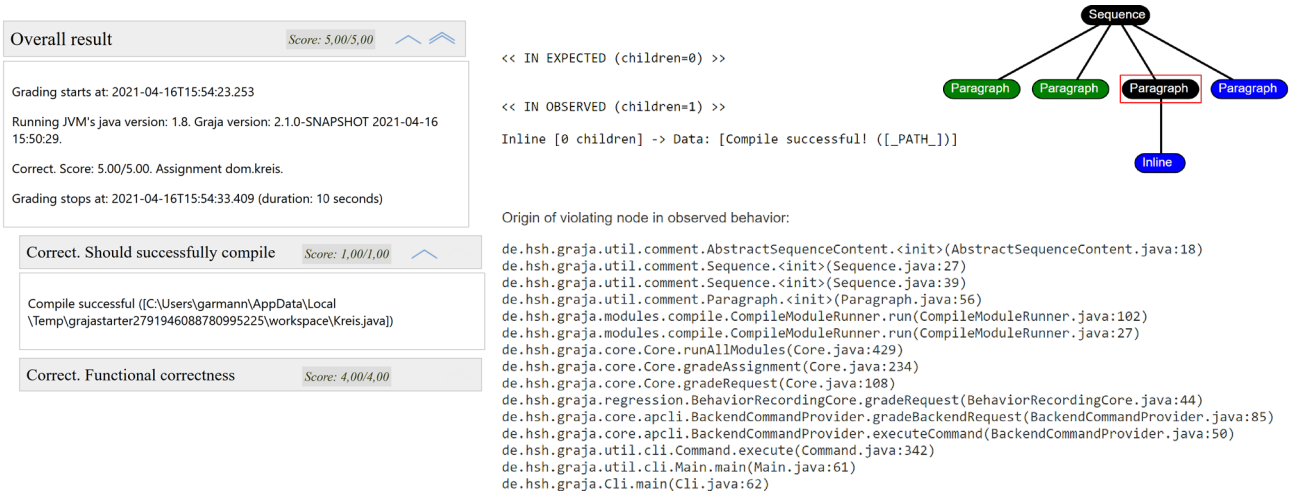

Abbildung 6: Erwünschte Regression nach fehlerhafter Wartung des Frameworks F. Diese führt zu einer unerwünschten Ausgabe des Pfades der übersetzten Datei (links). Auf der rechten Seite ist ein Auszug der internen Repräsentation der Ausgabe als DOM dargestellt. Blaue Knoten sind beobachtete Knoten die im erwarteten DOM nicht existieren. Rechts oben erkennt man die Eliminierung des Dateipfades durch die in Abschnitt 3.1 erwähnte Normalisierung.

5 Ausblick

Zukünftig muss der Vorschlag zur Erweiterung des ProFormA-Formats um Elemente, die den Regressionstest unterstützen, durch weitere prototypische Implementierungen in weiteren Gradern untermauert werden. Insbesondere die Darstellung von Informationen zur Lokalisierung der Ursache entdeckter Regressionen bedarf weiterer Forschung [7]. Die Baumdarstellung in Abbildung 6 ist sehr spezifisch auf Graja zugeschnitten. Hier müssen andere Darstellungen gefunden werden, die von Grader-Interna abstrahieren. Zudem ist es in der derzeitigen Implementierung noch nicht immer möglich, den der Lokalisierung dienenden Stacktrace automatisch im richtigen Moment zu erzeugen, so dass zusätzlicher Implementierungsaufwand pro Aufgabe anfällt.

Der Regressionstest für variable Aufgaben bedarf weiterer Forschung, um die Eignung des in Abschnitt 3.6 genannten pragmatischen Ansatzes zu untersuchen.

6 Quellen

[1] Garmann, R.: Der Grader Graja. In Bott, O.; Fricke, P.; Priss, U.; Striewe, M. (Hrsg.): Automatisierte Bewertung in der Programmierausbildung. Digitale Medien in der Hochschullehre, ELAN e.V., Waxmann, 2017.

[2] Garmann, R.: Ein Datenformat zur Materialisierung variabler Programmieraufgaben. In: 4. Workshop „Automatische Bewertung von Programmieraufgaben“, Essen, 2019.

[3] Herschel, M.: Musterlösungen in Graja als Mechanismus für Regressionstests. In: Wissenschaftliche Schriften der Hochschule Hannover, Hannover, https://doi.org/10.25968/opus-1721, 2020.

[4] ISO/IEC/IEEE International Standard for Software Engineering - Software Life Cycle Processes – Maintenance, 14764-2006, https://doi.org/10.1109/IEEESTD.2006.235774.

[5] Leung, H. K.; White, L.: Insights into regression testing. In Proceedings of the Conference on Software Maintenance, 60-69, IEEE, 1989.

[6] Rothermel, G.; Harrold, M. J.: Analyzing regression test selection techniques. IEEE Transactions on software engineering, 22(8), 529-551, 1996.

[7] Wong, W. E. et.al.: A survey on software fault localization. IEEE Transactions on Software Engineering, 42(8), 707-740, 2016.

[8] Yoo, S.; Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software testing, verification and reliability, 22(2), 67-120, 2012.