Generierung von Zusatzinformationen in automatischen Systemen zur Bewertung von Programmieraufgaben

Michael Striewe
paluno - The Ruhr Institute for Software Technology
Universität Duisburg-Essen
michael.striewe@s3.uni-due.de

Zusammenfassung: Gängige Systeme zur automatischen Bewertung von Programmieraufgaben geben Feedback und Punktzahlen meist basierend auf der Ausführung von Testfällen. Das Feedback könnte jedoch vielfältiger und genauer gestaltet werden, wenn weitere Informationen berücksichtigt werden, die leicht gewonnen werden können. Der vorliegende Beitrag diskutiert einige der Möglichkeiten konzeptionell und durch die prototypische Anwendung auf reale Daten. Das Ziel ist es, auf diese Weise genaueres, situationsbezogenes Feedback erzeugen zu können.

1 Einleitung

Automatische Systeme zur Bewertung von Programmieraufgaben werden üblicherweise eingesetzt, um vordefinierte Testfälle auszuführen und den Studierenden dadurch Rückmeldung zu geben, welche Anforderungen ihre Lösung erfüllt und welche gegebenenfalls noch nicht erfüllt sind [IAKS10]. Aus der Anzahl der Testfälle und gegebenenfalls auch einem ihnen zugewiesenen Schwierigkeitsgrad lassen sich zudem Punktzahlen und damit letztlich auch Noten ableiten, die auf klaren Fakten basieren und damit auch wenig angreifbar sind. Derartige Systeme werden allerdings nicht nur für Klausuren und prüfungsrelevanten Leistungen eingesetzt, sondern dienen häufig auch als Systeme zum Selbstlernen und Training in Blended-Learning-Szenarien oder sogar als alleinige Feedback-Systeme in reinen Online-Kursen. In diesem Rahmen ist eine Richtig/Falsch-Bewertung einer Lösung zwar noch immer relevant, aber ihr zur Seite treten weitergehenden Hinweise, Hilfestellungen und Zusatzbewertungen. Beispielsweise kann eine zusätzliche Stilprüfung vorgenommen werden, die auf die Verletzung von Codekonventionen hinweist.

Die genannten Beispiele aus statischer Überprüfung (Codestil) und dynamischer Überprüfung (Testfälle) schöpfen das Potenzial automatischer Techniken jedoch nicht aus und

lassen vor allem Zusatzinformationen unberücksichtigt, die während der Durchführung der Überprüfungen leicht gewonnen werden können. Dadurch wird auch die Möglichkeit vergeben, zusätzlich zu einer klaren Bewertung weitergehende Hinweise auf Fehlerursachen oder allgemeines Verbesserungspotenzial anzugeben. Dies ist insbesondere auch in den Fällen schade, in denen eine eingereichte Lösung zwar alle funktionalen Eigenschaften erfüllt und damit alle Testfälle besteht, hinsichtlich anderer Kriterien wie zum Beispiel Performanz oder Umfang des Codes noch verbessert werden kann. Umgekehrt kann sich aus einer Zusatzinformation ein konkreter Hinweis zur Behebung von Fehlern ergeben: Besteht eine Lösung nicht alle Testfälle, ist aber in Umfang und Komplexität mit der Musterlösung vergleichbar, dürfte der Fehler eher im Detail der Implementierung zu suchen sein, während ein auffallend kurzes und einfach strukturiertes Programm mit denselben fehlgeschlagenen Testfällen eher darauf hindeutet, dass eine grundsätzliche Herausforderung der Aufgabenstellung noch nicht verstanden wurde.

Der vorliegende Artikel erläutert daher anhand von einfachen Informationen, die während er Durchführung statischer oder dynamischer Überprüfungen leicht gewonnen werden können, beispielhaft die mögliche Generierung und Nutzung von Zusatzinformationen, die eine Richtig/Falsch-Bewertung ergänzen können. Insbesondere wird dabei in Abschnitt 2 auf die Messung der Ausführungsdauer, in Abschnitt 3 auf die Bestimmung der Code-Coverage von Testfällen und in Abschnitt 4 auf die Messung klassischer Softwareproduktmetriken wie zyklomatische Komplexität eingegangen. Es wird aufgezeigt, wie diese Informationen gewonnen werden können und welcher inhaltliche Mehrwert für Studierende und Lehrende daraus entstehen kann. Die vorgestellten Techniken werden jeweils anhand realer Daten aus dem Prüfungssystem JACK [SBG09] im Sinne einer Machbarkeitsstudie evaluiert, die den Einsatz dieser Techniken in kommenden Lehrveranstaltungen vorbereiten soll.

2 Ausführungsdauer

Die Bestimmung der Ausführungsdauer ist einer der einfachsten Möglichkeiten, Zusatzinformationen zu gewinnen, da einreichte Lösungen in den allermeisten Bewertungssystemen ohnehin zur Ausführung gebracht werden. Die Messung kann dabei in Zeiteinheiten oder in Programmschritten erfolgen, was beides individuelle Vorteile und Nachteile hat. Eine Messung in Zeiteinheiten setzt voraus, dass die Durchführung der Tests bei verschiedenen Lösungen immer auf demselben oder zumindest einem identischen System und immer unter denselben Bedingungen erfolgt. Abweichungen in der Systemauslastung oder der Leistungsfähigkeit des Systems führen zwangsläufig zu abweichenden Ergebnissen für identische Programme, was die Aussagekraft mindert. Eine Messung in Programmschritten unterliegt nicht diesem Risiko, ist aber deutlich schwieriger zu interpretieren, da einzelne Programmschritte nicht gleichwertig sein müssen und keine allgemeine Aussage getroffen werden kann, ob eine Lösung mit weniger Schritten besser ist als eine Lösung mit mehr Schritten.

Für die Implementierung in JACK wurde daher die Messung in Zeiteinheiten gewählt, wobei die Ausführung der Testfälle in JACK je nach Auslastung parallel auf identischen

	Aufgabe 1	Aufgabe 2	Aufgabe 3	Aufgabe 4	Aufgabe 5	Aufgabe 6
Median	3 sec.	7 sec.	13 sec.	21 sec.	25 sec.	175 sec.
25%-Quntil	3 sec.	5 sec.	9 sec.	21 sec.	15 sec.	153 sec.
75%-Quntil	3 sec.	7 sec.	21 sec.	23 sec.	33 sec.	181 sec.
Anzahl Testfälle	10	8	25	15	16	18

Tabelle 1: Ausführungszeiten der Tests verschiedener Aufgaben im WS 2012/13. Gemessen wurde jeweils die Summe der Ausführungsdauer aller Testfälle.

virtuellen Servern geschieht und daher für alle Lösungen vergleichbare Zeiten zu erwarten sind. Da in JACK bei der Ausführung von Testfällen eine Komponente zum Einsatz kommt, die Traces der Ausführung aufzeichnet (vgl. [SG11]), ist die Ausführungszeit generell deutlich höher als bei einer Ausführung ohne Aufzeichnung, so dass die Ausführungszeit lediglich in einer Granularität von Sekunden gemessen wird. Tabelle 1 zeigt Ausführungszeiten, die für sechs Aufgaben im Wintersemester 2012/13 gemessen wurden. Der zusätzliche Aufwand für die Gewinnung dieser Informationen war minimal, da nur wenige Zeilen Code in der Ausführungsroutine für die Testfälle ergänzt werden mussten.

Die so gewonnen Informationen können auf verschiedene Arten genutzt werden. Zum einen kann aus ihnen zusätzliches Feedback für Studierende generiert werden, beispielsweise wenn sie eine Lösung einreichen, deren Ausführungszeit außerhalb der angegebenen Quantile liegt. Liegt sie oberhalb des 75%-Qunatils, kann ein entsprechender Hinweis gegeben werden, dass die Performanz der Lösung noch verbessert werden kann. Liegt sie dagegen unterhalb des 25%-Quantils, kann lobendes Feedback und gegebenenfalls sogar Zusatzpunkte für eine besonders effiziente Lösung gegeben werden. Zu berücksichtigen ist, dass Lösungen mit Endlosschleifen, die vom Testtreiber nach einem Timeout abgebrochen werden, stark nach oben abweichende Zeiten aufweisen können und in diesen Fällen ein Hinweis auf eine mögliche Performanzverbesserung inhaltlich nicht zutreffend ist.

Neben dem Feedback an die Studierenden können die Daten aber auch genutzt werden, um die Qualität der Testfälle zu bestimmen. Im oben gezeigten Beispiel fällt auf, dass Aufgabe 6 erheblich höhere Laufzeiten aufweist als die anderen Aufgaben. Zwar ist eine tendenzielle Steigerung bei allen Aufgaben zu erkennen (und bei steigendem Schwierigkeitsgrad im Laufe des Semesters auch erwartbar), aber trotzdem weicht der Wert für Aufgabe 6 sehr deutlich nach oben ab. Tatsächlich konnte diese Abweichung auf ungünstig formulierte Testfälle zurückgeführt werden, die zu hohen Laufzeiten führten, ohne die eigentliche Testqualität zu verbessern.

3 Code-Coverage

Die Berechnung der Code-Coverage ist ein allgemeines Mittel zur Qualitätssicherung bei Softwaretests [Mye04]. Die Code-Coverage gibt dabei an, welcher Anteil des Programmcodes von mindestens einem Testfall tatsächlich aufgerufen wurde. Im Idealfall versucht man, eine vollständige Abdeckung zu erreichen. Im Fall von Programmieraufgaben muss

jedoch die Einschränkung gemacht werden, dass nur die Abdeckung für variable Teile der Lösung, die von den Studierenden erstellt werden, relevant ist. Als Vorgabe zur Verfügung gestellte Methoden, die die Studierenden benutzen dürfen, aber nicht müssen, sollten dagegen ausgenommen werden, weil sie die Werte sonst verfälschen. Gleiches gilt für Programmteile, die den Studierenden für lokale Tests zur Verfügung gestellt wurden, vom Testtreiber jedoch gar nicht aufgerufen werden.

Die Berechnung der Code-Coverage erfolgt in JACK als Nebenprodukt der bereits oben erwähnten Aufzeichnung von Traces. Da in diesen ohnehin festgehalten wird, welche Programmzeilen durchlaufen wurden, muss zur Berechnung der Abdeckung lediglich die Menge der abzudeckenden Zeilen als zusätzliche Information zur Verfügung gestellt werden. Im prototypischen Einsatz wurde dazu eine Blacklist verwendet, auf der pro Aufgabe alle Methoden verzeichnet waren, die nicht berücksichtigt werden sollten. Die Bestimmung der relevanten, ausführbaren Codezeilen erfolgte dann automatisch über Java Reflection. Der Vorteil dieses Ansatzes liegt darin, dass so insbesondere alle Hilfsmethoden, die von den Studierenden möglicherweise selbst erstellt wurden, in die Berechnung einbezogen werden. Für eine exemplarische Aufgabe aus dem Sommersemester 2012 wurde damit nachträglich die Code-Coverage bestimmt. Im Schnitt lag die Abdeckung bei 95,66% (Median 98,96%; 25%-Quantil 92,42%; 75%-Quantil 100%).

Grundsätzlich sollte davon ausgegangen werden, dass die zur automatischen Prüfung verwendeten Testfälle eine vollständige Abdeckung des Codes der Musterlösung(en) erreichen. Ist dies nicht der Fall, sinkt auch die Güte der Aussagen, die über studentische Lösungen getroffen werden können. Unter der Annahme, dass die Testfälle die Aufgabenspezifikation vollständig abdecken, deutet eine Code-Coverage von weniger als 100% darauf hin, dass die studentische Lösung Code enthält, der nicht zur Lösung der Aufgabe beiträgt. Im Feedback an die Studierenden können die betroffenen Zeilen farblich hervorgehoben werden und somit Hinweise auf mögliche Schwachstellen der Lösung geben. Die weitergehende Interpretation, ob der nicht abgedeckte Code tatsächlich überflüssig ist oder nur fälschlicherweise nicht aufgerufen wurde, kann allerdings nur in Verbindung mit der Kenntnis über eventuell fehlgeschlagenen Testfälle oder die Intention der Codeabschnitte erfolgen. Bisher wurde noch keine Untersuchung durchgeführt, ob die Studierenden die Hervorhebungen als hilfreich empfinden.

4 Softwareproduktmetriken

Der mögliche Einsatz von Softwareproduktmetriken bei der automatischen Bewertung von Programmieraufgaben ist keine neue Idee, sondern wurde bereits vor geraumer Zeit für die Qualitätssicherung und Unterstützung der Benotung [MY99] oder die Plagiatserkennung [Lea95] untersucht. Dabei gibt es eine Vielzahl von Metriken, die grundsätzlich eingesetzt werden können, wobei im Kontext von Programmieraufgaben Einschränkungen existieren: Programmieraufgaben arbeiten oft mit Codevorlagen, deren Inhalt Auswirkungen auf die Ergebnisse der Metriken hat. Aus diesem Grund sind manche objektorientierten Metriken [AC94] weniger geeignet, wenn beispielsweise die Zahl der Methoden und die Kopplung von Klassen ohnehin vorgegeben sind und somit keine zu-

sätzlichen Aussagen über die Qualität einer Lösung gewonnen werden können. Ähnliches gilt beispielsweise für die Halstead-Metriken [Hal77], die die Größe des Vokabulars messen (Anzahl verschiedener Operanden und Operatoren). Metriken, die diese Nachteile nicht haben, sind einfache Umfangsmetriken, bei denen der Umfang der Codevorlage abgezogen werden kann, oder auch die zyklomatische Komplexität [McC76].

Zum prototypischen Einsatz von Metriken wurde JACK um ein Modul erweitert, das durch Analyse des Syntaxbaumes einer Lösung die folgenden einfachen Metriken erhebt: zyklomatische Komplexität, Zahl der Statements, Zahl der Expressions, sowie Anzahl der Referenzen und Anzahl der Schleifenausdrücke, die beide eine Teilmenge der Statements beziehungsweise Expressions ausdrücken. Das Modul wurde bisher nur analytisch eingesetzt, ohne Feedback für die Studierenden zu generieren. Wie erwartbar ist, besteht zwischen einiger diesen Metriken eine hohe Korrelation. Für eine exemplarische Aufgabe aus dem Sommersemester 2012 korreliert die zyklomatische Komplexität sehr stark ($>$ 0,97) mit der Zahl der Expressions und der Zahl der Referenzen. Die Zahl der Statements korreliert ebenfalls deutlich ($>$ 0,9) mit der zyklomatischen Komplexität und der Zahl der Expressions. Die Zahl der Schleifenausdrücke weist zu den anderen Metriken Korrelationskoeffizienten zwischen 0,77 und 0,82 auf.

Trotz dieser Korrelationen können die Metriken genutzt werden, um Aufgaben auf Vergleichbarkeit zu prüfen, indem vergleichbare Aufgaben ähnliche Kennzahlen liefern sollten [SG13]. Ferner können auf der Basis einzelner Metriken Lösungen identifiziert werden, die sich deutlich von den üblichen Lösungen unterscheiden und zu denen ein spezielles Feedback generiert werden soll. Dabei kann es sich entweder um Lösungen handeln, die stark vom Durchschnitt abweichen oder die eine unübliche Kombination von Werten aufweisen. Außerdem können die Metriken eingesetzt werden, um den Fortschritt bei der Entwicklung einer vollständigen Lösung zu verfolgen, sofern ein Bewertungssystem mehrere Einreichungen erlaubt. In diesem Fall können Studierende beispielsweise einen zusätzlichen Hinweis erhalten, wenn ihre Lösung erheblich angewachsen ist, ohne zusätzliche Testfälle zu erfüllen.

5 Fazit und Ausblick

In diesem Artikel wurden verschiedene Möglichkeiten diskutiert, wie neben der Ausführung von Testfällen zusätzliche Informationen über eine Lösung in einem automatischen Bewertungssystem gewonnen werden können. Die Anwendbarkeit der Konzepte konnte prototypisch auf realen Daten gezeigt werden.

Daraus ergibt sich als Ziel für die weitere Entwicklung, diese Techniken regelmäßig einzusetzen und Konzepte zu entwickeln, wie aus der Kombination verschiedener Informationen exakteres, situationsbezogenes Feedback gewonnen werden kann. Dabei ist zu berücksichtigen, dass die Interpretation der verschiedenen gewonnen Daten oft nur in Kombination möglich ist, dann aber stärkere Aussagen zulässt als dies für die einzelne Information jeweils möglich wäre. Ferner ermöglicht die Kenntnis möglichst vieler Merkmale der Lösung, die Studierenden ebenfalls zu einer umfassenden Beschäftigung mit der Lösung zu motivieren.

Literatur

[AC94] Abreu, Fernando B. ; Carapuça, Rogério: *Object-Oriented Software Engineering: Measuring and Controlling the Development Process*. 1994

[Hal77] Halstead, Maurice H.: *Elements of software science*. Elsevier, 1977

[IAKS10] Ihantola, Petri ; Ahoniemi, Tuuka ; Karavirta, Ville ; Seppälä, Otto: Review of recent systems for automatic assessment of programming assignments. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. New York, NY, USA : ACM, 2010 (Koli Calling '10). – ISBN 978–1–4503–0520–4, S. 86–93

[Lea95] Leach, Ronald J.: Using metrics to evaluate student programs. In: *SIGCSE Bull.* 27 (1995), S. 41–43. http://dx.doi.org/10.1145/201998.202010. – DOI 10.1145/201998.202010. – ISSN 0097–8418

[McC76] McCabe, Thomas J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering,* 2 (1976), Nr. 4, S. 308–320. http://dx.doi.org/10.1109/TSE.1976.233837. – DOI 10.1109/TSE.1976.233837. – ISSN 0098–5589

[MY99] Mengel, Susan A. ; Yerramilli, Vinay: A case study of the static analysis of the quality of novice student programs. In: *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*. New York, NY, USA : ACM, 1999 (SIGCSE '99). – ISBN 1–58113–085–6, S. 78–82

[Mye04] Myers, Glenford J.: *The Art of Software Testing*. 2. Wiley, 2004

[SBG09] Striewe, Michael ; Balz, Moritz ; Goedicke, Michael: A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In: *Proceedings of the First International Conference on Computer Supported Education (CSEDU), 23 - 26 March 2009, Lisboa, Portugal* Bd. 2 INSTICC, 2009, S. 54–61

[SG11] Striewe, Michael ; Goedicke, Michael: Using Run Time Traces in Automated Programming Tutoring. In: Rössling, Guido (Hrsg.) ; Naps, Thomas L. (Hrsg.) ; Spannagel, Christian (Hrsg.): *Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29*, ACM, 2011. – ISBN 978–1–4503–0697–3, S. 303–307

[SG13] Striewe, Michael ; Goedicke, Michael: Analyse von Programmieraufgaben durch Softwareproduktmetriken. In: Spillner, Andreas (Hrsg.) ; Lichter, Horst (Hrsg.): *SEUH* Bd. 956, CEUR-WS.org, 2013 (CEUR Workshop Proceedings), S. 59–68