Herausgeber et al. (Hrsg.): DeLFI 2018 - Die 16. e-Learning Fachtagung Informatik,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 165

Zur automatischen Erkennung von Fehlkonzepten bei Java-Einsteigern durch Analyse von Speicher-Protokollen

Frank Höppner[1], Jan-Hendrik Hemmje

Abstract: Java-Anfänger entwickeln ihre eigene Vorstellung, was beim Ablauf eines Java-Programms passiert. Um diese Vorstellung in belastbare Bahnen zu lenken, kann eine schrittweise Protokollierung des Speicherinhalts genutzt werden. Durch Analyse eingegebener Speicher-Protokolle und der Syntaxbäume der Aufgaben konnte in einem Experiment gezeigt werden, dass bekannte Fehlkonzepte in studentischen Einreichungen anhand von Regeln erkannt werden können. Das nährt die Hoffnung, diese Fehlkonzepte künftig automatisch zu detektieren, geeignete Erklärungen bereitzustellen und passende Fragen zur Vertiefung auszuwählen, ohne dass ein manueller Eingriff erforderlich ist.

1 Einführung

Der Fokus dieser Arbeit liegt auf Grundlagen-Vorlesungen zur Programmierung mit Java, deren Teilnehmeranzahl zu groß ist, als dass eine individuelle Fehlerabstellung im Dialog mit dem Dozenten immer gewährleistet werden könnte. Im Laufe der Jahre hat sich in den betreuten Übungen der Eindruck festgesetzt, dass für einige Teilnehmer ohne vorherige Programmiererfahrung die Ebene der JUnit-Tests eine hohe Hürde darstellt. JUnit-Tests bieten den Vorteil, dass Studierende direkt Feedback zu ihrem Lösungsstand erhalten, auch wenn kein Laborbetreuer verfügbar oder Rückfragen möglich sind. Wenn jedoch die Programmierkonstrukte noch nicht ganz verstanden wurden, kann ein Studierender das Ziel "grüner Balken" nicht systematisch sondern nur zufällig erreichen. In Vorlesung und begleitenden Übungen hält man sich i.a. nicht zu lange mit Erläuterungen der einzelnen Sprachkonstrukte auf. Bei einem typisch breiten Spektrum von Anfängern und Erfahrenen, sind Übungsaufgaben schnell auf einem Niveau, das Einsteiger überfordert, die die Semantik der Sprachkonstrukte nicht sicher kennen.

Für Anfänger im Programmieren, aber auch um im Kurs insgesamt die Vorstellung von Programmabläufen abzugleichen, wurde ein Werkzeug *Java Memory Tracer* geschaffen [GH17]. Es soll sicherstellen helfen, die Semantik der einzelnen Sprachkonstrukte (an einer Modellvorstellung) eindeutig zu klären – noch *bevor* erste Programme geschrieben werden. Einigen bereitet das Verständnis kaum Probleme, andere benötigen mehrere Wiederholungen, um sich falsche Annahmen abzugewöhnen. Daraus ergibt sich die Anforderung des Dozenten, dass es möglichst viele Aufgaben zu Übungszwecken gibt, aber dadurch andererseits kein

[1] Ostfalia Hochschule, Fakultät Informatik, Am Exer 2, Wolfenbüttel, f.hoeppner@ostfalia.de

großer Aufwand auf Dozentenseite entsteht. Der *Java Memory Tracer* erlaubt bereits das Einstellen kleiner Java-Programme, ohne dass eine *richtige Lösung* von Hand erarbeitet werden müsste. Aber eine große Aufgabenmenge alleine führt noch nicht zu einer sinnvollen Nutzung, wenn Studierende mit der Menge an Aufgaben alleine gelassen werden. Neulinge benötigen Hinweise darauf, was noch mal nachgelesen werden sollte, Fortgeschrittene wollen nicht mit langweiligen Wiederholungen belästigt werden. In dieser Arbeit soll ein Beitrag zur Frage geleistet werden, ob/wie es möglich ist, automatisch eine Diagnose von bekannten Fehlkonzepten vorzunehmen und passende Aufgaben vorzuschlagen.

2 Unterstützungssysteme für die Programmierausbildung

Der hier verfolgte Ansatz zielt auf einen Einsatz *vor* den breit eingesetzten Kontrollen studentischer Abgaben mit bspw. JUnit-Tests und Checkstyle ab. Solche Vorstufen können Aufgaben zur Prüfung von Sprachkenntnissen sein [SKG17], indem Lückentexte ausgefüllt werden. Auch die Visualisierung eines Programmablaufs [Mo04, LJC10] kann helfen, ungeeignete Gedankenmodelle zu brauchbaren Modellvorstellungen abzuändern. Allerdings bleibt der Student bei reinen Visualisierungen passiv und es fehlt die Möglichkeit zur Selbstkontrolle. Das ist auch der Fall, wenn ein Programmablauf als *Trockenübung* ohne Rechner protokolliert wird; solche Werkzeuge verbessern das Verständnis signifikant [HJ13], erfordern aber eine manuelle Kontrolle durch einen Prüfer. Mit dem *Java Memory Tracer* [GH17] (vgl. Abb. 1) wurde eine Web-Anwendung geschaffen, an der Studierende solche Ablauf-Protokolle (in der Vorlesung oder im Selbststudium) eingeben und sofort Feedback bekommen. Für jede ausgeführte Zeile wird der Inhalt von Stack und Heap notiert.

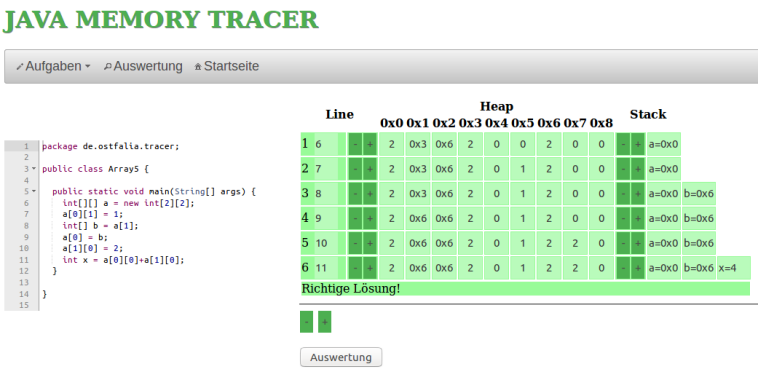

Abb. 1: Eingabe eines Ausführungs-Protokolls mit der Web-Applikation “Java Memory Tracer”

Der Detailierungsgrad des Protokolls mag auf den ersten Blick hoch erscheinen, aber das ist der Sprache geschuldet. Weil Unwissen über die interne Handhabung von bspw. Arrays zu Programmierfehlern führen kann, muss im Rahmen der Programmierausbildung dieses Verständnis vermittelt werden, insbesondere für Studierende der Informatik – Anforderungen für Schnupperkurse mögen andere sein.

3 Problemstellung

Bisher wurde der *Java Memory Tracer* genutzt, um Studierende interaktiv Speicherprotokolle ausfüllen zu lassen und individuelles, unmittelbares Feedback geben zu können. Auch eine Dozentensicht mit einer Übersicht von Fehlerhäufungen (für die Live-Bearbeitung einer Aufgabe innerhalb der Vorlesung) ist verfügbar.

Ein Werkzeug ist erfolgreich, wenn es die gewünschten Effekte bei den Studierenden erzielen kann *und* von den Studierenden auch genutzt wird. Dabei besteht die Gefahr, gute Studierende zu langweilen und Anfänger mit Problemen zu überfordern. Eine individuelle Empfehlung oder Zuordnung von Aufgaben könnte helfen, die Aufgaben am Lernstand des Einzelnen auszurichten und die Motivation zu erhalten. Da eine individuelle Aufgaben-Kontrolle und -Empfehlung durch einen Dozenten nicht skaliert, stellt sich die Frage, ob eine Automatisierung in diesem Kontext möglich ist. Es ergeben sich daraus zwei Teilprobleme:

Q1 Kann allein auf Basis der Eingaben, die Studierende beim Ausfüllen verschiedener Speicherprotokolle vornehmen, automatisch auf typische, bekannte Fehlkonzepte geschlossen werden?

Q2 Kann allein auf Basis des Java-Codes entschieden werden, welche Aufgabe bei Vorliegen eines Fehlkonzeptes geeignet ist, diesen Aspekt noch mal zu üben (um die Notwendigkeit einer händischen Annotation der Programme zu erübrigen)?

Kann die Frage Q1 positiv beantwortet werden, können also bekannte Fehlkonzepte in studentischen Antworten wiedererkannt werden, sind auch ergänzende Erklärungen leicht bereit zu stellen, bevor dann neue Aufgaben zur Selbst-Kontrolle auf Basis von Q2 empfohlen werden können.

4 Lösungsansatz

Eine Aufgabe besteht im *Java Memory Tracer* nur aus einem (kurzen) Java-Programm (kein weiterer Input). Daraus wird mit Hilfe des Java Debugging Interfaces eine Referenzlösung automatisch erstellt (analog zur studentischen Eingabe in Abb. 1). Zusätzlich kann mit Hilfe der Compiler API ein abstrakter Syntax-Baum (AST, abstract syntax tree) generiert werden. Beides zusammen stellt die verfügbaren Informationen pro Aufgabe dar.

Wird ein Speicherprotokoll eingegeben, erfolgt der Abgleich mit der automatisch erzeugten Referenzlösung. Da Fehler oftmals zu Folgefehlern im Protokoll führen, konzentrieren wir uns jeweils auf die Quelltextzeile mit dem ersten aufgetretenen Fehler. Die AST verschiedener Aufgaben sind natürlich verschieden, aber die Hypothese ist, dass die charakteristische Fehlersituation durch Teile des AST beschrieben werden kann, die sich dann in verschiedenen Aufgaben wiederholen. Ziel ist die Extraktion von Merkmalen des AST, deren Existenz mit bestimmten Fehlersituationen korreliert. Dazu wird mit den betroffenen Zeilen wie

folgt verfahren: Jedes Element im AST der betreffenden Quelltextzeile wird zu einem n-Tupel erweitert, indem die $n-1$ Vaterknoten im Baum aufgenommen werden. Wir benutzen hier nur $n \in \{2,3\}$. Wird bspw. im Protokoll ein falscher Wert für a nach Zeile 10 angegeben (vgl. Abb. 2), erzeugen wir wegen der Zuweisung (Zeile 10, Position 149 im AST) das Kontext-Tripel (Block, Expression-Statement, Plus-Assignment)$_{EL}$ und aufgrund der Multiplikation (auch Zeile 10, Position 156 im AST) (Plus-Assignment, Plus, Multiply)$_{EL}$. Der verwendete Index EL (Error Line) zeigt an, dass sich die Tripel direkt auf die Fehlerzeile beziehen. Wird ein falscher a-Wert im Protokoll hingegen in Zeile 13 erstmals protokolliert, wird der Fehler aber in der falschen Auswertung der **if**-Bedingung liegen. Darum werden zusätzlich Kontext-Tripel (If, Parenthesized, Less-Than)$_{PL}$ der zuvor ausgeführten Zeile aufgenommen und mit PL (previous line) markiert. Zusätzlich werden auch 2-Tupel aus der Inorder-Traversierung des AST-Teilbaums (pro Befehl) gebildet (notiert mit hochgestelltem Index SEQ). Ähnliche AST-Elemente werden außerdem generalisiert, bspw. Less-Than und Greater-Than zu X-Than, weil sie alle auf demselben Aspekt (Vergleich) basieren. Pro AST-Element einer Zeile werden alle besprochenen Tupel erzeugt; die Gesamtheit dieser Tupel beschreibt den lokalen Kontext, in dem der Fehler passiert ist.

```
1 package de.ostfalia.tracer;
2
3 public class Demo {
4
5     public static void main(String[] args) {
6         int a = 2;
7         int b = (a>2)?3:1;
8
9         if (b < a) {
10            a += b+a * a;
11        }
12
13        System.out.println(a);
14    }
15 }
```

```
| 37|    CLASS Demo
| 71|     METHOD main
| 66|      PRIMITIVE-TYPE VOID
| 85|      VARIABLE args
| 82|       ARRAY-TYPE
| 76|        IDENTIFIER String
| 91|      BLOCK
| 99|       VARIABLE a
| 95|        PRIMITIVE-TYPE INT
|103|        INT-LITERAL 2
|112|       VARIABLE b
|108|        PRIMITIVE-TYPE INT
|121|        CONDITIONAL-EXPRESSION
|116|         PARENTHESIZED
|118|          GREATER-THAN
|117|           IDENTIFIER a
|119|           INT-LITERAL 2
|122|         INT-LITERAL 3
|124|         INT-LITERAL 1
```

```
|131|       IF
|134|        PARENTHESIZED
|137|         LESS-THAN
|135|          IDENTIFIER b
|139|          IDENTIFIER a
|142|       BLOCK
|147|        EXPRESSION-STATEMENT
|149|         PLUS-ASSIGNMENT
|147|          IDENTIFIER a
|153|          PLUS
|152|           IDENTIFIER b
|156|           MULTIPLY
|154|            IDENTIFIER a
|158|            IDENTIFIER a
|169|      EXPRESSION-STATEMENT
|187|       METHOD-INVOCATION
|179|        MEMBER-SELECT println
|175|         MEMBER-SELECT out
|169|          IDENTIFIER System
|188|        IDENTIFIER a
```

Abb. 2: Beispiel-Programm (links) und abstrakter Syntaxbaum der Compiler-API (zweispaltig).

Neben der Analyse des AST kann aus dem Protokoll auch noch Information generiert werden, welche Änderungen beim Übergang zur aktuellen Zeile notwendig waren, bspw. die Änderung von Werten auf dem Heap oder dem Stack, die Vergößerung oder Verkleinerung des Stacks oder die Art der Daten (Referenz oder Wert).

Ein Datensatz mit studentischen Abgaben wird dann in eine Tabelle konvertiert. Die Attribute (Spalten) der Tabelle entsprechen allen Kontext-Tupeln, die jemals in irgendeiner Aufgabe aufgetreten sind. Eine Zeile entspricht einer Abgabe und eine Spalte wird auf `true` gesetzt, wenn der aufgetretene Fehler-Kontext das Kontext-Tupel enthalten hat. Für bekannte Fehlkonzepte werden die Zeilen um das vorliegende Fehlkonzept ergänzt. Dieser Datensatz wird anschließend analysiert, um Abhängigkeiten des Fehlkonzepts von den Kontext-Tripeln zu identifizieren. Dafür können Standard-Verfahren zur Datenanalyse benutzt werden [Be10, Ha09], vorzugsweise solche, die gut interpretierbare Ergebnisse liefern, wie etwa Regel-Lerner.

5 Evaluation

Für eine Auswertung wurden 6 Fehlkonzepte exemplarisch untersucht. Es wurden je 5 Protokolle ausgefüllt, wie sie Studierende mit dem jeweiligen Fehlkonzept ausfüllen würden. Der zuvor beschriebene Datensatz wurde mit einem WEKA-Regel-Lerner [Ha09] analysiert, um möglichst einfache Regeln für jedes einzelne Fehlkonzept abzuleiten. Tabelle 1 zeigt die Fehlkonzepte und die identifizierten Bedingungen, die mit den Fehlkonzepte korrelieren. Die letzte Spalte zeigt, auf wieviele Protokolle die Bedingung zutrifft, aufgeteilt nach r richtigen und f falschen Zuordnungen ($r : f$).

ID	falsche Vorstellung / Annahme und korrelierender Ausdruck	r:f
FK1	Die innerhalb eines Blocks angelegten Variablen sind auch nach Beendigung des Blocks noch verfügbar. (Block, Block, Expression_Statement)$_{PL}$	4:0
FK2	Kein Unterschied zwischen Post- und Pre-In/Decrement (PrePost_IncDec, Identifier)$^{SEQ}_{EL}$ $\vee$ (Parenthesized, X_Than, PrePost_IncDec)$_{PL}$	5:1
FK3	Die Startzeile einer for-Schleife (mit Update- und Prüf-Teil) wird nur einmal initial ausgeführt. (For_Loop, Block, Expression_Statement)$_{PL}$ $\wedge$ (Block, Block_End)$^{SEQ}_{EL}$	5:0
FK4	Auswertungsfehler, hier Missachtung von Punkt- vor Strichrechnung (Operator-Priorität) (X_Literal, Identifier)$^{SEQ}_{PL}$ $\vee$ (Variable, X_Arithm_Op, X_Arithm_Op)$_{EL}$	5:0
FK5	Bei Funktionsaufrufen werden auf dem Stack nur die Variablen der aktuell ausgeführten Funktion protokolliert (Stack_Grew) $\wedge$ ((Identifier, Identifier)$^{SEQ}_{EL}$ $\vee$ $\neg$ (Block, Variable, Primitive_Type)$_{EL}$)	5:0
FK6	Bei Array-Zuweisungen wird automatisch eine tiefe Zuweisung ausgeführt (alle Elemente des Arrays werden kopiert). (Stack_Ref_Changed)	3:0

Tab. 1: Fehler und korrelierende Kontext-Tripel.

Das Kontext-Tripel zu FK1 bezieht sich auf die Zeile *vor dem Fehler* (Index PL) und fordert, dass wir uns in einem 2-fach verschachtelten Block befinden. Ein Block wird schon durch die `main`-Funktion eingeführt, einen weiteren Block brauchen wir zwingend, um die Sichtbarkeit von Variablen (vor Funktionsende) enden zu lassen. Dass das Kontext-Tupel als letzte Anweisung ein Expression Statement fordert, liegt nur daran, dass die Beispielaufgaben (bei denen Sichtbarkeit auf die Probe gestellt wird), kaum andere Sprachkonstrukte enthalten. Damit werden 4 von 5 Situationen richtig erkannt. Zu FK2 gibt es zwei Kontext-Tripel, wobei das erste erwartbar ein Pre- oder Post-Increment in der Fehlerzeile (EL) einfordert. Alternativ muss in der Zeile *vor dem Fehler* (PL) ein Klammerausdruck mit einem Vergleich vorkommen, der ein Pre- oder Post-Increment benutzt. Das macht ebenfalls Sinn, weil das Ergebnis eines Vergleichs einer **if**-Anweisung im Protokoll nicht festgehalten wird, aber den nächsten Schritt beeinflusst – eine falsche Auswertung zeigt sich also erst in der Folgezeile.

Für FK3 bedeuten die Bedingungen sinngemäß, dass wir in der letzten Ausführungszeile eine Anweisung innerhalb einer Schleife ausgeführt haben und nun am Ende des Schleifenblocks angelangt sind. Wenn hier schon die Schleifenvariable erhöht wird (und nicht zur for-Zeile zurückgekehrt wird), deutet das auf eine Fehlinterpretation der **for**-Semantik (FK3). Für

FK4 wird eine Variablendeklaration gefordert (X_Literal generalisiert int-, double-, float-Typen) oder eine "komplexe Rechenoperation", die aus zwei verschachtelten arithmetischen Operationen besteht. Letzteres ist erforderlich, weil für einen Fehler bzgl. "Punkt- vor Strichrechnung" beide Operationen in einem Ausdruck vorkommen müssen. Auch sehr zutreffend ist die Bedingung für FK5 (alte Variablen auf Stack nicht protokolliert): Es wird ein Wachstum auf dem Stack gefordert und eine Identifier-Liste (Parameterliste eines Funktionsaufrufs) ohne eine Variablendeklaration (die auch für den Stackaufbau verantwortlich sein könnte). Insgesamt sind die gefundenen Bedingungen weitestgehend sinnvoll und erlauben auf der betrachteten Fallmenge eine recht sichere Klassifikation.

6 Fazit

Für bekannte Fehlkonzepte ist es gelungen, aus einigen Beispielen für Fehlkonzepte Regeln abzuleiten, die es ermöglichen, fehlerhafte Abgaben recht sicher den Fehlkonzepten zuzuordnen (Frage **Q1**). Einige Regeln sind inhaltlich sehr treffend, andere nutzen noch Eigenschaften der verwendeten Beispielprogramme. Hier kann eine größere Menge von Beispielen mglw. Abhilfe schaffen. Weil die Regeln nur Eigenschaften des Programms abfragen, können Programmen mit eben diesen Elementen identifiziert und als Aufgaben zur Selbstkontrolle herangezogen werden (womit Frage **Q2** positiv beantwortet ist).

Diese Arbeit wird vom Zentrum für erfolgreiches Lehren und Lernen unterstützt, welches im Rahmen des Qualitätspakt Lehre (Förderkennzeichen 01PL 16059) gefördert wird.

Literaturverzeichnis

[Be10] Berthold, Michael R; Borgelt, Christian; Höppner, Frank; Klawonn, Frank: Guide to Intelligent Data Analysis. Springer, 2010.

[GH17] Goltermann, Robert; Höppner, Frank: Internalizing a Viable Mental Model of Program Execution in First Year Programming Courses. CEUR Workshop Proceedings, (ABP):1–9, 2017.

[Ha09] Hall, Mark; Frank, Eibe; Holmes, G.; Pfahringer, B.; Reutemann, P.; Witten, I.: The WEKA Data Mining Software: An Update. SIGKDD Explorations, 11(1), 2009.

[HJ13] Hertz, Matthew; Jump, Maria: Trace-Based Teaching in Early Programming Courses. Proceedings of the 44th ACM Technical Symposium on Computer Science Education, S. 561–566, 2013.

[LJC10] Lessa, Demian; Jayaraman, Bharat; Czyz, Jeffrey K.: JIVE: A pedagogic tool for visualizing the execution of Java programs. Bericht, University of New York at Buffalo, 2010.

[Mo04] Moreno, Andrés; Myller, Niko; Sutinen, Erkki; Ben-Ari, Mordechai: Visualizing programs with Jeliot 3. Proceedings of the working conference on Advanced visual interfaces - AVI '04, S. 373, 2004.

[SKG17] Striewe, Michael; Kramer, Matthias; Goedicke, Michael: Ein Lückentext-Test zur Beherrschung einer Programmiersprache. In: DeLFI. S. 261–266, 2017.