Ein Lückentext-Test zur Beherrschung einer Programmiersprache

Michael Striewe[1], Matthias Kramer[2] und Michael Goedicke[3]

Abstract: Dieser Beitrag präsentiert und diskutiert erste Erfahrungen mit einem experimentellen Ansatz, der das sprachwissenschaftliche Konzept der C-Tests auf das Verständnis von Programmiersprachen überträgt. Es werden sowohl die Konstruktion der Aufgaben als auch erste vorläufige Ergebnisse erläutert, die eine weitere Untersuchung dieses Aufgabentyps motivieren.

Keywords: Programmierausbildung, Codeverständnis, C-Test, Lückentext, Kompetenzmessung

1 Einleitung

Programmieranfänger sehen sich beim Erlernen ihrer ersten Programmiersprache verschiedenen Herausforderungen ausgesetzt: Das allgemeine Paradigma der gewählten Sprache muss verstanden werden, Programmverhalten muss in Datenstrukturen, Datenflüsse und Kontrollflüsse innerhalb dieses Paradigmas umgesetzt werden und nicht zuletzt muss die Syntax der Sprache beherrscht werden. Letztgenannter Punkt gilt genauso für erfahrene Programmierer, die eine neue Programmiersprache lernen, selbst wenn sie dabei an einem vertrauten Programmierparadigma festhalten.

Dass das Erlernen der Syntax eine Barriere für Anfänger sein kann, ist gut bekannt [Mc01, SS13]. Daher ist es teilweise möglich, aus der Art und Häufigkeit von Syntaxfehlern bei der Programmkonstruktion auf das Niveau der Sprachbeherrschung zu schließen [SH15]. Für einen diagnostischen Test erscheint dieses Vorgehen aber zu ungenau, da die Programmkonstruktion im Rahmen einer Programmieraufgabe neben der reinen Beherrschung der Programmiersprache auch andere Anforderungen stellt. Es erscheint daher wünschenswert, ein Testinstrument zu entwickeln, mit dem Schwächen in der Syntaxbeherrschung ermittelt werden können, ohne dass die Lernenden dazu eigenständig vollständigen Programmcode erzeugen müssen. Entsprechende Aufgaben könnten dann als Ergänzung zu klassischen Programmieraufgaben eingesetzt werden.

Aus den Sprachwissenschaften sind für diesen Einsatzzweck sogenannte C-Tests bekannt, um das Niveau der Beherrschung einer (Fremd-)Sprache zu bestimmen [KR82]. Sie werden z.B. eingesetzt, um Anfängern einen Sprachkurs auf dem richtigen Level

[1] Paluno - The Ruhr Institute for Software Technology, Universität Duisburg-Essen, Campus Essen, michael.striewe@paluno.uni-due.de
[2] Didaktik der Informatik, Universität Duisburg-Essen, Campus Essen, matthias.kramer@uni-due.de
[3] Paluno - The Ruhr Institute for Software Technology, Universität Duisburg-Essen, Campus Essen, michael.goedicke@paluno.uni-due.de

zuzuweisen [Kl85]. Der Einsatz ist nicht auf eine bestimmte Sprache oder Sprachfamilie beschränkt, sondern wurde in zahlreichen Sprachen mit verschiedenen Notationen verwendet [Ba14]. Auch wenn Programmiersprachen formale und keine natürlichen Sprachen sind, erscheint es aufgrund der positiven Erfahrungen mit C-Tests bzgl. Reliabilität, Validität und Objektivität und trotz der existierenden Kritik an diesen Tests [BM15] vielversprechend, das Konzept auch hier zu erproben. Der vorliegende Beitrag demonstriert daher eine Adaption und experimentelle Erprobung des Konzepts von C-Tests für das Erlernen einer Programmiersprache. Die vorläufigen Ergebnisse zeigen, dass dieser Aufgabentyp tatsächlich diagnostisches Potenzial hat und motivieren so eine weitere Erforschung.

Der Beitrag ist wie folgt gegliedert: Abschnitt 2 diskutiert den Forschungskontext und verwandte Arbeiten. Abschnitt 3 erläutert die Aufgabenkonstruktion für die adaptierten C-Tests für Programmiersprachen. Abschnitt 4 stellt eine Beispielaufgabe und die darauf basierenden bisherigen Evaluationsergebnisse vor. Abschnitt 5 schließt den Beitrag mit einem Fazit und einem Ausblick auf nachfolgende Forschungsaktivitäten ab.

2 Forschungskontext und Related Work

Bereits 1986 beschrieb du Boulay [Du86] die Aspekte, die beim Erlernen von Programmierung parallel eingeübt werden müssen und daher nicht selten in Überforderung und Frustration münden. Neben dem Verstehen des Programmablaufs sowie dem Entwickeln allgemeiner Problemlösungsstrategien spielt das Erlernen einer *Notation*, also der Syntax und Semantik einer Programmiersprache, eine wichtige Rolle. Zum einen stellt der Umgang mit der Syntax nachweislich ein wesentliches Hindernis beim Erlernen einer Programmiersprache dar (siehe z.B. [Mc01] und [SS13]). Zum anderen scheint das Wissen um syntaktische Bausteine und deren Semantik allein nicht auszureichen, um diese zu funktionierenden Programmen zusammenzusetzen [Wi96]. Der korrekte Umgang mit einer Programmiersprache, d.h. Programmierbausteine syntaktisch und semantisch sinnvoll miteinander kombinieren zu können, scheint daher mutmaßlich eine notwendige Teilfertigkeit zu sein, die für das erfolgreiche Erlernen von Programmierung von wesentlicher Bedeutung ist. Hier wird auch die inhaltliche Nähe zum linguistischen C-Test offensichtlich.

Betrachtet man nun Programmierkompetenz als multidimensionales latentes Konstrukt [KHB16], so erscheint es sinnvoll, durch vielfältige Testitems im Rahmen eines formativen Assessments festzustellen, welche Ausprägungen in den verschiedenen Teildimensionen zu (Miss-)Erfolgen führen und welche daraus resultierenden individuellen Lernbedürfnisse die Studierenden haben. Insbesondere können so bei einem späteren summativen Assessment (bspw. der Abschlussklausur eines Semesters) bessere und gezieltere Rückschlüsse auf generelle Fehlvorstellungen und Probleme gezogen werden, was wiederum zu einer möglichen Verbesserung der Lehre beiträgt.

Es ist klar, dass für eine umfassende Testung von Programmierkompetenz eine bloße

Adaption des C-Tests für die Programmierung allein nicht ausreicht [Kl04]. Dennoch kann ein solcher Test als Baustein in einem größeren Assessment wertvolle Hinweise auf die Leistung in Kompetenzfacetten wie *Verstehen gegebener Quelltexte* bzw. *syntaktisches Verständnis einer Programmiersprache* liefern. Zudem stellt er durch das Format eine ideale Schnittstelle zwischen den Fertigkeiten *Quelltexte interpretieren* und *Quelltexte produzieren* dar, was ebenfalls vermuten lässt, dass durch einen solchen Test verschiedene Fertigkeiten abgeprüft werden. Hartig & Harsch konnten für C-Tests in Fremdsprachen bereits empirisch nachweisen, dass multidimensionale Modelle die Varianzen besser erklären können als unidimensionale [HH09, S. 61].

3 Aufgabenkonstruktion

Das Konzept der C-Tests in ihrer üblichen Form sieht vor, in einem Text beginnend mit dem zweiten Satz in jedem zweiten Wort die hintere Hälfte des Wortes durch eine Lücke zu ersetzen. Das Füllen einer solchen Lücke mit einer korrekten Lösung erfordert dabei sowohl Kenntnisse im Vokabular der Sprache, um das fragmentierte Wort zu erkennen oder aus dem Kontext zu erschließen, als auch in vielen Fällen der Grammatik, um die passende Endung zu bestimmen. Eine direkte Übernahme dieses Konzeptes erscheint für Programmiersprachen nicht sinnvoll, da das Vokabular eines Programms in der Regel so aufgebaut ist, dass sich aus dem ersten Teil eines Wortes sofort ohne Kenntnis von Grammatik und Kontext der fehlende Teil ergibt (z.B. bei Schlüsselworten), oder aber der zweite Teil nahezu willkürlich gewählt werden kann (z.B. bei Bezeichnern). Ferner treten im Programmcode nicht nur Worte im klassischen Sinne auf, sondern auch Operatoren für Zuweisungen, Vergleiche oder logische Verknüpfungen, die eine wichtigere Rolle haben als Satzzeichen in einem natürlichsprachlichen Text.

Daher wurde eine Adaption des Verfahrens gewählt, bei der eine gewisse Menge von Wörtern komplett in eine Lücke umgewandelt wird. Als Wörter werden dabei Schlüssel-Worte sowie weitere beliebige Zeichenketten behandelt, die spezifisch pro Programmier-sprache oder sogar pro Quelltext zu definieren sind. Im Kontext von Java erscheint es beispielsweise sinnvoll, Zeichenketten wie `String`, `System` oder `println` als er-setzbare Worte zu behandeln, auch wenn sie keine Schlüsselworte der Programmier-sprache, sondern vorgegebene Typen und Methoden der Java API sind. Ferner kann es sinnvoll sein, Operatoren wie == oder `&&` sowie beliebige Klassennamen in Lücken umzuwandeln, sofern deren Lösung aus dem Programmcode geschlossen werden kann.

4 Beispielaufgabe und Evaluation

Eine auf diesem Ansatz basierende Beispielaufgabe zur Programmiersprache Java mit insgesamt 13 Lücken unterschiedlichen Charakters zeigt Abbildung 1. Sie zeigt einen Screenshot aus dem zur Evaluation verwendeten E-Assessment-System in der Ansicht für Lernende.

Betrachten Sie den folgenden Programmcode, der an einigen Stellen Lücken enthält. Ergänzen Sie diese Lücken sinnvoll.

```
public class Bank {
     public static final int MAX_KONTEN = 10;
     int anzahl = 0;

     Konto[] konten =              Konto[MAX_KONTEN];

     private Konto kontoSuchen(int kontoNr) {
          for (int i = 0; i < konten.length; i++) {
                         (konten[i]!=null && konten[i].getKontoNr() == kontoNr) {
                                konten[i];
               }
          }
          return null;
     }

     public boolean kontoAnlegen(Konto neuesKonto) {
          if (kontoSuchen(neuesKonto.getKontoNr()) == null && anzahl < MAX_KONTEN) {
               for (int i = 0; i < konten.length; i++) {
                    if (konten[i]              null) {
                         konten[i] = neuesKonto;
                         anzahl++;
                         return true;
                    }
               }
          }
                         false;
     }

     public boolean kontoLoeschen(              kontoNr) {
          for (int i = 0; i              konten.length; i++) {
                         (konten[i]!=null && konten[i].getKontoNr() == kontoNr) {
                    konten[i] = null;
                    anzahl--;
                    return true;
               }
          }
                         false;
     }

     public double berechneVermoegen(Konto[] konten) {
          double vermoegen = 0.0;
                         (int i = 0; i < konten.length; i++) {
               Konto k = konten[i];
               vermoegen += k.kontoStand;
          }
                         vermoegen;
     }

     private              ausgabeAllerKonten() {
          for (Konto i : konten)
                         (i != null)
                    System.out.println("Konto " + i.getKontoNr()
                              + " hat einen Kontostand von: " + i.getKontoStand());
     }
}
```

Abb. 1: Screenshot der zur Evaluation verwendeten Beispielaufgabe. Die erwarteten korrekten Antworten sind in der Kopfzeile von Tabelle 1 ersichtlich.

In 11 der Lücken fehlen Schlüsselworte der Sprache Java, in den verbleibenden zwei fehlen Vergleichsoperatoren. Die fehlenden Schlüsselworte sind viermal das `return`-Statement, dreimal `if` und je einmal `void`, `new`, `int` und `for`. Die Lücken erfordern eine unterschiedlich umfangreiche Betrachtung des Programmcodes: Der Einsatz von `return`, `if`, `for` und `new` lässt sich aus der jeweiligen Zeile direkt ableiten, da andere denkbare Einträge in der jeweiligen Lücke keine syntaktisch korrekte Lösung bilden

würden. Für die Lösung der Lücke für `void` ist es dagegen notwendig, die betroffene Methode zumindest soweit zu verstehen, dass erkannt wird, dass in dieser Methode keine Rückgabe erfolgt. Die Lücke für `int` kann schließlich nur nach genauerer Betrachtung des Datenflusses geschlossen werden. Dies erfordert auch Sorgfalt beim Erstellen der Aufgabe, da Lücken mit Datentypen ggf. nicht eindeutig lösbar sind, wenn in einer Methode eine lokale Variable deklariert und anschließend zurückgegeben wird. Werden nun sowohl bei der Deklaration als auch in der Signatur die Typnamen zu einer Lücke geändert, könnte jede Lösung korrekt sein, die beide Lücken identisch füllt. Ebenfalls zur Sicherstellung der Eindeutigkeit wurde beim Erstellen der Beispielaufgabe auf die Tilgung von Sichtbarkeits-Modifiern verzichtet, da die korrekte Lösung dazu ohne weitere Kenntnis des Programmkontextes nicht zu bestimmen ist.

Zur experimentellen Evaluation haben 11 Personen die Beispielaufgabe bearbeitet. Bei diesen handelt es sich um Studienanfänger, Auszubildende und Schüler, die alle geringe bis mäßige Erfahrung (max. ein Schuljahr/eine Vorlesung) in Java hatten. In den Ergebnissen in Tabelle 1 ist ersichtlich, dass sowohl in der Summe als auch in der Verteilung der Fehler unterschiedliche Leistungen erzielt wurden. Beispielsweise wurde Lücke 6 dreimal falsch geschlossen, während bei den vier `return`-Lücken insgesamt nur zwei Fehler auftraten. Ferner gibt es Hinweise auf unterschiedliche Personenparameter, da Testperson 11 alle `if`-Lücken falsch geschlossen hat und alle anderen korrekt, während alle anderen Personen zusammen nur einen Fehler bei einer `if`-Lücke machten.

Person	L1 `new`	L2 `if`	L3 `return`	L4 `==`	L5 `return`	L6 `int`	L7 `<`	L8 `if`	L9 `return`	L10 `for`	L11 `return`	L12 `void`	L13 `if`	Σ
#1	1	1	1	1	1	1	1	1	1	1	1	1	1	13
#2	1	1	1	1	1	1	1	1	1	1	1	1	1	13
#3	1	1	1	1	1	0	1	1	1	1	1	1	1	12
#4	0	1	1	1	0	1	0	1	1	1	1	0	0	8
#5	1	1	1	1	1	0	1	1	1	1	1	1	1	12
#6	1	1	1	0	1	0	1	1	1	1	1	1	1	11
#7	1	1	1	1	1	1	1	1	1	1	1	0	1	12
#8	1	1	1	1	1	1	1	1	1	1	1	1	1	13
#9	1	1	0	0	1	1	1	1	1	1	1	1	1	11
#10	1	1	1	1	1	1	1	1	1	0	1	1	1	12
#11	1	0	1	1	1	1	1	0	1	1	1	1	0	10
Σ	10	10	10	9	10	8	10	10	11	10	11	9	9	

Tab. 2: Ergebnisse des Experiments. Eine 1 steht für eine korrekt gefüllte Lücke, eine 0 für eine falsch oder gar nicht ausgefüllte Lücke.

Die geringe Anzahl an Daten verbietet eine weitergehende statistische Analyse. Die Ergebnisse lassen jedoch erwarten, dass weitere Experimente relevante Beziehungen zwischen Personenparametern und Charakteristika der Lücken aufdecken könnten.

5 Fazit und Ausblick

Es konnte gezeigt werden, dass die Konstruktion von C-Tests mit einigen Anpassungen

auch für Programmiersprachen sinnvoll möglich ist. Die vorläufigen Ergebnisse legen nahe, dass das Potenzial der Aufgaben sowie die Einflussfaktoren für die Schwierigkeit der Lösung solcher Aufgaben weiter untersucht werden sollte. Insbesondere ist zu prüfen, ob die Zahl der Fehler ohne weitere Betrachtung der falsch gefüllten Lücken ein gutes Maß für die Beherrschung einer Programmiersprache ist und welcher Quelltext-Umfang mit welcher Frequenz von Lücken als Testwerkzeug geeignet ist. Es ist zu erwarten, dass sich dabei auch Unterschiede je nach Programmiersprache ergeben.

Literaturverzeichnis

[Ba14] Baghaei, Purya: Construction and validation of a C-Test in Persian. In (Grotjahn, Rüdiger, Hrsg.): Der C-Test: Aktuelle Tendenzen, S. 299–312. 2014.

[BM15] Baur, Rupprecht; Mashkovskaja, Anna: C-Test-Kritik reviewed. In (Böcker, Jessica; Stauch Anette, Hrsg.): Konzepte aus der Sprachlehrforschung – Impulse für die Praxis, S. 435-449, 2015.

[Du86] Du Boulay, Benedict: Some difficulties of learning to program. Journal of Educational Computing Research, 2(1), S. 57–73, 1986.

[HH09] Hartig, Johannes; Höhler, Jana: Multidimensional IRT models for the assessment of competencies. Studies in Educational Evaluation, 35(2), S. 57–63, 2009.

[Kl85] Klein-Braley, Christine: C-Tests as placement tests for German university students of English. AKS-Rundbrief, 13/14, S. 96–100, 1985.

[KR82] Klein-Braley, Christine; Raatz, Ulrich: Der C-Test: ein neuer Ansatz zur Messung allgemeiner Sprachbeherrschung. AKS-Rundbrief 4, S. 23–37, 1982.

[KHB16] Kramer, Matthias; Hubwieser, Peter; Brinda, Torsten: A Competency Structure Model of Object-Oriented Programming. In: Learning and Teaching in Computing and Engineering (LaTICE), 2016 International Conference on IEEE, S. 1–8, 2016.

[Mc01] McCracken, Michael; Almstrum, Vicki; Diaz, Danny; Guzdial, Mark; Hagan, Dianne; Kolikant, Yifat Ben-David; Laxer, Cary; Thomas, Lynda; Utting, Ian; Wilusz, Tadeusz: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. ACM SIGCSE Bulletin, 33(4), S. 125–180, 2001.

[SH15] Schulten, Bastian; Höppner, Frank: Zur Einschätzung von Programmierfähigkeiten – Jedem Programmieranfänger über die Schultern schauen. In: Proceedings des Zweiten Workshops „Automatische Bewertung von Programmieraufgaben“, Wolfenbüttel. 2015.

[SS13] Stefik, Andreas; Siebert, Susanna: An empirical investigation into programming language syntax. ACM Transactions on Computing Education (TOCE), 13(4), S. 19, 2013.

[Wi96] Winslow, Leon E.: Programming pedagogy - a psychological overview. ACM Sigcse Bulletin, 28(3), S. 17–22, 1996.