Debugging-Aufgaben – Programmfehler als Lernchance

Hanno Schauer

Mons-Tabor-Gymnasium
Von-Bodelschwingh-Straße 35
56410 Montabaur
hanno.schauer@uni-duisburg-essen.de

Abstract: Der Beitrag präsentiert einen (neuen) Aufgabentyp für den Unterricht in Algorithmik/Programmierung, die sogenannten Debugging-Aufgaben. In Debugging-Aufgaben wird mithilfe des Debuggers einer Programmierumgebung der Quelltext eines mit Fehlern versehenen Programms analysiert und von Fehlern bereinigt. Hierbei sind die die Schülerinnen und Schüler gefordert, sich intensiv und entdeckend mit den Ursachen der Fehler bzw. den zugehörigen (Programmier-) Konzepten auseinanderzusetzen. Der Beitrag integriert Unterrichtserfahrungen an einem rheinland-pfälzischen Gymnasium.

1 Realität und Anspruch in Algorithmik/Programmierung

Die Algorithmik/Programmierung ist ein basaler Baustein jedes Informatikkurses – dies gilt an Schulen ebenso wie an Universitäten. Die Algorithmik/Programmierung ist hierbei konzeptuelles und methodisches Fundament ebenso wie auch Stolperstein: Denn einerseits bauen nachfolgende Reihen auf den Inhalten der Algorithmik/Programmierung auf; dies gilt gleichermaßen für allgemeine Daten- und Kontrollstrukturen wie auch die jeweilige Programmiersprache. Andererseits weist gerade die grundlegende Reihe der Algorithmik/Programmierung eine tendenziell hohe Misserfolgsquote auf [Mo11]: Vornehmliche Ursache des Scheiterns der Schülerinnen oder Schüler ist hierbei die (unzureichende) Beherrschung der konkreten Programmiersprache (oder Notationsform), nicht das Verständnis der algorithmischen Grundstrukturen. Lehrende in der Algorithmik/Programmierung stehen mithin stets vor der Herausforderung zu verhindern, dass die Eigenheiten der jeweiligen Programmiersprache den Unterricht unangemessen dominieren.

Die geschilderte Problematik ist nicht neu und es existierten erprobte didaktische Mittel hierfür. Generell ist der einschlägige Unterricht so zu strukturieren, dass allgemeine Konzepte der Algorithmik einerseits und deren Umsetzung in einer Programmiersprache anderseits für Schülerinnen und Schüler gut unterscheidbar bleiben. Zur Unterstützung speziell der Programmierung existiert darüber hinaus eine ansehnliche Zahl an Softwarewerkzeugen, die darauf abzielen, die Hürden der Programmierung für Anfänger zu reduzieren. Hierbei ist zu denken an (1) Mini-Languages mit eingeschränktem Befehlssatz (z. B. *Robot Karol*[1], *Kara*[2]), an grafische Programmiersprachen, welche syntaktisch

[1] Siehe: http://www.schule.bayern.de/karol/
[2] Siehe: http://www.swisseduc.ch/informatik/karatojava/index.html

fehlerhaften Code von Haus aus nicht ermöglichen (z. B. *BYOB*[3]), oder an Entwicklungsumgebungen, die eine integrierte Pflege des Quellcodes eines Programms und eines zugehörigen Datenmodells (i. d. R. eines Klassendiagramms) erzwingen (z. B. *BlueJ*[4]) bzw. ermöglichen (z. B. *Java Editor*[5]). Auch Verbindungen der genannten Ansätze sind existieren – bspw. kombiniert *Greenfoot*[6] Ansätze einer Mini-Language mit der modellbasierten Entwicklung, *Scratch*[7] wiederum offeriert eine grafisch notierte Mini-Language.

Nachfolgend soll mit den sogenannten Debugging-Aufgaben ein (neuer) Aufgabentyp der Algorithmik/Programmierung vorgestellt werden. Während der Bearbeitung einer Debugging-Aufgabe untersucht der Lerner den Quelltext eines lauffähigen, aber fehlerhaften Programms mithilfe des Debuggers (s)einer Entwicklungsumgebung. Hierbei gilt es, die Fehlerquellen innerhalb des Quelltextes des Programms zu lokalisieren, die Fehler zu analysieren und schließlich zu korrigieren. Im Unterschied zu den eingangs genannten Ansätzen verringern Debugging-Aufgaben nicht die Hürden der Programmierung, sondern leiten Schülerinnen und Schüler dabei an, ausgesuchte Eigenschaften einer Programmiersprache, einer Kontrollstruktur oder einer Datenstruktur entdeckend und so intensiv zu untersuchen, wie es die Behebung des jeweiligen Fehlers erfordert. Debugging-Aufgaben sind kongruent zu den vorgenannten Ansätzen und lassen sich bei Bedarf gut mit diesen kombinieren.

Die Vorstellung der Debugging-Aufgaben selbst erfolgt in Kapitel 3. Dieses vorbereitend gibt Kapitel 2 einen Überblick über Debugging-Tätigkeiten und -werkzeuge. Kapitel 4 evaluiert auf Basis einer Schülerbefragung und Unterrichts-erfahrungen des Autors das Konzept der Debugging-Aufgaben. Der Beitrag schließt mit einem Ausblick (Kapitel 5).

2 Debugging und Debugger

Unter Debugging (wörtlich: „entwanzen“) subsumiert die Softwaretechnik Tätigkeiten einer gezielten Fehlersuche und -eliminierung in compilierten, also grundsätzlich lauffähigen Programmen. Einschlägige Vorgehensweisen sind diesbezüglich (händische) ‚Schreibtischtests‘, das Programmieren von Fehlerabfragen, die Auswertung von Log-Files und der Einsatz dedizierter Debugging-Werkzeuge (z. B. [GB+09], [Ba05]).

Debugging-Tätigkeiten richten sich auf die Eliminierung von Laufzeit- und Semantikfehlern[8], nicht aber auf Syntaxfehler, da letztere bereits beim Compilieren erkannt werden: Laufzeitfehler verhindern die reguläre Ausführung eines Programms, da sie dieses entweder zum Absturz – bspw. nach Division durch Null – oder zum Halten bringen – z. B. in einer Endlosschleife. Mit Semantikfehlern behaftete Programme hingegen wer-

[3] Siehe: http://snap.berkeley.edu/
[4] Siehe: http://www.bluej.org/
[5] Siehe: http://www.javaeditor.org
[6] Siehe: http://www.greenfoot.org
[7] Siehe: http://scratch.mit.edu/
[8] Neben *Semantikfehler* ist auch der Begriff *Logischer Programmfehler* gebräuchlich (z. B. [Ba05, S. 167]).

den performant ausgeführt, das Programmverhalten entspricht aber nicht dem vom Programmierer Gewollten – z. B. indem falsche Werte berechnet werden.

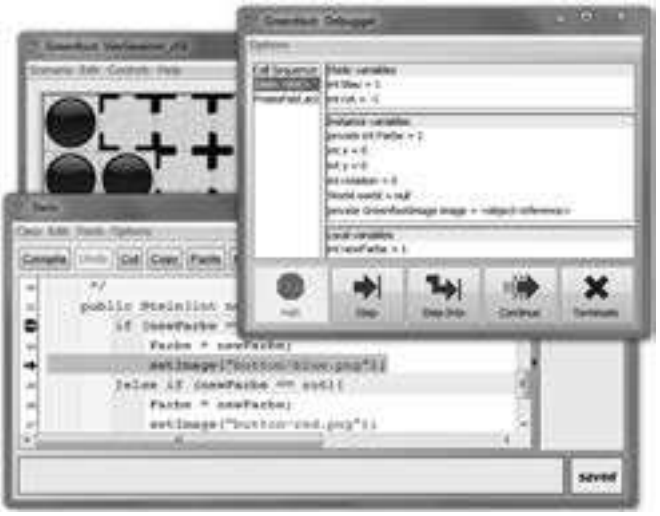

Abbildung 1: Debugging in Greenfoot: Applikation (hinten), Quelltext mit Haltepunkt (Mitte), Debugger mit aktuellen Variablenzuständen und Steuertasten des Tracemodus (vorne)

Werkzeugunterstützung erfährt das Debugging durch dedizierte Debugging-Werkzeuge – häufig in einer Progammoberfläche integriert, dem *Debugger*. Entsprechende Funktionalitäten sind ein üblicher Bestandteil (fast) jeder Programmierumgebung. Moderne Debugger beinhalten auch sehr elaborierte Funktionalitäten – bspw. zur Analyse von Stapelverarbeitung und Funktionsaufrufen, von Speicherabbildern oder Prozessorzuständen. Im und für den Schulunterricht genügt es, sich auf die folgenden drei Kernfunktionalitäten eines Debuggers zu fokussieren: (1) *Haltepunkte* zu setzen, bei deren Erreichung ein laufendes Programm angehalten wird, (2) gestoppte Programme im *Tracemodus* Schritt für Schritt zu steuern, sowie (3) den *internen Zustand* von Variablen und Ausdrücken bzw. dessen Veränderung während der Programmausführung im Tracemodus (quasi live) zu beobachten (vgl. auch Abbildung 1). Die nachfolgenden Ausführungen rekurrieren (nur) auf die vorgenannten Kernfunktionalitäten.

In der Literatur der Didaktik der Schulinformatik ebenso wie in einschlägigen Schulbüchern werden der Debugger bzw. das Debugging eher am Rande behandelt. Einerseits wird die (in der Pädagogik selbstverständliche) Bedeutung einer unter-richtlichen Fehlerkultur und des damit ermöglichten Lernens aus Fehlern auch für die Informatik betont (z. B. [Hu06], [Mo04]). Eine Verbindung der Fehlerkultur mit der Nutzung von Debuggern wird in den diesbezüglichen Quellen nicht hergestellt. Andererseits werden Debugger und Debugging in verschiedenen Quellen als Werkzeug bzw. Technik der Programmierung erwähnt und einführend erläutert (z. B. [En06], [Ha10]). Eine vertiefende Nutzung im Unterricht, so wie sie nachstehend vorgeschlagen wird, findet indes nicht statt.

3 Debugging-Aufgaben

Im Folgenden sollen Debugging-Aufgaben vorgestellt und aufgezeigt werden, wie diese gewinnbringend im Unterricht eingesetzt werden können. Kapitel 3.1 beschreibt diesbezüglich das didaktische Konzept Debugging-Aufgabe. Kapitel 3.2 nennt einschlägige, erfolgreich getestete Anwendungsfelder. Kapitel 3.3 umreißt schließlich auf Unterrichtserfahrungen basierende Regeln guter Praxis für Debugging-Aufgaben.

3.1 Gegenstand

In einer Debugging-Aufgabe gilt es, den Quelltext eines fehlerbehafteten Programmes mithilfe des Debuggers einer (der gewohnten) Programmierumgebung zu entdecken und zu korrigieren. Die Lerner müssen den oder die Fehler im Quelltext lokalisieren, analysieren und bereinigen. Das Vorgehensmodell einer Debugging-Aufgabe ist diesbezüglich wie folgt:

1) **Fehler aufdecken / rekonstruieren:** Durch das testweise Anwenden des Programms der Debugging-Aufgabe rekonstruieren die Schülerinnen und Schüler das in der Aufgabenstellung beschriebene bzw. angedeutete Fehlerverhalten des Programms und grenzen es ggf. auf bestimmte auslösende Bedingungen (z. B. bestimmte Benutzereingaben) ein.

2) **Fehlerquelle lokalisieren:** Ab einem zuvor gesetzten Haltepunkt werden das Programm im Tracemodus solange Schritt für Schritt durchlaufen und die Werte relevanter Variablen überwacht, bis eine Fehlfunktion des Programms (Laufzeitfehler) oder unpassende Variablenwerte (Semantikfehler) beobachtet und die auslösende, fehlerhafte Anweisung damit lokalisiert werden kann.

3) **Fehlerhaften Quelltext analysieren:** Nun gilt es, die Stelle(n) des Quelltextes, an der/denen ein fehlerhaftes Programmverhalten beobachtet wurde, so intensiv zu auf Fehler und deren Ursachen zu analysieren, dass eine Korrektur des oder der Fehler möglich wird.

4) **Fehler korrigieren:** Schließlich ist der Quellcode so abzuändern, dass der oder die vorliegende(n) Fehler nicht mehr auftreten.

Obiges Vorgehensmodell ist gegebenenfalls mehrmals zu durchlaufen, insbesondere bei mehreren Programmfehlern. Rücksprünge zu vorhergehenden Phasen innerhalb des Vorgehensmodells sind nicht nur möglich, sondern üblich.

Debugging-Aufgaben erfordern die Einarbeitung in einen fremderstellten, teils nur eingeschränkt lauffähigen Quelltext (Laufzeitfehler). Es ist deshalb vielfach sinnvoll, den Schülerinnen und Schülern zusätzlich das Programm erläuternde Materialien an die Hand zu geben – beispielsweise konzeptuelle Modelle (z. B. Use Case- und Klassendiagramme), compilierte Musterlösungen des Programms oder, bei realitätsnah gehaltenen Aufgaben, auch eine strukturierte Anforderungsspezifikation (siehe auch Kapitel 3.2.3).

Der Quelltextumfang einer Debugging-Aufgabe ist typischer Weise deutlich größer (Faktor 2 bis 3) als die Menge Code, die ein durchschnittlicher Lerner innerhalb einer Schulstunde selbst zu verfassen vermag. Hierbei gilt, dass je übersichtlicher, je strukturierter und je besser dokumentiert der Quelltext einer Debugging-Aufgabe ist, desto umfangreicher kann er tendenziell gestaltet werden (siehe auch Kapitel 3.3).

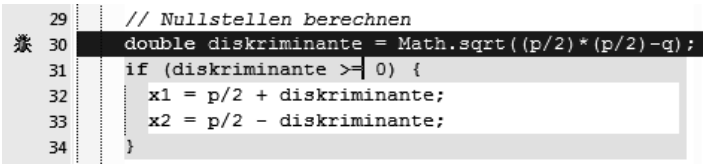

Abbildung 2: Ausschnitt aus dem (fehlerhaften) Quelltext einer Debugging-Aufgabe

Das Beispiel in Abbildung 2 zeigt einen Ausschnitt einer in das Thema einführenden Debugging-Aufgabe. Es sollen Nullstellen einer quadratischen Funktion mithilfe der p-q-Formel berechnet werden. Allerdings wurden im Quelltext zwei Fehler untergebracht, ein Laufzeit- und ein Semantikfehler. Beim ersten Testen des Programms würde das Programm regelmäßig abstürzen. Nach Setzen eines Haltepunkts in Zeile 30 und Ausführung dieser Zeile im Tracemodus würde die Quelle des Laufzeitfehlers (hier: Wurzel aus einer negativen Zahl) ebenfalls in Zeile 30 entdeckt werden. Diese könnte dann – z. B. durch das Ziehen der Wurzel erst in den Zeilen 32 ff. – korrigiert werden. Ein erneuter Programmtest würde nun den Semantikfehler in Form falsch berechneter Nullstellen offenbaren. Auch dieser wäre schnell gefunden; schon die Betrachtung der p-q-Formel nämlich offenbart ein fehlendes Minus jeweils vor „p/2“ (Zeilen 32 und 33).

3.2 Anwendungsgebiete

Debugging-Aufgaben empfehlen sich in mehrfacher Hinsicht für den Unterricht. Sie unterstützen bei der Vermittlung der Programmiertechnik Debugging (Kapitel 3.2.1). In der Algorithmik/Programmierung leiten sie Schülerinnen und Schüler bei der Analyse von Eigenheiten und Fehlerquellen einer Programmiersprache bzw. ausgewählter algorithmischer Konzepte an (Kapitel 3.2.2). In der Softwaretechnik sind sie ein empfehlenswerter Aufgabentyp für die Phasen Test und Wartung des Softwarelebenszyklus (Kapitel 3.2.3). Zudem haben sie sich als knifflige Zusatzaufgaben bewährt (Kapitel 3.2.4).

3.2.1 Debugging-Kenntnisse vermitteln

Bevor der Debugger im Unterricht genutzt werden kann, gilt es, den Schülerinnen und Schülern dessen Kernfunktionalitäten näherzubringen – also Haltepunkte, Tracemodus, Variablenbelegung (siehe auch Kapitel 2). Für die Vermittlung von Debugging-Tätigkeiten als solches empfehlen sich (einfache) Anwendungsaufgaben für den Debugger, sprich (einfache) Debugging-Aufgaben. Ein geeigneter Zeitpunkt für eine entsprechende Unterrichtseinheit ergibt sich nach Erfahrung des Autors dann, wenn alle grundlegenden Kontrollstrukturen der Programmierung behandelt worden sind. Der Um-fang einer diesbezüglichen Einheit ist mit zwei bis vier Unterrichtsstunden überschaubar.

Es sei angemerkt, dass eine Einführung in das Debugging als wichtige Programmiertechnik auch dann sinnvoll ist, wenn im weiteren Unterrichtsgang Debugging-Aufgaben nicht genutzt werden sollten.

3.2.2 Fehlerquellen erkennen

Debugging-Aufgaben führen Schülerinnen und Schüler in einem entdeckenden Lernprozess an die im Quelltext der Aufgabe untergebrachten Fehlerquellen; diese sind dann solange zu untersuchen, bis es gelingt den (verstandenen) Fehler zu korrigieren. Diesbezüglich sind Debugging-Aufgaben gut geeignet, um Eigenheiten und typische Fehlerquellen ausgesuchter Programmierkonzepte durch Schülerinnen und Schüler entdecken und ergründen zu lassen. Im Unterricht des Autors wurden Debugging-Aufgaben unter anderem erfolgreich eingesetzt, um den Unterschied von Deklaration und Initialisierung einer Variable herauszuarbeiten, um den (fehlschlagenden) Zugriff auf nicht vorhandene Ressourcen (Elemente eines Arrays, Dateien) zu demonstrieren und, um rekursive Funktionsaufrufe durch Schülerinnen und Schüler (entdeckend) nachvollziehen zu lassen.

Testweise gelang es im Unterricht des Autors sogar, Schülerinnen und Schüler eines Informatik-Grundkurses die softwaretechnisch komplexen Fehlerquellen von Rollenbeziehungen in Klassenhierarchien (vgl. [Fr00]) sowie der Ko- und Kontravarianz (vgl. [Pr01]) exemplarisch aufdecken und ergründen zu lassen – ohne Fehlerkorrektur allerdings.

3.2.3 Test und Wartung in der Softwareentwicklung

In einer Reihe zur Softwaretechnik haben Schülerinnen und Schüler aufgrund ihrer Vorerfahrungen in der Programmierung i. d. R. ein gutes Verständnis für die Herausforderungen und Tätigkeiten der Planung und Entwicklung von Software. Die Bedeutung des Testens und der (langfristigen) Wartung von Software ist ihnen intellektuell meist ebenfalls präsent, allerdings fehlt nicht selten, ein implizites, erfahrungsbasiertes Verständnis dieser Phasen. Hier bewähren sich Debugging-Aufgaben in zweifacher Weise. Im Vorfeld der Reihe erzeugen sie praktische Erfahrungen im Testen und Pflegen von Software. Während der Reihe selbst bieten sich Debugging-Aufgaben als praktische Übungen in den entsprechenden „Phasen“ an. Aus Sicht des Autors sind hierbei anspruchsvolle Debugging-Aufgaben empfehlenswert, welche an den Original-Kontext des Software-Testens anknüpfen – z. B. mit Blick auf die Einhaltung einer gegebenen Softwarespezifikation oder die Bearbeitung einer fiktiven Fehlermeldung („Trouble Ticket“).

3.2.4 Zusatzaufgaben

Debugging-Aufgaben erfordern die Beschäftigung mit fremdem Quellcode, weshalb sie sich fast beliebig zeitintensiv gestalten lassen. Die Besprechung der Aufgaben lässt sich dagegen meist in kurzer Zeit erledigen – schließlich gilt es (im Wesentlichen) nur, den oder die Fehler aufzuzeigen. Hierbei ist das Erklären eines Fehlers i. d. R. auch für Schülerinnen und Schüler nachvollziehbar, wenn sie die Aufgabe nicht oder nicht vollständig bearbeitet haben. Diese Eigenschaft empfiehlt Debugging-Aufgaben – auch aus der

Erfahrung des Autors heraus – in besonderer Weise als Zusatzaufgaben – z. B. um schnelle Programmierer im Unterricht sinnvoll zu beschäftigen.

3.3 Empfehlungen guter Praxis

Wie bei (fast) allen didaktischen Instrumenten, hängt auch bei Debugging-Aufgaben der (Bildungs-) Erfolg nicht zuletzt von der gelungenen Gestaltung und dem angemessenen Einsatz des Instruments ab. Einschlägige Unterrichtserfahrungen des Autors sind diesbezüglich in den nachfolgenden Regeln guter Praxis zusammengefasst und ergänzen die Evaluation des Konzepts (siehe Kapitel 4).

Der richtige Schwierigkeitsgrad

Das Austarieren des Schwierigkeitsgrads einer Debugging-Aufgabe ist nicht trivial: Aufgaben, bei denen es Schülerinnen und Schülern nicht gelingt, einen Fehler (in absehbarer Zeit) zu analysieren, führen kaum zu Lernerfolg, eher zu Frustration. Aufgaben mit geringer Komplexität wiederum werden vielfach ohne Debugger und ohne intensivere Fehleranalyse gelöst – z. B. durch simples Ausprobieren. Als Faustregeln hat es sich diesbezüglich bewährt, Quelltexte in der zwei- bis dreifachen Länge dessen zu nutzen, was eine mittelschnelle Schülergruppe[9] in einer Unterrichtsstunde typischer Weise zu verfassen vermag. Andererseits hat es sich als sinnvoll herausgestellt, nur solche Quelltexte in einer Debugging-Aufgabe zu nutzen, bei denen auch ein erfahrener Programmierer einen Debugger einsetzen würde.

Bezüglich der Ausrichtung des Schwierigkeitsgrades ist ebenfalls zu beachten, dass Schülerinnen und Schüler durch ihr mathematisches Vorwissen mathematische Laufzeitfehler (Division durch Null, Wurzel aus einer negativen Zahl) gut verstehen und schnell erkennen. Entsprechende Aufgaben bieten sich somit vornehmlich für den Einführungsunterricht in das Debugging an.

Gestaffelte Fehler in gestaffelter Schwierigkeit

Im Unterricht des Autors differierte die Bearbeitungszeit für Debugging-Aufgaben zwischen einzelnen Schülergruppen häufig sehr stark – dem Eindruck nach deutlicher als bei Programmieraufgaben. Diesbezüglich zeigten sich solche Debugging-Aufgaben als besonders geeignet, die mehrere Fehler in gestaffelter Schwierigkeit enthielten – z. B. ein Laufzeitfehler und ein Semantikfehler. Bei entsprechenden Aufgaben zeigte es sich als wahrscheinlich, dass auch langsamere Aufgabenlöser Teilergebnisse bereits erreichen, noch bevor die schnelleren Löser die Aufgabe als Ganzes bewerkstelligen. Gleichzeitig wurde aus Schülerrückmeldungen deutlich, dass Schülerinnen und Schüler Debugging-Aufgaben besonders dann als sinnvollen Lerngegenstand erachten, wenn während der Aufgabe ein steter Lösungsfortschritt klar zu erkennen ist. Umgekehrt verunsicherten Aufgaben mit logisch ineinander verwobenen Programmfehlern insbesondere weniger programmiersichere Schülerinnen und Schüler merklich.

[9] Im Unterricht des Autors arbeiten Schülerinnen und Schüler ausschließlich in Kleingruppen am PC.

Struktur – Struktur – Struktur

Das Einarbeiten in fremden Quellcode gelingt grundsätzlich leichter, wenn dieser wohl strukturiert und modularisiert ist, ausführlich kommentiert wurde und gut leserlich implementiert ist. Ergänzende Formen der Dokumentation, insbesondere konzeptuelle Modelle wie Schnittstellenbeschreibungen, Use Case- oder Klassendiagramme unterstützen die Einarbeitung zusätzlich. Es empfiehlt sich diesbezüglich in mehrfacher Hinsicht, Schülerinnen und Schülern in Debugging-Aufgaben wohl strukturierte Quelltexte und zusätzliche Dokumentation zu Verfügung zu stellen: Erstens, weil dadurch die Bearbeitung der Aufgabe erleichtert wird. Zweitens zeigte sich, dass je leichter verständlich ein Quelltext gestaltet war bzw. je zugänglicher er durch begleitende Dokumente wurde, desto umfangreicher konnte er sein, so dass auch größere Programmtexte regelmäßiger Teil des Informatikunterrichts werden können. Drittens erfahren Schülerinnen und Schüler den Nutzen von Struktur, Modellierung und Dokumentation insbesondere dann sehr deutlich, wenn sie, wie in Debugging-Aufgaben, in einer direkten Weise hiervon abhängig sind bzw. hiervon profitieren.

Klare Arbeitsaufträge

Auch im Debuggen geübte Schüler neig(t)en zuweilen dazu, die im Quelltext versteckten Fehler ohne den Debugger finden und korrigieren zu wollen, was bei umfangreicheren Debugging-Aufgaben i. d. R. nicht gelang und die eingesetzte Lernzeit somit weitgehend unproduktiv verging. Es hat sich diesbezüglich als sinnvoll und hinreichend herausgestellt, Debugging-Aufgaben klar als solche zu betiteln und in diesen – auch gegenüber damit vertrauten Schülern – dezidiert die Verwendung des Debuggers einzufordern.

4 Evaluation und Einordnung

Debugging-Aufgaben sind keine gesonderte didaktische Methode oder ein eigenständiges Unterrichtskonzept, sondern (nur) ein spezieller Aufgabentyp, der an geeigneter Stelle in den Unterricht integriert wird (siehe hierfür Kapitel 3.2). Mit Ausnahme der Vermittlung des Debuggings als solches bestimmen das Debugging bzw. Debugging-Aufgaben keine Unterrichtseinheit, sondern ergänzen diese. Gleichzeitig sind Debugging-Aufgaben aktuell noch in einer – wenn auch gereiften – Entwicklungsphase. Vor diesem Hintergrund erfolgte die Evaluation der Debugging-Aufgaben nicht durch eine statistische Messung des durch Debugging-Aufgaben erzeugten Lernerfolges – z. B. durch die vergleichende Evaluation zweier Lerngruppen –, sondern anhand einer Schülerbefragung und der (qualitativen) teilnehmenden Beobachtung durch den Autor.

Die hier grundgelegten Unterrichtserfahrungen entstammen drei Informatikgrundkursen mit 11 (Kurs A), 16 (Kurs B) bzw. 23 Schülerinnen und Schülern (Kurs C), in denen der Autor Debugging-Aufgaben einsetzte und testete: Alle Kurse erhielten eine Einführung in das Debugging (siehe Kapitel 3.2.1) und hatten Debugging-Aufgaben im weiteren Fortgang der Algorithmik/Programmierung zu bearbeiten (siehe Kapitel 3.2.2 und 3.2.4). Nur ein Kurs, Kurs A, bearbeitete Debugging-Aufgaben bislang in der Softwaretechnik-Reihe (siehe Kapitel 3.2.3).

In allen drei Kursen fand eine Schülerbefragung statt, in welcher die Schüler aufgefordert waren, ihre Erfahrungen mit Debugging-Aufgaben in wenigen Worten zu formulieren. Das Feedback des ersten Kurses (Kurs A) floss direkt in die Verbesserung derjenigen Aufgaben ein, die den Folgekursen präsentiert wurden. Aus den Rückmeldungen der beiden weiteren Kurse wurden – mit Blick auf den vorliegenden Beitrag – die folgenden fünf Aussagen im Plenum ausgewählt, welche nach Ansicht der Anwesenden geeignet sind, die geäußerten Argumente prägnant zu verdichten:

- *„Man weiß, wie man sich selbst helfen kann."*
- *„Einige kritische Fehlerquellen wurden mir besser verständlich."*
- *„Fehlererkennen ist noch nicht das Fehlerbeseitigen. Dies ist eine zusätzliche Schwierigkeit."*
- *„Das Lesen und Verstehen fremder Programme war eine ganz eigene, neue Herausforderung"*
- *„Debugging-Aufgaben bringen Abwechslung in den Unterricht. Selbst programmieren ist aber interessanter."*

Einschlägige Unterrichtserfahrungen des Autors flossen bereits in die Kapitel 3.2 und 3.3 in Form von sinnvollen *Anwendungsgebieten* bzw. *Empfehlungen guter Praxis* für Debugging-Aufgaben ein. Die dort aufgeführten Eigenschaften von Debugging-Aufgaben bzw. des zugehörigen Unterrichts zählen zu den Inhalten einer Evaluation des Konzeptes. Sie sollen aber an dieser Stelle nicht wiederholt werden. Der Leser wird stattdessen gebeten, den Inhalt der beiden genannten Kapitel in seine Gesamtwürdigung mit einzubeziehen. Im Folgenden werden ergänzend noch zwei (weitere) prägende Merkmale von Debugging-Aufgaben erläutert, die sich nicht in die Inhalte der vorstehenden Kapitel integrierten:

Klarer Fokus auf Details: Debugging-Aufgaben befördern auf Seiten der Schülerinnen und Schüler entdeckendes Lernen. Der Lösungsweg der Aufgaben ist i. d. R. durch die gesetzten Fehler und die Funktionsweise des Debuggers aber eng vorgegeben. En gros sind Debugging-Aufgaben also keine offenen Aufgaben. Sie fokussieren vielmehr auf konkrete, identifizierbare Fehler und den diesbezüglichen Programmausschnitt. Intensiv analysiert werden hierbei in aller Regel (nur) die fehlerhaften Quelltextausschnitte.

Nicht unbeträchtlicher Erstellungsaufwand: Die Erstellung der Debugging-Aufgaben des Autors war – verglichen mit gängigen Programmieraufgaben – mit deutlich erhöhtem Aufwand verbunden. Denn es galt, zuerst ein fehlerfreies Programm sowie begleitende Dokumentation zu erstellen, wobei der Quelltextumfang denjenigen typischer schulischer Programme merklich übersteigt (vgl. Kapitel 3.1). Erst dann wurden die für den Unterricht relevanten Fehler im Quelltext verankert. Die Erstellung einiger, vom Autor genutzter Debugging-Aufgaben nahm jeweils mehr als einen Arbeitstag in Anspruch.

5 Fazit und Ausblick

Debugging-Aufgaben haben sich im Unterricht des Autors bewährt; sie sind dort mittlerweile fester Bestandteil. Sie haben sich hier als (sinnvolle) Ergänzung all der Unterrichtsreihen gezeigt, in denen intensiv programmiert wird – im Unterricht des Autors sind dies die Algorithmik/Programmierung, die Softwaretechnik und die (ebenfalls programmierintensiv gehaltene) Kryptologie.

Die geschilderten Vorteile von Debugging-Aufgaben verbinden sich mit der deutlichen Herausforderung eines teils sehr beträchtlichen Erstellungsaufwands. Diesbezüglich ist es weder sinnvoll, noch erwartbar, Debugging-Aufgaben von jeder Informatiklehrkraft allein kreieren zu lassen. Hier ist vielmehr die Informatik-Didaktik-Community gefordert, den Praktikern vor Ort qualitätsgesicherte und gut dokumentierte Aufgaben zu Verfügung zu stellen. Einen ersten Schritt in diese Richtung plant der Autor selbst. Ein an Informatiklehrer und -lehrerinnen gerichteter Artikel, der die Anwendung von Debugging-Aufgaben im Unterricht anhand von Beispielen praxisnah aufzeigt, ist in Vorbereitung. Parallel hierzu will der Autor ausgewählte Debugging-Aufgaben online zu Verfügung stellen.

Literaturverzeichnis

[Ba05] Balzert, H.: Lehrbuch Grundlagen der Informatik. Elsevier, München, 2005.

[En06] Engelmann, L (Hrsg.); Bartke, P., Burkhard, H.-D. et al.: Informatik – Gymnasiale Oberstufe. Duden, Berlin, Frankfurt/Main, 2006.

[Fr00] Frank, U.: Delegation: An Important Concept for the Appropriate Design of Object Models. JOOP – Journal of Object-Oriented Programming. 13 (2000) 3, S. 13-18.

[GB+09] Grechenig, T.; Bernhart, M.; Breiteneder, R.; Kappel, K.: Softwaretechnik. Pearson, München et al., 2009.

[Ha10] Hattenauer, R.: Informatik für Schule und Ausbildung. Perason, München et al., 2010.

[Hu06] Humbert, L.: Didaktik der Informatik – mit praxiserprobtem Unterrichtsmaterial. Teubner, Wiesbaden, 2006.

[Mo04] Modrow, E.: Einführung in die Informatik – Teil 2 – Java-Anweisungen und -Ausdrücke. Virtuelle Lehrerweiterbildung Informatik in Niedersachsen, Scheden, 2004. [http://www.vlin.de/material_2/Struktogramme-Java.pdf, 03.03.2013]

[Mo11] Modrow, E.: Visuelle Programmierung – oder: Was lernt man aus Syntaxfehlern? Thomas, M. (Hrsg.): Informatik in Bildung und Beruf (INFOS 2011).: Gesellschaft für Informatik, Bonn, 2011, S. 27-36.

[Pr01] Prasse, M: Entwicklung und Formalisierung eines objektorientierten Sprachmodells als Grundlage für MEMO-OML. Fölbach, Koblenz, 2001.