Internalizing a Viable Mental Model of Program Execution in First Year Programming Courses

Robert Goltermann[1], Frank Höppner[2]

Abstract: While there are many tools available to evaluate student solutions to programming exercises by means of unit tests, there seems to be only little support for earlier stages. Students have to *apply* their knowledge about instructions, data organization, program flow when programming – but usually we have little to no means to check beforehand if they have fully understood the instructions they are using. While creativity and problem solving skills are also necessary to write correct programs, a lack of understanding the mechanisms behind instructions surely prevents a student from delivering correct solutions. We suggest a web-based tool that lets the students enter their expectations of what is happening at execution time. The solution is automatically generated from source code and the students receive immediate feedback. The tool may be used for training, assessment and in-class discussion and feedback.

1 Introduction

The technical support for an automated grading of exercises in first year programming courses seems to concentrate on validating solutions of programming exercises submitted by the students. The sources are compiled and a number of unit test cases are executed; the number of satisfied test cases provides a straightforward mechanism for grading. But students who are new in programming often struggle with automated tests (yet another tool, yet another type of error message). We often observe that they misunderstand the exercises (either because they have read it superficially or because terms do not (yet) trigger the same associations as they do with more experienced students), so they have a hard time in fulfilling the tests. The binary feedback (red/green) does not reward progress below the correctness threshold. To get good grades they are doomed to get a green bar, so they often start to apply trial and error rather than thinking about their code and testing it on their own. Fighting the red bar may distract beginners from concentrating on the core concepts of the programming language.

Unit tests are not so helpful in distinguishing the different reasons of failure. We can think of at least three types of reasons: (T1) misunderstanding of the exercise, (T2) overlooked intricacies of the problem, (T3) erroneous understanding of the programming language. A misunderstanding of the exercise (T1) may be prevented by handing out test cases in source code or examples in plain text together with the exercise, so they can align their own perception of the task with the tests. Problem (T2) addresses a core competence of computer scientists: get trained to keep track of all special cases our code needs to cover

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

to work correctly. We are providing many exercises to our students in order to train this competence. However, if a student has not yet fully understood the fundamental elements of the language (T3), it is impossible to write a correct solution other than by chance.

For the last problem (T3) someone has to identify what the student got wrong about the language/statements, so that the misconception can be corrected. Once the problems have been resolved, the student may be able to solve the task correctly. But a submission system with test-based grading does not offer this kind of help and therefore puts an additional burden on the students, which makes them struggle even more with programming. It raises the pressure and – given the strict time constraints – students may start to copy and paste solutions from fellow students rather than practicing themselves. As the submission systems cannot distinguish both cases, the next step is often a call for an automated plagiarism check (plagiarism as a cry for help [Vog09]).

What are we doing to prevent problem (T3)? The lecturer explains the statements, gives examples and encourages the students to solve first tasks – but cannot supervise all students individually. If the courses are not too large, additional staff may offer help and discover misconceptions regarding the lifetime or visibility of variables, the way arrays are handled, the exact semantics of the for-loop, etc. But too many students simply do not want to ask for help. So we face the situation that students start to program before they confidently know the mechanism behind the statements they use. Asking straightaway for own solutions may be asking for too much. Instead, we should convince ourselves first that the semantics is well understood. If the mental model of how instructions operate is wrong, a working solution cannot be found by *thinking about the program* (because this may lead to correct solutions in the wrong model, which typically tend to be wrong solutions in the correct model), but only by frustrating and demotivating trial and error.

If writing own programs is on the 'application level' of the Bloom taxonomy then we have automated test-based grading as a tool to support grading this skill level. But the 'comprehension level' precedes the 'application level' and it seems that there is not much tool support at this stage. We know that many lecturers use pencil and paper approaches to clarify what actually happens at execution time. So did the second author in his lectures. But it is impossible to provide individual feedback to all students whenever new elements of the language have been introduced. More tool support at the comprehension level would be welcome. This paper describes such a web-based tool.

2 Mental Model of Program Execution

Many instructors report about difficulties of teaching programming. We are not focusing on difficult concepts such as concurrency but assignment, conditions, iteration, recursion, arrays, objects and so forth. For such a simple concept like assignment, test questions were developed and presented to students and matched against a broad variety of possible *mental models* (such as right-to-left copy, left-to-right copy, equality, ...) [BDS08]. One of the results was that many students in their study did not stick *to the same model*, but changed between different models from question to question. Succeeding in the course was attributed to a consistent behavior (even when it was wrong initially).

This study of Dehnadi and Bornat encouraged others "to study the range and consistency of mental models held by students of both simple (value assignment) and more challenging (reference assignment) programming concepts towards the end of a first-year course" [MFRW07]. In other disciplines (e.g. physics) it is considered useful to have different models for the same phenomenon (cf. wave-particle duality of light). There it might be appropriate to switch between different models in different situations. It becomes an interesting research question how to combine the models and what the origins of (non-viable) mental models might be [CG81]. Such knowledge might then be used to reveal how a learning processes must look like. Instructors may adopt their material according to the mental models they are facing. In this line of argumentation the authors of [MFRW07] state that "the problem of helping students to convert this diverse and persistent range of non-viable mental models into viable ones presents a real challenge to computer science educators".

However, we see a fundamental difference between physics and programming. With physics there is no single universal model and therefore reasoning by analogy might be the best path to follow. With programming, however, we have a human-manufactured artificial environment that behaves deterministically and the question is only how detailed a mental model should be, but we do not have to rely on analogies to successfully predict a programs outcome. Rather than learning how a students mental model may look like (to see if it is viable and to correct it if necessary), we consider it more efficient to establish a viable model straight from the beginning.

This vision is shared by [LKPC04], where a number of instructor guidelines are presented, e.g.: (1) Show behavior, no analogy (e.g. do not use the analogy of boxes for variables). (2) Build concepts first, jargon last (e.g. terms alone may provoke wrong presumptions). (3) Constant repetition. The last point is quite important, because reasoning by analogy connects to ideas that have been consolidated over the past (which is why we may find them easier to argue with). Constant repetition may be the only way to prevent non-viable concepts from infiltrating the students mental model of program execution.

We consider approaches such as Jeliot [MMSBA04] or Jive [LJC10] as steps into this direction, which actively support the construction of a viable mental model by visualizing what actually happens at execution time, but the student remains passive. A very popular technique (although there might not be so many publications about it) are hand-written charts illustrating the flow of execution and how variables changes correspondingly. The usage of such techniques may lead to a significant increase in student performance [HJ13].

3 Memory Traces in Programming Courses

3.1 The Protocol

In our lectures we have used a hand-written memory trace successfully for several years in first-year programming courses. It has been designed to give a precise idea of what is actually going on while the program gets executed. It drops many details, but is realistic

and complex enough to detect problems novices typically face. Fig. 1 shows how such a memory trace is organized: It consists of the source code and the execution order right next to it. The students number the lines of the source code in the order in which they get executed. Source lines receive multiple step numbers if the line gets executed more than once (e.g. due to a loop or repetitive function call). In the grid on the right, the horizontal axis corresponds to the execution steps that were written next to the source code lines. Every column of the grid corresponds to a snapshot of the memory at a specific step. The content of stack and heap are logged above and below the x-axis.

For instance, in the program shown in Fig. 1 the line **int** [] a=**null**; is executed first, which is why it gets the step number 1 (green). In the trace (right part) the column marked 1 contains the content of stack and heap *just after step 1 has been executed completely*. As no objects are in the heap yet it is empty, but the new local variable a resides on the stack. In Step 2 (next line) variable a gets a reference to a new array object in the heap. The content can be placed anywhere on the heap as long as the value of a points to the correct heap position (virtual reference 0x4). (The size of the array precedes the elements.)

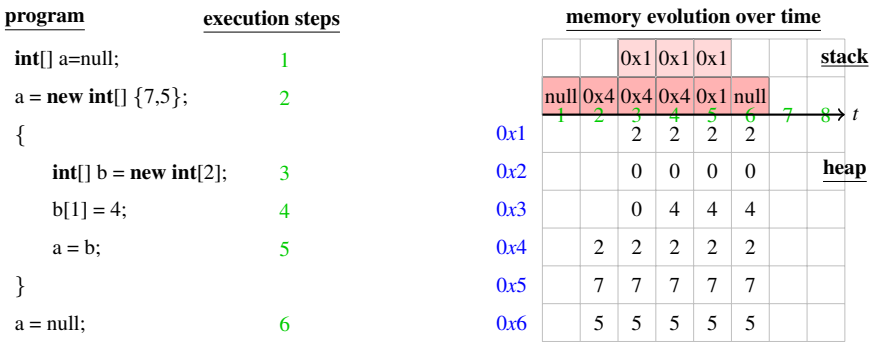

Fig. 1: Paper version of a memory trace for offline work.

Many other types of useful visualization exists (e.g. UML object diagrams). We use them at a later point in time when the number of attributes of variables, objects and methods increases. It is visually more appealing to use arrows to indicate which reference points to which object, but on the other hand such object diagrams do not capture the evolution of memory over time: they constantly change while explaining the execution. If a student makes a copy of the diagram it does not capture the discussion completely, whereas the memory traces can be used to comprehend the execution step by step at home. Following [LKPC04] (concepts first, jargon last) we clarify the concept of memory addresses before using phrases such as 'variable a is pointing to ...' (visualized by arrows).

.

3.2 Use of Memory Traces in the Lectures

Past Use. Whenever a new concept is introduced in class and its semantic has been explained, one or more traces are filled out on the blackboard interactively by the students.

The instructors explanations are not necessarily understood by every student exactly as they were meant. So filling out the trace immediately afterwards is important to spot differences in the perception.

During the course, the lecturer presents various small *challenges* to the students, which have to be answered via classroom response systems (aka 'clickers'). The challenges are more difficult questions that students easily get wrong in the first try. Once the voting has demonstrated that the students do not come to the same conclusion, the log is filled out on the blackboard. This offers the opportunity for every student to identify the individual reason for the wrong answer, covering (T1), (T2) and (T3) – but the lecturer usually does not know which type of problem applies to how many students.

To resolve misconceptions early, the traces were used in mid-term exams. Grading the traces provides individual feedback, but manual grading is extremely time consuming. Asking for general feedback (like: how many cells did you get wrong?) gives an impression how well the trace was filled out in general, but gives no idea of what was wrong in particular. (Quite often mistakes are made at points that were already introduced earlier, which calls for more practice and more grading.) The students were generally encouraged to fill out more traces and hand them in (in particular if a misconception was detected), but this offer is only seldomly accepted (so the missing scalability of manual grading does not really become a problem).

Intended Future Use. Ideally, a web-based tool provides multiple ready-to-use examples that can be filled out by the students at any time they want. A first example is provided and discussed with the whole class, a second example will be filled out by the students individually (using their notebooks). A few minutes later, the lecturer can see a filled-out memory trace where the entries are color-coded depending on the number of wrong entries. He may then decide to explain the topic in greater detail or point to students to earlier sections and other tracing tasks they should handle. (The application itself could also suggest related tasks.) Ideally, the tool provides support to let the lecturer drop a few lines of Java code and let all students fill out the trace spontaneously.

As manual grading is no longer necessary, this simplifies continuous feedback. Students can improve their understanding by filling out additional traces with immediate feedback (avoiding an uncomfortable situation of asking an advisor to grade a trace once more). Being part of the homework, students get more and more used to filling out traces correctly (cf. Sect. 2: repetition!), so the comprehension of integral parts of the language improves. Eliminating problems of type (T3) hopefully enables students to better cope with unit tests in the future. Being able to clearly state the expectations of what happens next is fundamental for successful debugging sessions at later stages of the course, too.

4 Web-Based Memory Trace

We have implemented a web application that enables students to fill out the memory trace online. Although the web interface does not yet allow to drop new programs as a spon-

taneous in-class exercise, the technical foundations are already laid. It is not necessary to provide a handcrafted solution but the Java code is executed and the memory trace is generated automatically using the Java Debugging Interface (JDI)[3].

```
1  int s = 4;
2  {
3    int j = s*3;
4    s+=++j;
5  }
6  System.out.println(s);
```

Fig. 2: Change of local variable in last line of block.

```
1  int i = 2;
2  int a = f(i);
3  if (a>0) {
4    int j=3*(i%7); a+=j;
5  }
6  System.out.println(a);
```

Fig. 3: Local variable in one-liner block.

```
1  int y;
2  if (x>2) {
3    y=0;
4  } else {
5    y=2;
6  }
```

Fig. 4: Deferred initialization of local variable.

While some parts of the implementation were straightforward, some other properties of the paper-based memory trace where somewhat difficult to retain. The memory trace contains all values *after having executed* the code line associated with this step. This is different from how the debugger behaves normally: it stops at the beginning of a code line and after stepping over the line it stops again at the beginning of the next code line to be executed. In line 4 of Fig. 2 the last statement of a code block changes a local variable. Stepping over line 4 causes the debugger to stop no earlier than at line 6. At this point in time, the local variable j does no longer exist and the change of its value (line 4) cannot be observed. An even worse situation is shown in Fig. 3 where a block contains (for educational purposes) a local variable (that must be reported on the stack). Using the standard step-over procedure, the content of i is never observable. A third problem is shown in Fig. 4, where the initialization of local variable y is deferred to line 3 or 5. As the debugger does not show uninitialized variables, y is not visible at line 1 or 2. While the last problem can be circumvented by simply providing *some* initialization for any local variable, the first two cases were more difficult to solve by using a mixture of line breakpoints and guarded variables.

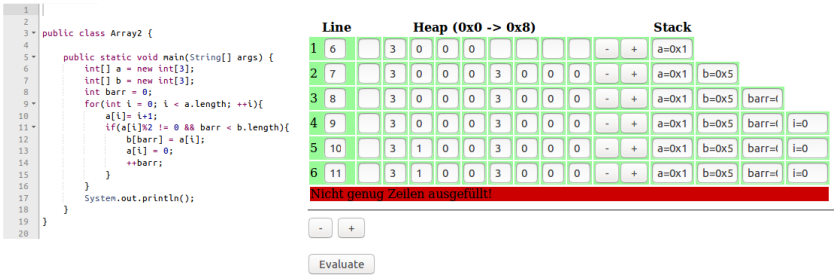

Fig. 5: Example of a partly filled memory trace in the web frontend.

A screenshot of a partly filled memory trace in the web application is shown in Fig. 5. Compared to Fig. 1 the layout is tilted: One row (rather than one column) stores the heap and stack content per step. The students can extend the trace and stack content by clicking on the respective +/− buttons. Upon clicking '*Evaluate*' all steps that were entered so far are evaluated and correct lines are shown in green, erroneous lines in red. Currently we

[3] `https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/`

provide no further hints on the location of errors within the line to prevent trial and error. A red line should make the students think over the corresponding step.

5 First Results

We are in an early stage where the first course was able to test the online tool. We received important feedback regarding the usability that has to be considered in upcoming versions.

In a first test scenario we divided a group of students into two groups A and B (of roughly equal size, first term students, fourth week of term). Based upon a questionnaire about previous knowledge in programming we took care that both groups had similar skills on average. The memory traces have been used in class during the four weeks. All students were given 30 minutes time for practicing, group A got printed exercises (and was encouraged to discuss with other students) and group B used the online tool. Afterwards, a printed exercise was given to all students. (The final exercise was in the same format as all the other printed exercises, but somewhat different from how the memory trace had to be entered in the online form.) We analyzed the results with respect to some key points. One aspect was the identification of the life span of a variable (variable disappears from the stack if defining block ends). The results are shown below. There is a statistically significant difference between both groups (χ^2 independence test).

	practice		
	paper based	tool based	
variable life span wrong	11	3	14
variable life span correct	12	26	38
	23	29	

$\chi^2 = 7.353, p < 1\%$

Students were also asked to provide feedback at the end of the test. Only students that had used the tool provided answers to these questions. Fig. 6 shows the distribution of answers for two examples: We asked for the students to rate the sentences “The tool was helpful for my preparation” (left) and “The paper-based version would have been sufficient.” (right). Answers were provided on a Likert scale where 1 means ’strongly disagree’ and 5 means ’strongly agree’. Overall, the students appreciated the tool support.

Fig. 6: Student rating of the sentences “The tool was helpful for my preparation” (left) and “The paper-based version would have been sufficient.” (1=strongly disagree, 5=strongly agree).

6 Conclusions

Without having a clear understanding of what happens in for-loops, array instantiation or reference assignment, programs written by the students will only be correct by chance. This holds particularly for those who have never programmed before, but also for those who have already some experience but a non-viable mental model. For both groups it is important to clarify the semantics beyond any doubts to solve subsequent, increasingly complex programming tasks successfully. After all, it is one of the advantages of computer science that a well-defined semantic exists. Filling out a memory trace can help, but only if the author gets feedback from the staff, which is typically expensive and therefore done infrequently. The availability of such an online tool enables us to acquire a necessary level of repetition (including feedback) to ensure that programming statements are fully understood before they are applied. The tool is helpful in front of the class (all students fill out the trace simultaneously and we immediately highlight difficult parts in the trace) as well as assessment situations. It prepares for upcoming debugging sessions and may save shy students from asking questions in front of the class. The first feedback was encouraging. For upcoming courses we plan to include a couple of weekly tracing exercises for homework for every new concept that was introduced in class. We also plan to guide the students in resolving problem on their own (e.g. provide explanations once the same type of error has been made *n* times) and automatically suggest follow-up exercises.

References

[BDS08] Richard Bornat, Saeed Dehnadi, and Simon. Mental models, consistency and programming aptitude. *Conferences in Research and Practice in Information Technology Series*, 78(1986):53–61, 2008.

[CG81] Allan Collins and Dedre Gentner. How people construct mental models '. *Cultural models in language and thought*, pages 243–265, 1981.

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

[LJC10] Demian Lessa, Bharat Jayaraman, and Jeffrey K. Czyz. JIVE: A pedagogic tool for visualizing the execution of Java programs. 2010.

[LKPC04] Andrew K Lui, Reggie Kwan, Maria Poon, and Yannie H Y Cheung. Saving weak programming students. *ACM SIGCSE Bulletin*, 36(2):72, 2004.

[MFRW07] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmers. *Proceedings of the 38th SIGCSE technical symposium on Computer science education - SIGCSE '07*, page 499, 2007.

[MMSBA04] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. *Proceedings of the working conference on Advanced visual interfaces - AVI '04*, page 373, 2004.

[Vog09] D Vogts. Plagiarising of source code by novice programmers a "cry for help"? *Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2009*, (October):141–149, 2009.