PandionJ: A Pedagogical Debugger Featuring Illustrations of Variable Tracing and Look-ahead

André L. Santos
Instituto Universitário de Lisboa (ISCTE–IUL)
PORTUGAL
andre.santos@iscte-iul.pt

Hugo S. Sousa
e.Near
PORTUGAL
hugossousa92@gmail.com

ABSTRACT

We present PandionJ, a pedagogical debugger for Java with innovative features regarding how the program state information is presented to users. We consider aspects that are either not available or not fully automated in existing debuggers (pedagogical or not), such as illustration of the history of variable values and look-ahead of their future state. Our approach relies on static analysis of code in order to infer variable roles, relationships, and behavior. This information is used to render illustrations of program state that existing debuggers are not capable of providing without requiring additional user input.

CCS CONCEPTS

• **Social and professional topics** → *Computing education*; • **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

programming pedagogy, debuggers, visualization

ACM Reference Format:
André L. Santos and Hugo S. Sousa. 2017. PandionJ: A Pedagogical Debugger Featuring Illustrations of Variable Tracing and Look-ahead. In *Proceedings of 17th Koli Calling International Conference on Computing Education Research (Koli Calling 2017).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3141880.3141911

1 MOTIVATION

We carried out a study with programming instructors that revealed illustration patterns in their explanations of program execution [6]. Many of these illustration patterns either have no representation or are not fully automated (i.e. additional user input is necessary) in existing pedagogical debuggers and animation tools (e.g., [1, 3–5]). Further, despite the numerous pedagogical tools for program visualization that have been proposed, such tools are not widely used in teaching [2]. One the main reasons given by instructors is that they prefer to develop their own visualizations. Based on our study, we speculate that one of the possible justifications for this is that the illustrations developed by instructors (drawings or animations) tend to be considerably richer than the ones provided by the tools.

Koli Calling 2017, November 16–19, 2017, Koli, Finland

ACM ISBN 978-1-4503-5301-4/17/11…$15.00
https://doi.org/10.1145/3141880.3141911

Based on the assumption that the information drawn by programming instructors in their explanations is useful for student comprehension, we believe that a pedagogical debugger whose visualizations resemble those could be beneficial to the learning process. The goal of our approach is to incorporate in a debugger several illustration patterns used by programming instructors, automating them with the purpose of helping on programming education (debugging exercises, understanding runtime errors, comprehension of elementary algorithms).

2 DEBUGGING ENVIRONMENT

PandionJ consists of a view of debugger-provided information that renders the state of program variables in graphical way, taking into account variable information obtained from static analysis of the source code. Our proof-of-concept implementation supports Java and was built as a plugin for Eclipse, that integrates with its debugger infrastructure (see Figure 1).

The information provided by a debugger engine is simply a set of variable-values pairs held in call stack frames. In order to automatically render our illustrations (i.e., without relying on any additional information provided by users), we collect additional information about variables, related to their roles [5] (e.g., Gatherer, Stepper, Most-Wanted Holder), how they relate to other variables (e.g., an iterator for array indexes), and their behavior (e.g., constant, incrementer/decrementer). PandionJ performs static analysis of the code under execution in order to obtain such information. When illustrating program state, the variable values provided by the debugger are depicted according to the information extracted from the source analysis. The illustrations do not rely on any sort of source code annotations or naming conventions on the user code. To our knowledge, there are no pedagogical debuggers available that combine static analysis with runtime information in order to provide fully automated visualizations of program state.

Variables are segmented according to the active frames of the call stack. For each stack frame, values (variables of primitive type) are represented as name-value pairs, while references are connected to the objects (including arrays) they are pointing to (if not null). Value variables may be depicted differently according to their role in the method, its relationship with array variables, or predicted behavior. Figure 1 presents an example of PandionJ in action, showing an executing function for summing the values of an array within a given interval of array indexes [a, b] (parameter validation is ignored here). The execution is suspended at a point where the loop is about to perform its third iteration.

Tracing. PandionJ features visualizations that enable users to trace the current value of a variable. One case pertains to array index iterators (Stepper ă[5]), which are depicted next to the position of

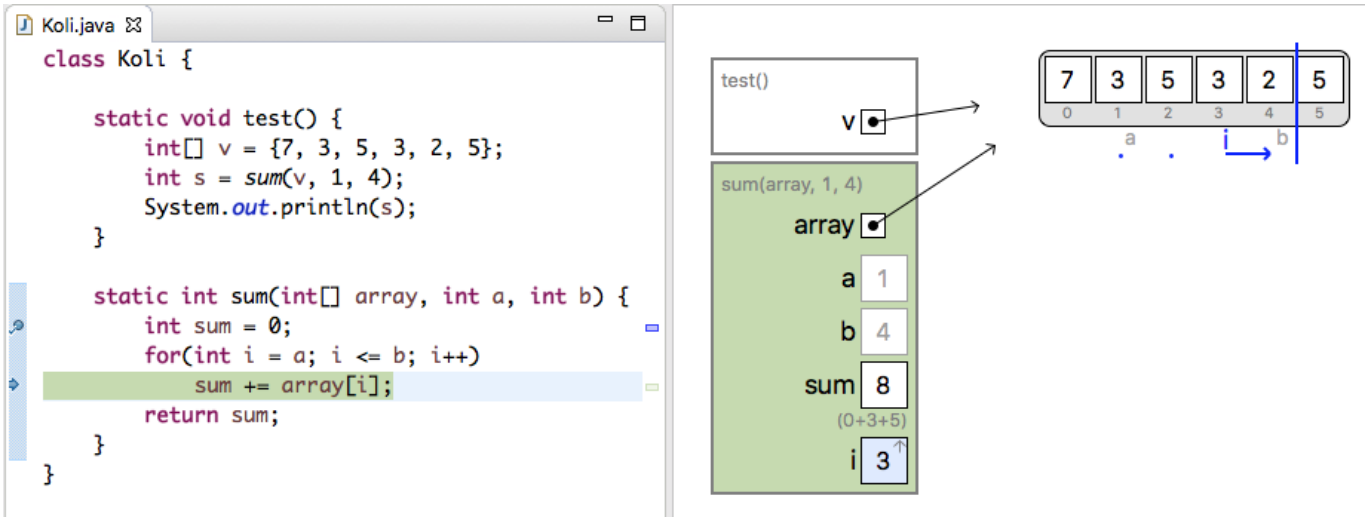

Figure 1: PandionJ illustrating a function for summing the values contained within a given interval of array indexes.

the target array, while the trace of its previous values is visible (trailing dots). In the example, variable i is classified as such, given that it is only used to access positions of the array v. Parameters a and b are considered array index accessors, and are also represented next to the respective array positions. Although they are not used to access the array directly, they are used to initialize an iteration variable (i = a) and constrain its growth (i <= b).

Another tracing feature relates to Gatherer variables [5], whose values result from a successive accumulation of values. In the example, variable sum is classified as such given that it is modified only through an accumulation instruction (+=). The tool displays the parcels that decompose the current value of the variable. Yet another case, not illustrated in the example, is when the variable is identified as a Most-Wanted Holder ă[5], for which the tool displays an history of the successive values that were superseded by the newer ones.

Look-ahead. PandionJ features visualizations that anticipate variable behavior, in order to enable users to look-ahead the future state of the variable. Array index iterator variables may be classified as *incrementers* or *decrementers* if all the instructors that modify them are increments/decrements. In the example, variable i is classified as an incrementer, since its value only changes through the instruction i++. Using this information, the tool depicts a *direction* of the array iterator with an arrow pointing to the right/left (incremental/decremental). In addition to this classification, an array iterator might have an associated *bound*, which consists of an expression that constrains the upper/lower value of an array iterator. In the example, variable bis a (closed) bound for the incrementer variable i. We can also see variable i with a right-direction pointing to its bound (represented by the bar), which in this case is a variable (b), but it could be any side-effect-free expression.

Variables whose value never changes (so-called *constants*, or Fixed Valuesă[5]) are also represented differently (grayed out), despite if they were explicitly programmed as such (using the final keyword in Java). In the example, the constants are the function parameters (also classified as array index accessors, as explained).

3 CONCLUSIONS

Notice that the four variables of the example function sum are all integers but have very different purposes in the function. Whereas a conventional debugger would render these equally in a list of name-value pairs, PandionJ depicts them differently according to their behavior. While our visualizations are similar in some ways to previous approaches, PandionJ has the advantage of not requiring any user input for rendering the illustrations as such. For instance, PlanAni [5] introduced different variable visualizations according to variable role, however the scripts have to be manually developed by instructors. jGRASP [1] renders arrays and iterators in a similar way as ours, but the bindings of iterators to arrays have to be manually defined by users and there is no representation of variable direction.

REFERENCES

[1] James Cross, Dean Hendrix, Larry Barowski, and David Umphress. 2014. Dynamic Program Visualizations: An Experience Report. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 609–614.

[2] Essi Isohanni and Hannu-Matti Järvinen. 2014. Are Visualization Tools Used in Programming Education?: By Whom, How, Why, and Why Not?. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14)*. ACM, New York, NY, USA, 35–40.

[3] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. 2003. The Jeliot 2000 program animation system. *Computers and Education* 40, 1 (2003), 1–15.

[4] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. 2007. VILLE: A Language-independent Program Visualization Tool. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88 (Koli Calling '07)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 151–159.

[5] Jorma Sajaniemi. 2002. An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02) (HCC '02)*. IEEE Computer Society, Washington, DC, USA.

[6] André L. Santos and Hugo S. Sousa. 2017. An Exploratory Study of How Programming Instructors Illustrate Variables and Control Flow. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research.*