An Exploratory Study of How Programming Instructors Illustrate Variables and Control Flow

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

Hugo Sousa
e.Near
PORTUGAL
hugossousa92@gmail.com

ABSTRACT

We present an exploratory study that investigated how programming instructors illustrate program execution, namely variable manipulation and control flow. The study involved tasks where instructors were asked to doodle illustrations of program execution in a sheet of paper for given Java methods with example input/output. We found eight illustration patterns, some of which relate to specific variable roles (Gatherer, Stepper for array indexing, Most Wanted Holder). By analyzing existing pedagogical animation and debugging tools, we conclude that their visualizations do not have a broad representation of the aspects evidenced by the illustration patterns of instructors, hinting that human illustrations tend to be considerably richer.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Human-centered computing** → *Empirical studies in visualization*;

KEYWORDS

program state visualization, variable roles, pedagogical debuggers

ACM Reference Format:
André L. Santos and Hugo Sousa. 2017. An Exploratory Study of How Programming Instructors Illustrate Variables and Control Flow. In *Proceedings of 17th Koli Calling International Conference on Computing Education Research (Koli Calling 2017).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3141880.3141892

1 INTRODUCTION

A considerable amount of programming instructor time is spent helping students understand and debug code as part of their learning process. Instructors explain countless times how programming instructions manipulate variables, either in paper (or screen) for individuals and small groups, or on whiteboards (or screen projection) for a class audience. A study that analyzed the history of student attempts when solving exercises [6], revealed that among the typical misconceptions and errors, there are frequent difficulties with arrays and program state.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Koli Calling 2017, November 16–19, 2017, Koli, Finland
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5301-4/17/11…$15.00
https://doi.org/10.1145/3141880.3141892

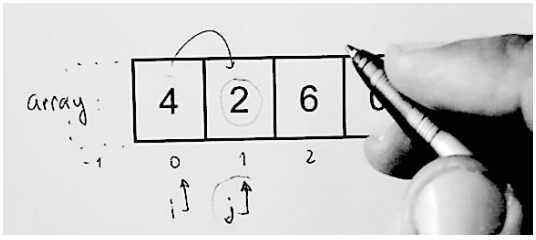


Figure 1: A programming instructor doodles the execution of the insertion sort algorithm.

The outcome of executing a piece of code is is often explained by instructors through a step by step manual simulation, where each change in the program state is illustrated as an aid for comprehension. These simulations resemble to a certain extent the process carried out by program debuggers, and that leads to the question: *why don't the students run the debugger themselves?* Conventional debuggers provided by professional IDEs (e.g., in the Java realm: Eclipse, Netbeans) are generally considered not suitable for novices, due to presentation aspects and information overload. Anecdotal evidence from teaching experience at our institution reveals that most students from the first two years of undergraduation avoid using them.

Several pedagogical debuggers and animation tools (e.g., [3, 7, 9, 10]) were proposed to mitigate the problem of understanding code. However, a recent survey [4] revealed that no more than 20% of programming courses use visualization tools. The most common reason given by instructors for not doing so is that they prefer to create their illustrations in other ways (e.g., whiteboard). Given that drawing by hand is typically more laborious than using a dedicated tool, this fact may hint that existing tools may not yet exhibit adequate properties for the intended purpose (illustrate explanations). We believe that pedagogical debuggers could be further improved by approximating their visualizations to the ones that programming instructors draw manually.

We conducted an exploratory study to investigate how programming instructors illustrate program variables and control flow when explaining the execution of Java methods. We gathered six programming instructors at our institution as participants. They were given seven tasks that consisted of doodling in a sheet of paper the execution of a given Java method, explaining its result (see Figure 1). All the explanations were recorded in video (audio included) and analyzed to mine illustration patterns. Some illustrations of program state are "natural" given the widespread notations used in teaching materials, as for instance, depicting arrays horizontally

from left to right and variables that point to array positions. However, little is systematized regarding the illustration of less obvious variable usages. Another study investigated student illustrations on exam papers [8], and revealed the frequent presence of trace annotations as an aid to their comprehension. To our knowledge, there are no previous studies that investigated instructor illustrations. Our contribution consists of a small exploratory investigation from which illustration patterns can be drawn (despite the reduced number of study participants). The fact that there are illustration patterns consists of a hint that what they are evidencing is likely to be relevant with respect to program comprehension.

Variable roles [10] were proposed as categories that describe the variable behavior in the execution of a program. As an example, an integer variable can serve clearly distinct purposes (roles), such as iterating over the indexes of an array, accumulating the result of a summation, or holding the maximum value during a search. There are reports of increased success when variable roles are explicitly adopted in introductory programming courses [2, 12]. Our study revealed that variables having specific *roles* had the tendency to be depicted using similar illustrations. We also found that variables with the same role are depicted differently when they are used in different situations, e.g. a Stepper variable [10] is illustrated in a clearly different way when used for iterating an array.

Another contribution of this paper is an analysis of existing pedagogical debugger and animation tools in the light of our study results. We conclude that several aspects that we found prevalent in the study results are not represented in existing tools. We argue that the illustration patterns of instructors can be used as guidelines for improving pedagogical debuggers, namely by embodying graphical illustrations that resemble these patterns.

2 STUDY

2.1 Participants

We contacted all the Assistant Professors that were teaching an undergraduate programming course during the last academic year at our institution. We managed to have six instructors (5 male, 1 female) volunteering for the study as participants, which are involved in different courses that relate to programming. Two instructors teach Introduction to Programming, another two instructors teach Object-Oriented Programming (second programming course at our institution), one instructor teaches Algorithms and Data Structures, and finally, one instructor teaches Concurrent and Distributed Programming (third programming course at our institution). The background of the participants in terms of their own undergraduate studies (university and degree) is fairly diverse, given that they graduated from 4 different institutions (and 5 distinct degrees).

We had one session with each participant that lasted approximately 30–45 minutes. A video (audio included) was recorded for each participant task for post-analysis. The camera was aiming at the sheets of paper where participants were doodling during their explanations (see Figure 1). Participants were asked to talk and draw freely, as if they were giving explanations to a student. Their participation consisted of explaining the execution of methods, given example arguments for their parameters and a corresponding result returned by the method.

Table 1: Study tasks.

Task	Method goal
T1	Calculate the factorial of a number (non-recursive).
T2	Calculate Fibonacci sequence number (recursive).
T3	Swap the values of two positions of an array.
T4	Check if an array contains a given number.
T5	Determine the maximum value contained in an array.
T6	Sort an array using the Insertion Sort algorithm.
T7	Sum all the elements of a two-dimensional array.

2.2 Tasks

Each participant had to accomplish seven tasks, each of which consisted of explaining the execution of a static method written in Java. Each task was given in its own sheet of paper (portrait orientation), containing the method code and an example invocation in the upper half, and a free space for doodling illustrations in the lower half. In cases when the algorithm contained an array, we provided the depiction of the array filled with the values of the example invocation. There was no time limit for each task.

The tasks were selected among the set of exercises of the introduction to programming course taught at our institution. Our goal was to have a small sequence of tasks with increasing difficulty and that the session would not last more than one hour. On the other hand, we selected methods in order to include cases covering most of variable roles [10]. This gave us some confidence that the set of examples was reasonably wide in terms of the behavior of variables, and hence, that the sample was representative of introductory programming exercises.

Table 1 presents the methods' goals that were associated to each task. All methods used integers (or integer arrays) as data types in their implementations. Tasks T1–2 did not involve arrays, whereas tasks T3–6 did, and task T7 involved a two-dimensional array. T2 was the only task where implementation was based on recursion. Task T6 was intentionally more challenging than the other tasks, and our goal with this task was to evaluate if explanation styles differ when facing a more complex method.

2.3 Illustration patterns

We analyzed the recorded videos thoroughly in order to mine patterns of how participants illustrated program variables during their explanations. Table 2 summarizes the most prominent patterns found in the illustrations, indicating how many instructors have used it on each task, and relating the patterns to variable roles when applicable.

A pattern was considered as such if there was a task where at least half of the participants (3 out of 6) drew an instance of it. We allowed some degree of variations between instances of a pattern, as long as their intention was clearly the same (for instance, writing array indexes below or above the array positions are considered equivalent). By "default", participants depicted variables with their name followed by a box holding its value. When we state that a particular variable had no special representation means that it was either not represented at all or that it was depicted using the "default" box.

Table 2: Illustration patterns found in our exploratory study: name and description, instances found in each task, example illustrations from the sessions, related variable role (if applicable).

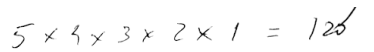
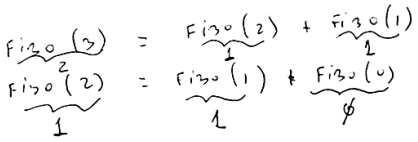
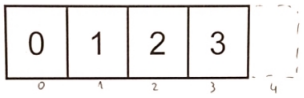
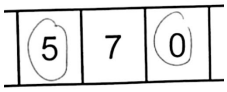
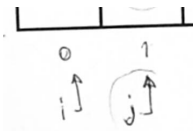
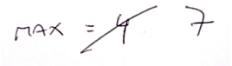
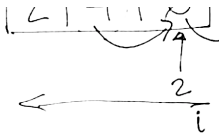
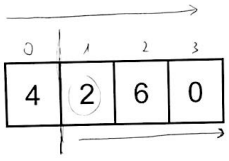
Illustration pattern	Instances	Example illustration	Related variable role
Accumulation terms trace (ATT). Writing the terms that compose an accumulation, leaving a trace of values that lead to the final result.	T1 (4)		Gatherer
Recursive call trace (RCT). Representing the tree of recursive calls, including the result of each one.	T2 (4)		
Array index values (AIV). Writing array indexes next to the array positions.	T3 (5), T4 (5), T5 (4), T6 (5), T7 (3)		
Array index parameter (AIP). Marking an array position whose index is given by a (fixed value) parameter.	T3 (6)		Fixed Value (array indexing)
Array index iterators (AII). Writing iteration variables pointing to the array positions/indexes.	T4 (5), T5 (3), T6 (6), T7 (6)		Stepper (array indexing)
Search hit history (SHH). Stroking previous stored value during a search, leaving a trace of previous search hits that were replaced by better values.	T5 (3)		Most Wanted Holder
Array index direction (AID). An arrow indicating the direction of an array index iterator (forward / backward).	T6 (3), T7 (3)		Stepper (array indexing)
Array iteration bound (AIB). A bar that divides an array according to the upper/lower bound of an array index iterator, typically in combination with the AID pattern.	T6 (3)		Stepper (array indexing)

Table 3: Representation of illustration patterns found in our study in existing pedagogical debugger and animation tools. (✓: clearly represented, ⋆: partially represented)

Tool	ATT	RCT	AIV	AIP	AII	SHH	AID	AIB	Remarks (on partial representation)
BlueJ			✓						
ViLLE		⋆							No explicit relation between stack frames (RCT).
jGRASP			✓		⋆				AII requires manual configuration.
Jeliot		⋆	✓		⋆				No explicit relation between stack frames (RCT). AII only explicit when positions are being accessed.
PlanAni			✓			✓	⋆	⋆	AID and AIB without graphically relation with the array.

2.3.1 Trace history of variable values. Several participants have written the accumulation terms next to a Gatherer variable [10] of task T1 for calculating the product. The terms were used to trace the result to the execution, as it was clear in the oral explanations.

Half of the participants depicted a Most Wanted Holder variable [10] of task T5 by striking out the previous values as they were replaced by higher ones. As with the case of the Gatherer variable (T1), the trace of the variable value was left as a means to assist the oral explanation of the process that lead to the final result.

On the recursive function (T2), several participants drew a tree-like scheme to explain the recursive calls and how the results of the base cases were propagated to the recursive ones. The returned value of the calls also served as a means to trace the terms that led to the final result.

2.3.2 Array index values and iterators. Participants had a strong tendency for writing the values of array indexes (below or above) as an aid for their explanations, given that there were several tasks where most participants have done so. Some instructors also wrote the values that go one unit beyond the array bounds (lower: -1, upper: array length), to emphasize and warn about errors pertaining to array access at invalid indexes.

In the case of the parameters to access array positions (T3), half of the participants have marked (circle-like) the positions that were being swapped. The fact that a circle was used is perhaps only due to being more quick to draw. From the oral explanations, it did not seem that there was any particular reason for using the circle instead of another figure.

A Stepper variable [10] is often used to iterate over the indexes of an array. A great majority of participants illustrated the state of this kind of variable by writing it near the associated array position (below or above, with or without an arrow). Notice that in the more complex tasks (T6, T7), on which two array index iterators were involved, *every* participant has used this illustration form.

2.3.3 Index iterator direction and bounds. When faced with the more complex cases (T6, T7, involving more variables), participants have used additional and more elaborate forms of illustrating index iterators for arrays. On T6, which was clearly the most complex due to the sorting algorithm, several participants took some time to understand themselves the given code (despite that they knew the insertion sort algorithm), and from the recordings it seemed that the illustrations were helping themselves to understand the code.

Half of the participants have used an arrow to point the "direction" of the index iterator (forward / backward) according to the values that the variable would hold as the execution progresses. This pattern was found both on T6, presumably to help coping with complexity, and on T7 where array index iterators had two distinct directions (rows and columns).

Another illustration used by half of the participants on T6 consisted of a "bar" dividing the array for marking the upper/lower bound of an array index iterator. From the oral explanations, it was clear that these bars were used to emphasize that the array iteration is being performed on a subset of the array positions, while also referring to properties of the current state of the array (more concretely, in insertion sort, meaning that a part of the array delimited by the bar was already sorted).

2.3.4 Observations. While the Stepper variables for iterating over array indexes were systematically illustrated near the arrays, the depiction of the Stepper variable of task T1 was not depicted in any particular way by participants. This hints that despite that having the same role, a variable can be depicted very differently.

Task T4 had a Temporary variable and a One Way Flag variable [10] which did not have any particular way of being depicted by participants. However, in the oral explanations of some participants it was clear the emphasis on stating that "this is a temporary variable" (former) and that "the change of value in the variable only happens once" (latter).

Our exploratory study reveals that certain aspects of program state are being evidenced by programming instructors through particular illustrations that stand out from a uniform representation of variable values. We argue that these aspects are likely to be relevant with respect to comprehension, given that otherwise, instructors would not draw them consistently.

3 ILLUSTRATIONS IN PEDAGOGICAL TOOLS

In this section we review a set of five existing pedagogical animation or debugger tools, focusing on analyzing the representation of the illustration patterns of our study in those tools. Note that these tools have other features that are not reviewed here. For each of the following tools, we downloaded the latest available version and tested the tools using the same examples of our study tasks. Table 3 summarizes our analysis, indicating which tools have representations for the different illustration patterns. We considered that a pattern is represented in a tool if the latter has a visual feature that highlights the program state information related to the pattern, even if the graphical representations do not closely resemble the drawings. When a tool has a feature that, despite being related to the illustration pattern, misses some of its essential aspects or requires additional user-provided information, we considered that the pattern is partially represented.

BlueJ [5], a widely-used programming environment, incorporates a debugger that works in the same style as a conventional one, but provides a simpler user interface that is targeted to novices. The program state information is a table of variables and their values for each element of the call stack, and hence, the difference to a conventional debugger (e.g., Eclipse) is essentially about user interface simplification. Variables are displayed uniformly regardless of their role in the algorithms under execution, while relations between variables are not made explicit. There is only one feature of BlueJ debugger that relates to one of our simplest illustration patterns, namely displaying arrays with their indexes (AIV). A controlled user study concluded that the use of BlueJ debugger had no significant impact on student learning [1].

ViLLE [9] is a debugger supporting multiple programming languages. Forward and backward execution steps are supported due to a custom built-in compiler, which supports a limited subset of the Java specification. While this custom compiler limits the scope of sample programs, it provides English explanations for each code line that are displayed while debugging a program. With respect to array visualization, even the most simple illustration pattern of AIV, represented in all other tools, is not present in ViLLE. We were not able to test recursion with our Fibonacci (program was

halting), but we assumed that the elements of the call stack would be displayed in the same way as in non-recursive calls.

jGRASP [3] is a pedagogical IDE supporting multiple programming languages that is capable of displaying complex data structures and visualize them under different forms. The user interaction with jGRASP is based on dragging objects of the stack frame into a canvas in order to visualize them. Depending on the type of object, one may decide how to visualize it by choosing from the available widgets. For instance, an array of integers may be visualized in the conventional form or as a bar chart. Visualization of variables holding array indexes is supported, but this requires manual intervention from the user (by providing a list of expressions). This detail can be considered as a disadvantage when the purpose is understanding unfamiliar code (in contrast to automatic inference).

Jeliot [7] is an animation tool focused on the demonstration of every aspect of program execution, including selection, loops, and expression evaluation. A controlled experiment revealed that an experimental group of students using Jeliot was able to provide better explanations of what would happen when running a piece of code involving multiple language constructs, when compared to the students of a control group. Despite the wide coverage in terms of programming constructs, Jeliot treats variables uniformly, i.e. the role of the variable in the program is not taken into account. For instance, an integer variable is displayed equally regardless of being used as a Stepper for iterating over the indexes of an array or as a Gatherer in a summation. There is a partial representation of AII because the relation of the variable with the array is only made explicit in the moment when an access to its positions is made.

PlanAni [11] is a program animation tool that takes into account variable roles in the animations, presenting role-based visualizations that have metaphoric graphical representations for each role, including the illustration patterns AIV and SHH (Most Wanted Holder role). Regarding the Stepper role, the related illustration patterns are only partially represented in PlanAni, given that the latter does not explicitly depict the relationship between a Stepper variable and the array that is the target of the iteration. This tool is the one that has the most representations of the illustration patterns. However, there is a major disadvantage, the role-based representation has to be manually written in a script (typically by programming instructors). This characteristic makes PlanAni not suitable for helping students to understand arbitrary code, such as debugging their own code.

4 CONCLUSIONS

Our exploratory study collected several illustration patterns that programming instructors draw when explaining variables and control flow, evidencing that different types of variables (roles) are illustrated differently. The fact that instructors are highlighting certain aspects of program behavior through specific illustrations hints that the latter are likely to be relevant for comprehension purposes. From our analysis, we conclude that the illustration patterns are not widely represented in the existing pedagogical tools. Further, most tools do not provide alternative illustration elements as an aid to understand program state.

Given the above reasons, existing pedagogical debugger and animation tools are clearly less rich in terms of illustrating program state than the manual illustrations. This fact might imply that, for the purpose of explaining program state and algorithms, pedagogical tools are still considerably far from being able to assume the role of programming instructors (even excluding the obvious differences related to the oral explanations). In the future, we believe that the effectiveness of pedagogical tools in terms of helping students to understand code/algorithms could be improved if the illustrations of program state would consider our (and possibly other) illustration patterns. The reason of our belief is that such a characteristic would approximate tool illustrations to the ones of humans, and in turn, that could help students to be more autonomous when trying to understanding code (using an enhanced debugger).

ACKOWLEDGEMENTS

We would like to thank the programming instructors of our institution that accepted to participate in our study, and the anonymous reviewers for their valuable comments on earlier drafts of this paper.

REFERENCES

[1] Jens Bennedsen and Carsten Schulte. 2010. BlueJ Visual Debugger for Learning the Execution of Object-Oriented Programs? *ACM Transactions on Computing Education* 10, 2, Article 8 (June 2010), 22 pages. DOI: http://dx.doi.org/10.1145/1789934.1789938

[2] Pauli Byckling and Jorma Sajaniemi. 2007. A Study on Applying Roles of Variables in Introductory Programming. In *2007 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007), 23-27 September 2007, Coeur d'Alene, Idaho, USA*. 61–68. DOI: http://dx.doi.org/10.1109/VLHCC.2007.31

[3] 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. DOI: http://dx.doi.org/10.1145/2538862.2538958

[4] 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. DOI: http://dx.doi.org/10.1145/2674683.2674688

[5] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology* 13, 4 (December 2003), 249–268. http://www.cs.kent.ac.uk/pubs/2003/2190

[6] Einari Kurvinen, Niko Hellgren, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. 2016. Programming Misconceptions in an Introductory Level Programming Course Exam. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. ACM, New York, NY, USA, 308–313. DOI: http://dx.doi.org/10.1145/2899415.2899447

[7] 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.

[8] Robert McCartney, Jan Erik Moström, Kate Sanders, and Otto Seppälä. 2005. Take note: the effectiveness of novice programmers annotations on examinations. *Informatics in Education* 4, 1 (2005), 69–86. http://urn.fi/URN:NBN:fi:aalto-201609294520

[9] 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. http://dl.acm.org/citation.cfm?id=2449323.2449340

[10] 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, 37–. http://dl.acm.org/citation.cfm?id=795687.797809

[11] Jorma Sajaniemi and Marja Kuittinen. 2003. Program animation based on the roles of variables. *SoftVis '03 Proceedings of the 2003 ACM symposium on Software visualization* (2003), 7. DOI: http://dx.doi.org/10.1145/774833.774835

[12] Juha Sorva, Ville Karavirta, and Ari Korhonen. 2007. Roles of Variables in Teaching. *Journal of Information Technology Education* 6 (2007), 407–423. http://search.ebscohost.com/login.aspx?direct=true