Enhancing Visualizations in Pedagogical Debuggers by Leveraging on Code Analysis

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

ABSTRACT

PandionJ is a pedagogical debugger that provides users with rich visualizations of program state that resemble teacher-drawn illustrations. Its design was driven by a study that investigated how programming teachers illustrate variables, transposing illustration patterns into the tool. These illustrations require static analysis of the source code to infer relationships between variables. We describe the innovative features of the tool regarding how it addresses program variables. The tool was adopted in an introductory programming course, and the pass rates of the first course edition using it were significantly higher when compared to the three previous editions.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Applied computing** → *Interactive learning environments*;

KEYWORDS

program visualization, pedagogical debuggers, CS1

ACM Reference Format:
André L. Santos. 2018. Enhancing Visualizations in Pedagogical Debuggers by Leveraging on Code Analysis. In *18th Koli Calling International Conference on Computing Education Research (Koli Calling '18), November 22–25, 2018, Koli, Finland.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3279720.3279732

1 INTRODUCTION

Studies have reported that students face difficulties concerning execution tracing and understanding program state [7, 18]. Whenever a program is not doing what is expected, one has to perform some sort of debugging activity in order to find the cause and eventually fix it. Debugging is an essential part of the programming activity, not only at a learning stage. However, from a novice perspective, debugging will likely to be more challenging because in addition to the lack of algorithmic skills, a student is also in the process of learning programming constructs. Therefore, the apprentice might struggle with these two difficulties at once.

Koli Calling '18, November 22–25, 2018, Koli, Finland

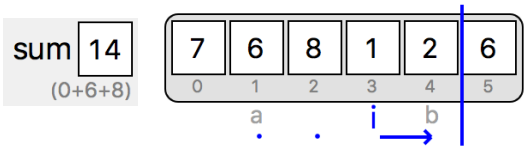
ACM ISBN 978-1-4503-6535-2/18/11…$15.00
https://doi.org/10.1145/3279720.3279732

We believe that a debugger designed for pedagogical purposes can provide users with more information regarding program state (i.e. its variables) than what is generally offered by conventional debuggers. In this way, a debugger not only informs the user of *what* are the values of the variables (what conventional debuggers do), but also, to some extent, *why* do they had, have, and will have, certain values and behavior (our aim). If a debugger provides a sufficiently rich illustration of program state, it is assuming part of the task of a programming teacher, when he or she is drawing illustrations of program state to help students understanding program execution.

We developed PandionJ [1] [17], a pedagogical debugger for Java that provides users with a representation of program state that not only displays variable values, as in a conventional debugger, but it also uses different representations according to the role of each variable and its relationship with other variables. The goal is to provide information that aids on understanding the current and future state of the variables, to the extent of what is possible using static and dynamic analysis of the program under execution. The design of PandionJ was driven by the results of our previous study [16] that investigated how programming teachers illustrate program state (by hand). The results of this study helped us to prioritize which aspects of program state to address and how to illustrate them. The most common hand-drawn illustration patterns of variable behaviour were transposed to our debugger, so that they are provided automatically without the aid of an instructor.

Figure 1 presents an example illustration provided by PandionJ for a function that sums a range of values of an array. The function

[1] Name inspired by *Pandion haliaetus* (known as osprey or fish eagle/hawk), a bird of prey whose vision is specialized for detecting under water objects as an assistance for hunting. Freely available at: http://pandionj.iscte-iul.pt




```
static int sumRange(int[] v, int a, int b) {
   int sum = 0;
   for(int i = a; i <= b; i++)
      sum = sum + v[i];
   return sum;
}
```

Figure 1: PandionJ illustration of code: function to sum the values contained in a range of array indexes.

receives a reference to an array (v), and a range of indexes (a, b) for which a sum is computed. The array is iterated partially (from index a to b) and the values at those indexes are summed up in a variable (sum), which will hold the function result. There are four integer variables in the function, with three different *variable roles* [11]. Variables a and b are Fixed Values that refer to array positions. Variable sum is a Gatherer, whose value is computed through incremental summation. Variable i is a Stepper for iterating the target range of array indexes ([a, b]).

As opposed to a conventional debugger, which would render the values of these variables uniformly (a value for each variable, usually grouped in a table), PandionJ illustrates them distinctly. This is achieved by statically analyzing the code under execution to infer roles and relationships between variables, and it does not require any user input or previous preparation. The way to depict the different types of variables mimics the drawing patterns that were recurrently observed in our previous study [16].

The main novelty of PandionJ is the capability of automatically provide rich illustrations of program state that resemble teacher-drawn ones. In contrast to the state of the art, these illustrations establish relationships between variables (e.g., in Figure 1 variables a, b, and i are associated to the array (v)) and foresee future state (e.g., the next values that variable i will hold are depicted).

PandionJ was adopted at the Introduction to Programming course of our institution. In the first experimental edition, the pass rates of first-time students have increased over previous course editions (improvements ranging from 18% to 27%). Given that the course was stable regarding contents, exercises, and form of evaluation, we are confident that the improvement of success is due, at least partially, to the effect of adopting PandionJ.

In this paper we describe the main features of PandionJ, with a focus on its innovative aspects. Further, we detail the context in which the tool was adopted and the results of using it during one course edition.

2 RELATED WORK

A debugger, which nowadays is typically integrated in a programming environment, allows users to execute a program step-by-step, while displaying the current program state (values of variables) as the execution progresses. A debugger can be seen as a visualizer of program execution, despite that the way information is displayed is not typically designed for that purpose.

The debuggers that are part of professional IDEs (e.g., Eclipse[2], Netbeans[3]) can be considered too overwhelming for a novice. Pedagogical programming environments have provided simpler debuggers, as for instance in BlueJ [6]. While this tool provides a simpler and more friendly debugging environment for novices, the information that is being provided to the user is not considerably different than the one offered by a conventional debugger: the call stack, a list of variables therein, and their current values. Variables are displayed uniformly regardless of their role in the algorithms under execution.

ViLLE [9] is also a pedagogical debugger, supporting multiple languages (including Java). In addition to the conventional features of a debugger, it embodies features that help on understanding the code, through a textual explanation that is automatically generated for each code line. It is also suitable to integrate pop-up questions linked to the program execution so that they appear at predefined locations. The tool supports backward steps, i.e. moving back to the previously executed instructions, an usability improvement over many of existing debuggers that allows users to quickly replay a part of the program.

Jeliot [8] is a program animation system that although not designed as a debugger, it allows users to move forward in the animation step-by-step. Therefore, for the purpose of understanding execution, we consider that Jeliot may have the same role as a debugger. However, this tool goes deeper in illustrating all the details, namely with respect to computation of expressions, which most debuggers do not. A controlled experiment revealed that using Jeliot improves the learning process of mediocre students.

DrJava [10] is a pedagogical programming environment for Java that has its own debugger and an interaction console, on which users can type expressions and check the result immediately. The interaction console can be used while the program is suspended in debug mode.

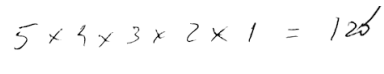
jGRASP [4] is a pedagogical programming environment for Java (and other languages) also with its own debugger. The main novelty consists in allowing users to drag variables into a canvas where arrays and objects are displayed graphically, offering alternative visualization widgets for rendering them (e.g., an array may be visualized in the default way or as a bar chart). Iteration variables (integers) may be associated to the indexes of an array widget, but this has to be manually performed by users.

Our approach to pedagogical debuggers has a major difference to the previously mentioned tools (BlueJ, ViLLE, Jeliot, DrJava, jGRASP), consisting of the capability of inferring variable behavior and relationships automatically from the source code. We rely on code analysis to collect information, in order to render more elaborate illustrations that differentiate variables with different roles, establish relationships between variables, and foresee future variable state.

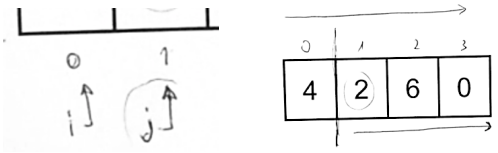
PlanAni [12] is a program animation environment that was designed to teach *variable roles* [11]. This tool embodies different iconic representations for each variable role. As opposed to the previous tools, PlanAni cannot be considered a debugger, because the animations are based on predefined scripts (developed by teachers), and hence, it is not suitable for students to debug their own code. The assignment of variable roles in the original version of the tool was not automatic, it had to be defined as part of the scripts. Later on, additional research was carried out to propose automatic forms of classifying variables according to their role [2, 5].

The approach of PlanAni is the one that relates more to ours, in terms of having different visual representations for variables with distinct roles. However, to our knowledge, there was no rationale for the iconic representations used in PlanAni for the different roles. In our approach, the representation of roles is based on an empirical investigation of how programming teachers illustrate them. These teacher-drawn illustrations in some cases are clearly different for a same role, depending on the context and relationship with other variables.

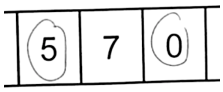
[2] http://eclipse.org
[3] http://netbeans.org



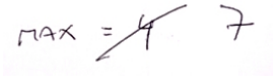
(a) Gatherer



(b) Stepper (array indexing)



(c) Fixed Value (array indexing)



(d) Most-Wanted Holder

Figure 2: Illustration of variables with different roles. Examples of teacher illustrations taken from [16].

3 ILLUSTRATION OF VARIABLE ROLES

Variable roles [11] are categories that describe variable behavior in the execution of a program. A role implies that there is a pattern in the variable behavior, i.e. how its values are going to be assigned. Variable roles may contribute to success when adopted in teaching introductory programming courses [1, 19].

Our previous study [16] found that there are drawing patterns among programming teachers that relate to some of the variable roles, in the sense that variables with the same role tend to be illustrated in similar ways. Figure 2 presents a summary of illustration patterns found in the study with example illustrations collected during the study. Notice how the purpose of the variable (role) influences how it is depicted by teachers, leading to distinct representations. The study was of exploratory, and it only identified patterns related to the roles Gatherer, Stepper (array indexing), Fixed Value (array indexing), and Most-Wanted Holder. In PandionJ, we only address these roles, while further investigation could lead to illustration patterns of other roles and relationships, which could be the basis for future enhancements of the tool.

Gatherer variables hold a value that is calculated using a series of values (e.g., `sum` variable of Figure 1). These tend to be illustrated showing all the terms in the calculation of the current value. In a sense, this is a trace that explains the current variable value (see Figure 2a).

Stepper variables hold a value that is incremented or decremented by a known value (e.g., `i` variable of Figure 1). The study reports that a Stepper, if used for array indexing, will most likely be represented next to the array. Further, the illustration may include arrows that inform about the direction of the variables (backward or forward), and bars that delimit the the growth or decrease of the variable (see Figure 2b).

Fixed Value variables hold a value that does not change after the first assignment (also often referred to as *constants*). The study reports that in the particular case of Fixed Values for array indexing, teachers tend to mark those next or at the array positions (see Figure 2c).

Most-Wanted Holder variables hold a value that is the current best match during a search (e.g., a variable holding the highest value during the iteration of an array). Given that every new value that is assigned to the variable superseded the previous best match, the illustrations tend to include a trace of previous variable values (see Figure 2d).

4 USER CODE ANALYSIS

One of the goals of PandionJ is to mimic the teacher-drawn illustrations (such as those in Figure 2). We do not aim at being exhaustive in terms of addressing all the variable roles, but rather to focus on those that are relevant for the visualizations. On the other hand, a simple binary classification of variable roles is not enough for our purposes. A deeper analysis is necessary to infer relationships between variables, so that we can relate them visually. For instance, considering the example of Figure 1, the tool has to be aware that the variable `i` is a stepper used for indexing array referenced by variable `v`, and that variables `a` and `b` are fixed values that refer to positions of the array and represent the interval [minimum, maximum] of variable `i`. The technical details of the static code analysis that performs this are out of the scope of this paper. In this section we describe which information we are capable of extracting from the code and the criteria for assigning the variable roles.

Figure 3 presents a conceptual model (as a class diagram) of the information collected by our code analysis for each Java method. All local variables of the method are analyzed, including parameters. We consider four types of *Value Holders* (having a Java primitive

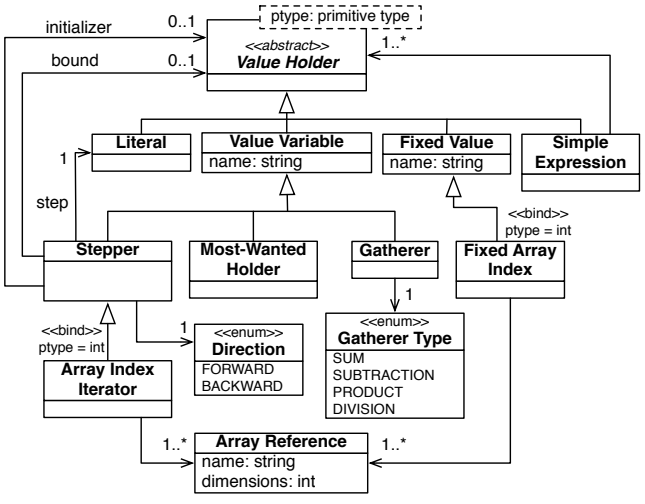


Figure 3: Variable roles and relationships collected through static analysis.

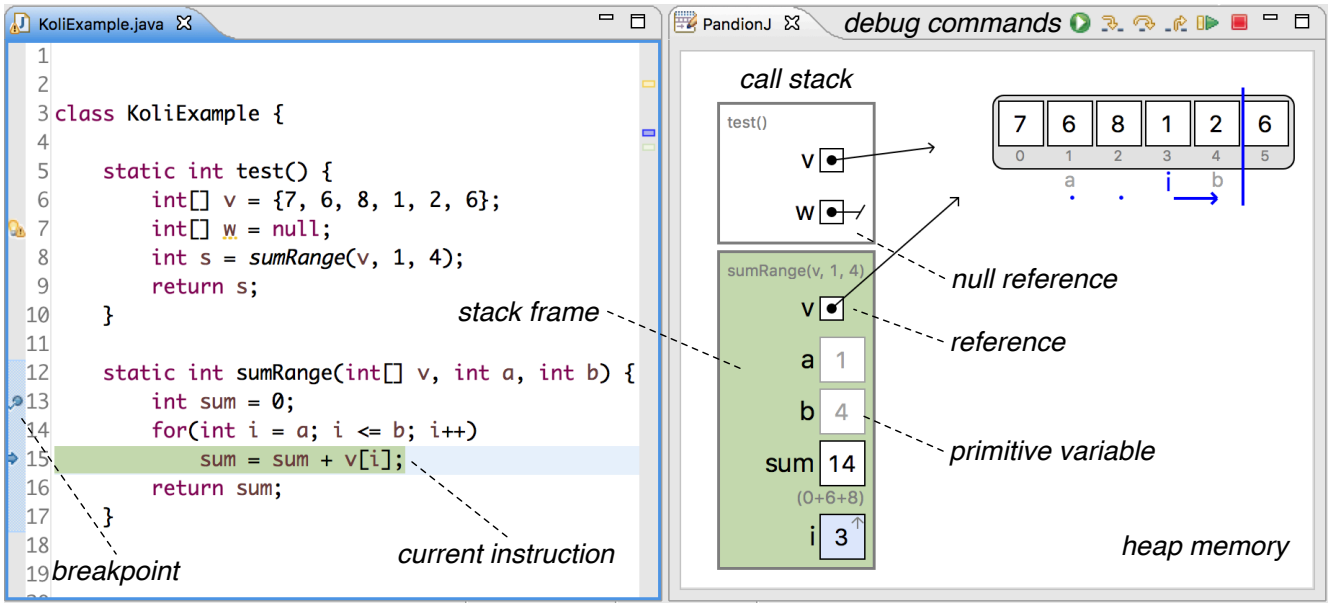


Figure 4: PandionJ debugging environment (Eclipse plugin); Gatherer, Stepper, Fixed Values (array indexing)

type): *Literals* (values hard-coded in the source), *Fixed Values* (constants), *Value Variables*, and *Simple expressions* (i.e. that not include method calls). Additionally we consider *Array References* for single or multi-dimensional Java arrays to which other variables are related. A variable is classified as a Fixed Value when one of the following conditions is met: (a) it is declared as `final`[4], (b) it is effectively a constant, i.e. even if not declared `final` it is assigned only once (when declared), or (c) it is a method parameter that is not assigned (a constant in the context of the method execution).

For each *Value Variable*, if a role can be determined when analyzing the code, the variable assumes one of the subtypes: *Stepper*, *Most-wanted holder*, or *Gatherer*. Otherwise, the variable is considered unclassified.

A *Stepper* is a variable that holds values that are determined sequentially, through fixed increments or decrements. A variable is classified as a Stepper when all its assignments except initialization consist of either exclusively adding or subtracting a same constant value to the current one. In addition to the classification, our analysis infers the *Direction* of the Stepper (*Forward* or *Backward*) and the *step* value. We support different syntactic possibilities for incrementation/decrementation (e.g., `i++`, `i += 1`, `i = i + 1`). If the Stepper *initializer* is a Value Holder, we collect that information. If the Stepper is being modified inside a loop, we also collect a *bound* Value Holder that constraints the growth or decrease of the Stepper based on the loop condition.

An *Array Index Iterator* is a special case of Stepper that consists of an integer serving the purpose of iterating the indexes of an array. While a Stepper variable might not have any relation to other variables, an Array Index Iterator is associated with one or more array references.

[4]A Java keyword that enforces that a variable can only be assigned once and exactly once.

A *Fixed Array Index* is a special case of Fixed Value that consists of an integer that is used as an array index. If the initializers or bounds of Array Index Iterators are Fixed Values, the latter are considered Fixed Array Indexes.

A variable is classified as a Gatherer when the following conditions are met: (a) all its assignments include the variable as part of the assigned expression; (b) all assignments are of the same accumulation type (*Sum*, *Subtraction*, *Product*, or *Division*). We support different syntactic possibilities for the accumulation assignments (e.g., for summation: `x = x + n`, `x = n + x`, `x += n`), and collect the gatherer type.

A variable is classified as Most Wanted Holder when the following conditions are met: (a) all the assigned expressions neither include the variable nor consist of a constant value; (b) all assignments except the initialization are guarded by a conditional `if` statement; and (c) the variable is used in the condition of that same `if` statement.

5 PANDIONJ

PandionJ was developed as an extension of Eclipse, and uses its debugger engine to run programs. However, we provide an alternative user interface to render the program state. Our interface combines the information of variable values provided by the debugger engine with the information of variable roles obtained through our static analysis (explained in the previous section).

Figure 4 presents a screenshot of PandionJ. The left hand part shows the code editor of Eclipse, whereas the right hand part shows our view. Each frame of the active call stack is displayed as a container holding its variables. Primitive types are represented as boxes with a value, whereas references as pointers to their targets. Null references appear as "disconnected" (e.g., variable `w`). By hovering a variable one can see a tooltip with the role information (if any).

Allocated objects (Java heap memory) are represented in the right hand part (e.g., the array object in the example).

In terms of user commands, PandionJ works exactly as the debugger of Eclipse, which provides the usual actions of *step in*, *step over*, and *step out* (return), *resume* and *stop* execution. Programs suspend at breakpoints set by the user, or when an exception occurs. The latter case is useful, because a user may execute the program without breakpoints and still observe the final state of the program at the moment when the exception is thrown.

5.1 Arrays

Arrays are rendered as containers with a box for each of its positions, each having its index number. Primitive arrays hold primitive variables, whereas reference arrays hold references. Over an array figure, numerous annotations may appear:

- **Stepper** variables that are array index iterators over the array, displayed at the position that matches the current value (e.g., Figure 4: variable `i`).
- **Fixed values** that refer to array positions (e.g., Figure 4: variables a and b)
- **Vertical bars** that indicate the lower/upper bound of a Stepper (e.g., Figure 4: the upper bound of variable `i`; Figure 5: the lower bound of variable `i`).
- **Horizontal arrows** that indicate the direction and destiny of a Stepper (e.g., Figure 4: forward direction of variable `i` towards 4 (b); Figure 5: backward direction of `i` towards -1).
- **Trailing dots** marking the previous values of a Stepper (e.g., Figure 4: variable `i` was initially 1, then became 2, and currently is 3).
- **Out-of-bounds position** marks that appear when a Stepper has a value that is outside the valid indexes of the array. The mark is red when an `ArrayIndexOutOfBoundException` occurs, indicating the invalid access (e.g., Figure 6).

Matrices in Java may be represented in terms of multi-dimensional arrays. PandionJ supports two-dimensional arrays by depicting the first dimension vertically, containing references to the second dimension elements (see Figure 7). These two dimensions of arrays are annotated as described previously, and we support Steppers that work over different dimensions (e.g., Figure 7: variable `i` for the first dimension and variable `j` for the second).

5.2 Variable Roles

Fixed Values, when displayed in the stack frame container, have a slightly different grayed out appearance, denoting that their value will not change (e.g., Figure 4: variables a and b). Further, these values may appear next to array positions, as explained previously.

Steppers, when displayed in the stack frame container, have an upwards or downwards arrow annotation to denote their direction (e.g., Figure 4: variable `i` forward; Figure 5: variable `i` backwards). Steppers may also be represented over the arrays, as explained previously. The direction arrows point to the last value that the

```
int[] v = {6, 8, 7, 4, 5};
int max = v[v.length-1];
int i = v.length-2;
while(i > -1) {
   if(v[i] > max)
      max = v[i];
   i--;
}
```

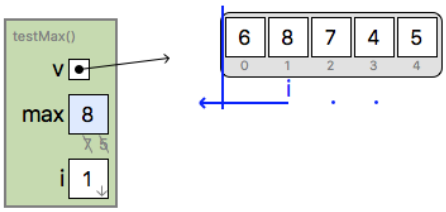


Figure 5: PandionJ illustration: Most-Wanted Holder, Stepper (array indexing, backward iteration)

```
int[] v = {1, 2, 3};
int sum = 0;
for(int i = 0; i <= v.length; i++)
   sum = sum + v[i];
```

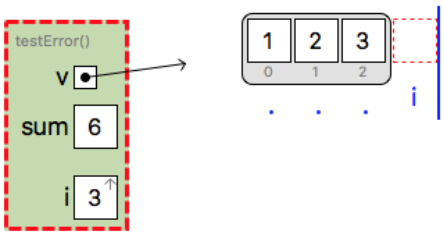


Figure 6: PandionJ illustration: Array out of bounds exception, Stepper (array indexing)

```
int[][] m = new int[3][4];
for(int i = 0; i < m.length; i++)
   for(int j = 0; j < m[i].length; j++)
      m[i][j] = 1;
```

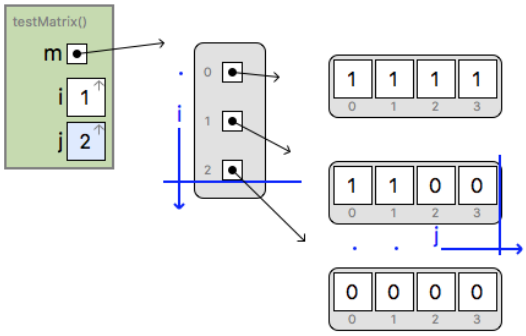


Figure 7: PandionJ illustration: Matrix, Stepper (two-dimensional array indexing)

Stepper is expected to have, according to the condition of the loop (if applicable). The vertical bar position is determined by the expression against which the Stepper is tested in the condition of the loop (if applicable). The arrow tip may be before or after the vertical bar, indicating that the last iteration matches the boundary value (e.g., Figure 4: `i <= b`) or that it matches the last value before the boundary (e.g., Figure 5: `i > -1`).

Gatherers are displayed with an annotation containing the accumulation terms that decompose the current value (e.g., Figure 4: variable `sum`). As the variable value is changed, the annotation is updated. The different arithmetic operations are supported.

Most-Wanted Holders are displayed with an annotation containing the history of its previous values (e.g., Figure 5: variable `max`). As a new assignment is made, the history annotation is updated.

5.3 Classes and Objects

The representation of objects in PandionJ borrows several ideas from our previous work on the AguiaJ tool [13, 15], namely with respect to distinct representation of static and instance fields, encapsulation or attributes, access modifiers, and interaction with the objects.

Objects are represented in containers that hold their instance fields (see Figure 8, a `Vector` object with its instance fields `elements` and `next`). Static fields of classes (e.g., `CAP`) are represented outside the objects, to emphasize that those are global and unique. The instance fields of an object are represented in the same way as explained in the previous sections.

When the active frame of the call stack is within the context of a certain object, it is displayed inside that object. This hints that not only the frame variables are accessible, but also the object attributes. The example in the figure has the current stack frame at method `Vector.add(...)`, after being called from `main()`.

Users may interact with the objects while the program is suspended in debug mode by invoking its public operations (available through the buttons). Private and public class members have a different treatment. Private methods are not available for invocation, while private fields only appear when the active stack frame reaches an object or the user explicitly "opens" the object for inspection. Public fields are always visible in an object. Public methods are available for invocation when the active stack frame is not at the object (otherwise method executions within a same object could be interleaved).

5.4 Widget Extensions

In order to allow programming exercises to be easily adapted to a *context* [3], PandionJ may be extended via plugins with widget extensions that provide alternative forms of visualization for objects and arrays. The possibility of developing such extensions relates to our previous work on the AguiaJ tool [14], which also supported this kind of extensibility. The extensions may address, for instance, the visual representation of images, charts, boards (for games), or a particular notation of a specific domain. Figure 9 presents examples of extensions. As steps are performed in the debugger, the widgets are updated accordingly. There are three types of extensions, each one addressing its own type of data: objects, primitive values, and arrays.

```
public class Vector {
   public static final int CAP = 3;

   private int next = 0;
   private int[] elements = new int[CAP];

   public void add(int e) {
      elements[next] = e;
      next++;
   }

   static void main() {
      Vector v = new Vector();
      v.add(7);
   }
}
```

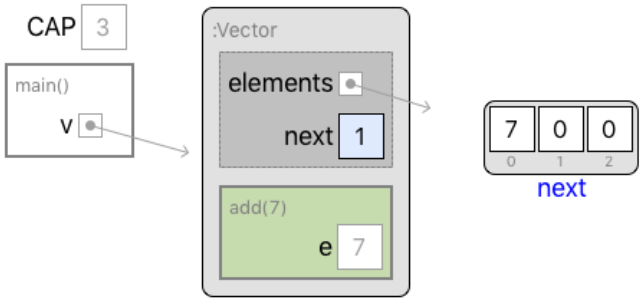


Figure 8: PandionJ illustration: Objects, encapsulation, and instance call stack frame.

```
Color red = new Color(255, 0, 0);
byte n = 32 >> 2; // @binary
char[] str = {'h','e','l','l','o'}; // @string
int[][] id = {{1, 0}, {0, 1}}; // @matrix
int[][] img = new int[50][128]; // @grayscaleimage
for(int y = 0; y < img.length; y++)
   for(int x = 0; x < img[y].length; x++)
      img[y][x] = x*2;
```

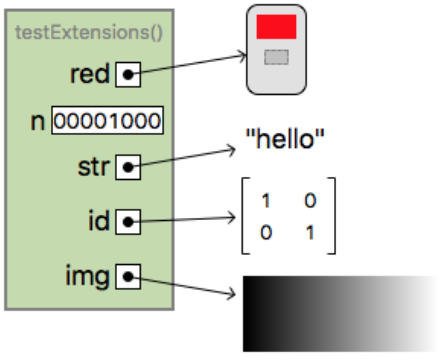


Figure 9: Extension examples for PandionJ.

For objects, an extension associates an object type with a widget, so that when objects of that type are displayed, the extension widget is rendered with it. In Figure 9 we can see the object referenced by variable `red` (of type `Color`, from Java libraries) being illustrated with a small rectangle filled with the color.

For primitive values and arrays, an extension defines a *tag* that will be used to annotate the variable declaration that the user wants to render using the extension. The extension widgets are displayed if the user annotates a variable declaration with a valid tag, and if the value or referenced array have a valid state at runtime. For instance, an extension for rendering algebra matrices (see Figure 9, `@matrix`) requires that on the second dimension there are no null arrays and that all lines have the same length.

Figure 9 presents one example of a primitive value widget extension where the variable `n` is being rendered in binary representation. The remaining three examples are array widget extensions. The array of characters is being rendered as a string (variable `str`), the first two-dimensional integer array is being rendered as an algebra matrix (variable `id`), whereas the second (variable `img`) is being rendered as a grayscale image.

6 EVALUATION

PandionJ was adopted at the Introduction to Programming course of our institution. The first time we used the tool was in the last course edition, which here we refer to as the *experimental* edition. As a form of evaluating the effectiveness of the tool, we compared the success in terms of pass rate of this experimental edition with the past three editions, which we refer to as the *control* editions. Our independent variable was the use of PandionJ, while the dependent variable was the course pass rate. In this section we explain the context in which we used the tool and we describe the obtained results.

6.1 Context

The Introduction to Programming course duration is 12 weeks. On each week there is a lecture of 1,5 hour and a lab class of 3 hours for practicing the topics of that week. The course contents are summarized in Table 1. Both the contents and exercises have been the same for the past four course editions, with the exception of minor corrections in the material. The AguiaJ tool [13] was being used as courseware in the control editions, both in lectures and lab classes. AguiaJ offers the possibility of fast experimentation with functions and objects, but lacks the integration with a debugger.

During the experimental edition, PandionJ was used both in lectures and lab classes. Lecturers use the tool to demonstrate programming constructs and algorithms, whereas in the lab classes students use the tool to solve exercises. Students had contact with PandionJ from the second week on.

In order write the final exam, students have to qualify for it by developing a small project individually. PandionJ was also used to develop the project. The number of students that deliver the project and fail the qualify is low (less than 10% in each edition). When a student does not deliver the project is in most cases a drop-out.

The final exam mark determines if a student passes the course. There are two exam dates, if a student fails on the first call, he or she may attend a second call that takes place about 2 weeks after. These evaluation rules were the same for the past four course editions.

Table 1: Introduction to Programming course contents.

Week	Topics
1	Data types, expressions, functions
2	Selection statements, loops
3	Function calls, recursion
4	Arrays
5	References, procedures
6	Matrices
7	Objects
8	Image manipulation
9	Class design, exceptions
10	Encapsulation
11	Interfaces
12	Revision

The fact that all the course content and evaluation rules were stable during the past four editions, made it suitable to evaluate the effect of using PandionJ, since the tool being used was the only aspect that changed in the experimental edition.

6.2 Results

We compare the results of first-time students, since they represent a more homogeneous cohort with less uncontrollable factors. Students with more than one enrolment, when compared to first-time students, are more likely to decide not to attend classes (they study on their own). Since we wanted to measure the effect of using the tool, we reduce the likelihood of taking into account students that did not actively use the tool by considering only first-time students.

Table 2 presents the pass rates of first-time students in the Introduction to Programming course over the last four editions.[5] Each table row line represents a course edition. PandionJ was being used as courseware in the experimental edition of 2017/2018. The second column contains the number of first-time students that enrolled in each edition. The third column contains the number of first-time students that made it to the final exam (the higher, the less drop-outs). The fourth column contains the number of first-time students that have passed the course, and the percentages are relative to the total of enrolments and to the number of students that has attended and passed the exam. The fifth column contains the pass rate improvement of the experimental edition when compared to each control edition ($\frac{\%exp - \%ctrl}{\%ctrl}$). The sixth column contains the p-values of two-tailed z-tests that compare the proportions of pass rates between each control edition and the experimental edition.

The experimental edition registered a pass rate of 66%, an average improvement of 22% when compared with the control editions (52%, 56%, 55%). The difference in pass rates between the experimental and each one of the control editions is statistically significant ($\alpha = 0.05$). When applying the Bonferroni correction to the three individual confidence intervals we obtain 0.98333 ($1 - \frac{0.05}{3}$). This

[5]The global pass rates also revealed significant improvements in the experimental edition. 2014/2015: 170/374 (47%); 2015/2016: 174/394 (44%); 2016/2017: 175/397 (44%); 2017/2018: 218/411 (53%)

Table 2: Pass rates of first-time students in the Introduction to Programming course over the last four editions.

Course Edition	Students Enrolled	Exam Attendance (% total)	Pass Rate (% total, % exam)	Exp. Edition Improvement	Significance (p-value)
2014/2015 (ctrl.)	257	207 (81%)	134 (52%, 65%)	27%	0.00076
2015/2016 (ctrl.)	281	189 (67%)	158 (56%, 84%)	18%	0.01352
2016/2017 (ctrl.)	275	204 (74%)	151 (55%, 74%)	21%	0.00560
2017/2018 (exp.)	288	231 (80%)	191 (66%, 83%)	-	-

indicates that our results are statistically significant with an overall confidence level of 0.95 ($\alpha = 0.05$), given that every p-value is lower than 0.01667 (1 − 0.98333).

Course success can be decomposed into two phases, the number of students that reached the exam, and among these, the number that have passed the exam. With respect to reaching the exam, we have had considerably less drop outs in the experimental edition when compared to two of the control editions and slightly more drop outs than in one of the control editions. With respect to passing the exam, the experimental edition had a higher pass rate than two of the control editions, and a slightly lower rate than one of the control editions In summary, the control editions when compared to the experimental edition had either significantly more drop outs (which are failures) or significantly less success in the exam.

Note that a situation of "almost drop-out" is likely to be a student with a higher chance of failing the exam (fragile programming skills). A drop-out is in many cases an "anticipated failure", where the student felt that it was not worth to take the effort to the end. In the control edition of 2014/2015 the drop-out rate was in the order of the experimental edition (81% vs. 80%), but the exam success rate difference was blatant (65% vs. 83%) — in other words, there was no significant difference in drop-outs, but the students of the control edition had apparently weaker preparation. Although we discriminate here the two phases for a deeper insight, we believe that the final result (pass rate) is the most relevant indicator to consider because of its robustness to these variations in drop-outs that difficult to reason about.

6.3 Threats to Validity and Discussion

There were several factors that we could not control as in a fully controlled experience. Conducting a whole-course learning experience as a fully controlled experience would face several challenges. Namely, administrative, such as the inability to randomly distribute students in control and experimental groups, because that distribution is not under control of the course administration. On the other hand, we could face legal issues, or at least complaints, by having students being treated differently within the same course edition, while being evaluated in the same way.

Despite that our evaluation was not a fully controlled experience, we believe that the similarity of the four course editions allows us to draw conclusions with reasonable confidence. Namely, the course editions were similar in terms of number of students, and equal in terms of course contents and evaluation rules.

The main aspect that we could not control was the potentially significant difference in terms of skills among the groups of students that have enrolled in each course edition. We address this threat by considering only first-time students (reducing variability), and by comparing the results with several course editions held previously.

There is an issue in our experience that can be considered as confounding variable, namely, to the activity of using a debugger systematically. When using PandionJ, students are actually running a debugger, executing their code step-by-step. Therefore, in the last course edition, students have observed more step-by-step executions than in the previous course editions (we cannot quantify, but from on-the-ground experience we can confidently state this). This new activity itself may have had an influence in the results. Hypothetically, the systematic use of a debugger by an apprentice could be beneficial to the learning process, and this could have been one of the factors that led to improvements in the pass rates, independently of PandionJ. It was not feasible to control this confounding variable in our process of evaluating the tool, because the tool is a debugger itself.

In order to have a control condition equivalent to the experimental condition with respect to the systematic use of a debugger, we would have to run a whole course edition where the debugger is being used (both in lectures and lab classes). However, we believe that this endeavour would face a considerable feasibility challenge. Our on-the-ground experience with promoting the use of a conventional debugger in introductory programming always faces a clear resistance from students. We argue that existing Java debuggers (e.g., what is available in Eclipse) are not fully suited for teaching settings, as there are usability issues that hinder a seamless and effective adoption.

7 LIMITATIONS AND FUTURE WORK

As with a conventional debugger, the program is suspended at the breakpoints. Although the current variable values are accessible, the history of their previous values is not. Therefore, PandionJ is not capable of rendering the illustrations taking that information into account, and hence, illustration aspects that are based on variable history (e.g., Gatherers, Most-Wanted Holders) do not appear as soon as the execution is suspended. They only appear when the user is executing the program step-by-step. Overcoming this limitation would require that the history of variable assignments is being collected while the program is running, independently of hitting breakpoints. This feature would be useful, because at any arbitrary suspension of the program one could have an explanation of the current values of the variables. While we do not consider it a priority, supporting this feature is a possible enhancement in the future.

Another limitation relates to the illustrations of array indexing. So far, we came across a few cases where certain variables that are conceptually related to a given array cannot be associated by

```
static int search(int v[], int n, int l, int r) {
   if (r >= l) {
      int m = l + (r - l) / 2;
      if (v[m] > n)  return search(v, n, l, m - 1);
      if (v[m] < n)  return search(v, n, m + 1, r);
      return m;
   }
   return -1;
}
```

Figure 10: Code analysis limitation example. Binary search function: `l` and `r` refer to positions of the array `v`, but that relationship cannot be (trivially) extracted from code.

means of the code analysis. As an example, Figure 10 presents a recursive function for performing binary search over an array. The parameter `n` is the value that is being searched on array `v`. The parameters `l` and `r` refer to the array range in which the search is going to be performed in the function call. Variable `m` is the position of the array at the middle of [`l`, `r`] that determines to which half the search will proceed (if the value is not found). While `m` can easily be associated with array `v` because it is used in the code as index (`v[m]`), variables `l` and `r` are not, because they are not used in that way — they are used to calculate `m`, which in turn is associated to `v`. However, one would like to see those variables illustrated next to the array. Given that apparently there is no unambiguous criteria to determine this type of association, we plan to offer the possibility of manually associating variables, as a workaround. This could be done directly in the code with an annotation, or at runtime through an action on the debugging environment (as in jGRASP[4]).

As discussed in the previous section, the activity of systematically using a debugger could by itself be beneficial in the learning process. We plan to conduct a follow-on study for comparing the use of a regular debugger versus PandionJ, in order to measure the effectiveness of the tool against a control condition that involves using a debugger.

8 CONCLUSIONS

Despite that our visualizations are related with variable roles, it was not our intent to address all the variable roles. Our main goal was to transpose prominent illustrations that are applied by teachers into PandionJ. With this respect, our automatic illustrations of program state have innovative features when compared to what is generally available in other debuggers (pedagogical or not).

Runtime information, i.e. variables and their values of the active stack frame, is not enough to deliver the kind of illustrations provided by PandionJ. Static code analysis is necessary to extract additional information related to variable roles and relationships, which are in turn used in combination with runtime information to render the illustrations.

After carrying out the first edition of our introduction to programming course using PandionJ, we conclude that its impact was clearly positive, given the significant improvement of the course pass rate.

REFERENCES

[1] 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

[2] Bishop C. and Johnson C. G. 2005. Assessing Roles of Variables by Program Analysis. In *Proceedings of the 5th Annual Finnish / Baltic Sea Conference on Computer Science Education, Kolin Kolistelut - Koli Calling*.

[3] Steve Cooper and Steve Cunningham. 2010. Teaching computer science in context. *ACM Inroads* 1 (March 2010), 5–8. Issue 1. DOI: http://dx.doi.org/10.1145/1721933.1721934

[4] 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

[5] Petri Gerdt and Jorma Sajaniemi. 2006. A Web-based Service for the Automatic Detection of Roles of Variables. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. ACM, New York, NY, USA, 178–182. DOI: http://dx.doi.org/10.1145/1140124.1140172

[6] 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

[7] 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

[8] 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.

[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] Eric Allen Robert, Robert Cartwright, and Brian Stoler. 2002. DrJava: A lightweight pedagogic environment for Java. In *SIGCSE Bulletin and Proceedings*. ACM Press, 137–141.

[11] 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

[12] 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

[13] André L. Santos. 2011. AGUIA/J: a tool for interactive experimentation of objects. In *Proceedings of the 16th annual conference on Innovation and technology in computer science education (ITiCSE '11)*. Darmstadt, Germany, 5. DOI: http://dx.doi.org/10.1145/1999747.1999762

[14] André L. Santos. 2012. An Open-Ended Environment for Teaching Java in Context. In *Proceedings of the 17th annual joint conference on Innovation and technology in computer science education (ITiCSE '12)*. Haifa, Israel.

[15] André L. Santos. 2014. Novel interaction metaphors for object-oriented programming concepts. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research, Koli, Finland, November 20-23, 2014*. 117–126. DOI: http://dx.doi.org/10.1145/2674683.2674693

[16] André L. Santos and Hugo Sousa. 2017. An exploratory study of how programming instructors illustrate variables and control flow. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research, Koli, Finland, November 16-19, 2017*. 173–177. DOI: http://dx.doi.org/10.1145/3141880.3141892

[17] André L. Santos and Hugo S. Sousa. 2017. PandionJ: a pedagogical debugger featuring illustrations of variable tracing and look-ahead. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research, Koli, Finland, November 16-19, 2017*. 195–196. DOI: http://dx.doi.org/10.1145/3141880.3141911

[18] Simon. 2011. Assignment and Sequence: Why Some Students Can'T Recognise a Simple Swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling '11)*. ACM, New York, NY, USA, 10–15. DOI: http://dx.doi.org/10.1145/2094131.2094134

[19] 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