Issues in Informing Science and Information Technology Volume 5, 2008

Using Roles of Variables to Enhance Novice's Debugging Work

Mikko-Jussi Laakso
Turku Centre for Computer Science and University of Turku, Turku, Finland
milaak@utu.fi

Lauri Malmi
Helsinki University of Technology, Espoo, Finland
lauri.malmi@tkk.fi

Ari Korhonen
Helsinki University of Technology, Espoo, Finland
ari.korhonen@tkk.fi

Teemu Rajala
Turku Centre for Computer Science and University of Turku, Turku, Finland
temira@utu.fi

Erkki Kaila
University of Turku, Turku, Finland
ertaka@utu.fi

Tapio Salakoski
Turku Centre for Computer Science and University of Turku, Turku, Finland
sala@utu.fi

Abstract

Abstract Debugging skill is an essential part of the programming skills. It is also highly related with program comprehension skills. In this paper we present a novel tool, called ViLLE, which supports learning debugging by promoting students' understanding of target program. ViLLE combines visual debugging features with the support for roles of variables. These roles promote activating schemas of variable use in programs. In addition, ViLLE supports automatic presentation of the target program in different programming languages, even in pseudo code or with textual explanations. This, in turn, helps in building more general and abstract understanding of program structures and their relation to problem domain concepts. The key features of the tool are presented, followed by a discussion of how the tool should be used in programming education.

Introduction

Programming is a complex cognitive skill. Most students face a lot of new challenges in learning the basic skills required to design and implement even small programs. Several extensive international studies have confirmed this, not to speak of results that have been reported in dozens of studies – mostly of single courses – presented in computing education conferences. For example, Valentine (2004) surveyed and classified total of 444 papers published in SIGCSE Technical Symposium conferences in years 1984-2003, all of which were related to teaching introductory programming, including different teaching methods, tools, experiments, new kinds of assignments, etc. Extensive studies include the McCracken et al. (2001) working group research with 216 students in 4 universities. The results indicate an alarming number of failures in simple programs the students were requested to code. A few years later another study was carried out by Lister et al. (2004). In this case, the topic of investigation was the students' understanding of execution of simple programs. The results of 556 students from 12 institutions indicate that the students had severe problems understanding even the smallest of the code fragments. Thus, it seems that first and second year students have serious shortcomings in both reading and writing skills of programs. It is therefore not surprising that Tenenberg et al. (2005) found out in their study – concerning 21 institutions and 300 students – that students cannot design even simple programs after their introductory courses.

What makes learning to program so complex, and how should we tackle this problem in education? Obviously much of the complexity follows from the fact that programming includes many different types of tasks, including problem solving, conceptual analysis of problem domain, program design, detailed temporal time splitting of actions, developing and combining algorithms and data structures, understanding language issues – both syntax and semantics, writing program code, testing and finally debugging it. Mastering all of these requires a lot of training and experience, which cannot be acquired during a single introductory course. Moreover, programming requires thinking with abstract concepts, which is not easy for all novices.

du Boulay (1989) classified some of these challenges by identifying five different subfields of programming skill that a novice student has to learn to work effectively. Firstly, he must gain a general understanding in what programming is about and what computers can do. Secondly, he needs to understand the principles of how programs execute within a computer. du Boulay used a term *notional machine*, which means a general model of computer internals and program execution – including how memory is used for storing variables, how statements and procedures are executed etc. Thirdly, computer programs are written using programming languages, artificial formal notations. Each of these has its own syntax and semantics that must be understood. Fourthly, learning programming means acquiring a large set of schemas of how things – such as scanning an array of data to identify information, building a linked list or reading data from input source – are typically implemented. Knowledge of such schemas reduces the cognitive complexity of reading and writing programs, as the programmer can focus on composing larger chunks of code from smaller ones instead of thinking all the details simultaneously. Finally, programming requires practical skill – programmers need to know (and be able to use) special tools such as editors, compilers, profilers and debuggers for coding, compiling, testing and debugging programs.

Programming education has traditionally put a lot of effort in teaching the syntax of particular language. However, when recalling du Boulay's five subfields of programming skill, we note that issues concerning syntax mostly cover the last three areas, whereas the second one, understanding program execution, may easily be somewhat overlooked, or is at least less emphasized. However, it should be seen as an essential part in the programming skills.

In this paper, our focus is on the relation of program comprehension and debugging. A seemingly obvious conception is that good programming skills should always include good debugging skills. However, Ahmadzadeh, Elliman, and Higgins (2005) demonstrated that this relation is not that obvious. In their study, including almost 200 students, they found that less than 40 percent of students getting good marks on a programming course were able to identify and correct all errors in a relatively simple program in a controlled situation. This group was surprisingly small. Students who performed poorly on the course got even worse results. Thus, we conclude that there still seems to be a positive correlation with good debugging skills and good programming skills.

We discuss how to improve students' debugging practices with better program comprehension. Our assumption is that this, in turn, will improve their general performance in programming tasks. The debugging process inherently includes applying a mental model of program execution, i.e., the ability to follow execution of code, make predictions on variable values and observe their actual values to identify possible discrepancies. This requires a fair understanding on how the program executes on the notional machine, and especially how variable values are changed during the program execution. However, typical debuggers track only snapshots of variable values during program execution. They do not support program comprehension by identifying and emphasizing different schemas or by demonstrating the variable use. We are specifically interested in schemas called roles of variables, first presented by Sajaniemi (2002). The key idea is that almost all variables conform to one of the few identified behavioural patterns. For example, one variable may act as a counter, thus having the role of a *stepper*, and another stores a sum of calculations, thus being a *gatherer*. Or a value that is being looked for from a wider collection of data is stored in a *most-wanted-holder*. Thus, roles form an abstract form of variable behaviour, which describes program execution from the data point of view, whereas control structures and functions describe the program from the execution point of view. Thus, roles of variables identify valuable extra information of the program code to enhance the understanding of program behaviour. This information supports debugging by forecasting the expected changes in variable values during the program execution, making it easier to notice mismatch between the expected value and observed value. This way identifying faulty behaviour becomes easier.

Unfortunately, identifying roles of variables and following program execution are not easy tasks for novices. Therefore proper tools are required to illustrate both variable behaviour and control flow. A whole field of research, software visualization (For an overview, see for example, (Stasko, Domingue, Brown, & Price, 1998) has concentrated on examining and demonstrating program code, its structure, and the execution of code. Its two important subfields are *program visualization*, in which the focus of activity is on illustrating the dynamic behaviour of actual program code and variable values (see, for example, Jeliot (Moreno, Myller, Sutinen, & Ben-Ari, 2004), DDD (Zeller, 2001), jGrasp (Jain, James, Cross, Hendrix, & Barowski, 2006), and BlueJ (Zeller, 2001), and *algorithm animation* where the focus is on the visualization of dynamic behaviour of more abstract concepts: data structures and algorithms (see, for example, Animal (Rößling, Schüler, & Freisleben,, 2000), o JHAVE (Naps, Eagan, & Norton,, 2000), JAWAA (Akingbade, Finley, Jackson, Patel, & Rodger, 2003), Samba (Stasko, 1997)). However, many current tools, such as Jeliot, jGRASP, and Animal include features from both of these subfields. Where the goal of program visualization is on illustrating the execution of the target program, some tools have features better supporting the debugging process, such as proper control over the execution (e.g. DDD, jGRASP, Retrovue (Callaway, 2002)). These tools can be called visual debuggers.

The focus of current program visualization and visual debugging tools is on illustrating and controlling the dynamic behaviour of the target program in terms of control flow. Thus, these tools are good for visualizing control execution, control structures, and function calls. Visual debuggers often illustrate data structures as well. However, few tools support good depictions of program history, and history data of variable values, and effectively none support visualization of roles. The only tool we are aware of, is PlanAni (see, e.g., Sajaniemi & Kuittinen, 2003), which is more a tool for demonstrating roles of variables than a visual debugger.

In this paper, we present a new tool ViLLE that integrates facilities both from visual debuggers and the role analysis. Thus, the tool supports understanding programs in more versatile ways than any previous tool that we know. We discuss the features available in ViLLE and how it can be used in education.

In the next two sections, we present some research on program comprehension and roles of variables. In Section 4 we discuss debuggers and what is needed for efficient debugging. In Section 5, we present ViLLE and in Section 6 we discuss the use of ViLLE in education. Finally, some conclusions are given.

Program Comprehension

Computer programs are complex objects, and their internal structure and working can be understood from different points of view. One obvious view is to differentiate the syntax, semantics and pragmatics of a program, i.e., what are its constituents, how do they work and what are they used for in the target program. Another view could be to separate different levels of abstraction in programs. For example, a debugger can operate on single statements and single variable values, i.e., on a low level. A high level view could reveal the working of a whole software system giving aggregate information of memory usage and object populations. There could be different levels between these, such as method level, class level or architectural level descriptions of program structure and behaviour (Pacione, 2004). Shneiderman and Myers (1979) present a model of syntactic / semantic interaction. Where syntactic information is language dependent, semantic information is more general. It is multi-leveled, and the human understanding of a program is built by recognizing the function of program components and fragments as chunks. These pieces are aggregated until a description of the entire program is available.

Détienne discusses research of program comprehension in some detail in her book (Détienne, 2002). There are several theoretical approaches to explain how people understand code. First, the functional approach is built on the hypothesis that understanding a program means activating and instantiating knowledge schemas that present the generic knowledge a software expert possess. Such schemas can be either programming schemas or problem schemas. Détienne describes their significance, as follows: "The activity of understanding consists, in part, of activating schemas stored in memory, using indexes extracted from program's code, and inferring certain information starting from the schemas invoked."

The structural approach, on the other hand, views understanding a program as constructing a network of propositions, thus highlighting the importance of structural knowledge in understanding. This structural knowledge can be of control structures, or functional parts of the program like Input, Calculate and Output. The dominating aspect is, however, the program structure.

The third approach, mental models, stresses that we distinguish two different ways to understand a program: *the program model*, which is about the program structure and the *situational model*, which is related to the problem domain. Combining these two views is the key to understanding a program. Pennington (1987) evaluated the validity of this approach within a study of how professional FORTRAN and COBOL programmers understood code. She gave them a code with a general description what the code is about, but without any comments, and asked them to summarize its behaviour, followed by implementing certain changes in the code. Pennington observed that people who presented in their summaries both language level concepts and problem domain level concepts performed clearly better than those who concentrated only on language level concepts or problem domain concepts. She concluded that combining these two domains is essential for good program comprehension.

Finally, Détienne mentions a fourth approach: seeing program comprehension as problem solving. Here the focus is in the information selection process when reading a program, because programs are not read sequentially. Instead, people constantly jump forwards and backwards when reading code.

In this paper – as considering the activity of debugging – it seems evident that schemas related to use of variables are important because an inherent part of debugging is tracking values of variables, and asserting their correctness or faultiness. Thus we should emphasize such schemas in the debugging process. The mental model approach, making the connection between language level concepts and problem domain concepts, is important. A debugging tool should provide different views of the program, both low (language) level and high (pseudo code or even verbal descriptions) level, to aid the user to create bindings between the program model and the situational model.

Roles of Variables

Roles of variables are stereotypes of variable use in computer programs (Sajaniemi, 2002). The basic idea behind roles of variables is to distil expert programmers' tacit knowledge on variables and their use. A role does not encapsulate a unique task in some specific program, but merely a number of variables in many programs. Thus, a very small set of roles is enough to cover almost all variables encountered in programs written by novice programmers.

In the following, we give a brief overview to roles of variables. In program code variable acts as an identifier that can refer for example to a scalar value or an array. However, we encourage the reader to visit the *Roles of Variables Home Page* (http://cs.joensuu.fi/ saja/var roles/) to read a more comprehensive introduction to the role concept.

1. *Fixed value* is a variable which, after once initialized with a proper value, does not get a new one.
2. *Stepper* goes through a succession of values that are predictable and known in advance.
3. *Most-Recent Holder* is holding the latest value.
4. *Most-Wanted Holder*, on the other hand, is holding the most appropriate value encountered so far while examining a succession of unpredictable values.
5. *Gatherer* accumulates the effect of individual values.
6. *Follower* gets its new value based on an old value of some other variable.
7. *One-Way-Flag* has two possible values, its initial value, and some other value that is never changed anymore, if reached.
8. *Temporary* holds a value only for a very short time.

In addition, there are 3 other roles that are related to data structures.

9. *Organizer* stores data elements to be rearranged,
10. *Container* stores data elements to be added and removed, and
11. *Walker* traverses data structures.

Roles are cognitive concepts, which mean that different persons may have different interpretations of them. For example, a variable having the values from the sequence of Fibonacci numbers may be interpreted to be a Stepper by a mathematician, but a Gatherer by a programmer that see it summing up two Followers in each iteration. As long as this interpretation helps the programmer to grasp or explain the idea behind the variable, and build a new 'chunk' of knowledge, it is good for schema formation. While, expert programmers' (and instructors) have tacit knowledge on variables and their use, novices lack this knowledge, and thus need ways to make it explicit.

Roles can be taught in introductory programming courses gradually as they appear in examples. After this, roles can be used in program design, implementation, and debugging tasks all of which require program comprehension skills.

Sajaniemi and Kuittinen (2003) have studied program comprehension skills and they conclude that students who are taught programming with roles of variables outperform other students. They are not only able to describe the program behaviour better in terms of program summaries, but they also attain better mental model of the programs. Such a schema formation is necessary in order to be able to debug programs, as we are going to argue in the next section.

What is Debugging?

The ultimate goal of debugging process is to remove defects from computer programs (Chmiel & Loui, 2004). More precisely, it is a process of locating the exact position of the error and fixing it after the existence of error is verified by means of testing (Vessey, 1986). In addition, this process may include other tasks such as determining the cause of the error in order to fix it.

Debugging is often found and classified as a hard task to learn and master. This is due to its multifaceted nature. Ducasse and Emde (1988) have presented a classification of debugging knowledge sliced into seven categories: 1-2) knowledge of the intended program, and the actual program, 3) understanding of the programming language, 4) general programming expertise, 5) knowledge of the application domain, 6) knowledge of bugs, and 7) knowledge of debugging methods. Even with a quick glance to this list, it can be said that for novices most of these topics are unfamiliar and hard to grasp in the early phase of learning to program. Thus, it takes a lot of effort from a novice programmer to debug even simple programs. This is evidently one reason why novices find learning to program so time consuming and frustrating process (Johnson, 1990).

It can be said that the essential nature of debugging is testing hypotheses about what causes an error, where to find it, and finally how to fix it. These hypotheses are derived from programmer's mental model of the target program and its execution. In addition, novices are actively and continuously developing this mental model while gaining more experience of debugging and writing programs.

Difficulties in the debugging process are all about how to create relevant hypothesis and how to test them. Experienced programmers can easily find simply errors and narrow down the causes while novice programmers use trial and error method to debug programs (Lee & Wu, 1999). More over, they may even end up inflicting new bugs on the program during the course of trying to find the original ones (Gugerty & Olson, 1986). In addition, according to Smith and Webb (1995), experts are often able to make hypotheses about the most probable cause of an error. They are also skilled at isolating and identifying errors due to the experience they have. Thus, without any experience, novice's mental model of program code and its execution can be quite far a way from the ideal one and as a consequence debugging the program can be extremely difficult.

Gugerty and Olson (1986) argued that experts' superior debugging ability originates from their better skill to comprehend the program. In addition, according to Lee and Wu (1999) the program comprehension was often mentioned as the crucial skill for being able to debug efficiently. It is interesting, though, to compare this to the results obtained by Ahmadzadeh et al. (2005). In their study with almost 200 students, they found that less than 40 percent of students getting good marks on introductory course of programming were able to identify and correct all errors in a given relatively simple program in a controlled programming task. It would seem obvious that good performance in programming would imply a good skill in program comprehension, which in turn would imply good debugging skills. This research challenges this hypothesis at least for novices. However, their results about students performing badly in the course were that their debugging skills were clearly worse than those of "good programmers". We therefore conclude that there still seems to be a positive correlation with good debugging skills and good programming performance.

Nevertheless, our assumption is that better program comprehension aids students to create and test more easily their hypothesis that aim at discovering and correcting errors. These hypotheses, as aforementioned, are based on student's current mental model of the programming task. We note that the model may change and improve during the program comprehension process as student's understanding of the task increases. Moreover, we believe that there is a cycle that can be called a debugging-comprehension cycle, running from the beginning of the process of learning to program. Good comprehensions skills support the possibility to become a better debugger, because with them the student is able to create more relevant hypotheses. Correspondingly good debugging skills enhance student's ability to comprehend program execution because with them he/she can test hypotheses more efficiently and thus gain a better understanding of the program

Our assumption is supported by Kuittinen and Sajaniemi (2004) who noticed that using role information in basic programming courses promoted students' understanding about the whole program. Moreover, they showed that with the role concepts students can interpret program code in a more abstract and novel way regardless of the programming language or even the programming paradigm. The role concepts provide information about the behaviour of variables, which aids students to modify and develop their own mental model of the target program. For example, students often focus too much on the execution of single statements and the execution order instead of variables and how they behave. Therefore, in teaching, we should emphasize the behaviour of variables and point out mismatches on their actual use and definitions; if we know that the role of a certain variable is a stepper then the variable should act like a stepper.

To conclude, the role information aids student's to comprehend programs more deeply and with better program comprehension they can debug more efficiently. In addition, better debugging skills aid program comprehension even further by providing tools and ways to grasp the idea behind a program code.

This said, we note that recognizing roles when reading program code is far from easy for novices. Therefore, in order to use roles of variables effectively in teaching programming and debugging we need a tool that is able to identify automatically the roles within the program code written by the student. Moreover, the tool must also have functionalities to gather the dynamic information about the behavior of the variables like their values etc. The values need to be logged during the program execution, followed by comparing this information with the identified role of the variable. We also need a manner to interactively work with the roles to enhance student knowledge of how to recognize different roles and how to use them in practical debugging. For example, during the program execution, questions could be asked about the role of a particular variable and its previous, current or next values. Finally, to enhance the learning effect while teaching programming or debugging, we need a tool that provides these kinds of features and combines the presented approaches together.

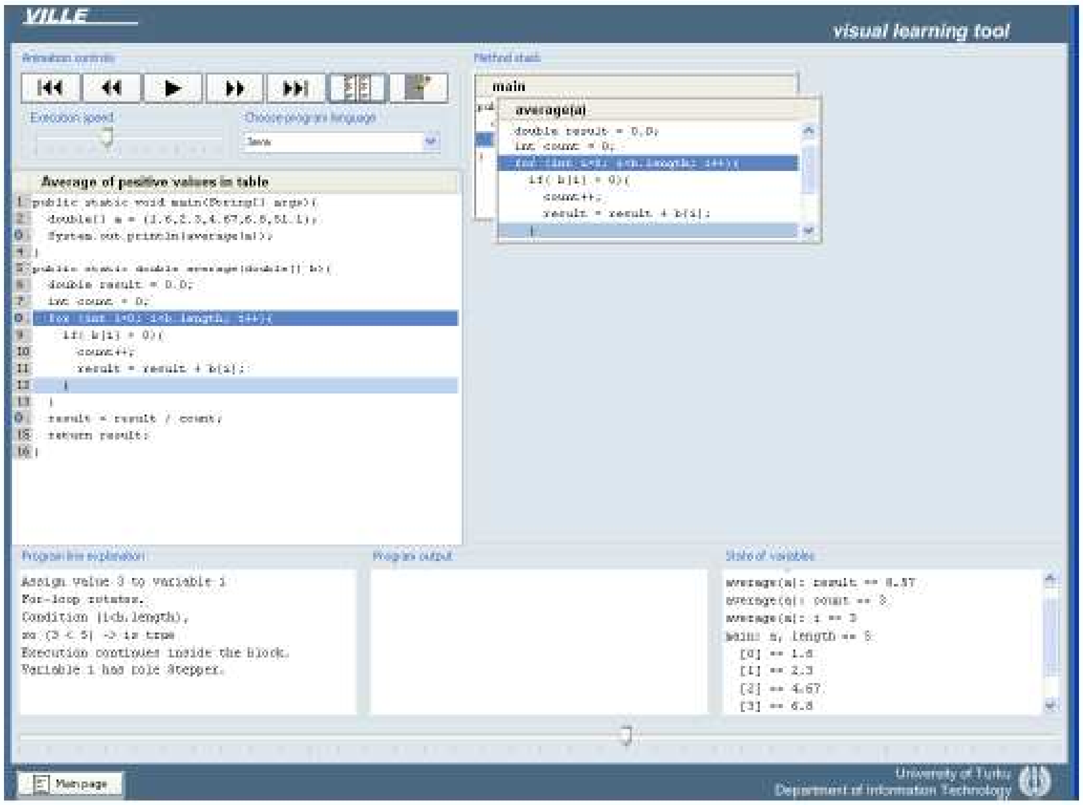
VILLE - A Tool for Debugging and Executing Program Code

ViLLE is a program visualization tool, which can be used to create and edit various programming examples, and to observe the events in the programs during their execution. The tool can be used both in lectures and in independent learning. Its main purpose is to support the learning process of novice programmers.

ViLLE supports typical features of a visual debugger, including controlled program execution, visualization of the execution path in the code and the values of variables (see figure 1). However, there are a number of novel features as well. Firstly, ViLLE automatically displays role information of the variables in the target program. Secondly, ViLLE supports presentation of the target program in many imperative programming languages (currently Java and C++) as well as in pseudo code in order to show the conceptual similarity among imperative programming languages. Finally, ViLLE allows execution of programs both forwards and backwards, which is an important benefit for debugging.

Currently the tool supports the following features of Java: basic variable types (int, float, double, Boolean), main aspects of the String class, conditional statements (if, else-if and else), loop structures (for, while and do-while), methods, one and two dimensional arrays, and records. These programming concepts cover majority of the topics usually included in the curriculum of a first programming course. ViLLE is not designed for teaching interaction of objects, but sequential execution of programs and algorithms. This is essential in order to grasp the basic understanding of programs regardless of selected programming language or even whether we teach imperative or/and object oriented programming. More detailed information about the tool can be found in Rajala, Laakso, Kaila, and Salakoski (2007) and it's effectiveness in programming learning is evaluated in Rajala, Laakso, Kaila, and Salakoski (2008).

Figure 1 shows the visualization view of ViLLE. On the left side of the view we see how the current line and the previously executed line are highlighted from the program code. A user can add breakpoints to program lines by clicking the line number from the code area. The buttons for controlling the visualization are situated in the upper left corner. Three text areas at the bottom of the view display an explanation of the current program event (including the role information of variables), program outputs, and the states of variables. Method calls are visualized with a call stack on the right side of the view. The call stack area can be replaced with a variable state area, which



Average of positive values in table

```
public static void main(String[] args){
  double[] a = {1.6,2.3,4.67,6.8,51.1};
  System.out.println(average(a));
}
public static double average(double[] b){
  double result = 0.0;
  int count = 0;
  for (int i=0; i<b.length; i++){
    if( b[i] > 0){
      count++;
      result = result + b[i];
    }
  }
  result = result / count;
  return result;
}
```

```
Assign value 3 to variable i
For-loop rotates.
Condition (i<b.length),
so (3 < 5) -> is true
Execution continues inside the block.
Variable i has role Stepper.
```

```
average(a): result == 8.57
average(a): count == 3
average(a): i == 3
main: a, length == 5
  [0] == 1.6
  [1] == 2.3
  [2] == 4.67
  [3] == 6.8
```

Figure 1: The visualization view of ViLLE

visualizes arrays and matrices graphically. The slider below the three text areas indicate, how far the execution has progressed. By moving the slider, the user can progress to any state of the program execution.

Key Features

Execution line by line, progress in program execution is represented by highlighting lines from the code. In addition, VILLE highlights the previously executed line with a different color. This makes following of the program execution easier especially in loops and for novices (Korhonen, Sutinen, & Tarhio, 2002).

Flexible control of visualization both forwards and backwards, the user can move one step at a time, both forwards and backwards in the execution of the program. Examples can also be run automatically with adjustable speed. In addition, VILLE has an execution slider with which the user can move to any state in program execution. These are features we have used to see in algorithm animation tools (see, e.g., Moreno et al., 2004; Rößling et al., 2000; Stasko 1997). However, moving backwards in the program execution is usually not possible in debuggers and it is missing from many program visualization tools.

Breakpoints, the user can set breakpoints into any program code line and move between them both forwards and backwards. This functionality enables debug-based control and observation of the program execution. Again, backward tracing between breakpoints is a novel feature, which is not available in other debuggers.

Code line explanation, every code line has a description in which all the program events related to the line are clearly explained verbally. Furthermore, all possible outputs and variable states are shown. This is also a feature that is absent in many similar applications.

Role information, information about the roles of variables is integrated into the code line description. According to Sajaniemi and Kuittinen (2004), this helps in learning programming, and enhances understanding of the program.

Some papers cover and indentify some typical programming errors for novices (see Sporher & Soloway, 1986; Joni, Soloway, Goldman, & Ehrlich, 1993). For example, Joni et al. (1993) have noted Array Index Variable bug, which is shown in Figure 2.

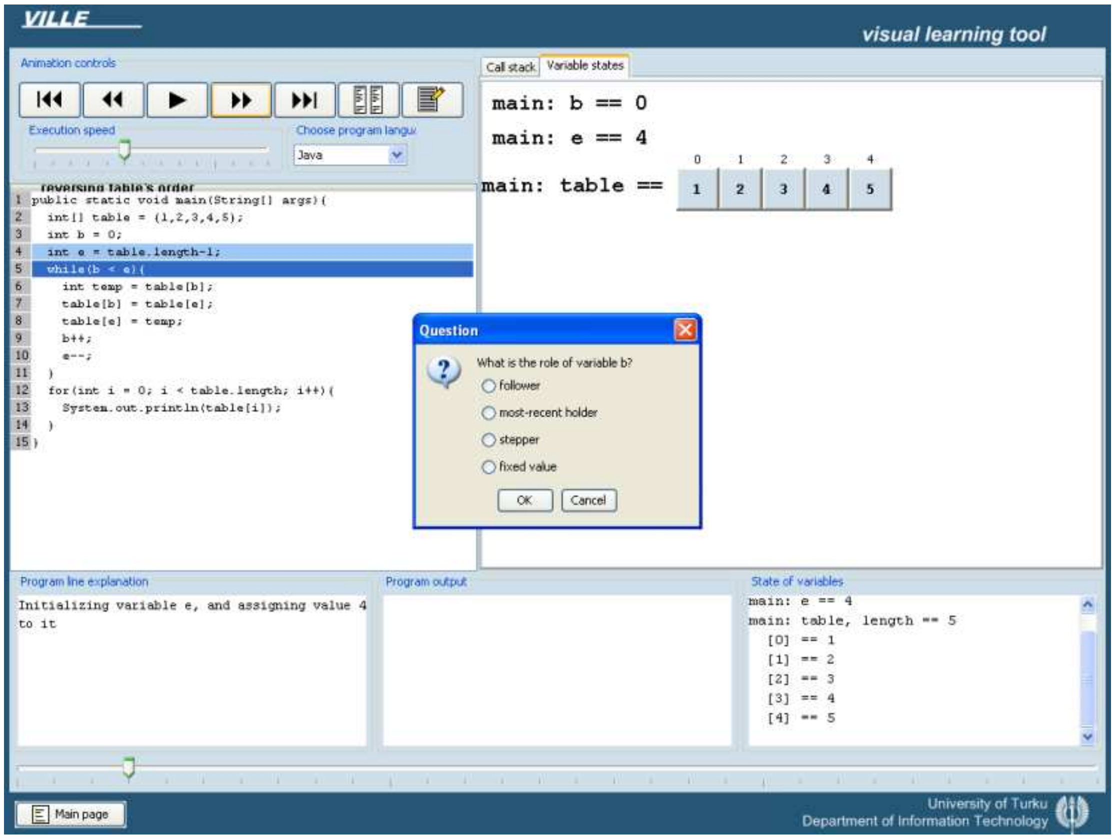
```
...
for (int i = 0; i < 100; i++){
    tbl[some_other_than_i] = 0;
}
```

Figure 2: Example of Array Index Variable bug in Java

In this case, the array is not referenced with loop variables as intended. With role information student can easily notice this type of error, because variable *i* should have role Stepper and now it has role of Fixed value. Similar to this, all errors in which the loop variable is not incremented can be found more easily with role information in hand.

Pop-up questions, one useful feature of VILLE is the possibility to create pop-up questions (see, e.g., Naps et al. 2000) for the programming examples. With the built-in editor, a teacher can create multiple choice questions and set them to trigger in certain states of the program execution.

Figure 3 shows an example of a pop-up question, which asks the user to select the correct role for a variable in the program.



reversing table's order

```
public static void main(String[] args){
  int[] table = {1,2,3,4,5};
  int b = 0;
  int e = table.length-1;
  while(b < e){
    int temp = table[b];
    table[b] = table[e];
    table[e] = temp;
    b++;
    e--;
  }
  for(int i = 0; i < table.length; i++){
    System.out.println(table[i]);
  }
}
```

```
main: b == 0
main: e == 4
main: table ==  | 1 | 2 | 3 | 4 | 5 |
```

What is the role of variable b?

- ◯ follower
- ◯ most-recent holder
- ◯ stepper
- ◯ fixed value

OK Cancel

Initializing variable e, and assigning value 4 to it

```
main: e == 4
main: table, length == 5
  [0] == 1
  [1] == 2
  [2] == 3
  [3] == 4
  [4] == 5
```

Figure 3: A pop-up question

Call stack, moving the program execution point between different methods due to function calls and returns is visualized with a call stack. When a method is called, a new window is opened in the call stack. The window remains in the call stack until the method is finished. When the execution returns to the caller, a return value is shown on top of the call stack. The visualization of the execution can be alternatively viewed in a parallel view with the program code viewed in two languages simultaneously.

Discussion

Programming is a complex cognitive skill, thus learning programming is hard. In this paper, we have argued that proper tools for debugging would promote program comprehension and clarify the students' mental model of program execution. The key idea is to improve students' debugging practices by providing novel debugging tools especially suitable for novice level programmers. The debugging process inherently includes the ability to follow execution of code as well as making predictions and observations on variable values. This requires a proper understanding on how the program executes on the notional machine, and especially how variable values are changed during the program execution. By using proper debugging tools, the student gradually comprehends the program code, which eventually helps developing better debugging skills. Ultimately, the aim of this program comprehension cycle is to promote the understanding of the principles of program execution in general--a learning task that we feel is overlooked or at least less emphasized in text books.

An initial step is required to get this debugging-comprehension cycle running. Our method is to use Roles of Variables to give the first impulse in this respect. The role information automatically attached to variables help novices to comprehend with the code. They can start following a single variable and verify its responsibility during the program execution. This way they can gradually improve the debugging skills until they are ready to take the next step, i.e., use their new skills to develop the mental model of the program even further.

Software visualization techniques can support the debugging process by supporting the control over the program execution. Tools adopting such techniques are called visual debuggers. The current program visualization and visual debugging tools are good for illustrating and controlling the dynamic behavior of the target program in terms of control flow. Thus, they focus on visualizing control execution, control structures, and function calls besides showing variable values. However, typically these tools do not include the role information of variables. ViLLE is a novel visual debugger that includes both visual debugging facilities and role analysis. The tool thus supports understanding programs in more versatile ways than any previous tool that we know.

ViLLE is a program visualization tool which incorporates the role information and the aforementioned features for enhancing novice level programming and debugging skills. We suggest that the role concept should be included in introductory programming course's curriculum, because good knowledge and understanding of the role concept enhances student's ability to comprehend programs in more abstract level. In addition, ViLLE provides a feature to generate pop-up questions about the role knowledge, the kinds like “what is the role of variable total”, “which sums up the values in a table” or “what is the role of variable i that travels through the values 1, 3, 5, 7, 9 during the execution of an algorithm”. In addition, questions such as "what values the variable iter encounters during the execution" can also be asked.

The system conforms also to several different teaching methods by promoting *the programming language independency paradigm*. This also aims for better comprehension of programs due to the fact that from student's point of view, it is not that important to learn how loops are defined in particular programming language, but far more importantly learn the basic principles behind loop structures regardless of the language. This is supported in ViLLE, as teachers can define pseudo languages of their own, suiting their needs better. The defined (pseudo) language can be visualized and executed like any other imperative language. Furthermore, these aspects can be emphasized by executing the example simultaneously in parallel view with two different programming languages. Thus, a student can notice that programming is not a skill of mastering the syntax of a programming language, but of mastering the basic concepts and semantics behind programs. This can also help the changeover from one programming language to another.

To support debugging and comprehension skills even more, ViLLE automatically generates a description of the executed code to aid understanding the purpose of every single code line. Not to mention that the tool holds also typical features related to debugging: breakpoints, step-by-step execution, moving between breakpoints both forward and backward, representation of states of variables, etc. The backward functionality, just to name one, is missing from many debugging tools, which often frustrates students especially in the early phases of learning.

We suggest that ViLLE should be used in the introductory programming courses to gather examples, to define a language if needed, to emphasize main points of programming language independecy paradigm, to support learning of role concepts, to boost effectiveness of code examples, and finally, to improve students ability to debug and comprehend programs.

Conclusions

In this paper, we have presented how ViLLE can be used to enhcance debugging skills of novices. ViLLE is a versatile tool for visual debugging, suitable for novice level programmers that are still forming their mental model of program execution. The tool promotes the use of roles of variables which is a novel concept for novice level programmers to grasp the essential characteristics of programs at glance. This role information aids learning of program comprehension and assists the students in their way to master programming and debugging skills. In addition, ViLLE supports programming language independency paradigm, in which the goal is to comprehend programs and their components in more abstract level.

Future Directions

So far we have a proof-of-concept tool to show that the visualizations and role information can be automatically extracted from the source code. However, experimental studies will be needed in the future to show that this new tool actually promotes learning and has the desired quality in teaching and learning process in general.

Acknowledgments

This work was supported by the Academy of Finland under grant number 111396.

References

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education,* 84-88, New York, NY, USA.

Akingbade, A., Finley, T., Jackson, D., Patel, P., & Rodger, S. H. (2003). JAWAA: Easy web-based animation from CS0 to advanced CS courses. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, SIGCSE'03*, 162-166, Reno, Nevada, USA.

Callaway, J. (2002). *Visualization of threads in a running Java program.* Master's thesis, University of California, June 2002.

Chmiel, R., & Loui, M. C. (2004). Debugging: From novice to expert. *Proceedings of the 35th SIGCSE Technical Symposium on Computer science education*, 17-21, New York, USA.

Détienne, F. (2002). *Software design – Cognitive aspects*. Springer-Verlag. ISBN 1852332530.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J. Spohrer (Eds.), *Studying the novice programmer* (pp. 283-299).

Ducasse, M., & Emde, A.-M. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. *Proceedings of the 10th International Conference on Software Engineering*, 162-171, Singapore.

Gugerty, L., & Olson, G. M. (1986). Debugging by skilled and novice programmers. *CH186 Proceedings,* 171–174.

Jain, J., James, I., Cross, H., Hendrix, T. D., & Barowski, L. A. (2006). Experimental evaluation of animated verifying object viewers for Java. *SoftVis '06: Proceedings of the 2006 ACM Symposium on Software Visualization*, 27-36, New York, NY, USA.

Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial Intelligence, 42,* 51-97.

Joni, S. N., Soloway, E., Goldman, R., & Ehrlich, K. (1983). Just so stories: How the program got that bug, *Proceedings of the SIGCUE/SIGCAS Symposium on Computer Literacy.*

Korhonen, A., Sutinen, E., & Tarhio, J. (2002). Understanding algorithms by means of visualized path testing. In S. Diehl (Ed.), *Software visualization: International seminar*, 256-268, Dagstuhl, Germany.

Kuittinen, M. & Sajaniemi, J. (2004). Teaching roles of variables in elementary programming courses. *SIGCSE Bulletin, 36*(3), 57-61.

Lee, G. C., & Wu, J. C. (1999). Debug it: A debugging practicing system. *Computers and Education, 32*, 165-179.

Lister, R., Seppälä, O., Simon, B., Thomas, L., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., & Sanders, K. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*(4), 119-150.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin, 33*(4), 125-180.

Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. *Proceedings of the International Working Conference on Advanced Visual Interfaces*, 373-376, Gallipoli (Lecce), Italy.

Naps, T.L., Eagan, J. R., & Norton, L. L. (2000). JHAVÉ: An environment to actively engage students in web-based algorithm visualizations. *Proceedings of the SIGCSE Session*, 109-113, Austin, Texas.

Pacione, M. J. (2004). Software visualisation for object-oriented program comprehension. *Proceedings of the 26th International Conference on Software Engineering*, 63-65, Los Alamitos, CA.

Pennington, N. (1987). Comprehension strategies in programming. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop*, 100-113.

Rajala, T., Laakso, M.-J., Kaila, E., & Salakoski, T. (2008). Effectiveness of program visualization: A case study with the ViLLE tool. (Manuscript submitted for publication.)

Rajala, T., Laakso, M.-J., Kaila, E. & Salakoski, T. (2007). VILLE – A language-independent program visualization tool. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research* (Koli Calling 2007), Koli National Park, Finland, November 15-18, 2007. Conferences in Research and Practice in Information Technology, Vol. 88, Australian Computer Society. Raymond Lister and Simon, Eds.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137-172.

Rößling, G., Schüler, M., & Freisleben, B. (2000). The ANIMAL algorithm animation tool. *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'00*, 37-40, Helsinki, Finland.

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments,* 37-39.

Sajaniemi, J., & Kuittinen, M. (2003). Program animation based on the roles of variables. *Proceedings of the 2003 ACM symposium on Software visualization*, 7-16, New York, USA.

Shneiderman, B., & Myers, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming, 8*(3),219–238. ISSN 0885-7458.

Smith, P. A., & Webb, G. I. (1995). Transparency debugging with explanations for novice programmers. *Proceedings of the 2nd Workshop on Automated and Algorithmic Debugging,* St. Malo.

Spohrer, J., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM, 29*(7), 624-632.

Stasko, J. T. (1997). Using student-built algorithm animations as learning aids. *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, 25-29, San CA, USA.

Stasko, J. T., Domingue, J. B., Brown, M. H., & Price, B. A. (1998). *Software visualization: Programming as a multimedia experience*. Cambridge, MA: MIT Press. ISBN 0-262-19395-7.

Tenenberg, J., Fincher, S., Blaha, K., Bouvier, D., Chen, T.-Y., Chinn, D., Cooper, S., Eckerdal, A., Johnson, H., McCartney, R., & Monge, A. (2005). Students designing software: A multi-national, multi-institutional study. *Informatics in Education*, *4*(1), 143-162.

Valentine, D. W. (2004). CS educational research: A meta-analysis of SIGCSE technical symposium proceedings. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 255-259, New York, NY, USA.

Vessey, I. (1986). Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man, and Cybernetics*, *16*, 621–637.

Zeller, A. (2001). Animating data structures in DDD. *Proceedings of the First Program Visuliztion Workshop – PVW 2000*, 69-78, Porvoo, Finland.

Biographies

Mikko-Jussi Laakso is a PhD student working as a researcher in a joint project of University of Turku and Helsinki University of Technology. He received his M.Sc (Computer Science) in 2003. His research interest covers program and algorithm visualization, learning environments, computer aided and automatic assessment in computer science education.

Lauri Malmi is a professor of Computer Science at Helsinki University of Technology. He received his D.Sc. (Tech.) degree from the same university in 1997. His current research concentrates mostly on computing education research, especially developing and evaluating tools for supporting programming education, and understanding various aspects of how students learn programming. Professor Malmi is leading the COMPSER research group (http://www.cs.hut.fi/Research/COMPSER/).

Ari Korhonen is a researcher and instructor at Helsinki University of Technology (HUT). He received his M.Sc. (Computer Science) in 1997, and his D.Sc. (Tech) diploma in 2003. His research includes data structures and algorithms in software visualization, various applications of computer aided learning environments and automatic assessment in computer science education.

Teemu Rajala is a PhD student at the University of Turku. He received his master's degree from the same university in 2007. His research focuses on visualization of programs and algorithmic problem solving.

Erkki Kaila is writing his master thesis on program visualization in programming learning in University of Turku. His research interests include program visualization systems and IT education.

Tapio Salakoski is a professor of Computer Science at University of Turku, where he received his Ph.D. in 1997. His main research focus has been in methodology development using machine learning and other intelligent techniques. He is leading a multidisciplinary research group studying various task domains, including problems related to human learning and computing education research.