# VILLE – A Language-Independent Program Visualization Tool

**Teemu Rajala** **Mikko-Jussi Laakso** **Erkki Kaila** **Tapio Salakoski**

Department of Information Technology
University of Turku
20014 Turku, Finland

{temira, milaak, ertaka, sala}@utu.fi

## Abstract

Visualization tools have proven to be useful for enhancing novice programmers' learning. However, existing tools are typically tied to particular programming languages, and tend to focus on low-level aspects of programming such as the changing values of variables during program code execution. In this paper we present a new program visualization tool, which provides a language-independent view of learning programming. Moreover, program execution can be viewed in two languages simultaneously. Complete with role information of variables, the tool supports the learning process at a more abstract level, thus emphasizing the similarities of basic programming concepts and syntax in all imperative programming languages.

*Keywords*: Language independency, teaching programming, novice programming, program visualization.

## 1 Introduction

Teaching programming has provided challenges for computer science education for many decades. Constructing and even understanding computer programs has proven to be highly non-trivial task for most learners (McCracken et al. 2001, Lister et al. 2004, Tenenberg et al. 2005). Many computer-based systems have been developed to aid the learning process, particularly for novice programmers. Existing systems use various visualizations and animation techniques to assist the learners in understanding the behaviour of program execution (Hundhausen et al. 2002).

In general, most visualization and animation systems are heavily dependent on a particular programming language, and can only visualize program execution in that language. However, the syntax and structure of basic programming concepts are very similar in all imperative programming languages. Those concepts include, for example, control structures (sequence, selection, and loops), statements, expressions, arrays, and methods. From a student's point of view it is not particularly important to learn how loops are defined and executed in a particular programming language; it is far more important to understand the basic principles behind the loop structure regardless of the programming language in question.

Grandell et al. (2006) have argued that in programming courses for novices, the syntax of the programming language should be as simple as possible. Simple syntax allows students to focus on learning the very concept of programming instead of struggling with excessive syntax. Thus in our opinion a simple pseudo-language could be used effectively as a first teaching language. When using a pseudo-language, the algorithm as well as the corresponding program code can be represented on a higher level of abstraction, as Boada et al. (2004) and Stern et al. (1999) have stated. However, as Garner (2006) has noted, a pseudo-language is often perceived as a language that can't be interpreted or executed.

Another abstraction of learning programming is provided by the roles of variables. Sajaniemi (2002) has defined a taxonomy of roles of variables, based on their behaviour during the execution of programs. The concept can be utilized regardless of programming language or even programming paradigm. Sajaniemi and Kuittinen (2003) have noticed that using the role information of variables in basic programming courses improves the learning process of students by enhancing their understanding of the program.

VILLE is a language-independent program visualization tool providing an abstract view of programming. It can be used both in lectures and for independent learning. It has a built-in syntax editor with which users can add new languages to the tool or modify the syntax of built-in languages (currently including Java, C++, and a pseudo-language). The visualizations can be viewed in any of the defined languages. To emphasize the language independency, VILLE has a parallel view displaying a program in two languages simultaneously. It is possible to trace program execution line by line and monitor program outputs and changes in variable values. To make visualization more effective and easily interpretable, there is an automatically generated textual description of each code line, including the role information of variables. VILLE comes with a set of predefined examples, which can be easily extended. In addition, VILLE's predefined or user-defined examples can be published on the web, allowing students to engage with a learning session at any time and place.

The structure of this article is as follows: section 2 presents related work, previous studies and related systems. VILLE and its features are presented in section

3. Section 4 presents the discussion, and finally section 5 presents the conclusions in brief.

## 2 Related work

Defining visualization is not a simple task. As Petre (1995, p. 34) has noted: "the question is not 'Is a picture worth a thousand words?', but 'Does a given picture convey the same thousand words to all viewers?' " Petre presents the concept of secondary cues, which provide additional information about visualizations. Ben-Ari (2001) claimed that graphical and textual descriptions have to be synchronized, because deciding which issues of the problem are relevant is a major problem for novice programmers. Naps et al. (2002) state that visualizations appear to be useful only if they can engage the learner into a learning session.

*Jeliot 3* (Figure 1) is a tool used in tracing the execution of Java programs. As the execution advances step by step, the evaluations of expressions are visualized with graphical symbols. Jeliot 3 is designed mainly to support the learning process of novice programmers. Kannusmäki et al. (2004) evaluated Jeliot 3 with qualitative methods and pointed out that only students without any previous programming skills were willing to use it. However, Jeliot 3 improved the novices' skills of perceiving if-statements and loops, understanding objects, and tracing errors from program code.

*JIVE* (Gestwicki & Jayaraman 2002) is a program visualization tool that in addition to code highlighting visualizes object structure and the calling sequence of methods. According to Gestwicki and Jayaraman, JIVE has proved to be a practical tool for program visualization and debugging.

*BlueJ* is an example of a *static* program visualization tool (Kölling et al. 2003). Unlike *dynamic* visualization tools such as Jeliot 3, JIVE and VILLE, static tools don't visualize program execution step by step, but instead focus on visualizing program structure and the relations between program components. BlueJ has a class view showing relations between classes and an object dock containing all initialized objects. According to Kölling et al. (2003), BlueJ is well suited to teaching programming with an objects-first approach.

Over the past few decades, many visualization and animation based applications have been developed, including *JavaVis* (Oechsle & Schmitt 2002) which uses object and sequence diagrams as visualizations, *ALVIS LIVE!*, based on the WYSIWYC (What You See Is What You Code) model and direct manipulation of program structures (Hundhausen & Brown 2007), and *Raptor* (Carlisle et al. 2005), a programming environment that uses dataflow diagrams for visualization. *JHAVE* (Grissom et al. 2003), *BALSA-II* (Brown 1988), *ZEUS* (Brown 1991), *XTANGO* (Stasko 1992) and *TRAKLA2* (Malmi et al. 2004) are algorithm animation systems, focusing on visualizing data structures and algorithms. In recent studies (Grissom et al. 2003, Laakso et al. 2005a, Laakso et al. 2005b) algorithm animation systems have been successfully applied to teaching data structures and algorithms.

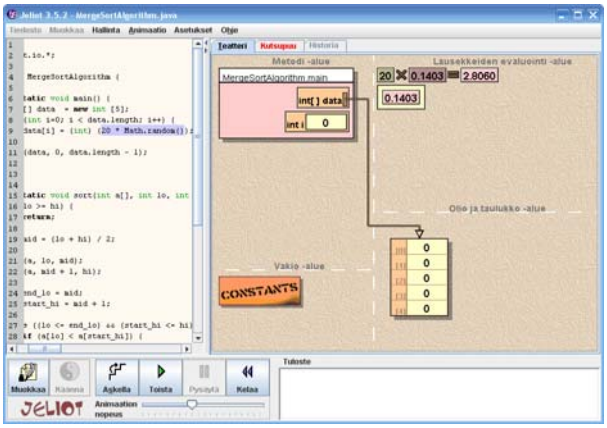


**Figure 1: User interface of Jeliot 3**

In conclusion, the tools most related to VILLE are Jeliot 3 and JIVE, which have the same basic purpose and several common features. However, remarkable differences still exist in the abstraction level of visualization. The features of these three tools are compared in detail in section 4.

## 3 The VILLE tool

VILLE is a program visualization tool, which can be used to create and edit programming examples and to observe events in the examples during their execution. Its main purpose is to support the learning process of novice programmers. Teacher can add programming examples to VILLE and then visualize their execution in lectures or over the web.

### 3.1 Key features

In this section we present VILLE's key features in four categories: level of abstraction, user interaction, tracing execution and customization. The categories reflect the main functions of features in this tool.

#### 3.1.1 Level of abstraction

**Language-independency**. One of the most important aspects of VILLE is the ability to view programming examples in several different programming languages. When observing program execution in different languages, a user can discover similarities in their basic functionalities. It is far more important for the novice programmer to learn how different programming concepts actually work than to focus on the syntactical issues of a specific language. We call this *the programming language independency paradigm*.

**Defining and adding new languages**. As built-in, VILLE supports Java, pseudocode and C++. The pseudocode's definition can be altered to suit a teacher's needs. It is also possible to define and add new programming languages to further extend the language support.

**The parallel view.** The program code execution can be viewed simultaneously in two different programming languages. This way the user can see how the execution

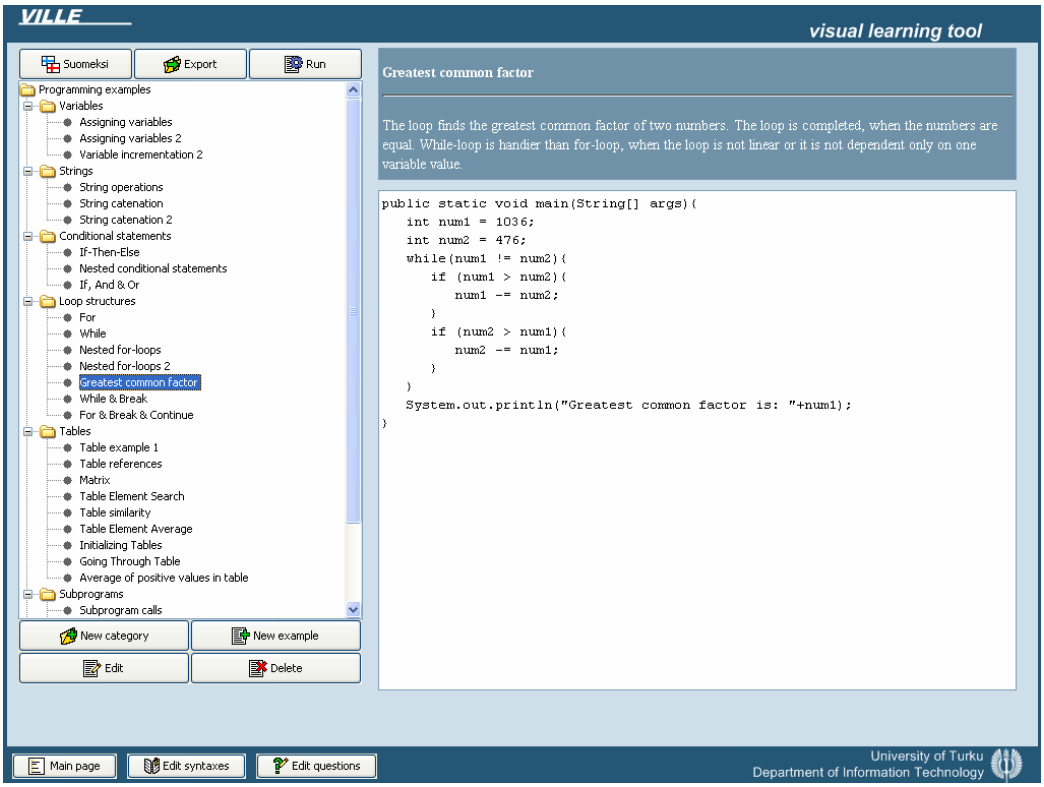


**Figure 2: Main view of VILLE**

progresses similarly regardless of syntactical differences between the languages.

**Role information**. The role information of variables is integrated into the code line explanation. According to Sajaniemi and Kuittinen (2003), the role information of variables helps learning and enhances understanding of the program.

### 3.1.2 User interaction

**Code editing**. Besides the example creation and editing view, the program code can also be edited in the visualization view, allowing users to trace the effects of changes in execution and visualization. The user's edits are not saved in the original example.

**Pop-up questions**. With the built-in editor the teacher can create multiple-choice questions and set them to trigger at certain states of the program execution.

**Flexible control of the visualization both forwards and backwards**. The user can move one step at a time, both forwards and backwards, in the execution of a program. Examples can also be run automatically with adjustable speed. Moving backwards in the program execution isn't usually possible in similar applications (e.g. Jeliot 3). Additionally, VILLE has an execution slider with which the user can progress to any state of the program execution.

### 3.1.3 Tracing execution

**Call stack**. The progress of the program execution 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 on the call stack. The window remains on the stack until the method is finished. When the execution returns to the caller, the return value is shown on top of the stack. The Call stack is especially useful in teaching *recursion*.

**Code line explanation**. Every code line has an automatically generated explanation, in which all the program events on the line are clearly explained. Furthermore, all possible outputs and variable states are shown. Code line explanation is a not a feature in most similar applications.

**Visualization row by row**. Progress of the program execution is visualized by highlighting rows in the code. In addition to highlighting the program row under execution, VILLE also highlights the previously executed row with a different colour. This makes the following of the program execution easier.

**Breakpoints**. The user can set breakpoints in program code lines and move between them, both forwards and backwards. This functionality enables debug-based control and observation of the program execution. Backward tracing between breakpoints is not a standard feature in program code debuggers.

### 3.1.4 Customization

**Example collection**. VILLE contains a predefined set of programming examples grouped into categories based on their subject. A user can create new categories and examples or edit the predefined ones. By creating and editing examples, the teacher can illustrate topics he thinks are essential in his programming courses.

**Publish examples**. With the export feature VILLE's examples can be saved to an example collection. The

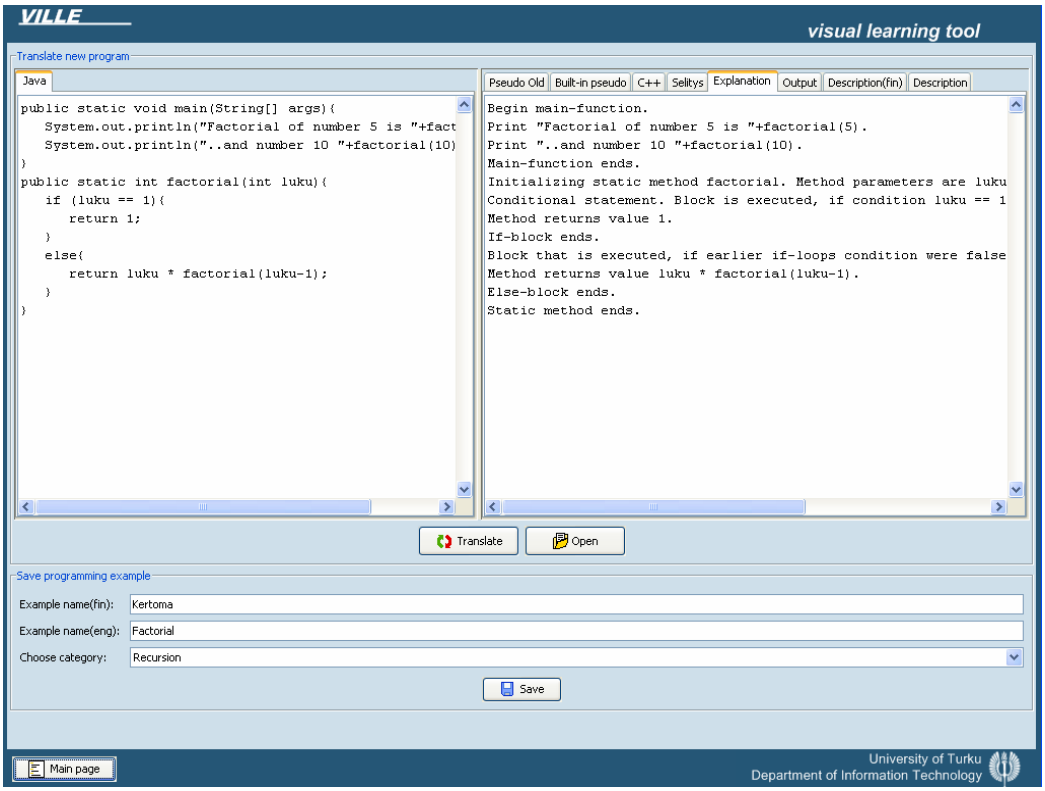


**Figure 3: Creation and editing view of programming examples**

example collection contains a version of VILLE with example creation and modification functions disabled; however, runtime modification is still enabled. The export feature can be used to publish a course's programming examples on the web for the students to use.

## 3.2 User interface of VILLE

VILLE's user interface consists of five separate views: the main view, the example creation and editing view, the visualization view, the syntax editor and the question editor.

### 3.2.1 Main view

When the application starts, the main view is loaded. On the left side of the view (Figure 2) are the programming example tree and buttons for controlling the application. Users can modify examples with the buttons below. The buttons above the examples can be used to change the language of the application between Finnish and English, to export the examples to an example collection, and to move to the execution of the chosen example. The right side of the view displays the description and code listing of a chosen example.

### 3.2.2 Example creation and editing view

In the creation and editing view (Figure 3) a user can add Java program code to the left text area; when the translate button is pressed, VILLE creates pseudocode and C++ translations (and, of course, translations to all the languages defined) and automatically generates explanations for each program line. The user can also write a general description for the programming example.

The translation of the program code is done with syntax definitions. There is a syntax definition for each programming language and also for the Finnish and English explanations. During the translation of the program, each code line is looked up from the Java syntax by using keywords, and then translated to other languages using the equivalent line in their syntax definitions. Thus, each language added to VILLE should define all the equivalent syntactical properties featured in VILLE's subset of Java syntax.

The events of a program code are solved by going through the program in its execution order and saving an execution event for each command. The events are used in the visualization view to control the visualization of the program execution.

The translation and execution tracing of programs is now possible only with Java. We are planning to add an option for translating code from the other defined languages in the near future. That will require a program component that parses the data stored in the variables of non-typed languages. After this the non-typed languages can be translated to Java, which can then be used in tracing the program execution events.

VILLE supports all the Java syntax necessary to teach introductory programming courses. It can handle the basic variable types, the main features of the String class, conditional statements, loop structures, tables and matrixes, methods, functions and records. With these programming concepts, the basic functionalities of programming can be well illustrated.

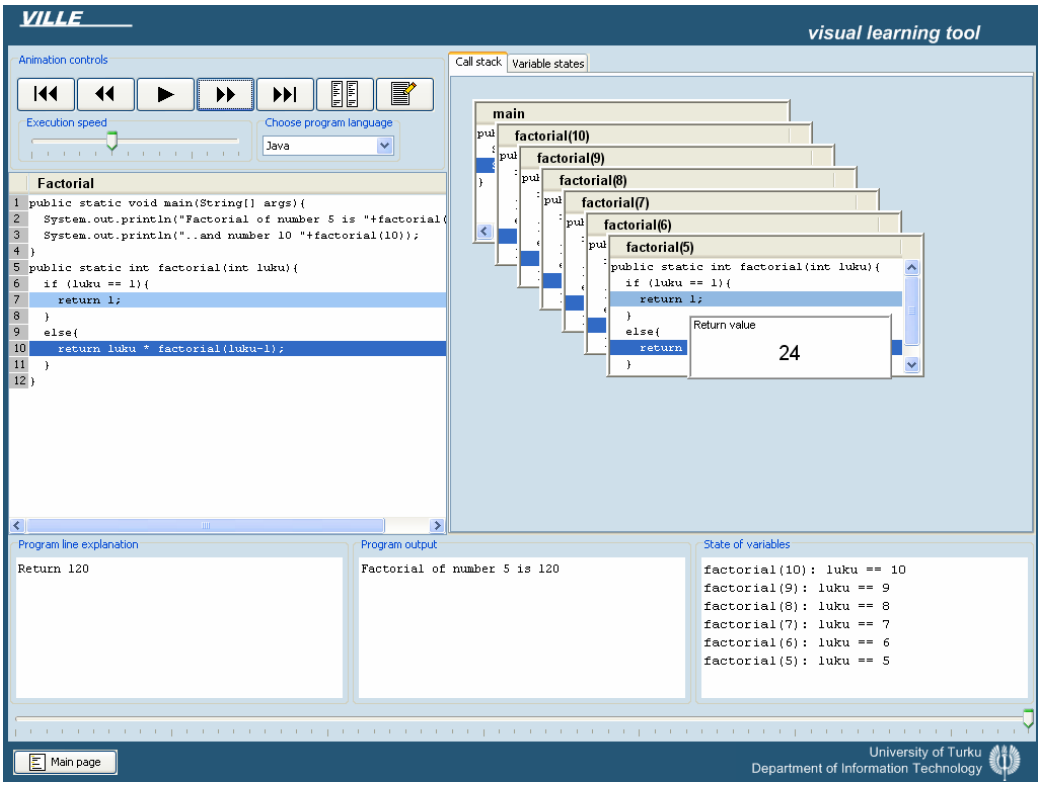


**Figure 4: Visualization view of VILLE in call stack mode**

### 3.2.3 Visualization view

In the visualization view (Figure 4) users can follow the execution of the programming examples. The control buttons for the visualization and the code listing of the programming example are located on the left side of the view. With the controls a user can start automatic program execution or alternatively move one step at a time either forwards or backwards in the program. The user can also add breakpoints to any code line and move between the breakpoints with controls similar to debuggers.

The control area can also be used to change the program code language to any language defined, even during the execution. On the right side of the view lies the call stack,

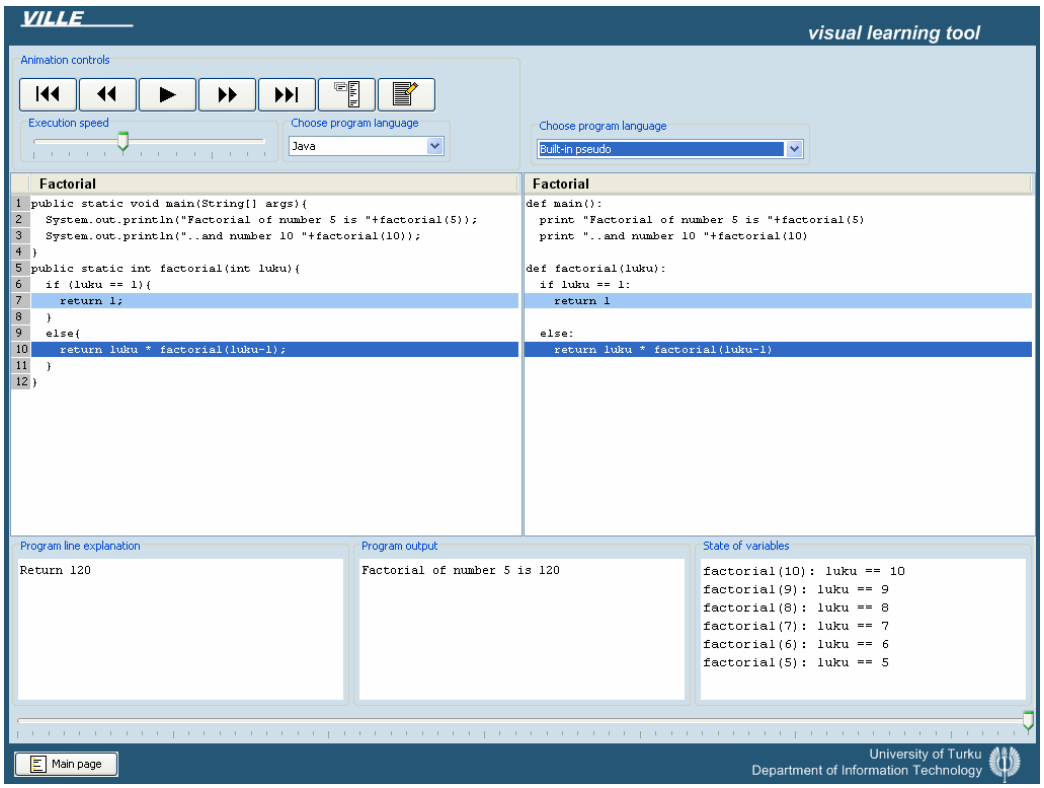


**Figure 5: Visualization view of VILLE in parallel mode**

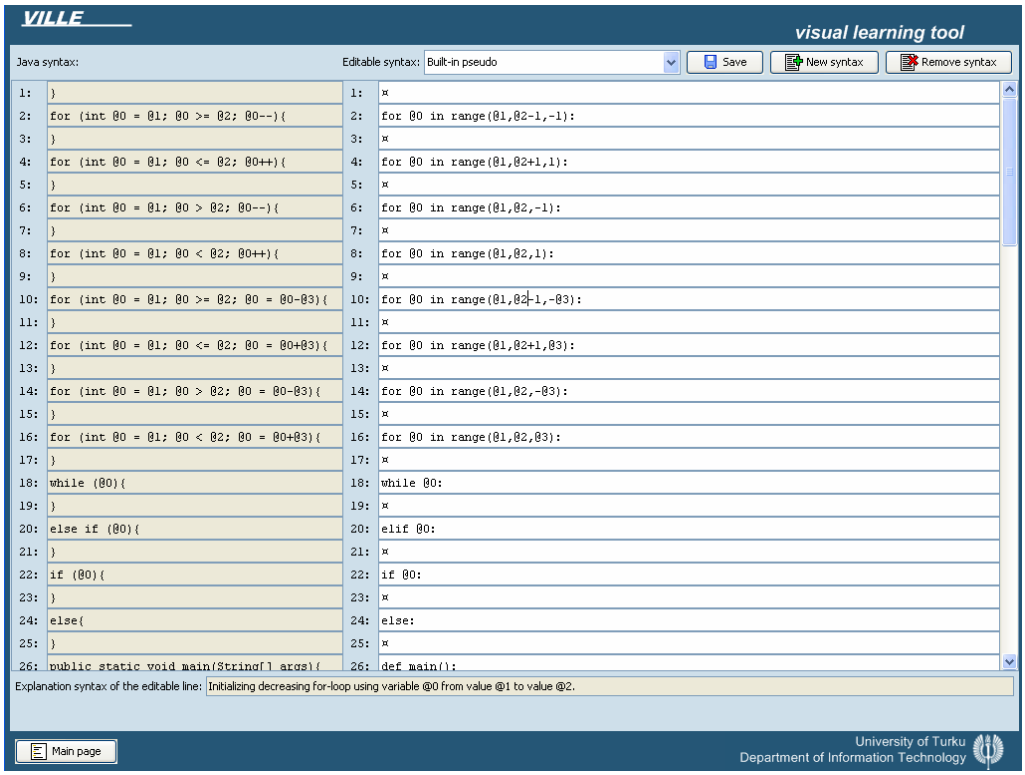


**Figure 6: Syntax editor view of VILLE**

on which the method calls are viewed in their own frames. The fields at the bottom of the view display the changes to program states, and the slider beneath those can be used to move around in the program execution.

The program execution in the visualization view can also be followed in so-called parallel mode (Figure 5), in which the program code is viewed in two selectable languages simultaneously; this way the syntax and the execution of the selected languages can be effectively compared.

### 3.2.4 Syntax editor

With the syntax editor (Figure 6) the teacher can add new programming languages to the system by defining their syntactical properties. The editor displays Java syntax lines on the left side. On the right side of the view, the user can select a syntax to modify or create completely new syntaxes. By comparing the Java syntax lines with matching lines in the modifiable syntax, the user can create new syntax lines understandable to VILLE.

### 3.2.5 Question editor

In the question editor view (Figure 7) a user can create multiple-choice questions and set them to trigger on selected code lines of a program. On the left side of the view the user can execute the program to a code line with controls similar to the visualization view, and then attach a multiple choice question to the code line. On the right side of the view the user can type in the question and the answer choices, select the choice count, and specify the right answer. All the created questions are displayed in the bottom right corner of the view.

## 4 Discussion

To enhance the learning process of novice programmers, the primary goal of VILLE has been to provide a higher level of abstraction by emphasizing the programming language independency paradigm. From the learner's point of view it is much more important to understand the principles behind basic programming concepts, such as loop control structures, regardless of the programming language. Thus, the use of a pseudo-language with less syntactical baggage than most actual programming languages is recommended for basic programming courses. With VILLE the teacher can define his own pseudo-language and then visualize program execution and its effects on the states of variables and the program output. However, because the program interpretation in VILLE is done with Java, the defined pseudo-languages should have corresponding program structures. The concept of the roles of variables gives a higher-level insight to programs, independent of programming paradigm, based on their variable behaviour. VILLE automatically generates a description with attached variable role information for every code line. This aids the interpretation of program execution as it acts as a secondary cue, aiding students in understanding the relations between programming concepts and program structures, which is essential in the process of learning to program.

Naps et al. (2002) have specified an engagement taxonomy that defines six different forms of learner engagement with visualization technology: 1) *no viewing*, 2) *viewing*, 3) *responding*, 4) *changing*, 5) *constructing* and 6) *presenting*. VILLE's feature set covers all these levels, except of course *no viewing*, which means that there is no visualization technology in use, and *changing*,

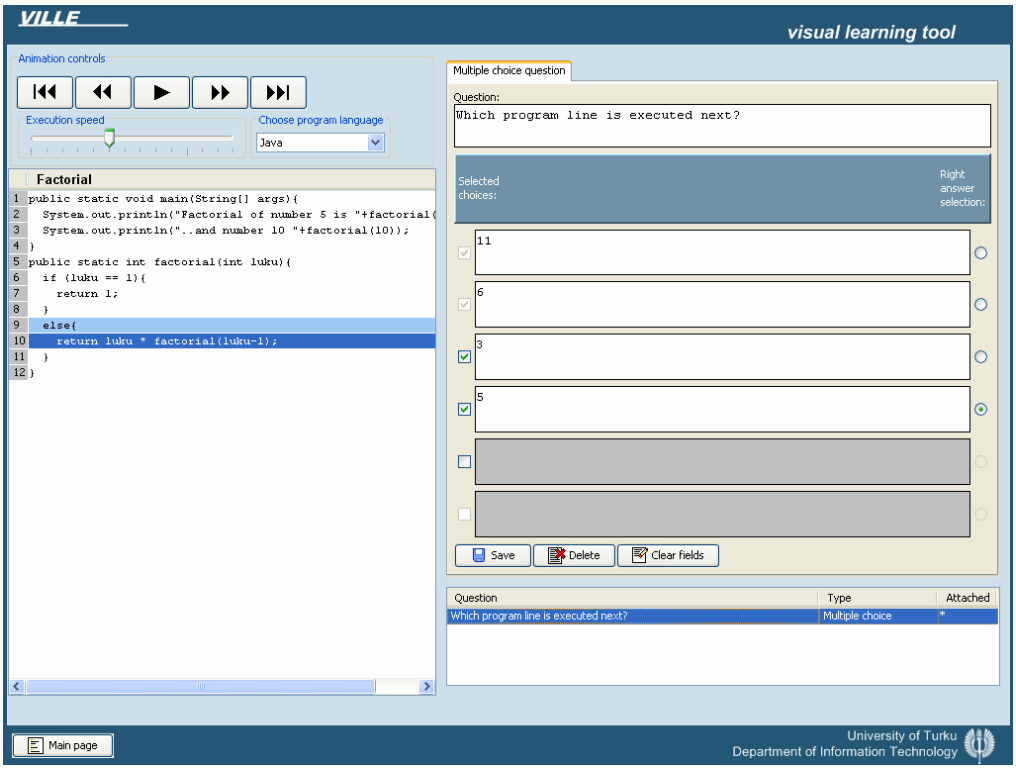


**Figure 7: Question editor view of VILLE**

which means that the system asks students for input to affect the execution of a program (this can, however, be achieved by altering the variable values in the program code). The majority of VILLE's features belong to the viewing category. Pop-up questions in the learner's perspective belong to the responding category. Students editing code in the visualization view and teachers creating new examples are clearly *constructing* visualizations, and one can engage in *presenting* just by demonstrating examples with VILLE to others.

To summarize, VILLE supports learning programming independent of the programming language. It offers customization features such as language and example creation, and provides interactivity by way of pop-up questions as well as interactive code editing to activate and engage the learner.

## 4.1 VILLE vs. Jeliot 3

VILLE and Jeliot 3 are applications that can trace step by step a program code execution, but there are some differences between these two novel tools.

From the language perspective, Jeliot 3 supports only Java, while VILLE supports Java, C++ and a user-definable pseudo-language, and the language support is easily extensible. Moreover, a user can view the selected example simultaneously in parallel with two different programming languages and compare their syntaxes. This way we can emphasize the language independency paradigm which aids the process of changing from one programming language to other.

The controls are very similar in both applications, but VILLE makes it possible to step backwards in execution and to progress to any point of the execution directly with an execution slider. The absence of backward tracing is often frustrating when executing programs.

Jeliot 3 uses graphical symbols to visualize changes in variable states, and the execution of a single statement is presented with more detail than in VILLE, which presents variable states in a textual form. Both tools highlight the code line under execution, but VILLE also highlights the previously executed line to help the tracing of the execution.

VILLE automatically generates a description line for every executed program code line. The description includes the role information of variables and dynamic information about variable states. This helps students to interpret events in the executed code lines.

Jeliot 3 supports asking the users for input. This is not possible in VILLE. However, with VILLE, students can be asked questions during program execution, which is not possible in Jeliot 3.

VILLE includes predefined examples that can be used directly through the user interface. These examples can also be published on the internet as an example collection. This feature is not found in Jeliot 3.

Table 1 presents a comparison between VILLE and Jeliot 3. We have also included JIVE in the comparison, because it's quite similar to Jeliot 3.

| | VILLE | Jeliot 3 | JIVE |
|---|---|---|---|
| **General** | | | |
| Supported languages | Java, pseudocode and C++ | Java | Java |
| Editable syntaxes | yes | no | no |
| Define new languages | yes | no | no |
| Examples | various built-in with a description; possible to add new ones | some included as files; possible to save new examples | possible to add new ones |
| Publish examples | yes | no | no |
| **Controls** | | | |
| Continuous running | yes | yes | yes |
| Adjustable speed | yes | yes | no |
| Reverse running | no | no | yes |
| Step forwards | yes | yes | yes |
| Step backwards | yes | no | yes |
| **Visualization** | | | |
| Call stack | yes | yes | yes |
| Program line explanation | yes | no | no |
| Graphical presentation of algorithm | no | no | yes |
| Variable values | yes | yes | yes |
| Role information of variables | yes | no | no |
| Program output | yes | yes | yes |
| Expression evaluation | verbal | graphical presentation | no |
| Program code viewed in | selectable language; alternative view with two languages | Java | Java |
| **Interaction with user** | | | |
| Editable programs in visualization state | yes | yes | no |
| 'Stop-and-think' questions | with pop-ups | no | yes |
| Ask input from user | no | yes | no |
| **Technical implementation** | | | |
| Implementation language | Java | Java | Java |
| Compiles examples with | built-in compiler | DynamicJava | existing JVM |
| Data model | XML | ASCII file | Java bytecode |

**Table 1: comparison between VILLE, Jeliot 3 and JIVE**

## 5 Conclusions

Learning to program is a challenging task, and a major step towards better learning is to go beyond syntactic features to understand the basic programming concepts. With this programming language independency paradigm, the similarities between the basic programming concepts in all imperative programming languages can be demonstrated, both syntactically and semantically. Furthermore, the understanding of the language independency principle should aid in adapting new programming languages and in changing from one language to another.

In the future, VILLE is going to be evaluated on the first programming courses at University of Turku. In addition to the learning performance the evaluation will focus on student engagement and the viability of VILLE's features.

In conclusion, VILLE promises an amendment to introductory programming courses by offering a chance to look at fundamental issues in an abstract way, and by allowing the teacher to create and use a programming language of his own.

## 6 Acknowledgment

This work was partially supported by Academy of Finland, project 121396, Automatic Assessment Technologies for Free Text and Programming Assignments.

## 7 References

Ben-Ari, M. (2001). Program Visualization in Theory and Practice. *Informatik/Informatique* **2**:8-11.

Brown, M.H. (1988). Exploring Algorithms Using Balsa II. *IEEE Computer*, **21**(5):14-36.

Brown, M.H. (1991). Zeus: A System for Algorithm Animation and Multi-View Editing. *In the Proceedings of IEEE Workshop on Visual Languages*, 4-9. New York: IEEE Computer Society Press.

Boada I., Soler J., Prados F. and Poch J. (2004). A Teaching/Learning Support Tool for Introductory Programming Courses. *In the Proceedings of the Fifth International Conference on Information Technology*

*Based Higher Education and Training. ITHET 2004*, 604-609.

Carlisle, M.C., Wilson, T.A., Humphries, J.W. and Hadfield, S.M. (2005). RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving. *In the Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA, 176-180.

Garner, S. (2006). The Development, Use and Evaluation of a Program Design Tool in the Learning and Teaching of Software Development. *Issues in Informing Science and Information Technology*, **3:**253-260.

Gestwicki, P. and Jayaraman, B. (2002). Interactive visualization of Java programs. *In Proceedings of Symposia on Human Centric Computing Languages and Environments*, 226-235.

Grandell, L., Peltomäki, M., Back, R.-J. and Salakoski, T. (2006). Why Complicate Things? Introducing Programming in High School Using Python. *In Proceedings of the 8th Australasian Conference on Computing Education*, Hobart, Australia, **52:**71-80.

Grissom, S., McNally, M. and Naps, T. (2003). Algorithm Visualization in CS Education: Comparing Levels of Student Engagement. *In Proceedings of the ACM Symposium on Software Visualization*, San Diego, California, s. 87–94.

Hundhausen, C.D. and Brown, J.L. (2007). What You See Is What You Code: A 'Live' Algorithm Development and Visualization Environment for Novice Learners. *Journal of Visual Languages and Computing*, **18**(1):22-47.

Hundhausen, C.D., Douglas, S.A. and Stasko, J.D. (2002). A Meta-study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing* **13:**259-290.

Kannusmäki, O., Moreno, A., Myller, N. and Sutinen, E. (2004). What a Novice Wants: Students Using Program Visualization in Distance Programming Course. *In Proceedings of the Third Program Visualization Workshop (PVW'04)*, Warwick, UK, 126-133.

Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, **13**(4).

Laakso, M.-J., Salakoski, T., Grandell, L., Qiu, X., Korhonen, A. and Malmi, L. (2005a). Multi-Perspective Study of Novice Learners Adopting the Visual Algorithm Simulation Exercise System TRAKLA2. *Informatics in Education*, **4**(1):49-68.

Laakso, M.-J., Salakoski, T. and Korhonen, A. (2005b). The Feasibility of Automatic Assessment and Feedback. *In Proceedings of Cognition and Exploratory Learning in Digital Age (CELDA 2005)*. IEEE Technical Committee on Learning Technology and Japanese Society of Information and Systems in Education. Porto, Portugal, 113-122.

Lister, R., Adams, S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L. (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGCSE Bulletin*, **36**(4):119-150.

Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppälä, O. and Silvasti, P. (2004). Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. *Informatics in Education*, **3**(2):267-288.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. (2001). A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. *ACM SIGCSE Bulletin*, **33**(4):125-140.

Naps, T.L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Velázquez-Iturbide, J. Á. (2002). Exploring the Role of Visualization and Engagement in Computer Science Education. *In Working group reports from ITiCSE on Innovation and Technology in Computer Science Education*, **35**(2):131-152.

Oechsle, R. and Schmitt, T. (2002). JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). *In Diehl, S. (Ed.), Software Visualization. vol.2269 of Lecture Notes in Computer Science*. Springer-Verlag, 176-190.

Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, **38**(6):33-44.

Sajaniemi J. (2002). PlanAni - A System for Visualizing Roles of Variables to Novice Programmers. *University of Joensuu, Department of Computer Science, Technical Report*, Series A, Report A-2002-4.

Sajaniemi, J. and Kuittinen, M. (2003). Program Animation Based on the Roles of Variables. *In Proceedings of the 2003 ACM Symposium on Software Visualization*, San Diego, California, 7-ff.

Stasko, J. (1992). Animating Algorithms with XTANGO. *ACM SIGACT News*, **23**(2):67-71.

Stern, L., Søndergaard, H. and Naish, L. (1999). A Strategy for Managing Content Complexity in Algorithm Animation. *In Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, Cracow, Poland, 127-130.

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