VisualJVM: A Visual Tool for Teaching Java Technology

Pedro Pablo Garrido Abenza, Angel Grediaga Olivo, and Bernardo Ledesma Latorre

Reference

- **ASCII Text**

Garrido Abenza, Pedro Pablo; Grediaga Olivo, Angel; Ledesma Latorre, Bernardo; "*VisualJVM: a visual tool for teaching Java technology*", Education, IEEE Transactions on, Vol.51, no.1, pp.86-92, Feb. 2008

- **BibTex**

```
@ARTICLE{visualjvm2008,
   author = {Garrido Abenza, P.~Pablo and Grediaga Olivo, Angel and
Ledesma Latorre, Bernardo},
    title = "{VisualJVM: a visual tool for teaching Java technology}",
  journal = {Education, IEEE Transactions on},
     year = {Feb. 2008},
   volume = {51},
   number = {1},
    pages = {86-92},
      doi = {10.1109/TE.2007.906601},
     ISSN = {0018-9359}
}
```

Abstract

This paper presents a laboratory session of an Advanced Programming course to introduce students to the technology involved with the Java programming language. In this special lab session the educational software tool VisualJVM is used, providing a graphical front-end to a Java virtual machine (JVM). This tool helps students learn about JVM architecture, learn how JVM works, and consequently, to understand why a Java program is platform independent. The student reaction to this experience was very positive and the authors are planning to use the tool in other contexts.

VisualJVM: A Visual Tool for Teaching Java Technology

Pedro Pablo Garrido Abenza, Angel Grediaga Olivo, and Bernardo Ledesma Latorre

Abstract—This paper presents a laboratory session of an Advanced Programming course to introduce students to the technology involved with the Java programming language. In this special lab session the educational software tool *VisualJVM* is used, providing a graphical front-end to a Java virtual machine (JVM). This tool helps students learn about JVM architecture, learn how JVM works, and consequently, to understand why a Java program is platform independent. The student reaction to this experience was very positive and the authors are planning to use the tool in other contexts.

Index Terms—Educational technology, Java language, object-oriented programming (OOP), program interpreters, user interfaces.

I. MOTIVATION

WITH the explosion of the Internet and the demand for platform independent software, Java technology has grown in popularity. Java has a wide range of applications and target platforms, ranging from PCs and Web servers to embedded devices and smart cards. Thus, there is a strong demand for knowledgeable Java programming engineers. Therefore, Java programming courses have been introduced into many curricula. Java is not only a programming language, but also a set of specialized platforms that are also known as Java runtime environments (JRE), which are necessary to execute Java programs. JRE is composed of the Java library of classes [Java application programming interface (API) implementation] and a Java program interpreter. The interpreter, also called Java virtual machine (JVM), is a program implemented for a specific platform that executes the multiplatform sequence of bytes generated by a Java compiler (*bytecodes*). This unique feature is the basis of the platform independence of Java programs, and can be difficult to understand for many students, even those able to write Java programs [1].

The primary motivation for the development of *VisualJVM* was to provide a graphical user interface (GUI) or front-end to a JVM implementation, which had been developed from scratch by the authors for research purposes. The role of the front-end was to facilitate the debugging and testing of the extensive source code written in C. This front-end shows at any time the internal state of the interpreter in a graphic and interactive mode. So *VisualJVM* is composed of the Java interpreter and the front-end; the former works as a back-end and is able to run together with or without the front-end. However, the use of this tool revealed that *VisualJVM* was very suitable for teaching some theoretical concepts involved with the Java programming language. Because of this the front-end was improved, giving special attention to factors like usability and readability, and a set of laboratory exercises were developed and successfully integrated into an Advanced Programming course for Telecommunications Engineering degrees at Miguel Hernandez University, Elche, Spain. Whereas the authors use this tool to debug the source code of their own JVM implementation, the students use *VisualJVM* as an educational tool to learn Java technology.

This paper is focused particularly on the benefits of *VisualJVM* as an educational tool, and is structured as follows. First, Section II examines other related tools and discusses the advantages of *VisualJVM*. Section III presents the lab session, while Section IV details the first exercise developed, and describes the use of the *VisualJVM* tool. The results of the student evaluation of the effectiveness of *VisualJVM* in enhancing the learning process are discussed in Section V. Finally, Section VI consists of concluding remarks and future plans.

II. RELATED TOOLS

Existing tools, like jclasslib Bytecode Viewer and Class Viewer for Java, could help in achieving the goals of this lab session. However, these tools only show static information about Java compiled classes; they do not show the internal state of the JVM during the execution of a Java program.

There are other tools that can be used to trace the execution of Java programs. Loggers, like Java Logging API and log4j, need to modify the source code so as to collect information at execution time. In contrast, since *VisualJVM* works with .class files (like any other JVM), a user can work with Java programs even when their source code is not available. Tracers, debuggers and profiler tools have been developed using Java platform debugger architecture (JPDA), including JTracor, jLouiss, Java-TraceIt! [2], extensible Java profiler (EJP), JProfiler, or hprof. None of these tools require a recompilation of the Java source code. However, much as in integrated development environments (IDEs) such as Microsoft Visual Studio (which includes Visual J#), Borland JBuilder, VisualCafe, NetBeans, or Eclipse, they work at the source code level and they do not show the inner workings of the JVM.

Javy [3] is a virtual environment that simulates the JVM, and allows users to learn the JVM structure and the Java language

P. P. Garrido Abenza is with the Department of Physics and Computer Architecture, Miguel Hernandez University, 03202 Elche, Spain (e-mail: pgarrido@umh.es).

A. Grediaga Olivo and B. Ledesma Latorre are with the Department of Computing and Information Technology, University of Alicante, 03690 San Vicente del Raspeig, Alicante, Spain (e-mail: angel.grediaga@ua.es; bernardo.ledesma@ua.es).

compilation. A Java class visualization tool has also been developed [4] that allows interaction with a visualization of any Java class through the instantiation of objects. Several applets that simulate JVM executing a few *bytecode* instructions are included in [5]. Unlike these tools, *VisualJVM* is not a simulator; when a Java program is loaded by the Java interpreter that program really is executed. In other words, *VisualJVM* has many of the benefits but none of the limitations of a simulation tool [6], [7].

Moreover, nearly all the previously mentioned tools have normally been developed for only one target platform, whereas *VisualJVM* is highly platform-independent because JVM is written in standard C and the front-end is based on the Qt 4.0 Opensource toolkit [8]. Such a front-end cannot be found in common JVM implementations (e.g., Sun Microsystems' JDK [9], IBM Jikes research virtual machine (RVM) [10], [11], Kaffe [12], Joeq [13], etc.). Specifically, the Jikes RVM is used in many teaching courses such as Advanced Compiler Techniques, Advanced Object-Oriented Programming, Compiler Design and Optimization, etc. However, to the knowledge of the authors, *VisualJVM* is the first JVM including a graphical interface that visualizes the internal state and the behavior of a real JVM. Thus, *VisualJVM* is very suitable for teaching purposes.

III. DESIGN OF THE LAB SESSION

In this section the design of the lab session is described. The context of the course and prerequisites are first specified, and then the learning objectives are described. Finally, the procedure to be followed and the necessary lab infrastructure are detailed.

A. Context of Course

The Advanced Programming course aims to teach third-year students the Java programming language, and to introduce them to good computer program design principles using the object-oriented programming (OOP) methodology. The course consists of a one-hour lecture and a two-hour lab session per week. Thus, this one-semester course (15 weeks) features an important practical component. Even though there are no official course prerequisites, the students are expected to have basic programming experience in either C or assembler; this background is acquired through several preceding courses. Some familiarity with abstract data types (ADT) (e.g., lists, stacks, or queues), although not required, would be helpful.

B. Learning Objectives

The two-hour lab session is not intended to give students a complete knowledge of JVM; this experience is, in fact, a unique opportunity for them to become familiar with Java technology. The learning objectives (LOs) are divided into two parts.

1) *Outside the JVM*: The goal of this first part is to reinforce theoretical concepts about Java technology.
   - Objective #1 (LO#1): To distinguish between an interpreted and a compiled language. Students will be able to differentiate between a compiler and an interpreter, in other words, to distinguish CPU-specific machine code from platform-independent *bytecodes*.
   - Objective #2 (LO#2): To understand why a Java program is platform independent. Students will prove to themselves that, unlike the C programming language, a Java program can be executed under two different operating systems (OSs) without recompiling the source code.
   - Objective #3 (LO#3): To differentiate the concepts of JVM, JRE, and the Java development kit (JDK), which are often confused by students.

2) *Inside the JVM*: Part 2 addresses issues related to the internal structure of the JVM. The goal is for students to get a general overview of how the Java programming language is implemented, since this is an unknown area for them.
   - Objective #4 (LO#4): To interpret the structure of the class file format, which defines the representation of a Java class. Students will be able to recognize binary files representing a Java class.
   - Objective #5 (LO#5): To become familiar with the JVM instruction set (*opcodes*). The JVM has a complex instruction set computer (CISC) architecture because JVM supports about 200 instructions, in contrast to the reduced instruction set computer (RISC), which supports fewer instructions. Java class files are, therefore, very compact. Students will be able to know the format of some of the *opcodes* and the operations they perform.
   - Objective #6 (LO#6): To identify and recognize the JVM architecture. Students will acquire an understanding of the runtime data areas used during the execution of a Java program (namely: frames, JVM stacks, the heap, or the runtime constant-pool).
   - Objective #7 (LO#7): To describe some of the processes involved with the execution of Java programs, such as the inner loop of the JVM, method invocations, passing arguments to a method, returning a value from a method, use of the operand stack of a frame (e.g., doing arithmetic operations), instancing objects, etc. Students will also verify that JVM is stack-oriented, meaning that, for example, the arithmetic instructions take the necessary operands from the operand stack and the results of the operation are pushed onto the operand stack.
   - Objective #8 (LO#8): To understand how the high-level Java source code is compiled to the low-level *bytecodes*. This LO would be useful for compiler writers and for compiler construction courses, but was not addressed in the lab session presented in this paper.

A different instrument was used to measure each of these parts, which will be discussed in Section V. Once the LOs were defined, the procedure and a set of exercises were designed to enable the students to attain them.

C. Procedure

The two-hour lab session presented in this paper takes place in the last week of the semester, when all the important features of Java technology have already been explained in lectures. Despite the fact that most of the students have a good knowledge of the Java programming language, their comments reveal that JVM is a *black box* for them, since the internal structure is an

obscure concept, and they have only a limited belief in the theoretical explanation given of the Java technology, specifically, the concept of Java platform independence.

In addition, to prepare the students for the new lab session, a brief slide presentation that introduced several concepts about the JVM architecture was given in a lecture during the preceding week. A first example, the "Hello World" program, was addressed in a step-by-step manner in this presentation. At the end of this introductory class the preparatory reading of the *VisualJVM* manual was assigned to the students as homework.

D. Lab Infrastructure

These four elements are required to accomplish this lab session: 1) the lab room must include one PC per student, able to boot two desktop OSs, including Windows and Linux (alternatively, making use of a Linux Server or Linux Live-CD is particularly useful for both students and educators alike); 2) the JDK, available for both platforms [9], which includes the JRE and provides several command line tools for the development of Java programs, including the Java compiler (javac), and a disassembler of Java classes (javap); 3) the JVM specification [14]; and 4) the *VisualJVM* binary and documentation for the specific platform. All the course material, as well as several other exercises, is available for downloading from the *VisualJVM* website: http://obelix.umh.es/pa/visualjvm.

IV. LAB EXERCISES

A lab session is presented in this section in which the students examine the execution of several ready-made Java programs. The first one is the "HelloWorld" program that just writes a "Hello World!!" message on the computer screen. This short program is usually the first to be attempted when learning any programming language [15].

```
//HelloWorld.java
class HelloWorld (
    public static void main(String[] args) (
        System.out.println("Hello, World!!");
    )
)
```

This HelloWorld.java file is compiled into a *bytecode* sequence and written to a HelloWorld.class file by using the Java compiler included in the J2SE [9] (javac command). Then, in the first exercise, several other command line tools included in the J2SE are used in order to execute the program, whereas in the second exercise, only *VisualJVM* is used. Both exercises are detailed below.

A. Lab Exercise: HelloWorld—Without VisualJVM

The student executes this program using the Java runtime interpreter (java command) as follows:

```
C:\java HelloWorld
Hello, World!!
```

Although the Java interpreter runs the program *opcode*-by-*opcode*, this process is not observable by the students. In order to understand how the JVM interpreting the *bytecodes* works, it is necessary to look into the HelloWorld.class file.

Identifying the class file is easy using any binary file editor and looking at the first four bytes of the file, which must be the hexadecimal magic number 0xCAFEBABE. However, the class file format is not simple and it is very difficult to analyze the rest of the bytes by hand, since they contain the version number, access flags, the runtime constant-pool, fields and methods declared by the class, generated *opcodes* for each method, etc. So to get these data more simply, instead of analyzing the binary files by using a binary file editor, the javap command tool (which is a disassembler of class files that parses them and prints out all the previously mentioned data) was used as follows:

```
C:\javap -c HelloWorld
Compiled from "HelloWorld.java"
class HelloWorld extends java.lang.Object{
HelloWorld();
Code:
    0: aload_0
    1: invokespecial #1; //Object."<init>"
    4: return
public static void main(java.lang.String[]);
Code:
    0: getstatic #2;    //Field
    out:LPrintStream;
    3: ldc #3;        //String Hello, World!!
    5: invokevirtual #4; //PrintStream.println
    8: return
}
```

At first glance the "HelloWorld" program seems very simple. However, following the execution is a complex and time-consuming task if done manually, due to the great amount of nested method invocations that are made from the println method. When a method is invoked (with either invokespecial or invokevirtual *opcodes*), the following steps are necessary.

1) Extract the corresponding class file (e.g., Object.class) from the ZIP or JAR files containing the Java API.
2) Use the javap tool to disassemble the class as shown previously, and look for the invoked method.
3) Follow the execution flow *opcode*-by-*opcode* by hand. If an invoke *opcode* is found, then go to step 1.

Because the JVM instruction set has about 200 *opcodes*, the students need to consult the JVM Specification [14] to learn which operation each *opcode* performs. In order to avoid wasting time, educators suggested skipping this exercise and going on with the next one, once the student realizes that putting this process into practice is not feasible.

B. Lab Exercise: HelloWorld—With VisualJVM

To illustrate the usefulness of *VisualJVM*, the "HelloWorld" program will again be examined in this section. To achieve
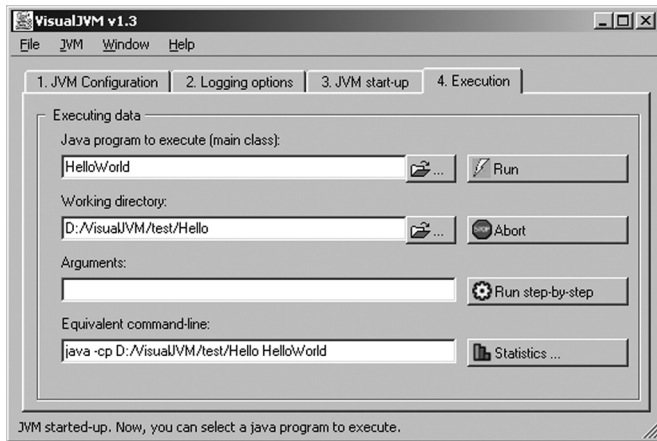
Fig. 1. Screenshot of the main window of *VisualJVM*.

Fig. 2. Screenshot of the class browser dialog.

LO#1 and LO#2, the exercise is first done using Windows OSs, and then repeated using Linux OSs. In both cases the same HelloWorld.class file is used (i.e., without recompiling the source code), so that the student realizes that an interpreted program is different from a compiled one.

Within *VisualJVM* (Fig. 1), the JVM must be configured and started up before a Java program can be executed. *VisualJVM* is not a full JRE, but only a JVM; in other words *VisualJVM* does not include the Java API classes. Therefore, the user must provide the exact location of the Java API classes, which differs according to the platform, library version, and developer (e.g., Sun Microsystems' JDK or GNU Classpath). This configuration can be set by either setting the VISUALJVM_JAVAPATH environment variable before calling *VisualJVM*, or by selecting the path from the application. After that the student can distinguish between JVM, JRE, and JDK (LO#3).

Before executing any Java program the interpreter must be started up, that is, loading and linking of the basic Java classes. A JVM implementation may choose to use lazy resolution (i.e., to resolve each symbolic reference in a class individually, only when used) or static resolution (i.e., to resolve all symbolic references at once, when the Java interpreter starts up). *VisualJVM* allows either of the two methods to be chosen. Static resolution implies that much greater delays are involved in the JVM start-up; the execution, however, will be faster later on.

Once the interpreter has been started up, no matter which kind of resolution is chosen, the student selects the .class file containing the main class of the Java program to execute (e.g., HelloWorld), and optionally, to specify the command line arguments passed to the main method (Fig. 1). Finally, the program can be executed by clicking either the [run] or [run step-by-step] button. In the latter case the execution dialog opens as a new window, allowing the user to step through the code *opcode*-by-*opcode* (not line-by-line at the source code level) to see how the execution progresses. This dialog, which will be explained later, has been very helpful in developing the interpreter, and also very instructive because this simple window gives the student a clear understanding of how JVM works.

The *Class Browser dialog* (Fig. 2) shows all the classes loaded into the JVM in three different ways: sorted in alphabetical order, grouped by t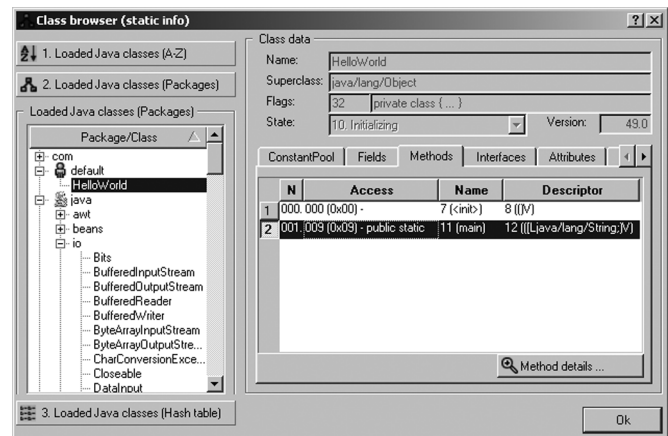he packages hierarchy, and clustered in buckets of the internal hash table of the JVM. In each case, when a class is selected, all the details are shown to the right, which helps students to reach LO#4. The information given for each class includes superclass, state (e.g., loaded, linked, resolved or initialized), visibility (e.g., public or private), the runtime constant-pool (*opcodes* refer to symbolic information in this table, which is similar to a symbol table for a conventional programming language), fields, methods, implemented interfaces, attributes, and the classes hierarchy. When an specific method is selected by double-clicking the method name in the list of methods defined for the class, the corresponding method details dialog appears, explained below. This allows the user to access the method code of any loaded class at any time more easily and faster than by using the javap tool.

The *Method details dialog*, which can be opened from either the class browser dialog (as explained previously) or from the execution dialog, displays data about a specific method, such as the disassembled code of that method and the descriptor (i.e., the parameters that the method receives and the type of the value that the method returns). The method's code is shown in table-form where each line corresponds to one instruction, including the one-byte *opcode* and associated mnemonic (e.g., $021(0 \times 15) - \text{iload}$), followed by zero to five bytes of additional arguments (e.g., "0" to specify the local variable #0), and a brief description of the operation that the instruction performs (e.g., load an "int" value from the specified local variable). This table is similar to that in the execution dialog described in (Fig. 3). This dialog is useful in familiarizing the student with the JVM instruction set, removing the need to consult the JVM Specification [14], and contributing to the attainment of LO#5.

When the user chooses to execute a program step-by-step, the *Execution dialog* appears (Fig. 3), showing the architecture of the JVM and allowing students to accomplish LO#6. The left-hand side of this dialog shows a list of all JVM threads and the private JVM stack of the selected thread (the current thread by default). A JVM stack stores frames, which are shown in a scrollable list. When a method is invoked (*current method*), a new frame is created and pushed into the stack of the current thread; this frame will be referred to as the *current frame*. When the method invocation is complete, the corresponding
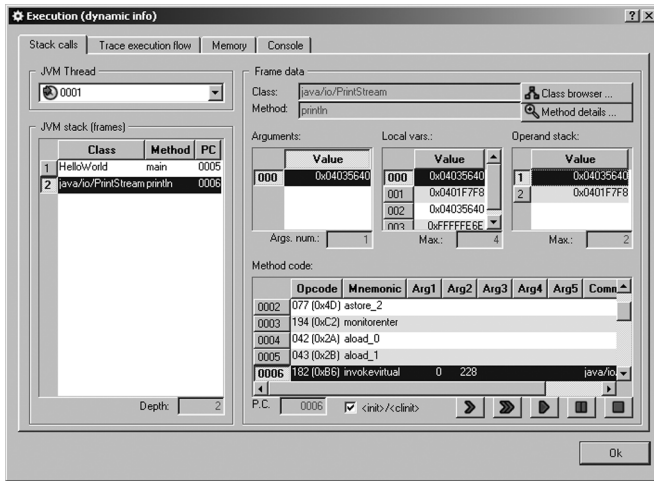
Fig. 3. Screenshot of the execution dialog.

Fig. 4. Screenshot of the statistics dialog.

frame is popped off the stack and deleted. The selected frame state is shown to the right of this dialog, including the associated method and *opcodes*, the program counter (PC) register, arguments received, local variables, operand stack, and a reference to the class of the current method (the *current class*). The highlighted code line shows the JVM instruction pointed to by the PC that will be executed next. Each time the student clicks the [step] or [step over] buttons a single *opcode* is executed and the execution stops again. Both behave identically if the current *opcode* does not contain a method call (`invoke`); otherwise, if the [step] button has been pressed, the next instruction will be the first one in the called method, but if the button pressed is [step over], the whole called method will be executed. By pressing the [run] button the execution will continue until the student, at any time, temporarily halts or permanently aborts the execution by clicking the [pause] or [stop] buttons, respectively. While execution is paused, student can examine the state of previous frames in the JVM stack, including data from their associated method, or visualize any data of any loaded class with the class browser dialog. The program execution can be resumed by pressing the [step] or [run] buttons. Thus, LO#7 can be easily achieved because at each step the student can, among other possibilities, watch the values of the local variables or the effect on the operand stack of a given *opcode* (e.g., arithmetic instructions), and understand processes like passing parameters when a method is invoked, or how a value is returned to the calling method when the called method ends. The program execution flow is recorded, allowing later analysis of the sequence of method calls made from start to finish. Finally, a summary report can be generated from statistics gathered during the program execution, showing how many times each *opcode* in the JVM Instruction Set has been executed, along with a brief description of that instruction (Fig. 4). These data could be used as a part of a performance profiler.

V. STUDENT EVALUATION AND RESULTS

Outcome data collected from written surveys used to evaluate the effectiveness of the aforementioned lab session is presented and discussed in this section. Students were surveyed using two
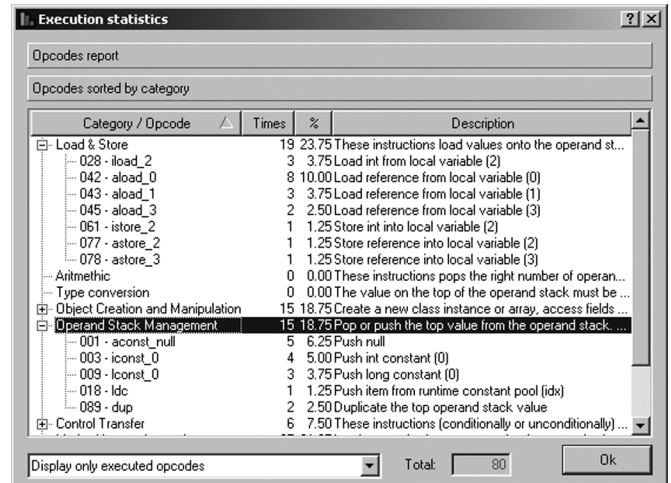
short duration paper-based surveys: 1) a posttest achievement and 2) an attitudinal survey. The following data correspond to the most recent semester, when enrollment was 69 students. Despite the fact that participation in the lab sessions of this course was optional, the rate of participation in this special session was high. There were 41 of 69 (59%) participants, whereas the mean of the regular sessions was 31 of 69 (about 45%). This participation ratio shows an increased enthusiasm on the part of the student before the session began.

A. Posttest Achievement

After doing the exercises the students were asked to take a test in order to examine their conceptual understanding of the theoretical concepts of Java technology. This test was in two parts, reflecting the two types of learning objectives defined previously. The first part, *Outside the JVM*, presented 7 multiple-choice items assessing student knowledge of the external interface of the JVM. The second part, *Inside the JVM*, comprised a set of numeric and short-answer items assessing student understanding of the JVM internal operation. In this second part, the student had to "fill in the gaps" with specific values from *VisualJVM* after executing some Java program (e.g., a local variable) or briefly explain some process.

The student responses to each item were compiled into the set of the defined learning objectives, and the mean scores are shown in Table I. As expected, the success rate on most of the learning objectives for part 1 was higher than those from part 2 because the latter was totally new for the students. Table I summarizes student performance on both parts scored between 0–10, and part 2 was more heavily weighed (60%) than part 1 (40%). According to the table the variability is low, since the SD is a little more than 1/4 of the overall mean, which is about 2/3 of the maximum possible score.

B. Attitudinal Survey

The students also answered a second questionnaire comprised of 10 items for the measurement and evaluation of the participant attitudes and satisfaction level. The instrument used for this purpose was a questionnaire based on a five-point Likert-scale,

TABLE I
POSTTEST ACHIEVEMENT; ITEM RESPONSES ($N = 41$)

Learning Objective Measured	PC
Part 1: Outside the JVM (M_1=7.11; SD_1=2.78)	
#1. Distinguishing between interpreted and compiled programs	.83
#2. Understanding why Java is platform independent	.84
#3. Differentiating the concepts of JVM, JRE, and JDK	.65
Part 2: Inside the JVM (M_2=5.89; SD_2=2.13)	
#4. Interpreting the structure of the class file format	.58
#5. Demonstrating familiarity with the JVM Instruction Set	.67
#6. Identifying and recognizing the JVM architecture	.67
#7. Being able to describe the processes involved with the execution	.50
Overall mean = M_1*0.4 + M_2*0.6 = 6.38; Overall SD = 1.77	

PC = Proportion Correct
M_i = Mean for Part i
SD_i = Standard Deviation for Part i

TABLE II
ATTITUDINAL SURVEY: FREQUENCY RESULTS ($N = 41$)

Questions/Answers prop	1	2	3	4	5	M	SD
1. Instructional content has been presented effectively	.05	.00	.17	.56	.22	3.90	0.92
2. I learned about the JVM architecture	.00	.00	.20	.46	.34	4.15	0.74
3. I enjoyed this experience	.02	.05	.39	.46	.07	3.51	0.82
4. Using the VisualJVM tool was easy	.02	.24	.27	.37	.10	3.27	1.03
5. The proposed practice activities were useful	.02	.07	.22	.54	.15	3.71	0.91
6. There was a sufficient number of exercises	.10	.10	.27	.32	.22	3.46	1.24
7. There was sufficient time to complete the exercises	.24	.24	.10	.17	.24	2.93	1.57
8. My level of involvement was high	.10	.12	.24	.29	.24	3.46	1.28
9. I would like to learn more about JVM	.00	.07	.20	.41	.32	3.98	0.92
10. This was a worthwhile learning experience	.05	.20	.10	.46	.20	3.56	1.15
Average	.06	.11	.21	.40	.21	3.59	-
Overall mean = 35.93; Overall SD = 4.96							

1—Strongly Disagree; 2—Disagree; 3—Undecided; 4—Agree;
5—Strongly Agree; M = Mean; SD = Standard Deviation

coded as shown in Table II, which summarizes the main results of the statements asked (all the sentences score on a positive scale).

Data analysis of responses: On this five-point scale, the average of these questions was above 3.59. Similarly, if the overall score can vary from 10 (totally negative attitude) to 50 (highly positive attitude), the obtained student scores were high, ranging from 25 to 49, with a mean of 35.93. Therefore, in general it can be said that the students had a favorable attitude to this experience.

The most-valued items: Nearly all the items show a very positive attitude. The most-valued items were numbers 2 and 9, which directly referred to the students' learning. Numbers 1 and 5, with a slightly lower score, indicate that the students understand the lab session materials and recognize the usefulness of the exercises, respectively.

The least-valued items: The lowest rated statement was number 7, which referred to the little time available. Realistically, the students can only conduct two exercises in this time-limited lab session (two hours), even though the time devoted to each of these is adequate to complete the work. Another lower-valued item was number 4, referring to the use of *VisualJVM*. The authors interpret these results as suggesting that the students enjoyed this experience, and that more time is necessary to learn about JVM architecture than is available within the scope of the Advanced Programming course.

Variability: From Table II the answers are quite homogenous, since overall SD (4.96) is about 1/7 of the mean (35.93).

Reliability: The reliability was estimated using a Cronbach's alpha score across all survey items, which is a measurement of the internal consistency of the survey. The obtained value of 0.58 indicates good reliability of the attitudinal survey.

C. A Comparative Study

This section presents the results of a comparative study in order to determine the impact this lab session has had on student achievement. This study used two groups: one control group from the academic year 2003–2004 ($N = 60$), which did not have the support of the *VisualJVM* tool, and one experimental test group from the academic year 2005–2006 ($N = 41$), which attended the lab session described previously (and whose data is shown in the previous sections). The instrument used was the comparison of the correct answers to specific questions in the common final exam of both groups. Only the questions relevant to the part 1 (outside the JVM) learning objectives were considered (i.e., LO#1, LO#2, and LO#3) because as mentioned previously, the part 2 (inside the JVM) learning objectives rely on new concepts explained only to the experimental group, and were not evaluated in the standard final exam.

The differences between both groups are shown in Table III. Students from the experimental group showed evidence of considerable improvement of Java technology knowledge, reaching

TABLE III
COMPARATIVE STUDY

	LO#1		LO#2		LO#3		Overall	
Group	M	SD	M	SD	M	SD	M	SD
Experim. (N=41)	.80	.20	.82	.22	.69	.25	.77	.23
Control (N=60)	.52	.23	.56	.23	.48	.21	.52	.22

M = Mean proportion of correct answers
SD = Standard Deviation

a higher score in terms of the mean percentage of correct answers on the final examination (77%) than students from the control group (52%).

VI. CONCLUSION AND FUTURE WORK

The *VisualJVM* tool, which consists of a graphical front-end and a JVM, is presented in this paper. The graphical front-end was initially designed for debugging purposes in JVM development and was then improved to be used for laboratory instruction. *VisualJVM* has been compared with other JVM implementations and related tools. The authors are not aware of any other tool that shows what is happening within the JVM during a Java program execution.

A lab session is described in detail, which has been integrated into an Advanced Programming course. The intended learning objectives have been met. Although measuring the impact of educational technology in student learning outcomes is often problematic [16], the results obtained with this lab support the conclusion that *VisualJVM* is a helpful educational tool for learning Java technology. Furthermore, from observations and informal conversations with students, many of them enjoyed performing these "hands-on" exercises. Although more time would be necessary to confirm this conclusion, this experience has proved to be an effective methodology to stimulate student interest, as a number of students were subsequently attracted to do their Master's theses on this topic.

Moreover, this tool is suitable for use in other courses, such as those on assembly language [17] and compiler construction [18]. The authors are planning to apply *VisualJVM* to such courses and to offer an advanced seminar for interested researchers or postgraduate students, which would provide an in-depth understanding of the fundamentals of virtual machine and Java technology.

Additional future work includes improving the front-end in response to student comments and the development of a more complete Java interpreter. In particular, to include the implementation of a garbage collection algorithm.

REFERENCES

[1] D. Friedman, M. Wand, and C. Haynes, *Essentials of Programming Languages*, 2nd ed. Cambridge, MA: MIT Press, 2001.

[2] D. Gonzalez-Pea and F. Fernandez-Riverola, "Understanding JPDA (debugging) & JVMTI (profiling) Java APIs within JavaTraceIt," in *Proc. IADIS Int. Conf. WWW/Internet*, Murcia, Spain, Oct. 2006, pp. 334–338.

[3] P. Gomez-Martin, M. Gomez-Martin, and P. Gonzalez-Calero, "Javy: Virtual environment for case-based teaching of JVM," in *Proc. 7th Int. Conf. Knowledge-Based Intelligent Information and Engineering Systems*, Oxford, U.K., 2003, vol. 2773, pp. 906–913.

[4] H. L. Dershem and J. Vanderhyde, "Java class visualization for teaching object-oriented concepts," *SIGCSE Bull.*, vol. 30, no. 1, pp. 53–57, 1998.

[5] B. Venners, *Inside the Java Virtual Machine*. New York: McGraw-Hill, 2000.

[6] C. O. Swift and R. W. Cook, "Sales management simulation: Bringing reality to the classroom," in *Proc. Society for Marketing Advances Meeting*, St. Petersburg, FL, Nov. 2004, pp. 195–198.

[7] K. J. Chapman and C. L. Sorge, "Can a simulation help achieve course objectives? An exploratory study investigating differences among instructional tools," *J. Educ. Bus.*, vol. 74, no. 4, pp. 225–230, Mar./Apr. 1999.

[8] J. Blanchette and M. Summerfield, *C++ GUI Programming With Qt 4*. Upper Saddle River, NJ: Prentice-Hall, 2006.

[9] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2005.

[10] J. Zhao, I. Rogers, C. Kirkham, and I. Watson, "Loop parallelisation for the Jikes RVM," in *Proc. 6th Int. Conf. Parallel and Distributed Computing Applications and Technologies*, Dalian, China, 2005, pp. 35–39.

[11] G. Cabri, L. Leonardi, and R. Quitadamo, "Enabling Java mobile computing on the IBM Jikes research virtual machine," in *Proc. 4th Int. Symp. Principles and Practices of Programming in Java*, Mannheim, Germany, 2006, pp. 62–71.

[12] A. Sartini and J. Waldron, "Dynamic profiling & comparison of Sun Microsystems JDK1.3.1 versus the Kaffe VM APIs," in *Proc. 1st Int. Symp. Information and Communication Technologies*, Dublin, Ireland, 2003, pp. 468–474.

[13] J. Whaley, "Joeq: A virtual machine and compiler infrastructure," *Sci. Comput. Program.*, vol. 57, no. 3, pp. 339–356, Sep. 2005.

[14] F. Yellin and T. Lindholm, *The JVM Specification*, 2nd ed. Reading, MA: Addison-Wesley, 1999.

[15] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[16] G. D. Haertel and B. Means, *Evaluating Educational Technology*. New York: Teachers College Press, 2003.

[17] C. Bredlau and D. Deremer, "Assembly language through the Java virtual machine," *SIGCSE Bull.*, vol. 33, no. 1, pp. 194–198, 2001.

[18] J. Engel, *Programming for the Java Virtual Machine*. Reading, MA: Addison-Wesley, 1999.

Pedro Pablo Garrido Abenza received the M.S. in computer science engineering from the University of Alicante, Spain, in 1997.

He is currently working toward the Ph.D. degree in physics, computing, and communication. He has been an Assistant Professor in the Department of Physics and Computer Architecture, Miguel Hernandez University, Elche, Spain, since 2001. His current research interests include Java programming language and reconfigurable computing. Specifically, he is focused on the development of a Java virtual machine for embedded systems.

Angel Grediaga Olivo received the M.S. degree in telecommunication engineering from the University of Alcala de Henares, Spain, in 1977, the M.S. degree in electronics engineering from the University of Valencia, Spain, in 1998, and the Ph.D. degree in computer architecture from Miguel Hernandez University, Elche, Spain, in 2002.

He is currently an Associate Professor in the Department of Computing and Information Technology, University of Alicante, Spain. His research interests focus on reconfigurable hardware in the area of computer science security, image processing, and robotics. He has authored numerous books and papers in specialized journals including *Lecture Notes in Computer Science* and *WSEAS Transactions on Circuits and Systems*.

Dr. Grediaga is a member of several international conference program committees including the Southern Conference on Programmable Logic (SPL) and International Business Information Management (IBIMA).

Bernardo Ledesma Latorre received the M.S. degree in computer science from the University of Valencia, Spain and the Ph.D. degree in computer architecture from Miguel Hernandez University, Spain, in 1994 and 2001, respectively.

He is an Associate Professor in the Department of Computing and Information Technology, University of Alicante, Spain. His current research interests include open software tools, Java development, security reconfigurable hardware, and image processing.