JavaMod: An Integrated Java Model for Java Software Visualization

Micael Gallego-Carrillo, Francisco Gortázar-Bellas, J. Ángel Velázquez-Iturbide
ViDo Group, Universidad Rey Juan Carlos, Madrid, Spain

{mgallego, pgortaza, a.velazquez}@escet.urjc.es

1 Introduction

Given the practical importance and complexity of object-oriented programming, there are many software visualization systems (VSs) for these languages. These systems use different forms of visualization to assist in understanding object-oriented applications. In particular, some VSs are designed to visualize programs written in the Java programming language (and they are often implemented in such a language, too). Java is an attractive language for visualization developers, because it is a "comfortable" language and it is simple to build visualizations in Java. In the particular case of Java VSs implemented in Java itself, there is an additional advantage: the Java Virtual Machine provides an interface to debug programs written in Java, namely JPDA (JPDA). This interface avoids the need of using external debuggers or of generating program traces. The former often involves obscure interfaces; the latter requires to introduce additional code within the target program in order to extract information at run-time.

Our ultimate goal is to build a infrastructure adequate to the comprehensive, flexible and systematic design of Java visualizations. Many Java VSs have been developed using different Java program representations, for instance, Evolve (Wang, 2002) based on Step (Brown, 2003), Jeliot (Myller, 2004) based on a Java interpreter, and JIVE (Gestwicki and Jayaraman, 2002) based on JPDA.

Our proposal provides an architecture to support three models of Java programs: source code, execution and trace. The final result is a set of APIs that allows working with a comprehensive model of any Java application in a uniform and homogeneous way. Note that we use the term "model" as synonymous for representation; it is inherited from the software engineering community, where representing entities is named modeling.

In the following sections, we briefly describe such an architecture. The second section outlines our three models: source code, execution and tracing. The third section sketches the construction of a debugger based on these models. The fourth section gives a comparison with VSs and tools. Finally, we summarize our future work.

2 Java Models

A program model is a representation of the program in a given programming language. Notice that such a model involves two programming languages: the language in which the target program is written and the modeling language. We are interested in building Java models of Java programs, so we use one only programming language for both roles.

A program can be modelled in different ways, depending on the point of view. We can distinguish the following three models:

- The *code model* provides a static representation of a program. It contains information that can be determined at compile time, such as subclass relationships. The most important code models are based on structural object information, abstract syntax trees (AST) like those generated by compilers constructed with SableCC (Gagno, 1998), or decorated ASTs (i.e. also including semantic information).
- The *execution model* provides a dynamic representation of a program. It contains information that can only be determined at run time, such as the value of variables. It

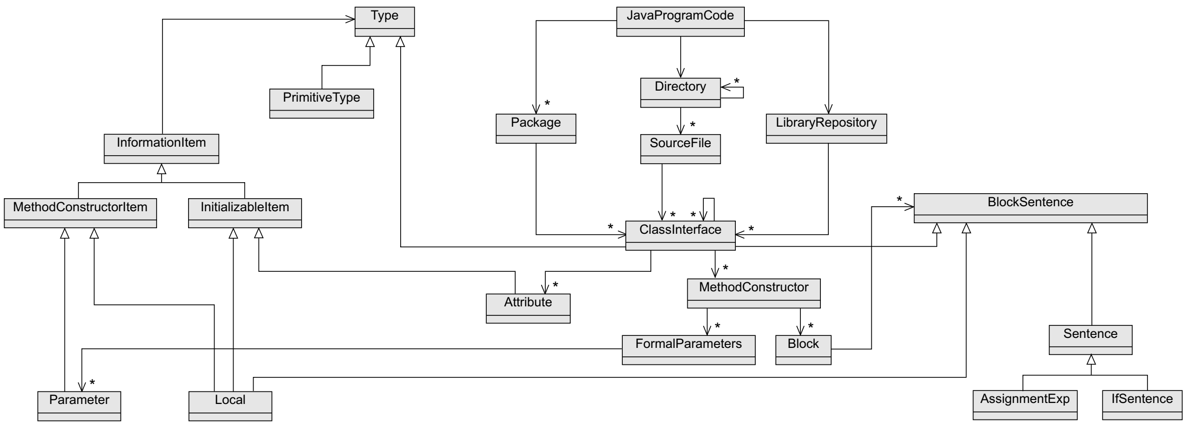

Figure 1: An overview of javaMod code model

is also common to include some code information, such as method names. The most important execution models are based on interpretation, debugging, or instrumentation (i.e. automatically modifying source code to introduce communication with the model at relevant points).

- The *trace model* also provides a dynamic representation of a program. The difference between the execution and the trace models of a given program is that the former only gives information about its current state of execution, while the latter gives information about its execution history. There are no agreed representations for this model. Existing models are based on their intended use, such as profiling or visualization. For instance, Omniscient Debugger (Lewis) is a trace system of Java programs for debugger purposes.

Our proposal for modeling Java programs in Java is called javaMod (`http://vido.escet.urjc.es/javamod`). The model allows representing a Java program with the three models. It is an integrated model, so that semantically related concepts that appear in the three models are explicitly related.

2.1 The javaMod Code Model

The code model of a program represents the contents of its source files and the libraries it uses. It is represented by an instance of the class `JavaProgramCode`. Each program element is modelled with lexical, syntactic and semantic information, as well as information about its location in the source file. Typically, some relevant physical information is also modelled, for example, paths of the `CLASSPATH` variable or path of the virtual machine.

All the elements of any source are modelled as decorated ASTs. The code model is based on classes such as `MethodConstructor`, `ClassInterface`, `Local`, `Attribute` and `Sentence` (examples of sentences are a control flow statement, an object instantiation, a method invocation, an assignment or a "break" sentence). This model is simple enough to be easily understood, because the elements present in source files and the modeling language are represented as objects of those classes (see Figure 1).

2.2 The javaMod Execution Model

The execution of a program is represented as an instance of the class `JavaProgramExec`. Entities present at any instant during the execution of a program are modelled. For instance, the value of local variables can be extracted from the execution stack represented in the model.

This model is based on classes such as `MethodConstructorExec` (that represents the execution of a method or constructor), `ClassInterfaceExec` (that represents a class or interface after being loaded in memory) or `LocalExec` (that represents the memory space of a local).

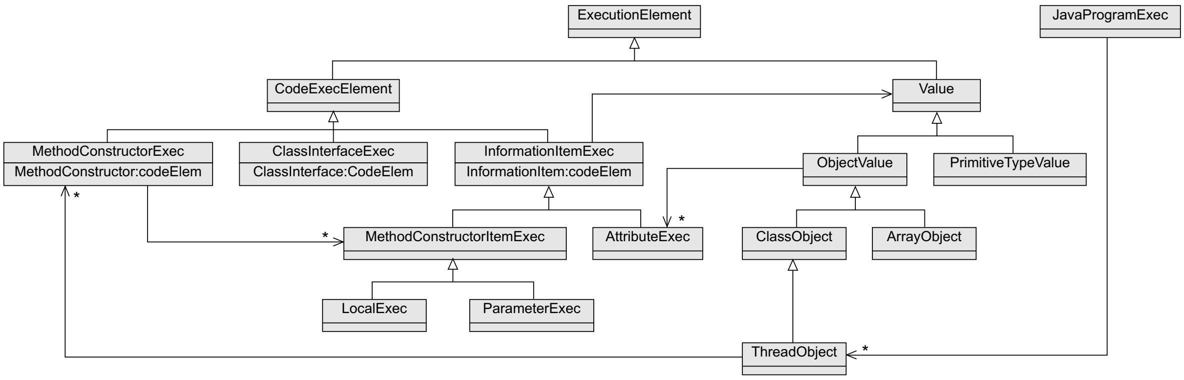

Figure 2: An overview of javaMod execution model

This model represents each thread of a program as an instance of `ThreadObject`. For each thread, its execution stack is represented as a list of `MethodConstructorExec` objects. For each method or constructor under execution, we can know the sentence that is being executed, the object that is serving the message, and the list of variables (see Figure 2).

Each object at the execution model is related to its corresponding object at the code model. For instance, each object of the class `MethodConstructorExec` is related to the object of class `MethodConstructor` it represents.

2.3 The javaMod Trace Model

The trace model records all the information that is generated during the execution of a program. Some pieces of information that can be determined by this model are the number of times a method has been executed, the values a variable has stored or the number of objects instantiated of a given class. Our trace model is in an advanced phase of elaboration, but it is not yet complete. However, we outline the services it will provide.

In this model, there is a class named `MethodConstructorTrace` that represents a method invocation along time, and therefore it stores the instant its execution started, the instant it finishes and the trace of each method invocation made from its body. The class `LocalTrace` represents the trace of the memory space allocated to a local variable, including the allocation time of such a memory space, the set of values stored and the instants they were assigned.

A program trace is represented as an instance of the class `JavaProgramTrace`, which contains a list of `ThreadObjectTrace` objects. Each `ThreadObjectTrace` hosts a tree with all the `MethodConstructorTrace` objects that represent each of the method invocations performed.

Recall that the trace model represents an execution record along the execution time of a program. Consequently, the information provided by this record at a given instant is equivalent to that provided by the execution model.

3 A View of the Debugger Process Using javaMod

In this section we show the use of javaMod to model and build an example application, namely a debugger. First, the overall debugger architecture and its design are presented and then its educational application is introduced.

3.1 Construction of a Java Debugger Using javaMod

To illustrate the versatility of javaMod, we have built a debugger (see Figure 3). It is divided into three blocks: components of the user interface, interest models, and javaMod. In Figure 3, we show component composition as circle ended lines. The user interface is a window that contains a tool bar to interact with the program being debugged (`JDebuggerToolbar`). Four dif-

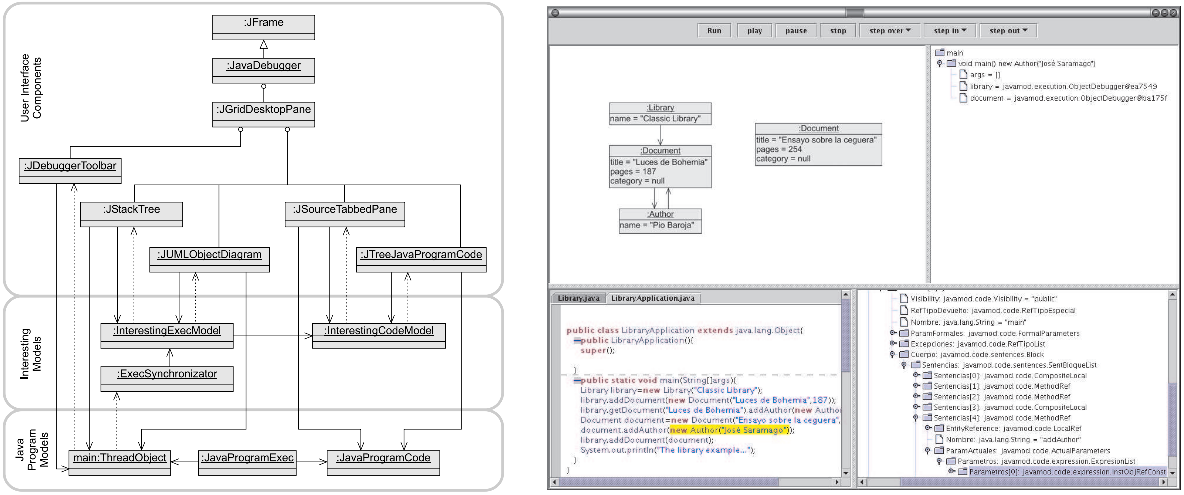

Figure 3: Java debugger architecture and snapshot

ferent, synchonized views of the program are shown: an object diagram (`JUMLObjectDiagram`), a tree showing the execution stack of the thread that executes the method main (`JStackTree`) and two views of the code model. `JTreeJavaProgram` shows a directory structure and each source file with its AST. `JSourceTabbedPane` shows source files with syntax highlighting. The development of this user interface would be the greatest burden in the work of the visualization designer.

We have used the Observer pattern (Gamma et al., 1997), in which the MVC architecture is based, to build javaMod. This pattern defines the generation of events to notify changes of state. In Figure 3 we show the association between subject and observer as dotted lines. We use this event model (in `JavaProgramExec`) to report changes of values in information items (locals, parameters and attributes). We also use it to report the beginning of method execution, the end of thread execution, etc. Our event model is based on the standard event model of Java defined in JavaBeans.

The `JavaProgramExec` instance allows initiating and finalizing the execution of the program. This instance models all the threads under execution in a program. For the sake of simplicity we only consider the main thread in this example, represented by an instance of `ThreadObject`. This class offers operations to pause execution, to resume it and to run step by step. The `ThreadObjects` trigger events when they change of state (paused to resumed or vice versa).

In this way, there is a `JavaProgramCode` that represents a Java program, a `JavaProgramExec` that represents an execution of that program, and a `ThreadObject` that represents the execution of the main thread. The user of the tool can control such a thread execution with a `JDebuggerToolbar`, which contains controls to resume, pause and manage a thread execution. The bar also associates event listeners to the thread to show its state (paused or active).

Typically, the debugger of integrated development environments (IDEs) shows, in the component that visualizes code, the next sentence to execute. In addition, the methods in the stack are shown in the components that visualize execution. The values of local variables of the method at the top of the stack are also commonly shown. This functionality is achieved in our tool by allowing components to access information related of the models.

The components in charge of visualizing the code model do not know or refer to any element of the execution model. As a consequence, the components of the graphical user interface of a tool are more modular and reusable. For instance, some visualizations in tools that do not require program execution can be built such as pretty-printers or metric gatherers.

The views of the code model and the execution model must also be synchronized. For

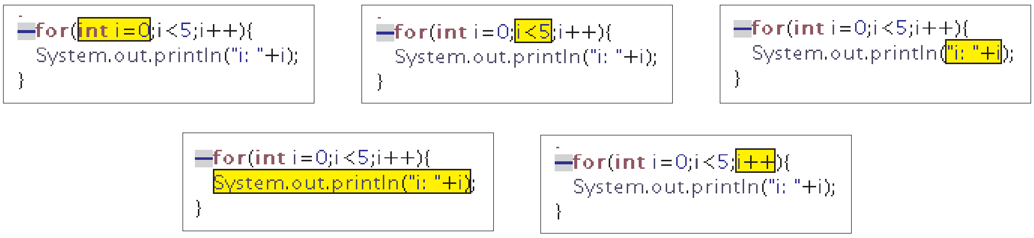

Figure 4: Structural debugger snapshot

instance, if the element of interest for the components that visualize the execution model is the execution of a method or constructor (`MethodConstructorExec`), the element of interest for the components that visualize the code model will be the definition of the method (`MethodConstructor`) or the next sentence (`Sentence`) to be executed.

The point of "interest" at any moment must also be identified, and it either is determined by the current state of execution of the program or is selected by the user interacting with one of the views. In the same way as the selected elements are managed in Java user interface (JFC) through `SelectionModels`, we have created `InterestingModels`.

Basically, the `InterestingModels` keep the element of interest and the components to which changes must be notified. The element of interest will show the current point of execution if it is indicated to the `InterestingExecModel`. An instance of the class `ExecSynchronizator` is used for this purpose. This instance will map the events of the main thread and the `InterestingExecModel`.

Provided there is an element of interest in the code model and an element of interest in the execution model, there is an `InterestingModel` for each of them. In addition, the `InterestingExecModel` is associated to the `InterestingCodeModel` to guarantee synchronization of the views. Establishing an interest element in the `InterestingExecModel` also implies that the corresponding element is established as the element of interest in the `InterestingCodeModel`.

Finally, the components that visualize each of these models must be associated to `InterestingModels`. This association is established directly by setting the interest element through the user interface or the other way around by means of events, so that the views are notified of a change in the interest element.

3.2 Using the Debugger in Education

From an educational point of view, the debugger we have built using javaMod has several advantages over traditional debuggers. It is a "structural debugger" (Gallego-Carrillo et al., 2004), in contrast to traditional line-based debuggers. A structural debugger allows inspecting the state of the program based on its syntactic structure. Consequently, the gap between the execution dynamics of a program and its static declaration is shorter. From an educational point of view, it allows instrumenting programs according to the operational semantics explained to students in the classroom. The comprehension of such static-dynamic relation is one of the problems most students have on learning programming. The facilities of our debugger could be applied to imperative languages in general, although it is currently applied to the Java language.

The construction of a Java debugger that implements the ideas of structural debugging has been possible by the way javaMod integrates the code model (with the syntactic information) and the execution one (with the capabilities of a traditional debugger). The use of the code model as an aid for the execution model allows for the inclusion of the operational semantics of the language in the debugger process.

For instance, suppose the *for* loop is to be explained in a classroom. In a *for* loop the initialization part is executed first and once, then the condition is evaluated and if it succeeds

Table 1: Some software visualization and educational tools

Tool Name	Code	Exec	Trace	Technologies
BlueJ (BlueJ)	√	√	×	JPDA
DrJava (DrJava)	√	√	×	DynamicJava (DynamicJava)
JGrasp (JGrasp)	√	√	×	DynamicJava & JPDA
ProfessorJ (ProfessorJ)	√	√	×	DrScheme plugin (DrScheme)
Jacot (Leroux et al., 2003)	×	√	×	JPDA
Omniscient Debugging	×	×	√	Instrumentation
JRat (JRat)	×	√	×	Instrumentation & others
Evolve	×	×	√	Step trace protocol
Fujaba (Fujaba)	√	√	×	JPDA

the body of the loop is executed. Afterwards, the loop variable is updated and the condition is evaluated again, and so on. In this context, a debugger showing what is the next part to be executed or evaluated could be really valuable for the students if used in conjunction with the theoretic explanations (see Figure 4).

Structural debugging allows performing operations such as showing what are the next suitable sentences to be executed (by taking into account where the program is stopped at the moment). As an example, if the program is stopped in the condition part of an if-statement at least two things can happen. If the condition evaluates to true, the then-part is executed, so the first sentence in that part could be highlighted. If the condition evaluates to false, then either the first sentence of the else-part (if exists) is executed or the first sentence after the if-sentence is executed, so they could also be highlighted. Even if the normal execution flow can be broken via an exception this can be detected with javaMod, and the corresponding catch block can be visualized too as a possible point to step to.

4 Related Work

The available Java VSs are based on their own specific architectures. Those that are capable of step-by-step debugging are usually based on JPDA. However, the management of source code and libraries and the execution trace is done without any standard. Moreover, the elements visualized have to be synchronized and this synchronization process has to be done by the tool itself. Table 1 shows some tools, identifies the program models each one manages and cites the technologies used to obtain such information from the Java program.

Some tools such as Fujaba, Evolve or BlueJ have a plugin architecture to include new functionality. Even so, they have not been constructed allowing other to build VSs just using its internal representation for Java programs (without making use of their user interface or installation procedure). JavaMod, however, is an API which has been designed to obtain all the relevant information from a Java program and to make this information available to build any kind of tool.

The models available for building tools are not integrated. Each one offers an specific functionality and it is not easy to integrate them. In Table 2 several APIs are shown focusing on the main aspect they are intended for.

5 Conclusions and Future Work

We have described a new approach to modelling Java programs in Java, called JavaMod. JavaMod allows defining three models: source, execution and trace. It has also been compared with advantage to other models. Currently, it provides a framework to build ambitious programming tools and visualizations. We have also illustrated it by applying it to build a structural debugger.

Table 2: Some Java program management APIs

API Name	**Definition**	**Execution**	**Trace**
Java Reflection (JavaReflection)	$\surd$	×	×
BCEL (BCEL)	$\surd$	×	×
Javassist (Javassist)	$\surd$	×	×
PMD (PMD)	$\surd$	×	×
RECODER (RECODER)	$\surd$	×	×
BARAT (BARAT)	$\surd$	×	×
OpenJava (OpenJava)	$\surd$	×	×
Eclipse JDT Core (JDT)	$\surd$	×	×
DynamicJava	×	$\surd$	×
BeanShell (BeanShell)	×	$\surd$	×
JPDA	×	$\surd$	×
Eclipse JDT Debug (JDT)	×	$\surd$	×
STEP	×	×	$\surd$

A useful feature that could be integrated into the code model consists in making it modifiable so that modifications are notified as events. This would permit that the code→debugging→execution→profiling cycle were integrated in one single model. We also plan to design educational tools that will make use of javaMod to generate graphical explanations of programs.

From a practitioner's point of view, it would be very useful to facilitate the use of the model in standard environments. To this aim, we are developing a tool to transform Java models into UML 2.0 (UML), as defined in the UML2 project by Eclipse (UML2), that provides serialization in the standard format XMI. Another interesting feature to make our model more useful consists in being able to incorporate it as a plug-in in the most relevant IDEs such as Eclipse (Eclipse) and NetBeans (NetBeans).

Finally, it would be useful to build a model generator, so that given a specification for a language, models were generated that included lexical, syntactic, semantic, and execution elements of the language.

6 Acknowledgements

This work is supported by the research projects TIC2000-1413 of the Spanish Research Agency CICYT and TIN2004-07568 of the Ministerio de Educación y Ciencia.

References

BARAT. URL `http://sourceforge.net/projects/barat`. (seen September 2004).

BCEL. URL `http://jakarta.apache.org/bcel/`. (seen September 2004).

BeanShell. URL `http://www.beanshell.org/`. (seen September 2004).

BlueJ. URL `http://www.bluej.org/`. (seen September 2004).

Rhodes H. F. Brown. STEP: A framework for the efficient encoding of general trace data. Master's thesis, McGill University, 2003. URL `http://www.sable.mcgill.ca/step/`.

DrJava. URL `http://www.drjava.org/`. (seen September 2004).

DrScheme. URL `http://www.drscheme.org/`. (seen September 2004).

DynamicJava. URL http://koala.ilog.fr/djava/. (seen September 2004).

Eclipse. URL http://www.eclipse.org/. (seen September 2004).

Fujaba. URL http://www.fujaba.de/. (seen September 2004).

Étienne Gagno. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, 1998.

Micael Gallego-Carrillo, Francisco Gortázar-Bellas, and J. Ángel Velázquez-Iturbide. Depuración estructural: Acercando la práctica a la teoría de la programación. 6th International Symposium on Computers in Education. To appear, 2004.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

Paul Gestwicki and Bharat Jayaraman. Interactive visualization of Java programs. In *Symposia on Human Centric Computing Languages and Environments*, pages 226–235, 2002.

JavaBeans. URL http://java.sun.com/products/javabeans. (seen September 2004).

JavaReflection. URL http://java.sun.com/j2se/1.4.2/docs/guide/reflection/. (seen September 2004).

Javassist. URL http://www.csg.is.titech.ac.jp/~chiba/javassist/. (seen September 2004).

JDT. URL http://www.eclipse.org/jdt/. (seen September 2004).

JFC. URL http://java.sun.com/products/jfc. (seen September 2004).

JGrasp. URL http://www.jgrasp.org/. (seen September 2004).

JPDA. URL http://java.sun.com/products/jpda/. (seen September 2004).

JRat. URL http://jrat.sourceforge.net/. (seen September 2004).

Hugo Leroux, Annya Réquilé-Romanczuk, and Christine Mingins. Jacot: a tool to dynamically visualise the execution of concurrent Java programs. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, 2003. ISBN 0-9544145-1-9.

Bil Lewis. Debugging backwards in time. URL http://www.lambdacs.com/debugger/debugger.html. (seen September 2004).

Niko Myller. The fundamental design issues of Jeliot 3. Master's thesis, University of Joensuu, 2004. URL http://cs.joensuu.fi/jeliot/.

NetBeans. URL http://www.netbeans.org/. (seen September 2004).

OpenJava. URL http://openjava.sourceforge.net/. (seen September 2004).

PMD. URL http://pmd.sourceforge.net/. (seen September 2004).

ProfessorJ. URL http://www.professorj.org/. (seen September 2004).

RECODER. URL http://recoder.sourceforge.net/. (seen September 2004).

UML. URL http://www.uml.org. (seen September 2004).

UML2. URL http://www.eclipse.org/uml2. (seen September 2004).

Qin Wang. Evolve: An extensible software visualization framework. Master's thesis, McGill University, 2002. URL http://www.sable.mcgill.ca/evolve/.