Software Documentation of Jeliot 3

Andres Moreno and Niko Myller

X.X.2003

University of Joensuu
Department of Computer Science
Special Project in Computer Science
Software Documentation

Contents

1 Introduction

2 Concept Design

3 The Components of the Jeliot 3 System

3.1 Visualization Engine

3.2 DynamicJava

Dynamic Java is a Java source code interpreter written in Java. This application is open source and can be freely obtained . At the moment, DynamicJava is almost fully compliant with Java specifications and supports multi-threading. Due to the lack of documentation (nothing is provided except an API documentation done by JavaDoc) it was necessary to take a look to the source code of the software to understand its behaviour. There we obtained the following relevant information:

DynamicJava consists of 7 different packages, where only five of them actually perform the interpretation: classfile, classinfo, interpreter, parser and tree. The other two (util and gui) are used to help the debugging of DynamicJava and to provide a nicer user interface to the interpreter. For example, the displayVisitor, included in the util package, provides a nice output from the syntax tree.

- **Classfile** contains all the classes for creating general purpose bytecode classes. The most important class is ClassFile which is the heart of the class creation process.
- **Classinfo** contains all the classes and interfaces for using reflection on Java or interpreted classes. This package is used during the compilation of the classes.
- **Interpreter** contains the classes for interpreting Java language statements. This is the most important package. It contains the most important visitors that will be explained later.
- **Parser** provides the classes that compose the default parser for the language. The parser itself is represented by the class Parser. All the Java files were generated by JavaCC 1.0. It creates the nodes of the tree to be traversed later by the interpreter visitors.
- **Tree** provides classes and interfaces for producing an abstract syntax tree. This package does not depend of any non standard java package.

The created tree consists of nodes, the main data structure used in DynamicJava. All nodes have common properties, the segment of source code where that node refers. Subclasses of this node are defined to address the unique properties of each different Java (e.g. staments and constructions). For example a node for any binary expression will also consist of the properties LEFT_EXRESSION and RIGHT_EXPRESSION.

The following Figure 1 explains the main relationships between the packages, the visitors and the main data flow.

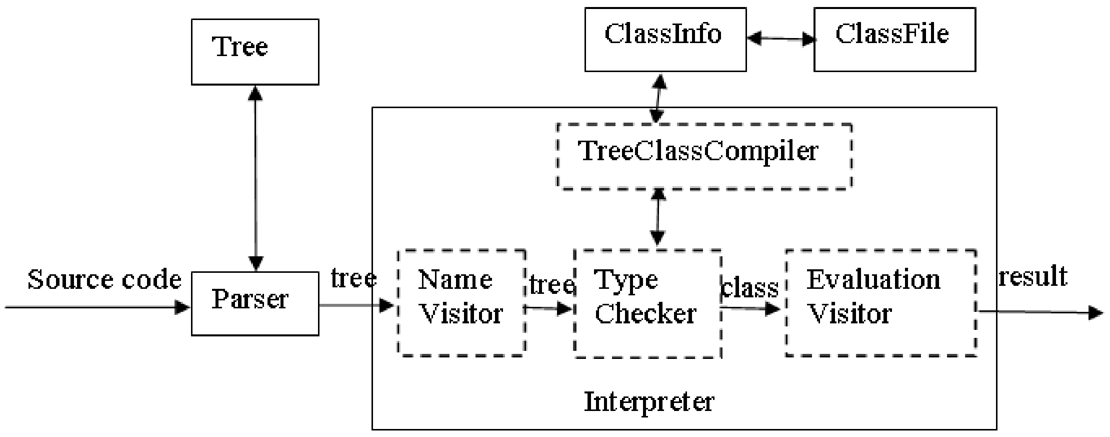

Figure 1: The package structure and dependencies between packages in DynamicJava.

As we can see in the image DynamicJava carries the source code through 3 visitors: NameVisitor, TypeChecker and, finally, EvaluationVisitor. We also see how EvaluationVisitor will receive a class from TypeChecker, the reason of this behaviour will be explained in the next paragraphs and in the section dedicated to the Concurrent Interpreter

3.3 NameVisitor

This tree visitor resolves the ambiguity in identifiers in a syntax tree. As declared, this visitor traverses the tree trying to find out syntactical ambiguities.

3.4 TypeChecker

This tree visitor checks the typing rules and loads the classes, fields and methods. This TypeChecker class is not only worried about typing rules. When visiting a class declaration, it invokes TreeCompiler, which compile the class into Java bytecode. However this compiling process alters the class and the formed byte code does not match the original source code of the class.

3.5 EvaluationVisitor

This tree visitor evaluates each node of a syntax tree. This visitor is the one that performs the evaluation and execution of the program. It usually starts by invoking the main method of the compiled class. We can easily observe how it traverses the syntax tree and modifies DynamicJava structures to store information and thus we can interfere with its normal interpretation to extract the information it produces while interpreting the source code.

4 Communication and Intermediate Code

4.1 Communication Model

4.2 Intermediate Code

As XML-based approach was discarded, a new language was to be designed in order to express the information extracted from the source code interpretation and pass it to a new ad-hoc interpreter. This interpreter will parse these instructions (m-code sentences) and give the "script" of the animation or "play" to the Director, which displays the actors on the screen. These cinematographic metaphors come from the previous versions of Jeliot. Director stands for the component that manages the different pieces of information (actors) on the screen.

The m-code syntax is quite simple. While the inner representations of the m-code commands are numbers, Java constants are used to refer to them. The usual m-code sentence will consist of:

- **Expression/Statement code**: A shortcut for every Java statement or expression is used: e.g. AE stands for Add Expression. The chosen names are heavily related to the ones used in DynamicJava.
- **Reference:** Every Expression/Statement sentence is identified by a number. This way nested statements and expressions can be formed up from previous m-code sentences.
- **Related References:** Most of the m-code sentences refer to previous m-code sentences. One Add Expression will refer to the references of both sides of expression. Flow-control statements will refer to a condition expression, and so on.
- **Value:** Most sentences will return the value resulting from the executing of an expression. If it is a flow control statement it will return a Boolean value indicating the result of the condition.
- **Type:** Every expression that has a result must specify its type.

- **Location:** This contains the location of the expression in the original source code file.

Some auxiliary m-code commands have been defined to simplify m-code interpretation, especially when referring to assignments and binary expression:

- **BEGIN:** Indicates beginning of an assignment or expression. It encapsulates nested expressions, literals or qualified names.
- **LEFT:** Indicates beginning of the left hand side of an expression.
- **RIGHT:** Indicates beginning of the right hand side of an expression.
- **TO:** Indicates the beginning of the assignment destination.
- **END:** States the end of the current program execution.

One typical assignment like a = b + 1; is coded as follows:

```
Begin|Assignment|1|1,1,1,10
Begin|AddExpression|2|1,5,1,10
Left|3
QualifiedName|3|b|1|int
Right|4
Literal|4|1|int|1,9,1,10
AddExpression|2|3|4|2|int|1,1,1,10
To|5
QualifiedName|5|a|UnknownValue|int
Assignment|1|2|5|2|int|1,1,1,10
```

In this example we find two new commands QUALIFIED NAME and LITERAL. QUALIFIED NAME refers to variables previously declared and LITERAL states for literal values, as numbers, characters or strings.

For a complete listing of commands and their descriptions see the Intermediate Language Specifications document..

4.2.1 Evaluation Visitor

As commented previously EvaluationVisitor was the main class to be modified. In the next subsections I will describe how M-code is produced for certain subsets of Java expressions and statements:

Static Method Call

Static method call is the entry point to the evaluation visitor. It is the visitor called when invoking the main method of a class. The first the I/O management is done. I/O facilities were to be built-in in DynamicJava, as they require special treatment in the Jeliot side. We have chosen to keep on using the I/O library provided by Jeliot 2000, nevertheless this can be changed by doing some simple changes in EvaluationVisitor. When visiting a static method call [public Object visit(StaticMethodCall node)], we first ask for the declaring class.

- If it is an Input class we ask the Director to provide the information requested by ways of one pipe that communicate both sides. We discriminate the type by the method name. Every input method has an equivalent in ECodeUtilities, where the value is actually read from the pipe. An m-code command named INPUT has been defined to request data of a given type from the director.
- If it is an Output class then the only method currently available is println. DynamicJava visits the argument and sends the resulting string to the Director with the command OUTPUT.

After that a stack is maintained by StaticMethodCall and Return visitors to manage multiple method calls (E.g. return object.method()). A reference number is pushed into the stack in every method call.

Later, the argument types are processed and m-code is produced to inform m-code interpreter which the types are. Finally DynamicJava will invoke the method with all the information. When the invocation ends a special statement is produced to indicate the end of static method call (SMCC).

However, when a static method call refers to a foreign method (no source code is provided for it), the normal invocation will only return the value, if it is not a void method. But to visualize the call properly we need to simulate the parameters passing and the method declaration m-code statements. Moreover, the value returned must be inside a return m-code statement, again for visualization purposes, so it is simulated too.

Return Statement

A return statement can contain a value to return or nothing at all (a void method or function). If there is something to be returned a BEGIN statement is produced before visiting the expression to be returned. Otherwise a simple return statement is produced with the special constant Code.NO_REFERENCE, so jeliot interpreter will not look for an expression.

A stack is maintained by StaticMethodCall and Return visitors to manage recursive method calls (E.g. return object.method()). A reference number is pushed into the stack in every method call. The return statement will pick it up from the top of the stack and that will identify the return statement.

Simple Assign Expression

A BEGIN statement is produced indicating the beginning of a new assignment. Then the right expression is visited and thus it produces its own m-code. A special statement TO is produced pointing the beginning of the left expression, where the value obtained interpreting the right expression will be stored. This left expression was not visited in the original DynamicJava, as it is not needed to modify the context. However we need to visualize that expression, so an "artificial" visit was added. An evaluating flag is set to show that it is an "artificial" visit. Finally we produce the assign code with references to both expressions.

Qualified Name

Qualified Names are the names already declared (e.g. variable names and any other identifier) and they are used in expressions. Its visitor was modified to take into account the evaluating flag. The reason of having two different behaviours is that DynamicJava throws an ExecutionError when visiting an uninitialized qualified name. That occurs an artificial visit is made in assignments. So when the evaluating flag is false the DynamicJava invokes display method to avoid unnecessary exceptions. However, both methods (visit and display) produce the same m-code.

Variable Declaration

Variable declaration does not cause big modifications to original source code. Only if an initialization expression is found we have to modify the normal process to visualize the initialization. This is done by simulating an assignment after the variable is declared.

Flow Control Statements

All flow control statements work similarly. Here, I will explain how a while statement produces its m-code. Other statements work in a similar way.

Firsts of all, the while statement node keeps the reference to the condition that will be visited and will determine whether to enter or not the body of the statement. If the condition holds the visitor produces a WHILE statement with TRUE as a value. Then the body is evaluated, producing its own m-code.

Break or continue visitors throw exceptions to be caught by the flow control statements. When they are caught their corresponding m-code statement is produced. This statement reflects where the break or continue has happened (WHILE, FOR, DO and SWITCH expression)

Boolean and Bitwise Unary Expressions

This group contains the not (!) and complement ( ) operators. There are two different possibilities. On the one hand the expression can be constant and not evaluation is performed by DynamicJava, and the generation of m-code is straightforward. However, as there is no expression to be referred, only a value is returned. Code.NO_REFERENCE is used to indicate this fact to the interpreter. On the other hand, when there is an expression to negate, a BEGIN statement is produced before the expression to be negated is visited. Finally the unary statement is produced returning the value and referring to the expression it affects.

Unary arithmetic expressions

This group contains increments and decrements (++, –). No special modifications were carried out in these visitors. They just generate a BEGIN statement and their own statement (PIE, PDE, PRIE,PRDE), that returns the modified value and the type.

Binary Expressions

This group comprises all boolean, bitwise and arithmetic binary operators. As usual, a BEGIN statement is produced, anticipating what the operator is. Then both sides of the expression are visited and their values are collected. Before each of these two visits there is one special m-code statement: a LEFT statement, for the left side, and a RIGHT statement for the right side. Finally the binary statement is generated referring both sides and the value resulting of applying the operator to both sides of the expression.

Compound Assignment Expressions

This group contains all bitwise and arithmetic compound assignments. Compound operators are for example +=, *=, /= and >=. The visitors of these compound assignments have been modified to produce m-code that decomposes the compound assignment into a simple assignment and a binary operation. E.g. a+=3-b will be interpreted as a=a+(3-b).

Then, the code of the visitors is just a composition of an assignment and a binary expression as its right hand side. This binary expression has as its sides the same ones that the compound assignment. As a result of these two fake or artificial visits are done to the left hand side of the compound assignment.

4.2.2 Tree Interpreter

This class contains methods to interpret the constructs of the language. This class is the one called from Jeliot to start interpreting a program or a single method. There are two main methods that have been modified.

interpret(Reader r, String fname)

This method receives the source code and invokes the three visitors DynamicJava consists of: NameVisitor, TypeCheker and EvaluationVisitor. This method catches the execution and parsing errors and generates m-code to notify Jeliot of the possible lexical, syntax or semantic errors that are found during the interpretation.

interpretMethod(Class c, MethodDescriptor md, Object obj, Object[] params)

Whenever a domestic method is invoked by the interpreter, this method will construct everything that allows interpreting it. The m-code generated by this method provides the names of the formal parameters so they are added to the method variables by the Jeliot interpreter. It also indicates the location of the method declaration in the source code by means of a special m-code statement: MD, method declaration. All these are the information that are generated in StaticMethodCall if the method is a foreign one. The flag inside is set to indicate that the currently interpreted method is a domestic method.

5 Managing Jeliot 3 System

5.1 Source Code Distribution

5.1.1 Directory hierarchy

Jeliot 3 is available as zip file containing all the sources and files needed to build it. After unzipping the file we will find the following directories:

- lib: Contains the tools needed for the automated build process.
- resources: Contains information messages used by DynamicJava to alert from errors.
- src: Contains the source code for Jeliot 3. It is divided into the following sub-folders:
  - docs: Contains the web pages that are used for the help page and the about page. It also contains the licenses under which Jeliot 3 is distributed.
  - examples: Contains the examples that will be available for users of Jeliot, new examples can be added here.
  - images: Those images used in the user interface of Jeliot.
  - jeliot: All source files related to Jeliot visualization engine(theatre), m-code interpretation (ecode) and the graphical user interface(gui).
  - koala: You will find here the modified source code of DynamicJava.

5.1.2 How to build Jeliot 3 from the source

Being used to the build tool used by DynamicJava (Ant), it was modified to build also Jeliot 3. Ant is a UNIX make-clone oriented to build Java programs based on an XML configuration file. When unzipping the source code, three files will appear on the destination directory:

- build.xml: This file defines the possible targets and its tasks that we want to perform with Ant. It includes some properties to customize the output. For example, you can rename the minor version of jeliot by modyfing the property named minor. For adding more targets you should refer to Ant manual page .

- build.bat and build.sh: These are the batch files that will invoke Ant with one of the arguments (targets) that you can provide to it:
  - compile: Jeliot 3 source code will be compiled and classes obtained will be located at classes subfolder. To run Jeliot 3 from this point you should enter the command "java jeliot.Jeliot" inside the classes subfolder.
  - dist: This argument will compile (if necessary), create the jar files and zip them. This way we will obtain a binary zip file of Jeliot 3 (Jeliot3$minor.zip)" and another zip file of Jeliot 3 source code (Jeliot3$minor-src.zip). These files are ready to be distributed without any modifications.
  - clean: deletes every file created by the build tool.

Notice that you must set JAVA_HOME to point at your Java Development Kit installation. You can do this by modifying a line in the build.bat file: set JAVA_HOME=e:\j2sdk1.4.2

A normal session could consist of the following commands:

```
c:\jeliot3>build compile
c:\jeliot3>cd classes
c:\jeliot3\classes>java jeliot.Jeliot // We get an instance of Jeliot running
c:\jeliot3\classes>cd ..
c:\jeliot3>build dist // Distribution files created
c:\jeliot3>cd Jeliot3 c:\jeliot3\Jeliot3>jeliot // We get an instance of Jeliot running
```

5.2 Non-Source Code Distribution

5.3 System Requirements

5.4 Installing and Running Jeliot 3 System

5.5 Extending Jeliot 3 System

6 Testing

References