Jeliot Program Animation System
Internal Documentation

Version 3.2

Niko Myller and Andrés Moreno García

4th December 2003

Contents

1 Introduction

Jeliot is a program animation system intended for teaching introductory programming. Programs are animated fully or semi-automatically, requiring no modifications or annotations on the part of the instructor or student. While this limits the flexibility of the animation, Jeliot is extremely simple to use so that it is easily accepted by true novices, as well as by their teachers who do not have to invest in learning how to prepare animations.

Jeliot is written in Java for portability and animates program that are written in Java. Jeliot uses a modified version of DynamicJava (Koala, 2002) (section 3.3) as a front-end and modified version of Jeliot 2000's animation engine (section 3.6) as its back-end. The user interface was also adopted from Jeliot 2000 (sections 3.1 and 3.2).

DynamicJava is a Java source code interpreter written in Java. This application is open source and can be freely obtained from `http://www.koala.ilog.fr/djava/`. At the moment, DynamicJava is almost fully compliant with Java language specifications and supports multi-threading.

To make these two separate systems communicate (section 4) with each other a new intermediate code (section 5and Intermediate Language Specification) and intermediate code interpreter (section 3.4)were designed. In this way the system should be more flexible for modifications and new extensions (sections 2 and 6).

1.1 History

The first member of the Jeliot family of program animation systems was called Eliot; it was developed by Erkki Sutinen and Jorma Tarhio and their students at the University of Helsinki. Eliot was written in C and used the Polka animation library. Later a version was written in Java and the name was changed to Jeliot. (This version is now called Jeliot I to distinguish it from later versions.)

Jeliot I was a flexible system: the user could choose the variables to be animated, the graphical form of the animated elements and multiple views of the same program. It proved too difficult for teaching novices, so a new version called

Jeliot 2000 was developed at the Weizmann Institute of Science by Pekka Uronen under the supervision of Moti Ben-Ari. A pedagogical experiment carried out by Ronit Ben-Bassat Levy proved the effectiveness of Jeliot in teaching introductory programming.

Jeliot 2000 only implemented a very small subset of the Java language. The current version Jeliot 3 is a re-implementation designed to significantly extend the range of Java features that are animated, in particular, to cover object-oriented programming. The design and implementation was carried out at the University of Joensuu by Niko Myller and André Moreno-García, under the supervision of Moti Ben-Ari and Erkki Sutinen. For an extensive discussion of the history of Jeliot, see (Ben-Ari et al., 2002).

1.2 Release Notes

version 3.2 contains a documentation and supports inheritance. In addition to the previously supported features it has a support for:

- Inheritance of classes.
- `super()` method calls at the beginning of a constructor.

Includes also minor bug fixes.

version 3.1 supports objects creation and object method calls. Line numbering was added for source code editor and viewer. In addition to the previously supported features it has a support for:

- User made classes, however, inheritance is not yet supported.
- Constructor calls.
- Object method calls.
- Object field access.

Includes also minor bug fixes.

version 3.01 is a maintenance release. Bugs of the initial release were fixed. Support for `switch` statement was added.

version 3.0 is the initial public release. It has a support for:

- Values of type `String`, all primitive types.
- One-dimensional arrays with primitive types or Strings as its component type.
- Expressions including all unary and binary operations except `instanceof`.
- All the control statements (`if`, `while`, etc.) except `switch` statement and conditional expression (`exp?exp1:exp2`).
- Method invocation, including recursive invocation.

Not implemented features are:

- Static variables.
- Calls to `super(...)`.
- Super field accesses.
- Arrays with components of reference type (except `String`).
- Conditional expressions (`exp?exp1:exp2`).
- Array initializers.
- Java 2 SDK API classes' methods cannot return object or array types.

2 Managing Jeliot System

In this chapter we explain what are the system requirements of Jeliot, what files and directories different distributions contain and how the source code distribution can be build.

2.1 System Requirements

2.2 End User Distribution

2.3 Source Code Distribution

2.3.1 Directory Hierarchy

Jeliot 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. It is divided into the following subfolders.

- `docs` Contains the web pages that are used for the help page and the about page. It also contains the licenses under which Jeliot 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` The modified source code of DynamicJava.

2.3.2 How to Build Jeliot 3 From the Sources

Jeliot uses the build tool called **Ant**. Ant is a UNIX make-clone oriented to build Java programs based on an XML configuration file (Apache, 2003a). 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 (Apache, 2003b).

build.bat and build.sh These are the batch files that will invoke Ant with one of the arguments (targets) that you provide to it:

compile Jeliot source code will be compiled and classes obtained will be located at classes subfolder. To run Jeliot from this point you should enter the command java jeliot.Jeliot inside the classes subfolder.

dist This argument will first compile the source files if necessary and create a jar file of the classes and compress them into zip file. This way we will obtain the binary executables in a zip file called Jeliot3${minor}.zip. Moreover, another zip file is created containing the source files of Jeliot Jeliot3${minor}-src.zip. Both of these files are ready to be distributed without any modifications.

clean deletes all the files created by the build tool.

Notice that you must set an envinroment variable JAVA_HOME to point at your Java Development Kit installation.

A normal session of compilation or distribution creation could consist of the following commands:

```
c:\jeliot3> build compile
```

The compiled class files are now in the folder `classes`.

```
c:\jeliot3> cd classes
c:\jeliot3\classes> java jeliot.Jeliot
```

Jeliot can be run with the command above. But if we want to make a distribution we need to do as follows and we will get two separate distributions as described above.

```
c:\jeliot3\classes> cd ..
c:\jeliot3> build dist
c:\jeliot3> cd Jeliot3
c:\jeliot3\Jeliot3> jeliot
```

3 The Structure of Jeliot System

We introduce here the different components of Jeliot systems. The system contains several packages and here we explain those that are most crucial in understanding the structure of Jeliot. Only package and classes related to communication between visualization engine and DynamicJava are introduced in the chapter 4.

The functional structure of the Jeliot 3 is shown in the Figure 1. A user interacts with the user interface and forms the source code of the program (1). The source code is sent to the Java interpreter and the intermediate code is extracted (2 and 3). The intermediate code is interpreted and the directions are given to the visualization engine (4 and 5). A user can control the animation by playing, pausing, rewinding or playing step-by-step the animation through the user interface (6). Furthermore, the user can input data, for example, an integer or a string, to the program executed by the interpreter (6, 7 and 8).

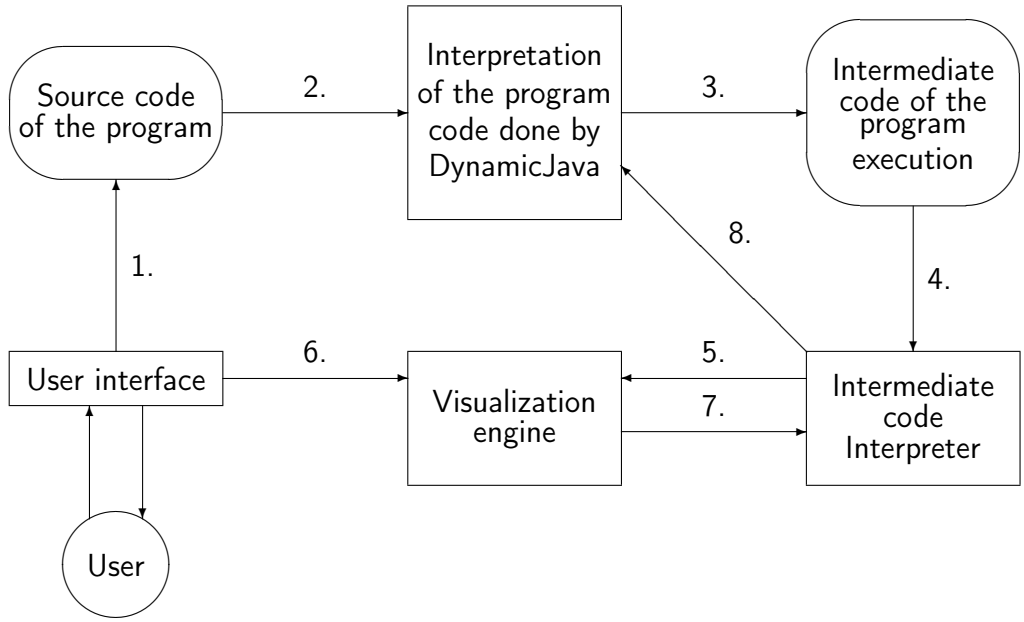

Figure 1: The functional structure of Jeliot 3.

Different packages of Jeliot 3 are introduced in this section in the same order as in the functional structure. In this way we hope that all the components and their meaning becomes clearer.

3.1 Jeliot Class

Class Jeliot in the package `jeliot` is the main class of the program. It combines the different components of the program together a deals with some communicational issues. When the program is started the class creates all the needed components and passes them as parameters to the user interface classes. It mainly handles the communication between the user interface (`jeliot.gui`) and the theater/animation engine (`jeliot.theatre`) classes. In addition to that it also invokes the thread handling the interpretation of the user program.

3.2 User Interface

The user interface of Jeliot is located in the package `jeliot.gui`. The structure of the user interface is shown in the Figure 2. The main class is `JeliotWindow` that extends `JFrame`. The frame is layed out by `BorderLayout`. A split pane (`JSplitPane`) is added in the center and a panel containing control panel (created in `JeliotWindow`) and output console (`OutputConsole`) is added in the south (bottom) border.

In the split pane the left side is used by code editor (`CodeEditor`) or code viewer (`CodeViewer`). Code editor is shown during editing of the program and code viewer during the animation of the program. Both of the text components have a line numbering component (`LineNumbers` on their row header. Moreover, code editor consist of editing panel that has a few buttons to load, save and edit the source code. The frame has also a menu bar that is constructed partially in `CodeEditor` class and partilly in `JeliotWindow` class.

On the right hand side of the split pane a animation engine called theater (`Theatre`) is normally shown. However, if any errors occur during the compilation or runtime of the program they are shown in the `Error viewer` (`ErrorViewer`) instead of theater.

`JeliotWindow` combines all the components in `jeliot.gui` package and creates the user interface. It also deals with most of the events happening during the runtime and delegates the commands forward to the appropriate classes (e.g. `Jeliot` or `CodeEditor`). Other classes in the `gui` package are related to one of the com-

ponents introduced in the previous paragraph. There are also three classes that are not currently in use in Jeliot 3, namely `LoadJeliot`, `DraggableComponent` and `TheatrePopup`.

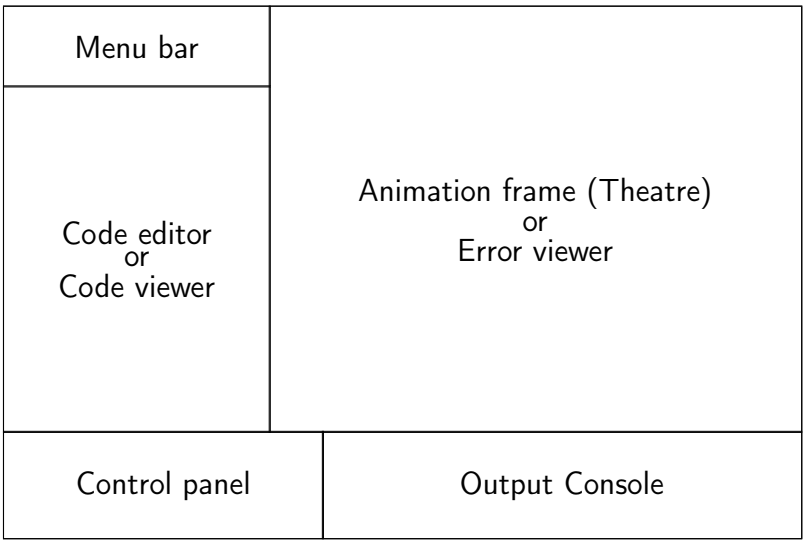

Figure 2: The structure of user interface in Jeliot 3.

3.3 DynamicJava

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`. The parser is 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`.

Figure 3 explains the main relationships between the packages, the visitors and the main data flow.

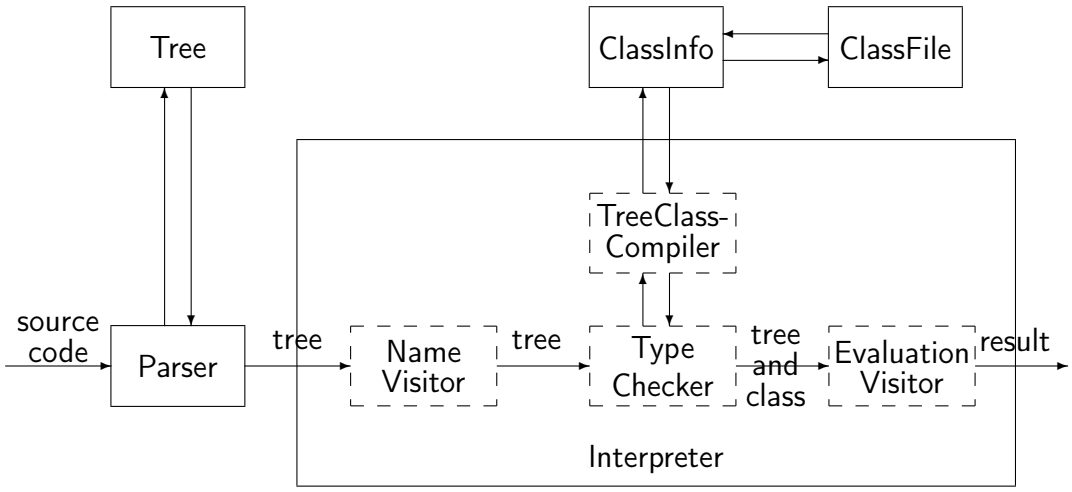

Figure 3: Packages, visitors and data flow in DynamicJava

As we can see in the image DynamicJava carries the source code through three 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.

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

The `TypeChecker` is a tree visitor that 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 compiles the class into Java bytecode. However, this compiling process *alters the class* and the formed *bytecode does not match the original source code* of the class.

The `EvaluationVisitor` is a tree visitor that 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.

3.3.1 Tree package

The `Tree` package has a class for every different Java language construct. These classes are organized also as a tree. The class Node is the one that every other class inherits, directly or indirectly. It defines only one thing, the location of the node in the source code, both the file and the position inside it. Most of the other classes also define properties that later are inherited by more specific classes.

The tree organization is shown in the Diagrams 4, 5, 6 and 7. These figures do not depict all the nodes. Especially those that do not add more information.

Figure 4 shows the different kind of declarations, initializers and types. Two main types are used by DynamicJava: Array type and and primitive types. String are not considered as a different type as it is a class. It is Java that supports it by normal String methods.

Figure 5 shows the primary expressions classes. Its super class, Expression, is also super class for binary and unary expressions. All properties are defined

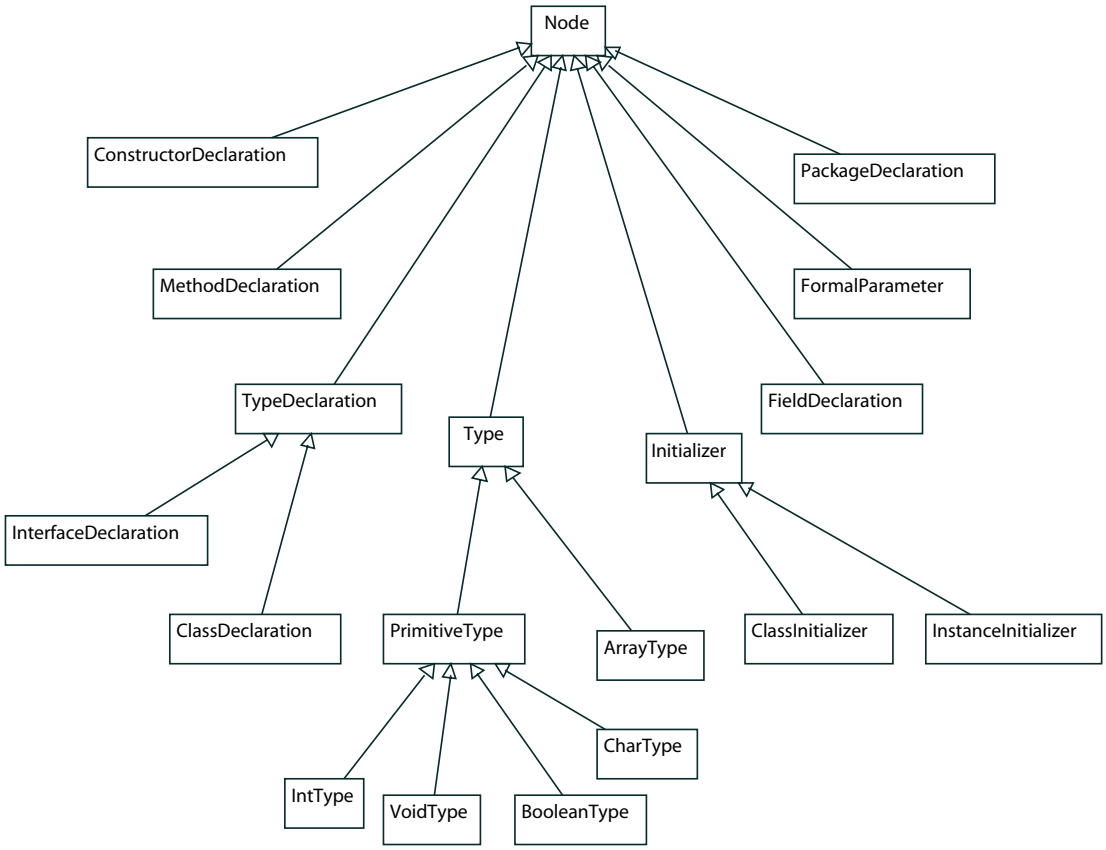

Figure 4: Several Tree classes. (Change the caption)

by Primary Expression subclasses. Some of those also implement interfaces, even more than one. Interfaces are used by DynamicJava to know when certain conditions hold. Those that implement `LeftHandSide` interface (`ArrayAccess`, `QualifiedName`, `FieldAccess`) are available to be the left hand side of an assignment. Those that implement Figure 5 shows the primary expressions classes. Its super class, Expression, is also super class for binary and unary expressions. All properties are defined by Primary Expression subclasses. Some of those also implement interfaces, even more than one. Interfaces are used by DynamicJava to know when certain conditions hold. Those that implement `LeftHandSide` interface (`ArrayAccess`, `QualifiedName`, `FieldAccess`) are available to be the left hand side of an assignment. Those that implement `ExpressionContainer` (`ObjectMethodCall`, `ReturnStatement`, `ConstructorInvocation`, `ObjectFieldAccess`, `ArrayAccess`, `UnaryExpression`) are those that need another expression to be completed. An array access needs a qualified name where to look for the data, that qualified name is the expression contained. The interface

`ExpressionStatement` is used only when DynamicJava parses the source code to build up the `ForStatement` node. `ExpressionContainer` (`ObjectMethodCall`, `ReturnStatement`, `ConstructorInvocation`, `ObjectFieldAccess`, `ArrayAccess`, `UnaryExpression`) are those that need another expression to be completed. An array access needs a qualified name where to look for the data, that qualified name is the expression contained. The interface `ExpressionStatement` is used only when `DynamicJava` parses the source code to build up the `ForStatement` node.

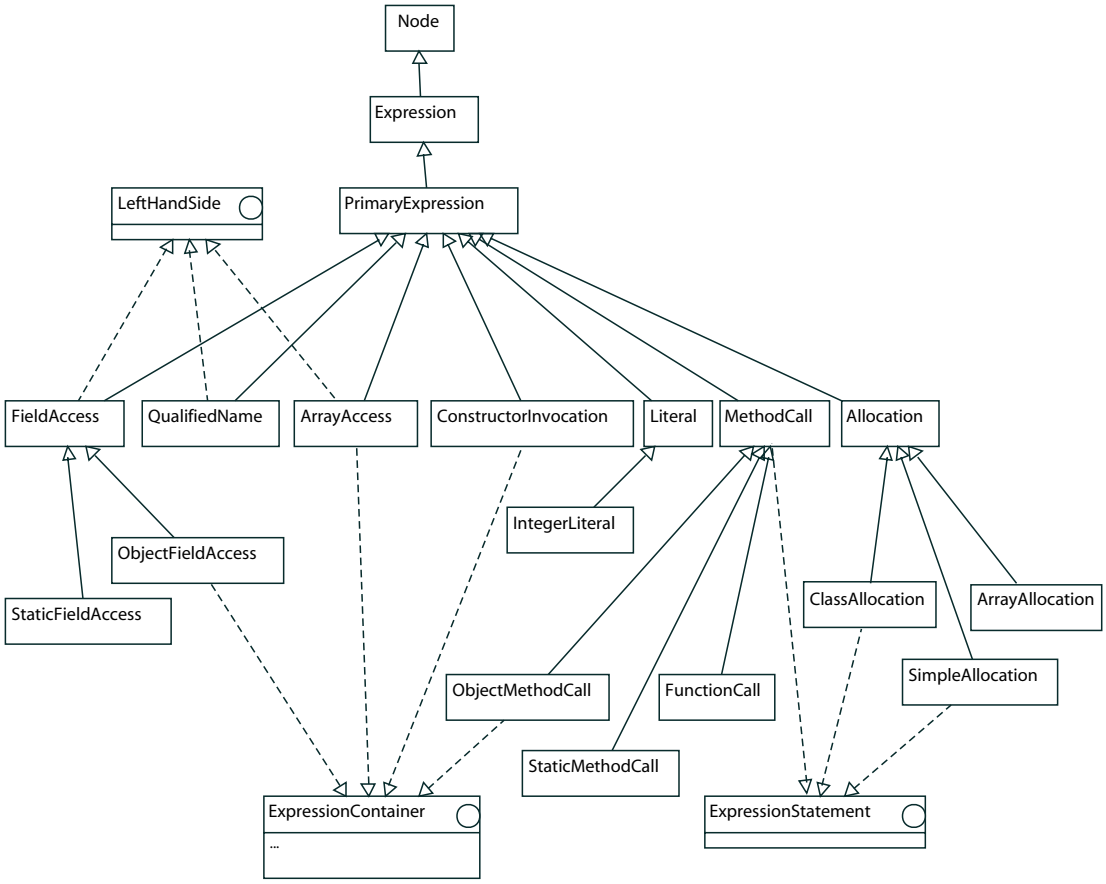

Figure 5: Primary expressions classes in `tree` package. (Change the caption)

Figure 6 contains all the possible Java statements. `ReturnStatement` is implementing `ExpressionContainer` because it needs that interface when it is returning an expression. pDo, pWhile and pFor statements are implementing the `ContinueTarget` interface in order to allow `ContinueStatements` inside of them.

Figure 7 illustrates unary and binary expressions. It only contains one as an example of each their subclasses. So `AndExpression` also refers to `OrExpression`

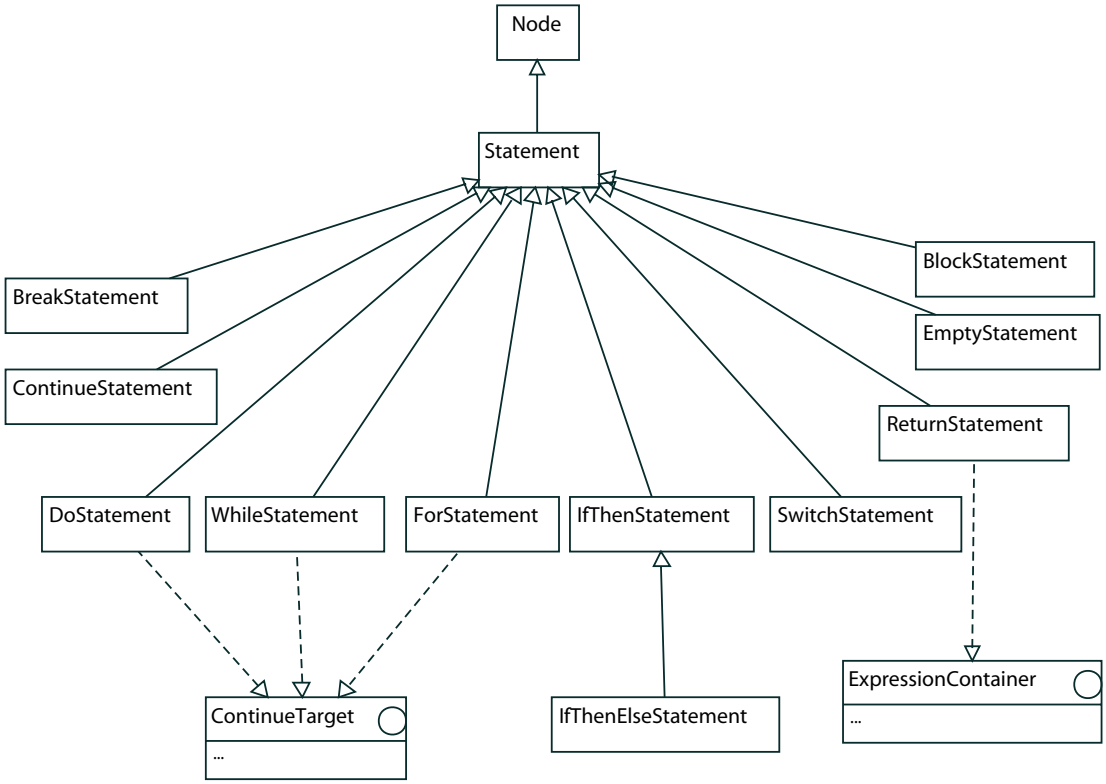

Figure 6: Statement classes in `tree` package. (Change the caption)

and so on.

3.4 Intermediate Code Interpreter

The package `jeliot.ecode` includes the classes used by the intermediate code interpreter. The class `Interpreter` contains the intermediate code interpreter and the other classes help the work with intermediate code generation, interpretation and on the other aspects of the program execution.

The intermediate code is generated by the modified version of DynamicJava. The code is then written into a pipe that can be read by the interpreter. As the code is completely machine written we know the form of the code and can interpret it easily. The intermediate code is introduced in the section 5. The code is read line by line and it is tokenized with the agreed token in the class `Code` that contains all the necessary constants for intermediate code generation. Then the first token tells what kind of statement is coming and how it should be processed. Then

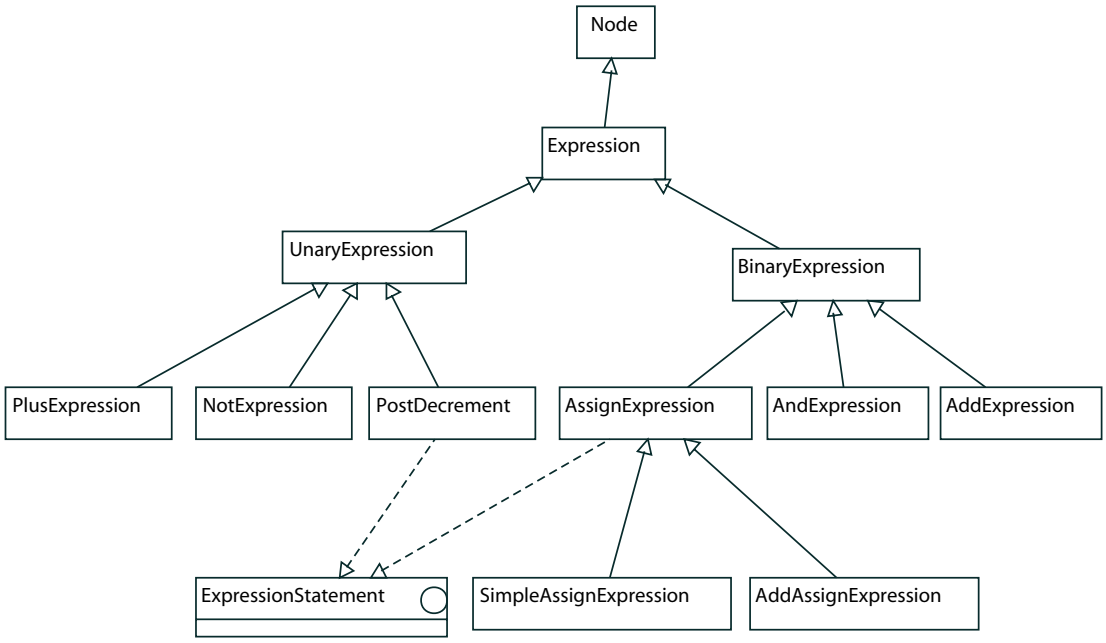

Figure 7: Unary and binary expression classes in `tree` package. (Change the caption)

the rest of the statement is processed accordingly and a part of the animation is shown if necessary.

There are several important data structures in the `Interpreter` class:

`commands` variable is a reference to a stack (`Stack`) that contains information about each read command. Command in this context tells, for instance, whether a operand of a binary operation is the left or right side of the operation.

`exprs` variable is a reference to a stack (`Stack`) containing the information about each read expression. The information consist of each expressions type (e.g. addition or subtraction operation), expression reference and the location in the code. It is used to identify which operands (i.e. values or variables) belong to which operation.

`values` variable is a reference to a hash table (`Hashtable`) that contains the values of literals, variables and operations encountered during the execution. Each of the value is stored in a form of `Value` object (containing also the `Actor`) and the expression reference is used as a key for the hash table.

These values are then used when an expression evaluation is animated.

`variables` variable is a reference to a hash table (`Hashtable`)that contains the visited variables as instances of the `Variable` class.

`instances`

`methodInvocation`

`postIncsDecs`

`currentClass`

`classes`

`currentMethodInvocation`

`objectCreation`

`returned, returnValue, returnActor, returnExpressionCounter` are variables for handling the return value of a method.

Class `ECodeUtilities` is used by the interpreter to help type identification and translating the intermediate code into the commands used in the visualization engine because they differ in some parts. It also handles some issues related to user input and communication between DynamicJava and the intermediate code interpreter.

3.5 Language Constructs

The classes of `jeliot.lang` package represent Java language constructs.

See Figure 8.

3.6 Visualization Engine

The package `jeliot.theatre` contains classes that are used by the animation component called `Theatre`. We will here introduce the most important classes in

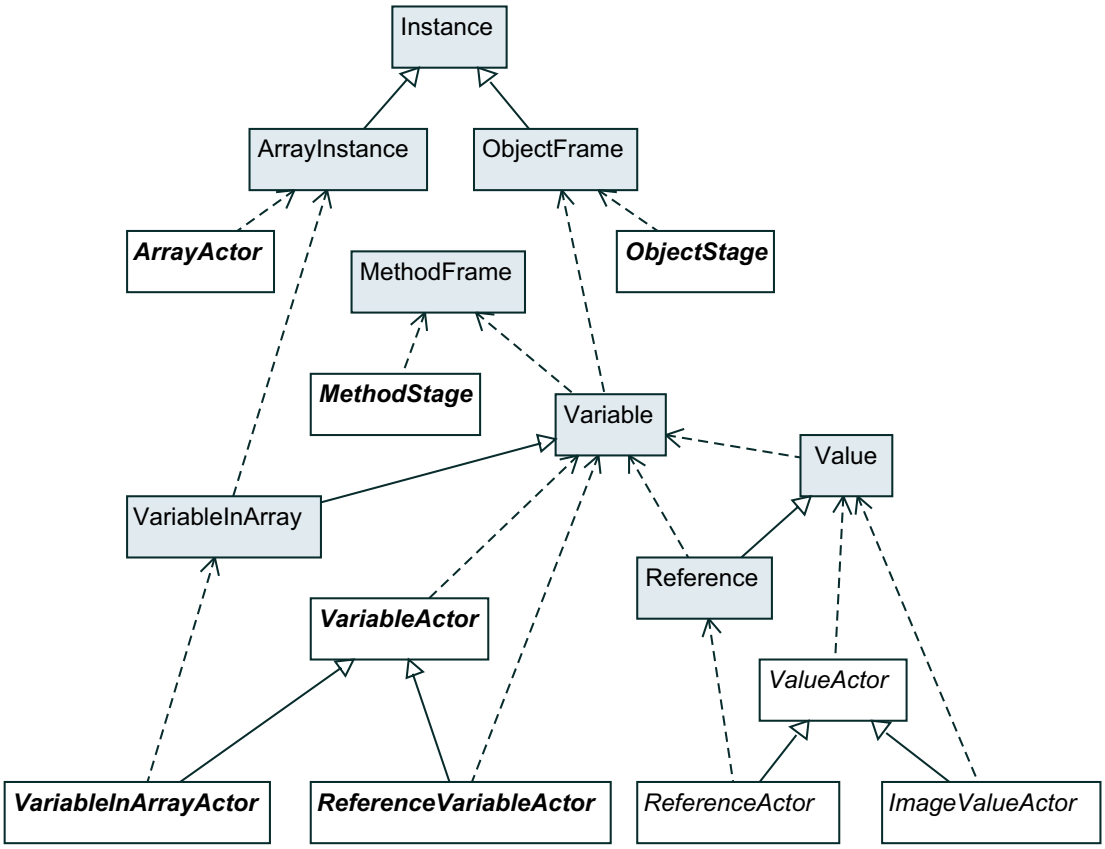

Figure 8: The class hierarchy of the language constructs (gray boxes and solid lines) and the `Actor`s (white boxes and solid lines) and their usage relations to each other (dashed lines). `Actor`s (white boxes) implementing actor container are written in bold and italics and the other `Actor`s on just italics.

the package and tell the meaning of these classes. For rest of the class see the documented source code.

See Figure 9.

3.6.1 Director class

`Director` has a methods for different kind of operations during the animation of the program (e.g. binary expression, unary expression, object creation and method call). These methods are called by the intermediate code interpreter (`ECodeInterpreter`). In each of the methods are given parameters that the method needs for the operation.

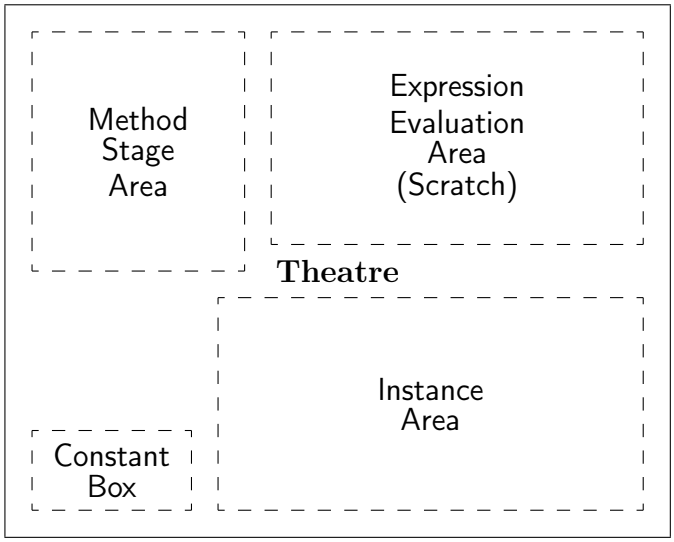

Figure 9: The structure of the animation frame (theatre) in Jeliot 3.

Normally, first the method creates actors through `ActorFactory` or uses existing actors. `Animation`s are created through `Actor`s' methods (e.g. `appear` or `fly`). Created animations are given to `AnimationEngine` to be run.Actors can be also added to other `ActorContainer`s (e.g. `Theatre`, `VariableActor` or `MethodStage`).

3.6.2 Actor classes

See Figure 10.

3.6.3 ActorContainer interface

See the Figure 11.

3.6.4 Animation class

See the Figure 13.

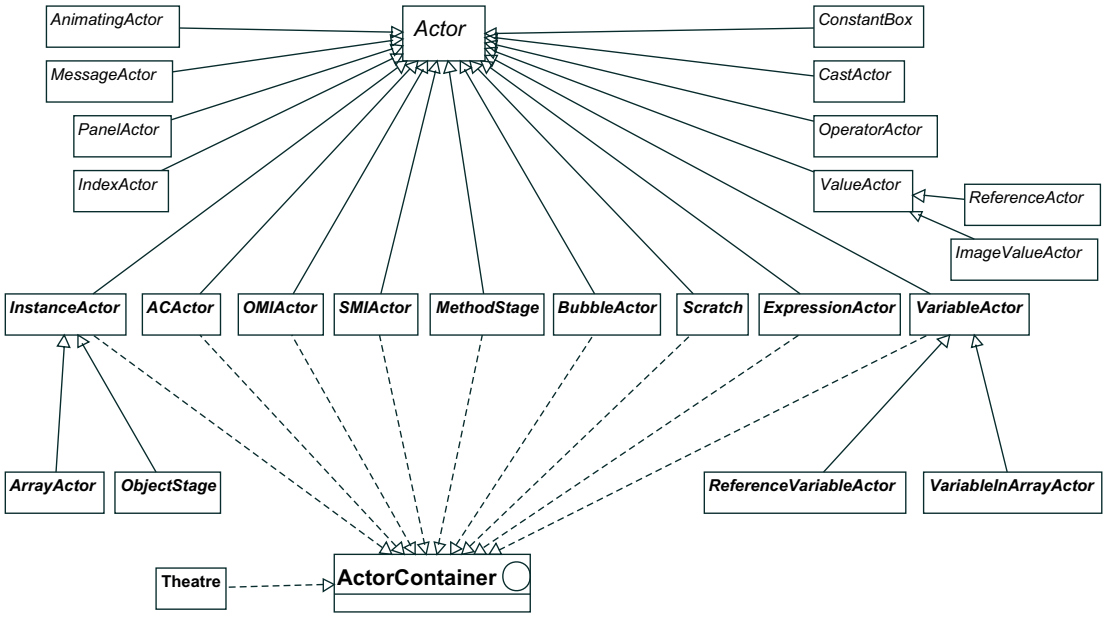

Figure 10: The class hierarchy of `Actor` class and `ActorContainer` interface.

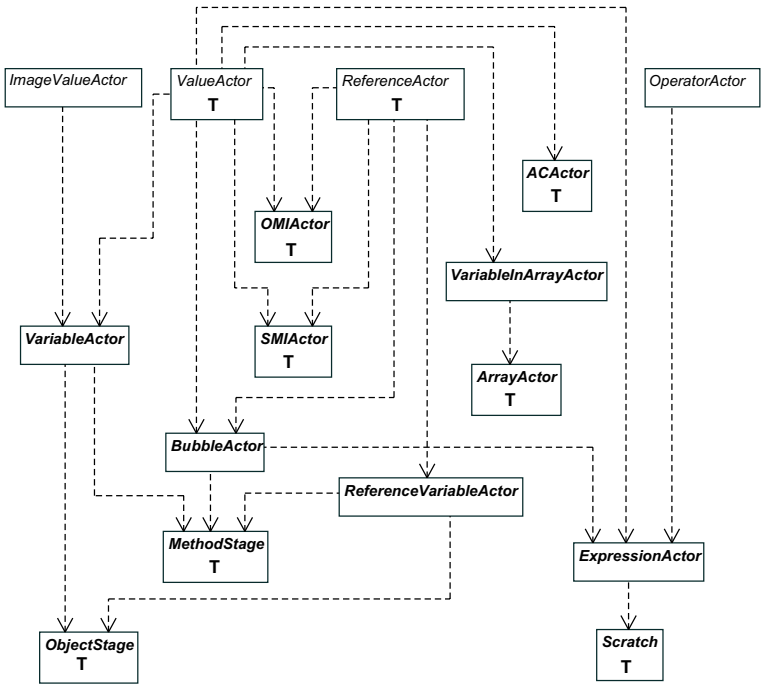

Figure 11: The inclusion relations between `Actors` and `Actors` implementing `ActorContainer` interface.

3.6.5 AnimationEngine class

See the Figure 13.

3.6.6 Theatre class

See the Figure 12.

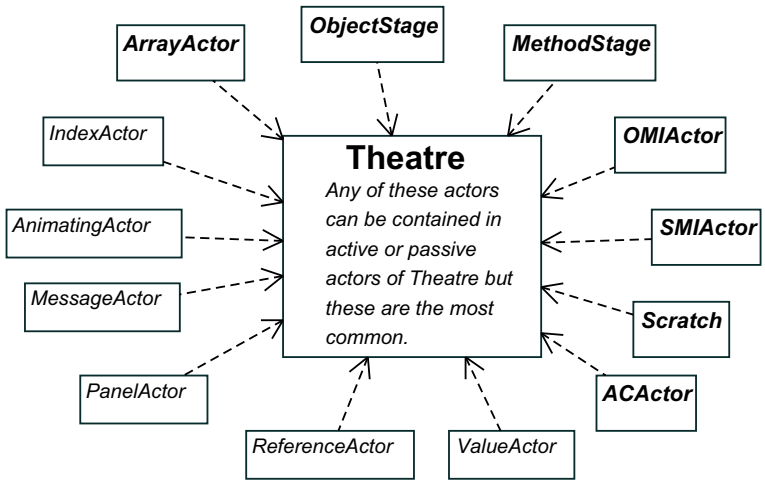

Figure 12: The `Actors` that are commonly included in the passive (`pasAct`) and active (`actAct`) `Actors`.

3.6.7 TheatreManager class

3.6.8 ActorFactory class

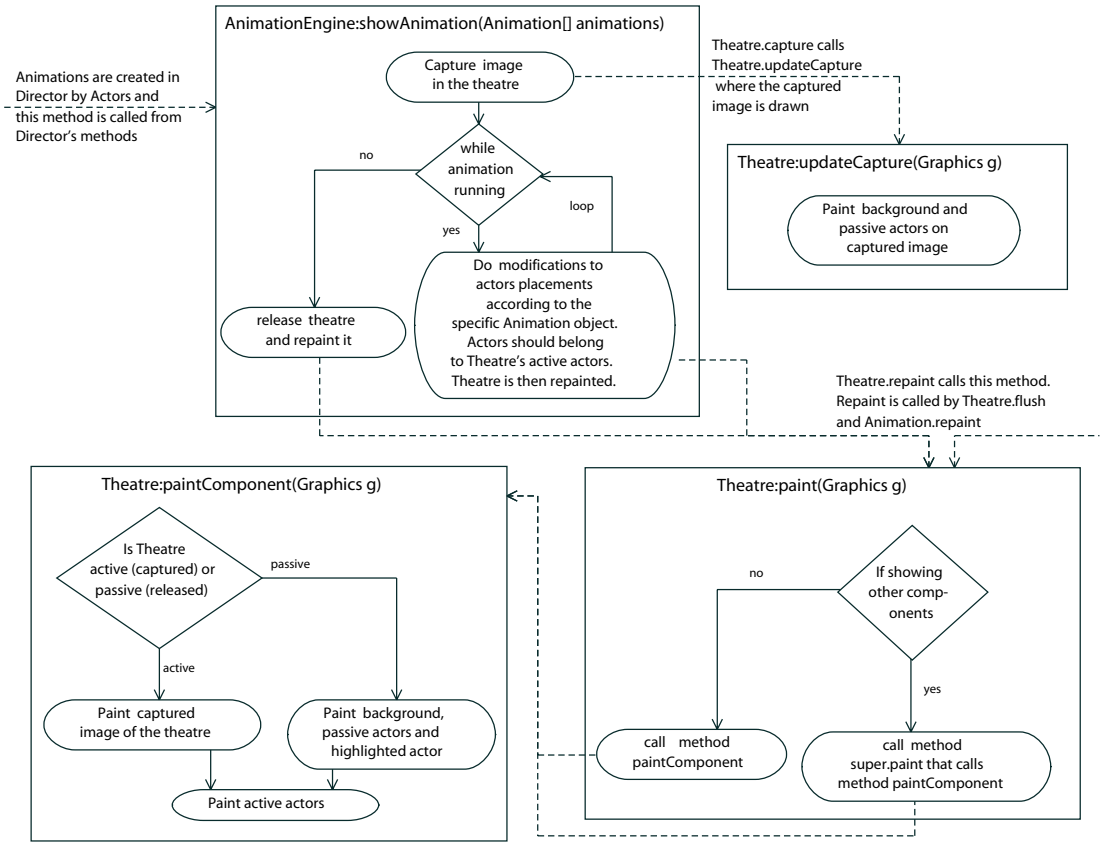

Figure 13: The structure of the animation engine in Jeliot.

4 Communication Model

Communication model here.

5 Intermediate Code (M-Code)

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 `Director`, which organizes the actors on the display. 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 rather 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`, which refers to variables previously declared and `LITERAL` which states for literal values, as numbers, characters or strings.

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

5.1 M-code production

5.1.1 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 `m-code interpreter` 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 `m-code interpreter` 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. 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. For example `a+=3-b` will be interpreted as `a=a+(-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.

5.1.2 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.

Add here what happens during the constructor call!

6 Extending Jeliot System

How do extend and what to extend?

References

Apache. Apache Ant. WWW-page, 2003a. `http://ant.apache.org` (Accessed 2003-27-10).

Apache. Apache Ant User Manual. WWW-page, 2003b. `http://ant.apache.org/manual/index.html` (Accessed 2003-27-10).

Mordechai Ben-Ari, Niko Myller, Erkki Sutinen, and Jorma Tarhio. Perspectives on Program Animation with Jeliot. In Stephan Diehl, editor, *Software Visualization, LNCS 2269*, pages 31–45. Springer-Verlag, 2002.

Koala. DynamicJava. WWW-page, 2002. `{http://koala.ilog.fr/djava/}` (Accessed 2003-16-07).