Jeliot Program Animation System
Internal Documentation

Version 3.2

Niko Myller and Andrés Moreno García

7th March 2004

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 and inner-classes.

To make these two separate systems communicate (section 4) with each other a new intermediate code (section 5 and Intermediate Language Specification) and an 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 (Ben-Bassat Levy et al., 2003).

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 documentation and supports inheritance, supports: inheritance of classes and `super()` method calls at the beginning of a constructor.

Version 3.1 supports objects creation and object method calls. Line numbering was added for source code editor and viewer. Supports: User made classes (inheritance is not yet supported), constructor calls, object method calls, object field accesses.

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. Supports: Values of type `String` and all primitive types, one-dimensional arrays with primitive types or strings as their 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 invocations, including recursive invocations.

Not implemented:

- Static variables.
- Calls to `super(...)` method with parameters.
- Super field accesses.
- Arrays with components of reference type (except `String`)
- Two or more dimensional arrays.
- Conditional expressions (`exp?exp1:exp2`).
- Array initializers.
- Java 2 SDK API classes' methods cannot return object (except `String` type) or array types (e.g. `object.getClass()` that returns a `Class` instance).

2 Managing the Jeliot System

In this chapter we explain the system requirements of Jeliot, the files and directories in the distributions and how the system can be built form the source code code distribution can be build.

2.1 Source Code Distribution

2.1.1 Directory Hierarchy

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

`lib` Contains the tools needed for the automated build process.

`resources` Contains messages used by DynamicJava.

`src` Contains the source code for Jeliot. It is divided into the following subdirectories.

- `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` 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.

2.1.2 How to Build Jeliot 3 From the Sources

Jeliot uses a 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 in the classes subdirectory. To run Jeliot from this point you should enter the command java jeliot.Jeliot inside the classes subdirectory.

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

3 The Structure of Jeliot System

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

The functional structure of the Jeliot 3 is shown in the Figure 1. A user interacts with the user interface and creates 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).

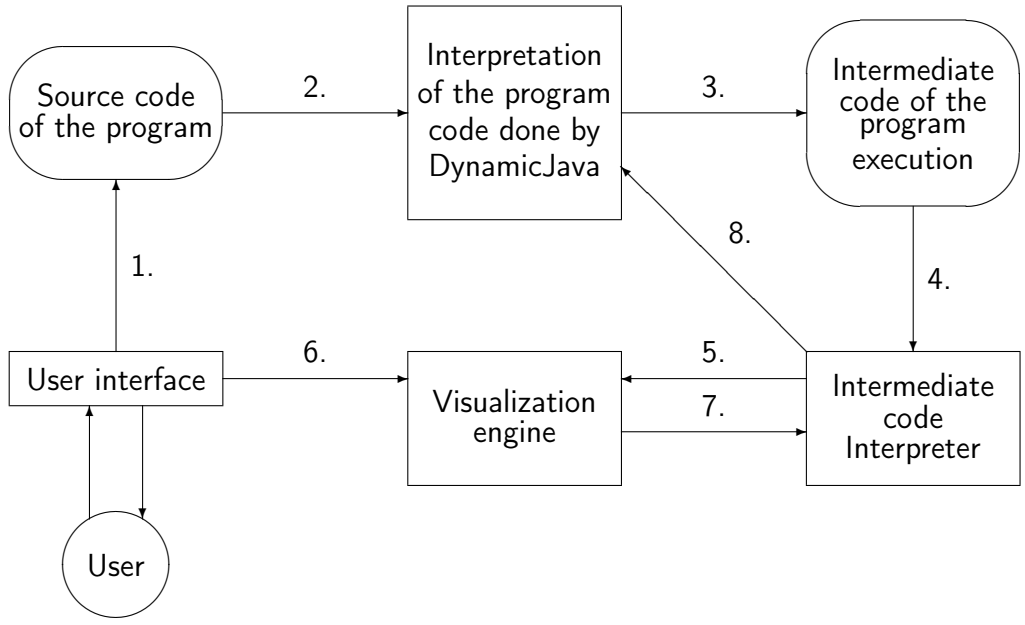

Figure 1: The functional structure of Jeliot 3.

Different packages of Jeliot 3 are introduced in this section in the same order as in the functional structure.

3.1 Class Jeliot

Class Jeliot in the package `jeliot` is the main class of the program. It combines the different components of the program together and handles the communication between different components. 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.theater`) classes. In addition to that it invokes the thread handling the interpretation of the user programs with the `jeliot.launcher.Launcher` class.

3.2 User Interface

The user interface of Jeliot is located in the package `jeliot.gui`. The structure of the user interface and the actual user interface are shown in the Figures 2 and 3. 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 the control panel (created in `JeliotWindow`) and the output console (`OutputConsole`) is added in the south (bottom) border.

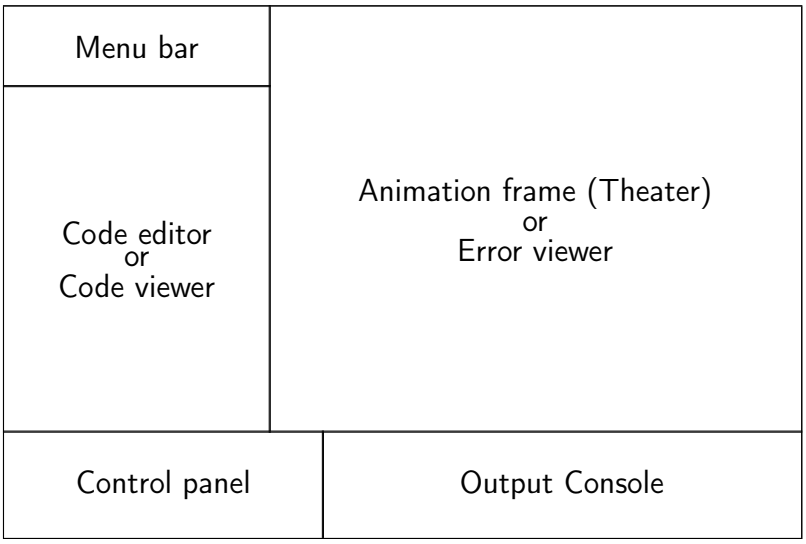

Figure 2: The structure of user interface in Jeliot 3.

In the split pane the left side is used by the code editor (`CodeEditor`) or the code viewer (`CodeViewer`). The code editor is shown during editing of the program

and the code viewer during the animation of the program. Both of the text components have a line numbering component (`LineNumbers` on their row header). The code editor consist of an editing panel that has buttons to load, save and edit the source code. The frame has also a menu bar that is constructed partially in the `CodeEditor` class and partially in the `JeliotWindow` class.

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

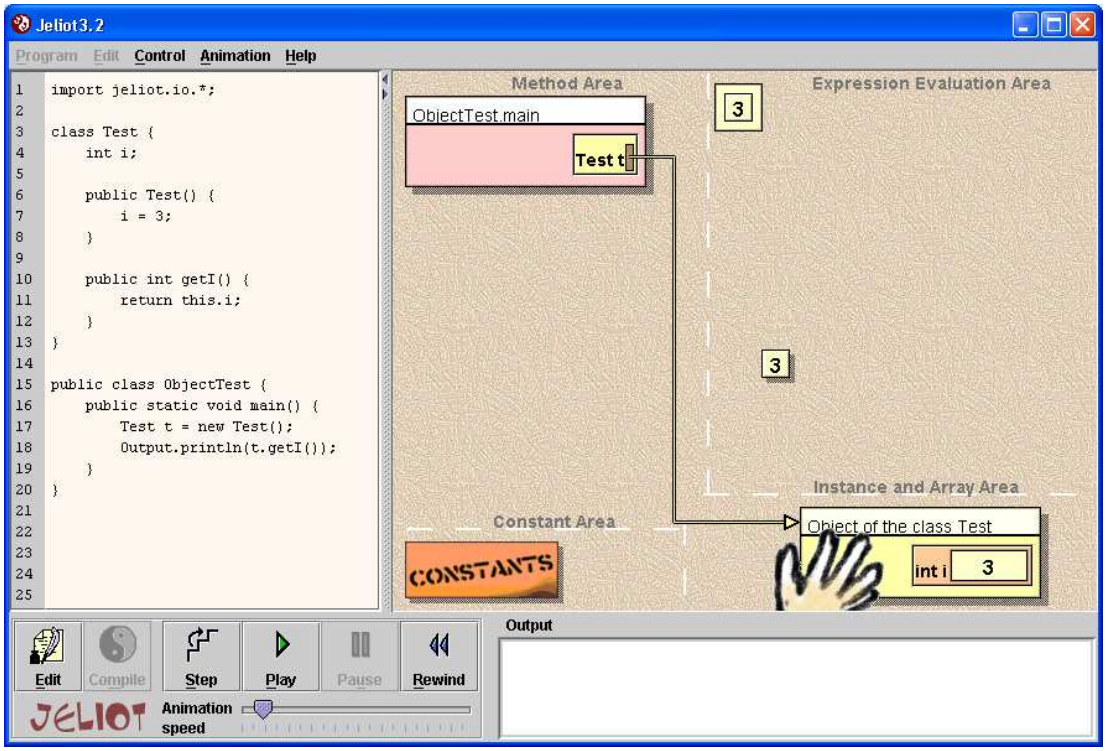

Figure 3: The user interface of Jeliot 3.

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

There are also three classes that are not currently in use in Jeliot 3, namely `LoadJeliot`, `DraggableComponent` and `TheaterPopup`. LoadJeliot class could be

used to show a splash screen in the startup of the program. DraggableComponent is an class that a DraggableComponent could extend to be dragged during the visualization. TheaterPopup class could be used to have a popup menu on the visualization frame. This popup menu could show, for example, methods of the object.

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`. This parser has been 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.

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

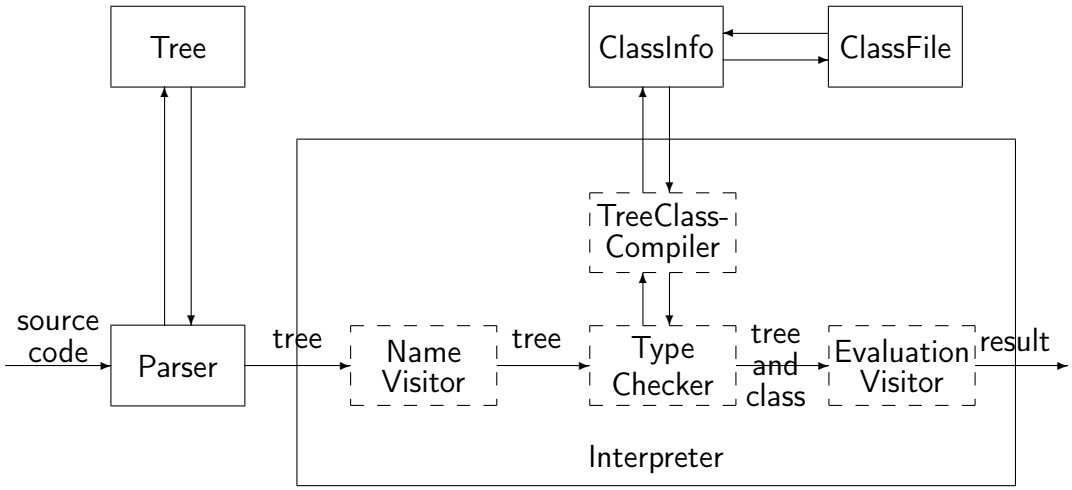

Figure 4: Packages, visitors and data flow in DynamicJava

As you can see, 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 for this behavior will be explained in the next paragraphs.

The NameVisitor is a tree visitor that resolves the ambiguity in identifiers in a syntax tree. 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 concerned 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 generated 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 contains a class for every different Java language construct. These classes build up the syntactical tree from the source code. These classes are also implemented as a tree. The root of this tree is the class `Node`. Every other class inherits it, directly or indirectly. `Node` only defines one property, the location of the node in the source code, both the filename and the position inside the file. Jeliot only accept single file programs, thus the filename is not needed. Most of the other classes also define properties that later are inherited by more specific classes.

The tree structure is shown in the Figures 5, 6, 7 and 8. These figures do not try to be exhaustive. Some classes have been taken away to not clutter the diagrams and because they were not needed to understand the tree structure.

Figure 5 shows the different nodes for declarations, initializers and types. Two main types are used by DynamicJava: Array type and and primitive types (`int`, `char`,...). The type String is not considered as a different type as it is a class. It is Java that supports it by normal String methods.

Figure 6 shows the primary expressions classes. Its super class, `Expression`, is also super class for binary and unary expressions. All properties are defined by `PrimaryExpression` subclasses. Some of those also implement interfaces, even more than one. Interfaces are used by DynamicJava to know when certain conditions hold. Nodes 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 complete and

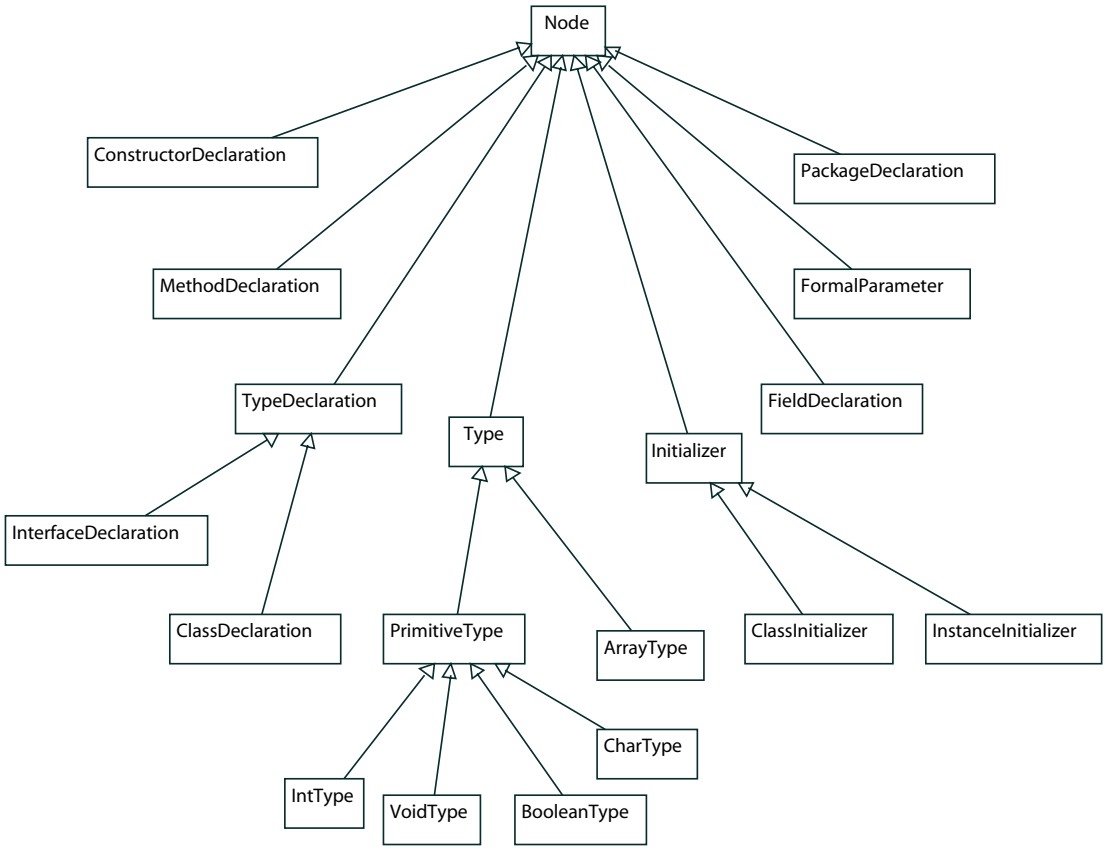

Figure 5: Several Tree classes. (Change the caption)

meaningful. For example, an array access (`buffer[3]` needs a qualified name (`buffer`) where to look for the data, that qualified name is the expression contained in `ArrayAccess`. The interface `ExpressionStatement` is used only when DynamicJava parses the source code to build the `ForStatement` node.

Figure 7 contains all the possible Java statements. `ReturnStatement` implements `ExpressionContainer` because it needs, that interface when it is returning an expression. `Do`, `While` and `For` statements implements the `ContinueTarget` interface in order to allow `ContinueStatements` inside of them.

Figure 8 illustrates unary and binary expressions. It shows one as an example of each their subclasses. So `AndExpression` also refers to `OrExpression` and so on.

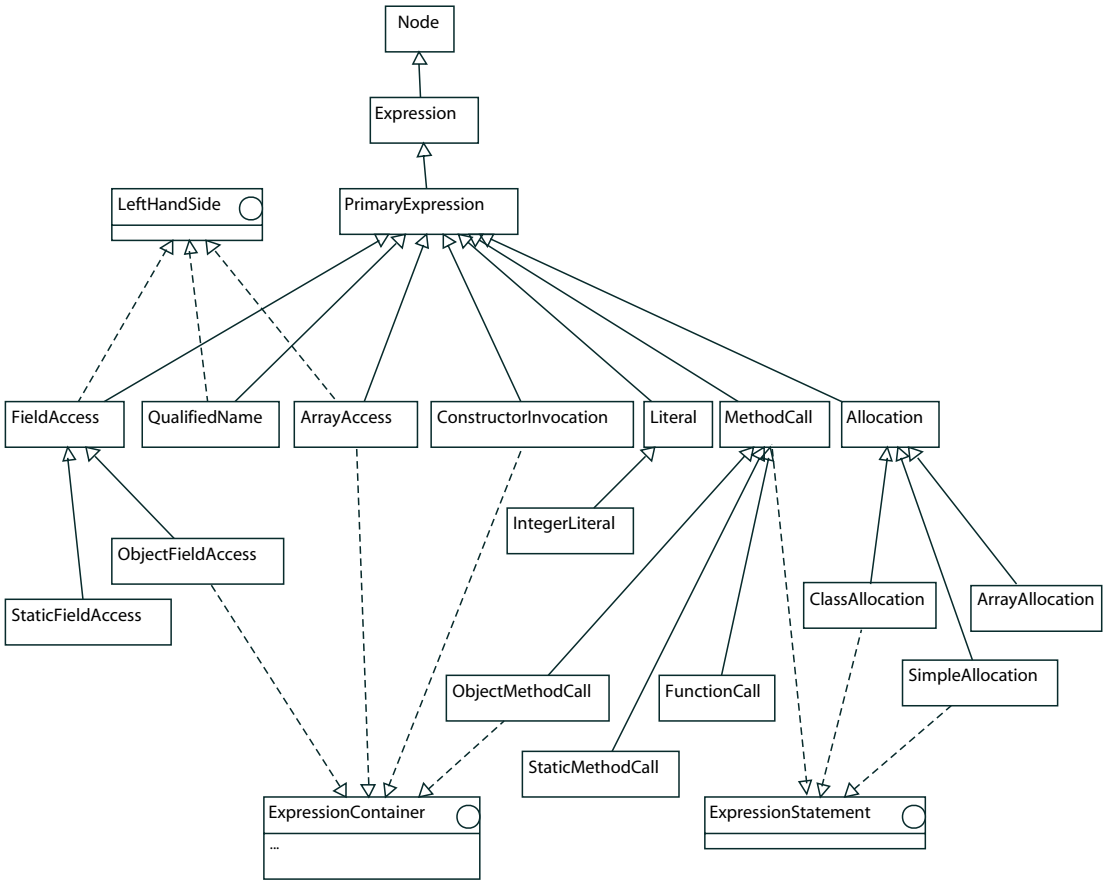

Figure 6: Primary expressions classes in `tree` package. (Change the caption)

3.4 Intermediate Code Interpreter

The package `jeliot.mcode` includes the classes used by the intermediate code interpreter. The class `Interpreter` contains the intermediate code interpreter and the other classes that 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. The rest of the statement is processed accordingly and a part of the animation is shown if

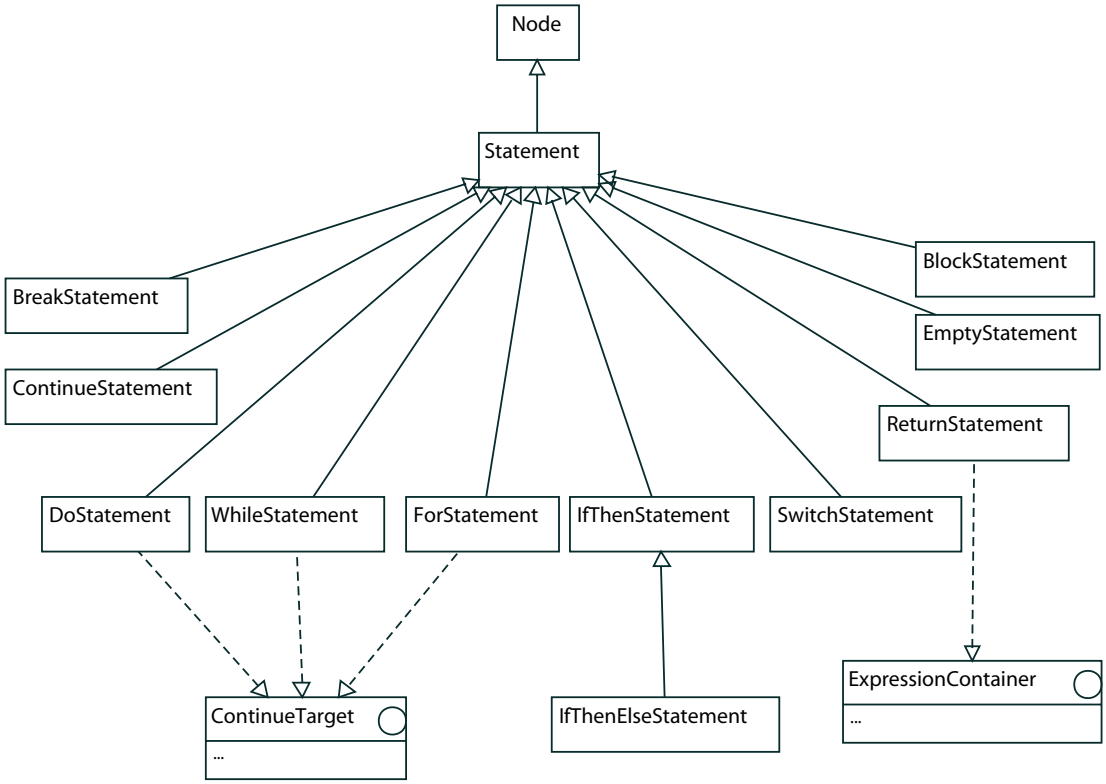

Figure 7: Statement classes in `tree` package. (Change the caption)

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 command read. 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 expression read. The information consists of each expression's type (e.g. addition or subtraction operation), reference and 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 values is stored in a form of `Value` object containing also

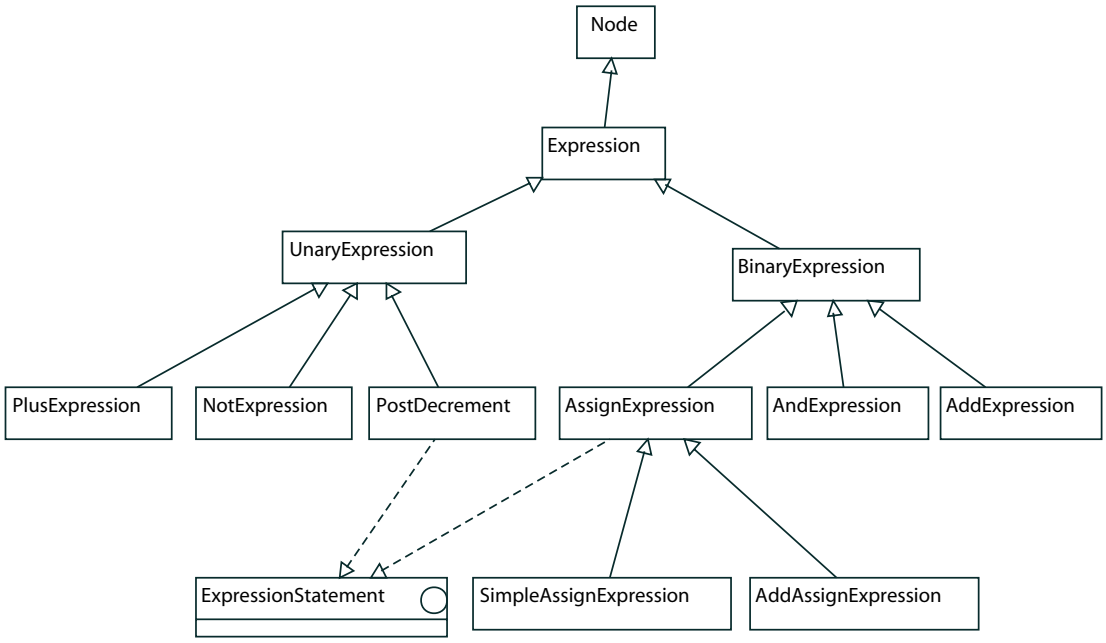

Figure 8: Unary and binary expression classes in `tree` package. (Change the caption)

`ValueActor` object of that value if it is already available. 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. Each instance also contains `VariableActor` instance as an attribute. The expression references are used as the keys in the hash table. These values are used for expressions when a variable reference is needed and the value of the variable is not enough, for example, in pre- and postincrements.

`instances` variable is a reference to a hash table (`Hashtable`) that contains the instances (an object of `Instance` class or its subclasses) that are currently instantiated in the program. The keys of the hash tables are the hash values of the instances, which means that the instances used in the program must generate unique hash values.

`currentMethodInvocation` variable is a refernce to an array which has a component type `Object`. Array `currentMethodInvocation` keeps track of all the information about the method invocation. The cells of the array contain

the following information: method name (`String`), Class/Object expression (`String`), Parameter values (Array of `String`s), Parameter types (Array of `String`s), Parameter names (Array of `String`s), Highlight information for invocation (`Highlight`), Highlight information for declaration (`Highlight`), Parameter expression references (`Integer`), Object reference if method is constructor or object method (`Integer`).

`methodInvocation` variable is a reference to a stack (`Stack`) that keeps the information of the method invocations both the current and the previous but still active invocations. Each time a method invocation is closed the stack is popped and the previous invocation record is set as the new `currentMethodInvocation`.

`postIncsDecs` variable is a reference to a hash table (`Hashtable`) that keeps the information about all the post increments or decrements that should be performed. This information is collected when a intermediate code command for post increment or decrement is read. The expression reference is used as the key in the hash table. During every expression involving variables, it is tested if the post increment or decrement should be animated after the variable value is used in the visualization.

`currentClass` variable is a reference to an instance of `ClassInfo` class that contains the information about the constructors, methods and fields that each class has.

`classes` variable is a reference to an instance of `Hashtable` that contains the `ClassInfo` instances of all the classes that are created.

`objectCreation` variable is a reference to an instance of `Stack` that contains the hash code of the current object allocation during the constructor call.

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

Class `MCodeUtilities` is used by the interpreter to help type identification and translating the intermediate code into the commands used in the visualization engine. For example, visualization engine uses same numbers for binary and unary expression and the difference is made with the method call, whereas in intermediate code binary and unary expressions have different numbers as their codes.

`MCodeUtilities` 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. They are mainly used to store and transfer the information about these language construct for the creation and storing of the corresponding `Actor`s (see section 3.6.2). The Figure 9 shows the class structure of the language construct classes and the corresponding `Actor`s that are used as attributes of the language constructs.

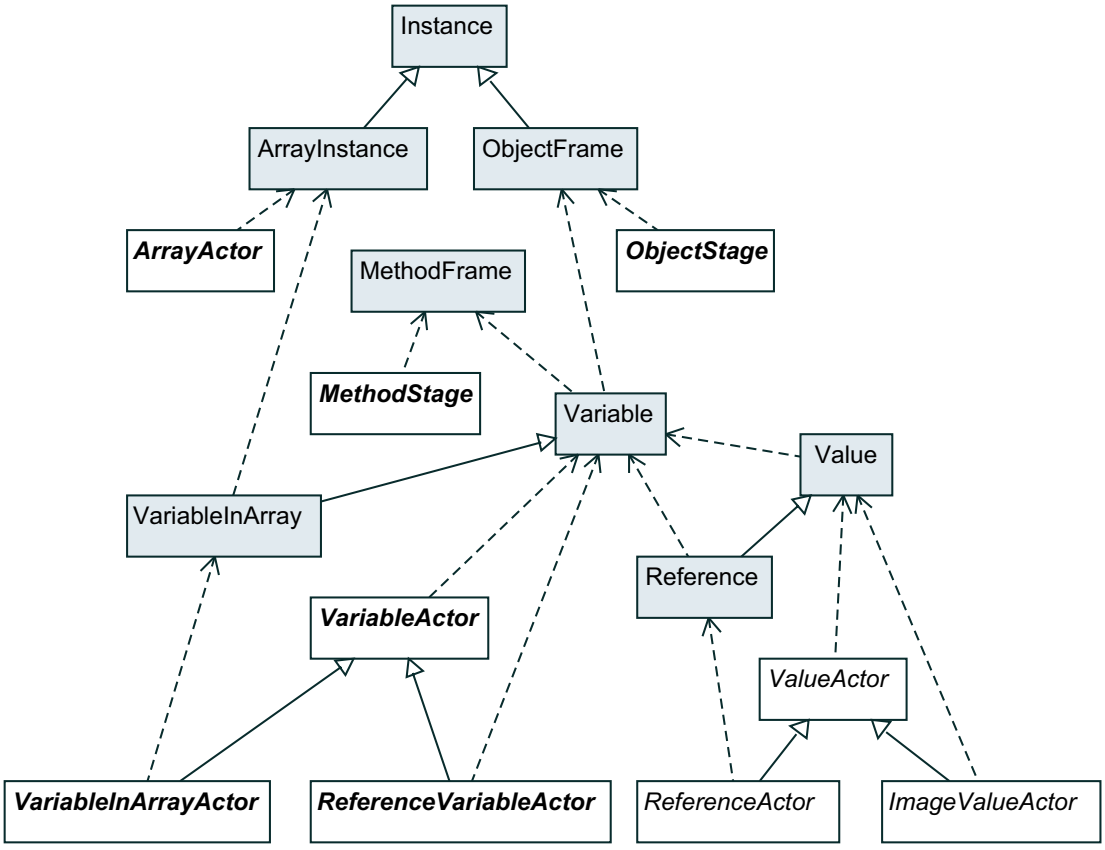

Figure 9: The class hierarchy of the language constructs (gray boxes and solid lines) and the `Actor`s (white boxes and solid lines) and their usage relations to each other (dashed lines). `Actor`s (white boxes) implementing actor container are written in bold and italics and the other `Actor`s on just italics (see section 3.6.2).

`Instance` class is the base class for all the instances (e.g. `ArrayInstance` and

`ObjectFrame`). The instance of the `ArrayInstance` class represents an array of n-dimensions. The instance of the `ObjectFrame` represents an object of a class that is created at runtime.

The instance of the `ClassInfo` class contains the information about a single Java class. It contains all the fields, methods and constructors of the class.

The instance of the `MethodFrame` class represents a method under execution. A method frame is created runtime each time a method is called.

The instance of the `Variable` class is an instance of a variable — a field or a local variable. A new variable is created runtime every time a local variable is declared. The instance of the `VariableInArray` is an instance of an array variable. A new array variables are created at runtime every time a new array is created.

The instance of the `Value` class represents any primitive type of value and values of String type and is the base class for reference values. The instance of the `Reference` is a value of a reference type meaning that all the references to the instances are objects of this class.

3.6 Visualization Engine

The package `jeliot.theater` contains classes that are used to form the animation of the program. We will here introduce the most important classes in the package and explain the meaning of these classes. For rest of the classes see their source codes.

3.6.1 Director class

`Director` has methods for different kinds 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 (`MCodeInterpreter`). In each of the methods are given parameters that the method needs for the operation.

Normally, the method first 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 the AnimationEngine to be run. Actors can be also added to other ActorContainers (e.g. Theater, VariableActor or MethodStage).

3.6.2 Actor classes

There is a large number of different Actor classes as can be seen from Figure 10.

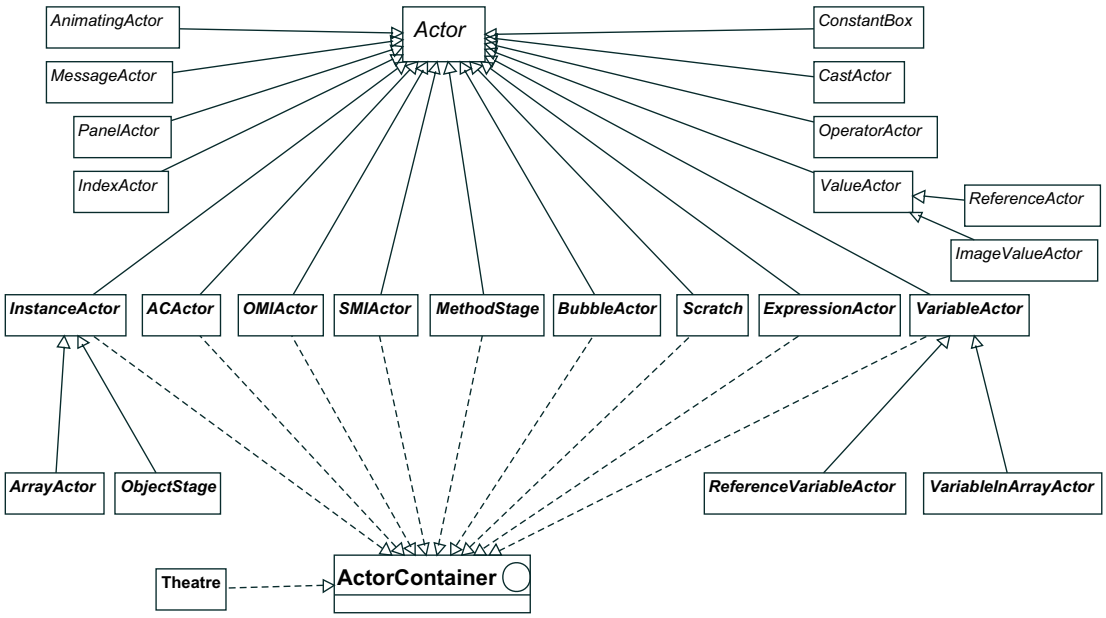

Figure 10: The class hierarchy of Actor class and ActorContainer interface.

Actor class is the base class for all the actors. The Actor class should not be used directly but indirectly with it's subclasses.

ValueActor is an actor that represents graphically the language construct Value. The Value's type is represented by the colors of the ValueActor and the Value's value is printed out as the String representation of the value. ImageValueActor is an actor that is used when a value actor should be an image. At the moment this only happens when the value of a variable is unknown visualized as "???". ReferenceActor shows the reference to some InstanceActor (e.g. ArrayActor or ObjectStage). They can be assigned to the ReferenceVariableActor instances or any other instance that is derived from the ReferenceVariableActor.

An instance of the OperatorActor class represents a operator in the expressions.

It can be a binary or unary operator and it is shown in the ExpressionActor with the operands.

Scratch controls the expression evaluation area. It allocates the space for each ExpressionEvaluationActor and possibly for other Actors that are there temporarily. ExpressionActor represents a single line of a scratch the evaluation area. It can contain any number of different Actors (e.g. ValueActor, ReferenceActor, OperatorActor, SMIActor, OMIActor or BubbleActor) inside that it renders.

VariableActor represent graphically the language construct Variable for primitive data types and Strings. ReferenceVariableActor represents graphically the variables of the reference type. It can bind ReferenceActor instances and render them. VariableInArrayActor represent graphically the language construct VariableInArray for primitive data types and Strings.

SMIActor represents graphically the static method invocation. The actor shows the method name and the parameters in a similar way as Java syntax just replaces the variable references with their actual values. OMIActor represents graphically the object method invocation. The actor shows the object reference, the method name and the parameters in a similar way as Java syntax just replaces the variable references with their actual values.

MethodStage is the graphical representation of the MethodFrame. It contains the local VariableActors and handles the scope changes.

InstanceActor is a base class for all the instances: ArrayActors and ObjectStage. An instance of this class should not be instantiated. ObjectStage is the graphical representation of the ObjectFrame. It contains the field of the object as VariableActors. ArrayActor represents the array instance and contains VariableInArrayActors for every index of the array. IndexActor shows the line between the array access' indexing expression result value and the array's actual index.

BubbleActor is used to move the return value from the MethodStage to the ExpressionActor on the Scratch (evaluation area).

CastActor handles the animation of the casting of the primitive values.

`ConstantBox` instance represents a place where all the literal constants appear during the animation.

`AnimatingActor` is a actor that may be used to show frame based animation. The output animation is currently uses this actor.

The `LinesAndText` actor draws the dashed lines and titles to separate explicitly the theater (animation frame) into four areas: constant area, method area, object and array area and expression evaluation area.

`MessageActor` shows all the textual messages to the user.

`PanelActor` represents the curtains of the theater and produces the opening and closing animations of the curtains.

3.6.3 ActorContainer interface

`ActoContainer` interface is implemented by classes that are going to contain `Actor`s as their fields and take care of their painting. This means that those classes having actors as their fields but not taking care of the painting of the actors are not implementing this class. The implementing classes are shown in Figure 10. The different containing relationships are illustrated in Figure 11.

3.6.4 ActorFactory class

This class handles the centralized creation of all the `Actor`s. This enables the centralized appearance handling for fonts, color and other appearance parameters. If the appearance parameters of the actors are going to be changed this class should be consulted first before any modifications to the actual `Actor` classes are made.

3.6.5 Animation class

`Animation` class represents one atomic animation in `Jeliot`. Animation means here any event that includes movement of actors or that is otherwise dependent of time. Examples of animation include moving an actor from one place to another

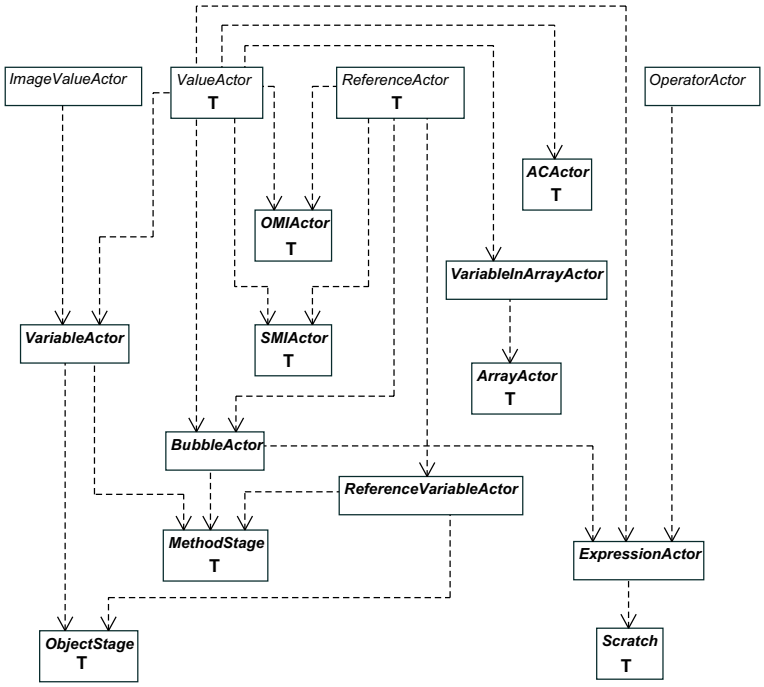

Figure 11: The inclusion relations between `Actor`s and `Actor`s implementing `ActorContainer` interface.

or flashing the colors of an actor as it is introduced to the theatre.

The animation is played by an instance of `AnimationEngine` class. The animation engine takes care of scheduling the animation. Animation class is the abstract superclass of various specialized animation classes. These subclasses must implement the `animate` method in which they make their changes to actors. The animation engine calls this method at even time intervals. If the animation has to do something in prior to starting the animation, especially if it has to set any parameters that depend on the duration of the animation or it has to add any actors to the theater, it may do this in its `init()` method. When the animation finishes, the engine calls its `finish` method.

See from Figure 12 how the `Animation` class relates to the animation formation.

3.6.6 AnimationEngine class

`AnimationEngine` schedules the animations represented by instances of `Animation` class. The engine is given an animation or an array of animations, and it

plays those animations. The speed and quality of the animation can be controlled by setting its volume (speed) and FPS (Frames Per Second) values. An engine's volume is the amount of action it gives to the animation objects each second. The higher the volume, the faster the animations will play.

An animation engine may be assigned a `ThreadController` instance. In this case, the engine checks with the controller after every step of animation calling its `checkPoint` method.

See from Figure 12 how the `AnimationEngine` relates to the animation formation.

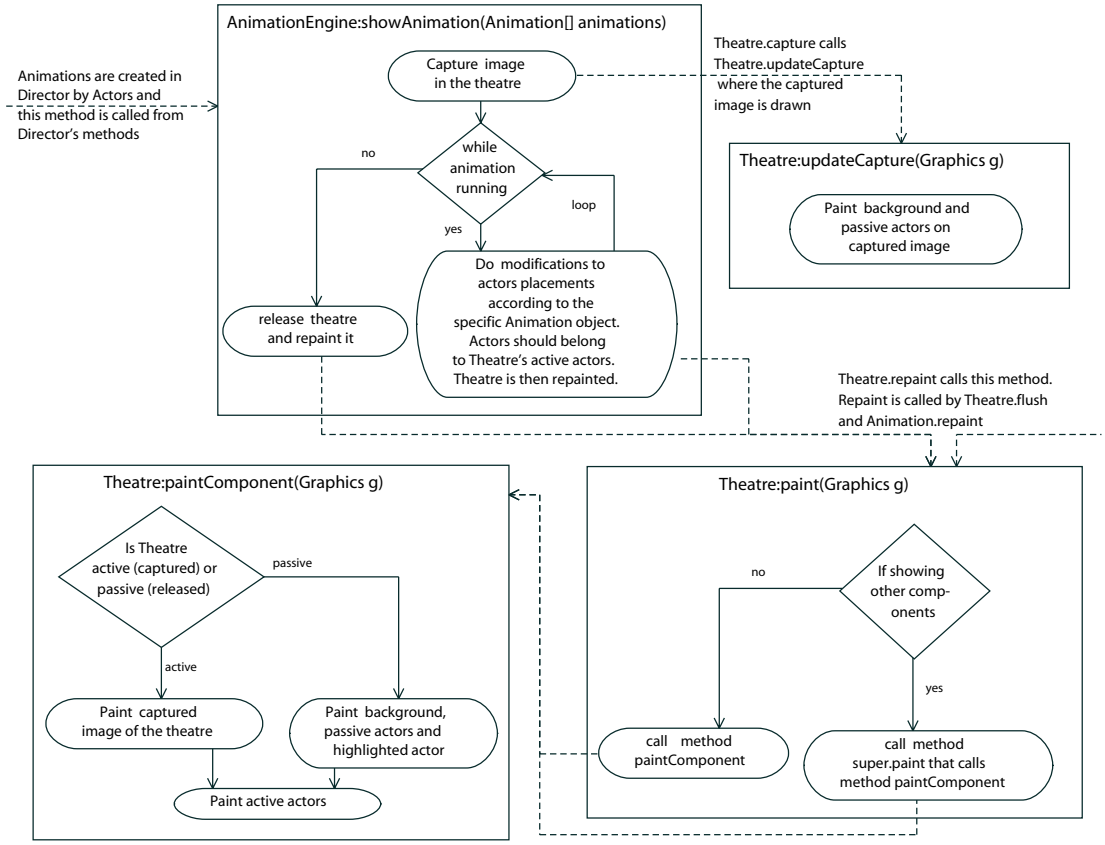

Figure 12: The structure of the animation engine in Jeliot.

3.6.7 Theater class

The `jeliot.theater` package contains a class `Theater` that extends `JComponent` class so that it can be entered inside the split pane in the user interface. It is the canvas for the animation and all the actors are drawn on it. The different

actors of the theater are shown on the theater in different areas that are shown in Figure 13. Furthermore, the `TheaterManager` allocates the places for the different actors on these areas.

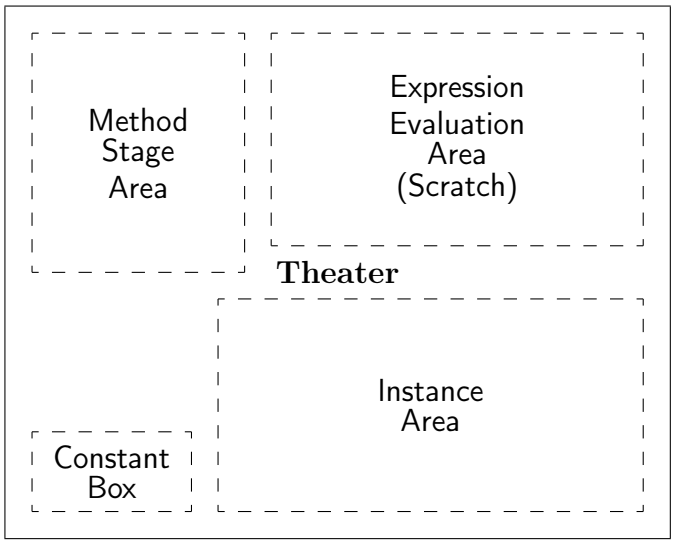

Figure 13: The structure of the animation frame (theater) in Jeliot 3.

`Theater` instance is a special case of an `ActorContainer` because actually `Theater` is not a subclass of the `Actor` class but still is implements `ActorContainer` interface. The `Theater` contains two different kinds of `Actor`s. The other actors are active (stored into `Vector actAct`) and the other actors are passive (stored into `Vector pasAct`). The active actors are the one that are drawn every time again when the new round of the animation is run. The passive actors are only stored in the background image in the beginning of the animation to optimize the drawing during the animation. See Figure 14 for those actors that are inserted as active actors during the animations.

See from Figure 12 how the `Theater` relates to the animation formation.

3.6.8 TheaterManager class

`TheaterManager` allocates the space for all `InstanceActor`s, `MethodStage`s, `Scratch`es and constants (`ConstantBox`), and also listens the `Theater` component for resizes so the the allocation of the space is valid after resizing of the `Theater` component.

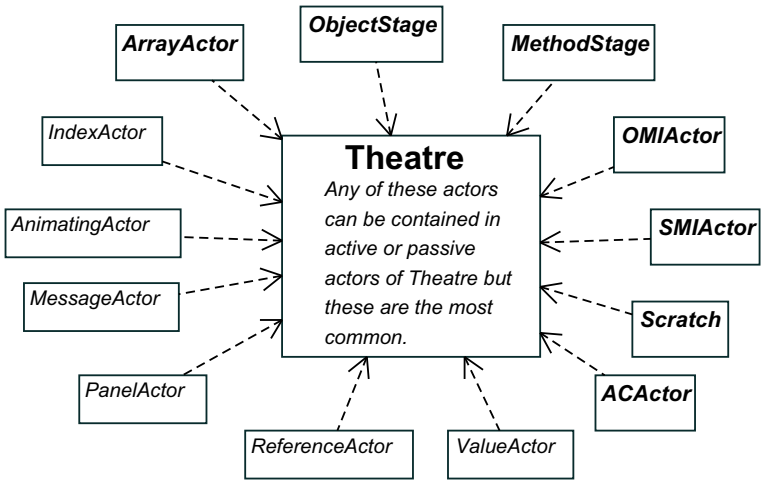

Figure 14: The `Actors` that are commonly included in the passive (`pasAct`) and active (`actAct`) `Actors`.

4 Communication Model

Communication model here. Work for Andrés

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 visualization 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 names chosen are closely 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 (Moreno and Myller, 2004).

5.1 M-code production

5.1.1 Evaluation Visitor

As previously noted, `EvaluationVisitor` was the main class that needed to be modified. In the next subsections, we will describe how m-code is produced for certain subsets of Java expressions and statements:

Static Method Call

The visitor method `StaticMethodCall` is the entry point to the evaluation visitor. It is the method called when invoking the main method of a class from `TreeInterpreter`.

Here is where I/O management takes place. 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`. So, to obtain the I/O method calls, we first ask for the declaring class when visiting a static method call [`public Object visit(StaticMethodCall node)`],

- If it is an `Input` class we ask the `m-code interpreter` to provide the information requested by ways of one pipe that communicates with both sides. We discriminate the type by the method name. Every input method has an equivalent in `MCodeUtilities`, 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 `ReturnStatement` 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` of these types. 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 the `StaticMethodCall` and `ReturnStatement` methods to manage recursive method calls (e.g. `return object.method()`). A reference number is pushed onto 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 having two different behaviors is that DynamicJava throws an `ExecutionError` when visiting an uninitialized qualified name. That occurs when an assignment method (as `SimpleAssign-Expression`) visits its left hand side for visualization purposes before it has a value. When the `evaluating` flag is false, DynamicJava invokes the `display` method to avoid unnecessary exceptions. However, both methods (`visit` and

`display`) produce the same m-code.

Variable Declaration

Variable declaration does not cause large 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, we will explain how a `while` statement produces its m-code. Other statements work in a similar way.

Firsts of all, the `WhileStatement` 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, so the generation of m-code is straightforward. However, as there is no expression to be referred, there is no node containing that constant, 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 will be. 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 as the compound assignment. As a result, 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 of DynamicJava: `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 needed to interpret 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. 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 calls!

6 Extending Jeliot System

Jeliot 3 is based on the idea that the interpreter and the visualization engine are separated with an intermediate code of the program execution. This allows the extension of the system by creating new visualization for the intermediate code. This section tries to help the extension or the modification of Jeliot 3 if that is needed. Currently, we have not added any subsection here for the extensions.

References

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

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

Ben-Ari, M., Myller, N., Sutinen, E., Tarhio, J., 2002. Perspectives on Program Animation with Jeliot. In: Diehl, S. (Ed.), Software Visualization, LNCS 2269. Springer-Verlag, pp. 31–45.

Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P. A., 2003. The Jeliot 2000 program animation system. Computers & Education 40 (1), 15–21.

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

Moreno, A., Myller, N., 2004. The intermediate language specification — mcode.