Beauty and the Beast: Translating Smalltalk to Java

Alexandre Bergel
RMoD Team, INRIA Lille-Nord Europe, France
alexandre.bergel@inria.fr

September 8, 2008

Abstract

The importance of Model-Driven Development (MDE) in Software Engineering has been in a constant increase. Model transformation, a key element of MDE, allows for the definition and implementation of the operations on models. This paper is about assessing how the Smalltalk programming language may be used as a modeling language to transform Smalltalk applications into Java ones.

To our knowledge, no attempt in translating Smalltalk programs into Java programs has been made. Such a translation raises several challenging issues related to the type system and the execution model.

This paper presents a general translating schema for Smalltalk to Java. In essence, each Smalltalk class is translated into a Java one; the root of the Smalltalk classes understands all Smalltalk messages that can be sent in order to satisfy the type system. Blocks are translated into inner classes. An implementation is also presented based on several case studies.

1 Introduction

Model transformation is a key facet of model-engineering. Technologies to perform model transformations range from conventional programming languages to specific transformation languages [7]. New general purpose and domain specific languages are regularly emerging.

Model transformation is a general appellation that covers different techniques and approaches. In this paper, we focus on transforming Smalltalk programs into corresponding Java ones. Mens *et al.* [5] qualify this program transformation as exogenous since the artifacts we are considering are programs and source codes are written in two different languages.

This paper presents a general schema for translating Smalltalk [3] into Java [4]. It describes a translation mechanism and discuses about a set of problematic and critical points that arise when translating a program written into a dynamic object oriented programming language to Java.

As shown with other language translator including Bigloo[1], Slang[2], and Pypy[3], such a translation cannot be realized on the full extend of the dynamic language. A subset has to be considered to cope with constraints of the underlying programming executing. The tricky part is to make this subset "big enough" to not make programmers loose their direction when developing an application intended to be translated.

To summarize our proposal, each single Smalltalk class is translated into a Java one. The top root Smalltalk class in Java is SmallObject. This class is the result of translating the Smalltalk Object class. SmallObject implements all Smalltalk messages that can be possibly sent. This inhibits the Java type checker on translated Smalltalk classes. Each Smalltalk block is translated into an inner class. For the strategy presented in this paper, the constraints imposed on Smalltalk are reduced to a minimal set:

- Reflection is only limited to introspection since Java does not provide tools for dynamic structure and be-

[1]http://www-sop.inria.fr/mimosa/fp/Bigloo/
[2]http://wiki.squeak.org/squeak/2267
[3]http://codespeak.net/pypy

havioral reflection. Note that doesNotUnderstand is supported by our translation.

- Blocks cannot refer to temporary variables since inner classes cannot refer to local variables that are not declared final.

Each of these constraints are discussed and analyzed further down in the paper. To our knowledge, this paper is the first one to present an efficient and portable translation schema into Java for Smalltalk. The results contained in this paper suggest that this approach is viable and scalable.

Experiments and the implementation provided in this paper use Squeak, a Smalltalk dialect open-source[4]. However, we believe the approach presented in this paper may be successfully applied to most of Smalltalk dialects.

The paper is organized as follows: First, Section 2 gives a general idea about the main lines of the translation. Then, subsections elaborate on crucial aspects of Smalltalk. Section 2.1 is dedicated to typing issues. Section 2.2 is dedicated to Smalltalk blocks. Section 2.3 presents Smalltalk object model versus the Java one. Section 2.4 relates on exception handling. Section 2.5 presents the importance to separate an application from the Smalltalk runtime. Section 3 describes the feasibility of the approach by offering various benchmarks. Section 4 presents a discussion and analysis. Section 6 concludes this paper.

2 Translating Smalltalk into Java

Despite different syntaxes, programming environments and application deployment mechanisms, the Smalltalk and Java programming languages share several commonalities. On the surface, Smalltalk and Java feature single inheritance, message-passing, field access and update, and self message send. They employ both a garbage collector and a similar set of control flow instructions. Both are object-oriented, therefore offer a notion of object, class, use inheritance and message passing.

A first approximation for translating a Smalltalk program into Java is easily obtained by mapping different language elements. For example, consider the definition of a class Counter in Smalltalk:

[4]http://www.squeak.org

```
Object subclass: #Counter
    instanceVariableNames: 'value'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CounterApplication'

Counter>> initialize
    super initialize.
    value := 0

Counter>> increment
    value := value + 1

Counter>> incrementBy: aNumber
    1 to: aNumber do: [:i |self increment]

Counter>> value
    ^ value

Counter>> display
    Transcript show: 'counter value=', value asString.
```

The class Counter inherits from the class Object and belongs to the class category CounterApplication. It has one instance variable (value) and defines five methods (initialize, increment, incrementBy:, value and display). The method initialize is automatically invoked when Counter is instantiated, it therefore acts as a kind of constructor. initialize calls initialize from its super class and sets value to 0. The method incrementBy: aNumber adds (in a rather inefficient way) the value carried out by aNumber to value. The message to:do: is sent to the immediate value 1 with aNumber and a block as arguments. The meaning of this call is to iterate from 1 to aNumber and execute self increment at each iteration. The method display prints a short message on the standard output stream.

With a not-so-sophisticated technique of code transformation, Counter may be translated into the following Java class definition:

```
public class Counter extends SmallObject {
    SmallObject value;

    public SmallObject initialize() { super.initialize(); value = 0; }

    public SmallObject increment () {
        value = value.plus (SmallInteger.getValue(1));
        return this;
    }

    public SmallObject incrementBy (SmallObject aNumber) {
        for (int _i = ((SmallInteger)SmallInteger.getValue(1)).value;
                _i <= ((SmallInteger)aNumber).value; _i++) {
```

```
            SmallObject i = new SmallInteger(_i);
            this.increment ();
        }
        return this;
    }

    public SmallObject value() { return value; }

    public SmallObject display () {
        Transcript.show (new SmallString("counter value=").
                append (value.asString ()));
        return this;
    }
}
```

This translation is obtained by using a dedicated visitor over the Smalltalk abstract syntax tree. This translation deserves few comments:

- When translated, Smalltalk objects are distinct from Java objects by inheriting from SmallObject. This class acts as a place holder for common operations on Smalltalk objects (e.g., equality, basic conversions, ...). As we shall see later, SmallObject will be populated with empty methods to satisfy Java's static type checker.
- In Smalltalk, every method call returns an object, the Java void type has no correspondence therefore. Every method takes instances of SmallObject as parameters and has SmallObject as return type.
- Primitive types such as integer, boolean, string, float are converted into their corresponding translated pair. The class SmallNumber is a subclass of SmallObject.
- The method initialize is naturally translated into a constructor.
- The message Number>> to:do: is translated into a Java for loop.
- The Java method call stack contains Smalltalk calls.

This example gives a first approximation of how Smalltalk may be translated into Java. The following subsections will focus on more problematic facets of Smalltalk related to the type system (Section 2.1), blocks (Section 2.2), the object model (Section 2.3), exception handling (Section 2.4), and cutting the application to be translated from the system (Section 2.5).

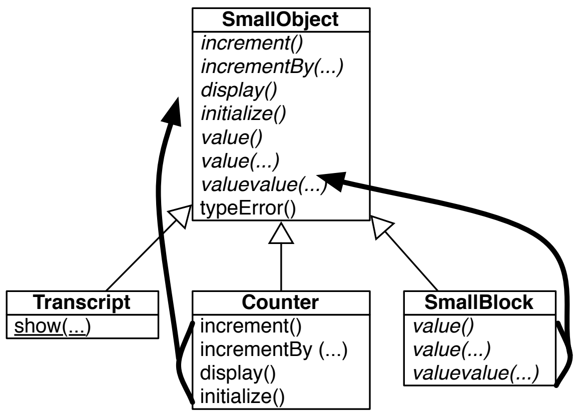

Figure 1: Each method in subclasses of SmallObject is defined in SmallObject as well (the underlined method is static).

2.1 Type

Smalltalk is traditionally classified as *dynamically typed* (also called *latently typed*). This means that its type system does not determine or even approximate values' type. The only certitude the programmer has, is that each Smalltalk expression will result in a Smalltalk object. One of the benefits of dynamic typing is to make source code concise and less verbose.

On the other side, Java is classified as a *statically typed language*. The benefit of static type checking is to prevent illegal operations (*e.g.,* sending a message that cannot be understood by the receiver) to happen at execution time.

Translating a program written into a dynamically typed language necessitates the addition of such annotations to satisfy the Java type system. The idea is to have a class as the superclass of all Smalltalk classes that understand all the messages that can possibly be sent. This is achieved by gathering all the methods defined in the application and the runtime (*i.e.,* all the subclasses of SmallObject), and then defining corresponding empty methods on the class SmallObject. This situation is illustrated in Figure 1: thick arrows shows the provenance of SmallObject's methods and italic methods are methods that call doesNotUnderstand when invoked.

For the class Counter given previously, the resulting

SmallObject will be defined as:

```
public class SmallObject extends Object {
    public SmallObject initialize()
        { return this.doesNotUnderstand (
            new Message ("initialize not implemented")); }
    public SmallObject increment()
        { return this.doesNotUnderstand (
            new Message ("increment not implemented")); }
    public SmallObject incrementBy(SmallObject aNumber)
        { return this.doesNotUnderstand (
            new Message ("incrementBy not implemented")); }
    public SmallObject value()
        { return this.doesNotUnderstand (
            new Message ("value not implemented")); }
    public SmallObject display()
        { return this.doesNotUnderstand (
            new Message ("display not implemented")); }
    ...

    public SmallObject doesNotUnderstand(SmallObject msg) {
        System.err.println(msg.printString());
        throw MessageNotUnderstood(msg);
        return null;
    }
    public SmallObject typeError() {
        System.err.println("Wrong type");
        Thread.dumpStack();
        System.exit(1);
        return null;
    }
}
```

The class Counter defines the methods initialize(), increment(), incrementBy(...), value(), display(). For each of these methods, an empty method having the same signature is created on SmallObject, the top most class of Smalltalk classes. The method show: is not defined in SmallObject since show: is a class side method, therefore translated into a static method in Java.

This mechanism makes each method in the Smalltalk application invokable on all Smalltalk objects. The default implementation of doesNotUnderstand throws an exception. typeError() is a convenient method that prints the Java stack with an error signaling a type error. This occurs when a method is called on an object that does not understand it. The definition of SmallObject above aborts the program execution in case of a message is not understood. Although some might feel the program interruption a bit rough, we assume such situation to be abnormal, revealing a wrong behavior for which pursuing the program execution is unlikely to resolve.

As we mentioned above, we take as hypothesis that the application to translate is self-contained. We therefore do not support separate compilation: a Smalltalk class cannot be translated in isolation. This has the benefit to statically know all the messages that can be sent. A simple pass on the Smalltalk abstract syntax tree detects methods that are not implemented.

2.2 Blocks

A block closure in Smalltalk (also simply called "block") is the aggregation of an expression, a list of variables and an environment. A block is similar in a sense to the lambda construct offered by Lisp languages and represents a reasonable approximation of mathematical functions. A block may be applied to a list of values, which evaluates its expression in its environment extends with the new bindings. The environment of a block is the current environment (usually a method or block activation) when the block is created. In the Counter example, the block [:i |self increment] is evaluated within the body of the to:do: method with the incrementing integer value.

Java's inner classes present few similarities with Smalltalk's blocks since they are both attached to an enclosing environment. A Smalltalk block is translated into a Java inner class, itself a subclass of SmallBlock. As an illustration, consider the simple Smalltalk block [:x | x + 10]. The translation into Java produces:

```
new SmallBlock () {
    public SmallObject value(SmallObject x) {
        return x.add(new SmallInteger(10));
    }}
```

Evaluation of this block is achieved by sending the message value(...) to it. The SmallBlock class offers several methods related to the evaluation of a block. The main methods are:

```
public class SmallBlock extends SmallObject {
    public SmallObject value()
        {System.err.println("Wrong number of arguments");
        return this.typeError(); }
    public SmallObject value(SmallObject o1)
        {System.err.println("Wrong number of arguments");
        return this.typeError(); }
    public SmallObject valuevalue(SmallObject o1,
                                  SmallObject o2)
        {System.err.println("Wrong number of arguments");
        return this.typeError(); }
    ...
}
```

A Smalltalk block is translated into an inner class, subclass of SmallBlock. One of SmallBlock's methods is overridden to proceed the evaluation of the block body. This method will be invoked to evaluate the block. The other evaluating methods of SmallBlock invoke typeError(), which raises a runtime error. This occurs when attempting to evaluate a block with the wrong number of arguments.

Translating blocks requires a special care when dealing with the self reference and accessing temporary variables. The this needs to be dereferenced to the outer object to access the current object. Consider the following illustrative method:

```
Counter>> reset
    [ value := self value - value ] value
```

This method is translated into Java as:

```
public SmallObject reset() {
    (new SmallBlock() {
        public SmallObject value() {
            return Counter.this.value =
                Counter.this.value().minus(Counter.this.value);
    }}).value();
}
```

A Smalltalk block may contain a return statement; if this block is executed, it returns not only from the block but from the lexically enclosing method (provided it's still active – if not, you get an exception). The following example illustrates this situation:

```
DiskEraser>> erase
    result := Dialog
        confirm: 'Are you sure you want to erase your hard drive?'
        onCancelDo: [ ˆself ].
    self erase.
```

Canceling the dialog will return self. Evaluating [ˆself] exits the method. Similarly to #Smalltalk[5], a compiler Smaltalk to CLI for .Net, non-local returns uses exception to prematurely exit the method. The erase method is translated as follows:

[5] http://www.refactory.com/Software/SharpSmalltalk

```
public SmallObject erase() {
    try {
        result := Dialog.theClass().confirmonCancelDo(
            "Are you sure you want to erase your hard drive?",
            new SmallBlock () {
                public SmallObject value() {
                    throw ReturnInBlock(this);
                }});
        this.erase;
        return this;
    } catch (ReturnInBlock e) { return e.getValue();}
}
```

A longer discussion on translating Smalltalk blocks into Java may be found in the Engelbrecht and Kourie's work [2].

2.3 Object model

Each object-oriented programming language comes with its object model. By *object model* we mean all the features provided by a programming language that are related closely or not to the concept of object. We will not present the differences between the Smalltalk and the Java object model. Several excellent works may be found in the literature [4, 8]. Instead, we list the features present in the Smalltalk object model that differs from the Java one or are result of a non-trivial translation:

- *Metaclasses* – In Smalltalk, classes are modeled as objects. A class is therefore an instance of another class, called a metaclass. Class references are manipulable as any plain object: message can be invoked on them and class reference may be passed as arguments. Metaclasses enables its instances to understand polymorphic calls.

  Java does not support metaclasses. Mapping each Smalltalk class method into a static Java method prevent therefore dynamically look up. Instead, we associate a Java object to each Smalltalk class. Each reference to a class returns this object. For example, consider the following Smalltalk code excerpt:

  ```
  Counter class>> fromFactory: aFactoryClass
      ˆaFactoryClass new newCounter
  ```

  The presence of class methods implies the presence of a meta-class. The translation will therefore be:

```
public class Counter extends SmallObject {
    private static SmallObject theClass =
            new Counter_Class();
    public static theClass () { return theClass; }
    ...
}

public class Counter_Class extends SmallObject_Class {
    public SmallObject fromFactory
                (SmallObject aFactoryClass) {
        return aFactoryClass.theClass().
            new().newCounter();
    }
}
```

- *Reflection* – Smalltalk supports an extensive panel of reflective features ranging from method reification to dynamic field and method addition [1]. Unfortunately, when projected to Java, Smalltalk's reflective capabilities are greatly diminished due to (i) the absence of mechanisms to alter the class behavior and structure (*i.e.,* intercession) and to (ii) a limited range of introspective features.

- *Current method context* – Smalltalk enables the method call stack to be reified using the dedicated thisContext pseudo variable. thisContext refers to an instance of the MethodContext Smalltalk class which contains information related to the last method call.

  Since the Java method activation context cannot be reified, thisContext is therefore not supported by our translator.

2.4 Exception handling

Java's exception handling model is a subset of Smalltalk's. In Java, the unique operation that can be performed on a caught exception is to be throw again (in order to be caught later on by a parent exception handler). Smalltalk offers a number of additional operations such as resume, restart, and retry, to resume the program execution at different location when an exception has been thrown. When translated into Java, Smalltalk exceptions are restricted to Java possibilities.

When translated into Java, exception classes are implemented as subclasses of SmallException is a subclass of SmallObject and is the root of all Smalltalk exception classes. The signal method is used to throw an exception. It throws an instance of a subclass of java.lang.RuntimeException, which is then later on caught by the method on:do:. We recall that Java runtime exceptions do not need to be declared in the Java method signature.

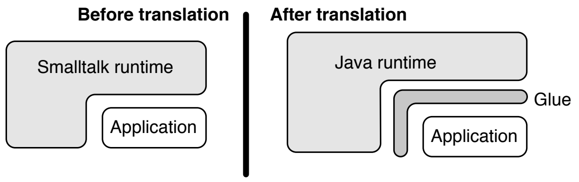

Figure 2: A glue between the translated Smalltalk application and the Java runtime simulates the original Smalltalk runtime.

2.5 Separating the application from the runtime

"Everything is an object" is probably one of the strongest pillar of the Smalltalk paradigm. Everything is an object, therefore every object is an instance of a class, which is also an object. Delimiting an application from the runtime is reputed to be a tough task that cannot be correctly automated (mainly because of the lack of a static type system).

In order to be translated into Java, a Smalltalk application needs to be correctly separated from the Smalltalk runtime. Although a translation of the whole Smalltalk image is possible, translating the primitives offered by the Squeak Smalltalk virtual machine into Java is far from being a trivial task. Smalltalk primitives related to concurrency, threads, graphics display largely differ from the ones provided by the Java virtual machine.

A Smalltalk application is closely tied to the runtime provided by the programming environment. This runtime comprises a large amount of features accessible within this environment such as the class collection hierarchy, streams, networking, input/output, language kernel, and graphical user interfaces. An application has naturally many connections with the runtime expressed in terms of class reference and use of literals objects. When translating a Smalltalk application into Java, a glue has to be layered between the Java runtime and the Smalltalk application. As illustrated in Figure 2, the glue offers class wrappers that make the Java runtime behaves as the Smalltalk one.

In order for an application to be translated, no *external reference* must be left unbounded by the glue. An external reference contained in the Smalltalk application that would not be defined in the glue cannot be bound to a definition, which would prevent the translation from being compilable.

For example, the Transcript global variable in Smalltalk is replaced by a Java Transcript class that offers static methods such as show(...), cr(). This class is part of the glue. Although the contract implicitly attached to the Transcript variable is not fully preserved with this glue (*e.g.,*Transcript is a variable whereas in Java it is a class), definitions provided by the glue preserve the original behavior of the Smalltalk application.

3 Benchmarks

This section gives an overview that compares the Smalltalk and Java version execution time length for a set of examples. We measured performances of four important facets of the translation: block evaluation, polymorphic calls, operations over primitive values (number, characters, boolean) and exception handling. Our benchmarks compare the Squeak's virtual machine with Java's[6]. Our goal is not to bash on Squeak, but rather to identify performance bottlenecks. The result of these benchmarks are summarized in the following table:

[6]The Squeak 3.8.12beta4U and the JVM 1.5.0_13 on a MacBook with 2Gb of memory were used for this experiment

Application	*Smalltalk (ms)*	*Java (ms)*	*Ratio*
Block closures	835	145	5.7
Polymorphic calls	379	85	4.4
Exception handling	8673	1256	6.9
Arithmetic operations	212	510	0.4
Athena (stack intensive)	2913	512	5.6
Athena (polymorphic calls)	951	269	3.5

We realized 2 sets of benchmarks. The first 4 benchmarks consist in evaluating 1 millions times an expression related to a particular aspect of the Smalltalk language:

- **Blocks closure** – As described previously (Section 2.2), Smalltalk blocks are translated into inner classes. The gain is of a factor 5.7.
- **Polymorphic calls** – Three message sends crawl over a class hierarchy (depth 8). The factor gain is 4.4.
- **Arithmetic operations** – The expression evaluated consists of 4 addition and 4 multiplication of integers. The Java version of this measurement is 2.4 times slower than the Smalltalk version (ratio 0.4).

  When translated into Java, Smalltalk numbers are implemented by wrapping Java values. This has the benefit to reuse the Java arithmetic operators and encoding of numbers. However, each arithmetic operation necessitates to unwrap the operands, performs the operation on Java values, then wraps the result into a SmallInteger. This combination of unwrapping and wrapping significantly raises the cost of number manipulation. We expect a type feedback mechanism or an inference mechanism to significantly circumvent this limitation.
- **Exception handling** – Throwing and catching an exception is naturally a costly operation. This operation is almost 7 times faster in Java. This is easily explained since the Java exception handling model

is simpler and does not require a reification of the method call stack.

In addition to these micro-benchmarks, we provide two benchmarks that employ Athena[7], a Smalltalk virtual machine written in Smalltalk. Virtual machine construction is probably one of the application domains that benefits the most from source code translation. Virtual machines are complex applications that require a significant software engineering effort. Athena consists of more than a thousand lines of Smalltalk code spread over four classes.

Two macro benchmarks are based on running Athena in two different settings. The first situation makes an intensive use of recursive method call. The later case heavily employs polymorphic calls. The corresponding gain factors are 5.6 and 3.5, respectively. These results correspond to the average performance gain an application benefit by being translated into Java.

This set of micro and macro benchmarks illustrates the practicability of the transformation process previously described without engaging a type inference mechanism.

4 Discussion

This section discusses and analyses various aspects of the code translation.

Constructor. Smalltalk does not provide a notion of constructor. Some dialects such as Squeak invokes initialize on the newly created object. The result of the expression Object new will be the result of Object>> initialize. It is expected that initialize returns the object initialize. For example, let us assume a method such as:

```
C>> initialize
    ˆ2
```

Evaluating C new will return 2, instead of an instance of C. When translated into Java, whatever the value returned by initialize, an instance of C will always be returned. This strategy used by our translator hasn't lead to any problematic situation so far.

Typing Smalltalk programs. To cope with the Java type system, SmallObject defines an empty method for each Smalltalk message that can be sent. This strategy thwarts the Java type system in statically detecting type misuses. This strategy assumes that the Smalltalk application is correctly typed since Java cannot type check it.

An alternative would be to leave the user annotates the Smalltalk program with Java type information. In Slang, the subset of Smalltalk that is mappable to the C language, requires the programmer to specify the C type of each variable. Methods are manually annotated in order to produce properly typed methods. For example, having self returnTypeC: 'char *' in a method will set the return type of the C function to char *.

Some approaches have been proposed to statically type check Smalltalk programs. For example, a popular implementation[8] has been proposed by Roel Wuyts. This type checker infers the type of instances variables based on messages sent. Unfortunately, knowing the type of instance variables will not leverage the mitigated performance on arithmetic operations.

Source code translation versus bytecode generation. Translating Smalltalk code into Java has few advantages over bytecode generation. For example, adjusting the generated code is easier over source code than bytecode. We initiated an experiment over Java micro edition, and the necessary adjustment were produced in a couple of hours.

5 Related Work

Kermeta. By being an extension of EMOF, the Kermeta workbench[9] is a metaprogramming environment based on an object-oriented domain specific language optimized for metamodel engineering. At the first glance, Kermeta looks like any programming language for general purposes: it includes control structure such as blocks, loop, conditional; it supports traditional object-oriented constructions; and enables interaction with Java. Kermeta offers specific constructions for expressing models such as *property*, which generalizes the notions of attributes, and associations (composite or not) that you can find in UML.

Xion. Xion is an action language for UML class diagrams; it is used to provide a high level platform inde-

[7]http://bergel.eu/athena

[8]http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper

[9]http://www.kermeta.org

pendent implementation of operations and methods [6].

6 Conclusion

This paper presents a strategy to translate Smalltalk into Java. The translation operates on a sub-set of Smalltalk. Few restrictions on the program to be translated are necessary to cope with the Java execution model.The case study we conducted suggests that these restrictions should not be significantly disturbing for the programmer.

New modeling languages are regularly popping up from the Modeling research community. The question rose by this paper is: *Whether Smalltalk may be successfully used in place were mainstream modeling languages succeeded?* This is a broad question that this paper partially answer by assessing model transformation and code generation toward Java.

Acknowledgement. We gratefully thank Noury Bouraqadi, Stéphane Ducasse, Serge Stinckwich, Mathieu Suen and Jan Vrany for their valuable reviews and comments.

References

[1] M. Denker. *Sub-method Structural and Behavioral Reflection*. phd thesis, University of Bern, May 2008.

[2] R. Engelbrecht and D. Kourie. Translating smalltalk blocks to java. *Software, IEE Proceedings*, 150(3):203–211, June 2003.

[3] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (Third Edition)*. Addison Wesley, 2005.

[5] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[6] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In S. K. L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, Oct. 2005. Springer.

[7] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jézéquel. On executable meta-languages applied to model transformations. Model Transformations In Practice Workshop, oct 2005.

[8] F. Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, Apr. 1996.