Language-Shifting Objects from Java to Smalltalk

An exploration using JavaConnect

Johan Brichau
Université catholique de Louvain
johan.brichau@uclouvain.be

Coen De Roover
Vrije Universiteit Brussel
cderoove@vub.ac.be

Abstract

Foreign-function interfaces enable programs in a host language to interoperate with programs written in another language and vice-versa. Two languages that feature such an interface to a common language can even interoperate by means of this common intermediate language. Smalltalk and Java, for example, both offer an interface to C and can therefore interoperate through a connection of these interfaces. However, the resulting interactions will need to occur at the level of abstraction offered by the (low-level) C interfaces, although both Java and Smalltalk are object-oriented languages. In this paper, we describe how the dynamic and reflective facilities of the Smalltalk language permits us to raise the level of abstraction of these inter-language interactions to seamless message passing between objects. In addition, we introduce and explore the concept of *language shifting objects* where Java objects are not only represented in Smalltalk but can partially or completely migrate from Java to Smalltalk. Language-shifting effectively migrates Java methods to Smalltalk methods such that original Java objects can end up "living" in the Smalltalk environment without needing a connection to their originating Java environment.

1. Introduction

Language interoperability is the ability of a program written in a host language to access constructs of a program that is written in another language. Language interoperability enables code reuse across language boundaries and therefore can help to improve the efficiency of the development process.

There are number of different techniques to language interoperability. Quite recently, language interoperability has been obtained by targeting languages to the same common runtime infrastructure. The Common Language Runtime (CLR), for example, is the execution environment for a variety of different programming languages and it specifically supports their interoperability. Similarly, there are several efforts to target the Java VM as the execution platform for other object-oriented languages, primarily to support interoperability with Java programs [1, 8, 7]. A less versatile approach is implemented by Corba and SOAP by running the programs in their own execution environment and offering support for communication and data exchange.

Another and more commonly adopted technique (that will be used in this article), is a foreign-function interface (FFI) that lets programs in a host language make calls to programs written in another language. Such an FFI typically targets interaction with programs in a single, specific foreign language, most often C. Some interfaces also provide a means for the foreign language to call back into the host language. As a result, whenever two different languages provide such a FFI to a third language, language interoperability between the first two languages can be established via the third language.

The JavaConnect system [3], that is presented in this paper, effectively relies on such a combination of Smalltalk's FFI (such as Visualworks' DLLCC [10]) and Java's native interface (JNI) [2] to allow Smalltalk programs to interact with Java programs. We will demonstrate how JavaConnect implements language interoperability through such a transitive combination of FFI's and how Smalltalk's dynamic and reflective features permit a high-level integration of Java objects in the Smalltalk environment. In particular, we describe and explore the concept of *language shifting objects* where Java objects migrate across the language barrier into Smalltalk such that a part or their entire set of methods of their class effectively executes in the Smalltalk environment rather than in the Java environment. To this end, Java methods are decompiled and translated to Smalltalk. Using such language-shifting, we can partially or completely migrate Java objects to Smalltalk to reduce the performance overhead involved with the FFI's and even "disconnect" an entire Java object structure from the running JVM.

The transitive combination of FFI's is not a novel nor unique feature exploited by the JavaConnect system. Several other similar libraries exist for other (dynamic) languages such as Lisp ([5]) and there are even alternative implementations in Smalltalk as well ([6, 4]). However, the JavaConnect implementation deliberately chose to plug its implementation into Smalltalk's meta-object protocol. The result is that Java classes and objects appear to live seamlessly in all existing Smalltalk infrastructure that relies on the meta-object protocol to perform its functionality.

In the following section, we provide an overview of the JavaConnect system and describe how it allows Smalltalk programs to interact with Java programs. Next, Section 4 introduces the concept of *language shifting objects* followed by Section 5 that introduces a number of applications of that technique.

2. JavaConnect

JavaConnect is a Smalltalk library that allows a seamless interaction with Java code from within Smalltalk. A Smalltalk application can manipulate any Java object and send messages to it, just as if it were a Smalltalk object. Its implementation relies on a connection between the Smalltalk and Java environments through their respective foreign-function interfaces. Smalltalk's FFI allows Smalltalk programs to interact with C programs and Java's JNI offers the ability for C programs to interact with Java programs. The transitive connection of these interfaces permits Smalltalk-Java interaction, as shown in Figure 1. The Java code thus executes on a regular Java VM and the Smalltalk code executes on a regular Smalltalk VM. Messages passed between Smalltalk and Java objects are actually translated into invocations of the FFI of both environments.

Figure 2 shows a Smalltalk code excerpt that implements exactly the same functionality as the Java code in Figure 3. The Smalltalk code excerpt can be executed in Smalltalk when the JavaConnect system is installed in the environment. JavaConnect is currently only available for Visualworks [10] but a port to Pharo/Squeak [9] is in preparation.

Although the technical ability to send messages across language boundaries merely requires the presence of a FFI, the seamless integration of Java objects in a Smalltalk environment was made possible because of the advanced reflective protocol in Smalltalk, allowing to wrap foreign (Java) values and make them behave like Smalltalk objects. In this section we outline the conceptual mapping of Java constructs into Smalltalk, which make the Java-Smalltalk interoperability seamless.

2.1 Language Interoperability

Although there are differences between the Smalltalk and Java languages, they both are class-based object-oriented programming languages with single inheritance and automatic memory management. In essence, Smalltalk and Java

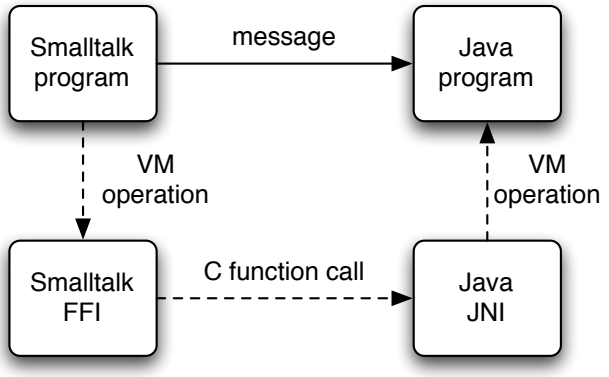

Figure 1. Smalltalk interacts with Java through the languages' FFI interfaces with C.

```
buf := JavaWorld.java.io.BufferedReader new_Reader:
        (JavaWorld.java.io.FileReader new_String: 'sometext.txt').
[((s := buf readLine) = nil) not]
  whileTrue:
    [line_count := line_count + 1.
    st := java.util.StringTokenizer new_String: s String: ' ,;.'.
    [st hasMoreTokens]
      whileTrue:
        [word_count := word_count + 1.
        s := st nextToken.
        char_count := char_count + s length]]
```

Figure 2. Smalltalk code interacting with Java objects for counting chars, words and lines of a file

```
buf = new BufferedReader(new FileReader(s));
while((s=buf.readLine())!=null) {
  line_count++;
  st=new StringTokenizer(s," ,;:.");
  while(st.hasMoreTokens()) {
     word_count++;
     s=st.nextToken();
     char_count+=s.length();
  }
}
```

Figure 3. Java code implementing the same functionality as Figure 2.

share the common object-oriented execution model where objects interoperate by sending messages. Therefore, language interoperability can best be established at this level, allowing Smalltalk objects to communicate with Java objects through message sending. Even Java's native types, instance creation operators and other Java peculiarities can be accomodated in this same model of interoperability, as we will demonstrate later on. We briefly outline the mapping for each Java language concept into Smalltalk.

Packages: Java packages define a name scope and, therefore, are very similar to Smalltalk namespaces. Given that namespaces are available in the Smalltalk at hand, JavaConnect will automatically define a namespace for each package that is available in the Java environment. In order to distinguish them from the regular Smalltalk namespaces, they are

harboured beneath the `JavaWorld` root namespace (as opposed to the `Smalltalk` root namespace). The java package `java.lang.util`, for example, is therefore accessible from Smalltalk as the `JavaWorld.java.lang.util` namespace. In a Smalltalk where no namespaces are available (such as Squeak), a dictionary could be created to simulate the namespaces for the Java packages.

Classes: Java classes are represented as Smalltalk classes with the same name. JavaConnect will dynamically generate such a "proxy" class in Smalltalk for each available Java class and add it to the namespace that represents its proper Java package. The Java `Date` class, for example, is accessible from Smalltalk as `JavaWorld.java.lang.util.Date`. From a user point-of-view, there is no observable difference between the manipulation of a normal Smalltalk class and such a "proxy" class that represents a Java class. For obvious reasons, JavaConnect uses a lazy generation mechanism, only generating the necessary "proxy" classes when they are referenced.

Inheritance: All of the aforementioned "proxy" classes that represent Java classes in Smalltalk are organized in an inheritance hierarchy that corresponds to the one defined by the original Java classes. The root of this inheritance tree is a convenience class named `JavaObject`.

Member classes: Java member classes are also represented as Smalltalk classes but their name is prefixed with their parent class(es) name(s) and includes the separator "_" between all the names. For example, the `Iterator` class, member class of the `List` class in the `java.lang.util` package, is represented in Smalltalk by the `JavaWorld.java.lang.util.List_Iterator` proxy class.

Methods: All methods declared by a Java class are represented as regular Smalltalk methods whose selector is an automatic conversion of the Java method name syntax into Smalltalk's message syntax. Furthermore, similar to any Smalltalk class, the static methods of the Java class can only be invoked on the corresponding proxy class itself, while the virtual methods can only be invoked on instance objects of that proxy class.

The conversion of Java method names to Smalltalk selectors occurs as follows:

- A Java method without arguments is trivially represented as a Smalltalk method with a unary selector. That selector is identical to the Java method name. For example, the Java method with signature `int length()` is represented in Smalltalk by the method with selector `#length`.
- A Java method with one or more arguments is mapped onto a Smalltalk method with a keyworded header. The first keyword is the concatenation of the Java method name and the short name of the type of the first argument, separated by a "_". The remaining keywords are the short names of the types of the remaining arguments. For example, the Java method with signature `int lastIndexOf(String,int)` is represented by the Smalltalk method that implements the selector `lastIndexOf_String:int:` .
- In some cases, notably in the case of overloaded methods, this naming conversion can result in different Java methods having the same header in Smalltalk. JavaConnect prevents this from happening and expands the simple names of the types in each keyword with their corresponding package name. Note that this only happens on rare occasions where methods are overloaded using types from different packages that have the same simple name.

Constructors: The public constructors of a Java class are represented using Smalltalk class methods. The name conversion scheme for constructors is identical to the one for methods, but the name of a constructor is always mapped onto "new". This means that zero-argument constructors are mapped onto the class method "new" and multiple argument constructors are mapped onto keyworded messages of which the first keyword is prefixed with "new_". For example, the Java constructor `Date(int, int, int)` is represented as the Smalltalk class method with selector `new_int:int:int:` (implemented on the `Date` class).

Instantiation: Java objects can be created from within Smalltalk by invoking the aforementioned constructors, or they can be obtained as return values from the invocation of any Java method. In Smalltalk, any Java object is represented by a Smalltalk object that is an instance of the appropriate proxy class (that represents a Java class). As a result, Java objects can be manipulated like any Smalltalk object. To ensure the correct functioning of object comparison, JavaConnect guarantees that identical Java objects are uniquely represented in the Smalltalk image.

Fields: Since Smalltalk does not have a mechanism for public instance variables, the public fields of a Java class are made available by automatically generated accessors methods. The accessor methods' selectors consists of the name of the field, prefixed with "get_"' and "put_". Static fields are accessible via class methods, while instance fields are accessible via instance methods.

Exceptions: Java exceptions are mapped onto Smalltalk exceptions. Smalltalk code can therefore reference and even throw a Java exception just like any Smalltalk exception.

Native types: The Java native types (int, boolean, float, char, ...) are transparently mapped onto instances of their Smalltalk equivalents (Integer, Boolean, Float, Character, ...) and vice-versa.

Callbacks: Smalltalk objects can be passed as arguments to Java methods *only* when the Java method has declared that argument using an interface type. This is because of a limitation of the Java language. Smalltalk objects are represented in Java by instances of dynamically generated `java.lang.Proxy` classes. Such proxy classes can only be generated for interface types, hence the restriction. Method invocations on these `java.lang.Proxy` instances trigger a

message send on the original Smalltalk object, using the same (but inverse) name conversion scheme as message sends from Smalltalk to Java.

Automatic Memory Management: Since both Java and Smalltalk have automatic memory management, a coordination mechanism needs to be set up such that each garbage collector knows which objects are still being referenced in the other language environment. This means that any Java object that is passed on to Smalltalk will be signaled to the Java environment as being retained in the Smalltalk environment, preventing its removal from memory. Whenever a Java "proxy" object is reclaimed from memory in Smalltalk, the Java VM is also informed that the corresponding Java object is no longer being retained in Smalltalk.

3. Reflection at Work

The previous section has outlined the mapping of Java language concepts onto Smalltalk language concepts to enable an interoperability between Java and Smalltalk at the object level. Although such an interoperability can be realized in Smalltalk without integrating the Java "proxy" classes in Smalltalk's meta-object protocol (MOP), JavaConnect has specifically chosen to pursue this implementation strategy. This means that JavaConnect represents Java classes in Smalltalk using exactly the same structure as the one used for Smalltalk classes (i.e. classes and their meta-classes). The result is that existing Smalltalk code will immediately be able to treat Java "proxy" classes as regular Smalltalk classes (given the appropriate method specializations, of course). In this section, we outline this implementation choice and reflect on the implications.

Figure 4 illustrates the implementation of JavaConnect as an extension of existing Smalltalk meta-objects. The JavaConnect classes are tinted gray while the original Smalltalk classes are left white. A first, simple part of the extension concerns the `JavaPackage` and `JavaCompiledMethod` classes, subclasses of `Namespace` and `CompiledMethod` respectively. These new meta-objects will represent Java packages and Java methods in the Smalltalk environment.

Next, JavaConnect integrates the Java "proxy" classes into Smalltalk as instances of the `JavaClass` class, which is a subclass of the existing `Class` class. More precisely, each such "proxy" class is an instance of its meta-class, which is actually a subclass of the `JavaClass` class (see Figure 4). These meta-classes of Java "proxy" classes are instances of the `JavaMetaclass` class. The example Java "proxy" class (`java.io.File`) that is included in the Figure illustrates this structure, which is exactly the way Smalltalk represents its own classes (also included in the Figure). It goes without saying that there is more to the implementation of JavaConnect than what is shown in Figure 4, but in the same way that this diagram forms the heart of Smalltalk's MOP, the shown extensions to the MOP are the core of how JavaConnect represents Java objects in Smalltalk.

This implementation strategy had some immediate advantages:

- Because Java packages are modeled as Smalltalk namespaces, all Java packages are immediately available in Smalltalk tools that reason about namespaces, such as the syntax highlighter and, of course, the namespaces lookup mechanism of the Smalltalk runtime environment. Enabling this interoperability required overriding only a single method on the `JavaPackage` class.
- Representing Java "proxy" classes using subclasses of the existing Smalltalk `Class` and `MetaClass` classes, has immediately enabled that Java "proxy" classes can be navigated with standard Smalltalk development tools. In addition, the instances of these classes can accessed and manipulated in exactly the same way as for normal Smalltalk instances and one can even create subclasses of a Java "proxy" class, using the standard tools and/or meta-object messages.
- The explicit representation of Java classes as a class–meta-class pair in Smalltalk does not only entail that Java static methods are implemented on the meta-class and Java virtual methods are implemented on the class itself. More importantly, it has allowed us to explicitly connect the Smalltalk meta-class with the `java.lang.Class` instance that reifies that class in Java. The meta-class of any Java "proxy" class therefore directly represents its Java counterpart as well.

4. Language Shifting

Although the ability to interact with Java from within the Smalltalk environment opens up the access to a vast array of Java libraries, there is a fair amount of runtime overhead involved with such external calls, compared to calling libraries native to the same language. A Smalltalk program that uses a data structure implemented in Java will eventually operate slower than an equivalent Smalltalk program that uses a Smalltalk data structure instead. In addition, the memory footprint will be larger as well since for each Java object that is manipulated by Smalltalk code, a corresponding Java "proxy" object needs to exist in the Smalltalk environment.

In order to reduce the overhead involved with using Java objects in a Smalltalk program, we introduce the concept of *language shifting objects*. In contrast to normal Java "proxy" objects, language-shifted objects have part (or all) of their *original* behavior executed in Smalltalk rather than in Java. In other words, some of the methods that can be invoked on it have *shifted* from Java to Smalltalk: they have been translated and can now execute in the Smalltalk VM. The result is that Java objects that are manipulated by a Smalltalk program do not always require to invoke behavior implemented in Java: the method executes entirely in Smalltalk, effectively eliminating the communication infrastructure overhead.

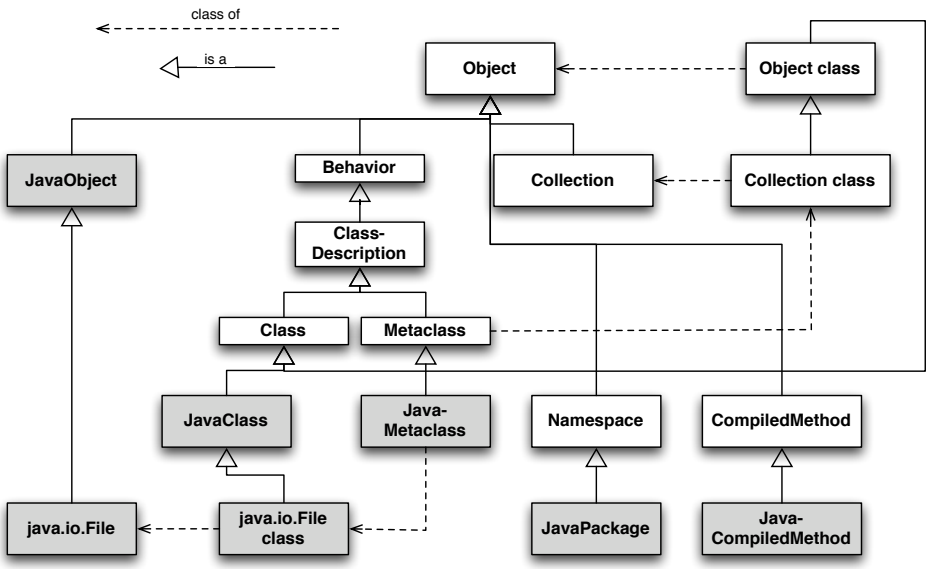

Figure 4. JavaConnect extends the Smalltalk MOP.

Evidently, the performance benefit does not hold for any Java method that is translated to Smalltalk code. If the translated Smalltalk code performs more calls to Java methods, the runtime overhead will increase. Therefore, language shifting often involves the translation of multiple Java methods that collaborate. At this time, the decision of which methods to translate lies in the hands of the Smalltalk developers.

The implementation of language shifting consists of two parts: translating method behavior and handling state of fields.

4.1 Translating Java methods

The translation process itself is implemented as follows. A Java decompiler decompiles the bytecode of the loaded Java class for which we want to perform language-shifting. The decompiled source-code is then parsed into a convenient abstract syntax tree from which the translated Smalltalk source for the desired Java methods is generated [1]. The generated Smalltalk source code is then compiled and the `JavaCompiledMethod` instance in the method dictionary is replaced by the resulting Smalltalk compiled method.

The source-to-source translation of Java to Smalltalk code is based on the conceptual mapping of language constructs employed by JavaConnect itself, which we described in Section 2.1. Here are the most important principles applied by the translation process:

- Invocations of public java methods are translated to Smalltalk messages according to the name translation scheme described in Section 2.1.
- Invocations of private java methods are translated in the same way but to ensure that the invocation will follow Java semantics (i.e. private methods are statically bound), the name of the declaring class is prepended to the Smalltalk selector. The corresponding method is added to the class as well.
- References to Java fields are replaced by calls to the (generated) Smalltalk accessor methods. If the fields are private or protected, accessor methods are also generated since the JNI allows to retrieve values of private and protected fields.
- References to Java class names are trivially translated to the corresponding references in Smalltalk.
- Uses of Java-specific values, such as all native types and `null`, are translated to the regular Smalltalk (object) counterparts.
- The use of Java arrays will be translated to the use of JavaConnect's representation for Java arrays in Smalltalk. This means that there is no mapping of Java arrays to Smalltalk collections.
- Java control-flow constructs such as `if-then-else`, `for`,`while`,`repeat`, etc. are translated to the equivalent Smalltalk idioms using messages such as `ifTrue:ifFalse`, `whileTrue:`,`to:do:`,etc.
- Java's try-catch construct is mapped to equivalent (nested) `on:do:` construct in Smalltalk.

At this time, some Java constructs that can be found in Java method bodies, such as type declaration statements and switch statements, are not handled. Similarly, synchronization primitives and break/continue statements require somewhat more complex transformations that probably can be

[1] The parsing step could be omitted if the decompiler supplies a reasonable Java AST.

implemented correctly, but they are not handled at this time. In such cases, the language-shifting fails. It is also important to note that generics are not an issue in this translation since they are a Java compiler facility and are not present in the bytecode of the loaded classes.

4.2 Handling Java fields

In addition to translating Java methods, JavaConnect allows to mark any method on a Java "proxy" class as a "cached" method. This means that the method will remember its return value and merely return that same value every time it is invoked on the same receiver, with the same arguments. This is particularly useful for accessor methods when we know that the values will remain constant. Such situations are not imaginary, as we discuss in the following section. The cached methods can also be applied when all Java methods of a class have been shifted to Smalltalk and when we know that manipulation of the values only occurs from within Smalltalk itself.

5. Applications of Language Shifting

Language-shifting is particularly useful when we can use it to reduce the amount of communication between the Smalltalk and Java environments. In this section, we briefly present a number of example applications in which we have applied the technique.

5.1 Static utility methods

Static Java methods often provide utility operations on values that are passed via the arguments only. The Eclipse JDT classes, for example, implement static methods that expose the meaning of flags encoded in integer numbers. Such methods have a typical signature like: `boolean isStatic(int)`. By language-shifting such methods, we were able to avoid the unnecessary call to Java code and keep the execution in the Smalltalk environment. In this particular case, the utility methods are frequently invoked by our Smalltalk code and since they operate on native integers, which are represented in Smalltalk by `Integer` instances, their internal execution therefore did also not require any communication with the Java environment anymore.

5.2 Visitor pattern on tree structures

The Visitor design pattern [11] relies on the implementation of so-called "accept" methods defined by all "visitable" classes. These methods implement a double-dispatch protocol: they invoke the appropriate visit method on the argument (i.e. the visitor). By language-shifting such "accept" methods, we can use a Smalltalk object as the visitor instead of the (foreseen) Java object and speed up visitations of entire tree structures. We applied this technique to language-shift the AST objects of the Eclipse JDT dom library and use a Smalltalk implementation for the expected ASTVisitor. This solution allowed us to reuse the existing accept methods (although automatically translated to Smalltalk) when we otherwise needed to implement them ourselves on the Java "proxy" classes.

5.3 Navigating tree structures

In the same context as the previous application, tree structures (such as ASTs) are frequently navigated through the invocation of `getParent()` to navigate to a node's ancestor and the invocation of (class-specific) methods to obtain a node's children. When the objects that make up the tree are implemented in Java and the traversal code is implemented in Smalltalk, the navigation process suffers some performance overhead. Fortunately, many tree structures (such as ASTs) will not change their structure. Every node will always have the same set of children. Therefore, we have cached those navigation methods, thereby eliminating the mentioned overhead. In addition, several other methods of the classes of the objects that make up the tree could be language-shifted.

Although we have not yet reached the point where all methods of the concerned classes have been language-shifted, when this happens we foresee that we can consider the entire language-shifted tree structure as completely Smalltalk-based and can theoretically "decouple" it from the Java environment.

6. Conclusion

We have presented how the JavaConnect system combines the foreign-function interfaces of Smalltalk and Java to permit Smalltalk code to interact with Java programs. Its implementation integrates with the Smalltalk meta-object protocol to achieve a seamless language interoperability. Using this system, we have explored the concept of *language shifting objects* where Java methods are translated to Smalltalk code, resulting in the ability to retain the execution of the interaction with Java objects in the Smalltalk environment itself. Such language shifting has some interesting applications and particularly targets runtime optimization but care must be taken which methods to shift since the translated Smalltalk implementation may require even more interactions.

References

[1] Clojure. http://clojure.org.

[2] Java native interface. http://java.sun.com.

[3] JavaConnect. http://www.info.ucl.ac.be/ jbrichau/javaconnect.html.

[4] JavaInSt. http://www.smalltalking.net/Goodies/VisualSmalltalk/.

[5] Jfli - A foreign language interface for Lisp. http://jfli.sourceforge.net/.

[6] JNIPort. http://jniport.wikispaces.com/.

[7] JRuby. http://www.jruby.org.

[8] st/JVM. http://www.smalltalkmigrations.com/stjvm.htm.

[9] The Pharo Project. http://www.pharo-project.org.

[10] Visualworks Smalltalk. http://www.cincomsmalltalk.com.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.