Generating Java Class Files from Smalltalk Objects

Author: Andrew Eidsness
Supervisor: Dr. W.R. LaLonde, B.A.Sc., M.A.Sc., Ph.D.

Date: April 19, 1999
Version: 1.1
Prepared For: Carleton University course 95.495B (Honour’s Project)

Preface

1. Abstract

Smalltalk is a mature object oriented language. Years of development have yielded a powerful, robust development environment that is suitable for a variety of projects. The financial sector appreciates its rapid development capabilities, as well as the ease with which changes can be implemented. At the other end of the spectrum, the embedded processing community likes the byte code interpreted nature of the language. A powerful VM can be used for development, and a custom VM can be written to run the same code in an embedded system.

While this is all good, twenty years of development has not produced a standard Smalltalk VM. The market is fragmented by several vendors, all producing a competing (and expensive) product. The Java language was founded on many of the same principles as Smalltalk, indeed many of its capabilities were taken from Smalltalk. Java is an object oriented language that is compiled to byte code. This byte code can be run on any machine implementing the proper code interpreter. The rapidly growing popularity of Java can be attributed to the availability of interpreters for a wide variety of platforms. Java is still in its infancy – performance, features, etc. can only improve. Additionally, Sun appears to be strong enough politically to prevent the Java market from becoming fragmented.

Java really consists of two components. There is a language (Java) and a byte code specification (Java Virtual Machine). We can consider the Java VM to be nothing more than a new platform (similar to any new product line; e.g., Sparc10, Sparc 2, etc). The most common way of generating Java byte code is to compile it from programs written in the Java language. However, the language is fully decoupled from the byte code, and this need not be the case. This project

proposes that it is possible to generate Java byte code from the Smalltalk syntax. Programs would continue to be written in the full Smalltalk environment, then this tool could be used to generate a fully functioning Java system.

2. Acknowledgements

I would like to thank Dr. LaLonde for his advice and encouragement throughout this project, especially during the preparation of this paper.

3. Table of Contents

4. List of Figures

5. List of Tables

1. Introduction

Since its recent introduction, Java has gained a huge following, and it is beginning to appear as though the Smalltalk community will embrace this new platform[1]. Witness the recent abandonment of Smalltalk by ObjectShare in its move to Java and a similar shift of focus at Carleton University.

Smalltalk's introduction twenty years ago prompted the object oriented revolution, and the language has been successfully used for numerous projects, this success has lead to a wide range of research intent on further improvements to the environment. Despite this success, and although the language provides an excellent development environment, Smalltalk does not seem to have lived up to its promise of a universal execution environment. Once cited reason has been the need for the high cost of the Smalltalk VM, a policy for which vendors have been widely criticized.

Similar to Smalltalk, the term Java has two meanings, it refers to both a machine-independent byte code specification, and a high level programming language which can be compiled to this language. It is the byte code specification (hereafter referred to as the Java VM) that has made Java so popular. The Java language itself is still in its infancy, and is constantly undergoing major revisions. This state of constant flux is blamed for the failure of a large number of projects, most notably Corel's Java Office Suite project. The conclusion seems to be that the Java language is still too immature for serious development.

[1] For an example see the Usenet thread at [USENET] which taken is from a recent discussion on comp.lang.smalltalk.

Below the language level, the Java VM is essentially unchanged since the initial version. In addition, all Java VM's are backwards compatible to the original version of the byte code, meaning that code targeted for the initial version of the VM will run on any subsequent VM. More importantly, contrary to the approach of the Smalltalk vendors, Sun freely distributes Java VM's for a wide variety of environments.

Hence we have identified the following points:

- Smalltalk provides an excellent development environment. Code reuse and an excellent debugging environment allow rapid application development and easy maintenance of existing applications.
- Code targeted to the Java VM can be executed on a diverse set of hardware platforms (including embedded environments such as USRobotic's PalmPilot). Further, the popularity of the Java VM makes it extremely likely that new platforms will provide their own implementations of the Java byte code interpreter.

These points seem to suggest that it would be advantageous to develop programs using some Smalltalk environment (ObjectShare's VisualWorks has been targeted for this project) and then deploy the application on the Java VM. By targeting version 1.0.2 of the VM we ensure that the code is not subject to future changes. Such a tool would, at the very least, provide an easy way to demo Smalltalk applications (since support for at least version 1.0.2 of the Java VM is included in all modern web browsers).

1.1 Supporting Smalltalk on a New Platform

In general, platform support is added to Smalltalk by ensuring that the final code can run on the new platform; e.g., Intel chips require one machine language, Motorolla chips another, and the

Java VM yet another. Figure 1a shows how Smalltalk would function on top of a new hardware platform. The Smalltalk byte code is unchanged, instead a new interpreter is created. Following this approach for the Java translation would lead to an implementation of the Smalltalk VM that runs on top of the Java VM, as shown in Figure 1b. Although feasible, this project proposes to reduce the overhead incurred by running two virtual machines; our goal instead is to compile the Smalltalk objects directly to Java byte code to create a system as in Figure 1c.

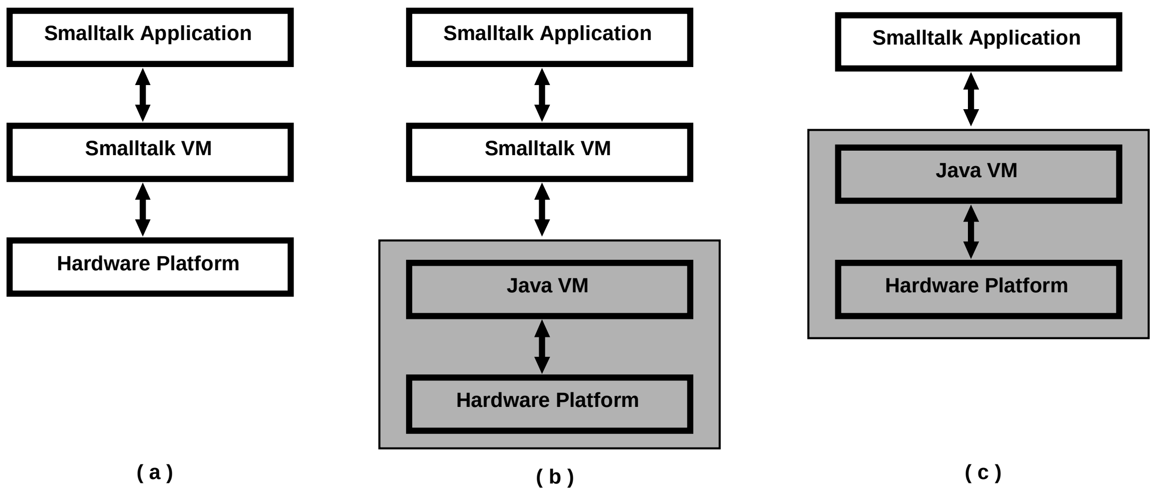


FIGURE 1: (a) The traditional way of running Smalltalk on a new platform, just write a new Smalltalk VM.
(b) How this would apply to the Java VM (NOTE the redundancy of two VM's).
(c) This project eliminates a step by compiling directly to the Java VM.

The second point to consider when supporting a new platform is the creation of user interface components for that platform. The development of UI components lies outside the scope of this project, however since the VisualWorks system has a well defined API for its user interface components, it is a relatively straightforward task to create Java classes which wrap the AWT[2] components with the appropriate interfaces. A rudimentary interface is provided with a Java class that is a scaled-down version of Smalltalk's Transcript window. This Transcript allows text output, and a limited form of text input. In addition, the Transcript class has been wrapped in a

[2] The abstract windowing toolkit is the package that supports graphical interfaces in Java programs.

Java Applet, so that the translated application can be embedded in any web page. Section 3 describes this process in more detail.

1.2 Document Overview

Section 2 begins by describing the trivial details involved in translating a Smalltalk application. The bulk of the section describes the more complicated parts of such a translation; e.g., how Smalltalk blocks are handled. Section 3 gives an idea of the abilities of the tool by describing the process of converting a moderately complex application to its Java equivalent. Appendix A describes the test plan which has been successfully handled by the tool.

Additionally, a web page has been created as a form of support for this paper. It includes links to relevant web sites as well as a working version of the sample application. This page is currently at [HONSUP].

2. Methodology

2.1 Introduction

Our goal is to develop a tool that is able to translate a working Smalltalk system to its equivalent Java byte code representation. We want the translated system to directly use the functionality of the Java VM as much as possible. However, due to language differences, it may be necessary to implement a partial Smalltalk VM in Java code. Our secondary goal is use Java byte codes as much as possible, making additions only when absolutely necessary. Additionally, all extensions will be implemented in Java, so they can run on top of any Java VM.[3]

2.2 Miscellaneous Details

This section defines the key language features, and explains the solution for all trivial problems. The following sub-sections describe the larger issues in more detail. Table 1 outlines some of the similarities and differences between the Smalltalk and Java language features.

Smalltalk	**Java**
Object oriented	object oriented
Memory managed by garbage collector	memory managed by garbage collector
Single inheritance of code and data	single inheritance of code and data
Dynamic binding	pseudo-dynamic binding
non-typed	typed
Instances access own methods via self	instances access own methods via this
Inheritance available on both instance and class sides	inheritance available on instance side only
Method lookup based on instance	method lookup based on declared type as well as instance

[3] As opposed to some projects which propose a modified Java VM.

all methods globally visible	some methods private to class or subclasses
all instance variables visible only to class and subclass	some instance variables globally visible
Equivalent of function pointers (#perform:)	no function pointers
parts of method selector mixed with method arguments	methods defined with complete method selector followed by list of arguments
Symbol as literal	N / A

TABLE 1: Comparison of some of the similarities and differences between Smalltalk and Java language features.[4]

2.2.1 Garbage Collection

Smalltalk and Java are virtually identical in the way that memory is managed. Memory is transparently allocated when a new instance is created, and transparently returned when an instance is no longer referenced. For this reason memory issues can be ignored.

It could be argued that this is one of the greatest similarities between Smalltalk and Java. Consider for example, the difficulties that would arise from attempting such a translation from C++ source code. Objects created and released with the *new* and *delete* keywords would be straightforward, but there is not obvious solution for dealing with other pointer manipulations.

2.2.2 Private vs. Public Members

In Java, methods can be declared as private (visible only within the class), protected (visible only within the class and its subclasses), or public (visible to everyone). In Smalltalk all methods are implicitly public. Therefore, all methods generated from a Smalltalk object will be declared public in the resulting class file.

[4] See [CHIMU] and [STIC] for a more thorough analysis.

Similar to methods, Java allows all member variables to be declared private, protected, or public. In Smalltalk, all member variables are visible only to the class and its subclasses. Thus member variables in all generated classes will be declared protected.

2.2.3 Dynamic Binding, Polymorphism, and Types

In both Smalltalk and Java, method lookup is performed symbolically, at run-time. In languages that are strictly compiled (e.g. C++) methods are invoked via memory offsets, which are determined at compile-time. Dynamic binding, although slower has the advantage of permitting some objects to be replaced without recompiling the entire system. It would appear that there is no issue. However in a strict sense, Java is not entirely dynamically bound. Section 2.4 explains the somewhat subtle differences, as well as the chosen solution.

2.2.4 Inheritance

In Smalltalk, the methods of a superclass (on both the instance and the class sides) can be invoked using the special object *super*. Java uses the super keyword to invoke a superclass' instance methods, but there is no way to directly reference a superclass' static methods, short of specifying the superclass itself; i.e., Java has no concept of inheritance on the static side. As described in section 2.6, the solution is to avoid using the Java static side.

Smalltalk implicitly provides access to a superclass' member variables; i.e., a superclass' instance and class variables can be accessed via a direct reference, no special keywords are required. The Java byte code requires the class where a member variable is defined to be supplied when a variable is referenced. Smalltalk ensures that subclasses do not reuse their parent's member variable names so given any instance variable name, we can find the defining class by examining the inheritance hierarchy.

2.2.5 Base Types

Smalltalk has several base types or literals, which are the ultimate contents of any memory location. Each of these types is represented by a specially written Java class. Table 2 lists these base classes and their corresponding Java literal class. Every instance of a base type is represented by a new instance of its corresponding literal class.

Smalltalk Base Type	Java Literal Class
String	visualworks.literals.String
Symbol	visualworks.literals.Symbol
Character	visualworks.literals.Character
Integer	visualworks.literals.Integer
Array	visualworks.literals.Array
Float	visualworks.literals.Float
Double	visualworks.literals.Double
FixedPoint	visualworks.literals.FixedPoint
Boolean	visualworks.literals.Boolean *(abstract)*
True	visualworks.literals.True
False	visualworks.literals.False
Nil	visualworks.literals.UndefinedObject

TABLE 2: Correspondance between Smalltalk base types and the Java classes that are used in the generated system.

In most cases, the Java literal class just wraps one of the primitive Java type (e.g., int, char, etc). Literals are represented with Java classes so that the Smalltalk interface can be made available to the generated classes (i.e., visualworks.literals.String implements the interface of Smalltalk's String class). These classes are compiled from Java source, the tool simply copies the class files to the target directory.

2.2.6 The Smalltalk #perform: Method

Smalltalk has the equivalent of function pointers with it's #perform: method. This method allows a selector to be built at runtime. We can implement this method in our generated classes as shown in the following Java pseudo-code:

```
public Object perform_(visualworks.literals.Symbol sym) {
   if(sel == #method1)
      return this.method1();
   else if(sel == #method2)
      return this.method2();
   . . .
}
```

2.2.7 Method Selector Mapping

A multi-part Smalltalk selector, say *#selectorArg1: arg2:*, must be mapped to a single string for Java. Additionally, Java does not allow colons in its selectors, so these must be replaced with underscores so we get *selectorArg1_arg2_(…)*. The arguments retain their order when put into the argument list.

Smalltalk allows some operators (e.g. @, -, and +) to be used as method selectors, while Java does not. In order to translate these operators to Java, they are replaced with a string, as shown in Table 3.

Smalltalk Operator	Java Selector String	Smalltalk Operator	Java Selector String
+	_plus	<	_lessThan
-	_hyphen	>	_greaterThan
*	_star	<=	_lessThanEqual
%	_percent	>=	_greaterThanEqual
/	_foreSlash	=	_equal
//	_doubleForeSlash	==	_doubleEqual
\\	_doubleBackSlash	~=	_tildeEqual

@	_at	&	_ampersand
**	_doubleStart	|	_pipe

TABLE 3: Mappings for translating Smalltalk operators to Java selector strings.

2.2.8 Smalltalk Primitives

Smalltalk methods are able to access *primitives*. These are essentially Smalltalk bytecodes that are visible at the application level. It is a straightforward task to catalogue the semantic meaning of each primitive, generate a snippet of Java bytecode to perform the same task, and then replace the primitive reference with the Java byte code when the Java class is generated. Due to the large number of primitives, this work is outside the scope of this project.

2.3 Java Class Files

This section introduces the class file concept and gives a brief description of how Smalltalk objects are translated. The languages are essentially compatible, in that both are based on objects which are composed of member variables and methods.

2.3.1 Namespaces

One area where Smalltalk and Java differ is namespaces. All Smalltalk classes are defined in a single namespace. Java uses packages to allow multiple namespaces. By putting all generated classes into the same package (we chose the name *visualworks*) we guarantee that the proper classes can be found and that there will be no naming conflicts with existing Java classes.

2.3.2 Object and Inheritance

In the generated class files, the Smalltalk inheritance hierarchy ties into the Java one directly below java.lang.Object. As illustrated in Figure 2, when visualworks.Object is generated, it inherits from java.lang.Object and the rest of the generated classes are derived from visualworks.Object in the same way as the original Smalltalk classes.

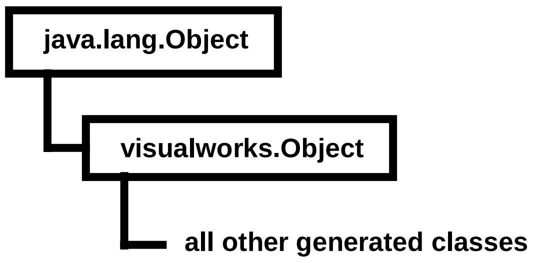


FIGURE 2: Inheritance hierarchy for generated classes, showing where it ties into the Java hierarchy.

2.3.3 Class Files

Java classes are represented with class files, with one class file for every Java class.[5]
Thus the result of running the tool on a Smalltalk system is a directory containing a collection of class files. The intent is for the generated classes to use the Java byte codes as much as possible. However, as described in the following sections this is not always possible. In some instances, it is necessary to enhance the Java VM using a few custom Java classes. These custom classes are included in the sub-packages of the base visualworks package but are pre-compiled, the tool simply copies them to the target directory.

2.3.4 Graphical Interfaces

The VisualWorks system creates graphical interfaces (GUI's) using component specifications and builders[6]. By creating component builders in Java, it would be possible to read the specs (once they have been translated from the Smalltalk source) and build the appropriate interface using Java components.[7]

GUI issues lie outside the scope of this project, so the Java builder has not been created, but it is interesting to briefly examine this issue. The GUIBuilders read the specifications and create the proper components. If Java files were created for each component (this would likely have to be done manually, since they must use the proper AWT classes) then the builders themselves could be translated from the Smalltalk source using the proposed tool. This is explored in more detail in section 4.1.1.

[5] While it is possible to define multiple classes in the same class file, only one of them can be declared public. All Smalltalk classes are public.
[6] This is the Builder pattern of [GAMMA 94] p. 97.
[7] Applied Reasoning [APPLIED] uses a variation of this technique to create a system where the Smalltalk code resides on the server and a Java interface is built on the client. The client generates Java events which are sent (via CORBA) to the server where all processing is performed.

2.4 Types

Java is a strictly typed language while Smalltalk is entirely untyped. This means that when a variable or method argument (i.e., a parameter) is translated from Smalltalk to Java, some type must be inferred. Some projects[8] have had some success doing this in a rigorous fashion, but we choose the simple solution of declaring everything as visualworks.Object, and letting the polymorphic abilities of the Java VM sort it out. The idea is that given an instance and a selector, the VM will find the correct code (from the instance's inheritance hierarchy).

If Java were entirely dynamically bound, this solution would be ideal, however in a strict sense this is not the case. When a Java method is invoked, both the classname and the method name must be supplied. The method table of the given class is examined (at runtime) to find the offset of the method (in terms of its index in the method table). The corresponding instance then invokes the code pointed to by the index. The method table is a collection of code pointers, where each entry corresponds to some symbolic method name. The method table is built by the VM when a class is loaded. Figure 3 illustrates this concept showing how it relates to inheritance.

[8] For example, [ORISA] performs an analysis of the code to hypothesize about types. They report being able to automatically make a type determination in most cases. The small remainder which cannot be decided, are brought to the user's attention to be manually defined.

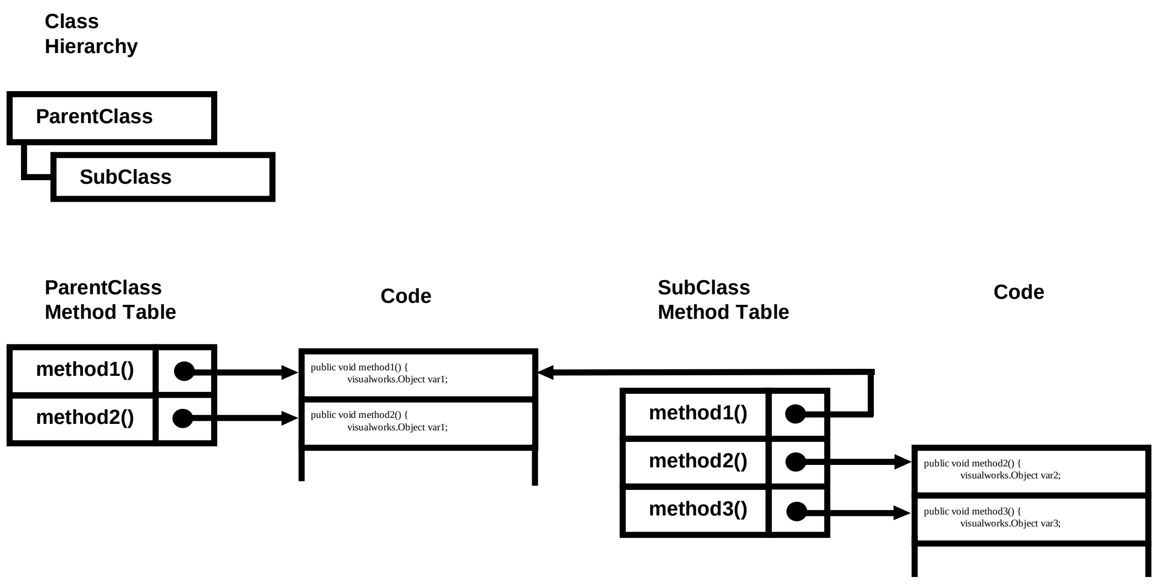

Class Hierarchy
ParentClass
SubClass
ParentClass Method Table
method1()
method2()
Code
public void method1() { visualworks.Object var1;
public void method2() { visualworks.Object var1;
SubClass Method Table
method1()
method2()
method3()
Code
public void method2() { visualworks.Object var2;
public void method3() { visualworks.Object var3;

FIGURE 3: Diagram showing how method tables are inherited. Even though SubClass doesn't implement method1(), it includes an entry in its table. This is not drawn to scale in terms of code locations, etc. it is just an abstract diagram used to illustrate the concept.

The following Java code shows how the method table in Figure 3 would be used.

```
ParentClass inst = new SubClass();
inst.method1();         // line A - to be referenced below
inst.method2();         // line B - to be referenced below
inst.method3();         // line C - this line will fail
```

From the compiler's viewpoint, *inst* is of type ParentClass, so it will use that as the class for all method invocations. Line A for example will compile to the following Java byte code:

```
invokevirtual visualworks/ParentClass/method1()V
```

Notice that ParentClasses' method table is used to determine the method index. Line A will find that the required method is the first in the table, so inst will invoke the code pointed to by the first entry of its method table; i.e., the first entry in the SubClass method table, which in this case is the same as that pointed to by ParentClass. In line B the method table of ParentClass will be consulted to determine that the required method is second in the table. When the code pointer is retrieved from the method table of inst, it will point to the new code since SubClass has overridden the method. Thus it should be clear that even though method3() is implemented by

SubClass, it cannot be found, since the bytecode refers to the table for ParentClass. The normal Java solution is to use typecasting as in:

```
((SubClass)inst).method3();
```

This is not a sufficient solution in our case, since it reintroduces the original problem of deciding on a type (in this case for the typecast).

The chosen solution is to guarantee that all selectors can be found in ParentClasses' method table. We can achieve this by generating visualworks.Object such that it implements every selector that is sent in the generated code. So when we translate a collection of Smalltalk objects, we generate a collection of all method selectors. The final step of the translation process is to build visualworks.Object such that it contains a method body for every selector. A class inherits its parent's method table so all method tables will have an entry for every selector. In the above example, ParentClass would have inherited an index for method3() from Object, so the sample code would evaluate correctly since its own entry for that index is the correct one. This also implies that the code in the methods generated in Object are never used, the methods are just placeholders in the method tables. However, for safety reasons we have generated code that throws an exception (as discussed in the next section).

2.4.1 DoesNotUnderstand Exception

The default Smalltalk behaviour when a method body is not found is to raise a *DoesNotUnderstand* signal. By adding code to throw a similar Java exception into the method bodies, we can emulate this behaviour in our generated code. The methods in Object will only be evaluated if the subclass does not implement the method, in other words, when the selector is not understood by the instance.

2.4.2 Summary

This solution is obviously less than optimal since it builds an Object class that is tailored to a specific set of class files. More importantly, it wastes a lot of memory, since every method table in the VM will contain an entry for every selector in the system. Despite of these drawbacks, this solution has been chose for its simplicity and the way it is able to emulate Smalltalk's behaviour.

2.5 Blocks

2.5.1 Introduction

A smalltalk block is an instance of an object (BlockClosure in VisualWorks) containing code which can be evaluated on demand, essentially a first class function. The Java VM does not support first class functions, so this behaviour must be emulated. This means that structures must be defined which permit the following behaviour:

- evaluate a section of code on demand
- pass a section of code between methods
- modify variables defined outside of a block (but within the scope that defines the block)
- cause a block's defining context to terminate

Section 2.5.2 contains a high-level description of the solution. Section 2.5.3 describes the evaluation of blocks. Section 2.5.4 discusses variable modification. Section 2.5.5 presents the details of how the defining context can be terminated. Section 2.5.6 concludes with a summary of the chosen solution.

2.5.2 Overview of Solution

This section proposes that the addition of a few Java classes is enough to provide all of the functionality of Smalltalk blocks.

As illustrated in figure 4, these base classes are essentially primitives that will be used by the generated classes. These base classes are used by the generated code, they are never directly referenced by the user (in this case the user is the Smalltalk programmer).

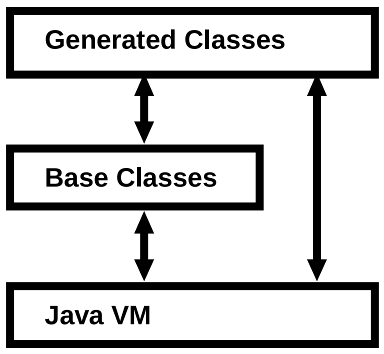


FIGURE 4: Diagram of relationship between class types in the resulting package. The base classes enhance some, but not all of the functionality of the basic Java VM.

For performance, the size of the base class set should be kept as small as possible. This restriction differentiates this project from others that attempt to implement a full Smalltalk VM on top of the Java VM,[9] and should yield much better performance.

2.5.3 Block Evaluation

In Java, all code evaluation is triggered via a message send. Thus when a class file is generated, we also generate a special blocks method. The body of this method contains the bodies of all blocks defined within the class. Each block is assigned a unique integer label. A switch statement is used to separate the block bodies within the special blocks method. During the generation phase, the label of each block is tracked, so that block evaluation becomes a simple matter of calling the special blocks method with the appropriate label as an argument.

As an example, consider translating the following Smalltalk code to equivalent Java code.[10] This example obviously lacks details, but instead illustrates the concept.

```
aMethod
```

[9] See [STX] for example.
[10] The generator does not actually create source Java code; this example merely illustrates the underlying structure of the byte code that is generated.

```
      [ true ] value.
      [ false ] value

   void aMethod() {
      self _blocks(0);
      self _blocks(1);
   }

   void _blocks(int i) {
      switch(i) {
         CASE 0:  return true;   break;
         CASE 1:  return false;  break;
      }
   }
```

The first step in automating this process is to create an object to wrap the message sends to the special blocks method. We define a class ContextBlock.java as follows (some details have been omitted for clarity).

```
   public class ContextBlock extends visualworks.Object {
      int blockIndex;
      visualworks.Object instance;

      public visualworks.Object value() {
         return instance._blocks(blockIndex);
      }
   }
```

ContextBlock.java also defines methods for the Smalltalk selectors #value:, #value:value:, #value:value:value:, and #valueWithArguments:. Passing arguments to a block is discussed in the next sub-section.

To complete this example, the Java version of #aMethod now becomes:

```
   void aMethod() {
      (new ContextBlock(this, 0)).value();
      (new ContextBlock(this, 1)).value();
   }
```

2.5.3.1 Block Arguments

Another detail is the ability to pass arguments to the blocks. We add a parameter to the special blocks method, which is just an array of objects that will be accessed by only those block bodies that expect arguments (this is ensured when the Smalltalk source code is written). In the body of *#value*, this will be an empty array, in *#value:* it will be an array with one element, etc.

2.5.3.2 Summary

Thus when the generator encounters a block, three things happen:

- the block is assigned a unique integer label
- the body of the block is added to the class' special blocks method
- Java bytecode is generated to create a new instance of ContextBlock using the block's integer label

Since instances of ContextBlock.java are ordinary Java objects, they can be passed between methods and classes in the same way as the Smalltalk counterpart.

2.5.4 Variable Accessing and Modification

The scope of a block is nested within that of the context where the block is defined. This means that a block is able to access variables that have been declared outside of the block but within the defining method. Any changes to variables performed within the block should be visible outside the block when the block terminates. For example, the following code should return the string *"inside the block"*, a behaviour that is consistent with the simple scoping rules of Smalltalk and Java (even C supports this form of scoping).

```
| local |
local := ’before’.
[ local := ’inside the block’ ] value.
^local
```

Unfortunately, providing this behaviour in the context of blocks is not trivial. Since, as described in section 2.5.3, the body of a block is evaluated in a method other than the defining one, the locals will not be visible in the special blocks method. The solution is to bypass the normal Java scoping mechanism and pass our own table of variable bindings. By using this table for all variables (even those local to the block), we can easily handle blocks defined within other blocks.

The class Scope.java has been defined to implement this binding table. Instances of Scope.java exist as a linked list, where each cell contains the variables defined by a single context, as shown in figure 5. The table is stored as an array, so variable references are nothing more than offsets into this array. The indices of a child start from the largest value of its parent.

```
| local temp |

temp := 'before'.
[ | nested | local := 'inside the block' ] value.

^local
```

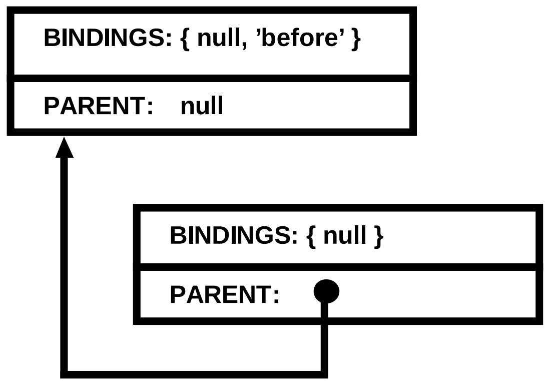


FIGURE 5: Example of how scopes can be nested, showing both the Smalltalk code and the corresponding Java structure. The bindings array is declared as type visualworks.Object, the contained string is actually of type visualworks.literals.String, which inherits from visualworks.Object.

The advantage of this approach is that it allows a context's variables to go out of scope without overhead. A block can modify its variables, as well as its parent's variables. The block's instance of Scope.java is destroyed when the block terminates, but the parent still has access to its own scope.

So in addition to a code pointer, a ContextBlock stores an instance of Scope.java. The value of scope is set when the block is defined, so that wherever the block is evaluated, the correct scope will be accessible. Also, since the block stores a reference to the scope (vs. a copy), all variable modifications will be visible when the block is evaluated.

Thus the final definition for the special blocks method is as follows. Notice that the third argument of the special blocks method is an array of block arguments (as described in section 2.5.3.1).

```
// This method is received by the owner of the block.
public visualworks.Object _blocks(
         int,                  // block index
         Scope,                      // context’s scope
         visualworks.Object []);     // arguments to block
```

Also ContextBlock.java must be refined to:

```
public class ContextBlock extends visualworks.Object {
   int blockIndex;
   Scope scope;
   visualworks.Object instance;

   // . . . some details omitted . . .

   // The #value method has no arguments so an empty array is used
as a
   // place holder.
   public visualworks.Object value() {
      visualworks.Object blockArgs[] = new visualworks.Object[0];

      return instance._blocks(blockIndex, scope, blockArgs);
   }

   // The #value: method has one argument, so a single element
array is
   // used.
   public visualworks.Object value_(visualworks.Object arg1) {
      visualworks.Object blockArgs[] = new visualworks.Object[1];
      blockArgs[0] = arg1;

      return instance._blocks(blockIndex, scope, blockArgs);
   }

   // similar methods for all other value selectors
}
```

2.5.4.1 Critique of Scope.java

Using this technique, all methods generated from a smalltalk object bypass the normal Java scoping mechanism, in favour of this custom scoping technique. This introduces a performance penalty, since all variables are now accessed at a higher level. However, performance is acceptable if the project goal is to be able to create demos of working Smalltalk systems. A demo (which is likely being accessed from some web page) is allowed to perform sub-optimally.

If the performance penalty can be overlooked, then this is a very powerful technique that can be used in a variety of situations. The key is that we are not generating source code so the emphasis can be on functionality instead of elegance (the execution of the compiled class files is not visible to the user).

2.5.5 Returning from Blocks

2.5.5.1 Introduction

A block without a carat (^) *implicitly* returns the result of the last computed value (i.e., the final line). A block with a carat *explicitly* forces the current value to be returned from the method that defined the block (i.e., a carat causes the block, as well as the method that defined the block, to terminate immediately).

This implies that a block can only use an explicit return if its defining method is still on the execution stack. For example, the following Smalltalk code will cause a *ContextCannotReturn* signal to be raised.

```
setBlock
   instanceVar := [ ^’block stored in an instance variable’ ]

useBlock
   self setBlock.
   instanceVar value
```

When the block is evaluated (in #useBlock), it will attempt an explicit return from the context of #setBlock. Since #setBlock has already completed, its context has been destroyed -- an error signal is raised.[11]

[11] In this case, an implicit return would execute without error.

2.5.5.2 Implementation of Implicit Returns

Implicit returns are essentially the default case. A normal Java method return from the special blocks method will return execution to the context that evaluated the block. Notice that this holds even if a block is being evaluated from within another block.

2.5.5.3 Implementation of Explicit Returns

Explicit returns require the ability to send a message up the evaluation stack to the context of the specific method that defined the block, we do this with a Java exception signal. When a context receives this signal, it should resume execution (terminating all contexts beneath it) and immediately return the contents of the signal. There may be an arbitrary number of contexts between the definition and the evaluation of a block, so the signal must be more than a simple return.

The Java exception mechanism is ideally suited for this. The intent of exceptions is to mark a section of code and then deal with any errors that occur before the end of the section. The following Java code illustrates this process.

```
try { this.wrapperMethod(); }
catch(Exception e) { return "an error occurred"; }

void wrapperMethod() throws Exception {
   this.innerMethod();
}

void innerMethod() throws Exception {
   if(badCondition)
      throw new Exception();
}
```

When the exception is thrown, it will send a signal directly to the catch block, bypassing the wrapperMethod. Exceptions that are not caught in code are eventually trapped by the Java VM.

Java contains definitions for about fifty types of exceptions.[12] We define a new class to represent the explicit return signal. By subclassing from RuntimeException, we obviate the need for the *throws* clause in the method definitions.

The solution then is to put a try block around the body of every method that defines a block. The corresponding catch block will extract and immediately return the value stored in the exception object.

If the context of the method that defined the block is no longer active, then the exception will not be caught in code (since the try block will already have completed normally) but rather by the Java VM. Thus by calling the exception ContextCannotReturn, we mimic the behaviour of Smalltalk.

A final detail to examine is dealing with nested blocks. The following Smalltalk code shows an example of this:

```
nestedBlocks
   [ [ ^ 'inside inner block' ] value.
      'inside outer block' ] value
   ^ 'after blocks'
```

The exception should only be caught by the defining method, never by the special blocks method (since that method just represents the body of some other block). Therefore a try is placed around the body of all methods that define blocks, except for in the special blocks method.

[12] See [FLAN96] pp. 349-364.

2.5.6 Summary

We have a technique for converting arbitrary Smalltalk blocks to fully functional Java counterparts. The solution requires the following three base classes:

- ContextBlock.java
- Scope.java
- ContextCannotReturn.java (to signal explicit returns)

The solution also requires a special blocks method to be created for every class file that is generated. This allows a block to be evaluated with an ordinary Java method.

2.6 Smalltalk Class vs. Java Static

Although similar, the behaviour of the static side of a Java class file is not precisely the same as the class side of a Smalltalk object. Table 4 highlights the differences and similarities.

Smalltalk	Java
class variables visible to all instances	static variables visible to all instances
class can be passed as an object	class cannot be passed as an object
class side has access to *self*	static side does not have access to *this*
class side inherits from superclass	static side does not inherit from superclass

TABLE 4: Comparison of Smalltalk and Java language features in terms of the class/static side.

As in section 2.5.4, where we bypass the Java scoping mechanism, we reconcile this Smalltalk/Java discrepancy by bypassing the Java static side. Every time a class file is generated, we also generate a special *metaclass*. The class side of every object is represented by a single instance of this metaclass. That is, a class method in ClassA becomes an instance method in MetaClassA.class.

The inheritance hierarchy of the metaclasses is the same as that of the classes. Those familiar with Smalltalk will recognize this as essentially the same way that Smalltalk handles the issue. The difference is that Smalltalk metaclasses are all instances of the same class, while we generate a new class file for each. Smalltalk classes are represented at a much higher level, while we want to use the basic Java resources as much as possible.

To ensure that there is exactly one instance of each metaclass, and that the instance is globally visible, we maintain a single, global, hashtable. Again, this is the same way that Smalltalk deals with the issue, and for consistency, this table is called *Smalltalk.* Each entry in the table is keyed by a string containing the name of the class. For example, an instance of MetaClassA is accessed with the key "ClassA".

2.6.1 The Smalltalk System Dictionary[13]

The two requirements of the system dictionary are that it be globally accessible and that there be only one instance. Both of these aims are met by creating a class called Smalltalk, with a hash table on its static side.

The first time that a Java class is referenced, the VM loads it from its class file (in the case of Java, this may involve reading over the internet). From this point, the VM always refers to its own local version of the class (permitting static variables, etc). There is an optional section of code that is evaluated one time (right after the class has been loaded). This code can be thought of as a class constructor instead of the more common instance constructor. For all generated classes, this method contains code to create an instance of the metaclass and put that instance into the system dictionary, with the class name as a key.

Since each class loads only its own metaclass, the section can be hardcoded and can therefore run very efficiently, making this the preferred technique. However, it requires the class to be accessed before the metaclass; i.e., a class must be referenced in order to make its metaclass accessible. In some cases, the metaclass may be accessed before the class, we deal with this by modifying the system dictionary accessor. If a non-existent key is accessed, the value of that key (i.e. the class name) is used to create and store the proper metaclass. The following code (taken from Smalltalk.java) illustrates this process.

```
      public static visualworks.Object at_(visualworks.Object key) {
```

[13] Hereafter, the Smalltalk system dictionary will be referred to as just the system dictionary to avoid confusion with the Smalltalk language.

```
        if(!systemDictionary.contains(key))
           systemDictionary.add(
              key,
              Class.forName("Meta" + key.toString()).newInstance());
        return systemDictionary.at(key);
     }
```

The real code adds some error handling to ensure that the metaclass definition is found, that there are no errors building the instance, etc., but this illustrates the basic idea.

To be clear, the hope is that the custom code will be used as much as possible, but if the metaclass is accessed before the class has been loaded, the general code in the accessor ensures that the metaclass is retrieved.

Smalltalk also uses the system dictionary for storing global variables (all Smalltalk classes are in fact stored in a global variable). Smalltalk.java can be used in the same way. However undefined globals (because of the above) are interpreted as classes. In general this is not a problem, since the source code is from a working Smalltalk application.

2.6.2 Summary

Every Smalltalk class, say ClassA, which is compiled to Java byte code creates two class files. One, ClassA.class, implements the instance side of the Smalltalk object. The other, MetaClassA.class, implements the class side of the Smalltalk object. A new base class, Smalltalk.java is introduced to manage the instances of the metaclasses. Smalltalk.java is also used to store all global variable bindings.

Since a Smalltalk class is represented by an instance of a Java class, it can be passed around in the same way as any other Java object. This behaviour is demanded by the Smalltalk code, and this solution is key to being able to translate Smalltalk objects to Java class files.

2.7 Inheritance

Smalltalk doesn't distinguish between the class and the instance side in terms of inheritance. Both sides are able to directly access their parent's member variables, and both sides can invoke an inherited method using the special object *super*. Java has a similar procedure for invoking inherited methods on the instance side.[14] Member variables can be directly accessed and super can be used to invoke the inherited methods. However, at the bytecode level, the inherited classname must be supplied. Figure 6 shows an example inheritance hierarchy. A method is defined then overridden in a derived class.

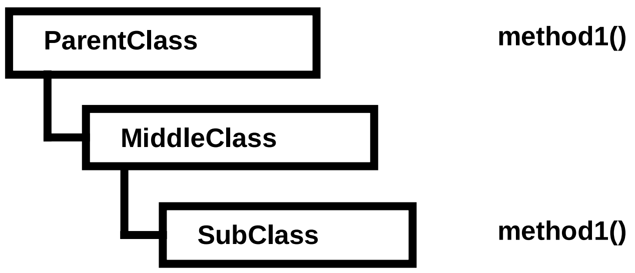


FIGURE 6: An example inheritance hierarchy which will be used to describe method and variable inheritance. ParentClass and SubClass provide implementations of method1(), but MiddleClass does not.

A Smalltalk method in SubClass could invoke ParentClass>>method1() using the special object *super*, as in:

```
super method1.
```

However the Java byte code requires that the proper classname be provided. Thus the bytecode:

```
invokespecial MiddleClass.method1()
```

would not work. The following should be used instead:

```
        invokespecial ParentClass.method1()
```

Since we translate classes not individually, but in groups this is not really a problem. Whenever a reference to the special object *super* is encountered in the Smalltalk code, we just look up the inheritance hierarchy to find the class that implements the selector. If the selector is not found, then Object is used as the implementor. In the final system, Object will implement all selectors using the 'DoesNotUnderstand' method body as described in section 2.4.1.

Member variables require a similar scheme, since at the bytecode level, the class must also be specified. The difference is that Smalltalk doesn't use the *super* object when retrieving an inherited member variable. The solution is to walk the inheritance hierarchy every time we generate code for member variable access and modification. Again, since the source is a working Smalltalk application, it has compiled correctly and therefore all member variables must be properly defined at an appropriate level in the tree.

[14] The static side is not applicable since, as described in section 2.6, it is not used in the generated class files.

3. Results

The techniques outlined in section 2 have been used to implement a tool capable of generating Java class files from Smalltalk objects. This tool performs the code generation but not the parsing. An existing parser was reused, so the tool starts not from the Smalltalk syntax, but from a parse tree. The tool has successfully handled the trivial test cases described in Appendix A. The intent of this section is to describe the translation of a more complex application.

3.1 Smalltalk Othello

3.1.1 Overview

The sample application is a two-player game of Othello[15], where both players use the same computer; i.e., there is no network communication. This application was chosen since it is moderately complex (the fileout of Smalltalk code is on the order of 100Kb). In addition, all user interaction is through the transcript, so we can ignore the more complex UI issues.

As shown in Figure 7, the application is comprised of six classes (which together define over 100 methods). Other than the absence of a true GUI, the application was written to take full advantage of Smalltalk's functionality. Blocks are used extensively, as in any other Smalltalk application. Additionally base Smalltalk classes such as OrderedCollection, Dictionary, and Set are used.

[15] See [ORUL] for the rules to the game of Othello.

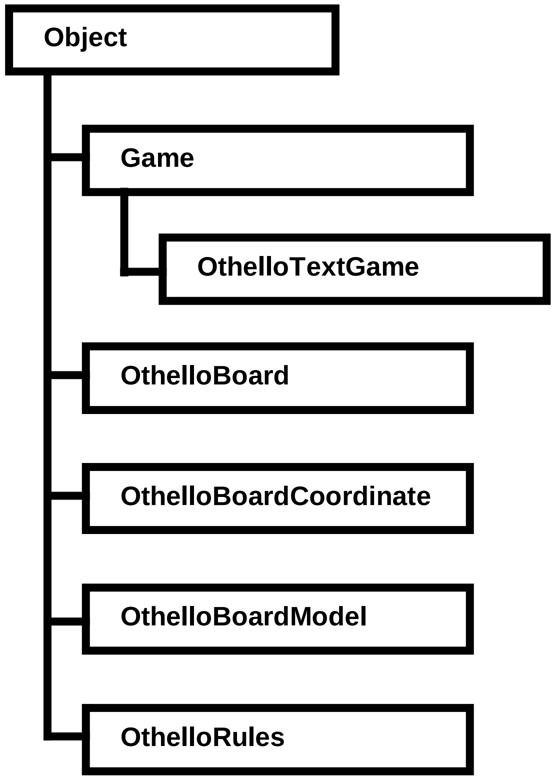


FIGURE 7: Architecture of the sample Smalltalk application.

3.1.2 The Translation Process

Since (as described in section 2.2.8) the tool is not able to translate primitive methods, it cannot translate the base Smalltalk classes such as OrderedCollection and Set. Instead, these classes were written in the Java language, and then compiled with a standard Java compiler. These custom created classes simply wrap their Java equivalents with the interface expected by the Smalltalk code. For example, visualworks.OrderedCollection.java stores an instance of java.util.Vector (Java's collection class) and contains a method definitions such as:

```
visualworks.Object add_(visualworks.Object);
visualworks.Object at_put_(visualworks.Object,
visualworks.Object);
visualworks.Object removeFirst();
. . .
```

These manually created classes include appropriate metaclasses to implement the class-side of the Smalltalk objects. For example, visualworks.__MetaOrderedCollection.java contains the following method definitions:

```
visualworks.Object with_(visualworks.Object);
visualworks.Object with_with_(visualworks.Object,
visualworks.Object);
. . .
```

Once the Smalltalk base classes are compiled, and the tool translates the Smalltalk code to Java, the UI has to be considered. We require a simple Transcript window that responds to #show:, #cr, and #tab. In addition the transcript has to be able to accept a highlighted line of text as input (the Smalltalk procedure for input is to highlight a snippet of code and *"do it"*). A Smalltalk transcript is able to execute any well formed Smalltalk syntax, our Java transcript is much more limited and in fact, only recognizes the two commands expected in the game:

- OthelloGame move: #d3[16]
- OthelloGame newGame.

The Transcript itself has been implemented with two classes, as shown in Figure 8.

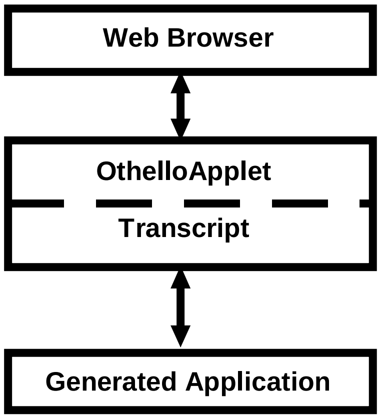


FIGURE 8: An illustration of the translated architecture, showing the interactions between the two classes forming the Transcript window.

[16] The argument can be anything in the range a1 to h8.

OthelloApplet is a pure Java class which inherits from java.applet.Applet, and as such can be embedded in any context appropriate to a Java applet (e.g., a web browser). It contains, among others the following method definitions:

```
public void show(java.lang.String);
public void cr();
public void tab();
```

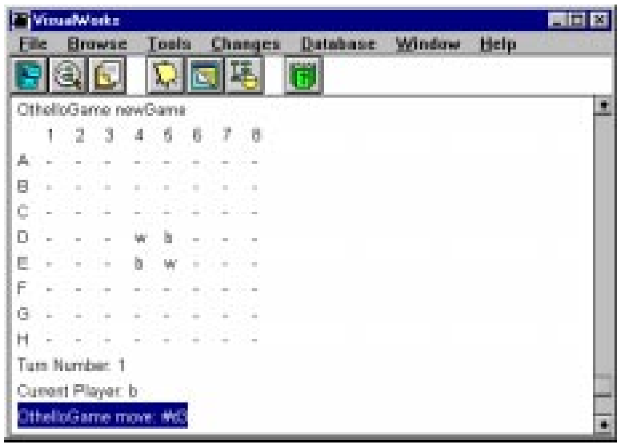
While OthelloApplet is responsible all interaction with the web browser, Transcript deals with the code translated from Smalltalk. The three methods cited above simply invoke their counterparts in the instance of Transcript (which is stored by OthelloApplet). When a user highlights some text and clicks the *"do it"* button, OthelloApplet invokes the following method in Transcript:

```
public void evaluate(java.lang.String);
```

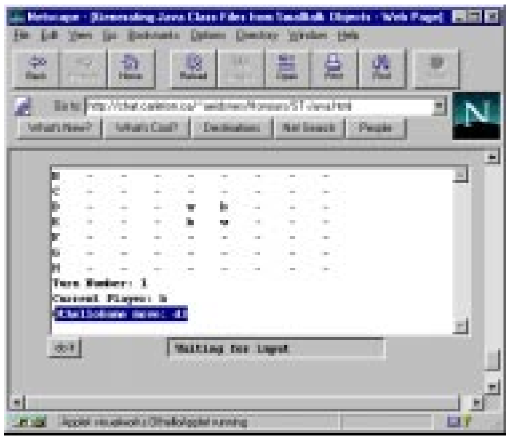
This method interprets the string (looking for one of the two commands), and invokes the proper method in the generated code.

Hence we have an application that can run on any Java VM (including those embedded in browsers) but was actually developed in Smalltalk. Figure 9 shows the original Smalltalk view as well as the view of the translated application.[17]

[17] This application is available at [HONSUP].



```
OthelloGame newGame
   1  2  3  4  5  6  7  8
A  -  -  -  -  -  -  -  -
B  -  -  -  -  -  -  -  -
C  -  -  -  -  -  -  -  -
D  -  -  -  w  b  -  -  -
E  -  -  -  b  w  -  -  -
F  -  -  -  -  -  -  -  -
G  -  -  -  -  -  -  -  -
H  -  -  -  -  -  -  -  -
Turn Number: 1
Current Player: b
OthelloGame move: #d3
```



```
B  -  -  -  -  -  -  -  -
C  -  -  -  -  -  -  -  -
D  -  -  -  w  b  -  -  -
E  -  -  -  b  w  -  -  -
F  -  -  -  -  -  -  -  -
G  -  -  -  -  -  -  -  -
H  -  -  -  -  -  -  -  -
Turn Number: 1
Current Player: b
OthelloGame move: d3
```

```
Waiting for input
```

FIGURE 9: Views of both the original Smalltalk Transcript window, and the application as an applet within a web browser.

4. Conclusion

This paper has proposed a method of translating Smalltalk applications into a form which can be executed on a Java VM. Smalltalk's extensive use of blocks, its non-typed nature, and the differences between Java's instance and static sides make such translation non-trivial indeed. As a test of these techniques, such a tool has been implemented and tested using a moderately complex applications.

Despite the successful translation of the sample application, the tool is still in the prototype stages. It is unable to deal with UI other than the basic text-based functionality provided by the transcript (and even this requires a small Java header to be manually implemented and compiled). Also, the tool is unable to translate any classes that use primitive methods. The following sections briefly outline proposed solutions.

4.1 Future Work

4.1.1 UI Builder

Smalltalk provides a two-step process to displaying a user interface. The interface is described in abstract terms, using interface specifications. These specifications describe the windows, scroll bars, buttons, etc that make up the interface. This specification is passed to a UIBuilder which builds the interface using components appropriate to the current platform. Smalltalk has a set of classes to invoke interface components on a Unix (motif) system, another set for a PC (windows), another for a Mac, etc. By implementing a set of these classes that are able to display the appropriate Java components, we could provide support for general UI's. This implementation would consist of nothing more than wrappers for AWT classes.

The UIBuilder itself is written in Smalltalk, and could therefore be converted to Java using the tool itself. Likewise, the specifications are extremely simple Smalltalk objects (in most cases, nothing more than an Array) and would be trivial to translate. In this manner it should be possible to provide support for general interfaces.

4.1.2 Smalltalk Base Classes

The ability to translate Smalltalk's base classes means that the tool must be able to deal with primitive methods. Once all primitives have been documented, it would be possible to create Java byte code strings which perform the same task. From this point it would be a trivial task to extend the tool to insert these byte code snippets every time the corresponding primitive is encountered.

This would allow the tool to translate any of the existing Smalltalk classes. A possible use of this would be to translate the Smalltalk collection hierarchy to make it available as a class library for tradition Java development.

5. References

[BERG97]	Berg Cliff, "How Do I Create a Java Bean", Dr. Dobb's Journal, September 1997.
[FLAN96]	Flanagan David, Java in a Nutshell, O'Reilly & Associates Inc., 1996.
[GAMMA94]	Gamma, et. al, Design Patterns - Elements of Reusable Object-Oriented Software, Addison Wesley Longman Inc., 1994.
[HIST]	Bergin Thomas & Gibson Richard, The History of Programming Languages, Addison Wesley Pub Co., 1996.
[LIND97]	Lindholm Tim & Yellin Frank, The Java Virtual Machine Specification, Addison Wesley, 1997.
[MEYE97]	Meyer Jon & Downing Troy, Java Virtual Machine, O'Reilly & Associates Inc., 1997.

5.1 Related Web Pages

[APPLIED]	Applied Reasoning: A way of creating a Java front end for a Smalltalk server: http://www.AppliedReasoning.com/cbframe.htm.
[CHIMU]	A syntax-level comparison of Java and Smalltalk: http://www.chimu.com/publications/JavaSmalltalkSyntax.html.
[HONSUP]	A web page with data relevant to this paper (including links to other sites listed in this section): http://www.cyberus.com/~eidsness/jasper.
[ORISA]	ORISA: A tool which forms hypotheses about variable types: http://www.softwarezentrum.de/orisa/Framed.Home.E.html.
[ORUL]	The rules to the game of Othello: http://www.armory.com/~iioa/othguide/faq/othellorules.html.
[STIC]	A language-level comparison of Java and Smalltalk: http://www.stic.org/Research/IDC97/stic.htm.
[STX]	A Java VM implemented in Smalltalk (link may be dead) http://home.t-online.de/home/exept/stja97_e.html.
[USENET]	A thread from comp.lang.smalltalk discussing the various way of converting from Smalltalk to Java. This discussion focuses on a source code level translation: <extremely long URL, use the link in [HONSUP] instead>

6. Appendix A: Test Plan

6.1 Introduction

This document describes a plan to test the ST-JAVA project. Java class files have been generated from several Smalltalk objects. These class files are described in section 2 and are the base of all test cases. Section 3 presents a table summarizing all test cases, and their current status. Section 4 describes each test case in more detail. For each case, there is a copy of the smalltalk source for applicable methods, the Java source (i.e. the test case driver), and the expected result. This document is intended to be used in concert with the test cases.

6.2 Overview

Several Java class files have been generated from Smalltalk objects. Additionally, a few support class files have been generated from Java source (e.g. Transcript, Array, etc.) These support classes are not shown here. See the main document for more information. There are three smalltalk classes; **VerySimpleTestParent**, **VerySimpleTest**, and **VerySimpleTestChild**. Figure A-1 describes the inheritance hierarchy for the generated class files. The following subsections show the Smalltalk source for each of the classes.

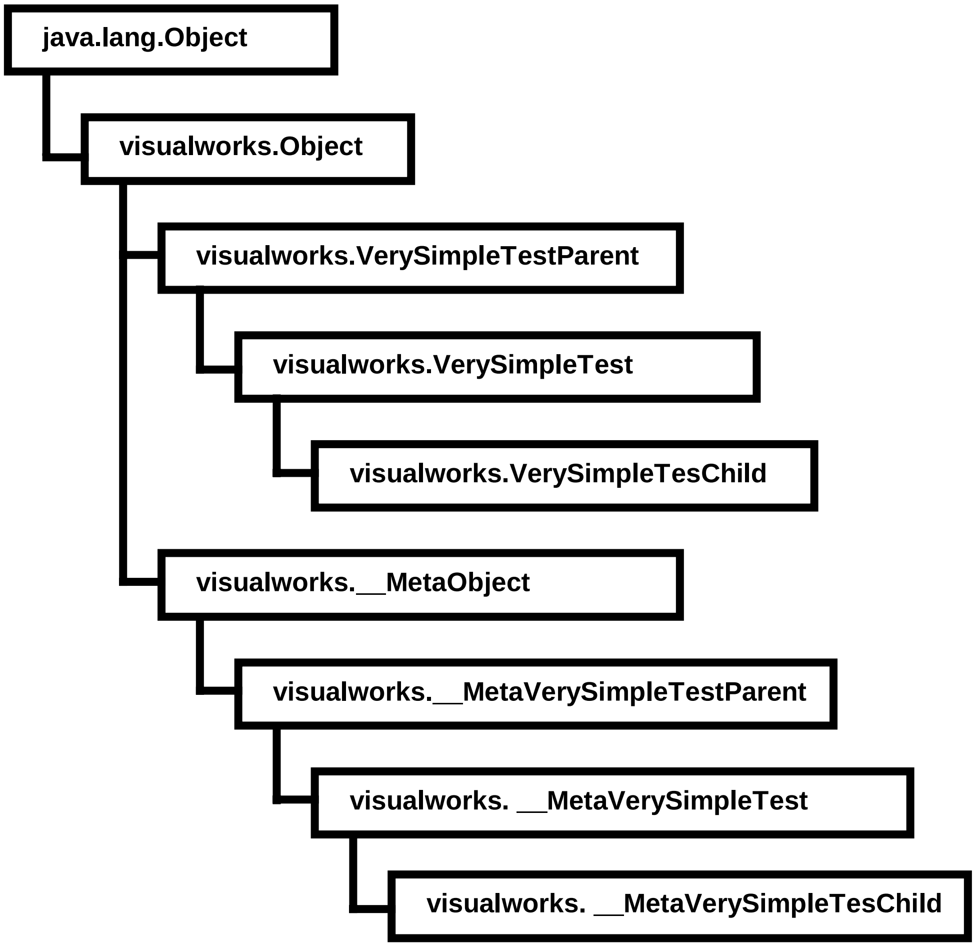

java.lang.Object
visualworks.Object
visualworks.VerySimpleTestParent
visualworks.VerySimpleTest
visualworks.VerySimpleTesChild
visualworks.__MetaObject
visualworks.__MetaVerySimpleTestParent
visualworks. __MetaVerySimpleTest
visualworks. __MetaVerySimpleTesChild

FIGURE A-1: Inheritance hierarchy for the test objects

VerySimpleTestParent

```
Object subclass: #VerySimpleTestParent
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'ST_Java-Samples'
```

VerySimpleTest

```
VerySimpleTestParent subclass: #VerySimpleTest
   instanceVariableNames: 'anInstVar '
   classVariableNames: 'AClassVar '
   poolDictionaries: ''
   category: 'ST_Java-Samples'
```

VerySimpleTestChild

```
VerySimpleTest subclass: #VerySimpleTestChild
   instanceVariableNames: ''
```

```
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ST_Java-Samples'
```

6.3 Test Case Status

Index	Test Case Title	Created On	Succeeded On	Page #
1	Local Variable Modification/Accessing	Feb 6, 1998	Feb 6, 1998	43
2	Basic Instance-side Block Evaluation	Feb 6, 1998	Feb 6, 1998	44
3	Instance Variable Modification/Accessing	Feb 6, 1998	Feb 6, 1998	43
4	Class Variable Modification/Accessing	Feb 6, 1998	Feb 6, 1998	44
5	Basic Class-side Block Evaluation	Feb 6, 1998	Feb 6, 1998	44
6	Class-side Messaging with Arguments	Feb 6, 1998	Feb 6, 1998	45
7	Evaluating Blocks With Arguments	Feb 6, 1998	Feb 6, 1998	45
8	Evaluating Nested Blocks	Feb 6, 1998	Feb 6, 1998	46
9	Evaluating a Block Stored in a Local Variable	Feb 6, 1998	Feb 6, 1998	46
10	Pass a Block as an Argument	Feb 6, 1998	Feb 6, 1998	47
11	Cascaded Messages	Feb 6, 1998	Feb 6, 1998	47
12	Instance-side Super Test	Feb 6, 1998	Feb 6, 1998	48
13	Class-side Super Test	Feb 6, 1998	Feb 6, 1998	50
14	Smalltalk Translated New	Feb 6, 1998	Feb 6, 1998	52
15	Messaging a Smalltalk Constructed Object	Feb 6, 1998	Feb 6, 1998	52
16	Chaining Message Sends	Feb 15, 1998	Feb 15, 1998	52

TABLE A-1: Test case titles with current status.

6.4 Test Case Descriptions

A small java program has been written to drive each test case. All test files can be found in the TestSuite directory. Each test case is described with three sections. The first lists the smalltalk source, the second gives the Java driver filename as well as the applicable code. The final section shows the expected result (this should be displayed on the window from which the test plan was executed). All test cases should be executed from a system prompt, using the java interpreter.

6.4.1 Local Variable Modification/Accessing

Smalltalk Source:

```
VerySimpleTest>>#valueToOne
    | local1 |
   local1 := 1.
   ^local1
```

Java Driver Program:

Test1.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
System.out.println(vst.valueToOne().toString());
```

Expected Result:

1

6.4.2 Basic Instance-side Block Evaluation

Smalltalk Source:

```
VerySimpleTest>>#useBlock
   [ ’VerySimpleTest - inside the block’ ] value.
   ^’VerySimpleTest - after the block’
```

Java Driver Program:

Test2.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
System.out.println(vst.useBlock().toString());
```

Expected Result:

VerySimpleTest - after the block

6.4.3 Instance Variable Modification/Accessing

Smalltalk Source:

```
VerySimpleTest>>#setInstVar
   anInstVar := ‘an instance variable value’
```

```
VerySimpleTest>>#getInstVar
   ^anInstVar
```

Java Driver Program:

Test3.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
vst.setInstVar()
System.out.println(vst.getInstVar().toString());
```

Expected Result:

an instance variable value

6.4.4 Class Variable Modification/Accessing

Smalltalk Source:

```
VerySimpleTest(class)>>#setClassVar
   AClassVar := ’a class variable value’

VerySimpleTest(class)>>#getInstVar
   ^anInstVar
```

Java Driver Program:

Test4.java:

```
visualworks.Smalltalk.at_(“VerySimpleTest”).setClassVar();
System.out.println(visualworks.Smalltalk.at_(
         “VerySimpleTest”).getInstVar().toString());
```

Expected Result:

a class variable value

6.4.5 Basic Class-side Block Evaluation

Smalltalk Source:

```
VerySimpleTest(class)>>#useBlock
   [ ^'VerySimpleTest(class) - inside block' ] value.
   ^'VerySimpleTest(class) - after block'
```

Java Driver Program:

Test5.java:

```
System.out.println(visualworks.Smalltalk.at_(
            “VerySimpleTest”).useBlock().toString());
```

Expected Result:

VerySimpleTest(class) - inside block

6.4.6 Class-side Messaging with Arguments

Smalltalk Source:

```
VerySimpleTest(class)>>#echoArg: anObject
   ^anObject
```

Java Driver Program:

Test6.java:

```
visualworks.Object result;
result = visualworks.Smalltalk.at_(“VerySimpleTest”).echoArg_(
         new visualworks.literals.Integer(10));
System.out.println(result.toString());
```

Expected Result:

10

6.4.7 Evaluating Blocks with Arguments

Smalltalk Source:

```
VerySimpleTest>>#useArgBlock
   ^[ :aBlockArg | aBlockArg ] valueWithArguments: #(654)
```

Java Driver Program:

Test7.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
System.out.println(vst.useArgBlock().toString());
```

Expected Result:

654

6.4.8 Evaluating Nested Blocks

Smalltalk Source:

```
VerySimpleTest>>#nestedBlock
   ^[[[[ ^true ] value ] value ] value. false ] value
```

Java Driver Program:

Test8.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
System.out.println(vst.nestedBlock().toString());
```

Expected Result:

true

6.4.9 Evaluating a Block Stored in a Local Variable

Smalltalk Source:

```
VerySimpleTest>>#localVarBlock
       | aBlock |
   aBlock := [ :arg | ^arg ].
   aBlock value: true
```

Java Driver Program:

Test9.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
System.out.println(vst.localVarBlock().toString());
```

Expected Result:

true

6.4.10 Pass a Block as an Argument

Smalltalk Source:

```
VerySimpleTest>>#useEvalBlock
   self evalBlock: [ :arg | ^arg ]

VerySimpleTest>>#evalBlock: aBlock
   aBlock value: true.
   ^false
```

Java Driver Program:

Test10.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
System.out.println(vst.useEvalBlock().toString());
```

Expected Result:

true

6.4.11 Cascaded Messages

Smalltalk Source:

```
VerySimpleTest>>#cascadeTest
   ^self getInstVar;
      valueToOne
```

Java Driver Program:

Test11.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
System.out.println(vst.cascadeTest().toString());
```

Expected Result:

1

6.4.12 Instance-side Super Test

This test is actually four tests rolled into one. It demonstrates that the super class method (and only the super class method) will be found when needed. The first sub-test case should succeed, the final three should be unable to find the proper method (and thus throw a DoesNotUnderstand exception).

Smalltalk Source:

```
VerySimpleTest>>#instanceParentExistTest
   Transcript show: 'VerySimpleTest>>#instanceParentExistTest -
START';  cr.
   super instanceParentExistTest.
   Transcript show: 'VerySimpleTest>>#instanceParentExistTest -
END'; cr

VerySimpleTest>>#instanceParentAbsentExistClassCurrentTest
   Transcript
      show:
'VerySimpleTest>>#instanceParentAbsentExistClassCurrentTest
                  - START';
      cr.
   super instanceParentAbsentExistClassCurrentTest.
   Transcript
      show:
'VerySimpleTest>>#instanceParentAbsentExistClassCurrentTest -
END';
      cr

VerySimpleTest>>#instanceParentAbsentExistClassTest
   Transcript show:
'VerySimpleTest>>#instanceParentAbsentExistClassTest
                  - START';   cr.
   super instanceParentAbsentExistClassTest.
   Transcript show:
'VerySimpleTest>>#instanceParentAbsentExistClassTest
                  - END';     cr

VerySimpleTest>>#instanceParentAbsentTest
   Transcript show: 'VerySimpleTest>>#instanceParentAbsentTest -
START';  cr.
   super instanceParentAbsentTest.
   Transcript show: 'VerySimpleTest>>#instanceParentAbsentTest -
END'; cr

VerySimpleTestParent>>#instanceParentExistTest
   Transcript show:
'VerySimpleTestParent>>#instanceParentExistTest';           cr

VerySimpleTest(class)>>#instanceParentAbsentExistClassCurrentTest
   Transcript show:

   'VerySimpleTest(class)>>#instanceParentAbsentExistClassCurrentT
est'; cr

VerySimpleTestParent(class)>>#instanceParentAbsentExistClassTest
   Transcript show:
```

```
   'VerySimpleTestParent(class)>>#instanceParentAbsentExistClassTe
st'; cr
```

Java Driver Program:

Test12.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();

vst.instanceParentExistTest();

System.out.println("");
try { vst.instanceParentAbsentTest(); }
catch(Throwable t) { System.out.println(t.toString()); }
System.out.println("");
try { vst.instanceParentAbsentExistClassTest(); }
catch(Throwable t) { System.out.println(t.toString());  }
System.out.println("");
try { vst.instanceParentAbsentExistClassCurrentTest(); }
catch(Throwable t) { System.out.println(t.toString());  }
```

Expected Result:

VerySimpleTest>>#instanceParentExistTest - START

VerySimpleTestParent>>#instanceParentExistTest

VerySimpleTest>>#instanceParentExistTest - END

VerySimpleTest>>#instanceParentAbsentTest - START

visualworks.DoesNotUnderstand: instanceParentAbsentTest

VerySimpleTest>>#instanceParentAbsentExistClassTest - START

visualworks.DoesNotUnderstand: instanceParentAbsentExistClassTest

VerySimpleTest>>#instanceParentAbsentExistClassCurrentTest - START

visualworks.DoesNotUnderstand: instanceParentAbsentExistClassCurrentTest

6.4.13 Class-side Super Test

This test is actually four tests rolled into one. It demonstrates that the superclasses' method (and only the superclasses' method) will be found when needed. The first sub-test case should succeed, the final three should be unable to find the proper method (and thus throw a DoesNotUnderstand exception).

Smalltalk Source:

```
VerySimpleTest(class)>>#classParentExistTest
   Transcript show: 'VerySimpleTest(class)>>#classParentExistTest
- START';   cr.
   super classParentExistTest.
   Transcript show: 'VerySimpleTest(class)>>#classParentExistTest
- END';  cr

VerySimpleTest(class)>>#classParentAbsentExistClassCurrentTest
   Transcript
      show:
'VerySimpleTest(class)>>#classParentAbsentExistInstanceCurrentTest
                  - START';
      cr.
   super classParentAbsentExistInstanceCurrentTest.
   Transcript
      show:
'VerySimpleTest(class)>>#classParentAbsentExistInstanceCurrentTest
                  - END';
      cr

VerySimpleTest(class)>>#classParentAbsentExistClassTest
   Transcript show:
'VerySimpleTest(class)>>#classParentAbsentExistInstanceTest
                  - START';   cr.
   super classParentAbsentExistInstanceTest.
   Transcript show:
'VerySimpleTest(class)>>#classParentAbsentExistInstanceTest
                  - END';     cr

VerySimpleTest(class)>>#classParentAbsentTest
   Transcript show: 'VerySimpleTest(class)>>#classParentAbsentTest
- START';   cr.
   super classParentAbsentTest.
   Transcript show: 'VerySimpleTest(class)>>#classParentAbsentTest
- END';  cr

VerySimpleTestParent(class)>>#classParentExistTest
   Transcript show:
'VerySimpleTestParent(class)>>#classParentExistTest';       cr

VerySimpleTest>>#classParentAbsentExistInstanceCurrentTest
   Transcript show:

   'VerySimpleTest>>#classParentAbsentExistInstanceCurrentTest';
cr
```

```
VerySimpleTestParent>>#classParentAbsentExistInstanceTest
   Transcript show:
      ’VerySimpleTestParent>>#classParentAbsentExistInstanceTest’;
cr
```

Java Driver Program:

Test13.java:

```
visualworks.Object vstClass =
visualworks.Smalltalk.at_(“VerySimpleTest”);

vstClass.instanceParentExistTest();

System.out.println(“”);
try { vstClass.instanceParentAbsentTest(); }
catch(Throwable t) { System.out.println(t.toString()); }
System.out.println(“”);
try { vstClass.instanceParentAbsentExistClassTest(); }
catch(Throwable t) { System.out.println(t.toString());  }
System.out.println(“”);
try { vstClass.instanceParentAbsentExistClassCurrentTest(); }
catch(Throwable t) { System.out.println(t.toString());  }
```

Expected Result:

```
VerySimpleTest(class)>>#classParentExistTest - START

VerySimpleTestParent(class)>>#classParentExistTest

VerySimpleTest(class)>>#classParentExistTest - END



VerySimpleTest(class)>>#classParentAbsentTest - START

visualworks.DoesNotUnderstand: classParentAbsentTest



VerySimpleTest(class)>>#classParentAbsentExistClassTest - START

visualworks.DoesNotUnderstand: classParentAbsentExistClassTest



VerySimpleTest(class)>>#classParentAbsentExistInstanceCurrentTest - START

visualworks.DoesNotUnderstand: classParentAbsentExistInstanceCurrentTest
```

6.4.14 Smalltalk Translated new (i.e. object construction without Java’s new)

Smalltalk Source:

```
VerySimpleTestChild(class)>>#new
   ^super new
```

Java Driver Program:

Test14.java:

```
System.out.println(visualworks.Smalltalk.at_(
         “VerySimpleTestChild”)._new().toString());
```

Expected Result:

visualworks.VerySimpleTestChild

6.4.15 Messaging a Smalltalk Constructed Object

Smalltalk Source:

```
VerySimpleTestChild(class)>>#new
   ^super new

VerySimpleTest>>#useBlock
   [ 'VerySimpleTest - inside the block' ] value.
   ^'VerySimpleTest - after the block'
```

Java Driver Program:

Test15.java:

```
System.out.println(visualworks.Smalltalk.at_(
            “VerySimpleTestChild”)._new().useBlock().toString());
```

Expected Result:

VerySimpleTest - after the block

6.4.16 Chaining Messages Sends

Smalltalk Source:

```
VerySimpleTest>>#chainedMessageTest
   ^self setInstVar getInstVar
```

```
VerySimpleTest>>#setInstVar
   anInstVar := ‘an instance variable value’

VerySimpleTest>>#getInstVar
   ^anInstVar
```

Java Driver Program:

Test16.java:

```
visualworks.Object vst = new visualworks.VerySimpleTest();
System.out.println(vst.chainedMessageTest().toString());
```

Expected Result:

an instance variable value