Issues in Translating Smalltalk to Java

R L Engelbrecht and D G Kourie
Object Technology Expertise Centre, Department of Computer Science, University of Pretoria, Pretoria 0002
dkourie@cs.up.ac.za, rudi@jupiter.cs.up.ac.za

Abstract: *A number of essential issues in translating Smalltalk to Java are addressed. A convention is proposed for mapping Smalltalk method selectors to Java method names. In addition, a Java class hierarchy that parallels the Smalltalk class hierarchy (including the metaclass objects) is suggested. These proposals are used to support ways of mapping both Smalltalk instance and class methods to their Java counterparts.*
Keywords: Smalltalk, Java, translation, Java byte code, instance methods, class methods, reflection, object oriented programming.

1. Introduction

Because of the availability of standardised Java Virtual Machines (JVM's) across a variety of platforms, languages other than Java are becoming as portable as Java itself. All that is required is a mechanism for translating source code written in the particular language into Java byte code (JBC). The resulting JBC can then be interpreted on any platform running a JVM. (See Lindholm and Yellin (1996), and Meyer and Downing (1997) for a comprehensive specification of the JVM.)

Translators to JBC as well as interpreters already exist for many source languages, including Ada (AppletMagic), BASIC (Halcyon Software), COBOL (Synkronix), C++ (Tilevich), Forth (Misty Beach Software), Scheme (Bothner). However, because of Smalltalk's unique characteristics, several challenging issues come to the fore when implementing a Smalltalk to JBC translator. The work of Chambers (1992) and Piamarta (1992) might offer some clues as to how certain problems might be resolved, but to the authors' knowledge no studies have fully resolved all the problems.

Smalltalk originated at Xerox PARC in the early 1970's. It is a dynamically typed programming language. Alan Kay, the chief architect of Smalltalk, summarises five basic characteristics of Smalltalk as follows (cited in Bergin and Gibson, 1987):

- Everything is regarded as an object.
- A program specifies a sequence of messages to be sent and received by a collection of objects, each object carrying out whatever action is implied by a message it receives.
- Each object has its own memory that may be made up of other objects.
- Every object has a type.
- All objects of a particular type can receive the same messages.

Full details on Smalltalk may be found in Goldberg (1981) and in Goldberg and Robinson (1983).

In contrast, Java is a relatively new, strongly typed language from Sun Microsystems, Inc. There are a number of similarities between the Smalltalk and Java environments, of which the following are perhaps the most pertinent.

1) *Object-oriented*: Smalltalk and Java are both object-oriented, dynamic languages.

2) *Interpreted*: Code produced in each of the environments is interpreted by a virtual machine. The standardised virtual machine used in Java is called the JVM and has already been mentioned above. There are also compilers in both environments that compile to native machine code for a specific platform.
3) *Garbage collection*: Objects that need no longer be retained in memory, do not need to be specifically removed by the programmer in order to free up memory. The environment automatically takes care of such memory management.
4) *Comprehensive class library*: Both Smalltalk and Java are released with an extensive set of classes available for reuse.
5) *Object references*: In general, objects are passed by reference (not by value) when a function call is made. Java has an exception in that when an object is one of a few primitive types (e.g. integer, double and float) then the object is passed by value.

This article supplements Smalltalk and other programming languages to JBC translation studies to date (e.g. Bothner, Hardwick and Sipelstein (1996) and Odersky and Wadler (1997)) by proposing solutions to key issues that have either not yet been resolved, or that have been resolved differently by other authors. Proposals put forward below are being implemented in a prototype Smalltalk to Java translator, which is in turn implemented in Smalltalk. However, the focus here is on broad design issues, rather than details of the prototype implementation. The general style of presentation is:

- to state a particular Smalltalk to JBC translation problem in generic terms;
- to provide examples of Smalltalk code that illustrate the problem;
- to suggest general Smalltalk to Java translation rules that resolve the problem (perhaps only partially); and finally
- to give Java code that illustrates the results of applying these rules.

Clearly, all derived Java code has its JBC equivalent. However, it is conceivable that a subset of Smalltalk code cannot be reasonably mapped onto Java code *per se*, but has to be mapped directly onto JBC. The present study provisionally excludes consideration of Smalltalk code that may be constrained in this way.

This article focuses on the following translation issues in turn. In section 2, a convention for mapping Smalltalk method selector names to Java method names is given. The next section addresses the matter of simulating Smalltalk objects in the Java typed environment. The translation of Smalltalk instance methods to Java instance methods is dealt with in section 4, while section 5 considers the translation of Smalltalk class methods to Java static methods. In section 6, some of the remaining problems in translating Smalltalk to Java are enumerated.

2. From Smalltalk method selectors to Java method names

In Smalltalk the method names are divided into three message groups: unary messages; binary messages; and keyword messages. A unary message is a message without arguments. A binary message is a message with a single argument and a selector that is one of a set of special single or double characters. A keyword message

has one or more arguments and a selector made up of a series of identifiers with trailing colons, one preceding each argument.

The first three examples in figure 1 illustrate messages belonging to each of the three respective message groups. The fourth example illustrates a keyword message with two arguments.

Unary message	frame minimize
Binary message	frame + field
Keyword message	frame moveTo: aNewLocation
Keyword message	frame replaceButton: button1 withNewButton: button2

Fig 1. Examples of Smalltalk Messages

It is relatively easy to devise rules for translating messages in each of the three message groups from their Smalltalk format to a suitable Java format. In general, each Smalltalk message sent to an object should be mapped to a Java invocation of the object's method using the Java notation <object>.<method_invocation>. The following rules are proposed for unambiguously translating the messages and their associated arguments and selectors to Java method invocations, including actual arguments where appropriate.

Note that these rules can also be used to deduce partially the corresponding Java method's declaration, although names for the formal parameters must be found with reference to the corresponding Smalltalk method's definition. Furthermore, for reasons that will later become clear, in declaring Java methods translated from their Smalltalk counterparts, it will be convenient to specify that they all return objects of type stj.Object.

1. A unary message is mapped directly to the equivalent Java method name without any arguments, i.e. minimize maps directly to the invocation minimize().
2. The selector of a binary message maps to a specially defined Java method name, the argument of the binary message becoming the actual argument of the corresponding Java method invocation. For example + argument1 maps to an invocation plus(argument1), where plus is a specially defined Java method name. In Smalltalk it is possible that the Integer class could redefine the behaviour of the + message. In Java, however, it is not possible to redefine the + keyword as it is part of the language definition. To provide for this Smalltalk functionality a lookup table will be used where + maps to plus and - maps to minus.
3. The sequence of identifiers in a selector of a keyword message maps to a single Java method name. This name is composed by joining the sequence of selector identifiers together as one long name, but replacing each occurrence of ':' by '_'. Furthermore, each argument of the keyword message becomes an actual argument (of type stj.Object – see below) of the Java method invocation. Thus, translations from Smalltalk methods to Java invocations will be as follows:
   moveTo:aNewLocation becomes moveTo_(aNewLocation)
   replaceButton:button1 withNewButton: button2 becomes
   replaceButton_withNewButton_(button1, button2)

If rule 1 or rule 2 maps to one of the reserved Java keywords, for example new or class, resulting in a method being named; class() or class(argument), it will be prefixed with stj_, resulting in stj_class() or stj_class(argument).

4. Note that these rules are indeed unambiguous. For example, if a unary Smalltalk message called plus existed, it would map to a Java invocation plus(), by the first rule. If a binary Smalltalk message + existed, it would map to a Java invocation plus(argument1) by the second rule. If a Smalltalk keyword message plus: argument1 existed it would map to the Java invocation plus_(argument1), by the third rule. In neither case is there any ambiguity with respect to the mapping in the example of the second rule.

Whenever one of the above Smalltalk messages is sent to the Smalltalk object frame, this corresponds to the invocation of a corresponding Java method using the syntax illustrated in figure 2 respectively:

Smalltalk code	Java Code
frame minimize	frame.minimize();
frame plus	frame.plus();
frame + field	frame.plus(field);
frame plus: field	frame.plus_(field);
frame moveTo: aNewLocation	frame.moveTo_(aNewLocation);
Frame replaceButton1: b1 withNewButton: b2	frame.replaceButton_withNewButton_(b1,b2);

Fig 2. Translations to Java method invocations

3. The Java class hierarchy for Smalltalk translations

It will be convenient to distinguish between Java objects derived from the Smalltalk translation, and other Java objects. In a Smalltalk system, all the objects have one root type, called Object. In the Java translation, this root type corresponds to a Java class denoted by stj.Object. It is a subclass of java.lang.Object and serves as the root class of all other Smalltalk-translated-to-Java objects. An arbitrary Smalltalk subclass of Object, say SomeClass will thus be translated to be a subclass of the Java class stj.Object and will be named stj.SomeClass. The translation algorithm should assure that this is done in a way that the original Smalltalk program's class hierarchy is retained.

The structure of the resulting Java class hierarchy is depicted in figure 3 below. Several advantages to be gained from this scheme will become apparent in later sections.

4. From Smalltalk- to Java instance methods

One of the matters to confront in the present context is the fact that Smalltalk is a dynamically typed language whereas Java is statically typed.

In the case of Smalltalk, therefore, local variables are not restricted to a specific class or type when they are declared. During runtime, an object of one class may be assigned to a local variable at some stage, and then an object of an entirely different class may be assigned to the same variable at a later point in the computation. Whenever a variable is used to represent an object that receives a message, then the message should obviously correspond to a method in the object's class or superclass. Since there is no restriction on what the object's class or superclass may be during

runtime, non-compliance with the foregoing results in a runtime error rather than a compile-time error.

In the Java case, the class of a variable is fixed at declaration: during runtime the variable can only be assigned an object of either its declared class, or of a subclass of its declared class. The variable's class declaration thus constrains the way in which the variable may be used to represent an object, and a violation of this constraint will be identified at *compile time*. In particular, if a variable representing an object is used as part of the syntax to invoke a Java method, and the method is not in the object's class or superclass (or superclass hierarchy), then a compile-time error results.

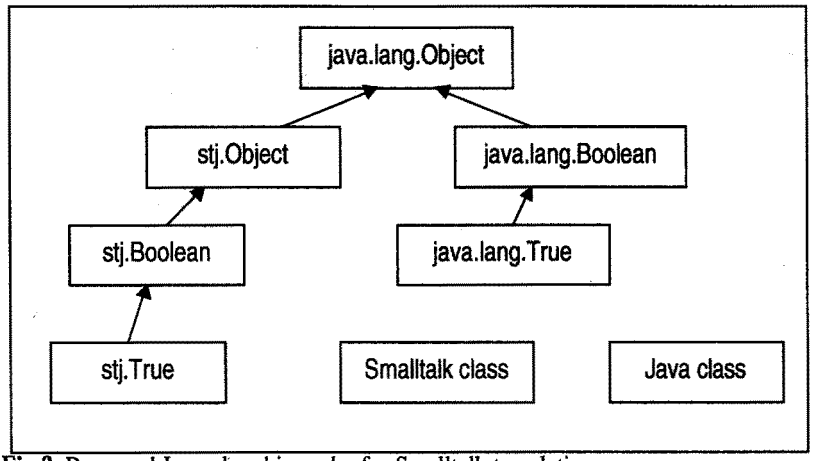

Fig 3. Proposed Java class hierarchy for Smalltalk translations

In summary, then, when a message is sent to an object in Smalltalk, the method to be executed is only determined at runtime by the virtual machine. In Java, the compiler restricts the callable methods by referring to the declared type of the object. As a consequence, two problems arise when doing a direct translation to Java: deciding what Java type should be assigned to Smalltalk-translated variables; and deciding how Java code should be constructed to simulate Smalltalk's runtime binding of methods. A solution to the first of these problems, and two solutions to the second problem are discussed below.

4.1 Java Types for Smalltalk Variables

While it might be possible, in some limited contexts, to infer a Java type that tightly constrains a Smalltalk variable, it is not possible to come up with a general scheme to do this. Consider, for example, the Smalltalk code in figure 4.

This purely hypothetical example consists of a method that accepts a Boolean argument `isBag` and an object as parameters. If `isBag` is true then an instance of `Bag` is assigned to the variable `aCollection`. The type of `aCollection` will be `Bag`. If `isBag` is false then `aCollection` will be assigned an instance of `Set`. The type of `aCollection` is then `Set`. Clearly, this runtime determination of the type of `aCollection` cannot be handled directly in Java. A solution in Java would be to declare the variable `aCollection` as the type of the most specific common superclass of both `Bag` and `Set`. In Smalltalk, `Collection` is the most specific superclass. However, it is not feasible to identify, as part of the translation

process, the set of possible classes that a variable might be typed as during runtime, and then to determine the most specific common superclass of classes in this set. It is far simpler merely to use the common superclass of all the Smalltalk objects in Java, namely stj.Object as the Java type of all Smalltalk local variables.

```
SampleClass > >returnCollection: isBag withObject: anInteger
    | aCollection |
    isBag  ifTrue: [aCollection := Bag new]
                ifFalse: [aCollection := Set new].
    aCollection add: anInteger
    ^aCollection
```

Fig 4. Smalltalk SampleClass example

The translated Java code of the above Smalltalk method is given in figure 5. Note that the translation rules lead to typing both isBag and anInteger as stj.Object. In order to carry out the test of the conditional statement, a Java class stj.True is defined (see figure 3). This class has a variable stj_true of type stj.True that has been initialized to a value equivalent to the Smalltalk object true. (Note: In Smalltalk there is one instance of the class True (namely true) and one instance of the class False (namely false). For a concise overview refer to Fussel.

```
public class SampleClass extends stj.Object {
 public stj.Object returnCollection_withObject_(stj.Object isBag,
                                                 stj.Object anInteger) {
     stj.Object   aCollection;
     try {
          if ( ((stj.Boolean)isBag).isTrue() )
               aCollection = new stj.Bag()
          else
               aCollection = new stj.Set();
     } catch(...) { /* isBag is not a standard boolean, lose */ }
     aCollection.add_(anInteger);
     return aCollection;                    }
```

Fig 5. Java SampleClass translation

In this Java code aCollection has been declared as type stj.Object. Since, by virtue of section 3, stj.Bag and stj.Set are both subclasses of stj.Collection, which in turn is a subclass of stj.Object, the respective assignments to aCollection will be accepted as correct by the compiler, provided that stj.Object or an appropriate subclass has a method defined as add_(stj.Object).

4.2 Reflection-based runtime method binding

The Java code above also contains an invocation to a method aCollection.add_(anInteger) that has been directly translated from the Smalltalk code above, using the translation rules of section 2. The assumption is that if and only if there was a Smalltalk instance method of add_ in the Smalltalk class Set (and/or Bag, and/or Collection and/or Object), then the same method would be translated into a Java instance method in the Java class

stj.Set (and/or stj.Bag, and/or stj.Object respectively). The Java translated code should behave as closely as possible in the Java environment to the original Smalltalk code in the Smalltalk environment, both at compile time and at run time. In particular, the Java environment should report an error at exactly the same point (compile time or run time) at which the Smalltalk environment would report it.

One possible approach to achieve this close simulation in Java would be to make use of reflection. (For an introduction to reflection refer to Bekker (1993) and Maes (1987).) With the release of the Java Development Kit (JDK1.1) by Sun Microsystems, Inc. a reflection API has been added whereby the Java environment can interrogate and act upon itself in various ways (JavaSoft (1997)). For example, a class can be interrogated for a list of all its methods; and an arbitrary method name can be assigned to a variable and used as a parameter in a call that, in turn, invokes whatever method that variable represents. All of this occurs at runtime. It is therefore possible to invoke methods of objects in a dynamic way.

Smalltalk incorporates Object>>perform: and Object>>perform:with: messages, whereby a message name can be constructed dynamically at runtime and then be sent to an object. Relying on the reflection API, the same functionality can be implemented in Java by adding matching methods to stj.Object called perform_(), perform_with_() etc. As an example, code for the perform_with_() method is given in figure 6.

```
public stj.Object perform_with_(java.lang.String methodName, stj.Object arg1) {
  stj.Object result = null;
  java.lang.reflect.Method method = null;
  java.lang.Class theClass = null;
  java.lang.Class[] argClasses = new java.lang.Class[1];
  java.lang.Object[] args = new java.lang.Object[1];
  theClass = this.getClass();
  args[0] = arg1;
  argClasses[0] = stj.Object.stj_nil.getClass();
  try { method = theClass.getMethod(methodName, argClasses); }
  catch (NoSuchMethodException e)
              { System.out.println("Error: NoSuchMethod"); }
  catch (SecurityException e) { System.out.println("Error: Security"); }
  try { result = (stj.Object) method.invoke (this, args); }
  catch (NullPointerException e) { System.out.println("Error: NullPointer); }
  catch (IllegalArgumentException e)
              { System.out.println("Error: IllegalArgument"); }
  catch (IllegalAccessException e)
              { System.out.println("Error: IllegalAccess"); }
  catch (InvocationTargetException e)
              { System.out.println("Error: InvocationTarget"); }
  return(result);          }
```

Fig 6. Java perform_with_ method

In essence, perform_with_() makes use of two reflection API methods within two try statements: getMethod and method.invoke respectively. The various associated catch

statements are required by the language definition, and should not distract from the overall understanding and logic of the perform_with_() method in the figure.

Consider the use of this reflective method in the context of the previous example (i.e. adding an element to a collection that may either be a bag or a set). Use of the perform_with_() method means that the translation of the Smalltalk code **aCollection add: anInteger** should be rendered as:

```
aCollection.perform_with_("add_", anInteger)
```

instead of as aCollection.add_(anInteger). The resulting Java system behaves exactly as its Smalltalk counterpart in the following senses:

- At compile time, no attempt is made to check that the arguments of perform_with_ make run-time sense, except to verify that actual argument types match those of the formal arguments. Compilation could be successful, even if there was no add_(anInteger) method in the entire system.
- If, at run time, aCollection is instantiated as an stj.Set object, then an stj.Set method, aCollection.add_(anInteger) is invoked. If such a method had not been defined in the stj.Set class, or in any predecessor class, then an appropriate run time error message is issued.

The foregoing remark applies *pari passu*, should aCollection have been instantiated as an stj.Bag object

It might be suspected that the above way of sending messages to objects in a truly dynamic way would be slow. In fact, tests done with the Sun JDK and Microsoft JDK indicate that it takes at least 200 times longer to invoke a method in this fashion, as compared with a normal method invocation. Until the JVM vendors provide optimised versions of the above methods used in the reflection API, this approach does not seem practical.

4.3 Superclass-based runtime method binding

This section discusses an approach to simulate Smalltalk's run-time instance method binding that is both simpler and more efficient than the reflection approach just discussed. It is therefore the preferred approach for use in the prototype under development. Two categories of methods are placed into the Java system:

1. As before, it is assumed that each method of Object and each method of its subclasses is translated into an equivalent Java method of stj.Object and its subclasses respectively.
2. In addition, for every *message* sent in the code of the Smalltalk system, a corresponding "default handler" method of the Java class stj.Object is constructed (provided that the message does not already have a corresponding method in stj.Object by virtue of 1. above.) This default method invokes the doesNotUnderstand method implemented on stj.Object. The implementation of doesNotUnderstand raises an exception to notify a debugger that an error occurred. This implies the complete Smalltalk program needs to be present during translation.

Thus, referring to the SampleClass example in figure 4, because aCollection add: anInteger appears in the Smalltalk code, the above rule specifies that a Java method add_ (stj.Object arg1) must be added to the class stj.Object, as illustrated in figure 7. below.

```
public class stj.Object {
  public stj.Object add_(stj.Object arg1) {
    System.out.println("Object > > doesNotUnderstand: #add:");
    return this;      } }
```

Fig 7. Superclass default method example

Recalling that the Smalltalk code **aCollection add: anInteger** was translated to the Java method invocation **aCollection.add_(anInteger)**, note that this invocation will always be regarded as type-correct by the Java compiler, irrespective of the class of object that the variable **aCollection** refers to (as long as the type of **aCollection** is a subclass of **stj.Object**). If, at some point during runtime, the variable **aCollection** refers to an instance of **Set**, and **Set** has no method called **add_**, then the **add_** method of the superclass **stj.Collection** will be used, or the **add_** method of **stj.Object** in that order.

By implementing the Java code in this way, not only are dynamically typed objects simulated, but the dynamic dispatch of messages at runtime is also simulated. Another benefit of this approach is that the speed of the resulting code approximates the speed of normal Java code with types in the variable declarations.

5. Smalltalk class method simulation

There is a subtle problem in translating Smalltalk class methods to Java. It is rooted in the fact that in Smalltalk, all classes are treated as first class objects. Java does not fully reflect this property. It is problematic if a straight translation is attempted that maps Smalltalk class methods to Java methods. The required static prefix in a Java class method declaration implies that the method does not possess dynamic properties such as those illustrated in the following Smalltalk example.

The example is based on a common practice in Smalltalk: to write a class method in a superclass that creates initialized objects for itself and its subclasses. Consider the **Vehicle** class in figure 8 below which is a superclass of the class **BMW**.

```
Vehicle is subclass of Object.
Vehicle class > > newInitializedObject
    | instance |
    instance := self new.
    instance initialize.
    ^instance
Vehicle > > initialize
    Transcript cr; show: 'Vehicle > > initialize called'.

BMW is subclass of Vehicle
BMW > > initialize
    Transcript cr; show: 'BMW > > initialize called'.
```

Fig 8. Smalltalk **Vehicle** class

Vehicle has the class method **newInitializedObject** and the instance method **initialize**. In the method **newInitializedObject**, an object is created by sending the message **new** to **self** (in this case **self** refers to the *class* object associated with the method) and a new instance of the class object is returned. Depending on the context in which the method executes,

it returns different types of objects. Evaluating Vehicle newInitializedObject returns an instance of the class Vehicle and BMW newInitializedObject will return an instance of the class subclass BMW. Furthermore, the former message calls Vehicle initialize, while the latter calls BMW initialize. Thus:

- Vehicle newInitializedObject will return an instance of Vehicle and print 'Vehicle >>initialize called', while
- BMW newInitializedObject will return an instance of BMW and print 'BMW >>initialize called'.

In attempting to simulate the above behaviour in Java, two approaches are outlined below. The first illustrates the problem caused by a direct translation to Java's static class methods, while the latter shows an alternative way in which dynamic class methods can be simulated in Java.

5.1 Static Java class methods

The approaches proposed in section 2, 3 and 4 to arrive at Java code from the Smalltalk code, indicate the following Java translations associated with Vehicle (and similar translations for BMW):

a) a Java class called stj.Vehicle for the Smalltalk class called Vehicle;
b) a Java subclass of stj.Vehicle called stj.BMW for the Smalltalk class called BMW;
c) in the Java class called stj.Vehicle, a Java class method called newInitializedObject;
d) in the Java class called stj.Vehicle, a Java instance method called initialize;
e) in the Java class called stj.BMW, another Java instance method called initialize;
f) the invocation: stj.Vehicle.newInitializedObject for any Smalltalk message Vehicle newInitializedObject;
g) the invocation: stj.BMW.newInitializedObject for any Smalltalk message BMW newInitializedObject.

Figure 9 shows these translations, where the Java class methods are declared with the required static prefix. A further class method called stj_Class() is provided in both stj.Vehicle and stj.BMW and is invoked from newInitializedObject in line 4. stj_Class() relies on the reflection API method forName (in lines 9 and 18 respectively) to return the class in which it is declared (either stj.Vehicle or stj.BMW). In line 4., the reflection API method newInstance() is then invoked to generate an instance of the returned class. In line 5., the initialize() method of this generated instance is invoked.

However, if tested with the following code, it will be found that this Java implementation does not behave as the Smalltalk counterpart.

```
1: stj.Object car = null;
2: car = stj.Vehicle.newInitializedObject();
3: car = stj.BMW.newInitializedObject();
```

Both in lines 2 and 3 is 'Vehicle>>initialize called' printed out. The reason for this is that Java static methods are truly static, resulting in Vehicle's stj_Class() being called in line 3. when one might have hoped that BMW's stj_Class() would be called instead. Consequently, an instance of Vehicle is then returned and not an instance of BMW. Clearly, then, an alternative approach to simulating Smalltalk class methods is required.

```
1.    public class stj.Vehicle extends stj.Object {
2.       public static stj.Object newInitializedObject () {
3.            stj.Object  instance = null;
4.            instance = (stj.Object)stj_Class().newInstance();
5.            instance.initialize();
6.            return instance;                                    }
7.       public static java.lang.Class stj_Class() {
8.            java.lang.Class thisClass = null;
9.            try { thisClass = java.lang.Class.forName("stj.Vehicle"); }
10.          catch (ClassNotFoundException e)
                  {System.out.println("Error: Class not found"); }     }
11.      public stj.Object initialize() {
12.           System.out.println("Vehicle > > initialize called");
13.           return this;                                          }
14.                                                               }
15.   public class stj.BMW extends stj.Vehicle {
16.      public static java.lang.Class stj_Class() {
17.           java.lang.Class thisClass = null;
18.           try { thisClass = java.lang.Class.forName("stj.BMW"); }
19.           catch (ClassNotFoundException e)
                  {System.out.println("Error: Class not found"); }   }
20.      public stj.Object initialize_() {
21.           System.out.println("BMW > > initialize called");
22.           return this;                                    }         }
```

Fig 9. Static Java Vehicle class

5.2 Dynamic Java class methods

The following indicates how the Java translation of Smalltalk classes can be designed to simulate dynamic binding of class methods. It is based on approximating in Java the Smalltalk class hierarchy (including meta classes). Smalltalk has the following (Goldberg and Robinson (1989, p269-p271)):

a) There are two kinds of objects in the system: those that can create instances of themselves (called classes) and those that cannot. Every object is an instance of a class.

b) Every class is a subclass of class **Object**. **Object** itself has no superclass.

c) Each class is itself an instance of a class, termed a metaclass. A metaclass has only one instance. The class **Object** is not excluded from this and also has a metaclass.

d) The hierarchy of metaclasses is rooted in the metaclass of **Object** and this hierarchy mirrors that of the associated class instances. However, whereas **Object** has no superclass, the metaclass of **Object** has a superclass called **Class**. All metaclasses are therefore subclasses of **Class**.

e) Metaclasses also being objects, are instances of a class called **Metaclass**.

The structure is depicted in figure 10, and includes the classes and metaclasses found in the Smalltalk system for **Vehicle** and **BMW**. Solid arrows in the figure represent subclass relationships, while dashed arrows represent instance relationships. Note carefully that **Object**'s metaclass is indeed a subclass of **Class**, in accordance with (d).

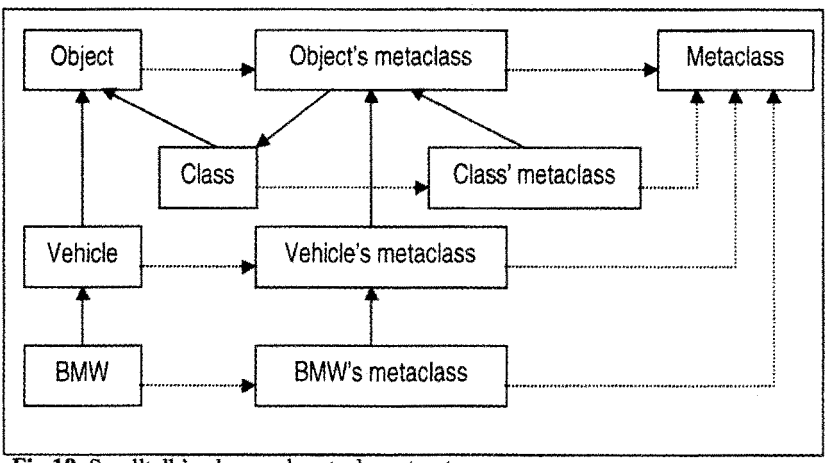

Fig 10. Smalltalk's class and metaclass structure

To provide for dynamic binding of Java class methods, it will be helpful to mirror this Smalltalk class hierarchy structure in the translated system, prepending each Java class by stj. as before. However, there will be no need to define a Java class stj.Metaclass. The name for the metaclass of Object will thus be stj.Object_metaclass and the name for the metaclass of Class will be stj.Class_metaclass.

The Java code is designed to simulate the metaclass of a class if and only if the class has a class method. This design allows for dynamically binding a class method when it is invoked at runtime. The principle is illustrated in figure 11 below, in terms of the previous Vehicle example.

Two classes, stj.Vehicle_metaclass and stj.BMW_metaclass have been defined with *instance* methods (which are thus dynamic) replacing the class methods (which had therefore been static) of the previously defined classes stj.Vehicle and stj.BMW respectively. In the case of stj.Vehicle_metaclass the relevant instance methods are newInitializedObject() and stj_new(). In the case of stj.BMW_metaclass the only relevant instance method is stj_new(). Note that the classes stj.Vehicle and stj.BMW are also defined, but each class defines only its original instance methods. (In each of these cases there is a single instance method, initialize().)

The result is that a Smalltalk class that has a class method is simulated by a composite of two classes in Java. The first Java class deals with Smalltalk instance methods as previously discussed. The second Java class contains the Smalltalk class methods in the form of Java instance methods. An instance of the Java class stj.Vehicle_metaclass has to be created before its instance methods can be used, and in this sense, the Java class behaves similarly to (i.e. simulates) a Smalltalk metaclass as described in (c) above.

The new version of the previously given code appears below.

```
1  stj.Object car = null;
2  car = (stj.Vehicle_metaclass.classInstance).newInitializedObject();
3  car = (stj.BMW_metaclass.classInstance).newInitializedObject();
```

Functionally, this code seems to achieve the same as before: it assigns an instance of Vehicle to the variable car, prints out "Vehicle>>initialize called" assigns an instance of BMW

to the variable car, and then prints out "BMW > > initialize called". However, the call to the class method newInitializedObject() is now bound at runtime, resulting in the right methods being called. Specifically, lines 4 and 5 will now invoke the right versions of stj_new() and initialize() respectively.

```
1  public class stj.Vehicle_metaclass extends stj.Object_metaclass {
2     public stj.Object newInitializedObject(){
3           stj.Object instance = null;
4           instance = this.stj_new();
5           instance.initialize();
6           return instance;                    }
7     public stj.Object stj_new() {
8           return new stj.Vehicle();}                    }
9
10 public class stj.Vehicle extends stj.Object {
11    public stj.Object initialize() {
12          System.out.println("Vehicle > > initialize called");
13          return this;              }          }
14
15 public class stj.BMW_metaclass extends stj.Vehicle_metaclass {
16    public stj.Object stj_new() {
17          return new stj.BMW(); }                    }
18
19 public class stj.BMW extends stj.Vehicle {
20    public stj.Object initialize() {
21          System.out.println("BMW > > initialize called");
22          return this;              }          }
```

Fig 11. Dynamic Java Vehicle class

Note that stj.Vehicle_metaclass.classInstance relates to an aspect of the Java metaclass simulation that, for reasons of brevity and clarity, has not been fully elaborated in figure 11. In the full version, a static variable called classInstance of the class stj.Vehicle_metaclass is declared. This variable is instantiated to an instance of stj.Vehicle_metaclass at start up time and may thereafter be used as in the context above. The *single* instantiation that occurs in the Java translation mirrors the fact that a Smalltalk class is a *single* instance of its corresponding metaclass. Arbitrarily to create multiple instances of stj.Vehicle_metaclass would violate the Smalltalk paradigm. The same applies for stj.BMW_metaclass.classInstance.

6. Future work

Several Smalltalk to Java translation issues have not been addressed in the previous paragraphs. The most important of these are briefly indicated below.

a) Smalltalk class variables can be represented as instance variables in the translated Java "metaclass" object. Thus, in order to translate a Smalltalk class variable, say AllVehicles, in the Smalltalk Vehicle class, an instance variable stj.AllVehicles should be added to the Java "metaclass" stj.Vehicle_metaclass. Locating a constructor in the

"metaclass" will ensure that the simulated class variable is initialised during runtime, as is the case for initialising the corresponding Smalltalk class variable.

b) Smalltalk class instance variables can be translated in the same way as class variables in (a). In Smalltalk all class instance variables start, by convention, with a lower case while class variables start with an upper case. Provided this naming convention is adhered to, there will be no conflict if the translation rule in a) is also applied to instance variables.

c) The Smalltalk method **allInstances** returns a collection of all the instances of a specific class. It is a class method. To implement **allInstances** in Java would involve enhancements to each class in the stj.Object hierarchy. Each class would have to keep track of its instances (i.e. objects) created by the use of **stj_new()**.

d) In Smalltalk it is possible to add a method dynamically (i.e. at runtime) to an object. In Java, a class has to be completely recompiled for methods to be added. Initial investigation resulted in a way of adding classes at runtime, but requires existing class instances to be migrated to the new version of the class.

These unresolved issues do not constitute a complete list. Indeed, several other issues require further study. For example, Budd (1987) discusses problems associated with implementing a Smalltalk compiler and virtual machine. Another important outstanding issue is that of translating Smalltalk blocks to Java. These issues are the subject of ongoing studies.

7. Conclusion

In one sense, this article provides the translation semantics of key Smalltalk constructs in Java. In doing so, it serves to highlight the similarities and differences between the two languages. The fact that a significant part of Smalltalk code can be translated to Java code by applying a few simple rules testifies to the similarities between the languages. Section 6 has identified further areas where, in principle, translation seems possible. There are additional areas where translation appears to be infeasible (for example, Smalltalk's **become:** method). Nevertheless, this work offers *prima facie* evidence that it will be possible to obtain reasonably efficient JBC versions of most Smalltalk programs. To this extent it will be possible to extend the range of platforms on which such Smalltalk programs can be run.

References

AppletMagic, *http://www.appletmagic.com.*

Bekker C, *Relationships and Reflection in the Object-Oriented Paradigm*, M.Sc. Dissertation, Department of Computer Science, University of Pretoria, 1993.

Bergin TJ and Gibson RG, *History of Programming Languages - II*, Addison-Wesley, 1996, Kay AC, *The Early History of Smalltalk.*

Bothner P, *Translating Smalltalk to Java*, http://www.cygnus.com/~bothner/ smalltalk.html

Bothner P, *Kawa, The Java-based Scheme system*, http://www.cygnus.com/ ~bothner/kawa.html.

Budd T, *A Little Smalltalk*, Addison-Wesley 1987.

Chambers C, *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, Ph.D. Thesis, Department of Computer Science, Stanford University, March 1992.

Fussel ML, *Java and Smalltalk syntax compared*, http://www.chimu.com/publications/JavaSmalltalkSyntax.html

Goldberg A and Robson D, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley 1983.

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

Goldberg A, *Introducing the Smalltalk-80 System*, Byte, Vol. **6**, No. 8, Aug. 1981

Halcyon Software, *http://www.vbix.com.*

Hardwick JC and Sipelstein J, *Java as an Intermediate Language*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1996.

JavaSoft, *The Java Core Reflection API and Specification*, http://java.sun.com. January 1997.

Lalonde W and Pugh J, *Inside Smalltalk: Volume 1*, Prentice Hall, Inc. 1990.

Lindholm T and Yellin F, *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley 1996.

Maes P, *Concepts and Experiments in Computational Reflection*, Proceedings of OOPSLA 87, 1987.

Meyer J and Downing T, *Java Virtual Machine*, O'Reilly & Associates, Inc. 1997.

Misty Beach Software, *http://www.mistybeach.com/Forth/JavaForth.html.*

Odersky M and Wadler P, *Pizza into Java: Translating theory into practice*, Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997.

Piamarta IK, *Delayed Code Generation in a Smalltalk-80 Compiler*, Ph.D. Thesis, Department of Computer Science, University of Manchester, October 1992.

Synkronix, *http://www.synkronix.com.*

Tilevich I, *http://pacevm.dac.pace.edu:80/~ny971734/c2j.html.*