# Implementing a Smalltalk to Java Translator

by

Roelof Lourens Engelbrecht

Submitted in fulfilment of the requirements for the degree Magister Scientiae

in the Faculty of Natural & Agricultural Science

University of Pretoria

Pretoria

June 2002

© University of Pretoria

# TABLE OF CONTENTS

**1. INTRODUCTION** ..... **8**

**2. FUNDAMENTALS** ..... **11**

2.1. SMALLTALK ..... 11
*2.1.1. Objects and Associated Concepts* ..... *13*
*2.1.2. Lambda closures, Smalltalk blocks and deferred execution* ..... *19*
*2.1.3. Execution Flow Control* ..... *20*
*2.1.4. Generating output* ..... *22*
2.2. JAVA ..... 24
*2.2.1. Objects and Associated Concepts* ..... *24*
*2.2.2. Inner Classes* ..... *29*
*2.2.3. Exceptions* ..... *32*
*2.2.4. Reflection* ..... *39*
2.3. SUMMARY ..... 39

**3. TRANSLATING FROM SMALLTALK TO JAVA** ..... **40**

3.1. METHOD SELECTORS ..... 40
3.2. CLASSES AND THE CLASS HIERARCHY ..... 43
3.3. DYNAMIC TYPES VERSUS STATIC TYPES ..... 44
*3.3.1. Type inference* ..... *45*
*3.3.2. Reflection-based runtime method binding* ..... *47*
*3.3.3. Superclass-based runtime method binding* ..... *51*
3.4. SIMULATING SMALLTALK CLASSES ..... 53
*3.4.1. Static Java class methods* ..... *54*
*3.4.2. Dynamic Java class methods* ..... *57*
*3.4.3. Smalltalk class variables* ..... *64*
*3.4.4. Smalltalk class instance variables* ..... *71*
*3.4.5. Smalltalk global variables* ..... *77*
*3.4.6. Java main methods* ..... *78*
3.5. TRANSLATING BLOCKCONTEXT OBJECTS ..... 79
*3.5.1. Simple block* ..... *81*
*3.5.2. Block with references to variables* ..... *83*
*3.5.3. Block with block arguments* ..... *86*
*3.5.4. Block contexts* ..... *88*
*3.5.5. Blocks with non-local returns* ..... *91*

*3.5.6. Nested blocks* ..... *98*
*3.5.7. Reference to self in block* ..... *103*
*3.5.8. Performance of blocks using exceptions* ..... *105*
3.6. SUMMARY ..... 106

**4. IMPLEMENTING A TRANSLATOR ..... 107**

4.1. THE SMALLTALK GRAMMAR ..... 110
*4.1.1. Method definition* ..... *110*
*4.1.2. Statements* ..... *111*
*4.1.3. Expressions* ..... *112*
*4.1.4. Blocks* ..... *112*
*4.1.5. Messages* ..... *113*
*4.1.6. Terminals* ..... *114*
*4.1.7. Reserved Identifiers* ..... *114*
4.2. LEXICAL AND SYNTAX ANALYSIS ..... 115
4.3. RECURSIVE DESCENT PARSER ..... 119
*4.3.1. MethodNode* ..... *122*
*4.3.2. BlockNode* ..... *124*
*4.3.3. AssignmentNode* ..... *125*
*4.3.4. VariableNode* ..... *127*
*4.3.5. MessageNode* ..... *128*
*4.3.6. SelectorNode* ..... *129*
*4.3.7. CascadeNode* ..... *129*
4.4. GENERATION OF JAVA CODE ..... 131
*4.4.1. MethodNode* ..... *132*
*4.4.2. Block Node* ..... *133*
*4.4.3. AssignmentNode* ..... *134*
*4.4.4. VariableNode* ..... *134*
*4.4.5. MessageNode* ..... *135*
*4.4.6. SelectorNode* ..... *136*
*4.4.7. CascadeNode* ..... *136*
4.5. OTHER TARGET LANGUAGES ..... 137

**5. CONCLUSION ..... 138**

5.1. RELATED WORK ..... 138
*5.1.1. Bistro* ..... *138*
*5.1.2. Smalltalk/JVM* ..... *138*
*5.1.3. Talks2* ..... *139*
*5.1.4. Smalltalk/X* ..... *139*

*5.1.5. Comparison with STJ* ........ *140*
5.2. FUTURE WORK ........ 142
*5.2.1. Smalltalk features* ........ *142*
*5.2.2. Translating the Translator* ........ *142*
5.3. SUMMARY ........ 143
**6. REFERENCES ........ 144**

# Abstract

A number of essential issues in translating Smalltalk to Java are addressed. The first chapter gives a brief overview of Smalltalk and Java with respect to the relevant language features that will be translated. In the next section a convention is proposed for mapping Smalltalk method selectors to Java method names. The dynamic nature of Smalltalk instance methods is compared with Java's static type model as well as a solution to simulate the dynamic nature of Smalltalk in Java. A Java class hierarchy that parallels the Smalltalk class hierarchy (including the metaclass objects) is suggested. A further proposal is given for translating the dynamic attributes of Smalltalk class methods to the same behaviour to Java. These proposals are used to support ways of mapping both Smalltalk instance methods, as well as Smalltalk class methods to their Java counterparts. Ways of translating Smalltalk class variables, Smalltalk class instance variables and Smalltalk global variables are illustrated.

A method for translating Smalltalk blocks to Java inner classes is implemented using Java exceptions to unwind the call stack. Various types of Smalltalk blocks are translated with increasing complexity. The various types of blocks translated are simple blocks; blocks with references to variables in the enclosing context; blocks with block arguments; blocks that need to refer to their own context executed from other contexts; blocks with multiple exit points as well as nested blocks. Some performance tests to illustrate the impact of using exceptions in Java are also reported.

The next section introduces the Smalltalk grammar with the necessary productions used to implement a parser. Lexical and syntax analysis are explained. A brief overview of a recursive descent parser is given where an example of Smalltalk source code is parsed and all the relevant parse nodes illustrated. The encoding in each parse node to Java source is shown.

The last section focuses on similar initiatives being pursued and compares the solutions in the dissertation against them. This dissertation focuses on key areas of the Smalltalk to Java translation process, but a few peculiar and unique Smalltalk features are not addressed. These are discussed in the last section and some suggestions are made on how the translations can be achieved.

# Samevatting

Verskeie probleme in die vertaling van Smalltalk na Java word ge-adresseer. Die eerste hoofstuk gee 'n kort oorsig oor Smalltalk en Java met klem op die relevante taal eienskappe wat van toepassing is op die vertaling. In die volgende afdeling word 'n oplossing voorgestel om die Smalltalk metode name te vertaal na Java metode name. Die dinamiese eienskappe van Smalltalk se instansiemetodes word vergelyk met Java se statiese tipe model so wel as 'n oplossing om die dinamiese eienskappe van Smalltalk in Java te simuleer. 'n Java klas-hierargie wat die Smalltalk klas-hierargie parallel (insluitend die metaklas-hierargie) word voorgestel. Nog 'n oplossing word voorgestel om die dinamiese eienskappe van Smalltalk klasmetodes na Java te vertaal en te verseker dat die vertaalde Java kode dieselfde gedrag openbaar as die Smalltalk kode. Hierdie voorstelle word gebruik om Smalltalk instansiemetodes, so wel as Smalltalk klasmetodes te vertaal na die gelyke Java kode. Verskillende maniere om Smalltalk klasveranderlikes, klasinstansieveranderlikes and Smalltalk globale veranderlikes te vertaal word ook geillustreer.

'n Oplossing om Smalltalk blokke te vertaal na Java binne klasse is geimplementeer deur Java se uitsonderingshanteringsmeganisme te gebruik. Verskeie tipes Smalltalk blokke word in trappe van toenemende kompleksiteit vertaal. Die verskeie bloktipes wat vertaal is, behels die volgende; eenvoudige blokke: blokke met verwysings na veranderlikes in die omsluitende konteks; blokke met blok argumente; blokke wat na hul eie konteks verwys en in ander kontekste uitgevoer word; blokke met verskeie uitkeerpunte sowel as geneste blokke. Spoedtoetse illustreer die impak wat die Java uitsonderingshanteringsmeganisme het op die vertaalde Java kode.

Die volgende afdeling stel die Smalltalk grammatika voor met die nodige produksies wat 'n herkenner implementeer. Leksikale- en sintaksanalise word verduidelik. 'n Kort oorsig van 'n rekursiewe herkenner word gegee waar 'n voorbeeld van Smalltalk kode herken word en al die relevante herkenningsnodes beskryf word. Die enkodering van elke herkenningsnode na Java kode word verduidelik.

Die laaste afdeling fokus op soortgelyke inisiatiewe wat besig is om dieselfde oplossings na te speur en vergelyk die oplossings in die verhandeling teen die huidige inisiatiewe. Die verhandeling fokus op sleutel areas van die Smalltalk na Java vertaalproses, maar 'n paar unieke Smalltalk eienskappe word nie geadresseer nie. Hierdie eienskappe word in die laaste afdeling bespreek en 'n paar voorstelle word gemaak oor hoe hierdie eienskappe vertaal kan word.

# Acknowledgements

All glory and praise be to Jesus Christ, without whom none of this would be possible.

I would like to thank my supervisor, Prof Derrick Kourie for all the necessary encouragement that made this work possible. It was his continuing support and enthusiasm that kept driving me towards completion. Without his drive, energy and immaculate talent for continuously picking up mistakes in the dissertation and pointing me in the right direction, this work would not have achieved the quality it has today.

Thanks are due to my wife, Natasha and my parents, Roelof and Loraine, who always believed in me, in what I wanted to achieve and kept on supporting me through this period.

I would also like to thank other parties at the University of Pretoria who at some stage interacted with me in my academic endeavors and played a major part in my education: Prof Roelf van den Heever with his sundowner topics and rich discussions, Prof Judith Bishop who lectured me on various topics throughout my undergraduate and postgraduate studies, Prof Jan Roos who showed me the interesting side of network architectures, Justine van den Bergh and Petra le Roux who had to put up with me in my years of undergraduate study and lastly but not least, Carel Bekker who mentored me through my undergraduate studies and introduced me to Smalltalk on a sunny Friday afternoon in April 1994.

# 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.) Terekhov and Verhoef (2000) argues in general that the complexities of source-to-source translations of programs are underestimated.

Translators to JBC as well as interpreters already exist for many source languages, including Ada (AppletMagic), COBOL (Synkronix), C++ (Tilevich), Forth (Misty Beach Software), Python (JPython), Scheme (Bothner) and SmallJava (Fussel). For a more current and comprehensive list of languages translating to JBC / Java refer to Tolksdorf (2002). Other translation to JBC / Java studies include Bothner et al. (1996), Odersky and Wadler (1997) and Hardwick and Sipelstein (1996).

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 Piumarta (1992) might offer some clues as to how certain problems might be resolved. More recent work is that of Boyd (2000) in which a few changes have been made to the Smalltalk grammar, the resulting language being renamed to Bistro. Bistro is thus a derivative of Smalltalk with support for types and there is a release of a translator that translates Bistro into Java source code. Another promising example of a Smalltalk to Java translator environment (named Smalltalk/JVM) has recently been released by Mission Software (2000). SPiCE by Yasumatsu and Koi (1995) is a solution for translating Smalltalk to C source. Waddington describes how the Java backend for GCC[1] is implemented. Section 5.1 addresses and compares various solutions for translating Smalltalk to other languages. The present dissertation supplements translation studies to date by proposing solutions to key issues that have either not yet been resolved, or that have been resolved differently by other authors.

Smalltalk 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)):

1. Everything is regarded as an object.

---

[1] GNU C Compiler – An open source C compiler for UNIX based operating systems

2. 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.
3. Each object has its own memory that may be made up of other objects.
4. Every object has a type.
5. All objects of a particular type can receive the same messages.

Full details on Smalltalk may be found in Goldberg (1981), Goldberg and Robinson (1983), Goldberg and Robinson (1989), Lalonde and Pugh (1990) and Pinson and Wiener (1988).

In contrast, Java is a relatively new, strongly typed language from Sun Microsystems, Inc. Java has been recognised as one of the most popular languages chosen by software engineers due, *inter alia*, to its simplicity in comparison to C/C++. The fact that Java is associated with the Internet through the inclusion of Java virtual machines in Web browsers has helped as well. Another area in which Java is gaining strong support is on the server side.

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 method is invoked. 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.

Proposals put forward below were implemented in a Smalltalk to Java translator, which is in turn implemented in Smalltalk. Broad design issues are discussed as well as the details of the implementation. The general style of presentation is:

1. to state a particular Smalltalk to JBC translation problem in generic terms;
2. to provide examples of Smalltalk code that illustrate the problem;
3. to suggest general Smalltalk to *Java* translation rules that resolve the problem;
4. to give *Java* code that illustrates the results of applying these rules; and finally

5. to implement a translator that translates Smalltalk code into Java

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 excludes consideration of Smalltalk code that may be constrained in this way.

This dissertation focuses on the following translation issues in turn. In Chapter 2 the Smalltalk and Java languages are introduced with examples illustrating the differences between the two languages. Chapter 3 introduces the following: a convention for mapping Smalltalk method selector names to Java method names; simulating Smalltalk objects in the Java typed environment; the translation of Smalltalk instance methods to Java instance methods; the translation of Smalltalk class methods to Java static methods and most importantly; the mapping of Smalltalk block closures to Java inner classes and Java exceptions. In Chapter 4 the Smalltalk grammar is explained with an explanation of a recursive descent parser. The chapter also describes how the Smalltalk front-end parser was modified and how a new back-end (translator) for the parser generating Java code was implemented.

It is assumed that the reader of this document has a good understanding of object-orientation and a background in an object-oriented language. The differences and nuances of Smalltalk and Java will be explained, but any further clarification should be sought in the references.

# 2. Fundamentals

A simple introduction to object-oriented software is given here. For a more comprehensive explanation refer to Meyer (1997).

An object consists of a set of attribute values and a set of operations that can be performed on the object. The state of the object is maintained by the attributes values stored in some private memory associated with the object. Depending on the nature of the object it will have different operations. For example, numeric objects will allow computational operations while data objects will allow operations for modifying and displaying information. The power of object-oriented computing lies in modeling the problem domain closely to the way it is defined in the real world without an additional mapping layer.

In the following two subsections, a brief overview is given of the two object oriented languages dealt with in this dissertation: Smalltalk and Java. The discussion focuses on the language features most relevant to the dissertation.

## 2.1. Smalltalk

Smalltalk originated at Xerox PARC in the early 1970's. The roots of Smalltalk can be traced back to applications like Sketchpad in 1962. Sketchpad implemented the concepts of clones and instances. In 1965 Simula was designed and built at the Norwegian Computing Center in Oslo by Ole-Johan Dahl and Kristen Nygaad. Simula was originally designed and implemented as a language for discrete event simulation, but was later expanded and re-implemented as a full-scale general purpose programming language. Although SIMULA never became widely used, the language has been highly influential on modern programming methodology. Among other things SIMULA introduced important object-oriented programming concepts like classes and objects, inheritance, and dynamic binding.

Smalltalk is based on principles of the Simula language. In 1972, Alan Kay (who had designed and built the first OOP-based personal computer called FLEX in 1967-68) and others at Xerox Palo Alto Research Center (PARC) created Smalltalk 72. This was later followed by Smalltalk 76, a completely object-oriented programming language. In 1980, Smalltalk 80, a uniformly object-oriented programming environment became available as the first commercial release of the Smalltalk language

Alan Kay had a vision for a simple, easy to understand language with the power of an object-oriented system. Kay's vision included making computers easier to use and in subsequent years he spent some time teaching Smalltalk to children and studied their problem solving skills being

applied to a visual development environment. The Apple Computer Vivarium Project was another project initiated by Ann Marion and Alan Kay and investigated the phenomena of learning. The system was called Playground. For an in depth discussion see Beck (1999:25-49).

The research carried out by Xerox (PARC) culminated in a seminal publication in the August 1981 issue of Byte magazine (Goldberg, 1981). The published articles described a Windows, Icon, Menu, Pointer (WIMP) based system, also known as a Graphical User Interface (GUI) where Smalltalk was introduced to the mainstream. Subsequent to the release of Smalltalk, Apple released a personal computer in 1983 called LISA. This model was modeled after prototypes at Xerox's PARC and featured a GUI and a mouse.

A group of employees at Xerox PARC persisted in developing a commercial implementation of Smalltalk in 1988 and so ParcPlace systems was formed. At the same time another company, Digitalk, created their own flavour of Smalltalk called Smalltalk/V and until the mid 1990's these two companies were the main providers of Smalltalk. IBM entered the Smalltalk market in 1994 with their VisualAge Smalltalk product aimed at corporate users.

Another interesting product called GemStone/S and based on Smalltalk is developed by GemStone Systems (formerly known as Servio Corporation). GemStone provides its own version of Smalltalk and implements a transparent object database.

Today Smalltalk is still in use, mostly at financial companies such as JP Morgan, Rand Merchant Bank, Equinox[2] and Numera Securities. These companies have a considerable investment in Smalltalk and more specifically, in GemStone. Smalltalk is regarded as one of the most efficient development environments for complex applications by those who use it.

Smalltalk never achieved the popularity of C/C++. Because of its characteristic of being an interpreted language, some people shied away from Smalltalk, making it not as popular as C++ and/or Java. The other hurdle could be that its different syntax makes people familiar with the C/Pascal syntax uncomfortable. The last aspect that probably prevented the wide adoption of Smalltalk is its unfamiliar, albeit powerful, working environment in which the focus is on defining messages that are sent to objects instead of an environment in which the focus is on compiling and executing source files.

---

[2] See http://www.equinox.co.za for an example of online financial transactions that are handled by GemStone Smalltalk

An open source implementation of Smalltalk called Squeak Smalltalk is available for all platforms at http://www.squeak.org. This is a close copy of the original Smalltalk-80 released in 1980 with plenty of modern class libraries. It provides a useful learning environment for those interested using Smalltalk. In the dissertation Squeak Smalltalk was used as the development environment while implementing the translator and the internal Squeak Smalltalk parser was modified and extended with an extra back-end producing translated Java code.

In Smalltalk almost everything that can be manipulated is represented as an object: the integers used to do numeric calculations; the strings used to display the result of the calculations; even the paragraphs and fonts used to store this document are represented as objects. In the following section, 2.1.1, the main concepts associated with Smalltalk objects are explained. The next section, 2.1.2, focuses on Smalltalk blocks, while section 2.1.3 deals with Smalltalk control structures. For a concise introduction to the Smalltalk grammar and syntax refer to section 4.1.

### 2.1.1. Objects and Associated Concepts

In Smalltalk, objects are "first class citizens". Objects may be classes, instances, abstract classes, meta-classes or even meta-objects. They are related to one another in an inheritance hierarchy and communicate with one another via messages. The subsections below elaborate on these themes.

#### 2.1.1.1. Classes

Each object belongs to a class – a certain type. This class describes the structure of all of its objects as well as their functionality. We refer to an object of a particular class as an instance of that class.

It is common practice to describe a class in terms of data and behaviour. In Smalltalk every object is an instance of a class. The data structure and behaviour (methods) defined on a class will be automatically assumed by the instances of the class.

The example below shows part of the definition of a class called `Rectangle`. It has four instance variables and two class variables. The instance variables characterize each instance of the class. Instance variables are encapsulated within their respective instance – i.e. they can only be manipulated from methods within the instance. An instance of this class would typically be used to represent some real-world rectangle, and its instance variables would define the properties (or attributes) and boundaries (dimensions) of the real-world rectangle.

```
Object subclass: #Rectangle
   instanceVariableNames: 'leftx rightx bottomy topy'
   classVariableNames: 'MaxWidth MaxHeight'
```

**Figure 1.** Definition of the Rectangle class

The class definition in Figure 1 specifies that a class named `Rectangle` is subclassed from a class named `Object`. The next line lists all the instance variables and the last line lists all the class variables.[3]

The lines below in Figure 2 create an instance, `rectangle`, of the `Rectangle` class.

```
| rectangle |
rectangle := Rectangle new.
```

**Figure 2.** Declaring a temporary variable and assigning an instance of Rectangle to it

This declares a temporary variable `rectangle`. The next line specifies that a method `new`, is sent to the `Rectangle` class and the result is assigned to the `rectangle` variable. The next section elaborates on this idea of sending messages to objects.

#### 2.1.1.2. Methods

An instance method is a piece of code belonging to a class that all its instances can execute. In this sense, the set of instance methods on a class defines the behaviour of the class instances. A class may also specify class methods. These methods define the behaviour of the class, as opposed to the behaviour of a class instance. A typical class method causes the class to create an instance of it.

In general, computation in Smalltalk is achieved by asking an object to perform a method by sending a corresponding message to the object. A simple message might be to ask a rectangle for its width. The object (the rectangle) that receives the message is known as the receiver. The message being sent, "`width`", is known as the selector.

---

[3] Keeping to the fundamental principles of Smalltalk where everything is an object and all operations are performed by messages sent to an object; the class `Object` is an object that understands the message `subclass:instanceVariableNames:classVariableNames:`. The arguments of the message are thus `#Rectangle`, `'leftx rightx bottomy topy'` and `'MaxWidth MaxHeight'`.

Continuing with the Rectangle example above, two instance methods of the Rectangle class are illustrated: width and initialiseLeft:right:top:bottom:. The width method computes and returns the width of a Rectangle instance, while initialiseLeft:right:top:bottom: provides a rectangle instance with initial values.

In Figure 3 below the syntax for defining instance methods is introduced. The name of the class, being Rectangle, is specified first. It is followed by the ">>"separator. The ">>"separator is followed by the name of the method. In Figure 3 the instance method named width is defined on the class named Rectangle as can be seen on the first line. If a class method needs to be defined, Rectangle is substituted with Rectangle class as illustrated in Figure 6 where a class method initialise is defined.

The second line in Figure 3 declares a temporary variable width used for temporary computations. One or more temporary variables can be defined inside the "|"separators. In the third line in Figure 3 the message abs is sent to the Integer object resulting from "rightx - leftx". The resulting absolute value is assigned to the instance variable called width. In the next statement, the value of width is returned to the sender of the message "width" to the rectangle instance.

```
Rectangle>>width
   | width |
   width := (rightx - leftx) abs.
   ^width
```

**Figure 3.** Defining an instance method named width on the Rectangle class

A useful instance method to define on Rectangle is a method that accepts all the initial values for a rectangle instance, which in turns initialises all the instance variables. Figure 4 below shows the implementation of initialiseLeft:right:bottom:top:.

```
Rectangle>>initialiseLeft: left right: right bottom: bottom top: top
   leftx := left.
   rightx := right.
   bottomy := bottom.
   topy := top.
```

**Figure 4.** Defining an instance method named initialiseLeft:right:bottom:top:

The following few lines in Figure 5 instantiates a Rectangle object. A new rectangle called rectangle is created by sending the message new to the class Rectangle, thereby invoking a class method (possibly inherited from a super class – see section 2.1.1.3) called new. The new rectangle instance, rectangle, is then initialised and asked for its width.

```
| rectangle |
rectangle := Rectangle new.
rectangle initialiseLeft: 10 right: 100 bottom: 10 top: 100.
rectangle width.
```

**Figure 5.** Creating a Rectangle instance, initialising it and asking it to compute its width

The convention in Smalltalk is to implement a class method or methods for instantiating objects and for initializing class variables. The following example defines a class method called `newLeft:right:bottom:top:` which creates a new instance of `Rectangle`. This allows validation to be performed on maximum rectangle sizes. An addition to this example is the use of class variables. Class variables are visible in both class and instance methods and serve to hold on to data that are available to all instances of a class. The first class method is named `initialise` and is called only once in the system to initialise the `MaxWidth` and `MaxHeight` variables. From then on all class and instance methods can access `MaxWidth` and `MaxHeight`. Note that the `initialise` class method has the same name as the instance method previously defined without conflicts.

```
Rectangle class>>initialise
  MaxWidth := 200.
  MaxHeight := 100.
```

**Figure 6.** Defining a class method `initialise` on the `Rectangle` class

The class method in Figure 7 creates an instance and then sends it the `initialiseLeft:right:bottom:top:` instance method. After the instance is initialised the instance is then returned to the sender of the message.

```
Rectangle class>>newLeft: left right: right bottom: bottom top: top
  | instance |
  instance := self new.
  instance
      initialiseLeft: left
      right: ( (right - left) > MaxWidth
                         ifTrue: [left + MaxWidth]
                         ifFalse: [right] )
      bottom: bottom
      top: ( (top - bottom) > MaxHeight
                         ifTrue: [bottom + MaxHeight]
                         ifFalse: [top] ).
  ^instance
```

**Figure 7.** Defining a class method `newLeft:right:bottom:top:` on the `Rectangle` class

An instance of the `Rectangle` class can now be created with one message to the class as can be seen in Figure 8.

```
Rectangle newLeft: 10 right: 100 bottom: 10 top: 100.
```

**Figure 8.** Creating an instance of Rectangle

### 2.1.1.3. Inheritance

Classes are arranged in a tree called the class hierarchy. This class hierarchy provides both a structural hierarchy and a functional hierarchy. A class inherits all variables (structure) and all methods (behaviour), from its superclass. This is in contrast with many later object orientation languages where inheritance is controlled by a variety of modifiers such as "private", "protected", etc. In Smalltalk the root class of all classes is `Object`. All classes are ultimately derived from Object. All classes that immediately inherit from Object are known as subclasses of `Object`. The complete structure of the Smalltalk class hierarchy is rather complicated and will be discussed in further detail in section 3.4

Structural inheritance means that a subclass of the `Rectangle` class discussed above, say `RoundedRectangle`, will inherit all the instance variables of Rectangle. Behavioural inheritance means that all the messages understood by `Rectangle` will also be understood by `RoundedRectangle`. The result is that `RoundedRectangle` will be able to be initialised with default values and return a width. A developer of a subclass may choose to redefine any inherited methods if so desired.

Encapsulation is an important attribute of object-orientation and enforces a discipline where instances may only be modified via methods on the instance. The methods to be used are known as the object's interface.

#### 2.1.1.4. Abstract classes

It is common practice to define some classes in an object-oriented system to be abstract. These classes serve the purpose of *abstracting* behaviour and locating it in a common class. This abstract class is made the superclass of all the subclasses that are required to exhibit the same behaviour. The useful feature is that an abstract class prevents the duplication of common code. A good example is the class `Magnitude` and its subclasses `Number` and `Date`. (Note: These and many other classes are provided by the Smalltalk system.) `Magnitude` is an abstract class and implements the max method. This instance method is invoked with one parameter. If the instance represents some value greater than the parameter, then the instance is returned, else the parameter is returned.

```
Magnitude>>max: aMagnitude
   "Answer the receiver or the argument, whichever has the greater
   magnitude."

   self > aMagnitude
      ifTrue: [^self]
      ifFalse: [^aMagnitude]
```

**Figure 9.** Defining the `max:` instance method on `Magnitude`

There is no need for the subclasses of `Magnitude`, `Number` and `Date`, to implement the `max` method as well since they inherit it from `Magnitude`. `Date`, `Number` or anything else that inherits from `Magnitude` need only implement the greater than (>) method. `Date` and `Number` instances will then automatically respond to the `max:` message.

The greater than (>) method for `Date` below complements the implementation of `max:` on `Magnitude` subclasses.

```
Date>> > aDate
"Answer whether aDate precedes the date of the receiver."
   year = aDate year
      ifTrue: [^day > aDate day]
      ifFalse: [^year > aDate year]
```

**Figure 10.** Implementation of greater than (>) to complement `max:` on the superclass

Abstract classes do not have instances. It is meaningless to create an instance of `Magnitude` or for that matter, even of `Object`. `Object` is thus another abstract class in the system and will stay an abstract class in the translated system.

#### 2.1.1.5. Meta-objects and meta-classes

Classes themselves are objects and are instances of the class `Metaclass`. A metaclass describes the class and the messages to which the class can respond. The messages invoke the methods that are known as class methods. To the compiler there is no difference between compiling instance and class methods. Instance methods are compiled in the context of the class and class methods are compiled in the context of the metaclass. This feature will be used in the same way when designing the translator and the generation of Java code.

### 2.1.2. Lambda closures, Smalltalk blocks and deferred execution

Blocks are Smalltalk's version of closures. They are similar to Lisp's anonymous lambda functions (Abelson, Sussman and Sussman, 1996:62-63). They are first class objects representing a piece of unevaluated code. These blocks can be executed at any time when needed.

Blocks can appear in any place where an expression is allowed. They can be assigned to variables, passed around and evaluated. They are instances of the class `BlockContext`. The syntax for defining a block looks like this:

```
[1 + 2]
```

**Figure 11.** Creating a block with one statement

The next example illustrates how a block can access variables outside itself.

```
| x |
x := 1.
[x := x + 1]
```

**Figure 12.** Creating a block with a reference to a variable "x" declared outside block

To evaluate the block it is sent the message `value`. Once the block has been evaluated the returned result is the result of the last statement executed in a block. A block can also accept arguments:

```
[:arg | x := x + arg]
```

**Figure 13.** Creating a block accepting one argument (`arg`)

When evaluating the block above, the result will be the value of the last statement executed, in this case, `x`, after assigning "`x + arg`" to `x`. This block will add `7` to the variable `x` when sent the message, `value: 7`.

In the section on translating Smalltalk blocks to Java, a number of additional features of a block will be explained. A good deal of effort has been spent in simulating this powerful feature of Smalltalk in the translated Java code.

### 2.1.3. Execution Flow Control

Smalltalk provides two types of flow control: conditional execution and looping. Conditional executions is provided by complementary definitions in the boolean classes. Looping is achieved by a combination of conditional execution and recursion.

#### 2.1.3.1. Conditional execution

Every object in Smalltalk is represented as an object of which `true` and `false` are prime examples. The following class hierarchy in Figure 14 shows `True` and `False` classes inheriting from `Boolean`. `Boolean` is an abstract class implemented with the common behaviour of `True` and `False`.

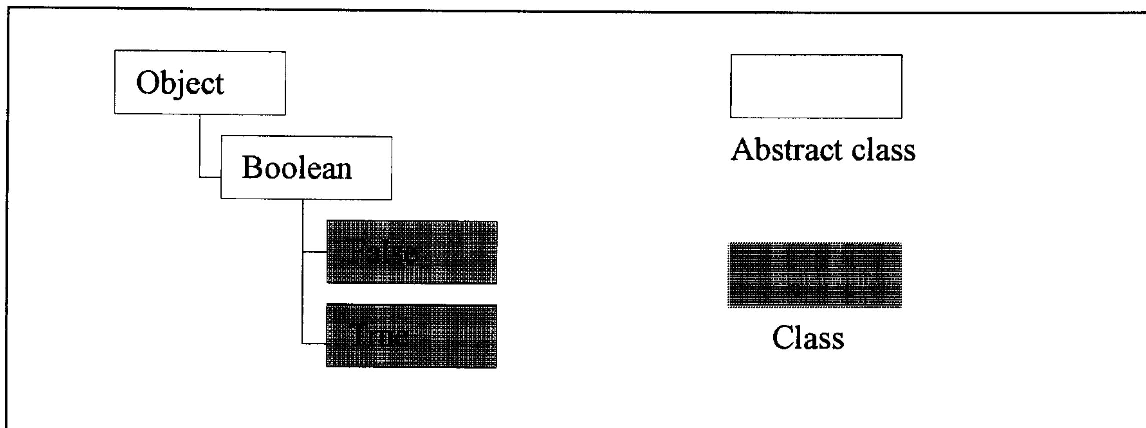


**Figure 14.** Boolean hierarchy with True and False subclasses

The `true` and `false` objects are each implemented as instances of `True` and `False`. There is only one instance of the `True` class, that being the `true` object. The same applies for the `false` object. This interesting artifact satisfies Smalltalk's principle of everything being an object and helps in implementing `True` and `False`.

```
x = 0 ifTrue: [^'Funds depleted']
```

**Figure 15.** Conditional execution based on the value of `x`

The example above shows how the variable `x` is compared to `0`. The result of the `=` message returns a *boolean* instance of the type `True` or `False`. This boolean instance is then sent the message `ifTrue:` and depending on the type of the instance a different implementation of `ifTrue:` is selected to execute. For example in class `True` we have

```
True>>ifTrue: aBlock
   ^aBlock value
```

**Figure 16.** Implementation of ifTrue: on the True class

so that the value of `[^'Funds depleted']` is returned if `x = 0` returns the `true` object. In class `False` we have the complementary definition

```
False>>ifTrue: aBlock
   ^nil
```

**Figure 17.** Implementation of ifTrue: on the False class

so that `nil` is returned if `x = 0` returns the `false` object.



i16034822X
b15432282

#### 2.1.3.2. Looping

Looping is provided by conditional recursion in messages understood by blocks. A simple loop that executes five times can be written as

```
| x |
x := 0.
[x < 5] whileTrue: [Transcript[4] cr; show: x printString. x := x + 1]
```

**Figure 18.** Illustrating a loop that prints `1` to `5` on the Transcript

The message `whileTrue:` is sent to a `BlockContext` instance (`[x < 5]`). The `whileTrue:` method is implemented recursively in `BlockContext` as

```
BlockContext>>whileTrue: aBlock
   ^self value
      ifTrue: [self whileTrue: aBlock].
```

**Figure 19.** Implementation of `whileTrue:` on `BlockContext`

Thus, the value of the block `[x < 5]` is evaluated, serving as the conditional of the `ifTrue:` method. If the result evaluates to true, then the value of the parameter block, `[Transcript cr; show: x printString. x := x + 1]` is evaluated, and the `whileTrue:` statement is executed recursively using the new value of `x`.

### 2.1.4. Generating output

When a Smalltalk program is required to output results to the standard output the following few lines of code are used:

```
Transcript cr; show: 'Hello World'
```

**Figure 20.** Printing `'Hello World'` on the Transcript

The `Transcript` variable is a reference to an instance of a `TextCollector` whose contents will be written to the graphical interface in a window.

---

[4] The Transcript reference is explained in section 2.1.4.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The first message that it is sent is `cr`. This tells it to start on a new line. The next message is `show:` which expects an object of type `String`. If numbers are to be written out then a simple `printString` message to the number object will return a `String` representation of it, ready for output. The next example prints a `3`.

```
Transcript cr; show: 3 printString
```

**Figure 21.** Printing an `Integer` on the `Transcript` by using `printString`

In the implementation it was decided to implement a subclass of `TextCollector` that transparently sent all requests to a file as well. This was useful in the testing framework where the output of all the Smalltalk classes was compared against that of the generated Java code to ensure the translator worked consistently. This new instance of `TextCollector` was named `STJTranscript`.

## 2.2. Java

Java, whose original name was Oak, was developed as a part of the Green project at Sun. Patrick Naughton, Mike Sheridan and James Gosling started it in December 1990. They were frustrated by the complexity and limitations of the C and C++ programming languages. At first the team thought to create an object-oriented development environment based on C++. In April 1991 the team decided on embedded systems software for smart consumer devices. James Gosling wrote the compiler called "Oak" and worked with the team to develop a runtime interpreter for what is known today as the Java language.

The first target for the Green project was to penetrate the smart consumer electronics market with an embedded operating system focusing on interactive television in particular. After failing to win a bid with Time Warner, the Green project was put on hold. At the same time the World Wide Web was starting to become popular and the Green team realised that the World Wide Web could be used as a delivery platform for applications based on their operating system. This meant that applications could be downloaded from a web server and executed in a web client.

Naughton implemented the first version of a Java based World Wide Web browser, named HotJava. It was wholly developed in Java and could execute Java applets. The HotJava browser was a showcase of the abilities of the Java language. It proved that Java could provide a secure, cross-platform execution environment for code to be downloaded and executed on the client. (SunWorld (1995)).

Java is similar to C/C++ in syntax. This section will explain only the features of Java necessary for the translation process by using the same examples as in section 2.1. An important aspect of Java is that it supports primitive types and object types. The primitive types are `int`, `long`, `double`, `float`, `boolean`, `byte`, `short` and `char`. In Smalltalk these types would be implemented as first class objects in the runtime environment.

### 2.2.1. Objects and Associated Concepts

Java differs from Smalltalk in that not every type is treated as a first class object. Java has a few primitive types, i.e. int, long and char. It is also not possible to send a message to a class object as in Smalltalk and expect the same behaviour as in Smalltalk. The normal response is to think that Java static methods will behave the same way as Smalltalk class methods, but they do not and section 3.4 discusses the issues with using static Java methods and why they do not behave in the same way as dynamic Smalltalk class methods. With the Java Reflection API introduced in Java

1.1 it became possible to load classes and create instances of a specific class via the Reflection API. An example of using the Reflection API is provided in section 2.2.4.

#### 2.2.1.1. Classes

For comparitive reasons the same examples will be used as in section 2.1. The class definition for `Rectangle` in Java is as follows

```
public class Rectangle {
   int leftx, rightx, bottomy, topy;
   static int MaxWidth, MaxHeight;
   }
```

**Figure 22.** Class definition of `Rectangle`

In contrast to Smalltalk where variables are dynamically bound, when declaring variables in Java it is required to specify the type[5] of the variable as well. To instantiate[6] an instance of Rectangle the following code is used

```
Rectangle rectangle = = new Rectangle();
```

**Figure 23.** Declaring a variable called `rectangle` of type `Rectangle`

---

[5] In Smalltalk classes play a similar role to types in Java. No distinction is made between types and classes in the translation process.

[6] It is not necessary to have a constructor for a Java class. If there is no constructor a default constructor is used that does nothing except for calling a superclass constructor.

#### 2.2.1.2. Packages

Java has the concept of packages where any class belongs to a certain package. To specify that Rectangle should be part of the graphics package the first line (before any class definitions) will have the package statement. Figure 24 shows the use of the package statement.

```
package graphics;

public class Rectangle {
   int leftx, rightx, bottomy, topy;
   static int MaxWidth, MaxHeight;
   }
```

**Figure 24.** Specifying that the Rectangle class belongs to the graphics package.

To access the Rectangle class in another area of the Java program the import statement must be used as illustrated below in Figure 25 and Figure 26.

```
import graphics.Rectangle;
```

**Figure 25.** Importing the Rectangle class from the graphics package

or

```
import graphics.*;
```

**Figure 26.** Importing all the classes in the graphics package

The class that is importing the graphics package can simply refer to Rectangle as if it was defined locally. If there is already another Rectangle definition in the local package then the reference to the Rectangle class must be fully qualified by the package name as shown below.

```
graphics.Rectangle rectangle = new graphics.Rectangle();
```

**Figure 27.** Referring to Rectangle by qualifying it with the graphics package name

When the package statement is omitted from a class definition the class belongs to the default package.

#### 2.2.1.3. Methods

To define a method in Java the definition of the method is included in the class definition as follows:

```
public class Rectangle extends Object {
   int leftx, rightx, bottomy, topy;
   static final int MaxWidth = 200;
   static final int MaxHeight = 100;

   public int width()
      {
      return abs(rightx - leftx);
      }

   public initialise()
      {
      leftx = 10.
      rightx = 100.
      bottomy = 10.
      topy = 100.
      }
   }
```

**Figure 28.** Definition of `Rectangle` as well as the `initialise` and `width` instance methods

Instance and class variables are defined in the class definition. The `static` modifier in the variable declaration indicates that the variable will be available to the class and instance methods. Any variable declared `final` (as in Figure 28 – `MaxWidth` and `MaxHeight`) in Java is a constant. A `final` (constant) variable may never be changed.

The following few lines declare a `Rectangle` reference, creates an instance, initialises it and asks it for its width.

```
Rectangle rectangle;
rectangle = new Rectangle();
rectangle.initialise();
rectangle.width();
```

**Figure 29.** Creating an instance of `Rectangle`, initialising it and asking it for its width

#### 2.2.1.4. Inheritance

To subclass from an existing class in Java the following syntax is used

```
public class Rectangle extends Object {}
```

**Figure 30.** `Rectangle` inheriting from class `Object`

In the translation process discussed in Chapter 3 all generated Java classes need to be accessible and the translator will generate the `public` class modifier by default. The other modifiers, `private`, `protected` and `final` will not be used in the context of the translated Java classes. More information about these modifiers can be found in Flanagan (1997:19-21).

#### 2.2.1.5. Abstract classes

The same principle of abstract classes in Smalltalk is found in Java. It is possible to define an abstract class in Java by using the abstract keyword as a modifier in the class declaration. Simply declare a class in the normal way and take care not to define any abstract methods with a method body. The abstract class declaration below has one abstract method and one normal method.

```
public abstract GraphicsObject extends Object {
   public abstract int computeArea();
   public boolean isViewable() {return true;};
   }
```

**Figure 31.** Declaring an abstract class with an abstract method `computeArea`

To use the class above will mean inheriting from it and overriding the abstract method, which can be done by a `Rectangle` class.

```
public class Rectangle extends GraphicsObject {
   public int computeArea() {return (rightx-leftx) * (topy-bottomy);};
   }
```

**Figure 32.** Rectangle inheriting from GraphicsObject and overriding computeArea

The effect of using abstract classes is that all subclasses of an abstract class, in this instance `GraphicsObject`, are required to implement the abstract method, in this case `computeArea()`.

#### 2.2.1.6. Meta-objects and meta-classes

It is not possible to treat a class in Java as a first class object. In other words, it is not possible to locate the class object at runtime and send messages to it in the same dynamic way as can be done in Smalltalk. In Java it is only possible to refer to the class at compile time and to invoke a static method on it. The reason for this is that in Java the reference to the class instance[7] is bound at compile time and will not be resolved dynamically at runtime as in Smalltalk. For a more complete discussion and examples of this problem and the challenges it poses in the translation process see section 3.4.

### 2.2.2. Inner Classes

Java does not have the powerful feature of Smalltalk known as blocks, but it does have inner classes that could be used to simulate Smalltalk blocks. Inner classes alone do not provide all the features that are required to translate Smalltalk blocks. As will be seen below, Java exceptions will be used in conjunction with inner classes.

Inner classes were added to the Java language specification as of version 1.1. With this addition to the language, an inner class can be defined as a member of another class, just as fields and methods can be defined as members of a class. It is possible to define a class within a block of Java code in the same way that local variables are defined in a block of code. Figure 33 will illustrate this with an example of an inner class. The four different types of inner classes are:

- Nested top-level classes
- Member classes
- Local classes
- Anonymous classes

For a complete explanation of the differences between all the different types of inner classes see Flanagan (1997:102-103). For the purpose of the translation the focus will be on local classes and anonymous classes.

---

[7] As will be explained in 3.4.2 each class in the virtual machine of either Smalltalk or Java can be modelled as an instance of another class – Metaclass.

#### 2.2.2.1. Local classes

A local class is an inner class defined in a block of Java code. The local class is only visible within the scope of the block of Java code. Because the local class is defined in a block of code it is similar to a local variable. The local class definition can thus be assigned to any other variable, as long as the types are compatible.

A local class has the following features:

- It is only visible and usable in the block of code in which it is defined – similarly to local variables.
- It can use any final variables or method parameters that are visible from the scope in which it is defined, be it a local, instance or global variable. It is not allowed to refer to any variable that is not declared final. The final variable can be referred to in assignment statements, but not assigned any other values.

In the following example a local class, `LocalClass`, is defined in the method, `createLocalClass`, of `Rectangle`.

```
public class Rectangle extends Object {
   int leftx, rightx, bottomy, topy;
   public int width() {…}  -defined as before
   public initialize() {…} -defined as before
   public createLocalClass() {
      class LocalClass {
         public localClassMethod() {
            return ("LocalClass method");
            }
         }
      LocalClass aClassObject = new LocalClass();
      return aClassObject;
      }
   }
```

**Figure 33.** Creating a local class and returning it as an object.

After creating an instance of `Rectangle` called `rectangle` and sending it the message `createLocalClass` a local class is defined and an instance of it is returned. Invoking `localClassMethod` on the returned instance will of course return the string "`LocalClass method`". The code below illustrates the example.

```
rectangle.createLocalClass().localClassMethod();
```

**Figure 34.** Invoking a method on a local class object

#### 2.2.2.1.1. Variable restrictions

Java has the concept of final variables. A final variable is very similar to a constant in other programming languages. When a variable is declared as `final` in Java its value cannot be changed. The use of `final` variables was necessary in translating Smalltalk blocks to Java inner classes.

There is one important aspect of local classes that involves accessing variables defined in the enclosing scope. The local class can only read the contents of the defined variables in the outer scope. It is not allowed to assign new values to it in the local class.[8] This restriction forces the use of an extra level of indirection as can be seen in section 3.5.2 on translating Smalltalk `BlockContexts`, by declaring an array variable as `final` and changing the contents of the array and not the array itself thus satisfying the constraint of `final` variables.

### 2.2.2.2. Anonymous classes

An anonymous class is essentially a local class without a name. Thus the variable restrictions discussed in section 2.2.2.1.1 apply as well. Instead of declaring a local class with one statement and then instantiating an instance of it to use in another statement, an anonymous class combines the two steps. Because it is not named it does have a side effect – it is instantiated and assigned once only.

The following code shows how the previous local class example can be rewritten to use an anonymous class.

---

[8] This makes the Java compiler's task easier by not allowing local classes to change the values of variables outside the declaration of a local class. The local class can only refer to final variables.

```
public class Rectangle {
[previous code here…]
   public createLocalClass() {
      LocalClass aClassObject = new LocalClass() {
         public localClassMethod() {
            return ("LocalClass method");
            }
         };
      return aClassObject;
      }
   }
```

Figure 35. Defining an anonymous local class and returning it

Once again, the code in Figure 35 will return the string "`LocalClass method`". Anonymous classes will be used extensively when translating Smalltalk blocks in section 3.5 where the translator is explained.

### 2.2.3. Exceptions

Java has exceptions built into the language that makes for powerful, robust code. Exceptions are used to signal that some sort of exceptional condition has occurred. For a more in-depth overview of exception handling in object oriented systems see Miller and Tripathi (1997:85-103). In Java the *throw* keyword is used to signal such an exception. To handle such an exception the *catch* keyword is used.

Exceptions propagate up through the lexical block structure of a Java method. If it is not handled then it will propagate up through the stack of method calls. An exception that is not handled by the block of code that throws the exception is propagated to the enclosing block of code. If the exception is not caught in the enclosing block of code it continues to propagate upwards. If it is not caught anywhere in the method then it is propagated to the invoking method, where it is again propagated through the block structure. If an exception is never caught it propagates all the way to the `main()` method of the running program. Depending on the Java virtual machine it may then cause the printing of an error message and a stack trace before causing the program to terminate.

The advantage of using exceptions is that it makes error handling more logical by grouping all the exception handling code in one place. Instead of testing the return value on every method invocation and handling it after every line, the code can be written more cleanly as shown in Figure 37. Compare traditional error handling given in `doSomething1` in Figure 36 below with the exception based error handling equivalent in doSomething2 in Figure 37.

```
public int doSomething1() {
   file  tempFile = new File("tempFilename");
   if (tempFile == null) {return -1;}
   result = tempFile.write("Test");
   if (result == null) {return -1};
   }
```

**Figure 36.** Traditional error handling

Figure 37 illustrates the difference between conventional error handling and using exceptions.

```
public int doSomething2() {
   try
      {
      File tempFile = new File("tempFilename");
      result = tempFile.write("Test");
      }
   catch (FileNotFoundException e1) {return -1;}
   catch (FileAccessException e2) {return -1;}
   }
```

**Figure 37.** Error handling using exceptions

In the case of `doSomething2` the error handling is not very complicated. It simply returns an error. For an interesting discussion of Java exceptions and their role in helping to free up resources acquired earlier in the code see Hunt and Thomas (2000:132-134).

#### 2.2.3.1. Exception objects

An exception is an object that is an instance of a subclass of `java.lang.Throwable`. `Throwable` has two standard subclasses: `java.lang.Error` and `java.lang.Exception`. The convention in Java is that subclasses of `java.lang.Error` are related to linkage, virtual machine or memory problems from which the system cannot recover gracefully. An application program should not catch these errors, but should merely terminate as these exceptions are related to errors outside the scope of the application program. Those errors that can be caught and recovered from gracefully are dealt with in predefined or user-defined exception subclasses of `java.lang.Exception`.

Since exceptions are objects, they contain data and define methods. Exceptions inherit from the `Throwable` class a String variable that is used to display explanatory messages about the exception that occurred. The fact that exception objects can contain data is used in the translation

process to be explained later, where exception objects are used to return BlockContext execution results.

### 2.2.3.2. Exception handling

A combination of try/catch/finally statements in Java are used to handle exceptions. The try statement encloses the block of code that needs to have its exceptions handled. The try block is followed by zero or more catch clauses that catch and handle specific classes of exceptions. The catch clauses are optionally followed by a finally block. The statements of the finally block are guaranteed to be executed.

Java requires that any method that can cause an exception must either catch the exception or specify the type of the exception with a throws clause in the method declaration – otherwise the compiler will not compile the method. This is useful if the code in the method does not want to handle the exception and simply wants to pass on the exception to the method where it was invoked. The method openFile below is an example

```
public void openFile() throws FileException {
   // Statements that might cause a FileException and not catch it.
   }
```

**Figure 38.** Definition of a method that might have statements that cause a FileException

Note that the exception class specified in a throws clause may be a superclass of all exception types thrown in the method's code. If a method throws either exception A or exception B, both of which are subclasses of C, then the method may specify both A and B in the throws clause or just C.

### 2.2.3.3. Throwing exceptions

Custom exceptions may be defined and later generated by the throw keyword. The throw keyword must be followed by an instance of Throwable or an instance of one of its subclasses. The code to create and throw an exception is seen in Figure 39.

```
Throw new CustomException("A Custom Exception occurred");
```

**Figure 39.** Creating an exception and throwing it

This example creates an object of type CustomException and passes a string to the default constructor where the string is assigned to an internal instance variable, in this case the default variable inherited from Throwable.

#### 2.2.3.4. Performance of exception handling

Because exception handling is used extensively in the translation process from Smalltalk `BlockContext` object to Java anonymous classes it was considered desirable to test the performance impact of this translation method on the generated target Java code.

The tests that were done involved a simple case where methods were enclosed in a `try` statement and followed by one or more `catch` clauses. They are described below.

A class `TestExceptionHandlingPerformance` class is defined with a constructor as well as with methods `testMethod`, `testMethod2`, `testMethodWithException` and `testMethodWithException2`; `testMethod` is a method without an exception, a clean invocation. The method `testMethodWithException` is a method that throws a `TestException`. The invocation of this method is wrapped in a `try` statement.

A variation on the test is to add a second level of method invocation without a `try` statement and declare the exception in the method declaration. The method `testMethod2` is invoked by `testMethod`. The same is done in the exception case with `testMethodWithException` invoking `testMethodWithException2`. The example in Figure 40 is the implementation of the performance test.

```
public class TestExceptionHandlingPerformance {
   long counter = 0;
   public TestExceptionHandlingPerformance() {
      super();
      counter = 0;
      }
   public static void main(java.lang.String[] args) {
      TestExceptionHandlingPerformance
               t = new TestExceptionHandlingPerformance();
      long start = 0;
      long end = 0;
      start = System.currentTimeMillis();
      for (long index = 0; index < 10000000; index++) {
         t.testMethod();
         }
      end = System.currentTimeMillis();
      System.out.println(
            "Method calls: " + (end-start) + " ms");
      start = System.currentTimeMillis();
      for (long index = 0; index < 10000000; index++) {
         try {t.testMethodWithException();}
         catch (TestException e) {};
         }
      end = System.currentTimeMillis();
      System.out.println(
            "Method calls with exception: " + (end-start) + " ms");
      }
   public long testMethod() {
      counter = counter + 1;
      testMethod2();
      return counter;
      }
   public long testMethod2() {
      return 0;
      }
   public long testMethodWithException() throws TestException {
      counter = counter + 1;
      testMethodWithException2();
      return counter;
      }
   public long testMethodWithException2() throws TestException {
      return 0;
      }
   }
```

**Figure 40.** Testing the performance of exception handling in Java

On a machine with the IBM JVM (JDK 1.2) the same code was executed 3 times and is noted as run 1 to run 3. The following results are returned – the time in milliseconds of invoking the methods 10 million times:

| Type of method invocation | Run 1 | Run 2 | Run 3 |
|---|---|---|---|
| Normal method invocation | 4060 | 4060 | 4070 |
| Exception wrapped method invocation | 4060 | 4170 | 4060 |

**Figure 41.** Results of running the code in Figure 40 three times in a row

The discrepancies in the numbers above are so small that they can be attributed to the VM overhead of maintenance and garbage collection between the different runs that is difficult to predict.

It is reassuring to see that simply wrapping statements in a try clause and adding catch clauses for handling the exceptions does not add any performance impediments. In the next test the impact of throwing an exception is measured by replacing the return statement with a throw statement as seen below.

```
public long testMethodWithException2() throws TestException {
   throw new TestException();
}
```

**Figure 42.** Modifying `testMethodWithException2` to throw an exception

In the throw statement an exception is thrown which is caught by the top level method call. The effect of embedding exceptions is also tested by throwing an exception after *n* levels deep in the call stack. In the next table test 1 shows the result of 1 level, test 2 shows the effect of 2 levels and so on until test 4. All the results are shown in milliseconds.

| Return method | Test 1 | Test 2 | Test 3 | Test 4 |
|---|---|---|---|---|
| Normal method return | 3570 | 4170 | 5000 | 5930 |
| Returning via throwing an exception | 67990 | 71620 | 74810 | 77720 |

**Figure 43.** Effect of nested levels of exceptions on performance

Plotting the results (see below) shows a linear relationship between the number of levels in the call stack and the time taken to return for both a normal return and returning via an exception. Note that the scale for the "Normal" graph is given on the left hand side vertical axis while the scale for the "Exception" graph is on the right hand side vertical axis.

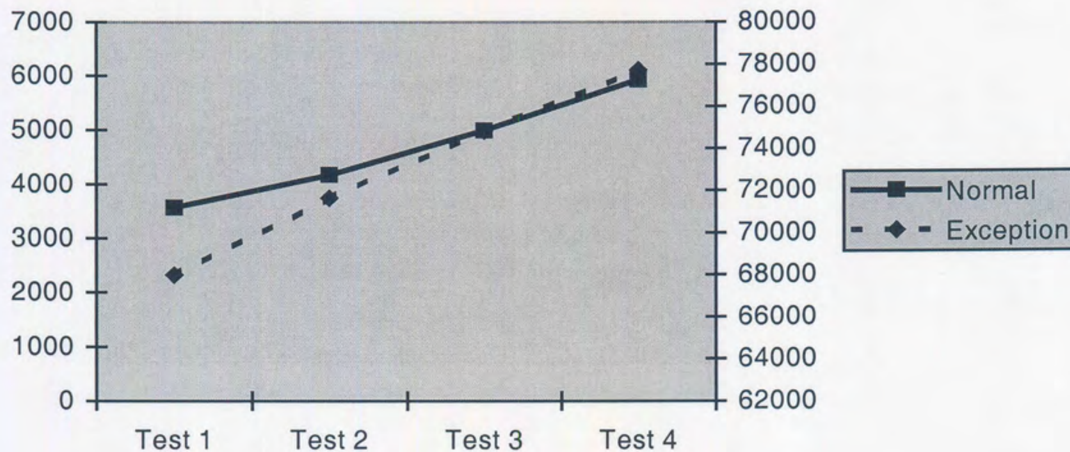


**Figure 44.** Graph showing the linear relationship between normal method exits and methods exiting via exceptions

From the above it is thus clear that a performance problem arises when throwing and handling an exception. This is due to locating the catch clause every time when throwing an exception and could pose an efficiency problem in certain circumstances. For more information about the performance issues with respect to exceptions in Java and optimising exception handling in Java see Ogasawara, et al (2001).

### 2.2.4. Reflection

With the introduction of Java 1.1 and later another API called the Reflection API was added to allow powerful constructs like instantiating classes at runtime or performing methods on objects without the virtual machine knowing at compile time what classes or methods will be needed.

Below is an example of instantiating an instance of `NewClass` which is not known at compile time. The string representation of it is printed on the standard output object.

```
java.lang.Class theClass = null;
Object theObject = null;

try { theClass = java.lang.Class.forName("NewClass"); }
catch (java.lang.ClassNotFoundException e) {};

try { theObject = (Object) theClass.newInstance(); }
catch (java.lang.InstantiationException e) {}
catch (java.lang.IllegalAccessException e) {};
System.out.println( theObject.toString() );
```

**Figure 45.** Using the Java Reflection API to load a class dynamically and invoke a method `toString()` on it

## 2.3. Summary

This chapter dealt with fundamental elements of the present study. It gave an overview of relevant Smalltalk and Java features. With this background, it is now possible to describe the approach that was followed in to translating Smalltalk code to Java code.

# 3. Translating from Smalltalk to Java

The purpose of this Chapter is to explain the mapping from Smalltalk source code to Java source code. It is divided into 5 subsections. The first 3 subsections are taken from a previously published article by Engelbrecht and Kourie (1998) with minor adaptations of paragraphs and diagrams. These sections deal with method selectors, classes and simulating a dynamically typed language (Smalltalk) in a statically type language (Java) respectively. Section 4 has been extensively rewritten. It deals with translating dynamic Smalltalk class methods into Java static methods and proposes alternative solutions. A few subsections have been added to section 4 to deal with different types of Smalltalk variables and the difference between Smalltalk and Java with respect to main methods. The rest of this chapter develops material that the article did not address. Specifically, section 5 discusses the different Smalltalk block closures and the problems associated with translating Smalltalk block closures into Java inner classes.

## 3.1. Method selectors

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 46 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` |

**Figure 46.** 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`.

**Rule 1**: A unary message is mapped directly to the equivalent Java method name without any arguments, i.e. `minimize` in Smalltalk maps directly to the invocation `minimize()` in Java.

**Rule 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()`, etc.

Methods such as `plus()` and `minus()` have to be inserted into a specially created Java class named `stj.Integer`. Each Smalltalk integer is mapped to an instance of this class. As a result the Smalltalk expression `1+2` will be mapped to the Java invocation `stj.Integer(1).plus(stj.Integer(2))`. Operations on other primitive Java types will be similar.

**Rule 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
```

translates to

```
moveTo_(aNewLocation)
```

and

```
replaceButton: button1 withNewButton: button2
```

translates to

```
replaceButton_withNewButton_(button1, button2)
```

If rule 2 or rule 3 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 '_' resulting in _class() or _class(argument).

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.

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 47 respectively:

| **Smalltalk code** | **Java translation** |
|---|---|
| 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 replace: b1 with: b2 | frame.replace_with_(b1,b2); |

**Figure 47.** Translations to Java method invocations

## 3.2. Classes and the class hierarchy

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 `SomeClass`. The translation algorithm should assure that this is done in a way that the structure of original Smalltalk program's class hierarchy is retained in the translated Java hierarchy.

Since Java classes are each translated to a file it is worth illustrating how the translated classes interact with the runtime classes. All the classes with an `stj` prefix belongs to the STJ runtime system and will be kept in the `stj` package. In contrast, translated classes will have no prefix and will thus be kept in the default package. Figure 48 illustrates the location of a translated class, `SomeClass.java`, with respect to the STJ runtime classes in the `stj` package.

```
SomeClass.java
...
metaclass\
   SomeClass.java
   ...
stj\
   Object.java
   Boolean.java
   Integer.java
   ...
stj\metaclass
      Object.java
      Boolean.java
      Integer.java
      ...
```

**Figure 48.** Illustrating location of translated Java class in the directory structure.

Note that the foregoing implies the existence of a package named `stj`. When declaring a class the prefix `stj` is not needed to qualify the class name. However, if a translated class inherits from or refers to a class in the `stj` package it needs to prefix the given class name with an `stj`. The references to metaclass\ and stj\metaclass in figure 48 will be further explained in section 3.4.

The structure of the resulting Java class hierarchy is depicted in Figure 49 below. Several advantages to be gained from this scheme will become apparent in later sections.

## 3.3. Dynamic types versus static types

One of the matters to confront when translating Smalltalk code to Java code 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.

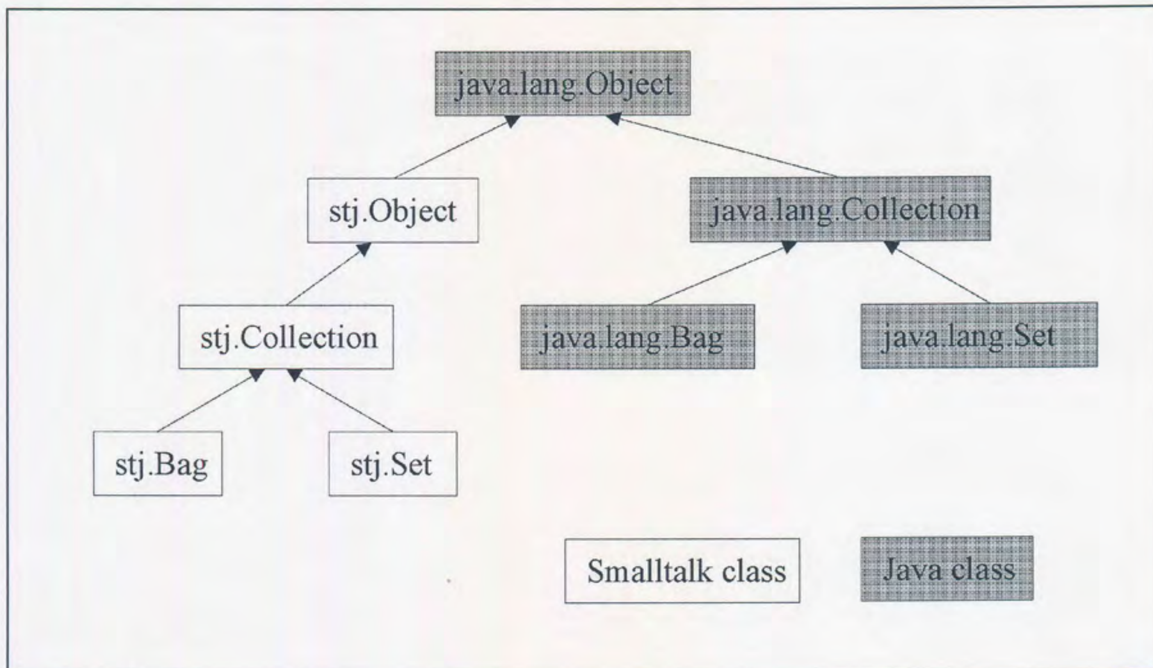


**Figure 49.** Proposed Java class hierarchy for Smalltalk translations

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.

### 3.3.1. Type inference

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 50.

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>>coll: isBag with: anInteger
   | aCollection |
   isBag
      ifTrue: [aCollection := Bag new]
      ifFalse: [aCollection := Set new].
   aCollection add: anInteger
   ^aCollection
```

**Figure 50.** Smalltalk `SampleClass` example

The translated Java code of the above Smalltalk method is given in Figure 51. 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. In the `stj.Object` class a variable with the name `__true` of type `stj.True` is 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 section 2.1.3.1.

```
public class SampleClass extends stj.Object
   {
   public stj.Object coll_with_(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;
      }
   }
```

**Figure 51.** Java `SampleClass` translation

In Figure 51 `aCollection` has been declared as type `stj.Object`. Since, by virtue of section 3.3, `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)`. The Java translation above is simplified to illustrate the issues of dynamic types in Smalltalk , thus the simplified translation of the `ifTrue:ifFalse:` Smalltalk keyword message and the naïve translation of the Smalltalk block. Section 3.5 will handle these translations in detail.

### 3.3.2. Reflection-based runtime method binding

The Java code in Figure 51 also contains an invocation to a method `Collection.add_(anInteger)` that has been directly translated from the Smalltalk code in Figure 50, 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 program 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 52.

```
public stj.Object perform_with_(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.__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);
   }
```

**Figure 52.** 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 Figure 52.

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)
```

**Figure 53.** Invocation of `add_` method on `aCollection` using reflection

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 via the call in Figure 53. 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 anticipated 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 confirm this. They indicate that it takes at least 100 times longer to invoke a method in this fashion, as compared with a normal method invocation. The code in Figure 54 tests the difference in performance between normal method invocations and reflection based method invocations:

```
public class TestPerformance extends stj.Object
  {
  public stj.Object testMethod(stj.Object arg1) {return arg1;}

  public static void main(java.lang.String[] args)
    {
    TestPerform t = new TestPerform();
    stj.Object arg1 = __true;
    long start = 0;
    long end = 0;
    start = System.currentTimeMillis();
    for (long index = 0; index < 100000; index++) {
      t.testMethod(arg1);
    }
    end = System.currentTimeMillis();
    System.out.println("Normal methods: " +(end-start)+ " ms");
    start = System.currentTimeMillis();
    for (long index = 0; index < 100000; index++) {
      t.perform_with_("testMethod", arg1);
    }
    end = System.currentTimeMillis();
    System.out.println("Reflection methods: " +(end-start)+ " ms");
    }
  }
```

**Figure 54.** Testing the performance of reflection based method invocations

| Type of method invocation | Result in milliseconds |
|---|---|
| **Normal method invocation** | 155 |
| **Reflection based method invocation** | 123985 |

**Figure 55.** Results of the performance test in Figure 54.

The table above shows the results and it can be clearly seen that the reflection based invocation is about 800 times slower than a normal method invocation. Until the JVM vendors provide optimised versions of the above methods used in the reflection API, this approach for ensuring run-time method binding does not seem practical. Nevertheless, the approach may be legitimately be used to translate `perform:with:` messages that occur in Smalltalk code.

### 3.3.3. 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 implementing the translator. Two categories of methods are placed into the Java system:

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.

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 rule 1 above in 3.1.) 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 that the complete Smalltalk program to be translated needs to be present during translation.

Thus, referring to the `SampleClass` example in Figure 50, 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`, which in turn invokes the `doesNotUnderstand_` method on `Object` as illustrated in Figure 56 below.

```
public class stj.Object
   {
   public stj.Object doesNotUnderstand_(java.lang.String messageName)
      {
      System.out.println("");
      System.out.println("Object>>doesNotUnderstand: " + messageName);
      System.exit(1);
      return this;
      }
   public stj.Object add_(stj.Object arg1)
      {
      this.doesNotUnderstand_( "add:" );
      return this;
      }
   }
```

**Figure 56.** 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 (i.e. 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 important benefit of this approach is that the speed of the resulting code executes at the same speed of normal Java code with types in the variable declarations. The table in Figure 57 illustrates the performance of dynamically typed methods and statically typed methods in milliseconds.

| **Type of method** | **Run 1** | **Run 2** | **Run 3** |
|---|---|---|---|
| **Dynamically typed method** | 5209 | 5310 | 5235 |
| **Statically typed method** | 5249 | 5164 | 5262 |

**Figure 57.** Performance of dynamically typed methods and statically typed methods

## 3.4. Simulating Smalltalk classes

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 58 below, which is a superclass of the class `BMW`.

```
Object subclass: #Vehicle
   instanceVariableNames: ''
   classVariableNames: ''.

Vehicle class>>newInitializedObject
   | instance |
   instance := self new.
   instance initialize.
   ^instance

Vehicle>>initialize
   Transcript cr; show: 'Vehicle>>initialize called'.

Vehicle subclass: #BMW
   instanceVariableNames: ''
   classVariableNames: ''.

BMW>>initialize
   Transcript cr; show: 'BMW>>initialize called'.
```

**Figure 58.** 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 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.

### 3.4.1. Static Java class methods

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

1 a Java class called Vehicle for the Smalltalk class called Vehicle;
2 a Java subclass of Vehicle called BMW for the Smalltalk class called BMW;
3 in the Java class called Vehicle, a Java class method called newInitializedObject();
4 in the Java class called Vehicle, a Java instance method called initialize();
5 in the Java class called BMW, another Java instance method called initialize();
6 the invocation: Vehicle.newInitializedObject() for any Smalltalk message Vehicle newInitializedObject;
7 the invocation: BMW.newInitializedObject() for any Smalltalk message BMW newInitializedObject.

Figure 59 shows these translations, where the Java class methods are declared with the required static modifier. A further two static methods each called __class(), are provided in Vehicle and BMW respectively. The two versions of __class() rely on the reflection API method forName() to return the class in which they are respectively declared (either Vehicle or BMW). In newInitializedObject a class to __class() is made, the intention being to invoke the __class() method corresponding to the class that qualifies newInitializedObject in a call. Thus Vehicle.newInitializedObject should return Vehicle and BMW.newInitializedObject should return BMW. The reflection API method newInstance() is invoked in the newInitalizedObject() method to generate an instance

of the returned class. In the next line the `initialize()` method of this generated instance is then invoked.

```
package stj;
public class Vehicle extends stj.Object {
   public static stj.Object newInitializedObject()
      {
      stj.Object  instance = null;
      instance = (stj.Object)__class().newInstance();
      instance.initialize();
      return instance;
      }
   public static java.lang.Class __class()
      {
      java.lang.Class thisClass = null;
      try { thisClass = java.lang.Class.forName("Vehicle"); }
      catch (ClassNotFoundException e)
         {System.out.println("Error: Class not found"); }
      }
   public stj.Object initialize()
      {
      System.out.println("Vehicle>>initialize called");
      return this;
      }
   }

public class BMW extends Vehicle
   {
   public static java.lang.Class __class()
      {
      java.lang.Class thisClass = null;
      try { thisClass = java.lang.Class.forName("BMW"); }
      catch (ClassNotFoundException e)
         {System.out.println("Error: Class not found"); }
      }
   public stj.Object initialize_()
      {
      System.out.println("BMW>>initialize called");
      return this;
      }
   }
```

**Figure 59.** Static Java `Vehicle` and `BMW` class

However, if the following code below in Figure 60 is executed, it will be found that this Java implementation does not behave as the Smalltalk counterpart.

```
| car |
stj.Object car = null;
car = Vehicle.newInitializedObject();
car = BMW.newInitializedObject();
```

**Figure 60.** Initialising `Vehicle` and `BMW` instances

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

### 3.4.2. 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 metaclasses). Smalltalk has the following (Goldberg and Robinson (1989, p269-p271)):

1. 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.
2. Every class is a subclass of class Object. Object itself has no superclass.
3. 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.
4. 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`.
5. Metaclasses also being objects, are instances of a class called `Metaclass`.

The structure is depicted in Figure 61, and includes the classes and metaclasses for the `Vehicle` and `BMW` classes as described previously.

The solid lines indicate a *subclass of* relationship in the class hierarchy, be it on the object level or the class level. The dotted lines indicate the *instance of* relationship between objects and their classes. Note carefully that `stj.metaclass.Object` is indeed a subclass of `stj.Class`, in accordance with (4).

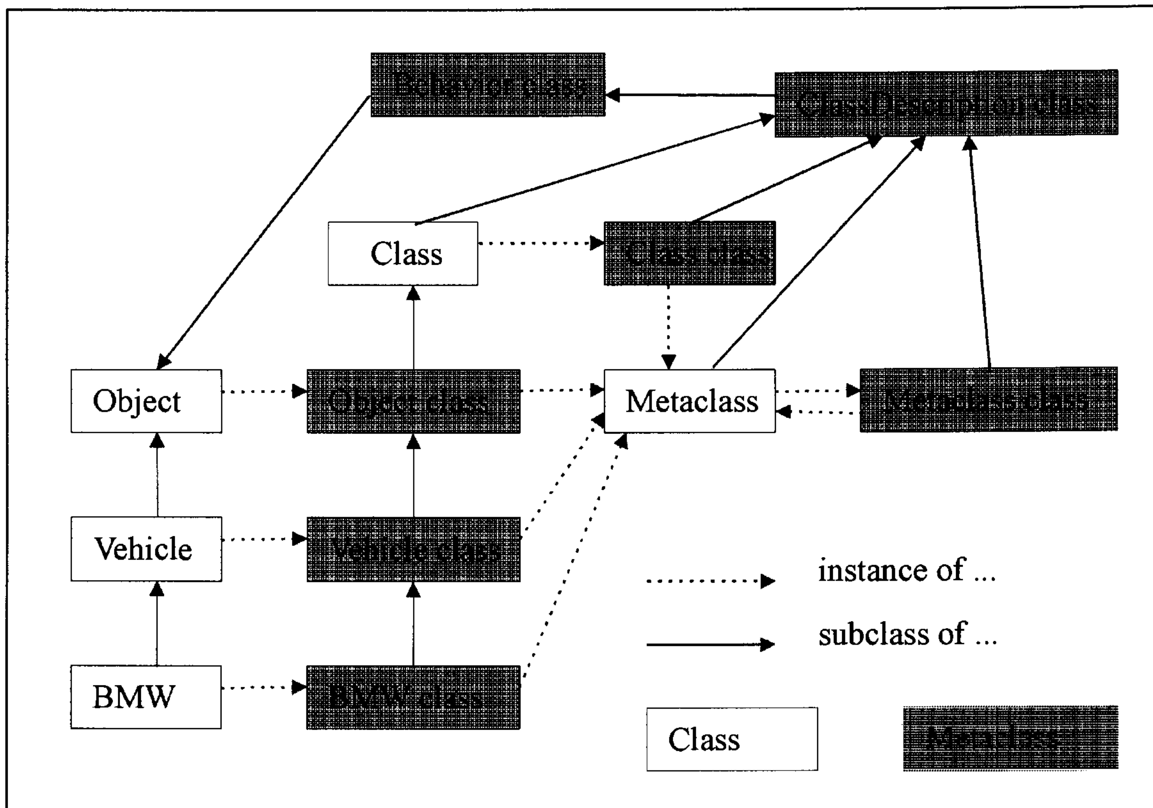


**Figure 61.** Smalltalk class and metaclass relationships

For the purpose of translating standard Smalltalk code the translated Java code will not have to simulate the Smalltalk reflection features exactly. This allows the implementation of a much simpler hierarchy that is illustrated in Figure 62.

To provide for dynamic binding of class methods, it will be helpful to mirror this Smalltalk class hierarchy structure in the translated Java system, prepending each Java class by "`stj.`" as before. However, there will be no need to define the Java class `stj.Metaclass`. An appropriate convention might have been to qualify all the metaclass subclasses of `stj.Class` by `stj.class`. Unfortunately the string `class` is a reserved keyword in Java and another scheme has to be used. A solution will be to store all the metaclasses in a separate package, calling it `stj.metaclass`. Thus, the class in package `stj.metaclass` named `Object` will be referenced as `stj.metaclass.Object`, the class named `Class` will be referenced as `stj.metaclass.Class`, etc.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

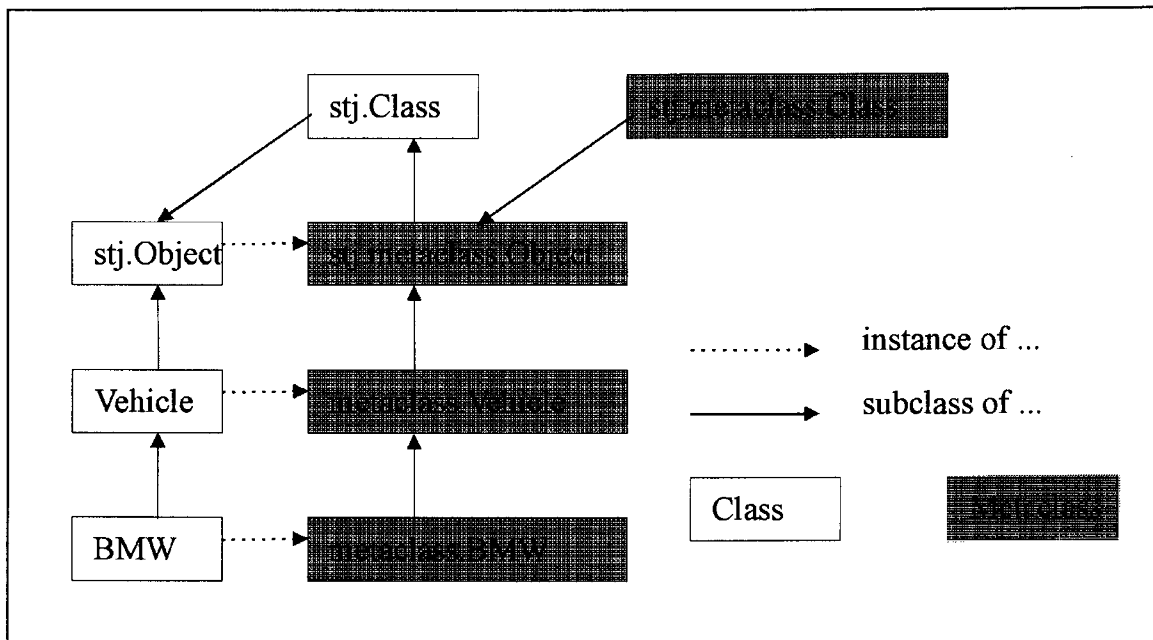


**Figure 62.** Simplified Java translation of the class and metaclass hierarchy

Another method that is needed is `__new()` to create a new instance of a specified class. This method is defined in `stj.Object` and overridden in `stj.metaclass.Object` since `stj.metaclass.Object` inherits from `stj.Class` which in turn inherits from `stj.Object`. As discussed previously the `stj.metaclass.Object` class inherits from `stj.Class` and since all metaclasses inherit from `stj.metaclass.Object` they will all share the behaviour of the `__new()` method. Figure 63 below shows the implementation of `__new()` in `stj.Object` and `stj.Class` as it also provides the class definition for `stj.Class`, `stj.metaclass.Object` and `stj.metaclass.Class`.

```
package stj;
public class Object
   {
   // other methods in stj.Object …
   public stj.Object __new()
      {
      this.doesNotUnderstand_("new");
      return __nil;
      }
   }

package stj;
public class Class extends stj.Object {}

package stj.metaclass;
public class Object extends stj.Class
  {
  public stj.Object __new()
    {
    java.lang.Class theClass = null;
    stj.Object      theObject = null;

    try {theClass = java.lang.Class.forName(this.__className());}
    catch (java.lang.ClassNotFoundException e) {};

    try {theObject = (stj.Object)theClass.newInstance();}
    catch (java.lang.InstantiationException e) {}
    catch (java.lang.IllegalAccessException e) {};

    return(theObject);
    }
  }

package stj.metaclass;
public class Class extends stj.metaclass.Object {}
```

**Figure 63.** Implementation of __new with corresponding class hierarchy

The Java code is designed to simulate the metaclass of a class that has a class method. This design allows for dynamically binding a class method that is invoked at runtime. The principle is illustrated in Figure 64 below, in terms of the previous `Vehicle` example.

Two classes, `metaclass.Vehicle` and `metaclass.BMW` have been defined with *instance* methods (which are thus dynamic) replacing the static methods of the previously defined classes `Vehicle` and `BMW` respectively. In the case of `stj.metaclass.Vehicle` the relevant instance

methods are `newInitializedObject()` and `__new()`. Note that the classes `Vehicle` and `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 `__new()` method in `stj.metaclass.Object` expects all metaclass classes to implement a method called `__className()`. This will return a Java string being the class name that the metaclass is to represent.

The result is that a Smalltalk class (such as `Vehicle`) that has a class method (such as `newInitializedObject`) is always simulated by a composite of two classes in Java. The first Java class named `Vehicle` deals with the Smalltalk *instance methods* on the Smalltalk vehicle class as previously discussed. The second Java class named `metaclass.Vehicle` deals with the translated Smalltalk *class methods* on the Vehicle class in the form of Java instance methods on `metaclass.Vehicle`. An instance of the Java class `metaclass.Vehicle` 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 (3) above.

```
package metaclass;
import stj.*;
public class Vehicle extends stj.metaclass.Object
    {
    public static stj.Object __class = new metaclass.Vehicle();
    public stj.Object newInitializedObject()
        {
        stj.Object instance = null;
        instance = this.__new();
        instance.initialize();
        return(instance);
        }
    public java.lang.String __className() {return("Vehicle");}
    }
import stj.*;
public class Vehicle extends stj.Object
    {
    stj.Object
    public stj.Object initialize()
        {
        STJTranscript.cr().show_((
            new stj.String("Vehicle>>initialize  called")));
        return(this);
        }
    }
package metaclass;
import stj.*;
public class BMW extends metaclass.Vehicle
    {
    public static stj.Object __class = new metaclass.BMW();
    public java.lang.String __className() {return("BMW");}
    }
import stj.*;
public class BMW extends Vehicle
    {
    public stj.Object initialize()
        {
        STJTranscript.cr().show_((
            new stj.String("BMW>>initialize called")));
        return(this);
        }
    }
```

**Figure 64.** Dynamic Java `Vehicle` and `BMW` class

It is now possible to invoke the class method `newInitializedObject()` on each class and the correct instance is initialized as illustrated below.

```
stj.Object  car = null;
car = metaclass.Vehicle.__class.newInitializedObject();
car = metaclass.BMW.__class.newInitializedObject();
```

**Figure 65.** Initialising a `Vehicle` and `BMW` instance

Functionally, the first line of the code achieves the same as before. The reference `metaclass.Vehicle.__class` refers to an instance of the `metaclass.Vehicle` (thus, the equivalent of a Smalltalk class). Thus `metaclass.Vehicle.__class.newInitializedObject()` invokes the `newInitializedObject()` method defined in `metaclass.Vehicle`. This method creates an instance of `Vehicle` and assigns it to the variable `car`. It also prints out "`Vehicle>>initialize called`". In a similar way the second line of code prints out "`BMW initialize called`" and assigns it to the variable `car`. However, the call to the class method `newInitializedObject()` is now bound at runtime, resulting in the right methods being called. Specifically the right implementations of `__className()` and `initialize()` is being invoked respectively.

Note that `metaclass.Vehicle.__class` relates to an aspect of the Java metaclass translation where a metaclass instance (class object) is created and class methods are implemented by implementing instance methods on the metaclass instance. This is the same way that Smalltalk implements class methods. The only difference is that in the Java translation the metaclass instance is resolved by referring to a static variable in the Java metaclass, namely `__class`, whereas in Smalltalk the parser resolves the metaclass instance by referring to a global dictionary in the system.

A static variable called `__class` of the class `metaclass.Vehicle` is thus declared. This variable is instantiated to reference an instance of `metaclass.Vehicle` 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. To arbitrarily create multiple instances of `metaclass.Vehicle` would violate the Smalltalk paradigm. The same applies in the case of `metaclass.BMW` instance.

### 3.4.3. Smalltalk class variables

The translation of Smalltalk class variables is analogous to the translation of Smalltalk class methods. The same pattern is applied where properties on a Smalltalk *class* are mapped onto properties on the Smalltalk class' metaclass *instance*. Thus, Smalltalk class variables can be represented as instance variables in the translated Java metaclass object. The Smalltalk class definition for `Vehicle` is shown in Figure 66 with a class variable `DefaultColour`. It is the convention in Smalltalk to begin all class variables with a capital to help distinguish class variables from instance variables when reading the source code. Note that a class variable can be accessed from instance methods and class methods.

```
Object subclass: #Vehicle
   instanceVariableNames: ''
   classVariableNames: 'DefaultColour'
```

**Figure 66.** Smalltalk class definition with a class variable

Using the same translation rules discussed in section 3.4.2 (see Figure 64) the translated Java class definition is generated in Figure 67. Only the Java metaclass translation is shown, since the Java class translation does not change from Figure 64.

```
public class metaclass.Vehicle extends stj.metaclass.Object
   {
   public stj.Object DefaultColour = null;
   public static stj.Object __class = new metaclass.Vehicle();
   public stj.Object newInitializedObject(){…}  -defined as before
   public java.lang.String __className() {return("Vehicle");}
   }
```

**Figure 67.** Java translation of Smalltalk class definition with a class variable

The `DefaultColour` variable definition is available to all instances of `Vehicle` and its subclasses. This results in the instances of the `BMW` class (subclass of `Vehicle`) to being able to access the `DefaultColour` variable as well. To initialise the `DefaultColour` variable (assuming `DefaultColour` is a string which needs to be initialized) the code in Figure 69 can be written in all instance methods of `Vehicle` and `BMW`. It is necessary to typecast the `__class` variable to the class in which the class variable is declared (in this case `metaclass.Vehicle`) since `stj.metaclass.Object` does not have any reference to the `DefaultColour` variable.

```
((metaclass.Vehicle)(metaclass.Vehicle.__class)).DefaultColour = new
stj.String("Black");
```

**Figure 68.** Initialising the class variable

```
((metaclass.Vehicle)(metaclass.Vehicle.__class)).DefaultColour;
```

**Figure 69.** Accessing the class variable

In Figure 70 the Smalltalk classes `Vehicle` and `BMW` are defined. Note that the class variable `DefaultColour` in `Vehicle` has been defined. In addition an instance variable `colour` is added in `Vehicle` for a vehicle to keep track of its colour. A class method `initializeClassVars` is defined in both `Vehicle` and `BMW` to initialise the colour instance variable to the default value in all their respective instances. However, the default for `Vehicle` is "Black" and the default for `BMW` is "Red".

```
Object subclass: #Vehicle
   instanceVariableNames: 'colour'
   classVariableNames: 'DefaultColour '

Vehicle class>>initializeClassVars
   DefaultColour := 'Black'.

Vehicle class>>newInitializedObject
   | instance |
   instance := self new.
   instance initialize.
   ^instance.! !

Vehicle>>colour
   ^colour

Vehicle>>initialize
   colour := DefaultColour.
   STJTranscript cr; show: 'Vehicle>>initialize called'.

Vehicle subclass: #BMW
   instanceVariableNames: ''
   classVariableNames: ''

BMW class>>initializeClassVars
   DefaultColour := 'Red'

BMW>>initialize
   colour := DefaultColour.
   STJTranscript cr; show: 'BMW>>initialize called'.
```

**Figure 70.** `Vehicle` and `BMW` definitions for an example of using a class variable

In Figure 71 below the `Vehicle` class is initialised and the string '`Black`' is assigned to the `DefaultColour` class variable. A `Vehicle` instance (`car1`) is created and initialised. When printing the colour of `car1` '`Black`' is printed. In the next section the `BMW` class is initialised and the colour '`Red`' assigned to the `DefaultColour` class variable. A `BMW` instance (`car2`) is then created and subsequently the string '`Red`' is printed. In the last section another `Vehicle` instance (`car3`) is created and due to the nature of class variables being shared between the class where it is defined and all the subclasses the `DefaultColour` class variable now has the value '`Red`' and `car3` will return '`Red`' when asked for its colour.

```
| car1 car2 car3 |
Vehicle initializeClassVars.
car1 := Vehicle newInitializedObject.
STJTranscript cr; show: car1 colour.
BMW initializeClassVars.
car2 := BMW newInitializedObject.
STJTranscript cr; show: car2 colour.
car3 := Vehicle newInitializedObject.
STJTranscript cr; show: car3 colour.
```

**Figure 71.** Illustrating the usage of a class variable

Executing the code in Figure 71 will result in the following output to the Transcript

```
Vehicle>>initialize called
Black
BMW>>initialize called
Red
Vehicle>>initialize called
Red
```

**Figure 72.** Output of Figure 71

The translation of Figure 70 is shown below in Figure 73. The `Vehicle` and `BMW` metaclasses are defined as well as the `Vehicle` and `BMW` classes.

```
package metaclass;
import stj.*;
public class Vehicle extends stj.metaclass.Object
   {
   public static stj.Object __class = new metaclass.Vehicle();
   public stj.Object DefaultColour;

   public stj.Object initializeClassVars()
      {
      ((metaclass.Vehicle)(metaclass.Vehicle.__class)).DefaultColour =
         (new stj.String("Black"));
      return(this);
      }
   public stj.Object newInitializedObject()
      {
      stj.Object instance = null;
      instance = this.__new();
      instance.initialize();
      return(instance);
```

```
      }
   public java.lang.String __className() {return("Vehicle");}
   }

package metaclass;
import stj.*;

import stj.*;

public class Vehicle extends stj.Object
   {
   stj.Object colour;

   public stj.Object initialize()
      {
      colour = ((metaclass.Vehicle)
         (metaclass.Vehicle.__class)).DefaultColour;
      STJTranscript.cr().show_(
         (new stj.String("Vehicle>>initialize called")));
      return(this);
      }
   public stj.Object colour()
      {
      return(colour);
      }
   public stj.Object __class() {return(metaclass.Vehicle.__class);}
   }

package metaclass;
import stj.*;
public class BMW extends metaclass.Vehicle
   {
   public static stj.Object __class = new metaclass.BMW();
   public stj.Object initializeClassVars()
      {
      ((metaclass.Vehicle)(metaclass.Vehicle.__class)).DefaultColour =
         (new stj.String("Red"));
      return(this);
      }
   public java.lang.String __className() {return("BMW");}
   }

import stj.*;
public class BMW extends Vehicle
   {
   public stj.Object initialize()
```

```
      {
      colour = ((metaclass.Vehicle)
         (metaclass.Vehicle.__class)).DefaultColour;
      STJTranscript.cr().show_(
         (new stj.String("BMW>>initialize called")));
      return(this);
      }
   public stj.Object __class() {return(metaclass.BMW.__class);}
   }
```

**Figure 73.** Translation of `Vehicle` and `BMW` classes in Figure 70

The code below is the Java equivalent of Figure 71 to illustrate the usage of a class variable.

```
stj.Object car1 = null;
stj.Object car2 = null;
stj.Object car3 = null;

metaclass.Vehicle.__class.initializeClassVars();
car1 = metaclass.Vehicle.__class.newInitializedObject();
STJTranscript.cr().show_(car1.colour());
metaclass.BMW.__class.initializeClassVars();
car2 = metaclass.BMW.__class.newInitializedObject();
STJTranscript.cr().show_(car2.colour());
car3 = metaclass.Vehicle.__class.newInitializedObject();
STJTranscript.cr().show_(car3.colour());
```

**Figure 74.** Illustrating the usage of a class variable in Java

When executing the code in Figure 74 the same output will be generated as in Figure 72.

Depending on the requirements of the program this may be the intended behaviour where `BMW` instances share the `DefaultColour` variable. If the requirement is indeed that `BMW` instances should have its own `DefaultColour variable` when creating new instances of `BMW` can be achieved in two ways:

The solution is by declaring another class variable `DefaultColour` in the class definition of `BMW` as in Figure 75.

```
Vehicle subclass: #BMW
   instanceVariableNames: ''
   classVariableNames: 'DefaultColour'
```

**Figure 75.** Adding class variable `DefaultColour` to the `BMW` class

If there are many subclasses of Vehicle, it may become tedious to declare class variables in all the subclasses of Vehicle. Smalltalk has another solution, which is a class instance variable. Section 3.4.4 illustrates the use of class instance variables.

### 3.4.4. Smalltalk class instance variables

Smalltalk has an interesting type of class variable, called the class instance variable. This variable is declared in a class in the same way as a class variable and all the subclasses will inherit this variable with the distinction that every subclass of the class will have its own copy of the class variable. The example in Figure 76 shows the declaration of the class instance variable `manualOrAutomatic` in the super class `Vehicle`. The two subclasses `BMW` and `Toyota` do not declare anything extra, but inherit implicitly their own copy of `manualOrAutomatic`. The net effect is that when a method, (either an instance or a class method), of `BMW` modifies `manualOrAutomatic` it has no impact on the contents of the `manualOrAutomatic` variable of the `Toyota` class, or the `manualOrAutomatic` variable of the `Vehicle` class. Another useful fact is that the class instance variable can be modified by both class methods and instance methods.

The example below in Figure 76 shows the modified class definition of `Vehicle` as well as extra methods that needs to be modified or added to the example in Figure 70.

```
Object subclass: #Vehicle
   instanceVariableNames: 'colour transmission'
   classVariableNames: 'DefaultColour '
   classInstanceVariableNames: 'manualOrAutomatic

Vehicle class>>manualOrAutomatic
   ^manualOrAutomatic

Vehicle class>>initializeClassInstanceVars
   manualOrAutomatic := 'Manual'

Vehicle>>initialize
   colour := DefaultColour.
   transmission := self class manualOrAutomatic.
   STJTranscript cr; show: 'Vehicle>>initialize called'.

Vehicle>>transmission
   ^transmission

Vehicle subclass: #BMW
   instanceVariableNames: ''
   classVariableNames: ''

BMW class>>initializeClassInstanceVars
```

```
   manualOrAutomatic := 'Automatic'

BMW>>initialize
   colour := DefaultColour.
   transmission := self class manualOrAutomatic.
   STJTranscript cr; show: 'BMW>>initialize called'.

Vehicle subclass: #Toyota
   instanceVariableNames: ''
   classVariableNames: ''

Toyota class>>initializeClassInstanceVars
   manualOrAutomatic := 'Manual'

Toyota>>initialize
   colour := DefaultColour.
   transmission := self class manualOrAutomatic.
   STJTranscript cr; show: 'Toyota>>initialize called'.
```

**Figure 76.** Extra methods to add to example in Figure 70 for the use of a class instance variable

After the enhancements are made for Vehicle, BMW and Toyota to support class instance variables the code in Figure 77 can be executed.

```
| car1 car2 car3 car4 |
Vehicle initializeClassInstanceVars.
car1 := Vehicle newInitializedObject.
STJTranscript cr; show: car1 transmission.
BMW initializeClassInstanceVars.
car2 := BMW newInitializedObject.
STJTranscript cr; show: car2 transmission.
car3 := Vehicle newInitializedObject.
STJTranscript cr; show: car3 transmission.
Toyota initializeClassInstanceVars.
car4 := Toyota newInitializedObject.
STJTranscript cr; show: car4 transmission.
```

**Figure 77.** Executing the code for showing the transmissions

The result of executing the code in Figure 77 is shown in Figure 78 below. Vehicle sets it's transmission type, then BMW sets it's own transmission type after which Vehicle's transmission type is still intact.

```
Vehicle>>initialize called
Manual
BMW>>initialize called
Automatic
Vehicle>>initialize called
Manual
Toyota>>initialize called
Manual
```

**Figure 78.** Output of Figure 77

Since a class instance variable has the same relationship to a metaclass as an instance variable to a class it is appropriate to have a translation rule whereby all class instance variables are translated as instance variables that belong to the metaclass. This allows all the subclasses of the metaclass where the class instance variable is defined to inherit that class instance variable as well. The translation of the methods in Figure 76 is shown below in Figure 79. In response to invocations translated from Figure 77, the output of Figure 78 is obtained.

```
package metaclass;
import stj.*;
public class Vehicle extends stj.metaclass.Object
   {
   public static stj.Object __class = new metaclass.Vehicle();
   public stj.Object manualOrAutomatic;
   public stj.Object DefaultColour;

   public stj.Object initializeClassVars()
      {
      ((metaclass.Vehicle)(metaclass.Vehicle.__class)).DefaultColour =
         (new stj.String("Black"));
      return(this);
      }
   public stj.Object manualOrAutomatic()
      {
      return(manualOrAutomatic);
      }
   public stj.Object initializeClassInstanceVars()
      {
      manualOrAutomatic = (new stj.String("Manual"));
      return(this);
      }
   public stj.Object newInitializedObject()
      {
      stj.Object instance = null;
```

```
        instance = this.__new();
        instance.initialize();
        return(instance);
        }
    public java.lang.String __className() {return("Vehicle");}
    }

import stj.*;
public class Vehicle extends stj.Object
    {
    stj.Object colour;
    stj.Object transmission;

    public stj.Object initialize()
        {
        colour = ((metaclass.Vehicle)
            (metaclass.Vehicle.__class)).DefaultColour;
        transmission = this.__class().manualOrAutomatic();
        STJTranscript.cr().show_(
            (new stj.String("Vehicle>>initialize called")));
        return(this);
        }
    public stj.Object transmission()
        {
        return(transmission);
        }
    public stj.Object colour()
        {
        return(colour);
        }
    public stj.Object __class() {return(metaclass.Vehicle.__class);}
    }

package metaclass;
import stj.*;
public class BMW extends metaclass.Vehicle
    {
    public static stj.Object __class = new metaclass.BMW();

    public stj.Object initializeClassInstanceVars()
        {
        manualOrAutomatic = (new stj.String("Automatic"));
        return(this);
        }
    public stj.Object initializeClassVars()
        {
```

```
      ((metaclass.Vehicle)
         (metaclass.Vehicle.__class)).DefaultColour = (new
stj.String("Red"));
      return(this);
      }
   public java.lang.String __className() {return("BMW");}
   }

import stj.*;
public class BMW extends Vehicle
   {
   public stj.Object initialize()
      {
      colour = ((metaclass.Vehicle)
         (metaclass.Vehicle.__class)).DefaultColour;
      transmission = metaclass.BMW.__class.manualOrAutomatic();
      STJTranscript.cr().show_(
         (new stj.String("BMW>>initialize called")));
      return(this);
      }
   public stj.Object __class() {return(metaclass.BMW.__class);}
   }

package metaclass;
import stj.*;
public class Toyota extends metaclass.Vehicle
   {
   public static stj.Object __class = new metaclass.Toyota();
   public stj.Object initializeClassInstanceVars()
      {
      manualOrAutomatic = (new stj.String("Manual"));
      return(this);
      }
   public java.lang.String __className() {return("Toyota");}
   }

import stj.*;
public class Toyota extends Vehicle
   {
   public stj.Object initialize()
      {
      colour = ((metaclass.Vehicle)
         (metaclass.Vehicle.__class)).DefaultColour;
      transmission = metaclass.Toyota.__class.manualOrAutomatic();
      STJTranscript.cr().show_(
         (new stj.String("Toyota>>initialize called")));
```

```
        return(this);
        }
    public stj.Object __class() {return(metaclass.Toyota.__class);}
    }
```

**Figure 79.** Translated Java code of Figure 76

### 3.4.5. Smalltalk global variables

Declaring a global variable in Smalltalk is straightforward and involves associating a value with a key in the Smalltalk dictionary. To create an empty list of colours and assign it to `VehicleColours` the following code will be used:

```
Smalltalk at: #VehicleColours put: Set new.
```

Accessing a global variable in Smalltalk can be done in one of two ways: either through the global Smalltalk dictionary or by referring to the name of the global variable directly. To access a global variable named `VehicleColours` via the first method and request its size, the following Smalltalk code will be used:

```
(Smalltalk at: #VehicleColours) size
```

The other way is to refer to the variable directly as in:

```
VehicleColours size
```

The method for translating Smalltalk global variables to Java is to use an object to hold on to all the global variables and access them in the same fashion as the first method above. The following variable declaration of Smalltalk is added to the `stj.Object` class definition.

```
public static stj.Object Smalltalk = new stj.Dictionary[9]();
```

It is now possible to refer to the Smalltalk dictionary in all the methods throughout the system in the following manner:

```
(Smalltalk.at_('VehicleColours')).size();
```

[9] `stj.Dictionary` will be an implementation of a Dictionary supplied by the STJ runtime classes.

### 3.4.6. Java main methods

To start a routine or program in Smalltalk it is customary to define a class method on a class that will create an instance of an object and send an initial message to it, as illustrated below in Figure 80. A method called `run` is defined in the class, say `Test`, and to start executing the `run` message is sent to the class -- i.e. `Test run`. This creates an instance of `Test` using the `self new` instruction, then sends messages to that instance, eventually printing out the result of the computation.

```
Test class>> run
   | instance function |
   instance := self new.
   function := instance getBlock: 1.
   Transcript cr; show: ((function value: 2) printString).
```

**Figure 80.** Smalltalk class method - `run`

In Java the same approach is followed as in C/C++ by having a main method which is the default entry point into a program. The translated Java source of Figure 80 is shown in Figure 81.

```
public Test extends stj.Object {
   public static void main(java.lang.String[] args) {
      stj.Object instance = null;
      stj.Object function = null;

      instance = metaclass.Test.__class.__new();
      function = instance.getBlock_((new stj.Integer(1)));
      Transcript.cr().show_(
         function.value_((new stj.Integer(2))).printString());
   };
   … rest of the class definition
}
```

**Figure 81.** Java static method - `main`

A decision was made to have a convention of the translator searching for a method with the name `javamain` on the class side of a Smalltalk class and if encountered it would translate that method into the Java `static main` method. Thus by renaming the run method above in Figure 80 to `javamain` the Smalltalk source will be translated in the correct fashion.

## 3.5. Translating BlockContext objects

Smalltalk defines a block of code as an object. It is possible to assign this object (block of code) to a variable as well, to pass it around in the system and to ask it to execute (i.e. to evaluate the block of code). When sent the `value` message, the block will execute in the enclosing context in which it was defined. When it executes, it therefore has to hold a reference to its enclosing context. A simple example of a statement is found in Figure 82.

```
1 + 2
```

**Figure 82.** Simple Smalltalk statement

When inspecting the results of evaluating the statement[10] in figure 3.23 the following is returned: an object of type `SmallInteger` with the value `3`.

To create a block out of the statement above is very easy – simply enclose the statement in square brackets as in Figure 83.

```
[1+2]
```

**Figure 83.** Simple Smalltalk block

To evaluate the block in Figure 83– just send it the `value` message as shown in Figure 84.

```
[1+2] value
```

**Figure 84.** Evaluating a block

When the result of Figure 84 is inspected, it returns the same result as the statement 1 + 2, namely an object of type `SmallInteger` with value `3`.

The result of Figure 83 is an object of type `BlockContext` with various instance variables. In Squeak Smalltalk the following instance variables are defined in a `BlockContext`:

---

[10] Note that most Smalltalk development environments allow the user to select pieces of Smalltalk code, to request that the code executes (i.e. evaluates) and to then inspect the results of the code's evaluation.

sender – not used in `BlockContext`

`pc` – program counter for keeping track of the VM's program counter

`stackp` – stack pointer

`nargs` – the number of arguments being passed to the block

`startpc` – starting program counter

`home` – holds on to the enclosing context and scope of where the block was defined

Of interest are the `nargs` and `home` instance variables. The `nargs` variable keeps track of how many arguments the block must receive as arguments for execution. In this respect, the block can act as a function in more traditional languages such as Pascal and C.

The discussion that follows below explains how the Java translation of Smalltalk blocks has been achieved. The discussion begins by giving a translation required for simple blocks (3.5.1). It then shows to translate a block that refers to variables outside the block (3.5.2) or that has arguments that are passed to the block (3.5.3). It then proposes a way of translating that ensures that the Java translation holds on to the enclosing context and scope of where the block was defined in the same way that Smalltalk does (3.5.4). Finally, it proposes a way of translating blocks that have multiple exit points (3.5.5). These different translation issues are illustrated in a step by step fashion from one subsection to the next, in each case requiring an enhancement of the approach proposed in the previous section.

### 3.5.1. Simple block

In the simplest case of a Smalltalk block it will execute a few statements and return a result, the result of the last statement executed. In this respect, the Smalltalk block can be treated like a function. An example of such a block is the following:

```
| simpleFunction |
  simpleFunction := [2 + 3].
  simpleFunction value
```

**Figure 85.** Simple Smalltalk block

Inspecting the result of Figure 85 will yield the `SmallInteger` with value 5. To translate Figure 85 it is necessary to introduce new translation rules for the targeted Java source.

Smalltalk has a class `BlockContext` and it thus follows that the translation should have an abstract superclass `stj.BlockContext` for all the behaviour common to the translated Java source. The implementation for the first iteration of `stj.BlockContext` will be as follows:

```
public class stj.BlockContext extends stj.Object
   {
   public stj.BlockContext() {}
   public stj.Object value()
      {
      return stj.Object.__nil;
      }
   }
```

**Figure 86.** A BlockContext definition in Java

An empty constructor method is needed for creating an instance of `stj.BlockContext` as well as a `value()` method that, in the present case, simply returns the translated nil Smalltalk object.

In seeking to translate the Smalltalk code in Figure 85, the contents of the `value()` method has to be translated differently. It has to be generated by, *inter alia*, applying the translation rules defined in sections 3.1 to 3.4. The resulting translation is shown in the Java excerpt in Figure 87.

```
stj.Object simpleFunction = null;
simpleFunction = new stj.BlockContext()
   {
   public stj.Object value()
      {
      return (new stj.Integer(2)).add((new stj.Integer(3)));
      } /* value() */
   };
simpleFunction.value();
```

**Figure 87.** Java translation of simple Smalltalk block shown in Figure 85.

Note the use of an anonymous inner class (discussed in section 2.2.2.2). It is used to define a class of type `stj.BlockContext` that differs from the default `stj.BlockContext` class. Specifically, the new class has a customised `value()` instance method. By virtue of the code fragment `new stj. BlockContext()` an instance of the anonymous class is created. This instance is assigned to the variable `simpleFunction`. The Java code `simpleFunction.value()` corresponds to the last line of the Smalltalk code in Figure 85. See section 2.2.2.2 for a more in-depth discussion on anonymous classes in Java.

### 3.5.2. Block with references to variables

To illustrate how a Smalltalk block holds on to the enclosing context, a block is defined (in Figure 88) that refers to two variables outside the definition of the block. The one is a variable that is local to the method that contains the block. The other is an argument to that method.

```
returnBlock: argument
   | simpleFunction localVariable |
   localVariable := 3.
   simpleFunction := [2 * localVariable + argument].
   ^simpleFunction
```

**Figure 88.** Block referring to variables defined outside the scope of a block

This simple example explains how the block holds on to the necessary variables and when executed it will refer to them and continue in its defined context. This happens at runtime. The result of executing the statements in Figure 88 is the block object referenced by the variable `simpleFunction`. In the block object, the variable `localVariable` is bound to `3`. Of course, to execute the statements, one would have to send a `returnBlock:` message to an appropriate object with the argument having some value, in which case the variable called `argument` would be bound to that sent value. Since the result is a block, it will be evaluated if it receives the `value` message. Thus the code `(returnBlock: 5) value` would return `11`.

To translate this block will require a slight modification in constructing a BlockContext object in Java. Because of the restrictions on anonymous inner classes where the class cannot refer to any variables outside the class unless the variables are declared final (see section 2.2.2.2), it is necessary to introduce another level of indirection. One way around the restriction is to declare an array of type `stj.Object` outside the anonymous class. The dimensions of the array correspond to the number of variables appearing in the block. For each such variable, an integer with the modifier `final` is declared outside the anonymous class, which will be used as an index of the array. An assignment to a variable outside the block in the Smalltalk code is simulated by an assignment to the array at the corresponding index in the Java code. Since the array is passed to the anonymous class as a parameter, and since its indices (being `final`) can be referenced within the anonymous class, assignment to the array can also occur within the anonymous class, simulating assignments to variables that take place within the block and that occur as a result of a value message to the block. The strategy is the following:

1. Tally the number of variables that are referenced in the block but are declared outside the block.

2. Create an array of type `stj.Object` with the name `__temp` and the number of elements in the array equal to the tally in step 1.

3. Assign a number to each variable and use this as the index into the array created in step 2. For this purpose, declare a local integer variable with modifier `final` and with name defined by a prefix `__idx_` and a suffix corresponding to the name of the Smalltalk variable.

4. Copy the value of the Smalltalk variable into the indexed element of the array whenever a Smalltalk assignment to that variable occurs.

5. Use the array with the index instead of the variable.

6. When constructing a block object pass the array in as an argument for the block to use. Due to Java's garbage collection features the array will be released when no other object (including the newly created block) refers to the array.

To save space in every method and prevent the generation of different named local temporary arrays it was decided that every object in the system have access to an array by declaring an array in `stj.Object` – called `__temp`. By referring to `this.__temp` the translator may keep on using the same name for the temporary array even when having nested levels of block definitions. The following declaration was thus added to `stj.Object` as seen in Figure 89.

```
public stj.object __temp[] = null;
```

**Figure 89.** Extra declaration added to `stj.Object`

An additional constructor for `stj.BlockContext` was also defined, as shown in Figure 90, allowing for an array to be passed as a constructor argument in order to initialise the `__temp` variable.

```
public class stj.BlockContext extends stj.Object
   {
   public stj.BlockContext() {…}  -defined as before
   public stj.BlockContext(stj.Object tempList[])
      {
      __temp = tempList;
      }
   public stj.Object value() {…}  -defined as before
   }
```

**Figure 90.** Augmented definition of `stj.BlockContext` with new constructor accepting temporary array

The translation for the Smalltalk code in Figure 88 is shown in the Java excerpt in Figure 91.

```
public stj.Object method1_(stj.Object argument)
   {
/* Translation of Smalltalk variable declaration starts here */
   stj.Object simpleFunction = null;
   stj.Object localVariable = null;

/* Translation that is triggered by the presence of a block */
   final int __idx_localVariable = 0;
   final int __idx_argument = 1;
   this.__temp = new stj.Object[2];
   this.__temp[__idx_localVariable] = localVariable;
   this.__temp[__idx_argument] = argument;

/* Translation of localVariable := 3 */
   this.__temp[__idx_localVariable] = (new stj.Integer(3));

/* Translation of simpleFunction := [2 * localVariable + argument] */
   simpleFunction = new stj.BlockContext(__temp)
      {
      public stj.Object value()
         {
         return ( new stj.Integer(2)).mul(
            this.__temp[__idx_localVariable]).add(
            this.__temp[__idx_argument]);
         } /* value() */
      };
/* Translation of ^simpleFunction */
   return(simpleFunction);
   }
```

**Figure 91.** Java translation of Figure 88.

### 3.5.3. Block with block arguments

Another example (Figure 92) will show how a variable is passed into a block as an argument and how the block uses the updated variable (argument) every time.

```
TestBlock>>getBlock: aVar
   | complexFunction localVariable |
   localVariable := aVar * 2.
   complexFunction := [:x | x * localVariable].
   ^complexFunction

TestBlock>>test
   | function |
   function := self getBlock: 1.
   STJTranscript cr; show: ((function value: 2) printString).
   function := self getBlock: 2.
   STJTranscript cr; show: ((function value: 2) printString).
```

**Figure 92.** Example of block accepting arguments

When executing the code in Figure 92 the Transcript window prints out `4` and `8` respectively. Note the use of the `value:` message to a block. If it is necessary to send two arguments to a block the `value:value:` message will be sent. The Squeak Smalltalk implementation caters for up to four arguments through the `value:value:value:value:` message and if more than four arguments are necessary then the `valueWithArguments:` message can be used which accepts an array with a variable number of arguments.

An extra rule is thus added to the block translating rules given above:

Count the number of arguments in the block and choose the appropriate `value_(stj.Object arg)` up to `value_value_value_value_(stj.Object arg1, stj.Object arg2, stj.Object arg3, stj.Object arg4)` method. If there are more than four arguments use the `valueWithArguments_(stj.Object args[])` method.

The previous rules remain applicable and their use results in the following translation:

```
public stj.Object getBlock_(stj.Object aVar)
   {
   stj.Object complexFunction = null;
   stj.Object localVariable = null;

   final int __idx_localVariable = 0;

   this.__temp = new stj.Object[1];
   final stj.Object __blocks[] = new stj.BlockContext[1];

   this.__temp[__idx_localVariable] = aVar.mul((new stj.Integer(2)));
   complexFunction = new stj.BlockContext(__temp)
      {
      public stj.Object value_(stj.Object x)
         {
            return(x.mul(this.__temp[__idx_localVariable]));
         } /* value() */
      };
   return(complexFunction);
   }

public stj.Object test ()
   {
   stj.Object function = null;

   function = this.getBlock_((new stj.Integer(1)));
   STJTranscript.cr().show_(function.value_((new
         stj.Integer(2))).printString());
   function = this.getBlock_((new stj.Integer(2)));
   STJTranscript.cr().show_(function.value_((new
         stj.Integer(2))).printString());
   return(this);
   }
```

**Figure 93.** Java translation of block accepting arguments

### 3.5.4. Block contexts

In Smalltalk a BlockContext is a first class object, it can be passed around as an argument and it has its own state. This allows BlockContext objects to be created in one context and to be executed in another context while retaining their own context. The next example illustrates this property:

```
TestBlock>>getABlock: aBoolean
   | x b |
   x := 0.
   aBoolean
      ifTrue: [b := [x+1]]
      ifFalse: [b := [x+2]].
   x := 100.
   ^b

TestBlock>>testBlock: aBoolean
   | block x v |
   block := self getABlock: aBoolean.
   x := 200.
   v := block value.
   ^v

TestBlock>>run
   | tb |
   tb := TestBlock new.
   Transcript cr; show: (tb testBlock: true) printString.
   Transcript cr; show: (tb testBlock: false) printString
```

**Figure 94.** Block with variables and long lived block context example

In Figure 94, `getABlock:`, creates a block with a reference to a local variable `x and it` returns with `^b` where `b` is a reference to a block. In another method, `testBlock`, this returned object is assigned to the variable called `block` and is subsequently sent the message `value`. The result is returned in the variable v and is eventually printed. The example serves to illustrate that a block executes in its defining context.

Consider the `getABlock:` method: In Smalltalk the `BlockContext` will hold onto the state of the variable `x`. This means that if `x` is modified after a block's definition (as, for example, as a result of the assignment `x := 100` in Figure 94), then the preceding block containing `x` will acquire the modified value of `x`. This behaviour has to be reflected in the translated Java code.

Because of the translation method, access to the variable `x` is indirect and through an array. Furthermore, both the block definition and the method definition hold on to the same array. They can thus both access and change the same variable.

Even though there is a variable called `x` in the context where the `value` message is sent (i.e the `testBlock` context), the block has already been bound to the variable `x` in the method `getABlock:`. Note, also, that the binding is dynamic. The variable `x` is changed from an initial value of `0` to a new value of `100` after the creation of the block and this change is reflected in the block when it is subsequently executed. As a result `TestBlock>>run` causes `101` and `102` to be displayed, not `1` and `2` as would have been the case if an `x` value of `0` was used, nor `201` and `202` as would have been the case if an `x` value of `200` was used.

The translation of Figure 94 is show in Figure 95.

```
public class TestBlock extends stj.Object
   {
   public stj.Object getABlock_(stj.Object aBoolean)
      {
      stj.Object x = null;
      stj.Object b = null;

      final int __idx_b = 0;
      final int __idx_x = 1;

      this.__temp = new stj.Object[2];
      this.__temp[__idx_x] = (new stj.Integer(0));

      aBoolean.ifTrue_ifFalse_(
      new stj.BlockContext(__temp)
         {
         public stj.Object value()
            {
            return( this.__temp[__idx_b]=new stj.BlockContext(__temp)
               {
               public stj.Object value()
                  {
                  return( this.__temp[__idx_x].add((new
                        stj.Integer(1))));
                  }) /* value() */
               };
            } /* value() */
         },
      new stj.BlockContext(__temp)
```

```
        {
        public stj.Object value()
           {
           return( this.__temp[__idx_b]=new stj.BlockContext(__temp)
              {
              public stj.Object value()
                 {
                 return( this.__temp[__idx_x].add((new
                       stj.Integer(2)));
                 } /* value() */
              };
           } /* value() */
        });
     this.__temp[__idx_x] = (new stj.Integer(100));
     return(this.__temp[__idx_b]);
     }

  public stj.Object testBlock_(stj.Object aBoolean)
     {
     stj.Object block = null;
     stj.Object x = null;
     stj.Object v = null;

     block = this.getABlock_(aBoolean);
     x = (new stj.Integer(200));
     v = block.value();
     return(v);
     }

  public stj.Object run()
     {
     stj.Object tb = null;
     tb = metaclass.TestBlock.__class.__new();
     Transcript.cr().show_(
        tb.testBlock_(stj.Object.__true).printString());
     Transcript.cr().show_(
        tb.testBlock_(stj.Object.__false).printString());
     }
  }
```

**Figure 95.** Java translation of Smalltalk blocks executing with their own context

### 3.5.5. Blocks with non-local returns

The example in Figure 96 below shows how a method called `range:` returns `10` if the argument `amount` is less than 100 (`< 100`); otherwise it returns `20`. This method therefore has two exit points.

```
range: amount
   ^amount < 100 ifTrue: [10] ifFalse: [20]
```

**Figure 96.** Implementation of `range:`

Suppose a new implementation of range: is required where it returns 10 if the argument `amount` is less than `100` (`< 100`), `20` if the argument `amount` is greater than or equal to `100` (`>= 100`) and less than `150` (< 150) and `30` if the argument amount is greater than or equal to `150` (`>= 150`). An implementation is shown in Figure 97.

```
range: amount
   ^amount < 100
      ifTrue: [10]
      ifFalse: [(amount >= 100 and: [amount < 150])
         ifTrue: [20]
         ifFalse: [amount >= 150
            ifTrue: [30]
            ]
         ]
```

**Figure 97.** New implementation of `range:`

Note that in this code, there are two exit points out if the block following the outer ifFalse: method: the exit returning 20 and the exit that returns 30. In general, both methods and blocks in Smalltalk are allowed to have multiple exit points. It frequently happens that an implementation such as in Figure 97 with multiple exit points can be rewritten to make it more readable, expressing the intent more clearly and using the minimum number of comparisons. What is required is the use of a so-called non-local return within blocks. The rewritten implementation is shown in Figure 98.

```
range: amount
   amount < 100 ifTrue: [^10].
   amount < 150 ifTrue: [^20].
   30
```

**Figure 98.** More readable implementation of `range:` - equivalent to Figure 97.

Note the use of a non-local return statement in the blocks themselves (indicated by the ^ symbol immediately following the [ symbol). The effect of this is that the block will return immediately and return out of the enclosing context in which it is defined. Assuming an argument of `95` is passed to `range:` If the block only had been executed, i.e. if the block had been defined as `[10]` instead of `[^10]`, then the method would continue execution and the result returned would be 30 and not 10 as expected. This is because Smalltalk always returns the result of the last statement executed in a method or a block. Even if the last statement was written as `amount >= 150 ifTrue: [30]` the result of evaluating it would return `nil` since 95 is not >= than 150.

The next example, figure 99, shows how Smalltalk blocks are bound to their enclosing context not only in the sense of relying on variable bindings in that context, but also in the sense of executing the non-local return in that context, no matter where the block was evaluated. In `method1` a block is created which returns the string `'non-local return'`. This block is then passed to `method2:` where it is evaluated by receiving the `value` message.

```
Object subclass: #TestBlock
   instanceVariableNames: 'resultVar'

TestBlock>>method1
   | block |
   resultVar := 'nothing'.
   block := [^ 'non-local return'].
   self method2: block.
   resultVar := resultVar , ' method1'.    ^ 'end of method1'11

TestBlock>>method2: aBlock
   aBlock value.
   resultVar := resultVar , ' method2'

TestBlock>>result
   ^resultVar

TestBlock>>run
   | tb |
   tb := TestBlock new.
   Transcript cr; show: tb method1.
   Transcript cr; show: tb result.
```

**Figure 99.** Example of Smalltalk block which returns immediately

When executing the `run` method of `TestBlock` the two lines shown in Figure 100 are written to the Transcript window.

```
non-local return
nothing
```

**Figure 100.** Output of executing the Smalltalk code shown in Figure 99.

Thus, the effect of evaluating the block is that the block returns immediately out of the context in which it is defined. (See the previous section with a similar example related to blocks and variables.) As a result, neither the assignment statement in `method2:` nor the last two statements in `method1` are executed. Consequently, `tb method1` returns the string `'non-local return'`, while `tb result` returns the initial value of the variable `resultVar`, namely `'nothing'`.

---

[11] In Smalltalk the ',' operator concatenates two strings and returns the concatenated result

Had the line `self method2: block` in `method1` been replaced by `block value` the outcome would have been the same.

One way of translating this type of Smalltalk block is to make use of the properties of Java exceptions, discussed in section 2.2.3. What has to be simulated is the fact that when a block returns, it returns out of the context in which it is defined and does not simply continue executing the statements that follow after the block evaluation statements. It is thus necessary to make a small change to the definition of `stj.BlockContext`. The final version of `stj.BlockContext` is shown in Figure 101.

```
package stj;
public class BlockContext extends stj.Object
   {
   public stj.Object __result = null;
   public BlockContext() {}
   public BlockContext(stj.Object tempList[])
      {
      __temp = tempList;
      }
   public stj.Object value() throws BlockException
      {
      return stj.Object.__nil;
      }
   }
```

**Figure 101.** Final definition of stj.BlockContext

The important difference is the fact that the `value()` method can now throw a `BlockException`. A `BlockException` will be used to keep track of which block threw the exception which will indicate if the method enclosing the block should return or not. The `BlockException` object will also hold on to the result of the last executed statement in a block. Figure 102 shows the definition of `stj.BlockException`. When a `BlockException` is created it expects two arguments in the constructor: the first one is a reference to the block that throws the exception; and the second one is the result of the last statement executed in the block before throwing the exception.

```
package stj;
public class BlockException extends Exception
   {
   public stj.Object outcome = stj.Object.__nil;
   public stj.Object block = stj.Object.__nil;
   public BlockException(stj.Object aBlock, stj.Object aResult)
      {
      super("Block Exception");
      outcome = aResult;
      block = aBlock;
      }
   }
```

**Figure 102.** Definition of `stj.BlockException`

Another few lines of code are necessary in the translation process. Two extra variables, `__blocks[]` and `__result` will be used. The `__blocks[]` variable is an array being used to refer to all the blocks being created in the translation process in each method. In the case of a non-local return the `__result` variable is assigned the result of the statement being returned via the non-local return and a throw statement is executed with the current block context being executed as an argument.

The next step is to enclose the code creating the block in a try clause and to provide a subsequent catch clause. This will allow the block to return (throw an exception) without executing the rest of the statements in the try block. Once an exception is thrown, the exception's block variable is compared to the block that was defined in the try clause. If the block variable is equal to the block variable in the exception object this means that the block defined in the try clause threw the exception and is signaling a return. This check is necessary for cases where there are multiple blocks defined and/or the blocks are evaluated a few levels down in the call stack. If there is no match between the exception's block variable and the blocks created in the try clause, the exception is passed on to the next level.. Figure 103 shows the necessary code to be added to the translation.

```
try {
   …// block is defined - __blocks[0] is assigned to the block instance
         block = __blocks[0] = new stj.BlockContext()
            {
            public stj.Object value() throws BlockException
               {
               __result = … /* result of the non-local return statement
*/
               throw new stj.BlockException(this, __result);
               } /* value() */
            };
   …// other statements
   }
catch (BlockException e)
   {  if (((BlockException)e).block == __blocks[0])
      return ((BlockException)e).outcome; throw e;
```

**Figure 103.** Extra code needed for translating immediate return Smalltalk blocks

The result of translating Figure 99 is shown below in Figure 104. The shaded areas illustrates the extra translation rules for blocks mentioned above in Figure 103.

```
public class TestBlock extends stj.Object
   {
   stj.Object resultVar;
   public static void main(java.lang.String[] args)
      {
      try {
         stj.Object tb = null;
         tb = metaclass.TestBlock.__class.__new();
         Transcript.cr().show_(tb.method1());
         Transcript.cr().show_(tb.result());
         }
      catch (BlockException e) {};
      } public stj.Object method1() throws BlockException
      {
      stj.Object block = null;
      final stj.Object __blocks[] = new stj.BlockContext[1];
      try {
         resultVar = (new stj.String("'nothing'"));
         block = __blocks[0] = new stj.BlockContext()
            {
            public stj.Object value() throws BlockException
               {
               __result = (new stj.String("'non-local return'"));
```

```
                throw new stj.BlockException(this, __result);
                } /* value() */
            };
        this.method2_(block);
        resultVar = resultVar.commaConcatenate
                                        ((new stj.String("' method1'")));
        return((new stj.String("'end of method1'")));
        }
    catch (BlockException e)
        {
        if (((BlockException)e).block == __blocks[0])
            return ((BlockException)e).outcome;
        throw e;
        }
    }
  public stj.Object method2_(stj.Object aBlock) throws BlockException
{
    aBlock.value();
    resultVar = resultVar.commaConcatenate
                                        ((new stj.String("' method2'")));
  return(this);
    }
  public stj.Object result() throws BlockException
    {
    return(resultVar);
    }
  }
```

**Figure 104.** Translated Java code of Smalltalk code shown in Figure 99.

It is necessary for the translator to make at two passes through the code. In the first pass, it counts the number of blocks needed to declare the size of the array `__blocks[]`. In the second pass, the translator generates the Java code.

### 3.5.6. Nested blocks

The occurrence of nested blocks in Smalltalk is common and it follows naturally that the translator must be able to translate it successfully. Since all blocks in Smalltalk are treated as objects, each with their own executing context it is not necessary in the translation to have different levels of exception handling, each with its own scope. The following example in Figure 105 illustrates a block accepting one argument being returned from within 2 levels of blocks.

```
TestBlock>>run
   | tb |
   tb := TestBlock new.
   STJTranscript cr; show: ((
      (tb testBlock: true arg2: true) value: 1) printString).
   STJTranscript cr; show: ((
      (tb testBlock: true arg2: false) value: 1) printString).
   STJTranscript cr; show: ((
      (tb testBlock: false arg2: true) value: 1) printString).
   STJTranscript cr; show: ((
      (tb testBlock: false arg2: false) value: 1) printString).

TestBlock>>testBlock: arg1 arg2: arg2
   arg1 = true
      ifTrue: [
         arg2 = true
            ifTrue: [^[:x | x * 3] ]
            ifFalse: [^[:x | x * 2] ]
      ]
      ifFalse: [
         arg2 = true
            ifTrue: [^[:x | x * 1] ]
            ifFalse: [^[:x | x * 0] ]
      ].
```

**Figure 105.** Nested block example

Since each block is handled without being dependent on the relative depth to other blocks the translation is straightforward and follows in Figure 106.

```
public class TestBlock extends stj.Object
   {
   public static void main(java.lang.String[] args)
      {
      try
         {
         stj.Object tb = null;
         tb = metaclass.TestBlock.__class.__new();
         STJTranscript.cr().show_(tb.testBlock_arg2_(
            stj.Object.__true, stj.Object.__true).value_((
               new stj.Integer(1))).printString());
         STJTranscript.cr().show_(tb.testBlock_arg2_(
            stj.Object.__true, stj.Object.__false).value_((
               new stj.Integer(1))).printString());
         STJTranscript.cr().show_(tb.testBlock_arg2_(
            stj.Object.__false, stj.Object.__true).value_((
               new stj.Integer(1))).printString());
         STJTranscript.cr().show_(tb.testBlock_arg2_(
            stj.Object.__false, stj.Object.__false).value_((
               new stj.Integer(1))).printString());
         }
      catch (BlockException e) {};
      }

   public stj.Object testBlock_arg2_(stj.Object arg1, stj.Object arg2)
throws BlockException
      {

      final int __idx_arg2 = 0;

      this.__temp = new stj.Object[1];
      this.__temp[__idx_arg2] = arg2;
      final stj.Object __blocks[] = new stj.BlockContext[10];

      try
         {
         arg1.equalsValue(stj.Object.__true).ifTrue_ifFalse_(
            __blocks[0] = new stj.BlockContext(__temp)
            {
            public stj.Object value() throws BlockException
               {
               __result = this.__temp[__idx_arg2].equalsValue(
                  stj.Object.__true).ifTrue_ifFalse_(
                        __blocks[1] = new stj.BlockContext(__temp)
                  {
                  public stj.Object value() throws BlockException
```

```
                        {
                        __result = __blocks[2] = new
stj.BlockContext(__temp)
                              {
                              public stj.Object value_(stj.Object x)
                                    throws BlockException
                                    {
                                    __result = x.mul((new
                                          stj.Integer(3)));
                                    return(__result);
                                    } /* value() */
                              };
                        throw new stj.BlockException(this, __result);
                        } /* value() */
                  }
               , __blocks[3] = new stj.BlockContext(__temp)
                  {
                  public stj.Object value() throws BlockException
                        {
                        __result = __blocks[4] = new
                              stj.BlockContext(__temp)
                              {
                              public stj.Object value_(stj.Object x)
                                    throws BlockException
                                    {
                                    __result = x.mul((new
                                          stj.Integer(2)));
                                    return(__result);
                                    } /* value() */
                              };
                        throw new stj.BlockException(this, __result);
                        } /* value() */
                  });
               return(__result);
               } /* value() */
            }
         , __blocks[5] = new stj.BlockContext(__temp)
            {
            public stj.Object value() throws BlockException
               {
               __result = this.__temp[__idx_arg2].equalsValue(
                  stj.Object.__true).ifTrue_ifFalse_(
                        __blocks[6] = new stj.BlockContext(__temp)
                  {
                  public stj.Object value() throws BlockException
                        {
```

```
                    __result = __blocks[7] = new
                          stj.BlockContext(__temp)
                          {
                          public stj.Object value_(stj.Object x)
                                throws BlockException
                                {
                                __result = x.mul((new
                                      stj.Integer(1)));
                                return(__result);
                                } /* value() */
                          };
                    throw new stj.BlockException(this, __result);
                    } /* value() */
              }
           , __blocks[8] = new stj.BlockContext(__temp)
              {
              public stj.Object value() throws BlockException
                    {
                    __result = __blocks[9] = new
                          stj.BlockContext(__temp)
                          {
                          public stj.Object value_(stj.Object x)
                                throws BlockException
                                {
                                __result = x.mul((new
                                      stj.Integer(0)));
                                return(__result);
                                } /* value() */
                          };
                    throw new stj.BlockException(this, __result);
                    } /* value() */
              });
           return(__result);
           } /* value() */
        });
     return(stj.Object.__nil);
     }
  catch (BlockException e)
     {
     if (((BlockException)e).block == __blocks[0])
        return ((BlockException)e).outcome;
     if (((BlockException)e).block == __blocks[1])
        return ((BlockException)e).outcome;
     if (((BlockException)e).block == __blocks[2])
        return ((BlockException)e).outcome;
     if (((BlockException)e).block == __blocks[3])
```

```
            return ((BlockException)e).outcome;
         if (((BlockException)e).block == __blocks[4])
            return ((BlockException)e).outcome;
         if (((BlockException)e).block == __blocks[5])
            return ((BlockException)e).outcome;
         if (((BlockException)e).block == __blocks[6])
            return ((BlockException)e).outcome;
         if (((BlockException)e).block == __blocks[7])
            return ((BlockException)e).outcome;
         if (((BlockException)e).block == __blocks[8])
            return ((BlockException)e).outcome;
         if (((BlockException)e).block == __blocks[9])
            return ((BlockException)e).outcome;
         throw e;
         }
      }
```

**Figure 106.** Translation of nested blocks example

### 3.5.7. Reference to self in block

In the translation process a number of extra rules is formed, one of them being that references to `self` in Smalltalk will translate to the `this` variable in Java. The rule is correct, except for the case where the `self` variable is referred to in the context of a block. By replacing the reference to `self` with `this`, the context of the BlockContext is accessed, instead of the outer context of the object in which the BlockContext is defined. This result in the wrong context being used with incorrect results. Figure 107 shows an example of referring to `self` in the block being passed to the `timesRepeat:` method.

```
TestBlock>>run
      100000 timesRepeat: [
          (self testBlock: true arg2: true) value: 1.
      ].

TestBlock>>testBlock: arg1 arg2: arg2
   "Same definition as before" …
```

**Figure 107.** Referring to `self` in a block definition

The translation process in this case will involve a call on all BlockContext objects in a method to test if a reference to `self` is made. In the case of a reference to `self`, an extra variable `self` will be added to the variable declaration section of the method. Since the `self` variable will be accessed in the BlockContext it must be declared `final` according to section 2.2.2.1.1. In Figure 108 the translation shows the declaration of the final `self` variable as well as the usage.

```
public class TestBlock extends stj.Object
    {
    public stj.Object run() throws BlockException
        {
        final stj.Object self = this;
            (new stj.Integer(100000)).timesRepeat_(
                __blocks[0] = new stj.BlockContext()
                {
                public stj.Object value() throws BlockException
                    {
                    self.testBlock_arg2_(stj.Object.__true
                        , stj.Object.__true).value_((new stj.Integer(1)));
                    return(__result);
                    } /* value() */
                });
            return(this);
            }
        catch (BlockException e)
            {
            if (((BlockException)e).block == __blocks[0])
                return ((BlockException)e).outcome;
            throw e;
            }
        }
    public stj.Object testBlock_arg2_(stj.Object arg1, stj.Object arg2)
        throws BlockException
        {
        // Same definition as before
        …
        }
    }
```

**Figure 108.** Translation of reference to `self` in BlockContext

### 3.5.8. Performance of blocks using exceptions

As discussed in 2.2.3.4 there is a marked difference in the Java VM when returning from a method via throwing an exception versus a normal return. Performance tests were done on the normal usage of blocks. The first test involved running the code in Figure 107 in the Squeak Smalltalk VM. The second test executed the translated Java code in Figure 108 to illustrate the impact of the return via exception mechanism. The last test measured the performance of returning via a normal Java `return` statement with the created block object as an argument. The results are tabulated below in Figure 109 and illustrates the overhead on the Java VM in locating the correct exception handler. The tests were executed three times and the average taken.[12]

| Execution strategy | Result in milliseconds |
|---|---:|
| **Squeak Smalltalk VM** | 925 |
| **Java translation with exception handling return** | 12685 |
| **Java translation with method return** | 985 |

**Figure 109.** Benchmark results of exception handling strategy in translating Smalltalk blocks

The results are not surprising since section 2.2.3.4 illustrated the marked difference in speed between a normal method return and locating an exception handler. The test above shows a dramatically reduced factor in the overhead between a normal method return and the exception handling strategy. It is envisaged that in normal Smalltalk examples the factor might even drop more to more acceptable levels, depending on the number of non-local returns that are translated and actually executed.

[12] The benchmarks were performed on an Intel Pentium 4 – 1GHz, running Linux (Red Hat 7.2) with Squeak 3.0 and version 1.3.1 of the Sun Java Development Kit.

## 3.6. Summary

The purpose of this Chapter was to explain the mapping from Smalltalk source code to Java source code. It is now possible to translate most Smalltalk programs to Java source that will execute in a Java Virtual Machine. Smalltalk features not translated will be addressed in 5.2.1. The area where Java is lacking is in its reflection capabilities and to treat a block of code as an object that will make BlockClosures easier to use. The use of blocks is integral to Smalltalk and the translated Java source of Smalltalk blocks is quite difficult to follow.

In Chapter 4 that now follows, some of the pertinent implementation issues will be addressed.

# 4. Implementing a translator

In order to translate from Smalltalk source code to Java it is necessary to develop a scanner and parser program to do the lexical and syntax analysis of the Smalltalk code. It was decided to use the Squeak Smalltalk system as a development environment. Adapting the already existing scanner and parser in the Squeak Smalltalk system shortened the implementation cycle of the new scanner and parser. The output of this code is a parse tree that is used to generate Java code. It is thus clear that there are two distinct parts in the translation phase from a source language to a target language. Wirth (1996:3) refers to these modules as the front end and back end.

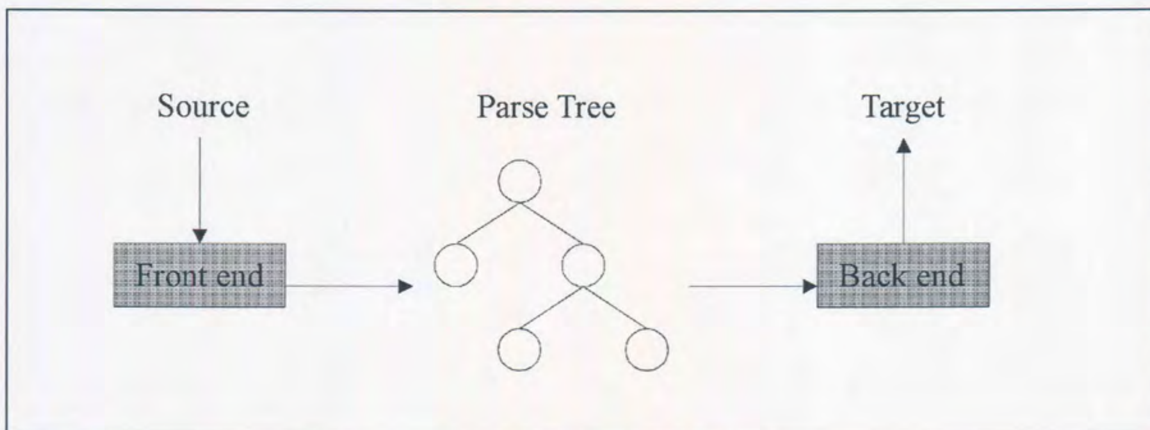


**Figure 110.** Illustrating front-end, parse tree and back-end of a parser

The diagram above shows the front end generating a parse tree as output and passing it to the back end. The advantage of de-coupling the source language and target architecture is that different source languages can have their own front ends generating parse trees for a single back end. The natural effect is that instead of implementing $m * n$ translators ($m$ source languages and $n$ targets) only $m + n$ translators need to be built.

The implementation of a front end deals with lexical analysis and syntax analysis. The lexical analyser reads the input source and groups the input stream into tokens. After the lexical analyser grouped the input into tokens the tokens are passed to the syntax analyser. The syntax analyser has two functions. The first one is to verify that the tokens are valid and permitted by the specification of the source language. For example, assume the following expression is passed to the lexical analyser:

```
A + * B
```

**Figure 111.** Expression with 4 tokens

After the lexical analysis the expression will appear to the syntax analyser as the following token sequence:

```
identifier + * identifier
```

**Figure 112.** Token sequence

Although the token sequence consists of valid tokens, the syntax is wrong and will be rejected by the syntax analyser.

The second function of the syntax analyser is to group the tokens into a hierarchical structure known as a parse tree. The following expression:

```
A + B * C
```

**Figure 113.** Expression resulting in parse tree in Figure 114

will be translated into a parse tree where certain parts of the token sequence are grouped together. Depending on the operator precedence the parse tree will be different. The diagram below shows the parse tree where the multiplication operator has a higher precedence over the plus operator as in languages like Java, C/C++ and Pascal. The second parse tree shows how the Smalltalk parser will generate the parse tree. Since Smalltalk does not have any operator precedence the syntax analyser parses the messages from left to right, which explains the + message being evaluated before the * message.

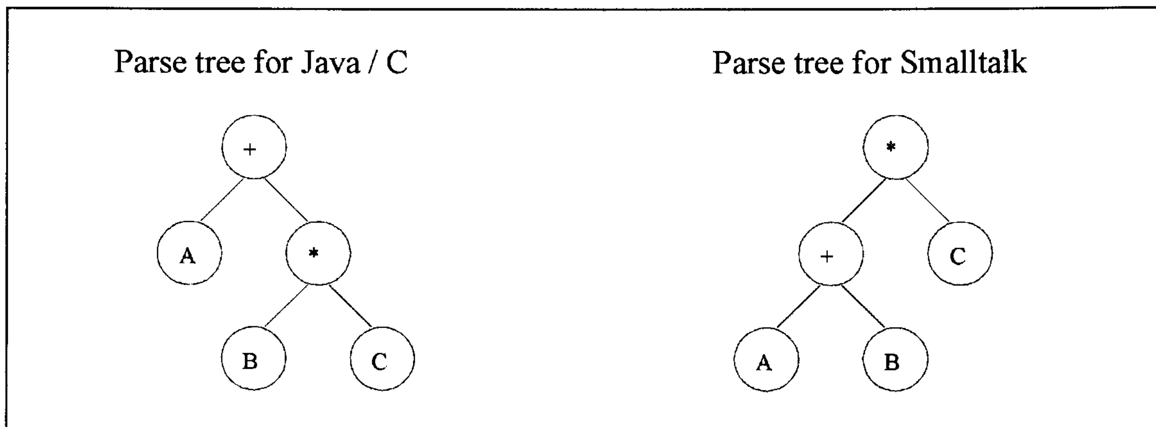


**Figure 114.** Parse tree illustrating operator precedence

Once the correct parse tree is constructed by the parser it is passed to the back end to generate code. The code generator will start with the root node and systematically traverse all the nodes in order to produce code in the target language. In section 4.4 the code generation process will be discussed. Before doing so however, a brief account of Smalltalk's grammar is presented in section 4.1, followed by a description of the lexical analyser (section 4.2) and parser (section 4.3) that were modified and adapted in Squeak Smalltalk.

It should be emphasised that these discussions are not intended to be detailed description of the Smalltalk to Java prototype translator that has been built to test the ideas spelt out in previous chapters. Instead, a CD accompanies this dissertation that contains all the source code of the Smalltalk to Java translator in a Squeak Smalltalk image, ready to be executed. The source code of all the samples used for illustrating the translator is contained in the image as well. In Smalltalk the whole environment with all the source code and runtime objects are persisted in a single file referred to as the image.

In Squeak Smalltalk all classes (with their source code) belong to a category. The STJ-Translator category contains the STJTranslator class. STJ-Compiler contains the Scanner (stj.Scanner), the Parser (stj.Parser) and all the parse nodes produced by the Parser. STJ-Tests contains all the classes used for testing the translator.

When a class is given to the translator (STJTranslator) instance as an argument the translator translates the Smalltalk source to two Java source files according to the guidelines discussed previously. The Java source files are the following: one for the metaclass (with all the class methods) and one for the class (with all the instance methods).

## 4.1. The Smalltalk grammar

The Smalltalk language itself is very simple. Kay (1996) reports that the simplicity was spurred by his challenge to his colleagues in 1968 to produce a clean and simple language with a grammar that could fit on a napkin. With some effort Kay and his team succeeded in being able to write down a simple grammar on a single page. This section explains the Smalltalk grammar as it is implemented today in Extended Backus-Naur Form (EBNF).

### 4.1.1. Method definition

A method definition consists of a method name that is known as the selector or name of the method. The method name is followed by optional temporary variable declarations and then a `<statements>` sequence.

```
<method definition> ::=
   <message pattern>
   [<temporaries>]
   [<statements>]
<message pattern> ::=
   <unary pattern> |
   <binary pattern> |
   <keyword pattern>
<unary pattern> ::= identifier
<binary pattern> ::= binarySelector identifier
<keyword pattern> ::= (keyword identifier)+
```

**Figure 115.** Method definition

A `<message pattern>` can be one of three types:

A `<unary pattern>` which consists only of an identifier. An example is '`print`' and an example of an associated message is '`object print`' which sends the `print` message to an `object`.

A `<binary pattern>` which consists of a binary selector and one argument. An example is '`+ 1`' and an example of an associated message is '`value + 1`' which sends the + message to an `Integer` object with another `Integer` object as an argument.

A `<keyword selector>` which consists of one or more keywords and the same number of arguments. For example '`value max: y`' means sending a message to the `value` object

requesting the maximum of itself and the argument y. In most other languages it will be written as 'max(value, y)'.

The list of temporary variables in a method is simply a list of variable names enclosed by '|'.

```
<temporaries> ::= '|' <temporary list> '|'
<temporary list> ::= identifier*
```

**Figure 116.** Definition of temporary variables

### 4.1.2. Statements

After the declaration of temporary variables it is followed by a list of statements. Each statement except the final statement is an <expression>. The last statement in the <statements> sequence may be either an <expression> or <return statement>. Each <expression> is separated from its following <statement> by a '.'. A period is optional following the last statement. If the last <expression> in a <statements> sequence is preceded by a '^' the <expression> forms a <return statement> and the result of the <expression> is the value of the return statement. For the purposes of the present study, a slight modification was made to <statements> in to the Smalltalk grammar to allow a <java statement> which contains lines of Java code. These lines of Java code are output directly in the translated code. This feature is useful for accessing Java only constructs and/or variables that are not defined in the Smalltalk system. These statements are used for debugging purposes in translations and are not part of the translation scheme *per se*.

```
<statements> ::=
   <return statement> ['.'] |
   (<expression> ['.' [<statements>]]) |
   <java statement>
<return statement> ::= '^' <expression>
<java statement> ::= '{' 'java code to be injected in the generated
code' '}'
```

**Figure 117.** Definition of statements

### 4.1.3. Expressions

An <expression> is either an <assignment> or a <basic expression>. A <basic expression> consists of a <primary> optionally followed by <messages> or <cascaded messages>. An <expression> describes a sequence of tokens that refers to an object or a computation that produces a reference to an object. An <expression> may optionally specify that its value is to be assigned to one or more variables.

An <assignment> is a variable name that is called the target of the assignment. An <assignment> may assign its value to multiple target variables by utilising multiple <assignment> clauses.

```
<expression> ::=
   <assignment> |
   <basic expression>
<assignment> ::= identifier ':=' <expression>
<basic expression> ::= <primary> [<messages> <cascaded messages>]
<primary> ::=
   identifier |
   <literal> |
   <block constructor> |
   ( '(' <expression> ')' )
```

**Figure 118.** Definition of an expression

The components of <basic expression> are described below.

### 4.1.4. Blocks

A block can be created and manipulated as an object. A block is always defined within another function, called its enclosing function. The result is that a block is always nested within a method or another block.

Evaluation of a block is finished when the last statement of the block has executed. The value of the last statement is returned as the value of the message sent that executed the block. If the last statement of the block was a <return statement> then the evaluation of the block's enclosing function is also terminated and the value of the <return statement> is used as the return value of the enclosing function.

Expressions within a block may refer to temporary variables and arguments in the enclosing function. In this way a block is an independent closure and any referenced variables are bound to the blocks context. These bindings will be maintained for as long as the block is used in the system.

```
<block constructor> ::= '[' <block body> ']'
<block body> ::= [<block argument>* '|'] [<temporaries>] [<statements>]
<block argument> ::= ':' identifier
```

**Figure 119.** Definition of a block

### 4.1.5. Messages

Messages cause the activation of a method. There are three different types of "message sends". They correspond to the three types; `<unary message>`, `<binary message>` and `<keyword message>`. Every message sent results in a value that corresponds to the result returned by the method. The *receiver* of a message is the value of the `<primary>` or the result returned by the message sent to the immediate left of a message's selector. The receiver is a reference to an object. The receiver and the arguments are evaluated before the message is sent and evaluated in a left-to-right order.

Unary messages require no arguments. Binary messages require one argument, and keyword messages take one or more arguments and are composed of a sequence of keywords followed by expressions. It is possible to construct a `<messages>` clause with multiple messages to be sent.

A *cascade* is a sequence of messages that are all directed to the same object. Only the first in such a sequence has an explicit receiver specified. The receiver of the subsequent messages is the same object as the receiver of the of the initial message in the sequence.

```
<messages> ::=
   (<unary message>+ <binary message>* [<keyword message>]) |
   (<binary message>+ [<keyword message>]) |
   <keyword message>
<unary message> ::= identifier
<binary message> ::= binarySelector <binary argument>
<binary argument> ::= <primary> <unary message>*
<keyword message> ::= (keyword <keyword argument>)+
<keyword argument> ::= <primary> <unary message>* <binary message>*
<cascaded messages> ::= (';' <messages>)*
```

**Figure 120.** Definition of messages

### 4.1.6. Terminals

The following productions are used in the lexical analyser to produce the tokens used in the syntax analysis phase.

```
keyword ::= identifier ':'
identifier ::= letter (letter | digit)*
binarySelector ::= binaryCharacter+
binaryCharacter ::= 'special characters, i.e. +, -'
letter ::= 'alphabetic characters'
digit ::= 'digits from 0-9'
```

**Figure 121.** Definition of terminal tokens

### 4.1.7. Reserved Identifiers

The following identifiers are reserved words in Smalltalk. They may only be used as a <primary> and are defined as follows:

**nil** A constant binding to a unique object. The scope of the binding is the entire program. Variables that have not been initialised initially are assigned this value.

**true** A constant binding to a unique object. The scope of the binding is the entire program.

**false** A constant binding to a unique object. The scope of the binding is the entire program.

**self** Within a method, a constant binding to the receiver of the message that activated the method. The scope of the binding is a single method activation. In a class method it is a constant binding to the associated class object.

**super** Within a method, a constant binding to the receiver of the message that activated the method. The scope of the binding is a single method activation. The binding of *super* is to the same object as the binding of self, but causes message lookup to start in the superclass of the class containing the method in which super appears, rather than starting in the class of the receiver.

## 4.2. Lexical and Syntax Analysis

To perform lexical analysis the Squeak Smalltalk Scanner was used and modified to include the changes made in the Smalltalk grammar for translating to Java.

The implementation of the Scanner is fairly straightforward. The next 4 methods, `initialize`, `step`, `scanToken` and `xDigit` are excerpts from the Smalltalk image. For full source code refer to the class named `stj.Scanner` in the Squeak-STJ image on the accompanying CD. It has a class variable, `TypeTable`, an instance of an `Array` with ASCII codes being the index and the values at the index of the array being a symbol describing the corresponding type of the *lexeme*. The array is initialised with statements such as those below – i.e. initialising the delimiter, digit and letter lexeme codes.

```
stj.Scanner class>>initialize
   TypeTable atAll: #(9 10 12 13 32) put: #xDelimiter. "tab lf ff cr
space"
   TypeTable atAll: ($0 asciiValue to: $9 asciiValue) put: #xDigit.
   TypeTable atAll: ($A asciiValue to: $Z asciiValue) put: #xLetter.
   TypeTable atAll: ($a asciiValue to: $z asciiValue) put: #xLetter.
```

**Figure 122.** Initialising the `stj.Scanner` table of types

An important method on the Scanner is the step method. This returns the current character in the input stream and increments the position in the stream to point to the lookahead character.

```
stj.Scanner>>step
   | t1 |
   t1 := hereChar.
   hereChar := aheadChar.
   source atEnd
      ifTrue: [aheadChar := 30 asCharacter]
      ifFalse: [aheadChar := source next].
   ^ t1
```

**Figure 123.** Implementation of the `step` (lookahead) method

The `scanToken` method is the most important method. It starts by scanning for white space (previously defined as `#xDelimiter`) and when the white space is finished the type of the next token is determined. If the token type determines that the token consists of more than one character (by having `x` as the first character in the description of the token type) the rest of the token is read from the stream by the perform method. Note the use of Smalltalk reflection where

the perform: message is sent to the object activating the relevant method associated with tokenType. A good example is when the scanToken method encounters a character that belongs to the #xDigit set. The scanToken method will read the first character of the number string and because it has an x as the first character the xDigit method (shown in Figure 125) will be performed on the scanner instance.

```
stj.Scanner>>scanToken
   [(tokenType := TypeTable at: hereChar asciiValue) == #xDelimiter]
      whileTrue: [self step].
   mark := source position - 1.
   (tokenType at: 1)
      = $x
      ifTrue: [self perform: tokenType]
      ifFalse: [token := self step asSymbol].
   ^ token
```

**Figure 124.** Implementation of scanToken

The xDigit method starts by simply setting the token type to #number. The highlighted line in Figure 125 reads the number from the stream. The next few lines test for end of stream (EOS) conditions. The readFrom: class method on Number accepts a stream (in this case the source from the Scanner) and then reads a number from the stream, assigning it to the token variable and then returning after increasing the current position in the source to the next token position.

```
stj.Scanner>>xDigit
   tokenType := #number.
   (aheadChar = EOS
      and:
         [source skip: -1.
         source next ~= EOS])
      ifTrue: [source skip: -1]
      ifFalse: [source skip: -2].
   token := Number readFrom: source.
   self step; step
```

**Figure 125.** Implementation of xDigit

Important changes made to the scanner, as illustrated in Figure 126, were to

1. relax the restriction of Smalltalk class names being capitalised and

2. allow full stops as part of the class name.

The reason for this was to simulate in Smalltalk the package concept in Java whereby classes with the same name can coexist in the system by having different package names prefixed to their name. It was thus quite useful as it allowed the Squeak Smalltalk system to have an `Object` class and a `stj.Object` class without conflicts. The `stj.Scanner` and `stj.Parser` classes (modified) could also be kept separate from the original `Scanner` and `Parser` classes. It allowed for transparent mapping between Squeak Smalltalk classes and their counterparts in the Java translation instead of introducing a prefix in the class name of every Squeak Smalltalk class.

```
stj.Scanner>>xLetter
   | t1 |
   buffer reset.
   [((t1 := typeTable at: hereChar asciiValue) == #xLetter or: [t1 ==
#xDigit])
      or: [t1 == #period and: [(typeTable at: aheadChar asciiValue)
               == #xLetter]]]
      whileTrue:
         [buffer nextPut: hereChar.
         hereChar := aheadChar.
         source atEnd
            ifTrue: [aheadChar := EOS]
            ifFalse: [aheadChar := source next]].
   (t1 == #colon
      or: [t1 = #xColon and: [aheadChar ~= $=]])
      ifTrue:
         [buffer nextPut: self step.
         tokenType := #keyword]
      ifFalse: [tokenType := #word].
   token := buffer contents
```

**Figure 126.** Implementation of `xLetter` on `stj.Scanner`

For more examples of different types of tokens and how the scanner handles them refer to the source code in the class `stj.Scanner` in the Squeak-STJ image. A full list of all the token types is shown in Figure 127.

```
xBinary
xColon
xDelimeter
xDigit
xDollar
xDoubleQuote
xLetter
xLitQuote
xSingleQuote
```

**Figure 127.** List of token types in `stj.Scanner`

## 4.3. Recursive Descent Parser

The recursive descent parsing technique is simple to understand and to implement. A top-down approach is followed in which the parser verifies that the syntax of the input stream is correct as it is read from left to right. The parser depends on a basic operation – step (look-ahead) – which is defined in Figure 123, section 4.2. It involves stepping through the source character by character and determining the token which has been scanned. The parser then verifies token types against EBNF requirements.

Another way to explain what a recursive descent parser actually does is the following: the parser performs a depth-first search of the derivation tree for the string being parsed. This provides the **descent** part of the name. The **recursive** property comes from how the parser is implemented, a collection of recursive procedures, where each recursive procedure represents a production as defined in the grammar in section 4.1. Thus for each non-terminal symbol in the grammar a procedure is defined which constructs a parse node from the stream of tokens.

```
| tb |
tb := TestBlock new.
STJTranscript cr; show: tb value printString.
```

**Figure 128.** Smalltalk example of code to be parsed

The code in Figure 128 will be used throughout this section as an example to illustrate the implementation of the adapted Squeak Smalltalk (`stj.Parser`) recursive descent parser.

A parse tree resulting from parsing the code in Figure 128 is shown in Figure 130. Each type of node (i.e. each type of ParseNode) illustrated in this parse tree will be discussed in the following subsections. The keys used throughout this section are shown in Figure 129.

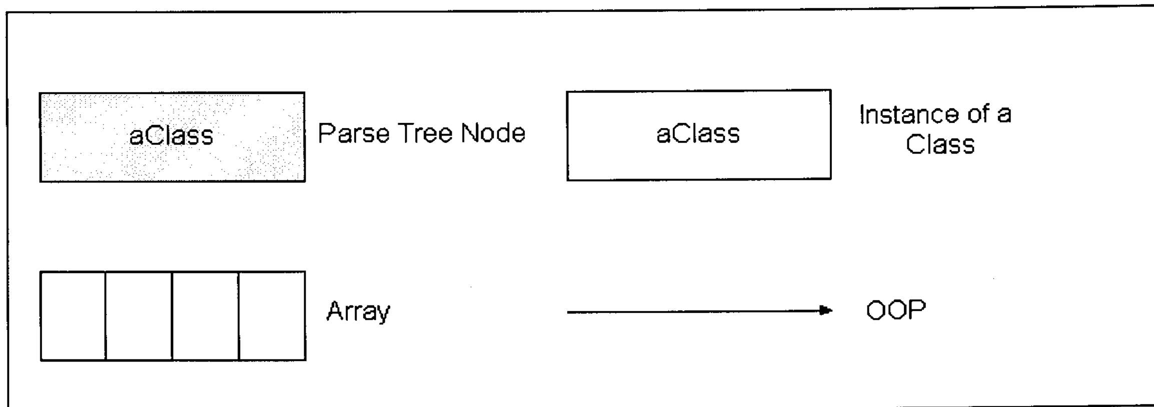


**Figure 129.** Key Nodes

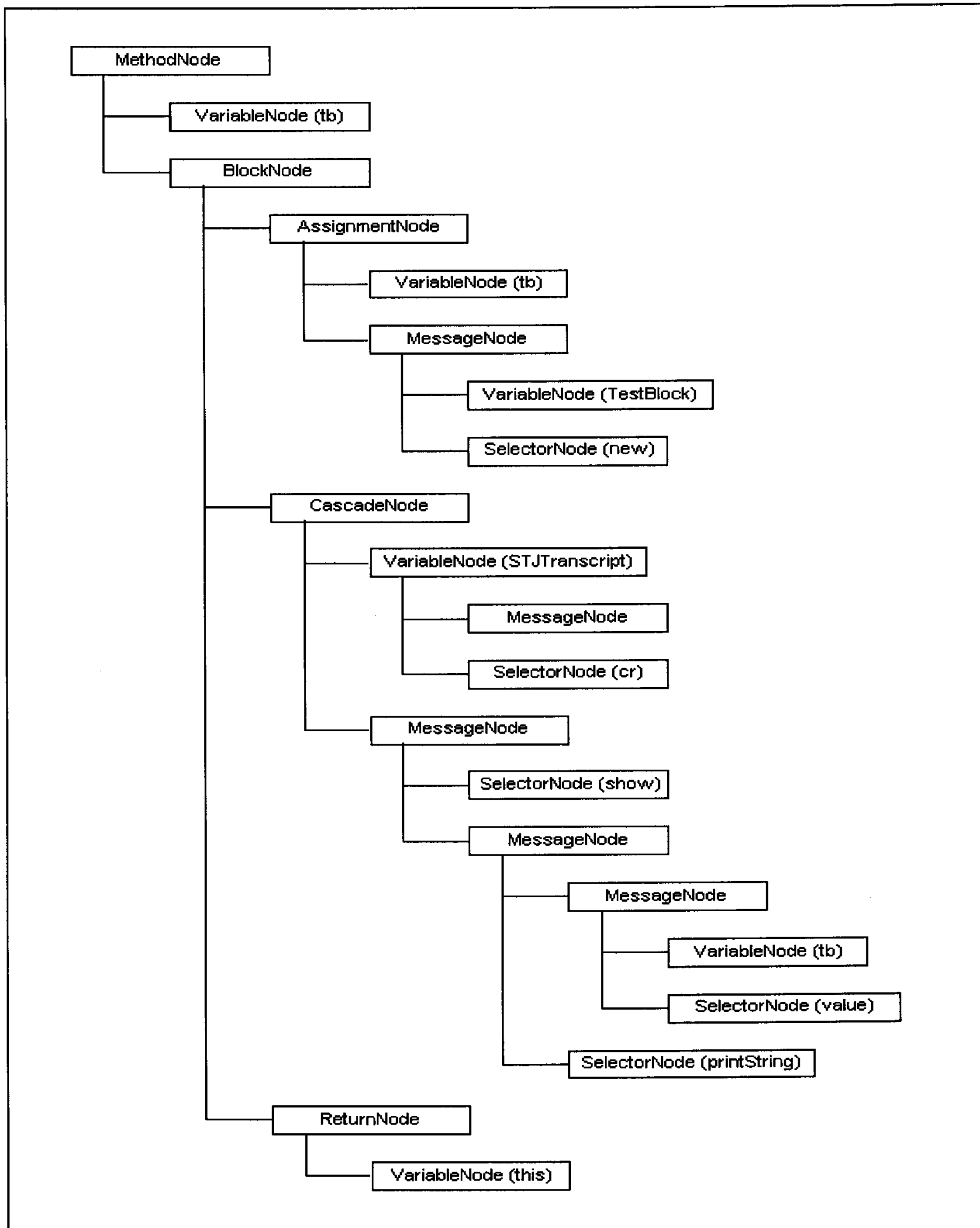


**Figure 130.** Parse tree of code in Figure 128

### 4.3.1. MethodNode

The first production of interest is `stj.Parser's` method named `method`. This method represents the production as defined in section 4.1.1. Because a method consists of a pattern, optional temporary variables and an optional list of statements (Squeak Smalltalk uses the block procedure for a list of statements); the method for parsing a method from a stream on `stj.Parser` follows the pattern seen in Figure 131.

```
stj.Parser>>method
   | selectorAndArgs temps block |
   selectorAndArgs := self pattern.
   temps := self temporaries.
   block := self block.
   ^stj.MethodNode new
      selector: (selectorAndArgs at: 1)
      arguments: (selectorAndArgs at: 2)
      temporaries: temps
      block: block.
```

**Figure 131.** Implementation of `method` on `stj.Parser`

In discussing nodes below, only the child node relationships will be illustrated. The other variables that belong to a Parse Node will not be shown. Thus in regard to `MethodNode`, `selector` and `argument` are not discussed further. The variables having child relationships are:

1) The list of temporaries that are part of the context of the method and

2) The block node with all the statements in the block

Figure 132 shows the list of temporaries and the block node.

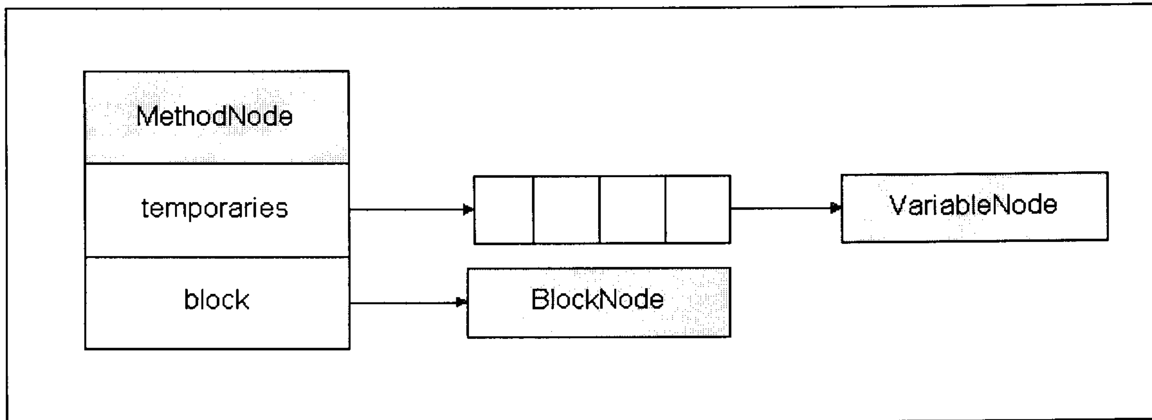


**Figure 132.** MethodNode

### 4.3.2. BlockNode

The next important production of `stj.Parser` is the method named `block`. A block can have an optional list of arguments, an optional list of temporary variables and an optional list of statements. The pattern in Figure 133 resembles the method for parsing a block from a stream on `stj.Parser`.

```
stj.Parser>>block
   | arguments temporaries statements |
   arguments := self blockArguments.
   temporaries := self temporaries.
   statements := self statements.
   ^stj.BlockNode new
      arguments: arguments
      temporaries: temporaries
      statements: statements.
```

**Figure 133.** Implementation of `block` on `stj.Parser`

Figure 134 illustrates the relationship between BlockNode and its children.

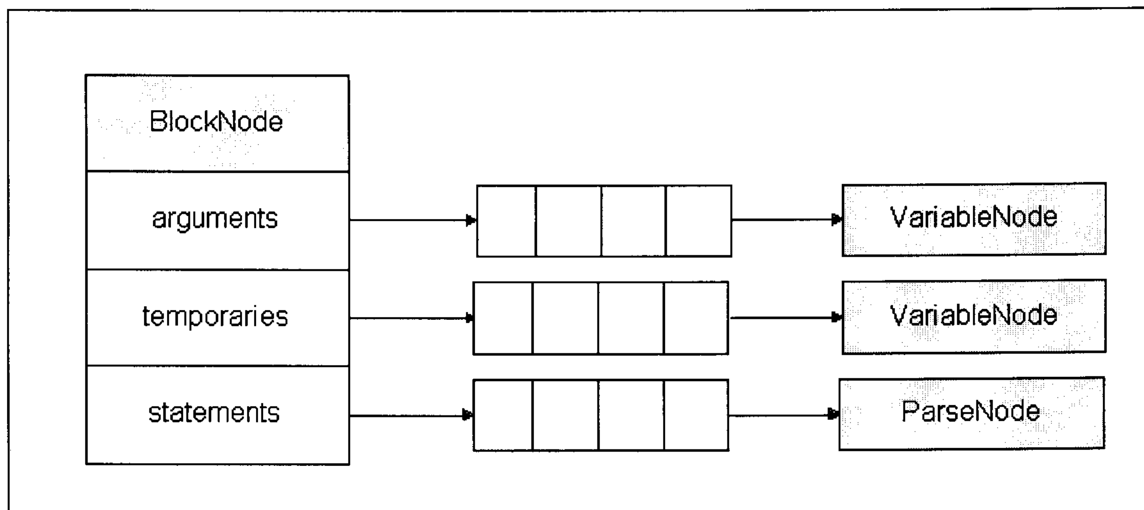


**Figure 134.** BlockNode

Note that the `statements` child is a list of ParseNodes and since ParseNode is the superclass of all nodes in the parser this means that any of the parser's node types can be in the statement list.

### 4.3.3. AssignmentNode

The next production is the method named `assignment`. It consists of a left hand side and a right hand side. The left hand side is the variable to which the result of the expression on the right hand side is assigned. The code in Figure 135 illustrates the `expression` production. It returns true if it does not encounter an expression, otherwise it returns one of three types of parse nodes; AssignmentNode, BraceNode or CascadeNode. The pattern for parsing the `assignment` production is shown in Figure 136.

```
stj.Parser>>expression
   (hereType == #word and: [tokenType == #leftArrow])
      ifTrue: [^self assignment: self variable].
   hereType == #leftBrace
      ifTrue: [self braceExpression]
      ifFalse: [self primaryExpression ifFalse: [^ false]].
   (self messagePart: 3 repeat: true)
      ifTrue: [hereType == #semicolon ifTrue: [self cascade]].
   ^ true
```

**Figure 135.** Implementation of `expression` on `stj.Parser`

```
stj.Parser>>assignment: varNode
   | parseNode |
   self assignmentToken.
   parseNode := self expression.
   ^stj.AssignmentNode new
      variable: varNode
      value: parseNode.
```

**Figure 136.** Implementation of `assignment` on `stj.Parser`

The illustration in Figure 137 shows the `variable` and `value` children. True to the production of an AssignmentNode the `value` child can be any type of parse node.

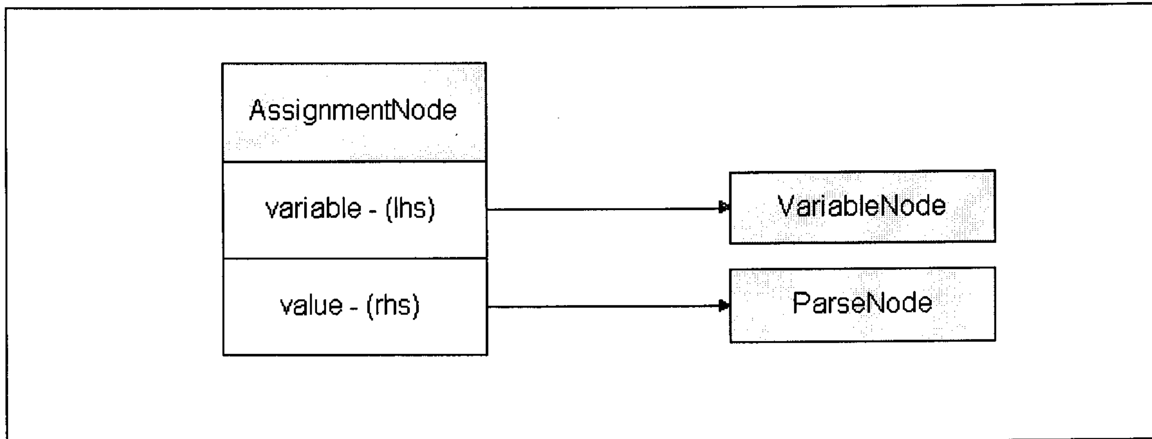


**Figure 137.** AssignmentNode

### 4.3.4. VariableNode

The production for a VariableNode, illustrated in Figure 138, is straightforward and involves keeping track of the variable name and other flags to indicate if it is a method argument or a block argument.

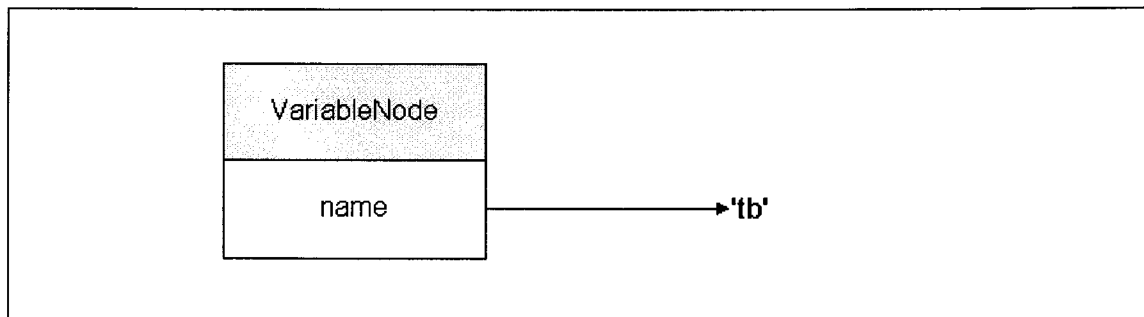


**Figure 138.** VariableNode

### 4.3.5. MessageNode

The next production is the method named `message` on `stj.Parser`. A MessageNode has to keep track of the receiver, the selector and arguments (if any). Because there are 3 types of messages the Parser needs to detect which type of message is being sent to the receiver.[13]

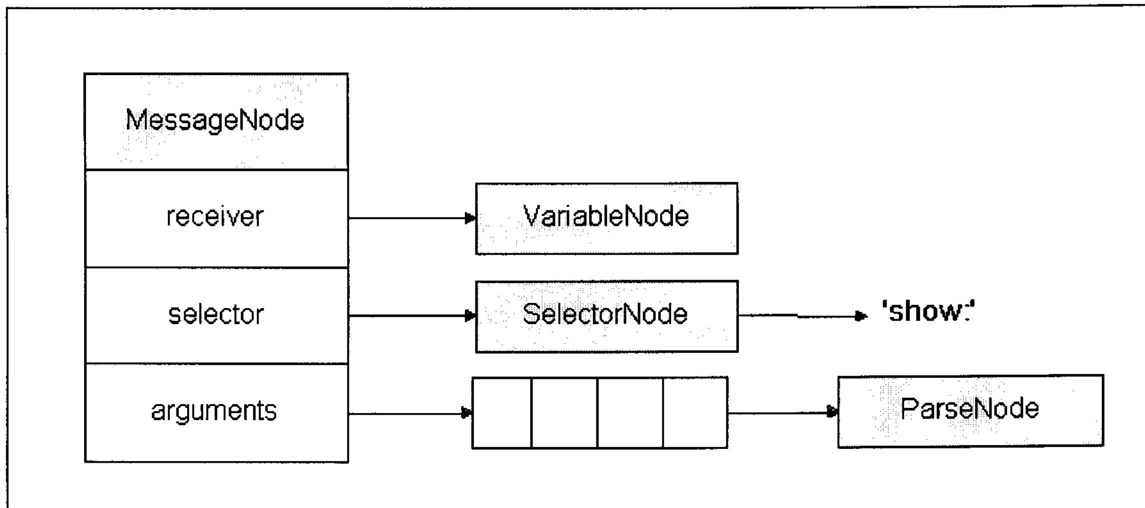


**Figure 139.** MessageNode

[13] Due to the long method list of message it is not repeated in here. The interested reader may refer to the source code on the enclosed CD for full details.

### 4.3.6. SelectorNode

A SelectorNode is very similar to a VariableNode as can be seen in Figure 140.

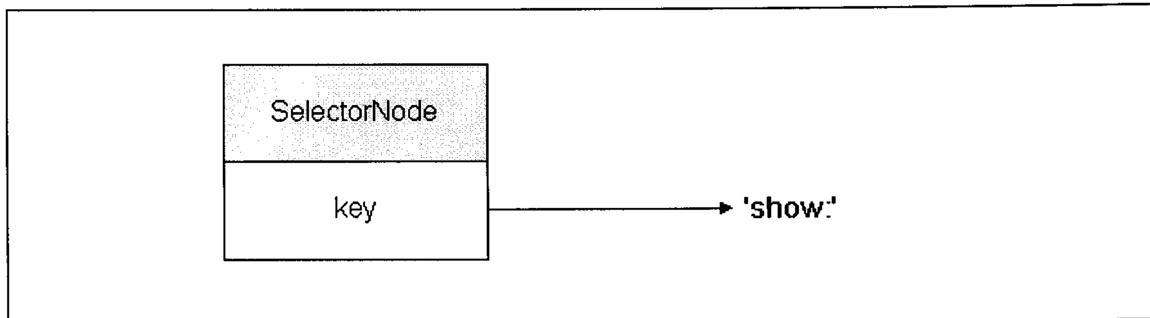


**Figure 140.** SelectorNode

### 4.3.7. CascadeNode

The production for the method named `cascade` on stj.Parser follows the pattern shown below in Figure 141. A CascadeNode has a receiver and a list of messages being sent to it. Figure 142 illustrates the simplicity of CascadeNode.

```
cascade
   | rcvr msgs |
   parseNode canCascade
      ifFalse: [^self expected: 'Cascading not'].
   rcvr := parseNode cascadeReceiver.
   msgs := stj.OrderedCollection with: parseNode.
   [self match: #semicolon]
      whileTrue:
         [parseNode := rcvr.
         (self messagePart: 3 repeat: false)
            ifFalse: [^self expected: 'Cascade'].
         parseNode canCascade
            ifFalse: [^self expected: '<- No special messages'].
         parseNode cascadeReceiver.
         msgs addLast: parseNode].
   ^stj.CascadeNode new
      receiver: rcvr
      messages: msgs
```

**Figure 141.** Implementation of `cascade` on `stj.Parser`

Figure 142 illustrates the simplicity of CascadeNode.

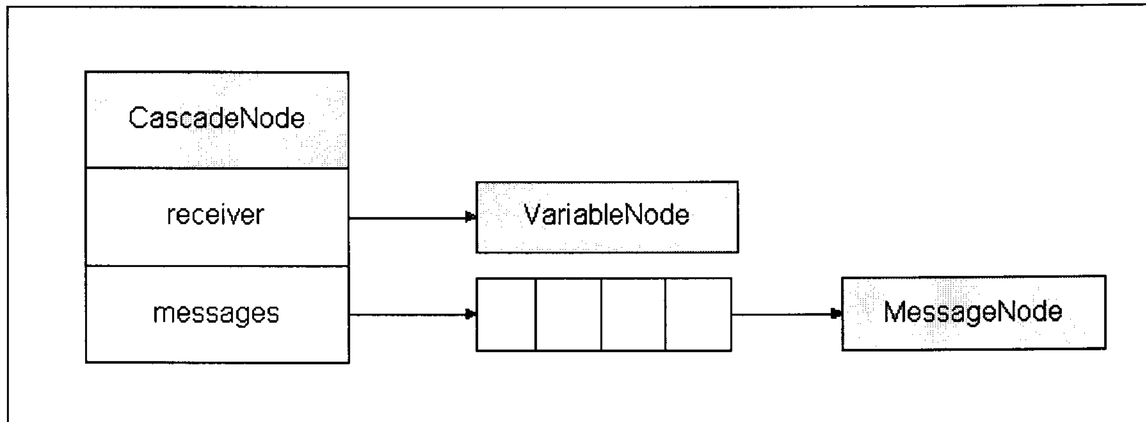


**Figure 142.** CascadeNode

## 4.4. Generation of Java code

In this section the encoding of the constructed parse nodes will be handled. Each parse node has values that indicate how the target Java code should be generated. The same parse nodes that were discussed in section 4.3 will be used. For the sake of brevity, only an extract of each encodeJava: method will be shown to illustrate the essence of the translation.

### 4.4.1. MethodNode

A MethodNode will write out all the temporary variables and assign null to them. The next step is to determine the number of blocks that will be created and then an array, __blocks[], will be created to refer to these blocks. The block child is encoded and then, depending whether there are blocks or not, the catch- and corresponding throw clauses are generated.

```
stj.MethodNode>>encodeJava: aStream
   | allBlocks allTempVars |
   temporaries do: [:temp |
      aStream nextPutAll: 'stj.Object '.
      temp encodeJava: aStream.
      aStream nextPutAll: ' = null;'; cr.
   ].
   aStream cr.
   allBlocks := block allBlocks.
   1 to: allBlocks size do: [:index | (allBlocks at: index)
blockNumber: (index-1)].
   allBlocks isEmpty ifFalse: [
      aStream nextPutAll: 'final stj.Object __blocks[] = new
stj.BlockContext[', allBlocks size printString, '];'; cr; cr.
      aStream nextPutAll: 'try'; cr.
      aStream tab.
      aStream nextPutAll: '{'; cr.
   ].
   block encodeJava: aStream.
   allBlocks isEmpty ifFalse: [
      aStream untab.
      aStream nextPutAll: '}'; cr.
      aStream nextPutAll: 'catch (BlockException e)'; cr.
      aStream tab.
      aStream nextPutAll: '{'; cr.
      1 to: allBlocks size do: [:index |
         aStream nextPutAll: 'if (((BlockException)e).block ==
__blocks[', (index-1) printString, ']) return
((BlockException)e).outcome;'; cr.
      ].
      aStream nextPutAll: 'throw e;'; cr.
      aStream untab.
      aStream nextPutAll: '}'; cr.
   ].
```

**Figure 143.** Implementation of encodeJava: on stj.MethodNode

### 4.4.2. Block Node

Every block in a method is assigned an index and assigned to an index in the `__blocks` array. The next step is to test if the block refers to variables defined outside its own context. If so a `__temp` array referring to these variables is passed as an argument to the block when creating it. The number of arguments is determined and the appropriate method signature is generated by the `valueMethod`. The body of the block with all the statements is generated next. The last step is to determine if the block has a non-local return and if so, an `stj.BlockException` is thrown. Otherwise a normal return is made.

```
encodeJava: aStream
   blockNumber notNil ifTrue: [aStream nextPutAll: '__blocks[',
blockNumber printString, '] = '].
   self needsTempVarArray
      ifTrue: [aStream nextPutAll: 'new stj.BlockContext(__temp)';cr.]
      ifFalse: [aStream nextPutAll: 'new stj.BlockContext()'; cr.].
   aStream tab; nextPut: ${; cr.
   aStream nextPutAll: self valueMethod; cr.
   aStream tab; nextPut: ${; cr.

   self encodeJavaForStatementsInBlock: aStream deleteReturnStatement:
returns topLevelBlock: false staticMethod: false.

   returns
      ifTrue: [aStream nextPutAll: 'throw new stj.BlockException(this,
__result);'; cr]
      ifFalse: [aStream nextPutAll: 'return(__result);'; cr].
   aStream nextPutAll: '} /* value() */'.
   aStream untab.
   aStream cr; nextPutAll: '}'.
   aStream untab.
```

**Figure 144.** Implementation of `encodeJava:` on `stj.BlockNode`

### 4.4.3. AssignmentNode

Generation of Java code for an `AssignmentNode` is straightforward and illustrates the recursive nature of this generation method. The `encodeJava:` message is sent to the `variable` child, the assignment operator is generated and the `encodeJava:` message is sent to the `value` child.

```
encodeJava: aStream
   self variable encodeJava: aStream.
   aStream nextPutAll: ' = '.
   self value encodeJava: aStream.
```

**Figure 145.** Implementation of `encodeJava:` on `stj.AssignmentNode`

### 4.4.4. VariableNode

A `VariableNode` has to first check if any of the reserved keywords are being referred to. If so some direct substitutions are made. The next step is to test if a class name is being used. If so the appropriate prefix and suffix is generated. In the case of a class variable, its class has to be determined as well and used in the substitution.

```
encodeJava: aStream
   | className class |
   name = 'self' ifTrue: [^aStream nextPutAll: 'this'.].
   name = 'nil' ifTrue: [^aStream nextPutAll: 'stj.Object.__nil'].
   name = 'true' ifTrue: [^aStream nextPutAll: 'stj.Object.__true'].
   name = 'false' ifTrue: [^aStream nextPutAll: 'stj.Object.__false'].
   "Are we dealing with a class name?"
   ((Smalltalk at: self javaName asSymbol ifAbsent: [])
      isKindOf: Class)
      ifTrue: [^aStream nextPutAll: 'metaclass.' ,
         self javaName , '.__class'].
   "Are we dealing with a class variable?"
   class := STJTranslator classCurrentlyTranslated.
   className := class classNameOfClassVariableDefinition: name.
   className notNil
      ifTrue: [aStream nextPutAll: '((metaclass.' , className , ')
         (metaclass.' , className , '.__class)).' , self javaName]
      ifFalse: [aStream nextPutAll: self javaName].
```

**Figure 146.** Implementation of `encodeJava:` on `stj.VariableNode`

### 4.4.5. MessageNode

A `MessageNode` has two main parts, the receiver and the selector. The receiver is sent the `encodeJava:` message. The selector is sent `encodeJava:` as well and then the arguments need to be generated. For each argument the `encodeJava:` message is sent as well.

```
encodeJava: aStream
   | tempArgs firstArg |
   self receiver notNil ifTrue:
      [self receiver encodeJava: aStream.
      aStream nextPutAll: '.'.].
   self selector encodeJava: aStream.
   aStream nextPutAll: '('.
   tempArgs := self arguments asOrderedCollection.
   (tempArgs size >= 1) ifTrue: [
      firstArg := tempArgs at: 1.
      tempArgs removeFirst.
      firstArg encodeJava: aStream.
   ].
   tempArgs do: [:arg | aStream cr; nextPutAll: ', '.
      arg encodeJava: aStream].
   aStream nextPutAll: ')'.
```

**Figure 147.** Implementation of `encodeJava:` on `stj.MessageNode`

### 4.4.6. SelectorNode

A simple check for the ',' message is done and substituted with 'commaConcatenate' and then the asJavaSelector method invoked.

```
encodeJava: aStream
   (key = #',') ifTrue: [key := 'commaConcatenate' asSymbol].
   aStream nextPutAll: key asString asJavaSelector.
```

**Figure 148.** Implementation of encodeJava: on stj.SelectorNode

### 4.4.7. CascadeNode

A CascadeNode can be translated into Java by cascading the method invocations as well. Thus the receiver is sent the encodeJava: message and then each message in the messages child is sent the encodeJava: message after generating a '.' before the message.

```
encodeJava: aStream
   receiver encodeJava: aStream.
   messages do:
      [:message |
      aStream nextPutAll: '.'.
      message encodeJava: aStream.].
```

**Figure 149.** Implementation of encodeJava: on stj.CascadeNode

## 4.5. Other Target Languages

It is quite possible to reuse most of the existing framework in the translator to produce a translator for another language. This can be achieved by replacing the `encodeJava:` sections on the nodes. The STJTranslator class will also have to be replaced by a new translator focussed on the new target language.

One target language that seems well-suited is the Python language. Python is similar to Smalltalk in that:

1) it has reflection,
2) it has a construct that is as powerful as Smalltalk blocks,
3) it translates to byte codes that executes on a virtual machine,
4) it is a dynamically typed language.

However, a full discussion of the merits or otherwise of various other possible target languages is beyond the scope of this dissertation. Instead, attention is now turned to other work done on Smalltalk to Java translation, as well as to future directions that could be taken by the present research.

# 5. Conclusion

## 5.1. Related Work

Due to the similarities of Java and Smalltalk other initiatives are busy with implementing Smalltalk to Java translators. (See Boyd (2000) and Mission Software (2002).) In this section the two environments known as Bistro and Smalltalk/JVM are introduced with a short overview on each. It is followed by a comparison between STJ (the current STJ Translator), Bistro and Smalltalk/JVM. The reason for choosing to discuss these two environments is that, as far as can be ascertained, they seem to be the most significant Smalltalk to Java endeavours to date. They have also apparently been developed contemporaneously and with the work discussed in this dissertation. Each turns out to have some similarities with STJ but there are also some differences.

### 5.1.1. Bistro

Bistro is a derivative of Smalltalk and although it does not conform to the ANSI Smalltalk specification it is still useful to compare against STJ. Bistro introduces types to Smalltalk and allows the developer to specify a type if needed for improved performance. In this respect it is similar to Typed Smalltalk by Johnson, Graver and Zurawski (1988).

Bistro allows the development of Java Servlets[14] and tight integration between Bistro source code and Java source code. Bistro allows the invocation of Java methods. Java code is generated from Bistro source code and the Java code must be compiled in a similar fashion to STJ.

At the time of writing the author confirmed a problem with Bistro translation's of Blocks that will be discussed in section 5.1.5.

### 5.1.2. Smalltalk/JVM

Smalltalk/JVM is a product developed by Mission Software (2002). It follows the Smalltalk paradigm of presenting the developer with class browsers, workspaces and a Transcript.

[14] A Java Servlet is a class in the Java Servlet Development Kit and allows a developer to develop Java code on the server that generates HTML pages for Web browsers.

Smalltalk/JVM compiles from a Smalltalk source file directly to a Java class file[15]. It has a comprehensive Smalltalk class library with the product. An inspector window is also implemented to inspect object values during runtime.

Other features include the ability for Smalltalk/JVM to call Java methods from Smalltalk methods and to call Smalltalk methods from Java methods. It is also possible for a Java class to inherit from a Smalltalk class and a Smalltalk class can inherit from a Java class. An example in the evaluation copy of Smalltalk/JVM is included to show how Smalltalk/JVM implements a Java Servlet in Smalltalk. Another example supplied with Smalltalk/JVM illustrates the tight integration of Smalltalk/JVM with Java and shows how to access the JDBC API from Smalltalk.

At the time of writing the author confirmed that, unlike STJ, Smalltalk/JVM does not handle dynamic Smalltalk metaclasses according to the Smalltalk standard.

### 5.1.3. Talks2

The third product, Talks2, has similar features to Smalltalk/JVM. The difference is that the Smalltalk and development environments run on top of the Java Virtual Machine, whereas Smalltalk/JVM is implemented in Smalltalk. The Architur company that is developing the product is currently attempting to have it translate the Squeak Smalltalk source code to Java. However, they admit to various limitations (See Architur (2002)). At the time of writing the product was in an alpha release and could not be used for comparative purposes in section 5.1.5.

### 5.1.4. Smalltalk/X

Another interesting product is Smalltalk/X by Gittinger (2002). It involves adapting the Smalltalk VM used by Smalltalk/X to interpret JBC as well. The result is that the VM can execute Smalltalk and Java code at the same time. The Java class hierarchy inherits from the Smalltalk Object class. A similar approach (in the opposite direction) is used to that used by STJ whereby translated Smalltalk classes inherit from the Java Object class.

---

[15] It seems possible to implement the same functionality in STJ simply by invoking the `javac` compiler directly after the translation process.

### 5.1.5. Comparison with STJ

#### 5.1.5.1. Method binding

Section 3.3 discusses the two methods explored for STJ. Bistro chose to use the reflection method, section 3.3.2, of translating most method calls into a `perform_with()` call. To achieve an acceptable level of performance, frequently used methods inherited from `Object` were invoked normally. Smalltalk/JVM opted to use the faster approach to invoking methods that was discussed in section 3.3.3, and which is used in STJ. In Smalltalk/JVM the `Object` class inherits from a class named `Any` and all new methods to be invoked are generated and added to the `Any` class.

#### 5.1.5.2. Dynamic Smalltalk metaclasses

Section 3.4 discusses how STJ approached the translation of Smalltalk classes by simulating metaclasses. The examples in Figure 56 and Figure 60 were used to test the capabilities of Bistro and Smalltalk/JVM. Bistro passed the test and managed to create a Vehicle and BMW instance and initialise them correctly. Smalltalk/JVM created a Vehicle instance, but could not create a BMW instance and invoke the correct initialise method.

#### 5.1.5.3. Simple block

Both Bistro and Smalltalk/JVM created and evaluated a block successfully. The example in Figure 85, section 3.5.1, was used. Bistro creates a different Block class, `ZeroArgumentBlock`, `OneArgumentBlock` and `TwoArgumentBlock` depending on the number of arguments passed into a block. Smalltalk/JVM translated directly to `class` files.

#### 5.1.5.4. Block with references to variables

The example in Figure 88, section 3.5.2, was used and translated. Bistro translated it and executed it successfully returning `11` as output. Smalltalk/JVM translated it, but the output was given as `16`. Further investigation is difficult, due to the direct translation to class files. However, the difference could perhaps be attributed to a simple problem like different precedence between '`*`' and '`+`'.

### 5.1.5.5. Block with block arguments

The example in Figure 92, section 3.5.3, was used and both Bistro and Smalltalk/JVM translated and executed it successfully. As noted before, Bistro makes use of different Block classes, depending on the number of arguments passed into a block.

### 5.1.5.6. Block contexts

A more difficult task is to translate a block within its enclosing context, and maintain the same behaviour as in Smalltalk. The example in Figure 94, section 3.5.4, was used. Bistro translated it successfully, but could not execute it correctly. Inspection of the translated Java code that Bistro generates, confirmed that the blocks being generated cannot refer to variables in the enclosing context while executing somewhere else in the program.

STJ solves this problem by using an array for all variables referred to in the block, declared outside the block, and passes this array to the block as an argument, effectively passing the enclosing context to the block.

Smalltalk/JVM translates and executes this example correctly.

### 5.1.5.7. Blocks with non-local returns

Another difficult translation to implement is when a block returns out of its enclosing context, by virtue of encountering a non-local return when it is evaluated. The example in Figure 99, section 3.5.5, was used. In STJ the approach taken was by throwing `BlockExceptions`, holding on to the block result.

The same approach was taken by Bistro. An exception, `MethodExit`, was defined and thrown whenever a non-local return was needed. Bistro was clever enough to subclass `MethodExit` from `RuntimeException` in Java, which prevents the method signatures from changing in the translated Java code.

The Smalltalk/JVM translator and environment executed the example correctly.

### 5.1.5.8. Nested blocks

The nested blocks example in Figure 105, section 3.5.6, was used to ensure that the translators could handle more than one level of block creation. Both Bistro and Smalltalk/JVM handled it successfully.

## 5.2. Future work

### 5.2.1. Smalltalk features

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 this will require existing class instances to be migrated to the new version of the class and ensuring that all references to the old class instances are updated to reference the newer version.

The `become:` method in Smalltalk allows the exchange of references to objects in the virtual machine. This is quite a useful feature and will ease the implementation of dynamically loading and unloading Java classes.

Another issue is that the translator does not support the identification of the classes and methods required to execute an application at runtime. The classes to be translated are specified by the user and all methods in the specified classes are translated.

Smalltalk has another feature on blocks to enable multi-threading and multiple processes. By sending the `fork` message to a block it executes in its own process. It is thus possible to assign a block (process) to a variable and treat it as a separate process. (See Goldberg and Robinson (1989, p250-p266).) More investigation will be necessary to support this feature in the translation of Smalltalk to Java code in STJ. Bistro has a few examples of Bistro code with multiple processes being translated to Java threads.

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.

### 5.2.2. Translating the Translator

It would be an interesting exercise to translate the implemented Smalltalk by using the translator on itself. The generated Java code (output 1) will then have the same functionality of the Smalltalk translator and would also be able to translate the Smalltalk translator and generate Java code again (output 2). By showing that the output of step 1 is equal to step 2 one could verify the generated Java translator code to be functionally equivalent to the original Smalltalk source code. This exercise was not possible with the current version of STJ. In order to carry out the experiment, it would be necessary to provide STJ not only with its own Smalltalk source code,

but also with the Smalltalk code for all the library classes that were used in its implementation. This was considered to be beyond the scope of the present study.

The Squeak Smalltalk system was constructed by a similar process known as bootstrapping. Refer to Ingalls, Kaehler and Kay (1997) for a full discussion on how this was achieved in Squeak Smalltalk.

## 5.3. Summary

It should be noted that the work described in this dissertation was started several years ago and was carried out on a part-time basis. In the intervening period, a number of other initiatives to translate Smalltalk to Java have come to light.

Section 5.1 examined the two most prominent initiatives. Examples of Smalltalk code were tested on these respective translators. The examples are from chapter 3, which had already pointed out how STJ achieves an appropriate translation in each case. An *ex post facto* finding was that in some cases, these initiatives followed similar translation strategies to STJ. However, it was also found that the rival translators do not always fully conform to standard Smalltalk semantics.

On the other hand, section 5.2.1 identified areas where translation is possible, but which have not been tackled by STJ. Indeed, some of the areas mentioned (e.g. blocks to enable multi-threading) are dealt with by rival translators. The same section points to at least one area where translation appears to be infeasible (namely, the `become:` method).

In one sense, then, STJ and the other translation systems discussed, provide the translational semantics of various Smalltalk constructs in Java. In doing so, the similarities and differences between the two languages are highlighted. 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. In other cases, such as translating blocks, the translation rules become far more complex. Nevertheless, while a comprehensive Smalltalk to Java translation system seems unlikely, this work suggests that it is be possible to obtain reasonably efficient JBC versions of most Smalltalk programs. By providing such translators, is becomes possible to extend the range of platforms on which legacy Smalltalk programs may be run.

# 6. References

Abelson, H., Sussman, G.J., Sussman, J. 1996. *Structure and Interpretation of Computer Programs*, McGraw-Hill, 1996.

Architur, 2002. *Talks2*, [Online]. Available: http://www.architur.de/talks2/

AppletMagic, 2002. *AppletMagic*, [Online]. Available: http://www.appletmagic.com

Beck, K., 1999. *Kent Beck's Guide to Better Smalltalk: A Sorted Collection*, Cambridge University Press, 1999.

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

Bothner, P. 1996. *Translating Smalltalk to Java*, [Online]. Available: http://www.cygnus.com/~bothner/smalltalk.html

Bothner, P. Hardwick, and Sipelstein. 2000. *The Kawa Scheme System*, [Online]. Available: http://www.cygnus.com/~bothner/kawa.html

Boyd, N. 2000. *Bistro – Integrating Smalltalk and Java*, [Online]. Available: http://www.jps.net/nikboyd/papers/bistro/index.htm

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

Chambers, C. 1992. *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

Engelbrecht, R. Kourie, D. 1998. *Issues in Translating Smalltalk to Java*, International Conference on Compiler Construction (CC) 1998, Springer-Verlag

Flanagan, D. 1997. *Java in a Nutshell*, O'Reilly, 2nd edition

Fussel, M.L. *Java and Smalltalk syntax compared*, 2002. [Online]. Available: http://www.chimu.com/publications/JavaSmalltalkSyntax.html

Fussel, M.L. *SmallJava*, 2002. [Online]. Available: http://www.chimu.com/publications/SmallJava/index.html

Gittinger, C. 2002. *Smalltalk/X*, [Online]. Available: http://www.exept.de

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

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

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

Hardwick, J.C., Sipelstein, J. 1996. *Java as an intermediate language*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Aug. 1996.

Hunt, A., Thomas, D. 2000. *The Pragmatic Programmer*, Addison-Wesley Longman Inc., 2000.

Ingalls, D., Kaehler, T., Kay, A. et al. *Back to the Future, The Story of Squeak, A Practical Smalltalk written in itself*, Proceedings of OOPSLA 97, 1997.

JavaSoft, 1997. *The Java Core Reflection API and Specification*, [Online]. Available: http://java.sun.com, Jan. 1997.

Johnson, R., Graver, J., Zurawski, L. 1988. *TS: an optimizing compiler for Smalltalk*, Proceedings of OOPSLA 88, 1988.

JPython, 2002. *JPython*, [Online]. Available: http://www.jpython.org

Kay, A.C. 1996. *The early history of Smalltalk*, Bergin, T.J. & Gibson, R.G., *History of Programming Languages*, **Vol. 2**, Addison-Wesley

Lalonde, W., Pugh, J. 1990. *Inside Smalltalk*, **Vol. 1**, Prentice-Hall Inc., 1990.

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

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

Meyer, B., 1997. *Object-Oriented Software Construction*, Prentice Hall, 1997.

Meyer, J., Downing, T. 1997. *Java Virtual Machine*, O'Reilly, 1997.

Miller, R., Tripathi, A. 2001. *Issues with Exception Handling in Object-Oriented Systems*, Proceedings of ECOOP '97, Finland, 1997.

Mission Software, 2002. *Smalltalk/JVM*, [Online]. Available: http://www.missionsoft.com

Misty Beach Software. 2002. *Misty Beach Forth*, [Online]. Available: http://www.mistybeach.com/Forth/JavaForth.html

Odersky, M., Wadler, P. 1997. *Pizza into Java: Translating theory into practice*, Proceedings of the 14th ACM Symposium on Principles of Programming Languages, France, Jan. 1997.

Ogasawara, T., Komatsu, H., Nakatani, T. 2001. *A Study of Exception Handling and its Dynamic Optimization in Java*, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Oct 2001.

Pinson, L., Wiener, R. 1988. *An Introduction of Object-Oriented Programming and Smalltalk*, Addison-Wesley, 1988.

Piumarta, I.K. 1992. *Delayed Code Generation in a Smalltalk-80 Compiler*, Ph.D. Thesis, Department of Computer Science, University of Manchester, Oct. 1992.

SunWorld, 1995. *Java: The inside story*, [Online]. Available: http://sunsite.cs.msu.su/sunworldonline/swol-07-1995/swol-07-java.html

Synkronix, 2002. *LegacyJ*, [Online]. Available: http://www.synkronix.com

Terekhov, A.A., Verhoef C. 2000. *The realities of language conversions*, IEEE Software, pp. 111-124, November 2000.

Tilevich, I. 2000. *Translating C++ to Java*, First German Java Developers' Conference Journal. [Online]. Available: http://sol.pace.edu/~tilevich/c2j.html

Tolksdorf, R. 2002. *Languages for the Java VM*, [Online]. Available: http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

Waddington, T. 2002. *Java Backend for GCC*, [Online]. http://archive.csee.uq.edu.au/~csmweb/ubqt.html#gcc-jvm

Wirth, N. 1996. *Compiler Construction*, Addison-Wesley, 1996.

Yasumatsu, K. 1996. *A translation method from Smalltalk into Interoperable C Code*, Ph.D. Thesis, Graduate School of Engineering Science, Osaka University, 1996.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Yasumatsu, K., Doi, N. 1995. *SPiCE: A System for Translating Smalltalk Programs into a C Environment*, IEEE Transactions on Software Engineering 21(11), pp. 902-912, 1995.