UNIVERSITY OF OSLO
Department of informatics

The SICS Java Port Project
-automatic translation of a large system from Smalltalk to Java

Master thesis
60 credits

Martin Skarsaune

01. May 2008
Abstract

The objective of the SICS Java Port Project is to create tools and methodology to translate a large financial software application from IBM VisualAge for Smalltalk to Java. My main motivation for this report is to present some of the major technical issues we have encountered as part of translation, present alternatives and the solution we chose. I would like to share some of the knowledge the project group has accumulated over the last four years.

Each problem will be briefly demonstrated, followed by a short discussion and the solution we ended up with and a demonstration of the results.

In our concrete project, the approach has been highly successful. We have found ways to overcome the most important language differences. The few problems that we have not been able or willing to solve have fortunately been limited to a manageable number of occurrences.

Although it is impossible to directly apply the experiences of this project to other Smalltalk systems, the results should be promising for others with a similar challenge at hand. Hopefully the reader will find the discussions relevant, and if not immediately reusable, at least serve as inspiration for developing custom solutions of their own.
**Foreword**

As I am writing this, we are more than halfway through a complete version test of the first ever Java version of the SICS application. The completion will mark the end of more than four years of hard work. From time to time the similarity to an exploration mission has struck me, with amazing discoveries along the way, never knowing exactly what lies around the next corner.

Although I get the honour of presenting some of the most interesting results of the SICS Java port project, the project has all the way been a real team effort. What I present is therefore collectively gathered knowledge. Throughout various stages of the project, the project room in Oslo has been filled with lively debate among the members of the core team on how to tackle different issues. In addition to myself the core team exists of Harald Sverdrup-Thygeson, Ian Meikle, Erling Skard, Øystein Eckhoff and Andi Thomas. In addition there have been numerous valuable contributions from other colleagues in Oslo, Italy and India.

Also many thanks to CSC’s (Computer Sciences Corporation) Reinsurance division for the willingness to go ahead with such a daring project, which has provided fascinating professional challenges for the ones involved, but undoubtedly a few headaches and sleepless nights for the managers responsible. First and foremost this includes Bjørnar Evenshaug and Yngvar Zahl, who have shielded the project group from external concerns and secured sufficient support and dedication for the project both internally and among the clients. Thanks for your never failing belief in the ability of the project group.

I also owe a great deal of gratitude to Professor Birger Møller-Pedersen at the University of Oslo for helping me to present my thoughts in a form that is hopefully comprehensible for others. Not an easy task!

To my loving wife Baili
## Table of contents

1  Introduction ........................................................................................................ 6
  1.1  Background .................................................................................................. 6
  1.2  The programming languages and runtime environments ............................ 6
  1.3  Previous research ....................................................................................... 10
  1.4  Terminology and notation .......................................................................... 11
  1.5  Methodology ............................................................................................... 12
  1.6  Parse tree transformation mappings ............................................................ 13
2  Basic Code Structure ......................................................................................... 17
  2.1  Classes ....................................................................................................... 17
  2.2  Instance variables ..................................................................................... 17
  2.3  Methods ..................................................................................................... 18
  2.4  Differences in the Object class .................................................................... 20
  2.5  Meta classes ............................................................................................... 22
3  Type system ....................................................................................................... 32
  3.1  Terms ......................................................................................................... 32
  3.2  Static Types ............................................................................................... 32
  3.3  Meta Types ................................................................................................ 33
4  Typing Issues ...................................................................................................... 36
  4.1  Primitive Types .......................................................................................... 36
  4.2  Different inheritance hierarchies in Smalltalk and Java ............................. 37
  4.3  Parameter types in overridden methods ....................................................... 40
  4.4  Multi Types ................................................................................................ 40
  4.5  Protocol supported by several classes across hierarchies ......................... 43
  4.6  Return type problems ................................................................................. 47
  4.7  Casting ........................................................................................................ 49
  4.8  Java Generics Syntax .................................................................................. 50
  4.9  Argument Types ......................................................................................... 55
5  Conditional expressions .................................................................................... 56
  5.1  Return ......................................................................................................... 57
  5.2  Assignment ................................................................................................ 58
  5.3  Expression context ..................................................................................... 59
  5.4  Void Context .............................................................................................. 62
  5.5  ifTrue:/ifFalse: in non void contexts ......................................................... 63
  5.6  Shortcut Boolean evaluation ..................................................................... 64
  5.7  Conditional loops ...................................................................................... 64
  5.8  Iteration loops ............................................................................................ 65
6  Cascaded message expressions ....................................................................... 66
7  Array Indexes .................................................................................................... 68
8  Blocks ................................................................................................................ 71
  8.1  Previous research ....................................................................................... 71
  8.2  Potential future support for closures in Java .............................................. 71
  8.3  Support for blocks with anonymous Java inner classes ........................... 72
  8.4  Static scope references .............................................................................. 72
  8.5  Non-local return ........................................................................................ 74
  8.6  In-lining blocks to optimize code ............................................................... 76
  8.7  References to self and super in Blocks ...................................................... 78
9  Numbers ............................................................................................................ 79
  9.1  Use of numbers in SICS and Smalltalk ....................................................... 79
  9.2  Number types in SICS ............................................................................... 79
  9.3  Computations and overflow ...................................................................... 80
  9.4  Project experiences ................................................................................... 82
10 Lacking features in Java .................................................................................. 83
  10.1  Mapping to corresponding method in Java .............................................. 83
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2</td>
<td>Mapping to utility method</td>
<td>83</td>
</tr>
<tr>
<td>10.3</td>
<td>In-lining behaviour</td>
<td>84</td>
</tr>
<tr>
<td>11</td>
<td>Class extensions</td>
<td>85</td>
</tr>
<tr>
<td>11.1</td>
<td>Some background</td>
<td>85</td>
</tr>
<tr>
<td>11.2</td>
<td>The class extension mechanism</td>
<td>85</td>
</tr>
<tr>
<td>11.3</td>
<td>Class extensions in Java</td>
<td>86</td>
</tr>
<tr>
<td>11.4</td>
<td>Base class extensions</td>
<td>86</td>
</tr>
<tr>
<td>12</td>
<td>UndefinedObject</td>
<td>87</td>
</tr>
<tr>
<td>13</td>
<td>Problematic Smalltalk methods</td>
<td>89</td>
</tr>
<tr>
<td>13.1</td>
<td>#doesNotUnderstand: redefinitions</td>
<td>89</td>
</tr>
<tr>
<td>13.2</td>
<td>Reference introspection and modification</td>
<td>90</td>
</tr>
<tr>
<td>13.3</td>
<td>Debugger methods</td>
<td>92</td>
</tr>
<tr>
<td>14</td>
<td>Exception handling</td>
<td>93</td>
</tr>
<tr>
<td>14.1</td>
<td>#signal and #signalWith</td>
<td>93</td>
</tr>
<tr>
<td>14.2</td>
<td>ExceptionalEvent hierarchy</td>
<td>93</td>
</tr>
<tr>
<td>14.3</td>
<td>Java checked exceptions</td>
<td>93</td>
</tr>
<tr>
<td>14.4</td>
<td>#when:do:</td>
<td>93</td>
</tr>
<tr>
<td>14.5</td>
<td>#ensure: and #atEndOrWhenExceptionDo:</td>
<td>95</td>
</tr>
<tr>
<td>14.6</td>
<td>Other unsupported exception semantics</td>
<td>95</td>
</tr>
<tr>
<td>14.7</td>
<td>#retry</td>
<td>97</td>
</tr>
<tr>
<td>15</td>
<td>Runtime Issues</td>
<td>99</td>
</tr>
<tr>
<td>15.1</td>
<td>Reflection</td>
<td>99</td>
</tr>
<tr>
<td>15.2</td>
<td>Classname resolving</td>
<td>100</td>
</tr>
<tr>
<td>15.3</td>
<td>Class loading and memory consumption</td>
<td>101</td>
</tr>
<tr>
<td>16</td>
<td>Summary</td>
<td>102</td>
</tr>
<tr>
<td>17</td>
<td>Conclusion</td>
<td>104</td>
</tr>
<tr>
<td>Appendix I</td>
<td>References</td>
<td>105</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background

This work is based on the knowledge gained in the SICS Java Port project, conducted at CSC’s Reinsurance R&D Division, located in Oslo Norway. Although most information should be relevant in other contexts, some background information about the actual undertaking might be of interest.

1.1.1 The system

SICS/nt is the world’s leading system for reinsurance administration with clients on all continents. SICS/nt is the clients main administrative system. Among the customers are some of the world’s largest reinsurance companies, such as Swiss Re and Allianz Re. The customers use SICS/nt to administrate all aspects of their business: Treaty management, claims and accounting.

SICS/nt comes in several versions: SICS/nt P&C for property and casualty reinsurance, SICS/nt Life for life reinsurance and SICS Cede for ceded reinsurance. SICS was originally developed by Storebrand in mainframe COBOL. The COBOL version is normally referred to as SICS Classic, and still in use by some clients. The system was completely rewritten as an interactive desktop application from 1996-1999, this time in IBM VisualAge for Smalltalk. CSC became owner of the product through the acquisition of Continuum in 1996.

Since release 3.0 (2003) the server versions have been available for the two products, allowing application integration with Web Services. Oslo is the main location for SICS’ research and development organization. SICS/Server shares code base with SICS/nt. The code base is huge with close to 20,000 classes, more than 220,000 methods and more than 700 database tables. IBM has described SICS as “to their knowledge - the largest software application ever built in IBM Smalltalk”\(^1\).

1.1.2 The project

The SICS4 Java Port project aims to create the 3\(^{rd}\) generation of SICS, this time on the Java platform. The translation is carried out from IBM VisualAge for Smalltalk version 6.0 to Java source code compatible with Java version 5 and later. The migration will be carried out by translation tools developed in-house as part of the SICS4 Java Port project. The rationale for choosing the Java platform will not be discussed in this report. The enormous code base rules out translation by hand.

1.2 The programming languages and runtime environments

1.2.1 Smalltalk

The Smalltalk language will be presented in most detail as we assume the reader is most familiar with Java. For readers unfamiliar with Smalltalk, (Kay 1996) tells the fascinating tale of a highly innovative era in computer science, which among other things gave birth to the Smalltalk language. For Kay’s group at the XEROX Palo Alto Research Center, aka. PARC, Smalltalk was but one component in their vision to apply information technology in daily life. At the same time the predecessors to personal computers, overlapping window user interfaces, Ethernet and laser printing were developed, so it should be of great interest to people in the computer industry to learn about their work. The group’s visions were truly ahead of their time, and in retrospect

\(^1\) [http://www.csc.com/industries/insurance/mds/mds222/](http://www.csc.com/industries/insurance/mds/mds222/)

\(^2\) Quote by IBM Smalltalk lab representative during CSC visit to IBM’s Smalltalk development unit in Raleigh, North Carolina, 1999
one may see that as other great innovations they have truly served to shape information technology as we know it today.

The Smalltalk universe continues to fascinate professionals and academics alike, though the software industry has largely turned to its successors Java and C# for commercial application. I will briefly attempt to describe and illustrate some profound characteristics of the language.

Object oriented
According to (Kay 1996), the original motivation for designing the language was a bet Kay made to define a powerful programming language in “a page”. In order to achieve this one needs a mechanism general and powerful enough to cover all the needs of a programming language. As an illustration of this there are only 5 reserved words in Smalltalk. Smalltalk demonstrates that all required constructs in a programming language may be expressed as objects and messages: “Smalltalk’s design--and existence--is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.” (Kay 1996), page 512. By decomposing everything into these basic building blocks, the language itself is extremely simple, elegant, consistent, powerful, flexible and extendable.

From a runtime point of view, all components of the system are objects, and everything “happens” by sending messages to objects. A small example to demonstrate this:

```smalltalk
^self hasParent ifTrue: [parent root] ifFalse: [self].
```

Example 1 Smalltalk if-else statement

Example 1 shows the implementation of a method called root. We will manually execute the method step by step.

First a few words about so-called messages:
Each message must be specified by a name or message selector in Smalltalk terminology. Selector indicates that the message name is used at runtime to select a method among the methods defined in the hierarchy. For convenience, symbols are used for method selectors. A Symbol is a unique string instance in the global symbol table. Symbol literals in code are given by the string value prefixed by a hash character (#). It is therefore common to use symbol notations for message selectors. Message selectors are divided in 3 categories:

- Unary selectors may only be used for messages without arguments. Unary selectors are alphanumeric and must begin with a letter.
- Binary selectors must have exactly one argument. Binary selectors are not alphanumeric and are mostly used for arithmetic e.g. #+ or logical methods e.g. #>.  
- Keyword selectors are used for methods with one or more arguments. There must be one keyword per argument e.g. the method for adding a value in a KeyedCollection has this signature at: key put: value. Each keyword must be alphanumeric, begin with a character and end with a colon. The actual message selector used at runtime is the keywords concatenated, in the example above: #at:put:.
Going back to the example, similarly to the reserved word this in Java, self refers to the object instance which receives the message #root, commonly referred to as the receiver in Smalltalk. Therefore the expression self hasParent means that the unary message named #hasParent is sent to the same object that received the #root message. Smalltalk does not have void methods, so there will always be a return value, or an answer to stick with Smalltalk's message analogy. Naturally the role of the object that sends the message is referred to as the sender. If there is no explicit return from the method (return statements begin with '^'), the receiver will answer itself (verb form this time).

In turn the keyword message ifTrue: [parent root] ifFalse: [self] is sent to the return value. The latter message consists of several parts: The message selector #isTrue:ifFalse:, and the two arguments [parent root] and [self].

A piece of Smalltalk code enclosed in square brackets is referred to as a Block. Blocks have some similarities to higher order functions, such as static scoping, return value, and possibly parameters. In addition they are first class objects, which in this case are passed as message arguments. Note how the entire method can be built only by sending messages to objects. The observant reader will by now have figured out how if-else is implemented in Smalltalk. The method #ifTrue:ifFalse: is of course implemented differently in the two subclasses of Boolean.

```
#ifTrue: trueBlock ifFalse: falseBlock

^trueBlock value
```

Example 2 Implementation of #ifTrue:ifFalse: in class True

```
#ifTrue: trueBlock ifFalse: #falseBlock

^falseBlock value
```

Example 3 Implementation of #ifTrue:ifFalse: in class False

The method #value on Block evaluates the code in the block and answers the answer of the final statement. Applied on our method that means if #hasParent answers false, #root will answer the receiver, if #hasParent answers true we answer by recursively sending the message #root to parent.

Simple and natural
Of course measures such as the readability and simplicity of a programming language are highly subjective. Still, the previous example goes some way in illustrating how Smalltalk was designed to make programming easier to understand. Experiments of Smalltalk used for problem solving for non IT professionals and children in the seventies gave some promising results (Kay 1996). To some extent all programming languages make use of words from natural language to show intent or consequence, such as: if, else, goto, while, return and so on. Still, note how Smalltalk used in the right way goes a long way to achieve “intention revealing messages” (Beck 1997). See how standard language messages do not break the flow noticeably, and blend in with application specific messages. Also notice how multiple message arguments must be separated by keywords which describe the role of each argument.
1.2.2 Similarities
Java was chosen as target language for several good reasons. In many respects Java and its competitor C# have kept many of the most successful and important features of the predecessor Smalltalk.

Object oriented
Most importantly Java is an object oriented language. Dynamic method lookup is even relied upon for control flow in Smalltalk, so any language which does not provide this is ruled out. Methods and classes are in Java by default not final, so there is no need to explicitly declare methods virtual as in C++. Both languages also share a single inheritance model. With a few exceptions the Object class of both languages is the ancestor of all other classes.

Interpreted
Neither language is normally compiled directly into executable code targeted for one particular machine architecture. Instead the compiler transforms the source code into byte codes, which rely on a virtual machine to execute.

Garbage collected
Neither language requires explicit memory management by the program. The virtual machine’s garbage collector reclaims memory of unreferenced objects.

Memory model
The memory model of Smalltalk and Java are quite similar. Objects are allocated on the heap. Composite objects and data structures consist of references to other objects. Local variables and arguments in the execution stack merely contain references to objects allocated on the heap.
In Java this is not true for primitive values. In fact the VisualAge for Smalltalk environment is also optimized to allow direct representation of immediate object values, see (IBM 2002), page 287. Immediate objects include true, false, nil and instances of Character and SmallInteger. Neither language allows pointer arithmetic or out-parameters. Note that all the immediate object values in Smalltalk are immutable.

Range check of indexed structures
For both languages the virtual machine will raise an exception if the program tries to address or store to an index outside the bounds of an Array or a String.

Reflective Object model
In both languages the objects are aware of their type. This is required for dynamic method lookup and in Java also for runtime type checks. The program may also introspect various attributes of the class at runtime.

1.2.3 Differences
In this section we are mostly interested in differences that pose a challenge for a translation from Smalltalk to Java.

Dynamic vs. Strong typing
Java requires static type information of all objects. The Smalltalk language on the other hand is dynamically typed. We therefore need type information for all the Smalltalk source code to be able to generate Java source code for SICS. In section 4 we will see that determining the correct Java type at compile time and runtime is not always as easy as one may think.
Null Value
In Smalltalk object references initially reference nil, similarly in Java reference null, unless the declaration contains an assignment. There is, however, a big difference in the null values, as nil in Smalltalk is a proper object capable of responding to messages, while invoking methods on null in Java will give a NullPointerException. This has several implications for generated code, which is discussed in section 12.

Indexing
Indexable structures are addressed from 1 to size in Smalltalk, but from 0 to size – 1 in Java. This is discussed in section 7.

Meta classes
The notion of meta class is an important feature of Smalltalk. We will demonstrate how Java has no direct counterpart, and a pattern to implement a similar structure in Java is given in section 2.5.

Numbers
Smalltalk provides number types and semantics that suit a financial application like SICS well. The main challenges when translating to Java is: the absence of corresponding types, different semantics and strong typing. An application specific strategy to overcome these problems is presented in section 9.

Syntax
Example 1 demonstrated that even a fundamental construct like if-else is implemented radically different in Smalltalk than other programming languages. In section 5 we will see how the source code must in some cases be transformed to suit Java’s built-in control structures.

Blocks
Example 1 also demonstrated the use of blocks. In addition there are also some characteristics of blocks not easily or effectively replicated in Java, this is discussed in section 8.

1.3 Previous research
Several systems for translation of Smalltalk to other programming languages have been made in the past:

The ones I have found references to include (Yasumatsu and Doi 1995) which describes SPiCE, a system for translation of Smalltalk applications to C. The system focuses on runtime support for Smalltalk requirements such as dynamic method invocation, activation records and garbage collection as part of the translated stand alone application. The resulting C applications performed comparatively to the most effective Smalltalk VM at the time.

(Wang, Yang et al. 2001) describe s2cpp, a system for translation of Smalltalk to C++.

Specific issues in translation from Smalltalk to Java have also been discussed earlier:

(Bothner 1996) discusses a few central translation issues such as dynamic method dispatch and replacement of Smalltalk base classes.
(Engelbrecht and Kourie 1998) gives an excellent introduction to the topic. It has the most comprehensive coverage of translation issues and solutions that I have come across. (Engelbrecht and Kourie 2003) describes a solution for translating Smalltalk blocks to Java.

There are at least two currently commercially available translators for Smalltalk to Java:
- DirectJava from Object’ive (http://www.object-ive.com)
- SMT from Synchrony Systems (http://www.sync-sys.com)

1.4 Terminology and notation
To some extent Smalltalk terminology is used in the discussions. This may be unfamiliar for readers with a background from other object oriented languages. Some of the concepts are introduced in section 1.2.1, but repeated here one by one for clarity:

**Symbol**
Smalltalk has a special string class called Symbol for representing immutable constants in the global symbol table. Symbols are highly efficient to compare as two symbols with the same string value must be the same instance, so identity comparison is adequate. In code symbol literals are prefixed with a hash character ‘#’, e.g. #printOn:.

**Message**
All components in a Smalltalk system are objects. Everything that goes on inside the system are objects operating on other objects by sending them messages. In other programming languages this is commonly referred to as method invocation.

**Receiver**
The object that the message is sent to is referred to as the receiver.

**Sender**
The object sending the message is referred to as the sender. This will be an object of the class in which the code containing the message send is located.

**Answer**
The receiver will always respond to the message by returning a value. In Smalltalk this is commonly referred to as the answer. The verb form is also very common in method documentation, e.g. “Answer myself negated”.

**(Message) selector**
When a message is sent to an object the method name decides what method is invoked on the receiver. Therefore method name is commonly referred to as message selector or simply selector. The method selector is a symbol.

**Hook method**
The term hook method refers to a method that is specially provided to allow subclasses to override and customize their behaviour. There are often quite rigid constraints on how overrides should be carried out. Examples from Java include toString(), hashCode() and finalize().

**Smalltalk method references**
Smalltalk methods are referred in the text by their selector. As the selector is represented as a Symbol in Smalltalk code, I have chosen symbol notation also in the text. By convention all method selectors have a leading lower case letter. In addition references are normally highlighted in bold face to stand out, for instance: #printOn.
For references to the implementation of a method in a particular class, IBM VisualAge for Smalltalk’s convention is used: Class name, followed by two larger than signs, followed by message selector, for instance: **SequenceableCollection>>#at:**
For class methods the class is the meta class, represented by the name of the instance class suffixed by class, for instance: **CfsReadFileStream class>>#open:**

**Java method references**
For Java method references we attempt to unambiguously reference the method by its signature, which means it’s name and unnamed parameter types enclosed by parenthesis.
For instance: **equals(Object)**
In some cases the parameter types may be substituted with three dots, if it is fairly evident which method it must be based on the context.

The notation itself does not distinguish between static and instance methods, but the enclosing text should explicitly state that the method is static, if not one may assume that it is an instance method. The method may be qualified by class, for instance: **String.equals(Object).**

**Type references**
The type system in the Java Port project is naturally based on the existing Smalltalk class hierarchy. However for the discussions here and the work in the Java Port project it is helpful to distinguish type references from class references. Type references are therefore given as symbols, while references without a leading hash mean the class.
For instance: **#SequenceableCollection**, refers to to the type. By convention all class and type names in the system have a leading uppercase character, so all such symbols in the text reference types. In addition there are 3 type lowercase symbols as stand in representation for some Java primitive types: **#int, #boolean and #long.**

**Class References**
Classes are referenced by the unqualified class name. The notation itself does not distinguish between Smalltalk and Java classes, but the name will normally be unique for one language, if not the context should make it clear.

**Code Samples**
Smalltalk code samples are bounded by a rectangle with grey background. Java code samples are bounded by a rectangle and reserved words are coloured by default settings of the Eclipse integrated development environment.

1.5 **Methodology**
The discovery of and solution to most translation problems has followed the pattern described below:

1. Identification of problem.
   Some problems were known up front due to detailed knowledge of the two programming languages. Others were realized due to inconsistent behaviour in the translated application.

2. Team discussions about solutions to the problem.

3. Implementation of solution in the translation toolset.

4. Continuous evaluation of chosen solutions. Some solutions were refined, others abandoned.
This report is written after four years of work on the topic. The emphasis is on practical issues. The most important and interesting translation problems of the SICS Java port project are presented quite independently of each other. Each translation problem is roughly presented according to the order below, though not all items apply to all problems.

1. Description of the problem. A short example may be used to demonstrate why it is a problem
2. Reference to previous work on the topic
3. Discussion of alternative solutions.
4. The preferred solution in the SICS Java port project
5. Project experiences with the chosen solution and possibly changes done along the way.

1.6 Parse tree transformation mappings

Although I will try to make discussions about translation issues relevant outside the SICS system and the translation tools developed as part of the Java port project, the chosen solutions are naturally affected by the capabilities of the translation tools. The solution to most translation issues presented in this report, rely on one basic principle. I will therefore briefly describe the component that implements this principle and give a short example.

1.6.1 Mappings

The toolset allows the user to specify that a Smalltalk message expression should be transformed into something else. We refer to each of these rules as a mapping. A mapping may be created for a given method selector in a class. The mapping applies to all message expression nodes in the parse tree of that message selector and the type of the receiver is either that class or a subclass. It is also possible to possible to specify more than one mapping for a selector, in which case the different mappings must be qualified by predicates. If another mapping is specified for the same selector in a subclass it takes precedence over the mapping for the superclass. Note, this is only based on static type information in the parse tree. Mappings are not polymorphic and offer no guarantee about the type of the object at runtime.

1.6.2 Implementation Details

The mapping contents are actually stored as a Smalltalk source string, resembling a method. The Smalltalk parser turns into a method parse tree at runtime, which contains all the necessary information for the mapping engine to transform the node into a new sub-tree.

The predicates are specified as source code strings resembling Smalltalk blocks. The blocks are compiled and evaluated at runtime. The block must answer either true or false, indicating whether the mapping is a valid.

Here is an example of a mapping:
Picture 1

Picture 1 shows a trivial mapping defined for the selector #= in the Object class. The upper text field contains the mapping method source. The lower text field contains the mapping predicate, which in this case always answers true. As this mapping is defined in the Object class, it applies to all message expressions with selector #= as all receiver types are subtypes of Object. As a result such message expressions are transformed into message expressions with the selector #areEqual: directed to the class ComparisonUtilities instead.

Let’s demonstrate what will happen to the parse tree during generation. There will be many examples of mappings later while we cover translation issues to give more thorough understanding. The main principle is very intuitive. Assume we have the mapping above and the Smalltalk method below:

```
= otherObject
"^<Boolean>"

^self value = otherObject value
```
Diagram 1

Diagram 1 shows the method parse tree and the mapping parse tree side by side. Arrows indicate which parse nodes substitute others.

Let’s describe the order of events step by step:

When the generation reaches the “==” message expression, there exists a mapping for that expression. The Java generator will take the parse tree of the mapping, and insert the main expression of the mapping in place of the original message expression. We see that references to the “receiver” and “arguments” of the mapping “method” are substituted with nodes from the original parse tree. Notice the similarity to method invocations. The parse tree after the mapping has taken place can be seen in diagram 2.
Diagram 2

If the mapping in picture 1 contained more than one reference to either this or anObject, there would be a problem. As both the receiver and argument of the original message expression are also message expressions, it would be a mistake to insert any of those expressions in more than one location. This would duplicate the message expression in Java and execute the method more than once at runtime. Clearly that could cause all sorts of problems. So in such cases the transformation engine will declare and assign a temporary value for the expression first. The temporary variable is then used within the mapping.
2 Basic Code Structure

We begin by looking at how the main components of a Smalltalk program are converted to Java.

2.1 Classes

As we know Smalltalk and Java are object oriented languages. Programs in both languages are therefore made up of a collection of classes. As a main rule each Smalltalk class has a corresponding Java class. We will later see that in some cases we need more than one Java class to represent a Smalltalk class.

2.1.1 Class and package name

By default all translated classes have the same class name as in Smalltalk. There exists a special hook method that may be overridden to give a class another name in Java. As all class names are defined in a flat namespace in Smalltalk, SICS classes tend to have quite long names.

IBM VisualAge for Smalltalk organizes classes in applications. An application is not a namespace (like Java packages), but a module of related classes. Each class must be defined in (belong to) one application. The translated classes are organized in packages in Java. The package name is derived from the application of the class in Smalltalk. It is possible to change the Java package name of all classes in the application by a special hook method. It is also possible to modify the package name individually per class with a hook method.

We also allow the user to distinguish between Java type name and class name. By type name we mean the Java type that is used to represent the type. By Java class name we refer to the actual class of objects of that type. The class name is only required for instantiation of the object. By default the Java type name and class name are the same.

Separating interface from implementation is a proven Java strategy to achieve greater flexibility and looser coupling. If the Java class name differs from the Java type name of a SICS class, the Java generator will produce two Java files:

1. A Java interface with all the method signatures
2. A Java class which implements the interface and all the methods

Distinction between class and type name also has consequences for references to the type. For instance the Smalltalk class OrderedCollection has Java type (java.util.)List. Therefore all references to the type in Java use List. But to instantiate an object of the class we need the class that actually implements the type, so the statement:

```
OrderedCollection new
```

is translated as:

```
new ArrayList<..>();
```

2.2 Instance variables

Instance variables are only accessible from within the class and subclasses in Smalltalk. We use the Java access modifier protected, as it offers the closest match
with regard to encapsulation. It is not possible to address instance variables in other instances than the receiver in Smalltalk, this is different in Java.

2.3 Methods

2.3.1 Generation process for a method

In the VisualAge for Smalltalk environment source code is stored, versioned and compiled individually per method. It is therefore natural for the Java generation process to translate the methods one by one. As all Smalltalk source code is stored in the methods, this is where the bulk of the work occurs. The generation process for a method follows the steps described below in a single pass. The resulting Java methods are kept in a temporary structure, as we cannot write the Java source file until all required imports are known.

Parsing the source code

Due to Smalltalk's dynamic nature, a powerful parser is integrated in the VisualAge for Smalltalk environment. The parser and compiler are used to compile the methods in the image and to evaluate Smalltalk statements on the fly.

The syntax trees built by the parser therefore contain nodes with sufficient semantic information for the Smalltalk compiler to generate byte code. There are for instance dedicated parse node classes that represent #ifTrue:ifFalse: and #whileTrue:. From a grammar point of view, both the foregoing are ordinary message expressions. From a translation point of view however, it is clearly an advantage that these special methods are clearly distinguishable.

The parser is also used by the code formatter, and so it preserves a number of items that are particularly useful in a source to source translation, such as parenthesis and comments.

Assigning types to nodes of the parse tree

As will be described in the next chapter the toolset includes interactive tools and a type repository that stores information about types of the Smalltalk code. Error conditions and type incompatibilities should be manually fixed prior to translating the code. So in the Java generation phase there should normally not be any type errors left. The statically known types for the given method is combined with rules to recursively add type information to all nodes in the parse tree.

Recursively applying translation rules to the parse tree

The translation process proceeds recursively down the methods parse tree. For each parse node the Java generator will query a database of transformation mappings. If there is a mapping, it will transform the parse tree before passing control back to the Java generator. If there is no mapping, the responsibility goes back to the node itself, see next section.

Translating the remaining parse nodes directly to Java

As we noted above, if there is no special handler for a given node, the node itself must generate Java. The entire parse node class hierarchy extended with methods for generating Java. Class extensions is a feature of the VisualAge Smalltalk environment, that allows addition of methods to existing classes, it is described in more detail in section 11. This follows good Smalltalk object oriented practice, as the nodes best know how to represent themselves in Java. Most of the low level syntax is generated from parse nodes. Some examples of this are: constant literals, assignments, variable references and method invocations.
2.3.2 Method names
Method names are translated according to the rules below, unless there is defined a rename mapping, for the method which takes precedence. A rename mapping is simply a mapping with the rename flag set. In addition to transforming all senders of the method, all implementers are also renamed accordingly. Mappings may only rename methods if the message expression it maps to meets these requirements:

- Has the same receiver. i.e. this
- Has the same number of arguments

In addition if the a method or variable name happens to be a Java reserved word, it is renamed according to a global mapping table.

Unary method selectors
The Java method name convention for unary methods is trivial enough. The method names are unchanged from Smalltalk. This gives an unambiguous one to one mapping between unary Smalltalk methods and Java methods without parameters.

We did at one point consider whether we should prefix all instance variable getters with “get” to comply with Java beans convention. In addition (SUN 1999) states that method names should be verbs. In the end, however, we chose not to do this, much due to problems with reflection discussed later.

Binary method selectors
For binary methods there is really no option but to rename the methods, as the selectors are not valid in Java. For instance implementations of #= are mapped to #equals:. Fortunately there are only a limited number of binary selectors and only a few of them are implemented in SICS classes. We have chosen to require rename mappings, as that is the most flexible solution and it also ensures that senders of the method are transformed to the new selector. There is, however, no way on the Java side to know what the selector was in Smalltalk. That could only be achieved by having global rename rules to unique Java method names as proposed by (Engelbrecht and Kourie 1998), and make the rules available at runtime in Java. We have not prioritized this.

Keyword method selectors
Keywords selectors are converted in the following manner
1. Colons from the end of keywords are removed
2. The first character of all keywords except the first are converted to uppercase
3. The keywords are concatenated together

Our convention for keyword methods is a bit more problematic. Table 1 shows a few examples of Java method names corresponding to Smalltalk keyword selectors. Note how the second and third examples produce the same method name in Java. The method signatures are still unique in Java as one has one parameter and the other has two.

<table>
<thead>
<tr>
<th>Keyword selector</th>
<th>SICS4 Java method name</th>
<th>(Engelbrecht and Kourie 1998)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#at:put:</td>
<td>atPut</td>
<td>at_put</td>
</tr>
<tr>
<td>#addChildOfParent:</td>
<td>addChildOfParent</td>
<td>addChildOfParent</td>
</tr>
</tbody>
</table>

Table 1
However, we quickly came to realize that it is no longer possible to unambiguously know what the method selector was in Smalltalk. (Engelbrecht and Kourie 1998) on the other hand consistently substitute the semicolons in keyword selectors with underscores, this allows unambiguous two-way mapping between Java method names and Smalltalk method selectors.

It is also clear that the method names in Java can get very long. Most of the time the keywords appended to the end do not enhance the readability, as the logical link to the argument they describe is lost. Alternatively one might discard all but the first keyword. That would give shorter method names, but the risk of method name clashes and the number of overloaded method signatures would increase.

As for getters we have not prioritized renaming instance variable setters according to Java beans conventions.

**Problems for reflection**
In our discussions up to this point regarding method names in Java, we have focused on backward traceability to Smalltalk method selectors. The primary reason for this is the use of reflection in SICS code. This problem is discussed further in section 15.1.

### 2.4 Differences in the Object class
All classes in the VisualAge for Smalltalk environment inherit from Object, except for a few very special cases. All classes in Java inherit from `java.lang.Object`. So it should be straightforward to let subclasses of `Object` in Smalltalk, inherit from `java.lang.Object` in Java. However the hook methods defined in the two base classes are in many cases different. For instance: Java objects alter their default string representation by redefining the `toString()` method. Smalltalk classes on the other hand must override `#printOn:` to achieve the same. So if a class that overrides `#printOn:` was translated as a direct subclass `java.lang.Object` in Java, it would compile and run, but the `printOn(...)` method would never be invoked. The behaviour is therefore not consistent with Smalltalk. We therefore have to put a layer in between to intercept the Smalltalk hooks and reflect that behaviour in the corresponding Java hook methods. This gives the following convention:

- All classes that inherit directly from `Object` in Smalltalk, inherit from `com.csc.sics.framework.lang.CscObject` in Java.

Assume class A inherits from `Object` in Smalltalk. This gives the situation illustrated in diagram 3 below:
As a consequence all but a few translated classes directly or indirectly inherit from `CscObject`. This also allows us to define Smalltalk functionality which is not present in Java in `CscObject`. This assumes that there is a default implementation in the Object class in Smalltalk, and that it is only overridden in other classes that are also translated. It does for instance not work for `String` and `Boolean`, which are not translated and therefore outside our control in Java. In addition we do not want to clutter the interface of the base class unnecessarily.

The only classes that do not inherit from `CscClass` are classes that inherit from some other class which is not translated. For instance our custom subclasses of `OrderedCollection` inherit directly from `java.util.ArrayList` in Java in order to act as proper collections. In such cases it is the responsibility of the developer to correctly handle any Smalltalk overrides such as `#printOn` etc.

**Alternatives**

(Engelbrecht and Kourie 1998) propose porting the entire Smalltalk class library to Java. All classes would then inherit from a class called `stj_Object` which is the top of the hierarchy of translated classes in Java. Among other things this allows simulation of Smalltalk’s dynamic typing. The approach would also effectively solve a lot of the other problems discussed, such as lacking features in java and class extensions. In SICS4 we have, however, chosen to use as much as possible of the existing Java class library directly.

Other alternatives to porting include custom subclasses of non final base types or wrappers around the base class objects. However, wherever the system interfaces with infrastructure libraries such as GUI or database, these custom objects must be converted back and forth to the standard Java objects. So we quickly decided against those strategies.
2.5 Meta classes

Meta classes are an essential and quite unique feature of Smalltalk. An introduction to the concept is given in (Goldberg and Robson 1989) p. 76. I will attempt to describe some characteristics of meta classes and their runtime organisation:

1. All components of a Smalltalk environment are objects.
2. All objects are instances of a class.
3. Due to 1, classes are also objects, often referred to as class objects. There is only one instance of each class object, which stored under its name in the global Smalltalk dictionary. The code `Smalltalk at: #String` is equivalent with `String`, both return a reference to the `String` class instance.
4. Due to 2 a class object is an instance of a meta class. A meta class is an instance of the class `MetaClass`. The `MetaClass` has an instance variable `primaryInstance` which refers to its instance, or class object (see above).
5. The class of a `MetaClass` is referred to as `MetaClass class` (a `Class`), and it’s class is `MetaClass`, so in an endless chain sending the `#class` message, the result will alternate between these two objects. This fact is slightly simplified in diagram 4, which shows that `MetaClass` is the class of `MetaClass`. The important thing is however to point out that there are not infinite meta levels.

Diagram 4

The diagram shows class and instance relations in Smalltalk.

Here is a short example that demonstrates the class relations, by repeatedly asking for an objects class and prints the result to the transcript

```
| obj |
obj := 5.

Transcript cr; print: obj. "5"
Transcript cr; print: obj class. "SmallInteger"
```
Although these relationships are crucial to the Smalltalk environment, admittedly even experienced developers may get confused from time to time, and it is a challenge to keep the terminology consistent as everything we refer to can be several things at the same time depending on perspective. Let’s therefore focus back on the practical consequences:

2.5.1 Class methods

According to the diagram above class methods are defined by the meta class, which sole instance is the class object. In day to day programming, however, we normally think of the class and meta class as one entity with instance methods and variables and class methods and variables. We sometimes also refer to class-side or instance-side in order to clarify; this terminology is practical the standard VisualAge code browsers use a toggle button to switch between the list of class methods and instance methods. Based on the previous discussion we can quickly establish some central characteristics that our implementation of class objects must support:

1. The class object must be globally accessible by name
2. It must be possible to transfer references to the class object
3. Instances of the class must have access to their class object
4. The class object must be capable of responding to the class’ class methods. This includes any user defined class method.
5. Class methods must be polymorphic and late bound, this implies that there is a “self” of the class methods, namely the class object.

At first sight there are some evident candidates for implementing class methods and class objects in Java.

Static methods

Java classes may define static methods. Static methods do not require an instance of the class, but can be sent globally in the form ClassName.staticMethodName(...). Java static method are however incapable of supporting characteristic 5, late bound method dispatch. This has been thoroughly demonstrated by (Engelbrecht and Kourie 1998). Further static methods do nothing to support the class as an object. Characteristics 1 to 4 are therefore also not fulfilled.

java.lang.Class

The Java class library includes several facilities to perform reflection and to introspect classes at runtime. The relevant classes are located in the java.lang and java.lang.reflect packages. Classes are represented by instances of java.lang.Class, in the remainder referred to as Class with a capital C in bold face. Each object is associated with an instance of Class, and there are even dedicated instances for the primitive types, void and array types. Given the built in support for associating Class instances to all Java objects, we are pretty close to fulfil the Smalltalk characteristics we started out with.

- All components of the environment are objects: This applies to everything but the primitive data types
• All objects are instances of a class: invoking `getClass()` returns an instance of `Class` which represents the class.

• Classes are also objects, implemented as singletons and globally accessible by name: There is exactly one instance of `Class` that represents each class, and the instance is globally available, using this syntax `ClassName.class`. There is however a significant difference: all instances of `Class` are instances of the same class, whereas each Smalltalk class object is an instance of its meta-class.

In short we see that the `Class` class and `getClass()` method provide good support for runtime representation of a systems classes as objects. So among the criteria we originally set up instances of `Class` fulfill 1 to 3. However, as we have noted all the class objects are instances of `Class`, it is therefore not possible for the instances to support user defined class methods. Instances of `Class` are therefore not fully able to substitute Smalltalk classes.

2.5.2 Dedicated classes to represent meta classes

The conclusions in the preceding sections and a solution to the problems have already been described by (Engelbrecht and Kourie 1998). The solution is to split the class into two classes in Java, one which contains instance side behaviour and state, and another that defines class side behaviour and state. We have adopted the solution proposed by (Engelbrecht and Kourie 1998) with some minor adjustments. Below are some implementation details, an example of this can be seen in example 4 and example 5.

• The “instance” class keeps the name of the Smalltalk class, the meta class is suffixed “Class”. This is slightly different from the naming convention proposed by (Engelbrecht and Kourie 1998), where the meta class is suffixed “_metaclass”. In our project however we felt that class names describe a class' instances, not the class itself. For instance the base classes in Java and Smalltalk are called Object not Class. Therefore it is logical to name the meta class “Class”, as its sole instance is a class.

• Class objects are implemented as singletons, global access is possible by invoking a public static method: `ClassNameClass.getInstance()`, (Engelbrecht and Kourie 1998) propose access through a final static variable, however we felt that using a method invocation is more flexible.

• All class objects inherit from `CscClass`. `CscClass` defines shared behaviour for all classes, and ensures that class side contracts from Smalltalk are preserved in a similar manner to `CscObject` for the instance side. `CscClass` inherits from `CscObject`, as Class inherits from Object in Smalltalk.

• Class objects have a protected default constructor, it ensures that:
  o A reference to its superclass class object is set up
  o The method `initializeClass()` is invoked this represents the Smalltalk class method `#initialize` which is invoked when the class is loaded. The default implementation in `CscClass` is empty.

Instances of the class have reference to the class object, through the polymorphic method `getKlass()`, e.g. the expression `anObject class` is translated as `anObject.getKlass()`. The `getKlass()` method returns the value of the static final variable `klass`, this ensures that the class object is loaded and instantiated before any instances of the class are created. The method is called `getKlass()` because the method name `getClass()` is already taken by the method in Object which returns the `Class`. 
Class variables and class instance variables
Smalltalk has two kinds of class variables: class variables and class instance variables. A class variable is shared among all instances of the class, the class itself, all subclasses and their instances. According to the former class variables must be accessible from both class methods and instance methods. We have also seen that the class objects are implemented as singletons. In other words there will be exactly one instance of the class object, and one instance of each of its subclasses. Class variables must therefore be declared static, in order to ensure that one copy is shared among all the class object instances in the hierarchy. Class instance variables on the other hand are only accessible from class methods, and there is one separate copy per subclass. Class instance variables are therefore declared protected.

Then we are left with the challenge to have the variable be accessible from instance methods. There are only two access modifiers in Java that may allow instances of class direct access to a variable of another class outside the hierarchy: The package modifier (default if no other modifier is given) which will give access to other classes in the same package or the public modifier which gives all classes access. As there are cases in SICS where the two classes reside in different packages we had to use the public modifier. So we see that package protected access is more restrictive than public, nevertheless none of them offer the encapsulation of class variables found in Smalltalk. That is however something we have to live with, due to our strategy for implementing meta classes. So it is important to stress that this is not a language defect in Java, but a consequence of our translation strategy. As the code is automatically generated from Smalltalk, we are at least sure that there will be no abusive access in the translated code. There is of course no guarantee of how the variables will be used in Java in the future, however it is questionable how relevant the Smalltalk semantics on this point will be in the future development of the system. Some examples of declaration and access of class and class instance variables are given below:

First the class declaration:

```java
CnuObject subclass: #MartinClassVariableAccessExamples
classInstanceVariableNames: 'classInstanceVariable '
instanceVariableNames: ''
classVariableNames: 'ClassVariable '
poolDictionaries: ''
```

Secondly we created simple getters and setters for ClassVariable as class and instance methods, and getter and setter for classInstanceVariable as class methods. Example 4 shows the resulting meta class in Java:

```java
public class MartinClassVariableAccessExamplesClass extends CscClass {

    public static final MartinClassVariableAccessExamplesClass instance =
    new MartinClassVariableAccessExamplesClass();

    public static Object ClassVariable;
    protected Object classInstanceVariable;

    protected MartinClassVariableAccessExamplesClass () {
        superclass = CnuObjectClass.getInstance();
    }

    @SmalltalkSelector("#classInstanceVariable")//$NON-NLS-1$
    public Object classInstanceVariable() {
```
Example 4 Class side access to class variables

We see that class and class instance variables in class methods are handled just like instance variables in instance methods, apart from the fact that the class variable is declared static. Example 5 below shows how the instance getter and setter for ClassVariable are translated. Notice how the class variable must be prefixed with the meta class.

Example 5 Instance side access to class variables

2.5.3 Classes without meta class

Although the meta class solution described in the previous section, is vital to implement several important hierarchies in SICS, there are also simpler hierarchies and more lightweight classes that do not define a lot of class side behaviour. In such cases it is hard to justify the complexity and overhead of splitting the class into two separate
entities and generating boilerplate code to wire the two together. As we have decided to utilize standard Java APIs as much as possible, there will be a mixture of classes with or without meta class in the system anyway. For instance neither the String nor Boolean class have meta classes. It is therefore possible to control whether meta classes should be generated or not for a certain hierarchy by overriding the class method #usesClassObject to answer true or false, false being the default. The setting also applies to subclasses unless overridden. For classes without meta classes there are a number of differences to what was described in the previous section:

- Class methods are translated as static methods
- Class method invocations from instance methods of the form: self class doXxx is translated as doXxx() without any prefix.
- A warning will be given if a any class method is overridden in a subclass
- Senders of #class to a receiver which type does not use class objects, will be translated as getClass(), which will return the objects java.lang.Class object. By type we mean the type assigned to the receiver of the #class message in the parse tree. Only the static type information is available to the Java generator. This does not give any guarantees about the actual type of the object at runtime!
- Class variables are static
- There is no support for class instance variables, as there are no instances of the class or it’s subclasses

2.5.4 Constructors
Here we define a constructor in Smalltalk as a class method that answers a new instance of the class. However, in Java constructors have very special characteristics and requirements:

- There is special syntax for constructors
- Each constructor must call another constructor or a constructor of the superclass, if not the default constructor of the superclass is implicitly called
- If no constructors are declared, the class will have a default (parameter less) constructor, but if any constructors with parameters are declared then the default constructor must be explicitly declared as well.
- Constructors are not inherited.

The solution for translation of constructors is entirely different for classes with meta classes and classes without. The two are therefore described separately.

Constructors in classes with meta classes
In this case constructors are not implemented by Java constructors, but by ordinary instance methods defined on the meta class, so in fact none of the above mentioned constraints apply. The class CscClass is the base class for all meta classes. It includes a Class instance variable called instanceClass, which holds a reference to the Class object that represents the instance side. In practice the reference is set up based on naming conventions in SICS. The method basicNew() uses reflection to invoke the default constructor on the class and return an instance.

Alternatively we could override basicNew() in all meta classes, with the concrete subtype as return type, and explicitly invoke the constructor of the instance class. However we did not want to repeat this method throughout the hierarchy, as there
would be more code to maintain. The implicit coupling between meta class and class, also gives us some flexibility with regard to handling code dependencies.

The benefit of built-in support for basicNew() in our meta class hierarchy is evident: all existing constructors can be translated as is. As they are implemented as normal methods, they are inherited and none of the constraints regarding Java constructors apply. The only downside is that in most cases the sender needs to make a down cast, as the constructors are inherited from a more abstract level. For instance the Smalltalk expression “CnuNote new” is far more complex in Java:

```
(CnuNote)CnuNoteClass().getInstance().newInstance().
```

**Constructors in classes without meta classes**

For classes without meta classes we cannot implement constructors as normal methods. Instead proper Java constructors must be created. It is however very difficult to automatically generate such constructors, as the Smalltalk methods do not fulfil the syntax requirements for Java constructors. Example 6 illustrates this:

```plaintext
newFor: anObject

| sample |

sample := self basicNew.
sample initializeFor: anObject.

^sample
```

**Example 6 Smalltalk constructor with temporary variable**

The Java generator does not perform sufficiently sophisticated semantic analysis to convert example 6 into a Java constructor with the same behaviour. Therefore a number of manual steps must be taken.

For overrides of #new the solution is quite straightforward:

1. Implement a default constructor in Java with corresponding behaviour
2. Check the “No Java generation” flag for the class method #new

This behaviour is then also inherited for all subclasses.

For constructors with parameters, which most user defined constructors in SICS are, additional manual effort is required.

**In lining the constructor**

The first option is to inline the constructor. This is achieved by the following steps:

1. Create a mapping that in-lines the behaviour, see examples below
2. Check the “No Java generation” flag for the original class method

Whether in-lining is acceptable depends on a number of things. First of all, Java constructors with parameters have a number of desirable qualities.

- It is a convenient way to initialize a new instance.
- It prohibits instantiations without the required input.

In lining the constructor on the other hand does not provide or enforce this. In addition, in-lined constructors duplicate the logic to initialize the new object in all senders, so the number of senders is also relevant. Alternatively one could map to a reusable method to initialize the object. Below are examples of a plain inline mapping and a mapping that delegates to a static method to configure the object.

```smalltalk
| superExample childExample |
superExample := SampleClassWithConstructor newFor: 'super'
childExample := SampleSubclassWithConstructor newFor: 'child'
```

**Example 7**

Example 7 shows a Smalltalk example of sending an inherited constructor with argument. Assume the constructor of Example 6 is defined in the `SampleClassWithConstructor` class, and inherited by the subclass `SampleSubclassWithConstructor`.

**Example 8: In-lined constructor mapping**

Example 8 shows an inline mapping for the constructor in example 6. Notice how the mapping is an exact copy of the implementation apart from the missing return in the last statement. This is common for all inline mappings. If an explicit return was part of the mapping it could cause a premature exit from the sender method when the parse tree of the method and the parse tree of the mapping are merged.

```java
SampleClassWithConstructor superExample = new
SampleClassWithConstructor();
superExample.initializeFor("super");

SampleSubclassWithConstructor childExample = new
SampleSubclassWithConstructor();
childExample.initializeFor("child");
```

**Example 9**

Example 9 shows the result in Java of the inline mapping for the constructor. We see that the encapsulation and responsibility of the Smalltalk constructor is lost.
Example 10

Example 10 shows a variation of the inline mapping that delegates initialization of the object to a reusable function.

```java
SampleClassWithConstructor superExample = new SampleClassWithConstructor();
SampleClassWithConstructor.initializeWith(superExample, "super");
SampleSubclassWithConstructor childExample = new SampleSubclassWithConstructor();
SampleSubclassWithConstructor.initializeWith(childExample, "child");
```

Example 11

Example 11 shows the result of the inline mapping with a dedicated initialization method. The advantage is not obvious in this case, as it only contains one statement.

Creating Java constructors with parameters

The approach that gives highest code quality, but requires the most work is to create proper Java constructors to replace the Smalltalk constructors with arguments. The following steps are required:

1. Manually implement and test the constructor in Java.
2. Provide the source code and imports for the constructor in the designated hooks for the Java generator. The constructor will be part of the generated class.
3. Check the “No Java generation” flag for the original class method
4. Map the original class method to the message expression _java_new: with _: as separator keywords if there is more than one parameter.
5. Repeat step 1 and 2 in all subclasses, the subclass constructors may simply invoke the corresponding super constructor.

Of the above, step 4 is without doubt the most cryptic. A couple of non-intuitive technical workarounds of the mapping tool are used:

- The _java_ prefix is omitted by the Java generator. It may be used to map directly to Java methods. For instance, if a part of a mapping is a message send of a method with the same name in Java, the Java generator could end up in endless recursion. In addition it signals to the Java generator that a number of checks are not required. The constructor #new: is defined in Object with an Integer argument. So in our example, if we mapped directly to #new: with a string argument, there would be a problem.
- As the mappings are represented as Smalltalk methods, they may only be expressed in valid Smalltalk syntax. Separator keywords are required for each argument in keyword message expressions. So in order to map to Java methods with several parameters _: may be used as keyword separator for all but the first keyword.
The Java generator will treat any method that meets the following requirements as a Java constructor:

- It must be a class method
- The receiver must be a class that does not have a meta class
- The Java method name is “new” according to the method name transformation rules given in section 2.3.2

If the above applies the Java generator will convert the message expression to a Java constructor invocation in the following fashion:

ClassName new: <parameters> -> new ClassName(<parameters>)

The result on our existing example would be:

```
SampleClassWithConstructor superExample = new SampleClassWithConstructor("super");
SampleSubclassWithConstructor childExample = new SampleSubclassWithConstructor("child");
```

Example 12

Example 12 shows that a proper Java constructor with mapping clearly gives the best result. However more manual effort is required than in the previous approaches.
3 Type system

Assigning types to elements of the system is perhaps the most important undertaking of the SICS4 project. The following two chapters will:

- Introduce the type system used in SICS4, this is required knowledge for understanding the translation issues that will be discussed later
- Present and discuss a few of the concrete problems experienced in the typing work

3.1 Terms

3.1.1 Typed item
The abstract term typed item refers to a Java syntactic element that requires type information. This can be any of:

- Instance variable
- Method return value
- Method parameter
- Temporary variable

3.1.2 Typed node
Prior to generation of Java code, the Smalltalk syntax tree is first processed in order to add type information to all nodes, for further information see section 2.3.1. It is also possible to manually override the type of a given node. The node is identified by its in-order traversal index in the tree. Once a typed node in a method is created by the user, it is available among the other typed items of that method.

3.2 Static Types

3.2.1 Object type
The object type is by far the most commonly used type. The objects types are based on the Smalltalk class hierarchies, with a few exceptions. The type is represented by the class name as a symbol, for instance #String, #Boolean or #SicsBusinessPartner. As the object types are concrete they are quite easy to map to Java.

3.2.2 Class type
Class types are analogous to object types, but distinguished by an underscore suffix. As the classes in Smalltalk are first class objects, we need a type for each class object. For instance the type #CnuObject denotes a typed item that may contain the CnuObject class or one of its subclasses.

3.2.3 Formal/Actual types
As we will see later there are situations where constraints or limitations in the Java language type system require us to use one formal type, but we know for sure something more specific about the actual type, information which we want to keep in the type system and Java generation process. We use formal/actual types (due to lack of a better term) to gap the differences in the type system of the SICS Java port and the Java type system. Formal/Actual types are represented as one type followed by another type separated by a slash, e.g. #Object/SELF.

The first type, called the formal type is the type that will be used in Java declarations, and is the only type that will be visible in the resulting Java code. The actual type is used to aid the automatic assignment of types and Java generation.
3.2.4 Composite (Generic) types

Collection type
The collection type is used for ordinary Smalltalk collections such as Set, Bag and OrderedCollection. The collection member type must be specified after the collection type, separated by a colon. e.g: #Array: String. Parenthesis can be used around the item type to specify an arbitrary complex type, e.g: #Array:(Array: String). The item type is mandatory for collections, if it is not specified it will be treated as #UNKNOWN, e.g. #OrderedCollection:UNKNOWN, as it is not completely typed.

KeyedCollection/Association type
The KeyedCollection and Association type are identical to the collection type, but with two member types, key and value. Example: #Dictionary: Symbol: CnuAbstractProcess denotes a dictionary with subclasses of CnuAbstractProcess keyed on Symbols.

Extends type
The Java generic upper bounded generic type is also supported as part of generic collection types. For instance #Collection(EXTENDS: : CnuAbstractPersistentDomainObject) corresponds to Collection<? extends CnuAbstractPersistentDomainObject>. It does not make sense to use the extends type for anything else than item types of generic types.

#Block (JacSpecifiedBlockType)
The #Block type is a composite type with some special characteristics. The first child element specifies the return value. Remaining elements specify the argument types left to right.
#Block: VOID: String denotes a single argument block that accepts a string as argument, and has no specific return type.

3.3 Meta Types

3.3.1 #UNKNOWN
Initially all type slots contain the value #UNKNOWN. The enclosing method/class will not be considered complete until all #UNKNOWN types are replaced with proper types.

3.3.2 #VOID
Methods or blocks with no return value are typed as #VOID. Strictly speaking messages and blocks always have a return value in Smalltalk. If a method does not include any explicit returns, the receiver itself will be returned. The last statement of a block will always be returned or nil if the block is empty. However there are plenty of cases where assigning return value is problematic or meaningless:
- Discarded return values, such as #do: blocks sent to collections
- Methods with side effects only (for instance so called “do-it” methods, and instance variable setters)
- Methods that offer no clear guarantees about the return value, where one of the two conditions above also applies. This corresponds to the return value UNSPECIFIED of (NCITS 1997) section 5.1.2.3 Return value specification.
- There is no clear intent or guarantee about the return value. For instance there are examples of implementations of methods that explicitly return different types of objects with no clear semantics. This is normally done just to terminate execution of the method
3.3.3 #ERROR

Expressions that raise errors are typed as #ERROR. The #ERROR type is acceptable with any required type, but does not assist type inference. For instance if a method only contains the statement:

```smalltalk
someMethod
self error: ‘Should not come here’.
```

The return type is initially set as #ERROR, unless implementers higher up or down enables the typing tool to figure out the correct return type. A method return value of type #ERROR is considered incomplete and must be manually typed. As stated above the statement is acceptable by the compiler for any assigned return type, as the translated Java would be:

```java
public String someMethod()
{
    throw new Error("Should not come here");
}
```

3.3.4 #ANY

#ANY corresponds to the Java 1.5 generic wildcard type. If a method accepts an ordered collection containing any sorts of elements the argument will be typed as #OrderedCollection:ANY which will be translated as List<?>. It is an error to type anything else than a generic element with #ANY.

3.3.5 #SELF

Any item typed as #SELF must be the same type as the receiver/enclosing class. The vast majority of methods with return type #SELF actually return the receiver itself. The type system, however, does not impose such semantics. For instance the #copy method will always return an object of the same kind and is therefore typed as #SELF. All Smalltalk methods without explicit return, return the receiver. All methods without explicit return are therefore initially typed as #SELF. The #SELF type logically corresponds to the `<RECEIVER>` protocol specified in (NCITS 1997) section 5.1.3.3. The #SELF type works excellent during type inferencing, however the Java type system has no such concept. A discussion of problems related to the #SELF type is included in section 4.7 Casting.

3.3.6 #SELFINST

Similarly to #SELF, #SELFINST is relative to the receiver/enclosing class. #SELFINST is only used on the class objects, and requires an item to be an instance of the receiver class. Notably all constructors are typed as #SELFINST. So the expression

```smalltalk
CnuNote new
```

Is of type #CnuNote as the class literal ‘CnuNote’ is of type #CnuNote_, and #new has return type #SELFINST.

3.3.7 #SELFCLASS

The opposite of #SELFINST is #SELFCLASS, namely the method that returns the class of an object. This type would normally only be used on the #class method. For instance the expression:
where the expression `self businessProcess` is of type `#CnuAbstractBusinessProcess`, is of type `#CnuAbstractBusinessProcess`.

3.3.8 #GEN1

The #GEN1 type specifies the first generic type of the enclosing object.
Example: `SequenceableCollection>>#at: anIndex`, is typed as #GEN1.
So the expression

```
oc at: 1
```

is of type #String if `oc` is of type `#OrderedCollection:String`.

3.3.9 #GEN2

Similarly, the #GEN2 type is used as return type for methods that return the second generic type of the enclosing object. For instance `KeyedCollection>>#at: aKey` is typed as #GEN2.
So the expression

```
dict at: 'hello'
```

is of type #Boolean if `dict` is of type `#Dictionary:String:Boolean`.

3.3.10 #ARG

A method typed as #ARGn will always return an object of the same type as argument n (indexed from 1). For instance Smalltalk semantics for `Collection>>add:` methods, states that the argument should be returned from the method. The return type is therefore #ARG1, as the expression will always be of the same type as the first argument. The expression

```
oc add: 'first'
```

is therefore of type #String, as the first argument is of type #String. Section 4.9 gives alternatives for realizing this concept in Java.

3.3.11 #BLOCKVALUE

#BLOCKVALUE is used to specify the type resulting from evaluating a block. This meta-type is only required where we do not statically know the types of a block, thus it is typically references some other metatype, e.g. #BLOCKVALUE:ARG1.
For instance `KeyedCollection>>#at: aKey ifPresent: aBlock` allows the sender to specify a one argument block to be evaluated if there is an entry for `key aKey`, otherwise `nil` is returned. The method return type is therefore #BLOCKVALUE:ARG2, i.e. the result of evaluating argument 2 which must be a block. `aBlock` must be of type #BLOCK:ANY:GEN2 i.e. a block that takes an argument of the keyed collection value type and returns a value of any type. The expression

```
formatters at: self class ifPresent: [:aHandler | aHandler format: self]
```

where `formatters` is of type `#Dictionary:Class:Handler` and the block argument is of type `#Block:String:Handler`, will be of type #String.
4 Typing Issues

This section will present some decisions and challenges in the project related to translating Smalltalk code to the strongly typed Java environment. The internal type system described in section 3 is an intermediate step to help us in this process. There are both issues related to assigning correct types of the internal type system and to realize our internal type model in the Java type system.

4.1 Primitive Types

Smalltalk as a language does not have primitive types. In theory instances of Character, SmallInteger, Float, Boolean and even UndefinedObject (nil) are atomic immutable objects that are transferred by reference rather than by value. There are no built-in language operators as the required behaviour is defined as methods on the classes.

In practice however Smalltalk virtual machines will optimize “primitive” objects. The user does not have to write Boolean new: or SmallInteger new:. The primitive object instances are automatically provided by the environment for constants in code such as true, false, 1, $a, 0.2f. In the reference Smalltalk virtual machine implementation (Goldberg and Robson 1983) the first bit of object references is used to indicate whether it actually is an object reference or a SmallInteger value. Basic integer arithmetic functions built into the virtual machine can then operate directly on the integer value. Some operations even have optimized byte codes for very common integer operands such as 0 and 1. In fact the VisualAge for Smalltalk environment is also optimized to allow direct representation of immediate object values. Immediate objects include true, false, nil and instances of Character and SmallInteger, see (IBM 2002), page 287.

Java however, does distinguish between primitive types and objects. The built-in arithmetic and logic operators can only be used on primitive types. At the same time it is not possible to write generic algorithms for primitive values, as they are not proper objects. Therefore Java provides wrapper classes for the primitive types. Instances of the wrapper classes can be used to encapsulate one corresponding primitive value. For instance new Integer(2), new Boolean(true), new Characer("b") etc., there are also convenient static valueOf(…) methods to convert to corresponding wrapper object instances. The question is: what should primitive values in Smalltalk code be typed as? On one hand primitives are required for computations; on the other hand reflection and generic procedures require proper objects. The most significant semantic language difference regarding primitive values is that Java primitives can never be null. Variables are initialized to a default value if the user does not provide one. Booleans are by default false, and the number primitives to some “0” value. Smalltalk primitives on the other hand are initialized as nil. It would therefore be extremely hazardous to type Smalltalk primitives as Java primitives, as primitive values would be different. This might result in all kinds of imminent and subtle errors in the translated program. Therefore the “primitive” Smalltalk types are by default translated as the Java wrapper types such as Boolean and Integer.

We have already seen that Java primitives may not be null. It is therefore natural to use primitives in local contexts where one is in control of initialization. This removes uncertainty about potential null values and the overhead of auto-boxing (see next section) is avoided. Unfortunately the SICS Java Port tools do not have sufficiently semantic analysis capability to determine such cases. There are, however, provided types corresponding to the Java primitives such as #int and #boolean, that may manually be assigned to instruct the Java generator to use Java primitives.
4.1.1 Autoboxing

What about arithmetic computations? Is it very tedious with the wrapper classes? Fortunately the auto-boxing feature was introduced in Java 1.5. In most cases the compiler will be able to automatically insert boxing and un-boxing operations in the bytecode. Example 13 shows manual boxing and un-boxing of integers required prior to Java 1.5, whereas example 14 shows how auto-boxing drastically simplifies the source code.

```
List integers = new ArrayList(3);
// store integers in generic list
integers.set(0, Integer.valueOf(1));
integers.set(1, Integer.valueOf(2));
// retrieve integers from generic list and store at index 2
integers.set(2,
    Integer.valueOf((Integer) integers.get(0)).intValue() +
    ((Integer) integers.get(1)).intValue()));
```

**Example 13** Manual boxing and unboxing of integers

```
List<Integer> integers = new ArrayList<Integer>(3);
// store integers in generic list
integers.set(0, 1);
integers.set(1, 2);
// retrieve integers from generic list and store at index 2
integers.set(2, integers.get(0) + integers.get(1));
```

**Example 14**
Example 14 shows how integers can be automatically boxed and unboxed by the compiler. Note that use of generics is required to make use of un-boxing from a generic data structure in this example.

4.2 Different inheritance hierarchies in Smalltalk and Java

4.2.1 Subclassing and Subtyping

As Smalltalk is a dynamically typed language, there is not really any concept of types. So can any object be used anywhere in the system? Certainly not, all objects are instances of a certain class and are capable of responding to messages implemented in that class. If an object of an incompatible class is used, the **#doesNotUnderstand:** method will be invoked by the virtual machine on invocations of methods not implemented on that class, and an error is signalled.\(^3\)

Specialization (sub-classing) is described as an “is a” relationship. Neither Smalltalk nor Java supports multiple inheritance. So if class B extends class A, an instance of class B “is” at the same time an instance of A. The opposite is not true, so instances of A are not instances of B. With regard to typing, this means that objects of class B are type compatible with A, but objects of class A are not type compatible with B. The same applies in Java, so in the case of A and B, this rule would be enforced by both the compiler and the run-time environment. From this it clearly follows that the main type convention is that types follow the inheritance structure of the Smalltalk classes.

\(^3\)Strictly speaking, it is possible to override **#doesNotUnderstand:** to do something else, however that represents a special case.
However, as we started with, any Smalltalk object may be used in a given scope as long as it is capable of responding to all messages sent to it within that scope. So in the method below the type of anObject could be defined as all objects capable of responding to the messages (#printString, #value, #name).

```smalltalk
doStuff: anObject

Transcript cr;
show: 'Object named: ';
show: anObject name;
show: ' is: ';
show: anObject printString.
^anObject value
```

On a general basis however, we do not operate with these kinds of types, as this may open for a lot of coincidental types that do not correctly reflect the intention or semantics of the code. One could envision using such temporary types internally in data flow analysis to aid automated type inference or static analysis, however this is not done in our toolset.

However, the example does indicate that there may be situations where the type requirements of an object are not based on inheritance order. Indeed (NCITS 1997), speaks of “protocols” not classes. This is particularly true for the stream and collection class hierarchies, see diagram 5.

![Diagram 5 Collection protocol hierarchy, taken from (NCITS 1997) page 156](image)

There are also a lot of application specific more or less formalized protocols in the SICS system. These protocols contain a set of messages and semantic requirements for those messages. If a class fully implements a given protocol, there is also an “is a” relationship to that protocol. This closely resembles Java interfaces, without compile time or run-time verification apart from potential #doesNotUnderstand: invocations. To cater for this, our typing model permits specifying type compatibilities across inheritance hierarchies.

In addition we follow Java guidelines to type certain kinds of objects by interfaces for greater flexibility. For instance (java.util.)List is used for most kinds of SequenceableCollection in Smalltalk and (java.util.)Map is used for subclasses of KeyedCollection (Hashtable-like objects).
4.2.2 #String, #Symbol and #EsAtom

String, Symbol and EsAtom are very similar classes in Smalltalk. Symbol inherits from String, but EsAtom is not related to the others by inheritance. Strings are mutable in Smalltalk, and therefore suitable for storing string data. Symbol and EsAtom on the other hand are immutable strings that are mostly used internally in the application. They are kept in their respective global symbol table, so that there will never be two identical instances in the system. For atoms and symbols it is therefore sufficient to use identity test (==) for comparisons. Atoms and symbols serve the same purpose. Atoms are distinguished by a double hash prefix (##). The distinction is of a technical nature. The packager assumes all symbols potentially relevant for reflection (#perform:), and gives warnings for symbol names with no implemented method of that name. Developers are therefore advised to use atoms for symbolic values used for purposes other than reflection.

In Java there is only one String class: (java.lang.)String. By invoking intern(), a reference to the string with that value in the global symbol table is returned, so there is no need for separate classes for symbols and atoms. Therefore all these 3 classes are mapped to (java.lang.)String. One might argue that this makes the code less restrictive, as operations currently not supported by EsAtom are suddenly available. For instance the code might crash in Smalltalk in one place, but be more forgiving in Java. However we do not consider this a problem, as we do not consider this particular case semantically important.

4.2.3 #SequenceableCollection

As previously noted most SequenceableCollection classes are mapped to (java.util.)List. This also applies to Array. We have chosen List over using Java arrays for Array, as using Java arrays would be too tedious (missing a lot of functionality). This is yet again an example where the code will be more forgiving in Java, as add(T) is specified in the List interface, but is not possible for Smalltalk arrays. In addition instances of Array and OrderedCollection are mixed quite a bit in SICS in Smalltalk, e.g. a method may return an OrderedCollection of data in some cases but an empty array (#()) in others. So we do not keep the distinction between Array and OrderedCollection. As above we do not attach a lot of semantic importance to this fact.

SortedCollection on the other hand, is treated specially. SortedCollection is a collection where the elements may be retrieved by index as in other lists, but the elements may change position when objects are added or removed in order to preserve ordering of the elements. Strictly speaking this may not be a proper List according to computer science theory, but for all practical purposes they are treated as lists within Smalltalk applications. In this case however we have chosen to map these lists to SortedCollection, as we feel that the original intent and semantics of the code is best reflected that way. So if an instance variable is typed as SortedCollection, it will not be possible to assign other kinds of List to it.

Some might see this as a violation of common conventions on Java collection interfaces. Many will argue that the collection interfaces are merely a requirement on method signatures, and all other aspects are merely “implementation issues”.

4.2.4 The Collection Hierarchy Incompatibilities

In Smalltalk KeyedCollection is a subclass of Collection. This means that all keyed dictionaries are also collections, and are capable of responding to the Smalltalk Collection protocol messages. The general contract is that methods such as #do:, #select: and #collect: operate on the values of the KeyedCollection. In Java however subtypes of (java.util.)Map are generally not subtypes of (java.util.)Collection.
For example in SICS an `OrderedCollection` is used for the `children` instance variable in the class `CnuTreeLikeObject`. As the name implies the class and subclasses typically form hierarchical structures. However one subclass initializes the same instance variable with a `LookupTable` (KeyedCollection). This is largely unproblematic in Smalltalk as both `LookupTable` and `OrderedCollection` share the `Collection` interface.

But how should we type the `children` instance variable? In practice this must be handled like a multi type instance variable, see section 4.4.3. The same problem applies to `String` which is a subclass of `SequenceableCollection(List)` and therefore also a subclass of `Collection` in the Smalltalk class hierarchy. String is not related to List in Java. To summarize, an excerpt of the `Collection` hierarchy with the two problem cases is given below:

```
Object
  Collection
    KeyedCollection
      LookupTable
    Dictionary
    SequenceableCollection
      ArrayedCollection
      EsString
    String
```

4.2.5 ExceptionalEvent hierarchy

There are also differences in the exception hierarchies in Smalltalk and Java. In Java each exception type is represented as a dedicated type, but in Smalltalk exceptions are `instances` of the same class. This also poses some serious type problems. Further details can be found in section 14.1.

4.2.6 Number types

The number hierarchies in Smalltalk and Java are quite different. In addition, we have taken certain application specific decisions regarding suitable types and computation semantics in the SICS Java port project. Further details can be found in section 9.

4.3 Parameter types in overridden methods

The `#=` method in Smalltalk can be overridden just like `equals` in Java. Most SICS implementations of `#=` assume that the parameter is an object of the same class and do a field by field comparison. The parameter is therefore initially typed as `#SELF`, this resolves all type issues and compiles in Java without problem. So the implementation of `CnuRate>>>#=` has Java signature `public boolean equals(CnuRate)`.

Is there anything wrong with that? The experienced reader will notice that this implementation of equals on `CnuRate` is an `overload` not an `override`. In Java overrides must have identical method signature. As Smalltalk is not typed, it is sufficient to have identical message selector to override a method. In order for `CnuRate` to properly override `equals` the argument must be of type `Object` in Java. The parameter should therefore be typed as `#Object/CnuRate` or `#Object/SELF` to give the type tool a hint to cast to `#CnuRate` where required within the method body.

In addition we add an `@Override` annotation to the generated code, this gives a compilation error if no matching super implementation is found.

4.4 Multi Types

Various known situations where multi types occur are presented in this section.
4.4.1 Definition of multi type
Within a given scope, a typed item may contain objects for which there is no common (Java) super type that defines the entire set of messages sent to the object.

4.4.2 Multi type argument with branches
As Smalltalk method arguments are not typed, overloaded method signatures are not possible. There are however examples in SICS of methods, where different kind of objects can be used as argument for the sake of convenience. (It is highly questionable whether this is good programming practice in Smalltalk, but we still have to deal with these methods). The body of the method will typically contain conditional branches that process the argument differently based on the type of argument. Example 15 below gives some examples of how such type checks may be done in Smalltalk:

```smalltalk
doStuffTo: x
  x isString ifTrue: [...]
  "or"
  x class = Array ifTrue: [...]
  "or"
  (x isKindOf: Number) ifTrue: [...]  
  "or even worse"
  (x respondsToSelector: #myMessage) ifTrue: [x myMessage].
```

Example 15 Multi type argument

Assume there is no single type for x in example 15 that supports all branches. There are two approaches to solve this problem:

Type argument as common superclass
This approach will typically result in typing the argument as #Object, and perform type tests and casts for the different branches. This can to some extent be done automatically.

Refactor as overloaded methods
Refactoring the method as several overloaded methods, may be a more elegant solution. Whether this is a good solution or not depends mostly on the method structure. For instance what is the ratio of custom branches compared to common functionality?
We have examples in SICS where a method can take an argument of either type A or B. Furthermore, a logically equivalent object of type B can be quite easily derived from an object of type A. For such cases the original logic can be kept in a method with argument type B. In addition we create an overloaded sibling method with argument type A. This method converts the A instance into a B instance, and invokes the method for argument type B. All reorganizing of code requires manual effort though.

4.4.3 Multi type Instance variable
There are also examples in SICS where an instance variable, its getter, setter and various other methods referencing the instance variable are defined in a superclass. A subclass, however, uses the same instance variable to store a different type of object. That will of course not be possible in Java, as the compile time and run-time checks will not allow it.
We have two alternate ways to solve this:

**Common Supertype and type casts**

**Directions:**

1. Type the instance variable and getter as a common superclass, `#Object` in the worst case.
2. Do specific type casts where required for all usage in both the superclass and subclasses.
3. In addition analyze the hierarchy. Bear in mind that code implemented in a given class may also be invoked on instances of its subclasses.

This approach may worsen the code quality in some cases. I will give a short example to illustrate: Assume a hierarchy with 50 classes. An instance variable called ‘x’ with corresponding getter and setter is defined in a common superclass. Two leaf classes are later introduced. The new classes have no need for the current implementation of ‘x’. Instead the instance variable, getter and setter are reused for the new classes’ own concept of ‘x’.

Let me first add that this case clearly represents abuse of existing code and poor design also in Smalltalk. Still we are not always able to clean up everything in our way as part of the Java port project, so occasionally we have to deal with situations like this. As a consequence Java code quality is suboptimal as the code is less precise, heavily relies on type casts and therefore loses out on compile time and runtime type checks. These disadvantages apply to all 50 classes, although really only required by two very specific subclasses.

**Introduce new dedicated instance variable in subclass**

Fortunately SICS classes rarely reference instance variables directly, apart from in getters, setters and `#initialize`. An alternative solution to the example given above with an instance variable called ‘x’:

1. Analyze the subclass. Hopefully this leads to the conclusion that the subclass never relies on inherited methods that rely on ‘x’ apart from the getter and setter.
2. Create an instance variable, getter and setter with a new name and correct type in the subclass. We assume ‘x’ is consistently replaced with ‘myX’.
3. Create a mapping for the getter and setter of ‘x’ in the subclass to corresponding ‘myX’ methods. This will dispatch all local senders to the new methods.
4. In addition, override the original ‘x’ methods to given a runtime exception if ever invoked in the subclass. If this should still happen, the conclusion drawn in step 1 is wrong.

**4.4.4 Multi type temporary variable reuse**

There are some examples in code where a temporary variable may reference objects of incompatible types during the course of the method. Most cases seem to be simply caused by laziness by the developer. For cases like this there are two options:

1. Type the variable as something more abstract, in many cases `#Object`. Then make type casts in all the places where the temporary is treated as a more specific type. This is similar to the first approach for multi type arguments.
2. Rewrite the method to add more dedicated temporary variables. This requires manual effort and great care.
4.4.5 Error/status return values

There are some examples in the standard library where a method may return a different kind of object to indicate an error. This is illustrated in the example below:

```
| fs |
fs := CfsReadFileStream open: 'c:\temp\test.txt'.
fs isCfsError ifFalse: [....]
```

The method `CfsReadFileStream class>>#open:` normally returns an instance of `CfsReadFileStream`. If an error occurs, an instance of `CfsError` is returned instead. The developer should always check the type before attempting to read from the stream.

There are also a few rare examples in SICS code where return of a special kind of object signals an error condition. One way to solve this: type the return value as some common supertype, in our example `#Object`. The sender must type check the result and cast from there on. On the other hand this gives less precise code due to a few exceptional cases. One programming guideline states that exception handling should be a separate concern, and affect the normal flow of code as little as possible. Another alternative is therefore:

1. Rewrite the implementation to throw a runtime exception instead.
2. Retype the implementation to the type of the “normal” result, in our case `#CfsReadStream`
3. Rewrite senders to properly catch the exception.

4.5 Protocol supported by several classes across hierarchies

As we have previously noted, the only “type” requirement for Smalltalk objects, is that they are able to properly respond to all messages sent to them. Unfortunately this flexibility may lead to code which is difficult to execute in a strongly typed environment. There are pieces of code in the system in where an object is not constrained to one single class hierarchy. This may range from a well defined interface or “protocol” to one single method invocation within a method. Conditional branches may further complicate the picture, as it is difficult to predict what kinds of objects might enter the different branches. First we will look at some causes for these kinds of situations.

4.5.1 Reuse (misuse?)

SICS is an extensive system, with many functional areas. From time to time similar requirements appear in completely different parts of the system. In some cases there is an attempt to reuse whole or parts of existing functionality designed for some part of the system, for something completely different. To be able to do this the new object must provide implement all the methods sent within the scope of the functionality to reuse.

4.5.2 Protocols supported by only some subclasses
Diagram 6

Diagram 6 shows a poorly designed class hierarchy. See how there are similarities among classes in different branches of the hierarchy. \#isCircular answers false in Shape, and is overridden to answer true in Circle and Sphere. Example 16 below shows a method which sends messages to a shape object.

describeShape: aShape on: aWriteStream

aWriteStream nextPutAll: ‘the shape ‘.

aShape isCircular

ifTrue: [ 
    aWriteStream
        nextPutAll: ‘ is circular with radius ‘;
    print: aShape getRadius.

aShape numberOfDimensions = 3

ifTrue: [ 
    aWriteStream
        nextPutAll: ‘, surface area ‘;
    print: aShape calculateSurfaceArea;
    nextPutAll: ‘ and volume ‘;
    print: aShape calculateVolume
]

ifFalse: [ 
    aWriteStream
        nextPutAll: ‘ and area ‘;
    print: aShape calculateArea]]

Example 16
Example 16 shows a method with an argument called \texttt{aShape} of the abstract type \texttt{Shape}, which is sent a number of messages. The method does not reflect good object oriented practice. Still there are examples of such hierarchies and code in SICS. Let’s take a look at the messages sent within the method body:

- \texttt{#isCircular} is defined on the abstract superclass \texttt{Shape}. So the method invocation will compile in Java.
- \texttt{#getRadius} is implemented on 2 of 4 leaf classes. Furthermore the two classes reside in different sub-trees. Different solutions to this problem are discussed below.
- \texttt{#numberOfDimensions} is also implemented on \texttt{Shape}.
- \texttt{#calculateSurfaceArea} is only defined for \texttt{Sphere}. If the message is only understood by one unique subclass/sub-tree, the Java generator inserts a downcast. In this particular case the result is: \texttt{((Sphere)aShape).calculateSurfaceArea()}.  
- \texttt{#calculateVolume} is constrained to the sub-tree \texttt{ThreeDimensionalShape}. Note due to the enclosing predicates and the previous message send we see that the object: must in fact be an instance of \texttt{Sphere}. Still, as a general rule, we do not make more specific casts than required. So we cast to \texttt{ThreeDimensionalShape}.
- \texttt{#calculateArea} seems to be supported for all concrete subclasses of \texttt{TwoDimensionalShape}, but there is no implementation on \texttt{TwoDimensionalShape}. This problem is discussed below in section 4.5.4.

4.5.3 Dedicated interface

This is the preferred solution for cases of methods only defined in some subclasses. In the hierarchy of diagram 6 we define an interface called \texttt{Circular}. The interface is implemented by \texttt{Circle} and \texttt{Sphere}. Technically this is achieved by the procedure given below:

1. Define a new class in the Smalltalk image called \texttt{Circular} to represent the \texttt{#Circular} interface type. After the class is defined, the type \texttt{#Circular} may be assigned to any typed item.
2. Implement a special hook in \texttt{Circle} and \texttt{Sphere} to inform the Java generator that they implement the interface \texttt{Circular}. This is required in order to generate correct implements information in the class signature. In addition it aids the type compatibility rules of the typing tool.

Dedicated interfaces are in many ways the optimal solution as they enhance type safety and give good structure to the code. On the other hand the process is quite time consuming and it can be hard to figure out good descriptive names for the numerous interfaces that are created this way. There are currently close to 450 such interfaces in SICS.

4.5.4 \#subclassResponsibility method in common superclass

Assume we have a hierarchy with one superclass and five subclasses. Three of the subclasses implement method \texttt{#x}. In a given scope an object may be an instance of any of the three subclasses that implement \texttt{#x}, but not the two other classes. We can be fairly sure of this as the message \texttt{#x} is sent and that would crash in Smalltalk if the object did not understand it. It is quite tempting to add an implementation of \texttt{#x} in the superclass. The implementation would simply send the message \texttt{#subclassResponsibility} to itself.
The method \texttt{#subclassResponsibility} is defined in Object. If it is ever invoked, a runtime exception will be signalled. This is the standard approach to enforce abstract methods in Smalltalk, as the language itself has not concept of abstract methods. Note that our solution in this case deviates slightly from the standard use of \texttt{#subclassResponsibility}, as some subclasses do not support the method. Still, the Java code will compile and the code would fail consistently with Smalltalk at runtime. One might argue that this is no “worse” than Smalltalk, where there is no compile time type check of message sends.

Indeed (Engelbrecht and Kourie 1998) propose a solution where all methods in the entire system are defined on the common superclass. Even though the situation will not be worse than Smalltalk, it certainly introduces uncertainty in the code. One might argue that this uncertainty is worse in Java, where one would not normally expect publicly available methods to fail by intent. We also risk “contaminating” the superclass interface with large numbers of methods that are only relevant for a few subclasses. For instance it would be hard to justify adding such a method if only three out of a hundred subclasses implemented it. I also think most would agree that to add the method \texttt{#getRadius} on \texttt{Shape} to solve the \texttt{Circular} problem, would be highly misleading.

Sometimes we are lucky however. From time to time it turns out that all subclasses implement the method, so there is an implicit contract present, but there is not defined any \texttt{#subclassResponsibility} method on the superclass. In addition the superclass should be abstract. Although the Smalltalk language does not enforce abstract classes like Java, abstract classes are still quite common in an application like SICS. This situation is internally referred to as \textit{missing subclass responsibility}. There are several examples of this in the standard class library. For instance \texttt{#Boolean} has two subclasses \texttt{#True} and \texttt{#False}. Several of the Boolean protocol operations such as \texttt{#and} are only defined on \texttt{#True} and \texttt{#False}, not in \texttt{#Boolean}. This does not cause any problems in Smalltalk, as there will never be an instance of the class \texttt{Boolean} at runtime, only the instances \texttt{true} and \texttt{false}. This, however, poses a challenge in a strongly typed environment. Likewise the message \texttt{#calculateArea} is understood by all subclasses of \texttt{TwoDimensionalShape} above. So the hierarchy will benefit from adding a subclass responsibility implementation on \texttt{TwoDimensionalShape}.

Once in a while there is an opportunity to create a superclass implementation that does not throw a runtime exception. The alternative is to create a sensible default implementation on the superclass. Whether this is possible relies on the intent and semantics of the method. Assume a method called \texttt{#isCube} is defined in \texttt{ThreeDimensionalShape} which answers \texttt{false}, and an override in \texttt{Cube} that answers \texttt{true}. In this case it would be unproblematic to move the implementation that answers false up to \texttt{Shape}, as \texttt{Square} and \texttt{Circle} are certainly not cubes. In some cases this can be an elegant solution, but it should be used with great care.

\textbf{4.5.5 Inline instanceof tests}

Imagine an extreme case:

\begin{itemize}
\item There are only 2 implementers in the entire system of a method. For instance \texttt{#getRadius} in the example above
\item The implementers do not reside in one unique hierarchy
\item The message is sent in only one place in the entire system.
\item The receiver may be an instance of either one of the implementers
\end{itemize}
The code will not compile if we type as one of the implementers, as the other is not compatible. Alternatively one could use instanceof tests and type cast to the specific types. This is not a recommended approach in SICS, but it has been used from time to time. Note that the existence of only one sender in this example does in itself not justify this approach. Still number of senders is part of the practical considerations we must make from time to time, and few senders certainly reduce the consequences.

Assume the sender of the method looks something like this:

```java
getRadius: aShape
^aShape getRadius
```

It is possible to rewrite the method to handle both types in Java:

```java
Double getRadius(Shape aShape) {  
    if(aShape instanceof Circle)  
        return ((Circle)aShape).getRadius();  
    else  
        return ((Sphere)aShape).getRadius();  
}
```

4.5.6 Utility/dispatch class
There exists an alternative to the former solution. The type tests and downcasts can be refactored as an external method, which we may invoke in place of the original invocation. There are several advantages over the in-lined approach:

- It is easier to add support for more types
- It can be invoked from several places

Just like the former solution it does not reflect good object oriented practice. So it is not a preferred solution in SICS. However it is still quite commonplace due to the base class extensions translation issue described in section 11.4.

4.5.7 Reflection
Coming back to our starting point once more: Smalltalk only requires that each object is able to properly respond to messages sent to it at runtime. One interesting mechanism that can bypass all the typing problems and reproduce this behaviour is reflection. The class of the object does not matter as long as it defines a method with that name. This idea was also discussed but in the end rejected by (Engelbrecht and Kourie 1998). For reasons that will be discussed in more detail in section 15.1, we have had to remove reflection based code from SICS in the transition to Java mostly due to performance problems. So using reflection to overcome multi type problems is not a viable option.

4.6 Return type problems
4.6.1 Incompatible return type
One of the most useful new features of Java 1.5 is the ability to use subtypes as return type for overrides in subclasses. For details see (Gosling 2005), section 8.4.5 Method Return Type. In most cases this turns out well, as subclasses by nature are specializations, and therefore method overrides tend to have more specific return types. This is also a great advantage for the generated code, as it reduces the need for casting the return type of messages sent to subclasses, and it aids the type inference
due to more precise information about return types. But, if the return type of a method override in a subclass is not the same or a proper subtype of the return type of super, there will be a compilation error in Java.

Assume we have a class called `RemoteOperationFailure` with a method called `#errorCode` of type `#String`. Assume there is a subclass of `RemoteOperationFailure` called `RemoteDatabaseOperationFailure`. `RemoteDatabaseOperationFailure` overrides `#errorCode` but the return type is `#Integer`. Again, this constructed example shows an inconsistently designed hierarchy, which is of course bad also in Smalltalk. Still there are real world examples where there is justification for inconsistent return types.

To resolve the problem, a few steps must be taken:

1. Retype the super implementation return value to a common super type. `RemoteOperationFailure>>#errorCode` must in our case have return type `#Object`. This will solve the compilation error in `RemoteDatabaseOperationFailure>>#errorCode`
2. Consider using a formal/actual return type in the super implementation with the former return type as actual type. In our case this means `#Object/String`. As a result `#errorCode` message expressions where the receiver type is `#Object` are cast to `#String` where required.
   NB: This relies solely on static type information, not the actual runtime object.
3. Fix any remaining errors in senders by manual typecasts.

4.6.2 Unintended overrides

There are a few odd examples where a subclass implementation is not an intended override. Class `Object` defined the hook method `#release`. It may be overridden to remove interest from any events in the publish/subscribe protocol used for user interface updates. However, the class `CnuVersion` has a getter named `#release` that returns a String with the SICS release number. Technically this is still an override in Smalltalk, the method will be called during user interface cleanup. But as `CnuVersion>>#release` is without side effect, this does not do any harm. We consider it an unintended override, as it was written for a completely different purpose. There are two steps we have to take to get rid of the problem.

1. The toolset has an option called “Not an override”, that may explicitly be set in cases like this. This removes warnings about return type, and omits the `@Override` annotation in the generated code.
2. Create a `rename mapping` to give the method another name in Java.

Fortunately these problems are quite rare, and most have been easily solved so far in the project.

4.6.3 Missing logical branch

As Smalltalk is a dynamically typed language, there are of course no type constraints on the return value of methods. We have also seen that unless the last statement of a method either returns or signals an exception, the receiver itself will be the return value of a method. Similarly the Java generator will insert “return this” at the bottom of such methods. The Java method return types may in some cases reveal that the method will in fact never return the receiver. The method `SicsAccountingAbstractPortfolioEarningCurve>>#refUnitOfTime` is shown in example 17:
Example 17 Invalid implicit return self

We see that the method returns an instance of SicsRefUnitOfTime in 4 logical branches. Initially we have to assume that if none of the #ifTrue: blocks are entered, and that the receiver itself of type #SicsAccountingAbstractPortfolioEarningCurve may be returned from the method. After analyzing all senders we quickly reach the conclusion that the latter cannot be the case. Therefore we assume that the four conditions cover all possibilities. But the Java compiler will not tolerate the return type #SicsRefUnitOfTime unless all branches return an object of that type.

We can type the method return type as #Object. That will remove the compilation error. The result must be cast to #SicsRefUnitOfTime in senders where required. This reduces type safety and gives in our opinion lower quality code.

Therefore we want to modify the method to meet Java type requirements. If we believe the four branches cover all possibilities, we could indeed remove the last #ifTrue: test and return the contents of the block directly. However there is a slight possibility that there could be introduced an error in the Smalltalk version later. If we remove the test the Smalltalk and Java versions will be inconsistent. Instead we add this statement at the bottom of the method:

```
self throwUnsupportedCondition: 'Invalid state'
```

This is translated as:

```
throw new CnuUnsupportedConditionException("Invalid State ");
```

This removes the compilation error, and ensures that both Java and Smalltalk fail consistently.

4.7 Casting

4.7.1 CAST: nodes

One may manually add casts to typed nodes by setting the CAST flag. This is done by beginning the type symbol with uppercase CAST, e.g. #CAST:String.

4.7.2 Problems with #SELF/SEFLINST

The method #copy is typed as #SELF, this means that the return value will be of the same type as the receiver. However this is a generic method in Smalltalk and does therefore not have to be redefined on all subclasses. It is for instance defined in SicsExchangeRate>>#copy, and typed as #SELF. So the signature in Java is public SicsExchangeRate copy(). When #copy is sent to a receiver of the subtype #SicsDayRate, the Java compiler will assume that the result is of type #SicsExchangeRate, based on the method signature. We know that in fact the type is #SicsDayRate. In such cases Java code generator will insert a downcast.
There is a similar casting requirement for #SELFINST, in the case of constructors in meta classes, described in section 2.5.4.

As for the #copy example above, the solution is quite different for base classes such as String. As this function is not available in the Java class, #copy message sends are instead mapped to the method static <T> T clone(T obj) defined in CopyUtilities, with the original receiver as argument. Details on why this is done are given in section 11.4. For now let’s focus on the typing issue. The method signature for clone(T) uses a generic type T for both return type and parameter. So in Java the return type is given by the parameter type. And so there is no need to cast the return value in Java, as the transformed message expression in a sense preserves the #SELF return type of the original expression.

4.8 Java Generics Syntax

An important addition to the Java Language in version 5 (1.5) is generics. See (Bracha 2004) for a brief introduction. Generics allow development of functionality reusable for several data types. The most prominent reuse of generics in SICS is the utilization of the new generic collection classes. This has several advantages:

1. Better types in the system, there is no doubt as to what types of elements a collection is supposed to contain.
2. Less type casts in source code (although the compiler will put in casts for the runtime code).
3. Compiler warnings on unsafe access to collections.

In the SICS type system Collections, Maps, Blocks and a few other types as described in section 3.2.4 are translated to specified generic types in Java. E.g. #SequenceableCollection:String is translated as List<String> and #Block:Boolean:Object is translated as OneArgumentBlock<Boolean, Object>.

4.8.1 Problems with Generics and Subtyping

Section 3 “Generics and Subtyping” in (Bracha 2004) demonstrates that for instance List<String> is not a subtype of List<Object>, even though the element type String is a subtype of Object. So an expression of type List<String> may not be assigned to, sent as parameter or returned from a method with type List<Object>. Naturally the same applies for all generic types. I will assume this important subtype relationship is well understood from this point on. If the concrete example above does not seem obvious, refer to (Bracha 2004).

Interestingly enough this problem does not apply to Java arrays. As arrays are type checked at runtime the syntax rules are less complicated. For instance String[] is a subtype of Object[], see for instance (Mitchell 2003) page 397.

Collections with different element types is a quite typical problem in SICS, where we have deep hierarchies and a lot of shared framework code. For instance a persistence framework method might require a Collection<CnuAbstractPersistentDomainObject> parameter, a collection of any kind of SICS persistent object. Senders of the method would in most cases provide a more specific argument, such as Collection<SicsCountry>, where SicsCountry is a subclass of CnuAbstractPersistentDomainObject. This does not compile in Java. The remainder of this section will look at various strategies to solve such situations.
Use of Wildcards
The solution for cases like the one above according to (Bracha 2004), is to use a bounded wildcard. The framework method should use the type Collection<? extends CnuAbstractPersistentDomainObject>. As Collection<SicsCountry> is a proper subtype, the code compiles without any unsafe warnings. There is support for upper bounded generic types in the type system, see section 3.2.4. Downcasts of upper bounded generic types are also possible, but give an unchecked warning. So this piece of code is unsafe but compiles:

```java
public void downCastItems(
    final List<? extends CnuAbstractPersistentDomainObject> parameter)
{
    final List<SicsCountry> countries=(List<SicsCountry>) parameter;
}
```

With the existence of unchecked warnings in the code, the Java compiler will no longer be able to guarantee type safety. Still, as the further discussion below will reveal, we do not consider it feasible within a reasonable amount of time to assign all types to a system as large and complex as SICS without any unchecked warnings.

In addition, compile time warnings are only part of the picture. Some SICS framework components rely heavily on reflection. Java 1.5 generic syntax is a compile time feature only. There are no traces of the formal parameters in the compiled byte code. So for instance if a method with a parameter of type Collection<String> is invoked by reflection, the element type of the actual parameter is not type checked.

As a conclusion we realize that the way we use Java generics syntax in SICS offers no guarantees against type cast errors involving generic types in SICS. Still, it has several desirable features such as, increased code quality, simplicity and documentation.

CollectionUtilities narrow/widen
There is however a serious drawback to the use of wildcards. If a wildcard is specified as the generic formal argument of a type, invoking any method where a parameter is of the formal argument type will give a compilation error. The Collection method boolean add(E e) is an example of such a method. Therefore, this code does not compile:

```java
public void addObject(Collection<? c)
{
    c.add("value");
}
```

See (Bracha 2004) for a demonstration of why that is not allowed. So in our example we have to be absolutely certain that no elements are added to the Collection within the scope of the declaration where wildcards are used. Unfortunately our toolset does not perform adequately sophisticated semantic analysis to detect such cases.

Even with manual type assignment it can be very difficult and time consuming to use wildcard generic types correctly, as the call graphs within SICS can be very deep. It is not a feasible solution for the SICS system as a whole as it we believe it would take too much effort. In fact even if a similarly complex system was written from scratch, it is in my view difficult to get all generic types right and completely avoid unsafe warnings. It is very difficult to foresee the consequences of local type decisions, with complex call graphs. For instance
To conclude we do not use upper bounded generic types by default, as it is hard to predict where it may cause problems. Still, the type model includes upper bounded generic types, and may be manually assigned to solve complex type problems in an elegant manner.

We still appear to be stuck with impossible supertype – subtype transitions in the generated code. We will look at a way around this problem. First of all, the Java generation tool naturally assumes that all type information either manually assigned or automatically deducted is correct. Of course there will be instances of incorrectly assigned types, but it is not the task of the Java generator to question or analyze the types. So for instance if the type on the left side of an assignment is #OrderedCollection:String and the right hand side is of type #OrderedCollection:Object, we have to trust the left hand side. This does not mean that the type of the right hand side is incorrect. There may be a number of reasons why the right hand side has a less restrictive type, for instance it is defined at a different level of abstraction, imposed or inherited method signature within a hierarchy, etc. So we assume that the types on both sides of the assignment are logically correct. So our job is to make the right hand side compatible with the left hand side. Ok, let’s try to simply cast it.

```
List<Object> general=Collections.EMPTY_LIST;
List<String> specific;
specific=(List<String>)general;
```

**Example 18 Invalid generic subtype cast**

The code in example 18 gives the compilation error: “Cannot cast from List<Object> to List<String>.”. The compiler will only allow casts to a subtype, an interface or a supertype. As List<String> is neither of these the cast above will not be allowed. In fact the only generic types for which the cast (List<String>) compiles (with a warning) is the wildcard type List<?> or the unsafe type (omitting formal argument) List. A way to achieve our goal is therefore to go via a third variable which goes via one of these types. We may use unsafe type List for this purpose, see the example below.

```
final List unsafeList=general;
specific=(List<String>)unsafeList;
```

Now the code is able to compile, but with an unsafe warning as we have previously noted. Although the just in time compiler would most likely be able to optimize the code by removing the redundant assignments above, we see that this reduces the clarity of the original source code and would create unsafe warnings throughout the code base. We have therefore chosen to implement this rather common operation in one utility method, shown below.

```
public static <T> List<T> narrow(final Class<T> tClass, final List<?> wideList) {
    return (List<T>) wideList;
}
```

Note that the method does not actually “do” anything, the parameter is returned unmodified!

Now the unsafe warning will only be raised in the implementation of narrow, not for the senders.
In addition the method signature helps the compiler to decide whether the conversion is a valid “downcast” as the second argument has type as `List<? super T>`. The keyword `super` is used in lower bounded types. `List<? super T>` means a list where the elements must be a super type of T. Trying to narrow a `List<String>` to `List<Boolean>` gives a compilation error, just like casting a `String` to `Boolean` would.

There is also another utility method called “widen” where the argument has type `List<? extends T>`, which is used to “convert” to a list with a more general element type. There is also a method called “narrowToInterface”, where there is no particular relation between the element types, just as the Java compiler will allow cast from any interface type to another interface type.

There are also corresponding methods for `(java.util.)Collection` and `(java.util.)Set`. The Java generator inserts the appropriate method according to the type information available in the Smalltalk syntax tree. The assignment in example 18 is now carried out like this:

```java
specific = CollectionUtilities.narrow(String.class, general);
```

Note that a `java.lang.Class` literal is used to specify the return type to the compiler, for details on this feature refer to (Bracha 2004), section 8 Class Literals as Run-time Type Tokens.

As a conclusion the `CollectionUtilities.narrow` method and its siblings are not particularly elegant or desirable, but they allow us to get around Java generics syntax problems that would otherwise be very difficult to resolve. We have earlier concluded that we do not see it as feasible to translate the entire SICS code base without any unsafe warnings.

### 4.8.2 Checked collections

We have already noted that Java 1.5 generics syntax is a compile time feature. There is no trace of the formal arguments in the compiled bytecode, and so there are no runtime type checks. The class `java.util.Collections` provides type checked collections through the `checkedList(List)` etc. methods. These collections ensure that only objects that are compatible specified element types may be added at runtime.

It would be quite easy to make use of these methods in the Java generator wherever collections are instantiated. For instance the code ‘OrderedCollection new’ of type `#OrderedCollection:String` could be instantiated like this in Java (an example of ordinary instantiation is given for comparison):

```java
//normal unchecked instantiation
new ArrayList<String>();
//checked collection (normal collection wrapped by type checker)
Collections.checkedList( new ArrayList<String>(), String.class);
```

We have however chosen not to use this feature in the translated code, for several reasons. First of all the type checking decision code must be invoked at instantiation time. We may not in all cases know the correct type at instantiation time. Secondly this changes the internal structure of the objects, as the original collection is wrapped. Although this wrapping is completely “transparent”, there could potentially be problems
in reflection based code, type checking code etc. There is also a slight performance overhead by the delegation from the wrapper.

4.8.3 Return type incompatibilities

As we have previously seen #List:String is not a subtype of #List:Object. Assume we have an abstract class A with the method #items with return type #List:Object. Class B extends A, but overrides #items with return value #List:String. As we learned in section 4.6 the return type must be the same type or a subtype of the super implementations return type. By now we know that #List:String is not a subtype of #List:Object. Clearly the code will not compile.

Our options are the same as for the subtype problems we discussed in section 4.8.1. We can type the super implementation with a wildcard type. However we have seen that that is not always an option. The only other solution we have found is to use an actual/formal type as element type, in our case: #List:(Object/String). This gives the signature List<Object> items() in class B, but indicates to the Java generator it may insert a narrow to #List:String where required.

4.8.4 Parameter type incompatibilities

Just as for other parameters in overrides, the type must be an exact match. So in this case we have to use the actual/formal element type solution we proposed for the return type problem above.

4.8.5 Use of generics in blocks

The problem with subtypes and generics is even more profound for blocks. Assume that we have a framework method that requires an argument of the type #Block:CnuAbstractWrapper:CnuAbstractDomainObject, both the formal generic arguments types are the root of their respective hierarchies. Senders of the method must pass an argument block with an argument of type #CnuAbstractDomainObject and returns an object of type #CnuAbstractWrapper.

```java
void wrapperBlock(OneArgBlock<CnuAbstractWrapper,
CnuAbstractDomainObject> wrapperBlock)
{
    //...
    wrapperBlock.value(object);
    //...
}
```

Now let’s assume that one sender of this method has a block with a much more specific signature #Block:CnuDistributedTaskStatusWrapper:CnuDistributedTask. Both the return and argument type of the block are concrete leaf subclasses of the corresponding abstract root classes defined in the framework block argument type. We assume the concrete block is body is in the sender method, and return and argument types can be exactly inferred from the local syntax tree. Given this information the Java generator declares a block with the following signature:

```java
new OneArgBlock<CnuDistributedTaskStatusWrapper,CnuDistributedTask>()
{
    public CnuDistributedTaskStatusWrapper value(CnuDistributedTask arg)
    {
        //...
    }
};
```
Now from our previous discussion about problems with generic subtypes, we know that the concrete block is not a valid argument for the method wrapperBlock as it is not type compatible. How can we make the actual argument block compatible?

Let’s first look at the return type. As for the other subtype problems we could use an upper bounded wildcard for return type, in our case:

```java
features extends CnuAbstractWrapper).
```

However that is not really necessary. If the return statement of the actual block is compatible with `CnuAbstractTaskStatusWrapper`, it is certainly also compatible with the supertype `CnuAbstractWrapper`. If this was not the case, there would surely be a type error somewhere. So we adapt the return type signature of the local block to fit with this requirement.

What about the argument type? We could try to impose the abstract supertype on the block argument. But that will give compilation errors if methods only defined in the concrete subclass are sent within the block.

What about using a bounded wildcard type for the argument? Clearly that is meaningless. I will be impossible to invoke the `value(...)` method on the block, which is the sole purpose of using a block.

So how about doing something similar to `CollectionUtilities.narrow` for blocks? We have decided against that. Instead we type the block argument as a formal/actual type.

So in our concrete example the block argument should be of type `#CnuAbstractDomainObject/CnuDistributedTask`. The formal type is used in declarations in the Java source code, so in out example the block will appear as `OneArgBlock<CnuAbstractWrapper, CnuAbstractDomainObject>`. In other words identical to the signature in the framework method. However the actual type is available for the type inference algorithms to know what type the argument really is. In our example this means that if a message is sent to the block argument which is not defined for `#CnuAbstractDomainObject`, we know we can downcast the argument towards `#CnuDistributedTask` to find an implementation.

The problem with of abstract block requirement and concrete actual blocks is quite common in SICS. Admittedly much of the benefit of generics is lost if we have to cast the argument in the block body. So at one point we considered simply giving up on generics for blocks. A one argument block type would simply appear as `OneArgBlock`. However in the end we decided that the benefits outweighed the problems with generics. For instance in our example we felt that `OneArgBlock<CnuAbstractWrapper, CnuAbstractDomainObject>` communicates more intent that `OneArgBlock<Object,Object>`.

### 4.9 Argument Types

The Java 1.5 generics syntax even allow us to implement argument types. For instance the return type of the method `CnuAbstractBusinessProcess#registerObject` is typed as `#ARG1` as the method should return the argument or a proxy for it and the argument is typed `#CnuAbstractPersistentDomainObject`. The Java signature for this method translated as:

```java
public <ARG1 extends CnuAbstractPersistentDomainObject>
registerObject(ARG1 arg)
```

So if the message is sent with an argument of type `#CnuNote`, the whole message expression is also of type `#CnuNote`. The type parameter ARG1 is defined in the method signature, for further details see (Bracha 2004), section 5 Generic Methods.
5 Conditional expressions

As we have previously touched upon, all control structures in Smalltalk are in fact methods. Some may in fact not actually be carried out as method calls when the byte codes are interpreted by the virtual machine, but from a grammar point of view they are still clearly method calls, or message sends in Smalltalk terminology. We also know that Smalltalk methods always return a value. If there is no explicit return, the receiver itself will be the return value. As a consequence of the two former: conditional statements return a value! This means that such expressions in Smalltalk:

- Can be assigned to a variable
- Can be returned from a method
- Can be the receiver of other messages
- Can be the argument of a message

Although the above still applies, there are a few examples of Smalltalk syntax that one in practice hardly ever use the return value of. For instance, it is not immediately evident what the return value of \#whileTrue: or \#timesRepeat: should be.

Now, what is the consequence for translating the Smalltalk code to Java? We will in the following discussions focus on \#ifTrue:ifFalse: as that is the most common control structure. As we have already noted \#ifTrue:ifFalse: expressions return a value. In Java however an \if/\else statement does not return a value. Interestingly there is an alternative to \if/\else in Java that does return a value: the ternary conditional operator. Example 19 below shows how the ternary conditional operator can be used in all the situations described above.

```java
String ternaryExamples(Object value) {
    // assignment
    final String var = value == null ? null : value.toString();
    // receiver and argument
    (value == null ? null : value.toString()).equals(
        value == null ? null : value.toString());
    // return from method, note: nested ternary
    return value == null ? (var == null ? "" : var) :
        value.toString();
}
```

**Example 19 Java ternary conditional operator**

Initially we attempted to use the ternary conditional operator in the generated code whenever possible. In the end however we chose not to use it for the following reasons:

- It is in our opinion more difficult to read, for instance look at the nested example above
- In SICS, most \#ifTrue:ifFalse: statements cannot be translated into the ternary conditional operator, as they are too complex. For instance variable declarations, assignments, more than one statement etc. are not allowed. Therefore only a small portion of \#ifTrue:ifFalse: statements would be translated into the ternary conditional operator. Due to this, we felt that it was better to translate all \#ifTrue:ifFalse: statements in a consistent manner.

Back to the original problem, what should we do to an \#ifTrue:ifFalse: expression in the different situations above? First of all we need to classify the different kinds of situations, as the solutions are slightly different for the different cases. The discussion
will touch upon a few implementation details of the Java generator, but this should also be helpful for understanding the problem.

The Java generation occurs in a top-down traversal of the parse nodes, and the nodes have no reference to their parent. The generation routine therefore has to signal to the nodes below what kind of situation we are in. This is achieved by adding a helper object onto a stack. The helper object communicates in which of the four kinds of contexts the code is generated. This can be either:

- Return context, used for return statements (marked by a leading ^ caret )
- Assignment context, is used if we are generating code for the right hand side of an assignment
- Expression context, is used if we are generating code for the receiver or argument of a message send
- Void context is used if the result of the expression is not used

5.1 Return
Let's look at a return example, this kind of code can be seen quite frequently in SICS:

```java
ifTrueIfFalseReturn

^self collection isNil ifTrue: [>()] ifFalse: [self collection]
```

Example 20 Conditional return from method

We see that the entire statement in example 20 including the #ifTrue:ifFalse: expression is returned. One solution that would work in all the situations described above, is to use a temporary variable to store the value of the branches and return the variable that immediately after the if/else statement. In fact in the prototype Smalltalk to C++ translator described by (Wang, Yang et al. 2001), there is extensive use of temporary variables. It is argued that that is not a problem as the compiler should be able to remove any redundant variables.

```java
public List<Integer> ifTrueIfFalseReturnWithTemp() {
    List<Integer> tmp = null;
    if (this.collection() == null) {
        tmp = EMPTY_LIST;
    } else {
        tmp = this.collection();
    }
    return tmp;
}
```

Example 21 Java conditional return statement with temporary variable

The experienced Java developer will have noticed that this is not optimal, as it introduces an extra temporary variable and complicates the method in a way that the original intent of the Smalltalk statement less evident. The natural thing would of course be to make sure that the two branches return separately as in example 22.
```java
public List<Integer> ifTrueIfFalseReturn() {
    if (this.collection() == null) {
        return EMPTY_LIST;
    } else {
        return this.collection();
    }
}
```

**Example 22 Java return statement carried out in conditional branches**

I will briefly give some details on how this is implemented:

When the Java generator enters a return statement, a return context helper object is pushed on the stack to signal this fact to the child nodes. It is up to the subnodes if and how they react to this fact. In the case of `#ifTrue:ifFalse:` the node itself knows that method returns are best taken care of within the branches. It will therefore make sure that both its branches return. This responsibility would be transferred down if there were deeper nested branches in the example above. After it has added return to both branches it sets a flag on the helper object to signal that return is taken care of. Finally when control comes back to the return statement node, it will consult the helper object whether it is necessary to add an explicit return.

All the helpers objects also respond to exceptions being thrown and explicit returns within the branches, as the two also have consequences for the value being returned. In the cases where there is a type requirement for the value returned, that is also specified on the generation helper. This can be used to perform casts or conversions in the sub-node.

### 5.2 Assignment

The solution for assignments is quite similar to that of return. Let’s use the same expression as in the return case in an assignment:

```java
| col |
| col := self collection isNil ifTrue: [#()] ifFalse: [self collection].
```

**Example 23 Conditional assignment**

The implementation follows the same pattern as for return. A helper object that represents the assignment is pushed onto the stack. The helper object keeps a reference to the variable. The `#ifTrue:ifFalse:` parse node sees itself best fit to perform the assignment. It picks up the variable from the helper object and performs the assignment in both branches. It then sets a flag on the helper object to signal that the assignment has been carried out. Finally the assignment expression then knows whether the assignment has been taken care of or if it should output code to assign the variable. The resulting Java code is shown in example 24.
public void ifTrueIfFalseAssignment() {
    List<Integer> col = null;
    if (this.collection() == null) {
        col = EMPTY_LIST;
    } else {
        col = this.collection();
    }
}

Example 24 Conditional assignment in Java

5.3 Expression context
The solutions to the foregoing return and assignment examples were pretty intuitive. The following example may seem more confusing to the average Java developer. In this case it is not so trivial to find a Java counterpart. The example uses the same conditions as the foregoing, with a slight difference in the argument to be able to distinguish it from the receiver. Though admittedly example 25 looks quite bizarre, it is in no way more complex very commonplace SICS source code.

ifTrueIfFalseReceiverAndArgument

^((self collection\(^1\) isNil\(^2\)
  ifTrue: [#()]
  ifFalse: [self collection])
  includesAll\(^7\) (self collection\(^4\) isNil\(^5\)
  ifTrue: [#(1 2 3)]
  ifFalse: [self collection])

Example 25 Conditional receiver and argument

Message expressions are marked with a number to indicate execution order. The message expressions within logical branches are not numbered. The same numbers are visible in the parse tree below. A simplified parse tree for the method expression looks like this:
Diagram 7
Diagram 7 shows the parse tree of message expression with `#ifTrue:ifFalse:` in both receiver and argument. The diagram is focused on message expressions. The message expression nodes are distinguished by a message selector prefixed with `#` and a number that indicates the execution order.

Let's give a brief walkthrough of the flow, experienced Smalltalk programmers may skip this:

Receiver
1. The message `self collection` is sent
2. The result is evaluated for `nil` by sending the message `isNil`
3. `#ifTrue:ifFalse:` is sent to the result, if `true` the expression evaluates to an empty array, for `false` to the message expression `self collection`.

Argument
4. The message `self collection` is sent
5. The result is evaluated for `nil` by sending the message `isNil`
6. `#ifTrue:ifFalse:` is sent to the result, if true the expression evaluates to the three element array `#(1 2 3)`
7. The message expression `containsAll:` is sent to the result of 3 with argument from 6 above.

Now, how should this be realized in Java? We have already concluded that we prefer not to use the ternary conditional operator. For the assignment and return cases the easiest solution was to leave the responsibility to the conditional branches. Is it also an option in this case? In our case there are two branches for the receiver and two branches for the argument. This gives a total of 4 combinations of receiver and argument. We would have to repeat the message send 4 times. In addition one of the if-tests must also be duplicated in both branches of the outer if test. Clearly this would not give optimal code. In addition traces to the original Smalltalk logic are less evident. Therefore it seems like the best solution is to introduce a temporary variables to store the expression values. The Java generator produces the code shown in example 26:
public Boolean ifTrueIfFalseReceiverAndArgument() {
    List<Integer> list = null;
    if (this.collection() == null) {
        list = EMPTY_LIST;
    } else {
        list = this.collection();
    }
    Collection<?> col = null;
    if (this.collection() == null) {
        col = asList(1, 2, 3);
    } else {
        col = this.collection();
    }
    return (list).containsAll(col);
}

Example 26 Java code for conditional receiver and argument

Implementation details

Message expressions are generated according to Smalltalk execution order. First the receiver is generated, then the arguments one by one. The same procedure applies in each of these cases:

First we push an expression context helper object onto the stack. The helper object keeps a reference to the original expression sub-tree and a reference to a required type. The required type for the receiver is the type the same as that of the original expression. For arguments the required type is given by the argument types in the method signature.

The actual receiver or argument node may decide that it is too complex to be evaluated inline. If that is the case, a new temporary variable is created. Then the expression value is assigned to it, and returned in place of the original expression to the enclosing context. As can be seen above the declaration and assignment of the temporary variable must be done prior to its use in the message expression. As the numbers indicate, the same execution order as in Smalltalk is preserved, but the order of source code is different.

The example of return and assignment were specific to #ifTrue:ifFalse:. The use of a temporary to hold an expression value that we just saw is in fact required in many cases. For instance it applies to all mappings that transform the original expression to more than one statement.

There are also a few other interesting observations to be made:

1. The type of the receiver is that of the original expression List<Integer>, but the type of the argument is Collection<?> as that is the requirement of the method containsAll in Java. The argument is typed as general as possible to avoid redundant type casts.

2. The example also shows a shortcoming in the Java generator. In the last statement there are parentheses around the receiver list. This is a consequence of the algorithm outlined above, the enclosing context is not aware that the contents of the receiver expression will be substituted in the latest minute.

There was actually a subtle but very serious bug present in the Java generator for more than a year. As we have previously hinted, the temporary variables and extra statements before the message expression are introduced on a need-to basis.
So what happens if we have a receiver that can be evaluated inline but an argument that requires set up as in example 27.

```plaintext
simpleReceiverIfTrueIfFalseArgument

^self collection^ includesAll:^ (self collection isNil ifTrue: [ #(1 2 3) ] ifFalse: [ self collection ])
```

**Example 27 Simple receiver but conditional argument**

Originally the argument’s branches would be evaluated first, due to the need for a temporary, but the receiver was evaluated as part of the message expression at the end. That is not the same execution order as in Smalltalk! As it took more than a year before someone spotted the bug, probably indicates that it is very seldom that the execution order of the receiver and argument actually does matter. To make sure that the receiver is evaluated first, the code is now generated like this:

```java
public Boolean simpleReceiverIfTrueIfFalseArgument() {
    List<Integer> list = this.collection();
    Collection<?> col = null;
    if (this.collection() == null) {
        col = asList(1, 2, 3);
    } else {
        col = this.collection();
    }
    return list.containsAll(col);
}
```

As can be seen, a temporary must also be used for the receiver, to be able to execute before the argument.

### 5.4 Void Context

If the result of an `ifTrue:ifFalse:` or any other message expression is discarded, in other words there is not attempt to assign, return or otherwise use the value, a void context helper object is pushed onto the stack. In contrast to the other scenarios, this does not require any special action on the generated Java code, but in some cases it greatly simplifies translation. For instance in the mapping for `Collection>>#add:` we have to cater for the fact that in Smalltalk, `add:` returns the argument being added, but in Java `Collection.add(T)` returns a boolean that indicating whether the collection was modified. However if there is no attempt to use the return value, we can generate Java code without any modifications. So the second mapping for `add:` is an optimization for this case:
Notice how the predicate checks the current type of generation helper object, and how the _java_ prefix is used to prevent endless recursion.

5.5 **ifTrue::ifFalse: in non void contexts**

From time to time there are cases where the return value of an **ifTrue: or ifFalse:** expression is used. In cases like this one might wonder what the return value is if the condition fails. **True>>#ifTrue:** is implemented like this:

```
ifTrue: alternativeBlock
```

```
^alternativeBlock value
```

**False>>#ifTrue:** is implemented like this:

```
ifTrue: alternativeBlock
```

```
"Since the receiver is false, answer nil."
```

```
^nil
```

We see that the expression should return nil in the failing case. We must therefore add a nil expression in Java. This is easiest achieved with a mapping for **ifTrue:** in a void context, there is also a corresponding mapping in the **ifFalse:** case.

```
ifTrue: alternativeBlock
```

```
"@MAPPING (29.04.2005 12:03:33)"
```

```
this ifTrue: alternativeBlock ifFalse: [nil]
```

**Picture 3**

**Mapping for #ifTrue: in non void case**

Due to this mapping there will be added an else clause in Java that returns or assigns nil in such cases.
5.6 Shortcut Boolean evaluation

In Smalltalk shortcut Boolean evaluation is achieved through the keyword messages #and: and #or:. The methods require a Boolean receiver and a block that evaluate to a Boolean result. The corresponding Java operators && and || require two Boolean expressions as operands. Not all the block arguments in and: and or: statements in Smalltalk can be translated into simple Boolean expressions in Java however. Examples of blocks that are too complex to be part of shortcut Boolean evaluation:

- Have more than one statement
- Declare variables
- Contain conditional statements
- Are transformed in a way that includes any of the above

The Java generator tool includes a recursive procedure that will test whether a block is too complex for shortcut Boolean evaluation. The test is used as predicate in the mappings for these cases. For instance and: is mapped like this in the complex case:

```smalltalk
and: aBlock
    this ifTrue: aBlock ifFalse: [false]
```

So for instance if used in an assignment, the result in Java looks like this

```java
if (<condition1>) {
    var = <condition2>;
} else {
    var = false;
}
```

5.7 Conditional loops

The block method #whileTrue: is used to evaluate a block argument as long as the receiver block evaluates to true. #whileFalse: is similar but requires the receiver block to evaluate to false instead of true. I will use #whileTrue: as example in the remainder. The syntax and semantics of #whileTrue:

```
[predicateBlock] whileTrue: [doBlock]
```

In each iteration the predicateBlock is evaluated first, followed by the doBlock. This is fairly straightforward to map to a Java while loop:

```java
while (<predicateBlock>) {
    <doBlock>;
}
```

But we encounter the same problem as for #and: if the predicate block is too complex. So we can use the same predicate in a mapping which transforms the code quite dramatically. The result is not particularly elegant in Java, see example 28.
while (true) {

<predicateBlockStatements>;
if ({predicateBlockResult}) {
  <doBlock>;
} else {
  break;
}
}

Example 28 #whileTrue: translated to Java

Similarly as for expression context predicateBlockStatements represents any declarations and statements that must be generated prior to the evaluation and predicateBlockResult the part of the predicate that is simple enough to fit into an if test. For instance assume the predicateBlock contained the Boolean expression in example 25. See how it is translated to Java in example 26. Only the very last statement “(list).containsAll(col)” would be valid syntax for the if-test. All the statements leading up to that must be executed prior to the if-test.

5.8 Iteration loops

Smalltalk has several constructs that allow a block to be evaluated a specified number of times. A few examples are given below.

| 5 timesRepeat: [...]. |
| 1 to: 5 do: [:num | ...]. |
| “same as the previous, but loops over an instance of Interval” |
| (1 to: 5) do: [:num | ...]. |
| 1 to: self max by: 2 do: [:num | ...]. |
| “same as the previous, but loops over an instance of Interval” |
| (1 to: self max by: 2) do: [:num | ...]. |

Example 29 Smalltalk iteration based loops

These constructs of example 29 are quite straightforward to map to Java for loops. There is however one crucial difference. In the last two examples the iteration occurs from number 1 by 2 to the upper limit given by answer of self max. In Smalltalk the integer value of self max is passed to the looping construct before iteration begins. If we created a Java for loop with predicate “num <= self.max()”, the max method would be invoked on every iteration. If the value returned from max changed during the course of the iteration, the translated Java would behave inconsistently with Smalltalk. Therefore, in cases like this, we must store such values in a final temporary variable which we use in the iteration. This includes message expressions, instance variables and local variables which are assigned within the loop block.
6 Cascaded message expressions

Cascaded messages are a nice feature of Smalltalk. For programmers unfamiliar with Smalltalk cascaded message expressions, refer to (NCITS 1997), section 3.4.5.3.3 Cascades. A cascaded message expression has one common receiver, but a sequence of messages sent to that receiver.

If the receiver is anything but a variable, a global reference or a constant literal we need a temporary variable in Java to store a reference to the receiver between the messages sends. During translation each message pattern is converted to a message expression with the temporary as receiver. The last message expression gives the return value, unless as in most cases it is the message #yourself which will return the receiver. Examples are given of cascades in the different contexts. Here is a typical cascade returned from a method

```
cascadeReturn

^OrderedCollection new
  add: 1;
  add: 2;
  yourself
```

We see that we need a temporary variable in Java

```
public List<Integer> cascadeReturn() {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(2);
    return list;
}
```

An expression case:

```
cascadeExpression

self collection: ( 
    OrderedCollection new
    add: 1;
    add: 2;
    yourself)
```

Is translated as:

```
public void cascadeExpression() {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(2);
    this.collection(list);
}
```

Note how the temporary variable is given type based on the type of the receiver, not on the requirements of the argument type as in the ifTrue:ifFalse: expression example. This is done because of the high probability that the the cascaded messages might
have more specific type requirements than the argument type of the enclosing expression.

An assignment example:

```java
cascadeAssignment

| col |

col :=
    ArrayCollection new
    add: 1;
    add: 2;
    yourself
```

Is translated as:

```java
public MartinSampleClass cascadeAssignment() {
    List<Integer> col = null;
    col = new ArrayList<Integer>();
    col.add(1);
    col.add(2);
    return this;
}
```

Note how the variable being assigned to is used within the cascade. The generator will do this to save a temporary variable, unless:

- The variable is referenced within the cascaded expression, in which case it might erase the original value too early
- It is an instance variable, where guaranteeing that the above may not occur is too complex. For instance a message within the cascade might directly or via other messages modify the value of the instance variable.
7 Array Indexes

Arrays and other index-able objects begin from index 1 in Smalltalk. In Java on the other hand, the elements of arrays begin from 0 as in C. So if we use standard Java collections, the translated code will store and retrieve indexed elements from incorrect positions. So we have to take this into account in the generated code.

The first idea that comes to mind for resolving this problem is quite intriguing. Simply ignore index 0 of all indexed structures in Java. The application code should only be concerned about elements from index 1 and up. But we quickly come to realize that this approach might do more harm than good. Here are a number of reasons:

- The size of all lists must be one larger than in Smalltalk. This will affect all code that relies on the size of lists
- Standard iterators and the new enhanced for loop would still include the first elements
- It would also be impossible to impose this behaviour on existing Java classes and libraries. For instance strings in Java are indexed from 0
- The resulting Java code would be difficult to understand and maintain

The only feasible solution is therefore to generate code according to Java conventions, realizing that the index values in code represent the Smalltalk index. It is therefore evident that code which accesses items cannot be translated as is. The solution is to subtract one from the index for all access by index. Likewise one must be added to the index returned by methods that report indexes, such as #indexOf:. A few examples of the two and how this in some cases produces rather clumsy code are given below.

![Picture 4](image1.png)

**Picture 4**
1 is subtracted from the index in the mapping for `SequenceableCollection>>#at:`.

![Picture 5](image2.png)

**Picture 5**
Likewise 1 is added to the index returned by `SequenceableCollection>>#indexOf:`.
A few examples affected by the mappings above are given below:

```
indexAccess

self collection at: 2. "Access by constant index, notice how the
generator computes the result directly"
^self collection indexOf: 3 "Retrieve index of element"
```

Is translated as:

```
public int indexAccess() {
    this.collection().get(CollectionUtilities.sizeOf(this.collection()) - 1);
    /* Access by expression */
    this.collection().get(1);
    /*
     * Access by constant index, notice how the generator computes
     * the result directly
     */
    return this.collection().indexOf(3 /* Retrieve index of element */ + 1);
}
```

Notice how the constant 1 is added and subtracted inline. In the second case, we can
easily compute the new constant as part of Java generation, this produces
this.collection().get(1); instead of this.collection().get(2 - 1), which would look quite
ridiculous.

Similar index manipulation is also required for strings.

```
urlQueryPart: anUrl

| index |

anUrl copyFrom: (anUrl indexOf: $?) to: anUrl size. "Index retrieval
immediately followed by access: + 1 - 1 not neccesary"

index := anUrl indexOf: $? . "But the generator is not able to
optimize when the index is stored in a variable"

^index > 0 ifTrue: [anUrl copyFrom: index to: anUrl size] ifFalse: [''].
```

Is translated as:

```
public String urlQueryPart(final String anUrl) {
    anUrl.substring(anUrl.indexOf('?'),
    StringUtilities.lengthOf(anUrl));
    /* Index retrieval immediately followed by access: + 1 - 1 not neccesary */
    int index = anUrl.indexOf('?') + 1;
```
/*
 * But the generator is not able to optimize when the index is
 * stored in
 * a variable
 */
if (index > 0) {
    return anUrl.substring(index - 1, StringUtilities.lengthOf(anUrl));
} else {
    return "";
}

So we have seen that the mappings are able to consistently manipulate indexes in a predictable manner. However the Java generator does not possess sufficient semantic analysis capabilities to figure out the intent of the code. In some cases this gives rather Java-unlike code, like in the example below:

iteratingByIndex

1 to: self collection size do: [:index |
    (self collection at: index) even ifFalse: [self collection removeAtIndex: index]]

Which is translated as:

public void iteratingByIndex() {
    int intVal = CollectionUtilities.sizeOf(this.collection());
    for (Integer index = 1; index <= intVal; index++) {
        // Test whether number is even
        if (!((this.collection()).get(index - 1) % 2) == 0)) {
            this.collection().remove(index - 1);
        }
    }
}

Clearly this a lot more complicated than a standard for loop to iterate over a list or an array in Java, such as: for(int i=0;i<this.collection().size; i++). The only problem is that the Java generator does not know that the index is only used for array iteration. As far as it's concerned the loop index might be used for anything, so it must preserve the same value as in Smalltalk.

The experiences regarding index manipulation so far in the project are very good. There has been much less problems regarding this than one might fear. On the down side, as one may see above the code may clearly confuse Java developers. It is also a fear that in the future when SICS is maintained in Java, there will be a mixture of index values with Smalltalk and Java semantics. Fortunately the vast majority of index manipulation occurs within a very limited scope, so it is easy to figure out the semantics by looking at the code. For cases where the generator produces suboptimal or Java-unlike code like we have seen above, perhaps Java static analyzers or manual effort could be used to clean up the code on the Java side.
8 Blocks

Blocks are essential to Smalltalk. All conditional processing is achieved through blocks. The Java programming language has no direct counterpart to Smalltalk blocks. Handling of blocks is one of the most challenging and interesting issues any translation from Smalltalk to Java must tackle.

8.1 Previous research
Several strategies for bringing Smalltalk blocks to Java have been developed.

8.1.1 Block syntax for Java
(Al-Ekram and Askari) propose a Smalltalk like syntax for blocks in Java. This is done by extending the existing Java syntax with a custom developed syntax for blocks. This requires the use of a source code transformer to pre-process the source code to normal Java syntax before compilation. The blocks are in fact implemented as Java inner classes, the runtime implications of this are not discussed.

We will later see that Java syntax for implementing blocks with anonymous inner classes is neither compact nor easy to read. It is therefore tempting to resort to this dedicated block syntax. For the SICS4 project however there are some practical considerations against it. For instance debugging information about line numbers could become misleading.

8.1.2 Runtime issues for block translation
(Engelbrecht and Kourie 2003) describes a strategy for implementing true Smalltalk block semantics in Java. The solution even includes general solutions for the two predominant runtime obstacles non local return and static scope references see section 8.4 and 8.5. The authors even test out their examples with two Smalltalk to JVM compilers, and demonstrate that neither produces correct behaviour.

8.1.3 Blocks as functions
(Wang, Yang et al. 2001) describes the s2cpp system, a Smalltalk to C++ generator. S2cpp generates functions from Smalltalk blocks. Pointers to these functions are passed to the conditional and loop constructs. s2cpp creates instances of the predefined BlockClass class in the cases where a block object is required. In fact common conditional operators such as #whileTrue: and #ifTrue:ifFalse: are implemented as functions, with function pointers as arguments.

As s2cpp is a translator from Smalltalk to C++ there is however limited applicability for the SICS4 project. Java neither has function pointers or higher order functions. In addition we will see that in the SICS4 project, we prefer to use built in Java conditional operators wherever possible. That gives code which is easier to understand and more effective.

8.2 Potential future support for closures in Java
(Bracha, Gafter et al. 2006) propose adding support for closures in JRE 7 (Dolphin).
The proposal includes a new syntax for closures and addresses runtimes issues such as non-local return and static scope references discussed in sections 8.4 and 8.5. If the Java language and runtime environment is extended according to the proposal, we would be able to translate Smalltalk blocks in a much more elegant manner. However the SICS4 project must be completed before this feature is available, so we have to rely on the current Java platform. One could hope that some code transformation tools for Java would be able to convert anonymous inner classes, to the new compact syntax. However the transformations and manual solutions we have chosen for out of scope return, static scope access and inline blocks are very difficult to reverse automatically, unless the whole code base is regenerated from Smalltalk.
8.3 Support for blocks with anonymous Java inner classes

The Java generator is capable of translating Smalltalk blocks to anonymous inner classes where block objects are required. The number of arguments decides what kind of block class is used. Recall that the type assigned to a block reveals how many arguments it has. The first generic parameter denotes the return type and the following the argument types. The table below gives a few examples:

<table>
<thead>
<tr>
<th>Type</th>
<th>Java class signature</th>
<th>value() method signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Block:Boolean</td>
<td>ZeroArgBlock&lt;Boolean&gt;</td>
<td>Boolean value()</td>
</tr>
<tr>
<td>#Block:Boolean:Object</td>
<td>OneArgBlock&lt;Boolean,Object&gt;</td>
<td>Boolean value(Object)</td>
</tr>
<tr>
<td>#Block:String:string:Object</td>
<td>ThreeArgBlock&lt;String,string,object&gt;</td>
<td>String value(String,Object)</td>
</tr>
</tbody>
</table>

As different classes are used for different number of arguments, it will for instance be impossible to pass a two-argument block to a function which requires a one argument block as argument. Examples of generated code will be given later.

8.4 Static scope references

The block has reference to the immediately enclosing instance (self) (Gosling 2005), instance and class variables, this is also the case in Java. In addition the Smalltalk block has read/write access to temporaries of the enclosing method. That is not possible in Java. Anonymous inner classes in Java can only access arguments and temporaries that are final. In effect the block only gets access the actual value/reference. That is safe as when the variable is final it cannot be reassigned. So the value referenced by the block correctly references the value as it was when the enclosing method was executed. The actual activation record of the method may already be popped of the execution stack. Smalltalk’s activation records on the other hand are in fact garbage collected just as other objects. As a slightly annoying consequence in everyday work, the variables of the enclosing method are not available when debugging inner classes.

8.4.1 Making parameters and temporary variables final

Due to these requirements, method parameters are made final when we generate Java. This is unproblematic as method and block arguments cannot be reassigned in Smalltalk. In addition the Java generator eagerly makes temporaries final based on an analysis if the parse tree. It is also possible to force the generator to make a temporary final if the analysis failed to identify such a case. This is also unproblematic as the compiler will give an error if the user is wrong.

8.4.2 Reference Object

However sometimes blocks assign to temporary variables of the enclosing method. As we will touch upon later this is a situation that we try to avoid by all means, however sometimes there is no other alternative. As one might envision the solution is to introduce one level of indirection. (Yasumatsu and Doi 1995) and later (Engelbrecht and Kourie 2003) propose a generic solution to the problem: Parameters and temporaries of the enclosing method are arranged as elements of an array. The array is final. The block may then freely access the elements of the array by index. For the SICS4 project, we feel that the array access method was quite too difficult to read. It may work better in a source to Java byte code translation. In SICS however write access to temporary variables of the enclosing method is not handled by default. Initially the code is generated as is, in other words the assignment carried out within the block with no special handling. This will problem will be brought to the developer’s attention by a Java compilation error. The developer
must then decide whether using a block object is the best solution in Java. If that is the case then the developer may choose to support write access to the variable by marking it with the comment `REFERENCEOBJECT`. This instructs the Java generator to handle the temporary to do a number of things. First the temporary is made `final`. In place of the original object the variable contains an `AtomicReference`. All read access to the variable is substituted with `variable.get()` and all assignments to the variable are substituted with `variable.set(<value>)`. Note: The class `java.util.concurrent.atomic.AtomicReference` was chosen for this purpose, simply because it is a handy reference object that allows changing the referenced object and it has a generic type signature. It does not reflect any semantics related to concurrency. To sum up, the reference object solution is neither desirable nor elegant but it does the job. Example 30 below illustrates this.

```java
addChildScopeOfCoverWith: aMappedDataObject
   " aMappedDataObject <AbtXmlMappedObject>
   ^<SicsAbstractBusinessScopeOfCover>"

| soc |

(self businessProcessForAddingDefinitionIn: aMappedDataObject)
ifNotNil:
[:bp |
   self
   executeProcessForDescription: bp
   ifFinishedWithCommitDo:
      [:childProcess | soc := childProcess scopeOfCover]].

Example 30 Block assignment to temporary variable

Example 30 shows a method where a block must be translated as an anonymous inner class in Java as it is a callback that may or may not be called at a later point in time. This gives the following code:

```java
public SicsAbstractBusinessScopeOfCover addChildScopeOfCoverWith(
   finalInputElementWSI aMappedDataObject) {
   final AtomicReference<SicsAbstractBusinessScopeOfCover> soc = new
   AtomicReference<SicsAbstractBusinessScopeOfCover>();
   CnuChildProcessDescription bp = this
   .businessProcessForAddingDefinitionIn(aMappedDataObject);
   if (!((bp == null))) {
      this.executeProcessForDescriptionIfFinishedWithCommitDo(bp,
         new OneArgBlock<Object, CnuAbstractBusinessProcess>() {
            public Object value(
               final CnuAbstractBusinessProcess childProcess) {
               return soc
            .set(((SicsBusinessScopeOfCoverCreateDirectorBp) (childProcess))
               .scopeOfCover());
            });
      }
      return soc.get();
   }
}
```

Example 31
Example 31 shows assignment to temporary variable from within block. Notice how the method `get()` must be called wherever the value of `soc` is needed, and `set(Object)` wherever `soc` is assigned a value.

8.5 Non-local return

Smalltalk blocks are capable of returning from the method which they are defined. This is normally part of ordinary control flow such as `#ifTrue:ifFalse:`, iterations etc. However it actually also possible to evaluate the block from an arbitrary level deep down in the stack, to force immediate return from the declaring method. The experienced reader will notice that this is very similar to the control flow of exceptions. Indeed (Engelbrecht and Kourie 2003) propose a general solution to support this behaviour by throwing a special exception, which is caught in the declaring method. In the SICS Java port project, however, we do not generally use this approach for a number of reasons:

- It could easily be broken by or break other exception handling code
- The generated code is very complex and not very intuitive
- Java exceptions should only be used for exceptional cases not as a means for normal control flow.
- Exceptions are also an inefficient means to return from a method, measurements by (Engelbrecht and Kourie 2003) indicate a factor of more than 10 compared to normal method return.

When the generator encounters a block with a non local return it inserts the statement: `"this.outOfScopeReturnInBlock()"`, which will give a compilation error to attract the attention of the developer. The developer must then decide whether it is possible to remove the need for a block or use one of the strategies for non local return which will be presented shortly.

8.5.1 Setting flag in enclosing method

This solution requires some manual effort, and must be carried out per method. The procedure is as follows:

- Add a temporary variable to the method, and mark it as a `REFERENCEOBJECT`
- Replace non local returns with assignments to this variable within the block
- Immediately after the expression containing the block, the enclosing method must test whether the variable has been assigned to and if so return

As with the reference object solution described in section 8.4.2, it would have been possible to implement this behaviour automatically, however since the resulting code is slightly awkward, we want it to be a conscious decision by the developer to use it. Below is an example which is very hard to solve in any other way:

```mit)(* getNewWorksheetID: aWorksheet
  " answers a new WS-ID number "

  self
  worksheetType: aWorksheet type;
  executeProcessForDescription: self class getNextIdentifierChild
  ifFinishedWithCommitDo: [:child | ^childProcess]
  elseDo: [:child | ^nil]```
The two block arguments are in effect framework call backs to the sender. According to the guidelines the method can be rewritten like this:

```
getNewWorksheetID: aWorksheet
" @@REWRITE_TS@@ (01.03.1999 17:18:51) "

" answers a new WS-ID number "

|result|
result := nil.
self
worksheetType: aworksheet type;
executeProcessForDescription: self class getNextIdentifierChild
ifFinishedWithCommitDo: [:child| result := childProcess]
elseDo: [:child| result := nil].
^result
```

In this method there are no statements after the expression with blocks, so there is no need to test whether to return or not. The resulting Java code looks like this:

```java
public SicsAccountingWorksheetIdentifierGetNextBp getNewWorksheetID(
    final SicsAccountingWorksheet aWorksheet) {
    final AtomicReference<SicsAccountingWorksheetIdentifierGetNextBp> result = new AtomicReference<SicsAccountingWorksheetIdentifierGetNextBp>();
    result.set(null);
    this.worksheetType(aWorksheet.type());
    this.executeProcessForDescriptionIfFinishedWithCommitDoElseDo(
        this.getKlass().getNextIdentifierChild(),
        new OneArgBlock<Object, CnuAbstractBusinessProcess>() {
            public Object value(final CnuAbstractBusinessProcess child) {
                return result.set(
                    (SicsAccountingWorksheetIdentifierGetNextBp) childProcess);
            },
            new OneArgBlock<Object, CnuAbstractBusinessProcess>() {
                public Object value(final CnuAbstractBusinessProcess child) {
                    return result.set(null);
                }
        });
    return result.get();
}
```

8.5.2 Non local return by throwing Exception

According to observations done by (Engelbrecht and Kourie 2003) non local return from blocks have several similarities with Java exceptions. In fact there are several examples of such use of blocks in SICS. Although this is not strictly a non local return problem, typical cases also involve non local return. In such cases we resort to Java exceptions. The method `beginTransactionIfError` requires a block that will be evaluated if an error condition occurs. The implementation of the method is replaced
with the method `beginTransactionThrows()` which throws a Java exception under the conditions that would previously evaluate the block. A mapping is then created for the original method:

![Mapping for error block](image)

The mapping will apply to all senders, and inline the block as part of a `#when:do:` statement which is translated as `try/catch`.

### 8.6 In-lining blocks to optimize code

As we have previously established there are quite a few problems associated with implementing blocks as anonymous inner classes, and we therefore want to remove the need for blocks as much as possible. We will illustrate this with a trivial example that neither involves non-local return nor static scope references:

```ruby
#(1 2 3) collect: [:num | num * 2]
```

**Example 32** Common usage of block in Smalltalk

A naive approach to produce the same behaviour in Java:

```java
CollectionUtilities.collect(
    Arrays.asList(1, 2, 3),
    new OneArgBlock<Integer, Integer>()
    {
        public Integer value(Integer num)
        {
            return num*2;
        }
    });
```

Clearly this is not optimal. At this point we are only concerned with the calling syntax. The implementation of `CollectionUtilities` and `OneArgBlock` are irrelevant. We see that straightforward implementation of blocks in Java is achieved by an anonymous class. A number of things should be said about the use of an anonymous inner to solve the problem above:

First regarding the source code:
1. The source code is definitely not very intuitive
2. The syntax is quite messy; a lot of syntactic noise surrounds the two lines with actual content.
3. Depending on formatting conventions there are 3 to 4 levels of indentation in this tiny example.

As we have seen, (Al-Ekram and Askari) propose a more compact notation, we have however concluded not to use it in the SICS4 project. There are also some runtime implications:
1. Code for the inner class will be stored in a separate class file
2. Each invocation of the code above will instantiate an instance of the inner class
3. The instance of the block will reference the immediately enclosing instance, see (Gosling 2005) section 8.1.3. This is not a problem in this particular example, but might in general impact garbage collection in ways not easily realized by the developer if references to blocks are passed around in the application.

These problems are inherent to the use of non-static inner classes in Java. It is however interesting that neither runtime consequence 2 nor 3 is really required in the example above. The behaviour of the block is purely functional and does not depend of the state of the enclosing object. So it would in theory be possible to define the block as a static inner class, and store it in a variable like this:

```java
static class CollectBlock extends OneArgBlock<Integer, Integer> {
    public Integer value(Integer num) {
        return num * 2;
    }
};
public static final CollectBlock COLLECT_BLOCK = new CollectBlock();
```

The block object can then be reused over and over again without any reference back to the sender. The block declaration must, however, take place outside the method. That would make the source code even more difficult to follow.

Going back to the original example we quite easily see that it is easy to produce similar behaviour in Java without need for anonymous inner classes. The solution to the problem in SICS4 is therefore to inline the behaviour of `#collect` in the sending method. The mapping for `#collect` is given below:

```plaintext
collect: aBlock
| transformedList |

transformedList := OrderedCollection new.
this do: [:each | transformedList add: (aBlock value: each)].
transformedList
```

**Example 33 In-line mapping for #collect:**

There are some technical details within the mapping for `#collect` that may be confusing, so a short walkthrough will be given:

1. A new temporary variable called "transformedList" is created for the resulting collection
2. The elements of the original collection are iterated over
3. The result of applying the block on each element is added to the new collection
4. The new collection is returned to the calling context

The result of applying the mapping is shown below:

```java
List<Integer> transformedList = new ArrayList<Integer>();
for (final Integer num : asList(1, 2, 3)) {
    transformedList.add(num * 2);
}
```
We see that the generated code is in fact more compact and a lot easier to follow than the original block based. This is particularly important for Java programmers without Smalltalk background. In addition this solves the potential problems of non-local return and static scope references discussed above. Blocks in-lining is therefore the preferred approach. One might also expect some performance gain by doing the whole operation in-line, however that does not appear to be significant. The numbers below result from executing the **#collect**: example with the different approaches:

<table>
<thead>
<tr>
<th>Approach</th>
<th>One million invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smalltalk (VAST 6.0)</td>
<td>438 ms</td>
</tr>
<tr>
<td>JRE Sun 1.6.0</td>
<td>329 ms</td>
</tr>
<tr>
<td>JRE IBM 1.5</td>
<td>469 ms</td>
</tr>
<tr>
<td>JRE JRockit 1.6.0</td>
<td>923 ms</td>
</tr>
<tr>
<td>Original block based</td>
<td></td>
</tr>
<tr>
<td>Static reused block</td>
<td></td>
</tr>
<tr>
<td>In-lined</td>
<td></td>
</tr>
</tbody>
</table>

Some might argue convincingly that in-lining is a bad practice from a maintenance point of view: the same behaviour is replicated in all the senders! Indeed it should be used with care. The advantage of implementing the behaviour in one place and reusing it elsewhere is obvious. It gives the freedom to modify the implementation as long as the expected behaviour is kept. However for something as basic as **#collect**: it is hard to see how the implementation could ever be modified or optimized in the future. So in this particular example it is a price we are willing to pay in order to get more compact and readable Java code.

### 8.7 References to self and super in Blocks

Smalltalk blocks are not separate classes like Java anonymous inner classes. References to **self** or **super** within blocks therefore naturally refer to the enclosing instance. If method invocations within a Java anonymous inner class was directed to **this** or **super** however, that would mean the anonymous class or superclass, not the enclosing class. We therefore have to prefix **this** and **super** with the name of the enclosing class, for example: `CnuNoteImpl.super` in order to address the enclosing instance. Class names are usually quite long in SICS, this is due to the huge number of classes and the flat namespace structure of Smalltalk. The prefix to **this** may therefore be very long. We have therefore opted to omit **this** prefix to method invocations within blocks translated as anonymous inner classes. That is ok in Java as the methods and instance variables of the lexically enclosing class are also visible to the inner class. There is however a huge risk if the method is also implemented by the inner class: The method will be invoked on the inner class instead. This will not give any compilation error, just wrong behaviour. Of course this also includes methods defined on `Object` such as `equals(Object)`, `hashCode()` or `toString()`. So far in the project this problem has not occurred.
9 Numbers

SICS is a financial application. It is therefore most important to preserve correct semantics for integer and decimal types in Java. This has proven to be quite cumbersome given the available types and operators on the Java platform. Floating point number types are not allowed in SICS. The reader should be aware that the content of this section is highly reflected by application specific decisions. It may serve as inspiration for developing type systems for other financial applications. But it is most probably not directly applicable to other applications.

9.1 Use of numbers in SICS and Smalltalk

A very high level view of the treatment of numbers in SICS will be given. A number value may enter the system from a few different sources: read from the database, entered by the user, or stored as a constant in code. If the number is a decimal (instance of ScaledDecimal), the number of decimal places (scale) is always specified. The scale is given by the database column definition. If the number is entered by the user, there will be a fixed number of decimal places in the user interface number entry field. For constants in code the scale is 0 by default for constants without any decimal part and 15 (which is the maximum) for numbers with a decimal part. It is also possible to explicitly specify the scale by suffixing the value by ‘s’ and the scale as an integer value e.g. 2.5s4.

The computational semantics of IBM VisualAge for Smalltalk may initially seem quite confusing or even hazardous for readers mostly experiences with strongly typed languages, primitive types, integer and floating point arithmetic. We will see that it is in fact both logic and in some cases more reliable than its Java counterparts. More detailed semantics regarding the different operations will be given below. However as a general description, the correct algorithm and result type of a computation is always based on the input types. Note that this is different from the decision to use integer or floating point arithmetic in Java based on the operand types, which is chosen at compile time.

As for number output, the number values are converted/truncated to suit the required format, e.g. a database column or user interface number field. A simplified view of the life cycle of a number value in SICS:

- Enters the system as an integer or decimal
- Is part of computations that dynamically pick the best algorithm and result type
- Is converted/truncated back into an integer or decimal for display or storage to the database

Note that the actual instances of the number types are immutable in Smalltalk, just like in Java.

Note that Integer is an abstract class in VisualAge for Smalltalk. Integer values in the range [-1 073 741 824 , 1 073 741 823] are instances of SmallInteger, and outside the range instances of LargeInteger. The Number protocol defined by (NCITS 1997) defines a comprehensive set of operations. All the number types implement this protocol, although the implementation of each message may be radically different. Numeric values of different types may therefore be used interchangeably.

9.2 Number types in SICS

As we have mentioned there are two kinds of numbers in SICS, decimals and integers.

For decimals the choice of type is simple enough. All decimal values in SICS are instances of ScaledDecimal in Smalltalk. The class (java.math.)BigDecimal provides
arbitrarily precision decimal numbers in Java. For mostly historical reasons decimal values for which we want to use BigDecimal in Java should be typed as #DomainDecimal.

For integer values we need to take the probable maximum value into account. As we have previously touched upon integers in SICS may be either SmallInteger or Largelnteger in Smalltalk. The Java primitive int is a signed 32 bit two’s-complement. We see that the range is sufficient for all Smalltalk SmallInteger values. Therefore if we are sure that a variable will never contain a value outside the int range, we type it as #TechInteger which will be represented as Integer in Java. If however a variable may potentially contain a value that exceeds the int range, we cannot type it as #TechInteger. For instance there are database columns in the SICS data model that allow integers up to 20 characters in length. Such cases must be typed as #DomainInteger which is translated as (java.math.)BigInteger in Java.

Integer and BigInteger both inherit from java.lang.Number, but there is no common interface apart from that. This means that items must be typed as either #TechInteger or #DomainInteger, and all values assigned must be of the same type. This is radically different from Smalltalk where the integer classes and indeed all number classes share the same interface. So even if an integer value is within the int range, it must be an instance of BigInteger.

9.2.1 Conversions
As we have touched upon the Java number types are not easily interchangeable. In addition it can be difficult to consistently use the same number type for a given value all the way from the database to GUI, due to the complex call graphs. Therefore conversion of numeric values from one type to an equivalent value of another number type is required from time to time. The Java generator will therefore insert conversion methods where required. In cases where a value is converted from a BigDecimal or BigInteger to int the conversion will explicitly check for lost precision and throw an exception if that is the case.

9.3 Computations and overflow
9.3.1 Integer overflow
If the result of an addition or multiplication of two instances of SmallInteger, exceeds the range of SmallInteger, the result will be an instance of Largelnteger. In Java however computations with two int operands will always give an int result, even if an overflow occurs. We will see examples of this shortly. There is not even an overflow bit to check for. It is therefore the responsibility of the programmer in each case to choose a sufficiently large number type. Similarly when we assign types in SICS, it is crucial to choose number types that for all cases preserve precision and range.

Knowing the dynamic treatment of number types in Smalltalk, correct type assignment is sometimes a tricky task. Should one only resort to BigDecimal and BigInteger in Java, there would be no risk of overflow. However there are still cases where int is needed, such as for array indexes. In addition working with BigInteger is quite cumbersome and there is an overhead both in terms of memory and instructions compared to the primitive int type. It would also be possible to use a custom utility that temporarily made use of BigInteger during computations, and converted back to int or threw an exception if there was overflow. That would however also incur the same overhead and make things overly complex in most scenarios. Therefore we have chosen to use int where applicable, and BigInteger where we anticipate very large values or due to business rules. An example of overflow is given below:
Example 34 Smalltalk Integer overflow

This translates as:

```
Integer a = 1000000000;
Integer b = 10;
Integer c = a * b; //ERROR! gives 1410065408
```

Example 35 Java integer overflow

9.3.2 Division and Fraction

In C or Java the division operator “/” has very different semantics and implementation for integer and floating point operands. In Smalltalk however integer division is clearly distinguished by the selector #/. Normal division is implemented by the method #/ on all number types. For integers the result type depends on whether there is a remainder. If there is no remainder the result will be either SmallInteger or LargeInteger depending on the magnitude. If there is a remainder, the result will be of type Fraction. As the name implies Fraction is an exact representation of a fraction. As the other number types Fraction implements the Number protocol, and may be used interchangeably with other numbers.

If a Fraction instance is subsequently used in a computation where the result is without any fractional part, the result will be an Integer. Looking back at the initial description of numbers in SICS, we see that Fractions also fit nicely in. At any time there may be Fraction values resulting from computations in the system. When a Fraction value is presented in the GUI or written to the database, it must be rounded or truncated to fit the specified precision.

Smalltalk

<table>
<thead>
<tr>
<th>receiver / argument</th>
<th>SmallInteger</th>
<th>LargeInteger</th>
<th>ScaledDecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallInteger</td>
<td>SmallInteger/Fraction</td>
<td>Integer/Fraction</td>
<td>ScaledDecimal</td>
</tr>
<tr>
<td>LargeInteger</td>
<td>Integer/Fraction</td>
<td>Integer/Fraction</td>
<td>ScaledDecimal</td>
</tr>
<tr>
<td>DomainDecimal</td>
<td>ScaledDecimal</td>
<td>ScaledDecimal</td>
<td>ScaledDecimal</td>
</tr>
</tbody>
</table>

Table 2 Return type matrix for division in Smalltalk.

Java

<table>
<thead>
<tr>
<th>receiver / argument</th>
<th>Integer</th>
<th>BigInteger</th>
<th>BigDecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>BigDecimal</td>
<td>BigDecimal</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>BigInteger</td>
<td>BigDecimal</td>
<td>BigDecimal</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>BigDecimal</td>
<td>BigDecimal</td>
<td>BigDecimal</td>
</tr>
</tbody>
</table>

Table 3 Return type matrix for division in SICS in Java.

It is easy to see that we have not tried to mimic the dynamic return types of Smalltalk divisions in the Java version. The primary reason is the limited interface of the common
Number class in Java. One could envision a solution where all Number values were typed as Number, and all computations would be carried out by generic procedures, that would take the type of the operands into account. For instance division would be carried out by a method with the interface public Number divide(Number, Number). We feel however that this would lead to too much overhead, and not be a good solution for Java in the long run. In addition we want to constrain number types in many cases. For instance the instance variables 'accountingYear' or 'layerNumber' should only contain Integer values. For division of integers it is impossible to know in advance whether there will be a remainder or not. Therefore BigDecimal is used for the result in all cases. BigDecimal correctly represent integer values, and represent an approximation of fractional values. The method NumberUtilities.divide is defined for all number types. The result is represented in IEEE 754R Decimal128 format (34 digits). The largest persistent numeric values in SICS may be up to 22 digits. The chance of rounding errors should therefore be very slim. Below is a simple demonstration of the behaviour of division in Java.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.</td>
<td>5.</td>
<td>Fraction (5/3)</td>
<td>SmallInteger (15)</td>
</tr>
</tbody>
</table>

Example 36 Integer division in Smalltalk

```plaintext
Integer a = 3;
Integer b = 5;
BigDecimal c = NumberUtilities.divide(b, a);
//c == 1.66666666666666666666666666666667
BigDecimal d = NumberUtilities.multiply(c, 9);
//d == 15.00000000000000000000000000000003
```

Example 37 Integer division in Java

As a matter of fact floating point representation would give a more effective and precise result in the concrete example above. However the Java generator does not perform sufficiently sophisticated static analysis of the code to predict that the value c will immediately be part of a multiplication. Therefore we still prefer to keep intermediate results as BigDecimal even for division. The same computation using floating point arithmetic is shown below:

```plaintext
int a = 3;
int b = 5;
double c = ((double)b) / a;
//c == 1.66666666666666666666666666666667 (string representation of double)
double d = c * 9;
//d == 15.0 (string representation of double)
```

9.4 Project experiences

Given the big differences in the computation model in Java and Smalltalk, I must admit I was quite nervous to see whether our computation model produced the identical results under all circumstances. Incorrect calculation results are of course the most critical error that may occur in a system like SICS. At the time of writing we are more than halfway through a complete version test of the system. As part of the version test, the business analysts run various comprehensive calculation jobs in the system. I am not aware of any incorrect calculations so far.
10 Lacking features in Java

The Visual Age Smalltalk class library is extensive and rich in functionality. If we for instance look at the number of defined and inherited instance methods in the String class defined in the CLDT application (core language), the number is 204. If we include methods from all extensions (see section 11) of the String class the number is 1446. In comparison the String class in Java provides 51 public instance methods. Although this is not necessarily a precise measurement of the functional richness of the String class, it clearly shows that in a system as large as SICS, there will be senders of String methods with no direct counterpart in Java. This creates a challenge for translation of senders of methods that do not exist in Java. If such message expressions are mapped directly to method invocations in Java, there will be compilation errors. So some replacement must take place. As we have already touched upon, we want to utilize standard built in Java classes as much as possible, so recreating the entire Smalltalk class library is not an option. In most cases it is not possible or desirable to create subclasses of the normal Java classes either.

10.1 Mapping to corresponding method in Java

Fortunately there are cases where there exists a method with identical behaviour in Java. For instance senders of String>>#asUppercase can safely be mapped to String.toUpperCase(), #hash may be mapped to hashCode() etc. It is quite trivial to create mappings for such cases: For instance String>>#asUppercase is mapped as:

```plaintext
asUppercase
```

```plaintext
this toUppercase
```

10.2 Mapping to utility method

Unfortunately the vast majority of Smalltalk methods have no Java counterpart or the counterpart may have different semantics. As we concluded above, we want to utilize standard Java classes wherever possible. So how can we provide more functionality to these classes, without modifying them? The answer is to implement the required behaviour outside the class. There are existing examples of this in the Java class library. The Smalltalk Collection hierarchy is perhaps the one with the largest gap in terms of functionality between Smalltalk and Java. The class java.util.Collections contains reusable functions for collection classes. As Java collections are specified by already quite bloated interfaces, it might be considered impractical to require implementing even more methods although they might be useful. Functionality that is quite simple to implement outside the object itself, can even be reused in a more efficient manner, as one implementation can do for all. Although the above would seem reasonable for most Java programmers, it is in fact in stark contrast to the extreme object oriented attitude of Smalltalk. Due to object oriented ideals such as loose coupling, strong coherence and tight encapsulation it is in most cases preferred to let the object implement required operations itself. Combined with dynamic typing, polymorphic methods and class extensions described below, this gives very elegant solutions to many problems. In addition, in my personal view, the code is far easier to read when all function calls are directed to responsible objects, rather than passing the object around as a parameter to a function.

Still we have more or less concluded that the solution in Java has to be procedural. Example 38 gives an example of this.
public static Object printOn(final Object obj, final IAppendableStream appendable) {
    if (obj == null) {
        appendable.append("");  
    } else if (obj instanceof CscInterface) {
        ((CscInterface) obj).printOn(appendable);  
    } else if (obj instanceof DateMidnight) {
        DateAndTimeUtilities.printOn((DateMidnight) obj, appendable);  
    } else if (obj instanceof TimeOfDay) {
        DateAndTimeUtilities.printOn((TimeOfDay) obj, appendable);  
    } else if (obj instanceof BigDecimal) {  
        appendable.append(((BigDecimal) obj).stripTrailingZeros().toPlainString());  
    } else if (obj instanceof String) {  
        StringUtilities.printOn((String) obj, appendable);  
    } else {  
        appendable.append(obj.toString());  
    }  
    return obj;  
}

Example 38 Utility method for printOn:

As we have touched upon, #printOn: is the hook used in Smalltalk to modify an objects string representation. The implementation of #printOn: will affect the object’s #printString, but not the other way around. The method is therefore extremely crucial. In section Error! Reference source not found. we saw how we are able to intercept these hooks in our own classes by inheriting from CscObject. That does however not intercept existing Java classes outside our control. So we have to provide behavior for such classes in the utility method. The method above is a very good example of the typical pattern of utility methods:

1. Handling null:
   Handling null is perhaps the most important reason for dispatching to utility methods, for details refer to section 12.

2. Handling our own classes:
   In the example above we test for instance of CscInterface which is implemented by CscObject which is the ancestor of all generated classes. For less common methods, we often create a dedicated interface. Generated classes that implement the method will also implement the interface. In such cases we test for instance of the interface before we dispatch back to the object.

3. Handling Java base classes:
   The bulk of the method will normally provide the required behaviour for existing Java classes. The logic for each class may be performed in-line or in a separate subroutine. It is of course important to arrange instanceof tests with the more specific first.

4. Handling default case:
   In the example above appending the object’s toString to the stream is a sensible default implementation. Other utility methods may choose to throw an exception if none of the type tests match.

10.3 In-lining behaviour

Although we have concluded in various discussions that in lining behaviour should be done with care, there are however some rare examples where it is the best alternative. Section 8.6 demonstrates how Collection>>#collect: may be in-lined by a mapping. This is an example of functionality that is present in the Collection class in Smalltalk, but not in Java.
11 Class extensions

11.1 Some background

The code base in a VisualAge for Smalltalk development image is organized and versioned in a hierarchy. The system is known as ENVY. Methods editions are the atomic entity. Whenever the source code of a method is modified, a new edition is created and stored in the source repository. The new edition is released as current edition in the owning class. At the same time the source is compiled, and a new edition of the CompiledMethod is stored in the Class. It is possible to browse editions of a method across different editions of the containing class. The next entity is the class. While working on a class (modifying methods or signature), the class must be an open edition.

Classes can be versioned. Once a class is versioned it is no longer possible to modify its methods. Classes are defined in applications. In order to modify a class the application must be an open edition. Once an application is versioned it is not possible to modify any of its classes. All classes must be versioned prior to versioning the application. The top level entity is called a configuration map.

A configuration map is a specification of application versions. All the specified applications must be versioned to version a configuration map. A configuration map may require a specified edition of one or more other configuration maps. We see that from a version of a configuration map we can deduct the exact version down to the method level. Configuration maps can be used to load a particular version of source code into a development image, or as basis for building a runtime image.

11.2 The class extension mechanism

In addition to the hierarchy described above, classes can also be extended to other applications. It is important not to confuse class extensions with specialization/subclassing which is the relation often associated with the verb extend in object oriented programming.

So applications may contain two sorts of entities: class definitions and class extensions. Class extensions are versioned and released in the enclosing application just like class definitions.

Each class can only be defined in one application, but may be extended in several. The class definition specifies instance variables, class variables, superclass and methods. Class extensions may only contain methods. The class extensions depend on the class definition, but the class definition has no knowledge of the extensions. The class extensions are completely independent of each other, unless there are dependencies among the applications in which they are defined.

Given the above, class extension is an extremely powerful way to add functionality to existing classes without modifying the original class definition. An example of this is the Java generator reuses the standard parse nodes created by the built-in parser in IBM VisualAge for Smalltalk. Java generation logic is simply added to class extensions of the existing parse node classes. As the class extensions belong to applications which in turn may be specified by configuration maps, it is possible to handle large numbers of class extensions effectively.

If we load the configuration map “Java Generation” in a development image, the extensions of the parse nodes and all other Java generation logic is present. Other development configurations that do not include the Java generation configuration map, contain no trace of this functionality. Applications and extensions are only relevant in
the development environment. In a runtime image there are only classes and methods. Only methods defined in applications that are part of the build configuration are included.

In SICS we have several applications with class extensions for development or test purposes. These are omitted in the build specification so there is no trace of them in the running system. Similarly class extensions are used to isolate specific functionality for the different flavours of the SICS product (SICS P&C, SICS Life and SICS GR). The SICS build shipped to the customer is to a large extent stripped of functionality not relevant for the purchased product.

11.3 Class extensions in Java
Although several exotic ideas on how to split up the classes according to Smalltalk applications in Java have come and gone, we have decided to keep things simple and translate all methods of a class as part of the same class in Java. This applies to the methods of all extensions of the class in Smalltalk. This gives the obvious advantage that all the methods can be translated as is. At the same time it increases coupling and complicates dependencies across the application. For instance a class might include code related to all versions of the product, user interface and SICS/Server. This might require certain libraries in the class path at compile time and run time, which are not actually in use in a given setup.

11.4 Base class extensions
Base class extensions are a bigger challenge. This feature has been used to a large extent in SICS. Useful and product specific functionality has been added to fundamental classes such as Boolean and String. The observant reader will have noticed that this problem is in fact identical to the problem with lacking features in java, described in section 9 When a Java class lacks a method of the corresponding Smalltalk class, it is from a translation point of view irrelevant whether that method is part of the class library or has been added as part of the SICS application.
12 UndefinedObject

Smalltalk variables initially reference the special object nil of class UndefinedObject. Similarly Java object references are initialized as null. Any attempt to invoke a method on a Java variable referencing null, the virtual machine will throw a NullPointerException. However in Smalltalk nil is a proper object capable of responding to messages.

1. Defined in the UndefinedObject class
2. Defined in extensions (see section 11) of the UndefinedObject class
3. Inherited from Object, includes extensions as above

In theory all methods supported by UndefinedObject, should produce the same behaviour in Java. Null Object is a well known pattern for reducing null tests in code. However this is not easily implemented in Java due to the strong typing. One would need a separate null object implementation for each class in the system. In addition we would have to wrap and un-wrap the Null objects in all transitions to platform functions and frameworks such as the database and GUI layer. We have come up with 3 alternatives for handling null receivers:

1. Map to a utility method that handles the null case, a typical example is given in section 10.2
2. Use a mapping to inline a null test
3. Leave the generation as is and rewrite with null test in each case that fails

For case 2 and 1 (if only used to handle null), the mapping normally contains a predicate so the mapping will not apply in cases where the receiver may evidently not be null such as self, super and constants. The reader may be puzzled by alternative 3, however there are some cases where the probability of a null receiver seems highly unlikely, and the project team wanted to avoid complicating senders throughout the system. In any case, due to the very limited static analysis capability of the Java generator, both strategy 1 and 2 will be used unnecessarily in numerous cases, which the human eye immediately detects. If the developers are aware of the intention of the test or utility method, such cases can be manually during normal development in Java. Potential NullPointerException and redundant null checks are also features of some static code analyzers.

Perhaps the most prominent nil handling mapping in the entire system was actually presented in Error! Reference source not found. as an example of a mapping. The code below highlights the need for this special handling:

```
| obj1 obj2 |

obj1 := nil.
obj2 := 'hello'.

obj1 = obj2. "false"
obj2 = obj1. "false"
(obj1 = obj2) == (obj2 = obj1)
ifTrue: [Transcript show: 'commutative!'].
```

Example 39 Smalltalk equality tests involving nil
The first comparison in example 39 answers false because nil is an instance of \texttt{UndefinedObject} which is a direct subclass of \texttt{Object}. It therefore inherits the implementation of \texttt{#=} from \texttt{Object} which uses identity comparison of the two objects (\texttt{#==}). As \texttt{nil} is a singleton, the only argument for which \texttt{#=} will answer true is therefore \texttt{nil}.

The second comparison answers false because \texttt{#=} on \texttt{EsString} includes a type check (sending \texttt{#isString}). Please note that most, but not necessarily all implementations of \texttt{#=} take \texttt{nil} into account.

A direct translation into Java might be done like this:

```java
final Object obj1=null;
final Object obj2="hello";

obj1.equals(obj2);//NullPointerException!
obj2.equals(obj1);//false

if(obj1.equals(obj2) ==
obj2.equals(obj1))//NullPointerException
    System.out.print("commutative!");
```

We see that the Java code will cause \texttt{NullPointerException} in several locations. Therefore if we cannot guarantee that neither the receiver or the argument can be null, we must dispatch to the utility method \texttt{ComparisonUtilities.areEqual(Object, Object)} instead.
13 Problematic Smalltalk methods

13.1 #doesNotUnderstand: redefinitions

As we have previously touched upon, due to Smalltalk’s dynamic type system, there is no compile time check that the receiver of a message actually implements that message. If the receiver does not implement the method, the virtual machine will invoke the #doesNotUnderstand: method instead. The argument is an instance of Message, which contains the method selector and an array of arguments. The Message implements the method #sendTo:, which will perform itself on the argument. The #doesNotUnderstand: mechanism is an extremely powerful feature of Smalltalk, used in research on topics such as aspect oriented programming. There are a few overrides of #doesNotUnderstand: in SICS. These cases are easily classified into two categories:

13.1.1 Proxy Objects

#doesNotUnderstand: can be overridden to implement a transparent proxy. For instance the SICS persistency layer defers database reads to the last possible moment in order to prevent unnecessary overhead. For this purpose, a proxy is initially created in place of a proper entity object which is referenced from some other object in the data model. It is only after a message is sent to the object, that the real entity object is created and populated with persistent data. From that point on, the proxy will delegate all messages to the real object. Proxy classes serving this sort of purpose typically do not inherit from Object, in order to intercept all message sends in #doesNotUnderstand.

As a replacement for these kinds of objects, we have made use of the Java feature java.lang.reflect.Proxy, present since version 1.3. The proxy mechanism has some similarities with #doesNotUnderstand: overrides. It allows the application to intercept all method invocations against instances of proxy objects. The runtime organisation of a typical proxy is quite different in Smalltalk and Java, this is shown in diagram 8.

![Diagram 8 Instance diagram of proxies in Smalltalk and Java](http://cglib.sourceforge.net)

Java proxies may only act as interfaces, not classes. This constraint has the most serious implications for translation. For applications where this requirement cannot be met, there are also other alternatives such as the bytecode manipulation library CGLIB.

For the SICS persistency framework, the requirement to type as interfaces can be met. As the translation is automatic, the interface/implementation separation is easily done for the entire class hierarchy.

All interfaces that the proxy implements must be declared when it is instantiated. For Smalltalk proxies however there is no limitation on what sort of objects may be proxied, due to the dynamic type system. This is for instance a problem when we proxy a

relation to an object which may be any subclass in a hierarchy. The information about
the concrete subclass is defined in the database row which we want to defer reading.
The proxy must therefore be instantiated as the super interface. It is not possible to
down cast the proxy to a subclass, even though the real object is of that subclass, as
the proxy is only declared to support the super interface. For cases where down cast
was required we had to explicitly discard the proxy and reference the real object
instead. This was fairly unproblematic as the relational proxies are stored in instance
variables and only accessed through getters.

13.2 Reference introspection and modification

13.2.1 #allInstances

The message #allInstances can be used to get references to all instances of a class in
the Smalltalk image. Fortunately the method #allInstances is only used by tools in the
SICS development environment, not in the delivered product. It would be possible to
keep track of instances, by keeping references to all created instances either in some
weak data structure in the class object or a central component. For example we could
add this code to CscClass:

```java
private WeakIdentityMap instances = new WeakIdentityMap(101,0.5f);
public Object basicNew()
{
    final Object instance=this.basicNewInternal();
    this.instances.put(instance, null);
    return instance;
}
public Collection<CscClass> allInstances()
{
    this.instances.purgeBeforeRead();
    return instances.keySet();
}
```

However for the SICS product we have not found justification for maintaining this
information at runtime.

13.2.2 #allReferences

The message #allReferences answers a collection of all objects that reference an
Object in Smalltalk. It is also fortunate that there are no senders of #allReferences in
SICS production code. The relatively slow implementation of #allReferences in IBM
VisualAge for Smalltalk may be part of the reason for this. Typically invoking
#allReferences takes more than one second on a high performance PC. This is
acceptable for debugging, but not in a running system.
Likewise it has been possible to inspect references to an object in the debugger
through the Java Platform Debugger Architecture (JPDA) since version 6.0
. Although it might be possible to make use of this, it does not seem like a
recommendable approach. Let me stress that I have not actually tried it.

13.2.3 #become:

There are some rare senders of #become: in SICS. Fortunately the usage is not
crucial, and was simple enough to replace with alternative logic. To support #become:
globally in a Java application does not seem feasible. The same conclusion is drawn by
(Engelbrecht and Kourie 1998). It could of course be possible if one had a solution for
#allReferences (above), went through and modified all the references. For applications
that rely heavily on #become: the situation is more difficult.

However given that #become: is used only for certain kinds of objects for a specific
purpose, it should be possible to solve. For instance for objects that are substituted
with an instance of a different class with the same interface, a proxy or delegate pattern could be used. For an object which is shared among many other objects, some sort of reference object could be used to update all references simultaneously. Note that solutions like these require analysis and refactoring of the application.

13.2.4 subclasses / allSubclasses
In a Smalltalk development or runtime image classes may be queried not only for superclasses, but also subclasses. The method subclasses answer the class’ direct descendents, while the method allSubclasses answers all direct and indirect subclasses. Java classes however are not aware of existing subclasses in the running application.

The two methods are used to some extent in the SICS system. For instance on application startup, all subclasses of CnuDesktopFunction are tested whether they are valid for the given configuration and user, this in turn determines what functional areas of SICS are available to the user. Initially I did not want to globally support retrieval of subclasses in the Java version. The idea was that in cases like the one above, we are not really interested in the subclasses as such; they are merely a means in order to find out the valid desktop functions. A similar dynamic registration of components could easily be achieved by dependency injection or a plugin architecture. In the long run however we discovered too many cases where subclasses were needed for all sorts of purposes, so it was decided that the most realistic approach to the problem within the project’s timeframe would include support for retrieving subclasses.

We decided right at the beginning that we did not want explicit references from a superclass to its descendants. This would give unwanted circular dependencies and would suffer from unnecessary class loading overhead. Due to our experiences with class loading overhead (section 15.3) searching the entire class path for subclasses was out of the question. One must also remember that subclasses in Smalltalk only include the subclasses that are loaded/packaged in the image. This is highly relevant in SICS, where there are 4 product brandings P&C, Life, Gr and Cede. In addition each product comes as Workstation, SysAdmin and SICS/Server. So the subclasses of a given class may vary a great deal based on configuration.

The initial solution for this was to generate a method with a list of strings containing the qualified names of all the subclasses. The method would only be included for classes where we knew it was needed. At runtime only the classes that were present on the class path would be returned. allSubclasses would be supported by recursively asking for subclasses.

However at the time of writing, it seems unlikely that it will be very difficult to deploy SICS versions completely stripped of classes belonging to other products or configurations. The primary reason for this are the dependencies strong typing and class extensions (section 11) give us in Java.

Instead a utility has been developed that keeps track of the class hierarchies. The component may be queried for subclasses of a given class at runtime. Relevant classes are filtered by naming conventions. Subclass information is retrieved from serialized meta data that have been previously built by going through all classes in the classpath.

13.2.5 canUnderstand: and respondsTo:
In Smalltalk it is possible to test whether an object responds to a given message selector, similarly whether a class can understand a message selector. The test returns true if the method is defined or inherited by the class. It is also possible to use
reflection to do similar tests in Java, but according to our experiences with reflection in the translated application (section 15.1) this is both highly inefficient and also quite unreliable. So although there is put in support for these two methods, they are highly discouraged. There are also examples of code where #respondsTo: is used as a “type test” for ad hoc interfaces (section 4.5) or just to be on the safe side or simply plain laziness by the developer. For example this method:

```smalltalk
depositPortfolioReserve

^(self businessProcess respondsTo: #depositPortfolioReserve)
  ifTrue: [self businessProcess depositPortfolioReserve]
  ifFalse: [nil]
```

**Example 40 Use of #respondsTo: to test for object protocol**

The example above would best be implemented in Java by creating an interface with the method `depositPortfolioReserve()` and use of `instanceof` and cast.

### 13.3 Debugger methods

#### 13.3.1 #halt and #halt:

In VisualAge for Smalltalk breakpoints are implemented by the exceptional event **ExHalt**, this will bring up the debugger in a development image unless the exception is handled. Breakpoints are normally specified by selecting the option “Break” from the popup menu of a method browser, which will add a red dot in the left margin. However it is also possible to achieve this programmatically by sending the message **#halt** or **#halt:***. There are some examples in SICS of code that attempts to raise the developers attention of something suspicious. For example:

```smalltalk
warnOfPrivateCollectionAccess

System isRuntime
  ifFalse: [self halt: 'Private collection access - this is not allowed - please change the code. You should use CnuOneToManyRelation methods categorized as API.]
```

We have chosen to try to support this sort of behaviour in Java, although it highly is questionable whether it is good programming practice. The Java exception **SicsHaltException** can be used for this purpose. Developers should specify an exception breakpoint for **SicsHaltException**. This will immediately bring up the debugger if **SicsHaltException** is thrown. It is however not possible to resume the process as in Smalltalk. So it should be used with great care. If one instead mapped **#halt** to a globally accessible method, for instance the static method **SicsSystem.halt()** or **SicsSystem.halt(String)** it would be possible to both stop and resume the code, given that all developers placed a breakpoint in that method.

#### 13.3.2 #inspect

VisualAge for Smalltalk has two interactive ways of inspecting an objects state. During debugging or evaluation of code, one may select a piece of code and choose the popup item “Inspect”. This will bring up an inspector, a window where the values of the resulting objects instance variables can be investigated. It is also possible to bring up an inspector programatically by sending the message **#inspect**. We have not found any way to support this kind of behaviour in Java.
14 Exception handling

Although there are similarities between exception handling in Smalltalk and other programming languages, the discussions below will quickly illustrate that there are several characteristics of Smalltalk exception handling that are not easily recreated in Java. Although the Java generator has support for the most common exception handling constructs used in SICS, there are certainly subtle differences in the two languages that pose considerable risk. Manual QA is therefore carried out for exception handling code to ensure that the output is correct. This is required as correct semantics of the exception handling code is extremely crucial to the application. For fine print semantics of exception handling in VisualAge for Smalltalk refer to (IBM 2002) section “Exception handling classes” (page 26 – 32).

14.1 #signal and #signalWith:

In Smalltalk exceptions are raised by sending one of the #signal messages (#signal, #signalWith:, #signalWith:with: etc.) to an ExceptionalEvent. For instance

```
CnuExAll signalWith: ‘parent cannot be nil’
```

Is translated as:

```
throw new CnuException("Parent cannot be nil ");
```

14.2 ExceptionalEvent hierarchy

One major difference between Smalltalk and Java, is that the receiver of the #signal message in Smalltalk always is an instance of ExceptionalEvent, where as exceptions in Java are represented as separate classes. The instances of ExceptionalEvent are however organized in static hierarchy. We have therefore attempted to create a mapping from ExceptionalEvent instances to Java exception classes, using language standard Java exceptions wherever possible. Problems however occur when the inheritance order is different in the two languages, possibly producing wrong order of catch statements in Java. Fortunately the compiler will inform us if this results in unreachable code. This is just one of the reasons why we feel that it is difficult to guarantee correctness of automatically translated exception handling code.

14.3 Java checked exceptions

Another problem also arises of the fact that we attempt to map to existing Java exceptions whenever possible: The Java generator has no support for checked exceptions. A checked exception is a direct or indirect subclass of Throwable, which is neither a direct or indirect subclass of either RuntimeException or Error. Smalltalk has no concept of or support for checked exceptions. There are at least two serious syntactic issues that arise when checked exceptions are used.

1. If a checked exception is thrown as part of a method, this fact must either be declared in the method signature or the exception must be caught within the scope of the method.
2. Similarly it is not allowed to catch a checked exception unless some method within the try block declares that it may throw that exception. It is however allowed to catch the tops of the checked exception hierarchy Exception and Throwable.

14.4 #when:do:

The Smalltalk construct for catching exceptions is the #when:do: method in defined for blocks. There are also versions with more than one pair of when:do: keywords:
#when:do:when:do:when:do: etc. Each when: keyword expects an instance of ExceptionalEvent or ExceptionAllEventCollection (see below), specifying the kind of exception, and each corresponding do: argument must be a one argument block that specifies what to do given such an exception. Such block arguments will be referred to as do-blocks for the remainder of this section for simplicity. The do-block argument is an instance of Signal. For all non resumable exceptions the do-block must send the message #exitWith: to the Signal argument, or re-signal the signal (see below). This is required as all expressions in Smalltalk must return a value. As the execution of the receiver block terminated before completion due to an exception the do-block must specify what value to return from the expression. Below is an example of #exitWith:

```smalltalk
| result |
result :=
[self doExtensiveSearch]
    when: ExCLDTObjectNotFound
do: [:sig |
    Transcript
    cr;
    show: 'Some problem in search: '; 
    print: sig.
    sig exitWith: nil]
```

is translated as:

```java
Object result = null;
try {
    result = this.doExtensiveSearch();
} catch (NoSuchElementException sig) {
    System.out.println();
    System.out.print("Some problem in search: ");
    System.out.print(sig);
    result = null;
}
```

### 14.4.1 ExceptionalEventCollection
In addition to an exceptional argument the when: keyword argument can be a collection of exceptional events, where the do-block applies to all. ExceptionalEventCollection instances can be created and concatenated by the binary selector "|". For example:

```smalltalk
[self doCalculations]
    when: DivideByZero | IllegalXxx
do: [:sig | System error: 'Problem in calculation: ', sig printString]
```

Java requires one catch statement per exception type. The code must therefore be reorganized to handle the exceptions one by one.

### 14.4.2 Resignalling signals
A common approach in exception handling in both Smalltalk and Java include raising caught exceptions again. This is often done to perform logging or cleanup state, and
raise the original exception. This can be done by sending the Signal one of the `#signal` methods with or without arguments. There is of course support for re signalling Signals in the Java generation. There are however a couple of problems one must be aware of. The example below will highlight both these problems:

```
[self doStuff]
when: ExAll
do: [:sig |
  self logError: sig.
  sig signalWith: 'An error occured in doStuff']
```

Example 41 Re-signal Signal with arguments in Smalltalk

Which is translated into:

```
try {
  doStuff();
} catch (Throwable sig) {
  logError(sig);
  throw sig;
}
```

Example 42 Throwing caught exception in Java

We immediately see two problems with this code.

1. First of all `ExAll` is mapped to `Throwable` which is a checked exception type in Java. There will therefore be a compilation error when it is thrown again, as the method does not declare that it `throws Throwable`.

2. If one of the signal methods with arguments are sent to the Signal, for example `#signalWith:` the arguments of the signal are modified. We have chosen to ignore this in the Java generation, so the supplied argument is simply discarded.

14.5 `#ensure:` and `#atEndOrWhenExceptionDo:`

`#ensure:` statements and the alternative `#atEndOrWhenExceptionDo:` are quite easily mapped to Java try-finally statements. For example

```
| isBusy |
[isBusy := true.
self dowork] ensure: [isBusy := false]
```

is simply translated as:

```
boolean isBusy;
try {
  isBusy = true;
  this.doWork();
} finally {
  isBusy = false;
}
```

14.6 Other unsupported exception semantics

The rather complex example below highlights several advanced exception semantics that we are unable to support with Java exceptions. The cases will be covered one by one.
SampleResumableException ifNil: [
    SampleResumableException :=
    ExAll newChild
    resumable: true;
    defaultHandler: [:sig | Transcript cr; show:
        'Default exception handler'];
    resumable: true;
    yourself].

"Unhandled example"
Transcript cr; cr; show: '1 Before unhandled'.
SampleResumableException signal.
Transcript cr; show: '2 After unhandled'.

"Example with local exception handler"
[ Transcript cr; cr; show: '3 Before handled'.
    SampleResumableException signal.
    Transcript cr; show: '4 After handled'].
    when: self resumableException
do: [:sig | Transcript cr show: 'Local handler block'].

"Example with local exception handler that aborts"
[ Transcript cr; cr; show: '5 Before aborted'.
    SampleResumableException signal.
    Transcript cr; show: '6 After aborted'].
    when: SampleResumableException
do: [:sig |
        Transcript cr show: 'Local exit handler block'.
        sig exitWith: nil].

Example 43 Complex use of Smalltalk exceptions
The output of the code is:

"Unhandled example"
1 Before unhandled
Default exception handler
2 After unhandled
"Example with local exception handler"
3 Before handled
Local handler block
4 After handled
"Example with local exception handler that aborts"
5 Before aborted
Local exit handler block
14.6.1 Resumable exceptions and default exception handlers

In the example above we see that SampleResumableException is sent the message #resumable: true. In addition VisualAge for Smalltalk allows specification of a default handler per ExceptionalEvent instance by sending the #defaultHandler: method. The method expects a one argument block as argument, similar to the block of a when:do: specification. In this case the default handler will simply print some text to the transcript. The example shows 3 cases:

- If there is no exception handler when the exception is signalled the default handler is used, the code proceeds from the point where the exception was signalled
- If there is set up a local exception handler, it will take precedence over the default handler. The code still resumes from the point where the exception was signalled.
- If a local exception handler sends the message #exitWith: to the signal, the execution will be terminated up to the level where the exception is handled

We have not found any generic way to implement this sort of behaviour in Java. There was one unique case in SICS where a resumable exception is used in this way. We concluded that the easiest solution to the problem was to refactor the code to achieve the same behaviour in a less exotic way.

14.7 #retry

It is also possible to send the message #retry to the signal. This will evaluate the protected block once more. We have not found any practical solution for this, so the existing cases in SICS have been rewritten. A simple solution to this problem is a recursive method call, with a sound exit condition. The example below builds on the previous and shows the #retry semantics in Smalltalk:

```
| secondTry |
secondTry := false.
[
  Transcript
  cr;
  cr;
  show: '7 Before retry'.
  secondTry ifFalse: [SampleResumableException signal].
  Transcript
  cr;
  show: '8 After retry']
when: SampleResumableException
do: [:sig | Transcript
  cr;
  show: 'Just about to retry...'.
  secondTry := true. sig retry]
```

Example 44 Signal retry

The code gives the following output:
7 Before retry
Just about to retry...
“The exception is not signalled the second time”
7 Before retry
8 After retry
15 Runtime Issues

15.1 Reflection

Due to Smalltalk’s dynamic method resolving, explicit method invocation and method invocation by reflection are largely equivalent. Use of reflection therefore does not lead to much increased complexity or overhead. To some extent reflection is used by the SICS frameworks to achieve loose coupling and configurable behaviour, but as a main rule explicit method invocation is preferred over reflection, as it generally makes the code easier to understand. We shall see that the reflection problems are even direr in Java.

15.1.1 Resolving method names

All data in the Java version of SICS should be identical to the Smalltalk version. This includes string constants in code. As a consequence the first argument of the #perform:, represents the method selector used in Smalltalk. Smalltalk method selectors are unique. In Java the method signature is made up of both the name and the parameter types. As a result the reflection methods in Java for getting reference to Method objects required for setting up method invocation by reflection require specification about parameter types as Class objects. That information is of course not part of the string which contains method selector. The methods #perform:, #perform:with: etc. are mapped to InvocationUtilities.perform(Object, String, Object ... args), which contains the logic required to find the proper Method object, carry out the invocation and handle any exceptions that may occur. The part with highest overhead and uncertainty is searching for the correct Method.

As we concluded in section 2.3.2, only unary method names remain unchanged and therefore traceable to Smalltalk method selectors in Java. So for unary methods (distinguishable by an empty argument array), InvocationUtilities queries the Java class directly for a method of that name. For binary and keyword methods or any method which has been renamed by a rename mapping the situation is more complicated.

As we are not able to provide sufficient information for an exact lookup without parameter types, a brute force algorithm is used. We iterate over all accessible methods of the receiver. Each method is then tested for compatibility. The first match is returned. The matching uses a combination of information. First we look for the method annotation @SmalltalkSelector, which embeds the original selector for all translated methods. Note that this will work even for renamed methods. If that does not produce a match, the method name transformation rules used by the Java generator in the reflection utility. The rules are applied on the method selector. If the name matches and the actual arguments are compatible for the parameter types in the Method object, the method is accepted. So we are in fact able to set up method invocation even in Java, but with a considerable cost, which is demonstrated below.
15.1.2 Performance

A simple test case demonstrates the overhead in reflection use for the various cases discussed above, the results are presented in table 4, below.

<table>
<thead>
<tr>
<th></th>
<th>Smalltalk</th>
<th>Sun JDK 1.7</th>
<th>IBM SDK 5.0</th>
<th>jrockit-R27.2.0-jre1.6.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>All explicit</td>
<td>47</td>
<td>31</td>
<td>31</td>
<td>78</td>
</tr>
<tr>
<td>Getter reflection</td>
<td>31</td>
<td>1 719</td>
<td>704</td>
<td>1 234</td>
</tr>
<tr>
<td>Setter reflection</td>
<td>31</td>
<td>326 000</td>
<td>453 834</td>
<td>83 079</td>
</tr>
<tr>
<td>Keyword reflection</td>
<td>47</td>
<td>360 250</td>
<td>625 475</td>
<td>171 828</td>
</tr>
<tr>
<td>Renamed reflection</td>
<td>32</td>
<td>80 531</td>
<td>373 948</td>
<td>43 531</td>
</tr>
<tr>
<td>All reflection</td>
<td>140</td>
<td>763 188</td>
<td>1 457 637</td>
<td>307 844</td>
</tr>
<tr>
<td>Cached reflection</td>
<td>n/a</td>
<td>672</td>
<td>640</td>
<td>453</td>
</tr>
</tbody>
</table>

Table 4 Reflection overhead in Smalltalk and Java

All measurements given in table 4 are in milliseconds for one million invocations. The “All Reflection” row, is a test suite which contains the four cases above, but run within the same loop. The row “Cached reflection” uses local caching of Method objects, so reflection method lookup is exclusive. Still the overhead of using reflection is significant.

15.1.3 Experiences and conclusion

Due to the uncertainty and overhead demonstrated above, the project team has decided to refactor code to replace unnecessary reflection with explicit method invocations. The initial focus is on the main bottle necks, but other cases should be incrementally fixed based on common sense and cost benefit considerations. In SICS reflection is often used for configurable behaviour. There are many other patterns one may use to achieve configurable behaviour such as call backs, method overrides and blocks.

15.2 Classname resolving

As we have previously touched upon, all classes in our VisualAge for Smalltalk environment belong to one flat namespace: the global Smalltalk dictionary. The Java generator assigns Java packages to the generated classes. This is derived from the application that the classes are defined in, unless specifically overridden. This mapping rule is incorporated in the generation phase. On the Java side however there is no trace of this. So the code below is a challenge for Java generation:

```plaintext
| className |

className := String.
className asClass new
```

Although this kind of code is not very common in SICS, there are a few occurrences. Some occurrences have also been refactored to avoid resolving classes by name, typically by passing the class objects around instead of the class name. But still, a few rare cases exist where we need to resolve classes from names. One solution would be to replace the contents in all string references to Smalltalk classes in code with qualified Java class names instead.

Unfortunately this is not possible. In a few very dynamic parts of the system meta data that include class names are stored to the database. Bear in mind that the primary objective of the SICS4 project is to produce a version of SICS running on Java with identical behaviour as the Smalltalk version. In fact it must be possible to have a
Smalltalk instance and a Java instance running side by side on the same database. Hence the Smalltalk class names must be preserved in such cases. Instead the Java generator produces a text file that maps Smalltalk class names to the corresponding qualified Java class names. The static utility method `ClassMapper.asClass(String)` can be invoked at runtime to get hold of the appropriate Java class for a class name. Eventually this problem should be resolved, when support for the Smalltalk version ceases such class names will most probably be replaced with fully qualified class names.

15.3 Class loading and memory consumption

The current Smalltalk runtime version of SICS is packaged as one large runtime image file. The image includes required methods of both SICS classes and base classes. Right after the virtual machine is started, the entire image file must be loaded into memory. This is required regardless of what and how much functionality is actually used. In most respects this is similar to a statically compiled application. As a result the applications footprint is quite extensive and there is no reuse across instances. Java's dynamic class loading was therefore initially viewed as an improvement. Only the classes utilized by the application are loaded. There is however considerable overhead in the Java class loading process, sufficient to manifest itself as considerable delay the first time a new area of the system is accessed. At the time of writing there is ongoing work in order to reduce the impact of this problem. This includes attempts to load larger chunks of classes in the background.
16 Summary

In this report we have looked at a few of the most important translation challenges encountered in the SICS Java port project. Here is a brief summary of the most important lessons learned.

Central hook methods in the Object class of Smalltalk and Java are different. For instance the methods #printOn: and toString(). Translated objects would therefore behave inconsistent in the two languages. The solution is to let all translated classes directly or indirectly inherit from the same class. This superclass overrides the Java hooks and invoke the corresponding Smalltalk hooks to trigger overrides in subclasses.

A Smalltalk meta class can be represented in Java by a dedicated class. Smalltalk class methods are implemented as instance methods on the Java meta class. Smalltalk class variables are implemented as static variables in the Java class object, while class instance variables are protected. The Java meta class uses the singleton pattern to be globally accessible.

Smalltalk constructors are difficult to translate automatically for hierarchies without meta classes. In most cases such constructors require manual translation.

We have concluded that Java primitive data types cannot be used in the translation by default, as they are initialized to a default value instead of null. This is less of a problem after the auto boxing feature was included in Java.

The method signature of method overrides must be identical to the super implementation. This must be enforced as part of assigning types to the system. In addition the @Override annotation can help us detect such errors by causing a compilation error for lost overrides.

We have also looked at different strategies for handling cases where no single Java type covers all the kinds of objects that may occur at runtime. These problems are referred to as multi-type problems.

For implicit Smalltalk protocols across inheritance hierarchies, we have concluded that dedicated Java interfaces are required. The toolset has support for defining such interfaces, and the Java compiler ensures compliance.

We have seen some of the advantages of Java generic syntax. For these reasons we have decided to use generics extensively in the translated code. But we have also seen that generics types can be hard to get right in complex call graphs and deep hierarchies. The Java port toolset is not sufficiently sophisticated to resolve these issues automatically. Therefore we have implemented some pragmatic workarounds for issues with generic types that would otherwise be very hard to resolve.

Smalltalk syntax is entirely based on message expressions. In addition, methods always return a value. As a result all expressions return a value. In many cases the corresponding Java syntax cannot be used as expressions. For instance an if-else statement cannot be used as an expression. In many cases this requires transformation of the source code in order to be able to use standard Java syntax. Similarly cascaded Smalltalk message expressions require a temporary variable in most cases in Java.
Arrays and strings are indexed differently in Java and Smalltalk. The first element is stored at index 0 in Java and at index 1 in Smalltalk. The solution to the problem is relatively simple. Mappings can be used to add and subtract one from the indexes in the required locations.

There are several problems regarding translation of Smalltalk blocks to Java. Anonymous inner classes can be used to implement blocks in Java. Anonymous inner classes do not fully support two vital requirements of Smalltalk blocks, namely non local return and static scope references. In addition the syntax of anonymous inner classes is much less compact than Smalltalk blocks. We have shown possible workarounds for the two problems mentioned above, but also how mappings can greatly enhance the Java code by removing the need for blocks where applicable.

We have also demonstrated how mappings may be used to handle incompatibilities in the standard class library of the two languages. Lacking functionality can also be a result of application specific extensions of the Smalltalk library. Mappings may solve this problem in different ways:
- Map to a corresponding method with a different name
- Map to a utility method that implements the same functionality
- In-line the functionality

Utility methods may also be used to implement the behaviour of nil in Smalltalk. Invoking any method on null will give a NullPointerException in Java.

Decimal numbers and integers are quite different in Smalltalk and Java. Most notably, due to the dynamic typing of Smalltalk, different kinds of numbers can be mixed in computations. Even though this sort of behaviour is not possible with Java number types, we have put in some more flexibility and sanity checking by special utility methods for computations.

Exception handling is also quite different in Smalltalk and Java. There are a number of features of Smalltalk exception handling that are not easily reproduced in Java. Although the Java generator has support for translation of Smalltalk exception handling. Manual QA of the translated code is therefore recommended in order to uncover any subtle differences.

There certainly exists more exotic Smalltalk functionality which we have not found feasible to support such as #become:
17 Conclusion

As this report consists of a collection of odd practical problems and discussions it does not really build up to any grand conclusion. A few solutions have been proposed, which have worked well in our case and may be of use for others. I do not claim that our solutions are revolutionary or necessarily better than what other developers would come up with given the time and opportunity. Hopefully the reflections presented are worthwhile reading for others with an interest in this problem area.

I must once again disappoint the reader with the absence of silver bullets. Like most other software development problems it really boils down to basic problem solving skills and common sense. As in all other development tasks the goal is to come up with an acceptable solution meeting the requirements given the constraints in time and resources.

Although I have tried to keep most discussions independent of the product, I suppose the reader is by now interested in the outcome of the project. I will therefore give my subjective assessment of the relative success of the approach chosen to meet the overall goal of the project. As I am writing this we are halfway through a full blown version test of the first version of SICS running on the Java platform.

Most importantly, I am positive that our approach has been cost effective in terms of preserving correctness and business value of the product compared to translation by hand or a complete remake. This cannot be stressed enough. This belief is based on experiences from the last 10 months of testing and debugging the Java version.

- Only a tiny fraction of the overall problems originate from errors in the generation logic. But once such an error is found, it may be fixed once for all.
- In fact the majority of problems originate from subtle differences in the infrastructure frameworks (primarily GUI) that were rewritten from scratch due to change of platform. This goes back to the original point: automatically translated mature code does in my experience generally contain less errors than new code.

Secondly, assigning types and inserting correct casts has required a larger effort than we originally anticipated. This is highly correlated with code quality. In general the automatic and manual type assignment routines have been highly successful for most code that follows good object oriented practice and internal guidelines, while some pieces of problem code have been exceptionally time consuming. I am under distinct impression that the issues and effort related to assigning types in SICS is greater than the combined effort of all the other issues discussed in this report. Unfortunately I am not able or permitted to publish hard numbers to document this view.

Admittedly some of the generated code is less Java like than Java code written from scratch by humans, and some of the intentions of the Smalltalk code are less visible. The word "code generator" sounds bad in many ears, so let me again emphasize that we have strived very hard to output high quality, readable and maintainable Java code adhering to standards. If not we could just as well have created a compiler to Java byte code. We have also seen that rewrite rules (mappings) can go a long way in transforming source code to the style of another language, see for instance section 8.6.

To sum up, we were able to find general solutions for the most important translation issues from Smalltalk to Java required by our application. Fortunately in our case the Smalltalk features we were not able or did not want to support on the Java platform, such as #become: where manageable in number and quite easily substituted.
Appendix I. References


