SELF 4.0
Smalltalk Emulator

Mario Wolczko
Mario.Wolczko@Sun.Com

This document outlines the facilities present in the Smalltalk emulator included with Self 4.0. It should serve as a user guide. The document assumes some familiarity with Smalltalk. A forthcoming paper will describe the implementation in more detail.

Copyright (c) 1995, Sun Microsystems, Inc. All Rights Reserved.

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043 USA

R**ESTRICTED RIGHTS LEGEND:** Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 (Oct. 1988) and FAR 52.227-19(c) (June 1987).

SOFTWARE LICENSE: The software described in this manual may be used internally, modified, copied and distributed to third parties, provided each copy of the software contains both the copyright notice set forth above and the disclaimer below.

DISCLAIMER: Sun Microsystems, Inc. makes no representations about the suitability of this software for any purpose. It is provided to you "AS IS", without express or implied warranties of any kind. Sun Microsystems, Inc. disclaims all implied warranties of merchantability, fitness for a particular purpose and non-infringement of third party rights. Sun Microsystems, Inc.'s liability for claims relating to the software shall be limited to the amount, if any of the fees paid by you for the software. In no event will Sun Microsystems, Inc. be liable for any special, indirect, incidental, consequential or punitive damages in connection with or arising out of this license (including loss of profits, use, data, or other economic advantage), however it arises, whether for breach of warranty or in tort, even if Sun Microsystems, Inc. has been advised of the possibility of such damage.

1 Overview

The Smalltalk emulator is a part of the Self system which mimics the core of a Smalltalk system using Self objects. It was built for a variety of reasons:

1. To demonstrate that Self, a prototype-based system, is capable of providing all the features of a class-based system.
2. To illustrate how Self's advanced implementation can benefit other object-oriented languages.
3. To provide a free, open, high-performance Smalltalk implementation to educators.
4. To provide a non-trivial sample applications demonstrating the use of the Self language and graphical user interface.

The Smalltalk emulator is not intended to be a production-quality Smalltalk environment, but just a demonstrator of what is possible. There are many facilities and features missing that would be required to make a complete Smalltalk system, but most of them are fairly straightforward extensions of what is present. Indeed, it is hoped that educators might use the current implementation to teach how Smalltalk works, and to set exercises to extend the system in interesting ways. The author would be most interested to hear of such work.

2 What the user sees

On the Self background menu is an item, 'Smalltalk', which pops up a second, pinnable menu. This menu can be used to get Smalltalk browsers, workspaces, and the system transcript (where printed output appears).

The browser is a modest facsimile of the Smalltalk-80 system browser. It organizes the Smalltalk classes and methods into class categories and protocol. You can use it to browse Smalltalk methods, edit and enter methods, and to enter and edit classes.

The workspace can be used to enter Smalltalk expressions, which are translated into Self for evaluation. The transcript is a special kind of workspace in which strings sent to the Smalltalk object, Transcript, appear.

2.1 Browser operation

As with a conventional Smalltalk browser, clicking on an item in one of the top four list panes causes the list to the right to update. Note that all items in the list appear on the screen; there is no list scrolling (scrolling would be desirable, but there was no time to add it). When all panes have a selection, a method appears in the edit pane at the bottom. When only some of the panes have a selection, one of a variety of different templates appears in the edit pane. The contents of the edit pane can be altered, which will cause a standard Self 'counterfactual' panel to appear; clicking the green button accepts the change, clicking the red button cancels it. Meta-return is a keyboard shortcut for accept. When accepted, the Smalltalk code is translated into Self.

2.1.1 Browser menus

The following menu items are available in the respective list panes when one or more items are in the list:

- Category pane: *update* updates the list of categories in the browser. As with most Smalltalk systems, this must be done manually (unlike Self outliners, which are automatically updated).
- Class pane: *remove* removes the class and its subclasses. Beware: no confirmation is requested! *Add protocol* is used to add a new message category to the class. A template method is inserted into the new protocol, ready to be edited.
- Protocol pane: *add* gives you another way to add a protocol.
- Selector pane: *remove* removes the selected method; again, no confirmation is requested.
- Edit pane: *cut*, *copy* and *paste* do the obvious things. *Do it* executes the selection as a Smalltalk expression. *Print it* also executes the selected expression, but also prints the result in the transcript. *Inspect it* also executes the selected expression, but then opens an inspector on the resulting object. Note that unlike most other Smalltalk implementations, execution is asynchronous, meaning that you can do other things while it is taking place.

The workspace and transcript have the same menu as the browser's edit pane.

2.2 Inspectors

An inspector provides an object-level view onto a Smalltalk object. The name of the object's class appears in the label. In the left hand pane is a list of the object's instance variables, preceded by the item *self*. Selecting one of these items causes the value of that item to printed in the right pane. An expression can be entered into the right pane. This expression, if accepted, will be evaluated and the result will replace the old value of the instance variable selected in the left pane.[†]

2.3 Filing code in

The Smalltalk sources provided with the system are modified versions of the GNU Smalltalk library, version 1.1.1. You can file in other code using the expression `'fileName' fileIn`. The system will look in all directories in `smalltalkEmulator sourceDirPath` for `fileName` or `fileName.st`. The file-in format used is that of ParcPlace Smalltalk, and the language accepted is (supposed to be) compatible with ParcPlace Smalltalk (with the exceptions noted below). To convert GNU sources, you need to insert a space between pairs of exclamation marks.

3 Example

Use the background menu to get the Transcript, a browser and a workspace. In the workspace, type the expression `'richards.st' fileIn` and then 'do it'. In the transcript will appear messages

[†] Inspecting indexed instance variables is not yet implemented.

as the various classes and categories in that file are installed. When it is done, update the browser using the left pane menu, find the category `RichardsBenchmark`, then the class `RichardsBenchmark`, then the class method `start` (in protocol *instance creation*). You can select and execute this expression. The benchmark will run and print a time in the transcript. If you execute it repeatedly, you should notice the time improve dramatically as the Self re-compiler optimizes the code. You may want to try running this benchmark in another Smalltalk system for comparison.

4 Limitations

Due to the experimental nature of the Smalltalk emulation system (it was constructed in just a few weeks), there are many restrictions and limitations.

4.1 Linguistic restrictions

With the exceptions noted below, the system accepts ParcPlace Smalltalk syntax, including

- literal byte arrays: `#[1 3 4]`
- := for assignment
- `super` only as the receiver of a message
- block temporaries: `[ :a || t | ... ]`
- string literals as symbol names: `#'a symbol'`
- nil, true, false and strings in literal arrays: `#(nil 'foo')`
- optional # signs in nested literal arrays: `#((1) #(2))`

4.1.1 Syntax

The grammar used to build the parser does not capture some of Smalltalk's syntactic quirks, so the following cases are *not* accepted:

1. Assignment without a preceding space. You must write `x :=` with a space between the `x` and the `:=`.
2. Subtraction without a preceding space. You must write `x - 1` with a space between the minus sign and numeral.
3. Spaces in block arguments. No space is allowed in `:x`.

4.1.2 No pool dictionaries

Pool dictionaries are not supported. Who uses them anyway?

4.1.3 No class instance variables

If you don't know what this means, you certainly don't care.

4.1.4 No thisContext

The thisContext pseudovariable is not supported. The Self model for access to execution state is quite unlike that of Smalltalk's; emulating Smalltalk's is well beyond the scope of this system.

4.2 Numeric restrictions

There are no double-precision floats, nor fractions. Fractions could be added if a suitable Smalltalk implementation was available, together with some coercions in the Self arithmetic operations. The syntax for exponent format integers is not supported.

Integers and floats are represented using Self integer and float objects. The objects do not have a Smalltalk-like class, so you cannot add Smalltalk methods to them, nor can you send them the message `class`.† You can, however, add Self methods to the Self objects known as `traits smallInt`, `traits integer`, `traits float` and `traits number`, which will apply to Smalltalk numbers too. Many compatible methods are provided, however. Beware of subtle inconsistencies, especially that `printString` returns a Self string not a Smalltalk one (you can use `asSmalltalkString` to convert).

4.3 No classes True, False, Boolean

As with integers and floats, the objects `true` and `false` are represented using the Self objects of those names. They do not have a class, and cannot have Smalltalk methods added to them. As with numbers, however, you can add Self methods to `true` and `false`. The cautions listed in the previous sections also apply.

4.4 Non-lifo blocks need to be distinguished; weak support

The Self Virtual Machine does not currently support non-lifo blocks, that is blocks which are executed after their home context has returned. A facility is provided to mimic non-lifo blocks by the emulator; however, you need to tell the emulator which blocks may be non-lifo. The following example will illustrate:

```
nonLifoBlockExample
    | c |
    c := SortedCollection new.
    c sortBlock: [ :a :b | a > b ].
    ^c

Object new nonLifoBlockExample add: 1; add: 2
```

If you try running this, you will get a debugger with the message 'Self 4.0 cannot run a block after its enclosing method has returned'. To fix this, we need to tell the emulator to handle the block specially, which we do by sending it the message `asNonLifoBlock:`

† It would be possible to implement this; anyone interested should ask me how.

```
nonLifoBlockExample
    | c |
    c := SortedCollection new.
    c sortBlock: [ :a :b | a > b ] asNonLifoBlock.
    ^c
```

Non-lifo blocks also suffer a further limitation: you cannot access any of the variables in the enclosing method (but you can access self or its instance variables). This can be a problem if you have code from versions of Smalltalk in which you are not allowed to declare temporaries inside blocks:

```
nonLifoBlockExample2
    | c t |
    c := SortedCollection new.
    c sortBlock: [ :a :b | t := a + 1. t > b ] .
    ^c
```

In this case we not only need to annotate the block as non-lifo, but also move the temporary into the block:

```
nonLifoBlockExample2
    | c |
    c := SortedCollection new.
    c sortBlock: [ :a :b || t |
              t := a + 1. t > b ] asNonLifoBlock.
    ^c
```

Normal blocks (i.e., blocks which do not outlive their home context) do not suffer from this restriction.

4.5 Methods incompatible with Self

Some methods in the numeric classes are already present in Self, with subtly different meanings. The selectors /, & and | are mapped by the emulator to /=, && and ||.

4.6 Environment limitations

The tools and facilities provided are very rudimentary. Syntax errors are notified by the generated Smalltalk parser; the error messages are not very friendly. Runtime errors cause a Self debugger to appear, and the generated Self code is visible in the debugger.

Many extensions to the system are possible:

- Scrolling could be added to the browser's list panes.
- Navigation tools such as senders/implementors/messages/find class/find method could be added to the browser menus. These could be built by extending the Self tools (sendersMorph, implementorsMorph, etc.).
- Filing out Smalltalk code.

- Renaming categories, classes and protocols.
- If you change the definition of a class, existing instances are not updated, and existing values of class variables are lost. This would not be difficult to fix using Self mirrors and enumerators.
- There are no classes for processes or semaphores, despite Self supporting such objects. It should be straightforward to wrap Smalltalk classes around the equivalent Self objects.

5 Extensions

5.1 Primitives

The syntax for a primitive is different from Smalltalk's. Where Smalltalk requires an integer, in the emulator you may write a string containing a Self expression. This string will be suffixed with the rest of the method, as a fail block. For example, the Smalltalk method

```
basicSize
  <primitive: '_Size'>
  self primitiveFailed
```

is translated into

```
basicSize = (_SizeIfFail: [self primitiveFailed])
```

You may also send messages between Self and Smalltalk objects. To allow you to break out of the Smalltalk world, a method, `selfLobby`, is provided in `Object` to return the Self lobby. You may also access the emulator itself via `smalltalkEmulator` should you want to try some reflective tricks!

6 Class library limitations

The class library provided contains a reasonable selection of collection classes, and basic infrastructure in Object, Behavior and Metaclass. However, reflective operations are not provided; these must be done via Self. Similarly, there are no user interface classes – you must call out to the Self morph world.

7 Bootstrapping

The entire Smalltalk system is bootstrapped from within Self by loading in the module `smalltalk.self`. This contains the Self code for the emulator. It builds the Smalltalk system by:

- Loading in the Self parser-generator, Mango.
- Using Mango to build a Smalltalk parser (using the grammar in `smalltalk.grm`). The parse trees are augmented with behavior for the Smalltalk-to-Self translator from the file `smalltalk_behavior.self`. Mango is then removed from the system to save space.

- Minimal versions of the kernel classes Object, Behavior and Metaclass are created in Self. Just enough structure is created to enable the creation of further subclasses. Special versions of String and Symbol (called BootstrapString and BootstrapSymbol) are installed, so that Smalltalk expressions creating new classes and methods can be executed.
- Bare-bones versions of UndefinedObject, Array, ByteArray, Character, String and Symbol are defined (with their superclasses). The new versions of String and Symbol replace BootstrapString and BootstrapSymbol.
- At this point, the emulator is capable of translating any Smalltalk sources (subject to limitations explained in previous sections), and bootstrapping is completed by reading in the remaining methods for the classes already created, and then by creating the remainder of the class hierarchy.

8 Acknowledgements

Thanks go to the other members of the Self group for suggestions and encouragement. Thanks also to Steve Byrne for putting out the GNU Smalltalk class library.

The task of designing the emulation system was made much easier by student projects (while the author was at the University of Manchester), which figured out the general problems of translating Smalltalk into other languages; I am indebted to Borek Vokach-Brodsky, Neil Cope and Ivan Moore for teaching me how to deal with the subtleties of Smalltalk. Their work is described in:

- Smalltalk Application Compilers, B. R. B. Vokach-Brodsky, M. I. Wolczko, Proc. TOOLS Pacific, Newcastle, Australia, Dec. 1990, pp. 69-78.
- Babel - A Translator from Smalltalk in to CLOS, I. Moore, M. Wolczko, T. Hopkins, Proc. TOOLS USA, Santa Barbara, CA, Aug. 1994.