Smalltalk: Overview and Implementation Issues

Heinrich Moser*

October 2002 – January 2003

Starting with the very first evaluator, implemented as a thousand-line BASIC program in 1972, Smalltalk was the first computer language based entirely on objects and messages. This paper gives an overview of Smalltalk and some of the design issues Smalltalk implementors had to face.

1 Introduction

Smalltalk consists of

1. an object-oriented *programming language*,
2. a *virtual machine* to run programs written in that language,
3. a *class library*, providing access to low-level functions and facilitating development, and
4. a graphical *user interface* and *programming environment* included in the class library.

In respect to points 1-3, Smalltalk is similar to recently developed object-oriented languages such as Java or C#. However, it includes a GUI which contains a development environment and debugging facilities; even the GUI itself can be modified and analyzed using Smalltalk.

Smalltalk implementations include the *virtual machine (VM)* and the *virtual image (VI)*. The virtual machine is hardware/operating system dependant and interprets programs written in the Smalltalk language and compiled to Smalltalk binary code; thus, the VM includes the memory management and other primitive functions which must run on top of the operating system. The virtual image contains the class library in

*E-mail: `mail@heinzi.at`

Smalltalk binary code; therefore, it needs the VM to run, but it is machine-independent. It includes data structures, the user interface, and development tools.

Theoretically, a Smalltalk virtual machine could be written for any hardware platform conforming to the following requirements as laid out in [GR83], the so-called “Blue Book”:

- a bitmap display
- a pointing device with three buttons
- a keyboard
- a disk
- a real time clock and a millisecond timer

Therefore, it is not surprising that implementations of Smalltalk are available for a wide variety of platforms, including DOS, OS/2, Windows, Macintosh[1], Unix, BeOS, NeXT and PDA operating systems.

There is another facet in which Smalltalk differs from the object-oriented languages we usually use today: A program is not developed by starting a text editor, writing it down, compiling it, and starting it, but by manipulating the Smalltalk environment itself. The virtual image, representing the current state of the system, is a collection of objects being created, destroyed and modified. As shown in figure 1, a program called System Browser can be used to examine and modify the classes currently present in the running virtual image.

Smalltalk has evolved from its first version Smalltalk-72 to Smalltalk-80, which is the current standard. The historical development of the language and its virtual machine is outlined in [Ing83].

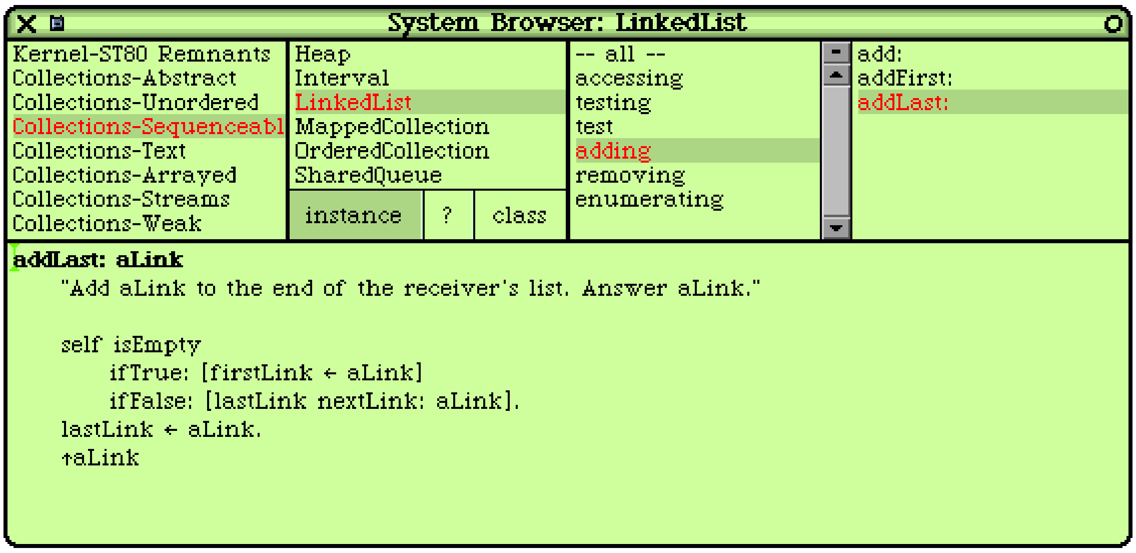
1.1 Implementing Smalltalk

The Smalltalk-80 virtual machine specification [GR83] defines the required behavior of a Smalltalk-80 interpreter and provides a “model interpreter”. The implementation details, however, are left to the implementor. Many of the design decisions that have to be made are laid out in [WB83].

One approach – to translate the specified model interpreter into an appropriate implementation language – creates a usable interpreter in a short period of time. However, its performance might be disappointing; therefore, another approach might be taken by designing a more efficient interpreter based on the experience gained from the model implementation.

Other design decisions implementors have to face are the hardware and the implementation language. The system will need to support the required amount of main memory

[1]On the Macintosh and other systems with less than three mouse buttons, additional buttons can be emulated by holding down a control key while clicking.



```
addLast: aLink
    "Add aLink to the end of the receiver's list. Answer aLink."

    self isEmpty
        ifTrue: [firstLink ← aLink]
        ifFalse: [lastLink nextLink: aLink].
    lastLink ← aLink.
    ↑aLink
```

Figure 1: System Browser

and the processor should provide an adequate number of registers in order to optimize performance by caching variable values. As the Smalltalk specification requires a bitmap display device, hardware support for bitmap operations can also improve performance.

The choice of an implementation language opens up the usual dilemma faced by software developers: A high-level programming language facilitates the coding whereas usage of a low-level language might yield better performance.

Tektronix Smalltalk

One of the companies that implemented Smalltalk was Tektronix Inc. They developed a Smalltalk-80 interpreter based on a Motorola 68000 processor and less than 1 megabyte of RAM. The virtual machine was written mainly in Pascal, although the arithmetic primitives were coded in assembly language to simplify the handling of the SmallInteger bit (see section 3.1 for details). It was cross-compiled on a DECSYSTEM-20 and later re-written and optimized entirely in assembly language. Their experience with the development process is outlined in [McC83].

The Tektronix implementors decided not to use an operating system on the host system, thus having full control of the hardware and the memory. The drawback of this decision was that the file system and other basic features had to be implemented (rather than just interfaced).

Apple Smalltalk

Another Motorola 68000 implementation was done by Apple Computer Inc. Their work was basically a translation from the Xerox specification to 68000 assembly language with only one small optimization regarding the message lookup cache. Very thorough performance measurements were made during the implementation. The results can be found in [MC83].

Berkeley Smalltalk

The University of California, Berkeley, also implemented a Smalltalk interpreter they called "Berkeley Smalltalk". They used a DEC VAX-11/780 as the host platform and wrote their virtual machine in the C programming language under Berkeley Unix. They believed that a straightforward implementation of the specification could not achieve acceptable performance. Proving their point, they published a comparison of a by-the-book interpreter and their implementation ([UP83]).

DEC VAX/Smalltalk-80

The Corporate Research Group at Digital Equipment Corporation (DEC) also created its own implementation. The first virtual machine was written on a PDP-11/23, implemented in assembly language as closely as possible to the official specification. It turned out to have three major performance bottlenecks: the memory manager, which spent most of its time adjusting the memory map because of the PDP-11's memory design; the lack of writable microcode hardware, which forced the developers to use assembly language instead; and the reference counter, which created a large amount of memory management overhead.

The second assembly code implementation was based on a VAX 11/780, the same host system on which Berkeley Smalltalk was developed. DEC used VMS as the operating system, a word size of 32 bits, and an incremental compacting garbage collector ([BS83]).

VAX/Smalltalk-80 was adapted to co-exist with other applications on a timesharing system, something for which it was not designed. Wait messages were inserted into code loops waiting for mouse buttons and a hibernate primitive was added to the idle process.

Squeak

Squeak is a recently developed, freely available implementation of Smalltalk, whose virtual machine has been written entirely in Smalltalk.

The goal of the Squeak developers was to design an easy-to-use educational software environment that could even be programmed by children. In order to ensure portability and to ease debugging, the Squeak kernel was written in Smalltalk and translated into C. To avoid having to emulate the complete Smalltalk language in C, Squeak was written in a subset of Smalltalk which did not include support for objects and message sending.

The machine-dependent part of the virtual machine that must be written directly in C has about 1700 lines of code[2] which must be changed when porting Squeak to another hardware platform. The machine-independent rest of the VM (about 6500 lines of code: interpreter, object memory, graphics libraries) is written in Smalltalk and can be translated into C when compiling the VM. Only a few weeks after the release of

[2]Figures are taken from the December 1996 release, version 1.18.

the Macintosh version of Squeak, ports for various UNIX and Windows systems were available.

Details about the development of Squeak are laid out in [IKM+97].

2 Language

In contrast to most popular object-oriented languages, which distinguish between native data types and class-based objects, everything in Smalltalk is an object. This is partly due to the fact that one of the Smalltalk paradigms is to write as much as possible in Smalltalk itself. Only parts that must be implemented in machine code (such as I/O primitives or basic arithmetic operations) are part of the virtual machine.

2.1 Objects and Messages

The first version of the language was Smalltalk-72, a language based entirely on classes, objects and messages passing between objects.

Message Passing Example: For example, the expression `5 squared` sends the message `squared` to the object `5`, which belongs to class `SmallInteger`. The class `SmallInteger` is defined to return `self * self` upon receiving the message `squared`, which, in this example, yields the new SmallInteger object `25`.

Smalltalk-74, following only two years later, included some updates to the virtual machine. In 1976, there was also a redesign of the Smalltalk language (Smalltalk-76): Classes became real objects and support for inheritance was added. Furthermore, the syntax was updated and some constraints limiting the maximum number of variables and literals were removed. Smalltalk-78 included a new method `become`, which allowed two objects to exchange their identity (see below). Finally, Smalltalk-80 removed non-ASCII characters from the language.

2.2 User Interface

Smalltalk-72's graphic capabilities relied on `Textframes` to display text and `Turtles` to draw lines. In Smalltalk-74, an advanced library called `BitBlt` was introduced.

In an effort to reduce the size of the machine-code containing the kernel, BitBlt was almost entirely ported to Smalltalk code, thereby becoming part of the virtual image. This step had become necessary, because Smalltalk-78 was supposed to run on a portable computer and the designers did not want to transcribe the whole kernel to the portable computer's machine code.

Squeak

Support for sound was not included in the Smalltalk-80 specification; however, it has been implemented in Squeak. Since sound processing is a very compute-intensive process, the algorithms were implemented in the Smalltalk subset which can be translated into C code. Therefore, they can either be run directly within Squeak or translated and linked into the virtual machine for better performance.

Squeak also uses BitBlt for displaying purposes and added support for color: table-based color with 1, 2, 4 or 8 bits and RGB color with 16 or 32 bits. An additional library WarpBlt[3] was implemented which speeds up rotation and scaling of images and adds support for anti-aliasing techniques (e.g. smooth edges and calculate average colors). WarpBlt was written in Smalltalk and afterwards translated into C for performance reasons.

3 Virtual Machine

The Smalltalk Virtual Machine is used to establish the link between programs compiled to Smalltalk byte code and the underlying operating system. Its major tasks include memory management and garbage collection.

3.1 Object Memory

The original Smalltalk-72 used direct memory addresses as object pointers. Objects were reference counted (see Section 3.3) and returned to the linked list of free storage after no longer being accessible.

Smalltalk-76 included OOZE ("Object-Oriented Zoned Environment"), a virtual memory system. Class information was encoded directly into nine bits of the object pointer. This had the advantage of not having to include a class pointer within the object, thereby saving some storage space. However, it did limit the number of classes possible and the maximum number of instances per class.

Smalltalk-78, on the other hand, used an indexed object table. Therefore it could easily support a new primitive method called `become:`, which would switch the identity of two objects by simply exchanging the contents of their object table entries. For reasons of efficiency, the replacement of direct object pointers by an object table was also used to store SmallIntegers (integer values between -16384 and 16383) as values rather than as objects. If an object pointer had the low-order bit set, it was not pointing at an object in the object table but it was rather a SmallInteger, whose value was defined by the remaining 15 bits. This had the consequence that pointers to two SmallInteger objects containing the same value were actually equal.

Object Pointer Example: `000000000010100` is an object pointer to the object table entry with index 20. Object table indices are always even. However, `000000000010101` is a SmallInteger containing the value 10.

Example for the `become:` Message: Imagine the following piece of code being executed.

[3]WarpBlt was supposed to be a "warp drive" for BitBlt, named after the engine of the starship Enterprise in the popular science-fiction TV show *Star Trek*.

```
x := 'X'
y := 'Y'
p1 := x
p2 := x

p1 become: y
```

After issuing the last statement, `x`, `p1` and `p2` point to the string object `'Y'` whereas `y` points to `'X'`.

[WB83] outlines other possibilities for object pointer formats. The internal format is invisible to the Smalltalk programmer; therefore, individual implementations can use alternative formats and still conform to the formal specification. For example, the decision to put the SmallInteger tag bit into the least significant bit (LSB) was made because the original implementation hardware (a Xerox processor) reflected the value of this bit in its condition codes. As we will see below, other implementations might have reasons to change that position in order to optimize the performance for their desired host system. They might put the tag into the most significant bit (MSB, which is the sign-bit position for 16-bit 2's complement processors) and use a conditional branch to test for the tag. Another possibility would be to keep the tag in the LSB but invert the meaning (i.e. 1 = object pointer, 0 = SmallInteger). This way, values can be added and subtracted without requiring a conversion. The result is another valid SmallInteger in object pointer format. Inverted meaning and putting the tag into the MSB would result in an object pointer value that could be used directly as a direct index into a table of 8-bit values.

DEC VAX/Smalltalk-80

Due to the fact that DEC VAX/Smalltalk-80 was based on 32-bit words, the DEC implementation used 31-bit object names, using the 32nd bit as the SmallInteger tag. This enabled Smalltalk to use 31-bit long SmallIntegers, thereby eliminating the need for LargePositiveIntegers as indices to large arrays.

The implementors also decided to use some kind of "virtual object" system, in order to allow for a very large number of objects. The designer's goal was to use a hybrid system of object swapping and paging in order to get a low faulting rate and to have Smalltalk run on paged systems.

Squeak

The Squeak object memory was designed with 32-bit direct object pointers, i.e. without an object table. In order to reduce overhead, a variable-length object header is used. A small class table is included for frequently used classes thereby avoiding the need to store a full 32-bit class pointer in every object. Nevertheless, not using an object table makes the `become:` operation much more costly, because rather than just switching two entries in the object table, the whole memory must be scanned in order to replace every pointer to one object with a pointer to the other.

3.2 Object Allocation

There are two cases in which new memory needs to be allocated:

1. A new object is being created explicitly by calling the `new` primitive.
2. A new activation record must be allocated because a method has been invoked. This is accomplished by creating a new context object (which behaves like any other Smalltalk object).

A new object pointer must be found to refer to the new object. The memory manager must then find an area of free storage in the object memory that can be used to store the new object. Finally, internal data structures must be updated and the new object be initialized with a null value. Since every method call causes a new context object to be created, performance can be raised considerably by speeding up this process. Another bottleneck is located within the graphics routines, which cause a large number of small Point objects to be created.

Tektronix Smalltalk

Tektronix decided to cut on the context creation expenses. Complete context objects are only created if they are really needed, i.e. if a method references the active context or causes another context to be activated.

Berkeley Smalltalk

The Berkeley Smalltalk optimization is based on the assumption, that when an object of a certain size (for example, the size of a small context object) is freed, a new object of that size will be requested in the near future. Thus, pools of objects with small sizes (0-40 fields) are being used. When freeing an object, the memory manager returns it to the linked list of objects of this size but the object table entry is being kept. If a new object needs to be allocated and the corresponding pool is empty, the Unix storage allocator is invoked, delivering ten new objects for the pool. On the other hand, when no more object table slots are available, objects in the free pool are being removed.

3.3 Garbage Collection

A new object is allocated by sending a `new` message to its class. To explicitly deallocate an object is not possible, because this could result in errors such as variables still referencing an object which no longer exists. Thus, the *reference counting* algorithm is used to ensure that objects that are no longer in use will be destroyed.

Reference Counting: Every object has a counter that is incremented whenever a variable is assigned a pointer to that object. As soon as the variable is destroyed or reassigned another value, the reference counter is decremented. As soon as the counter reaches zero, the object can be destroyed. Smalltalk's reference counting algorithm is laid out in more detail in [WB83].

It is critical for the garbage collector to be very stable and reliable. Freeing objects while they are still being referenced can lead to errors that are difficult to debug because they appear much later, when the damage has already been caused and part of the still-in-use memory has been overwritten. On the other hand, not freeing unused memory leads to memory leaks.

By the time Smalltalk-80 was implemented, the classical garbage collection method called *mark and sweep* (see [Coh81] for details) was considered unacceptable because this process would consume a considerable period of time whenever storage was reclaimed. The reference counting model was chosen because it distributed the overhead time over the normal operation of the system.

[WB83] suggests that garbage collection is an area where the performance of the Smalltalk system can be improved significantly. This observation has been confirmed by [MC83] and [UP83], whose performance benchmarks show that a considerable part of the execution time is spent reference counting. One proposed technique is called *deferred reference counting* (see [DB76]), which reduces the number of necessary reference count operations.

Although the formal specification's Smalltalk model uses reference counting, other implementations have decided to use a different approach to garbage collection. The drawbacks of reference counting are that it does not work for circular references and that it fails once the reference count field cannot hold the number of references (which can either be solved by a sufficiently large reference count field, an overflow detection mechanism, or an additional mark-and-sweep garbage collector).

Tektronix Smalltalk

The very first Tektronix implementation included the reference-counter provided by the formal specification (translated into Pascal) and a simple, recursive mark-and-sweep algorithm to get rid of circular references.

The second version of Tektronix Smalltalk used a separate reference-count table and the above mentioned *deferred reference counting* technique to limit the reference-counting overhead.

Berkeley Smalltalk

Berkeley Smalltalk implemented the following enhancements to speed up reference counting:

- The reference-count variables were enlarged from 8 to 32 bits to avoid overflow checking.

- The SmallInteger flag was moved from the LSB to the MSB to avoid a previously necessary right shift. (E.g. 10 being represented as `10...001010` rather than `00...010101`.) SmallIntegers can now be seen as negative object pointers; therefore, only a simple signed comparison (pointer <= 3) is necessary to determine whether an object has to be reference-counted or not. (Object pointers 0 to 3 (*invalid object pointer*, *nil*, *true* and *false*) do not need to be reference counted.)

- The object table was split up into separate arrays for the reference counts, the addresses and the flags.
- The concept of *destructive moving* was used to pass a return variable from the callee's to the caller's context. This avoids the overhead of incrementing the reference counter when adding the variable to the caller's context and decrementing it when destroying the callee's context.
- Stack management has been improved to decrease the amount of required context scans and to reduce the number of reference-count operations.

DEC VAX/Smalltalk-80

DEC decided to implement an incremental, compacting garbage collector based on the Baker Garbage Collector ([Bak78][4]). The basic idea is to split the memory into two spaces (the Fromspace and the Tospace) and copy the "root" objects from one space to the other. Objects referenced by these root objects or other already copied objects are also being copied. Therefore, everything that is transitively accessible will be copied; the rest is garbage and will be reused when, on the next garbage collector run, the locations of the two spaces are switched (a so-called "flip").

However, this algorithm proved to be inefficient, because context objects would fill up free space rapidly. The authors of [Bak78] suggest using the Lieberman-Hewitt Garbage Collector ([LH83]) instead. This adaptation would divide the available memory into a lot of little Baker spaces (i.e. spaces, that are garbage collected using the Baker algorithm). One space would contain objects of about the same age. This would result in all the small new context objects being in the same small space. This space would frequently be garbage collected; however, there would be no need to move the large objects contained in the other spaces.

A context reclamation algorithm has been implemented to avoid the strain on the garbage collector caused by frequent creation and destruction of context objects: If the active context object does not need to be referenced by other objects, it can be reclaimed immediately upon returning and be stored in a linked list of free context objects.

Squeak

Instead of reference counting, Squeak uses a two-generation mark-and-sweep approach to achieve garbage collection, based on the method used by Apple Smalltalk. Mark-and-sweep basically means that periodically the memory is scanned and all objects that have any variable pointing at them are marked. Unmarked objects are removed.

[4]If you do not want to read the entire paper, [BS83] provides a small overview of the algorithm used.

4 Further Information

The classic source of information about Smalltalk-80 are the “Blue Book” [GR83], the “Green Book” [Kra83], and the “Red Book” [GR84]. They are out of print, but some chapters of the Blue Book have been made available online: `http://users.ipa.net/~dwighth/smalltalk/bluebook/bluebook_imp_toc.html`. Further Smalltalk information can be found at `http://www.smalltalk.org` and at one of the Smalltalk user groups around the world (`http://www.smalltalk.org/usergroups.html`).

The Squeak executables, its source code and further information about the project can be found at `http://www.squeak.org`.

5 Conclusion

Through its years of development, Smalltalk has become a mature object-oriented environment with the general concepts already being present in the very first implementation. It has been shown that Smalltalk can be created for any platform that meets the minimum requirements and that the reference implementation’s performance can be dramatically increased by use of more efficient memory management algorithms and by taking advantage of the host processor’s characteristics. Recent implementations show that this design can be used to create easy to use environments including the features (sound, graphics, TCP/IP etc.) one has come to expect from a modern class library-based development system.

References

[Bak78] H. G. Baker. Actor systems for real-time computation. Technical Report MIT/LCS/TR-197, 1978.

[BS83] Stoney Ballard and Stephen Shirron. The design and implementation of vax/smalltalk-80. In Krasner [Kra83], chapter 8.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys (CSUR)*, 13(3):341–367, 1981.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[GR83] Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[GR84] Adele Goldberg and D. Robson. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[IKM$^+$97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 318–326. ACM Press, 1997.

[Ing83] Daniel H. H. Ingalls. The evolution of the smalltalk virtual machine. In Krasner [Kra83], chapter 2.

[Kra83] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[MC83] Richard Meyers and David Casseres. An mc68000-based smalltalk-80 system. In Krasner [Kra83], chapter 10.

[McC83] Paul L. McCullough. Implementing the smalltalk-80 system: The tektronix experience. In Krasner [Kra83], chapter 5.

[UP83] David M. Ungar and David A. Patterson. Berkeley smalltalk: Who knows where the time goes? In Krasner [Kra83], chapter 11.

[WB83] Allen Wirfs-Brock. Design decisions for smalltalk-80 implementors. In Krasner [Kra83], chapter 4.