The Smalltalk-76 Programming System Design and Implementation

Daniel H. H. Ingalls
XEROX Palo Alto Research Center
Palo Alto, California

Abstract
This paper describes a programming system based on the metaphor of communicating objects. Experience with a running system shows that this model provides flexibility, modularity and compactness. A compiled representation for the language is presented, along with an interpreter suitable for microcoding. The object-oriented model provides naturally efficient addressing; a corresponding virtual memory is described which offers dense utilization of resident space.

INTRODUCTION

The purpose of the Smalltalk project is to support children of all ages in the world of information. The challenge is to identify and harness metaphors of sufficient simplicity and power to allow a single person to have access to, and creative control over, information which ranges from numbers and text through sounds and images. In our experience, the SIMULA notion of class and instance is an outstanding metaphor for information *structure*. To describe *processing*, we have found the concept of message-sending to be correspondingly simple and general. Rather than provide this organization as a "feature" in an existing system, we have taken these two metaphors as the point of departure for the Smalltalk programming language. The result is a lively interactive system which provides its own text editing, debugging, file handling and graphics display on a personal computer.

The Smalltalk language is object oriented rather than function oriented, and this often confuses people with previous experience in computer science. For example, to evaluate <some object>+4 means to present +4 as a message to the object. The fundamental difference is that the object is in control, not +. If <some object> is the integer 3, then the result will be the integer 7. However, if <some object> were the string 'Meta', the result might be 'Meta4'. In this way, meaning rides with the objects of the system, and code remains an abstract form, merely directing the flow of communication. As we shall see, this separation is a key factor in the ability of a system to handle complexity.

THE PRINCIPLE OF MODULARITY

No part of a complex system should depend on the internal details of any other part. The design of the Smalltalk language supports this principle through uniform reference to *objects*, sending *messages* to obtain results, and through organization of object descriptions and computational methods into *classes*. Factoring of information structure and behavior is provided by the implementation of *subclassing*.

All references in the Smalltalk language are to objects, which may be atomic, or may consist of several named *fields* which refer in turn to other objects. Though the only truly atomic datum is the *bit*, it is often appropriate for such simple objects as names and numbers to be considered atomic. For example, in a piece of code which reads the y-coordinate of a point in two dimensions, nowhere should there be instructions such as: "load the second word relative to point p." Such an instruction would not perform properly if it encountered a point represented in polar coordinates.

Communication is the metaphor for processing in the Smalltalk language. Objects are created and manipulated by sending messages. The same model describes activities ranging from ordinary arithmetic to communicating processes in separate machines. The response to a message is implemented by a *method*, which reads or writes some data field, or sends further messages to achieve the desired response. The communication metaphor supports the principle of modularity, since any attempt to examine or alter the state of an object is sent as a message to that object, and the sender need never know about internal representation. For example, points represented in polar coordinates r and theta would have a method which responds to the message, y, by returning r*theta cos.

The examples in this paper use a syntax which gives higher precedence to messages of fewer arguments. Therefore theta first receives the message cos (remember, cosine is a property of angles, not a function!), and then r receives the message * with one argument, theta's cosine.

Every object in Smalltalk is created as an *instance* of some class. The class holds the detailed representation of its instances, the messages to which they can respond, and methods for computing the appropriate responses. The only information remaining to be stored in an instance is the set of values for its named fields. The class is the natural unit of modularity, as it describes all the external messages understood by its instances, as well as all the internal details about methods for computing responses to messages and representation of data in the instances.

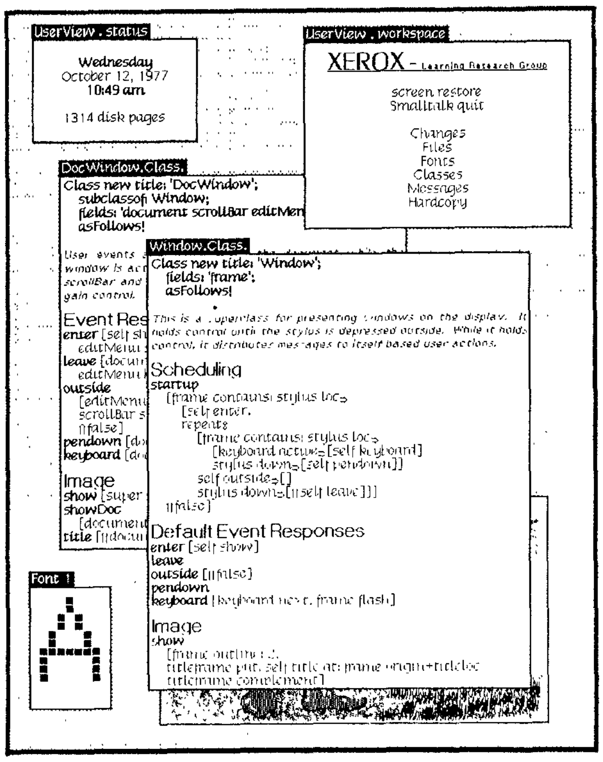
In the Smalltalk language, a class can be declared to be a subclass of another class, and thus inherit all the traits of that superclass. The subclass can then add traits of its own, can override those it wishes, and can still invoke the overridden ones from within its code. This capability leads to a highly factored system.

THE REACTIVE PRINCIPLE

The salient feature of Smalltalk is that all objects are active, ready to perform in full capacity at any time. Nothing of this aliveness should be lost at the interface to the human user of the system. In other words, all components of the system must be able to present themselves to the user in an effective way, and must moreover present a set of simple tools for their meaningful alteration.

The Smalltalk analogy to a library of useful functions is a set of well developed superclasses from which most of the system classes are derived. The most general of these, class Object, provides default behavior for printing any object, or examining and altering its contents on a display screen.

In addition to a keyboard and display screen, the Smalltalk system provides a stylus for pointing at the screen. The Smalltalk system includes a class Window which establishes a uniform model for interaction with these devices. From Window, users can define new subclasses with behavior specific to their content. The inheritance from class Window avoids a lot of replicated code and furnishes a uniform model for reactive control over new objects added by the user.



```
Class new title: 'DocWindow';
   subclassof Window;
   fields: 'document scrollBar editMen
   asFollows!
```

```
Class new title: 'Window';
   fields: 'frame';
   asFollows!

This is a superclass for presenting windows on the display. It
holds control until the stylus is depressed outside. While it holds
control, it distributes messages to itself based user actions.

Scheduling
startup
   [frame contains: stylus loc⇒
      [self enter.
      repeat⇒
         [frame contains: stylus loc⇒
            [keyboard active⇒[self keyboard]
            stylus down⇒[self pendown]]
         self outside⇒[]
         stylus down⇒[⇑self leave]]]
   ⇑false]

Default Event Responses
enter [self show]
leave
outside [⇑false]
pendown
keyboard [keyboard next. frame flash]

Image
show
   [frame outline: 2.
   titleframe put: self title at: frame origin+titleloc.
   titleframe complement]
```

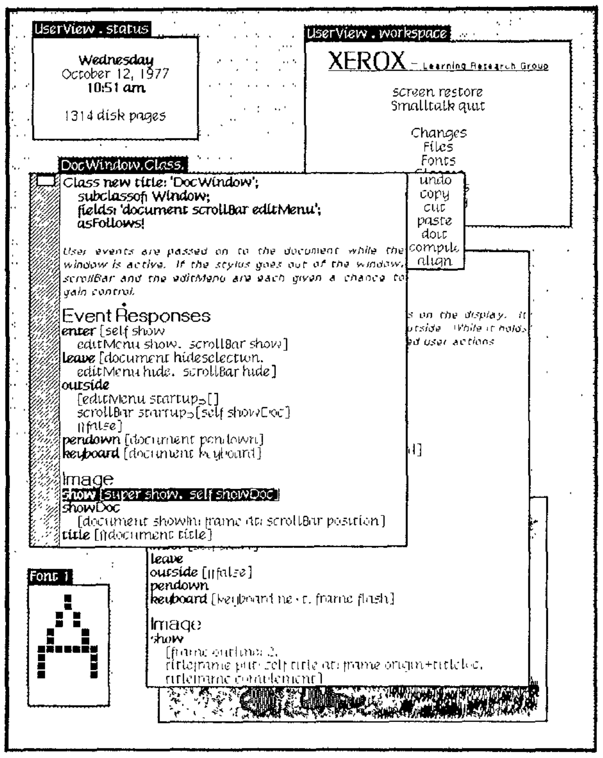
Figure 1. A typical Smalltalk display

Figure 1 shows a typical display screen from a Smalltalk system. Each of the rectangular areas is an instance of class Window which provides a basic protocol for user interaction:

- if the stylus is depressed within the window, the content is redisplayed, thus showing it "in front of" the other windows;
- once active, the window becomes the recipient of all user actions, such as depressing the stylus, typing on the keyboard, or entering sensitive areas around the window;
- if the stylus is depressed outside the window, control is given up so that another window may be awakened.

The class Window also provides a basic protocol for handling such actions as moving itself, changing its size, printing its contents, and terminating its appearance on the screen.

The reader may find himself evaluating the user interface presented in the figures. This misses the point, which is the aptness of communicating objects in describing the situation. Many other approaches to display handling are used with Smalltalk, and they are not the subject of this paper.

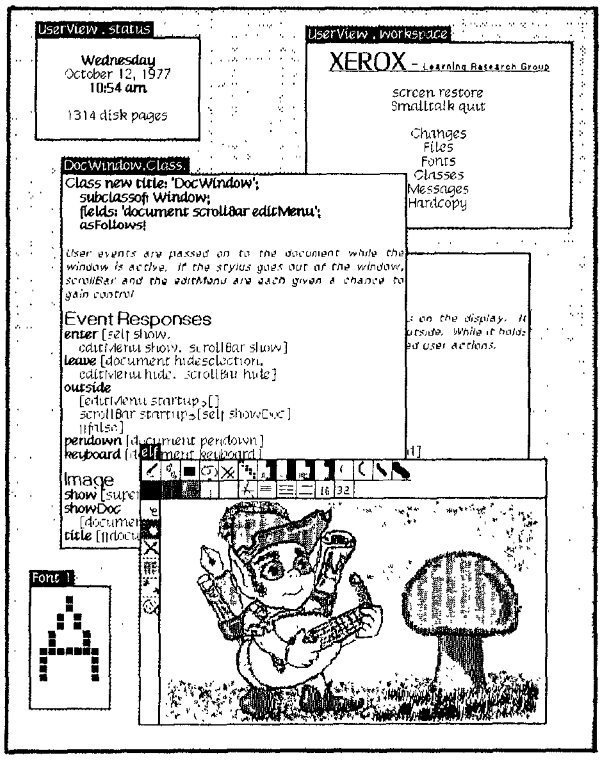


```
Class new title: 'DocWindow';
   subclassof Window;
   fields: 'document scrollBar editMenu';
   asFollows!

User events are passed on to the document while the
window is active. If the stylus goes out of the window,
scrollBar and the editMenu are each given a chance to
gain control.

Event Responses
enter [self show
   editMenu show. scrollBar show]
leave [document hideselection.
   editMenu hide. scrollBar hide]
outside
   [editMenu startup⇒[]
   scrollBar startup⇒[self showDoc]
   ⇑false]
pendown [document pendown]
keyboard [document keyboard]

Image
show [super show. self showDoc]
showDoc
   [document showin: frame at: scrollBar position]
title [⇑document title]
```

Figure 2. Editting text

Each of the windows shown in the figure belongs to some subclass of Window; the subclass handles the specific content. In figure 2, for example, the user has awakened a window for editing the text of a class. Specific to this content is the ability to select text with the stylus, scroll the text up and down by entering the scroll bar at the window's left, and send specific editing messages by entering the menu at its right. The class has thus provided a simulation of itself as structured text (which is needed for printing anyway). This text in turn furnishes a text editor to display the text and allow it to be manipulated. Finally, the window provides a spatial channel for the flow of information in both directions between the user and the subject of investigation. As shown in figure 2, the user has just drawn the stylus through the text of the message show, and the editor has responded by highlighting the text and noting the selection internally.

In figure 3, the currently active window supports freehand drawing. It provides a large menu from which to choose brush shape and paint tone to be applied when the stylus is depressed. Another window interfaces to one of the character fonts, allowing the user to design new fonts at will. Yet another displays the time of day. Each of the windows brings its own semantics to the uniform "syntax" of stylus motion and keyboard action available to the investigator. In this way, the underlying metaphor of communicating objects can be seen to operate all the way up to the level which corresponds to a conventional operating system.



```
Wednesday
October 12, 1977
10:54 am

1314 disk pages
```

```
XEROX - Learning Research Group

screen restore
Smalltalk quit

Changes
Files
Fonts
Classes
Messages
Hardcopy
```

```
Class new title: 'DocWindow';
   subclassof: Window;
   fields: 'document scrollBar editMenu';
   asFollows!

User events are passed on to the document while the
window is active. If the stylus goes out of the window,
scrollBar and the editMenu are each given a chance to
gain control

Event Responses
enter [self show.
   editMenu show. scrollBar show]
leave [document hideselection.
   editMenu hide. scrollBar hide]
outside
   [editMenu startup⇒[]
   scrollBar startup⇒[self showDoc]
   ⇑false]
pendown [document pendown]
keyboard [document keyboard]

Image
show [super
showDoc
   [documen
title [⇑docu
```

Figure 3. Painting a picture

EXAMPLE OF A SMALLTALK CLASS: Rectangle
As we have seen above, most programming in Smalltalk is done using a structured editor. Rather than go into the details of this process, we show here an excerpt from class Rectangle, show how it supports the Window example above, and describe in detail the operation of one of its messages.

A Smalltalk class is defined by giving it a name, and naming the fields of its instances. Following this is an optionally categorized list of the messages to which the class responds. Each message consists of a pattern followed by Smalltalk code within brackets.

```
Class new title: 'Rectangle';
 fields: 'origin corner'.

Access to fields
origin  [⇑origin]        "⇑ means return"
corner  [⇑corner]
origin: origin corner: corner
        "no code; just store into the instance"

Testing
contains:pt"return true if pt is inside me"
        [⇑origin≤pt and: pt≤corner]
empty  [⇑corner≤origin]

Combination
inset: delta
        [⇑origin+deltarect: corner-delta]
intersect: r
        [⇑(origin max: r origin) rect:
            corner min: r corner)]

Image
clear:color [primitive]  "display operation"
outline: width
        [(self inset: (-1⊡-1)*width)
             clear: black.
        self clear: white]
moveto: pt
        [corner ← corner-origin+pt.
        origin ← pt],
```

Rectangles have two fields: **origin** is the lower left corner, and **corner** is the upper right corner. Because both **origin** and **corner** are Points, Rectangle builds on a data domain which is already graphically interesting.

```
Class new title: 'Point';
 fields: 'x y'.      "Cartesian coordinates"

Access to fields
x[⇑x]
y[⇑y]
x: x y: y

Testing
≤ pt "return true if I am below/left of pt"
        [⇑x≤pt x and: y≤pt y]

Combination
rect:c      "make up a new rectangle"
        [⇑Rectanglenew origin: self corner: c]

Point arithmetic
+ pt     [⇑Point new x: x+pt x y: y+pt y]
- pt     [⇑Point new x: x-pt x y: y-pt y]
```

The scheduling of windows shown in figures 1-3 is implemented in Window's response to the message **startup** (visible in figure 1). The chief element in the code is the expression

```
frame contains: stylus loc
```

which checks whether the stylus location (a Point) is within the frame (a Rectangle) of the current Window. The code for Rectangle **contains:** builds, in the procedural domain, on the meaning of ≤ for Points. The first test determines that **origin** is below and to the left of the Point **pt**, and the second test determines that **pt** is below and to the left of **corner**. The reader will want to know here that arithmetic messages (+, -, etc.) get sent before keyword messages (the identifiers ending in colon). Therefore, after the two ≤ tests have been made, the conjunction of the two results defines the case that the rectangle contains the Point **pt**.

Smalltalk treats all objects which are not false as true. This is effected rather easily in class Object by messages of the form:

```
and: x [self=false⇒[⇑false] ⇑x],
```

which are inherited by all other classes. The implication arrow is a primitive operation which compiles into a branch conditional on the distinguished object, false.

The other messages to Rectangle are presented so that interested readers can pursue the message **outline: w**, which causes an outline of width **w** to appear around the rectangle on the screen. This message is used by class Window to clear and outline its frame.

These readers will want to know that if several keyword messages appear at a given level, they combine to provide a message with multiple arguments. For example, class Number provides a message

```
⊡ arg [⇑Point new x: self y: arg]
```

for creating Points simply. This code sends the compound message x:y: to the new Point with the two arguments, self and arg.

The Benefits of the Message Discipline
Adding a new class of data to a programming system is soon followed by the need to print objects of that class. In many extensible languages, this can be a difficult task at a time when things should be easy. One is faced with having to edit the system print routine which (a) is difficult to understand because it is full of details about the rest of the system, (b) was written by someone else and may even be in another language, and (c) will blow the system to bits if you make one false move. Fear of this often leads to writing a separate print routine with a different name which then

must be remembered. In our object-oriented system, on the other hand, printing is always effected by sending the message **printon:** s (where s is a character stream) to the object in question. Therefore the only place where code is needed is right in the new class description. If the new code should fail, there is no problem; the existing system is unmodified, and can continue to provide support.

The code for moving a Rectangle (**moveto:**) could be more efficient if **corner** were relative to **origin**, so that it did not need to be relocated. This could be accomplished by changing names from

```
        fields: 'origin corner'
to
        fields: 'origin extent',
```

and updating the code from

```
        moveto: pt   [corner ← corner-origin+pt.
                      origin ← pt]
to
        moveto: pt   [origin ← pt].
```

In a conventional organization, all the code in the system will have to be recompiled because it used to count on finding the corner coordinates in the second field. In fact most systems would require *rewriting* because access to the corner (which used to be a simple load or store) becomes a computation:

```
        corner       [⇑origin+extent]
        corner:      [extent← pt-origin]
```

The class organization and message discipline ensure that if the original message protocol is supported, then all code outside the class will continue to work without even recompiling. Moreover, the only changes required will all be within the class whose representation is being changed.

Modularity is not just an issue of "cleanliness." If N parts depend on the insides of each other, then *some* aspect of the system will be proportional to N-squared. Such interdependencies will interfere with the system's ability to handle complexity. The phenomenon may manifest itself at any level; difficulty of design, language overgrown with features, long time required to make a change or difficulty of debugging. Messages enforce an organization which minimizes interdependencies.

Another benefit of leaving message interpretation up to the target objects is type independence of code. In the Rectangle example, the code will work fine if the coordinates are Integer or FloatingPoint, or even some newly-defined numerical type. This allows for much sharing of code, and the ability to have one object masquerade as another.

The Power of Subclasses

The utility of type-independent code is exploited by the subclass organization in Smalltalk. For example, the superclass Number implements the messages

≤ ≥ ≠ max: min:

in terms of the basic comparisons, <, =, and >. Consequently, after defining only the basic comparisons, any new subclass of Number can respond to the full numerical protocol. Other messages in the superclass Number, such as ⌾ (which creates a Point) and **to:by:** (which creates an interval), similarly allow coordinates of Points, and limits of for-loop ranges to be FloatingPoint, Fraction, or any other kind of Number.

The factoring provided by subclassing leads to cleanliness and compactness. It can also increase efficiency through multiplied use of high-speed system code. For instance, we provide microcode support for reading and writing the next item in a Stream. Now by making class DiskFile a subclass of Stream, DiskFile inherits this very efficient code for its most common operations. The superclass Stream sends itself the message **pastend** when it reaches the end of its character buffer, and this gives the subclass DiskFile a chance to read a new record off the disk by overriding the meaning of **pastend**.

Besides allowing subclasses to override messages in their superclasses, Smalltalk provides for access to the overridden messages. In figure 2, the selected text defines the message **show** to DocWindow,

```
        show  [super show. self showDoc]
```

which overrides Window's definition. The token, **super**, indicates that the following message should be looked up beginning with the class above. In this case it allows DocWindow to invoke Window's **show** code to put a frame around the window before telling the document to display its text.

Further properties of Smalltalk subclassing, including multiple inheritance of traits, are beyond the scope of this paper.

IMPLEMENTATION

The challenge of implementing the Smalltalk language is to provide acceptable performance without compromising the simplicity of the underlying metaphors. Beyond the laudable goals of modularity and factoring, four pragmatic issues affect the performance of the Smalltalk system: storage management, compact object code, message handling, and a decent virtual memory.

Storage Management

Requiring the programmer to manage the allocation and deallocation of objects is out of the question in a true high-level language. It is a sure source of errors and it clutters the code with irrelevant pragmatics. Instead, the Smalltalk kernel maintains reference counts for all objects, and frees them to be reused when the count goes to zero. We chose reference counting over garbage collection because the computational overhead is more uniform in real-time, and the behavior with little free storage is better. The dominant flow of data in a programming system is usually small integers, either representing characters or results of simple arithmetic and indexing. The Smalltalk system reclaims small integers without reference counting (they are special object references); this one wild-card (present in some LISP systems) reduces the overhead to a reasonable level.

The reclamation of cyclic structures cannot be handled by reference counting, but we have not found this to be a problem. The class architecture makes it especially convenient to manage objects which are going to participate in cyclic structures. For each cyclic link field, an appropriate **unlink** message can be defined which propagates through the structure breaking the cycle. It is true that this discipline corresponds to explicit storage management, but it is well localized, and hence presents little problem.

Compact Object Code

Dear to our hearts is the ability to run Smalltalk on personal computers. In this arena, compactness of the system is crucial and worth sacrificing some raw speed to achieve. Moreover, our experience with virtual memory systems is that overall performance actually improves with compactness owing to reduced swapping of the workingset. To this end, the Smalltalk system translates source code into compact syllables corresponding to the elements of the language.

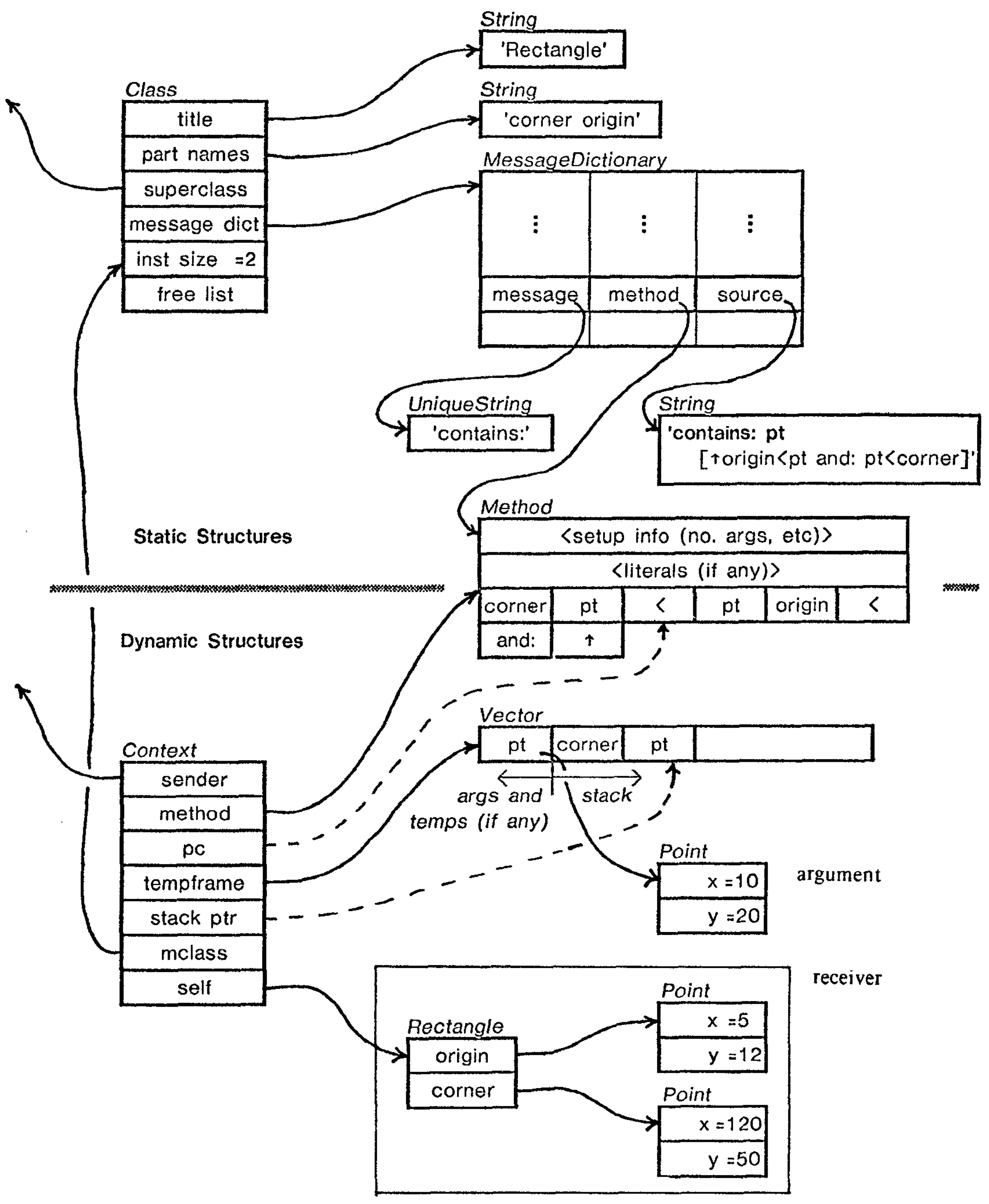


Figure 4 - Smalltalk Structures

The top of figure 4 illustrates the static structure in Smalltalk-76 corresponding to Rectangle **contains:**. One of the fields of the class points to an object called a MessageDictionary. This is a hash table whose keys are the message names (UniqueStrings) and whose value parts include the corresponding compiled method and also the source code. The compiler translates the source code into syllables arranged in postfix order using what the class knows about instance fields and other variable locations. Including some setup information discussed below, the code body for computing **contains:** occupies 14 code syllables or, in our system, seven 16-bit words.

The choice of encoding is made by translating every common item into a single syllable, and then providing escapes for cases where this is impossible. The codes fall into the following basic categories:

1	Load relative to receiver (self)
2	Load relative to temp frame
3	Load relative to literals
4	Load indirect literals
5	Load relative to context
6	Load constant
7	Send literal message
8	Send special message
9	Short jumps
10	Long jumps
11	Control ops
12	Escapes

The loads pick up an object reference and push it onto an evaluation stack, and the message codes either cause a primitive operation or invoke another context, depending on the actual message and the receiver on top of the stack. The jumps come in two flavors, unconditional and conditional, the latter taking place only if the top of stack is false, but popping the stack in any event. The long jumps are followed by an extension syllable. The control ops include returning to caller, pop stack, and store, which is followed by a load-like syllable describing the destination field. The escapes work like the long jumps, extending the loads and sends by a full syllable of offset. An area is set aside in the method for literals (object references which do not fit into a syllable), and these serve when necessary as an escape to refer to unusual constants, external references or message tokens. Our example (which does not use any literals) would be encoded as

```
origin    load  receiver.0
corner    load  receiver.1
pt        load  temp.0
≤         send  special  message.7
and:      send  special  message.12
⇑         do  control.3
```

The encoded methods typically occupy one quarter the space of the original source code. In implementations with no virtual memory, therefore, it would be important to reconstruct the source from the code syllables.

Message Handling
Execution of the code syllables is relatively simple, and the lower half of figure 4 illustrates the dynamic environment. When a Rectangle receives the message **contains:**, an instance of class Context is allocated to serve much the function of a conventional call frame. Its **sender** field points to the context of its caller, to which the final value of its computation will be passed with the return (⇑) operator. The **self** field points to the object who is receiving this message; in our example it is a Rectangle. The **mclass** field in this example is simply the class Rectangle, but in the case of messages which are inherited from superclasses, it indicates the class from which the message was inherited (this is needed to provide the *super* access described earlier). **Method** points to the object code for the method being executed, and **pc** is a syllable offset into it indicating the current progress of execution. **Tempframe** points to a Vector of temporary storage for the computation. Its access is partitioned into three areas: arguments passed from its caller, other temporaries required for the computation (none in our example), and the stack area used in evaluating the syllables. **Stackptr** is the offset indicating the current top of stack.

The Smalltalk system gets a great deal of mileage out of the fact that Contexts are objects of full capability, just like all the other parts of the system. For debugging, the Context is in a position to reveal the location of the pc, the contents of the stack, the values of temporary variables and the current state of the receiver. For analysis purposes, the operation of the interpreter is simulated Smalltalk by a loop which sends the message **step** to the current context:

repeat⁸ [current ← current step].

An outline of this simulation appears in an appendix to this paper.

Most of the task of interpretation is implicit in the syllable descriptions above. The one complex operation worthy of further clarification is how a new context gets set up to send a message to a new receiver. Our example of figure 4 will serve us again: **corner** has been put on the stack, and **pt** above it is the top item in the stack, and the next syllable pointed to by **pc** indicates that the message ≤ is to be sent. The very first thing that happens (since this is a common arithmetic operation) is that **pt** is checked to see if it is an Integer allowing a fast primitive operation; but it is not. The more general process then begins of hashing the token ≤ into the MessageDictionary of **pt**'s class, Point. A match is indeed found, and the resulting code object can be installed in the new context along with **pt** (popped off the stack) in the **self** field, class Point in the **mclass** field, and the previous context in the **sender** field.

The setup parameters in the beginning of the method are computed at compile time, and serve to initialize the new environment. They tell how much space is needed for the **tempframe** and how many arguments are expected (one here). After the arguments have been moved from the previous stack to the beginning of the tempframe, the setup supplies initial values of **stackptr** and **pc**, and the new context is ready to run. It will eventually terminate by encountering a *return* syllable. Then the value (true or false) on the top of the stack will get pushed back onto the sender's stack, and the computation of Rectangle **contains:** will resume where it left off.

Efficient Message Handling
It should be obvious that to carry the message sending metaphor down into integer arithmetic will result in a slow system on any non message oriented computer. The implementor must cheat, but not get caught. In other words, the user of the system should not perceive any non-uniformity.

To achieve acceptable performance, it is important that the microcode interpreter *not* send messages to Contexts. It performs all the same steps, but in the fastest way possible.

The Smalltalk system achieves its speed by the assignment of special code syllables to the most frequently executed operations. For instance, the message + is assigned a special code, and the (microcode) interpreter responds to that code by immediately checking whether both receiver and argument are of class Integer. If so, it does the addition in-line and proceeds immediately without even looking up the symbol plus, let alone creating a new context. If the check fails, then control passes to the normal message sending code, thus insuring the appearance of uniformity.

Virtual Memory

One of the most satisfying aspects of a system built of objects is the naturally efficient use of address space. Since a Smalltalk reference can only point to an object as a whole, the Smalltalk system can indicate 2↑16 distinct objects using 16-bit object references. With a mean object size of 20 words, the system can address over a million words, and this size is a nice match to the size of common personal computer disk drives. In implementing this virtual memory, we decided to swap single objects, thus achieving maximum utilization of resident storage at the possible expense of slower establishment of workingset. Without a few tricks this scheme could never work.

First, a poor man's Huffman code associates the class of an object with its object reference, thus avoiding a 1-word class-pointer overhead on every object. The address space is given to classes for their instances in *zones* of 128 with the same high-order bits. It is therefore possible to index a table with the high order bits of any object reference and discover the class of that object.

Second, variable length classes have many such zones, one for each of the common small lengths. In this way the overhead of a length word is avoided on the small sizes. For larger sizes, the zones cover octave ranges (i.e., 9-16, 17-32, and so on) and an extra word specifies the exact length.

Third, these zones of address space are mapped into contiguous locations on the disk. Since all the objects are instances of the same class, they have the same size; hence the page and offset on the disk can be computed from the low order bits of the object reference. In this way non-resident objects can be located on the disk to be brought into core. For the octave sizes, a little wasted space on the disk is traded for the ability to compute the disk offset location.

Fourth, the zoning provides a solution to the problem of making free space without thrashing the disk. Here is the problem: there are 1000 objects in core, roughly 20 words each, and we suddenly need 500 words of space; if we have to write out 25 objects (assuming we can even decide which ones), how do we keep from doing 25 disk seeks to do it? Our solution is to purge core by zone, thus writing many objects together on a page, and sweeping over the disk in order of pages. Moreover, a single bit on each object meaning "I've been touched since my zone was last purged" provides a nice aging criterion for selecting objects to be purged.

CONCLUSION

We have known for some time that the uniform model of communicating objects leads to a naturally integrated programming system. A very gratifying aspect to our work is the demonstration that this approach can also result in an efficient implementation. The Smalltalk system has supported large applications for simulation, information retrieval, text editing, musical composition and animation without encountering significant barriers to complexity, and this is our strongest indication of the worth of the language. Beyond this I must add that programming in Smalltalk is fun. On the one hand, the act of assembling expressions into statements and then into methods is not very different from conventional programming. On the other hand, the experience is totally different, for the objects which populate and traverse the code are active entities, and writing expressions feels like organizing trained animals rather than pushing boxes around.

Acknowledgement

Many people have contributed to the Smalltalk system. Alan Kay has been expounding the basic principles for about a decade now, and the rest has unfolded through the love and energy of the whole Learning Research Group. Smalltalk-76 was implemented by the author with the assistance of Ted Kaehler, Diana Merry, and Dave Robson. Several other people at PARC have contributed significantly, most notably Larry Tesler and Peter Deutsch.

References

Kay, A. *FLEX, a flexible extensible language*
M.S. thesis, Univ of Utah, May, 1968 (Univ. Microfilms).
Kay, A. *The Reactive Engine*
PhD. thesis, Univ of Utah, Sept., 1969 (Univ. Microfilms).
Learning Research Group. *Personal Dynamic Media.*
SSL76-1, Xerox PARC, Palo Alto, Calif., April 1976.
Birtwistle, G., Dahl, O.-J., Myhrhaug, B., Nygaard, K.
Simula Begin. Auerbach, Philadelphia, Pa., 1973.
Fisher, D. A. *Control Structures for Programming Languages.*
PhD. thesis, Carnegie-Mellon Univ. Pittsburg, 1970
Liskov, B. and Zilles. S. *Programming with Abstract Data Types.* SIGPLAN Notices, April 1974, 50-59.
Liskov, B. *An Introduction to CLU.*
CSG Memo 136, MIT LCS, Febuary 1976.
Greif, I. and Hewitt, C. *Actor Semantics of PLANNER-73.*
ACM SIGPLAN-SIGACT Conf., Palo Alto, Calif., Jan 1975.
Steiger, R. *Actor Machine Architecture.*
M.S. thesis, MIT Dept. EECS, June 1974.
Deutsch, L. P. *A LISP Machine with Very Compact Programs.*
IJCAI, Stanford, Calif., August 1973.
Deutsch, L. P. and Bobrow, D. *An Efficient Incremental Automatic Garbage Collector.* CACM September 1976.

APPENDIX: A Smalltalk Simulation of the Interpreter

```
Class new title: 'Context';
   fields: 'sender method pc tempframe stackptr mclass receiver';
   asFollows!
```

Contexts carry the dynamic state of Smalltalk processes. They are accessed in efficient ways by the microcode interpreter. At the same time, they are instances of a perfectly normal Smalltalk class. In this way, the full generality of Smalltalk can be applied to examining and tracing the progress of Smalltalk execution.

The code below differs from the actual code in Smalltalk-76 in that it corresponds to the slightly simplified categories of the text, and has not been carefully checked for off-by-1 errors.

Beyond the specifics in the text, the interested reader will want to know:
- *"◦" is the subscript message, as in: tempframe◦lobits*
- *except for assignment, "←" is treated as an agglutinating message part, as in: t◦i ← self pop*
- *the ⇒ symbol indicates conditional execution; if the preceding value is true, then the following body of code is executed, and control exits the outer (!) brackets. This "if-only" form serves to build dispatch tables as in the message "next" below*
- *the default value returned from any message is "self", the receiver of the message. Other values may be returned with the "⇑" symbol.*

The messages "instfield: n" and "instfield: n← val", which are used below to read and write the n-th field of an instance, clearly violate the principle of modularity. This reflects that the buck stops here, and these primitive messages appear nowhere else in the system.

Access to Fields

```
sender: sender method: method pc: pc tempframe: tempframe stackptr: stackptr
   mclass: mclass receiver: receiver    "initializes all fields"
```

Simulation of the Interpreter

```
step | byte lobits                      "dispatch on next code syllable"
   [byte ← self nextbyte.
   lobits ← byte|16.
   byte/16=1⇒[self push: receiver instfield: lobits];        "load from instance"
          =2⇒[self push: tempframe◦lobits];                   "load from temps (and args)"
          =3⇒[self push: (method literals: lobits)];          "load from literals"
          =4⇒[self push: (method literals: lobits) value];    "load indirectly from literals"
          =5⇒[self push: self instfield: lobits];             "load from this Context"
          =6⇒[self push: ☞(¯1 0 1 2 10 true false nil)◦lobits];  "frequent constants"
          =7⇒[⇑self send: (method literal: lobits)];
          =8⇒[⇑self send: (SpecialMessages◦lobits)];          "frequent messages"
          =9⇒[lobits<8⇒[pc← pc+lobits]                        "short jump forward"
                self pop⇒[] pc← pc+lobits-8];                 "short branch if false and pop"
          =10⇒[lobits<8⇒[pc← lobits-3*256+self nextbyte+pc]"long jump forward and back"
                self pop⇒[pc← pc+1];                          "skip extension byte on true"
                pc← lobits-11*256+self nextbyte+pc];          "long bfp"
          =11⇒[lobits=0⇒[self pop];                           "pop stack"
                =1⇒[self store: self top into: self nextbyte]; "store"
                =2⇒[self store: self pop into: self nextbyte]; "store and pop"
                =3⇒[sender push: self top.  ⇑sender]]       "return value to sender"
   ]
store: val into: field | lobits            "same encoding as above"
   [lobits ← byte|16.
   field/16=1⇒[receiver instfield: lobits ← val];      "store into instance"
          =2⇒[tempframe◦lobits ← val];                  "store into temps (and args)"
          =3⇒[user notify: 'invalid store'];            "can't store into literals"
          =4⇒[(method literals: lobits) value ← val];"store indirectly through literals"
          =5⇒[self instfield: lobits ← val]             "store into this Context"
   ]
send: message | class meth callee t i      "send a message"
   [class ← self top class.
   until⦂ (meth← class lookup: message) do⦂ "look up the method"
      [class← class superclass.             "follow the superclass chain if necess"
      class=nil⇒[user notify: 'Unrecognized message: '+message]]
   [meth primitive⇒                         "If flagged as primitive, then do it"
      [self doprimitive: meth  ⇒[⇑self]]].  "If it fails, proceed with send"
   callee← Context new                      "create new Context, and fill its fields"
      sender: self method: meth pc: meth startpc
      tempframe: (t← Vector new: meth tframesize) stackptr: meth startstack
      mclass: class receiver: self pop.
   for⦂ i to: meth nargs do⦂                "pass arguments"
      [t◦i← self pop]
   ⇑callee]                                 "return new Context, so it becomes current"
nextbyte       "step pc and return next code syllable"
   [⇑method◦(pc← pc+1)]
```

Stack-related Messages

```
push: val       "push value onto top of stack"
   [tempframe◦(stackptr← stackptr+1) ← val]
top   "return value on top of stack"
   [⇑tempframe◦stackptr]
pop | t         "pop value off stack and return it"
   [t← tempframe◦stackptr.
   stackptr← stackptr-1. ⇑t]
```