A Taste Of Smalltalk:

Object-Oriented Programming From The Bottom Up

One of the things that surprised me as I read through this and the other object-oriented articles in this issue, was the similarities among languages. The ideas of objects, classes, and messages are definitely key. It's where things are bound (fancy term for when space is allocated) that seems to change from interpreter (Smalltalk) to compiler (C++). Check out all these pieces, I think you'll really get a sense of these unique languages.

In the early 70s, the Xerox Palo Alto Research Center in California began developing a new programming language — Smalltalk. Several versions later, in 1981, they released Smalltalk-80. In August '81, *BYTE* devoted an entire issue to Smalltalk-80, introducing the micro computer world to object-oriented programming.

There are now several versions of Smalltalk; we list them (and other sources) at the end of this article.

What's In A Smalltalk?

When you buy a Smalltalk, you get:

By Jan Steinman and Barbara Yates
Tektronix, Inc.
P.O. Box 500, Mail Sta. 50-470
Beaverton, OR 97077

(1) A virtual image file which contains the source code for a group of core classes.

(2) A starter set of programming tools.

(3) An interpreter.

You can also add on packages of additional classes. Classes are the voices of Smalltalk.

Smalltalk is both a programming language and a programming environment, consisting of, at least: an editor, a debugger, and an interface to a file system. Ask any Smalltalk programmer what's special about Smalltalk, and he'll invariably mention how much fun it is.

Fundamental Talk

Basically, we program in Smalltalk by sending "messages" to "objects," which are "instances" of "classes."

Some classes represent fundamental objects such as integers, characters, and reals (or floating point numbers). The names of classes are always capitalized. So the corresponding classes for these basic types are Integer, Character, and Float.

Other classes represent more abstract objects. For example, Process and Stream. There's also a rich set of classes of Collections: for instance, Array, String, and Dictionary.

Your Smalltalk system *is* effectively its classes — from Cursor, to InputSensor, to Compiler, to View. You send messages to instances of classes, and things happen.

There's some confusion (isn't there always?) about classes because there's no standard class library. The same class may go by a different name in a different implementation. And Smalltalk implementations may contain fewer than one hundred or as many as several hundred classes.

Later we'll go a little deeper into classes and clear up some of the mystery.

> In Smalltalk, there's no program per se. At least in the beginning, think in terms of modifying the behavior of existing classes, rather than creating new data structures and operations.

Messages

Messages are represented symbolically by selectors. Different types of message selectors correspond to the number of objects involved in the message.

A unary message has one object, the object receiving the message. A binary message has two objects — the receiver and one argument.

Keyword messages are sent to the receiver with one or more arguments. Each part of a keyword selector ends with a colon. For example, when an object can't reply to a message, it sends itself the message —

```
doesNotUnderstand:
```

Precedence

Smalltalk uses precedence rules to resolve ambiguities —

- 1. expressions in parentheses,
- 2. unary messages,
- 3. binary messages,
- 4. keyword messages,
- 5. equal precedence messages are sent left-to-right,
- 6. variable assignment happens last. The symbols ":=" or "<-" (left-arrow) are assignment operators on different systems.

So in the Smalltalk statement —

```
stream := ReadStream on:
  (Array with: 'Joe','Blow') first.
```

the parenthetical expression must be manipulated first.

Within it is a binary message. So the concatenation selector "," is sent to the String 'Joe' along with the String argument 'Blow', resulting in the new String 'JoeBlow'.

This new String is then sent as an argument of the keyword message with: to the class Array, which responds with a new Array object containing the String 'JoeBlow'.

The unary selector "first" is then sent to the newly created Array, returning the first (and only) object in the Array, which is the String, 'JoeBlow'.

This String is then sent as an argument to the on: message to the class ReadStream. ReadStream responds with a new ReadStream object, which is assigned to the variable "stream".

Creating Classes & Behaviors

When we create a class, we (programmers) decide what kinds of data are needed to describe the instances of that class. The private data of instances is stored in instance variables.

All instances of a class can share data, as well. Shared data is stored in class variables.

The programmer decides the behavior of a class, and chooses the selec-

tor names of messages to send to the class (or to instances of the class) in order to start the behavior.

Methods are the procedures (or functions) of Smalltalk which execute when a message is delivered. In Smalltalk-80, methods are grouped into categories according to their function.

Programs & Objects

In traditional, structured languages, you write programs by defining data structures and the operations you want to perform on that data.

In Smalltalk, there's no program per se. At least in the beginning, think in terms of modifying the behavior of existing classes, rather than creating new data structures and operations.

An object is a private, persistent state that behaves in a particular manner through a publicly accessible interface.

The arrangement of the state of one object is unknown to other objects. In other words, you program by focusing on the way an object behaves (or manipulates data).

Messages Determine Behavior

The behavior of an object is determined by the messages an object can reply to. You can simulate message passing in many languages, but Smalltalk is the only popular language where message passing is the only means of communication among objects.

It's important to note that a message selector exists apart from an object. So the particular object a message is sent to is very likely indeterminable at the time the method is written.

Editor's note: For instance, in C++, memory for an object has to be allocated at compile time, whereas the size of a smalltalk object changes while the program is running.

This polymorphism is the biggest reason there are no Smalltalk compilers in common use, and why those which do exist (in research labs) place restrictions on message passing. It's also cited by backers of late-binding languages (like Smalltalk) as the key that turns a language truly object-oriented. In this sense, neither C++ nor Ada are truly object-oriented.

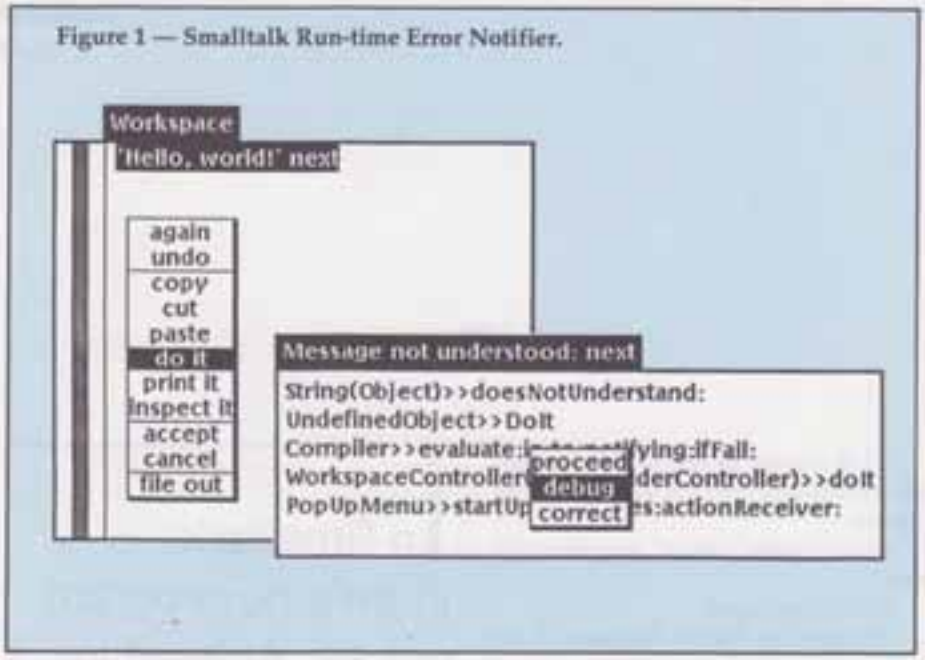
A Smalltalk message, like printString, can be sent to any object (regardless of class), and the object will respond by identifying itself with a String of some sort.

Modems talk to each other this way. And if you unintentionally reach a modem, when you're trying to reach a fellow human being, the modem will let you know that it doesn't understand by hanging up!

In Smalltalk, when an object can't reply to a message, it hangs up, sending itself the doesNotUnderstand: message, which we can intercept for error recovery.

For example, sending the message "next" to the string 'Hello, world' brings up a notifier window. (See Figure 1.)

Figure 1 — Smalltalk Run-time Error Notifier.



Inheritance

How does an object know about messages like printString and doesNotUnderstand:? Through inheritance, a key element of any object-oriented language.

The Smalltalk interpreter directly supports hierarchical inheritance.

Like everything else in Smalltalk, classes are objects. You create a new class by sending a class creation message to some other class. Your class becomes a subclass of that other class, inheriting all its behavior.

At the root of the class hierarchy is the class Object, which understands the messages common to all objects, such as printString and doesNotUnderstand:.

And Where Do Objects Come From?

Most real-world objects have fairly long lifetimes. With the exception of radioactive Polonium-214 (which has a half-life of 164 microseconds), or chocolate ice cream (which in our house, has a full life about as long), real objects tend to stick around. Not so with Smalltalk objects.

They may be created hundreds or thousands of times each second. When they're no longer needed, they become garbage. Fortunately, a garbage collector knows when garbage is garbage and reclaims the memory.

In a world of routine object creation and destruction, garbage collection is sorely needed.

So you (the programmer) create objects deliberately. And the Smalltalk system creates objects behind your back, as it answers messages.

While the usual activity of Smalltalk programming is defining and refining the behavior of a class of objects, the usual activity of executing Smalltalk code is creating objects and sending them messages. It's not unusual for a large Smalltalk application to generate hundreds of thousands of objects.

For example, you might do a "print it" on the statement "newString := 'Hello,', 'world!'". (See Figure 2.)

This sends the String 'Hello,' the concatenation message "," with the String argument 'world!'. It replies (in the best of Smalltalk worlds) with the brand new String object 'Hello, world!', which is stored in the temporary variable newString.

Then, the workspace window closes, and the new String object is collected as garbage.

The One-Line Database

Let's create some more useful objects. (See Figure 3.)

Execute

Figure 2 — Create a New Object by Sending a New Message

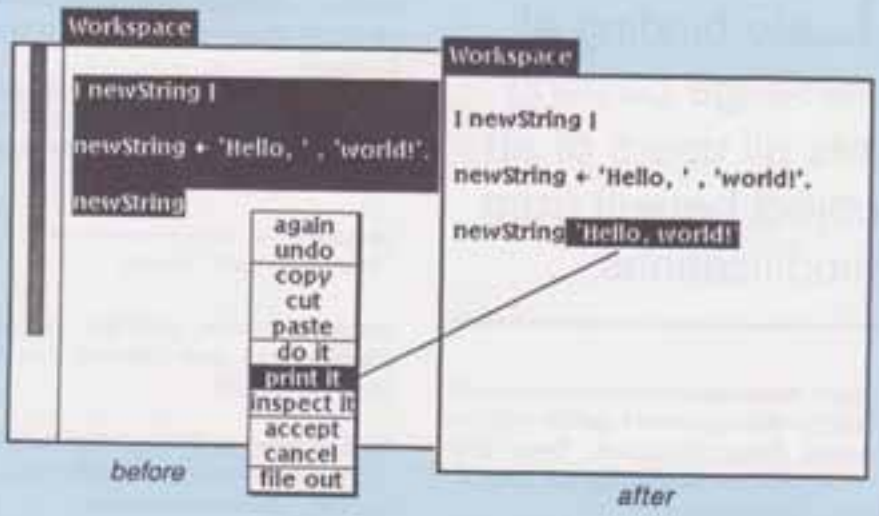
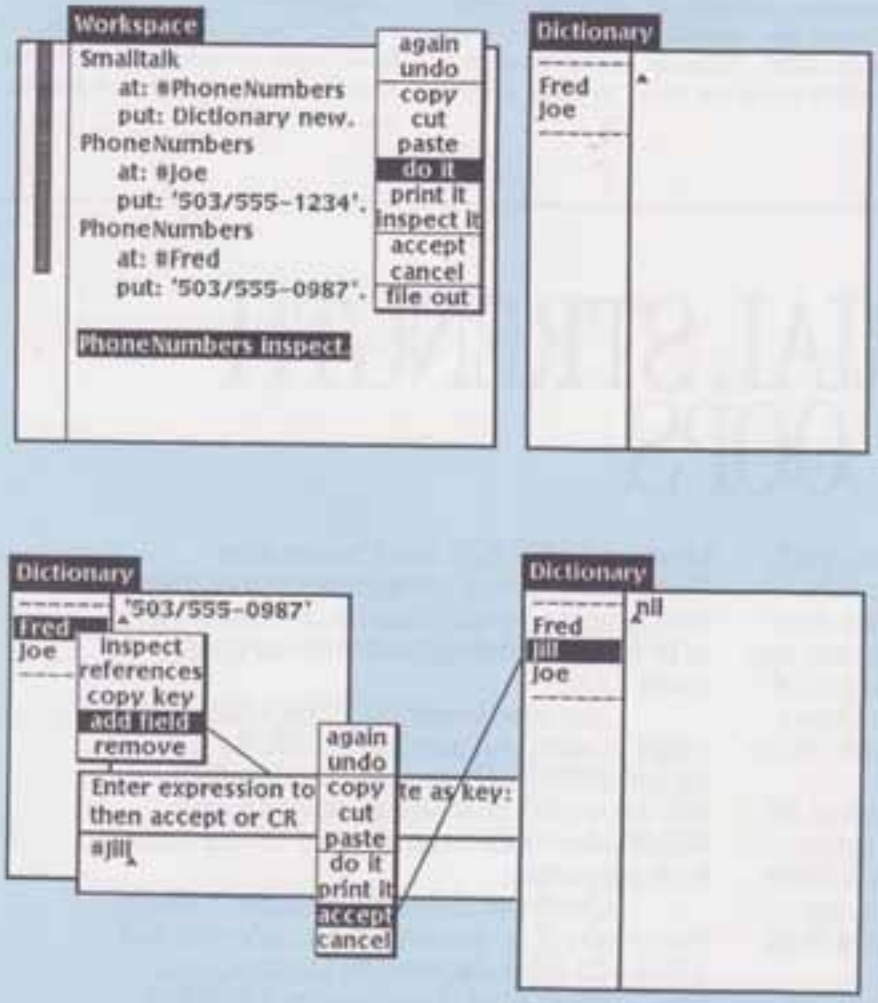


Figure 3 — The One-line Database in Action.



```
Smalltalk
  at: #PhoneNumbers
  put: Dictionary new.
```

by highlighting the text and choosing "do it" from the menu.

This is Smalltalk for — create a new global variable named PhoneNumbers and place a newly created Dictionary object in it.

Now you can begin adding objects to your new Dictionary.

To retrieve an object from the PhoneNumbers Dictionary, "print it" using the following statement :

```
PhoneNumbers at: #Joe.
```

What's Going On Here?

We're creating objects by sending messages to other objects. The names #Joe and #Fred are actually objects of class Symbol (similar to String), whose instances are unique in the system.

Once you've created #Joe, any mention of #Joe, is a reference to the existing Symbol, #Joe.

When you send the message at:put: to the new Dictionary object PhoneNumbers with two arguments, it creates a new Association object that stores the first Symbol argument together with the second String argument. These key-value pairs are very useful for symbolically representing data.

> Late-binding of message passing lets all users of an object benefit from modifications.

Although we're using a Symbol as the key and a String as the value in this example, any object can become associated with any other object.

All objects are equipped to answer common messages (in addition to printString and doesNotUnderstand:). One particularly valuable one is "inspect," which forces an object to open a window on its private state. This is so useful during debugging that the standard Smalltalk debugger includes an Inspector for the object whose method is being debugged.

Executing "PhoneNumbers inspect" will pop up an Inspector that allows you to add, remove, and modify entries in your database. So —

```
(Smalltalk at: #PhoneNumbers put:
  Dictionary new) inspect.
```

creates a simple database manager, complete with user interface! Not bad for a line of code!

An Environment Of Reusable Parts

The key to Smalltalk productivity is that components are reusable.

The Class Inspector is a good example. Programming often requires examining or modifying an object's private state and this is the task of Inspector. (Without modification it's also a debugger.)

"So," you might counter, "my Speedo QuickWindows Mark III Library from Ne'erdowell Software lets me do that in C."

Perhaps, but does it let you easily modify and extend the library so that already existing applications can use the changes? Probably not, unless your C is late-binding. Late-binding of message passing lets all users of an object benefit from modifications

To get an idea of how powerful this is, I'll briefly describe the abilities of the major classes in Smalltalk.

Magnitude's Classes

Numeric classes are all subclasses of Magnitude. Magnitude does very little, it's really an abstraction for all classes of objects that know how to compare themselves with each other. Abstract superclasses, like Magnitude, are common in Smalltalk.

Below Magnitude in the hierarchy are the classes that deal with quantities, like Date, Time, Character, and Number. Date and Time contain methods for printing and comparing dates and times, as well as special methods for obtaining partial information, such as day of the week.

The class methods "Date today" and "Time now" return a new object representing the date (or time) a message was sent.

Class Character contains methods for converting a character to or from ASCII, or for determining if it's an alphanumeric, vowel, or whatever. (This class has been successfully modified to support Oriental character sets, with more than 256 characters.)

Class Association, mentioned earlier, might seem out of place as a subclass of Magnitude. But associations connect or associate a symbolic name with an object. And such symbolic names need order. So two Associations need to know how to compare themselves.

Think of Associations as entries in the index of a book, where keywords serve as references to page numbers.

Numbers

Class Number is an abstract superclass for objects which know about arithmetic. You would never create an instance of Number (since it's an abstract class), but through inheritance, Number provides the arithmetic ability you would associate with a number.

Number has subclasses that do most of the work. Float provides floating-point capability, and Fraction represents a ratio between two Numbers. Fractions may seem quaintly archaic. They are, until you have to deal with non-integer quantities that don't lend themselves to floating point.

For example, using Fractions, you can add one penny to the sum of the National Debt. Attempting to do the same using floating point math would result in no change, which is convenient to politicians, but hardly accurate. In particular, Fraction provides a good superclass for creating a fixed-point class.

The Number subclass Integer is another abstract class, which serves as a superclass for LargePositiveInteger, LargeNegativeInteger, and SmallInteger.

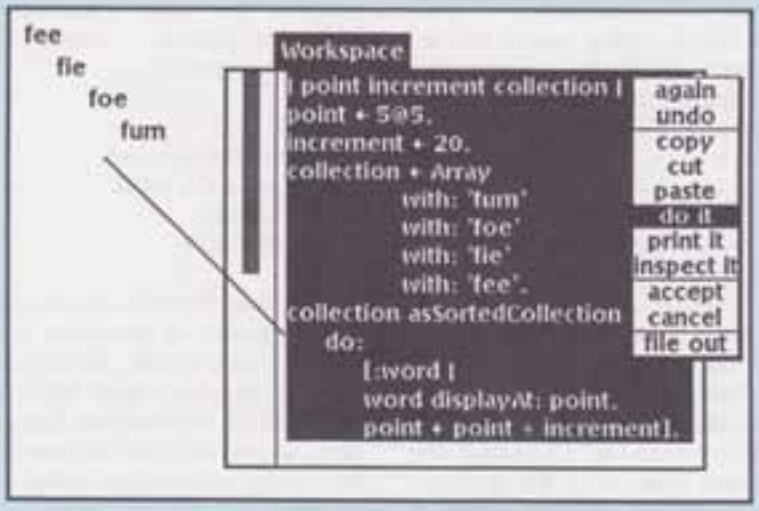
For most purposes, it's safe to treat all these the same (with the understanding that larger integers take more memory and execution time).

Most modern systems allow SmallIntegers between -2^{30} and 2^{30-1}, although some older versions of Smalltalk limit SmallIntegers to the range of -2^{14} to 2^{14-1}

Numeric Conversions

One of the nicer things about

Figure 4 — A Few Features of Collections at Work.



takes on the value of the next object in "collection." So, each object in "collection" is displayed on the screen at the location "point," and "point" is incremented by "increment."

Miscellaneous Classes

Before we move up to the classes that do windows and graphics, we need to mention a few housekeeping classes. Stream and its subclasses are invaluable for sequential access to a Collection.

Streams

While all Collections understand enumeration behavior, sometimes it's useful to be able to interrupt, skip forward or backward, or terminate the enumeration of a Collection.

Streams maintain a position in their Collection and are useful for modeling sequential things like files or terminals. Stream, itself, is an abstract class and only provides instance creation. It tests to see if you're at the end of the Collection, does basic enumeration, and provides accessing methods for its subclasses.

ReadStream gives you access to a Collection. The "next" message, for instance, returns the next object in the Collection, and the "next:" message takes an Integer argument and returns the specified number of sequential objects in the Collection.

WriteStream adds write access using the selectors nextPut: and nextPutAll:. ReadWriteStream and FileStream understand both reading and writing messages, one on a Collection, and the other on a file.

Boolean is an abstract superclass, with the subclasses True and False. These objects understand boolean logic. They are the objects that conditional branching and looping messages are sent to.

UndefinedObject has a single instance, known as nil. nil is used to initialize objects. If you get an error notifier that mentions UndefinedObject, chances are you forgot to initialize a variable. And so a message was sent to nil.

Blocks

BlockContext (or Context on some systems) is simply a way of moving code around the system. Conditional branching and looping, enumeration, error recovery, and many other functions are created from Blocks. For example, SortedCollections contains a Block of code that compares two component objects in the SortedCollection to determine their order.

Blocks are very useful, but they can also waste a lot of memory and execution time. Blocks can take arguments, which change their state (e.g., during enumeration). The number of arguments specified when the block is created and number sent during execution must be the same.

Graphics

A whole group of classes represent graphic objects. Point, mentioned earlier, represents a position in an area. Rectangle contains two Points marking opposite corners. Form contains a bitmap, making up the graphic image.

BitBlt manipulates Forms, and is how graphics get displayed. Don't use BitBlt as a model for your own classes — it's better used as an example of how Smalltalk code can be non-object-oriented!

User Interface Classes

The most important user interface class is ParagraphEditor on Smalltalk-80 systems and TextEditor on Smalltalk/V systems. It handles text editing and entry for the whole system.

SelectionInListController (ListSelector on Smalltalk/V) provides a simple way to choose an item out of a list. Unlike pop-up or pull-down menus, these lists are scrollable. You'll see these throughout the system as "subviews," or windows within other windows.

The Inspector class (mentioned earlier) is one of the most versatile performers in the user interface group. It allows you to examine and modify the private state of an object. It combines a SelectionInListView, to allow selecting a variable from a list of all variables of the object, with a ParagraphEditor, used to view or modify the selected variable's contents. Its subclass DictionaryInspector allows selection of the inspected object by key.

FileList lets you examine files and directories, and typically includes a list of files or directories and a window for editing a selected file.

The important aspect of having the source code to all the classes that make up the environment is that you can customize it, even the user interface, to suit yourself.

Don't like the way a menu is set up or want more items on it? Change it! Want to change the way the window panes look? Do it! Want to make the interface to the file system work differently? Go ahead!

Decision Points For Potential Users

Primarily, Smalltalk is a language and environment for rapid program development. In other words, it's ideal for hackers. Some studies show that Smalltalk programmers are about five times more productive than Pascal or C programmers, and up to 25 times more productive than folks working in assembly.

On the flip side, Smalltalk applications aren't easy to separate from the environment, and this makes delivering applications difficult. The Smalltalk environment is large, and has a (partially deserved) reputation for being slow.

Smalltalk is also attacked for being wordy, but usually by those who think you need to write a lot of code. Code reusability, combined with copy/cut/paste editing, means that an experienced Smalltalk programmer generally spends less time typing than a C programmer.

"Make it work, then make it fast," is something we all need to repeat to ourselves from time to time. Smalltalk makes it work like no other, and when it comes time to make it fast, *there is help*. All popular versions of Smalltalk include some means for access to assembly code, and Tektronix and Digitalk Smalltalk include full access to all system calls.

If you decide to buy a Smalltalk implementation and work for someone with deep pockets, consider getting a Smalltalk-80 variant. ParcPlace Smalltalk-80 is available on many platforms, but at a kilobuck a pop, it's probably not a personal purchase.

If your employer has "really deep" pockets, buy a Tektronix bitmap workstation. Smalltalk's included. And the Tektronix hardware and Unix port have been designed for Smalltalk, so it's better integrated.

If you're running on a PC clone or a Mac II, then Smalltalk-V is your best bet. It's inexpensive, well-supported, and the Mac II and 80286 versions are very fast. The environment, while differing considerably from Smalltalk-80, shares a large group of Smalltalk-80 classes. The version that runs on vanilla 8088 PCs is cheap, but lacks the speed and object space necessary for large applications.

Vendors

Digitalk, Inc., has sold more copies of its version of Smalltalk, called Smalltalk/V, than any other Smalltalk on the market. It runs on a number of platforms, and has been pared down especially for PCs. The version we bought came with 99 classes. It has very clearly written, nicely formatted documentation — some people would pay for the manual alone what we paid for the package, since you can't buy the manual without the software.

Smalltalk/V requires an IBM-PC, PC/XT, PC/AT, or compatible and PC-DOS or MS-DOS — costs $99.95. Smalltalk/V286 operates with the DOS operating system on AT class computers and requires a minimum of 1-1.5 MB of memory — costs $199.95. Digitalk's Macintosh Smalltalk/V runs on the Mac II, the Mac SE, and the Mac Plus; it will be available in the Fall — costs $199.95. There are applications kits available for color, communications, and other goodies — they cost $49.95 each. For more info —

Digitalk, Inc.
9841 Airport Boulevard
Los Angeles, CA 90045
1-800-922-8255
1-213-645-1082

ParcPlace Systems sells Smalltalk-80 for a number of platforms manufactured by Sun, Apollo, Hewlett-Packard and others, plus Apple Macintosh II, Plus, and SE workstations. See publications section for their address to write for specific information. Prices are over $1,000.

Tektronix was one of four companies that contributed to the development of Smalltalk-80 in the late 70s and early 80s. Apple, Digital Equipment Corporation, Hewlett-Packard and Tektronix were test sites for early versions of the system Xerox was developing — they each were expected to develop a full implementation of a written specification of the interpreter.

Tektronix has been developing its own implementation of the system since then, extending it from a version that Xerox PARC released in 1983. They have a very extensive class library, collection of tools, and reusable components. Tektronix Smalltalk is only available on Tektronix workstations.

A Little Smalltalk is a version of Smalltalk that runs under UNIX on line-oriented (conventional) terminals. The interpreter is written in C. It's available for distribution cost on 9-track tape in "tar" format. This Smalltalk doesn't support graphics (no windows, no mouse), but it's an inexpensive way to learn object-oriented programming. For more info —

Smalltalk Distribution
Department of Computer Science
Oregon State University
Corvallis, OR 97331

Periodicals

BYTE magazine, August 1981 issue. Entire issue devoted to Smalltalk-80. A collection of articles that provide an interesting and authoritative introduction to Smalltalk.

HOOPLA! (Hooray for Object-Oriented Programming Languages!) is published

quarterly by Object-Oriented Programming for Smalltalk Application Developers Association (OOPSTAD). Annual individual membership in the U.S. is $25.

HOOPLA!
P.O. Box 1565
Everett, WA 98206-1565, USA
electronic bulletin board, 300/1200 baud: 1-206-252-9048

Journal of Object-Oriented Programming (JOOP) is published bimonthly. U.S. annual individual subscription is $49.

JOOP
P.O. Box 968
Fort Washington, PA 19034
1-800-345-8112

SCOOP is a newsletter published by Digitalk, Inc., for their Smalltalk/V customers who've sent in their signed warranty card. It contains new product and upgrade information, bug fixes, programming tips, and other news.

The *Smalltalk-80 Newsletter* is published by ParcPlace Systems (PPS), Xerox Corporation. It focuses on new PPS products and services and applications written in this vendor's Smalltalk-80.

ParcPlace Systems
2400 Geng Road
Palo Alto, CA 94303
1-415-859-1000 (in CA)
1-800-822-7880

theActiveView is a quarterly newsletter available free of charge to Tektronix Smalltalk customers. It focuses on features of Tektronix' implementation of Smalltalk-80, programming tips column applicable to Smalltalks from all vendors, updates on documentation and training classes offered by Tektronix. Send a postcard with your name, company name, address and phone number to:

Software Productivity Technologies
Tektronix, Inc.
P.O. Box 500, M.S. 50-470
Beaverton, OR 97077
ATTN: theActiveView -mc

Books

Budd, Timothy. *A Little Smalltalk.* Reading, MA: Addison-Wesley, 1987.

Goldberg, Adele and Robson, David. *Smalltalk-80: The Language and Its Implementation.* Reading, MA: Addison-Wesley, 1983.

Goldberg, Adele. *Smalltalk-80: The Interactive Programming Environment.* Reading, MA: Addison-Wesley, 1984.

Pinson, Lewis J. and Wiener, Richard S. *An Introduction to Object-Oriented Programming and Smalltalk.* Reading, MA: Addison-Wesley, 1988.

♦ ♦ ♦