

Building Control Structures in the Smalltalk-80 System

L Peter Deutsch
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

Just as *data structures* refer to the ways that we group data together by using simple *objects* to represent more complex objects, *control structures* refer to the ways a programmer can build up complex sequences of *operations* from simpler ones. The easiest example of a control structure is sequencing: do something and then do something else. Two other familiar examples are the *conditional* structure (if some condition is true, do something, otherwise do something else) and the *loop* (do something as long as some condition remains true).

Most languages provide a few common control structures, typically sequencing, conditional, looping, and procedures, but no way for a programmer to define new structures. One useful control structure that many languages omit is the simple *case* statement (given *N* alternative things to do, numbered from 1 to *N*, and a variable *K*, do the *K*th thing). If the language doesn't provide a case statement, you can always simulate it with a long string of conditionals, but it makes your program harder to read. Other useful control structures are much more difficult to simulate if the language fails to provide them.

The Smalltalk-80 language and system (which will be called simply "Smalltalk") is one of the few languages in which a programmer can invent and implement, with relative ease, new control structures that aren't provided by the system implementors. The rest of the article illustrates this point with examples that have actually been run on a Smalltalk-80 implementation.

What's Built In

Smalltalk provides very few built-in control structures. There is the conditional structure, implemented as follows:

```
someCondition ifTrue: [somethingToDo]
someCondition ifFalse: [somethingToDo]
someCondition
  ifTrue: [somethingToDo]
  ifFalse: [somethingElseToDo]
```

and the simple loop:

```
[someCondition] whileTrue: [somethingToDo]
[someCondition] whileFalse: [somethingToDo]
```

The most powerful tool for building new structures is the *block*. Two examples are:

```
[somethingToDoLater]
```

and:

```
[:anArgumentName| somethingToDoLater]
```

The block allows a caller to pass to the implementor of a control structure a piece of code to be executed (possibly with arguments, as in the second example) at an appropriate time.

Case Statement

Our first example is the case statement described before. We would like a construct that includes an indexed collection of blocks for the expected cases and another block for the situation where the index is out of range. Without any particular trouble we can have a construct like this:

```
someExpression
  case: (Array with: [case1] with: [case2] with:
            [case3])
  otherwise: [somethingElse]
```

where [somethingElse] gets evaluated if someExpression isn't 1, 2, or 3. Then the definition is simple. We add a message to the existing class Number. In order to distinguish adding methods to existing classes from creating new classes, we will label templates "existing" if they are to be seen as partial templates adding new methods to existing classes.

Table 1 shows the code necessary to add the case method to the class Number. As far as the control struc-

FOR ONLY \$129.95 Learn Computing From The Ground Up

Build a Computer kit that grows with you, and can expand to 64k RAM, Microsoft BASIC, Text Editor/Assembler, Word Processor, Floppy Disks and more.

EXPLORER/85

Here's the low cost way to learn the fundamentals of computing... the important skills you'll need to learn and move you advances in computer skills. For just \$129.95 you get the advanced design Explorer/85 motherboard with all the features you need to learn how to write and use programs. And it can grow into a system that is a match for any personal computer on the market. Look at these features: 8085 Central Processing Unit, the microprocessor "heart" of the Explorer/85 [join the millions who buy and use the 8080/8085 this year alone]! Four 8-bit plus one 6-bit input/output ports from which you can input and output your programs, as well as control exterior switches, relays, lights, etc. A cassette interface so you store and load programs you've learned to write. A large 2.000 byte operating system/monitor makes it easy to learn computing in several important ways: • It allows simpler, faster writing and entering of programs • It permits access by you to all parts of the system so you can check on the status of any point in the program • It allows tracing each program step by step, with provision for displaying all the contents of the CPU (registers, flags, etc.) • ... and it does much more!

You get all this in the starting level (Level A) of the Explorer/85 for only \$129.95. Incredible! To use, just plug in our 8VDC power supply and terminal and you're off-and-ready — if you don't have them, see our special offers below.

- Level A computer kit (Terminal Version) ... \$129.95 plus \$3 P&I.*
- Level A kit (Hex Keypad/Display Version) ... \$129.95 plus \$3 P&I.*

LEVEL B — This "building block" converts the mother-board into a two-slot \$100 bus (industry standard) computer. Now you can plug in any of the hundreds of \$100 cards available.

- Level B kit ... \$49.95 plus \$2 P&I.*

\$100 bus connectors (two required) ... \$4.95 each, postpaid.

LEVEL C — Add still more computing power; this "building block" mounts directly on the mother-board and expands the \$100 slots.

- Level C kit ... \$39.95 plus \$2 P&I.*
- \$100 bus connectors (five required) ... \$4.95 each, postpaid.

LEVEL D — When you reach the point in learning that requires more memory, we offer two choices: either add 4k of a memory directly on the mother-board, or add 16k to 64k of memory by means of a single \$100 card, our famous "JAWS."

- Level D kit: (CHECK ONE) ... 4k on-board ... \$49.95 plus \$2 P&I.* 16k \$100 "JAWS" ... \$149.95 plus \$2 P&I.* 32k \$100 "JAWS" ... \$199.95 plus \$2 P&I.* 48k \$100 "JAWS" ... \$249.95 plus \$2 P&I.* 64k \$100 "JAWS" ... \$329.95 plus \$2 P&I.*

LEVEL E — An important "building block;" it activates the 8k ROM/EPROM space of the mother-board. Now just plug in our 8k Microsoft BASIC or your own custom program.

- Level E kit ... \$5.95 plus 50¢ P&I.*

Microsoft BASIC — It's the language that allows you to talk English to your computer! It is available three ways:

- 8k cassette version of Microsoft BASIC (requires Level B and 12k of RAM minimum; we suggest a 16k \$100 "JAWS" — see above) ... \$64.95 postpaid.

8k ROM version of Microsoft BASIC (requires Level B & Level E and 4k RAM; just plug into your Level E sockets. We suggest either the 4k Level D RAM expansion or a 16k \$100 "JAWS") ... \$99.95 plus \$2 P&I.*

Disk version of Microsoft BASIC (requires Level B, 32k of RAM, floppy disk controller, 8" floppy disk drive) ... \$225 postpaid.

TEXT EDITOR/ASSEMBLER — The editor/assembler is a software tool (a program) designed to simplify the task of writing programs. As your programs become longer and more complex, the assembler can save you many hours of programming time. This software includes an editor, which edits the programs you write, makes changes, and saves the programs to cassette tapes. The assembler performs the clerical task of translating symbolic code into the computer-readable object code. The editor/assembler program is available either in cassette or a ROM version.

- Editor/Assembler (Cassette version; requires Level "B" and 8k (min.) of RAM — we suggest 16k "JAWS" — see above) ... \$59.95 plus \$2 P&I.*
- Editor/Assembler (ROM version, supplied on a \$100 card; requires Level B and 4k RAM (min.) — we suggest either Level D or 16k "JAWS") ... \$99.95 plus \$2 P&I.*

8" FLOPPY DISK — A remarkable "building block." Add our 8" floppy disk when you need faster operation, more convenient program storage, perhaps a business application, and access to literally thousands of programs and program languages available today. You simply plug them into your Explorer/85 disk system — it accepts all IBM-formatted CP/M programs.

- 8" Floppy Disk Drive ... \$499.95 plus \$12 P&I.*
- Floppy Controller Card ... \$159.95 plus \$2 P&I.*
- Disk Drive Cabinet & Power Supply ... \$69.95 plus \$3 P&I.*

Drive Cables (set up for two drives) ... \$25.00 plus \$1.50 P&I.*

CP/M 2.2 Disk Operating System: includes Text Editor/Assembler, dynamic editor, and other features that give your Explorer/85 access to thousands of existing CP/M-based programs ... \$150.00 postpaid.

NEED A POWER SUPPLY? Consider our AP-1. It can supply all the power you need for a fully expanded Explorer/85 (note: disk drives have their own power supply). Plus the AP-1 fits neatly into the attractive Explorer steel cabinet (see below).

- AP-1 Power Supply kit (8V @ 5 amps) in deluxe steel cabinet ... \$39.95 plus \$2 P&I.*

NEED A TERMINAL? We offer you choices: the least expensive one is our Hex Keypad/Display kit that displays the information on a calculator-type screen. The other choice is our ASCII Keyboard/Computer Terminal kit, that can be used with either



NETRONICS Research & Development Ltd.
333 Litchfield Road, New Milford, CT 06776



1. Plug in Netronics' Hex Keypad/Display
2. Add Level B to convert to S100
3. Add 4k RAM
4. Plug in Level E here; accepts Microsoft BASIC or Editor/Assembler in ROM
5. Add two S100 boards
6. Add your own custom circuits (prototyping area)
7. Connect terminal

a CRT monitor or a TV set (if you have an RF modulator)

- Hex Keypad/Display kit ... \$69.95 plus \$2 P&I.*

ASCII Keyboard/Computer Terminal kit featuring a full 128 character set, u/l case, full cursor control, 75m video output, convertible to baud rate, selectable baud rate, RS232-C or 20 mA/I/O, 32 or 64 characters by 16 line formats ... \$149.95 plus \$3 P&I.*

- Steel Cabinet for ASCII Keyboard/Terminal ... \$19.95 plus \$2.50 P&I.*
- RF Modulator kit (allows you to use your TV set as a monitor) ... \$59.95 postpaid.
- 12" Video Monitor (10MHz bandwidth) ... \$139.95 plus \$5 P&I.*
- Deluxe Steel Cabinet for the Explorer/85 ... \$49.95 plus \$3 P&I.*
- Fan for cabinet ... \$15.00 plus \$1.50 P&I.*

ORDER A SPECIAL-PRICE EXPLORER/85 PAK—THERE'S ONE FOR EVERY NEED.

- Beginner Pak (Save \$20.00) — You get Level A (Terminal Version) with Monitor Source Listing (\$25 value) AP-1, 5-amp. power supply, Intel 8085 User's Manual ... (Reg. \$199.95) SPECIAL \$179.95 plus \$4 P&I.*
- Experimenter Pak (Save \$53.40) — You get Level A (Hex Keypad/Display Version) with Hex Keypad/Display, Intel 8085 User's Manual, Level A Hex Monitor Source Listing, and AP-1, 5-amp. power supply ... (Reg. \$279.95) SPECIAL \$219.95 plus \$6 P&I.*
- Special Microsoft BASIC Pak (Save \$103.00) — You get Level A (Terminal Version), B, D (4k RAM), E, 8k Micros in ROM, Intel 8085 User's Manual, Level A Monitor Source Listing, and AP-1, 5-amp. power supply ... (Reg. \$439.70) SPECIAL \$329.95 plus \$7 P&I.*

ADD A ROM-VERSION TEXT EDITOR/ASSEMBLER (Requires Levels B and D or \$100 Memory) ... \$99.95 plus \$2 P&I.*

- Starter 8" Disk System — Includes Level A, B floppy disk controller, one CDD 8" disk-drive, two drive cables, two S100 connectors; just add your own power supplies, etc. ... \$199.95 plus \$13 P&I.* 32k Starter System ... \$104.95 plus \$13 P&I.* 48k Starter System ... \$145.95 plus \$13 P&I.*
- Add to any of above Explorer steel cabinet, AP-1 five amp. power supply, Level C with two S100 connectors, disk drive cabinet and power supply, two sub-D connectors for connecting your printer and terminal ... (Reg. \$225.95) SPECIAL \$199.95 plus \$13 P&I.*
- Complete 84k System: Wired & Tested ... \$1650.00 plus \$28 P&I.*
- Special Complete Business Software Pak (Save \$25.00) — Includes CP/M 2.2 Microsoft BASIC, General Ledger, Accounts Receivable, Accounts Payable, Payroll Package ... (Reg. \$1325) SPECIAL \$999.95 postpaid.

*P&I stands for "postage & insurance." For Canadian orders, double this amount.

Continental Credit Card Buyers Outside Connecticut:

TO ORDER Call Toll Free: 800-243-7428

To Order From Connecticut, or For Technical Assistance, call (203) 354-9375

★ (Clip and mail entire ad) ★

SEND ME THE ITEMS CHECKED ABOVE

Total Enclosed (Conn. Residents add sales tax): \$ _____
Paid by:

- Personal Check Cashier's Check/Money Order

- VISA MASTER CARD (Bank No. _____)

Acct. No. _____ Exp. Date _____

Signature _____

Print Name _____

Address _____

City _____

State _____ Zip _____

ture goes, this is all there is to it: alternativeBlocks is an array of blocks, and the method in Number simply picks the appropriate one to evaluate. However, the syntax looks clumsy. We might like to have something that looks more like the following:

someExpression

case: [case1], [case2], [case3]

otherwise: [somethingElse]

One way to do this is to arrange for appropriate interpretation of the message , (comma) by BlockContext (table 2a) and to invent a new subclass of IndexedCollection (table 2b) that will also interpret , appropriately. We also have to add some protocol to BlockContext to handle the situation of a single block. Note that double quote marks delineate comments in Smalltalk.

As a matter of style, we generally discourage syntactic embellishments of this kind: their implementation tends to be obscure and they don't add that much to the ease of writing programs.

Generator Loops

Many languages provide a kind of loop called a generator, which sets a variable to successive values generated by some algorithm each time through the interior of the loop. The familiar kind of loop that runs through successive integers from 1 to N is one such example. Another example is looking at successive elements of a linked list, or any ordered collection.

Smalltalk actually provides simple generators of the form:

someCollection do:

[:anElement| doSomethingWithTheElement]

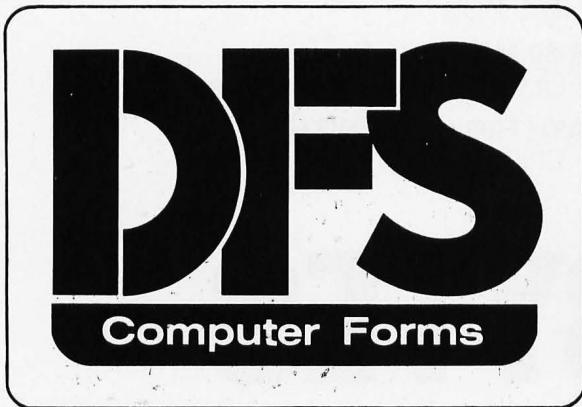
but it is instructive to see how we could have constructed them ourselves. This could be accomplished by having each kind of collection object implement the message do:

class name (existing)	Number
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
	"none added here"
instance messages and methods	
control	
case: alternativeBlocks otherwise: aBlock	
(self >= 1 and: [self <= alternativeBlocks size])	
ifTrue: [↑(alternativeBlocks at: self) value]	
ifFalse: [↑aBlock value]	

Table 1: Template showing additions to existing class Number.

Small Business Systems User!

**WHEN BUYING CHECKS,
STATEMENTS AND
INVOICES — LOOK FOR
THIS MARK:**



**ON THE DOOR OF YOUR
COMPUTER STORE**

DFS Computer Forms are

- Sold by a Local Business
- Satisfaction Guaranteed
- Available in Small Quantities
- Compatible with Existing Software
- Very Economical



P.O. Box 643 • Townsend, MA. 01469

directly; we would get simple arithmetic loops by using do: with an Interval, a kind of collection that represents a bounded arithmetic progression. Using do: is convenient when we know that we want to look at all the elements of the collection, and do the same thing to each one. For example IndexedCollection might implement do: as shown in table 3.

However, if we want to retain more flexibility in controlling the generation process, there is a better way. We define the notion of a *supplier*, which will deliver the elements of a collection one at a time in response to messages. The protocol (set of messages and their intended meanings) for suppliers consists of the messages:

s atEnd

class name (existing)	BlockContext
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
	"none added here"
instance messages and methods	
<i>constructing</i>	
, aBlock	↑ BlockCollection with: self with: aBlock
<i>accessing</i>	
size	"Behave like a BlockCollection with self as the only element" ↑ 1
at: Index	"Behave like a BlockCollection with self as the only element" index = 1 if True: [↑ self]. self error: 'Subscript out of bounds'
class name (existing)	BlockCollection
superclass	IndexedCollection
instance variable names	"none defined here"
class messages and methods	
	"none defined here"
instance messages and methods	
<i>constructing</i>	
, aBlock	↑ self add: aBlock

Table 2: Templates showing additions to existing class BlockContext (2a) and the creation of a class template for class BlockCollection (2b).

SORCERER SOFTWARE

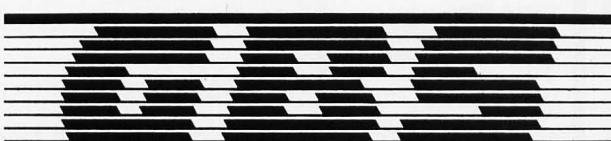
from QUALITY SOFTWARE

All Programs Are On Cassette

VISI-WORD by Lee Anders

From preparing short letters to writing a book, word processing becomes easy and inexpensive using VISI-WORD, a cassette based word processor. VISI-WORD is designed to interface with just about any printer you can attach your Sorcerer to. VISI-WORD can accept control characters, which allows you to issue special commands to those printers with graphics controls, font control, and the like. A special feature of VISI-WORD, from which it gets its name, is the "command display off" feature. This command eliminates all special end of line markers and other non-printing characters and automatically performs right-justification, centering, and indenting right on the video, so that you can see what your text will look like before it is printed. Other features of VISI-WORD include four separate buffers (to assist with form letters, boiler plating, and shifting text around), automatic page numbering and titling, partial print, and locating strings.

\$59.95



General Business System

by Lee Anders

GBS is a general purpose programming system that can be used for many business applications. Use this system to create, edit, format, and print mailing lists. Or set up an inventory system, an accounts receivable file, or a payroll system. Or use it to enter orders. Delete, modify or append records, and then summarize and tabulate the results. You design (with the help of an extensively documented user manual) a system of records. Then use the power of GBS to compute, sort, select, merge, add, and modify your data. GBS will provide you with the kind of fast, accurate, flexible tools you always knew a home computer could provide. Four example application programs are included. Of course, you don't need to use GBS for business. You can use it for personal finance, club or personal record keeping, or almost any type of problem that involves the management of records. Written in machine language with flexible cassette interfacing, this program requires a Sorcerer with at least 32K of memory.

\$99.95

FORTH for the Sorcerer. Now Sorcerer owners can enjoy the convenience and speed of the fascinating FORTH programming language. Based on fig-FORTH and adapted for the Sorcerer by James Albanese, this version uses simulated disk memory in RAM and does not require a disk drive. Added to standard fig-FORTH are an on-screen editor, a serial RS-232 driver, and a tape save and load capability. Numerous examples are included in the 130 pages of documentation. Requires 32K or more of RAM.

\$59.95

ARROWS AND ALLEYS™ by Vic Tolomei

The latest of Quality Software's great arcade games for the Sorcerer is ARROWS AND ALLEYS, by Vic Tolomei. You drive your car in a maze of alleys. Your task is to eliminate a gang of arrows that constantly pursues you. You have a gun and the arrows don't, but the arrows are smart and they try to stay out of your sights and will often attack from the side or from behind. Eliminate the arrows and another, faster gang comes after you. Four levels of play. Requires 16K or more of RAM.

\$17.95

We have more than 20 programs for the Sorcerer
PLEASE WRITE FOR OUR CATALOG



QUALITY SOFTWARE

6660 Reseda Blvd., Suite 105, Reseda, CA 91335
Telephone 24 hours, seven days a week: (213) 344-6599

HOW TO ORDER: If there is no SORCERER dealer near you, you may order directly from us. MasterCard and Visa cardholders may place orders by telephone. Or mail your order to the address above. California residents add 6% sales tax. **Shipping Charges:** Within North America orders must include \$1.50 for shipping and handling. Outside North America the charge for airmail shipping and handling is \$5.00. Pay in U.S. currency.

*The name "SORCERER" has been trademarked by Exidy, Inc.

which returns true if there are no more elements to be supplied, and:

s next

which returns the next element. Now we can build our do: operation as shown in table 4. Then each kind of collection needs to implement the message asSupplier, which returns an appropriate supplier. Table 5 shows what a supplier might look like for IndexedCollection (including its creation). If an attempt is made to read past the end, position will be incremented beyond the size of the collection, and next will provoke an error when the at: tries to

class name (existing)	IndexedCollection
superclass	Collection
instance variable names	"none added here"
class messages and methods	
	"none added here"
instance messages and methods	
enumeration	
do: aBlock	
	index limit
	index ← 1.
	limit ← self size.
	[index <= limit] whileTrue:
	aBlock value: (self at: index).
	index ← index + 1]

Table 3: Template showing additions to existing class IndexedCollection.

class name (existing)	Collection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
	"none added here"
instance messages and methods	
enumeration	
do: aBlock	
	supplier
	supplier ← self asSupplier.
	[supplier atEnd] whileFalse:
	aBlock value: supplier next]

Table 4: Template showing additions to existing class Collection.

access an element beyond the size. An alternative approach, which gives a more useful error message at the expense of duplicating a check that `at` must perform anyway, is to define `next` as follows:

```
next | |
  position >= collection size
  ifTrue:
    [self error: 'Attempt to read beyond last
      element'].
  position ← position + 1.
  ↑ collection at: position
```

Similar supplier classes would be needed to provide generation capability for all of the different kinds of Collections.

With the supplier approach to generators, we can easily build a loop that sequences through two collections in parallel (see table 6). This would be very difficult if we did not have suppliers, but made collections implement `do:` directly. The problem is that while we could use `do:` to get one of the two collections to deliver its elements to a block of our choosing, there would be no way to get the other collection to deliver exactly one element each time the block is invoked.

Suppliers are so useful as a concept and as a protocol that Smalltalk actually includes them, under the name of

`ReadStream`. The important point is that we could have built them ourselves if the system implementors hadn't gotten there first.

Although the only kind of supplier we have constructed is one that sequences through a collection, other kinds of suppliers are possible: they just have to respond appropriately to `atEnd` and `next`. For example, one could imagine a supplier that selected elements at random from a collection in response to `next`.

Exceptional Conditions

One of the difficulties in designing programs that (at least appear to) work reliably is designing the control structures for handling "infrequent" events. An infre-

class name (existing)	IndexedCollection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	"none added here"
instance messages and methods	<i>enumeration</i> asSupplier ↑ IndexedCollectionSupplier of: self
class name	IndexedCollectionSupplier
superclass	Object
instance variable names	collection position
class messages and methods	<i>creation</i> of: aCollection ↑ self new of: aCollection
instance messages and methods	<i>creation</i> of: aCollection collection ← aCollection. position ← 0
accessing	atEnd ↑ position >= collection size
next	next position ← position + 1. ↑ collection at: position

Table 5: Templates showing additions to existing class `IndexedCollection` (5a) and the creation of a class template for class `IndexedCollectionSupplier` (5b).

JOIN A WINNING TEAM!

Datasoft is looking for new authors who are writing professional, innovative and high quality programs for the Atari, Apple and Radio Shack computers. Your program can join our lineup of successful selling programs such as:

Micro-Painter * SIGMON * Mychess App-L-ISP * Text Wizard * Le Stick

Datasoft is a professional software publishing organization with a full-time marketing and program support staff. You as an author receive royalties for your work. And you will be supported by an established dealer distributor network around the world.

So if you have Business, Educational, Utility or Entertainment software, call or write us for more information.

Datasoft Inc.TM

COMPUTER SOFTWARE
19519 Business Center Drive
Northridge, CA 91324
(213) 701-5161

quent event is any event which is (a) qualitatively different from what happens most of the time and (b) not so common that one wants to test for it in the normal flow of control. One example of an infrequent event is an ad-

class name (existing)	Collection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
	"none added here"
instance messages and methods	
<p><i>enumerating</i> with: anotherCollection do: aBlock</p> <pre> mySupplier itsSupplier mySupplier ← self asSupplier. itsSupplier ← anotherCollection asSupplier. [mySupplier atEnd or: [itsSupplier atEnd]] whileFalse: [aBlock value: mySupplier next value: itsSupplier next]</pre>	

Table 6: Template showing additions to existing class Collection.

class name (existing)	OrderedCollection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
	"none added here"
instance messages and methods	
<p><i>searching</i> maxBefore1000</p> <pre> supplier max value theLoop max ← 0. supplier ← self asSupplier. theLoop ← [[supplier atEnd] whileFalse: [value ← supplier next. value > 1000 ifTrue: [theLoop exit]. max ← max max: value]] withExit. theLoop value. "Actually do the loop block" ↑max</pre>	

Table 7: Template showing additions to existing class OrderedCollection.

ditional exit from a loop. Suppose we would like to write a searching loop that finds the maximum element of a collection of non-negative numbers but stops searching if it finds an element greater than 1000. Such a loop might be implemented as shown in table 7.

We want the block `[[supplier atEnd] ...]` to respond to the `withExit` message by giving back a blocklike object which we can assign to the variable `theLoop`. The ability to name this object allows us to exit from it midcourse. These `BlockWithExit` objects (see table 8) need to remember only two pieces of information: the original block, to execute in response to the `value` message, and where to send control if an `exit` message is sent.

The original statement `theLoop ← ...` doesn't actually

class name	BlockWithExit
superclass	Object
instance variable names	block exitBlock
class messages and methods	
creation	
wth: aBlock	
↑ self new with: aBlock	
instance messages and methods	
creation	
wth: aBlock	
block ← aBlock	
control	
value	
exitBlock ← [↑ nil]. "Exit to my caller if the block ever sends me the exit message"	
↑ block value "Actually do the computation"	
exit	
exitBlock value "Exit from the computation to the caller who sent the value message to me in the method just above"	
class name (existing)	BlockContext
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
	"none added here"
instance messages and methods	
control	
wthExit	
↑ BlockWithExit with: self	

Table 8: Templates showing the creation of a class template for class `BlockWithExit` (8a) and additions to existing class `BlockContext` (8b).

execute the loop: it creates a block whose code is [supplier atEnd] This block becomes the block variable of a new BlockWithExit as a result of the withExit message being sent. theLoop is set to the BlockWithExit just created. When theLoop is sent the message value, the value method in BlockWithExit first creates another block, the exitBlock, which, if evaluated, will return to the sender of value *regardless of how many other activations have intervened*. The value method in BlockWithExit then sends value to the original block, causing it to execute. If no exit is sent, the loop completes normally. If an exit is sent, the exitBlock is evaluated and control returns to the last statement of maxBefore1000, just as if the loop had completed.

Dynamic Binding

Another common kind of infrequent event is a request for information. For example, suppose we want to specify a default directory for disk files throughout some part of a program. We could pass this information as an argument through all intervening calls, but this would place an added burden (in time, space, and complexity) on many parts of the program that have no interest in this information. An alternative would be to set a global variable before starting the computation, and reset it afterwards; unfortunately, if the computation is interrupted (say by something like the loop exit construct we described earlier), this leaves the variable with the wrong value. Ideally, we would like to set up a structure that

will get control if the default information is ever needed, without getting in the way of the rest of the program. Such an arrangement is called *dynamic binding*. We will illustrate how it can be used both for data and control.

Suppose we want to write something such as the following:

```
#defaultDirectory bindTo: 'Smith' in:  
[someComputation]
```

and then have the file system be able to ask for the current default directory by:

```
#defaultDirectory binding
```

Since we want the binding of defaultDirectory to 'Smith' to last only for the duration of someComputation, it follows that in order to find the binding of a dynamic variable, we must examine the data structures that Smalltalk uses to represent the state of a computation. In

class name	Binding
superclass	Association "Provides key and value variables, and messages for accessing them"
instance variable names	"none defined here"
class messages and methods	
creation	
of: aSymbol to: aValue in: aBlock ↑ self new of: aSymbol to: aValue in: aBlock	
instance messages and methods	
initialization	
of: aSymbol to: aValue in: aBlock key ← aSymbol. value ← aValue. ↑ aBlock value "Actually does the computation"	
class name (existing)	Symbol
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	
binding	
bindTo: value in: aBlock ↑ Binding of: self to: value in: aBlock	

Table 9: Templates showing creation of a class template for class Binding (9a) and additions to existing class Symbol (9b).

A LOGIC ANALYZER FOR \$395?



YES!
O W L
LA 1600-A
High Speed
16 Channels

Interfaces to dual channel scope or Apple computer.

- 10 MHZ capture rate
- Gold plated connectors and clips
- Stores 16 words of 16 bits
- Crystal controlled internal clock
- 1, 0, X compare word bit selection
- Time domain display
- Data domain display*
- Hex display*
- Internal and external trigger modes

*Options with use of Apple computer

Comes complete with interconnecting cables;
logic probe clips, diskette for Apple computer,
and operating instructions.

— Send for FREE brochure —

Osborne Wilson Labs.
508 Waterberry Drive
Pleasant Hill, California 94523
(415) 932-5489



class name (existing)	Symbol
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
	"none added here"
instance messages and methods	

binding

```

binding | context |
context ← thisContext. "Start here. thisContext is a
machine register"
[context = nil] whileFalse:
  (((context receiver isMemberOf: Binding)
    and: [context selector = #of:to:in: "Is it a
      binding..."])
   and: [context receiver key = self]]) "...of this
     variable?"
  ifTrue: "Yes, return its value"
    [↑ context receiver value]
  ifFalse: "No, go on to the next context in the
    chain"
    [context ← context sender].
  self error: ('No binding for' concatenate: self )

```

Table 10: Template showing additions to existing class Symbol.

particular, even though many messages may be sent in someComputation before the file system needs to find the binding of defaultDirectory, there must be some way to search the stack of methods that have been started but not completed, looking for whatever represents the binding of defaultDirectory. In Smalltalk, each element of this stack is a MethodContext object, and the variable in a MethodContext that refers to its caller is called its *sender*. So searching this stack just means checking the current context's sender, its sender, and so on, until we find a binding of the variable. We know we have found a binding when we recognize a MethodContext in which the receiver of the message is a Binding (see tables 9a and 9b), and which was created in response to a particular message. During this computation (↑ aBlock value in table 9a), a MethodContext will exist in which the receiver is the Binding and the message is of:to:in:. This is how we recognize a binding in the stack of MethodContexts. The searching process is shown in table 10.

Note that by combining dynamic binding with the ability to name exit points (eg: by doing #theExit bindTo: to create a BlockWithExit), we can arrange for dynamically bound exceptional events to stop a computation in midstream. More complicated arrangements that allow the parts of the computation being stopped to clean up after themselves are also easy to construct.

Coroutines

Generator loops are an example of producer/consumer

TALK'S >< INEXPENSIVE

ECHO^{T.M.} SPEECH SYNTHESIZERS

Don't limit your computer! Let it speak its mind with an ECHO SPEECH SYNTHESIZER. There are now three new additions to the ECHO family: the ECHO-80 (TRS-80 MODEL I), the ECHO-GP (general purpose serial/parallel), and the ECHO-100 (S-100). These join the already popular ECHO II (Apple).

All ECHO SYNTHESIZERS use a combination of Texas Instrument's LPC synthesis and phoneme coding to produce an unlimited vocabulary while using a minimal amount of memory. New male and female phonemes and TEXTALKER™ software (converts English text to speech) make them easier to use than ever before.

Speech applications are virtually unlimited, including education, games, and aiding the handicapped. The flexibility and low price of the ECHO SYNTHESIZERS make them the logical choice for adding speech to your system. For further information see your dealer or contact Street Electronics Corporation.



STREET ELECTRONICS
CORPORATION

3152 E. La Palma Ave., Suite C
Anaheim, CA 92806 (714) 632-9950

structures: the supplier produces elements, and the program that invoked the loop construct consumes them. As we saw earlier, one way to do this is to assign responsibility as follows:

Producer

implements: do: aBlock
delivers values by: aBlock value: theNextElement

Consumer

receives values using: [:elementName]
 dosomethingToTheElement]

Under this arrangement, the producer can use any desired control structure internally, just by sending the message:

aBlock value: theNextElement

to the block whenever a new element has been generated; the consumer, however, is confined to executing the same block for each element. The other arrangement reverses the situation:

Producer

implements: atEnd, next
delivers values by: returning a value from next

Consumer

receives values using: producer next

Under this arrangement, the producer has to use instance variables, rather than control variables, to remember what state it is in, but the consumer can call for new elements using any control pattern it wants.

The control structure *coroutines* allows both the producer and consumer to use any control pattern. Notice that in the first arrangement, the producer has to retain its argument aBlock to be able to send it value: for each element; in the second arrangement, the consumer has to retain the producer to be able to send it next for each element. In the coroutine arrangement, both sides retain a common object called a *port*. The purpose of the port is to remember the control state of one partner while the other partner is running. Let us now build a port in which the consumer invokes the producer with the messages next and atEnd, and the producer invokes the consumer with the messages nextPut: anElement and markEnd. A loop in this implementation might look similar to the following:

Consumer

```
| first second |
portForProducer ← someCollection asProducer.
"Here is a sample loop that takes elements two at a time"
[portForProducer atEnd]
whileFalse:
  [first ← portForProducer next.
   second ← portForProducer next.
   "Do something with first and second"]
```

Producer Collection

```
asProducer | port |
port ← Port new.
port producer: [CollectionProducer of:
  self with: port].
"Create a new process for the producer"
↑ port
```

CollectionProducer

```
of: aCollection with: portForConsumer | |
"Here is a sample loop that generates elements three at a
time"
[someCondition]
whileTrue:
  [portForConsumer nextPut:
   someComputation1.
   portForConsumer nextPut:
   someComputation2.
   portForConsumer nextPut:
   someComputation3].
portForConsumer markEnd
```

The code in both consumer and producer can involve any combination of loops, messages, or other control structures: the consumer can request a new element at any time with portForProducer next, and the producer can deliver an element any time it has control with port-

computer case company

comp case

• RS205

- AP101 Apple II with Single Disk Drive \$109
- AP102 Apple II with Double Disk Drives 119
- AP103 Apple II, 9 inch Monitor & Double Drives ... 129
- AP104 Apple III, two additional Drives & Silentype 139
- AP105 12 inch monitor plus accessories 99
- RS201 TRS-80 Model I, Expansion Unit & Drives.... 109
- RS202 TRS-80 Monitor or TV set 84
- RS204 TRS-80 Model III 129
- RS205 Radio Shack Color Computer 89
- P401 Paper Tiger 440/445/460 99
- P402 Centronics 730/737 - Line Printer II/IV 89
- P403 Epson MX70 or MX80 89
- CC90 Matching Attaché Case 75

computer case company

5650 INDIAN MOUND CT. COLUMBUS, OHIO 43213 (614) 868-9464

MasterCard VISA

ForConsumer nextPut: anElement. Interleaving a consumer that wants pairs of elements with a producer that generates triplets is a very simple example of the freedom that both partners enjoy in this arrangement.

To implement Port we need to consider how Smalltalk

class name	Port
superclass	"none added here"
instance variable names	consumerSemaphore producerSemaphore nextElement endMark
class messages and methods	"none defined here"
instance messages and methods	<p><i>initialize</i></p> <p>producer: aBlock </p> <p>"Assume we are running in the consumer process, so create a new process for the producer."</p> <p>endMark ← false.</p> <p>consumerSemaphore ← Semaphore new.</p> <p>producerSemaphore ← Semaphore new.</p> <p>producerSemaphore signal. "So producer will proceed the first time"</p> <p>aBlock fork</p> <p><i>consumer</i></p> <p>next anElement </p> <p>consumerSemaphore wait. "Wait for producer to deliver an element"</p> <p>endMark ifTrue: "No more elements"</p> <p>[self error: 'Attempt to read past last element'].</p> <p>anElement ← nextElement.</p> <p>producerSemaphore signal. "Restart producer"</p> <p>↑ anElement</p> <p>atEnd </p> <p>consumerSemaphore wait. "Wait for an element or end mark"</p> <p>consumerSemaphore signal. "Doesn't consume the element"</p> <p>↑ endMark</p> <p><i>producer</i></p> <p>nextPut: anElement </p> <p>producerSemaphore wait. "Wait for consumer to have taken last element"</p> <p>nextElement ← anElement.</p> <p>consumerSemaphore signal "Restart consumer"</p> <p>markEnd </p> <p>producerSemaphore wait.</p> <p>endMark ← true.</p> <p>consumerSemaphore signal</p>

Table 11: Class template for class Port.

allows us to get hold of our current control state, since whenever control goes from consumer to producer or vice versa, we have to save the state of the partner that is giving up control. For just such purposes, Smalltalk provides a primitive notion of a *process*, an entity which has its own control state and can be suspended and resumed. The usual way to create a new process is with:

aProcess ← [someComputation] newProcess.

The process can then be started up by:

aProcess resume

and it will compute "in parallel" with the current computation until it finishes someComputation or it (or some other process) executes:

aProcess terminate

which stops it midflight. Alternatively:

[someComputation] fork

creates and starts an unnamed process that will proceed until the computation finishes.

To allow processes to synchronize their control or their use of data in an orderly way, Smalltalk provides *semaphores*. A semaphore logically represents the current availability of a finite resource: aSemaphore signal indicates that one unit of the resource has just become available, and aSemaphore wait indicates that the currently running process needs to take one unit of the resource and must wait if none is available (presumably until some other process does aSemaphore signal). A useful special case of this is a semaphore that always holds either 1 (meaning a resource is available) or 0 (meaning it is unavailable).

As an aside, we note that semaphores could have been implemented in Smalltalk (ie: not as primitive entities) at a considerable cost in performance: we only need the ability to temporarily guarantee that no other process could run aside from the one currently running (on this processor in a multiprocessor system). Smalltalk provides semaphores at a primitive level because they are such a help in building multiprocess systems that we wanted people to feel free to use them without worrying about their cost.

Given processes and semaphores, we are ready to implement Port (see table 11). The producer and consumer will each run in a process of their own, and we will use semaphores to make sure only one of them is running at a time. (The reader can easily imagine and might enjoy thinking about a version of coroutines which allows the producer to "get ahead" of the consumer. This requires a queue between the two, like the SharedQueue we will develop later.) The "resource" controlled by the semaphores will be free access to the variables in the port, nextElement and endMark, under the following arrange-

Consumer	Producer
"In portForProducer next:"	
consumerSemaphore wait.	
"Semaphore started with 0, consumer waits."	
	"In port nextPut:"
	producerSemaphore wait.
	"Semaphore started with 1, now has 0"
	nextElement ← anElement.
	consumerSemaphore signal.
	"Semaphore started with 0, now has 1"
anElement ← nextElement.	
producerSemaphore signal.	
"Semaphore had 0, now has 1"	
↑ anElement	
	"nextPut: returns, producer proceeds."
	"Later, producer does another port nextPut:"
	producerSemaphore wait.
	"Semaphore goes from 1 to 0 again"

Table 12: Dialog between consumer and producer objects using the Port defined in table 11.

ment: when consumerSemaphore has a 1, it means nextElement has something in it (or endMark has been set) and the consumer needs to run; when producerSemaphore has a 1, it means nextElement is vacant and the producer needs to run. Notice that the next and nextPut: methods are very similar.

A partial trace through an exchange of control would look like the dialog shown in table 12. Note that if the producer reached the second wait before the consumer took the first element, the producer would wait until the consumer did the producerSemaphore signal. A full discussion of how semaphores should be used to produce minimum waiting, minimum process switching, and correct synchronization is beyond the scope of this article; one important and useful special case will be presented in

the following section.

Monitors—Asynchronous Structures

Even in personal computer systems there are often reasons to allow for the possibility of several things happening "at once" (ie: not synchronized with each other beforehand). The best examples involve communication with other users. For example, your machine could be listening for incoming messages through a network connection. But even on an isolated personal machine, you would like to be able to start the system on a time-consuming project (like printing on a hardcopy device) and continue to do interactive work. As we saw before, Smalltalk provides the ability to create independent processes and set them going "in parallel," and provides

WE'VE DONE IT AGAIN: 64K OF S-100 STATIC RAM — PLUS Z80A CPU BOARD — FOR AN AMAZINGLY LOW \$899!

That's right, \$899...and **Mullen Computer Products** makes it possible! We bought a batch of production overrun Z80A* CPU boards, along with static RAM boards with some minor cosmetic defects, from a leading IEEE-696/S-100 board manufacturer. For expanding present systems or upgrading older systems to the current state-of-the-art, these boards are the lowest priced around. But you don't sacrifice one bit of quality—here's what we mean:

64K Static RAM—Includes bank select and extended addressing for maximum flexibility. Fast low power operation, many convenience features.

32K STATIC RAM BOARDS ALSO AVAILABLE SEPARATELY FOR \$399.

Same boards as described above; all that's needed to complete these fast, low power memories is to insert some ICs into the board's sockets and solder in a few other parts. With instructions and all components. Don't miss out on this exceptional memory offer! (Limited quantity, first-come-first-served.)

CPU Board—Runs at 4 MHz and takes full advantage of all Z-80A features. Features include maskable interrupts, extended addressing, RAM/ROM sockets for optional RAM/ROM, and much more.

Boards may be assembled or partially assembled; instructions included on how to complete partially assembled boards (takes less than an evening's work), along with all other documentation. Best of all, you'll end up with the same reliable, low power performance you've come to expect from the boards made by this prominent company.

Don't miss out on the CPU/memory deal of the year—these are limited quantity, first-come-first-served.

MULLEN Computer Products
BOX 6214, HAYWARD, CA 94544

CONDITIONS OF SALE: VISA* /Mastercard* accepted; call (415) 783-2866. Include \$1.50 for shipping and handling. California residents add tax.
*Zilog is a trademark of Zilog

class name	Queue
superclass	Object
instance variable names	array writer reader
class messages and methods	
creation	
new: size	
↑ self allocate init: size	
instance messages and methods	
initialization	
Init: size	
array ← Array new: size.	
reader ← 0.	
writer ← 0	
access	
removeFirst	
reader ← reader + 1.	
↑ array at: reader	
addLast: anElement	
writer ← writer + 1.	
array at: writer put: anElement	

Table 13: Class template for an initial implementation of class Queue.

Process A	Process B
reader ← reader + 1.	reader ← reader + 1.
↑ array at: reader	↑ array at: reader

Table 14: Execution of the removeFirst method using the implementation of table 13.

class name	SharedQueue
superclass	Object
instance variable names	array writer reader accessSemaphore
class messages and methods	
creation	
new: size	
↑ self allocate init: size	
instance messages and methods	
initialization	
Init: size	
array ← Array new: size.	
reader ← 0.	
writer ← 0.	
accessSemaphore ← Semaphore new.	
accessSemaphore signal "Give it the baton"	

Table 15: Class template for class SharedQueue.

semaphores for synchronizing their behavior.

From semaphores we can easily build a more useful construct, called a *monitor*. The purpose of a monitor is to allow several processes to communicate with a data structure without getting in each other's way; failing to provide for this is another common source of bugs—consider the simple-minded implementation of a queue given in table 13. (The reader should ignore the obvious bugs: there is no check for an empty queue or for exceeding the size of the array.)

Suppose two processes both try to remove an element at about the same time, and the removeFirst method gets executed as shown in table 14 (the flow of time is vertical down the page, interleaving the statements executed by process A in the left column and process B in the right). One element is skipped—and one is returned twice! The solution to this problem is to consider “permission to update the state of the queue” as a resource that only one process can hold at any given time, like the baton in a relay race. So we can construct a safe Queue by giving it a semaphore that starts out with one unit of the resource (see table 15).

A pattern we will encounter in the implementation of SharedQueue will be to reserve a resource during the execution of a piece of code:

someSemaphore wait.	"Acquire the resource"
someComputation.	
someSemaphore signal.	"Release the resource"

The code someComputation is called a *critical section*. We would like to be able to write the previous code fragment as:

someSemaphore critical: [someComputation].

class name (existing)	Semaphore
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	"none added here"
instance messages and methods	
critical sections	
critical: aBlock result	
self wait. "Acquire the resource"	
result ← aBlock value. "Do the computation, save the result"	
self signal. "Release the resource"	
↑ result "Return the result of the computation"	

Table 16: Template showing additions to existing class Semaphore.

The system actually provides this message to Semaphore, with a straightforward implementation which is shown in table 16. It is then easy to appropriately modify the two messages in SharedQueue (see table 17).

If two processes try to access the queue, the interchange shown in table 18 occurs (with a few steps left out). Note that the variable `anElement` is a local variable, and since the two processes have different contexts (despite the fact that they share the same instance of SharedQueue in this example), the variable `anElement` in Process A is different from `anElement` in Process B.

class name	SharedQueue
superclass	"none defined here"
instance variable names	"none defined here"
class messages and methods	
	"none defined here"
instance messages and methods	
access	
removeFirst <code>anElement</code>	
↑ <code>accessSemaphore critical:</code> "Reserve access for the duration of the block"	
[<code>reader ← reader + 1.</code>	
<code>array at: reader]</code>	
addLast: anElement	
<code>accessSemaphore critical:</code> "Reserve access for the duration of the block"	
[<code>writer ← writer + 1.</code>	
<code>array at: writer put: anElement]</code>	

Table 17: Class template for class SharedQueue.

Final Comments

Many languages don't have the flexibility we've just described; others, such as assembly language, have great flexibility at the expense of readability. What is it about the Smalltalk-80 language and system that makes all of the foregoing both possible and fairly readable? Three things come to mind:

- The existence of blocks, with and without arguments, and the simple square-bracket notation for writing them. This makes it possible to pass a piece of code to the implementor of a control structure, which can then execute the code whenever and however it is appropriate. ALGOL and LISP have constructs which capture some, but not all, of the power of blocks.
- The ability to manipulate the control state directly, as in the dynamic binding example. Of course disaster can result if you aren't careful, but a challenge like this is necessary to exploit the full power of your imagination. InterLISP (a widely used LISP dialect) has facilities which capture some of the power of Smalltalk in this area.
- The accessibility of the entire system to modification. Several of the examples we've described involve adding messages to fundamental classes like `Object` and `BlockContext`. Restraint is important here too. Several LISP systems derive tremendous power from this kind of openness.

Of course, we pay a price for all this flexibility and simplicity. A discussion of the time and space cost of blocks, visible control state, and a completely accessible system is beyond the scope of this article; we will just observe that the elementary instructions which implement control structures (branch, call, and return) take about the same proportion of the total execution time in a typical Smalltalk-80 implementation as they do in more conventional languages that don't use globally optimizing compilers. ■

Process A

```

accessSemaphore wait
"Semaphore now has 0 units of resource"
reader ← reader + 1.

anElement ← array at: reader.
accessSemaphore signal.

"Process B can proceed now, but immediately
reacquires the semaphore"

↑ anElement

```

Process B

```

accessSemaphore wait.
"Waits here"

```

```

reader ← reader + 1.
anElement ← array at: reader.
accessSemaphore signal.
↑ anElement

```

Table 18: Execution of the `removeFirst` method using the implementation of table 17.