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 Kth 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-

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.

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	

```
constructing
, aBlock | |
    ↑BlockCollection with: self with: aBlock
accessing
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 ifTrue: [↑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	

```
constructing
, 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).*

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

```
enumeration
asSupplier | |
    ↑ IndexedCollectionSupplier of: self
```

class name	IndexedCollectionSupplier
superclass	Object
instance variable names	collection position
class messages and methods	

```
creation
of: aCollection | |
    ↑self new of: aCollection
```

instance messages and methods

```
creation
of: aCollection | |
    collection ← aCollection.
    position ← 0

accessing
atEnd | |
    ↑position > = collection size
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).*

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 additional 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 (existing)	Collection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	

```
enumerating
with: anotherCollection do: aBlock
    | mySupplier itsSupplier |
    mySupplier ← self asSupplier.
    itsSupplier ← anotherCollection asSupplier.
    [mySupplier atEnd or: [itsSupplier atEnd]]
        whileFalse:
            [aBlock
                value: mySupplier next
                value: itsSupplier next]
```

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	

```
searching
maxBefore1000
    | 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
```

Table 7: *Template showing additions to existing class* OrderedCollection.

class name	BlockWithExit
superclass	Object
instance variable names	block exitBlock
class messages and methods	

```
creation
with: aBlock | |
    ↑ self new with: aBlock
```

instance messages and methods	

```
creation
with: 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
withExit | |
    ↑ 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).*

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*

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-

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

```
initialize
producer: aBlock | |
    "Assume we are running in the consumer process, so create a
    new process for the producer."
    endMark ← false.
    consumerSemaphore ← Semaphore new.
    producerSemaphore ← Semaphore new.
    producerSemaphore signal. "So producer will proceed
    the first time"
    aBlock fork
consumer
next | anElement |
    consumerSemaphore wait. "Wait for producer to deliver
    an element"
    endMark ifTrue: "No more elements"
        [self error: 'Attempt to read past last element'].
    anElement ← nextElement.
    producerSemaphore signal. "Restart producer"
    ↑ anElement
atEnd | |
    consumerSemaphore wait. "Wait for an element or end
    mark"
    consumerSemaphore signal. "Doesn't consume the ele-
    ment"
    ↑ endMark

producer
nextPut: anElement | |
    producerSemaphore wait. "Wait for consumer to have
    taken last element"
    nextElement ← anElement.
    consumerSemaphore signal "Restart consumer"
markEnd | |
    producerSemaphore wait.
    endMark ← true.
    consumerSemaphore signal
```

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

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 | anElement |
    ↑ accessSemaphore critical: "Reserve access for the duration of the block"
        [reader ← reader + 1.
        array at: reader]
addLast: anElement | |
    accessSemaphore critical: "Reserve access for the duration of the block"
        [writer ← writer + 1.
        array at: writer put: anElement]
```

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	Process B
accessSemaphore wait "Semaphore now has 0 units of resource"	
reader ← reader + 1.	
	accessSemaphore wait. "Waits here"
anElement ← array at: reader. accessSemaphore signal. "Process B can proceed now, but immediately reacquires the semaphore"	
	reader ← reader + 1. anElement ← array at: reader. accessSemaphore signal. ↑ anElement
↑ anElement	

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