A New Object-Oriented Programming Language: *sh*

Jeffrey S. Haemer
Canary Software, Inc.

Abstract

Many have frittered away their time on C++, while overlooking the new, POSIX.2-required, object-oriented language: **sh.** As will be clear from the enclosed code, the name may allude to the fact that the author would be embarrassed to have anyone find out about it.

This paper introduces a tiny, object-oriented programming system written entirely in POSIX-conforming shell scripts.

1. Overview

Object-oriented programming is currently all the rage [King89]. Though we normally use languages designed specifically for the task, they aren't always necessary. Here, we illustrate this point by doing object-oriented programming in the shell.

In what follows, object classes are shell scripts and objects are running processes. Methods are invoked by messages passed to objects through FIFOs (named pipes). The methods themselves are implemented as shell functions; function polymorphism is guaranteed because separate programs have separate name spaces. A class hierarchy is provided by the file system itself.

Sensible default actions are taken by objects when they're sent messages for which they lack explicitly defined methods. Debugging code can be added to objects on the fly, that is, after they've been created.

While the system is unconventional, only a toy, and downright slow, its implementation is straightforward and its use instructive. For example, figure 1 shows an implementation of the examples used in Roger Sessions' Summer, '93 USENIX Invited Talk [Sessions93].

Sessions' talk used application code size as one measure of the advantages of object-oriented programming. By that measure, and with the same examples, this system is better than C++. In fact, the core of the entire system, the two shell scripts *create* and *send*, total a little over 100 lines of code. If you don't find your favorite OOP feature, it may not be very hard to add it.

```
$ cat animalia
new animal pooh bugs

send pooh setName Pooh Bear
send pooh setFood Hunny
send bugs setName Bugs Bunny
send bugs setFood Carrots

for i in pooh bugs
do
      send $i getName
      send $i getFood
done

echo

new dog Snoopy
new littleDog Toto
new bigDog Lassie

for i in Snoopy Toto Lassie
do
      echo $i says
      send $i bark
      echo
done

destroy bugs pooh
destroy Snoopy Toto Lassie
```

```
$ ./animalia
My name is: Pooh Bear
My favorite food is: Hunny
My name is: Bugs Bunny
My favorite food is: Carrots

Snoopy says
Unknown Dog Noise

Toto says
woof woof
woof woof

Lassie says
WOOF WOOF
WOOF WOOF
WOOF WOOF
WOOF WOOF
WOOF WOOF
```

Figure 1. Animalia

2. Design

As a foundation, we begin by reviewing the three basic object-oriented features: encapsulation, function polymorphism, and inheritance. These requirements, plus sloth — an eagerness to let UNIX and the shell do as much of the job as possible — lead directly to most major design decisions.

- Encapsulation

  All object-oriented systems provide data abstraction and encapsulation; they let programmers create and operate on objects that have user-defined types, while hiding all knowledge about how those operations and types are implemented. Programs are prevented from manipulating an object's internal data structures except through the methods that operate on those objects.

  UNIX processes are attractive candidates for objects. Each running, UNIX process has its own name and address space; it's impossible to look at or tweak the insides of an already running process unless the process is running under a debugger.

- Objects and the object-class hierarchy

  Object-oriented programming systems try to maximize code reuse by letting new data types inherit methods and the data structures they operate on from their "parent" classes. This sort of inheritance produces a tree-structured class hierarchy. Implementing class hierarchies means picking a way to define trees.

  UNIX has two ubiquitous tree structures that seem plausible choices, either of which might be worth barking up: the file system and the process tree. Deciding to make object classes conceptually distinct from objects, and making the latter simply instances of object classes, guides our choice. In this system, executing processes are objects; programs are their definitions, the object classes. The file system implements the class hierarchy. Although a process can be any executing program image, in this system all objects will be shell scripts.

- Polymorphism

  A good object-oriented system lets the programmer extend the suite of object classes without changing existing software. Programs can operate on new kinds of data that get defined long after the code is written. The same operation may be implemented different ways for different kinds of data, but the invocation of the operation is identical. What separate compilation provides for functions, object-oriented programming provides for data.

For example, if each new object responds to *pass()*, there is no need to change base code from

```
pass(X)
```

to

```
if (object_type(X)==BILL)
    pass_bill(X);
else if (object_type(X)==BUCK)
    pass_buck(X);
else if (object_type(X)==GAS)
    ...
```

C++ and UNIX device drivers implement this sort of behavior by using tables of function pointers. *read()* is always *read()*, whatever the device, because the kernel figures out what routine to call based on the device that's being read from.

Many other object-oriented systems, like Smalltalk, implement polymorphism by passing messages between objects, which interpret the messages at runtime. Thus, the instruction

```
X pass
```

works whether X is a bill or a buck, because it just sends a message. Both bill and buck understand the message "pass," but each implements the operation with a data-type-specific method.

Here, we take this latter approach. Class definitions are shell scripts. Methods are shell functions. Messages are just the names of the functions, sent as ASCII strings. Two different scripts are free to use identical names for completely different functions.

One side-effect of this approach is that messages sent to objects that are *not* names of functions are interpreted as other sorts of executable statements: built-ins and shell commands. On first blush, this seems like a horrible bug; in practice, and to my surprise, it feels like a feature.

3. Implementation

Each object is a simple, infinite loop:

forever
read message from FIFO
execute it as a command

The FIFOs are created in a pre-arranged spot in the file system and have names tied to the names of their corresponding objects. Messages are sent with the program *send*. The command

`$` *send pooh setFood Hunny*

just writes the message `setFood Hunny` to pooh's input channel `/tmp/ipc/pooh/in.`

Object creation is trickier, but not by much. Each object class is a shell script, stored in a directory tree where the directory and subdirectory names are class and subclass names. Each directory contains one script, named `class`, that defines the methods for the class corresponding to that directory. A request to create an object of type *name* starts up a process like this:

use ***find*** *to find directory* ***name***

source all the class methods
from the root of the class tree
down to that definition

create a FIFO tagged to
the name of the object

loop forever, reading and executing messages
(as shown above).

Taken together, `send` and `create` are currently only a little over 100 lines of shell code. (The `new` and `destroy` commands, used in the example in the first section, are just loops that call `create` and `send exit` for each of their arguments.)

A handful of interesting things fall out of implementing objects this way.

- Process manipulation commands can be used to handle objects. You can search for objects with **ps** and destroy them with **kill -9**.
- Every object understands normal shell commands. Not only can you see if an object is alive, but you can see if it's paying attention with commands like this:

  `$` *send pooh date*
  `Sun Jan 23 21:00:25 MST 1994`

- You can kill objects with `exit`. This is enormously comforting. (Since I haven't gone to great lengths to make this system any more bullet-proof than it deserves to be, it's also enormously necessary.)
- You can add methods to objects on the fly. This sort of thing actually works:

  `$` *send X 'zzazz() { echo foo }'*
  `$` *send X zzazz*
  `foo`

  From time to time, this last feature has proven itself a useful debugging tool.

4. Real Code

Enough abstract chatter. Let's see some code.

4.1. send

send, shown in figure 2, sends messages to objects.

```
# send a message

case $1 in
  -d) msgtype=D
    shift ;;
  *)  msgtype=C ;;
esac

T_NAME=$1
shift

T_DIR=/tmp/ipc/$T_NAME
```

```
T_IN=$T_DIR/in
T_OUT=$T_DIR/out
USAGE=\
"usage: $(basename $0) obj msg"

abort() { # print and bail
  echo $* 1>&2
  exit 1
}

test $# -gt 0 || abort $USAGE

test -d $T_DIR ||
abort $T_NAME: no such object

echo -e $msgtype "$*" > $T_IN
test $msgtype = "D" || cat $T_OUT

exit 0
```

Figure 2. send

All object I/O channels are in subdirectories of */tmp/ipc*. Each named object has a subdirectory that corresponds to its name, `T_DIR`, and all files associated with that object are within that directory. By default, each object has at least an input channel, `T_IN`, through which messages arrive, and an output channel, `T_OUT`, to which returns are sent.

The code shown above sets environment variables to point at the right channels, and then, after a brief sanity check, echoes its argument into the input channel and reads the objects response from the output channel.

There are at least two restrictions of this design. First, the name space is global to the system; only one object on the entire system can be called "foo" at any given time. Second, there's no provision for tying returns to the messages that elicited them; if two objects send messages to a third at nearly the same time, there isn't any way to guarantee that the return value one of the senders retrieves corresponds to the message it sent. A more sophisticated implementation might nest object directories as subdirectories under the directories of the objects that created them, and use a more sophisticated messaging scheme to provide a virtual circuit between the messenger and the messagee.

Even after accepting these limitations, at least two problems require immediate solution.

The first of these is the deadlock that arises when object A sends a message to object B and object B, or some object further down the line, sends a message to object A before object B has replied to A's original message. In these cases, object A cannot read the incoming message because it is blocked reading B's output channel. The general case is a general problem, but in a some cases object A doesn't really need B's answer, and can go on to listen for incoming messages as soon as it dispatches a message to B. For just such cases, **send** accepts a flag, *-d*, that means "don't wait for an answer." **Send** prepends a 'D' to such "datagrams," and replies are neither expected nor supplied.

The second problem is trickier. In the absence of special arrangements, an open of a FIFO for writing will only complete when that FIFO has an available reader. Consider, then, what happens when object A sends a message to itself, using a command like

```
echo $msg > /tmp/ipc/A/in.
```

The **echo** will block, awaiting a reader, preventing A from ever executing the read that would move **echo** past the block.

The current implementation side-steps this problem by providing each object with a built-in version of **send**. Whenever an object notices that it is sending a message to itself, it executes the message directly instead of trying to write the message to its own input channel. (See figure 7.) An alternative to this would be putting **echo** in the background, but that would use up process slots, a resource that this system already strains. Another alternative might be writing substitutes for **read** and **echo**.

4.2. create

More complex than **send** is **create**, shown in figure 3.

```
# create a new object

O_DIR=/tmp/ipc/$2
O_IN=$O_DIR/in
O_OUT=$O_DIR/out

O_BIN=${O_BIN:-$HOME/obj/bin}
O_CLASS=$1
O_NAME=$2
O_PATH=/bin:/usr/bin:$O_BIN
O_ROOTS=${O_ROOTS:-$HOME/obj/objs}
export O_BIN O_CLASS O_NAME
```

```
export O_PATH O_ROOTS

USAGE=\
"usage: $(basename $0) class obj"

abort() {
  echo $* 1>&2
  exit 1
}

test $# -eq 2 || abort $USAGE

# cleanliness is next to godliness
cleanup() {
  trap "" 0 1 2 3 15
  rm -rf $O_DIR
  exit 0
}

event_loop() {
  trap "cleanup" 0 1 2 3 15
  while read pkt < $O_IN
  do
    type=${pkt%% *}
    msg=${pkt#[A-Z] }

    if test $type = "D"
    then
      PATH=$O_PATH eval $msg
    else
      PATH=$O_PATH eval $msg \
        >$O_OUT
    fi
    # hack around BSDI timing bug
    sleep 1
  done
}

get_obj_chain() {

  # find object and superclasses
  IFS=:
  set $O_ROOTS
  IFS='   '
  for i
  do
    test -f $i/class || continue
    obj_root=$(
      find $i \
        -type d \
        -name $O_CLASS \
        -print
    )
    test -n $obj_root && break
  done
  test -z $obj_root && abort $USAGE

  # now set up paths
  d=$obj_root
  O_PATH=$O_PATH:$d
  O_DEFS=$d
  while test "$d" != "$i"
  do
    d=${d%/*}
    O_DEFS=$d" $O_DEFS"
    O_PATH=$O_PATH:$d
  done

  unset d i
}

# create message channels
make_channels() {
  test -d $O_DIR &&
    abort "Duplicate $O_NAME"
  mkdir -p $O_DIR ||
    abort "Can't make $O_DIR"
  mkfifo $O_IN ||
    abort "Can't make $O_IN"
  mkfifo $O_OUT ||
    abort "Can't make $O_OUT"
}

# build the object from definitions
mkobj() {
  for d in $O_DEFS
  do
    . $d/class 2>/dev/null
  done
}

get_obj_chain
make_channels
mkobj
event_loop &
```

Figure 3. create

Following initialization and sanity checks, **create** makes four function calls to create an object. The first, *get_obj_chain*, sets the variable `O_DEFS` to a list of directory names, starting at the root object directory, that end in the directory that defines the class. For the class *littleDog*, from our earlier example, `O_DEFS` would be set to `objs objs/animal objs/animal/dog objs/animal/dog/littleDog`

Next, *make_channels* creates the input and output channels used by `send`.

Third, *mkobj* visits the directories in `O_DEFS`, reading class definitions. Because of the order in which *get_obj_chain* sets up `O_DEFS`, methods defined in subclasses supplement or override those defined in parent classes.

Finally, *event_loop* loops infinitely, reading messages and writing responses on the pair of message queues set up by *make_channels*. If the message is the name of a function call — a method — that function is invoked. Otherwise, *eval* looks for a shell built-in or a UNIX command to execute.

A disadvantage of the approach sketched above is that there isn't an easy way to say "use my parent's definition of this method"; when an object definition overrides a method defined by a parent, that parental method becomes completely unavailable.

In an earlier version of this code, *event_loop* looked like this:

```
event_loop() {
    trap "cleanup" 0 1 2 3 15
    while read msg < $O_ICHAN
    do
        eval $msg > $O_OCHAN
    done
    cleanup
}
```

In the version shown in figure 3, *get_obj_chain* stores the path to the directory that contains the object definition in `O_PATH`, for later use in creative ways, including prefixing it to the path used by **eval**. Having that path available makes it possible to back up through the class hierarchy searching for a parental method. I've experimented with this, but the trick isn't entirely satisfactory; a more sophisticated implementation should find a more interesting way to use `O_PATH` or `O_DEFS` to gain access to parental-class methods.

Like **send**, which has a single, global namespace for objects, **create** uses a global name space for object classes. The system will not support two different definitions of class "dog". On the other hand, users can point at their own object-class definitions by setting `O_ROOTS`, even invocation by invocation.

Another limitation of this system is that it is restricted to single inheritance; each class has one and only one parent class — that of its parent directory. Although links might make it possible to provide an interesting way to implement and explore multiple inheritance, everyone knows that multiple inheritance is a bad idea [Cargill91].

4.3. new and destroy

Returning now to the example, we can show **new** (figure 4) and **destroy** (figure 5).

```
# create a set of objects

USAGE=\
"new class obj [obj ...]"
abort() {
  echo $* 1>&2
  exit 1
}

test $# -ge 2 ||
abort "$USAGE"

class=$1
shift

for i
do
  create $class $i
done
```

Figure 4. new

```
# destroy a set of objects

USAGE=\
"destroy obj [obj ...]"
abort() {
  echo $* 1>&2
  exit 1
}

test $# -ge 1 ||
abort "$USAGE"

for i
do
  send $i destroy
done
```

Figure 5. destroy

As advertised earlier, each of these is a simple loop. The earliest version of destroy was even simpler:

```
for i
do
    send $i exit
done
```

This works because each object inherits the methods understood by the base class — the shell augmented by a few basic methods. The current version calls *destroy* instead, which lets each object define its own destructor. (The base class defines a simple default *destroy*, shown in the next section.)

One alternative to explicit calls to **destroy** for each object would be to make **new** keep track of objects it has created and let **destroy** destroy everything.

Because of encapsulation, the easiest implementation would incorporate *new* as a method in a more sophisticated base class. An odd side-effect of this is that classes could redefine *new*.

4.4. hop: a simple class

Having constructed the infrastructure, let's look at a simple class definition (figure 6).

```
$ cat objs/hop
# class hop

hop() {
  if test "$*" = "on pop"
  then
    echo -n "Stop!  "
    echo "You must not hop on pop."
  else
    echo "hippity hop"
  fi
  return 0
}
$ create hop X
$ send X hop
hippity hop
$ send X hop on pop
Stop!  You must not hop on pop.
$ send X foo
$ send X exit
$ send X hop
X: no such object
```

Figure 6. Class hop

The entire class definition is a single method: *hop*.

Although this is an elementary example, it illustrates a few interesting points:

(1) Defining methods is easy; it doesn't require a lot of special syntax.

(2) Class definitions are small. While code size is not the only measure of the quality of a programming language — else we would all program in APL — code size strongly affects maintenance and debugging efforts; bug frequency per line appears to be roughly constant across languages [Brooks75]. Less code, fewer bugs.

As an illustration, Sessions contrasts the code to make a dog bark in C:

```
void printDog(dog *thisDog,
  int dogType)
{
  printf("\n%s says\n",
    getName((dog *) thisDog));
  switch dogType {
    case DOG:
      dogBark(
        (dog *) thisDog);
      break;
    case LITTLEDOG:
      littleDogBark(
        (littledog *)
        thisDog);
      break;
    case BIGDOG:
      bigDogBark(
        (bigdog *) thisDog);
      break;
  }
}
```

with the code to do the same job in C++:

```
void printDog(dog *thisDog)
{
  printf("\n%s says\n",
    thisDog->getName());
  thisDog->bark();
}
```

Here's the same code in the shell:

```
echo $thisDog says
send $thisDog bark
```

(3) We've chosen to ignore nonsense requests. The consequences of changing that decision can be explored by toying with *event_loop* in

create

(4) Methods can have arguments.

This example becomes even more interesting when we notice that we can invoke methods that aren't defined by the class. As figure 7 shows, methods defined by parent classes (in this case, the class defined in directory *objs*) are inherited by their subclasses

```
$ create hop X
$ send X self
X
$ send X class
hop
$ cat objs/class
# fundamental methods

abort() { # print and bail
  echo $* 1>&2
  exit 1
}

class() {
  echo $O_CLASS
}

debug() {
  echo $O_NAME: $*
}

defs() {
  for d in $O_DEFS
  do
    cat $d/class 2>/dev/null
  done
}

_destroy() {
  test $# -eq 0 && exit 0
  test $0 = "self" && exit 0

  for i
  do
    send -d $i destroy
  done
}

destroy() {
  _destroy $*
}

self() {
  echo $O_NAME
}

send() {  # send a message

  case $1 in
    -d) msgtype=D;
      shift ;;
    *)  msgtype=C ;;
  esac

  T_NAME=$1
  shift
  T_DIR=/tmp/ipc/$T_NAME
  T_IN=$T_DIR/in
  T_OUT=$T_DIR/out

  USAGE="usage: send obj msg"
  test $# -gt 0 || abort $USAGE

  if test "$T_NAME" = "$O_NAME" ||
    test "$T_NAME" = "self"
  then
    PATH=$O_PATH eval $*
    return 0
  fi

  test -d $T_DIR ||
  abort $T_NAME: no such object

  echo -e $msgtype "$*" > $T_IN
  test $msgtype = "D" || cat $T_OUT
}
```

Figure 7. The base class

Most of the methods defined in `objs/class` are simple utility methods. The motivation for the one long method, *send*, was given in section 4.1.

4.5. animals

The code for to our original animal example shows inheritance in practice (figure 8).

```
$ cat objs/animal
# base animals methods

name=$O_NAME
food="Unknown food."
```

```
setName() {
  name=$*
}
getName() {
  echo My name is: $name
}
setFood() {
  food=$*
}
getFood() {
  echo My favorite food is: $food
}
$ cat objs/animal/dog
# dog: all bark, no bite

bark() {
  echo Unknown Dog Noise
}
$ cat objs/animal/dog/littleDog
# a little dog

bark() {
  echo woof woof
  echo woof woof
}

$ cat objs/animal/dog/bigDog
# a BIG DOG

bark() {
  for i in 0 1 2 3 4
  do
    echo WOOF WOOF
  done
}
```

Figure 8. Animal Objects

Here, the class *animal* defines methods for setting and getting an animal's name and favorite food; the subclass *dog* adds a way to make the animal bark, and sub-sub-classes for little and big dogs replace that method with ones that generate size-appropriate noises.

5. Applications

> Sir, a woman's preaching is like a dog's walking on his hinder legs. It is not done well; but you are surprised to find it done at all.
>
> Boswell's *Life of Johnson, vol 1,* p 428, 31 July 1763

"Cute idea," you say, "but is this good for implementing real applications?" Probably not. Still, it seems worth sniffing around to see what sorts of things besides barking dogs might be interesting to implement with it.

5.1. Starting small ...

When I was soliciting suggestions for interesting applications to implement, Doug Pintar, of Aztec Engineering, laconically suggested **emacs**. While an **emacs** implementation might not fit within the page-limit length imposed by this conference, I can include a more contained, but logically equivalent, application. Appendix A shows the code for a Turing machine. The text below, sketches the implementation of each of the classes. The example, taken from the nearest automata theory text to hand [Manna78], recognizes strings of the form

$$a^n b^n$$

5.1.1. Turing machine The machine itself is an object that creates a tape object and five nodes. After initializing all the objects, loading the tape with an input string and the nodes with their transition tables, it starts up by telling the first node to go, and then awaits an announcement of success or failure from some node down the line. When the announcement arrives, the machine writes the result as output; destroys the nodes it has created; and exits.

5.1.2. Tape

The tape itself is trivial. Input data are stored as a string, and there are a handful of methods to move along the tape and to read or write at the current position. (We've tried to avoid mixed-case disease, which seems endemic to object-oriented programmers, but *Read* requires an initial, upper-case 'R' because *read* is reserved by the shell. *Write* is just following suit.)

The position is just a numeric index into the string, maintained using the POSIX shell's built-in arithmetic facilities.

5.1.3. Node Nodes, too, are objects. When called on, a node reads the current cell on the tape and looks up the entry for the character it reads in a dictionary of transitions that it creates and maintains (another object, described in the next section). Transitions are a triple containing a character to write into the current cell, a direction to move, and a new node to call. The current node writes the prescribed character into the cell, moves either left or right, and then calls on another node (possibly itself) to handle the next cell. If the dictionary reveals that the node has reached a decision to accept or reject the input string, then instead of passing control to a new node, the current node sends the message *accept* or *reject* to the original Turing machine.

The absence of returns and lack of a single, centralized transition table lend a palpably unstructured aura to the process.

Although this implementation uses a colon-separated array to store the three pieces of information associated with each possible input character and teases them apart with **cut**, performance could be improved somewhat by using the shell's prefix- and suffix-shaving operators — *${PARAMETER%%expression}* and friends — to do the parsing without recourse to subprocesses.

5.1.4. Dictionary

A dictionary stores its entries as shell variables whose names are constructed on-the-fly from the words being defined. The command

```
$ send $DICTIONARY define and dumb
```

turns into the assignment

```
def_and=dumb
```

In a better world, the shell might have arrays (indeed, the Korn shell does), but POSIX shells aren't required to have them, and this work-around is good enough for our example.

5.2. ... then getting smaller.

Exploiting the ability of objects to learn new methods at run-time, we can also create a simpler but more tantalizing application. The code shown in figure 9 shows a method that sends itself to another object: a virus.

```
$ cat oneline
# put everything on one line

tr -s ' \n\t' '[ *]'
$ infect()
> {
> send $1 \"$(typeset -f infect|oneline)\"
> }
$ new null X
$ infect X
$ send X typeset -f infect
infect ()
{
  send $1 "$(typeset -f infect |
  oneline)"
}
$ new null Y
$ send X infect Y
$ send Y typeset -f infect
infect ()
{
  send $1 "$(typeset -f infect |
  oneline)"
}
```

Figure 9. A Simple Virus

The script **oneline** is a work-around for two implementation problems. First, with the code shown here any methods learned at runtime must fit on a single line. Second, shell quoting conventions make it annoyingly difficult to fit the **tr** command inside the function itself.

(We confess to having resorted to occasional non-standard shell extensions to avoid other lengthy circumlocutions, particularly *echo -e* which interprets many of the usual shell escape characters like '\n', and *typeset*, All these shell scripts run under **bash**, a publicly available POSIX-conforming shell, and the extensions are those provided by **bash.** Other POSIX shells provide analogous extensions.)

A more sophisticated *infect* would go out and hunt for other objects to infect, A more sophisticated *create* routine would permit multi-line messages.

5.3. Summary

Neither of these applications is particularly long (or useful), but each illustrates the capability and extensibility of this relatively simple system, and the power and flexibility of the shell as a programming language.

As a parting note, we observe that, the two applications can be used in consort: despite its simplicity, limitations, and implementation dependencies, the virus shown above can be used to infect the Turing machine described above to give it a cold in its nodes.

6. *send Paper exit*

This is hardly a complete system. On the other hand, it's so simple that an average undergraduate who's already familiar with UNIX at the shell level should be able to play with objects without first having to wrap his mind around a conventional OOPS like C++ or Smalltalk. A really good undergraduate should be able to enhance it in interesting ways without a course in compiler theory. I've suggested several such enhancements in this paper.

What's more, even though the exercise seems akin to making a sow's ear out of a silk purse, it illustrates that the shell has more power than many people give it credit for. That said, I'll raise anew a question posed by Steve Johnson at the Winter '94 USENIX conference: "Will object-oriented programming replace the shell?" Johnson intended the question to be rhetorical, but I harbor the suspicion that object-oriented shells, and other shells that break from the conventional-programming-language model, are fruitful areas of research [Budd89]. Mashey showed that creating a shell that was a real programming language was exactly the right idea, and that people would use a well-designed shell early and often. [Mashey76]. Given that, I'm surprised that nearly all widely available shells today still use C, ALGOL, or pocket-teller machines as their models.

(A notable exception is Doug Gwyn's "Adventure Shell." Though not a wild success as a programmer's shell, it has spawned, after a trip through a maze of twisty passages, the development of MOOs.)

Personally, I've long wanted to run "the spread-shell" but I haven't any idea what such a thing would do. If you write one, please send it to me.

Acknowledgements

My thanks to Dave Taenzer for patient explanations about objects, Jim Oldroyd and Doug Pintar for humorous discussions about object-oriented applications, Dick Dunn for working me into a lather about them, and Rob Pike for an apposite quote. Thanks also to Mike Karels for fixing FIFOs in BSDI on the spot at a POSIX meeting.

References

[Budd89] Tim Budd. "The design of an object-oriented command interpreter," *Software Practice and Experience*, 19(1), pp. 35-51 (January 1989).

[Brooks75] Fredrick P. Brooks, Jr.. *The Mythical Man-Month: Essays on Software Engineering,* Addison Wesley, Reading, Massachusetts. pp. 93-94 (1975).

[Cargill91] T. A. Cargill, "The Case against multiple inheritance in C++," *Computing Systems*, Vol 4(1), pp. 69-82 (1991).

[Johnson94] Steve Johnson, "Objecting to Objects" USENIX Winter Conference Invited Talks Submitted Notes, San Francisco, January 1994, pp. 41-61.

[King89] Roger King, "My Cat is Object-Oriented," in *Object-Oriented Languages, Applications, and Databases,* W. Kim & F. Lochovsky, eds., Addison-Wesley, New York. (1989).

[Manna78] Zohar Manna, *Mathematical Theory of Computation,* McGraw-Hill, New York. pp. 22-23 (1978).

[Mashey76] J. R. Mashey, "Using a command language as a high-level programming language." *Proceedings of the Second International Conference on Software Engineering*, San Francisco, California. pp. 177-181 (October 1976).

[Sessions93] Roger Sessions, "An Introduction to Object-Oriented Programming and C++" USENIX Summer Conference Invited Talks Submitted Notes, Cincinnati, June 1993, pp. 29-38.

Biography

Jeffrey S. Haemer is an independent consultant in Boulder, Colorado. He works, writes, and speaks on the interrelated topics of internationalization, POSIX, open systems, software portability, and porting. Dr. Haemer has given UNIX programming tutorials since 1988 and is a frequently featured speaker at such well-attended industry forums as Expo Kuwait, USENIX, and the Romanian Open Systems Exposition. He currently serves as the USENIX organizational representative to the POSIX effort.

Appendix A: A Turing Machine

```
# turing machine
#   recognizes a^n b^n

MACHINE=$O_NAME
TAPE=${MACHINE}_T
export MACHINE TAPE

destroy() {
  #debug destroy $*
  _destroy $TAPE s1 s2 s3 s4 s5
  exit
```

```
}

accept() {
  echo ACCEPT!
  destroy
}

reject() {
  echo REJECT!
  destroy
}

new tape $TAPE
new node s1 s2 s3 s4 s5

send $TAPE load aabb

# Hard-wire the nodes.
# It'd be nicer to have this
# load from a file.

send s1 transition a A:right:s2
send s1 transition _ X:right:accept

send s2 transition B B:right:s2
send s2 transition a a:right:s2
send s2 transition b B:left:s3

send s3 transition B B:left:s3
send s3 transition a a:left:s4
send s3 transition A A:right:s5

send s4 transition a a:left:s4
send s4 transition A A:right:s1

send s5 transition B B:right:s5
send s5 transition _ X:right:accept

send -d s1 goto
```

Figure A1. Turing Machine

```
# Turing machine tape

unset S
typeset -i n
typeset -i j

right() {
  let n=n+1
  return 0
}

left() {
  if test $n -le 1
  then
    echo HALT
    return 1
  else
    let n=n-1
    return 0
  fi
}

load() {
  S=$1
  let n=1
  return 0
}

print() {
  echo $S
  let j=n
  while let j=j-1
  do
    echo -n ' '
  done
  echo '^'
  return 0
}

Write() {
  let left_neighbor=n-1
  let right_neighbor=n+1
  left=$(echo $S |
    cut -c -$left_neighbor)
  right=$(echo $S |
    cut -c $right_neighbor-)
  S=${left}$1${right}
  return 0
}

Read() {
  if test $n -gt ${#S}
  then
    echo '_' > $O_OUT
  else
    echo $S |
      cut -c $n >$O_OUT
  fi
  return 0
}
```

Figure A2. Turing Machine Tape

```
# Turing machine node
```

```
XITIONS=dict_$O_NAME
new dict $XITIONS

transition() {
  send $XITIONS define $*
  return 0
}

destroy() {
  send -d $XITIONS destroy $*
  _destroy $*
}

goto() {
  SYMBOL=$(send $TAPE Read)
  ACTION=$(
    send $XITIONS lookup $SYMBOL
  )

  debug $SYMBOL, $ACTION

  if test -z "$ACTION"
  then
    send -d $MACHINE reject
    return 0
  fi

  OUT_CHAR=$(echo $ACTION |
    cut -f 1 -d:)
  DIRECTION=$(echo $ACTION |
    cut -f 2 -d:)
  NEXT_STATE=$(echo $ACTION |
    cut -f 3 -d:)

  if test $NEXT_STATE = "accept"
  then
    send -d $MACHINE accept
    return 0
  fi

  send $TAPE Write $OUT_CHAR
  send $TAPE $DIRECTION

  send -d $NEXT_STATE goto
  return 0
}
```

Figure A3. Turing Machine Node

```
# Small dictionary

dictionary() {
  set | sed -n 's/^def_//p'
  return 0
}

define() {
  eval def_$1="$2"
  return 0
}

lookup() {
  eval echo $"def_$1"
  return 0
}
```

Figure A4. Simple Dictionary