The Problem with Menus

by Jeffreys Copeland and Haemer

Recently, we've experienced a series of problems involving menus. "What?" ask the wags in the audience. "Choosing one from column A is too complicated? Having trouble with the French translations?" Good questions. We tend to order à la carte at Chinese restaurants, and we are both notoriously bad at French, to the extent that one of us isn't even allowed to attempt to speak it at home.

But the question is, given a product with an embedded controller, what's the best way to do menu selections from the front panel? You know the kind of thing we're talking about: the microwave oven that lets you choose 17 different cooking levels from a 10-key pad, or the VCR that has to be programmed from the digits zero to nine, using the plus and minus keys on a remote control. In both cases, the trick is accomplished with hierarchical menus, which allow you to traverse a chain of commands like "VCR setup," "set VCR clock" or "time" to cure the blinking "12:00" on the display.

We had a recent experience that led us to these questions:

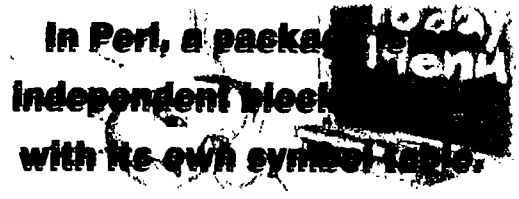
- How can we prototype menus without having to build the whole controller?
- What's a good test harness for menus?
- How can we encapsulate the menu text for easy translation from one country's language to another?
- What's an easy way to translate the prototype into something to feed into the controller?

Jeffrey Copeland (copeland@alumni.caltech.edu) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

If you're paying particular attention, or have had experience with this sort of embedded software before, then you've probably guessed that the menu text we were looking at was rendered in a quasi-assembler language, which requires item counts at every step and very little error or redundancy checking. Certainly, this is a step above programming the menus by twiddling bits directly but having a higher-level programming language–a "little language" in Jon Bentley's phrase–for the menus would be a big step forward.

As has been our habit lately, we wrote some code in Perl to handle the prototype solution to the problem. It's a fairly large program compared with our past examples, so it will take us a couple of months to cover it. But first we should explain some features of Perl that we'll be using for the first time.



Some Background

The code for this problem is explicitly in Version 5 Perl. Many of the object-oriented features that we'll be using aren't available in earlier versions of the language.

That said, we need to review some of the scoping rules and new tricks that we'll be using. Bear with us if you already know these. This is as much for our benefit as yours. An old professor of ours referred to the "Johnson Effect," named after Prof. Johnson, who always managed to learn more than his students when he taught a new course.

First, we'll make extensive use of Perl packages. In Perl, a package is an independent block of code, with its own symbol table. (The object-oriented among you can think of this as an object.) The neat trick here is that the symbol table is just an associative array: For package `Dir`, the symbol table is `%Dir::`. Thus, anywhere in the program, we can refer to the variable `blat` from package `Dir` as `%Dir::blat`. The default package is `main`.

To this we add the notion of a stack frame. Each separate block–surrounded by curly braces–is a new stack frame. So, with

```
#! /opt/local/bin/perl -w

$x = 42;
warn "$x";

{
        local($x) = 99;
        my($y) = 43;
        warn "$x $y";
}

warn "$x $y";
```

we get output of

```
Identifier "main::y" used only once: possible typo
at xx line 12.
42 at xx line 4.
99 43 at xx line 9.
Use of uninitialized value at xx line 12.
42 at xx line 12.
```

Two points to note: First, by specifying `$x` as `local` in the middle block, its scope is restricted–any changes we make do not get reflected outside the block. Second, by tagging the `$y` in that same block as `my`, we make it a variable on the stack in *that* block–it shows up as uninitialized later.

That should be enough to get us started, we'll add more variations as we go along.

What's a Menu?

There are a number of real-world models we can use to visualize the way menus are laid out. The most obvious way is in a tree form. We need to be able to mark leaves as having been selected in several different ways:

- We need to be able to choose from several leaves on the same branch. In the VCR example, do we set the audio to "hi-fi," "normal" or "mix"?
- We need to be able to choose a branch–for example, we invoke the "tuner setup" item.
- We need to be able to provide data–we set the clock or program the VCR to record something later.

After a bit of fumbling about, Haemer had the leap of insight that got us what we needed. After the 14th cup of coffee, he said, "Hey! We're implementing a file system." The leaves are inodes. The data at each menu item is a file. The file status information is not the access permissions but tells us whether an item is set. With that in mind, things begin to fall into place.

Let's start with the Perl package for inodes. Inside the inode, there are attributes and a pointer to the data stored in the inode. We can build functions to change that information, such as: `$ino->chdata`. This is analogous to the UNIX functions *chgrp* and *chown*, for example.

We start with the normal declarations:

```
#! /bin/perl -w

# An "Inode" is a data object that
# looks a little like a UNIX inode.

package Inode;
```

Note: What is called a class in C++, is called a package in Perl. We immediately follow this by providing a method to create a new inode:

```
sub new {
    my($type) = @_;

    my($self) = {};
    $self->{'data'} = [];
    $self->{'ftype'} = "ETYPE";
    bless $self;
}
```

This puts the object on the stack with *my* and initializes it. The type is ETYPE, our error type. We then *bless* the new object, which tags it as a member of this class.

Next, we need a method to change the "file type" and "data" of our inode:

```
sub chftype {
    my($self, $ftype) = @_;

    $self->{'ftype'} = $ftype;
}

sub chdata {
    my($self, $data) = @_;

    $self->{'data'} = $data;
}
```

This is pretty straightforward. We take the arguments and make them local stack variables, and then drop them into the appropriate place in the data structure.

If we have existing data, we may want to add more to the tagged array. In the VCR example, we add an item to the list of programs to record. Again, this is very simple:

```
sub append {
    my($self, @data) = @_;

    push(@{$self->{'data'}}, @data);
}
```

We are left with the relatively complicated procedure to get information out of the *Inode*. By analogy, we call this `ls`. This method always takes an indentation level so that we can get nested listings of our inodes pointing to other inodes.

```
@default_args = ('ftype', 'data');

sub ls {
    my($self, $indent, @args) = @_;
    my($val);
    $indent ||= 0;

    @args = @default_args unless (@args);
    foreach (@args) {
        $val = $self->{$_};
        next unless ($val);
        print "\t" x $indent . "$_: ";
        print "$val\n";
    }
}
```

Notice the useful trick for printing several tabs by using the `x` operator in a `print` statement. Similarly, to get the data out of our inodes, we use the *cat* method. Again, we can provide an indent level:

```
sub cat {
    my($self, $indent) = @_;
    $indent ||= 0;
    $data = $self->{'data'};
    foreach $line (@$data) {
        print "\t" x ($indent), $line;
    }
}
```

Now we test it. One of the claimed advantages of object-oriented programming is that we can carefully encapsulate our classes and test them individually before we use them in larger programs. In this spirit, we bundle some test code into the package:

```
$TEST = 1;
END {
 if ($TEST) {
    print "== Testing package Inode\n\n";
    print "Create a file:\n";
    $ino = new Inode;
    $ino->chftype("regular");
    $ino->ls;

    print "Put something in it\n";
    $ino->chdata(["Hello\n"]);
    print "Contents:\n";
    $ino->cat(1);

    print "Append a line:\n";
    $ino->append("there\n", "world\n");
    $ino->cat(1);
    }
}

1;
```

This gives us the reassuring test output:

```
== Testing package Inode
Create a file:
ftype: regular
data: ARRAY(0xa9958)
Put something in it
Contents:
     Hello
Append a line:
     Hello
     there
     world
```

Simulating UNIX Directories

Next, we add a layer on top of our *Inode* package to simulate UNIX directories.

```
#! /bin/perl -w
use Inode;

package Dir;
@ISA = qw( Inode );

sub new {
     my($type) = @_;

     my($self) = new Inode;
     $self->chftype("directory");
     bless $self;
     $dot = {'ino'=>$self, 'name'=>"."};
     $dotdot = {'ino'=>$self, 'name'=>".."};
     $self->append($dot, $dotdot);
     $self;
}
```

We begin our *Dir* package by setting the ISA array. This tells us where else to look for a method if it's not in the current package. We also provide a creator. This creates a directory by first creating an *Inode* of type directory. It adds pointers to the directory node and its parent: We call these "dot" and "dotdot" to match our directory analog.

Next, we need a query function to tell us if the given argument is a directory or its parent.

```
  sub dotname {
     my($name) = @_;

     if ($name eq '.' || $name eq '..') {
         return 1;
     } else {
         return 0;
     }
}
```

We also need a method to provide the directory listing, as in Listing 1.

Listing 1

```
sub ls { # always takes an indenting level
  my($self, $indent, @args) = @_;
  local(@dflags) = grep(/^-/, @args);
  local(@margs) = grep(!/^-/, @args);

  if (grep(/-d/, @dflags)) {
    Inode::ls($self, $indent, @margs);
  } else {
    foreach $file (@{$self->{'data'}}) {
      local($name) = $file->{'name'};
      print "\t" x $indent ."Name: $name\n";
      $ino = $file->{'ino'};
      if ($ino->{'ftype'} eq 'regular') {
         $ino->ls($indent+1, @margs);
      } else { # a little complicated
         if (dotname($name)) {
            $ino->ls($indent+1, @margs,@dflags, '-d');
         } elsif (grep(/-R/, @dflags)) {
            $ino->ls($indent+1, @margs, @dflags);
         } else {
            $ino->ls($indent+1, @margs, @dflags, '-d');
            }
         }
      }
  }
}
```

This is a recursive subroutine that uses the ls subroutine of the *Inode* method to help produce the listing. Generally, we use the *Inode* routine if we're looking at an inode with the name of "." or ".."; otherwise, we run through the individual items in the inode.

We're out of space for the moment, so we'll continue where we left off next month, and finish covering the Dir package and Menu package built on top of it.

In the meantime, we're expecting simultaneous visits from a passel of relatives, including metalsmiths in both families. We hope your month is as entertaining as ours will be. ▲