The Problem with Menus, Part 3

by Jeffreys Copeland and Haemer

As you'll no doubt recall, we've been discussing menus. Not the kind you see at restaurants but the kind found in embedded controllers, like those in a microwave or VCR.

As we've explained in previous columns, this all came about because a few months ago we found ourselves having to prototype the front-panel menus for a new system, and it was easier to do so by analogy in Perl than using the baroque one-step-removed-from-assembly-code language that the system itself used.

We'll warn you that we still haven't written the compiler from our prototype system to the baroque native one. We're leaving that as an exercise for later.

The critical realization we have made is that menus are really a form of directory tree. To that end, we have written a skeletal file system in Perl using inodes and directories. Now, we're ready to layer our menu system on top of the code for implementing a directory tree. (Refer to the previous two months' columns for the structure of the base `Inode` and `Dir` classes.)

Making Menus

We begin our menu class with the normal declarations as shown in Listing 1.

We make this a subclass of our directory class and declare the beginning of a new `Menu` class. We also tell Perl that we are inheriting methods from the `Dir` class. Note that Perl's `ISA` array contains a list of packages from which this package inherits methods. Copeland keeps

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.*

Listing 1

```
#!/bin/perl -w
# $Id: Menu.pm,v 1.1 1996/03/17 20:36:09 jsh Exp $
# A "Menu" is a directory with added features:
#    markable items
#    a "current" item,
#    traversal left, right, up, and down
```

forgetting this and ends up having to look it up in the manual. Thus,

```
use Dir;

package Menu;
@ISA = qw( Dir );
```

Of course, we need a constructor for the menu item. Remember that we create the object with new and then bless it to declare it as an item generated by the current package. In the case of a new menu item, we make the file type "menu."

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

    my($self) = new Dir;
    $self->chftype("menu");
    bless $self;
}
```

We need to know what menu item we are pointing to in each submenu–that is, what entry we're pointing at in each directory. How can we tell which one it is? We rely on a routine pointname to return the current item. How does pointname know the current item? The current item is the first one in the directory. For example,

```
sub pointname {
    my($self) = @_;

    $data = $self->{'data'};
    @{$data}[0]->{'name'};
}
```

This is probably not the best implementation of the current pointer. Wehave discussed this among ourselves and come up with a few alternative ways of implementing it.

Exercise for the reader: Come up with at least one other way of marking the entry of interest at the current menu level. Because we have done some data-hiding by using the object-oriented package construct of Perl, it should be easier to change the implementation without causing vast destruction to the code.

Of course, we also need to be able to navigate within our menu. Thus, we have left, right, up and down procedures. We begin with left, which creates a circular shift of the entries in the directory to the left, skipping the current directory and its parent, for those entries for which Dir::dotname(...) returns true. For example,

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

    $data = $self->{'data'};
    return if (@{$data} == 2); # just '.' and '..'
    do {
        $sfx = pop(@$data);
        unshift(@$data, $sfx);
    } while
        (Dir::dotname($self->pointname));
}
```

Our routine for shifting right is almost identical, except we shift() and push() instead of pop()ing and unshift()ing to get the data to rotate to the right. For example,

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

    $data = $self->{'data'};
    return if (@{$data} == 2); # just '.' and '..'
    do {
       $pfx = shift(@$data);
       push(@$data, $pfx);
    } while
        (Dir::dotname($self->pointname));
}
```

After that, up is simplicity itself: We just go up a level in the directory tree.

```
sub up {
    my($self) = @_;
    my($up) = $self->namei('..');
    $up;
}
```

down is a little more complicated. If we aren't looking at a menu type node, we exit with an error. However, in the mainline code, we move to the child of this node and

to the next menu item on the right.

```
sub down {
    my($self) = @_;
    $data = $self->{'data'};
    $ino = @{$data}[0]->{'ino'};
    if ($ino->{'ftype'} eq "menu") {
        my($down) = $ino;
        $down->right unless ($self eq $down);
        return($down);
    } else {
        return(-1);
    }
}
```

A fundamental thing we need to do with menu items is mark them. We really need to make that part of our handling of inodes, so we go back to the Inode package to add this functionality. For example,

```
package Inode;

# An extension to Inode to permit "marking" Inodes.
sub chmark {
    my($self, $mark) = @_;
    $mark = '*' unless ($mark);

    if ($self->{'mark'}) {
        unset($self->{'mark'});
    } else {
        $self->{'mark'} = $mark;
    }

}
@default_args = (@default_args, 'mark');

package Menu;
```

Notice that we've just bodily included this package switch in the single source file. We must remember to switch back to the Menu package when we have finished with the Inode additions.

That's about it for the Menu package itself. There just remains the task of testing.

Adding Some Code to Test

We complete the Menu package by adding some code to test it. As we've done before, we just include this test in the same source file, setting the value of $TEST to zero most of the time.

```
$TEST = 1;
END {
    if ($TEST) {
```

We begin by creating a menu, and listing it–remember a menu is a directory and inherits all the directory methods. Therefore, we can use the directory ls function to list the contents of the menu.

```
print "== Testing package Menu\n\n";
print "Create a menu:\n";
$menu = new Menu;
$menu->ls(1,'-d');
```

Next, we add some items to the menu, and list it again.

```
print "Now add some items:\n";

$ino = new Inode;
$ino->chdata(["This is item 1\n"]);
$ino->chftype("regular");
$menu->ln($ino, "Foo");

$ino = new Inode;
$ino->chdata(["This is item 2\n"]);
$ino->chftype("regular");
$menu->ln($ino, "Bar");

$ino = new Inode;
$ino->chdata(["This is item 3\n"]);
$ino->chftype("regular");
$menu->ln($ino, "Mumble");

$menu->ls(1);
```

We follow this by marking an item in the menu and showing the mark.

```
print "Next, mark an item:\n";
$ino = $menu->namei("Foo");
$ino->chmark;

$menu->ls(1);
```

We cycle through the items at the current level of the menu, first widdershins, then deasil.

```
print "Find the point item:\n";
print $menu->pointname, "\n";
print "A problem! Not once we move, though\n";

for ($i = 0; $i < 5; $i++) {
    print "Move left:\n";
    $menu->left;
```

```
    print $menu->pointname, "\n";
}

for ($i = 0; $i < 5; $i++) {
    print "Move right:\n";
    $menu->right;
    print $menu->pointname, "\n";
}
```

We add a submenu tree, to test that functionality.

We begin by adding a regular file in the menu/directory. Then we make it into a submenu. Again, we list the directory tree to convince ourselves that we've made the directory tree correctly.

```
print "Next, add a sub-menu:\n";
$submenu = new Menu;
$ino = new Inode;
$ino->chdata(["This file is in a subdir\n"]);
$ino->chftype("regular");
$submenu->ln($ino, "ZZazz");
$submenu->ls(1);

print "Now make it a submenu,\n";
$menu->ln($submenu, "Sub-menu");
$menu->ls(1, '-R');
```

Last, we check our menu system to ensure that we can go up and down the menu hierarchies.

```
print "find it,\n";
while($menu->pointname ne "Sub-menu") {
      $menu->left;
}

print "Pointname is ", $menu->pointname, "\n";
print "descend it,\n";
$menu = $menu->down;
print "Pointname is now ", $menu->pointname, "\n";
print "and ascend again.\n";
$menu = $menu->up;
print "Pointname is now ", $menu->pointname, "\n";
```

We finish by closing up our big `if()` statement and our `Menu` package.

```
      }
  }

  1;
  __END__
```

To reassure us, the correct test output is as follows:

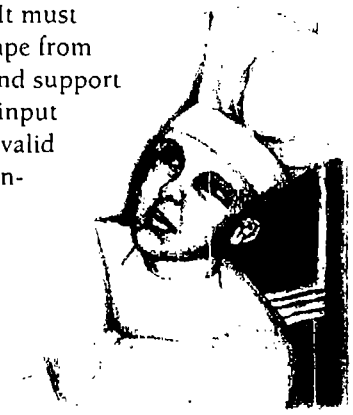
```
== Testing package Menu

Create a menu:
    ftype: menu
    data: ARRAY(0xbc42c)
Now add some items:
    Name: .
          ftype: menu
          data: ARRAY(0xbc42c)
    Name: ..
          ftype: menu
          data: ARRAY(0xbc42c)
    Name: Foo
          ftype: regular
          data: ARRAY(0xdde0c)
    Name: Bar
          ftype: regular
          data: ARRAY(0xddecc)
    Name: Mumble
          ftype: regular
          data: ARRAY(0xddf38)
Next, mark an item:
    Name: .
          ftype: menu
          data: ARRAY(0xbc42c)
    Name: ..
          ftype: menu
          data: ARRAY(0xbc42c)
    Name: Foo
          ftype: regular
          data: ARRAY(0xdde0c)
          mark: *
    Name: Bar
          ftype: regular
          data: ARRAY(0xddecc)
    Name: Mumble
          ftype: regular
          data: ARRAY(0xddf38)
Find the point item:
A problem! Not once we move, though
Move left:
Mumble
Move left:
Bar
Move left:
Foo
Move left:
Mumble
Move left:
Bar
Move right:
Mumble
Move right:
Foo
```

```
Move right:
Bar
Move right:
Mumble
Move right:
Foo
Next, add a sub-menu:
    Name: .
          ftype: menu
          data: ARRAY(0xddf98)
    Name: ..
          ftype: menu
          data: ARRAY(0xddf98)
    Name: ZZazz
          ftype: regular
          data: ARRAY(0xe29dc)
Now make it a submenu,
    Name: Foo
          ftype: regular
          data: ARRAY(0xdde0c)
          mark: *
    Name: Bar
          ftype: regular
          data: ARRAY(0xddecc)
    Name: Mumble
          ftype: regular
          data: ARRAY(0xddf38)
    Name: .
          ftype: menu
          data: ARRAY(0xbc42c)
    Name: ..
          ftype: menu
          data: ARRAY(0xbc42c)
    Name: Sub-menu
          Name: .
                ftype: menu
                data: ARRAY(0xddf98)
          Name: ZZazz
                ftype: regular
                data: ARRAY(0xe29dc)
          Name: ..
                ftype: menu
                data: ARRAY(0xbc42c)
find it,
Pointname is Sub-menu
descend it,
Pointname is now ZZazz
and ascend again.
Pointname is now Sub-menu
```

Finishing Up Our Work

While we have completed the basic menu software–meaning that we have a way to test the menu trees we develop–we still don't have a simple language for *laying out* the menus. Furthermore, as we mentioned earlier, we still haven't built a compiler from our menu package to the embedded one that we were originally working with.

The second of these is our problem, but the first one is an exercise for the reader: Construct a simple language to describe an embedded menu system. It must allow submenus, the marking of an item–in the VCR example, I want the counter displayed on-screen–and allow selecting from a list of items. For example, whether I'm connected to a cable or an antenna. It must also allow input values–tape from channel 4, for example–and support range checking for those input values–27 o'clock is an invalid time and 18 hours is an unlikely program duration.



We're just about done with our discussion on menus, next month we'll be leaving Perl behind and taking a look at code in C++. ▲