26 – Top-Down Design

As programs get larger and more complex, they become more difficult to design, code, and maintain. As with any large project, it is often a good idea to break large, complex tasks into a series of small, simple tasks. Let's imagine that we are trying to describe a common, everyday task, going to the market to buy food, to a person from Mars. We might describe the overall process as the following series of steps:

1. Get in car.
2. Drive to market.
3. Park car.
4. Enter market.
5. Purchase food.
6. Return to car.
7. Drive home.
8. Park car.
9. Enter house.

However, a person from Mars is likely to need more detail. We could further break down the subtask "Park car" into this series of steps:

1. Find parking space.
2. Drive car into space.
3. Turn off motor.
4. Set parking brake.
5. Exit car.
6. Lock car.

The "Turn off motor" subtask could further be broken down into steps including "Turn off ignition," "Remove ignition key," and so on, until every step of the entire process of going to the market has been fully defined.

This process of identifying the top-level steps and developing increasingly detailed views of those steps is called *top-down design*. This technique allows us to break large complex tasks into many small, simple tasks. Top-down design is a common method of designing programs and one that is well suited to shell programming in particular.

In this chapter, we will use top-down design to further develop our report-generator script.

Shell Functions

Our script currently performs the following steps to generate the HTML document:

1. Open page.
2. Open page header.
3. Set page title.
4. Close page header.
5. Open page body.
6. Output page heading.
7. Output timestamp.
8. Close page body.
9. Close page.

For our next stage of development, we will add some tasks between steps 7 and 8. These will includethe following:

- System uptime and load. This is the amount of time since the last shutdown or reboot and the average number of tasks currently running on the processor over several time intervals.
- Disk space. This is the overall use of space on the system’s storage devices.
- Home space. This is the amount of storage space being used by each user.

If we had a command for each of these tasks, we could add them to our script simply through command substitution.

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<html>
```

```
        <head>
                <title>$TITLE</title>
        </head>
        <body>
                <h1>$TITLE</h1>
                <p>$TIMESTAMP</p>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
        </body>
</html>
_EOF_
```

We could create these additional commands in two ways. We could write three separate scripts and place them in a directory listed in our PATH, or we could embed the scripts within our program as *shell functions*. As we have mentioned, shell functions are “mini-scripts” that are located inside other scripts and can act as autonomous programs. Shell functions have two syntactic forms. First, here is the more formal form:

```
function name {
      commands
      return
}
```

Here is a simpler (and generally preferred) form:

```
name () {
      commands
      return
}
```

where *name* is the name of the function and *commands* is a series of commands contained within the function. Both forms are equivalent and may be used interchangeably. The following is a script that demonstrates the use of a shell function:

```
#!/bin/bash

# Shell function demo

function step2 {
     echo "Step 2"
     return
}

```

```
10  # Main program starts here
11
12  echo "Step 1"
13  step2
14  echo "Step 3"
```

As the shell reads the script, it passes over lines 1 through 11 because those lines consist of comments and the function definition. Execution begins at line 12, with an `echo` command. Line 13 *calls* the shell function `step2` and the shell executes the function just as it would any other command. Program control then moves to line 6, and the second `echo` command is executed. Line 7 is executed next. Its `return` command terminates the function and returns control to the program at the line following the function call (line 14), and the final `echo` command is executed. Note that for function calls to be recognized as shell functions and not interpreted as the names of external programs, shell function definitions must appear in the script before they are called.

We'll add minimal shell function definitions to our script, shown here:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    return
}

report_disk_space () {
    return
}

report_home_space () {
    return
}

cat << _EOF_
<html>
    <head>
        <title>$TITLE</title>
```

```
    </head>
    <body>
        <h1>$TITLE</h1>
        <p>$TIMESTAMP</p>
        $(report_uptime)
        $(report_disk_space)
        $(report_home_space)
    </body>
</html>
_EOF_
```

Shell function names follow the same rules as variables. A function must contain at least one command. The `return` command (which is optional) satisfies the requirement.

Local Variables

In the scripts we have written so far, all the variables (including constants) have been *global variables*. Global variables maintain their existence throughout the program. This is fine for many things, but it can sometimes complicate the use of shell functions. Inside shell functions, it is often desirable to have *local variables*. Local variables are only accessible within the shell function in which they are defined and cease to exist once the shell function terminates.

Having local variables allows the programmer to use variables with names that may already exist, either in the script globally or in other shell functions, without having to worry about potential name conflicts.

Here is an example script that demonstrates how local variables are defined and used:

```
#!/bin/bash

# local-vars: script to demonstrate local variables

foo=0   # global variable foo

funct_1 () {

    local foo # variable foo local to funct_1

    foo=1
    echo "funct_1: foo = $foo"
}
```

```
funct_2 () {

    local foo # variable foo local to funct_2

    foo=2
    echo "funct_2: foo = $foo"
}

echo "global:  foo = $foo"
funct_1
echo "global:  foo = $foo"
funct_2
echo "global:  foo = $foo"
```

As we can see, local variables are defined by preceding the variable name with the word `local`. This creates a variable that is local to the shell function in which it is defined. Once outside the shell function, the variable no longer exists. When we run this script, we see these results:

```
[me@linuxbox ~]$ local-vars
global:  foo = 0
funct_1: foo = 1
global:  foo = 0
funct_2: foo = 2
global:  foo = 0
```

We see that the assignment of values to the local variable `foo` within both shell functions has no effect on the value of `foo` defined outside the functions.

This feature allows shell functions to be written so that they remain independent of each other and of the script in which they appear. This is valuable, because it helps prevent one part of a program from interfering with another. It also allows shell functions to be written so that they can be portable. That is, they may be cut and pasted from script to script, as needed.

Keep Scripts Running

While developing our program, it is useful to keep the program in a runnable state. By doing this, and testing frequently, we can detect errors early in the development process. This will make debugging problems much easier. For example, if we run the program, make a small change, then run the program again and find a problem, it's likely that the most recent change is the source of the problem. By adding the empty functions, called *stubs* in programmer-speak, we can verify the logical flow of our program at an early stage. When constructing a stub, it's a good idea to include something that provides feedback to the programmer, which shows the logical flow is being carried out. If we look at the output of our script now:

```
[me@linuxbox ~]$ sys_info_page
<html>
    <head>
         <title>System Information Report For twin2</title>
    </head>
    <body>
         <h1>System Information Report For linuxbox</h1>
         <p>Generated 03/19/2009 04:02:10 PM EDT, by me</p>



    </body>
</html>
```

we see that there are some blank lines in our output after the timestamp, but we can't be sure of the cause. If we change the functions to include some feedback:

```
report_uptime () {
        echo "Function report_uptime executed."
        return
}

report_disk_space () {
        echo "Function report_disk_space executed."
        return
}

report_home_space () {
        echo "Function report_home_space executed."
        return
}
```

and run the script again:

```
[me@linuxbox ~]$ sys_info_page
```

```
<html>
    <head>
        <title>System Information Report For linuxbox</title>
    </head>
    <body>
        <h1>System Information Report For linuxbox</h1>
        <p>Generated 03/20/2009 05:17:26 AM EDT, by me</p>
        Function report_uptime executed.
        Function report_disk_space executed.
        Function report_home_space executed.
    </body>
</html>
```

we now see that, in fact, our three functions are being executed.

With our function framework in place and working, it's time to flesh out some of the function code. First, here's the `report_uptime` function:

```
report_uptime () {
    cat <<- _EOF_
        <h2>System Uptime</h2>
        <pre>$(uptime)</pre>
        _EOF_
    return
}
```

It's pretty straightforward. We use a here document to output a section header and the output of the `uptime` command, surrounded by `<pre>` tags to preserve the formatting of the command. The `report_disk_space` function is similar.

```
report_disk_space () {
    cat <<- _EOF_
        <h2>Disk Space Utilization</h2>
        <pre>$(df -h)</pre>
        _EOF_
    return
}
```

This function uses the `df -h` command to determine the amount of disk space. Lastly, we'll build the `report_home_space` function.

```
report_home_space () {
    cat <<- _EOF_
         <h2>Home Space Utilization</h2>
         <pre>$(du -sh /home/*)</pre>
         _EOF_
    return
}
```

We use the `du` command with the `-sh` options to perform this task. This, however, is not a complete solution to the problem. While it will work on some systems (Ubuntu, for example), it will not work on others. The reason is that many systems set the permissions of home directories to prevent them from being world-readable, which is a reasonable security measure. On these systems, the `report_home_space` function, as written, will work only if our script is run with superuser privileges. A better solution would be to have the script adjust its behavior according to the privileges of the user. We will take this up in the next chapter.

Shell Functions In Your `.bashrc` File

Shell functions make excellent replacements for aliases, and are actually the preferred method of creating small commands for personal use. Aliases are limited in the kind of commands and shell features they support, whereas shell functions allow anything that can be scripted. For example, if we liked the `report_disk_space` shell function that we developed for our script, we could create a similar function named `ds` for our `.bashrc` file:

```
ds () {
   echo “Disk Space Utilization For $HOSTNAME”
   df -h
}
```

Summing Up

In this chapter, we have introduced a common method of program design called top-down design, and we saw how shell functions are used to build the stepwise refinement that it requires. We also saw how local variables can be used to make shell functions independent from one another and from the program in which they are placed. This makes it possible for shell functions to be written in a portable manner and to be *reusable* by allowing them to be placed in multiple programs; this is a great time saver.

Further Reading

- The Wikipedia has many articles on software design philosophy. Here are a couple of good ones:
  http://en.wikipedia.org/wiki/Top-down_design
  http://en.wikipedia.org/wiki/Subroutines