26

Top-Down Design
Diseño de arriba hacia abajo

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 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:
A medida que los programas se hacen más grandes y complejos, se vuelven más difíciles de diseñar, codificar y mantener. Al igual que con cualquier proyecto grande, a menudo es una buena idea dividir las tareas grandes y complejas en una serie de tareas pequeñas y simples. Imaginemos que estamos tratando de describir una tarea común y cotidiana, ir al mercado a comprar comida, a una persona de Marte. Podríamos describir el proceso general como la siguiente serie de pasos:

1. Get in car.
   Sube al coche.
2. Drive to market.
   Conducir al mercado.
3. Park car.
   Aparcar
4. Enter market.
   Entrar en el mercado
5. Purchase food.
   comprar comida
6. Return to car.
   Volver al coche
7. Drive home.
   Conducir a casa
8. Park car.
   Aparcar el coche
9. Enter house.
   Entrar en la casa

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:
Sin embargo, es probable que una persona de Marte necesite más detalles. Podríamos desglosar aún más la subtarea "Estacionar el automóvil" en esta serie de pasos:

1. Find parking space.
   Encuentra plaza de aparcamiento.
2. Drive car into space.
   Conduce el coche al espacio.
3. Turn off motor.
   Apague el motor.
4. Set parking brake.
   Ponga el freno de mano.
5. Exit car.
   Salir del coche
6. Lock car.
   Bloquear el coche.

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.
La subtarea "Apagar el motor" podría dividirse en pasos que incluyen "Apagar el encendido", "Quitar la llave de encendido", etc., hasta que cada paso del proceso completo de salida al mercado se haya definido completamente.

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.
Este proceso de identificar los pasos de nivel superior y desarrollar vistas cada vez más detalladas de esos pasos se denomina diseño de arriba hacia abajo. Esta técnica nos permite dividir grandes tareas complejas en muchas tareas pequeñas y simples. El diseño de arriba hacia abajo es un método común de diseño de programas y uno que se adapta bien a la programación de shell en particular

In this chapter, we will use top-down design to further develop our report-generator script.
En este capítulo, utilizaremos el diseño descendente para desarrollar aún más nuestro script generador de informes.

Shell Functions Our script currently performs the following steps to generate the HTML document:
Funciones de Shell Nuestro script actualmente realiza los siguientes pasos para generar el documento HTML:

1. Open page.
   abrir página
2. Open page header.
   abrir página de cabeza
3. Set page title.
   establecer titulo de página
4. Close page header.
   Cerrar página de cabeza
5. Open page body.
   abrir cuerpo página

6. Output page heading.
   Encabezado de la página de salida.
7. Output timestamp.
   Marca de tiempo de salida.
8. Close page body.
   Cerrar el cuerpo de la página.
9. Close page.
   Cerrar página.

For our next stage of development, we will add some tasks between
Para nuestra próxima etapa de desarrollo, agregaremos algunas tareas entre

steps 7 and 8. These will include the following:
pasos 7 y 8. Estos incluirán lo siguiente:

System uptime and load. This is the amount of time since the last shut- down or reboot and the average number of tasks currently running on the processor over several time intervals.
Tiempo de actividad y carga del sistema. Esta es la cantidad de tiempo desde el último apagado o reinicio y el número promedio de tareas que se están ejecutando actualmente en el procesador durante varios intervalos de tiempo.

Disk space. This is the overall use of space on the system's storage devices.
Espacio del disco. Este es el uso general del espacio en los dispositivos de almacenamiento del sistema.

Home space. This is the amount of storage space being used by each user.
Espacio para el hogar. Esta es la cantidad de espacio de almacenamiento que utiliza cada usuario.

If we had a command for each of these tasks, we could add them to our script simply through command substitution.
Si tuviéramos un comando para cada una de estas tareas, podríamos agregarlas a nuestro script simplemente mediante la sustitución de comandos.

```
#!/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:
Podríamos crear estos comandos adicionales de dos formas. Podríamos escribir tres scripts separados y colocarlos en un directorio listado en nuestro PATH, o podríamos incrustar los scripts dentro de nuestro programa como funciones de shell. Como hemos mencionado, las funciones de shell son "mini-scripts" que se encuentran dentro de otros scripts y pueden actuar como programas autónomos. Las funciones de shell tienen dos formas sintácticas. Primero, aquí está la forma más formal:

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

Here is the simpler (and generally preferred) form:
Aquí está la forma más simple (y generalmente preferida):

```
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 ashell function:
donde nombre es el nombre de la función y comandos es una serie de comandos contenidos dentro de la función. Ambas formas son equivalentes y pueden usarse indistintamente. El siguiente es un script que demuestra el uso de una función shell:

```
1 #!/bin/bash
2
3 # Shell function demo
```

```
4
5 function step2 {
6   echo "Step 2"
7   return
8 }
9
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.
A medida que el shell lee el script, pasa por las líneas 1 a 11 porque esas líneas constan de comentarios y la definición de la función. La ejecución comienza en la línea 12, con un comando de eco. La línea 13 llama a la función de shell step2, y el shell ejecuta la función como lo haría con cualquier otro comando. El control del programa luego se mueve a la línea 6 y se ejecuta el segundo comando de eco. A continuación se ejecuta la línea 7. Su comando de retorno termina la función y devuelve el control al programa en la línea que sigue a la llamada a la función (línea 14), y se ejecuta el comando de eco final. Tenga en cuenta que para que las llamadas a funciones se reconozcan como funciones de shell y no se interpreten como nombres de programas externos, las definiciones de funciones de shell deben aparecer en el script antes de que se llamen.

We'll add minimal shell function definitions to our script, shown here:
Agregaremos definiciones mínimas de funciones de shell a nuestro script, que se muestran aquí:

```
#!/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.
Los nombres de las funciones de shell siguen las mismas reglas que las variables. Una función debe contener al menos un comando. El comando de retorno (que es opcional) satisface el requisito.

Local Variables
Variables locales

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 accessible only within the shell function in which they are defined and cease to exist once the shell function terminates.
En los scripts que hemos escrito hasta ahora, todas las variables (incluidas las constantes) han sido variables globales. Las variables globales mantienen su existencia durante todo el programa. Esto está bien para muchas cosas, pero a veces puede complicar el uso de funciones de shell. Dentro de las funciones de shell, a menudo es deseable tener variables locales. Las variables locales son accesibles solo dentro de la función de shell en la que están definidas y dejan de existir una vez que termina la función de shell.

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.
Tener variables locales le permite al programador usar variables con nombres que pueden existir, ya sea en el script globalmente o en otras funciones de shell, sin tener que preocuparse por posibles conflictos de nombres.

Here is an example script that demonstrates how local variables are defined and used:
A continuación, se muestra un script de ejemplo que demuestra cómo se definen y utilizan las variables locales:

```
#!/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:
Como podemos ver, las variables locales se definen precediendo el nombre de la variable con la palabra local. Esto crea una variable que es local a la función de shell en la que está definida. Una vez fuera de la función de shell, la variable ya no existe. Cuando ejecutamos este script, vemos estos resultados:

```
[~]$ 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.
Vemos que la asignación de valores a la variable local foo dentro de ambas funciones de shell no tiene ningún efecto sobre el valor de foo definido fuera de las funciones.

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.
Esta característica permite escribir funciones de shell para que permanezcan independientes entre sí y del script en el que aparecen. Esto es valioso porque ayuda a evitar que una parte de un programa interfiera con otra.

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.
También permite escribir funciones de shell para que puedan ser portátiles. Es decir, se pueden cortar y pegar de un guión a otro, según sea necesario.

Keep Scripts Running
Mantenga las secuencias de comandos en ejecución

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:
Mientras desarrolla nuestro programa, es útil mantener el programa en un estado ejecutable. Al hacer esto y probar con frecuencia, podemos detectar errores al principio del proceso de desarrollo. Esto hará que los problemas de depuración sean mucho más fáciles. Por ejemplo, si ejecutamos el programa, hacemos un pequeño cambio, luego ejecutamos el programa nuevamente y encontramos un problema, es probable que el cambio más reciente sea la fuente del problema. Al agregar las funciones vacías, llamadas stubs en el lenguaje del programador, podemos verificar el flujo lógico de nuestro programa en una etapa temprana. Al construir un stub, es una buena idea incluir algo que proporcione retroalimentación al programador, que muestre que se está llevando a cabo el flujo lógico. Si miramos la salida de nuestro script ahora:

```
[~]$ sys_info_page
<html>
    <head>
        <title>System Information Report For twin2</title>
    </head>
    <body>
        <h1>System Information Report For linuxbox</h1>
        <p>Generated 03/19/2018 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:
vemos que hay algunas líneas en blanco en nuestra salida después de la marca de tiempo, pero no podemos estar seguros de la causa. Si cambiamos las funciones para incluir algunos comentarios:

```
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:
y vuelva a ejecutar el script:

```
[~]$ sys_info_page

<html>
    <head>
        <title>System Information Report For linuxbox</title>
    </head>
    <body>
        <h1>System Information Report For linuxbox</h1>
        <p>Generated 03/20/2018 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.
ahora vemos que, de hecho, nuestras tres funciones se están ejecutando.

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:
Con nuestro marco de funciones en su lugar y en funcionamiento, es hora de desarrollar parte del código de funciones. Primero, aquí está la función report_uptime:

```
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.
Es bastante sencillo. Usamos un documento aquí para generar un encabezado de sección y el resultado del comando de tiempo de actividad, rodeado por etiquetas <pre> para preservar el formato del comando. La función **report_disk_space** es 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.
Esta función utiliza el comando df -h para determinar la cantidad de espacio en disco. Por último, crearemos la función **report_home_space**.

```
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.
Usamos el comando du con las opciones -sh para realizar esta tarea. Sin embargo, esto no es una solución completa al problema. Si bien funcionará en algunos sistemas (Ubuntu, por ejemplo), no funcionará en otros. La razón es que muchos sistemas establecen los permisos de los directorios personales para evitar que se puedan leer en todo el mundo, lo cual es una medida de seguridad razonable. En estos sistemas, la función report_home_space, tal como está escrita, funcionará solo si nuestro script se ejecuta con privilegios de superusuario. Una mejor solución sería hacer que el script ajuste su comportamiento de acuerdo con los privilegios del usuario. Trataremos de esto en el próximo capítulo.

Shell Functions in Your .bashrc File
Funciones de shell en su archivo .bashrc

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:
Las funciones de shell son excelentes reemplazos para los alias y en realidad son el método preferido para crear pequeños comandos para uso personal. Los alias están limitados en el tipo de comandos y características de shell que admiten, mientras que las funciones de shell permiten cualquier cosa que pueda ser programada. Por ejemplo, si nos gustó la función de shell report_disk_space que desarrollamos para nuestro script, podríamos crear una función similar llamada ds para nuestro archivo .bashrc:

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

Summing Up
Resumen

In this chapter, we 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.
En este capítulo, presentamos un método común de diseño de programas llamado diseño de arriba hacia abajo, y vimos cómo se usan las funciones de shell para construir el refinamiento paso a paso que requiere. También vimos cómo se pueden usar las variables locales para hacer que las funciones de shell sean independientes entre sí y del programa en el que están ubicadas. Esto hace posible que las funciones de shell se escriban de manera portátil y sean reutilizables al permitir que se coloquen en múltiples programas; esto es un gran ahorro de tiempo.
