UnixShellPatterns

by JimCoplien, AlanRobertson?, and GreggWonderly + MichaelLindner

Lucent Technologies, Bell Laboratories Innovations

VISITORS: We welcome your comments. Please add them at the bottom of this page. Please identify yourself as the contributor. Thanks!

These patterns have been submitted to PLoP96 (except for **CleanupTempFiles?**, which was added by Michael Lindner).

Introduction

Shell has emerged as a family of programming languages for the UNIX Operating System in the same sense that JCL emerged as a programming language for mainframe job control in the 1960s. It is a family of languages because we find variant syntaxes and capabilities in the Bourne shell, the Korn shell, the C shell, and others. Most of the patterns described here apply equally well to all these variants.

These patterns do not form a full pattern language nor are they a comprehensive collection of patterns for the "architecture" of a Shell program. They cover frequent Shell programming problems that often are resolved only with expert insight. These patterns emerged from an iterative editing process served by a shared writable web page. The technology we used employs naming conventions to distinguish links between pages. Names like UsePasteInsteadOfSed? and SubShell? are examples of such links. In this web environment--popularized by Ward Cunningham's WikiWikiWeb (http://c2.com/wiki), such naming has become a cultural convention that is retained here in the printed form. Also, if there are other co-authors out there who failed to sign their names to these patterns while posted in public forum, we'd like to thank them as Unknown Authors. Many thanks to Stephen Hill for comments, and to RussellCorfman, who served as the PLoP/96 shepherd for these patterns.

Glossary

SetUid?: A process mode that allows a process, started by one user, to run under the auspices and permissions of another user. SetUid? permissions appear in the program file's mode.

SubShell?: A Shell running in a process that is a child process of some parent process, also running a shell, often arranged so the two Shells can communicate through a pipe.

ShellVariables?: Identifiers known to Shell that can be bound to values such as strings and integers. Their value can be referenced with a leading dollar sign; e.g. **$DATE**.

StandardOutput?, StandardError?, StandardInput?, stdout,' 'stderr, stdin: StandardOutput? is an abstract resource used by programs to print results where they will most likely be read; for an interactive user, this is the user terminal; for a batch job, it would correspond to a line printer or file. It is also called stdout. Other resources include StandardError? (stderr), the standard resource for error output, usually tied to StandardError?, and StandardInput? (stdin), the standard resource for user input (usually the keyboard, but it can be a file).

Daemon: a program that is usually not interactive and that usually runs for a long time. For example, **gtty** daemons are long-lived processes that wait for users to log in on a line.

The Pattern Intents

TestingAgainstTheNullString?: how to test for the null string while avoiding aliasing problems caused by Shell command interpolation rules

CallByReferenceSubshellVariables?: how to effectively export values from a SubShell? to its parent shell.

UsingXargsForRepetition?: how to efficiently unfold execution loops for many forms of Shell commands like **grep, ls**, etc.

UseIFSInsteadOfCut: an expressive and efficient way to extract delimited data fields from input text

UsePasteInsteadOfSed?: avoiding paste limitations in sed that cause truncated lines when generating long lines

UseExecToAvoidSubshell?: the Shell equivalent of tail recursion of FORTRAN chaining

SaveFileDescriptorsToAvoidSubshell?: using Shell input/output capabilities that reuse file descriptors and their bindings to avoid costly Shell spawns

IsolateDaemonsFromTerminals?: avoiding naive problems with hidden input/output assignments in programs designed to run as detached background processes

UniqueDelimiters?: make sure that the contents of an archive file don't alias the EOI delimiter used in scripts.

CleanupTempFiles?: Making sure temp files do not get left behind when your shell program exits -- added by Michael Lindner

---

TestingAgainstTheNullString?

The null string is an exceptional and sometimes problematic ShellVariable? value.

Consider the following code:

```
if [ $a = "" ]; then echo "null string"; fi
```

If **$a** is indeed the null string, the Shell won't be able to recognize and interpret the command, since **test** will be called with syntactially invalid arguments. The command would appear to the shell like this if **$a** were the null string:

```
if [ = "" ]; then echo "null string"; fi
```

We can fix this with quotes:

```
if [ "$a" = "" ]; then echo "null string"; fi
```

but if the contents of the string **$a** is unconstrained, and it contains something like **!=*' or '*-a** or some other option to the test operator, it will cause the shell to issue a syntax error, and not perform the evaluation the programmer desired.

Therefore:

Use double aliasing to test ShellVariable? values:

```
if [ x$a = x ]; then echo "null string"; fi
if [ x$a = xBamboozled ]; then echo "Bamboozled"; fi
case x$a in
   xwpasskey) ... ;;
   x) ... handle null string;;
esac
```

Security Note:

This type of care is especially important when writing shell scripts that are intended to be executed SetUid?, or to be executed by a networking daemon, or CGI-BIN script, as a hostile user can obtain unintended privileges through crafty probes into carelessly-written shell scripts.

CallByReferenceSubshellVariables?

The encapsulation of SubShell? processes makes it difficult for them to communicate through ShellVariables? with the rest of the system. Consider the following Shell code:

```
abortf=0
$CGIDIR/post-undecode | (
  security=""
  while read x; do
    case $x in
      #
      # wpasskey is for attempts
      # to write pre-existing pages
      #
      "wpasskey") passkey=`line`;
        if [ "$passkey" ]; then
          security="WriteProtected?=$passkey";
        fi;;
      "writesecure") writesecure=`line`;;
      . . .
```

```
      "Abort_save") line >/dev/null;
                    abortf=1;;
      "text")        break;;
    esac
  done
  if [ "$abortf" = "1" ]; then
    echo '<html><head>'
    echo'<title>'$term
    echo 'Save Canceled</title></head>'
    . . .
    exit
  fi
)
if [ "$abortf" = "1" ]; then
  ... take error action
fi
```

The last *if* test can fail, because the changes made to abortf inside the SubShell? (delimited by parenthesis) are lost when the SubShell? exits. Some Shells, like Korn shell (ksh), handle this well, but one can't depend on the ubiquitousness of ksh (pity). One could communicate such conditions through StandardOutput?, but we'd rather use StandardOutput? for stream data, passing more voluminous data. We could also use AuxillaryFiles?, but those have problems with efficiency, reentrancy, cleanup, etc.

Therefore: Communicate error conditions from SubShells? to their parents using return error codes:

```
$CGIDIR/post-undecode | (
  abortf=0  # right scope
  security=""
  while read x; do
    case $x in
      #
      # wpasskey is for attempts to write
      # pre-existing pages
      #
      "wpasskey")    passkey=`line`;
        if [ "$passkey" ]; then
          security="WriteProtected?=$passkey";
        fi;;
      "writesecure") writesecure=`line`;;
      . . .
      "Abort_save")   line >/dev/null; abortf=1;;
      "text")        break;;
    esac
  done
  if [ "$abortf" = "1" ]; then
    echo '<html><head>'
    echo '<title>'$term 'Save Canceled</title></head>'
    . . .
    exit 65
  fi
)
if [ "$?" = "65" ]; then
  ... take error action
fi
```

UsingXargsForRepetition?

Shell efficiency is often greatly impeded by **fork/exec** overhead. Consider the following code:

```
while
  read filename
do
  ls -ld $filename
done
```

The *while* loop makes it possible to invoke the same Shell command several times with the same general parameters, conveniently parameterized as ShellVariables?. In this code, a **fork*' and '*exec** system call is performed for every file name in standard input. For loops with very simple commands, like **ls*' in this case, the '*fork/exec** overhead is the vast majority of the run time. It would be nice if **ls** could handle multiple files in a single invocation:

```
ls a b c
```

is the same as

```
ls a; ls b; ls c
```

a property true of many Shell commands (**ls, cat, echo,** etc.). But there is no syntactically convenient way to bunch up parameters coming through a pipeline. We could "bunch" parameters ourselves:

```
while
   read filename1 filename2 filename3 filename4
do
   ls -ld $filename1 $filename2 $filename3 $filename4
done
```

but it's difficult to handle the leftovers when the number of data items is not an even multiple of the number in the bunch (4 in this case).

Therefore: Use xargs to "vectorize" loops. The following line replaces the loop above.

```
xargs ls -ld
```

Efficient shell scripts often resemble APL, they consist of vector operations.

UseIFSInsteadOfCut

The cost of using the **cut(1)** program for several consecutive operations is high. In UseXargsForRepetition?, each iteration parameter was separated from the next by a newline. Unfortunately, the **read** Shell utility cuts on whole line boundaries. We can input lines with **read** and then split them using **cut**, separating fields by their delimiters:

```
while read line
do
   f=`cut -f1 -d';'`
   d=`cut -f2 -d';'`
   c=`cut -f4 -d';'`
   ...
done
```

However, this spawns a Shell process for each parameter on each line, and most of the real time will be expended in process creation. It violates intuition to create a process to pick off each parameter. Yet we must create a new process to use **cut**.

Therefore, use the IFS variable and **read/line/set**. Use this instead of the above code:

```
IFS=';'
while read f d junk1 c junk2
do
   ...
done
```

The use of **junk2** is important to make sure that fields following 'c' are not put into the value of 'c'.

UsePasteInsteadOfSed?

When joining multiple lines of text into a single line of delimiter-separated fields, the line length can silently violate Shell buffer sizes. The construct:

```
echo `grep -l foo *`
```

causes Shell to strip the newlines from the SubShell? executed within the grave accents, replacing them with spaces; **echo** posts the result to StandardOutput?. One can then use **sed** to transform the spaces into a delimiter of choice:

```
echo `grep -l foo *` | sed 's/ /:/g'
```

However, this can lead to lines of unbounded length, which can break Shell buffer limitations. The command will create truncated output when the buffer exceeds the line buffer length in **sed(1)**.

Therefore: when joining multiple lines of text into a single line with a one character separator, the **paste(1)** program should be used instead of **sed(1)** if the length of the line is unbounded. Use:

```
grep -l foo * | paste -sd':' -
```

to avoid truncated lines.

UseExecToAvoidSubshell?

It can be costly for the parent process to wait while a child process completes its work, particularly if the parent is already done. Shell processes often progress in a sequence of programs, one working with the output of the proceeding one. Some of these programs started as monolithic Shell scripts that were broken into multiple scripts for maintainability. If there is a large volume of serial data, it makes sense to use PipesAndFilters?. In other cases, the environment of the first scripts provides input for later scripts. Since earlier scripts defer to later scripts to finish the work, they do no work and produce no output after starting their successor scripts. These scripts form a structure of FORTRAN chaining, or tail recursion. As with tail recursion, each step in the chain remains until its child process terminates--by default, Shells **wait(2)** for the deaths of all their children. This creates an unnecessarily large number of processes. Yet to fold the code of the child process back into the parent process would re-introduce the monolithic structure.

Therefore: Take a cue from FORTRAN chaining, and reuse the same process every time, just as chaining re-uses the same memory. You can do this using exec. So for some script **specialJoin:**

```
cat $DATABASE | (
while expr $NRECORDS \> 0; do
   . . . .
done
)
```

instead of writing:

```
export DATABASE=database_A
export NRECORDS=20
specialJoin
```

do this in the first script instead:

```
export DATABASE=database_A
export NRECORDS=20
exec specialJoin
```

SaveFileDescriptorsToAvoidSubshell?

The encapsulation of SubShell? processes makes it difficult for them to communicate through ShellVariables? with the rest of the system. Consider the following fragment of Shell code:

```
while
  read line
do
  lastline=$line;
done < $TMPFILE
echo "Last line is $lastline"
read stdinline
echo "Next Line of stdin is $stdinline"
```

In this case, the variable *$lastline* is lost through the creation of a SubShell? to perform the I/O redirection for the while loop. Yet, at the same time, the conditions associated with UseExecToAvoidSubShell? are not met, since the **read stdinline** references standard input after the I/O redirection had taken place.

Therefore: Save file descriptors to avoid a SubShell?. The following code saves the stdin file descriptor in file descriptor 3, and restores it after the *exec*.

```
exec 3<&0 <$TMPFILE
while
  read line
do
  lastline=$line;
done
echo "Last line is $lastline"
exec 0<&3 3<&-
read stdinline
echo "Next Line of stdin is $stdinline"
```

Notes: This is a variant of UseExecToAvoidSubshell? that works for the case that the **stdin/stdout/stderr** that is being redirected will be needed again. This is the technique the Korn shell uses to avoid the problems associated with creating SubShells? to perform I/O redirection.

IsolateDaemonsFromTerminals?

Daemon processes can be affected by the terminal they were started from. Consider the following daemon startup script:

```
echo "Starting Console Log Daemon"
osmcat  /dev/osm > /var/spool/console &
```

If this script is started from a terminal (other than the console), problems can arise when the person who started it logs off, hits break or when **osmcat** tries to write to standard error (or to read from standard input) after the user starting logs off.

Therefore: Isolate daemon processes from terminals. If **osmcat** is badly designed, all of the following safeguards are necessary to ensure it functions correctly when started from a terminal:

- redirect **stdin, stdout, stderr**
- ignore signals 1, 2, 3 and 15 (unless 15 is used to trigger graceful shutdown)

```
echo "Starting Console Log Daemon"
trap '' 1 2 3 15
osmcat  /dev/osm > /var/spool/console 2>/dev/console </dev/null &
```

Notes: These techniques are also relevant to other languages, but such startup scripts are most commonly written in Shell.

The **nohup(1)** tool handles only the **HUP (1)** and **TERM (15)** signals.

UniqueDelimiters?

When combining several input streams into a single file, so each part can be separated later, you want to separate the parts with a unique delimiter:

```
cat <<'THISISADELIMITER'
. . . .
. . . .  stuff
. . . .
THISISADELIMITER
```

However, there is no guarantee that the delimiter you choose isn't already a line in the file. You could embed ends-of-file in the file, but reading past ends-of-file often causes strange behavior.

Therefore: Prepend every line of the archive file with any character different from the first character of the delimiter; that is sufficient to make aliasing impossible:

```
#!/bin/sh
PATH=/bin:/usr/bin
echo '# To unbundle, sh this file'
for i
```

```
do
    echo "echo $i 1>&2"
    echo "sed 's/.//' >$i <<'//GO.SYSIN DD $i'"
    sed 's/^/-/' $i
    echo "//GO.SYSIN DD $i"
done
```

As a result, there is no aliasing; the archive file can have explicit delimiters that make it more self-documenting and easy to persue with an editor.

Many thanks to BrianKernighan?, who is author of the **bundle** script.

CleanupTempFiles?

Often it is necessary to save intermediate results of a shell script in a file. I refer below to a file used for this purpose as a **temp file**. Creating temp files in the current direectory is a bad idea, because the user may not have write permission in the current directory. Creating temp files in /tmp introduces the problem of file name collision, if a fixed file name is used.

In either event, the shell script author should ensure that the temp files are removed when the shell script exits, even if the exit is an abnormal one.

With that in mind, I use the following pattern for dealing with temp files:

```
TMPDIR="${TMP:-/tmp}/$$"
trap 'rm -rf "$TMPDIR"' 0
trap ' exit ' 1 2 15
mkdir "$TMPDIR"
```

The first line defines a (hopefully) unique directory in which to keep temp files. The default is to keep them under the /tmp directory, but that can be overriden by the $TMP environment variable. The next line arranges for the entire directory and its contents to be removed when the shell script exits. The third line arranges for the shell script to exit on signals 1 (hangup), 2 (interrupt), or 15 (terminate). The spaces around the word "exit" are there because some versions of the Korn Shell treat the word exit immediately following the trap command to mean the **EXIT** trap number.

Finally, the directory is created. Now, the shell script can create tempfiles of the form **$TMPDIR/xxxx** without worry.

Added 6/19/96 by Michael Lindner[1]

Add your comments here.
