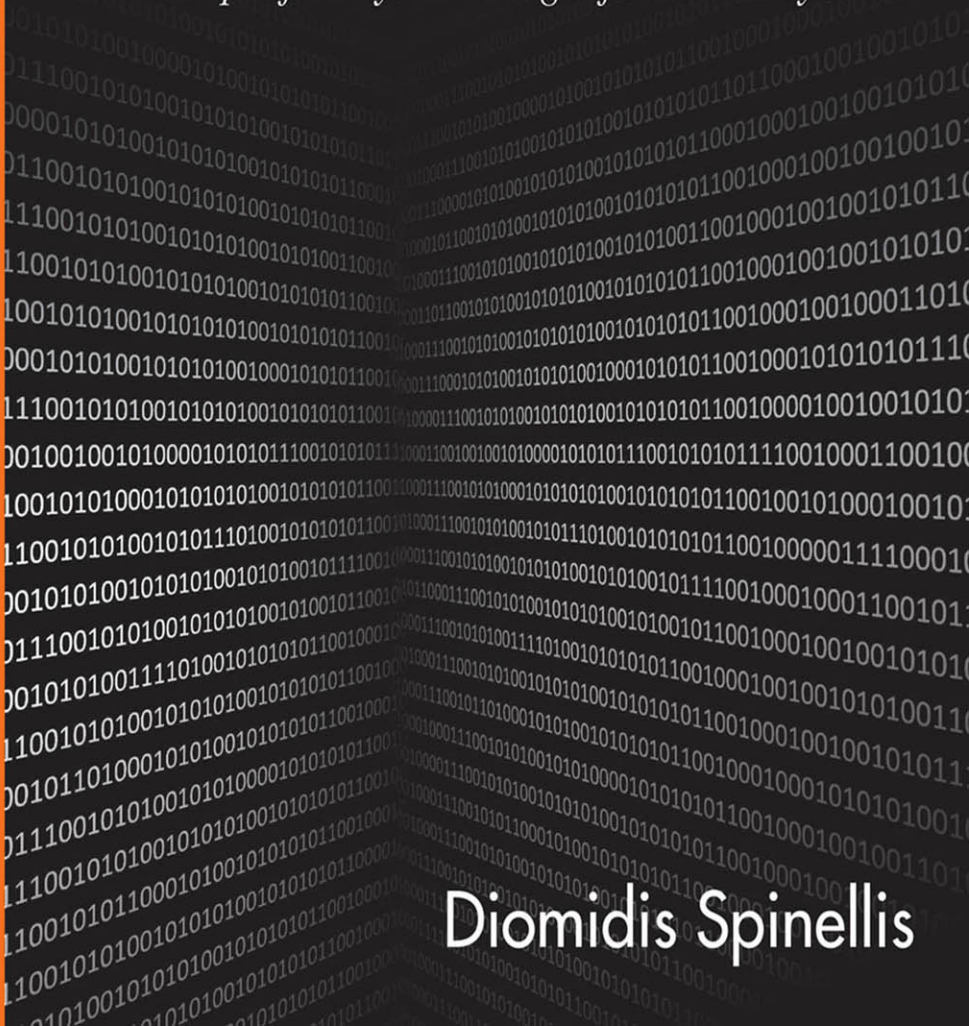
Effective SOFTWARE DEVELOPMENT SERIES
Scott Meyers, Consulting Editor

Effective DEBUGGING

66 Specific Ways to Debug Software and Systems



Diomidis Spinellis

Effective Debugging

This page intentionally left blank

3 General-Purpose Tools and Techniques

Although specialized debugging tools can be friendly and efficient, general-purpose ones often have an edge because you can use them to quickly solve a wide variety of development and operations problems in diverse languages and platforms. The tools described in this chapter trace their origin to Unix, but are nowadays available on most systems including GNU/Linux, Windows, and OS X. The flexibility, efficiency, and wide applicability they offer you justifies investing time and effort to master them. A succinct guide you can use for this purpose is Joshua Levy's collaboratively edited text, "*The Art of Command Line*." Assuming you know the basics of Unix command-line use and regular expressions, this chapter focuses on the specific tools and methods you'll use when debugging.

Item 22: Analyze Debug Data with Unix Command-Line Tools

When you're debugging you'll encounter problems no one has ever seen before. Consequently, the shiny IDE that you're using for writing software may lack the tools to let you explore the problem in detail with sufficient power. This is where the Unix command tools come in. Being general-purpose tools, which can be combined into sophisticated pipelines, they allow you the effortless analysis of text data.

Line-oriented textual data streams are the lowest useful common denominator for a lot of data that passes through your hands. Such streams can be used to represent many types of data you encounter when you're debugging, such as program source code, program logs, version control history, file lists, symbol tables, archive contents, error messages, test results, and profiling figures. For many routine, everyday tasks, you might be tempted to process the data using a powerful Swiss Army knife scripting language, such as *Perl*, *Python*, *Ruby*, or the

Windows *PowerShell*. This is an appropriate method if the scripting language offers a practical interface to obtain the debug data you want to process and if you're comfortable to develop the scripting command in an interactive fashion. Otherwise, you may need to write a small, self-contained program and save it into a file. By that point you may find the task too tedious, and end up doing the work manually, if at all. This may deprive you of important debugging insights.

Often, a more effective approach is to combine programs of the Unix tool chest into a short and sweet pipeline that you can run from your shell's command prompt. With the modern shell command-line editing facilities, you can build your command bit by bit, until it molds into exactly the form that suits you.

In this item, you'll find an overview of how to process debug data using Unix commands. If you're unfamiliar with the command-line basics and regular expressions, consult an online tutorial. Also, you can find the specifics on each command's invocation options by giving its name as an argument to the *man* command.

Depending on the operating system you're using, getting to the Unix command line is trivial or easy. On Unix systems and OS X, you simply open a terminal window. On Windows, the best course of action is to install *Cygwin*: a large collection of Unix tools and a powerful package manager ported to run seamlessly under Windows. Under OS X, the *Homebrew* package manager can simplify the installation of a few tools described here that are not available by default.

Many debugging one-liners that you'll build around the Unix tools follow a pattern that goes roughly like this: fetching, selecting, processing, and summarizing. You'll also need to apply some plumbing to join these parts into a whole. The most useful plumbing operator is the pipeline (`|`), which sends the output of one processing step as input to the next one.

Most of the time your data will be text that you can directly feed to the standard input of a tool. If this is not the case, you need to adapt your data. If you are dealing with object files, you'll have to use a command such as *nm* (Unix), *dumpbin* (Windows), or *javap* (Java) to dig into them. For example, if your C or C++ program exits unexpectedly, you can run *nm* on its object files to see which ones call (import) the `exit` function.

```
# List symbols in all object files prefixed by file name
nm -A *.o |
# List lines ending in U exit
grep 'U exit$'
```

The output, such as the example below, is likely to be more accurate than searching through the source code.

```
cscout.o:           U exit
error.o:           U exit
idquery.o:          U exit
md5.o:            U exit
pdtoken.o:          U exit
```

If you're working with files grouped into an archive, then a command such as *tar*, *jar*, or *ar* will list you the archive's contents. If your data comes from a (potentially large) collection of files, the *find* command can locate those that interest you. On the other hand, to get your data over the web, use *curl* or *wget*. You can also use *dd* (and the special file /dev/zero), *yes* or *jot* to generate artificial data, perhaps for running a quick benchmark. Finally, if you want to process a compiler's list of error messages, you'll want to redirect its standard error to its standard output or to a file; the incantations `2>&1` and `2>`*filename* will do this trick. As an example, consider the case in which you've changed a function's interface and want to edit all the files that are affected by the change. One way to obtain a list of those files is the following pipeline.

```
# Attempt to build all affected files redirecting standard error
# to standard output
make -k 2>&1 |
# Print name of file where the error occurred
awk -F: '/no matching function for call to Myclass::myFunc/
  { print $1}' |
# List each file only once
sort -u
```

Given the generality of log files and other debugging data sources, in most cases you'll have on your hands more data than what you require. You might want to process only some parts of each row, or only a subset of the rows. To select a specific column from a line consisting of fixed-width fields or elements separated by space or another field delimiter, use the *cut* command. If your lines are not neatly separated into fields, you can often write a regular expression for a *sed* substitute command to isolate the element you want.

The workhorse for obtaining a subset of the rows is *grep*. Specify a regular expression to get only the rows that match it, and add the `-v` flag to filter out rows you don't want to process. You saw this in Item 21: "Fix

All Instances of a Problem Class," where the sequence was used to find all divisions apart from those where the divisor was `sizeof`.

```
grep -r ' / ' . |
grep -v '/ sizeof'
```

Use *fgrep* (*grep* for fixed strings) with the `-f` flag if the elements you're looking for are plain character sequences rather than regular expressions and if they are stored into a file (perhaps generated in a previous processing step). If your selection criteria are more complex, you can often express them in an *awk* pattern expression. Many times you'll find yourself combining a number of these approaches to obtain the result that you want. For example, you might use *grep* to get the lines that interest you, `grep -v` to filter out some noise from your sample, and finally *awk* to select a specific field from each line. For example, the following sequence processes system trace output lines to display the names of all successfully opened files.

```
# Output lines that call open
grep '^open(' trace.out |
# Remove failed open calls (those that return -1)
grep -v '= -1' |
# Print the second field separated by quotes
awk -F\" '{print $2}'
```

(The sequence could have been written as a single *awk* command, but it was easier to develop it step-by-step in the form you see.)

You'll find that data processing frequently involves sorting your lines on a specific field. The *sort* command supports tens of options for specifying the sort keys, their type, and the output order. Once your results are sorted, you then efficiently count how many instances of each element you have. The *uniq* command with the `-c` option will do the job here; often you'll postprocess the result with another sort, this time with the `-n` flag specifying a numerical order, to find out which elements appear most frequently. In other cases you might want to compare results between different runs. You can use *diff* if the two runs generate results that should be the same (perhaps the output of a regression test) or *comm* if you want to compare two sorted lists. You'll handle more complex tasks, again, using *awk*. As an example, consider the task of investigating a resource leak. A first step might be to find all files that directly call `obtainResource` but do not include any direct calls to `releaseResource`. You can find this through the following sequence.

```
# List records occurring only in the first set
comm -23 <(
```

```
  # List names of files containing obtainResource
  grep -rl obtainResource . | sort) <(
  # List names of files containing releaseResource
  grep -rl releaseResource . | sort)
```

(The `$(...)` sequence is an extension of the *bash* shell that provides a file-like argument supplying, as input, the output of the process within the brackets.)

In many cases, the processed data is too voluminous to be of use. For example, you might not care which log lines indicate a failure, but you might want to know how many there are. Surprisingly, many problems involve simply counting the output of the processing step using the humble *wc* (word count) command and its `-l` flag. If you want to know the top or bottom 10 elements of your result list, then you can pass your list through *head* or *tail*. Thus, to find the people most familiar with a specific file (perhaps in your search for a reviewer), you can run the following sequence.

```
# List each line's last modification
git blame --line-porcelain Foo.java |
# Obtain the author
grep '^author ' |
# Sort to bring the same names together
sort |
# Count by number of each name's occurrences
uniq -c |
# Sort by number of occurrences
sort -rn |
# List the top ones
head
```

The *tail* command is particularly useful for examining log files (see Item 23: "Utilize Command-Line Tool Options and Idioms" and Item 56: "Examine Application Log Files"). Also, to examine your voluminous results in detail, you can pipe them through *more* or *less*; both commands allow you to scroll up and down and search for particular strings. As usual, use *awk* when these approaches don't suit you; a typical task involves summing up a specific field with a command such as `sum += $3`. For example, the following sequence will process a web server log and display the number of requests and average number of bytes transferred in each request.

```
awk '
# When the HTTP result code is success (200)
# sum field 10 (number of bytes transferred)
```