Make it right using Tcl

Software testing with Tcl for Apache Rivet

David N. Welton

Any sufficiently complex software system has bugs, and those of us who aspire to produce high quality work also seek to not only minimize these, but guarantee that our code does what we say it ought to.

One proven way to eliminate bugs, and ensure that code behaves as documented is to test the program. Easy enough to do by hand, when there isn't much functionality. However, when the system grows more complex, and there are many possible environmental factors with various permutations, it quickly becomes obvious that we need to automate our testing.

This article aims to provide the reader with a few notions about software testing in general, and then concentrates on a specific example, using the test suite written in Tcl for Apache Rivet, in order to demonstrate a real-world approach to testing a particular program.

Software testing

Testing is often an afterthought, and even for large, complex systems, or expensive, proprietary software, testing is never going to directly generate revenue, or add new features to the program, and so it has to compete for scarce developer time. Even in free software work, more often than not, it's more fun to spend time hacking on a cool new feature rather than writing test cases to make sure everything works exactly as advertised.

This means that it's important to try to get the most from the time you dedicate to testing - to get better value for money. In technical terms, it is desirable to maximize is the "code coverage" for the time invested. Coverage refers to how much of the code is exercised by the test suite. Think of running a simple program with options -x, -y, and -z. If you run it with the -x option, the "paths" through the other options will not be taken, and you won't know if they really work or not. So you have to expand your code coverage to all three code paths. Generalizing, the two main approaches are "white box" and "black box" testing.

Testing is often an afterthought, and even for large, complex systems, or expensive, proprietary software, testing is never going to directly generate revenue, or add new features to the program, and so it has to compete for scarce developer time

White box testing is testing a program based on knowledge of its internal workings. For example, writing a series of tests that give different input to each C function in a program, and the checking to ensure that it behaves correctly. Obviously, you need the source code, and the ability to rebuild the program in order to do this type of testing. One of the most important reasons to use this approach is that it is theoretically possible to test most or all of the code paths in the program. In practice though, the effort required to do this may be significant - imagine that you have a C function that takes a struct as input, and that struct is in turn generated by other functions, and so on. It's clear that things can quickly get more comlpicated.

Black box (or "functional") testing involves running a program against a specification to make sure it produces the correct output. For instance, testing to ensure the `ls` program works correctly. A very simple test would be to create a directory with files of known sizes and creation times, run the `ls` program, and compare its output with the known contents of the directory.

Apache Rivet

Apache Rivet is a server-side Tcl system for the creation of dynamic web pages. Think JSP or PHP, but using Tcl, a free, multi platform, general-purpose scripting language. For example:

```
<b><? puts "The date is: [clock format \
  [clock seconds]]" ?></b>
```

It is best to test the software with as little modification to the environment as possible, meaning that the test suite will run using the copy of Apache already installed on the computer - having to create a special copy of Apache would defeat the purpose.

The goal of the test suite is to be able to start the web server with configuration options of our choosing, send HTTP requests, receive answers, stop the server, and then create a report. Because it's so tightly integrated with Apache and Tcl, white box or unit test would be difficult. It would be very laborious to create and fill in all of the arguments that are passed to each function in the C code, because they usually reference complex C structures such as the Tcl interp struct, or the Apache request struct, which rely, in turn, on lots of configuration and set up. The effort required to make most of this work would probably be more than that involved in creating Rivet itself! So, a "black box" testing style would provide more coverage for the time dedicated to it. From the test suite, there much required, as the real work is in devising clever ways to test as much of Rivet's functionality as possible. An application is required that allows program tests to be performed quickly and flexibly, and provides a lot of tools for interacting with Rivet and Apache:

- Reading and writing files in order to manipulate Rivet's configuration files.
- Process control, to control the Apache process itself.
- Sockets and an implementation of the HTTP protocol in order to send requests to the Rivet-enabled web server.
- Good string matching and regular expression support.

Being a fan of Tcl, I choose.

The Tcl Test Suite ships as part of the core Tcl distribution, and is an excellent base upon which to build a series of tests for all kinds of applications

The Tcl test suite

The Tcl Test Suite ships as part of the core Tcl distribution, and is an excellent base upon which to build a series of tests for all kinds of applications. As noted Tcl expert Cameron Laird says:

> Tcl tests automobile engines, emergency telephone circuits, chemical sensors, microprocessors, rocket components, industrial ovens, and much, much more. In the absence of any other knowledge, ANY software project should think of Tcl as its first choice for testing.

What does the suite itself provide? It defines several Tcl commands that are used in order to create and run a series of tests written in the Tcl language, and generate reports based on the results. It gives control over exactly which tests are wanted, and how the output is to be organized, as well as a number of commands to control the testing environment (files, directories, output, errors, and so on).

Returning to the example of testing `ls`, the following code contains some simplistic tests for the `ls` command.

```
# Load the tcltest package, and import its commands
# into the current namespace.

package require tcltest
namespace import tcltest::*
```

```
# Set up a directory named 'testdir' for the tests
# to use.
tcltest::makeDirectory testdir

# ls -l output regexp for two files that will
# be created.
set lslformat "^-rw-rw-r-- +1 $tcl_platform(user) \
        +\\w+ +1 \(.+\) A$ ^-rw-rw-r-- +1 \
        $tcl_platform(user) +\\w+ +1 \(.+\) B$"
# perms username group time filename

# Create two empty files.
proc makefiles {} {
  makeFile {} testdir/A
  makeFile {} testdir/B
}

# Delete the same two files.
proc delfiles {} {
  removeFile testdir/A
  removeFile testdir/B
}

# Running ls on an empty directory should return
# an empty string result.
test ls-1.1 {empty dir ls test} {
  exec ls testdir
} {}

# Run ls on two files.
test ls-2.1 {ls two files} {
  makefiles
  exec ls testdir
} {A
B}

# Delete the files for the next run.
delfiles

# Run ls -l on two files. Matching on this is more
# complex because of the data and user. The username
# from comes from $tcl_platform(user). To match the
# date, save the time the files were created, and
# check that against the time reported.

test ls-3.1 {ls -l test} {
    makefiles
    # Save the file creation time.
    set filecreatetime [clock seconds]

    # Run ls -l
    set result [exec ls -l testdir]

    # Check the results against the regular expression,
    # and add that to the result.
    lappend testresult \
        [regexp -line $lslformat $result \
             match matchtime]

    # If the regexp matched, have a look at the time
    # it returned, and compare it with the time we
    # have in memory.
    if { $testresult == 1 } {
        # clock scan reads the formatted time string
        # and returns a number of seconds.
        set lsfiletime [clock scan $matchtime]
        # Make sure the numbers aren't too different.
        # A small difference is ok here, because the
        # 'clock' command might have been run a little
        # later than when the files that were created,
        # and more importantly, because ls -l does not
        # include seconds in its results.
        lappend testresult \
          [expr {abs($filecreatetime - $lsfiletime) \
                 > 100}]
    }
    set testresult
} {1 0}

delfiles

# Clean up tests and print results.

cleanupTests
```

In this script, the test cases are defined by the "test" command. In this code fragment, for instance

```
test ls-2.1 {ls two files} {
        makefiles
        set result [exec ls testdir]
        set result
        puts $result
} {
A
B
}
```

`ls-2.1` is the short name of the test, which is followed by a description of the test, the test case body, and finally, the expected result.

The body of the code in this example does nothing more than create the two files and run `ls`. The result should be A B, each one on its own line. When the test is run, it should show that it passed all of the tests:

```
@ashland [~] $ tclsh ./lstests.test
lstests.test:
Total 3
Passed 3
Skipped 0
Failed 0
```

Tcl has lots of facilities for interacting with the outside world, so you can open sockets, look at files, execute programs and so on

If there had been a failure, it might look something like this:

```
@ashland [~] $ tclsh ./lstests.test

==== ls-2.1 ls two files FAILED
==== Contents of test case:

        makefiles
        exec ls testdir

---- Result was:
A
B
C
---- Result should have been (exact matching):
A
B
==== ls-2.1 FAILED

lstests.test: Total 3   Passed 2   Skipped 0   Failed 1
```

In this case, there is a third file (for the sake of argument), and so the test fails. The test suite shows which test failed, what the expected results were, and how many of the tests passed. It is also clear that in the 3rd test, the "correct output" is not just the results of the `ls -l` command. It can be calculated whether the `ls -l` output is satisfactory by both testing it against the regular expression, and examining the time. So instead of a basic string match, return the results of the "real" matching performed in the test body and "match" against those.

Because this framework is so simple, it can be adopted to any number of situations. All that is required is find a way to make the test perform an action, and then compare that result with the expected outcome. Tcl has lots of facilities for interacting with the outside world, so you can open sockets, look at files, execute programs and so on.

Testing Apache Rivet

In order to test Apache Rivet, it's necessary to be able to create a controlled environment in which to run the tests, then run through them, stopping and starting the server as the tests dictate. An Apache module Rivet can be either: compiled directly into the web server, or, of course, loaded at run time when compiled as a shared object. In either case, it is just one component of the web server, which is a complex system. For this reason, it is necessary to ensure that testing it gives the same results, even across diverse systems, so that programmers can count on it to always execute their code correctly.

The first task is to set up a minimalist environment for Rivet to run in by creating configuration files with as little in them as possible. Since the aim is to automate testing, advantage is taken of several Apache features in order to automatically generate these configuration files.

It's necessary to be able to create a controlled environment in which to run the tests, then run through them, stopping and starting the server as the tests dictate

```
@ashland [~/workshop/tcllib/modules/log] $ apache -V
Server version: Apache/1.3.26 (Unix) Debian GNU/Linux
Server built: Aug 12 2002 21:19:30
Server's Module Magic Number: 19990320:13
Server compiled with....
-D EAPI
-D HAVE_MMAP
       -D HAVE_SHMGET
       -D USE_SHMGET_SCOREBOARD
-D USE_MMAP_FILES
-D HAVE_FCNTL_SERIALIZED_ACCEPT
-D HAVE_SYSVSEM_SERIALIZED_ACCEPT
-D SINGLE_LISTEN_UNSERIALIZED_ACCEPT
       -D HTTPD_ROOT="/usr"
-D SUEXEC_BIN="/usr/lib/apache/suexec"
       -D DEFAULT_PIDLOG="/var/run/apache.pid"
    -D DEFAULT_SCOREBOARD="/var/run/apache.scoreboard"
       -D DEFAULT_LOCKFILE="/var/run/apache.lock"
       -D DEFAULT_ERRORLOG="/var/log/apache/error.log"
       -D TYPES_CONFIG_FILE="/etc/mime.types"
       -D SERVER_CONFIG_FILE="/etc/apache/httpd.conf"
       -D ACCESS_CONFIG_FILE="/etc/apache/access.conf"
       -D RESOURCE_CONFIG_FILE="/etc/apache/srm.conf"
```

The command `apache -V` gives a variety of information about Apache's compile time options, which can then be put to use...

```
@ashland [~/workshop/tcllib/modules/log] $ apache -l
Compiled-in modules:
        http_core.c
        mod_so.c
        mod_macro.c
suexec: enabled; valid wrapper /usr/lib/apache/suexec
```

`apache -l` reports the modules that are already compiled into the Apache build. On my Debian system, very little is actually compiled in, as you can see - it's all loaded as modules, in order to make the Apache install as flexible as possible.

Thanks to this information, it's possible to figure out if Rivet has been compiled into the server, and where the server's main configuration file is. In this case, it can be seen that mod_so is present, meaning modules can be loaded, Rivet is not built into the executable, and that the main configuration file is at `/etc/apache/httpd.conf`. Were Rivet compiled directly into the main executable, a line like "mod_rivet.c" would have been seen, and the next step could be taken. If there were no mod_rivet, and no mod_so to load other modules, it would have been necessary to error out! The Tcl code for retrieving this information - apachetest::getcompiledin - isn't very complex, and is really just there to sort through the output of "apache -l". Since there is a way to load shared objects, take a look at what happens next.

The `getcompiledin` procedure tells which modules are part of the apache build, and that Rivet isn't one of them. Given that the standard use of the test suite is to test a change that has just been made to the Rivet source code, this situation is actually sensible, because it's easier to quickly compile Rivet as a module, and then include that, than make a big, fat build of Apache. In fact, given the Rivet directory layout:

```
tcl-rivet/
src/
        tests/
```

It's easy to find the Rivet module that was just built – the test suite is always run in the "tests" directory, so the new shared object just built is only "one directory over" or in Unix parlance, "../src".

When writing a test suite for your own code, this sort of arrangement is helpful, so that you don't have to "tell" (via command line options or some other means) your test suite where everything is, making it even faster to run. Getting back on track, what's built in and where Rivet is are known, but to run all the tests we want to perform, a few other Apache modules are needed: mod_log_config, mod_mime, mod_negotiation, mod_dir mod_access, and mod_auth, which are all part of the standard Apache distribution. Even if they aren't compiled in, no problem, in apachetest::getloadmodules, sift through the configuration file, looking for the locations of those modules that are needed, or error out of they're not there, because they really are required!

Now that it's known where everything required is, a minimalistic configuration file can be written in the tests/ directory. I'll come back to this file and its contents in a moment. You should now be able to run Apache with Rivet loaded up, which is a pretty good place to start the tests! Here, another obstacle is encountered though. Apache is a "daemon" (i.e., on Unix, a program that runs in the background) and you want to be able to start and stop Apache at will, so it can't just go "running loose". Fortunately, once again there is a command line option that comes in quite handy: "-X". When apache is run in this way, it doesn't go into the background, but instead runs as one process in the foreground, which is a big help. Apache still needs to be launched in such a way that the program doesn't block (stop and wait) while it is running, because then you couldn't go about testing. Tcl's exec command permits this with no problems:

```
   set serverpid [eval exec $binname -X -f \
                 [file join [pwd] server.conf] \
                 $options >& apachelog.txt & ]
```

What is being done here is the apache server process is being executed in the background (via the & at the end of the command line), and with the -f command line option, telling it to use the configuration file just created, instead of the standard one. The process id must be monitored, so that it can killed off when required, and any output redirected (there shouldn't be much) to a file. The configuration file created is very simple, just enough to make Rivet and a few other modules, that it interacts with, work. Rivet has several configuration directives that it is desirable to test, of course, and furthermore, you need to be able to modify several other lines in the standard configuration that is used. In order to be flexible, utilize a template file, called, `template.conf.tcl`, which has several variables that are filled in when loading up the file. After which, the file is then written to the tests/ directory, where it is ready to be used.

```
# Minimal config file for testing

# Parsed by makeconf.tcl

ServerRoot "$CWD"

PidFile "$CWD/httpd.pid"

ResourceConfig "$CWD/srm.conf"
AccessConfig "$CWD/access.conf"
```

```
Timeout 300

MaxRequestsPerChild 0

$LOADMODULES

Port 8081

ServerName localhost

DocumentRoot "$CWD"

<Directory "$CWD">
Options All MultiViews
AllowOverride All
Order allow,deny
Allow from all
</Directory>

<IfModule mod_dir.c>
DirectoryIndex index.html
</IfModule>

AccessFileName .htaccess

HostnameLookups Off

ErrorLog $CWD/error_log

LogLevel debug

LogFormat "%h %l %u %t \\"%r\\" \
           %>s %b \\"%{Referer}i\\" \
           \\"%{User-Agent}i\\"" combined
CustomLog "$CWD/access_log" combined
```

After all of this, it's evident why it was desirable to automate it, eh? By way of explanation, you can see that the file is littered with `$CWD`, which you should replace with the current working directory where the tests are located. All of the test and .rvt files are located in one big directory (perhaps it's time to sort things out a bit more!), so that locating them is no problem - set `ServerRoot` and `DocumentRoot` to CWD. `$LOADMODULES`, which is replaced by directives to load the modules required - the same ones Apache regularly uses. Aside from the configuration files, .test and .rvt files, there are a few log files, but (currently) nothing is required of those. There is no other preparation required for our environment. For Rivet, there is no cleanup to perform either, but you should always consider this, in order to have a pristine condition to start from, after your tests have run.

Finally, the first test is ready to be run. Of course, tradition calls for "Hello, world" at this point, and who are we to buck the trend? The `hello.rvt` file looks like this:

```
<?
# hello-1.1
puts ''Hello, World\n''
# i18n-1.1
puts ''< À È Ì Ò Ù - El Burro Sabe Más
Que Tú!\n''
?>
<p>ÆüËÜ¸ì(EUC-JP Japanese text)</p>
```

Leaving be the Spanish and Japanese texts (used to test internationalization features) for the moment, take a look at what happens in hello.test:

```
set testfilename1 hello.rvt

test hello-1.1 {hello world test} {
 set page [::http::geturl \
  "${urlbase}$testfilename1"]
 regexp -line {^Hello, World$} \
  [::http::data $page] match
 set match
} {Hello, World}
```

Which is really pretty simple.

First, the URL with Tcl's http package is fetched:

```
set page [::http::geturl "${urlbase}$testfilename1"]
```

The page data is run through a regular expression, and put the results in the "match" variable...

```
regexp -line {^Hello, World$} [::http::data $page] match
```

... which is then returned.

```
 set match
} {Hello, World}
```

The result of this test should be "Hello, World". If the "match" variable doesn't contain that text, the test will fail. Most of the tests are similar to this one. HTTP isn't a terribly complicated protocol, so for most cases, all that is necessary is to do is send a request, and check what is received back from the server. An effort has been made to test as many different features of Rivet as possible, including:

- The ability to handle internationalized content - at the moment, a few accented characters are sent, and some Japanese text.

- Binary data, which is tested by sending and receiving a JPEG of Fishbone's Angelo Moore! This required the use of some code to do a file upload in Tcl, which I gratefully borrowed from Jeff Hobbs. To make sure the result is what it is supposed to be, compare the original file with the uploaded or received version.
- Error messages. Just to make sure that the error message works correctly.
- Apache's built-in environmental variables like (like `DOCUMENT_ROOT`).
- Passing variables through both GET and POST.
- Cookies. These two were very easy, thanks to Tcl's http package.
- Parsing and including secondary files, with and without non-ASCII characters. Rivet is able to "include" other Rivet (mixed HTML and Tcl) files, parsing them correctly as it does.
- Internationalization in general. This was a difficult one - despite speaking Italian, I'm a native English speaker and am not too concerned about creating content with non-ASCII character sets. The value of the free software community was proven when two Japanese users, Taguchi Takeshi and Makoto Satoh provided me with help in understanding the issues involved, and how Rivet needed to behave to meet their needs.
- Uploading files.

Getting Apache's configuration directives right in Rivet was also somewhat of a chore, and it was necessary to ensure that they weren't interacting badly with one another. To cover all the possible permutations, the creation of tests, looping through and trying different combinations, was automated:

```
foreach Config $ConfigList {
 incr i
 test config-auto-${i}.1 {config test} {
  ... body ...
 }
}
```

`i` is a counter variable, that is used in creating the test name: `config-auto-${i}.1`, and incremented `(incr i)` every time around.

HTTP isn't a terribly complicated protocol, so for most cases, all that is necessary is to do is send a request, and check what is received back from the server

Conclusion

Hopefully, this article has stimulated some interest in a subject that at first glance doesn't appear to be all that exciting. It still may not be as fun or glorious as adding the greatest new kernel feature, but can be an entertaining challenge to try to devise the best ways to test your program's features one at a time. Of course, the benefits to your code quality will speak for themselves.

Copyright information

About the author

David N. Welton lives and works in Padova, Italy as a consultant specializing in free software technologies (Linux, Apache, Tcl, Python, C, etc...) providing programming, training and strategic consulting services. He is an active participant in the Tcl comunity, has been a part of the Debian project since 1997, and has been the Vice President (cat herder?) of Apache Tcl with the Apache Software Foundation since 2001. He may be found on the web at the DedaSys web site (http://www.dedasys.com/davidw/)