Baseline Testing

Gene Michael Stover Jay F. Smith

created Tuesday, 7 October 2003
updated Sunday, 12 October 2003

Contents

Jay: Is my use of "test case" correct? How about "test harness"? I haven't inserted section headers yet. Thought it'd be better to flesh out the prose & see what organization it takes before inserting them.

You want to automate some of your testing? Baseline testing is a simple technique that handles many testing situations.

With baseline testing, you start with an input/output pair that are known correct. The output part of the pair is called the baseline. To perform the test, you send the input data through the System Under Test (SUT) & compare its output with the baseline. With some predefined & expected exceptions, any differences are suspicious; those differences are probably errors & must be investigated.

Jay: Will the telcos sue us for telling the world about the existence of switches in this example?

Here is a simple example: The SUT is an application that coalesces the records from wireless telephony switches & produces per-call records that are more easily processed by a billing system. (The computers which control the wireless telephony network are called switches, & they produce highly detailed records of activity, usually with more than one record for what a billing system would consider a single call.) For your test, your input file contains real switch records, selected & arranged to test one or more features of the SUT. Your baseline file (which is the expected output) contains the per-call records you have derived from the input file. You run the input file through the SUT, save the output, & compare that output to the baseline. The *diff* program is probably adequate to compare actual output & the baseline in this case, but sometimes you need a special comparison program. We'll talk more about that later in this article.

Here are the general steps to creating a baseline:

1. Design your Test Case.

2. From the test case, derive the input data.
3. Create your baseline data, either by deriving it from the test case or run it through the SUT & verify it "by hand".
4. Save the test case, the input data, & the baseline in your database.

Here are the steps to execute a baseline test. These steps should be done with an automated testing harness when possible:

1. Retrieve the input data file for your test case.
2. Run that input data through the SUT.
3. Compare the actual output to the baseline.

That's all there is to it. It's kind of a "duh" technique, when you think about it, but there are some tricks to maximizing the range of situations in which baseline testing is applicable. So let's do some more examples.

Sometimes, the actual output will contain fields that you know will differ from the corresponding fields in the baseline. You'll need to ignore some fields when comparing the actual output with the baseline. That's easily done by filtering those fields from both the baseline & the actual output before comparing them with *diff*. If your output files are CSV or a similar form, the filtering program takes less than ten trivial lines of Perl code.

A common case of such fields is time-stamps, but with time-stamps, you have another option. Part of the inputs for the test can be the time at which the baseline was created. To run the test, your test harness can set the system clock to that value before it runs the input through the SUT. This works unless the resolution of the time-stamps is fine enough that they are affected by the inevitable small differences in processing time, & even with time-stamps of low resolution, it is possible that a record will be a border-line case, sometimes with one time-stamp, sometimes with another, so resetting the clock is of limited use. Also, system administrators hate the idea.

Jay: Is this a good example, or would another be better? The example I have here isn't really baseline, I think. We need some example that shows files that don't have records. Or an example that shows a complex application, if we can do that in the space for an article.

Here's a more complex example that uses multiple files. In it, let's test an implementation of *tar*.[1] Our newly implemented, fictional *tar* program is *bltar* (BaseLine TAR). The idea behind this test is to verify that a pre-existing implementation, *tar*, can extract an archive that *bltar* created. This might be a special case of baseline testing because the input files are the baseline itself.

We use *bltar* to create a tarball, then pipe that file into *tar* to extract it into another directory. We compare the directories with "*diff -r*". It's best to let your test framework run the test, but by way of explanation, it would work

[1]Sure, *tar* is a done deal, but that doesn't mean it doesn't make a good example.

as if we ran this shell command: "`bltar cf - testdir |(cd /tmp; tar xf -); diff -r testdir /tmp/testdir`". Any differences should be considered suspicious; they are almost certainly errors.

So far, the examples have used files, but baseline testing isn't limited to files. Here's another example that isn't testing a file-processing system. It requires some custom hardware for a project testing a television tuner that responds to a remote control.

Jay: Since I don't know hardware testing very well (though I do it at work currently, I guess), I think this example sounds artificial. Maybe I can write it better later, after thinking about it, or maybe we could think of some other example. Anyway, seems like if I was reading the article to learn about baseline testing from scratch, & I was skeptical, this example would not convince me.

The television tuner is the SUT. The company developing the tuner creates a computer-controlled test rig that can generate signals like the remote control would. Another part of the test rig monitors the signal emitted by the tuner & emits the channel whenever the tuner changes it; those emissions are stored in a file. The input file tells the test software what signal the mock-up remote control should emit. The baseline file, as always, is a record of a previous good run or is constructed from the test case. For this example, it contains the channel numbers we expect from the signals emitted by the remote control.

1 Conclusion

Need a conclusion.

References