Do it our way
Linux.conf.au 2007

"I've tried it your way, how about you try it my way?" — Erik de Castro Lopo

"No, it will slow me down." — clueless colleague

Authors: Robert Collins, Erik de Castro Lopo, Peter Miller

Contents

Testing is fundamentally an engineering exercise - one tests software as part of the validation of the software. Putting time and effort into testing is not free - it has many costs: The time to write the tests, the time to execute them, code changes to support the tests. It makes sense then, when spending time on testing, to aim for the largest return in the shortest possible time.

Software testing is a hot topic today, and is often associated with 'agile methods', 'extreme programming' or 'test driven design'. While the authors generally practice 'test first development', the techniques described are independent of these methodologies. Instead the techniques are about getting back to the basics of testing - using good sense to make the best use of your time in achieving your testing goals.

Most software developers test their code in some fashion. Sometimes those tests are ad-hoc - e.g. running the tool by hand - and sometimes they are not - e.g. an automated test suite. We focus on automated test suites in this paper because they offer a high ratio of repeated runs to developer time, compared with manually executed or ad-hoc tests.

If you are a software developer and already do extensive automated testing, you can look to get some new tricks and tips, or possibly just refinements of your existing catalogue, from this paper. To get the most out of this paper you should be willing to experiment with the recommendation contained within it. Experiment with changes to your build system, code and architecture. Play with the ideas in a junk-code area, and become familiar enough with each technique to perform it well before you decide whether it is for, or not for you. If you don't already have extensive automated testing of your code, please persevere, or play with the techniques in a small environment before applying them to your complete codebase.

What makes up a good test

At the heart of a test, there lie assertions about the expected behavior of the system under test. A good test will do much more than simply test those assertions; it will test them in a manner that is reliable, fast, low-maintenance and most importantly when failures occur, the test will assist correcting the failure without requiring interactive access to the machine the test is failing on.

When tests fail for unrelated reasons - for instance, a parsing test failing because the system clock was fast triggering an assert outside the parser - it often results in massive cascade errors when something is wrong, making it harder to diagnose the correct problem as it is now a needle in a haystack. The overall approach in testing then should be to have each test focus on a specific condition, using only the required code to produce this. A related issue occurs when tests sporadically fail - that is when they are not completely repeatable. Isolating the test from external factors can help with this repeatability by making the test less sensitive.

Improving the quality of diagnosis provided by a test depends on the language the test is written in, the problem domain being worked in, available infrastructure and more. It is worth putting some considerable effort into the test infrastructure to ease analysis. For instance, in *bzr* when a test fails, we get a detailed log of all operations carried out during the test, including both client and server logs, and a traceback at the point of failure, with a human readable printout of the failing condition - often in a diff-like format.

Getting started

When writing tests choose the easiest to write test to be the first one written. Its not always obvious what really constitutes easiest though. Consider a test which validates correct calculation of file freshness in a HTTP proxy. Without fine control over the internal time within the proxy, its almost impossible to write a test that will execute quickly. At this point one could write a slow test easily, or start development efforts to gain control over the internal time of the proxy. As Voltaire wrote, 'the perfect is the enemy of the good', and the choice in this example is between having a slow test quickly, or having a quick test slowly, and unless test suite performance is currently an issue, adding the slow test quickly will give more immediate returns.

Easiest here is possibly best defined as 'takes the least time'. By writing the easiest test first, even if the test is not ideal, its goals - whether its new functionality, or increasing coverage - are achieved as quickly as possible. Successive iterations can improve the test, once it exists.

The test patterns described later provide tools to tackle testing situations, and these all aim to make the process easier, via identifying specific scenarios where whats easiest changes.

Important infrastructure

Testing infrastructure naturally fits into a number of groups. Firstly there is at-the-programmers fingers tools - the test runner and similar low level tools. Secondly there are larger scale tools, such as automated test running during commit, or a website tracking test pass and failures over time.

While the basics of running tests is very simple, its not enough to provide a smooth and easy to use experience. Since the most common user of a test suite is the developer, having a good UI is important to make interacting with them pleasant during development. Tests may be run dozens of times in an hour, a small improvement in usability here can have significant impact.

Test execution

The very first thing is to make your tests able to run without human intervention. That is, after starting a single (or group) of test(s), you should not need to observe or interact with the test as it executes for it to complete. After it (or they) complete the exit code should reflect whether the tests all passed or not. This achieves several things:

- It helps makes it easy to run the test(s).
- Tests can be run as part of larger processes - such as creating releases.
- Tests are more compatible with IDEs and other environments, which often have glitches in their support for tty/stdin usage in subprocesses.

Depending on the language your tests are written in, this may be a trivial consideration. For other languages or environments it may be harder. Doing this requires you to stub out all the input and output done by the code being tested. For instance, in a C program you can either setup a socketpair over fd's 0 and 1 around the code being testing, letting you supply input and output, or provide an interface within your program that your tests link in a different implementation, or you can run each test via a separate invocation of your program and have the tests be shell scripts.

Test environment

The next important thing is to ensure there is some means of running all your tests in a reproducible, clean environment. Doing this allows the entire test suite to be run robustly and helps reduce spurious results between different machines. If your program is relatively isolated, with simple requirements this is easy. More complex environments can be quite tricky to achieve this with. If it can't be achieved, at least document clearly what is needed. As an example, if the program being tested is a GTK program, you will need an X server running for all the UI layer in the program. One way of achieving this is to run up the VNC X server as part of the test suite. Ideally all your tests will run when the developer does *make check*, or whatever your analogous command is. Having the tests also runnable via some other command can be useful too, for when you need to provide options to the test runner.

At this point all your tests can be run, anywhere, easily. Hooking the test suite into the build process can be useful now - any install can be sure that the tests are passing, and you can get widespread feedback on the test results.

As test suite sizes increase, the time to run all the tests becomes progressively longer. During a fast development cycle it can be very useful to only run the tests that a developer expects to be affected. So you should provide a mechanism to run a subset of the tests in the test suite. We recommend an unanchored regex as the simplest way of specifying these tests, but there are many different approaches - choose one that fits your project well. It can be useful to categorize tests. For example, *bzr*, a project one of the authors work on, has performance tests which are considered separate to regression tests. This helps keep the regression test suite running fast, while the performance tests which often have to perform a large amount of processing to get valid results are accessed by passing *--benchmark* to the test runner.

Designing a test

When designing a test, the aim should be to meet the following objectives:

- Ensure correct behavior and/or absence of incorrect behavior.
- Be repeatable.
- Should help isolate any problems found.
- Should aid diagnosability.
- Be easy to implement and extend.

The easiest way to ensure that a software module exhibits correct behavior is to provide it with known input data, retrieve the modules output data and compare the output with expected results. This can be done with as many input data sets as required to cover the full working space of the module.

When checking for the absence of incorrect behavior a common approach is to provide input data that tests corner cases and make sure that each of these corner cases are handled correctly.

In both of the above cases we are providing the software module under test with known data. Supplying known data makes tests repeatable. Testing with (truly) random data generated on the fly is a bad idea, because while it may find a bug, fixing that bug will be difficult because the random data that triggered it cannot be re-created. In addition, generating large amounts of random data can impose significant performance performance hit. For sensible use of random data see "Finding test holes with random data".

Finally, the test should be easy to implement and extend. Usually this means working with a established unit testing framework along the lines of JUnit, PyUnit, CppUnit, Check (for C) etc. These frameworks are specifically designed to make adding and refactoring tests easy. The test suite will also require other test code to be written such as data validation and helper functions. Like any other piece of code these utility functions should be reused and re-factored to maximize code reuse.

Reporting

The reporting infrastructure should facilitate quality reporting, use of tests for release management and precommit checks, and interactive development. I recommend that you provide a number of different ways of running your tests, to match the needs of your project. You will probably need at least these two:

- Debugging the precise cause of a test failure - running some selected tests with verbose output or with a debugger (implying an interactive facility).
- Running via cron/precommit in an unattended environment. This should provide enough information to allow reproduction of failures, without attempting to do full analysis - as the results will typically need to be downloaded or otherwise viewed without full access to the test environment. Ideally it will also record the tests run in a manner useful for automated analysis by recording such things as the time the test was executed and the version of the software being tested.

For use in quality reporting, consider setting up a BuildBot, buildfarm or tinderbox environment, which runs the test suite on many different platforms at regular intervals (i.e. per commit, or daily, or on request). This can provide excellent insight and advance warning for problems in upcoming releases.

Almost all output is eventually read by a human though, so be sure to consider ease of use when doing this: make the most important information the easiest to ascertain!

It will also be useful for hooking the test suite into the release process - if the release is automated, have a test run in there. (i.e. 'distcheck').

Techniques

We've split the techniques into three categories:

- Test techniques for writing and running tests
- Code techniques for changing coding practices to facilitate testing and
- Process techniques to facilitate testing via your development process.

Test Techniques

Divide and Conquer the Code

Intent

All but the smallest pieces of software are composed to several modular pieces. Large systems may initially appear to be too large to start testing, particularly when you inherit a huge legacy system maintain.

By considering each of these modules one at a time, it becomes possible to perceive methods that may be applied to test these components.

It can also be that a single source file contains huge amounts of code. It is not necessary that it all be tested in a single huge unit test. Instead, write several unit tests.

Applicability

Divide and Conquer the Code should be used when:

- dealing with unfamiliar code bases
- the tests to be written involve many dependencies, with special mention for external dependencies
- your design is able to be changed to increase testability

Application

1. identify modularity in the code/design around the dependencies
2. change the code as needed to allow replacement of the modules you wish to replace
3. test each module you may replace explicitly and individually
4. test any modules that depend on other modules in this design via replacement test-modules that do not involve those external dependencies
5. test that your test modules and the real modules are substitutable. This can be done via:
    - a single end to end (“smoke”) test
    - or several integration tests
    - or a hand test (if the dependencies are really hard)

Strengths

- Gives the code author a way to think about testing the code, rather than seeing it as too big to know where to begin.
- Gives the tester fine grained results for analyzing test failures.

Weaknesses

- Some code is too difficult to modularize in a reasonable time frame.
- It requires the use of mock objects, or alternative code to link against, or alternative objects to be instantiated. In each case, it needs test-specific code, and the time investment to create this test code.

Implementation

The structure of source files in a project can greatly influence the ease or difficultly of using this technique.

In dynamic languages like python/ruby/perl it is possible to replace real modules with test modules at runtime without explicitly supporting alternative implementations. That said, it is preferable to have the use of alternative modules be an explicit supported part of the API because this increases the clarity of the test system.

In an object oriented compiled language, replacing modules is often accomplished by having an abstract interface class, and then deriving the actual code as another class, permitting an alternative derived class specifically as a mock object for testing. This mock code need not appear in the final code used by the installed code, but used only by testing programs.

In non object orientated compiled languages, it is often possible implement this technique to have one object file which implements the real functionality, and another source file which exports exactly the same API, but which is useful when linking to test programs to provide a mock implementation.

It may be that in a large system, a mixture of any or all of these techniques will be used across the entire set of tests.

Related Techniques

- Start small

Divide and Conquer the Test

Intent

Transition from large or complex tests to small focused tests in an incremental manner.

Applicability

There are times, both when implementing large pieces of new functionality, and when fixing subtle bugs with wide implications, when you will need to write a long and detailed test consisting of long sequences actions and validations. One common scenario is when a user provides a recipe for reproducing a fault which consists of many steps.

Divide and Conquer the Test should be used when:

- A single test performs setup, assertions, and then further state changes are made.
- A test uses another test as part of its state setup.
- A test being used to validate new functionality is not fully realized within a hours work.
- Are presented with more than a few places in the code to change in order for the test to pass. That is, you'll have more than a few outstanding changes in the code at once for this test.

Motivation

- Increase the clarity of tests,
- reduce the time to execute individual tests,
- increase isolation between tests,
- provide fully passing tests for incremental development, while an acceptance test is still not passing.

Application

- Copy the existing large test.
- In the copy, find group of first assertions, delete everything after.
- In the original remove first group of assertions but leave everything else.
- Check both the new and old tests still pass [or, if the large test was not passing, that the new smaller test is fully passing].
- Factor out common code in both tests.
- Check the tests are still passing.
- Optionally commit at this point to record the new smaller test.
- Repeat as needed.

Strengths

- Gives the code writer a way of thinking about testing large and difficult scenarios or changes..
- Gives the code writer a way of creating tests without having to consider the best way to structure them before writing code - the code can be written in the test and during the refactoring moved into the real code base. This works well when combined with Placeholder Results.

Weaknesses

When writing new code in conjunction with a large test and this pattern, it can be tempting to wait until the whole test passes before committing. This has a problem of infinite regression, it can keep getting put back - and nothing has been committed. To avoid this, overcome this temptation by committing when portions are working.

Related Techniques

- Placeholder Results

Start Small

Applicable when

- Low risk involved in making code changes when the tests show defects:
  - New functionality or new work being created.
  - Existing test suite is comprehensive and should protect against regressions.
- It is desirable to test small units of code individually, when possible, before testing combinations of these code units together, and so on as you ascend the application code stack.

Application

- First write tests that use limited functionality in the code base.
- Then write tests that use combined or higher layer functionality.
- Finally write 'smoke' tests to see its all integrated together.
- Write about the same number of tests at each level of granularity. That is, each unit of functionality should get around the same number of tests. Picking an arbitrary number of 10, you should have 10 end to end integration tests, 10 tests testing the module glue logic between modules, and then 10 tests for each individual module.

Intent

- For new functionality, it is valuable to add tests for simple, even obvious, test cases. While they may appear trivial to the author, they may not be so obvious to a later maintainer.
- It is also valuable to add tests to existing software which doesn't have any tests, and these tests pass against the existing (legacy) code base. Such tests may be added before bug fixing, to ensure that modifications can be made to the code without introducing a regression. (The bug fixes, of course, include appropriate tests, and those tests are different to regression guard tests discussed here.)

Also Known As

- You have to start somewhere.

Strengths

- This technique makes it clear what should be captured in automated tests. Testing a routine case is as important as testing each of the more interesting corner cases. Usually, only one routine case is required. However, some of the corner cases may fall into this "obvious" category, and those also need to be captured.

Weaknesses

- Start as simple as possible, but no simpler.
- Unit tests of simple cases is not sufficient.
- Tests of the combinations of smaller units is also required.
- However, higher levels of complexity should have their own tests.

Related Techniques

- Divide and Conquer the Code for working with complex module interactions.

Make it readable

Intent

Human readable output of tests allows inspection and validation without gymnastics.

Motivation

If you are able to easily inspect an error which has been logged in your test output for the correctness or otherwise of variable values, it helps quickly identify incorrect values without having to re-run your test under a debugger, or with special logging code enabled.

Application

If you are able to cast a test value into a string, you can perform a simple string comparison in your test assert. This makes the placeholder result simple to copy and paste.

There are two common ways of outputting a test value as a string, one is via a custom method or function to convert it to a string, and another is via the system provided "pretty printers" some languages have.

If a test outputs to standard output, and the output is validated, a printable ASCII result is more meaningful to humans (and maintainers). Another advantage of this is that *diff* (1) can be used to show differences, and is completely silent (and exits success) when there are no differences.

Strengths

- Human readability eases debugging and correction.

Weaknesses

- Often requires the tester to write structure dumping or formatting code. This is occasionally time consuming.
- Not much use for unstructured data like images and sound data.

Consequences

Simple textual comparisons may be used, which in non object orientated languages makes *Placeholder Results* simple to implement.

Related Techniques

- Placeholder Results

Placeholder Results

Applicable when

Use when generating test input is easy but generating the validated output is difficult. The solution is to create bogus expected output (the placeholder results), then validate the generated output of the software under test and finally replace the bogus expected output with the validated output.

Also Known As

- I'll know it when I see it.

Application

1. Write the test with an expected result such as “invalid output”. Note that the expected result may be a complex type, or just a string, depending on your testing environment. It should always be something that can be easily inspected for correctness: see Make it readable.
2. Run the test to check it fails.
3. Manually inspect the output from the program to ensure its a suitable output.
4. Replace your expected result with the known good output as you have confirmed it should be expected.

Related Techniques

- Make it readable for increasing clarity of test failures.

Finding test holes with random data

Applicable when

Finding corner cases is hard e.g. the code is unfamiliar territory or overly complex.

Motivation

A project with 100% code coverage by the test suite recently had a bug discovered. To create data suitable for identifying the root cause and expanding on it, a random generator can be used. The code coverage metric was not sufficient as a coverage measure because it usually does not reflect boundary conditions nor code path coverage, particularly in the highly modular systems.

Related Techniques

- Divide and Conquer the Code to test individual modules without complex external dependencies.

Application

1. Write something to generate random inputs for your API. e.g. two strings to diff in the case of testing a diff algorithm.
2. Write code to validate the behavior for the API given the random input and your API.
    - e.g. run the API under valgrind to detect access defects, and assert that API either returns a valid results or reports an error to the caller. In addition, a wrapper script around valgrind can check the valgrind output for reports of access violations.
3. Write a test driver which will output the random input that was used for each test failure - these should then be verified by a human, and turned into as many real tests as are needed to cover the exposed defects.

Known Uses

This technique was used to find corner cases in the array diffing logic of a domain specific diff tool created by one of the authors.

Code Techniques

Exit Status

Tests should always produce a meaningful exit status. This means that tests can be aggregated into shell scripts quickly and easily, when you want to run a subset of tests, *e.g.* for a “smoke test”.

Programs and executables produced by your project should only ever exit with a successful exit status if *nothing* went wrong. This means that unaltered programs can be used in a shell script or `Makefile`, and is essential for specialist test programs.

Intent

Make it as easy as possible to integrate pass/fail results from test suites into a projects build system. If your build system is to help you with testing, it needs to be able to act on the tests pass/exit status. The most accessible information, with the fewest dependencies, is the programs exist status.

It is possible to ignore a program’s exit status, and use *grep* (1) on the program’s output text, but this is much more complex and should be avoided.

This applies both to test programs, and also the programs which are to be executed by your users.

Strengths

Process status is a very simple API that is supported by nearly every build and testing tool today.

Weaknesses

None known.

Implementation

Return non-zero from a process when any error occurs. `Automake` provides a simple extension to the boolean nature of exit status by allowing a 77 exit status to mean “No result” - e.g. the tests could not be run on the current platform.

Process Techniques

Hoard Tests

Intent

To prevent previously tested code becoming less tested over time, it is necessary to zealously collect all the tests you perform, preferably in an entirely automated form, and commit them along with your code. Hoarding tests ensures that the number of tests in the system grows rather than shrinks.

Motivation

Tests capture how you system is supposed to work, in an objective and repeatable way. If you throw a test away, you may be throwing away test coverage, leaving later developers without the protection of the discarded test and no record of the test’s intent. Never throw a test away.

If you write a test to make sure something works, commit it to the code repository. Adding this test to the automated tests means this problem, or potential problem, will never recur.

Weaknesses

Some books on writing software advocate testing as scafolding in the source file, and once the code is done, the scafolding is removed. This can be accomodated by moving the tests from the source file to a dedicated test script.

Integrate With Builds

Intent

Make the act of building/releasing your software simultaneously check for errors and regressions. This alerts your users to errors on platforms you do not have, and allows you to be sure that your tests have been run before creating a tarball to ship.

Motivation

When a user on a new platform builds your software, they cannot be sure that there are no platform specific issues unless they can run your test suite trivially - and they wont know to do that unless you tell them.

Strengths

More error reports are recieved from users, and releases have tests always run (as opposed to sometimes run).

Weaknesses

Slow test suites can make building slow which may be a political problem. This can be ameliorated by providing a way to select a subset of tests to run - and thus allow knowledgable users to disable them.

Builds that are done via IDE's may have difficulties integrating the test suite because IDE's often directly invoke the compiler and linker bypassing the regular mechanism of Make or other tools.

Conclusion

Sotware testing is a much neglected area in the field of software engineering. In this paper we have attempted to document techniques that we have been using for years. An important part of this is an attempt to start to build a testing terminology so that developers can talk about 'Placeholder result' and 'Finding holes with random data' and other knowledgeable developers will know what they are talking about.