



# NITTY GRITTY UNIT TESTING

ALL YOU NEED TO KNOW TO START UNIT TESTING YOUR CURRENT PRODUCT

## IT'S ONE THING TO SEE A PERSON DRIVE

a manual transmission car and have a theoretical understanding of what the pedals do and how to change gears. It's another thing to drive on the street, safely and without stalling. There are certain activities that, to execute successfully, a person needs to have not only knowledge of all the details that go into making them happen, but also some hands-on experience. Unit testing is one of them.

The good news is unit testing is a lot simpler than driving stick. But there are a lot of small details that game programmers need to get right to do it successfully.

Even after reading about unit testing and being convinced of its benefits, programmers are not always sure how to get started. It's not my intention to convince you of the many benefits of unit testing (I hope you're already convinced), but rather, to describe some very concrete tips to help you get your hands dirty right away.



ILLUSTRATION BY JONATHAN KIM

## GOALS OF UNIT TESTING

- »» Unit tests are used for many kinds of testing:
- »» **CORRECTNESS TESTING:** to check that the code behaves as designed.
- »» **BOUNDARY TESTING:** to check that the code behaves correctly in odd or boundary situations.
- »» **REGRESSION TESTING:** to check that the behavior of the code doesn't change unintentionally over time.
- »» **PERFORMANCE TESTING:** to check that the program meets certain minimum performance or memory constraints.
- »» **PLATFORM TESTING:** to check that the code behaves the same across multiple platforms.
- »» **DESIGN:** tests provide a way to advance the code design and architecture (usually referred as test-driven development or TDD).
- »» **FULL GAME OR TOOLS TESTING:** technically this is a functional test, not a unit test, because it involves the whole program instead of a small subset of the code, but a lot of the same techniques apply.

Some developers use unit tests only for one of the reasons listed above, while others use many kinds of tests for a variety of reasons. It's important

to recognize that because there are so many different uses for unit tests, no single solution is going to fit everybody. The ideal setup for some of those situations is going to be slightly different than for others, but the basics are the same for all of them.

When working with unit tests, the main goals are:

- »» **Spent as little time as possible writing a new test,**
- »» **Be notified of failing tests, and see at a glance which ones failed and why,**
- »» **Trust the tests (have them be consistent from run to run and robust in the face of bad code), and**
- »» **Create a testing framework.**

## TESTING FRAMEWORK

»» Most of us have created one-off programs in the past to test some particularly complicated code. It's usually a quick command line program that runs through a bunch of cases and asserts that the results were correct for each one. That's the most barebones way of creating unit tests.

Unfortunately, it's also a pain, and it misses on most of the unit testing goals described in the previous section. Creating a new program just to run a new set of unit tests is a nuisance, we have to go out of our way to run the



tests, and it's usually out of date faster than the latest Internet meme. Perhaps this explains in part why a lot of programmers have an initial aversion to writing unit tests at all.

If you're considering writing even a small unit test, you should use a unit-testing framework. A unit-testing framework removes all the busywork from writing unit tests and lets you spend your time on the logic of what to test. This doesn't mean that the framework writes the tests for you—be wary of any tool that claims to do that! Rather, a unit-testing framework is simply a small library that provides all the glue for running unit tests and reporting the results. However, you still need to use your brain and do (some of) the typing. Sorry.

A quick search will reveal plenty of unit testing frameworks in your language of choice to choose from, most of which are free and open source so you can rely on them and modify them to suit your needs.

For C/C++ and game development, I strongly recommend starting with `UnitTest++`. Charles Nicholson and I wrote that framework a few years ago specifically with games and consoles in mind. Many game teams have adopted it for their games and tools, and it has been used on a lot of different platforms, including current and last generation consoles, Windows, Linux, Mac, and the iPhone. In most situations, it's a straight drop-in to the project to get you up and running.

If you end up using a different testing framework, or if you code your own, the techniques described here still apply, even if the syntax is slightly different.

## HELLO TESTS

» Writing your first test is easy as pie. Try this sample code:

```
#include <UnitTest++.h>

TEST(MyFirstTest)
{
    int a = 4;
    CHECK(a == 4);
}
```

To run it, you need to add the following line to your executable

somewhere (we'll talk more about the physical organization of tests in a moment):

```
int failedCount =
UnitTest::RunAllTests();
```

Done. Easy, right?

When you compile and run the program, you should see the following output:

```
Success: 1 test passed.
Test time: 0.00 seconds.
```

Let's add a failing test:

```
TEST(MyFailingTest)
{
    int a = 5;
    CHECK(a == 4);
}
```

Now we get:

```
/fullpath/filename.
cpp:17: error: Failure in
MyFailingTest: a == 4
FAILURE: 1 out of 2 tests
failed (1 failures).
Test time: 0.00 seconds.
```

That's great, but if we're going to diagnose the problem, we need to know the value of the variable `a`, and all the test is telling us is that it's not 4. We can change the `CHECK` statement to the following:

```
TEST(MyFailingTest)
{
    int a = 5;
    CHECK_EQUAL(4, a);
}
```

And now the output will be:

```
/fullpath/filename.
cpp:17: error: Failure in
MyFailingTest: Expected 4
but was 5
FAILURE: 1 out of 2 tests
failed (1 failures).
Test time: 0.00 seconds.
```

That's much better. Now we see both the error information and the value of the variable. Virtually all unit testing frameworks include different types of `CHECK` statements to get

more information when testing floats, arrays, or other data types. You can even make your own `CHECK` statement for your own common data types, such as colors or lists.

As a bonus, if you're using an IDE, clicking on the test failure message should bring you automatically to the failing test statement.

## WHEN TO RUN

» When to run unit tests will depend on what's being tested and how long it takes to test it. In general, the more frequently you run the tests, the better. The sooner you get feedback that something went wrong (maybe before code is checked in and is disseminated to the rest of the team), the easier it will be to fix. On the flip side, realistically, building and running a set of unit tests takes a certain amount of time, so it's important to find the right balance between feedback frequency and time spent waiting for tests.

At the very least, all tests should run once a day during the nightly build process in your build server. (You have a build server, don't you? If not, stop right here and read *The Inner Product* column, "The Heartbeat of the Project," in the August 2008 issue before you continue.) It doesn't matter how long they take or how many different projects you need to run. Just add the tests to the build script, and hook their output into the build results.

On the other extreme, you can build your tests every time you build the project and execute them as a post-build step. That way, any time you make a change to a project, all tests will execute and you'll see if anything went wrong. This approach works great, but I wouldn't recommend using it if the tests add more than a couple of seconds to the incremental build time; otherwise, they'll be slowing you down more than they help.

Most developers will find that the approach that makes the most sense for them is somewhere between these two extremes, for example, taking a small and fast subset of tests that are more likely to break, and running those with

every build. Whenever code is checked in to your version control system, the build server can run those tests, plus a few others that are slower. And finally, at night, you can bring out the big guns and run those really long and thorough tests that take a few hours to complete.

One way to lower the impact of constantly running tests is to separate those tests into different projects, or, if your framework supports them, into different test suites, which allows you to decide which sets of tests to execute at runtime.

## REPORTING RESULTS

» If a unit test fails and nobody notices, is it really an error?

Just running tests isn't good enough. We have to make sure that someone sees failures when they occur and fixes the problems.

Most unit-testing frameworks will let you customize how the failure errors are reported. By default, they will probably be sent to `stdout`, but you can easily customize the framework to send them to debug log streams, save them to a file, or upload them to a server.

Even more important than seeing the actual error messages is automatically detecting whether there were any failures. After running all the tests, there is usually some way to detect how many tests failed. The program that was running the tests can detect failures, print an error message, and exit with an error code. That error code will propagate to the build server and trigger a build failure. Hopefully by now alarms are ringing across the office, and someone is on his way to fix the problem.

## PROJECT ORGANIZATION

» When people start down the unit test path, they often struggle to figure out how to physically lay out the unit tests. In the end, it really doesn't matter how you do it too much as long as 1) it makes sense to you, 2) the final build doesn't contain any tests, and 3) the unit tests are easy to build and run.

My personal preference is to keep unit tests separate

from the rest of the code. Usually, I end up creating one file of tests for every `cpp` file, for example, `FirstPersonCameraController.h` and `.cpp` should have a corresponding `TestFirstPersonCameraController.cpp`. Since I use this convention regularly throughout all my code, I have a custom IDE macro to toggle between a file and its corresponding test file. I also put all the tests in a separate subdirectory to keep them as physically separate as possible.

I prefer to break up my code into several static libraries for each major subsystem: graphics, networking, physics, animations, etc. Each of those libraries has a set of unit tests, but instead of compiling them into the library, I make a separate project that creates a simple executable program. That project contains all the unit tests and links against the library itself, and in its main entry point, it calls the function to run all unit tests and returns the number of failures. This way, tests stay separate from the library, but they're still very easy to build.

If all your code is organized into libraries, and your game is just a collection of libraries linked together, that's all you need. But most games and tools have a fair amount of code that you might want to test in the project itself. Since the game is an executable, you can't easily link against it from a different project like we did before. In this case, I build the unit tests into the game itself, and I optionally call them whenever a particular command-line parameter like `-runtests` is present. Just be sure to `#ifdef` out all the tests in the final build.

### MULTIPLATFORM TESTING

» Running the tests on the same PC that you built the code on is straightforward. But unless you're only creating games and tools for that platform, you will definitely want to run your tests on different platforms as well.

Unit tests are an invaluable tool for catching slight platform

inconsistencies caused by different compilers, architecture idiosyncrasies, or varying floating point rules.

Unfortunately, running unit tests on a platform other than your build machine is usually a bit more involved and not nearly as fast as doing it locally. Start by compiling the tests for the target platform. This is usually not a problem since you're already building all your code for that platform, and hopefully your unit testing framework already supports it. Next, upload your executable with the tests and any other data required to the target platform and run it there. Finally, get the return code back to detect if there were any failures.

Surprisingly, that final step is often the trickiest part of the process on a lot of console development kits. If getting the exit code is not a possibility, you need to get creative by parsing the output channel, or even waiting for a notification on a particular network port.

Some target platforms are more limited than others in both resources and C++ support. One reason `UnitTest++` is a good choice for games is that it requires minimal C++ features (no STL), and it can be trimmed down even further (no exceptions).

For example, on past projects, running tests on PlayStation 3 SPUs was extremely useful, but required stripping down the framework to the minimum number of features. It also took some finagling to fit the library code plus all the tests into the small amount of memory available. We ended up changing the build rules for the SPUs so each test file created its own SPU executable (or module). We then wrote a simple main SPU program that could load each module separately, run its tests, keep track of all the stats, and finally report them.

Running a set of unit tests on the local machine can be an almost instant process, but running them on a remote machine is usually much slower and can take up to 20 seconds just with the overhead of copying them and launching the program

remotely. For this reason, most developers will prefer to run tests on other platforms less frequently.

### NO LEAKING ALLOWED

» Finally, if you're going to have all this unit testing code running on a regular basis, you might as well get as much information out of it as possible. I personally find it invaluable to keep track of memory leaks around the unit test code. You'll have to hook into your own memory manager or use the platform-specific memory tracking functions. The basic idea is to get one memory status before running the tests and another one afterward. If there are any extra memory allocations, you've probably got a leak. In that case, you can report it as a failed build by returning the correct error code.

Watch out for static variables or singletons that allocate memory the first time they're used. They

might be reported as memory leaks even though it wasn't what you were hoping to catch. In that case, you can explicitly initialize and destroy all singletons, or better, not use them at all and keep your memory leak report clean.

You're now armed with all you need to know to set up unit tests into your project and build pipeline. Grab a testing framework, and get your feet wet today. ☺

---

**NOEL LLOPIS** has been making games for just about every major platform in the last ten years. He's now going retro and spends his days doing iPhone development from local coffee shops. Email him at [nllopis@gdmag.com](mailto:nllopis@gdmag.com).

## resources

### UnitTest++ framework

<http://unittest-cpp.sourceforge.net>

size matters

[www.rtpatch.com](http://www.rtpatch.com)

RTPatch and Pocket Soft are registered trademarks of Pocket Soft, Inc.