Unit testing: why bother?

Unit testing is the practice of testing the components of a program automatically, using a test program to provide inputs to each component and check the outputs. The tests are usually written by the same programmers as the software being tested, either before or at the same time as the rest of the software.

> *Most unit tests are written using a test framework–a set of library code designed to make writing and running tests easier. Nearly all programming languages have at least one commonly used test framework. But you don't have to use one to do unit testing. All you need is something that can run a bit of your code, feed it some inputs, and check the results.*

What's unit testing good for?

Some developers do test-driven development, a process in which the unit tests are written before the rest of the code. This ensures that the code will be correct as written, and it provides a useful framework for thinking about how the code should be designed, what interfaces it should provide, and how its algorithms might work.

Unit tests written during software development provide early sanity-checking for code. A program to read data from a file and calculate a result might fail not only because the basic algorithm is wrong, but because the input is read wrongly or the code fails to deal with unusual cases like short or empty datasets. These things are very easy to test automatically.

Unit tests are also vital to regression testing, the business of ensuring that new changes in code don't break things that were working before. Regression testing is obviously especially important in team working, but it is surprisingly easy to break your own code without noticing it, even if you are working on your own. And because regression testing is next to impossible to do satisfactorily by hand (it's simply too tedious), it's an obvious case for automation through unit tests.

Finally, unit tests provide documentation for other developers. Even if, like most of us, you are too lazy to document your code thoroughly, a small set of unit tests conveys a lot of useful information about how it is designed and how you expect it to work.

Some practical tips

You can do unit testing without using a test framework, and this can be a good way to get started if the thought of learning a test framework seems too complicated. A framework saves time in the long run, and in a company context you'd always have one, but for your own use it dœsn't have to be something you must learn before you can start. You can just run a bit of the code, check the results, spit out hideous abuse when the results are wrong. You can even do it from a shell script if your program is simple enough.

Tests should be small—don't think you have to use bulky real-world data. If your function gets the right median value for inputs with one value, a small odd number of values, and a small even number, it'll work for bigger inputs too. Think of tricky small cases, not easy large ones. Picking good test cases can be an interesting exercise in its own right.

Writing the tests first (i.e. practising test-driven development) can be a real help during development especially if you're not yet clear on how the code should actually work. When you find yourself getting stuck trying to visualise how an algorithm should work or how other code should interact with it, consider whether you can approach it from the other end by describing what its output should look like. Sketch it out in the tests, then write the code until the tests pass.

Whenever you find a bug in "finished code", add a test for it. Make sure the test fails in the buggy code and passes when it is fixed. Areas of code you've found bugs in are more likely to be fragile in general, and bugs that have already been found are relatively highly likely to reappear.

When writing a new test, include something to make sure it is being run. For example, make it fail deliberately when you first write it. It's quite common to discover that the reason your tests are all passing is that they're not being run at all. (Overlooked in the build file, private instead of public, mistyped the method name: every testing framework has its set of common mistakes.) So, always do something to make sure your test is really working.

Don't ship code with tests that fail, even if it dœsn't matter that they fail. It's not uncommon, particularly in test-driven development, to change your mind during design about which tests are correct or relevant, or to make an initial implementation that only covers some of the test suite. But that means you end up with failed tests that you don't actually care about. Remove (or, at very least, document) them: anyone running your tests should be able to assume that a failed test indicates broken code.

Consider using a code coverage tool to check how much of your code is actually being tested. Coverage dœsn't tell you everything: it only measures static execution paths, but it can give you some idea of things you might have missed altogether.

Read the full article at: http://soundsoftware.ac.uk/unit-testing-why-bother/