# Unit Test

## Tools

Each language has it’s own particular unit testing tool. My favorites are Junit for Java, rspec for Ruby, NUnit for .Net, Midje for Clojure, and CppUTest for C and C++.

Whatever unit testing tool you choose, there are a few basic features they all should support.

1. It should be quick and easy to run the tests. Whether this is done through IDE plugins or simple command line tools is irrelevant, as long as developers can run those tests on a whim. The gesture to run the tests should be trivial.

For example, I run my CppUTest tests by typing command-M in TextMate. I have this command set up to run my makefile which automatically runs the tests and prints a one-line report if all tests pass. JUnit and rspec are both supported by IntelliJ, so all I have to do is push a button. For NUnit, I use the Resharper plugin to give me the test button.

2. The tool should give you a clear visual pass/fail indication. It doesn’t matter if this is a graphical green bar or a console message that says “All Tests Pass.” The point is that you must be able to tell that all tests passed quickly and unam biguously. If you have to read a multiline report, or worse, compare the output of two files to tell whether the tests passed, then you have failed this point.

3. The tool should give you a clear visual indication of progress. It doesn’t matter whether this is a graphical meter or a string of dots as long as you can tell that progress is still being made and that the tests have not stalled or aborted.

4. The tool should discourage individual test cases from communicating with each other. JUnit does this by creating a new instance of the test class for each test method, thereby preventing the tests from using instance variables to communicate with each other. Other tools will run the test methods in random order so that you can’t depend on one test preceding another. Whatever the mechanism, the tool should help you keep your tests independent from each other. Dependent tests are a deep trap that you don’t want to fall into.

5. The tool should make it very easy to write tests. JUnit does this by supplying a convenient API for making assertions. It also uses reflection and Java attributes to distinguish test functions from normal functions. This allows a good IDE to automatically identify all your tests, eliminating the hassle of wiring up suites and creating error-prone lists of tests.

## Clean Unit Test

Chapter 9 – Clean Code – Robert C. Martin

# Component Test

## Tools

These tools are for testing components at the API level. Their role is to make sure that the behavior of a component is specified in a language that the business and QA people can understand. Indeed, the ideal case is when business analysts and QA can write that specification using the tool.

More than any other tool, component testing tools are the means by which we specify what done means. When business analysts and QA collaborate to create a specification that defines the behavior of a component, and when that specification can be executed as a suite of tests that pass or fail, then done takes on a very unambiguous meaning: “All Tests Pass.”

### Fitnesse

My favorite component testing tool is FitNesse. I wrote a large part of it, and I am the primary committer. So it’s my baby.

FitNesse is a wiki-based system that allows business analysts and QA specialists to write tests in a very simple tabular format. These tables are similar to Parnas tables both in form and intent. The tests can be quickly assembled into suites, and the suites can be run at a whim.

FitNesse is written in Java but can test systems in any language because it communicates with an underlying test system that can be written in any language. Supported languages include Java, C#/.NET, C, C++, Python, Ruby, PHP, Delphi, and others.

There are two test systems that underlie FitNesse: Fit and Slim. Fit was written by Ward Cunningham and was the original inspiration for FitNesse and it’s ilk. Slim is a much simpler and more portable test system that is favored by FitNesse users today.

### Other Tools

I know of several other tools that could classify as component testing tools.

* RobotFX is a tool developed by Nokia engineers. It uses a similar tabular format to FitNesse, but is not wiki based. The tool simply runs on flat files prepared with Excel or similar. The tool is written in Python but can test systems in any language using appropriate bridges.
* Green Pepper is a commercial tool that has a number of similarities with FitNesse. It is based on the popular confluence wiki.
* Cucumber is a plain text tool driven by a Ruby engine, but capable of testing many different platforms. The language of Cucumber is the popular Given/When/Then style.
* JBehave is similar to Cucumber and is the logical parent of Cucumber. It is written in Java.

# Integration Test

## Tools

Component testing tools can also be used for many integration tests, but are less than appropriate for tests that are driven through the UI.

In general, we don’t want to drive very many tests through the UI because UIs are notoriously volatile. That volatility makes tests that go through the UI very fragile. Having said that, there are some tests that must go through the UI—most importantly, tests of the UI. Also, a few end-to-end tests should go through the whole assembled system, including the UI.

The tools that I like best for UI testing are Selenium and Watir.

# Continuous Build

Lately I’ve been using Jenkins as my Continuous Build engine. It’s lightweight, simple, and has almost no learning curve. You download it, run it, do some quick and simple configurations, and you are up and running. Very nice.

My philosophy about continuous build is simple: Hook it up to your source code control system. Whenever anybody checks in code, it should automatically build and then report status to the team.

The team must simply keep the build working at all times. If the build fails, it should be a “stop the presses” event and the team should meet to quickly resolve the issue. Under no circumstances should the failure be allowed to persist for a day or more.

For the FitNesse project I have every developer run the continuous-build script before they commit. The build takes less than 5 minutes, so this is not onerous.

If there are problems, the developers resolve them before the commit. So the automatic build seldom has any problems. The most common source of automatic build failures turns out to be environment-related issues since my automatic build environment is quite different from the developer’s development environments.