

Games from Within

An engineering look
at the game
development process



Search:

ARTICLES

[Agile development](#) (3)

[Book reviews](#) (6)

[C++](#) (14)

[Conferences and events](#)
(11)

[General](#) (6)

[Project management](#) (10)

[Software engineering](#) (9)

[Test-Driven Development](#)
(7)

[Tools](#) (7)

MOST POPULAR

[Stepping Through the
Looking Glass: Test-
Driven Game](#)

[Development \(Part 1\)
Even More Experiments
with Includes](#)

[So You Want to Be a
Game Programmer?](#)

[Exploring the C++ Unit
Testing Framework Jungle](#)

UPCOMING EVENTS

November 8-9, Montreal,

Exploring the C++ Unit Testing Framework Jungle

By Noel Llopis
28 December 2004

One of the topics I've been meaning to get to for quite a while is the applicability of test-driven development in games. Every time the topic comes up in conversations or mailing lists, everybody is always very curious about it and they immediately want to know more. I will get to that soon. I promise!

In the meanwhile I'm now in the situation that I need to choose a unit-testing framework to roll out for my team at work. So, before I get to talk about how to use [test-driven development](#) in games, or the value of unit testing, or anything like that, we dive deep into a detailed comparison of existing C++ unit-testing frameworks. Hang on tight. It's going to be a long and bumpy ride with a plot twist at the end.

If you just want to read about a specific framework, you can go directly there:

Canada:

[Montréal Game Summit](#)

(Speaker)

Nov 30-Dec 2, Brisbane,

Australia:

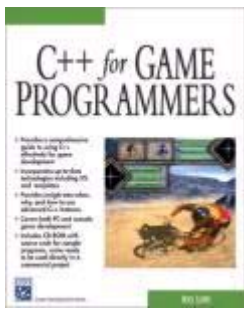
[Game Connect: Asia](#)

[Pacific](#) (Speaker)



SHAMELESS SELF-PROMOTION

Buy this fine book now from [Amazon](#).



LINKS

[Jim Tilander's Aurora](#)

[Agile Game Development](#)

[Al Patrick's Blog](#)

[GameDevBlog](#)

[GameArchitect.net](#)

[Joel on Software](#)

[Paul Graham](#)

[Gamasutra](#)

CURRENTLY READING

- [CppUnit](#)
- [Boost.Test](#)
- [CppUnitLite](#)
- [NanoCppUnit](#)
- [Unit++](#)
- [CxxTest](#)

Overview

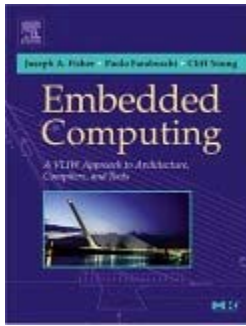
How do we choose a [unit-testing framework](#)

It depends on what

we're going to do with it and how we're going to use it. If I used Java for most of my work, the choice would be easy since [JUnit](#) seems to be the framework of choice for those working with Java. I don't hear them arguing over frameworks or proposing new ones very frequently, so it must be pretty good.

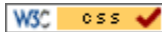
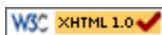
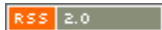
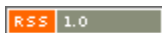
Unfortunately that's not the case with C++. We have our XUnit family member, CppUnit, but we're clearly not happy with that. We have more unit-testing frameworks than you can shake a stick at. And a lot of teams end up writing their own from scratch. Why is that? Is C++ so inadequate for unit testing that we have trouble fitting the XUnit approach in the language? Not





[About this site](#)

[About me](#)



like it's a bad thing, mind you. Diversity is good. Otherwise I would be stuck writing this under Windows and you would be stuck reading it with Internet Explorer. In any case, I'm clearly not the first one who's asked this question. [This page](#) tries to answer the question, and comes up with some very plausible answers: differences in compilers, platforms, and programming styles. C++ is not exactly a clean, fully supported language, with one coding standard.

A good way to start is to create a list of features that are important given the type of work I expect to be doing. In particular, I want to be doing test-driven development (TDD), which means I'm going to be constantly writing and running many small tests. It's going to be used for game development, so I'd like to run the tests in a variety of different platforms (PC, Xbox, PS2, next-generation consoles, etc). It should also fit my own personal TDD style (many tests, heavy use of fixtures, etc).

The following list summarizes the features I would like in a unit-testing framework in order of importance. I'll evaluate each framework on the basis of these features. Thanks to [Tom Plunket](#) for providing a slightly different view on the topic that helped me to re-evaluate the relative importance of the different features.

1. **Minimal amount of work needed to add new tests.** I'm going to be doing this all the time, so I don't want to do a lot of typing,

and I especially don't want to do any duplicated typing. The shorter and easier it is to write, the easier it'll be to refactor, which is crucial when you're doing TDD.

2. **Easy to modify and port.** It should have no dependencies with non-standard libraries, and it shouldn't rely on “exotic” C++ features if possible (RTTI, exception handling, etc). Some of the compilers we have to use for console development are not exactly cutting edge. To verify this one, I created a set of unit tests using each library under Linux with g++. Since most of the tests are written with Windows and Visual Studio in mind, it's not a bad initial test.
3. **Supports setup/teardown steps ([fixtures](#)).** I've adopted the style recommended by David Astels in his book [Test Driven Development: A Practical Guide](#) about using only one assertion per test. It really makes tests a lot easier to understand and maintain, but it requires heavy use of fixtures. A framework without them is ruled out immediately. Bonus points for frameworks that let me declare objects used in the fixture on the stack (and still get created right before the test) as opposed to having to allocate them dynamically.
4. **Handles exceptions and crashes well.** We don't want the tests to stop just because some code that was executed accessed some

invalid memory location or had a division by zero. The unit-testing framework should report the exception and as much information about it as possible. It should also be possible to run it again and have the debugger break at the place where the exception was triggered.

5. **Good assert functionality.** Failing assert statements should print the content of the variables that were compared. It should also provide a good set of assert statements for doing “almost equality” (absolutely necessary for floats), less than, more than, etc. Bonus points for providing ways to check whether exceptions were or were not thrown.
6. **Supports different outputs.** By default, I'd like to have a format that can be understood and parsed by IDEs like Visual Studio or [KDevelop](#), so it's easy to navigate to any test failures as if they were syntax errors. But I'd also like to have ways to display different outputs (more detailed ones, shorter ones, parsing-friendly ones, etc).
7. **Supports [suites](#).** It's kind of funny that this is so low in my priority list when it's usually listed as a prominent feature in most frameworks. Frankly, I've had very little need for this in the past. It's nice, yes, but I end up having many libraries, each of them with its own set of tests, so I hardly ever need this. Still, it certainly would be nice to have around

in case it starts getting slow to run the unit tests at some point.

Bonus: Timing support. Both for total running time of tests, and for individual ones. I like to keep an eye on my running times. Not for performance reasons, but to prevent them from getting out of hand. I prefer to keep running time to under 3-4 seconds (it's the only way to be able to run them very frequently). Ideally, I'd also like to see a warning printed if any single test goes over a certain amount of time.

Easy of installation was not considered a priority; after all, I only have to go through that once—it's creating new tests that I'm going to be doing all day long. Non-commercial-friendly licenses (like GPL or LGPL) are also not much of an issue because the unit test framework is not something we're going to link to the executable we ship, so they don't really impose any restrictions on the final product.

Incidentally, during my research for this article, I found that other people have compiled [lists of what they wish for in C++ unit-testing frameworks](#). It's interesting to contrast that article with this one and make a note of the differences and similarities between what we'd like to see in a unit test framework.

Ideal Framework

Before I start going over each of the major (and a few minor) C++ unit-testing frameworks, I

decided I would apply the philosophy behind test-driven development to this analysis and start by thinking what I would like to have. So I decided to write the set of sample tests in some ideal unit-testing framework without regard for language constrains or anything. In the ideal world, this is what I would like my unit tests to be like.

The simplest possible test should be trivial to create. Just one line to declare the test and then the test body itself:

```
TEST (SimplestTest)
{
    float someNum = 2.00001f;
    ASSERT_CLOSE (someNum, 2.0f);
}
```

A test with fixtures is going to be a bit more complicated, but it should still be really easy to set up:

```
SETUP (FixtureTestSuite)
{
    float someNum = 2.0f;
    std::string str = "Hello";
    MyClass someObject("somename");
    someObject.doSomethng();
}

TEARDOWN (FixtureTestSuite)
{
    someObject.doSomethingElse();
}

TEST (FixtureTestSuite, Test1)
{
    ASSERT_CLOSE (someNum, 2.0f);
    someNum = 0.0f;
}

TEST (FixtureTestSuite, Test2)
{
    ASSERT_CLOSE (someNum, 2.0f);
}
```

```
TEST (FixtureTestSuite, Test3)
{
    ASSERT_EQUAL(str, "Hello");
}
```

The first thing to point out about this set of tests is that there is a minimum amount of code spent in anything other than the tests themselves. The simplest possible test takes a couple of lines and needs no support other than a main file that runs all the tests. Setting up a fixture with setup/teardown calls should also be totally trivial. I don't want to inherit from any classes, override any functions, or anything. Just write the setup step and move on.

Look at the setup function again. The variables that are going to be used in the tests are not dynamically created. Instead, they appear to be declared on the stack and used directly there. Additionally, I should point out that those objects should only be created right before each test, and not before all tests start. How exactly are the tests going to use them? I don't know, but that's what I would like to use. That's why this is an ideal framework.

Now let's contrast it to six real unit-testing frameworks that have to worry about actually compiling and running. For each of the frameworks I look at the list of wanted features and I try to implement the two tests I implemented with this ideal framework. [Here is the source code for all the examples.](#)

CppUnit

CppUnit is probably the most widely used unit-testing framework in C++, so it's going to be a good reference to compare other unit tests against. I had used CppUnit three or four of years ago and my impressions back then were less than favorable. I remember the code being a mess laced with MFC, the examples all tangled up with the framework, and the silly GUI bar tightly coupled with the software. I even ended up creating a patch to provide console-only output and removed MFC dependencies. So this time I approached it with a bit of apprehension to say the least.

I have to admit that CppUnit has come a long way since then. I was expecting the worst, but this time I found it much easier to use and configure. It's still not perfect, but it's much, much better than it used to. The documentation is pretty decent, but you'll have to end up digging deep into the module descriptions to even find out that some functionality is available.

- 1. Minimal amount of work needed to add new tests.** This is one of the major downfalls of CppUnit, and, ironically, it's the highest-rated feature I was looking for. CppUnit requires quite a bit of work for the simplest possible test.

```
// Simplest possible test with CppUnit
#include <cppunit/extensions/HelperMacros>

class SimplestCase : public CPPUNIT_NS::
{
    CPPUNIT_TEST_SUITE( SimplestCase );
```

```

        CPPUNIT_TEST( MyTest );
        CPPUNIT_TEST_SUITE_END();

protected:
    void MyTest();
};

CPPUNIT_TEST_SUITE_REGISTRATION( Simplest

void SimplestCase::MyTest()
{
    float fnum = 2.00001f;
    CPPUNIT_ASSERT_DOUBLES_EQUAL( fnum, 2
}

```

2. Easy to modify and port. It gets mixed marks on this one. On one hand, it runs under Windows and Linux, and the functionality is reasonably well modularized (results, runners, outputs, etc). On the other hand, CppUnit still requires RTTI, the STL, and (I think) exception handling. It's not the end of the world to require that, but it could be problematic if you want to link against libraries that have no RTTI enabled, or if you don't want to pull in the STL.

3. Supports fixtures. Yes. If you want the objects to be created before each test, they need to be dynamically allocated in the setup () function though, so no bonus there.

```

#include <cppunit/extensions/HelperMacros.h>
#include "MyTestClass.h"

class FixtureTest : public CPPUNIT_NS::TestFixture
{
    CPPUNIT_TEST_SUITE( FixtureTest );
    CPPUNIT_TEST( Test1 );
    CPPUNIT_TEST( Test2 );
    CPPUNIT_TEST( Test3 );
    CPPUNIT_TEST_SUITE_END();
}

```

```

protected:
    float someValue;
    std::string str;
    MyTestClass myObject;

public:
    void setUp();

protected:
    void Test1();
    void Test2();
    void Test3();
};

CPPUNIT_TEST_SUITE_REGISTRATION( FixtureTest );

void FixtureTest::setUp()
{
    someValue = 2.0;
    str = "Hello";
}

void FixtureTest::Test1()
{
    CPPUNIT_ASSERT_DOUBLES_EQUAL( someValue,
    someValue = 0;

    //System exceptions cause CppUnit to stop
    //myObject.UseBadPointer();

    // A regular exception works nicely though
    myObject.ThrowException();
}

void FixtureTest::Test2()
{
    CPPUNIT_ASSERT_DOUBLES_EQUAL( someValue,
    CPPUNIT_ASSERT_EQUAL (str, std::string("Hello"));
}

void FixtureTest::Test3()
{
    // This also causes it to stop compilation
    //myObject.DivideByZero();

    // Unfortunately, it looks like the compiler
    // right at the beginning instead of the end
    // have to do it dynamically in the setUp()
    CPPUNIT_ASSERT_EQUAL (1, myObject.someValue);
    CPPUNIT_ASSERT_EQUAL (3, myObject.someValue);
    CPPUNIT_ASSERT_EQUAL (1, myObject.someValue);
}

```

2. Handles exceptions and crashes well.

Yes. It uses the concept of “protectors” which are wrappers around tests. The default one attempts to catch all exceptions (and identify some of them). You can write your own custom protectors and push them on the stack to combine them with the ones already there. It didn't catch system exceptions under Linux, but it would have been trivial to add with a new protector. I don't think it had a way to easily turn off exception handling and let the debugger break where the exception happened though (no define or command-line parameter).

3. Good assert functionality. Pretty decent.

It has the minimum set of of assert statements, including one for comparing floating-point numbers. It's missing asserts for less than, greater than, etc. The contents of the variables compared are printed to a stream if the assert fails, giving you as much information as possible about the failed test.

4. Supports different outputs. Yes. Has very-well defined functionality for “outputters” (which display the results of the tests), as well as “listeners” (which get notified while the tests are happening). It comes with an IDE-friendly output that is perfect for integrating with Visual Studio. Also supports GUI progress bars and the like.**5. Supports suites.** Yes.

Overall, CppUnit is frustrating because it's almost exactly what I want, except for my most wanted feature. I really can't believe that it takes so much typing (and duplicated typing at that) to add new tests. Other than that, the main complaint is the need for RTTI or exceptions, and the relative complexity of the source code, which could make it challenging to port to different platforms.

Boost.Test

Update: *I've revised my comments and ratings of the Boost.Test framework in light of the comments from Gennadiy Rozental pointing out how easy it is to add fixtures in boost.*

I'm a big fan of Boost, but I have to admit that it wasn't until about a year ago that I even learned that Boost was providing a unit testing library. Clearly, I had to check it out.

The first surprise is that Boost.Test isn't exclusively a unit-testing framework. It also pretends to be a bunch of other things related to testing. Nothing terribly wrong with that, but to me is the first sign of a “smell.” The other surprise is that it wasn't really based on the XUnit family. Hmm... In that case, it had better provide some outstanding functionality.

The documentation was top notch. Some of the best I saw for any testing framework. The concepts were clearly explained, and had lots of simple examples to demonstrate different features. Interestingly, from the docs I saw that Boost.Test

was designed to support some things that I would consider bad practices such as dependencies between tests, or long tests.

1. **Minimal amount of work needed to add new tests.** Almost. Boost.Test requires really minimal work to add new tests. It's very much like the ideal testing framework described earlier. Unfortunately, adding tests that are part of a suite requires more typing and explicit registration of each test.

```
#include <boost/test/auto_unit_test.hpp>
#include <boost/test/floating_point_comparison.hpp>

BOOST_AUTO_UNIT_TEST (MyFirstTest)
{
    float fnum = 2.00001f;
    BOOST_CHECK_CLOSE(fnum, 2.f, 1e-3);
}
```

2. **Easy to modify and port.** It gets mixed marks on this one, for the same reasons as CppUnit. Being part of the Boost libraries, portability is something that they take very seriously. It worked flawlessly under Linux (better than most frameworks). But I question how easy it is to actually get inside the source code and start making modifications. It also happens to pull into quite a few supporting headers from other Boost libraries, so it's not exactly small and self-contained.
3. **Supports fixtures.** Boost.Test eschews the setup/teardown structure of NUnit tests in favor of plain C++ constructors/destructors.

At first this threw me off for a loop. After years of being used to setup/teardown, and a fairly complex suite setup, I didn't see the obvious ways of using fixtures with composition.

Now that I've tried it this way I've come to like it almost better than setup/teardown fixtures. One of the great advantages of this approach is that you don't need to create fixture objects dynamically, and instead you can put the whole fixture on the stack.

On the downside, it's annoying to have to refer to the variables in the fixture through the object name. It would be great if they could somehow magically appear in the same scope as the test case itself. Also, it would have been a bit cleaner if the fixture could have been setup on the stack by the `BOOST_AUTO_UNIT_TEST` macro instead of having to explicitly put it on the stack for every test case.

```
#include <boost/test/auto_unit_test.hpp>
#include <boost/test/floating_point_comparison.hpp>
#include "MyTestClass.h"

struct MyFixture
{
    MyFixture()
    {
        someValue = 2.0;
        str = "Hello";
    }

    float someValue;
    std::string str;
    MyTestClass myObject;
};
```

```
BOOST_AUTO_UNIT_TEST (TestCase1)
{
    MyFixture f;
    BOOST_CHECK_CLOSE (f.someValue, 2.0f,
        f.someValue = 13;
}

BOOST_AUTO_UNIT_TEST (TestCase2)
{
    MyFixture f;
    BOOST_CHECK_EQUAL (f.str, std::string
    BOOST_CHECK_CLOSE (f.someValue, 2.0f

    // Boost deals with this OK and reports
    //f.myObject.UseBadPointer();
    // Same with this
    //myObject.DivideByZero();
}

BOOST_AUTO_UNIT_TEST (TestCase3)
{
    MyFixture f;
    BOOST_CHECK_EQUAL (1, f.myObject.s_cu
    BOOST_CHECK_EQUAL (3, f.myObject.s_ir
    BOOST_CHECK_EQUAL (1, f.myObject.s_ma
}
```

3. **Handles exceptions and crashes well.**

This is one of the aspects where Boost.Test is head and shoulders above all the competition. Not only does it handle exceptions correctly, but it prints some information about them, it catches Linux system exceptions, and it even has a command-line argument that disables exception handling, which allows you to catch the problem in your debugger on a second run. I really couldn't ask for much more.

4. **Good assert functionality.** Yes. Has assert statements for just about any operation you want (equality, closeness, less than, greater than, bitwise equal, etc). It even has support

for checking whether exceptions were thrown. The assert statements correctly print out the contents of the variables being checked. Top marks on this one.

5. **Supports different outputs.** Probably, but it's not exactly trivial to change. At least the default output is IDE friendly. I suspect I would need to dig deeper into the `unit_test_log_formatter`, but I certainly didn't see a variety of preset output types that I could just plug in.

6. **Supports suites.** Yes, but with a big catch. Unless I'm missing something (which is very possible at this point--if so make sure to let me know), creating a suite requires a bunch of fairly verbose statements and also requires modifying the test runner itself in main. Have a look at the example below. Couldn't that have been simplified to the extreme? It's not a big deal as this is my least-wanted requirement, but I wish I could label all the test cases in one file as part of a suite with a simple macro at the beginning of the file. Another minor shortcoming is the lack of setup/teardown steps for whole suites, which could come in really handy (especially if suite creation were streamlined).

```
#include <boost/test/unit_test.hpp>
#include <boost/test/floating_point_comparison.hpp>
using boost::unit_test::test_suite;
```

```
struct MyTest
{
```


[Feathers](#), the original author of CppUnit, got fed up with the complexity of CppUnit and how it didn't fit everyone's needs, so we wrote the ultra-light weight framework CppUnitLite. It is as light on the features as it is on complexity and size, but his philosophy was to let people customize it to deal with whatever they need.

Indeed, CppUnitLite is only a handful of files and it probably adds up to about 200 lines of very clear, easy to understand and modify code. To be fair, in this comparison I actually used a version of CppUnitLite I modified a couple of years ago (download it along with all [the sample code](#)) to add some features I needed (fixtures, exception handling, different outputs). I figured it was definitely in the spirit that CppUnitLite was intended, and if nothing else, it can show what can be accomplished by just a few minutes of work with the source code.

On the other hand, CppUnitLite doesn't have any documentation to speak of. Heck, it doesn't even have a web site of its own, which I'm sure is not helping the adoption rate by other developers.

1. **Minimal amount of work needed to add new tests.** Absolutely! Of all the unit-test frameworks, this is the one that comes closest to the ideal. On the other hand, it could be the fact that I've used CppUnitLite the most and I'm biased. In any way, it really fits my idea of minimum amount of work required to set up a simple test or even one

with a fixture (although that could be made even better).

```
#include "lib/TestHarness.h"

TEST (Whatever, MyTest)
{
    float fnum = 2.00001f;
    CHECK_DOUBLES_EQUAL (fnum, 2.0f);
}
```

2. **Easy to modify and port.** Definitely.

Again, it gets best of the class award in this category. No other unit-test framework comes close to being this simple, easy to modify and port, and at the same time having reasonably well separated functionality. The original version of CppUnitLite even had a special lightweight string class to avoid dependencies on STL. In my modified version I changed it to use `std::string` since that's what I use in most of my projects, but the change took under one minute to do. Also, using it under Linux was absolutely trivial, even though I had only used it under Windows before.

3. **Supports fixtures.** This is where the original CppUnitLite starts running into trouble. It's so lightweight that it doesn't have room for many features. This was an absolute must for me, so I went ahead and added it. I'm sure it could be improved to make it so adding a fixture requires even less typing, but it's functional as it stands. Unfortunately, it suffers from the problem that objects need to

be created dynamically if we want them to be created right before each test. To be fair though, every single unit-test framework in this evaluation has that requirement. Oh well.

```
#include "lib/TestHarness.h"
#include "MyTestClass.h"

class MyFixtureSetup : public TestSetup
{
public:
    void setup()
    {
        someValue = 2.0;
        str = "Hello";
    }

    void teardown()
    {
    }

protected:
    float someValue;
    std::string str;
    MyTestClass myObject;
};

TESTWITHSETUP (MyFixture,Test1)
{
    CHECK_DOUBLES_EQUAL (someValue, 2.0f);
    someValue = 0;

    // CppUnitLite doesn't handle system
    //myObject.UseBadPointer();

    // A regular exception works nicely t
    myObject.ThrowException();
}

TESTWITHSETUP (MyFixture,Test2)
{
    CHECK_DOUBLES_EQUAL (someValue, 2.0f);
    CHECK_STRINGS_EQUAL (str, std::string
}

TESTWITHSETUP (MyFixture,Test3)
{
    // Unfortunately, it looks like the 1
    // right at the beginning instead of
```

```
        // have to do it dynamically in the s
CHECK_LONGS_EQUAL (1, myObject.s_curr
CHECK_LONGS_EQUAL (3, myObject.s_inst
CHECK_LONGS_EQUAL (1, myObject.s_max
    }
```

2. **Handles exceptions and crashes well.**

The original CppUnitLite didn't handle them at all. I added minor support for this (just an optional try/catch). To run the tests without exception support it requires recompiling the tests with a special define turned on, so it's not as slick as the command-line argument that Boost.Test features.

3. **Good assert functionality.** Here is where CppUnitLite really shows its age. The assert macros are definitely the worst of the lot. They don't use a stream to print out the contents of their variables, so we need custom macros for each object type you want to use. It comes with support for doubles, longs, and strings, but anything else you need to add by hand. Also, it doesn't have any checks for anything other than equality (or closeness in the case of floating-point numbers).

4. **Supports different outputs.** Again, the original only had one type of output. But it was very well isolated and it was trivial to add more.

5. **Supports suites.** Probably the only framework that doesn't support suites. I never really needed them, but they would

probably be very easy to add on a per-file basis.

CppUnitLite is as barebones as it gets, but with a few modifications it hits the mark in all the important categories. If it had better support for assert statements, it would come very close to my ideal framework. Still, it's a worthy candidate for the final crown.

NanoCppUnit

I had never heard of NanoCppUnit until [Phlip](#) brought it up. From reading the feature list, it really appeared to be everything that I wanted CppUnitLite to be, except that it was better and ready to work out of the box.

The first point against NanoCppUnit is the awful “packaging” of the framework. If you thought that CppUnitLite was bad (not having a web page of its own), well, at least you could download it as a zip file. For NanoCppUnit you actually have to copy and paste the five files that make up the framework from a web page. I'm not kidding. That makes for some “lovely” formatting issues I might add. The documentation found in the web page wasn't exactly very useful either.

In any case, I continued my quest to get a simple test program up and running with NanoCppUnit. Out of the box (or out of the web page, rather) it's clearly aimed only at Windows platforms. I thought it would be trivial to fix, but changing it required more time than I thought at first (I

personally gave up when I started getting errors buried three macros deep into some assert statement). Unlike CppUnitLite, the source code is not very well structured at all, full of ugly macros everywhere, making it not trivial to add new features like new output types. Unless I'm totally mistaken, it even looks like it has sample code inside the test framework itself. Eventually I had to give up on running it under Linux, so my comments here are just best guesses by looking at the source code.

1. **Minimal amount of work needed to add new tests.** I think so. I'm not sure it's possible to create a standalone test that is part of a global suite, but at least creating a suite doesn't require manual registration of every test. This is (probably) the simplest possible test with NanoCppUnit.

```
struct MySuite: TestCase
{
};

TEST_(MySuite, MyTest)
{
    float fnum = 2.00001f;
    CPPUNIT_ASSERT_DOUBLES_EQUAL(fnum, 2
}
```

2. **Easy to modify and port.** Not really. Windows dependencies run deeper than it seems on the surface. The code is small, but it's messy enough that it's a pain to work with. I'm sure it can be ported with a bit of effort though since it's so small.

3. **Supports fixtures.** Yes. Setup and

teardown calls very similar to the modified version of CppUnitLite.

4. **Handles exceptions and crashes well.**

No idea since I wasn't able to run it. I see some try/catch statements in the code, but no way to turn them on or off. Probably no better than CppUnitLite.

5. **Supports different outputs.** Not really.

Everything is hardwired to use a stream that sends its contents to `OutputDebugString()` in Windows. I think the default output text is formatted to match the Visual Studio error format.

6. **Good assert functionality.** Yes. Good range of assert statements, including floating point closeness, greater than, less than, etc.

7. **Supports suites.** Yes. I don't know what's involved in just running a single suite though. Not a big deal either way.

One of NanoCppUnit's unique features is regular expression support as part of its assert tests.

That's very unusual, but I can see how it could come in handy. A few times in the past, I've had to check that a certain line of code has some particular format, so I had to `sscanf` it, and then check on some of the contents. A regular expression check would have done the job nicely.

Unfortunately, NanoCppUnit doesn't really live up to the standards of other frameworks. Right now it

feels too much as a work in progress, with too much missing functionality and not clearly structured code.

Unit++

The further along we get in this evaluation, the less Xunit-like the frameworks become. Unit++'s unique feature is that it pretends to be more C++-like than CppUnit. Wait a second, did I hear that right? More C++ like? Is that supposed to be a good thing? Looking back at my ideal test framework, it really isn't very much like C++ at all. Once I started thinking about that topic I realized that there really is no reason at all why the tests framework itself needs to be in C++. The tests you write need to be in the language of the code being tested, but all the wrapper code doesn't. That's a point that the next, and final, testing framework will drive home.

So, what does it mean to be more C++ like? No macros for a start. You create suites of tests by creating classes that derive from suite. That's the same thing we were doing in most other frameworks, really, but it was just happening behind the scenes. It really doesn't help me any to know that that is what I'm doing, and I would certainly not call it a "feature." As a result, tests are more verbose than they could be.

The documentation is simply middle-of-the-road. It's there, but it's not particularly detailed and it doesn't come loaded with examples.

- 1. Minimal amount of work needed to add new tests.** I'm afraid it gets failing marks for this. It requires manual registration of tests, and every test needs to be part of a suite. This makes adding new tests tedious and error prone (by writing a new test and forgetting to register it). I don't know about you, but with all the C++ cruft, I look at the code below and it's not immediately obvious what it does until I've scanned it a couple of times. The signal to noise ratio is pretty poor.

```
#define __UNITPP
#include "unit++.h"
using namespace unitpp;

namespace {

    class Test : public suite {
        void test1()
        {
            float fnum = 2.00001f;
            assert_eq("Checking float", fnum);
        }
    public:
        Test() : suite("MySuite")
        {
            add("test1", testcase(this, &Test::test1));
            suite::main().add("demo", this);
        }
    };
    Test* theTest = new Test();
}
```

- 2. Easy to modify and port.** So-so. It needs the STL and it pulls in some stuff like iostreams (which I remember having distinct problems with when I was working with STLPort). On the other hand the source code is relatively small and self-contained so it's certainly doable to port and modify if you're

willing to put in some time.

3. **Supports fixtures.** Another framework that I just can't see how to do fixtures with. Like Boost.Test, it seems to think that using the constructor and destructor for each class is all you need. A quick search for fixture or setup or teardown in the documentation doesn't reveal anything. I don't know if I'm totally missing something or if other people just write very different tests from me. I suppose I could create a new class for every fixture I want, put the setup in the constructor and the teardown in the destructor and inherit from it for every test case (and somehow figure out how to create an instance of that class and use it for each test run). It's probably possible, but it's not exactly trivial, is it? Again, the lack of fixtures puts this framework out of the running.
4. **Handles exceptions and crashes well.** Average. It manages to catch regular exceptions without crashing, but that's about it. No system exceptions in Linux. No way to turn it off for debugging.
5. **Supports different outputs.** I couldn't figure out how to do it from the documentation. There is probably a way to do it since it even supports GUI functionality , but it's not obvious (and there are no examples). Besides, by this point, having failed points 1 and 3, I wasn't really

motivated to spend a while learning the framework. Incidentally, this is one of the few frameworks whose default text output is not formatted correctly for IDEs like KDevelop.

6. **Good assert functionality.** It scrapes by with the minimum in this department. It provides equality and condition checks, but that's it. It doesn't even provide a float version of assert to check for “close enough.” At least it prints the contents of the variables to a stream correctly.

7. **Supports suites.** Yes, like most of them.

Overall, Unit++ is not really a candidate. Perhaps it's because it's not intended for the type of testing I intend to use it for, but it doesn't offer anything new over other frameworks and it has a lot of drawbacks of its own. The lack of fixtures is simply unforgivable.

CxxTest

After looking into a framework that tried to be different from XUnit (Unit++), I wasn't particularly looking forward to evaluating possibly the wackiest one of them all, CxxTest. I had never heard of it until a few days ago, but I knew that it required using Perl along the way to generate some C++ code. My spider senses were tingling.

Boy was I wrong!! Within minutes of using CxxTest and reading through its great documentation (the best by far), I was completely

convinced this was the way to go. This came as a complete surprise to me since I was ready to leave somewhat dissatisfied and pronounce a victor between CppUnit and CppUnitLite.

Let's start from the beginning. What's with the use of Perl and why is it different from CppUnit? Erez Volk, the author of CxxTest, had the unique insight that just because we're testing a C++ program, we don't need to rely on C++ for everything. Other languages, such as Java, are better suited to what we want to do in a unit-testing framework because they have good introspection (reflection) capabilities. C++ is quite lacking in that category, so we're forced to use kludges like manual registration of tests, ugly macros, etc. CxxTest gets around that by parsing our simple tests and generating a C++ test runner that calls directly into our tests. The result is simply brilliant. We get all the flexibility we need without the need for any ugly macros, exotic libraries, or fancy language features. As a matter of fact, CxxTest's requirements are as plain vanilla as you can get (other than being able to run Perl).

The code-generation step is also trivial to integrate into the regular build system. The wonderful documentation gives explicit step-by-step instructions on how to integrate it with make files, Visual Studio projects files, or [Cons](#). Once you have it set up, you won't even remember there's anything out of the ordinary going on.

Let's see how it stacks up against the competition.

1. **Minimal amount of work needed to add new tests.** Very good. It's almost as simple as the best of them. If I could nit-pick, I would have wished for an even simpler way to create tests without the need to declare the class explicitly. Since we're doing processing with a Perl script, there's no reason we couldn't have taken it a step beyond that and used a syntax even closer to my ideal test framework.

```
class SimplestTestSuite : public CxxTest
{
public:
    void testMyTest()
    {
        float fnum = 2.00001f;
        TS_ASSERT_DELTA (fnum, 2
    }
};
```

2. **Easy to modify and port.** CxxUnit requires the simplest set of language features (no RTTI, no exception handling, no template functions, etc). It also doesn't require any external libraries. It is also distributed simply as a set of header files, so there's no need to compile into a separate library or anything like that. Functionality is pretty well broken down and separated in the original source code, so making modifications should be fairly straightforward.
3. **Supports fixtures.** CxxUnit gets the “top of its class” label in this category. Not only does it support setup/teardown steps on a per-test level, but it also supports them at the suite

and at the world (global) level. Creating fixtures is pretty straightforward and just requires inheriting from a class and creating as many functions as you want starting with the letters “test.” To be really picky, I would have loved it if they had taken it a step further and, apart from simplifying the code a bit more, also inserted the setup and teardown code around the code for each test. That would have allowed us to work with those objects directly on the stack and their lifetime would have been managed correctly around each test. Oh well. Can't have everything.

```
#include "MyTestClass.h"

class FixtureSuite : public CxxTest::Test
{
public:
    void setUp()
    {
        someValue = 2.0;
        str = "Hello";
    }
    void tearDown()
    {
    }

    void test1()
    {
        TS_ASSERT_DELTA (someValue, 2.0f,
            someValue = 13.0f;

        // A regular exception works nice
        myObject.ThrowException();
    }
    void test2()
    {
        TS_ASSERT_DELTA (someValue, 2.0f,
            TS_ASSERT_EQUALS (str, std::string("Hello"));
    }
    void test3()
    {
        //myObject.UseBadPointer();
    }
}
```

```
        TS_ASSERT_EQUALS (1, myObject.s_c  
        TS_ASSERT_EQUALS (3, myObject.s_  
        TS_ASSERT_EQUALS (1, myObject.s_r  
    }  
  
    float someValue;  
    std::string str;  
    MyTestClass myObject;  
};
```

2. **Handles exceptions and crashes well.**

Great support. It catches all exceptions and prints information about them formatted like any other error (no system exceptions under Linux though). You can easily re-run the tests with a command-line argument to the Perl script to avoid catching exceptions and catch them in the debugger instead. It also gives you a custom version of every assert macro that lets you catch the exceptions yourself in case you ever need to do that.

3. **Supports different outputs.** Different outputs are supported by passing a parameter indicating which type of output you want to the Perl processing step. The default one (error-printer) was formatted correctly for IDE parsing, and you can use several others (including GUIs for those of you addicted to progress bars, a yes/no report, or a stdio one). Adding new output formatting sounds very straightforward and it's even covered in the documentation.

4. **Good assert functionality.** Again, it gets “top of its class” for this one. It has a whole suite of very comprehensive assert functions,

including ones for exception handling, checking predicates, and arbitrary relations. It even has a way to print out warnings which can be used to differentiate between two parts of the code calling the same test, or to print reminder “TODO” messages to yourself.

5. **Supports suites.** Yes. All tests are part of a suite.

Another feature supported by CxxUnit that I haven't had time to look into is some support for [mock objects](#). Anybody doing TDD knows the value of mock objects when it comes to testing the interactions between a set of objects. Apparently CxxUnit allows you to override global functions with specific mock functions (it gives an example of overriding `fopen()`). I don't think it helps any with regular classes; for those you're on your own.

So, what's not to like in CxxTest? Not much, really. Other than wishing that the test syntax were a bit tighter, the only thing to watch out for is what happens with large projects. If you follow the examples in the documentation, it will create a single runner for all the tests you give it. This can be problematic if you're going to be having thousands of tests, and then making one small change in one of them causes a full recompilation of all your code.

Update: After talking with Erez and re-checking the documentation, I realized this is already fully supported in CxxUnit. By default, when you

generate a test runner, it adds a main function and some global variables, so linking with other similar runners gives all sorts of problems.

However, it turns out you can generate a test runner with the `--part` argument, and it will leave out the main function and any other globals. You can then link together all the runners and have a single executable. I wonder if it would be worth going as far as creating a runner for every suite, or if it would be best to cluster suites together. Worth investigating at some point whenever I get enough tests to make a difference.

Conclusion

After going through all six C++ unit-testing frameworks, four stand out as reasonable candidates: CppUnit, Boost.Test, a modified CppUnitLite, and CxxTest.

Of the four, CxxTest is my new personal favorite. It fits very closely the requirements of my ideal framework by leveraging the power of an external scripting language. It's very usable straight out of the "box" and it provides some nifty advanced features and great assert functionality. It does require the use of a scripting language as part of the build process, so those uncomfortable with that requirement, might want to look at one of the other three frameworks.

CppUnit is a solid, complete framework. It has come a long, long way in the last few years. The major drawbacks are the relative verbosity for

adding new tests and fixtures, as well as the reliance on STL and some advanced language issues.

If what you need is absolute simplicity, you can do no wrong starting with CppUnitLite (or a modified version), and tweaking it to fit your needs. It's a well-structured, ultra-light framework with no external dependencies, so modifying it is extremely easy. Its main drawback is the lack of features and the primitive assert functionality.

If you're going to be working mostly on the PC, you don't expect to have to modify the framework itself, and you don't mind pulling in some additional Boost libraries, Boost.Test could be an excellent choice.

Should you roll your own unit-test framework? I know that Kent Beck recommends it in his book [*Test-Driven Development: By Example*](#), and it might be a great learning experience, but I just can't recommend it. Just as it's probably good to write a linked-list and a stack data structure a few times but I wouldn't recommend actually doing that in production code instead of using the ones provided in the STL, I strongly recommend starting with one of the three unit-testing frameworks mentioned above. If you really feel the need to roll your own, grab CppUnitLite and get hacking.

Whichever one you choose, you can really do no wrong with one of those three frameworks. The

most important thing is that you are writing unit tests, or, even better, doing test-driven development. To paraphrase Michael Feathers, code without unit tests is legacy code, and you don't want to be writing legacy code, do you?

 [unit_test_frameworks.tar.gz](#)

| [Comments \(22\)](#)

Enter your email address below to receive a notification whenever a new article is added.

All contents Copyright © 2003-2006 by Noel Llopis. All Rights Reserved.