Unci: a C++-based Unit-testing Framework for Intro Students

Don Blaheta
Longwood University
Farmville, VA, USA
blahetadp@longwood.edu

ABSTRACT

This paper describes Unci, a unit-testing language with a clean and minimal interface suitable for introducing beginning programming students to the ideas of unit testing and test-driven development. We detail why CppUnit, a common C++-based unit-testing framework, is not well-suited for beginners, and present Unci and explain how it addresses the weaknesses of CppUnit. Finally, we present a comparison of CS2 student performance in the two systems, showing that moving from CppUnit to Unci resulted in an approximate doubling in the proportion of students able to write an effective test suite for a lab assignment.

Categories and Subject Descriptors

D.2.5 [**Software engineering**]: Testing and debugging—*Testing tools*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Design, Human factors

Keywords

Test-driven development; unit testing; CS1; CS2

1. TEST-DRIVEN DEVELOPMENT IN INTRODUCTORY COURSES

Test-driven development (TDD) is by now a well-established framework for software engineering [1, 5], promoting the idea of developing an application's testing framework before and during, rather than after, the main code development; this is intended to solidify understanding of a project's requirements early in the project and improve communication about those requirements, as well as making the tests themselves less likely to be dependent on particular design choices, hence more likely to be robust. A number of educators have argued that TDD is pedagogically effective, even (or especially) in introductory programming courses [4, 7, 10]. Reasons for this mirror the reasons it is seen as effective as a general software engineering technique: for instance, students can better understand an example if it includes both an interface and an expected result [6], and having a written artifact of the *student's* understanding of the project requirements is helpful if the student needs to ask a question of a TA or professor (who can then quickly verify that the assumptions are correct before trying to answer the question) [11]. From a more cognitive perspective, an intro student—initially overwhelmed by the open-ended task of starting to write code—is well-served by having to first answer the more-constrained question of what a specific function call, with a specific, concrete parameter, should return. In addition, suites of discrete unit tests can be seen as an effective way to perform automatic grading of student work, decreasing grading load for the professor or teaching staff as well as giving the students more rapid feedback on their work [4].

Frameworks to support unit testing and test-driven development have been written for many programming languages. For Java, the original and most widespread is JUnit [8], although others exist to serve various specific niches, including at least one, tester [10], which was designed specifically for intro students. For C++, the unit testing landscape is much more fragmented. CppUnit [2], developed as a port of JUnit, is widespread but a variety of others exist.

At our institution, the introductory CS1/CS2 sequence is taught using C++. When deciding to integrate unit testing into these early courses, we looked at several of the available options, settling on CppUnit: with none of the options apparently targeting the intro programming student, we decided to at least choose one with some wider use.

2. PROBLEMS WITH CPPUNIT

In the Fall 2013 offering of our CS2 course, students were introduced to the concept of unit testing in week 3 (of 14); in their lab section during that week, they were given a template file similar to that shown in Figure 1 to work from, and instructions on how to adapt it to write test cases. Four of the remaining weekly labs contained specific additional instructions on CppUnit syntax or semantics, and two more of the labs did not give additional instructions but included "testing" among their rubric requirements.

We deemed it crucial to give a template for the students to use, not just the first time they were introduced to CppUnit

SIGCSE'15, March 4–7, 2015, Kansas City, MO, USA.

ACM 978-1-4503-2966-8/15/03 ...$15.00.
http://dx.doi.org/10.1145/2676723.2677228.

```
#include <cppunit/TestFixture.h>
#include <cppunit/TextTestRunner.h>
#include <cppunit/extensions/HelperMacros.h>
#include "Xxx.h"

class Xxx_test  : public CppUnit::TestFixture
{
  CPPUNIT_TEST_SUITE( Xxx_test );
  CPPUNIT_TEST( test_somethingMethod );
  CPPUNIT_TEST_SUITE_END();

  private:
    Xxx example1;

  public:
    void setUp()
    {
      example1 = Xxx (arg1, arg2);
    }

    void test_somethingMethod()
    {
      CPPUNIT_ASSERT_EQUAL ( 42, example1.something() );
    }
};

int main( int argc, char **argv)
{
  CppUnit::TextTestRunner runner;
  runner.addTest( Xxx_test::suite() );
  runner.run();
  return 0;
}
```

Figure 1: A minimal CppUnit file

but for the entire rest of the term. A minimal file using CppUnit to implement unit tests—if it is meant to stand alone—must in general contain:

- Three `#include` lines at the top
- A class that inherits from `CppUnit::TestFixture`
- A series of macro calls to build the test suite (where the macro arguments must be the name of the class itself as well as the names of each method designated as a test), and
- A four-line `main` function to bootstrap the test suite runner.

This boilerplate material must be present in addition to the declaration of any fixture variables, and a `setUp` method to give their initial values, and the test methods themselves. It was our hope that by providing a template file, the boilerplate could be abstracted away and the students could focus on the conceptually important part: what do objects of this type look like, and how should they behave?

There was, however, a great deal of friction involved in introducing CppUnit-based unit testing: problems that were not with the core concept of unit testing or with understanding of what the test cases ought to be, but with the framework itself. Some of these were relatively minor, such as misremembering which argument to `CPPUNIT_ASSERT_EQUALS` is the expression to check and which is the expected return value. But many were more major, causing subtle bugs or otherwise delaying or preventing them from addressing the "real" content of the lab.

For instance, a frequent student error was creating a test method and not adding it to the suite. This results in a syntactically valid class that compiles and runs, but doesn't execute that test—meaning the student either assumes a test is passing when it wouldn't, or notices the missing test and loses time trying to figure out what's wrong.

An even larger problem came when students inadvertently made errors in the macros constructing the test suite. Since C++ macros are based purely on text substitution, problems here result in error messages that are utterly incomprehensible to beginners. If a student writes `CPPUNIT_TEST` in place of `CPPUNIT_TEST_SUITE`—an easy mistake for a beginner who doesn't yet understand what these lines are doing—the resulting error message says "`'context' does not name a type`", which is unhelpful at best (although the error does at least indicate the correct line number). The reverse mistake is even worse: it causes the error "`expected '}' at end of input`" and points to the last line of the file. Other mistakes that yield unhelpful or misleading error messages include changing the header of a test method or omitting the inheritance from `TestFixture` in the class header. This sort of misdirection can cost a student hours of fruitless, wasted debugging time, and it prevents them from progressing with the lab.

A further set of problems arose from the fact that fixture values look just like (and technically are) instance variables. In most other circumstances, instance variables would be used for values that are meant to persist across method calls; but in the case of the test suite, this is almost exactly the opposite of what happens, since the variables are reset to their initial values before each test method is run. The reasons behind this are good and fairly interesting, but also subtle and somewhat sophisticated, and this is a source of confusion for students who have just been introducted to object-oriented design.

There are other C++-based unit testing frameworks, of course, but to a greater or lesser extent, they appear to suffer from the same problems in an intro context. For instance, UnitTest++[13] requires its own `#include` and boilerplate `main` function, albeit shorter than those for CppUnit, and works through the preprocessing system to create `SUITE` and `TEST` macros; the fixture, if any, is a user-defined class separate from the test suite (and must be named in the header to every test case). libunittest[9] is similar to UnitTest++ except that it can define its own `main`. Unit++[12] steers clear of the preprocessor but requires individual registration of each test, and uses some fairly exotic syntax (method pointers) and dispreferred techniques (global variables) to get all the pieces to talk to each other; it doesn't appear to support fixtures at all. xUnit++[14] leverages a number of C++11 features to offer extra power to the programmer, but they largely don't help the intro student; it is again macro-based, and fixtures must be built by hand and named at the top of each test case. CxxTest[3] takes the promising step of running as a preprocessor (so it doesn't need a boilerplate `main`) but again uses the C++ preprocessor for the assertions and requires substantial amounts of C++ code to wrap the tests; its fixtures are essentially identical to CppUnit's.

Many of these difficulties are more or less intrinsic to a library-based solution in a language like C++; the limited

```
#include "Xxx.h"

test suite Xxx_test
{
  fixture:
    Xxx example1 = Xxx(arg1, arg2);

  tests:
    test somethingMethod
    {
      check (example1.something()) expect == 42;
    }
}
```

Figure 2: A minimal Unci file

preprocessing capability in C++ itself doesn't let a library writer introduce new syntax or create constructs that would generate substantially reordered or restructured code. As a result, simply trying to fork the CppUnit development (or making a pull request to get changes into CppUnit proper) would not adequately address these issues.

3. A NEW UNIT-TESTING LANGUAGE

To address these concerns, we began developing Unci, a language specifically designed to be a clean interface for writing unit tests. Its syntax would be enough like C++ to feel "comfortable" to students working in that language, but would have the freedom to diverge from C++ when that would yield an improvement in clarity. The main design objectives for this new language were that:

- Users could include a fixture with arbitrary data and arbitrary number of distinct test cases
- Users could include arbitrary C++ code in tests
- It require minimal or no "boilerplate" code that is identical for every file
- Its syntax be accessible to beginners with no unit testing experience and relatively little programming background
- Its test syntax would clearly indicate what is being tested and what outcome is expected
- Any errors in Unci code would be flagged by the Unci compiler, with a helpful error message
- Any errors in embedded C++ code would be passed through to the C++ compiler, to yield the same error message as in other C++ code with the same mistake

The first two of these are present in CppUnit (and other testing libraries had various combinations of the first three), and it was important not to lose those central features in the course of designing the new language. The remainder were in service of our goal of producing a pedagogically appropriate introductory C++-based unit testing framework.

Figure 2 shows an Unci test file with exactly the same semantics as the CppUnit test file in Figure 1. The most obvious change is that all the parts of the CppUnit version that were either always the same or derivable from other parts of the file can be omitted in the Unci source, making it far shorter (and far less susceptible to typo bugs). In what remains, every piece is semantically important; and by using semantically-appropriate labels ("`test suite`" instead of "`class`", "`fixture`" instead of "`private`", etc), the beginner student can see and reinforce the connection to the terminology they're learning to talk about unit testing in general. Indeed, by separating the `fixture` section from the `tests` section and labelling them, we preserve a useful semantic distinction instead of having a fixture that is part `private` and part `public`, or a `public` section that includes both setup code and individual tests.

3.1 Parts of an Unci file

The top portion of an Unci file can contain `#include` directives, `using` directives (such as `using namespace std;`), and comments. Anything else is a syntax error; in particular, we wished to channel students away from declaring global variables in this context.

The remainder of the file is a block headed by `test suite` and given a name. The name is not significant but will be munged into a class name by preceding it with "`Unci_`".

The suite block contains two parts: the `fixture` and the `tests`. Both are optional, but if present they must occur in that order. Since the fixture is optional, it would be possible to write a suite comprised only of independent tests (perhaps in the weeks before fixtures are introduced). The reason the tests are optional is that in a data-first design pedagogy (as in Program By Design [11]) the students are expected to give examples of data first, before any tests are written, and it is useful to be able to compile at this stage.

Within the fixture, every variable must be separately declared in the form

type varname `=` *value-expr* `;`

This is true even if the type in question is a class type and the intended initial value is that provided by the default constructor. Because a class's default constructor is not automatically defined if any other constructor is defined, any code that implicitly requires the default constructor to exist may trigger a compiler error; this error is confusing to students just learning about default constructors (and implicit object construction and constructor overloading), because they don't understand what part of the code they've written is actually calling that constructor. By requiring an *explicit* call to the default constructor here (if that's what is intended), beginner programmers have a better understanding of why the compiler thinks they need to define it in the class definition.

For fixture values that cannot be constructed in a single expression (for instance, container data structures that need to be constructed empty and then have an `add` or `insert` method called), and for fixture data that requires cleanup, the user may define a `setup` block and/or a `teardown` block containing arbitrary C++ code, as shown in Figure 3.

The `tests` section (if present) must contain nothing but test blocks, each preceded by the word `test` and an arbitrary (but distinct) label. Some students were a little frustrated that they couldn't put fixture variable declarations between the test blocks, but this seemed to arise from a misunderstanding of the role of the fixture (that they are available to all tests, and that they reset to initial values before each test). As such, preventing anything but test blocks

```
fixture:
  Set<string>* onlyFoo = new VectorSet<string>();
  Set<int>* twoFive = new VectorSet<int>();

  setup
  {
    onlyFoo->add("foo");
    for (int i = 2; i <= 5; ++i)
      twoFive->add(i);
  }

  teardown
  {
    delete onlyFoo;
    delete twoFive;
  }
```

Figure 3: A sample Unci fixture

in the `tests` section seems to be a useful constraint as the students are developing their understanding of test-driven development.

The test blocks themselves will normally contain some number of check/expect statements, but these can be interspersed with almost completely arbitrary C++ code. The chief restriction is that `check` is a partially-reserved keyword and can't be used as variable or function name in a statement-initial position.

3.2 Check/expect statements

One of the major contributions of Unci is introducing to the C++ unit-testing ecosystem a new, uniform syntax specifically designed for testing: the `check`/`expect` statement. Regardless of what assertion is being made, these statements take the form

`check (` *expression* `)`

followed by an assertion about what is supposed to happen, headed by the keyword `expect`:

`expect ==` ***value***`;` A basic equality assertion. The given *value* must be of the same type as the *expression* being tested. This makes very explicit that the `==` operator is what is being used to perform the test.

`expect <` ***value***`;` By making the `==` explicit when that is wanted, we reserved a natural place in the syntax for the other comparison operators: `<`, `>`, `<=`, `>=`, and `!=`. Any of these can be used to make an assertion about an expected result.

`expect true;` Most testing libraries (including CppUnit) have a simple ASSERT form that represents an assertion that some expression is true, but it is helpful to make that expectation explicit. Intro students are often not yet comfortable with the idea of first-class boolean values; they find it strange to "assert an expression" but much more comfortable to "assert *that* an expression *is true*".

`expect false;` Many students at this level have yet to fully come to terms with the use of `not` to negate arbitrary boolean expressions, particularly in an assertion context. They also appreciate the symmetry between `expect true` and `expect false`, and this furthermore makes series of boolean check/expect statements easier to read and edit.

`expect about` ***value*** `+-` ***tolerance***`;` Floating point math is inexact, and test cases involving floating point numbers should properly include a tolerance on either side of the expected value. In this case `about` and `+-` are keywords and reinforce the idea that the testing operation is *not* using `==` (notwithstanding the fact that the analogous assertion in CppUnit is a three-parameter macro named `CPPUNIT_ASSERT_DOUBLES_EQUAL`, or that JUnit simply uses an `assertEquals` method with a third argument for this purpose).

```
tests:
  test contains
  {
    check (onlyFoo->contains("foo")) expect true;
    check (onlyFoo->contains("bar")) expect false;
    check (twoFive->contains(4)) expect true;
    check (twoFive->contains(7)) expect false;
  }
  test size
  {
    check (onlyFoo->size()) expect == 1;
    check (twoFive->size()) expect == 4;
  }
  test remove
  {
    twoFive->remove(5);
    check (twoFive->contains(2)) expect true;
    check (twoFive->contains(5)) expect false;
    check (twoFive->size()) expect == 3;
  }
  test unrelatedFloat
  {
    double frac = (1.0 / 6.0) * 10000;
    check (frac) expect about 10000.0 / 6 +- 0.00001;
  }
```

Figure 4: Sample Unci tests using fixture from Fig. 3

A sample `tests` section is shown in Figure 4, using the fixture from Figure 3 to test a few standard operations from the Set ADT as well as illustrating the use of the inexact "`expect about`" assertion.

This syntax is the one place we diverged from the most widespread unit-testing terminology, which usually uses some form of the word `assert` here. However, we wanted to be able to syntactically separate the testable expression from the expected outcome, and we wanted to avoid using established keywords with different meanings. (The C++ version of `assert`, while not technically a keyword, is a well-established part of the standard library, available in the `<cassert>` header.) Our usage of `check` and `expect` is inspired by, though not identical to, that seen in the Program By Design curriculum [10].

4. RESULTS

The initial roll-out of Unci was to the Spring 2014 sections of our CS2 course. The overall content of the course was

	N	No handin	Doesn't compile	No real test	Total no testing	Light tests	All one fn	Good tests	Total testing
F13/CppUnit	N=11	2 (18%)	1 (9%)	3 (27%)	55%	1 (9%)	1 (9%)	3 (27%)	45%
S14/Unci	N=27	3 (11%)	4 (15%)	0 (—)	26%	4 (15%)	5 (19%)	11 (41%)	74%

Table 1: Test suite quality on Lab 9

	N	No handin/ no suite	Doesn't compile	Fixture only	Total no testing	Some tests	All tests	Total testing
F13/CppUnit	N=11	2 (18%)	1 (9%)	4 (36%)	64%	0 (—)	4 (36%)	36%
S14/Unci	N=27	3 (11%)	2 (7%)	2 (7%)	25%	5 (19%)	15 (56%)	75%

Table 2: Test suite quality on Lab 11

unchanged from the previous term, and the student body comparable—in both cases coming directly out of a CS1 course the previous term—but all CppUnit-based content was replaced with Unci-based content. There was no longer a need to provide a template file for the students to copy whenever they wanted to create a new test suite, and the students were no longer getting stuck on misleading macro-related error messages.

In the assignment where unit testing was first introduced (the lab during week 3), there were fairly detailed instructions on what to type and how to build the test suite file. Perhaps as a result, both the CppUnit students and the Unci students were generally able to complete this by the end of the allotted week. A more interesting comparison can be seen in two assignments later in the term.

During week 9, the lab assignment was primarily focussed on the Set ADT: specifying what operations it should support, and then writing multiple implementations of it. Part of this process was to write appropriate unit tests for the data structure; but as the students had been practicing writing test cases for several weeks at this point, the lab instructions (during both terms) gave no detail on *how* to write the test suite. Student performance on the test suite portion of this lab diverged substantially between the two terms.

We coded all student submissions on this assignment, from both terms, into six categories (themselves grouped into two larger categories), presented in Table 1. A small number of students did not hand in the assignment at all. Of those who did, some had test suites that did not successfully compile, and others were technically able to make the file compile but couldn't figure out how to put any test cases in it. Students in these three categories clearly were not benefiting from test-driven development on this assignment (although in some cases they were able to successfully implement other parts of the assignment anyway).

The remainder of the students were able to implement at least some testing for their assignment. In some cases the testing was "light", in that there were multiple test functions or `test` blocks, but only with a single assertion—hence not a thorough test. A number of students implemented a series of statements and assertions that was reasonably thorough, but put all of them in a single function or block, thus not operating as a unit test and not taking advantage of the test fixture. Finally, some students were able to write multiple unit tests, covering multiple cases for most or all of the ADT operations. The proportion of students in the latter category went up considerably among the students who wrote their tests using Unci instead of CppUnit: only 27% of the students successfully navigated CppUnit to build a reasonably good test suite, while 41% were able to do so with Unci. The proportion of students who successfully did any testing at all on this assignment rose from 45% to 74%.

Two weeks later, students were given a lab involving binary trees and functions that operate on them, and once again they were expected to write a test suite from scratch with no new instructions on how to do so. Performance on this task is recorded in Table 2, with categories similar to those described above; here the third category refers to those students that successfully built binary trees for the test fixture but wrote no tests for them, and the fourth category covers students that wrote some tests but did not cover all five of the functions required for the lab. As with Lab 9, the best students in the CppUnit section were able to write test cases as requested, but the weak and middle students—who in many ways might benefit most from TDD—were unable to get any test cases written. In the Unci sections, however, most of the students were able to write all the necessary test cases and a wide majority succesfully wrote at least some test cases. The proportion of students able to run automated tests on their code here rose from 36% to 75%. The number of students in each group is low—11 in the CppUnit section, 27 in the Unci sections—but the performance difference is large enough to be persuasive.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented Unci, a new unit testing framework for C++ that is more accessible to introductory programmers than the alternatives. It provides a layer of abstraction so that one can write a test suite without understanding the implementation details of the testing engine itself; and the syntax it uses is powerful but constrained enough to steer inexperienced programmers toward good testing practices and (when possible) away from several easy-to-make beginner mistakes. When used in a CS2 programming course, students writing their tests in Unci were much more likely to be able to successfully write example data and test cases that would compile and test their code.

Unci is in many ways still what the entrepreneurs call a "minimum viable product". There are a number of areas where we plan to improve it. There are other assertion types that would be helpful, including array equality and containment, as well as testing that student code correctly throws exceptions. (Indeed, our parser already supports Unci syntax for these `expect` types, but the compiler does not yet

produce code for them.) It would be convenient to be able to compile multiple Unci test suites together into a single test executable. Currently, the Unci compiler outputs code that uses the CppUnit macro system, but it will be better to bypass the macros; it would perhaps eventually be good to bypass CppUnit entirely, although continuing to target CppUnit as an intermediate form may be helpful in the short term as we aim for integration with popular C++ IDEs. Finally, there are a few C++11 language features (notably the new curly-bracket-based uniform initialisation syntax) that the Unci parser does not correctly support, which needs to be fixed before those features start making their way into intro textbooks. We continue to use Unci in our CS2 course and consider it to be under active development.

We would also be interested in running larger-scale testing to confirm Unci's helpfulness for students in other C++-based intro curricula.

6. INSTALLATION AND AVAILABILITY

The code and installation instructions are available at our website.[1] The core tool is `unci-translate`, which reads `.u` files and compiles them into C++ code that makes use of the CppUnit library, printed to standard output. The more user-facing compilation tools are `uncic`, a shell script which compiles Unci files directly to object files (which include their own `main` but must be linked against the CppUnit library), and `compile`, which accepts both `.cpp` and `.u` files and dispatches them appropriately, producing an executable file. In our classes, the students are first introduced to the `compile` command as a sort of "one stop shop" for compiling their programs, but once they are introduced to `make` and the idea of a build manager, `uncic` is more appropriate.

The download also includes drop-in configuration files to set up syntax highlighting for Unci files in the Vim text editor.

[1] `http://cs.longwood.edu/~dblaheta/unci`

7. REFERENCES

[1] K. Beck. *Test-driven development: by example*. Addison-Wesley, Boston, 2003.

[2] CppUnit. Retrieved November 30, 2014 from `http://freedesktop.org/wiki/Software/cppunit/`.

[3] CxxTest. Retrieved November 30, 2014 from `http://cxxtest.com`.

[4] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *OOPSLA'03*, Anaheim, 2003.

[5] S. Hammond and D. Umphress. Test driven development: The state of the practice. In *ACMSE'12*, Tuscaloosa, 2012.

[6] D. S. Janzen and H. Saiedian. Test-driven learning: intrinsic integration of testing into the CS/SE curriculum. In *SIGCSE'06*, Houston, 2006.

[7] D. S. Janzen and H. Saiedian. Test-driven learning in early programming courses. In *SIGCSE'08*, Portland, 2008.

[8] JUnit. Retrieved November 30, 2014 from `http://junit.org`.

[9] libunittest. Retrieved November 30, 2014 from `http://libunittest.sourceforge.net`.

[10] V. K. Proulx. Test-driven design for introductory OO programming. In *SIGCSE'09*, Chattanooga, 2009.

[11] V. K. Proulx. Introductory computing: The design discipline. In I. Kalaš and R. Mittermeir, editors, *Informatics in Schools. Contributing to 21st Century Education*, volume 7013 of *Lecture Notes in Computer Science*, pages 177–188. Springer Berlin Heidelberg, 2011.

[12] Unit++. Retrieved November 30, 2014 from `http://unitpp.sourceforge.net/`.

[13] UnitTest++. Retrieved November 30, 2014 from `http://unittest-cpp.sourceforge.net`.

[14] xUnit++. Retrieved November 30, 2014 from `https://bitbucket.org/moswald/xunit`.