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

Don Blaheta
Longwood University
blahetadp@longwood.edu

6 March 2015

Unci: Intro C++ unit testing

Tests as a concrete specification

- "We agree you will do X"
- How will I know you have done X?
- How will *you* know you have done X?

Unci: Intro C++ unit testing

Tests as a progress report

- The contract includes 15 tests; how many are succeeding?
- Individual, easy-to-perform "unit" tests can be performed frequently
- Hard-to-perform tests will be postponed to the end

Unci: Intro C++ unit testing

Tests as a to-do list

- Specifications can change mid-project
- Client has additional requirements, clarifications
- Developer has "aha" moment: tricky edge case

Unci: Intro C++ unit testing

Example task: name processing

- Names are complicated (esp. globally)
- Analysis required to e.g. sort alphabetically
- “First” name, “Last” name
- Print last/sur-/family name in ALL CAPS

Unci: Intro C++ unit testing

Example task: name processing

Main task:

- `canonicalName` reformats a provided name into its preferred order but with the family name in all caps.

Example task: name processing

Main task:

- `canonicalName` reformats a provided name into its preferred order but with the family name in all caps

Subtasks:

- `firstName` extracts the "first name" from the provided name
- `lastName` extracts the "last name" from the provided name

Unci: Intro C++ unit testing

Showing my work: header file

```
1 #include <string>
2 using namespace std;
3
4 string canonicalName (string provided);
5 string firstName (string provided);
6 string lastName (string provided);
```

Unci: Intro C++ unit testing

Showing my work: "stubs"

```
 1 #include "names.h"
 2
 3 string canonicalName (string provided)
 4 {
 5   return "Not implemented yet";
 6 }
 7
 8 string firstName (string provided)
 9 {
10   return "Not implemented yet";
11 }
12
13 string lastName (string provided)
14 {
15   return "Not implemented yet";
16 }
```

Unci: Intro C++ unit testing

Examples of data

- "Adrienne Decker"
- "Kurt Eiselt"

Unci: Intro C++ unit testing

Examples of data

- "Adrienne Decker"
- "Kurt Eiselt"

Encode as:

```
string decker = "Adrienne Decker";
string eiselt = "Kurt Eiselt";
```

Unci: Intro C++ unit testing

Test cases

- What is the canonical, caps-ified version of our data examples?

Unci: Intro C++ unit testing

Test cases

- What is the canonical, caps-ified version of our data examples?
- For `decker`, it would be `"Adrienne DECKER"`

Unci: Intro C++ unit testing

Test cases

- What is the canonical, caps-ified version of our data examples?
- For `decker`, it would be `"Adrienne DECKER"`
  - So for `canonicalName(decker)`, I'd expect `"Adrienne DECKER"`

Unci: Intro C++ unit testing

Test cases

- What is the canonical, caps-ified version of our data examples?
- For `decker`, it would be `"Adrienne DECKER"`
  - So for `canonicalName(decker)`, I'd expect `"Adrienne DECKER"`
- For `eiselt`, it would be `"Kurt EISELT"`
  - So for `canonicalName(eiselt)`, I'd expect `"Kurt EISELT"`

Unci: Intro C++ unit testing

Test cases

- What is the canonical, caps-ified version of our data examples?
- For `decker`, it would be `"Adrienne DECKER"`
  - So for `canonicalName(decker)`, I'd expect `"Adrienne DECKER"`
- For `eiselt`, it would be `"Kurt EISELT"`
  - So for `canonicalName(eiselt)`, I'd expect `"Kurt EISELT"`

Encode as...

Unci: Intro C++ unit testing

The old way: CppUnit

```
 1 #include <cppunit/TestFixture.h>
 2 #include <cppunit/TextTestRunner.h>
 3 #include <cppunit/extensions/HelperMacros.h>
 4
 5 #include "names.h"
 6
 7 class test_names  : public CppUnit::TestFixture
 8 {
 9   CPPUNIT_TEST_SUITE( test_names );
10   CPPUNIT_TEST( test_basic );
11   CPPUNIT_TEST_SUITE_END();
12
13   private:
14     string decker = "Adrienne Decker";
15     string eiselt = "Kurt Eiselt";
16
17   public:
18     void test_basic()
19     {
20       CPPUNIT_ASSERT_EQUAL(string("Adrienne DECKER"),canonicalName(decker));
21       CPPUNIT_ASSERT_EQUAL(string("Kurt EISELT"),canonicalName(eiselt));
22     }
```

Unci: Intro C++ unit testing

The old way: CppUnit

```
22      }
23 };
24
25
26 int main( int argc, char **argv)
27 {
28   CppUnit::TextTestRunner runner;
29   runner.addTest( test_names::suite() );
30   runner.run();
31   return 0;
32 }
```

Unci: Intro C++ unit testing

A better way: Unci

```
 1 #include "names.h"
 2
 3 test suite names
 4 {
 5 fixture:
 6   string decker = "Adrienne Decker";
 7   string eiselt = "Kurt Eiselt";
 8
 9 tests:
10   test basic
11   {
12     check(canonicalName(decker)) expect == "Adrienne DECKER";
13     check(canonicalName(eiselt)) expect == "Kurt EISELT";
14   }
15 }
```

unci: Unit testing with a clean interface

Unci: Intro C++ unit testing

Existing frameworks: problems

- Such as

  CppUnit
  UnitTest++
  libunittest
  Unit++
  xUnit++
  CxxTest

- Boilerplate code
- Manual registration of test cases
- Macros (error messages)
- Fixture values as “instance variables”
- Intermediate/advanced C++ features (classes, templates, lambdas)

Unci: Intro C++ unit testing

Unci files

- Designed to have a minimum of "boilerplate"
- Two sections in most files:
  - List of examples ("test fixture")
  - List of test cases, grouped by unit

Unci: Intro C++ unit testing

Some unit tests

- Testing `canonicalName` with basic names

```
check (canonicalName(decker))
      expect == "Adrienne DECKER";

check (canonicalName(eiselt))
      expect == "Kurt EISELT";
```

Unci: Intro C++ unit testing

Some unit tests

- Other kinds of expectations

```
check (countValidItems(list))
      expect >= 1;

check (luftballon.isFlying())
      expect true;

check (triangle.hypotenuseLength())
      expect about 1.414 +- 0.001;
```

Unci: Intro C++ unit testing

Test-first design process

1. Write data examples

2. Write test cases

Unci: Intro C++ unit testing

Test-first design process

1. Write data examples
2. Write test cases
3. Run tests

Unci: Intro C++ unit testing

Test-first design process

1. Write data examples
2. Write test cases
3. Run tests
4. Do they all pass? If so, you're done
5. Otherwise, write a little piece of the program, and go to 3

Unci: Intro C++ unit testing

Test-first design process

1. Write data examples
2. Write test cases
3. Run tests
4. Do they all pass? If so, you're done
5. Otherwise, write a little piece of the program, and go to 3
   - Also, if you think of a new example, interrupt any step to go to 1
   - And if you think of a new test case, interrupt any step to go to 2

Unci: Intro C++ unit testing

Why Unci helps

Unci: Intro C++ unit testing

Why I thought Unci would help

Unci: Intro C++ unit testing

Why I thought Unci would help

- Getting bogged down in step 2 is a real drag
  - Less confusing format
  - Much better (more helpful) error messages
- "Interrupt" parts less disruptive
  - Add one thing, done

Unci: Intro C++ unit testing

Student performance

	No handin	Does not compile	No test	Total no testing
F13/CppUnit (N=11)	2 (18%)	1 (9%)	3 (27%)	55%

	Weak tests	Good tests	Total testing
F13/CppUnit (N=11)	2 (18%)	3 (27%)	45%

Unci: Intro C++ unit testing

Student performance

	No handin	Does not compile	No test	Total no testing
F13/CppUnit (N=11)	2 (18%)	1 (9%)	3 (27%)	55%
S14/Unci (N=27)	3 (11%)	4 (15%)	0 (—)	26%

	Weak tests	Good tests	Total testing
F13/CppUnit (N=11)	2 (18%)	3 (27%)	45%
S14/Unci (N=27)	9 (34%)	11 (41%)	74%

Unci: Intro C++ unit testing

Student performance

	No test suite	Does not compile	Fixture only	Total no testing
F13/CppUnit (N=11)	2 (18%)	1 (9%)	4 (36%)	64%

	Some tests	All tests	Total testing
F13/CppUnit (N=11)	0 (—)	4 (36%)	36%

Unci: Intro C++ unit testing

Student performance

	No test suite	Does not compile	Fixture only	Total no testing
F13/CppUnit (N=11)	2 (18%)	1 (9%)	4 (36%)	64%
S14/Unci (N=27)	3 (11%)	2 (7%)	2 (7%)	25%

	Some tests	All tests	Total testing
F13/CppUnit (N=11)	0 (—)	4 (36%)	36%
S14/Unci (N=27)	5 (19%)	15 (56%)	75%

Unci: Intro C++ unit testing

Thanks!

- Any questions?
- Any suggestions?
- `blahetadp@longwood.edu`