Testing ADTs in C++

Steven J Zeil

February 13, 2013


Outline


Outline I


Testing addContact

In our earlier test for addContact, we weren't particularly thorough:

```
TEST_F (MailingListTests, addContact) {
  mlist.addContact (jones);
  EXPECT_TRUE (mlist.contains("Jones"));
  EXPECT_EQ ("Jones", mlist.getContact("Jones").getName(
  EXPECT_EQ (4, ml.size());
}
```

- Missing functional case: adding a contact that already exists
- Missing boundary/special case: adding to an empty container
- Missing special cases: Adding to beginning or end of an ordered sequence


Adding Variety

Some of our concerns could be addressed by adding tests but varying the parameters:

```
TEST_F (MailingListTests, addExistingContact) {
   mlist.addContact (baker);
   EXPECT_TRUE (mlist.contains("Baker"));
   EXPECT_EQ ("Baker", mlist.getContact("Baker").getName(
   EXPECT_EQ (3, ml.size());
}
```


Generate and Check

A useful design pattern for producing well-varied tests is

```
for v: varieties of ADT {
   ADT x = generate(v);
   result = applyFunctionBeingTested(x);
   check (x, result);
}
```

- generate and check could be separate functions or in-line
  - depends on whether you can re-use in multiple tests


Generate and Check

```
for v: varieties of ADT {
   ADT x = generate(v);
   result = applyFunctionBeingTested(x);
   check (x, result);
}
```

- Most common "variety" would be size
  - Could also be different constructors (in which case you might not literally have a loop:

```
ADT x (constructor1);
result = applyFunctionBeingTested(x);
check (x, result);

ADT x2 (constructor2, ...);
result = applyFunctionBeingTested(x2);
check (x2, result);
```


Example: addContact

A more elaborate fixture will aid in generating mailing lists of different sizes:
fixture.cpp


Example: addContact - many sizes

testAdd1.cpp

- Here we generate mailing lists of a variety of sizes and add Jones to them
- At larger sizes, Jones is already in the list – functional case covered
- `sz == 0` covers one of our boundary/special cases


Example: addContact - ordering

testAdd2.cpp

- We an also explore adding to different positions (beginning, middle, end)
  - by varying which element we add


Outline I


Mutators and Accessors I

To test an ADT, divide the public interface into

- *mutators*: functions that alter the value of the object
- *accessors*: functions that "look at" the current value of an object
  - Occasional functions will fall in both classes


Organizing ADT Tests

The basic procedure for writing an ADT unit test is to

1. Consider each mutator in turn.
2. Write a test that begins by applying that mutator function.
   - Then consider how that mutator will have affected the results of each accessor.
   - Write assertions to test those effects.

Commonly, each mutator will be tested in a separate function.


Example: Unit Testing of the MailingList Class

mailinglist.h

Look at our `MailingList` class.

What are the mutators? What are the accessors?


MailingList Mutators

The mutators are

- the two constructors,


MailingList Mutators

The mutators are

- the two constructors,
- the assignment operator,


MailingList Mutators

The mutators are

- the two constructors,
- the assignment operator,
- `addContact`,


MailingList Mutators

The mutators are

- the two constructors,
- the assignment operator,
- `addContact`,
- both `removeContact` functions, and


MailingList Mutators

The mutators are

- the two constructors,
- the assignment operator,
- `addContact`,
- both `removeContact` functions, and
- `merge`.


MailingList Accessors

The accessors are

- `size`,


MailingList Accessors

The accessors are

- `size`,
- `contains`


MailingList Accessors

The accessors are

- `size`,
- `contains`
- `getContact`


MailingList Accessors

The accessors are

- `size`,
- `contains`
- `getContact`
- `operator==` and `operator<`


MailingList Accessors

The accessors are

- `size`,
- `contains`
- `getContact`
- `operator==` and `operator<`
- the output operator `operator«`


Unit Test Skeleton

Here is the basic skeleton for our test suite.

skeleton.cpp Now we start going through the mutators.


Testing the Constructors

The first mutators we listed were the two constructors. Let's start with the simpler of these.

Apply The Mutator

```
BOOST_AUTO_TEST_CASE ( constructor ) {
    MailingList ml;
      ⋮
```

First we start by applying the mutator.


Apply the Accessors to the Mutated Object I

Then we go down the list of accessors and ask what we expect each one to return.

```
BOOST_AUTO_TEST_CASE ( constructor ) {
    MailingList ml;
    BOOST_CHECK_EQUAL (0, ml.size());
    BOOST_CHECK (!ml.contains("Jones"));
    BOOST_CHECK_EQUAL (ml, MailingList());
    BOOST_CHECK (!(ml < MailingList()));
}
```

1. It's pretty clear, for example, that the `size()` of the list will be 0
2. `contains()` would always return false
3. The EQUAL test checks the `operator==`


Apply the Accessors to the Mutated Object II

4. We check for consistency of `operator<`

We can't check the accessors

- `getContact`
  - can't satisfy the pre-condition, and
- `operator«`
  - behavior unspecified


Testing the Copy Constructor

testCons1.cpp

1. We use the generate-and-check pattern.
2. Here we invoke the function under test.
3. The check function - we'll look at this in a moment
4. The second half of this is more subtle. I expect that copies are distinct. Once a copy is made, updating one object should not change the other.
   A failure suggests that we have done an improper shallow copy.

The Check Function for Copying

shouldBeEqual.cpp

1. Sizes should be equal
2. Boths lists should agree as to what they contain.
3. Relational ops - should be equal and not less/greater
4. Whatever they print, it should match


Testing the Assignment Operator

testAsst.cpp

- Very similar to testing the copy constructor

❶ But remember that assignment operators no only change the value on the left, but also return a copy of the assigned value.
  - We need to check both.

❷ Fortunately, we can re-use the check function developed for the copy constructor.


Testing addContact

testAdd.cpp

- Similar to version developed earlier, but now we go systematically through all the accessors
  - Needed to add checks on relational operators and output operator


And so on...

We continue in this manner until done.

fullTest.cpp


And so on...

We continue in this manner until done.

fullTest.cpp

- Yes, BTW, I did discover quite a few bugs in my MailingList code while putting this together.


Testing for Pointer/Memory Faults

Most destructors simply delete pointers, and pointer issues are particularly difficult to test and debug.

- One way that pointer/memory use can be tested is to employ specialized tools that replace the system functions for allocating and deallocating memory (`alloc` and `free`) with special versions that
  - keep track of all blocks of memory that have been allocated,
  - whether those blocks have been freed,
  - whether any block has been freed more than once, etc.


Tools for Testing Pointer/Memory Faults

- Purify is a well known commercial package for this purpose, but is not cheap.
- I have had good results with a free tool called LeakTracer.


Outline I


The Lowly Stub

- Most of what *Unit does is to provide a standard, convenient framework for writing *test drivers*.


The Lowly Stub

- Most of what *Unit does is to provide a standard, convenient framework for writing *test drivers*.
- But unit testing often requires *stubs* as well


The Lowly Stub

- Most of what *Unit does is to provide a standard, convenient framework for writing *test drivers*.
- But unit testing often requires *stubs* as well
  - supplying values/operations to the module under test


The Lowly Stub

- Most of what *Unit does is to provide a standard, convenient framework for writing *test drivers*.
- But unit testing often requires *stubs* as well
  - supplying values/operations to the module under test
  - processing outputs from the module under test


The Lowly Stub

- Most of what *Unit does is to provide a standard, convenient framework for writing *test drivers*.
- But unit testing often requires *stubs* as well
  - supplying values/operations to the module under test
  - processing outputs from the module under test
- We can avoid stubs by doing bottom-up integration


The Lowly Stub

- Most of what *Unit does is to provide a standard, convenient framework for writing *test drivers*.
- But unit testing often requires *stubs* as well
  - supplying values/operations to the module under test
  - processing outputs from the module under test
- We can avoid stubs by doing bottom-up integration
  - reduces independence of test suites


The Lowly Stub

- Most of what *Unit does is to provide a standard, convenient framework for writing *test drivers*.
- But unit testing often requires *stubs* as well
  - supplying values/operations to the module under test
  - processing outputs from the module under test
- We can avoid stubs by doing bottom-up integration
  - reduces independence of test suites
  - restricts development options


Where ADTs and Stubs Meet

In some ways, ADTs and OOP make it easier to do stubs.
A *mock object* is an object whose interface emulates a "real" class for testing purposes.
Mock objects may...

- be simpler than the real thing
- stand-in for classes not yet implemented
- decouple test suites
- offer mechanisms for data collection, thus improving testability


Example: Mailing List

In our MailingList tests, we relied on a *Contact* class.
Here is the real interface for *Contact*:
contactnmad.h


What Do We Need for the MailingList Test?

But in our tests, the only things we needed to do with contacts was

- create them with known names
- copy/assign them
- getName()
- compare them, ordering by name


A Mock Contact

So we made do with a simpler interface that

- made it easier to create and check contacts during testing
- but would still compile with the existing *MailingList* code

contact.h Replacing the *Name* and *Address* classes by simple strings.


OOP and Mocks

Object-oriented languages make it easier to create some mock objects

- Declare the mock as a subclass of the real interface
- Override the functions as desired

Example: for an application that drew graphics using the Java *java.awt.Graphics* API, I created the following as a mock class: **GraphicsStub.java** It basically writes each successive graphics call into a string, that can later be examined by unit test cases.


OO Mock Example

Here is a test using that mock:

linesegTest.java

- The (main) mock graphics object is created here.
- The function being tested in called here.
- Then, assertions like this one can test the info recrded in the mock object.


Mock Frameworks I

Increasingly, unit test frameworks are including special support for mock object creation.

- For example, the Google Mock Framework provides support for creating mock objects that automatically keep a log of calls made to them.
- And adds special functions and test assertions to both test that the expected calls were made and controlling what results will be returned from those calls.
  E.g., the following assertion says that a function `g.foo(x)` will be called 5 times, with x being 1 and 2 one time each.

Mock Frameworks II

```
EXPECT_CALL(turtle, foo(_))
    .TIMES(3)
    .WillRepeatedly(Return(150));
EXPECT_CALL(turtle, foo(1))
    .WillOnce(Return(100));
EXPECT_CALL(turtle, foo(2))
    .WillOnce(Return(200));
```

It also specifies what value should be returned on each call.

- If the function is called 6 times, or if one or more of the 5 expected calls are not made before the test case ends, the EXPECT_CALL assertion fails, just like other test assertions.


Outline I


Be Pro-active

Ideally, we have made it easier to write self-checking unit tests than to write the actual code to be tested.

- The goal is to encourage a philosophy of writing the tests before wirting the code.


Debugging: How Can You Fix What You Can't See?

The test-first philosophy is easiest to understand in a maintenance/debugging context.

- Before attempting to debug, write a test that reproduces the failure.
- How else will you know when you've fixed it?
- From a practical point of view, debugging generally involves running the buggy input, over and over, while you add debugging output or step through with a debugger.
  - You want that to be as easy as possible.
  - Doing this at the unit level, instead of in the context of the entire system, is often a dramatically better use of your time and effort.


Test-Writing as a Design Activity

Every few years, software designers rediscover the principle of writing tests before implementing code.
Agile and TDD (Test-Driven Development) are just the latest in this long chain.

- Writing tests while "fresh" yields better tests than when they are done as an afterthought
- Thinking about boundary and special values tests helps clarify the software design
  - Reduces the number of eventual faults actually introduced
- Encourages designing for *testability*
  - Making sure that the interface provides the "hooks" you need to do testing in the first place.