EECS 211
Unit Testing Notes

[Jump to Writing Clear Tests,
What about Private Functions?]

Testing

The typical approach to testing code is to write it, compile it, run it, and see if it seems to work. This is barely better than not testing at all. Doing serious testing this way is tedious and way too dependent on the patience, energy, skills, and fortitude of the programmer. Even worse, it depends on the coding being done on schedule -- and that never happens. As a result, there's just no time or incentive to do serious testing.

Many companies hire people to do nothing but run tests. These are the Quality Assurance (QA) teams. It's a job often given to interns and first-year co-op students. Scripts are written telling the QA team what to do and what to look for. This approach makes sense for things like checking the look and feel of an interface, but it's still inconsistent, expensive, and prone to error.

There's a better way that turns the code then test process on its head.

Test-Driven Development

Test-Driven Development (TDD) means that you write the tests **before** you write the code to pass the tests. By reversing the traditional order of code then test, you write code that is:

- better tested, because you have a larger library of tests, written at the start of development, rather than at the end when you're tired and pressed for time
- better designed, because you think about how it will be used first
- better documented, because the tests specify unambiguously what the code is supposed to do in every situation

Unit Testing Frameworks

For testing code thoroughly, quickly and consistently, the best approach is unit testing. A unit test is code that tests other code. Instead of running your program yourself, you run the unit test code. The test code will report any problems.

To make it easy to write and run unit tests, programmers have developed libraries for testing. While there have been many such libraries for decades, the JUnit library for Java has become the model for unit testing packages in a number of different

languages.

In this course, we'll use CppUnit, one of several C++ unit testing packages inspired by JUnit. For a list of many other options for C and C++ unit testing, see here.

Writing Good Tests

The set of tests used to test an application is commonly called a *test suite*. Any one test is usually neither good nor bad. What's important is the entire suite of tests.

What makes a test suite good? Two things: **coverage** and **clarity**. Many books and websites talk about test coverage. Clarity is less discussed.

Test Coverage

In general, every test should have a reason for being. Writing tests for 20 randomly picked pieces of data is not a good way to go. Such tests are redundant, inefficient, and incomplete.

Test coverage is a common metric for a test suite. There are formal definitions of different kinds of test coverage, but we'll use it here informally to mean the thoroughness of the test suite. Does the suite test for all possible things that might go wrong? You want a test suite that has good coverage and few if any redundant tests.

> For example, the rules for leap years are a little more complex than just "divisible by 4." Years divisible by 100 are not leap years, but years divisible by 400 are. A good test suite would check at least one year for each the following cases:
>
> - not divisible by 4
> - divisible by 4 but not 100
> - divisible by 100 but not 400
> - divisible by 400

Another important thing to write tests for are boundary cases. A boundary case is a situation at or very near the edge of some change in program behavior. "Off-by-one" errors are very common in programming.

For example, a common task is to convert numeric scores to letter grades, with A for 90 to 100, B for 80 to 89, and so on. Tests should be written for

- scores right on the boundaries, i.e., 100, 90, 89, 80, ...
- scores next to the boundaries, i.e., 88, 91, ...
- special cases, like 0
- scores past boundaries, like 101

Finally, always be sure to test with realistic typical values. If doing a gradebook, do grades that really occur. If a bank account, use account amounts that really occur. Testing with unrealistic values (1, 2, and 3, for example for grades or bank accounts) has two problems. First, it confuses maintainers who have to wonder if the variables are holding something different than what they claim to hold. Second, there may be issues that only arise with the kinds of values that normally occur, e.g., course names with colons, hyphens and spaces, or large integers.

Test Clarity

Unit tests are code. All code should be clear about what it is doing. It should be easy for a programmer reading your test suite to know what things it checks for, and easy to know where to add new tests.

Test Names

The best way to communicate what's being tested is through *well-named highly-focused* test functions. You don't want one big test function with line after line of assertions. Each test function should test for a particular situation that the application you're building needs to handle. The name of the function should communicate what situation it checks.

Some situations may affect several different aspects of an application. For example, when testing for an empty grade book, you'd probably want to separate out things like `testCalculateAverageNoGrades()` from `testPrintNoGrades()`.

As an example of what experienced test-driven developers do, here are the function names one programmer used for an employee application using PyUnit for Python. Note how even without reading the code, you get a good idea of what's being checked for, and how focused each test is:

```
testaccurately_report_hours_worked
testreports_active_status_once_hired
testreports_inactive_status_when_terminated
testrefuses_invalid_phone_numbers
testallows_nonwestern_names
testaccepts_long_names
testaccepts_unicode_name
testaccepts_unicode_in_addresses
testshould_reportsubordinate_employees
testshould_reportsupervisors_correctly
```

```
testaccepts_valid_birthdate
testrefuses_invalid_date_as_birthdate
testrefuses_future_birthdate
testreports_years_of_service
testrounds_years_of_service_downward
testrounds_months_of_service_downwards
testreports_vesting_based_on_months_of_service
testreports_role
testaccess_to_PII_requires_admin_access
testallows_variations_on_email_addresses
testallows_mutiple_phone_numbers
```

Test Organization

The worst -- but distressingly common -- approach to writting test code is this:

```
Account a1; // default limit of 2000
Account a2(1000);
Account a3(10000);
Account a4;
a4.setCreditLimit(5000);

CPPUNIT_ASSERT_EQUAL(2000, a1.creditLimit());
CPPUNIT_ASSERT_EQUAL(1000, a2.creditLimit());
CPPUNIT_ASSERT_EQUAL(10000, a3.creditLimit());
CPPUNIT_ASSERT_EQUAL(5000, a54.creditLimit());
```

With this code, you can't tell at a glance why the numbers expected are the right ones, or what's being tested for.

Put the assertions as close to the operation being tested as possible. That makes it easy to see what's being test. The ideal is to have the operation inside the assertion, e.g.,

```
CPPUNIT_ASSERT_EQUAL(10000, account.getCreditLimit());
```

With operations called for side-effect, put the assertion immediately following the operation:

```
account.setCreditLimit(200000);
CPPUNIT_ASSERT_EQUAL(20000, account.getCreditLimit());
```

This goes for assertions about constructors too! So the above bad code should've been written like this:

```
Account a1; // default limit of 2000
CPPUNIT_ASSERT_EQUAL(2000, a1.creditLimit());

Account a2(1000);
CPPUNIT_ASSERT_EQUAL(1000, a2.creditLimit());

Account a3(10000);
CPPUNIT_ASSERT_EQUAL(10000, a3.creditLimit());

Account a4;
a4.setCreditLimit(5000);
CPPUNIT_ASSERT_EQUAL(5000, a54.creditLimit());
```

What about Private Functions?

A fairly common question in unit testing is "How do I unit test private functions?" A

fairly standard response is "Don't do that." Opinions differ on this, but I think there are some good reasons why not to do it, or, if you do, why you have to be extra careful how you do it.

The first argument against unit testing private functions is that private functions are implementation decisions. As such, they should not be locked down. They should be open to change. In fact, the motto of Extreme Programming is "Embrace change." Unit testing supports change in that your code has 100's or 1000's of tests to help ensure that a change you make doesn't break anything."

Unit testing private functions makes them harder to change. You not only have to change the code but most likely you have to change the tests. For example, a private function to parse arithmetic expressions might have used characters to represent operators. Then later you decide it's better to use enumerated constants. Bam! Tests break, even though the public functions still pass all their tests.

The second argument is that unit testing private functions puts the focus on the wrong place. The question is not "does this private function do the right thing?" The real question is how it affects the public API. For example, suppose we had a rational number class. A common internal function for such a package is a greatest common divisor (GCD) function, to help reduce fractions to simplest terms. Mathematically, the GCD should always be positive, even if the arguments are negative. Suppose our GCD function gets that wrong. Does it matter? *Only if* the tests for the public rational number functions are wrong. Otherwise, it doesn't and shouldn't matter. We shouldn't be spending time to make the GCD function pass those tests.

The third argument is that unit testing private functions can, in many languages, force you to make functions public (or protected) so that they're available for testing. That's clearly bad. Depending on the language being used, there may be alternatives for this, but most require some extra effort or change in normal programming practice. This is not a zero-cost option, and can easily make maintenance more problematic.
