CS1632, LECTURE 6: UNIT TESTING, PART 2

BILL LABOON

HOW TO TEST THIS METHOD?

```
def double_me x
   x * 2
end
```

```
// Perhaps something like this…

def test_zero
   assert_equal 0, double_me(0)
end

def test_positive
   assert_equal 20, double_me(10)
end

def test_negative
   assert_equal -8, double_me(-4)
end
```

HOW ABOUT THIS?

```
class Foo
  attr_accessor :q1, :q2

  def initialize
    @q1 = 0
    @q2 = 0
  end

  def bar
    @q1 += 1
    @q2 -= 1
  end
end
```

```
# Note we are checking _side effects_ of the bar() method!
# This is allowed, although harder to test

def test_init
  f = Foo.new
  assert_equal 0, f.q1
  assert_equal 0, f.q2
end

def test_one_bar
  f = Foo.new
  f.bar
  assert_equal 1, f.q1
  assert_equal -1, f.q2
end
```

OK, HOW ABOUT THIS?

```
class Duck
  # d should be a Duck object, q is truthy or falsy
  def do_duck_stuff(d, q)
    if q
      d.quack
    else
      d.eat
    end
  end
end
```

WE NEED MORE ADVANCED TECHNIQUES

- Dummies
- Doubles
- Stubs
- Mocks
- Verification

TEST DOUBLES

- “Fake” objects you can use in your tests
- They can act in any way you want – they do not have to act exactly as their “real” counterparts

EXAMPLES

1. A doubled database connection, so you don't need to actually connect to the database
2. A doubled File object, so you can test read/write failures without actually making a file on disk
3. A doubled RandomNumberGenerator, so you can always produce the same number when testing

DOUBLES HELP KEEP TESTS LOCALIZED

- They let you test only the item under test, not the whole application, allowing you to focus on the current item.
- Remember, double objects of classes that the current class depends on; don't double the current class!
  - That would mean you are making a “fake” version of what you are testing

EXAMPLE

```
def test_delete_front_item
  ll = LinkedList.new
  ll.addToFront Node.new
  ll.deleteFront
  assert_equal nil, ll.getFront()
end
```

EXAMPLE

```
def test_delete_front_item
  ll = LinkedList.new
  ll.addToFront Minitest::Mock.new("mocked node")
  ll.deleteFront
  assert_equal nil, ll.getFront()
end
```

STUBS

Doubles are “fake objects”.

Stubs are “fake methods”.

STUBS

Stubbing a method says "hey, instead of actually calling that method, just do whatever I tell you.“

"Whatever I tell you" is usually just return a value.

EXAMPLE

```
def quack_alot(d, n)
  num_quacks = 0;
  n.times do
    num_quacks += d.quack
  end
  num_quacks
end
```

DEPENDENCY ON OTHER CLASSES == BAD

- Why?
  - If a failure occurs in a test, where is the problem?
    - This method?
    - Other method?
    - Yet another method that another method called?
    - Cannot be *localized*
  - What if quack() hasn’t been completed yet?

DOUBLE THE CLASS, STUB THE METHOD

```
def test_quack_alot

   mock_duck = Minitest::Mock.new("duck")
   def mock_duck.quack; 1; end
   val = quack_alot(mock_duck, 100)
   assert_equal val, 100

end
```

WE HAVE MADE THE TEST INDEPENDENT

We don't care about how quack() works, or Duck works, only our quackAlot() method.

If something goes wrong in Duck.quack, tests on THAT method will fail, not here.

Tests that break easily are called BRITTLE. If your tests depend on lots of other code working correctly, they are very brittle, and also make it difficult to know where the error actually is.

UNIT TESTS != SYSTEM TESTS

- The manual testing that you've already done is a system test – it checks that the whole system works
- This is not the goal of unit tests! Unit tests check that very small pieces of functionality work, not that the system as a whole works together.
- A proper testing process will include both –unit tests to pin down errors in particular pieces of code, system tests to check that all those supposedly-correct pieces of code work together.

VERIFICATION

- Note that this is different from the "verification" in "verification and validation". It’s also different than the “verification” used when checking that a developer actually fixed a defect. So this is the third definition of the term “verification” in this course, and it shan’t be the last.
- In this case, it means "verifying that a method has been called 0, 1, or n times.“
- A test double which uses verification is called a Mock. However, many frameworks (such as Mockito, the one we are using) don't have a strong differentiation between doubles and mocks. Technically, though, a mock is a specific kind of test double.

WHAT IS VERIFICATION?

- I like to think of it as “an assertion on the execution of the code”

EXAMPLE - A SLIGHTLY MODIFIED QUACK_ALOT

```
def quack_alot(d, n)
  n.times do
     d.quack
  end
  nil
end # What can we test here?

# Let's look at the sample code
```

EXAMPLE - A SLIGHTLY MODIFIED QUACK_ALOT

```
def test_quack_alot_mock
 mocked_duck = Minitest::Mock.new("mocked duck")
 mocked_duck.expect :quack, 1
 quack_alot mocked_duck, 1
 assert_mock mocked_duck
end
```

STRUCTURING UNIT TESTS

- Two philosophies:
  - Test only public methods.
    - This is the true interface to an object. We should be allowed to change the implementation details at will.
    - Private methods will be tested as a side effect of any public method calls.
    - Private methods may be difficult to test due to language/framework.
  - Test every method – public and private.
    - Code is code. The public/private distinction is arbitrary – you still want it all to be correct.
    - Unit testing means testing the lowest level; we should test as close to the actual methods as possible.

WE HAVE TO COME UP WITH A DECISION!

- Like most software engineering decisions - it depends. I don’t think that there is a right answer.

WHAT KINDS OF THINGS SHOULD I TEST ON THOSE METHODS?

- Ideally…
    - Each equivalence class
    - Boundary values
    - Failure modes
    - Any other edge cases

WHAT IF IT IS DIFFICULT TO TEST THINGS?

- It happens
- Especially when working with legacy code.
- Such is life.
- Don’t give up!

MY ADVICE

Try to add tests as soon as possible. DO NOT WRITE ALL OF YOUR CODE AND THEN TRY TO ADD TESTS.

Ideally, write tests before coding (TDD)!

Develop in a way to make it easy for others to test.

In legacy systems, add tests as you go. Don't fall into the morass!