CS1632, LECTURE 6: UNIT TESTING

BILL LABOON

MANUAL TESTING

- What we have been doing so far
- We write test plans
- A human executes them

AUTOMATED TESTS

- Mostly what we'll be doing from here on out
- We write tests which the computer executes for us

BENEFITS OF MANUAL TESTING

1. It's simple!
2. It's cheap (at first)
3. It's easy to set up
4. No additional software to learn or write
5. Flexible
6. Can focus on things users care about
7. Humans catch issues that programs may not notice

DRAWBACKS OF MANUAL TESTING

1. It is BORING
2. It can be unrepeatable
3. Some tasks are difficult to test manually, e.g.:
    1. Timing
    2. Individual methods, classes, objects
    3. Low-level interfaces
4. Human error is a possibility
5. It's time and resource-intensive

BENEFITS OF AUTOMATED TESTING

1. No chance for human error (during execution)
2. Fast test execution
3. Easy to execute once set up
4. Repeatable
5. Less resource-intensive during testing
6. Ideal for testing some things that manual testing is bad for

DRAWBACKS OF AUTOMATED TESTING

1. Requires extra time up-front
2. May not catch user-facing bugs
3. Requires learning tools and frameworks (but that's one of the things this class can help with)
4. Requires more skilled staff
5. Big issue: It only tests what it is looking for

SOLUTION: A MIXTURE

- Most teams will use both manual and automated tests
- Usually, the number of automated tests will far outnumber the number of manual tests

WHAT IS UNIT TESTING?

- A kind of automated testing
- Unit testing involves testing the smallest coherent "units" of code, such as functions, methods, or classes.
- It is white-box; you are looking at and testing the code directly.
- Ensures that the smallest pieces of the code work correctly (NOT that they work correctly with the rest of the system – very localized)

EXAMPLES

1. Testing that a .sort method sorts elements
2. Testing that passing a nil/null as an argument throws an exception
3. Testing that a formatNumber method formats a number properly
4. Checking that passing in a string to a function which expects an integer does not crash
5. Testing that a .send and .receive method exist on a class

UNIT TESTING

This is usually done by the developer writing the code, another developer (esp. in pair programming), or (very occasionally), a white-box tester.

WHAT'S THE POINT?

1. Problems found earlier
2. Faster turnaround time
3. Developer understands issues with his/her code
4. "Living documentation“
5. Able to tell if your changes caused issues elsewhere by running full test suite

MINITEST - OUR TESTING FRAMEWORK

- https://github.com/seattlerb/minitest
- Run “gem install minitest” or (better) add minitest to your Gemfile (see example) and run “bundle install”
- “…a complete suite of testing facilities supporting TDD, BDD, mocking, and benchmarking.
- Why Minitest? Relatively common, easy to learn, very fast, minimal.

MINITEST IS NOT THE ONLY UNIT TEST FRAMEWORK OUT THERE!

- Test::Unit (built-in)
- shoulda
- rspec
- Cucumber
- Ideas should apply to other testing frameworks easily

WHAT DO UNIT TESTS CONSIST OF?

- (optional) Set up code
- Preconditions
- Execution Steps
- Postconditions - a/k/a Assertions (a/k/a asserts, shoulds, musts)
- (optional) Tear down code

EXAMPLE (IN NATURAL LANGUAGE, NOT CODE)

I create two Integer objects, 1 and 1.

If I compare them with the equality operator, they SHOULD be equal.

(or "they MUST be equal.")

(or "I ASSERT that they will be equal")

POSTCONDITIONS = ASSERTIONS

- When you think "should" or "must", that is the assertion. It's what you're testing for.
- It's the EXPECTED BEHAVIOR of the unit test.
- When you execute the test, that's when you'll find out the OBSERVED BEHAVIOR.
- If the expected behavior matches the observed behavior, the test passes; otherwise it fails.

MINITEST ASSERTIONS

- Some assertions using MiniTest:
- assert_true
- assert_equals
- assert_includes
- assert_nil
- assert_raises

MINITEST ASSERTIONS

- You can also do the opposite with “refute” (like “assert not”)
- refute_true
- refute_equals
- refute_includes
- refute_nil
- refute_raises

TESTS ARE RUN IN RANDOM ORDER

- Make sure your tests are INDEPENDENT and SELF-CONTAINED
- Tests should be focused - one equivalence class, one method call
- Usually one or two assertions - rarely more than that
- Remember you are testing a small bit of code (a unit), not the whole system!

EXAMPLES IN SAMPLE_CODE/MINITEST_EXAMPLES

ADVANCED TECHNIQUES WITH MOCHA

- Dummies
- Doubles
- Stubs
- Mocks
- Verification

DUMMY

- Object that you pass in knowing that it won't be used
- Usually just nil or Object::new

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

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.

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?

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.

WHAT IS VERIFICATION?

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

EXAMPLES IN SAMPLE_CODE/MOCHA_EXAMPLES

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.

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!
- Solving this problem is the subject of the next two lectures.