Roberto Rojas Molina
A00354772

L3 Summary

Pragmatic Unit Testing (Chapters 1, 4, 5)

A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality in the code being tested. Unit tests are performed to prove that a piece of code does what the developer thinks it should do. Unit testing helps create better designs and drastically reduce the amount that you spend debugging and helps gain confidence in the code. It is the most effective technique to better coding. It is also a way to create “executable documentation” which is always in sync with the code. You need to ensure that the code does what you want all the time, this means even in the worst scenarios and unexpected parameters. The code we write should be dependable. Not only “think” we can depend on it, but actually be able to depend on, knowing for certain its strengths and limitations. Unit tests also allows to communicate the code’s intended use, being also a sort of executable documentation, which is always correct as long we keep running the tests and making sure they pass.

To do unit testing we need first to decide how to test the method, then write the test code itself before the implementation. Then the tests should become part of the system making sure all tests pass. Every test needs to determine whether it passed or not. The most common excuses for not testing are the following:

1. It takes too much time to write the tests: Writing tests take a bit of extra time, but is a lot faster and cheaper than the time spent isolating, debugging, reworking the bugs.
2. It takes too long to run the tests: Tests shouldn’t take long to run, if they do they probably need to be redesigned. Those tests that are not possible to run in a short time can run only once a day or in an automated build.
3. Legacy code is impossible to test: If unit tests cannot be written for legacy code means that the code needs to be refactored.
4. It’s not my job to test the code: Our job is to write and deliver quality code. We cannot ensure quality if the code is not tested.
5. I don’t really know how the code is supposed to behave so I can’t test it: Requirements need to be clarified before continue coding.

6. But it compiles: Compiler is syntax validator, it does not identify bugs in what the code is intended to do
7. I'm being paid to write code, not to write tests: Same as number 4.
8. I feel guilty about putting testers and QA staff out of work: QA's job is to work on testing automation, integration and end to end flows. Unit testing is up to the developer.
9. My company won't let me run unit tests on the live system: Unit tests should be design to run in our own system, with our own data or mock objects, anything else is not unit testing.
10. Yeah, we unit test already: Make sure that the tests are not actually integration tests, that they test exceptional conditions (not only the "happy path") and that they are maintained.

To strengthen our testing skills, we need to look we need to look at the following specific areas:

Right: Are the results right?

Result validation, check that the expected results are right. For sets of tests with large amounts of test data, you might want to consider putting the test values and/or results in a separate data file that the unit test reads in. Test data may be incorrect. When test data says you're wrong, double- and triple-check that the test data is right before attacking the code.

B: Are all the boundary conditions CORRECT?

Identifying boundary conditions is one of the most valuable parts of unit testing, because this is where most bugs generally live, at the edges. An easy way to think of possible boundary conditions is to remember the acronym CORRECT.

Conformance: Does the value conform to an expected format? Validate formatted string data such as e-mail addresses, phone numbers, account numbers, or file names. Consider what will happen if the data does not conform to the structure you think it should.

Ordering: Is the set of values ordered or unordered as appropriate? Another area to consider is the order of data, or the position of one piece of data within a larger collection.

Range: Is the value within reasonable minimum and maximum values? We need to avoid variables that allows to take on a wider range of values than we need. Don't use built in value types for bounded-integer values. We need to look for out of bound issues and have special attention when treating with indexes.

Reference: Does the code reference anything external that isn't under direct control of the code itself? We need to look out for any dependencies and application state. We should not make any assumptions about the global state of the application and make sure the app behaves as expected when the conditions are not met.

Existence: Does the value exist (e.g., is non-null, non-zero, present in a set, etc.)? It is a common issue that methods run into problems when expected data is not available. We need to check and validate what happens when we receive a null, zero, empty value, empty string, empty collection, etc. and handle those cases gracefully.

Cardinality: Are there exactly enough values? We also need to make sure that we handle those case where the amount of values is not exactly the expected.

Time (absolute and relative): Is everything happening in order? At the right time? In time?There are several aspects to time you need to keep in mind:

Relative time (ordering in time): This include issues of timeouts in the code

Absolute time (elapsed and wall clock): What if something you are waiting for takes "too much" time?

Concurrency issues: What will happen if multiple threads use this same object at the same time?

I: Can you check inverse relationships?

Some methods can be checked by applying their logical inverse. Be cautious when you've written both the original routine and it's inverse, as some bugs might be masked by a common error in both routines.

C: Can you cross-check results using other means?

When possible we can use different ways to get the same result. This allows us to create tests that rely on different methods, libraries or algorithms to verify our results.

E: Can you force error conditions to happen?

You should be able to test that your code handles real-world problems by forcing errors to occur. If our code needs to handle these sort of errors, then we need to test for them.

P: Are performance characteristics within bounds?

As input sizes grow problems become more complex. We should consider some rough tests just to make sure that the performance curve remains stable.

Testing your code

General rules of testing:

- A testing unit should focus on one tiny bit of functionality and prove it correct.
- Each unit test should be fully independent.
- Test should run fast.
- Learn your tools.
- Run the full suite before a coding session, then run it after.
- Implement hooks that run the test before pushing code.
- Write a test about what will be developed next before interrupting work.
- Use long and descriptive names for testing functions.
- Run the tests as much as running the code.
- Use testing code as an introduction for new developers.

unittest is a test module included in the python standard library.
doctest module searches for pieces of text that look like interactive Python sessions in docstrings.

Testing tools: py.test, hypothesis, unittest2, mock,