AUTOMATED TESTING WITH PYTEST

OR

MAKING PYTEST YOUR BATMAN

WHAT IS AUTOMATED TESTING?

WRITE SOME CODE

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

TEST IT OUT

```
In [1]: from fib import fib

In [2]: fib(1)
Out[2]: 1

In [3]: fib(3)
Out[3]: 2

In [4]: fib(4)
Out[4]: 3

In [5]: fib(5)
Out[5]: 5

In [6]: assert fib(2) == 1
```

TURN IT INTO SOMETHING YOU CAN RUN

```
In [9]: %save test_fib.py 1 6-8
```

```
# I can now re-run these tests whenever I want!
./test_fib.py
```

A TYPICAL SMALL AUTOMATED TEST (UNIT TEST)

A function or class with the following structure:

1. Arrange — set up inputs and preconditions
2. Act — do the operation
3. Assert — verify the results

Unit testing frameworks like unittest and pytest offer helpers, but that's really all there is to it.

WHAT'S THE BEST WAY TO DO AUTOMATED TESTING?

This gets controversial in a hurry.

TERMINOLOGY

Automated testing
: The opposite of manual testing

Unit testing
: Testing a single function, class, or method

Test-driven development
: Writing tests before you write the code

Integration testing or component testing
: Testing a group of related units together

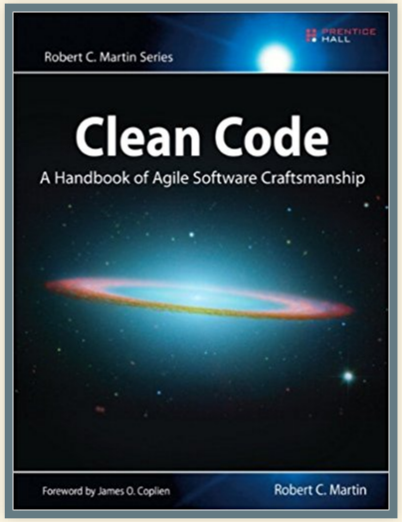
System testing or end-to-end testing
: Tests that are run against the entire system

There are terminology differences here; e.g., Uncle Bob says integration tests are a higher level than component tests.

All of these focus on functional testing — does the code do what it says it does? There's also non-functional testing - load testing, security testing, etc. A lot of that can be automated too, but that's outside of the scope of tonight's talk.

The Three Laws of TDD

(Test Driven Development)


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

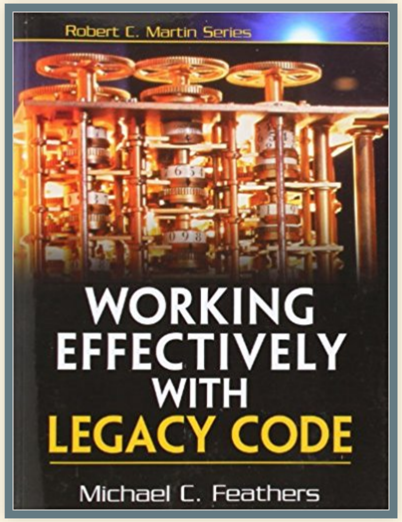
1. You may not write production code until you have written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
3. You may not write more production code than is sufficient to pass the currently failing test.

— Robert C. Martin, *Clean Code*

Here's one view. And Uncle Bob can be pretty dogmatic about it— he compares TDD to doctors washing their hands and strongly implies it's unprofessional not to do it. *Clean Code* is one of the best programming books I've read, but I'm not certain I agree. Writing tests ahead of time is one effective way of getting good test coverage, but it's not the only way.

Unit tests run fast. If they don't run fast, they aren't unit tests.

Other kinds of tests often masquerade as unit tests. A test is not a unit test if:



1. It talks to a database.
2. It communicates across a network.
3. It touches the file system.
4. You have to do special things to your environment (such as editing configuration files) to run it.

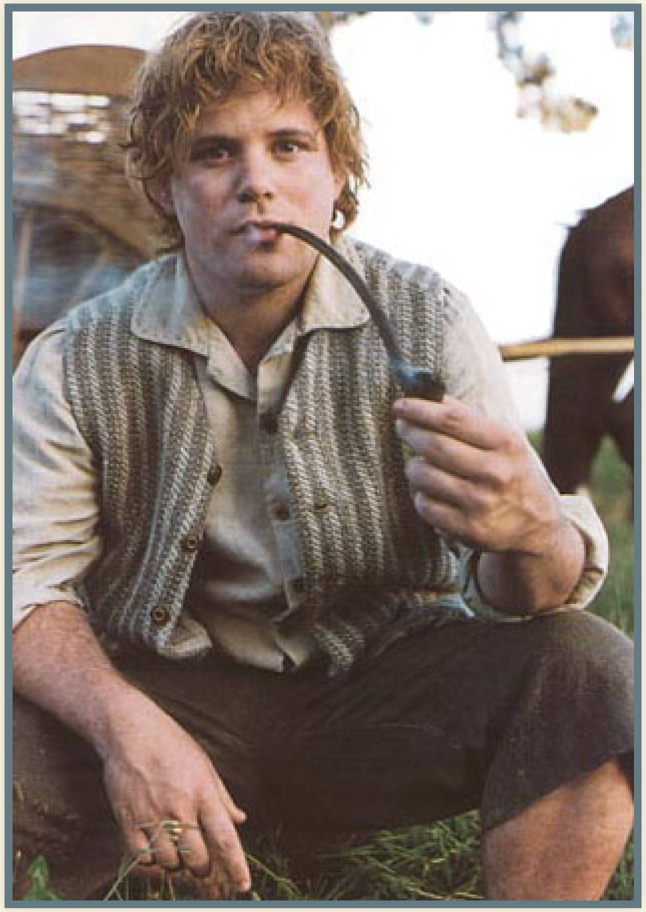
— Michael Feathers, *Working Effectively with Legacy Code*

Working Effectively with Legacy Code is a *fantastic* resource on how to take old, hard-to-test code and modernize it, so Feathers certainly knows his stuff. And there's a lot of truth to the argument that unit tests need to be so fast that you have no excuse not to run them. But I'd rather have a "fast enough" test that breaks these rules than jump through a lot of hoops trying to meet them.

[I'm] lazy, as all good engineers are wont to be. Being lazy means you don't want to find bugs at run time. Being lazy means you don't want to ever make the same mistake twice. Being lazy means making your compiler(s) work as hard as possible, so that you don't have to… If you treat it well, you can make the compiler your right-hand man, helper, conscience, your batman.

— Matthew Wilson, *Imperfect C++*

Wilson's a C++ developer, so he talks about “compilers,” but you can substitute “computers” in general here. This is closer to my view. As programmers, we automate things for a living — this extends to automating our own work.



[I'm] lazy, as all good engineers are wont to be. Being lazy means you don't want to find bugs at run time. Being lazy means you don't want to ever make the same mistake twice. Being lazy means making your compiler(s) work as hard as possible, so that you don't have to… If you treat it well, you can make the compiler your right-hand man, helper, conscience, your batman.

— Matthew Wilson, *Imperfect C++*

"Batman" is a term from the days of the British Empire, referring to a servant or orderly attached to a military officer.

FASTER FEEDBACK

All the ways our industry has improved our feedback…

- Interactive computing
- Syntax highlighting
- IDE code analysis
- Agile software development
- Continuous integration

And automated testing

PUTTING IT INTO PRACTICE

UNITTEST

```python
from fib import fib
import unittest

class FibTestCase(unittest.TestCase):
    def test_fib(self):
        self.assertEqual(fib(1), 1)
        self.assertEqual(fib(2), 1)
        self.assertEqual(fib(3), 2)
        self.assertEqual(fib(4), 3)
        self.assertEqual(fib(5), 5)

if __name__ == '__main__':
    unittest.main()
```

PYTEST BENEFITS

- Test discovery
- No need to inherit from TestCase
- Simpler assertions
- Plugins
- Test fixtures
- And lots of “quality of life” improvements

SWITCHING FROM UNITTEST

```
from fib import fib
import unittest

class FibTestCase(unittest.TestCase):
    def test_fib(self):
        self.assertEqual(fib(1), 1)
        self.assertEqual(fib(2), 1)
        self.assertEqual(fib(3), 2)
        self.assertEqual(fib(4), 3)
        self.assertEqual(fib(5), 5)

if __name__ == '__main__':
    unittest.main()
```

SWITCHING TO PYTEST

```python
from fib import fib

def test_fib():
    assert fib(1) == 1
    assert fib(2) == 1
    assert fib(3) == 2
    assert fib(4) == 3
    assert fib(5) == 5
```

SWITCHING TO PYTEST

1. `pip install pytest`
2. Run `pytest` instead of `./my_test_suite.py`
3. That's it!

(or use unittest2pytest)

pytest is drop-in-compatible with unittest, and it supports test discovery, so you don't need to do anything special to tell it what tests to run.

TEST DISCOVERY

By default:

- Search recursively under the current directory
- Any file named `test_*.py` or `_*test.py`
- Any class starting with `Test`
- Any function starting with `test_`

Actually, current versions of unittest do discovery too, although I've had better luck with pytest's.

UNITTEST ASSERTIONS

```
assertEqual                    assertIsInstance
assertNotEqual                 assertIsNotInstance
assertTrue                     assertAlmostEqual
assertFalse                    assertNotAlmostEqual
assertIs                       assertGreater
assertIsNot                    assertGreaterEqual
assertIsNone                   assertLess
assertIsNotNone                assertLessEqual
assertIn                       assertRegex
assertNotIn                    assertNotRegex
```

PYTEST ASSERTIONS

```
assert
```

PYTEST ASSERTIONS

pytest inspects the Abstract Syntax Tree (AST) to turn this...

```
a = 1
b = 2
assert a + b == 3
```

PYTEST ASSERTIONS

...into this.

```
@py_assert0 = 1
@py_assert2 = 2
@py_assert4 = @py_assert0 + @py_assert2
@py_assert6 = 3
@py_assert5 = @py_assert4 == @py_assert6
if not @py_assert5:
    @py_format8 = @pytest_ar._call_reprcompare(('==',), \
        (@py_assert5,), ('(%(py1)s + %(py3)s) == %(py7)s',), \
        (@py_assert4, @py_assert6)) % {
            'py3': @pytest_ar._saferepr(@py_assert2),
            'py1': @pytest_ar._saferepr(@py_assert0),
            'py7': @pytest_ar._saferepr(@py_assert6)}
    @py_format10 = ('' + 'assert %(py9)s') % {'py9': @py_format8}
    raise AssertionError(@pytest_ar._format_explanation(@py_forma
@py_assert0 = @py_assert2 = @py_assert4 = @py_assert5 = @py_asser
```

(You are not expected to understand this.)

This is magical, and magic code is often bad — it's even part of the Zen of Python that "explicit is better than implicit." But, sometimes, some magic can be nice; for unit testing in particular, it saves a lot of boilerplate.

PYTEST PLUGINS

pluginCompat.herokuapp.com

PYTEST COVERAGE

```
pip install pytest-cov
pytest --cov=. --cov-report=html:../cov_html
```

There's some controversy over how you should use code coverage and whether you should target a particular percentage, but at minimum, it gives valuable information into how effective your test suite is.

PYTEST-WATCH

Run `ptw`, and your tests will be automatically executed whenever your code changes.

PYTEST-XDIST

Run tests across multiple CPUs or hosts.

Run tests repeatedly on file change (similar to pytest-watch).

Run tests on multiple environments.

EXAMPLE: REPORTING SLOW TESTS

Report the 3 slowest tests, so that you know what to work on:

```
pytest --durations=3
```

EXAMPLE: EXPECTED ASSERTIONS

```python
def test_negative_numbers():
    with pytest.raises(ValueError):
        fib(-1)
```

CASE STUDY: AUTOMATING GUI TESTS WITH PYWINAUTO

So far, we've been focusing on unit tests. That's pytest's intended usage, and that ought to be where you spend most of your automated testing efforts. But pytest is also good for integration and system tests. I've been using it to do GUI test automation of a legacy C++ application that we maintain. This happens to be a good demonstration of some more unique pytest features.

PYTEST FIXTURES

```python
@pytest.fixture
def notepad():
    notepad = pywinauto.Application().start('notepad.exe')
    yield notepad
    notepad.close()

def test_copy(notepad):
    notepad.UntitledNotepad.Edit.type_keys('Hello')
    notepad.UntitledNotepad.Edit.type_keys('^A^C')
    assert pywinauto.clipboard.GetData() == 'Hello'
```

USING PYTEST WITH PDB

Optionally, install PDB++:

```
pip install pdbpp
```

Manually set a breakpoint:

```
import pdb; pdb.set_trace()
```

Or break on the first failure:

```
pytest --pdb -x
```

All of this makes for a very fluid coding style: Write code then manipulate it in IPython. Write code then let pytest-watch automatically run it. Run code then use pdb to manipulate it. Manipulate it in IPython then save it as new code.

FILTERING TESTS

```
pytest test_notepad.py
pytest test_notepad.py::TestFileIO::test_save
```

FURTHER READING

"Professionalism and Test-Driven Development," Robert C. Martin, IEEE Software, Vol. 24, Issue 3

UnitTest and SelfTestingCode, Martin Fowler

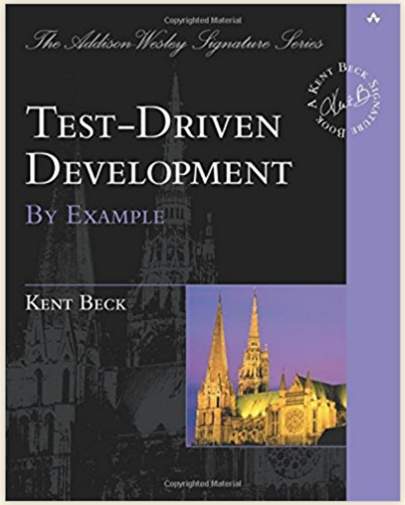
Why Most Unit Testing is Waste, James O. Copelien

TDD is dead. Long live testing, David Heinemeier Hansson

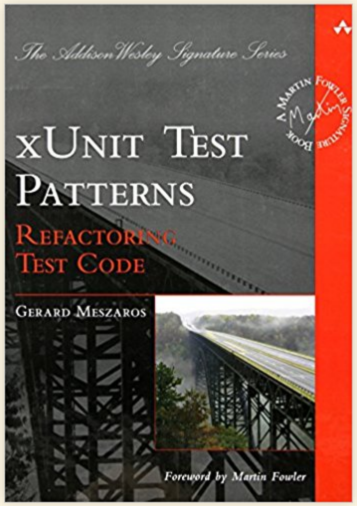
Just Say No to More End-to-End Tests, Mike Wacker

Write tests. Not too many. Mostly integration, Kent C. Dodds

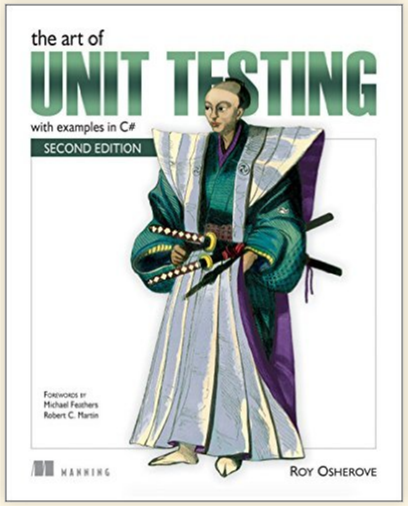
FURTHER READING


The Addison Wesley Signature Series
Test-Driven Development
By Example
Kent Beck
A Kent Beck Signature Book

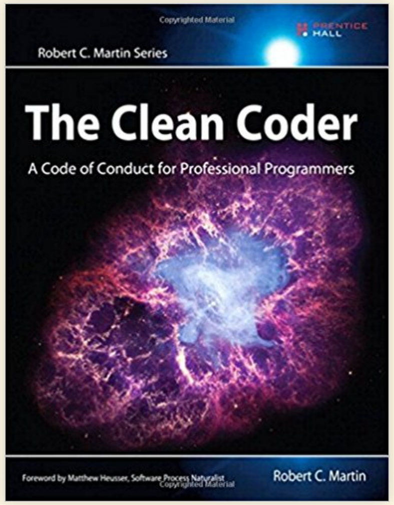
Test Driven Development: By Example, Kent Beck


The Addison Wesley Signature Series
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
A Martin Fowler Signature Book
Foreword by Martin Fowler

xUnit Test Patterns, by Gerard Meszaros



The Art of Unit Testing, by Roy Osherove


Robert C. Martin Series
Prentice Hall
The Clean Coder
A Code of Conduct for Professional Programmers
Foreword by Matthew Heusser, Software Process Naturalist
Robert C. Martin

The Clean Coder, by Robert C. Martin

IMAGE CREDITS

The Dark Knight © Warner Bros. 2017.

The Lord of the Rings film trilogy © New Line Cinema 2001, 2002, 2003