Test-Driven Repair

A discussion of test patterns for code that was written without tests.

Christopher Neugebauer
mail@chrisjrn.com
Australian
Home
What I knew about testing when I wrote the abstract?
The abstract was wrong.
Testing Done Right

What? Where? Why?
Test-Driven Development

- Step 1: Write a failing test
- Step 2: Write some code that makes the test pass
- Step 3: Repeat until all your code is done
Not Quite TDD

- Testing during review
- Testing as bug reports emerge
Structured Testing

- *Unit Testing*: One element of functionality works.
- *Integration Testing*: Unit-tested elements work together
Structured Testing

- Write unit tests first
- Write units
- Write integration tests
Why write tests early?

- Separated interfaces
- Clear separation of concerns (mocks, fakes, etc)
Why write tests early?

- I/O (if not all state) separate from business logic
  - Lots of unit tests (fast)
  - Relatively few integration tests (slow)
Why TDD?

You get tests from the moment you get code.
TDD Zealotry

“it is impossible BY DEFINITION to do Test-Driven Development after the code is written.”

— Tim Ottinger, author of ‘Clean Code’, on Twitter
LIES.
People have old code that should be tested.

The unfortunate thing is that much design and process advice assumes that your project is a blank page. In actuality, greenfield projects are noticeably rare. Most projects carry some amount of legacy code.

You *can* adopt early-stage testing on future releases.
So you have code that needs tests?
Untested Code

- Poor Interfaces
- Poor separation of concerns
- Dependencies tied together
Poor Interfaces

Lots of setup required to test one function
Poor Interfaces

The only way to execute code in one function may involve burrowing through other functions.
Poor Separation of Concerns

Impossible to test results of operations without altering state or performing I/O.
Tied Dependencies

You need to invoke code in other subsystems to run code in the subsystem under test.
When your code is coupled, "units" for testing do not exist.
If you don't have units, integration isn't well defined.
How Do You Test?
Test Coverings
Test Coverings

Tests that cover all of the subsystem under investigation.
Interface Testing

Testing everything that is reachable, with the interface you already have.
What is an Interface?

Anything that gives OK isolation to the code you want to test.
Interfaces for testing

GOOD: A custom test harness
Interfaces for testing

*OK*: Web APIs running in-place in your server
Interfaces for testing

*MEH*: A web front-end running in-place on your server
Any interface that lets you get tests in early is a good interface.
Units or Integrations?

Interface tests are neither. Do not expect unit-level precision.
What do you get then?
Invariants

Tests induce an invariant in the code under test.
Invariants

Invariants do not change after refactoring.
Invariants

Invariants only exist on code under test.
Unit tests give invariants with better precision

- You can tell what failed
- You can tell how you made it fail
Interface tests need lots of coarse-grained invariants

- You can tell that a case failed
- You can't (easily) tell what made it fail
Interface test invariants

- Relatively few invocation paths
- Lots of different inputs
- Classify into valid inputs and invalid inputs
It doesn't quite test what you want
but at least you *get* tests
Test Coverage

How you know what your tests actually test, and what your next test should be.
def cook_spam():
    print("1. Preheat the oven.")
    print("2. Put spam into oven")
    print("3. Sing the spam song")
    print("4. Tell customer their spam will be ready")
    print("5. Fight off horde of vikings")
    print("6. Serve the spam")
def cook_thing(thing):    # Needs four tests for full coverage
    if thing == "spam":
        cook_spam()
    elif thing == "eggs":
        cook_eggs()
    elif thing == "bacon":
        cook_bacon()

def cook_spam():        # Needs only one test
    print("1. Preheat the oven.")
    print("2. Put spam into oven")
    print("3. Sing the spam song")
    print("4. Tell customer their spam will be ready")
    print("5. Fight off horde of vikings")
    print("6. Serve the spam")
Use metrics that indicate effort.
Good interface tests will test every *branch*
Making Interface Tests Bearable
I/O and State

If it's embedded in your code, don't expect to remove it before you write tests.
Speed

Testing at the interface is sloooooow.
Correctness vs Speed

- A correct suite runs after every test case in a sanitised environment.
- A fast suite rarely sanitises.
Side-Effects of Isolation

- A slow suite will never get run
- A fast, but incorrect suite will waste developers' time.
Measure Every Test Case

- Setup time
- Execution time
- Teardown time
Cluster Unrelated Tests

Find tests that alter different parts of the environment.

Run them under the same test fixtures.
Cluster Unrelated Tests

Run each cluster, multiple times, in random orders.
Run Thoroughly Nightly
Don't make this your long-term testing strategy.
Refactor immediately.
A Refactoring Strategy
Uncle Bob's Clean Architecture

– Uncle Bob
Unclean Architecture

```python
def find_definition(word):
    q = 'define ' + word
    url = 'http://api.duckduckgo.com/?'
    url += urlencode({'q': q, 'format': 'json'})
    data = call_json_api(url)
    definition = data['Definition']
    if definition == u'':
        raise ValueError('that is not a word')
    return definition

def call_json_api(url):
    response = requests.get(url)  # I/O
    data = response.json()        # I/O
    return data
```

- **Brandon Rhodes, Clean Architecture in Python**
Clean Architecture

```python
def find_definition(word):
    url = build_url(word)
    data = requests.get(url).json()  # I/O
    return pluck_definition(data)

def build_url(word):
    q = 'define ' + word
    url = 'http://api.duckduckgo.com/?'
    url += urllencode({'q': q, 'format': 'json'})
    return url

def pluck_definition(data):
    definition = data['Definition']
    if definition == u'':
        raise ValueError('that is not a word')
    return definition

-Brandon Rhodes, *Clean Architecture in Python*
Functional Core--Imperative Shell


- Write processing logic as pure functions
- Isolate I/O as close to the top as you can.
Testing with FCIS

Unit tests are quick because pure functions are fast.
Testing with FCIS

Integration tests are easy because you can write lots of unit tests.
This is not just an architectural decision.
It's a refactoring strategy too
FCIS at Smaller Scales

- Find processing routines inside bigger functions
- Factor out testable functions
- Factor inner loops out of those too!
FCIS at Smaller Scales

- Keep I/O at the top
- Factor fast code into their own functions.
Test After You Refactor

These emergent functions are unit testable, so write some tests.
Once you refactor into units and test then
Your interface tests *become* integration tests.
Taking unstructured testing seriously means you'll start doing structured testing.
Tracking Down Bugs
Test Harnesses

Make it easy to translate a bug report into a test case.
Data Scales, Code Doesn't

Express cases as parameters to a function rather than writing new procedures.
Retain your cases

Keep these test cases, and run them regularly.
Not the talk I thought I'd give
Test-Driven Development

- Write Unit Tests
- Write Units
- Test end-to-end functionality with integration tests
Test-Driven Repair

- Write Integration Tests
- Refactor into units
- Write unit tests
Start with Test-Driven Repair
Enable Test-Driven Development
The End

Test-Driven Repair

Christopher Neugebauer
mail@chrisjrn.com