Unit Testing Makes Your Code Better

Greg Ward <greg@gerg.ca>
@gergdotca
PyCon 2014
Montreal, QC • Apr 12, 2014

http://gerg.ca/talks/testing-pycon/

It's for the common good

gh this

the ONION®
America's Finest News Source

A.V. CLUB | YouTube

ORTS · BUSINESS · SCIENCE/TECH · LOCAL · ENTERTAINMENT | search

Report: 98 Percent Of U.S. Commuters Favor Public Transportation For Others

NEWS · Science & Technology · Trends · Automotive · Public Transportation · ISSUE 44·27 ISSUE 36·43 ·

Nov 29, 2000

1.2K
215
36

WASHINGTON, DC–A study released Monday by the American Public Transportation Association reveals that 98 percent of Americans support the use of mass transit by others.

http://gerg.ca/talks/testing-pycon/

Assumption #1

You've at least started to drink the Kool-Aid™:

- either you're already writing unit tests
- or you're ready to start, with or without this talk

Assumption #2

(Corollary of Assumption #1)
You already get that unit testing helps make your code more correct.

I'm talking about “better” on a higher plane: æsthetics, elegance, beauty.

Elegance? Shmelegance!

Beautiful code is better code:

- easier to understand
- easier to extend
- easier to reuse

Plan of attack

Real-life case study:

- Examine some untested code
- Work through adding tests
- Understand how imperfect design → hairy tests
- Modify the design for simpler tests → better code

Background

- what is this code?
- why does it exist?
- where does it come from?
- what requirements does it meet?

What is this code?

- we[1] measure the Internet
- we ping all your public IPs every couple of months
- more relevant: we traceroute to 1.5m hosts all day, every day, from ~100 collectors around the world
- result: ~200m traces containing ~3b hops every day
- and then we try to make sense of this torrent of data

[1] www.renesys.com

A torrent of data

- clearly we need a super-duper, highly-advanced, next-generation, whiz-bang data storage and representation technology
- like... plain text?
- (ok, sure, we also use PostgreSQL and Redis when performance matters)

Staying sane with plain text

- keep it simple, stupid
- restrict the data tightly to avoid escaping
- stay consistent even as data and requirements evolve

Example 1: raw traceroute

T3 files contain one record (line) for every traceroute sent

```
T3      1394305276      icmp    0 \
172.18.79.139   vps01.nbo1        \
62.219.197.44   S                 \
172.17.28.6     0.924   2         \
41.139.255.94   0.583   3         \
[...]
62.219.197.44   201.326 14
```

(note variable number of fields, just to keep things interesting)

Example 2: daily summary

TIP1 files contain one record summarizing all the traces sent to a single target on a given day

```
TIP1    1395878400      67.212.64.4             \
67.212.64.0/24  45.50884          -73.58781      \
6077243 NA      CA      QC      Montréal        \
67.212.64.0/19  Peer1                           \
113     163     vps01.bos1=2,vps01.tlv1=1,…,vps01.hkg2=2
```

(note field with internal structure: key-value mapping)

Spot the similarities?

- tab-separated
- first column *of every line* is the data format
- every column has a name and a type
- most types are simple (str, int, float)
- some are complex (comma-separated string-to-int mapping)
- UTF-8 encoded
- often bzipped

Common format, common library

- we have a couple dozen of these formats
- writing a new parser from scratch for each one would be nuts
- hence: GenericLineParser
- with many subclasses: T3Parser, TIP1Parser, etc.

Requirements for GenericLineParser

- structured (columns and types)
- fast (simple files, but large: don't always want to parse that comma-separated string-to-int mapping)
- flexible (must be trivial to define new formats)

Good news: when I started testing, the code already met all of those requirements nicely.

Overview

```
class GenericLineParser(object):
    # -- main public interface
    def __init__(self, name, f=None,
                 fields=None, nfields=None, sep='\t'):
    def __str__(self):
    def parse(self, line=None, record=None, convert=True):
    def read(self, verbosity=1, convert=True,
             nreport=100000):

    # -- mostly internal methods
    def new(self):
    def convert_record_at_index(self, i, valin):
    def convert_record(self, nt, field_name):
    def convert_record_types(self, rec):
```

Easy to define new formats

```
tip1_fields = [
    ('DCV', str),
    ('ts', int),
    ('ip', str),
    ('routepfx', str),
    ('geopfx', str),
    ('latitude', float),
    ('longitude', float),
    [...]
    ('n_collectors', int),
    ('n_responses', int),
    ('collectors', CskvSIList)]

class TIP1Parser(GenericLineParser):
    def __init__(self, f=None, fields=tip1_fields,
                 nfields=None, sep='\t'):
        GenericLineParser.__init__(
            self, 'TIP1', f, fields, nfields, sep)
```

Where to start testing?

Well, you can't *test* an object if you can't construct it... so I like to start with the constructor

This goes double in cases like this one, with a non-trivial constructor (complex internal logic, sometimes does I/O)

About that constructor

```
def __init__(self, name, f=None,
             fields=None, nfields=None, sep='\t'):
   [...set some attrs...]
   if f is None:
   elif isinstance(f, basestring):
       try:
           if f.endswith('.bz2'):
           else:
       except IOError as e:
       except:
   elif isinstance(f, (file, bz2.BZ2File)):
   elif isinstance(f, Iterable):
   else:
       raise …
```

First reactions

- if this is a "line parser", why does it care so much about filenames?
- so... it's a line parser *and* a file opener *and* a file reader? hmmmm...

Testing the constructor

6 test cases for one method: definitely a code smell.

```
class TestGenericLineParser(unittest.TestCase):
   def test_constructor_minimal_args(self):
       '''construct with bare minimum arguments'''

   def test_constructor_filename(self):
       '''parser that will read from UTF-8 file'''

   def test_constructor_filename_bz2(self):
       '''parser that will read from bz2 UTF-8 file'''

   def test_constructor_error(self):
       '''tickle exception-handling code in constructor'''

   def test_constructor_file(self):
       '''pass constructor a file object'''

   def test_constructor_list(self):
       '''pass constructor a list of lines'''
```

Let's fix the constructor (a bit)

LineParsers parse lines. Something else should open files:

```
def zopen(name, mode='r'):
    '''Open a file, possibly compressed. Handles ".gz"
    files with gzip.GzipFile, ".bz2" files with
    bz2.BZ2File, and all other files with builtin
    open().'''

def uopen(name, mode='r'):
    '''Open a UTF-8 encoded file with strict error
    handling. read() returns unicode strings and
    write() expects unicode.'''

def uzopen(name, mode='r'):
    '''Open a possibly compressed, UTF-8 encoded file.
    (De)compression depends on the filename, as with
    zopen(). Uncompressed content must be UTF-8
    encoded, as with uopen().'''
```

(These are all straightforward and thoroughly unit-tested.)

Constructor, version 2

```
def __init__(self, name, f=None,
             fields=None, nfields=None, sep='\t'):
   [...set some attrs...]
   if f is None:
   elif isinstance(f, (file, bz2.BZ2File)):
   elif isinstance(f, Iterable):
   else:
     raise ...
```

Good: no longer cares about filenames at all (caller can use uzopen() directly).

Constructor tests, version 2

```
class TestGenericLineParser(unittest.TestCase):
   def test_constructor_minimal_args(self):
       '''construct with bare minimum arguments'''

   def test_constructor_filename(self):
       '''parser that will read from UTF-8 file'''

   def test_constructor_filename_bz2(self):
       '''parser that will read from bz2 UTF-8 file'''

   def test_constructor_error(self):
       '''tickle exception-handling code in constructor'''

   def test_constructor_file(self):
       '''pass constructor a file object'''

   def test_constructor_list(self):
       '''pass constructor a list of lines'''
```

Progress so far

- constructor is simpler and shorter
- other code can use zopen(), uzopen()
- now supports ".gz" files for free (or future compressed formats)
- less test code to maintain
- fewer code paths to worry about

Plenty more to do

- constructor: treat file/BZ2File/Iterable the same
- factor out progress logging
- shrink read() method to a trivial wrapper

fewer code paths = fewer, simpler tests = better code

OK, what's the big deal?

So I refactored some messy code. Whatever.

- writing the tests made me look deeper
- made me read the code very carefully
- made me see both the good side and the bad side

The “courage to refactor”

This is something unit-testing zealots like to boast about.

- sounds hokey
- sounds like something from a self-help book
- *but it's true!*
- I have absolutely no fear about tearing GenericLineParser to pieces and putting it back together again, even though I didn't write it

No happy ending... yet

Code that *uses* GenericLineParser:
almost completely untested → afraid to refactor

- easy to adapt existing clients of GenericLineParser to use uzopen()
- but because those client apps are untested, I cannot ensure that my change works
- best I can do: patch, ask maintainer to test for me

Thus: the job remains half done. ☹

Costs of not testing

- incorrect code (bugs caught late in the cycle)
- fear of refactoring
- code duplication (→ bug duplication)
- insufficient code reuse

Don't let this get you down!

- 1000 tests are better than 999 tests
- 1 test is *vastly* better than 0 tests
- unit tests will never cover everything (unless you're a wild-eyed maniacal crazed unit-testing fanatic[1])
- but you'll be pleasantly surprised by how much you can cover with some effort

[1] I do not recommend this

This talk is a Trojan horse

All of this has been said before.

- eXtreme Programming (XP)
- Test-Driven Development (TDD)
- Agile Manifesto
- blah blah blah

Conclusions

- *Of course* writing unit tests makes your code more correct—that's just obvious (right?).
- Less obvious: writing unit tests makes your code more beautiful (indirectly).
- Beautiful code is more reusable, more maintainable, more pleasant to work with.
- Beautiful code is *less expensive* (in the long run).

Contact & further reading

Greg Ward <greg@gerg.ca>
@gergdotca
(I work for www.renesys.com.)

- *Extreme Programming Explained* (Kent Beck, Cynthia Andres)
- *Refactoring: Improving the Design of Existing Code* (Martin Fowler, Kent Beck, et. al.)
- *Agile Software Development: Principles, Patterns, and Practices* (Robert C. Martin)
- *Growing Object-Oriented Software, Guided by Tests* (Steve Freeman, Nat Pryce)

http://gerg.ca/talks/testing-pycon/