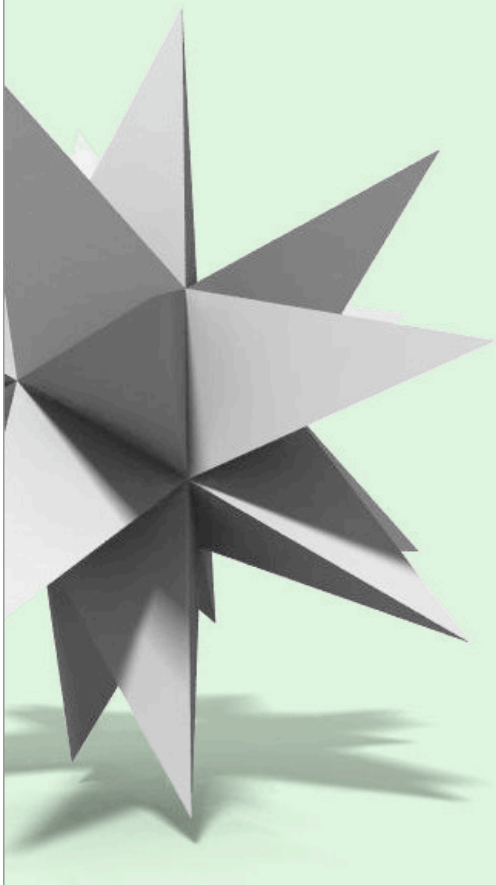


Tests & Testability

Ned Batchelder

Why so hard?

- Different needs
- Tests use your code differently than your product does
- Many internal interfaces
- Baby steps
- Have to decide what your code does!

Testability

- As important as the other -ilities
    - scalab-, usab-, portab-, maintainab-, compatab-, availab-, etc...
- Testable code means:
    - More testing
    - Fewer bugs
    - Better design
- Most of what you do to improve testability is good for the product

Testing lifecycle

- Select a piece of product code
- Configure it to run
- Feed it with defined inputs
- Run it in isolation
- Harvest its outputs
- Decide if it succeeded
- Fix it if it didn't succeed

Testability

- Anything that makes any of those steps easier/faster/better.
- Better tests are:
    - Convenient
    - Fast
    - Unambiguous
    - Repeatable

~ Examples ~

Typical command line

```
 1:  import blah_engine, optparse, sys
 2:
 3:  OK, ERR = 0, 1
 4:
 5:  def blah_main():
 6:      ret = OK
 7:      parser = optparse.OptionParser()
 8:      parser.add_option("-b", "--blah")
 9:      options, args = parser.parse_args()
10:
11:      if options.blah:
12:          print "Blah: %s" % options.blah
13:          ret = ERR
14:      else:
15:          blah_engine.run(args)
16:
17:      sys.exit(ret)
18:
19:  if __name__ == '__main__':
20:      blah_main()
```

Better command line

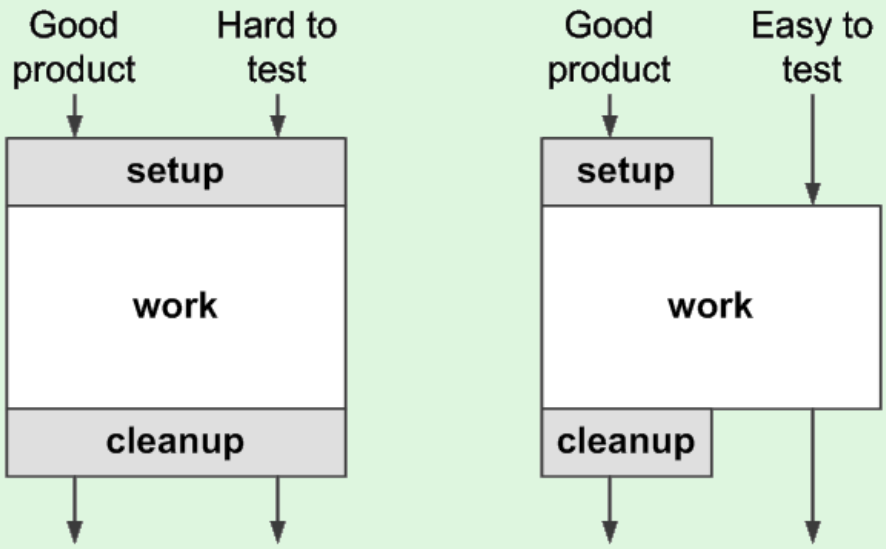
```
 1:  import blah_engine, optparse, sys
 2:
 3:  OK, ERR = 0, 1
 4:
 5:  def blah_main(argv):
 6:      ret = OK
 7:      parser = optparse.OptionParser()
 8:      parser.add_option("-b", "--blah")
 9:      options, args = parser.parse_args(argv)
10:
11:      if options.blah:
12:          print "Blah: %s" % options.blah
13:          ret = ERR
14:      else:
15:          blah_engine.run(args)
16:
17:      return ret
18:
19:  if __name__ == '__main__':
20:      sys.exit(blah_main(sys.argv[1:]))
21:
22:  def test_blah_main():
23:      assert blah_main([]) == 0
24:      assert blah_main(["-b","xyz"]) == 1
```

Split along the seams

```
 1:  # Tied up with files:
 2:
 3:  def write_it(self, filename):
 4:      """Write it to `filename`."""
 5:      f = open(filename, "w")
 6:      f.write(self.foo)
 7:      f.write(self.bar)
 8:
 9:  # Better separation:
10:
11:  def write_it(self, filename):
12:      f = open(filename, "w")
13:      self.write_it_to_file(f)
14:
15:  def write_it_to_file(self, f):
16:      f.write(self.foo)
17:      f.write(self.bar)
18:
19:  def test_write_it_to_file():
20:      f = StringIO.StringIO()
21:      obj.write_it_to_file(f)
22:      assert f.getvalue() == "foo bar"      # or whatever..
```

Exposing internal interfaces

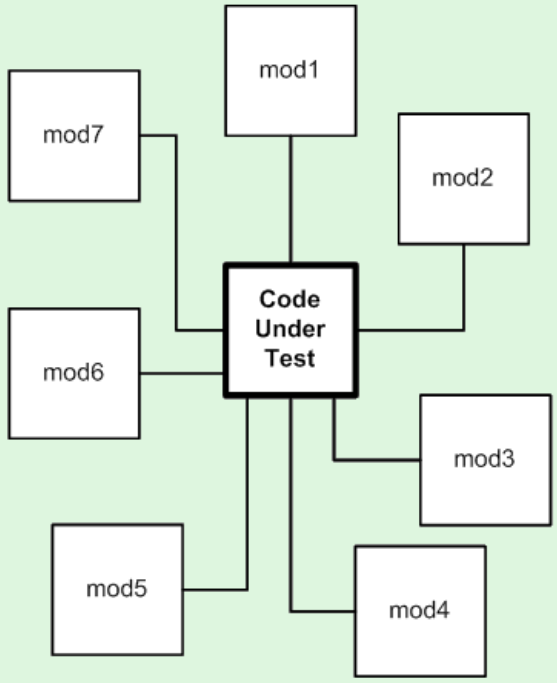
- Unpeeling the sandwich

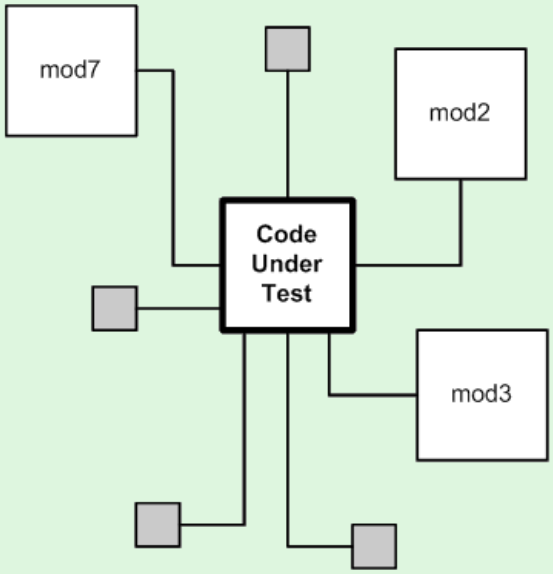


Mocks and dependency injection

```
 1:  import blah_engine, optparse, sys
 2:
 3:  OK, ERR = 0, 1
 4:
 5:  def blah_main(argv, engine=blah_engine):
 6:      ret = OK
 7:      parser = optparse.OptionParser()
 8:      parser.add_option("-b", "--blah")
 9:      options, args = parser.parse_args(argv)
10:
11:      if options.blah:
12:          print "Blah: %s" % options.blah
13:          ret = ERR
14:      else:
15:          engine.run(args)
16:
17:      return ret
18:
19:  if __name__ == '__main__':
20:      sys.exit(blah_main(sys.argv[1:]))
21:
22:  from mock import Mock
23:
24:  def test_blah_main():
25:      engine = Mock()
26:      blah_main(['foo'], engine=engine)
27:      engine.run.assert_called_with(['foo'])
```

Mocks and dependency injection





PDF reporter

```
 1:  # Inheritance is inflexible
 2:
 3:  class PdfWriter:
 4:      def drawstring(self, text): ...
 5:      def endpage(self): ...
 6:
 7:  class PdfReport(PdfWriter):
 8:      def make_report(self):
 9:          self.drawstring(self.title)
10:          for t in self.lines:
11:              self.drawstring(t)
12:          self.endpage()
13:
14:  # Composition gives you options and mockability
15:
16:  class PdfReport:
17:      def __init__(self, pdf=None):
18:          self.pdf = pdf or PdfWriter()
19:
20:      def make_report(self):
21:          self.pdf.drawstring(self.title)
22:          for t in self.lines:
23:              self.pdf.drawstring(t)
24:          self.pdf.endpage()
```

Display list PDF

```
 1:  class DisplayList:
 2:      def __init__(self):
 3:          self.display = []
 4:
 5:      def display_fn(self, fn_name, *args):
 6:          self.display.append((fn_name, args))
 7:
 8:      def play_display(self, output):
 9:          for fn, args in self.display:
10:              getattr(output, fn)(*args)
11:
12:  class ReportMaker(DisplayList):
13:      def make_report_display(self):
14:          self.display_fn('drawstring', self.title)
15:          for t in self.lines:
16:              self.display_fn('drawstring', t)
17:          self.display_fn('endpage')
18:
19:
20:  class PdfReport(ReportMaker):
21:      def __init__(self, pdf=None):
22:          self.pdf = pdf or PdfWriter()
23:
24:      def make_report(self):
25:          self.make_report_display()
26:          self.play_display(self.pdf)
```

Test-only code

- Sometimes it pays to use “if TESTING:”
- Pitfall: you’re changing the SUT
- Benefit: improved testability
- if TESTING examples:
    - in-memory SQLite instead of MySQL, much faster
    - Turn off caching for better repeatability
    - Disable deferred threading for better debugging
    - Extra assertions to catch coding errors

~ Taxonomy of Techniques ~

Better selection

- Modular code
- Well-defined interfaces between modules/classes/methods
- Do one thing well

Better configuration

- Isolate the code with dependency injection
- Supply fake/mock/stub dependencies
- Composition is better than inheritance
    - Can't mock out your base class
- Beware of implicit dependencies
    - sys.argv
    - sys.exit
    - stdout, stderr
    - os.environ

Better feeding

- Fixtures
- Take data from convenient sources
    - files, not filenames

Better running

- Test-only configurations
    - Single-threaded
- Fast running:
    - Mock out disks and networks

Better harvesting

- Beware of all your “outputs”
    - Files on disk
    - New records in db
    - Changed state in other objects
    - Logs

Better deciding

- Well-defined repeatable answers
- Gold files to compare against
- A rich collection of assert methods
- When a test fails, can you spot the difference?
    - Graphics
- PDF output
    - Also generate PNG that can be easily compared.

Better fixing

- Can you tell what was wrong with the output?
  - Richer asserts: assertMultiLineEqual
- Can you pinpoint (and re-run) the failing test?
  - Powerful test runners (nose, py.test)

Testing is a significant activity

- Test-Product ratio of 1:1 is not unreasonable
- You will write test framework code
    - Just accept it
    - You will need to test that code!
- Writing tests is hard
    - Invest in building infrastructure up front

Thank You

Questions?

http://nedbatchelder.com