CHAPTER 7

Test-Driven Development and Impostors

The previous chapter looked at the tools supporting TDD, but said little about TDD itself. This chapter will use several lengthy examples to show how tests are written, and along the way, you'll get to see how refactorings are performed. We'll also take a quick look at IDE refactoring support.

A consistent theme is code isolation through impostors. *Impostors*, or *test doubles*, are among the most powerful unit-testing techniques available. There is a strong temptation to overuse them, but this can result in overspecified tests.

Impostors are painful to produce by hand, but there are several packages that minimize this pain. Most fall into one of two categories based on how expectations are specified. One uses a domain-specific language, and the other uses a record-replay model. I examine a representative from each camp: pMock and PyMock.

We'll examine these packages in detail in the second half of the chapter, which presents two substantial examples. The same code will be implemented with pMock in the first example and with PyMock in the second example. Along the way, I'll discuss a few tests and demonstrate a few refactorings.[1] Each package imbues the resulting code with a distinct character, and we'll explore these effects.

Moving Beyond Acceptance Tests

Currently, all the logic for the reader application resides within the `main()` method. That's OK, though, because it's all a sham anyway. Iterative design methods focus on taking whatever functional or semifunctional code you have and fleshing it out a little more. The process continues until at some point the code no longer perpetrates a sham, and it stands on its own.

The `main()` method is a hook between Setuptools and our application class. Currently, there is no application class, so what little exists is contained in this method. The next steps create the application class and move the logic from `main()`.

Where do the new tests go? If they're application tests, then they should go into `test_application.py`. However, this file already contains a number of acceptance tests.

1. Here, I'm using the word *few* in a strictly mathematical sense. That is to say that it's smaller than the set of integers. Since there can be only *zero*, *one*, or *many* items, it follows that *many* is larger than the integers. Therefore, *few* is smaller than *many* (for all the good that does anyone).

The two should be separate, so the existing file should be copied to `acceptance_tests.py`. From Eclipse, this is done by selecting `test_application.py` in an explorer view, and then choosing Team ➤ Copy To from the context menu. From the command line, it is copied with `svn copy`.

The application tests are implemented as native Nose tests. Nose tests are functions within a module (a.k.a. a `.py` file). The test modules import assertions from the package `nose.tools`:

```
from nose.tools import *

'''Test the application class RSReader'''
```

The new application class is named `rsreader.application.RSReader`. You move the functionality from `rsreader.application.main` into `rsreader.application.RSReader.main`. At this point, you don't need to create any tests, since the code is a direct product of refactoring. The file `test_application.py` becomes nothing more than a holder for your unwritten tests.

This class RSReader has the single method `main()`. The application's `main()` function creates an instance of RSReader and then delegates it to the instance's `main(argv)` method:

```
def main():
    RSReader().main(sys.argv)


class RSReader(object):

    xkcd_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

    def main(self, argv):
        if argv[1:]:
            print self.xkcd_items
```

The program outputs one line for each RSS item. The line contains the item's date, the feed's title, and the item's title. This is a neatly sized functional chunk. It is one action, and it has a well-defined input and output. The test assertion looks something like this:

```
assert_equals(expected_line, computed_line)
```

You should hard-code the expectation. It's already been done in the acceptance tests, so you can lift it verbatim from there.

```
expected_line = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python"""
assert_equals(expected_line, computed_line)
```

The method `listing_from_item(item, feed)` computes the `expected_line`. It uses the date, feed name, and comic title. You could pass these in directly, but that would expose the inner workings of the method to the callers.

```
expected_line = \
    """Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python"""
computed_line = RSReader().listing_from_item(item, feed)
assert_equals(expected_line, computed_line)
```

So what do items and feeds look like? The values will be coming from FeedParser. As recounted in Chapter 6, they're both dictionaries.

```
expected_line = \
    """Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python"""
item = {'date': "Wed, 05 Dec 2007 05:00:00 -0000",
        'title': "Python"}
feed = {'feed': {'title': "xkcd.com"}}
computed_line = RSReader().listing_from_item(feed, title)
assert_equals(expected_line, computed_line)
```

This shows the structure of the test, but it ignores the surrounding module. Here is the listing in its larger context:

```
from nose.tools import *

from rsreader.application import RSReader

'''Test the application class RSReader'''

def test_listing_from_item():
    expected_line = \
    """Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python"""
    item = {'date': "Wed, 05 Dec 2007 05:00:00 -0000",
            'title': "Python"}
    feed = {'feed': {'title': "xkcd.com"}}
    computed_line = RSReader().listing_from_item(feed, title)
    assert_equals(expected_line, computed_line)
```

The method `list_from_item()` hasn't been defined yet. When you run the test, it fails with an error indicating this. The interesting part of the error message is the following:

```
test.test_application.test_list_from_item ... ERROR

======================================================================
ERROR: test.test_application.test_list_from_item
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/Library/Python/2.5/site-packages/➥
nose-0.10.0-py2.5.egg/nose/case.py", line 202, in runTest
    self.test(*self.arg)
  File "/Users/jeff/Documents/ws/rsreader/src/test/test_application.py",➥
 line 13, in test_listing_from_item
```

```
    computed_line = RSReader().listing_from_item(item, feed)
AttributeError: 'RSReader' object has no attribute 'listing_from_item'

----------------------------------------------------------------------
Ran 4 tests in 0.003s

FAILED (errors=1)
```

This technique is called *relying on the compiler*. The compiler often knows what is wrong, and running the tests gives it an opportunity to check the application. Following the compiler's suggestion, you define the method missing from `application.py`:

```
    def listing_from_item(self, feed, item):
        return None
```

The test runs to completion this time, but it fails:

```
FAIL: test.test_application.test_listing_from_item
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/Library/Python/2.5/site-packages/➥
nose-0.10.0-py2.5.egg/nose/case.py", line 202, in runTest
    self.test(*self.arg)
  File "/Users/jeff/Documents/ws/rsreader/src/test/test_application.py", ➥
line 16, in test_listing_from_item
    assert_equals(expected_line, computed_line)
AssertionError: 'Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python' != None

----------------------------------------------------------------------
```

Now that you're sure the test is checking the right thing, you can write the method body:

```
    def list_from_item(self, feed, item):
        subst = (item['date'], feed['feed']['title'], item['title'])
        return "%s: %s: %s" % subst
```

When you run the test again, it succeeds:

```
test_many_urls_should_print_first_results➥
 (test.acceptance_tests.AcceptanceTests) ... ok
test_no_urls_should_print_nothing➥
 (test.acceptance_tests.AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output➥
 (test.acceptance_tests.AcceptanceTests) ... ok
test.test_application.testing_list_from_item ... ok
```

```
----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK
```

The description of this process takes two pages and several minutes to read. It seems to be a great deal of work, but actually performing it takes a matter of seconds. At the end, there is a well-tested function running in isolation from the rest of the system.

What needs to be done next? The output from all the items in the feed needs to be combined. You need to know what this output will look like. You've already defined this in `acceptance_tests.py`:

```
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
```

Whenever possible, the same test data should be used. When requirements change, the test data is likely to change. Every location with unique test data will need to be modified independently, and each change is an opportunity to introduce new errors.

This time, you'll build up the test as in the previous example. This is the last time that I'll work through this process in so much detail. The assertion in this test is nearly identical to the one in the previous test:

```
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    assert_equals(printed_items, computed_items)
```

The function computes the `printed_items` from the feed and the feed's items. The list of items is directly accessible from the feed object, so it is the only thing that needs to be passed in. The name that immediately comes to my mind is `feed_listing()`. The test line is as follows:

```
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)
```

The feed has two items. The items are indexed by the key `'entries'`:

```
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
              'title': "Python"},
             {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
              'title': "Far Away"}]
    feed = {'feed': {'title': "xkcd.com"}, 'entries': items}
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)
```

Here's the whole test function:

```
def test_feed_listing(self):
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
              'title': "Python"},
             {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
              'title': "Far Away"}]
    feed = {'feed': {'title': "xkcd.com"}, 'entries': items}
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)
```

When you run the test, it complains that the method `feed_listing()` isn't defined. That's OK, though—that's what the compiler is for. However, if you're using Eclipse and Pydev, then you don't have to depend on the compiler for this feedback. The editor window will show a red stop sign in the left margin. Defining the missing method and then saving the change will make this go away.

The first definition you supply for `feed_listing()` should cause the assertion to fail. This proves that the test catches erroneous results.

```
def feed_listing(self, feed):
     return None
```

Running the test again results in a failure rather than an error, so you now know that the test works. Now you can create a successful definition. The simplest possible implementation returns a string constant. That constant is already defined: `xkcd_items`.

```
def feed_listing(self, feed):
    return self.xkcd_items
```

Now run the test again, and it should succeed. Now that it works, you can fill in the body with a more general implementation:

```
def feed_listing(self, feed):
    item_listings = [self.listing_for_item(feed, x) for x
                     in feed['entries']]
    return "\n".join(item_listings)
```

When I ran this test on my system, it succeeded. However, there was an error. Several minutes after I submitted this change, I received a failure notice from my Windows Buildbot (which I set up while you weren't looking). The error indicates that the line separator is wrong on the Windows system. There, the value is `\r\n` rather than the `\n` used on UNIX systems. The solution is to use `os.linesep` instead of a hard-coded value:

```
import os
...
def feed_listing(self, feed):
    item_listings = [self.listing_for_item(feed, x) for x
```

```
                    in feed['entries']]
    return os.linesep.join(item_listings)
```

At this point, you'll notice several things—there's a fair bit of duplication in the test data:

- xkcd_items is present in both acceptance_tests.py and application_tests.py.
- The feed items are partially duplicated in both application tests.
- The feed definition is partially duplicated in both application tests.
- The output data is partially duplicated in both tests.

As it stands, any changes in the expected results will require changes in each test function. Indeed, a change in the expected output will require changes not only in multiple functions, but in multiple files. Any changes in the data structure's input will also require changes in each test function.

In the first step, you'll extract the test data from test_feed_listing:

```
printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

def test_feed_listing(self):
    items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
              'title': "Python"},
             {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
              'title': "Far Away"}]
    feed = {'feed': {'title': "xkcd.com"}, 'entries': items}
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)
```

You save the change and run the test, and it should succeed. The line defining printed_items is identical in both acceptance_tests.py and application_tests.py, so the definition can and should be moved to a common location. That module will be test.shared_data:

```
$ ls tests -Fa
```

```
__init__.py      acceptance_tests.pyc      application_tests.pyc
__init__.pyc      acceptance_tests.py      application_tests.py
shared_data.py
```

```
$ cat shared_data.py
```

```
"""Data common to both acceptance tests and application tests"""

__all__ = ['printed_items']
```

```
printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
```

```
$ cat acceptance_tests.py
```

```
...
from tests.shared_data import *
...
```

```
$ cat application_tests.py
```

```
...
from tests.shared_data import *
...
```

The `__all__` definition explicitly defines the module's exports. This prevents extraneous definitions from polluting client namespaces when wildcard imports are performed. Declaring this attribute is considered to be a polite Python programming practice.

The refactoring performed here is called *triangulation*. It is a method for creating shared code. A common implementation is not created at the outset. Instead, the code performing similar functions is added in both places. Both implementations are rewritten to be identical, and this precisely duplicated code is then extracted from both locations and placed into a new definition.

This sidesteps the ambiguity of what the common code might be by providing a concrete demonstration. If the common code couldn't be extracted, then it would have been a waste of time to try to identify it at the outset.

The test `test_listing_for_item` uses a subset of `printed_items`. This function tests individual lines of output, so it's used to break the `printed_items` list into a list of strings:

```
expected_items = [
    "Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python",
    "Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away""",
]
printed_items = os.linesep.join(item_listings)
```

You save the change to `shared_data.py`, run the tests, and the tests succeed. This verifies that the data used in `test_feed_listing()` has not changed. Now that the data is in a more useful form, you can change the references within `test_listing_for_item()`. You remove the definition, and the assertion now uses `expected_items`.

```
def test_listing_from_item():
    item = {'date': "Wed, 05 Dec 2007 05:00:00 -0000",
            'title': "Python"}
    feed = {'title': "xkcd.com"}
    computed_line = RSReader().listing_from_item(feed, item)
    assert_equals(expected_items[0], computed_line)
```

You run the test, and it succeeds. The expectations have been refactored, so it is now time to move on to the test fixtures. The values needed by `test_listing_from_item()` are already defined in `test_feed_listing()`, so you'll extract them from that function and remove them from the other:

```
from tests.shared_data import *

items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
          'title': "Python"},
         {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
          'title': "Far Away"}]
feed = {'feed': {'title': "xkcd.com"}, 'entries': items}

def test_feed_listing(self):
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)

def test_listing_from_item():
    computed_line = RSReader().listing_from_item(feed, items[0])
    assert_equals(expected_items[0], computed_line)
```

Renaming

Looking over the tests, it seems that there is still at least one smell. The name `printed_items` isn't exactly accurate. It's the expected output from reading xkcd, so `xkcd_output` is a more accurate name. This will mandate changes in several locations, but this process is about to become much less onerous. The important thing for the names is that they are consistent.

Inaccurate or awkward names are anathema. They make it hard to communicate and reason about the code. Each new term is a new definition to learn. Whenever a reader encounters a new definition, she has to figure out what it really means. That breaks the flow, so the more inconsistent the terminology, the more difficult it is to review the code. Readability is vital, so it is important to correct misleading names.

Traditionally, this has been difficult. Defective names are scattered about the code base. It helps if the code is loosely coupled, as this limits the scope of the changes; unit tests help to ensure that the changes are valid, too, but neither does anything to reduce the drudgery of find-and-replace. This is another area where IDEs shine.

Pydev understands the structure of the code. It can tell the difference between a function `foo` and an attribute `foo`. It can distinguish between method `foo` in class `X` and method `foo` in class `Y`, too. This means that it can rename intelligently.

This capability is available from the refactoring menu, which is available from either the main menu bar or the context menu. To rename a program element, you select its text in an editor. In this case, you're renaming the variable `printed_items`. From the main menu bar, select Refactoring ➤ Rename. (It's the same from the context menu.) There are also keyboard accelerators available for this, and they're useful to know.

Choosing the Rename menu item brings up the window shown in Figure 7-1. Enter the new name **xkcd_output**.

Figure 7-1. *The Rename refactoring window*

At this point, you can preview the changes by clicking the Preview button. This brings up the refactoring preview window shown in Figure 7-2.

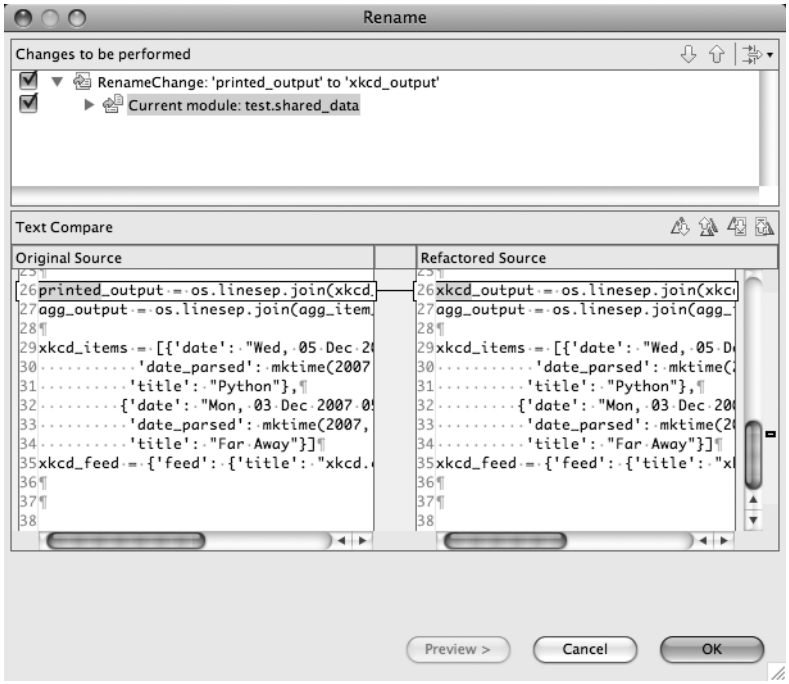

Figure 7-2. *The refactoring preview window*

Each candidate refactoring can be viewed and independently selected or unselected though the check box to its left. Pydev not only checks the code proper, but it checks string literals and comments, too, so the preview is often a necessary step, even with simple renames.

I find it edifying to see how many places the refactoring touches the program. It reminds me how the refactored code is distributed throughout the program, and it conveys an impression of how tightly coupled the code is.

When satisfied, click OK, and the refactoring will proceed. After a few seconds, the selected changes will be complete.

Overriding Existing Methods: Monkeypatching

The code turning a feed object into text has been written. The next step converts URLs into feed objects. This is the magic that FeedParser provides. The test harness doesn't have control over network connections, and the Net at large can't be controlled without some pretty involved network hackery. More important, the tests shouldn't be dependent on external resources unless they're included as part of the build.

All of these concerns can be surmounted by hacking FeedParser on the fly. Its parse routine is temporarily replaced with a function that behaves as desired. The test is defined first:

```
def test_feed_from_url():
    url = "http://www.xkcd.com/rss.xml"
    assert_equals(feed, RSReader().feed_from_url(url))
```

The test method runs, and it fails with an error stating that `feed_from_url()` has not been defined. The method is defined as follows:

```
def feed_from_url(self, url):
    return None
```

The test is run, and fails with a message indicating that `feed` does match the results returned from `feed_from_url()`. Now for the fun stuff. A fake parse method is defined in the test, and it is hooked into FeedParser. Before this is done, the real parse method is saved, and after the test completes, the saved copy is restored.

```
import feedparser
...
def test_feed_from_url():
   url = "http://www.xkcd.com/rss.xml"
    def  parse_stub(url): # define stub
        return feed
    real_parse = feedparser.parse # save real value
    feedparser.parse = parse_stub # attach stub
    try:
        assert_equals(feed, RSReader().feed_from_url(url))
    finally:
        feedparser.parse = real_parse # restore real value
```

The test is run, and it fails in the same manner as before. Now the method is fleshed in:

```
import feedparser
...
    def feed_from_url(self, url):
        return feedparser.parse(url)
```

The test runs, and it succeeds.

Monkeypatching and Imports

In order for monkeypatching to work, the object overridden in the test case and the object called from the subject must refer to the same object. This generally means one of two things. If the subject imports the module containing the object to be overridden, then the test must do the same. This is illustrated in Figure 7-3. If the subject imports the overridden object from the module, then the test must import the subject module, and the reference in the subject module must be overridden. This is reflected in Figure 7-4.

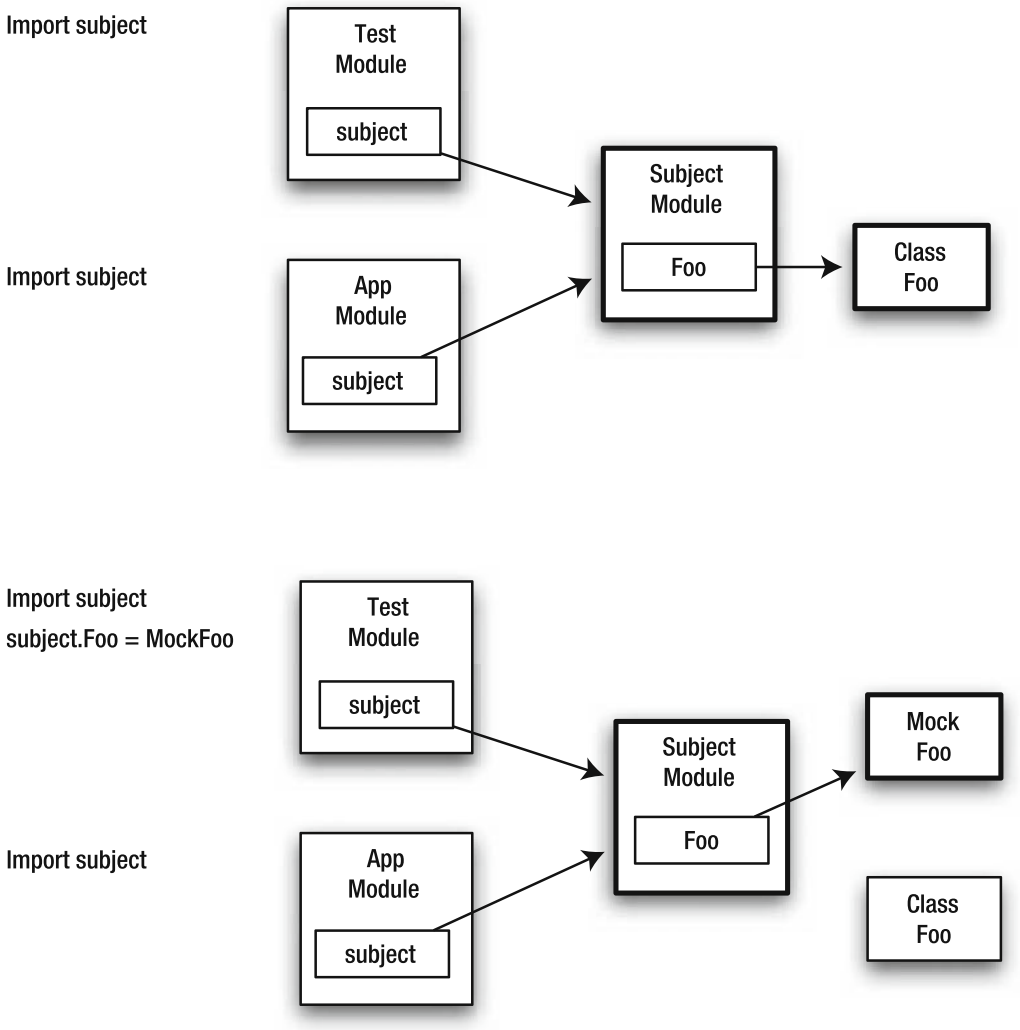

Figure 7-3. *Replacing an object when the subject imports the entire module containing the object*

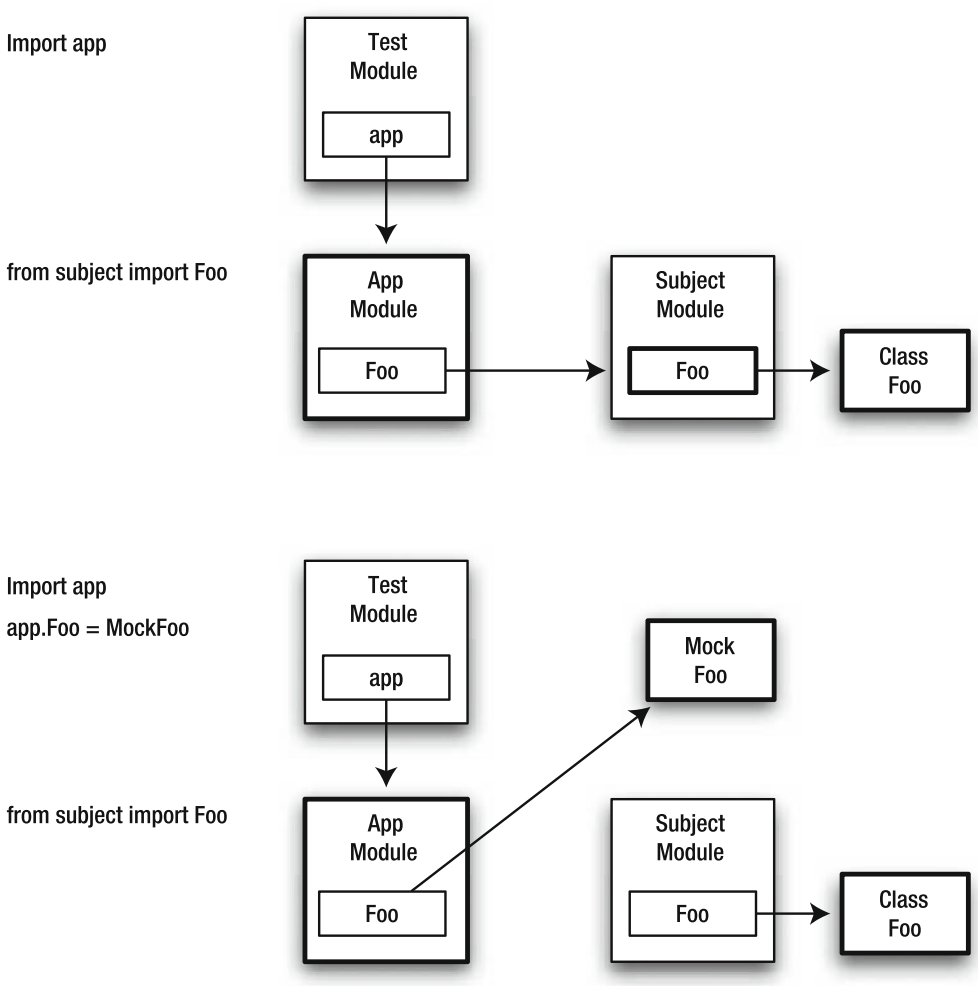

Figure 7-4. *Replacing an object when the subject directly imports the object*

It is tempting to import the subject module directly into the test's namespace. However, this does not work. Altering the test's reference doesn't alter the subject's reference. It results in the situation shown in Figure 7-5, where the test points to the mock, but the rest of the code still points to the real object. This is why it is necessary to alter the reference to the subject module, as in Figure 7-4.

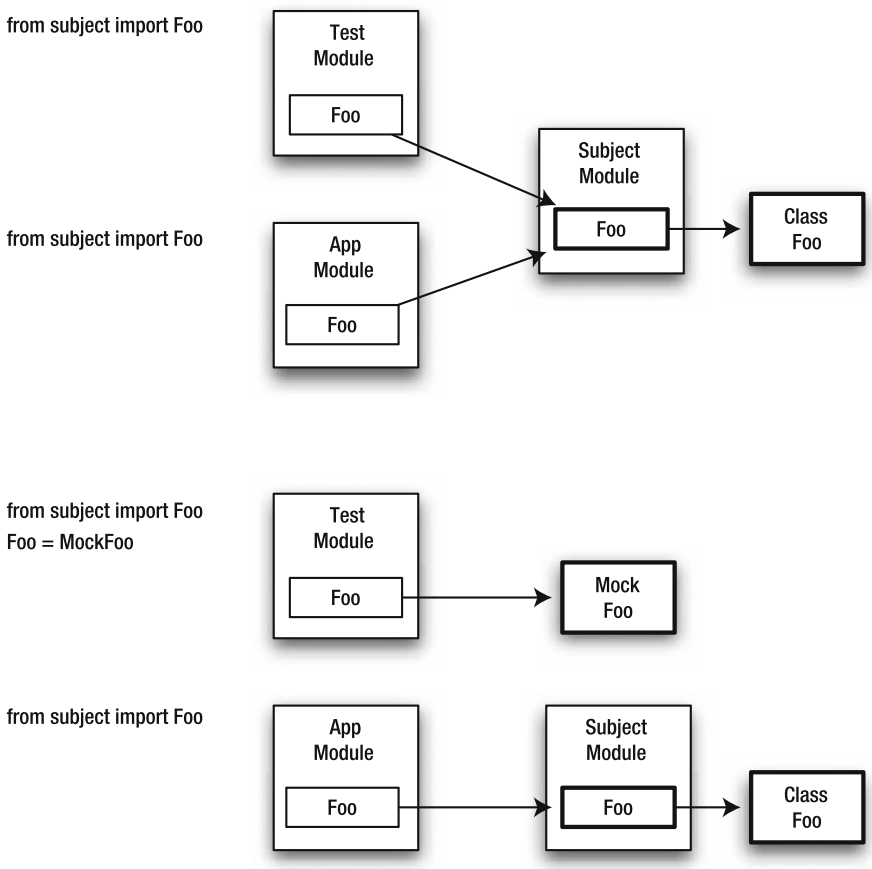

Figure 7-5. *Why replacing an object imported directly into the test's namespace doesn't work*

The Changes Go Live

At this point, URLs can be turned into feeds, and feeds can be turned into output. Everything is available to make a working application. The new `main()` method is as follows:

```
    def main(self, argv):
        if argv[1:]:
            url = argv[1]
            print self.listing_from_feed(self.feed_from_url(url))
```

The test suite is run, and the acceptance tests fail:

```
test_many_urls_should_print_first_results➥
(test.acceptance_tests.AcceptanceTests) ... FAIL
test_no_urls_should_print_nothing (test.acceptance_tests.AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output (test.acceptance_tests.AcceptanceTests)➥
 ... FAIL
```

```
test.test_application.test_list_from_item ... ok
test.test_application.test_list_from_feed ... ok
test.test_application.test_list_from_url ... ok
test.test_application.test_feed_from_url ... ok

...

======================================================================
FAIL: test_should_get_one_URL_and_print_output➥
(test.acceptance_tests.AcceptanceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/Documents/ws/rsreader/src/test/acceptance_tests.py",➥
line 25, in test_should_get_one_URL_and_print_output
    self.assertStdoutEquals(self.printed_items + "\n")
  File "/Users/jeff/Documents/ws/rsreader/src/test/acceptance_tests.py",➥
line 38, in assertStdoutEquals
    self.assertEquals(expected_output, sys.stdout.getvalue())
AssertionError: 'Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com:➥
Python\nMon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away\n' != '\n'

----------------------------------------------------------------------
Ran 7 tests in 0.015s

FAILED (failures=2)
```

The acceptance tests are now using real code, and the test cases have a problem.

Using Data Files

The failing tests are trying to access the file `xkcd.rss.xml`. This file doesn't exist, so the code is dying. These files should contain real RSS data that has been trimmed down to produce the expected results. I've done this already. You can simply download the file from `www.theblobshop.com/famip/xkcd.rss.xml` to a new directory, `src/test/data`.

With this file in place, the tests still fail. The acceptance tests need to specify the full path to the data file. The path is relative to the test being run, so it can be extracted from the test module's `__file__` attribute:

```
import StringIO
import sys
from unittest import TestCase

from test.shared_data import *
from rsreader.application import main

module = sys.modules[__name__]
this_dir = os.path.dirname(os.path.abspath(module.__file__))
```

```
xkcd_rss_xml = os.path.join(this_dir, 'data', 'xkcd.rss.xml')

class AcceptanceTests(TestCase):

    def setUp(self):
        self.old_value_of_stdout = sys.stdout
        sys.stdout = StringIO.StringIO()
        self.old_value_of_argv = sys.argv

    def tearDown(self):
        sys.stdout = self.old_value_of_stdout
        sys.argv = self.old_value_of_argv

    def test_should_get_one_URL_and_print_output(self):
        sys.argv = ["unused_prog_name", xkcd_rss_xml]
        main()
        self.assertStdoutEquals(expected_output + "\n")

    def test_no_urls_should_print_nothing(self):
        sys.argv = ["unused_prog_name"]
        main()
        self.assertStdoutEquals("")

    def test_many_urls_should_print_first_results(self):
        sys.argv = ["unused_prog_name", xkcd_rss_xml, "excess"]
        main()
        self.assertStdoutEquals(expected_output + "\n")

    def assertStdoutEquals(self, expected_output):
        self.assertEquals(expected_output, sys.stdout.getvalue())
```

With this change in place, the tests run, and they all succeed. The first pass at the application is complete. It can be installed and run from the command line.

Isolation

Isolating the components under test from the system at large is a major theme in unit testing. You've seen this with the method `feed_from_url()`. It has a dependency upon the function `feedparser.parse()` that was temporarily separated by the replacement of the function with a fake implementation.

These dependencies come in three main forms:

Dependencies can be introduced to functions and methods as arguments: In the function call `f(x)`, the function depends upon `x`. The object may be passed as an argument to other callables, methods may be invoked upon it, it may be returned, it may be raised as an exception, or it may be captured. When captured, it may be assigned to a variable or an attribute or bundled into a closure.

Dependencies can be introduced as calls to global entities: Global entities include packages, classes, and functions. In languages such as C and Java, these are static declarations, to some extent corresponding to the type system. In Python, these are much more dynamic. They're first-class objects that are not only referenced through the global namespace—they can be introduced through arguments as well. The method `f(x)` introduces a dependency on the package `os`:

```
def f(filename):
   x = os.listdir(filename)
```

Dependencies can be introduced indirectly: They are introduced as the return values from functions and methods, as exceptions, and as values retrieved from attributes. These are in some sense the product of the first two dependency classes. Modeling these is inherent in accurately modeling the first.

To test in isolation, these dependencies must be broken. Choosing an appropriate design is the best way to do this. The number of objects passed in as arguments should be restricted. The number of globals accessed should be restricted, too, and as little should be done with return values as possible. Even more important, side effects (assignments) should be restricted as much as possible. However, coupling is inescapable. A class with no dependencies and no interactions rarely does anything of interest.

The remaining dependencies are severed through a set of techniques known as mocking. *Mocking* seeks to replace the external dependencies with an impostor. The impostor has just enough functionality to allow the tested unit to function. Impostors perform a subset of these functions:

- Fulfilling arguments
- Avoiding references to objects outside the unit
- Tracking call arguments
- Forcing return values
- Forcing exceptions
- Verifying that calls were made
- Verifying call ordering

There are four categories of impersonators:

Dummies: These are minimal objects. They are created so that the system as a whole will run. They're important in statically typed languages. An accessor method may store a derived class, but no such class exists in the section of the code base under examination. A class is derived from the abstract base class, and the required abstract methods are created, but they do nothing, and they are never called in the test. This allows the tests to compile. In Python, it is more common to see these in the case of conditional execution. An argument may only be used in one branch of the conditional. If the test doesn't exercise that path, then it passes a dummy in that argument, and the dummy is never used in the test.

Stubs: These are more substantial than dummies. They implement minimal behavior. In stubs, the results of a call are hard-coded or limited to a few choices. The isolation of `feed_from_url()` is a clear-cut example of this. The arguments weren't checked, and there weren't any assertions about how often the method stub was called, not even to ensure that it was called at all. Implementing any of this behavior requires coding.

Mocks: These are like stubs, but they keep track of expectations and verify that they were met. Different arguments can produce different outcomes. Assertions are made about the calls performed, the arguments passed, and how often those calls are performed, or even if they are performed at all. Values returned or exceptions raised are tracked, too. Performing all of this by hand is involved, so many mock object frameworks have been created, with most using a concise declarative notation.

Fakes: These are more expansive and often more substantial than mock objects. They are typically used to replace a resource-intensive or expansive subsystem. The subsystem might use vast amounts of memory or time, with time typically being the important factor for testing. The subsystem might be expansive in the sense that it depends on external resources such as a network-locking service, an external web service, or a printer. A database is the archetypical example of a faked service.

Rolling Your Own

Dummies are trivial to write in Python. Since Python doesn't check types, an arbitrary string or numeric value suffices in many cases.

In some cases, you'll want to add a small amount of functionality to an existing class or override existing functionality with dummies or stubs. In this case, the test code can create a subclass of the subject. This is commonly done when testing a base class. It takes little effort, but Python has another way of temporarily overriding functionality, which was shown earlier.

Monkeypatching takes an existing package, class, or callable, and temporarily replaces it with an impostor. When the test completes, the monkeypatch is removed. This approach isn't easy in most statically typed languages, but it's nearly trivial in Python. With instances created as test fixtures, it is not necessary to restore the monkeypatch, since the change will be lost once an individual test completes. Packages and classes are different, though. Changes to these persist across test cases, so the old functionality must be restored.

There are several drawbacks to monkeypatching by hand. Undoing the patches requires tracking state, and the problem isn't straightforward—particularly when properties or subclasses are involved. The changes themselves require constructing the impostor, so this piles up difficulties.

Hand-coding mocks is involved. Doing one-offs produces a huge amount of ugly setup code. The logic within a mocked method becomes tortuous. The mocks end up with a mishmash of methods mapping method arguments to output values. At the same time, this interacts with method invocation counting and verification of method execution. Any attempt to really address the issues in a general way takes you halfway toward creating a mock object package, and there are already plenty of those out there. It takes far less time to learn how to use the existing ones than to write one of your own.

Python Quirks

In languages such as Java and C++, subclassing tends to be preferred to monkeypatching. Although Python uses inheritance, programmers rely much more on duck typing—if it looks like a duck and quacks like a duck, then it must be a duck. Duck typing ignores the inheritance structure between classes, so it could be argued that monkeypatching is in some ways more Pythonic.

In many other languages, instance variables and methods are distinct entities. There is no way to intercept or modify assignments and access. In these languages, instance variables directly expose the implementation of a class. Accessing instance variables forever chains a caller to the implementation of a given object, and defeats polymorphism. Programmers are exhorted to access instance values through getter and setter methods.

The situation is different in Python. Attribute access can be redirected through hidden getter and setter methods, so attributes don't directly expose the underlying implementation. They can be changed at a later date without affecting client code, so in Python, attributes are valid interface elements.

Python also has operator overloading. Operator overloading maps special syntactic features onto underlying functions. In Python, array element access maps to the function `__getitem__()`, and addition maps to the method `__add__()`. More often than not, modern languages have some mechanism to accomplish this magic, with Java being a dogmatic exception.

Python takes this one step further with the concept of protocols. *Protocols* are sequences of special methods that are invoked to implement linguistic features. These include generators, the `with` statement, and comparisons. Many of Python's more interesting linguistic constructions can be mocked by understanding these protocols.

Mocking Libraries

Mocking libraries vary in the features they provide. The method of mock construction is the biggest discriminator. Some packages use a domain-specific language, while others use a record-playback model.

Domain-specific languages (DSLs) are very expressive. The constructed mocks are very easy to read. On the downside, they tend to be very verbose for mocking operator overloading and for specifying protocols. DSL-driven mock libraries generally descend from Java's jMock. It has a strong bias toward using only vanilla functions, and the descendent DSLs reflect this bias.

Record-replay was pioneered by Java's EasyMock. The test starts in a record mode, mock objects are created, and the expected calls are performed on them. These calls are recorded, the mock is put into playback mode, and the calls are played back. The approach works very well for mocking operator overloading, but its implementation is fraught with peril. Unsurprisingly, the additional work required to specify results and restrictions makes the mock setup more confusing than one might expect.

Two mocking libraries will be examined in this chapter: pMock and PyMock. *pMock* is a DSL-based mocking system. It only works on vanilla functions, and its DSL is clear and concise. Arguments to mocks may be constrained arbitrarily, and pMock has excellent failure reporting. However, it is poor at handling many Pythonic features, and monkeypatching is beyond its ken.

PyMock combines mocks, monkeypatching, attribute mocking, and generator emulation. It is primarily based on the record-replay model, with a supplementary DSL. It handles generators, properties, and magic methods. One major drawback is that its failure reports are fairly opaque.

In the next section, the example is expanded to handle multiple feeds. The process is demonstrated using first pMock, and then PyMock.

Aggregating Two Feeds

In this example, two separate feeds need to be combined, and the items in the output must be sorted by date. As with the previous example, the name of the feed should be included with its title, so the individual feed items need to identify where they come from. A session might look like this:

```
$ rsreader http://www.xkcd.com/rss.xml http://www.pvponline.com/rss.xml
```

```
Thu, 06 Dec 2007 06:00:36 +0000: PvPonline: Kringus Risen - Part 4
Wed, 05 Dec 2007 06:00:45 +0000: PvPonline: Kringus Risen - Part 3
Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away
```

The feeds must be retrieved separately. This is a design constraint from FeedParser. It pulls only one at a time, and there is no way to have it combine the two feeds. Even if the package were bypassed, this would still be a design constraint. The feeds must be parsed separately before they can be combined. In all cases, every feed item needs to be visited once.

The feeds could be combined incrementally, but doing things incrementally tends to be tougher than working in batches. There are multiple approaches to combining the feeds, and they all fundamentally answer the question: how do you locate the feed associated with an item?

One approach places the intelligence outside the feeds. One list aggregates either the feeds or the fed items. A dictionary maps the individual items back to their parent feeds. This can be wrapped into a single class that handles aggregation and lookup. The number of internal data structures is high, but it works.

In another approach, the FeedParser objects can be patched. A new key pointing back the parent feed is added to each entry. This involves mucking about with the internals of code belonging to third-party packages.

Creating a parallel set of data structures (or new classes) is yet another option. The interesting aspects of the aggregated feeds are modeled, and the uninteresting ones are ignored. The downsides are that we're creating a duplicate object hierarchy, and it duplicates some of the functionality in FeedParser. The upsides are that it is very easy to build using a mocking framework, and it results in precisely the objects and classes needed for the project.

What routines are needed? The method for determining this is somewhat close to pseudocode planning. Starting with a piece of paper, the new section of code is outlined, and the outline is translated into a series of tests. The list isn't complete or exhaustive—it just serves as a starting point.

```
def _pending_test_new_main():
    """Hooking in new code to main"""

def _pending_test_get_feeds_from_urls():
    """Should get a feed for every URL"""

def _pending_test_combine_feeds():
    """Should combine feeds into a list of FeedEntries"""

def _pending_test_add_single_feed():
    """Should add a single feed to a set of feeds"""

def _pending_test_create_entry():
    """Create a feed item from a feed and a feed entry"""

def _pending_test_feed_listing_is_sorted():
    """Should sort the aggregate feed listing"""

def _pending_test_feed_entry_listing():
    """Should produce a correctly formatted listing from a feed entry"""
```

The string `_pending_` prefixing each test tells those reading your code that the tests are not complete. The starting underscore tells Nose that the function is not a test. When you begin writing the test, the string `_pending_` is removed.

A Simple pMock Example

pMock is installed with the command `easy_install pmock`. It's a pure Python package, so there's no compilation necessary, and it should work on any system.

A simple test shows how to use pMock. The example will calculate a triangle's perimeter:

```
def test_perimeter():
    assert_equals(4, perimeter(triangle))
```

pMock imitates the triangle object:

```
def test_perimeter():
    triangle = Mock()
    assert_equals(4, perimeter(triangle))
```

The expected method calls `triangle.side(0)` and `triangle.side(1)` need to be modeled. They return `1` and `3`, respectively.

```
def test_perimeter():
    triangle = Mock()
    triangle.expects(once()).side(eq(0)).will(return_value(1))
    triangle.expects(once()).side(eq(1)).will(return_value(3))
    assert_equals(4, perimeter(triangle))
```

Each expectation has three parts. The `expects()` clause determines how many times the combination of method and arguments will be invoked. The second part determines the method name and argument constraints. In this case, the calls have one argument, and it must be equal to `0` or `1`. `eq()` and `same()` are the most common constraints, and they are equivalent to Python's `==` and `is` operators. The optional `will()` clause determines the method's actions. If present, the method will either return a value or raise an exception.

The simplest method fulfilling the test is the following:

```
def perimeter(triangle):
    return 4
```

When you run the test, it succeeds even though it doesn't call the triangle's `side()` methods. You must explicitly check each mock to ensure that its expectations have been met. The call `triangle.verify()` does this:

```
def test_perimeter():
    triangle = Mock()
    triangle.expects(once()).side(eq(0)).will(return_value(1))
    triangle.expects(once()).side(eq(1)).will(return_value(3))
    assert_equals(4, perimeter(triangle))
    triangle.verify()
```

Now when you run it the test, it fails. The following definition satisfies the test:

```
def perimeter(triangle):
   return triangle.side(0) + triangle.side(1)
```

Implementing with pMock

To use mock objects, there must be a way of introducing them into the tested code. There are four possible ways of doing this from a test. They can be passed in class variables, they can be assigned as instance variables, they can be passed in as arguments, or they can be introduced from the global environment.

Test: Defining combine_feeds

Mocking calls to `self` poses a problem. This could be done with monkeypatching, but that's not a feature offered by pMock. Instead, `self` is passed in as a second argument, and it introduces the mock. In this case, the auxiliary `self` is used to help aggregate the feed.

```
def test_combine_feeds():
    """Combine one or more feeds"""
    aggregate_feed = Mock()
    feeds = [Mock(), Mock()]
    aggregate_feed.expects(once()).add_single_feed(same(feeds[0]))
    aggregate_feed.expects(once()).add_single_feed(same(feeds[1]))
    RSReader().combine_feeds(aggregate_feed, feeds)
    aggregate_feed.verify()
```

The test fails. The method definition fulfilling the test is as follows:

```
def combine_feeds(self, aggregate_feed, feeds):
    for x in feeds:
       aggregate_feed.add_single_feed(x)
```

The test now succeeds.

Test: Defining add_single_feed

The next test is `test_add_single_feed()`. It verifies that `add_single_feed()` creates an aggregate entry for each entry in the feed:

```
def test_add_singled_feed():
    """Should add a single feed to a set of feeds"""
    entries = [Mock(), Mock()]
    feed = {'entries': entries}
    aggregate_feed = Mock()
    aggregate_feed.expects(once()).create_entry(same(feed), same(entries[0]))
    aggregate_feed.expects(once()).create_entry(same(feed), same(entries[1]))
    RSReader().add_single_feed(aggregate_feed, feed)
    aggregate_feed.verify()
```

The test fails. The method `RSReader.add_single_feed()` is defined:

```
def add_single_feed(self, feed_aggregator, feed):
    for e in feed['entries']:
        feed_aggregator.create_entry(e)
```

The test now passes. There is a problem, though. The two tests have different definitions for `add_single_feed`. In the first, it is called as `add_single_feed(feed)`. In the second, it is called as `add_single_feed(aggregator_feed, feed)`. In a statically typed language, the development environment or compiler would catch this, but in Python, it is not caught. This is both a boon and a bane. The boon is that a test can completely isolate a single method call from the rest of the program. The bane is that a test suite with mismatched method definitions can run successfully.

The second test's definition is obviously the correct one, so you revise the first one. It is also apparent that the same problem will exist for `create_entry`, so you fix this expectation at the same time.

```
def test_combine_feeds():
    """Combine one or more feeds"""
    aggregate_feed = Mock()
    feeds = [Mock(), Mock()]
    aggregate_feed.expects(once()).add_single_feed(same(aggregate_feed),
        same(feeds[0]))
    aggregate_feed.expects(once()).add_single_feed(same(aggregate_feed),
        same(feeds[1]))
    subject = RSReader().combine_feeds(aggregate_feed, feeds)
    aggregate_feed.verify()
```

```
def test_add_singled_feed():
    """Should add a single feed to a set of feeds"""
    entries = [Mock(), Mock()]
    feed = {'entries': entries}
    aggregate_feed = Mock()
    aggregate_feed.expects(once()).create_entry(same(aggregate_feed),
        same(feed), same(entries[0]))
    aggregate_feed.expects(once()).create_entry(same(aggregate_feed),
        same(feed), same(entries[1]))
    RSReader().add_single_feed(aggregate_feed, feed)
    aggregate_feed.verify()
```

And the method definitions are also changed:

```
    def combine_feeds(self, feed_aggregator, feeds):
        for f in feeds:
            feed_aggregator.add_single_feed(feed_aggregator, f)

    def add_single_feed(self, feed_aggregator, feed):
        for e in feed['entries']:
            feed_aggregator.create_entry(feed_aggregator, feed, e)
```

In some sense, strictly using mock objects induces a style that obviates the need for `self`. It maps very closely onto languages with multimethods. While the second copy of `self` is merely conceptually ugly in other languages, Python's explicit `self` makes it typographically ugly, too.

Refactoring: Extracting AggregateFeed

The second `self` variable serves a purpose, though. If named to reflect its usage, then it indicates which class the method belongs to. If that class doesn't exist, then it strongly suggests that it should be created. In this case, the class is `AggregateFeed`.

You create the new class, and one by one you move over the methods from `RSReader`. First you modify the test, and then you move the corresponding method. You repeat this process until all the appropriate methods have been moved.

```
from rsreader.application import AggregateFeed, RSReader
...
def test_combine_feeds():
    """Should combine feeds into a list of FeedEntries"""
    subject = AggregateFeed()
    mock_feeds = [Mock(), Mock()]
    aggregate_feed = Mock()
    aggregate_feed.expects(once()).add_single_feed(same(aggregate_feed),
                                           same(mock_feeds[0]))
    aggregate_feed.expects(once()).add_single_feed(same(aggregate_feed),
                                           same(mock_feeds[1]))
    subject.combine_feeds(aggregate_feed, mock_feeds)
    aggregate_feed.verify()
```

The test fails because the class `AggregateFeed` is not defined. The new class is defined:

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    pass
```

The tests are run, and they still fail, but this time because the method `AggregateFeed.combine_feeds()` is not defined. The method is moved to the new class:

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    def combine_feeds(self, feed_aggregator, feeds):
        for f in feeds:
            feed_aggregator.add_single_feed(feed_aggregator, f)
```

Now the test succeeds. With mock objects, methods can be moved easily between classes without breaking the entire test suite.

Refactoring: Moving add_single_feed

The process is continued with `test_add_single_feed()`. You alter `test_add_single_feed` to create `AggregateFeed` as the test subject:

```
def test_add_single_feed():
    """Should add a single feed to a set of feeds"""
    entries = [Mock(), Mock()]
    feed = {'entries': entries}
    aggregate_feed = Mock()
    aggregate_feed.expects(once()).create_entry(same(aggregate_feed),
        same(feed), same(entries[0]))
    aggregate_feed.expects(once()).create_entry(same(aggregate_feed),
        same(feed), same(entries[1]))
    AggregateFeed().add_single_feed(aggregate_feed, feed)
    aggregate_feed.verify()
```

The test fails. You move the method from RSReader to `AggregateFeed` to fix this:

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    def combine_feeds(self, feed_aggregator, feeds):
        for f in feeds:
            feed_aggregator.add_single_feed(feed_aggregator, f)

    def add_single_feed(self, feed_aggregator, feed):
        for e in feed['entries']:
            feed_aggregator.create_entry(feed_aggregator, feed, e)
```

When you run the test it now succeeds.

Test: Defining create_entry

The next test is `test_create_entry()`. It takes an existing feed and an entry from that feed, and converts it to the new model. The new model has not been defined. The test assumes that it uses a factory to produce new instances. This factory is an instance variable in `AggregateFeed`. The object created by the factory is added to `aggregate_feed()`:

```
def test_create_entry():
    """Create a feed item from a feed and a feed entry"""
    agg_feed = AggregateFeed()
    agg_feed.feed_factory = Mock()
    (aggregate_feed, feed, entry, converted) = (Mock(), Mock(), Mock(), Mock())
    agg_feed.feed_factory.expects(once()).from_parsed_feed(same(feed),
        same(entry)).will(return_value(converted))
    aggregate_feed.expects(once()).add(same(converted))
    agg_feed.create_entry(aggregate_feed, feed, entry)
    aggregate_feed.verify()
```

The test fails, so you add the following code:

```
def create_entry(self, feed_aggregator, feed, entry):
    """Create a new feed entry and aggregate it"""
    feed_aggregator.add(self.feed_factory.from_parsed_feed(feed, entry))
```

And now the test succeeds.

Test: Ensuring That AggregateFeed Creates a FeedEntry Factory

`create_entry` has given birth to three new tests:

```
def _pending_test_aggregate_feed_creates_factory():
    """Verify that the AggregateFeed object creates a factory when instantiated"""

def _pending_test_feed_entry_from_parsed_feed():
    """Factory method to create a new feed entry from a parsed feed"""

def _pending_test_add():
    """Add an a feed entry to the aggregate"""
```

Checking to see if the `AggregateFeed` creates a factory seems like the easiest test to me, so we'll tackle it first, but it does take a little consideration of the program's larger structure.

Each entry in a feed will be represented by an instance of the class `FeedEntry`. The factory could be a function or another class, but that's probably making things a little too complicated. Instead, it will be a method within `FeedEntry`.

```
from rsreader.app import AggregateFeed, FeedEntry, RSReader
...
def test_aggregate_feed_creates_factory():
    """Verify that the AggregatedFed object creates a factory
       when instantiated"""
    assert_equals(FeedEntry, AggregateFeed().feed_factory)
```

The test fails because the `FeedEntry` class is not defined yet.

```
class FeedEntry(object):
    """Combines elements of a feed and a feed entry.
       Allows multiple feeds to be aggregated without losing
       feed specific information."""
```

The test now runs, but fails because `AggregateFeed.__init__` is not defined.

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    def __init__(self):
        self.feed_factory = FeedEntry
```

The test now passes.

Test: Defining add

The next test you'll write is `test_add()`.The `add()` method records the newly aggregated methods. At this point, the testing becomes very concrete.

```
from sets import Set
...
def test_add():
    """Add an a feed entry to the aggregate"""
    entry = Mock()
    subject = AggregateFeed()
    subject.add(entry)
    assert_equals(Set([entry]), subject.entries)
```

The test fails. The corresponding definition is as follows:

```
from sets import Set
...
def add(self, entry):
     self.entries = Set([entry])
```

The test passes this time. This definition is fine for a single test, but it needs to be refactored into something more useful.

Test: AggregateFeed.entries Is Always Initialized to a Set

The empty set should be defined when a feed is created. A new test ensures this:

```
def test_entries_is_always_defined():
    """The entries set should always be defined"""
    assert_equals(Set(), AggregateFeed().entries)
```

The test fails. You should modify the constructor to fulfill the expected conditions:

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    def __init__(self):
        self.entries = Set()
        self.feed_factory = FeedEntry
```

The test now succeeds. The next step is refactoring `add()`:

```
def add(self, entry):
     self.entries.add(entry)
```

The tests still succeed, so the refactoring worked.

Test: Defining FeedEntry.from_parsed_feed

Now it is time to verify the `FeedEntry` factory's operation. The required feed objects already exist within the tests, and you'll reuse them here.

```
def test_feed_entry_from_parsed_feed():
    """Factory method to create a new feed entry from a parsed feed"""
    feed_entry = FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_items[0]['date'], feed_entry.date)
    assert_equals(xkcd_items[0]['title'], feed_entry.title)
    assert_equals(xkcd_feed['feed']['title'], feed_entry.feed_title)
```

The test runs and fails. The method `from_parsed_feed()` is defined as follows:

```
@classmethod
def from_parsed_feed(cls, feed, entry):
    """Factory method producing a new object from an existing feed."""
    feed_entry = FeedEntry()
    feed_entry.date = entry['date']
    feed_entry.feed_title = feed['feed']['title']
    feed_entry.title = entry['title']
    return feed_entry
```

Test: Defining feed_entry_listing

At this point, `_pending_test_aggregate_item_listing()` jumps out from the list of pending tests. It pertains to `FeedEntry`, and it looks like `FeedEntry` has all the information needed.

```
def test_feed_entry_listing():
    """Should produce a correctly formatted listing from a feed entry"""
    entry = FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_listings[0], entry.listing())
```

The test fails. The new method, `FeedEntry.listing()`, is defined as follows:

```
def listing(self):
    return "%s: %s: %s" % (self.date, self.feed_title, self.title)
```

The test passes, so the example is one step closer to completion.

Test: Defining feeds_from_urls

At this point, there are a few tests left. URLs must be converted into feeds, feed entries must be converted into listings, and all of the new machinery must be hooked into the `main()` method. At this point, we'll try to finish off the `AggregateFeed` by focusing on the conversion of URLs to feeds.

The test is `test_get_feeds_from_urls()`. URLs are converted to feeds via `feedparser.parse()`. This can be viewed as a factory method. The dependency is initialized in a manner analogous to `feed_factory()`.

```
def test_get_feeds_from_urls():
    """Should get a feed for every URL"""
    urls = [Mock(), Mock()]
    feeds = [Mock(), Mock()]
    subject = AggregateFeed()
    subject.feedparser = Mock()
    subject.feedparser.expects(once()).parse(same(urls[0])).will(
        return_value(feeds[0]))
    subject.feedparser.expects(once()).parse(same(urls[1])).will(
        return_value(feeds[1]))
    returned_feeds = subject.feeds_from_urls(urls)
    assert_equals(feeds, returned_feeds)
    subject.feedparser.verify()
```

The test fails. The definition fulfilling the test is as follows:

```
def feeds_from_urls(self, urls):
    """Get feeds from URLs"""
    return [self.feedparser.parse(url) for url in urls]
```

The test succeeds.

Test: AggregateFeed Initializes the FeedParser Factory

The method `feeds_from_urls()` depends on the `feedparser` property being initialized, so a test must ensure this:

```
def test_aggregate_feed_initializes_feed_parser():
    """Ensure AggregateFeed initializes dependency on feedparser"""
    assert_equals(feedparser AggregateFeed().feedparser)
```

The test fails. The initialization method is updated:

```
def __init__(self):
    self.entries = Set()
    self.feed_factory = FeedEntry
    self.feedparser = feedparser
```

The test now succeeds.

Test: Defining from_urls

At this point, you should be asking yourself the following question: where does the list of feeds get aggregated? Any time you have a question like this, it suggests that you need to write a new test to answer the question. This test should check that the method gets `feeds_from_urls()` and that it combines those feeds. The test is as follows:

```
def test_from_urls():
    """Should get feeds from URLS and combine them"""
    urls = Mock()
    aggregate_feed = Mock()
    feeds = Mock()
    aggregate_feed.expects(once()).feeds_from_urls(same(urls)).\
        will(return_value(feeds))
    aggregate_feed.expects(once()).combine_feeds(same(aggregate_feed),
        same(feeds))
    AggregateFeed().from_urls(aggregate_feed, urls)
    aggregate_feed.verify()
```

The test fails, so you write the new method:

```
def from_urls(self, feed_aggregator, urls):
    """Produce aggregated feeds from URLs"""
    feeds = feed_aggregator.feeds_from_urls(urls)
    feed_aggregator.combine_feeds(feed_aggregator, feeds)
```

The test succeeds.

Refactoring: Reimplementing from_urls

Is there any functionality still unimplemented in `AggregateFeed`? For the moment, it doesn't appear so. However, I'm not comfortable with the code as it stands—it seems overly complicated. The discomfort comes from the interactions between `from_urls()`, `feeds_from_urls()`, and `combine_feeds()`. The data flow exhibits a Y shape, as shown in Figure 7-6.

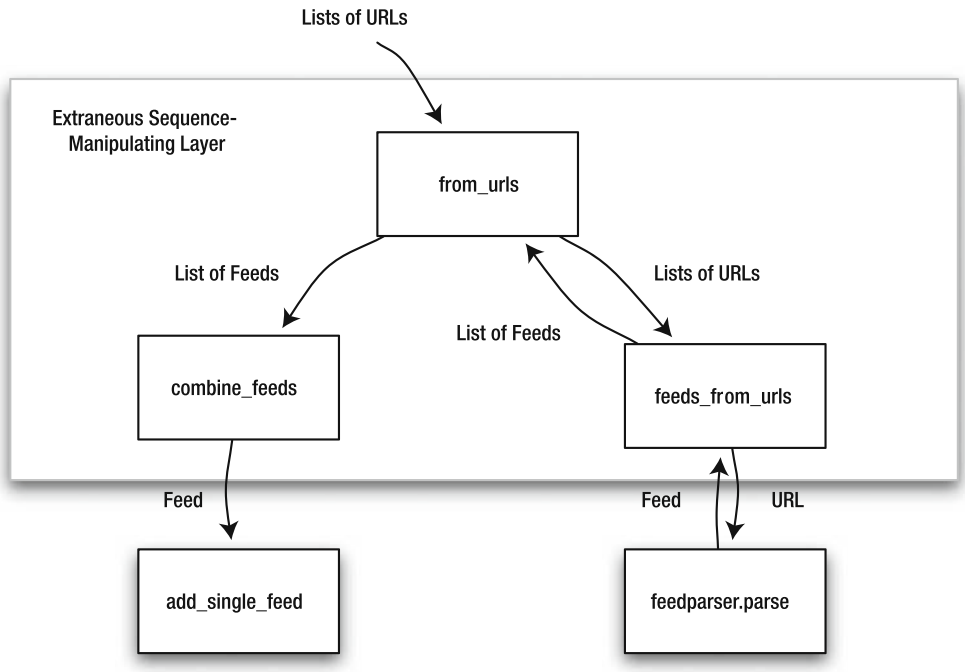

Figure 7-6. *Questionably complicated data flow*

A collection of URLs is passed down one leg, it is mapped to feeds, a collection of feeds is returned, and then the collection is fed down the other leg where another mapping is performed. It results in a layer of collection manipulations. This data flow pattern can often be transformed to a sequential set of mappings with only one iteration, as shown in Figure 7-7.

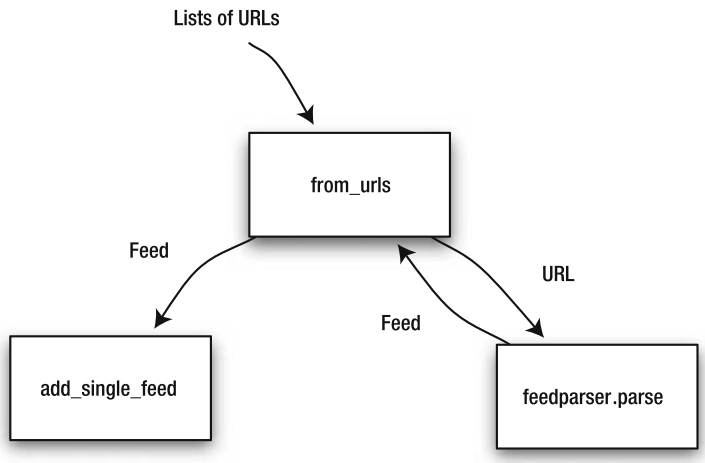

Figure 7-7. *Simplified data flow*

This rewritten test reflects the change in data flow:

```
def test_from_urls():
    """Should get feeds from URLs and combine them"""
    urls = [Mock(), Mock()]
    feeds = [Mock(), Mock()]
    subject = AggregateFeed()
    aggregate_feed = Mock()
    subject.feed_factory = Mock()
    #
    subject.feed_factory.expects(once()).parse(same(urls[0])).will(
        return_value(feeds[0]))
    aggregate_feed.expects(once()).add_single_feed(aggregate_feed, feeds[0])
    #
    subject.feed_factory.expects(once()).parse(same(urls[1])).will(
        return_value(feeds[1]))
    aggregate_feed.expects(once()).add_single_feed(aggregate_feed, feeds[1])
    #
    subject.from_url(aggregate_feed, urls)
    subject.feed_factory.verify()
    aggregate_feed.verify()
```

The test fails spectacularly. The new definition for `from_urls()` is as follows:

```
def from_urls(self, aggregate_feed, urls):
    """Produce aggregated feeds from URLs"""
    for x in urls:
        self.add_single_feed(aggregate_feed, self.feed_factory.parse(x))
```

The test passes. This definition of `from_url()` bypasses `feeds_from_urls()` and `combine_feeds()`. Those two methods and the corresponding tests can be removed. The excised code can always be retrieved from the source repository if it turns out to be needed later. (You have been checking in the code regularly, haven't you?)

One of the primary benefits of using mock objects is the style produced in the preceding example. In this style, mappings are composed and the composition action itself is tested. Without mock objects, writing for testability forces the code into a more expansive style where mappings are performed on sequences and the resulting sequences are examined. An additional layer then coordinates those sequencing operations.

Refactoring: Condensing Some Tests

What remains to be done? The tests `test_aggregate_feed_creates_factory()` and `test_aggregate_feed_initializes_feed_parser()` both verify that dependencies are initialized correctly, so they can be combined. The combined test is as follows:

```
def test_aggregate_feed_dependency_initialization():
    """Should correctly initialize dependencies"""
    assert_equals(FeedEntry, AggregateFeed().feed_factory)
    assert_equals(feedparser, AggregateFeed().feedparser)
```

The test passes. The other two tests are removed, all tests still pass, and AggregateFeed is complete. The next set of tests examine printing.

Test: Formatting Feed Entry Listings

The printing tests should verify that individual feed listings are combined correctly, and that an unsorted feed produces sorted listings. Most of the test data for these tests already exists, so it is reused. Checking sorting on one key only requires two data points, so only two data points are used. This data is not in sorted order. Printing is a distinct category of functionality, so the printing methods will be put in a separate class.

```
from rsreader.application import AggregateFeed, FeedEntry, FeedWriter, RSReader
...
def test_aggregate_feed_listing_should_be_sorted():
    """Should produce a sorted listing of feed entries."""
    unsorted = [FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[1]),
                FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[0])]
    aggregate_feed = AggregateFeed()
    aggregate_feed.entries = unsorted
    aggregate_listing = FeedWriter().entry_listings(aggregate_feed)
    assert_equals(xkcd_output, aggregate_listing)
```

The test fails with an error, which reports that FeedWriter doesn't exist.

```
class FeedWriter(object):
    """Prints an aggregate feed"""

    def entry_listings(self, aggregate_feed):
        """Produce a sorted listing of an aggregate feed"""
        return None
```

With FeedWriter and entry_listings() defined, the test fails with an equality mismatch. The result is now faked:

```
    def entry_listings(self, aggregate_feed):
        """Produce a sorted listing of an aggregate feed"""
        entries = aggregate_feed.entries
        return os.linesep.join([entries[1].listing(),
            entries[0].listing()])
```

The test passes. The question is now how to sort the entries by date. The feed entries contain the date as a printable string, so these can't be compared directly. Fortunately, FeedParser converts them into a comparable form.

The relevant field is date_parsed. You put this field into the test data, and then you modify the FeedEntry conversion routines. The definition of xkcd_items in shared_data.py becomes the following:

```
xkcd_items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
           'date_parsed': mktime(2007, 12, 5, 5, 0, 0, 2, None),
          'title': "Python"},
```

```
        {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
         'date_parsed': mktime(2007, 12, 3, 5, 0, 0, 2, None),
         'title': "Far Away"}]
```

The tests all pass. Now `test_feed_entry_from_parsed_feed()` is modified:

```
def test_feed_entry_from_parsed_feed():
    """Factory method to create a new feed entry from a parsed feed"""
    feed_entry = FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_items[0]['date'], feed_entry.date)
    assert_equals(xkcd_items[0]['date_parsed'], feed_entry.date_parsed)
    assert_equals(xkcd_items[0]['title'], feed_entry.title)
    assert_equals(xkcd_feed['feed']['title'], feed_entry.feed_title)
```

This test fails, so you rewrite `FeedEntry.from_parsed_feed()` as follows:

```
@classmethod
def from_parsed_feed(cls, feed, entry):
    """"Factory method producing a new feed entry from a feedparser entry"""
    feed_entry = FeedEntry()
    feed_entry.date = entry['date']
    feed_entry.date_parsed = entry['date_parsed']
    feed_entry.feed_title = feed['feed']['title']
    feed_entry.title = entry['title']
     return feed_entry
```

The test suite passes. You can now modify the method `entry_listings()` to support sorting:

```
def entry_listings(self, aggregate_feed):
    """Produce a sorted listing of an aggregate feed"""
    sorted_entries = sorted(aggregate_feed.entries,
                            key=lambda x: x.date_parsed,
                            reverse=True)
    return os.linesep.join([x.listing() for x in sorted_entries])
```

The test suite still passes. Now the printing behavior must be verified.

Test: Defining print_entry_listings

`print_entry_listings()` takes a listing from `FeedWriter` and prints it to `stdout`. `stdout` will be contained in an instance variable. The test needs to mock out both calls to `FeedWriter` to capture `stdout`, and it needs to mock out a call to `FeedWriter.entry_listings()`. `stdout` will be contained in an instance variable.

```
def test_print_entry_listings():
    """Verify that a listing was printed"""
    subject = FeedWriter()
    (feed_writer, aggregate_feed, listings) = (Mock(), Mock(), Mock())
    subject.stdout = Mock()
```

```
    feed_writer.expects(once()).entry_listings(same(aggregate_feed)).\
        will(return_value(listings))
    subject.stdout.expects(once()).write(same(listings))
    subject.stdout.expects(once()).write(eq(os.linesep))
    subject.print_entry_listings(feed_writer, aggregate_feed)
    feed_writer.verify()
    subject.stdout.verify()
```

The test fails. The printing code is implemented:

```
def print_entry_listings(self, feed_writer, aggregate_feed):
    """Print listing"""
    self.stdout.write(feed_writer.entry_listings(aggregate_feed))
    self.stdout.write(os.linesep)
```

The test succeeds. `FeedListing.stdout` needs to be initialized, and the next test ensures this.

Test: FeedWriter Initializes the stdout Attribute

The `stdout` attribute must be initialized when the `FeedWriter` creates itself.

```
def test_feed_writer_intializes_stdout():
    """Ensure that feed writer initializes stdout from sys.stdout"""
    assert_equals(sys.stdout, FeedWriter().stdout)
```

The test fails. The initialization code is written as follows:

```
class FeedWriter(object):
    """Prints an aggregate feed"""

    def __init__(self):
        self.stdout = sys.stdout
```

The test succeeds. The final printing test ensures that nothing is printed if there are no entries.

Test: Empty AggregateFeeds Generate No Output

When the feeds have no items, then the program should produce no output. This test ensures that.

```
def test_feed_writer_prints_nothing_with_an_empty_feed(self):
    """Empty aggregate feed should print nothing"""
    subject = FeedWriter()
    (feed_writer, aggregate_feed) = (Mock(), Mock())
    subject.stdout = Mock()
    aggregate_feed.expects(once()).is_empty().will(return_value(True))
    subject.print_entry_listing(feed_writer, aggregate_feed)
    aggregate_feed.verify()
    subject.stdout.verify()
```

The test fails. The method is updated:

```
def print_entry_listings(self, feed_writer, aggregate_feed):
    """Print listing"""
    if not aggregate_feed.is_empty():
        self.stdout.write(feed_writer.entry_listings(aggregate_feed))
        self.stdout.write(os.linesep)
```

The test succeeds, but `test_print_entry_listing()` fails because it doesn't mock out the call to `aggregate_feed.is_empty()`. The broken test now reads as follows:

```
def test_print_entry_listing():
    """Verify that a listing was printed"""
    subject = FeedWriter()
    (feed_writer, aggregate_feed, listings) = (Mock(), Mock(), Mock())
    subject.stdout = Mock()
    aggregate_feed.expects(once()).is_empty().will(
        return_value(False))
    feed_writer.expects(once()).entry_listings(same(aggregate_feed)).\
        will(return_value(listings))
    subject.stdout.expects(once()).write(same(listings))
    subject.stdout.expects(once()).write(eq(os.linesep))
    subject.print_listings(feed_writer, aggregate_feed)
    feed_writer.verify()
    subject.stdout.verify()
```

The test now succeeds. The new method `AggregateFeed.is_empty()` must be tested.

Test: Defining is_empty

The preceding test used the `is_empty()` method. This is a good time to create it.

```
def test_is_empty():
    """Unsure is empty works"""
    aggregate_feed = AggregateFeed()
    assert aggregate_feed.is_empty()
    aggregate_feed.add("foo")
    assert not aggregate_feed.is_empty()
```

The test fails, so you define `is_empty()` as follows:

```
def is_empty(self):
    """True if set is empty, and False otherwise"""
    return not self.entries
```

The test now succeeds, and printing is complete.

Test: Defining new_main

The `new_main()` method must be tied into the existing program. It should get `AggregateFeed` objects from a set of URLs, and then it should print them.

When argv is passed into the method, it contains extra information that must be removed. The first argument is the program name, and the subsequent arguments are the URLs to be read. This test ensures that the unneeded leading argument is stripped off.

```
def test_new_main():
    """Main should create a feed and print results"""
    args = ["unused_program_name", "x1"]
    reader = RSReader()
    reader.aggregate_feed = Mock()
    reader.feed_writer = Mock()
    reader.aggregate_feed.expects(once()).from_urls(same(reader.aggregate_feed),
         eq(["x1"]))
    reader.feed_writer.expects(once()).print_listings(same(reader.aggregate_feed))
    reader.new_main(args)
    reader.aggregate_feed.verify()
    reader.feed_writer.verify()
```

The test fails. The new_main() function is as follows:

```
def new_main(self, argv):
    self.aggregate_feed.from_urls(self.aggregate_feed, argv[1:])
    self.feed_writer.print_listings(self.aggregate_feed)
```

The test succeeds. Now you have to check that aggregate_feed and feed_writer have been initialized.

Test: The Application Initializes Dependencies

The application needs to initialize its dependencies, and this test ensures that it does so:

```
def test_rsreader_initializes_dependencies():
    """RSReader should initialize dependencies"""
    reader = RSReader()
    assert isinstance(reader.aggregate_feed, AggregateFeed)
    assert isinstance(reader.feed_writer, FeedWriter)
```

The test fails. The __init__ method is implemented as follows:

```
class RSReader(object):
    """The Application"""

    def __init__(self):
        self.aggregate_feed = AggregateFeed()
```

And the test gets a bit further. The method is expanded:

```
    def __init__(self):
        self.aggregate_feed = AggregateFeed()
        self.feed_writer = FeedWriter()
```

And the test succeeds. At this point, the new application code is complete, but the script is still executing the old main() method.

Refactoring: Making new_main the New main[2]

You're now ready to make the changes active by replacing `main()` with `new_main()`. This happens with the tests, too—you rename the `test_new_main()` method to `test_main()`:

```
def test_main():
    """Main should create a feed and print results"""
    args = ["unused_program_name", "x1"]
    reader = RSReader()
    reader.aggregate_feed = Mock()
    reader.feed_writer = Mock()
    reader.aggregate_feed.expects(once()).from_urls(same(reader.aggregate_feed),
       eq(["x1"]))
    reader.feed_writer.expects(once()).print_listings(same(reader.aggregate_feed))
    reader.main(args)
    reader.aggregate_feed.verify()
    reader.feed_writer.verify()
```

The test fails. You fix this by renaming `new_main()` to `main()`:

```
def main(self, argv):
    self.aggregate_feed.from_urls(self.aggregate_feed, argv[1:])
    self.feed_writer.print_listings(self.aggregate_feed)
```

The new tests all succeed, as do the acceptance tests. The old tests fail, but that's OK since you're about to remove them. You should take note of the failing tests, and then remove the extraneous code that these methods test. If more tests fail, then you may have removed too much. If the acceptance tests still pass, then you should remove the failing tests for old functionality. At this point, the program is complete.

A Simple PyMock Example

A simple test shows how PyMock is used. The example will calculate a triangle's perimeter:

```
def test_perimeter():
    assert_equals(4, perimeter(triangle))
```

PyMock must be initialized before you can use it in a function or method. This is done with the `use_pymock` decorator:

```
@use_pymock
def test_perimeter():
    assert_equals(4, perimeter(triangle))
```

Here, PyMock is used to imitate `triangle`:

2. War is the new peace. Politics is the new gossip. Pink is the new black. Whales are the new sushi.

```
@use_pymock
def test_perimeter():
    triangle = mock()
    assert_equals(4, perimeter(triangle))
```

PyMock uses a record-replay mechanism. Expectations are set either by performing calls exactly as they are expected to be replayed, or by describing them with a DSL similar to pMock. I'll demonstrate the former here:

```
@use_pymock
def test_perimeter():
    triangle = mock()
    triangle.side[0]; returns(1); once()
    triangle.side[1]; returns(3); once()
    assert_equals(4, perimeter(triangle))
```

The direct recording makes it easy to record multistep sequences. The actions here have two steps: the property `side` is retrieved, and then `side.__getitem__()` is called.[3] PyMock uses `__eq__()` to compare most arguments, with mocks being the sole exception; they are compared by identity. The return values and counts are specified using additional functions.

After expectations have been set, the mock is switched from record mode to replay mode:

```
@use_pymock
def test_perimeter():
    triangle = mock()
    triangle.side[0]; returns(1); once()
    triangle.side[1]; returns(3); once()
    replay()
    assert_equals(4, perimeter(triangle))
```

Here's the simplest method fulfilling the test:

```
def perimeter(triangle):
    return 4
```

The test passes, but it shouldn't. You want to verify that the `triangle` object was used, and you do this by calling `verify()`. This function checks all recorded expectations. This contrasts with pMock, in which the `verify()` method must be called for each mock that you want to check.

```
def test_perimeter():
    triangle = Mock()
    triangle.side[0]; returns(1); once()
    triangle.side[1]; returns(3); once()
    replay()
    assert_equals(4, perimeter(triangle))
    verify()
```

3. Every call to a mock returns a new mock unless you specify something else using `returns()` or `raises()`.

Now the test fails. The following definition satisfies the test:

```
def perimeter(triangle):
   return triangle.side[0] + triangle.side[1]
```

It is worth noting that the call count is optional. If it is not specified, then `once()` is assumed, so the following code is equivalent to the preceding test:

```
def test_perimeter():
    triangle = Mock()
    triangle.side[0]; returns(1)
    triangle.side[1]; returns(3)
    replay()
    assert_equals(4, perimeter(triangle))
    verify()
```

Monkeypatching

PyMock directly supports monkeypatching existing classes, attributes, and properties. This is one of the primary distinctions between PyMock and pMock. This allows your code to temporarily override an existing package or attribute, without having to explicitly inject dependencies. Monkeypatching is done with the `override()` function, as shown here:

```
@use_pymock
def test_dirpaths(self):
    root = 'path'
    listed_directories = ['one', 'two']
    expected_paths = [os.path.join(root, 'one'), os.path.join(root, 'two')]
    override(os, 'listdir'); os.listdir(root); returns(listed_directories)
    replay()
    assert_equals(expected_paths, dirpaths(root))
    verify()
```

The definition fulfilling the test is

```
def dirpatch(root):
    return [os.path.join(root, x) for x in os.listdir(root)]
```

Saying the Same Thing Differently

PyMock supports a second syntax to define expectations. It is closer to pMock's, and it suffers similar limitations. It is restricted to simple function calls, but it is more concise when monkeypatching. The previous test becomes the following:

```
@use_pymock
def test_dirpaths(self):
    root = 'path'
    listed_directories = ['one', 'two']
    expected_paths = [os.path.join(root, 'one'), os.path.join(root, 'two')]
```

```
    override(os, 'listdir').expects(root).returns(listed_directories)
    replay()
    assert_equals(expected_paths, dirpaths(root))
    verify()
```

The `override` expression now completely specifies the expected call. The arguments are specified with `expects()`, and the return values are specified with `returns()`.

Calls on mock objects are specified with the `method()` function:

```
def test_perimeter():
    triangle = Mock()
    method(triangle, 'side').expects(0).returns(1)
    method(triangle, 'side').expects(1).returns(3)
    replay()
    assert_equals(4, perimeter(triangle))
    verify()
```

This test specifies this function:

```
def perimeter(triangle):
   return triangle.side(0) + triangle.side(1)
```

Implementing with PyMock

The implementation with PyMock largely mirrors the pMock implementation, but there are significant differences. Monkeypatching eliminates the necessity of introducing a second variable to hold the mock. Instead, the mocked method is attached directly to an instance using the `override()` function. This leads to a simpler call structure.

Monkeypatching also allows tests to mock external modules in place. In cases where the dependencies are never expected to change, this leads to a simpler application. Modules are referenced directly rather than being referenced through variables.[4]

The tests will start with `from_urls()`. This method wasn't in the pending tests originally brainstormed. It was discovered along the way, but it is a good place to start. It obviates the need for both `get_feeds_from_urls()` and `combine_feeds()`, so those pending tests are discarded. `test_feed_entry_listing()` duplicates `test_feed_listing_is_sorted()`, so that will be ignored, too. There is nothing to be gained from repeating material with no new insight, so the list of pending tests is now as follows:

```
def _pending_test_new_main():
    """Hooking in new code to main"""

def _pending_test_from_urls():
    """Should retrieve feeds and add them to the aggregate"""
```

4. Some might say, "Rather than being injected as dependencies," but we're not dealing with Java here.

```
def _pending_test_get_feeds_from_urls():
    """Should get a feed for every URL"""

def _pending_test_combine_feeds():
    """Should combine feeds into a list of FeedEntries"""

def _pending_test_add_single_feed():
    """Should add a single feed to a set of feeds"""

def _pending_test_create_entry():
    """Create a feed item from a feed and a feed entry"""

def _pending_test_feed_listing_is_sorted():
    """Should sort the aggregate feed listing"""

def _pending_test_feed_entry_listing():
    """Should produce a correctly formatted listing from a feed entry"""
```

Test: from_urls and Mocking External Modules

From the very beginning, you'll make strong use of `override()`:

```
import feedparser
...
@use_pymock
def test_from_urls():
    """Should retrieve feeds and add them to the aggregate"""
    urls = [dummy(), dummy()]
    feeds = [dummy(), dummy()]
    subject = RSReader()
    #
    override(feedparser, 'parse').expects(urls[0]).returns(feeds[0])
    override(subject, 'add_single_feed').expects(feeds[0])
    #
    override(feedparser, 'parse').expects(urls[1]).returns(feeds[1])
    override(subject, 'add_single_feed').expects(feeds[1])
    #
    replay()
    subject.from_urls(urls)
    verify()0
```

The `dummy()` calls create dummy objects. They're impostors with no functionality that exist only to be used as arguments. In actuality, the objects returned from `dummy()` are full-fledged mock objects—just the same as those returned from `mock()`—but the factory method makes the intention clear to the test's readers.

The test fails to execute, but defining `from_urls()` is insufficient. The `override()` function verifies the existence of `add_single_feed()`, so it must be defined before the test can run:

```
    def from_urls(self, feeds):
        """Combine a set of parsed feeds"""

    def add_single_feed(self, feed):
        """Add a single parsed feed"""
```

The test now raises a verification failure. The full definition satisfying the test is as follows:

```
    def  from_urls(self, feeds):
        """Combine a set of parsed feeds"""
        for f in feeds:
            self.add_single_feed(f)

    def add_single_feed(self, feed):
        """Add a single parsed feed"""
```

The tests runs, and it succeeds.

Test: Defining add_single_feed

The next test characterizes `add_single_feed()`.The method `add_single_feed()` should call `create_entry()` once for each entry in the feed the caller passes in.

```
@use_pymock
def test_add_single_feed():
    """Should create a new entry for each entry in the feed"""
    reader = RSReader()
    entries = [dummy(), dummy()]
    feed = mock()
    feed.entries; returns(entries)
    override(reader, 'create_entry').expects(entries[0])
    override(reader, 'create_entry').expects(entries[1])
    replay()
    reader.add_single_feed(feed)
    verify()
```

The test fails to execute because `create_entry()` hasn't been defined yet. Here is a minimal definition:

```
    def add_single_feed(self, feed):
        """Add a single parsed feed"""

    def create_entry(self, feed, entry):
        """Add a single entry"""
```

The test now fails with a verification exception. The method is defined as follows:

```
    def add_single_feed(self, feed):
        """Add a single parsed feed"""
        for x in urls:
            self.add_single_feed(feedparser.parse(x))
```

```
    def create_entry(self, feed, entry):
        """Add a single entry"""
```

The test now passes.

Refactoring: Moving Methods to a New Object

At this point in the pMock example, it was clear that these methods belonged in their own class. The explicit dependency that had to be passed in like a second `self` is missing. Monkeypatching bypasses the need to inject a dependency, but it was this very dependency that made it clear where these methods belonged.[5] The trade-off is that the resulting code looks more Pythonic and less alien.

Refactoring: Moving add_single_feed

Moving the methods to another class highlights another difference involved with gleeful monkeypatching. Methods have to be moved in twos. One method is moved, and the monkeypatched one is temporarily duplicated. I prefer to start with the last implemented method. The test becomes the following:

```
from rsreader.app import AggregateFeed, RSReader
...
def test_add_single_feed():
    """Should create a new entry for each entry in the feed"""
    subject = AggregateFeed()
    entries = [dummy(), dummy()]
    feed = mock()
    feed.entries; returns(entries)
    override(subject, 'create_entry').expects(feed, entries[0])
    override(subject, 'create_entry').expects(feed, entries[1])
    replay()
    subject.add_single_feed(feed)
```

The test fails because `AggregateFeed` hasn't been created. The class is created, and then the test fails because the methods haven't been defined. At this point, I'll move over the methods `add_single_feed()` and `create_entry()`. The first is a complete method, while the second is a stub. One of the implicit goals in TDD is never breaking more than one test at a time. If other tests depended on `RSReader.create_entry()`, then I would leave a copy behind, and I would only remove it after altering those tests.

```
class RSReader(object):
    """The Application"""

    def from_urls(self, urls):
        """Transform URLS into feeds"""
```

5. You could restrict your usage of `override()` to standard library modules such as `os` or `sys`, but that doesn't seem to work in practice—it's too tempting a tool.

```
        for x in urls:
            self.add_single_feed(feedparser.parse(x))

    def add_single_feed(self, feed):
        """Add a single parsed feed"""


class AggregateFeed(object):
    """Several parsed feeds combined"""

    def add_single_feed(self, feed):
        """Add a single parsed feed"""
        for e in feed.entries:
            self.create_entry(feed, e)

    def create_entry(self, feed, entry):
        """Add a single entry"""
```

The test succeeds.

Refactoring: Moving from_urls()

The method `from_urls()` belongs in the class `AggregateFeed`, so you modify `test_from_urls()` to expect this change:

```
@use_pymock
def test_from_urls():
    """Should retrieve feeds and add them to the aggregate"""
    urls = [dummy(), dummy()]
    feeds = [dummy(), dummy()]
    subject = AggregateFeed()
    #
    override(feedparser, 'parse').expects(urls[0]).returns(feeds[0])
    override(subject, 'add_single_feed').expects(feeds[0])
    #
    override(feedparser, 'parse').expects(urls[1]).returns(feeds[1])
    override(subject, 'add_single_feed').expects(feeds[1])
    #
    replay()
    subject.from_urls(urls)
    verify()
```

The test fails, so you move `from_url()` from RSReader to `AggregateFeed`. There are no more tests depending on the stub method `add_single_feed()`, so you remove it:

```
class RSReader(object):
    """The Application"""
```

```
class AggregateFeed(object):
    """Several parsed feeds combined"""

    def from_urls(self, urls):
        """Transform URLS into feeds"""
        for x in urls:
            self.add_single_feed(feedparser.parse(x))

    def add_single_feed(self, feed):
        """Add a single parsed feed"""
        for e in feed.entries:
            self.create_entry(feed, e)

    def create_entry(self, feed, entry):
        """Add a single entry"""
```

The test succeeds.

Test: create_entry() and Mocking Class Constructors

With pMock, it was necessary to use a factory to introduce the relationship between `AggregateFeed.create_entry()` and the `FeedEntry` objects it produces. With PyMock, the constructor for `FeedEntry` is mocked out directly.

I can argue that using a factory makes for a better design by explicitly capturing the dependency. I can also argue that it is overkill for this application. When the class is referenced in more than one place, or when it comes time to introduce a second kind of element, then a factory may be the better choice.

```
import rsreader.application
...
@use_pymock
def test_create_entry():
    """Should create an entry and add it to the collection"""
    subject = AggregateFeed()
    (feed, entry) = (dummy(), dummy())
    new_entry = dummy()
    override(rsreader.application, 'FeedEntry')\
        .expects(feed, entry)\
        .returns(new_entry)
    override(subject, 'add').expects(new_entry)
    replay()
    subject.create_entry(feed, entry)
    verify()
```

The test fails to run because `FeedEntry` is not defined. The definition is as follows:

```
class FeedEntry(object):
    """Combines elements of a feed and a feed entry.
    Allows multiple feeds to be aggregated without losing feed specific
    information."""
```

The test progresses a bit further before failing with an error. It now complains that `add()` isn't defined, so you stub it out:

```
    def create_entry(self, feed, entry):
        """Add a single entry"""

    def add(self, entry):
        """Add an entry"""
```

The test now fails with a verification error. You complete `create_entry()`:

```
    def create_entry(self, feed, entry):
        """Add a single entry"""
        self.add(FeedEntry(feed, entry))

    def add(self, entry):
        """Add an entry"""
```

With this definition, the test succeeds.

Tests: Defining add and AggregateFeed.__init__

These two tests are almost exactly the same as with pMock, with only the slightest differences between the `add()` implementations. I'll just summarize them here:

```
from sets import Set

from nose.tools import *
from pymock import mock, override, replay, returns, verify, use_pymock

from rsreader.application import AggregateFeed, FeedEntry, RSReader

...

    @use_pymock
    def test_add():
        """Add an a feed entry to the aggregate"""
        entry = mock()
        subject = AggregateFeed()
        subject.add(entry)
        assert_equals(Set([entry]), subject.entries)

    def test_entries_is_always_defined():
        """The entries set should always be defined"""
        assert_equals(Set(), AggregateFeed().entries)
```

The code satisfying these tests is as follows:

```
from sets import Set
...

    def __init__(self):
        self.entries = Set()
...

    def add(self, entry):
        """Add to the set of entries"""
        self.entries.add(entry)
```

With these definitions, the tests once again successfully run to completion.

Test: Defining FeedEntry.__init__

`FeedEntry` construction has been mocked out, but the `FeedEntry` constructor doesn't exist yet. With pMock, there is no way to mock `__init__()`, so the constructor was a class method. With PyMock, you can mock `__init__()` directly:

```
def test_feed_entry_constructor():
    """Verify settings extracted from feed and entry"""
    subject = FeedEntry(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_items[0]['date'], subject.date)
    assert_equals(xkcd_items[0]['title'], subject.title)
    assert_equals(xkcd_feed['feed']['title'], subject.feed_title)
```

The test fails, so you redefine `__init__` as follows:

```
    def __init__(self, feed, entry):
        self.date = entry['date']
        self.feed_title = feed['feed']['title']
        self.title = entry['title']
```

The test now succeeds.

Test: Defining listing

The next test ensures that feed entry listings are correctly formatted:

```
def test_feed_entry_listing():
    """Should produce a correctly formatted listing form a feed item"""
    subject = FeedEntry(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_listings[0], subject.listing())
```

The test fails. The code producing the correct listing is as follows:

```
    def listing(self):
        return "%s: %s: %s" % (self.date, self.feed_title, self.title)
```

The test succeeds.

Test: entry_listings Should Be Sorted

The next set of tests implement printing. The first test, and the ripple of changes resulting from it, are nearly the same as in the pMock example. Those tests are as follows:

```
from rsreader.application import AggregateFeed, FeedEntry, FeedWriter, RSReader

...

def test_feed_entry_constructor():
    """Verify settings extracted from feed and entry"""
    subject = FeedEntry(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_items[0]['date'], subject.date)
    assert_equals(xkcd_items[0]['date_parsed'], subject.date_parsed)
    assert_equals(xkcd_items[0]['title'], subject.title)
    assert_equals(xkcd_feed['feed']['title'], subject.feed_title)
...

def test_aggregate_feed_listing_should_be_sorted():
    """Should produce a sorted listing of feed entries."""
    unsorted_entries = [FeedEntry(xkcd_feed, xkcd_items[1]),
                        FeedEntry(xkcd_feed, xkcd_items[0])]
    aggregate_feed = AggregateFeed()
    aggregate_feed.entries = unsorted_entries
    assert_equals(xkcd_output, FeedWriter().entry_listings(aggregate_feed))
```

The following change is made to the test data:

```
xkcd_items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
           'date_parsed': mktime(2007, 12, 5, 5, 0, 0, 2, None),
          'title': "Python"},
         {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
          'date_parsed': mktime(2007, 12, 3, 5, 0, 0, 2, None),
          'title': "Far Away"}]
```

The changes and additions, made hand-in-hand with the preceding tests, are the following:

```
class FeedEntry(object):
    """Combines elements of a feed and a feed entry.
    Allows multiple feeds to be aggregated without losing feed specific
    information."""

    def __init__(self, feed, entry):
        self.date = entry['date']
        self.date_parsed = entry['date_parsed']
        self.feed_title = feed['feed']['title']
        self.title = entry['title']
```

```
    def listing(self):
        return "%s: %s: %s" % (self.date, self.feed_title, self.title)


class FeedWriter(object):
    """Prints an aggregate feed"""

    def entry_listings(self, aggregate_feed):
        """Produce a sorted listing of an aggregate feed"""
        sorted_entries = sorted(aggregate_feed.entries,
                                key=lambda x: x.date_parsed,
                                reverse=True)
        return os.linesep.join([x.listing() for x in sorted_entries])
```

Once these changes have been made, the tests succeed.

Test: Defining print_entry_listings

With pMock, printing was performed through a local copy of `sys.stdout`, which was held in `FeedWriter.stdout`. This instance variable was replaced with a mock. PyMock can easily mock the method in situ.

```
def test_print_agg_feed_listing_is_printed():
    """Should print listing of feed entries"""
    unsorted_entries = [FeedEntry(xkcd_feed, xkcd_items[1]),
                        FeedEntry(xkcd_feed, xkcd_items[0])]
    aggregate_feed = AggregateFeed()
    aggregate_feed.entries = unsorted_entries
    override(sys.stdout, 'write').expects(xkcd_output + os.linesep)
    replay()
    FeedWriter().print_entry_listings(aggregate_feed)
    verify()
```

The test fails. It reports `This object can't be modified`. There are some objects that Python can't monkeypatch. As a result, the object itself has to be mocked:

```
@use_pymock
def test_print_entry_listing():
    """Should print listing of feed entries"""
    unsorted_entries = [FeedEntry(xkcd_feed, xkcd_items[1]),
                        FeedEntry(xkcd_feed, xkcd_items[0])]
    aggregate_feed = AggregateFeed()
    aggregate_feed.entries = unsorted_entries
    override(sys, 'stdout')
    method(sys.stdout, 'write').expects(xkcd_output + os.linesep)
    replay()
    FeedWriter().print_entry_listings(aggregate_feed)
    verify()
```

The test fails because the method isn't defined yet. The method declaration is as follows:

```
def print_entry_listings(self, aggregate_feed):
    """Print an entry_listing to sys.stdout"""
```

The test now fails in the desired way. The full definition for the method is as follows:

```
def print_entry_listings(self, aggregate_feed):
    """Print an entry_listing to sys.stdout"""
    sys.stdout.write(self.entry_listings(aggregate_feed) + os.linesep)
```

The test succeeds, but the method isn't complete.

Test: print_entry_listings Should Do Nothing with Empty Feeds

As defined, the method will print `os.linesep` when the `AggregateFeed` is empty. It should print nothing. This test expresses that requirement:

```
def test_print_entry_listing_does_nothing_with_an_empty_aggregate():
    """Ensure that nothing is printed with an empty aggregate"""
    empty_aggregate_feed = AggregateFeed()
    override(sys, 'stdout')
    replay()
    FeedWriter().print_entry_listings(empty_aggregate_feed)
    verify()
```

This test makes use of negative assertions. No actions are defined on the mock in `sys.stdout`, ensuring that an error will arise if any are performed on it. As currently defined, `print_entry_listing()` always writes `os.linesep` to `sys.stdout`, so the test fails. The subject is redefined as follows:

```
def print_entry_listings(self, aggregate_feed):
    """Print an entry_listing to sys.stdout"""
    if not aggregate_feed.is_empty():
        entry_listings = self.entry_listings(aggregate_feed)
        sys.stdout.write(entry_listings + os.linesep)
```

Both `print_entry_listings()` tests now fail because the method `AggregateFeed.is_empty()` doesn't exist, so you define the method as follows:

```
class AggregateFeed(object):
    """Several parsed feeds combined"""

    def __init__(self):
        """Define factory"""
        self.entries = Set()

    def is_empty(self):
        """True if empty, False otherwise"""
        return not self.entries
```

The tests now pass, but there is a problem with them.

Test: is_empty and the Unproven Test

The `is_empty()` method is only tested indirectly. If it is ever broken, then the malfunction will show itself indirectly through the previous two tests. Failures should be exhibited directly, so it needs to be tested directly:

```
def test_is_empty():
    """Ensure that is_empty reports emptiness as expected"""
    empty_aggregate_feed = AggregateFeed()
    non_empty_aggregate_feed = AggregateFeed()
    non_empty_aggregate_feed.add("foo")
    assert empty_aggregate_feed.is_empty() is True
    assert non_empty_aggregate_feed.is_empty() is False
```

The test succeeds, which is a problem. It is unproved that this test fails. There are two approaches to fixing this. One is to go back several steps and mock out `is_empty()`, and slowly build up the tests. The other way is to break `is_empty()`, and verify that the test breaks. You do this by changing the return value to `None`, running the test, verifying that it failed, changing it to `True`, running the test, verifying that it failed again, and then putting back the real implementation.

This second approach is sometimes required when using automatic refactorings, particularly method or class extractions. The newly created methods and classes don't have any direct tests, so these tests must be created de novo. You'll often have to verify that the tests fail by breaking the newly refactored code and then restoring it.

Test: new_main, Hooking It All Together

The test for `new_main()` is precisely analogous to the pMock example. It verifies that the first argument is stripped off, and that the appropriate calls are made to the `AggregateFeed` and `FeedWriter` objects.

```
@use_pymock
def test_new_main():
    """Hook components together"""
    args = ["unused_program_name", "u1"]
    subject = RSReader()
    subject.aggregate_feed = mock()
    subject.feed_writer = mock()
    method(subject.aggregate_feed, 'from_urls').expects(["u1"])
    method(subject.feed_writer, 'print_entry_listings').\
         expects(subject.aggregate_feed)
    replay()
    subject.new_main(args)
    verify()
```

The test fails. The method satisfying the test is as follows:

```
def new_main(self, argv):
    """Read argument lists and coordinate aggregates"""
    self.aggregate_feed.from_urls(argv[1:])
    self.feed_writer.print_entry_listings(self.aggregate_feed)
```

The test succeeds.

Test: RSReader Initialization

One more test remains. The new `main()` method depends on the two aggregates. These dependencies must be initialized and verified. The test is as follows:

```
def test_rsreader_dependency_initialization():
    """Ensure that dependencies are correctly initialized"""
    assert isinstance(RSReader().aggregate_feed, AggregateFeed)
    assert isinstance(RSReader().feed_writer, FeedWriter)
```

The test fails. The method fulfilling the test is as follows:

```
class RSReader(object):
    """The Application"""

    def __init__(self):
        self.aggregate_feed = AggregateFeed()
        self.feed_writer = FeedWriter()
```

The test succeeds.

Finishing Up: Activating the New Functionality

The new `main()` function is complete. The entire new application has been wired together in parallel with the existing code. Now it is time activate it. The old `test_main()` is no longer needed. It is removed, and `test_new_main()` is renamed to `test_main()`. Here's the new test:

```
@use_pymock
def test_main():
    """Hook components together"""
    args = ["unused_program_name", "u1"]
    subject = RSReader()
    subject.aggregate_feed = mock()
    subject.feed_writer = mock()
    method(subject.aggregate_feed, 'from_urls').expects(["u1"])
    method(subject.feed_writer, 'print_entry_listings').\
        expects(subject.aggregate_feed)
    replay()
    subject.main(args)
    verify()
```

The test fails. The old `main()` method is removed and `new_main()` is renamed to `main()`:

```
def main(self, argv):
    """Read argument lists and coordinate aggregates"""
    self.aggregate_feed.from_urls(argv[1:])
    self.feed_writer.print_entry_listings(self.aggregate_feed)
```

With this, thePyMock application is complete.[6]

Other pMock and PyMock Features

pMock and PyMock have a number of features that haven't been covered here. Most notable are exception mocking and playback limits. PyMock also supports setter mocking and generator mocking.

Raising Exceptions with pMock

pMock uses the `raise_exception(exc)` function in the `will` clause to declare that the method raises an exception when played back:

```
def test_raising_exception():
    """Raise an exception"""
    m = Mock()
    m.expects(once()).whoops().will(raise_exception(Exception()))
    assert_raises(Exception, m.whoops)
    m.verify()
```

Raising Exceptions with PyMock

The PyMock equivalents are the `raises(exc)` method and function:

```
@use_pymock
def test_raising_exception():
    """Raise an exception"""
    m = mock()
    method(m, 'whoops').expects().raises(Exception())
    replay()
    assert_raises(Exception, m.whoops)
    verify()
```

Using the direct recording interface, this same test would be the following:

```
@use_pymock
def test_raising_exception():
    """Raise an exception"""
```

6. Go take a break. Get yourself a beer. Write someone a letter. Call your friends. You deserve it. You made it through that slog. I'm going to do the same in a few pages. I'll catch up with you.

```
    m = mock()
    m.whoops(); raises(Exception())
    replay()
    assert_raises(Exception, m.whoops)
    verify()
```

Playback Counts with pMock

pMock recognizes three different calling policies: `once()`, `at_least_once()`, and `never()`. All are used in the `expects` clause.

Playback Counts with PyMock

PyMock recognizes the following playback counts: `once()` (the default), `one_or_more()`, `zero_or_more()`, `set_count(int)`, and `at_least(int)`. Only the last two require explanation. The call `set_count()` specifies the precise number of playbacks expected, and the call `at_least()` specifies the minimum number of playbacks expected.

Mocking Attribute Setters with PyMock

PyMock mocks property getting and setting using both the raw recording style and a declarative style. Here are three different setter expressions:

```
@use_pymock
def test_setting_attributes():
    """Set attributes"""
    m = mock()
    m.f = 1
    m.f = 2; raises(Exception())
    set_attr(m, 'f', 3).once()
    replay()
    m.f = 1
    try:
        m.f = 2
    except Exception:
        pass
    m.f = 3
    verify()
```

Here are two getter expressions:

```
@use_pymock
def test_getting_attributes():
    """Get attributes"""
    m = mock()
    m.g; returns(1)
    get_attr(m, 'h').returns(2)
    replay()
```

```
    assert_equals(m.g == 1)
    assert_equals(m.h == 2)
    verify()
```

Mocking Generators with PyMock

With PyMock, generators are most clearly mocked with the declarative style:

```
@use_pymock
def testGeneratesWithRaisedTermination(self):
    m = mock()
    method(m, 'f').expects().generates(1, 2)
    replay()
    g = m.f()
    assert_equals(1, g.next())
    assert_equals(2, g.next())
    assert_raises(StopIteration, g.next)
```

You specify a terminal exception using the `ending` keyword:

```
@use_pymock
def testGeneratesWithRaisedTermination(self):
    m = mock()
    method(m, 'f').expects().generates(1, 2, ending=StopMe())
    replay()
    g = m.f()
    assert_equals(1, g.next())
    assert_equals(2, g.next())
    assert_raises(StopMe, g.next)
```

If you use the recording mode, the first example would be

```
generator(m.f(), [1, 2])
```

For the second example, the equivalent line would be

```
generator(m.f(), [1, 2], StopMe())
```

Using PyMock with unittest

PyMock defines a subclass of `unittest.TestCase` called `PyMockTestCase`. By subclassing it, all test methods are automatically configured to use PyMock. There is one caveat. `PyMockTestCase` uses `setUp()` and `tearDown()` to configure the mocking machinery and to restore monkey-patches. If you have defined your own `setUp()` or `tearDown()` methods, then they must use `super` to call the appropriate methods in the parent class.

Summary

I walked you through several TDD examples in this chapter, demonstrating how to use mock objects in excruciating detail. Throughout the chapter, example refactorings were shown and briefly discussed. Along the way, I briefly introduced the automatic refactoring tools available in Pydev.

The primary focus of the chapter was on code isolation through *impostors*. Impostors are test objects that replace application objects. Impostors, sometimes called *test doubles*, come in four different flavors. In order of increasing complication, they are *dummies*, *stubs*, *mocks*, and *fakes*.

Two mock object packages were introduced. They reflect the two different schools of thought. One focuses on restricting functionality to improve design, and the other focuses on testing ease. pMock reflects the first school of thought, and PyMock reflects the second.

pMock is modeled on Java's jMock library. It uses a *DSL* to declare expectations. PyMock is modeled on Java's EasyMock library, although it has acquired some of jMock's trappings. It uses a record-replay model. Calls are specified by performing them, and then the mock is switched into replay mode.

I compared the two packages by building the same example with each. The results were noticeably different, reflecting the capabilities and constraints of each package. This emphasizes that TDD and mock objects are design tools.

One technique introduced along the way was *monkeypatching*. Monkeypatches replace part of an existing object to alter that part's behavior. I demonstrated this technique using PyMock's `override()` function extensively in the latter half of the chapter.

Using TDD leads to higher test coverage, but it's easy to backslide. Programmers tend to be optimists (if you weren't, then you'd go insane), so they tend to overestimate test coverage. The next chapter examines tools for measuring test coverage, and how to deploy them in your development environment.

Consistent style is also important if multiple people are working on a body of code. The next chapter also examines how to enforce these stylistic guidelines through Subversion pre-commit triggers. In general, developers want high test coverage and consistent style, but without feedback, errors creep into the work of even the most diligent. Automation can provide much of this feedback.