unittest is Broken: Toward Composable, Shareable Test Framework Extensions

Collin Winter
Google, Inc
collinwinter@google.com

0. Abstract

In this paper I examine the shortcomings and core design flaws of JUnit and JUnit-derived test frameworks, such as Python's PyUnit (aka unittest), focusing specifically on the programmer's ability to extend the framework. I then discuss the requirements for an extensible testing framework and introduce `test_harness`, an alternative framework designed from the ground up to address these fundamental issues. Finally, examples drawn from real-world, PyUnit-based test suites are reformulated using `test_harness` to demonstrate the power of the new framework.

1. Introduction

When writing test suites, it is often desirable to mix in additional functionality in order to augment the tests themselves, e.g., the ability to mark certain tests as "expected to fail" or to indicate that a given test should skipped in the presence of a given condition. While many testing frameworks exist that allow the creation of such augmentations and extensions, the design of these frameworks greatly limits the ability of the test writer to combine extensions in order to create the ideal testing environment for the current project. This lowers the incentive to share such framework extensions with the wider development community, thus both inhibiting software reuse and increasing the barriers to properly testing certain kinds of code.

In this paper, I will explore the requirements for a testing framework that will allow extensions to be easily shared and combined, in the process contrasting these principles with those found in existing popular testing frameworks. In section 5, I introduce `test_harness`, an implementation in Python of the ideas set forth in earlier sections. Finally, I discuss further ideas and future directions for `test_harness`.

2. Terminology

For the purposes of this paper, the following terms and definitions will be used:

- A "test" is a piece of code that exercises some other code and verifies it performed as expected. In JUnit terms, a "test" is a single test method on a test class [HT03].
- A "test suite" is any collection of tests.
- A "test runner" is any mechanism that iterates over a test suite, running each individual test and collecting the results.
- "Combining" or "composing" framework extensions means any process by which multiple extensions are applied to the same test suite.

3. What Has Come Before

The dominant family of testing frameworks is Kent Beck's SUnit framework for Smalltalk [Beck99] and its many progeny: JUnit (for Java), PyUnit a.k.a. unittest (Python) [Pur03], RUnit (Ruby), Test::Unit (Ruby), CPPUnit (C++), HUnit (Haskell), SchemeUnit (Scheme) [WSG02], etc., of which JUnit is the most well-known and the basis for most of the other fooUnit frameworks. (I will use "xUnit" to refer to these systems collectively.) I will restrict my focus to SUnit, JUnit and their derivatives; HUnit and SchemeUnit are written in functional languages and so operate under a vastly different set of assumptions and restrictions than do JUnit and co.

While xUnit systems allow test authors to extend the framework, most such extensions tend to be exceedingly difficult to compose.

One area where this difficulty is prominent is in handling exceptions: in xUnit, one piece of code catches and categorized the exception (e.g., runtime error, test failure, etc.) and then uses another piece of code to record this categorization by calling, e.g., `addPass()`, `addFailure()`, `addError()`. In order to extend this system, both the code that catches the exception and the code that records the category must be kept in sync, and any effort to combine multiple extensions must be applied equally across both systems, in addition to whatever test runner mechanism is provided. Ruby's Test::Unit framework [Tal03] makes this somewhat easier using an observer pattern [GHJV95], though this is less than ideal since it does not allow one extension to prevent other extensions from mishandling a given exception; such control is necessary when implementing extensions that use exceptions to signal from one piece of code to another (see section 5.IV for examples).

Another sticking point in xUnit comes when implementing extensions that run a single test multiple times while only recording a single run (useful for shaking out nondeterministic tests). Because xUnit uses a single method, `run()`, to execute the test, test-specific set-up and tear-down code as well as to handle raised exceptions, writing an extension that desires to modify any part of the test life cycle is required to reimplement a significant portion -- if not all -- of the `run()` method of one of the provided `TestCase` classes. This leads to a system of extension composition that looks suspiciously like repetitive extension rewriting.

4. Toward Composable Extensions

Having enumerated some of the problems with existing test frameworks, let's look at the requirements for a framework that avoids these issues. The chief goal of such our new framework is to make it possible to share and easily combine extensions to the framework.

Extensions to our framework should be able to operate in concert with any number of other extensions as well as they can alone, that is, as the sole extension. In order to achieve this, it is essential that each extension need no information about its environment; each extension should be as self-contained and independent as possible, responsible only for its own functionality and nothing more. What a given extension doesn't handle, it should trust that some other part of the system will handle that.

In order to avoid the problems inherent in xUnit's single `run()` method, a new separation of concerns must be devised. Examining the life cycle of a test suite and its components, we see the following abstract structure (expressed in pseudo-Python):

```
for test in test_suite:
  prepare_for(test)
  try:
    run_single_test(test)
  except Exception as e:
    handle_exception(e)
  finally:
    finished_with(test)
```

This is effectively the cycle seen in xUnit, where the body of the loop is subsumed into a single function call. In order to maximize the flexibility and composability of framework extensions, these steps must be as explict as possible, so that an extension has to override only the portions it is interested in and no more.

These guidelines are fairly abstract and so encompass a number of more minor points. These will be discussed in more depth in the next section.

5. Introducing `test_harness`

`test_harness` is a Python-language implementation of the above guidelines with some elaboration. The current version is focused on running tests and reporting the results; some issues outside of this area have received only light treatment, in particular test discovery and test aggregation. These issues will be discussed in section 6.

A link to the `test_harness` source can be found in the References section [Win07].

5.I. High-level Overview

The `test_harness` framework consists of a single class, `TestRunner`, which is subclassed by extensions. This class is responsible for all facets of the test lifecycle outlined above: iterating over the test suite, setting up for a test, running the test, handling any exceptions, tearing down after the test, doing it all over again.

`TestRunner`'s primary purpose is to lay out an API for subclasses to follow; it implements only minimal functionality. In particular, it is responsible for defining the two default exception categories, "error" and "failure"; the former indicates a runtime exception, and the latter signals that an assertion did not hold. The minimal nature of the core framework means that most functionality that is traditionally considered "built-in" to other frameworks (e.g., terminal output of test results) is implemented here as extensions to `test_harness`.

`test_harness` extensions are composed using cooperative multiple inheritance. For example, defining an extension that logs test results to a console and to a database and supports TODO tests (see section 5.IV) is achieved like so:

```
from test_runner import ConsoleRunner
from test_runner import DBLogRunner
from test_runner import TodoRunner

class MyRunner(TodoRunner, DBLogRunner, ConsoleRunner):
  pass
```

Communication between the several extensions takes place through a chain of responsibility pattern [GHJV95], implemented via superclass calls. This is most clearly evident in the mechanism for classifying a raised exception: if an extension does not recognize the given exception, it is expected to hand off the exception object to the next class in the method resolution order [Sim02]. For a further example, see the `RefcountRunner` example in section 5.III.

5.II. Extension Isolation

One key aspect of making extensions as independent as possible is the exception handling facility. `test_harness` allows each exception-consuming exception to handle only those exceptions it is interested in, trusting that other parts of the system will take care of the rest.

Each extension defines a list of categories it will assign to exceptions. Each category is a string that will be used by output extensions when labeling exceptions or when summarizing the results of a test run.

Taking an extension for TODO tests as an example, the relevant sections of code look something like this:

```
class TodoRunner(test_harness.TestRunner):
 categories = ['todo pass', 'todo fail']

 def handle_exception(self, test, exc_info):
  exc_type = exc_info[0]

  if issubclass(exc_type, TodoPassedException):
   self.log_exception('todo pass', test, exc_info)
  elif issubclass(exc_type, TodoFailedException):
   self.log_exception('todo fail', test, exc_info)
  else:
   super(TodoRunner, self).handle_exception(test, exc_info)
```

If a given test raises an exception, the runner's `handle_exception()` method will be invoked with the test and the return value from Python's `sys.exc_info()` function [VRo06] as arguments. If the raised exception was an instance of either `TodoPassedException` or `TodoFailedException` (both specific to `TodoRunner`), the runner will use the `log_exception()` method to categorize the exception. If the exception is of some other type, the runner passes it on, in the expectation that some other runner will handle it. (The `TestRunner` base class handles all exceptions that reach it.)

5.III. The Test Life Cycle

The test cycle is similar to the one described in section 4, though with some minor modifications.

```
def run(self, test_suite):
 for test in test_suite:
  self.pre_test(test)
  try:
   self.run_test(test)
  except Exception as e:
   self.handle_exception(e)
  finally:
   self.post_test(test)

def run_test(self, test):
 try:
  test.set_up()
  test.run()
 finally:
  test.tear_down()
```

This shows the two relevant methods from the `TestRunner` class, `run()` and `run_test()`. Notice that we have separated test-specific set-up and tear-down code into their own steps in the sequence (the `set_up()` and `tear_down()` calls in `run_test()`), distinct from per-test set-up and tear-down that is specific to extensions (the `pre_test()` and `post_test()` calls in `run()`). This differentiation is important when writing extensions that, e.g., rerun individual tests multiple times.

One extension that takes advantage of this is `RefcountRunner`, which helps to detect reference count-related bugs in C-language Python extension modules by repeatedly running a single test. The relevant method of the runner looks something like this:

```
 repeats = 7

 def run_test(self, test):
  ref_counts = [0] * repeats
  up = super(RefcountRunner, self)

  for r in range(7):
   up.run_test(test)
   run_garbage_collection()
   ref_counts[r] = get_total_refcount()

  for r in range(1, self.repeats):
   if ref_counts[r] != ref_counts[0]:
    raise RefcountFailed(ref_counts)
```

Here, test repetition is achieved by looping over a superclass call, which is expected to handle the work of actually running the test. In xUnit, where no differentiation is made between test- and extension-specific set-up/tear-down, such a strategy would result in each test being reported numerous times. Separating these concerns leads to simpler, more comprehensible extensions.

5.IV. Comparison With PyUnit

5.IV.i. Conceptual Size

Compared to xUnit (in particular, PyUnit), there are fewer concepts and classes for an extension author to keep track of in `test_harness`: `test_harness` has only one class, `TestRunner`, whereas PyUnit has four: TestCase, TestRunner, TestSuite and TestResult. Their interaction is not especially complicated, but it does add to the complexity involved in construction PyUnit extensions.

5.IV.ii. Ease of Composition

This is the true test of the `test_harness` design.

In examining the relative ease of combining multiple extensions, two user-defined extensions will be considered: an extension to mark tests as "expected to fail" (hereafter, "TODO support"), and the `RefcountRunner` example from section 5.III.

All code samples referred to below can be found in [Win07].

5.IV.ii.a. TODO Support

The desired functionality is the ability to mark certain tests as "expected to fail": if the test fails, report it, but don't consider it a problem; on the other hand, if the test passes unexpectedly, alert the user.

Implementing this in terms of PyUnit requires 140 lines of Python, including five PyUnit classes and four support functions and classes. The same functionality, implemented as a `test_harness` extension, requires 61 lines of Python. The same four support functions and classes from the PyUnit implementation are used, and only a single `test_harness TestRunner` subclass had to be defined.

5.IV.ii.b. `RefcountRunner`

The desired functionality is the ability to run an individual test multiple times, keeping track of the Python interpreter's global reference count after every run. If the global reference count is not consistent across every run, an exception should be raised; ideally, this exception will be counted as a "refcount-failed" error, rather than a regular "assertion failure" exception.

Implementing this in terms of PyUnit requires 117 lines of Python, consisting of four PyUnit classes and one support class. The same functionality, implemented as a `test_harness` extension, requires 36 lines of Python. The same support class from the PyUnit implementation is used, and only a single `test_harness` `TestRunner` subclass had to be defined.

5.IV.ii.c. Composition

Combining the functionality of the two PyUnit extensions proved difficult, requiring a significant amount of new code to be written in order to achieve both sets of functionality. Of the 197 total lines of the combined PyUnit extension, over half (105 lines) was new code. The primary trouble spot was in catching and categorizing raised exception, an area necessitating almost a complete rewrite.

By contrast, composition of the two `test_harness` extensions looks like this:

```
from test_harness.runner.refcounting import RefcountRunner
from test_harness.runner.todo import TodoRunner

class MyRunner(RefcountRunner, TodoRunner):
  pass
```

These four lines achieve the same functionality and level of composition that took 197 lines and considerable frustration with PyUnit.

5.V. Comparison With Decorators

Several reviewers of early drafts of this work have pointed out that the basic functionality of several examples cited above, specifically TODO tests and `RefcountRunner`, can be trivially implemented as Python decorators. While this is certainly true, a framework-based implementation offers several advantages over decorators:

1. Decorators must be manually added to each test they should be applied to, whereas `test_harness` extensions can be swapped in and out behind the tests (as it were), with only minimal changes to the test suite as a whole. In contrast, removing, e.g., decorator-based TODO support from a test suite would require deleting every single occurrence of the decorator across the entire suite, whereas removing that support from `test_harness`-based testing infrastructure would require deleting a single import and removing a base class from a class definition.
2. Decorator-based implementations do not interact well with code that reports the results of a testing run. These decorators must work within the exception categories established by the reporter code, whereas a proper `test_harness` extension is able to define custom exception categories that will be picked up by reporter extensions.

6. Future Direction

An alpha release of `test_harness` has already been made available to the public [Win07], and the response has generally been positive. Several users have submitted feedback based on their experience using `test_harness` in their own projects, and a follow-up release is planned that will incorporate their comments.

One high-priority item for future releases of `test_harness` is some measure of compatibility with PyUnit. This would most likely take the form of a `TestCase`-alike class that would provide the expected assertion methods. Because of the number of components in PyUnit (`TestRunner`, `TestCase`, `TestResult`, `TestSuite`, etc), full compatibility is prohibitively difficult, though the minimal solution should be sufficiently comprehensive to enable relatively easy migration from PyUnit to `test_harness`.

Another important facet that has thus far been omitted from `test_harness` is a comprehensive test discovery strategy. Test discovery is an orthogonal process to test execution and so has fallen outside the purview of my active interest. There are several Python-language test discovery systems such as nose [XXX] and py.test [XXX], and these will be consulted for any future work on test discovery in `test_harness`.

The ideal endpoint would be to have `test_harness` replace PyUnit in the Python standard library. PyUnit compatibility is obviously a prerequisite for this. Several core Python developers have communicated that they are interested in seeing `test_harness` folded into the standard library, and it is my hope that a gentle PR campaign -- of which this paper is a part -- will win over the general Python community.

7. Conclusion

`test_harness` offers considerable advantages to test engineers when compared to xUnit frameworks. The ability to trivially composed any number of extensions encourages authors to make their extensions available to a wider developer community, creating a larger pool of extensions to choose from when it comes time to create just the right testing environment for a given project.

While advanced framework users will receive a significant benefit from using `test_harness` over, e.g., PyUnit, the impact on more basic test environments is probably insufficient to motivate a migration. It is hoped that future work, such a PyUnit compatibility layer and inclusion in the Python standard library will make it possible to bring `test_harness`'s advantages to a wider audience.

8. References

[Beck99]. Kent Beck. *Kent Beck's Guide to Better Smalltalk* chapter 21. SIGS Reference Library. Cambridge University Press, 1999. http://www.xprogramming.com/testfram.htm

[GHJV95] - Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HT03] - Andrew Hunt and David Thomas. *Pragmatic Unit Testing In Java with JUnit*. The Pragmatic Programmers, 2003.

[Pur03] - Steve Purcell. PyUnit. In the Python standard library, version 2.5. http://www.python.org

[Sim02] - Michele Simionato. The Python 2.3 Method Resolution Order. http://www.python.org/download/releases/2.3/mro/

[Tal03] - Nathaniel Talbott. Test::Unit - Ruby Unit Testing Framework. In the Ruby standard library, version 1.8.6. http://www.ruby-lang.org/

[VRo06] - Guido van Rossum. sys -- System-specific Parameters and Functions. http://docs.python.org/lib/module-sys.html

[Win07] - Collin Winter. test_harness -- Next-gen Testing Infrastructure for Python. http://oakwinter.com/code/test_harness/

[WSG02] - Noel Welsh, Francisco Solsona and Ian Glover. SchemeUnit and SchemeQL: Two Little Languages. In *Third Workshop on Scheme and Functional Programming*, Pittsburg, Pennsylvania, USA, October 2002.