CHAPTER 11

Functional Testing

Functional testing is about building the right code. It is as important as unit testing, but it gets far less press. It breaks down into the three rough categories of *acceptance testing*, *integration testing*, and *performance testing*. I won't examine performance testing at all, and I'll only discuss integration testing in passing. This chapter's real meat is acceptance testing using PyFit, a functional and integration testing tool. So what is integration testing, and how do integration tests differ from acceptance tests?

Integration testing determines if large chunks of the application fit together correctly. It's like fitting together a few pieces of a broken mug before you try to glue the entire thing together. If you can't fit together the big chunks, then you know you can't reassemble it all. These sorts of tests are often not specified up front, but written by programmers or testers as the project proceeds.

Acceptance tests are begun before the program is written. In a perfect world, they serve as the outline for all the new features in an iteration of development. They are written in conjunction with the customer. Acceptance tests are an adjunct to stories. The stories are brief descriptions that provide a roadmap for the feature, but they don't supply anything concrete that can be automatically verified. That's where the tests come in.

The stories serve as a starting point for the discussion between the developers and the customer. These two hash out the details. The customer supplies the needed inputs and broad behaviors of the product. The customer comes from a high level, and the developer comes from a low level. Their goal is to meet in the middle in a place that captures the essence of the feature in way that the customer can understand, yet in enough detail that it can be quantified for testing. The product of this discussion is one or more acceptance tests.

Running Acceptance Tests

Acceptance tests occupy a different place in the build infrastructure than unit tests. The build fails if unit tests fails, but the product fails if acceptance tests fail. The build must always work, but the product doesn't have to work until delivery, so acceptance tests are not expected to pass with every build.

However, acceptance tests do yield useful information when run. Their successes and failures suggest how close the product is to completion. This information is interesting to developers in that it allows them to know how close they are to completion, but it's also interesting to customers. It should be available to both, and it should be produced regularly, but it doesn't need to be produced with every build.

The injunction against running functional tests with every build is even more important when you consider that functional tests are often slower than unit tests. Often they are orders of magnitude slower, in some instances taking literally days to run against mature products. Functional test farms are not unheard of with large products. Running them quickly can be a major engineering effort. At least one person I've spoken with has been porting their testing infrastructure to cloud computing environments such as Amazon's EC2 so that they can acquire hundreds of testing machines for short periods of time. Fortunately, I haven't had to confront such monsters myself.

This problem is remedied by adding a second kind of build to your continuous integration servers. The builds you've seen until now construct the software and then run the unit tests. I'll refer to them as *continuous builds*. The new builds do this, but they also run the functional tests after the unit tests complete. I'll refer to these as *formal builds*. Formal builds should run regularly, at least daily and preferably more often, and the results should be published to the customer.

PyFit

FIT (Framework for Integrated Tests, `http://fit.c2.com/`) is a tool developed by Ward Cunningham to facilitate collaboration between customers and developers. Tests are specified as tables, which are written in a tool the customer is familiar with. Developers or testers use these documents to write the tests. These tables are extracted from the documents, and they drive the acceptance tests. The end results are similarly formatted tables.

FITNESSE

FIT has given rise to a system called FitNesse, which is built around a wiki. Tests are entered as wiki pages, meaning that the people writing the tests need to learn a new tool, and they need to have access to the wiki in order to write tests. (Frankly, wikis are awful places to write tables.) Running the acceptance tests requires access to the wiki, too.

The real drawback for me is that the tests are independent of the code. It isn't possible to reproduce the acceptance criteria for a previous revision of the product. This may work for small groups or projects in which there is only ever one version of the system deployed at a time. While this is true of many hosted products, it's not true of many other software systems, particularly those that I work with.

FIT was originally produced for Java, but blessed clones have been created for many other languages. PyFit, written by John Roth, is Python's rendition. This flavor of FIT is well adapted to running from within the build. FIT has four components:

- *Requirement documents* are created by customers in conjunction with the developers. They specify the tests as tables, defining expected inputs and outputs, as well as identifying the associated test fixture.

- *Fixtures* are created by developers, and testers perform tests upon the applications.
- *Test runners* extract test data from tables in the requirement documents and then feed the data into the associated fixtures.
- *Reports* are created when the test runners are executed.

The relationship between these is shown in Figure 11-1.

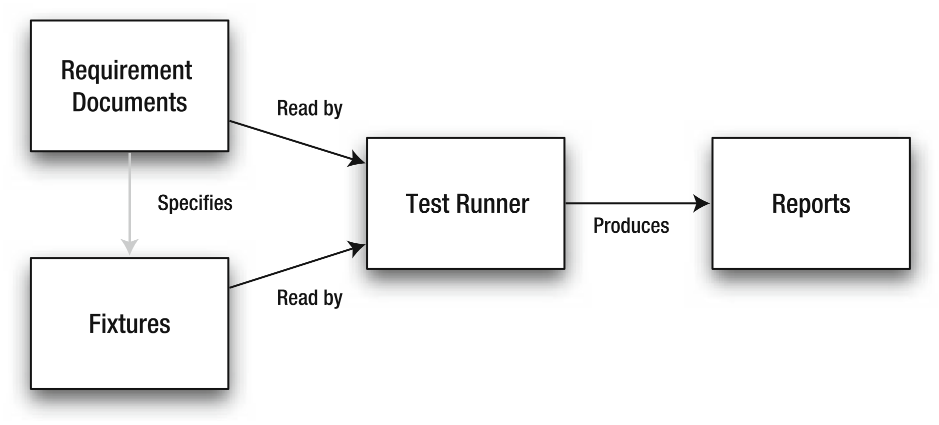


Figure 11-1. *The components of FIT*

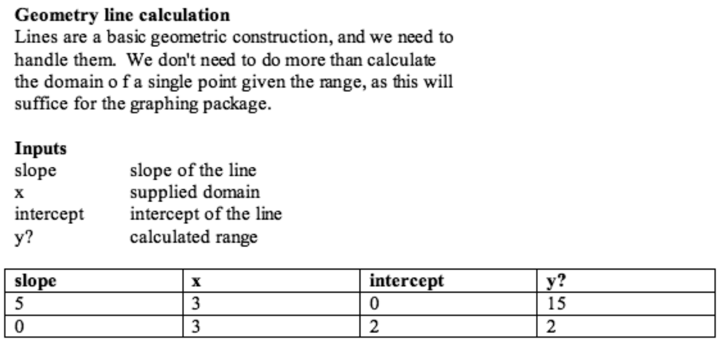
These components are common across all FIT implementations. Requirement documents can come from the filesystem or from FitNesse servers (see the "FitNesse" sidebar). In this chapter, they'll all be coming from a local filesystem.

Writing Requirements

With some FIT implementations, the requirement documents can be in many different formats. The test tables can be extracted from Microsoft Word documents, Excel spreadsheets, and HTML documents. Any format capable of representing tables can theoretically be used as a source document as long as a converter is supplied.

My favorite document source is a spreadsheet. Spreadsheets are eminently capable of creating, manipulating, and formatting tables, and everyone knows how to use one. In particular, the customers I work with have extensive experience with them.

Now for the bad news: PyFit doesn't support them. With PyFit, you get one choice: HTML files. The good news is that there are many tools that will edit HTML files so that you don't have to get your hands dirty. Figure 11-2 is a FIT spec being written with Microsoft Word.



Geometry line calculation
Lines are a basic geometric construction, and we need to handle them. We don't need to do more than calculate the domain o f a single point given the range, as this will suffice for the graphing package.

Inputs

slope	slope of the line
x	supplied domain
intercept	intercept of the line
y?	calculated range

slope	x	intercept	y?
5	3	0	15
0	3	2	2

Figure 11-2. *Writing an HTML spec document using Microsoft Word*

First and foremost, nonprogrammers are the intended audience for this document. It has a format that you'll use again and again:

- The first section describes the purpose of the acceptance test in human terms. It gives a background for everything that follows.
- The second section describes the variables used in the test. On one side are the names, and on the other are descriptions. Variable names ending in ? are results calculated by the test fixture.
- One or more tables follow. These define the acceptance criteria.

Notes may be freely mixed within the document. The documents may be as simple or as fancy as you desire. When the FIT runner processes the document, it will extract the tables and use them to drive the fixtures.

These specifications are not intended to be data-driven tests that exhaustively examine every possible input and output. The rows should specify interesting conditions. This data should emphasize the things that are important to determine about the test. It is a waste of everyone's time to supply 50 or 60 rows when only a few are necessary to convey a complete explanation of how the feature is supposed to work.

So who is everyone? Everyone includes the customers, the developers, and the testers. The preceding spec would most likely be created by the customer. It's rough and it needs refinement. The document might be shuttled back and forth by e-mail a few times while people discuss the possibilities. Eventually, the team huddles around someone's laptop and hashes out a finished version. The precise process by which this happens isn't important.

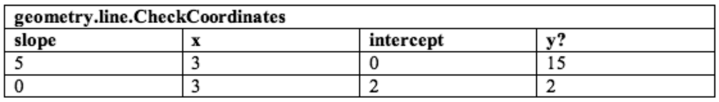
What matters is that a discussion happens between all stakeholders. This requirement document serves as the centerpiece for discussion. It forces everyone to decide on a concrete description of the feature.

Along the way, the team creates a common vocabulary describing the application and its actions. This vocabulary defines the system metaphor. At first, this vocabulary grows quickly, but the birth rate of new terms declines quickly.

Because the group creates these documents, they are at a level that all parties can understand. Each party involved will pull the documents in their own directions. The customers will want them to be too abstract, and the developers will want them to be too concrete. It will take a while before the participants learn to choose the right level. This is a good time to use people with both customer- and application-facing experience, such as sales engineers. They can serve as arbitrators early in the process.

This is FIT's magic. The documents are abstract enough that nontechnical people can grasp them and learn to write them with familiar tools, and they're detailed enough to produce tests from. Their level of abstraction allows them to serve as design specifications, and their level of detail makes them sufficient to replace technical requirement documents and test plans. Since they can be executed, they serve as formal acceptance criteria, which can be verified through automation. This also means that they won't fall out of date.

There is one crucial thing missing from the specification in Figure 11-2. FIT has no way of knowing which test fixture to use. This information is added to the form by the developer when they begin writing the test implementation. The new table is shown in Figure 11-3.



geometry.line.CheckCoordinates			
slope	**x**	**intercept**	**y?**
5	3	0	15
0	3	2	2

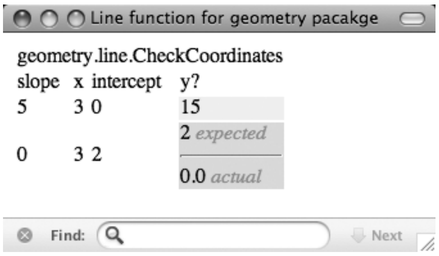
Figure 11-3. *A fixture binding has been added to the table.*

In the figure, a fixture binding has been added to the beginning of the table. Customers know that this line is techie magic stuff, so they avoid modifying it.

The first row in Figure 11-3 binds the fixture `geometry.line.CheckCoordinates` to the values in the table. A developer or tester creates the fixture. This is a simple (but broken) example.

```
from fit.ColumnFixture import ColumnFixture
class CheckCoordinates(ColumnFixture):
    _typeDict={
        "slope": "Float",
        "x": "Float",
        "intercept": "Float",
        "y": "Float",
        }
    slope = 0.0
    x = 0.0
    intercept = 0.0
    def y(self):
        return self.slope * self.x
```

When FIT runs, it produces an XML summary document and an HTML page for every test. The HTML page for this test and fixture is shown in Figure 11-4.



geometry.line.CheckCoordinates			
slope	x	intercept	y?
5	3	0	15
0	3	2	2 *expected* / 0.0 *actual*

Figure 11-4. *A FIT report for a broken test*

Successful test results are shown in green and unsuccessful ones are shown in red.

A link to these documents may be supplied to the customers and to management. This provides them with a self-service view into the status of the current development iteration. Once an organization adapts to using FIT as its primary specification tool, these reports supplant many status meetings and other formal communications.

At the beginning, customers and management will have to become familiar with the rhythms and patterns with which features are fulfilled. This will take time, and that needs to be made up front. Care should be taken when introducing the process, and expectations should be managed carefully.

Once everyone is comfortable with the process, it has social benefits. Developers and managers will feel less need to pester developers for project statuses. Developers will feel less harried and less pressured, and it will give them a greater sense of control.

It will also give customers and managers the feeling of more control, too. Instead of harrying development (which management likes doing as little as development likes receiving), they can look to the day's reports. Regular meaningful feedback that they can retrieve empowers them, and it builds trust in their team.

A Simple PyFit Example

The previous section has hopefully given you a good feeling for what FIT does and how it can benefit you. Unfortunately, setting up FIT is more complicated than it needs to be. The documentation is patchy at best, and if you're not using FitNesse, it fails to address many implementation questions, particularly to do with running PyFit from a build. Fortunately, this lack of information conceals a simple process.

It should be easy to set up a build on a new machine. Each additional package that you have to install to perform a build is a potential barrier. Each step is another delay when new people start on the project, when a machine is rebuilt, or when a new build server is added. Each new package has to be back-ported to all the existing build environments, too. There is little as frustrating as updating your source and discovering that you need to install a new package, so your build should carry its own infrastructure whenever possible.

The best way to learn about PyFit is to work with it. You can install it via `easy_install`, but none of the executables will work, and it won't be accessible to the build. Instead you'll install it into the `tools` directory that was created for JsUnit in Chapter 10, and the build will run it from there.

As I write, the current version is 0.8a2, and you can download it directly from `http://pypi.python.org/packages/source/P/PyFIT/PyFIT-0.8a2.zip`.

Note As of this writing, an earlier version is also available from the Download Now section of the FIT web site, at `http://fit.c2.com`. Hopefully, it will be up to date by the time you read this.

```
$ curl -o /tmp/PyFIT-0.8a2.zip -L➥
http://pypi.python.org/packages/source/P/PyFIT/PyFIT-0.8a2.zip
```

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  962k  100  962k    0     0   314k      0  0:00:03  0:00:03 --:--:--  356k
```

```
$ cd /Users/jeff/Documents/ws/rsreader/tools
$ ls -F
```

```
jsunit/
```

```
$ unzip /tmp/PyFIT-0.8a2.zip
```

```
Archive:  /tmp/PyFIT-0.8a2.zip
  inflating: PyFIT-0.8a2/PKG-INFO
  inflating: PyFIT-0.8a2/README.txt
  ...
  inflating: PyFIT-0.8a2/fit/tests/VariationsTest.py
  inflating: PyFIT-0.8a2/fit/tests/__init__.py
```

```
$ ls -F
```

```
PyFIT-0.8a2/     jsunit/
```

The tests will always run with the version of PyFit in `tools`, so the version information in the file name is superfluous.

```
$ mv PyFIT-0.8a2 pyfit
$ ls -F
```

```
pyfit/        jsunit/
```

Finally, you can remove the ZIP file that you downloaded earlier:

```
$ rm /tmp/PyFIT-0.8a2.zip
```

At this point, you should check the `pyfit` directory and all of its contents.

Giving the Acceptance Tests a Home

Unlike unit tests, acceptance tests do not run every time the code is built. They are run when the developer needs to see the results, or when iteration progress is checked. The latter is typically done on a regular basis by a special build. This means that the acceptance tests must be separated from unit tests. You can do this by creating a directory for acceptance tests:

```
$ cd /Users/jeff/Documents/ws/rsreader
$ mkdir acceptance
$ ls -F
```

```
acceptance/    ez_setup.py       setuptools-0.6c7-py2.5.egg
build/           javascript/          src/
dist/            setup.cfg            thirdparty/
ez_setup.py      setup.py             tools/
```

You must have locations to store requirement documents, fixtures, and reports, and they should be separate:

```
$ mkdir acceptance/requirements
$ mkdir acceptance/fixtures
$ mkdir acceptance/reports
```

You should check these into your source repository at this point.

Your First FIT

Requirement documents are at the heart of FIT. There are a number of different families of tests that can be created with FIT. The type of test I'm showing you how to create is a column fixture. A *column fixture* is a table in which each column represents a different input or output to the test. Each row is a different combination of these values.

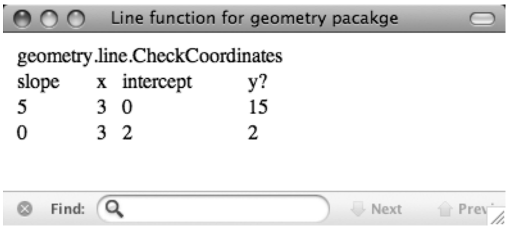
You're limited to HTML documents at this time—it's the format that PyFit currently understands. This doesn't mean that you have to write them by hand, though. Microsoft Word, Adobe Dreamweaver, or any tool capable of reading and writing HTML will speed the job along. The requirement document shown following is written to the file `acceptance/requirements/geometry/line.html`:

```
$ cat acceptance/requirements/geometry/line.html
```

```
<html>
    <head>
        <title>Line function for geometry pacakge</title>
    </head>
    <body>
        <table>
             <tr><td colspan="4">geometry.line.CheckCoordinate</td></tr>
             <tr>
                   <td>slope</td>
                   <td>x</td>
```

```
                    <td>intercept</td>
                    <td>y?</td>
             </tr>
             <tr>
                   <td>5</td>
                   <td>3</td>
                   <td>0</td>
                   <td>15</td>
             </tr>
           <tr>
                   <td>0</td>
                   <td>3</td>
                   <td>2</td>
                   <td>2</td>
              </tr>
         </table>
     </body>
</html>
```

The test-specific information is printed in bold. As with all HTML, it's much easier to interpret in a browser, as Figure 11-5 shows.



geometry.line.CheckCoordinates			
slope	x	intercept	y?
5	3	0	15
0	3	2	2

Figure 11-5. *The simple test table shown in a browser*

This table represents column fixture, and it has three important parts:

- The first row has one cell that reads `geometry.line.CheckCoordinates`. This designates the fixture for the test.
- The second row assigns variable names to columns. Columns names ending in ? are expected results from the tests.
- Each subsequent row contains a set of test values. A test fixture is constructed for each one, and the values from each column are used to either prepare the test or compare the results.

Column fixtures are commonly used to represent business process. A friend's favorite FIT examples come from a marketing department that he worked in. The tables consisted of criteria for when coupons should be given to customers. The conditions were involved, but walking a person through the process answered many questions about which values were important and which were not. They were all easily represented as columns.

There are other kinds of fixtures, and they have different behaviors, but they're all specified as tables.

The Fixture

A fixture can be thought of as a kind of command object. A fixture is created, the test values are set in the fixture, and then command methods are called. The variable names and command method names match those in the table.

Here, the fixtures are placed into `acceptance/fixtures`, which is the root of a Python package tree. Since it is a Python package tree, each subdirectory needs to have an `__init__.py` file.

```
$ mkdir acceptance/fixtures/geometry
$ touch acceptance/fixtures/geometry/__init__.py
$ mkdir acceptance/fixtures/geometry/line
$ touch acceptance/fixture/geometry/line/__init__.py
```

You'll notice something weird here—the binding line in the table is `geometry.line.CheckCoordinates`. It would be reasonable for you assume that it specifies the class `CheckCoordinates` in the module `geometry.line`, but you would be wrong. It actually specifies the class `CheckCoordinates` in the module `geometry.line.CheckCoordinates`. Go figure. Here is the fixture's code:

```
$ cat acceptance/fixtures/geometry/line/CheckCoordinates.py
```

```
from fit.ColumnFixture import ColumnFixture

class CheckCoordinates(ColumnFixture):
    _typeDict={
        "slope": "Float",
        "x": "Float",
        "intercept": "Float",
        "y": "Float",
        }

    slope = 0.0
    x = 0.0
    intercept = 0.0

    def y(self):
        return self.slope * self.x + self.intercept
```

The fixture breaks into three parts. All of the names correspond to columns in the requirements table minus any meaningful punctuation, such as the trailing ? on test results.

- The first section declares the test's variables and their types. This reflects FIT's strongly typed Java heritage. The argument types could be inferred from member definitions.
- The second section declares and initializes defaults for the variables that FIT supplies from each row in the requirements table.
- The third section defines the methods that produce results. Each is named after the corresponding column in the requirements table.

For this fixture, you can think of the execution process as following these steps:

```
f = CheckCoordinates()
converted_row = converted_values(row, f._typeDict)
f.slope = converted_row['slope']
f.x = converted_row['x']
f.intercept = converted_row['intercept']
f.recordAssertionResults('y?', converted_row['y?'], f.y())
```

The reality is more complicated, but this captures the essence of the process.

Running PyFit

PyFit supplies many different programs for running tests. These are located in `tools/pyfit/fit`. You want `FolderRunner.py`, which reads requirement documents from one directory and writes the finished reports to another. `FolderRunner.py` came from a ZIP file, and on UNIX systems this means that the execute bit isn't set, so you'll have to call Python to run it.

```
$ python tools/pyfit/fit/FolderRunner.py acceptance/requirements acceptance/reports
```

```
Result print level in effect. Files: 'e' Summaries: 't'
Total tests Processed: 0 right, 0 wrong, 0 ignored, 0 exceptions
```

This indicates that no tests were found. By default, `FolderRunner.py` searches only the top-level directory you specified, but `line.html` is in a subdirectory. The `+r` flag tells `FolderRunner.py` to search for tests recursively.

```
 $ python tools/pyfit/fit/FolderRunner.py +r acceptance/requirements➥
acceptance/reports
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Processing Directory: geometry
0 right, 0 wrong, 0 ignored, 1 exceptions line.html
Total this Directory: 0 right, 0 wrong, 0 ignored, 1 exceptions
Total tests Processed: 0 right, 0 wrong, 0 ignored, 1 exceptions
```

This is better. It found the test—however, it should display `1 right, 0 wrong, 0 ignored, 0 exceptions`. Obviously there's a problem here, so you'll want to look at the results. The test was in `acceptance/requirements/geometry/line.html`, so the results are in `acceptance/reports/geometry/line.html`.

```
$ cat acceptance/reports/geometry/line.html
```

```
...
            <tr><td colspan="4" class="fit_error">geometry.line.CheckCoordinates➥
<hr>The module 'geometry.line' was not found.</td></tr>
...
```

The relevant message in the preceding report is shown in bold. The module `geometry.line` wasn't found because the directory `acceptance/fixtures` isn't on the `PYTHONPATH`. This is something that can be easily fixed:

```
$ export PYTHONPATH=acceptance/fixtures
$ python tools/pyfit/fit/FolderRunner.py +r acceptance/requirements➥
acceptance/reports
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Total tests Processed: 1 right, 0 wrong, 0 ignored, 0 exceptions
```

This is exactly what you wanted. The tests have run and the results have been generated.

Making It Easier

You're going run PyFit frequently. Remembering all those values is a hassle, and you're certainly not going to want to type them over and over again, so you should create a script to do it for you.

The tool is going to have to go someplace, and you don't have a location for generic tool scripts yet. You can put them in a subdirectory of `tools`:

```
$ mkdir tools/bin
```

The script is called `tools/bin/accept.py`. The first version is shown in Listing 11-1.

Listing 11-1. *The First Pass at the Acceptance Testing Script*

```
#!/usr/local/bin/python

from subprocess import Popen
import sys

cmd = "%(python)s %(fitrunner)s +r %(requirements)s %(reports)s"
expansions = dict(python=sys.executable,
                  fitrunner='./tools/pyfit/fit/FolderRunner.py',
                  requirements='./acceptance/requirements',
                  reports='./acceptance/reports')
env = dict(PYTHONPATH='acceptance/fixtures')

proc = Popen(cmd % expansions, shell=True, env=env)
proc.wait()
```

On UNIX systems, you must make the script executable before it can be run:

```
$ chmod a+x tools/bin/accept.py
```

Before you can verify that it's doing the right thing, you need to remove `PYTHONPATH` from your shell's environment:

```
$ unset PYTHONPATH
```

At this point, running `FolderRunner.py` by hand results in an exception again:

```
$ python tools/pyfit/fit/FolderRunner.py +r acceptance/requirements➥
acceptance/reports
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Processing Directory: geometry
0 right, 0 wrong, 0 ignored, 1 exceptions line.html
Total this Directory: 0 right, 0 wrong, 0 ignored, 1 exceptions
Total tests Processed: 0 right, 0 wrong, 0 ignored, 1 exceptions
```

However, running the script works:

```
$ tools/bin/accept.py
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Total tests Processed: 1 right, 0 wrong, 0 ignored, 0 exceptions
```

There is still a problem, though. The script will only run from the project's root directory:

```
$ cd tools
$ bin/accept.py
```

```
/Library/Frameworks/Python.framework/Versions/2.5/Resources/Python.app/Contents/➥
MacOS/Python: can't open file './tools/pyfit/fit/FolderRunner.py':➥
[Errno 2] No such file or directory
```

All the paths are relative, which is good. It means that the tools work no matter where the project is placed on the filesystem. It's bad, however, in that the script will only run from the project's root directory. You could require people to run it from there, but that's inconvenient—people forget rules like that, so this isn't a particularly robust solution.

You could extract the root directory from an environment variable and require people to set that, but this creates problems for people working on multiple branches. In order to move between them, they'll have to reconfigure their environment.

A more robust solution determines the directory from the path. The revised program is shown in Listing 11-2.

Listing 11-2. *The Script accept.py Now Runs from Any Directory Within the Project*

```
#!/usr/local/bin/python

import os
from subprocess import Popen
import sys

def bin_dir():
    return os.path.dirname(os.path.abspath(__file__))

def find_dev_root(d):
    setup_py = os.path.join(d, 'setup.py')
    if os.path.exists(setup_py):
       return d
    parent = os.path.dirname(d)
    if parent == d:
       return None
    return find_dev_root(parent)

dev_root = find_dev_root(bin_dir())
if dev_root is None:
     msg = "Could not find development environment root"
     print >> sys.stderr, msg
     sys.exit(1)
os.chdir(dev_root)

cmd = "%(python)s %(fitrunner)s +r %(requirements)s %(reports)s"
expansions = dict(python=sys.executable,
                  fitrunner='./tools/pyfit/fit/FolderRunner.py',
                  requirements='./acceptance/requirements',
                  reports='./acceptance/reports')
env = dict(PYTHONPATH='tools/pyfit/fit:acceptance/fixtures')

proc = Popen(cmd % expansions, shell=True, env=env)
proc.wait()
```

The project's root is the ancestor of the `tools/bin` directory that contains `setup.py`. The functions `bin_dir()` and `find_dev_root(directory)` perform this search. The solution is a mouthful, but it can be used over and over again. When you need to reuse it, you should move it into a common module that gets installed with your development environment.

The script now runs from anywhere:

```
$ cd /tmp
$ /Users/jeff/Documents/ws/rsreader/tools/bin/accept.py
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Total tests Processed: 1 right, 0 wrong, 0 ignored, 0 exceptions
```

Besides learning a general technique, you've made the job of running from Buildbot that much easier.

FIT into Buildbot

In a world with limitless computing resources, there would only be one kind of a build. That build would execute the program's full test suite. It would run every unit test, acceptance test, functional test, and performance test. In the real world, however, there are rarely enough resources to do this.

Large mature projects often have full test suites that take hours if not days to run. Many of the tests thirstily consume resources. For obvious reasons, performance tests are consummate gluttons. Functional tests for products such as embedded systems or device drivers may require specialized hardware.

While the size of the unit test suite is roughly proportional to the size of the code base, the size of the functional test suite is proportional to the code base's age. As the code ages, the functional suite bloats. Eventually, it may grow so large that massive parallelization is the only solution to running it in a reasonable time.

Functional tests need to be broken out long before then, and acceptance tests do, too. The team needs regular progress reports for the acceptance tests, so the build containing them is produced at regular intervals.

Casting back to Chapter 5, you'll recall that we set up a build master and a build slave named `rsreader-linux`. The build system produced builds for both Python 2.4 and Python 2.5. The builds were triggered whenever a change was submitted to the Subversion server.

You're about to add a second kind of build that includes the acceptance tests. This build will run several times a day, and it will run even if there are no recent changes.

You do this by defining the following items in the Buildbot configuration file `master.cfg`:

- A *schedule* that determines when a *builder* runs
- A *build factory* that constructs a build
- *Build steps* that the build performs
- A *builder* that ties together a build factory, a slave, and a factory

Before you can run a builder, you need to set up the infrastructure on the slave.

Preparing the Slave

Each builder needs a unique build directory under `/usr/local/buildbot/slave/rsreader`. The previous two were `full-py2.4` and `full-py2.5`. This will be a Python 2.5 builder, and you should name it `acceptance-py2.5`.

```
slave$ cd /usr/local/buildbot/rsreader/slave
slave$ sudo -u build mkdir acceptance-py2.5
```

The builder needs its own Python installation. You can copy that from `full-2.5`.

```
slave$ sudo -u build cp -rp full-py2.5/python2.5 acceptance-py2.5/python2.5
```

Now you can set up the builder.

Run New Builder, Run!

There's an old maxim about software: Make it run. Make it run right. Make it run fast. It applies to configuration, too. Right now, you want to focus on making the builder run. Later, you can make it run right. This prevents you from conflating basic configuration errors with the mistakes you make while hacking out a new builder, so you should configure the build like the existing ones. They are defined by `/usr/local/buildbot/master/rsreader/master.cfg` as follows:

```
b1 = {'name': "buildbot-full-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.5",
      'factory': pythonBuilder('2.5'),
      }
b2 = {'name': "buildbot-full-py2.4",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.4",
      'factory': pythonBuilder('2.4'),
      }
c['builders'] = [b1, b2,]
```

Adding the new definition gives you this:

```
...
b2 = {'name': "buildbot-full-py2.4",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.4",
      'factory': pythonBuilder('2.4'),
      }
b3 = {'name': "buildbot-acceptance-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "acceptance-py2.5",
      'factory': pythonBuilder('2.5'),
      }
c['builders'] = [b1, b2, b3]
```

After doing this, you should reload the Buildbot configuration to see if you've introduced any errors:

```
master$ buildbot reconfig /usr/local/buildbot/master/rsreader
```

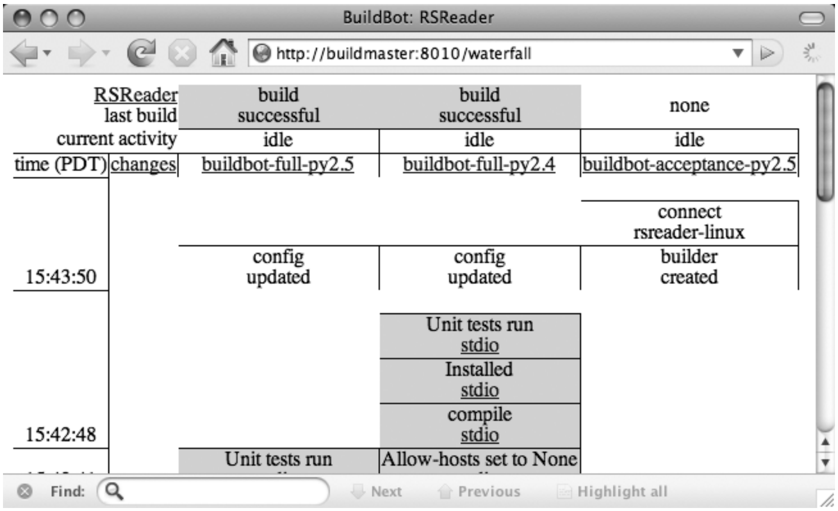
```
sending SIGHUP to process 52711
2008-05-05 15:59:56-0700 [-] loading configuration from /usr/local/buildbot/master
2008-05-05 15:59:56-0700 [-] updating builder buildbot-full-py2.4: factory changed
...
Reconfiguration appears to have completed successfully.
```

The reconfig worked, so you can safely continue. The new builder must be scheduled. Here is the section of `master.cfg` containing the new scheduler definition:

```
####### SCHEDULERS

from buildbot.scheduler import Nightly, Scheduler
c['schedulers'] = []
c['schedulers'].append(Scheduler(name="rsreader under python 2.5",
                                 branch=None,
                                 treeStableTimer=60,
                                 builderNames=["buildbot-full-py2.5"]))
c['schedulers'].append(Scheduler(name="rsreader under python 2.4",
                                 branch=None,
                                 treeStableTimer=60,
                                 builderNames=["buildbot-full-py2.4"]))
c['schedulers'].append(Scheduler(name="Acceptance tests under python 2.5",
                                 branch=None,
                                 treeStableTimer=60,
                                 builderNames=["buildbot-acceptance-py2.5"]))
```

At this point, you should reconfigure the master. A quick look at the waterfall display in Figure 11-6 shows that the builder is online.



	RSReader last build	build successful	build successful	none
	current activity	idle	idle	idle
time (PDT)	changes	buildbot-full-py2.5	buildbot-full-py2.4	buildbot-acceptance-py2.5
				connect rsreader-linux
15:43:50		config updated	config updated	builder created
			Unit tests run stdio	
			Installed stdio	
15:42:48			compile stdio	
		Unit tests run	Allow-hosts set to None	

Figure 11-6. *The new builder is alive.*

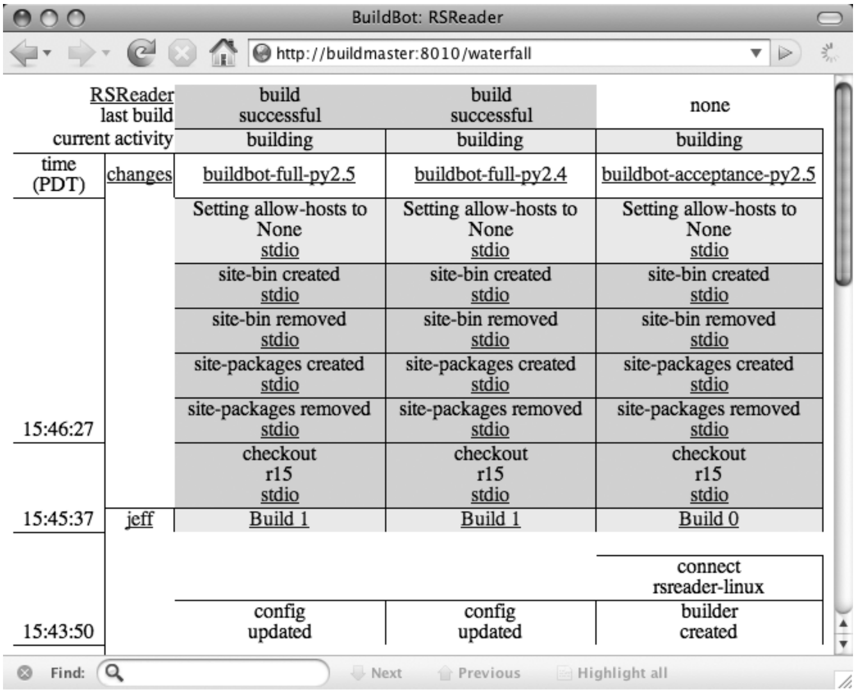
You send a notification to verify that the builder actually works:

```
master$ buildbot sendchange --master buildmaster:4484 -u jeff -n 30 setup.py
```

```
change sent successfully
```

You can watch the build happen on the waterfall display.

Figure 11-7 shows the build completing on my system. You should see something similar. Now that you know the builder runs, you have to make it run right. Running the acceptance tests requires making a new builder factory.


BuildBot: RSReader

http://buildmaster:8010/waterfall

	RSReader last build	build successful	build successful	none
	current activity	building	building	building
time (PDT)	changes	buildbot-full-py2.5	buildbot-full-py2.4	buildbot-acceptance-py2.5
		Setting allow-hosts to None stdio	Setting allow-hosts to None stdio	Setting allow-hosts to None stdio
		site-bin created stdio	site-bin created stdio	site-bin created stdio
		site-bin removed stdio	site-bin removed stdio	site-bin removed stdio
		site-packages created stdio	site-packages created stdio	site-packages created stdio
15:46:27		site-packages removed stdio	site-packages removed stdio	site-packages removed stdio
		checkout r15 stdio	checkout r15 stdio	checkout r15 stdio
15:45:37	jeff	Build 1	Build 1	Build 0
				connect rsreader-linux
15:43:50		config updated	config updated	builder created

Figure 11-7. *The new builder builds.*

The current builder factory does most of what you want. You can easily leverage this. The new builder will call the old one and then add its own steps.

```
def pythonBuilder(version):
    python = python_(version)
    nosetests = nosetests_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)

    f = factory.BuildFactory()
...
    f.addStep(ShellCommand,
              command=[python, "./setup.py", "test"],
              description="Running unit tests",
              descriptionDone="Unit tests run")
    return f
```

```
def pythonAcceptanceBuilder(version):
    f = pythonBuilder(version)
    f.addStep(ShellCommand,
              command=[python_(version, "./tools/bin/accept.py"],
              description="Running acceptance tests",
              descriptionDone="Acceptance have been run")
    return f
```

You should reload the master to ensure that you haven't made any mistakes. The new builder definition still references the old builder factory, so you make the following change to hook in the new one:

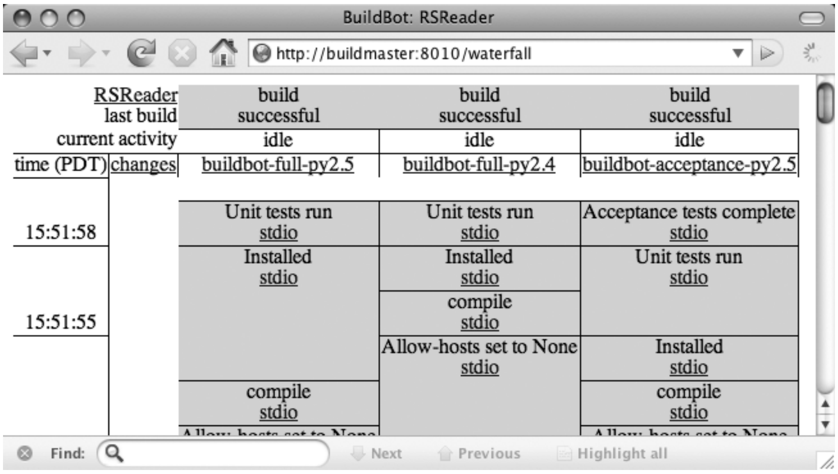
```
b3 = {'name': "buildbot-acceptance-py2.5",
         'slavename': "rsreader-linux",
         'builddir': "acceptance-py2.5",
         'factory': pythonAcceptanceBuilder('2.5'),
      }
```

Now you should reload the master again and verify that you didn't make an error. With that verified, you should trigger a build again:

```
master$ buildbot sendchange --master buildmaster:4484 -u jeff -n 30 setup.py
```

```
change sent successfully
```

When the build completes, you'll see the new step, as shown in Figure 11-8.



RSReader last build		build successful	build successful	build successful
current activity		idle	idle	idle
time (PDT)	changes	buildbot-full-py2.5	buildbot-full-py2.4	buildbot-acceptance-py2.5
15:51:58		Unit tests run stdio	Unit tests run stdio	Acceptance tests complete stdio
		Installed stdio	Installed stdio	Unit tests run stdio
			compile stdio	
15:51:55			Allow-hosts set to None stdio	Installed stdio
		compile stdio		compile stdio

Figure 11-8. *PyFit has been run.*

The output from the build step shows that the build has been run successfully, but reports aren't available.

Making the Reports Available

When the PyFit step completes, the results are left in `acceptance/reports`. They're HTML documents, so you can publish them by copying them into a web server's document tree. Since you'll invariably have more than one build slave, you'll need to centralize the reporting.

Fortunately, this is easy to do with Buildbot. It can copy files from the slaves to the master, and you can use the master's internal web server to present the reports.

You'll want to track the project's progress over time, so you'll want to publish all the acceptance reports simultaneously. Copying them to a fixed location on the master won't work. If you do that, then the latest reports will overwrite the previous reports, so you'll need to copy each to a different location.

The combination of the Python version and Subversion revision make a useful identifier. These should effectively be unique. You'll copy the results to

```
public_html/rsreader/acceptance/%(subversion_revision)s-py%(python_version)s/reports
```

Buildbot makes build-related information such as the branch and the revision available through build parameters. The documentation would lead you to believe that the feature is complete, but it isn't.[1] The mechanisms for setting and reading build properties are in place, but the final pieces that set them are missing. Along the way, you're going to have to supply some of this machinery yourself.

The publishing process has the following steps:

- The slave packages the reports.
- The master copies the reports from the slave.
- The master unpacks and moves the reports to the web server.

Each of these corresponds to a build step.

Packaging the Reports

The build tool `accept.py` puts the reports into `acceptance/reports`. These files need to be zipped into a single archive so that the next step can copy them over. At this point, only this directory must be copied, but that is likely to change in the future. It's temping to issue shell commands directly, and this will work. However, to do this, Buildbot must understand how the build is structured.

Using formal interfaces allows developers to restructure the build without having to modify the build server. A red flag should go up when you find yourself tweaking the build system to manipulate build's internal structure. Instead you resolve the issue by putting the details within the build itself; this often involves creating some sort of tool.

You're about to create a tool called `package_reports.py`. Like `access.py`, it lives in `rsreader/tools/bin`. Here's the code:

1. This is true as of Buildbot 0.7.7.

```
#!/usr/local/bin/python

import os
from subprocess import Popen
import sys

if len(sys.argv) != 2:
    msg = """Usage: %s ZIPFILE

This must be run from the project's root.
"""
    print >> sys.stderr, msg % sys.argv[0]
    sys.exit(0)

filename = sys.argv[1]
reports = os.path.join("acceptance", "reports")
proc = Popen(['zip', '-r', filename, reports])
proc.wait()
```

Currently it just zips up the directory `acceptance/reports` into the file `reports.zip`, but this still hides the layout from the build scripts. It takes the name of the ZIP file as its only argument, and it runs from the slave's `builder` directory. You should check this file in.

The build step is straightforward:

```
def reportsFile():
    return "reports.zip"

 def pythonAcceptanceBuilder(version):
    f.addStep(ShellCommand,
              haltOnFailure=True,
              command=[python_(version), "./tools/bin/accept.py"],
              description="Running acceptance tests",
              descriptionDone="Acceptance tests complete")
   f.addStep(ShellCommand,
              haltOnFailure=True,
              command=[python_(version),
                  "./tools/bin/package_reports.py",
                 reportsFile()],
              description="Packaging build reports",
              descriptionDone="Build reports packaged")
```

After you make this change, you should force a build, and it should succeed.

Retrieving the Reports

The Buildbot installation will have acceptance tests for both Python 2.4 and 2.5. These builds will run at the same time, so you need to keep the retrieved ZIP files separate. If you don't, then they may stomp on each other.

An easy way to do this is to create an upload directory for each builder. You'll name these directories `uploaded-py`*`Version`*. The version corresponds to the Python version (e.g., 2.5). These directories must be created before the copy happens.

The `FileUpload` build step copies the files. You supply the source file name on the slave, and the destination file name on the master. These can be absolute or relative path names. On the slave, they are relative to the build directory, and on the master, they are relative to the server root directory.

```
def reportsFile():
    return "reports.zip"

def makeUploadDirectory(version):
    uploads = uploadDirectory(version)
    if not os.path.exists(uploads):
       os.mkdir(uploads, 0750)

def reportsFileLocal(version):
    return os.path.join(uploadDirectory(version), reportsFile())

def uploadDirectory(version):
    return "uploaded-py%s" % version

def pythonAcceptanceBuilder(version):
    f.addStep(ShellCommand,
              haltOnFailure=True,
              command=[python_(version), "./tools/bin/accept.py"],
              description="Running acceptance tests",
              descriptionDone="Acceptance tests complete")
   f.addStep(ShellCommand,
              haltOnFailure=True,
              command=[python_(version),
                  "./tools/bin/package_reports.py",
                 reportsFile()],
              description="Packaging build reports",
              descriptionDone="Build reports packaged")
f.addStep(FileUpload,
              haltOnFailure=True,
              slavesrc=reportsFile(),
              masterdest=reportsFileLocal(version))
```

Once you've made these changes, you should restart the master and trigger a build. Once the build succeeds, you're ready to publish the reports.

Publishing the Reports

You'd think that unpacking a ZIP file to a directory would be an easy job. Unfortunately, you'd be wrong. The command you're trying to run is

```
unzip -qq -o -d %(destination)s %(zipfile)s
```

The -qq option silences all output, and the -o option overwrites existing files without prompting. The ZIP file will be unpacked in the directory specified by the -d option, which also happens to create any missing directories.

The destination location contains the revision number and the Python version. This path is public_html/rsreader/%(revision)s-py%(version), and it is relative to Buildbot's root directory on the master. Getting the revision number is the first hurdle.

Getting the Revision

The Buildbot 0.7.7 documentation suggests that this information is in the "revision" build property. BuildStep.getProperty() and BuildStep.setProperty() form the core of the build properties system, but only custom tasks can use them. ShellCommand classes have another access mechanism: command strings are wrapped in the WithProperties class, and this class expands them at runtime.[2]

The documentation suggests that the "revision" property is set by the SVN build step. Alas, that is not true. You must create a customized SVN build step to set this property.

SVN calls svn checkout to create the build directory. One of the last lines from this command is r'^Checked out revision \d+\.', where \d+ is the revision number. All you need to do is search the log for that pattern.

```
import re
from StringIO import StringIO
...
class SVNThatSetsRevisionProperty(SVN):

     checked_out_line = re.compile("^Checked out revision (\d+)\.")

     def createSummary(self, log):
        for line in StringIO(log.getText()).readlines():
           found = self.checked_out_line.search(line)
           if found:
               self.setProperty("revision", found.group(1))
        return SVN.createSummary(self, log)
```

The method createSummary(log) gives you access to the log just after it completes and just before Buildbot makes any decisions about the step's status. There are quite a few other hook methods that let you intercept a step's control flow.

It is time to replace the old SVN step with the new one:

```
def pythonBuilder(version):
    python = python_(version)
    nosetests = nosetests_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)
```

2. Used thusly: ShellCommand(command=["rm", "-rf", **WithProperties**("uploaded-py%(revision)s")]).

```
    f = factory.BuildFactory()
    f.addStep(SVNThatSetsRevisionProperty,
       baseURL="svn://repos/rsreader/",
       defaultBranch="trunk",
       mode="clobber",
       timeout=3600)
    f.addStep(ShellCommand,
              command=["rm", "-rf", site_pkgs],
              description="removing old site-packages",
              descriptionDone="site-packages removed")
...
```

You'll see no change when you fire off the build this time.

Publishing the Build

The publishing step only runs on the master. Buildbot doesn't provide much support for this, but it's not too hard. You'll start with a very simple build step:

```
from buildbot.process.buildstep import BuildStep
from buildbot.status.builder import SUCCESS
...
class InstallReports(BuildStep):

    def start(self):
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

This step ties it into the build:

```
def pythonAcceptanceBuilder(version):
   ...
    f.addStep(FileUpload,
              haltOnFailure=True,
              slavesrc=reportsFile(),
              masterdest=reportsFileLocal(version))
    f.addStep(InstallReports, haltOnFailure=True)
```

Run it, and it should produce a successful green build step.

You'll need to publish to a URL. On my system, this URL is `http://buildmaster.theblobshop.com:8010/rsreader/%(revision)s-py%(version)s`. You'll use a fixed URL the first time, and you'll subsequently parameterize it:

```
class InstallReports(BuildStep):
    url = "http://buildmaster.theblobshop.com:8010/rsreader/18-py2.5"

    def start(self):
        self.setUrl("reports", self.url)
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

Parameterizing the URL requires the code revision and Python version. The customized SVN step supplies the revision via the build parameter. The build factory supplies the version as an argument, as with other steps in the build factory.

```
class InstallReports(BuildStep):
    url = "http://buildmaster.theblobshop.com:8010/rsreader/18-py2.5"

    def __init__(self, version, **kw):
        self.version = version
        BuildStep.__init__(self, **kw)
        self.addFactoryArguments(version=version)

    def start(self):
        self.setUrl("reports", self.url)
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

When you initialize a build step, you must always pass on the other arguments to the parent. Not doing this leads to unpredictable behavior.

Now you have to change how the factory calls the build step.

```
def pythonAcceptanceBuilder(version):
   ...
    f.addStep(FileUpload,
              haltOnFailure=True,
              slavesrc=reportsFile(),
              masterdest=reportsFileLocal(version))
    f.addStep(InstallReports,
              haltOnFailure=True,
              version=version)
```

Now the build step has access to the revision and version, so you can finally parameterize the URL:

```
class InstallReports(BuildStep):
    url = "http://buildmaster.theblobshop.com:8010" \
       "/rsreader/%(revision)s-py%(version)s"

    def __init__(self, version, **kw):
        self.version = version
        BuildStep.__init__(self, **kw)
        self.addFactoryArguments(version=version)
    def expansions(self):
        return {'revision': self.getProperty('revision'),
                'version': self.version}

    def start(self):
        self.setUrl("reports", self.url % self.expansions())
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

The step now expands the URL. You'll want the step to go yellow while it runs.

```
    def start(self):
        self.step_status.setColor("yellow")
        self.step_status.setText(["Publishing reports", "Unzipping package"])
        self.setUrl("reports", self.url % self.expansions())
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

Oh yeah, and you'll want to unzip the file too. There is a catch, though. Buildbot uses the Twisted framework. Twisted is an asynchronous interaction system, and it extensively manipulates operating system signals. This interferes with the normal subprocess calls, resulting in strange exceptions whenever you invoke any asynchronous operations—like checking a process's exit code.

Luckily, Twisted supplies process-handling methods. These methods include `getProcessOutput()`, `getProcessValue()`, and `getProcessOutputAndValue()`. These live in the package `twisted.internet.utils`. You'll use `getProcessValue()` first:

```
import os
from twisted.internet.utils import getProcessValue
...
class InstallReports(BuildStep):
    url = "http://buildmaster.theblobshop.com:8010/" \
       "rsreader/%(revision)s-py%(version)s"
    dest_path = "public_html/rsreader/%(revision)s-py%(version)s"
...
    def start(self):
        self.step_status.setColor("yellow")
        self.step_status.setText(["Publishing reports", "Unzipping package"])
        dest = self.dest_path % self.expansions()
        if not os.path.exists(dest):
            os.makedirs(dest, 0755)
        cmd = "/usr/bin/unzip"
        zipfile = os.path.abspath(reportsFileLocal(self.version))
        args = ("-qq",
                "-o",
                "-d", dest,
                zipfile)
        getProcessValue(cmd, args)
        self.setUrl("reports", self.url % self.expansions())
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

When you run this step, it should succeed. If it succeeds, you'll find the results in the directory `public_html/rsreader/%(revision)s-py2.5/acceptance/reports`, under the Buildbot master's directory.[3] What if it doesn't succeed, though? You haven't checked the results of `getStatusValue()`, so you don't know. There's nothing special about checking the results, however:

```
from buildbot.status.builder import FAILURE, SUCCESS
...
    def start(self):
        self.step_status.setColor("yellow")
        self.step_status.setText(["Publishing reports", "Unzipping package"])
        dest = self.dest_path % self.expansions()
        if not os.path.exists(dest):
            os.makedirs(dest, 0755)
        cmd = "/usr/bin/unzip"
        zipfile = os.path.abspath(reportsFileLocal(self.version))
        args = ("-qq",
                "-o",
                "-d", dest,
                zipfile)
        result = getProcessValue(cmd, args)
        if result == 0:
            self.setUrl("reports", self.url % self.expansions())
            self.step_status.setColor("green")
            self.step_status.setText("Reports published")
            self.finished(SUCCESS)
        else:
            self.step_status.setColor("red")
            self.step_status.setText("Report publication failed")
            self.finsished(FAILURE)
```

If your build failed before, then it's worth trying it again. You should see a red step without a URL this time. Once you have this step running, you can refactor it:

```
def start(self):
    self.begin()
    self.make_report_directory()
    results = self.unzip()
    if results == 0:
        self.succeed()
```

3. On my system, this is `/usr/local/buildbot/master/rsreader`.

```
   else:
        self.fail()

def begin(self):
    self.step_status.setColor("yellow")
    self.step_status.setText(["Publishing reports", "Unzipping package"])

def make_report_directory(self):
    dest = self.dest_path % self.expansions()
    if not os.path.exists(dest):
        os.makedirs(dest, 0755)

def unzip(self):
    cmd = "/usr/bin/unzip"
    zipfile = os.path.abspath(reportsFileLocal(self.version))
    args = ("-qq",
              "-o",
              "-d", self.dest_path % self.expansions(),
              zipfile)
    return getProcessValue(cmd, args)

def succeed(self):
    self.setUrl("reports", self.url % self.expansions())
    self.step_status.setColor("green")
    self.step_status.setText("Reports published")
    self.finished(SUCCESS)

def fail(self):
    self.step_status.setColor("red")
    self.step_status.setText("Report publication failed")
    self.finsished(FAILURE)
```

As usual, you should verify the changes by running the build. You no longer need to run tests on this builder, so you can put it on a regular schedule.

Getting Regular Builds

Regularly timed builds are run with the `Nightly` scheduler. It runs builds at times specified by a combination of `dayOfMonth`, `dayOfWeek`, `month`, `hour`, and `minute`. If a value isn't specified, then it isn't used to match the date. The exception is `minute`, which defaults to `0`. If you've ever used the UNIX `cron` command, then you'll be right at home.

All of this makes more sense with a few examples. This will run every March at 6:42 PM and 9:42 PM:

```
month=3, hour=(18, 21), minute=42
```

This will run at 6:00 AM on every Monday that falls on the second or third day of the month:

```
dayOfMonth=(2, 3), dayOfWeek=0, hour=6
```

I like to run acceptance builds several times a day:

- Once in the morning so that people know the project status at the beginning of the day, including any changes that people made the night before
- Once at lunch to pick up the morning's work
- Once near the end of the day to pick up the afternoon's changes and report them before everyone goes home

Scheduling this takes just a few lines:

```
from buildbot.scheduler import Nightly, Scheduler
c['schedulers'] = []
c['schedulers'].append(Scheduler(name="rsreader under python 2.5",
                                 branch=None,
                                 treeStableTimer=5,
                                 builderNames=["buildbot-full-py2.5"]))
c['schedulers'].append(Scheduler(name="rsreader under python 2.4",
                             branch=None,
                             treeStableTimer=5,
                             builderNames=["buildbot-full-py2.4"]))
c['schedulers'].append(Nightly(name="python 2.5 acceptance builds",
                               builderNames=["buildbot-acceptance-py2.5"],
                               hour=(7, 12, 17), minute=0)
```

The odds are that you'll have to wait several hours for this to trigger a build. You can test `Nightly` by setting the hours and minutes to times just a minute or two in the future, but don't forget to restore them when you've finished your tests.

What's Left?

You went through a great deal of effort to ensure that a 2.4 builder could easily configured, but I'm not going walk you through the rest of the process. You should know more than enough at this point to set it up, and it's a great exercise. As you do, test each change. Buildbot configuration is a complicated path, and it's easy to get lost unless you keep track of each step.

Summary

FIT is a system for specifying and running functional tests. It consumes requirement documents with which it drives testing fixtures and produces reports. The system focuses upon the requirement documents. In an optimal situation, the customer produces them with the assistance of other team members. This process serves as the basis of detailed design discussions. Along the way, they create a common vocabulary, which can be considered the core of the system metaphor.

The requirement documents can also play the roles of design documents and testing plans. Developers and testers create test fixtures that connect these plans to the larger code base. Once the fixtures are developed, they can be run from the build. The output from these runs serves as a progress document for customers and for management. Watching as the requirements go from red to green over the course of an iteration instills them with confidence.

PyFit is the Python version of FIT. While the FIT framework theoretically copes with any kind of document that contains tables, PyFit only supports HTML. The documents are read from the filesystem or a FitNesse server. FitNesse is a wiki server for writing and running FIT tests. While appealing for very small projects, it doesn't cope well with branching, and disconnected operation isn't really possible. For these reasons, I prefer placing FIT tests into the source.

Tying PyFit into a build is theoretically an easy thing to do, but there are many intricacies. In the system you worked with, the tests run on the build slaves, and the results are packaged into ZIP files there, too. Scripts encapsulate the intricacies of both tasks and hide the details from the build server. The results are presented by the build master using Buildbot's internal web server. To do this, the ZIP file is copied to the build master, and then unpacked into the web server's document tree. This forms the basis for the project's dashboard.