CHAPTER 6

Testing: The Horse and the Cart

This chapter describes unit testing and test-driven development (TDD); it focuses primarily on the infrastructure supporting those practices. I'll expose you to the practices themselves, but only to the extent necessary to appreciate the infrastructure. Along the way, I'll introduce the crudest flavors of agile design, and lead you through the development of a set of acceptance tests for the RSReader application introduced in Chapter 5. This lays the groundwork for Chapter 7, where we'll explore the TDD process and the individual techniques involved.

All of this begs the question, "What are unit tests?" Unit tests verify the behavior of small sections of a program in isolation from the assembled system. Unit tests fall into two broad categories: programmer tests and customer tests. What they test distinguishes them from each other.

Programmer tests prove that the code does what the programmer expects it to do. They verify that the code works. They typically verify behavior of individual methods in isolation, and they peer deeply into the mechanisms of the code. They are used solely by developers, and they are not be confused with customer tests.

Customer tests (a.k.a. acceptance tests) prove that the code behaves as the customer expects. They verify that the code works correctly. They typically verify behavior at the level of classes and complete interfaces. They don't generally specify *how* results are obtained; they instead focus on *what* results are obtained. They are not necessarily written by programmers, and they are used by everyone in the development chain. Developers use them to verify that they are building the right thing, and customers use them to verify that the right thing was built.

In a perfect world, specifications would be received as customer tests. Alas, this doesn't happen often in our imperfect world. Instead, developers are called upon to flesh out the design of the program in conjunction with the customer. Designs are received as only the coarsest of descriptions, and a conversation is carried out, resulting in detailed information that is used to formulate customer tests.

Unit testing can be contrasted with other kinds of testing. Those other kinds fall into the categories of functional testing and performance testing.

Functional testing verifies that the complete application behaves as expected. Functional testing is usually performed by the QA department. In an agile environment, the QA process is directly integrated into the development process. It verifies what the customer sees, and it examines bugs resulting from emergent behaviors, real-life data sets, or long runtimes.

Functional tests are concerned with the internal construction of an application only to the extent that it impinges upon application-level behaviors. Testers don't care if the application was written using an array of drunken monkeys typing on IBM Selectric typewriters run through a bank of badly tuned analog synthesizers before finally being dumped into the source repository. Indeed, some testers might argue that this process would produce better results.

Functional testing falls into four broad categories: exploratory testing, acceptance testing, integration testing, and performance testing. *Exploratory testing* looks for new bugs. It's an inventive and sadistic discipline that requires a creative mindset and deep wells of pessimism. Sometimes it involves testers pounding the application until they find some unanticipated situation that reveals an unnoticed bug. Sometimes it involves locating and reproducing bugs reported from the field. It is an interactive process of discovery that terminates with test cases characterizing the discovered bugs.

Acceptance testing verifies that the program meets the customer's expectations. Acceptance tests are written in conjunction with the customer, with the customer supplying the domain-specific knowledge, and the developers supplying a concrete implementation. In the best cases, they supplant formal requirements, technical design documents, and testing plans. They will be covered in detail in Chapter 11.

Integration testing verifies that the components of the system interact correctly when they are combined. Integration testing is not necessarily an end-to-end test of the application, but instead verifies blocks larger than a single unit. The tools and techniques borrow heavily from both unit testing and acceptance testing, and many tests in both acceptance and unit test suites can often be characterized as integration tests.

Regression testing verifies that bugs previously discovered by exploratory testing have been fixed, or that they have not been reintroduced. The regression tests themselves are the products of exploratory testing. Regression testing is generally automated. The test coverage is extensive, and the whole test suite is run against builds on a frequent basis.

Performance testing is the other broad category of functional testing. It looks at the overall resource utilization of a live system, and it looks at interactions with deployed resources. It's done with a stable system that resembles a production environment as closely as possible.

Performance testing is an umbrella term encompassing three different but closely related kinds of testing. The first is what performance testers themselves refer to as performance testing. The two other kinds are stress testing and load testing. The goal of performance testing is not to find bugs, but to find and eliminate bottlenecks. It also establishes a baseline for future regression testing.

Load testing pushes a system to its limits. Extreme but expected loads are fed to the system. It is made to operate for long periods of time, and performance is observed. Load testing is also called volume testing or endurance testing. The goal is not to break the system, but to see how it responds under extreme conditions.

Stress testing pushes a system beyond its limits. Stress testing seeks to overwhelm the system by feeding it absurdly large tasks or by disabling portions of the system. A 50 GB e-mail attachment may be sent to a system with only 25 GB of storage, or the database may be shut down in the middle of a transaction. There is a method to this madness: ensuring recoverability. Recoverable systems fail and recover gracefully rather than keeling over disastrously. This characteristic is important in online systems.

Sadly, performance testing isn't within this book's scope. Functional testing, and specifically acceptance testing, will be given its due in Chapter 11.

Unit Testing

The focus in this chapter is on programmer tests. From this point forward, I shall use the terms *unit test* and *programmer test* interchangeably. If I need to refer to customer tests, I'll name them explicitly.

So why unit testing? Simply put, unit testing makes your life easier. You'll spend less time debugging and documenting, and it results in better designs. These are broad claims, so I'll spend some time backing them up.

Developers resort to debugging when a bug's location can't be easily deduced. Extensive unit tests exercise components of the system separately. This catches many bugs that would otherwise appear once the lower layers of a system are called by higher layers. The tests rigorously exercise the capabilities of a code module, and at the same time operate at a fine granularity to expose the location of a bug without resorting to a debugger.

This does not mean that debuggers are useless or superfluous, but that they are used less frequently and in fewer situations. Debuggers become an exploratory tool for creating missing unit tests, and for locating integration defects.

Unit tests document intent by specifying a method's inputs and outputs. They specify the exceptional cases and expected behaviors, and they outline how each method interacts with the rest of the system. As long as the tests are kept up to date, they will always match the software they purport to describe. Unlike other forms of documentation, this coherence can be verified through automation.

Perhaps the most far-fetched claim is that unit tests improve software designs. Most programmers can recognize a good design when they see it, although they may not be able to articulate why it is good. What makes a good design? Good designs are highly cohesive and loosely coupled.

Cohesion attempts to measure how tightly focused a software module is. A module in which each function or method focuses on completing part of a single task, and in which the module as a whole performs a single well-defined task on closely related sets of data, is said to be highly cohesive. High cohesion promotes encapsulation, but it often results in high coupling between methods.

Coupling concerns the connections between modules. In a loosely coupled system, there are few interactions between modules, with each depending only on a few other modules. The points where these dependencies are introduced are often explicit. Instead of being hard-coded, objects are passed into methods and functions. This limits the "ripple effect" where changes to one module result in changes to many other modules.

Unit testing improves designs by making the costs of bad design explicit to the programmer as the software is written. Complicated software with low cohesion and tight coupling requires more tests than simple software with high cohesion and loose coupling. Without unit tests, the costs of the poor design are borne by QA, operations, and customers. With unit tests, the costs are borne by the programmers. Unit tests require time and effort to write, and at their best programmers are lazy and proud folk.[1] They don't want to spend time writing needless tests.

1. Laziness is defined by Larry Wall as the quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it.

Unit tests make low cohesion visible through the costs of test setup. Low cohesion increases the number of setup tasks performed in a test. In a functionally cohesive module, it is usually only necessary to set up a few different sets of test conditions. The code to set up such a condition is called a *test fixture*. In a random or functionally cohesive module, many more fixtures are required by comparison. Each fixture is code that must be written, and time and effort that must be expended.

The more dependencies on external modules, the more setup is required for tests, and the more tests must be written. Each different class of inputs has to be tested, and each different class of input is yet another test to be written.

Methods with many inputs frequently have complicated logic, and each path through a method has to be tested. A single execution path mandates one test, and from there it gets worse. Each if-then statement increases the number of tests by two. Complicated loop bodies increase setup costs. The number of classes of output from a method also increases the number of tests to be performed as each kind of value returned and exception raised must be tested.

In a tightly coupled system, individual tests must reference many modules. The test writer expends effort setting up fixtures for each test. Over and over, the programmer confronts the external dependencies. The tests get ugly and the fixtures proliferate. The cost of tight coupling becomes apparent. A simple quantitative analysis shows the difference in testing effort between two designs.

Consider two methods named `get_urls()` that implement the same functionality. One has multiple return types, and the other always returns lists. In the first case, the method can return `None`, a single URL, or a nonempty array of URLs. We'll need at least three tests for this method—one for each distinct return value.

Now consider a method that consumes results from `get_urls()`. I'll call it `get_content(url_list)`. It must be tested with three separate inputs—one for each return type from `get_urls()`. To test this pair of methods, we'll have created six tests.

Contrast this with an implementation of `get_urls()` that returns only the empty array `[]` or a nonempty array of URLs. Testing `get_urls()` requires only two tests.

The associated definition for `get_content(url_list)` is correspondingly smaller, too. It just has to handle arrays, so it only requires one test, which brings the total to three. This is half the number of the first implementation, so it is immediately clear which interface is more complicated. What before seemed like a relatively innocuous choice now seems much less so.

Unit testing works with a programmer's natural proclivities toward laziness, impatience, and pride. It also improves design by facilitating refactoring.

Refactorings alter the structure of the code without altering its function. They are used to improve existing code. They are applied serially, and the unit tests are run after each one. If the behavior of the system has changed in unanticipated ways, then the test suite breaks. Without unit tests, the programmer must take it as an article of faith that the program's behavior is unchanged. This is foolish with your own code, and nearly insane with another's.

The Problems with Not Unit Testing

I make the bald-faced assertion that no programmer completely understands any system of nontrivial complexity. If that programmer existed, then he would produce completely bug-free code. I've yet to see that in practice, but absence of evidence is not evidence of absence, so that person might exist. Instead, I think that programmers understand most of the salient features of their own code, and this is good enough in the real world.

What about working with another programmer's code? While you may understand the salient features of your code, you must often guess at the salient features of another's. Even when she documents her intent, things that were obvious to her may be perplexing to you. You don't have access to her thoughts. The design trade-offs are often opaque. The reasons for putting this method here or splitting out that method there may be historical or related to obscure performance issues. You just don't know for sure. Without unit tests or well-written comments, this can lead to pathological situations.

I've worked on a system where great edifices were constructed around old, baroque code because nobody dared change it. The original authors were gone, and nobody understood those sections of the code base. If the old code broke, then production could be taken down. There was no way to verify that refactorings left the old functionality unaltered, so those sections of code were left unchanged. Scope for projects was narrowly restricted to certain components, even if changes were best made in other components. Refactoring old code was strongly avoided.

It was the opposite of the ideal of collective code ownership, and it was driven by fear of breaking another's code. An executable test harness written by the authors would have verified when changes broke the application. With this facility, we could have updated the code with much less fear. Unit tests are a key to collective code ownership, and the key to confident and successful refactorings.

Code that isn't refactored constantly rots. It accumulates warts. It sprouts methods in inappropriate places. New methods duplicate functionality. The meanings of method and variable names drift, even though the names stay the same. At best, the inappropriate names are amusing, and at worst misleading.

Without refactoring, local bugs don't stay restricted to their neighborhoods. This stems from the layering of code. Code is written in layers. The layers are structural or temporal. Structural layering is reflected in the architecture of the system. Raw device IO calls are invoked from buffered IO calls. The buffered IO calls are built into streams, and applications sip from the streams. Temporal layering is reflected in the times at which features are created. The methods created today are dependent upon the methods that were written earlier. In either case, each layer is built upon the assumption that lower layers function correctly.

The new layers call upon previous layers in new and unusual ways, and these ways uncover existing but undiscovered bugs. These bugs must be fixed, but this frequently means that overlaying code must be modified in turn. This process can continue up through the layers as each in turn must be altered to accommodate the changes below them. The more tightly coupled the components are, the further and wider the changes will ripple through the system. It leads to the effect known as *collateral damage* (a.k.a. *whack-a-mole*), where fixing a bug in one place causes new bugs in another.

Pessimism

There are a variety of reasons that people condemn unit testing or excuse themselves from the practice. Some I've read of, but most I've encountered in the real world, and I recount those here.

One common complaint is that unit tests take too long to write. This implies that the project will take longer to produce if unit tests are written. But in reality, the time spent on unit testing is recouped in savings from other places. Much less time is spent debugging, and much less time is spent in QA. Extensively unit-tested projects have fewer bugs. Consequently, less developer and QA time is spent on repairing broken features, and more time is spent producing new features.

Some developers say that writing tests is not their job. What is a developer's job then? It isn't simply to write code. A developer's job is to produce working and completely debugged code that can be maintained as cheaply as possible. If unit tests are the best means to achieve that goal, then writing unit tests is part of the developer's job.

More than once I've heard a developer say that they can't test the code because they don't know how it's supposed to behave. If you don't know how the code is supposed to behave, then how do you know what the next line should do? If you really don't know what the code is supposed to do, then now probably isn't the best time to be writing it. Time would be better spent understanding what the problem is, and if you're lucky, there may even be a solution that doesn't involve writing code.

Sometimes it is said that unit tests can't be used because the employer won't let unit tests be run against the live system. Those employers are smart. Unit tests are for the development environment. They are the programmer's tools. Functional tests can run against a live system, but they certainly shouldn't be running against a production system.

The cry of "But it compiles!" is sometimes heard. It's hard to believe that it's heard, but it is from time to time. Lots of bad code compiles. Infinite loops compile. Pointless assignments compile. Pretty much every interesting bug comes from code that compiles.

More often, the complaint is made that the tests take too long to run. This has some validity, and there are interesting solutions. Unit tests should be fast. Hundreds should run in a second. Some unit tests take longer, and these can be run less frequently. They can be deferred until check-in, but the official build must always run them.

If the tests still take too long, then it is worth spending development resources on making them go faster. This is an area ripe for improvement. Test runners are still in their infancy, and there is much low-hanging fruit that has yet to be picked.

"We tried and it didn't work" is the complaint with the most validity. There are many individual reasons that unit testing fails, but they all come down to one common cause. The practice fails unless the tests provide more perceived reliability than they cost in maintenance and creation combined. The costs can be measured in effort, frustration, time, or money. People won't maintain the tests if the tests are deemed unreliable, and they won't maintain the tests unless they see the benefits in improved reliability.

Why does unit testing fail? Sometimes people attempt to write comprehensive unit tests for existing code. Creating unit tests for existing code is hard. Existing code is often unsuited to testing. There are large methods with many execution paths. There are a plethora of arguments feeding into functions and a plethora of result classes coming out. As I mentioned when discussing design, these lead to larger numbers of tests, and those tests tend to be more complicated.

Existing code often provides few points where connections to other parts of the system can be severed, and severing these links is critical for reducing test complexity. Without such access points, the subject code must be instrumented in involved and Byzantine ways. Figuring out how to do this is a major part of harnessing existing code. It is often easier just to rewrite the code than to figure out a way to sever these dependencies or instrument the internals of a method.

Tests for existing code are written long after the code is written. The programmer is in a different state of mind, and it takes time and effort to get back to that mental state where the code was written. Details will have been forgotten and must be deduced or rediscovered. It's even worse when someone else wrote the code. The original state of mind is in another's head and completely inaccessible. The intent can only be imperfectly intuited.

There are tools that produce unit tests from finished code, but they have several problems. The tests they produce aren't necessarily simple. They are as opaque, or perhaps more opaque, than the methods being tested. As documentation, they leave something to be desired, as they're not written with the intent to inform the reader. Even worse, they will falsely ensure the validity of broken code. Consider this code fragment:

```
a = a + y
a = a + y
```

The statement is clearly duplicated. This code is probably wrong, but currently many generators will produce a unit test that validates it.

An effort focused on unit testing unmodified existing code is likely to fail. Unit testing's big benefits accrue when writing new code. Efforts are more likely to succeed when they focus on adding unit tests for sections of code as they change.

Sometimes failure extends from a limited suite of unit tests. A test suite may be limited in both extent and execution frequency. If so, bugs will slip through and the tests will lose much of their value. In this context, *extent* refers to coverage within a tested section. Testing coverage should be as complete as possible where unit tests are used. Tested areas with sparse coverage leak bugs, and this engenders distrust.

When fixing problems, all locations evidencing new bugs must be unit tested. Every mole that pops out of its hole must be whacked. Fixing the whack-a-mole problem is a major benefit that developers can see. If the mole holes aren't packed shut, the moles will pop out again, so each bug fix should include an associated unit test to prevent its regression in future modifications.

Failure to properly fix broken unit tests is at the root of many testing effort failures. Broken tests must be fixed, not disabled or gutted.[2] If the test is failing because the associated functionality has been removed, then gutting a unit test is acceptable; but gutting because you don't want to expend the effort to fix it robs tests of their effectiveness. There was clearly a bug, and it has been ignored. The bug will come back, and someone will have to track it down again. The lesson often taken home is that unit tests have failed to catch a bug.

Why do people gut unit tests? There are situations in which it can reasonably be done, but they are all tantamount to admitting failure and falling back to a position where the testing effort can regroup. In other cases, it is a social problem. Simply put, it is socially acceptable in the development organization to do this. The way to solve the problem is by bringing social pressures to bear.

Sometimes the testing effort fails because the test suite isn't run often enough, or it's not run automatically. Much of unit testing's utility comes through finding bugs immediately after they are introduced. The longer the time between a change and its effect, the harder it is to associate the two. If the tests are not run automatically, then they won't be run much of the

2. A test is gutted when its body is removed, leaving a stub that does nothing.

time, as people have a natural inclination not to spend effort on something that repeatedly produces nonresults or isn't seen to have immediate benefits.

Unit tests that run only on the developer's system or the build system lead toward failure. Developers must be able to run the tests at will on their own development boxes, and the build system must be able to run them in the official clean build environment. If developers can't run the unit tests on their local systems, then they will have difficulty writing the tests. If the build system can't run the tests, then the build system can't enforce development policies.

When used correctly, unit test failures should indicate that the code is broken. If unit test failures do not carry this meaning, then they will not be maintained. This meaning is enforced through build failures. The build must succeed only when all unit tests pass. If this cannot be counted on, then it is a severe strike against a successful unit-testing effort.

Test-Driven Development

As noted previously, a unit-testing effort will fail unless the tests provide more perceived reliability than the combined costs of maintenance and creation. There are two clear ways to ensure this. Perceived utility can be increased, or the costs of maintenance and creation can be decreased. The practices of TDD address both.

TDD is a style with unique characteristics. Perhaps most glaringly, tests are written before the tested code. The first time you encounter this, it takes a while to wrap your mind around it. "How can I do that?" was my first thought, but upon reflection, it is obvious that you always know what the next line of code is going to do. You can't write it until you know what it is going to do. The trick is to put that expectation into test code before writing the code that fulfills it.

TDD uses very small development cycles. Tests aren't written for entire functions. They are written incrementally as the functions are composed. If the chunks get too large, a test-driven developer can always back down to a smaller chunk.

The cycles have a distinct four-part rhythm. A test is written, and then it is executed to verify that it fails. A test that succeeds at this point tells you nothing about your new code. (Every day I encounter one that works when I don't expect it to.) After the test fails, the associated code is written, and then the test is run again. This time it should pass. If it passes, then the process begins anew.

The tests themselves determine what you write. You only write enough code to pass the test, and the code you write should always be the simplest possible thing that makes the test succeed. Frequently this will be a constant. When you do this religiously, little superfluous functionality results.

No code is allowed to go into production unless it has associated tests. This rule isn't as onerous as it sounds. If you follow the previously listed practices then this happens naturally.

The tests are run automatically. In the developer's environment, the tests you run may be limited to those that execute with lightning speed (i.e., most tests). When you perform a full build, all tests are executed. This happens in both the developer's environment and the official build environment. A full build is not considered successful unless all unit tests succeed.

The official build runs automatically when new code is available. You've already seen how this is done with Buildbot, and I'll expand the configuration developed in Chapter 5 to include running tests. The force of public humiliation is often harnessed to ensure compliance. Failed builds are widely reported, and the results are highly visible. You often accomplish this through mailing lists, or a visible device such as a warning light or lava lamp.

Local test execution can also be automated. This is done through two possible mechanisms. A custom process that watches the source tree is one such option, and another uses the IDE itself, configuring it to run tests when the project changes.

The code is constantly refactored. When simple implementations aren't sufficient, you replace them. As you create additional functionality, you slot it into dummied implementations. Whenever you encounter duplicate functionality, you remove it. Whenever you encounter code smells, the offending stink is freshened.

These practices interact to eliminate many of the problems encountered with unit testing. They speed up unit testing and improve the tests' accuracy. The tests for the code are written at the same time the code is written. There are no personnel or temporal gaps between the code and the tests. The tests' coverage is exhaustive, as no code is produced without an associated set of tests. The tests don't go stale, as they are invoked automatically, and the build fails if any tests fail. The automatic builds ensure that bugs are found very soon after they are introduced, vastly improving the suite's value.

The tests are delivered with the finished system. They provide documentation of the system's components. Unlike written documents, the tests are verifiable, they're accurate, and they don't fall out of sync with the code. Since the tests are the primary documentation source, as much effort is placed into their construction as is placed into the primary application.

Knowing Your Unit Tests

A unit test must assert success or failure. Python provides a ready-made command. The Python `assert` expression takes one argument: a Boolean expression. It raises an `AssertionErrror` if the expression is `False`. If it is `True`, then the execution continues on. The following code shows a simple assertion:

```
>>> a = 2
>>> assert a == 2
>>> assert a == 3
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

You clarify the test by creating a more specialized assertion:

```
>>> def assertEquals(x, y):
... assert x == y
...
>>> a = 2
>>> assertEquals(a, 2)
>>> assertEquals(a, 3)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in assertEquals
AssertionError
```

Unit tests follow a very formulaic structure. The test conditions are prepared, and any needed fixtures are created. The subject call is performed, the behavior is verified, and finally the test fixtures are cleanly destroyed. A test might look like this:

```
def testSettingEmployeeNameShouldWork():
     x = create_persistent_employee()
     x.set_name("bob")
     assertEquals("bob", x.get_name)
     x.destroy_self()
```

The next question is where the unit tests should go. There are two reasonable choices: the tests can be placed with the code they test or in an isolated package. I personally prefer the former, but the latter has performance advantages and organizational benefits. The tools to run unit tests often search directories for test packages. For large projects, this overhead causes delays, and I'd rather sidestep the issue to begin with.

unittest and Nose

There are several packages for unit testing with Python. They all support the four-part test structure described previously, and they all provide a standard set of features. They all group tests, run tests, and report test results. Surprisingly, test running is the most distinctive feature among the Python unit-testing frameworks.

There are two clear winners in the Python unit-testing world: unittest and Nose. unittest ships with Python, and Nose is a third-party package. Pydev provides support for unittest, but not for Nose. Nose, on the other hand, is a far better test runner than unittest, and it understands how to run the other's test cases.

Like Java's jUnit test framework, unittest is based upon Smalltalk's xUnit. Detailed information on its development and design can be found in Kent Beck's book *Test-Driven Development: By Example* (Addison-Wesley, 2002).

Tests are grouped into `TestCase` classes, modules (files), and `TestSuite` classes. The tests are methods within these classes, and the method names identify them as tests. If a method name begins with the string `test`, then it is a test—so `testy`, `testicular`, and `testosterone` are all valid test methods. Test fixtures are set up and torn down at the level of `TestCase` classes. `TestCase` classes can be aggregated with `TestSuite` classes, and the resulting suites can be further aggregated. Both `TestCase` and `TestSuite` classes are instantiated and executed by `TestRunner` objects. Implicit in all of this are modules, which are the Python files containing the tests. I never create `TestSuite` classes, and instead rely on the implicit grouping within a file.

Pydev knows how to execute unittest test objects, and any Python file can be treated as a unit test. Test discovery and execution are unittest's big failings. It is possible to build up a giant unit test suite, tying together `TestSuite` after `TestSuite`, but this is time-consuming. An easier approach depends upon file-naming conventions and directory crawling. Despite these deficiencies, I'll be using unittest for the first few examples. It's very widely used, and familiarity with its architecture will carry over to other languages.[3]

3. Notably, it carries over to JavaScript testing with JSUnit in Chapter 10.

Nose is based on an earlier package named PyTest. Nose bills itself primarily as a test discovery and execution framework. It searches directory trees for modules that look like tests. It determines what is and is not a test module by applying a regular expression (`r'(?:^|[\\b_\\.%s-])[Tt]est' % os.sep`) to the file name. If the string `[Tt]est` is found after a word boundary, then the file is treated as a test.[4] Nose recognizes `unittest.TestCase` classes, and knows how to run and interpret their results. `TestCase` classes are identified by type rather than by a naming convention.

Nose's native tests are functions within modules, and they are identified by name using the same pattern used to recognize files. Nose provides fixture setup and tear-down at both the module level and function level. It has a plug-in architecture, and many features of the core package are implemented as plug-ins.

A Simple RSS Reader

The project introduced in Chapter 4 is a simple command-line RSS reader (a.k.a. aggregator). As noted, RSS is a way of distributing content that is frequently updated. Examples include new articles, blog postings, podcasts, build results, and comic strips. A single source is referred to as a feed. An aggregator is a program that pulls down one or more RSS feeds and interleaves them. The one constructed here will be very simple. The two feeds we'll be using are from two of my favorite comic strips: xkcd and PVPonline.

RSS feeds are XML documents. There are actually three closely related standards: RSS, RSS 2.0, and Atom. They're more alike than different, but they're all slightly incompatible. In all three cases, the feeds are composed of dated items. Each item designates a chunk of content. Feed locations are specified with URLs, and the documents are typically retrieved over HTTP.

You could write software to retrieve an RSS feed and parse it, but others have already done that work. The well-recognized package FeedParser is one. It is retrieved with `easy_install`:

```
$ easy_install FeedParser
```

```
Searching for FeedParser
Reading http://pypi.python.org/simple/FeedParser/
Best match: feedparser 4.1
...
Processing dependencies for FeedParser
Finished processing dependencies for FeedParser
```

The package parses RSS feeds through several means. They can be retrieved and read remotely through a URL, and they can be read from an open Python file object, a local file name, or a raw XML document that can be passed in as a string. The parsed feed appears as a queryable data structure with a `dict`-like interface:

4. The default test pattern recognizes `Test.py`, `Testerosa.py`, `a_test.py`, and `testosterone.py`, but not `CamelCaseTest.py` or `mistested.py`. You can set the pattern with the `-m` option.

```
>>> import feedparser
>>> d = feedparser.parse('http://www.xkcd.com/rss.xml')
>>> print d['feed']['title']
```

```
xkcd.com
```

```
>>> print len(d['items'])
```

```
2
```

```
>>> print [x['title'] for x in d['items']]
```

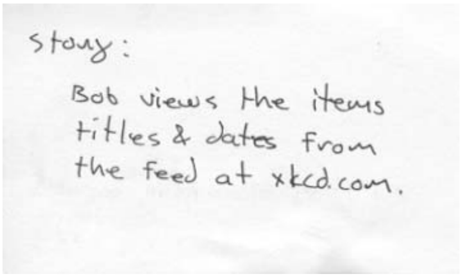
```
[u'Python', u'Far Away']
```

```
>>> print [x['date'] for x in d['items']]
```

```
[u'Wed, 05 Dec 2007 05:00:00 -0000', u'Mon, 03 Dec 2007➥
05:00:00 -0000']
```

The project is ill defined at this point, so I'm going to describe it a bit more concretely. We'll start simply and add more features as the project develops. For now, I just want to know if a new comic strip is available when I log in. (I find it really depressing to get the Asia Times feed in the morning, and comics make me happy.)

Let's make a story. User stories describe new features. They take the place of large requirements documents. They are only two or three sentences long and have just enough detail for a developer to make a ballpark estimate of how long it will take to implement. They're initially created by the customer, they're devoid of technical mumbo jumbo, and they're typically jotted down on a note card, as in Figure 6-1.



Story:

Bob views the items titles & dates from the feed at xkcd.com.

Figure 6-1. *A user story on a 3 × 5 notecard*

Developers go back to the customer when work begins on the story. Further details are hashed out between the two of them, ensuring that the developer really understands what the customer wants, with no intermediate document separating their perceptions. This discussion's outcomes drive acceptance test creation. The acceptance tests document the discussion's conclusions in a verifiable way.

In this case, I'm both the customer and the programmer. After a lengthy discussion with myself, I decide that I want to run the command with a single URL or a file name and have it output a list of articles. The user story shown on the card in Figure 6-1 reads, "Bob views the titles & dates from the feed at xkcd.com." After hashing things out with the customer, it turns out that he expects a run to look something like this:

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

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

I ask the customer (me), "What should this look like when I don't supply any arguments?" And the customer says, "Well, I expect it to do nothing."

And the developer (me) asks, "And if it encounters errors?"

"Well, I really don't care about that. I'm a Python programmer. I'll deal with the exceptions," replies the customer, "and for that matter, I don't care if I even see the errors."

"OK, what if more than one URL is supplied?"

"You can just ignore that for the moment."

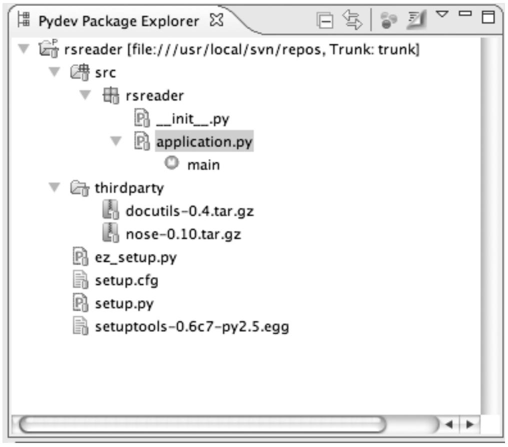
"Cool. Sounds like I've got enough to go on," and remembering that maintaining good relations with the customer is important, I ask, "How about grabbing a bite for lunch at China Garlic?"

"Great idea," the customer replies.

We now have material for a few acceptance tests. The morning's work is done, and I go to lunch with myself and we both have a beer.

The First Tests

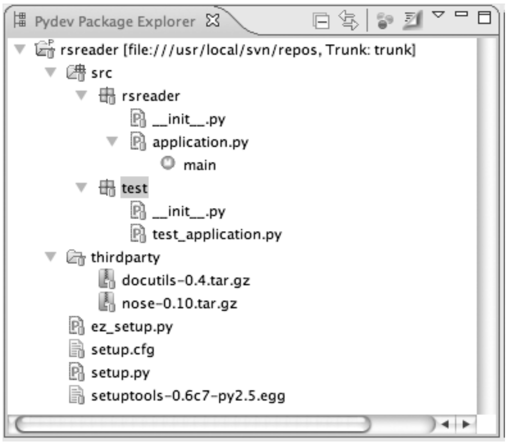
In the previous chapter, you wrote a tiny fragment of code for your application. It's a stub method that prints "woof." It exists solely to allow Setuptools to install an application. The project (as seen from Eclipse) is shown in Figure 6-2.



```
rsreader [file:///usr/local/svn/repos, Trunk: trunk]
  src
    rsreader
      __init__.py
      application.py
        main
  thirdparty
    docutils-0.4.tar.gz
    nose-0.10.tar.gz
  ez_setup.py
  setup.cfg
  setup.py
  setuptools-0.6c7-py2.5.egg
```

Figure 6-2. *RSReader as last visited*

Instead of intermixing test code and application code, the test code is placed into a separate package hierarchy. The package is `test`, and there is also a test module called `test.test_application.py`. This can be done from the command line or from Eclipse. The added files and directories are shown in Figure 6-3.



```
rsreader [file:///usr/local/svn/repos, Trunk: trunk]
  src
    rsreader
      __init__.py
      application.py
        main
    test
      __init__.py
      test_application.py
  thirdparty
    docutils-0.4.tar.gz
    nose-0.10.tar.gz
  ez_setup.py
  setup.cfg
  setup.py
  setuptools-0.6c7-py2.5.egg
```

Figure 6-3. *RSReader with the unit test skeleton added*

RSReader takes in data from URLs or files. The acceptance tests shouldn't depend on external resources, so the first acceptance tests should read from a file. They will expect a specific output, and this output will be hard-coded. The method `rsreader.application.main()` is the application entry point defined in `setup.py`. You need to see what a failing test looks like before you can appreciate a successful one, so the first test case initially calls `self.fail()`:

```
from unittest import TestCase

class AcceptanceTests(TestCase):

    def test_should_get_one_URL_and_print_output(self):
        self.fail()
```

The test is run through the Eclipse menus. The test module is selected from the Package Explorer pane, or the appropriate editor is selected. With the focus on the module, the Run menu is selected from either the application menu or the context menu. From the application menu, the option is Run ➤ Run As ➤ "Python unit-test," and from the context menu, it is Run As ➤ "Python unit-test." Once run, the console window will report the following:

```
Finding files... ['/Users/jeff/workspace/rsreader/src/test/test_application.py']➥
 ... done
Importing test modules ... done.

test_should_get_one_URL_and_print_output➥
 (test_application.AcceptanceTests) ... FAIL
```

```
======================================================================
FAIL: testShouldGetOneURLAndPrintOutput (test_application.AcceptanceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/workspace/rsreader/src/test/test_application.py",➥
 line 6, in testShouldGetOneURLAndPrintOutput
    self.fail()
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.000s
```

The output shows that one test was run, and it took less than 1 ms. As expected, the test failed.

This example starts with a bang; a very complicated bang. Feeding input into the program and reading the output is the most complicated thing done in this chapter. The print statement writes to `sys.stdout`. The test should capture `sys.stdout`, and then compare the output with the expectations.

`sys.stdout` contains a file-like object. The test replaces this object with a `StringIO` instance. `StringIO` is a file-like object that accumulates written information in a string. This string's value can be extracted and compared with the expected value.

Care must be taken when doing this. If the old value of `sys.stdout` is not restored, then it will be lost, and no more output will be reported. Instead of going to the console, the output will accumulate in the inaccessible `StringIO` object. A first pass looks something like this:

```
import StringIO
import sys
from unittest import TestCase

from rsreader.application import main

class AcceptanceTests(TestCase):

    def test_should_get_one_URL_and_print_output(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"""

        old_value_of_stdout = sys.stdout
        try:
            sys.stdout = StringIO.StringIO()
            main()
            self.assertEquals(printed_items + "\n",
                              sys.stdout.getvalue())
        finally:
            sys.stdout = old_value_of_stdout
```

The core statements of the test are in bold. When run, this test fails as expected. The important line of output reads as follows:

```
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' !=➥
 'woof\n'
```

As hoped, the `printed_items` list does not match the recorded output. The test shows that the output, `woof`, was indeed captured, though. The most questionable part of the test mechanics has been checked.

The test isn't complete, though. The URL needs to be passed in through `sys.argv`. `sys.argv` is a list, and the first argument of the list is always the name of the program—that's just how it works. The single URL will be the second element in the list. `sys.argv` is also a global variable, so it needs the same treatment as `sys.stdout`:

```
class AcceptanceTests(TestCase):

    def test_should_get_one_URL_and_print_output(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"""

        old_value_of_stdout = sys.stdout
        old_value_of_argv = sys.argv
        try:
            sys.stdout = StringIO.StringIO()
            sys.argv = ["unused_prog_name", "xkcd.rss.xml"]
            main()
            self.assertEquals(printed_items + "\n",
                              sys.stdout.getvalue())
        finally:
            sys.stdout = old_value_of_stdout
            sys.argv = old_value_of_argv
```

Running the method shows the same results as before—the test fails with an equality mismatch.

The test method is complete. Now, what is the simplest possible way to make the code pass the test? The simplest way is by faking the results. The new code is shown in bold.

```
def main():
    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"""
    print xkcd_items
```

This change is saved, and the test case is run again:

```
Finding files... ['/Users/jeff/Documents/ws/rsreader/src/test/➥
test_application.py'] ... done
Importing test modules ... done.

test_should_get_one_URL_and_print_output➥
 (test_application.AcceptanceTests) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The test case has become pretty gruesome. Fixtures are set up, and if the setup succeeds, then they must be torn down afterward. This is part of the standard pattern described earlier, though, and unittest addresses these situations. It provides a mechanism to remove this code from the test case. This uses the magical `setUp(self)` and `tearDown(self)` methods. If defined, they are called at the beginning and end of every unit test. `TearDown()` will only be skipped under one condition, and that is when `setUp()` is defined yet fails. In that case, the entire test is abandoned.

I'll demonstrate the refactoring in two steps. First, the `sys.stdout` code will be moved into the `setUp()` and `tearDown()` methods, and then the `sys.argv` code will be moved to them:

```
class AcceptanceTests(TestCase):

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

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

    def test_should_get_one_URL_and_print_output(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"""

        old_value_of_argv = sys.argv
        try:
            sys.argv = ["unused_prog_name", "xkcd.rss.xml"]
            main()
            self.assertEquals(printed_items + "\n",
                              sys.stdout.getvalue())
        finally:
            sys.argv = old_value_of_argv
```

Running this test succeeds. With the assurance that nothing is broken, the second refactoring is performed:

```
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):
        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"""

        sys.argv = ["unused_prog_name", "xkcd.rss.xml"]
        main()
        self.assertEquals(printed_items + "\n", sys.stdout.getvalue())
```

Running the test again demonstrates that nothing has changed. The test still passes, and the test is notably cleaner. The `try` block has been removed, and the test method retains only code related to the test itself.

The next test focuses on empty input. Casting back to the use case discussion, there should be no output when there are no URLs or files specified. The test for that condition is quite compact:

```
    def test_no_urls_should_print_nothing(self):
        sys.argv = ["unused_prog_name"]
        main()
        self.assertEquals("", sys.stdout.getvalue())
```

Running the test produces the following output:

```
Importing test modules ... done.

test_no_urls_should_print_nothing➥
(test_application.AcceptanceTests) ... FAIL
test_should_get_one_URL_and_print_output➥
 (test_application.AcceptanceTests) ... ok

======================================================================
FAIL: test_no_urls_should_print_nothing (test_application.AcceptanceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/Documents/ws/rsreader/src/test/test_application.py",➥
line 30, in test_no_urls_should_print_nothing
    self.assertEquals("\n", sys.stdout.getvalue())
AssertionError: '\n' != 'Wed, 05 Dec 2007 05:00:00 -0000:➥
xkcd.com: Python\nMon, 03 Dec 2007 05:00:00 -0000: xkcd.com:➥
 Far Away\n'

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

Summaries for both tests are shown. It is clear that the new test failed, while the old test succeeded. The new test failed because the hard-coded output is a constant. The main routine needs to be changed. It should print nothing when no user arguments are passed. `main()` only needs to distinguish between two options, so we fake it a little more:

```
def main():
    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"""
    if len(sys.argv) == 2:
        print xkcd_items
```

The tests are run, and the second test now passes. There is a third acceptance test that was discussed. In that case, more than two feeds are passed in, but only the first is reported. The new test case reads as follows:

```
    def test_many_urls_should_print_first_results(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"""

        sys.argv = ["unused_prog_name", "xkcd.rss.xml", "excess"]
        main()
        self.assertEquals(printed_items + "\n", sys.stdout.getvalue())
```

The test is run, and it fails. It fails because the `main()` method explicitly checks for a length of 2. In all other cases, it prints nothing. This is corrected by checking for any nonempty user argument list:

```
def main():
    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"""
    if sys.argv[1:]:
        print xkcd_items
```

With this change, all three acceptance tests pass. There is now a solid framework for writing the rest of the application. At this point, the application can be installed with `python ./setup.py install`, or the local installation can be put into development mode with `python ./setup.py develop`, and the application runs. This can be verified from the command line. It's running in a brain-dead, bogus, dirt-simple way, but it can be refined into useful functionality from there.

```
$ rsreader
$ rsreader xkcd.rss
```

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

```
$ rsreader xkcd.rss something.useless
```

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

There are still things to clean up. Tests will have to be rewritten in the future, so they must be maintainable. Tests serve as documentation, too, so they must also be readable. They obey the same rules as the application code, and if refactoring is neglected, then the tests will rot.

There are two duplications within the tests. The constant `printed_items` can be lifted out of the first and third tests, and the lines comparing the captured `sys.stdout` can be extracted into a new method, which I'll call `assertStdoutEquals`. After these refactorings, the tests look like this:

```
class AcceptanceTests(TestCase):

    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 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(self.printed_items + "\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(self.printed_items + "\n")

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

These tests are run, they still execute, and the application still runs. The smell of duplication has been removed, but there are still things to be done before work can begin on the rest of the application.

Finding Tests with Nose

Running tests manually within Eclipse is fine for a brief tutorial, but it doesn't cut it for day-to-day work. Triggering the tests takes time and attention, and it breaks flow. This is a no-no. Tests should run automatically in the local development environment, and they must run automatically in the official build environment.

Running the unit tests cannot be automated unless it can be done from the command line. unittest does this very poorly, but that's all right. As noted earlier in this chapter, the Nose test package is much better at doing this.

The Nose test runner is `nosetests`. By default, it searches for tests starting in the current working directory. It identifies tests by name, and it only searches for them in valid Python packages. The search begins in a directory, and it extends recursively into any subdirectories that contain `__init__.py` files. Nose runs unittest `TestCase` classes, so we can use it to run the current acceptance tests from the command line. The next few lines are executed from the project root:

```
$ nosetests
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.048s

OK
```

Other directories are specified using the `-w` switch. This is particularly useful when working on a large project where searching for packages consumes a noticeable amount of time, or when for one reason or another, Nose is being run from a working directory that is not the project root.

```
$ nosetests -w src/test
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.048s

OK
```

By default, `nosetests` is quiet. It prints one dot for each successful test. An `F` is printed if a test assertion fails, and an `E` is printed if a test has an error that prevents it from running to completion. Stack traces are printed when a test fails or errors out. Developers quickly acquire an addiction to watching the little dots stream across the page.

Nose is made more vociferous using the `-v` switch. Instead of printing a string of dots, it prints one line for each test. It prints the test name and test module, or the doc string followed by a status that may be one of `ok`, `fail`, or `error`.

```
$ nosetests -w src/test -v
```

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

----------------------------------------------------------------------
Ran 3 tests in 0.002s
```

Nose also intercepts `stdout` and `stderr`, which sometimes isn't desired. Sometimes you'll need to see messages propagated by other modules, and other times you'll want to watch debugging messages you've inserted. At times like these, you can turn off output capture with the `-s` switch.

Skipping Slow Tests

The majority of tests in a test suite will run in a matter of seconds, but a small minority will take seconds or tens of seconds. This is far too long for the development environment. As long as the continuous build system takes up the slack, and as long as code is committed regularly, it is usually a win to skip these slow tests for most runs.

Nose provides the facility to do this through attribute tags. *Attribute Tags* are bits of metadata attached to test methods. In actuality, they're just additional attributes. I'll add the following test to demonstrate the feature:

```
def test_simulates_performing_a_timeout(self):
    import time
    time.sleep(5)
```

Running `nosetests` shows that the test runs, and it is clear that it significantly lengthens the suite's runtime:

```
$ nosetests -w src/test
```

```
....
----------------------------------------------------------------------
Ran 4 tests in 5.004s

OK
```

The next code snippet shows the slow test with a tag attached:

```
def test_simulates_performing_a_timeout(self):
    import time
    time.sleep(5)
test_simulates_performing_a_timeout.slow = True
```

`nosetests`'s `-a` option gives the tags meaning. It runs all tests matching an expression:

```
$ nosetests -w src/test -a slow
```

```
.
----------------------------------------------------------------------
Ran 1 test in 5.003s

OK
```

This is exactly the opposite of what you wanted. The option can be negated by prefixing it with an exclamation point. Most shells attach meaning to `!`, so it must be escaped with a backslash:

```
$ nosetests -w src/test -a \!slow
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

This is the desired result. Using attribute tags, the slow tests are designated, and they are skipped with the `-a` option, but the specification is pretty ugly. Fortunately, Python's decorators come to the rescue.

Decorators were introduced in Python 2.4. They are functions that (in the simplest case) take a function as input and return a function as output. They are used to modify existing methods. The most familiar usage specifies class methods. The following code defines a decorator that sets the `slow` attribute to `True`:

```
def slow(f):
    f.slow = True
    return f
```

Using this decorator, the tagged test becomes

```
@slow
def test_simulates_performing_a_timeout(self):
    import time
    time.sleep(5)
```

I find this definition much clearer, and running the previous Nose command proves that it works.

The `-a` option is implemented through a standard Nose plug-in named `attrib`. It has more features than I've demonstrated so far. In addition to testing a tag's existence, the tag's value may be checked for equality or inequality. The tag's value may also be a sequence. In this case, the tag expression checks for membership. This behavior is summarized in Table 6-1.

Table 6-1. *Tag Expressions*

Tag Option	Tag on Class	Executed by Tests?
`-a slow`	`test.slow = True`	Yes
`-a slow`	`test.slow = False`	No
`-a slow`	`test`	No
`-a \!slow`	`test.slow = True`	No
`-a \!slow`	`test.slow = False`	Yes
`-a \!slow`	`test`	Yes
`-a slow=a`	`test.slow="a"`	Yes
`-a slow=a`	`test.slow="b"`	No
`-a slow=a`	`test`	No
`-a slow=a`	`test.slow = ["a", "b"]`	Yes
`-a slow=a`	`test.slow = ["b"]`	No
`-a slow=\!a`	`test.slow = ["a"]`	No
`-a slow=\!a`	`test.slow="a"`	No
`-a slow\!=a`	`test.slow="a"`	No
`-a slow=\!a`	`test`	Yes

Integrating the Tests into the Environment

There are three times unit tests need to be run. First, they need to be run when changes are saved in the development environment. The test run should be fast, so this set of tests is restricted to those that run in a matter of seconds. Sometimes this means restricting them to a local area of the project, but often it means simply skipping those that take too long to run.

The full unit test suite should be run before changes are committed. This ensures that the code submitted works completely in the developer's environment. It catches bugs that show up in slow tests, and prevents them from reaching the shared codeline. It gives developers confidence that their changes won't break the build.

Finally, the official build system must run the complete unit test suite as part of every build, and the build must fail if any unit tests fail. This ensures that all tests pass in a clean environment. This also provides a mechanism to police unit-testing policy. Purely social pressures help, but the humiliation of a broken build is the big stick.

Running Tests After Every Change

Eclipse contains a mechanism intended to produce incremental builds. Eclipse activates builders when projects change. *Builders* take a list of changes since their last invocation, and then perform an update task. This may be an extension to Eclipse or a program run external to the IDE. The builder mechanism is a hook to run the unit tests after every change.

Additional builders are defined from the project properties menu. The project properties are accessed through the application menu or the context menu when the project is selected in the Package Explorer. From the application menu, the menu item is Project ➤ Properties. From the context menu, the menu item is Properties. This opens the window shown in Figure 6-4.

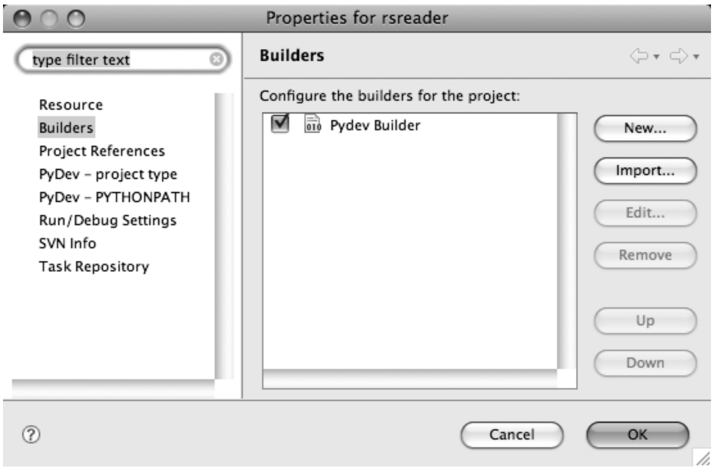


Figure 6-4. *The project properties window*

The Builders menu item is selected from the menu on the left, which brings up the panel shown. Clicking the New button brings up the window shown in Figure 6-5, from which the kind of builder is chosen.

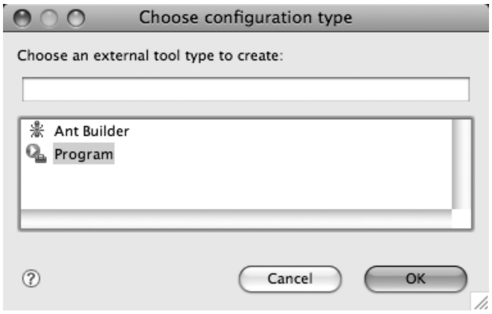
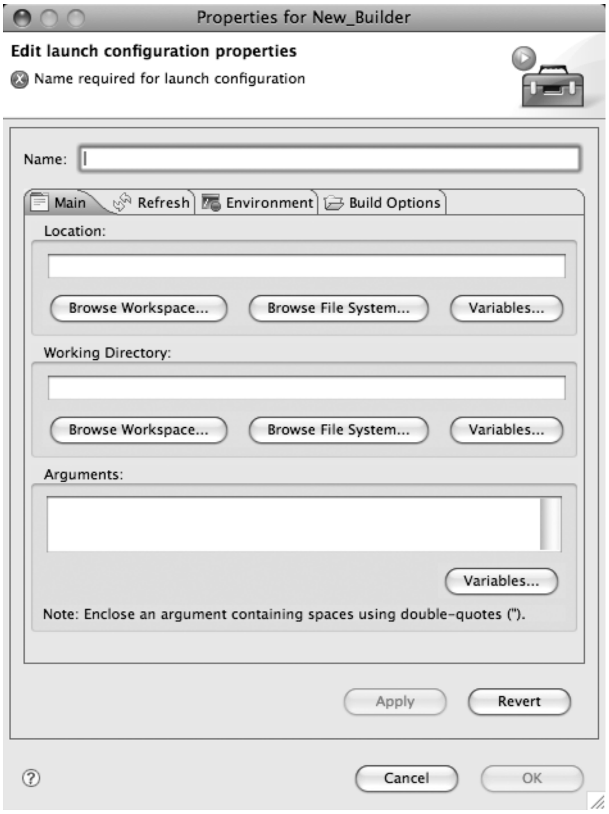

Choose configuration type
Choose an external tool type to create:
Ant Builder
Program
Cancel
OK

Figure 6-5. *Choosing the kind of external builder to create*

Out of the box, Eclipse offers two choices: Ant Builder and Program. Ant Builder calls Java's Ant build tool, and it understands how to parse the output, giving much more interesting output. I personally work on one mostly Python project that uses Ant as a build harness for historical reasons, but I prefer to use Setuptools when given the chance.

Ant is not being used in this project, though. Instead, tests are run through Nose, so the generic Program option is the correct choice. Clicking the OK button brings up the builder properties window, shown in Figure 6-6.



Name required for launch configuration

Name:

Note: Enclose an argument containing spaces using double-quotes (").

Figure 6-6. *The builder properties window*

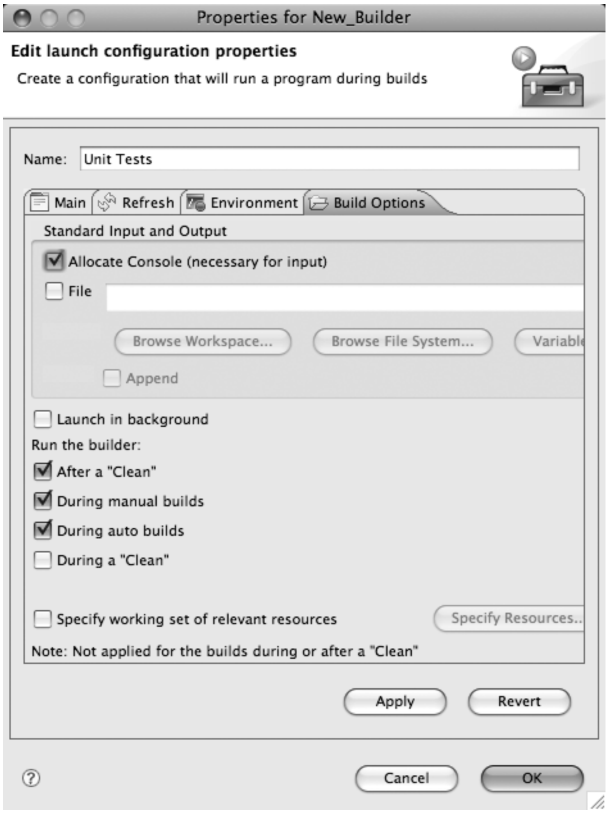
The builder requires a name, so I'll call it Unit Tests. This name is for human consumption, and it has no significance to the IDE. In the Main tab, there are three fields to be filled in:

Location: This is the path to the `nosetests` binary. On UNIX systems, this can be found by executing the command `which nosetests`. On my Mac OS X system, the path is `/Users/jeff/bin/nosetests`, as specified by my `~/.pydistutils.cfg` file.

Working Directory: This is the top-level project directory. This is specified using the Eclipse variable expression `${workspace_loc:/rsreader}`. Using the Browse Workspace button to select this directory gives the same results.

Arguments: In this field, four options are passed: `-w src/test -v -s -a \!slow`. The option `-w src/test` specifies that Nose should only look for tests in the test directory. The option `-v` yields verbose output, showing all the tests, and the `-s` option ensures that any interesting output is sent to the console. The `-a \!slow` option specifies that only fast tests will be run (i.e., tests not marked as `slow`).

By default, the builder is only invoked after a clean build and during manual builds. For this purpose, it should be run when autobuilds are triggered, which happens after any change to the project. This setting is changed on the Build Options tab, which is shown in Figure 6-7.



Properties for New_Builder

Edit launch configuration properties

Create a configuration that will run a program during builds

Name: Unit Tests

Standard Input and Output

- [✓] Allocate Console (necessary for input)
- [ ] File
- [ ] Append
- [ ] Launch in background

Run the builder:

- [✓] After a "Clean"
- [✓] During manual builds
- [✓] During auto builds
- [ ] During a "Clean"
- [ ] Specify working set of relevant resources

Note: Not applied for the builds during or after a "Clean"

Figure 6-7. *The Build Options tab*

In Figure 6-7, the "During auto builds" check box has already been selected. It is the last selected check box in the window. If the unit tests take too long to execute, or you find that they interrupt your flow too much, then they may be backgrounded by default. This option is just below the console settings at the top. A console should always be allocated so that the results are clearly visible, and this is also the default.

With this change, the builder definition is complete. Clicking the OK button closes the builder properties window, and the focus returns to the project properties window shown in Figure 6-8.

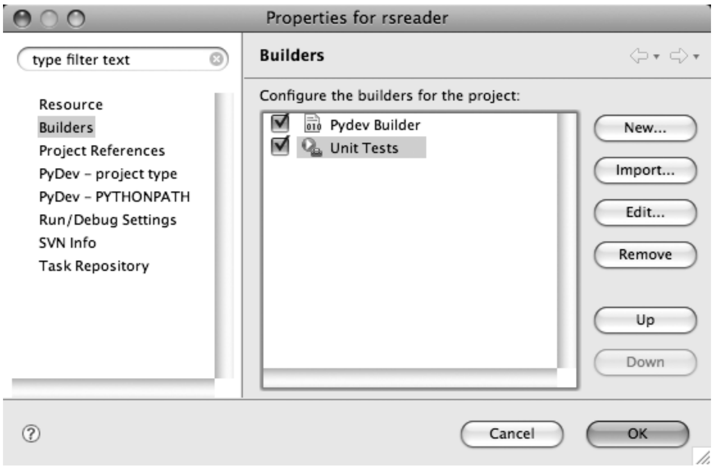
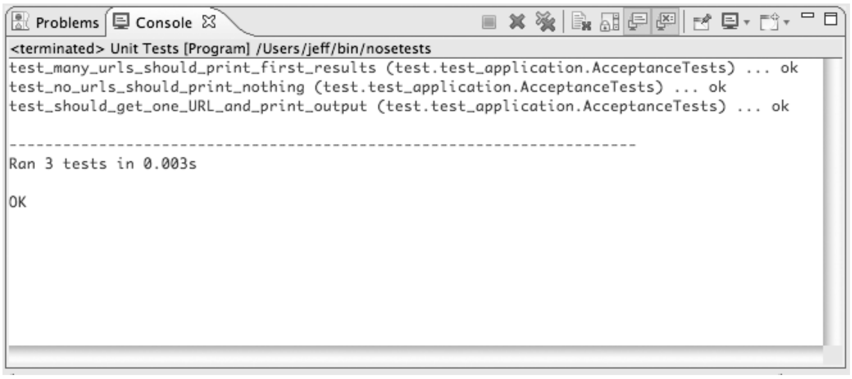


Figure 6-8. *The Unit Tests builder is defined and active.*

The Unit Tests builder is now available in the Builders list, and the check box to its left indicates that it is active. The order in the window determines when each builder is invoked, with the first invoked at the top and the last invoked at the bottom. You can reorder the list by selecting a builder and using the Up and Down buttons to change its position in the list.

Clicking OK saves the changes. This constitutes a change in the project, so the new builder launches immediately, and the test output is shown in the console, as in Figure 6-9.



```
<terminated> Unit Tests [Program] /Users/jeff/bin/nosetests
test_many_urls_should_print_first_results (test.test_application.AcceptanceTests) ... ok
test_no_urls_should_print_nothing (test.test_application.AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output (test.test_application.AcceptanceTests) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK
```

Figure 6-9. *The console showing the Unit Tests builder output*

Running the Complete Test Suite in Development

The complete test suite should be run before changes are committed. This is done through the build harness. This complete test suite check can be triggered from the command line or through Eclipse on demand. The changes to the harness are minimal and restricted to one line in `setup.py`.

Setuptools supports running unit tests through the command `setup.py test`. This command builds the package locally, and then runs a specified set of tests against this local installation. The tests are specified through the `test_suite` property. Nose provides a test collector that plugs into this property. The change to `setup.py` is shown here:

```
...
install_requires = [
                    'docutils == 0.4',
                    'nose == 0.10.0',
                    ],

# use nose to run tests
test_suite='nose.collector',

# metadata for upload to PyPI
...
```

Notice that `setup.py` already requires Nose, so it is always guaranteed to be there. The dependency could have been specified through the `tests_require` property. This property specifies packages that will only be installed for testing. Had Nose been installed through those routes, it would not be generally available for development work. Nose isn't required for production, but it doesn't hurt to bundle it along, and it makes it much easier to set up a development environment.

```
$ python ./setup.py test
```

```
running test
running egg_info
writing requirements to src/RSReader.egg-info/requires.txt
writing src/RSReader.egg-info/PKG-INFO
writing top-level names to src/RSReader.egg-info/top_level.txt
writing dependency_links to src/RSReader.egg-info/dependency_links.txt
writing entry points to src/RSReader.egg-info/entry_points.txt
writing manifest file 'src/RSReader.egg-info/SOURCES.txt'
running build_ext
test_many_urls_should_print_first_results (test.test_application.➥
AcceptanceTests) ... ok
test_no_urls_should_print_nothing (test.test_application.➥
AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output (test.test_application.➥
AcceptanceTests) ... ok
```

```
----------------------------------------------------------------------
Ran 3 tests in 0.014s

OK
```

The next step makes the full test run available through Eclipse. Eclipse can run arbitrary programs from within the IDE and report the results. Eclipse calls these programs *external tools*. External tools are created and run through the application menu or the external tools button and drop-down on the toolbar. The external tools option is the little green play button with a toolbox in the lower-right-hand corner. It is shown in Figure 6-10.

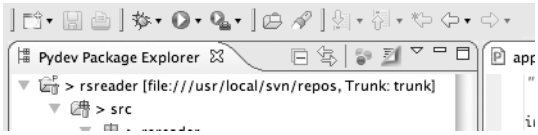
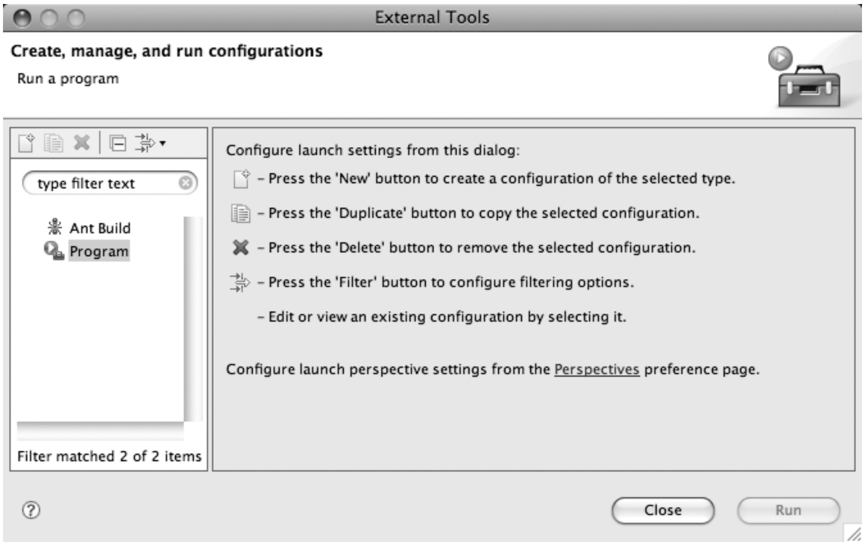


Figure 6-10. *The external tools button on the toolbar*

You create a new external tool through the application menu by selecting the Run ➤ External Tools ➤ External Tools Dialog menu item. From the drop-down button, the menu item is External Tools Dialog. This brings up the dialog shown in Figure 6-11.


External Tools

Create, manage, and run configurations

Run a program

type filter text

Ant Build

Program

Filter matched 2 of 2 items

Configure launch settings from this dialog:

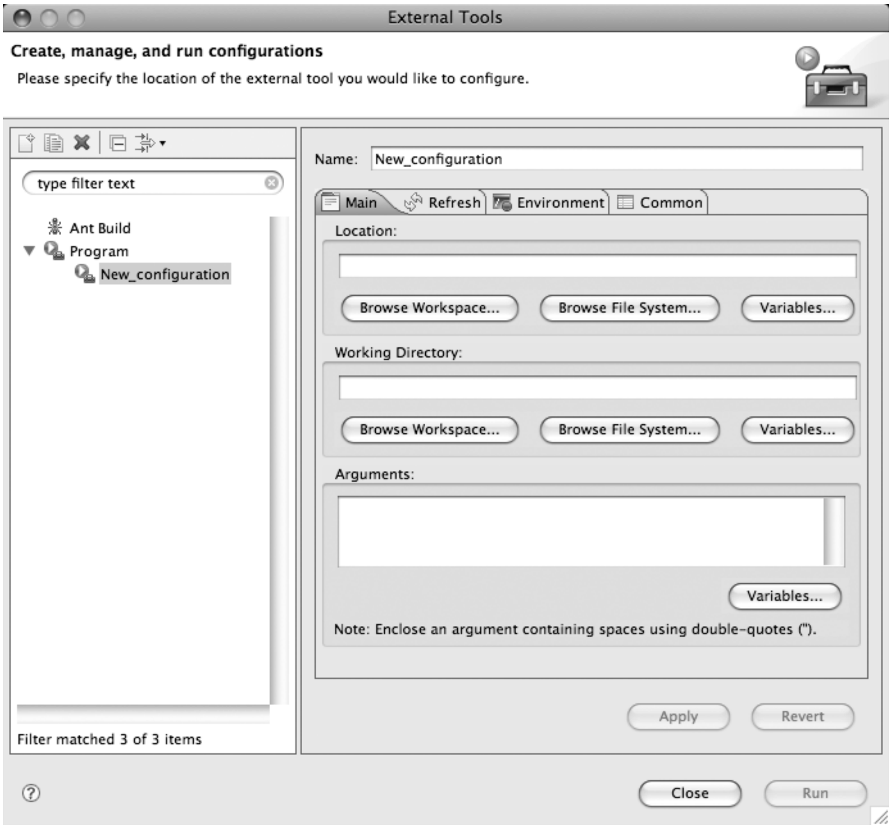
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

Figure 6-11. *The External Tools dialog*

The right half of the window contains basic instructions for getting started. The symbols there refer to the toolbar on the left. As with builders, there are two categories. One invokes Java's Ant and interprets the results, and the other executes arbitrary programs such as Python.

Clicking the leftmost icon on the toolbar or double-clicking the Program menu item creates a new program configuration and replaces the instructions with an editing panel. This is shown in Figure 6-12.


External Tools

Create, manage, and run configurations

Please specify the location of the external tool you would like to configure.

Name: New_configuration

Note: Enclose an argument containing spaces using double-quotes (").

Filter matched 3 of 3 items

Figure 6-12. *Defining a new external tool*

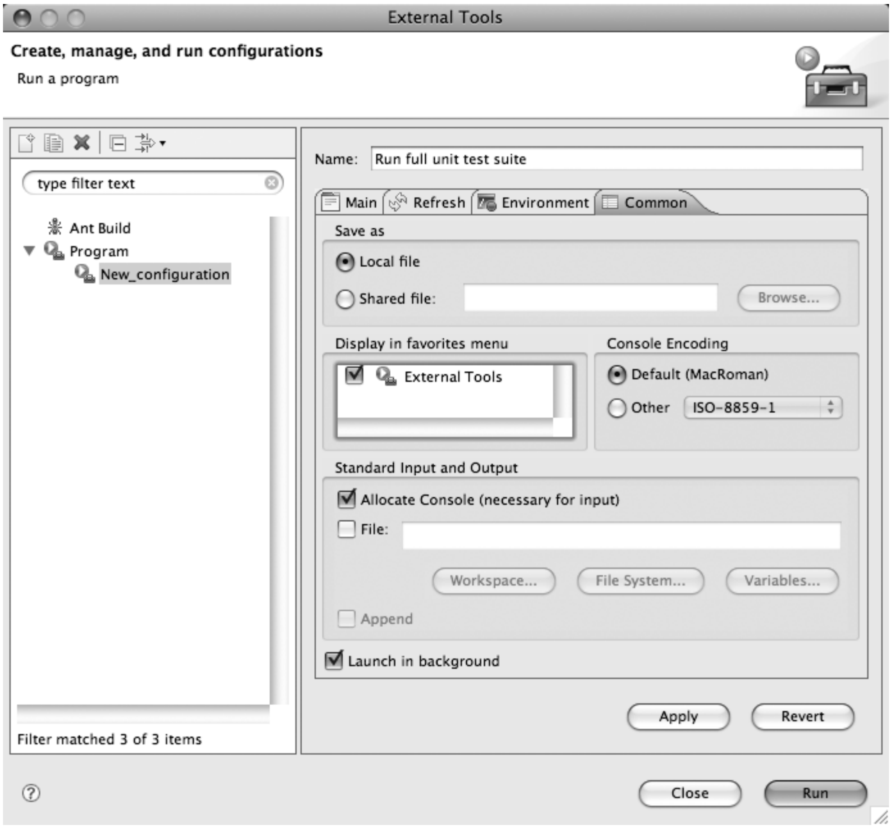
This window has strong similarities to the builder definition window. In fact, the Main pane is identical, except that the name defaults to New_configuration. In this case, the name for the new configuration will be "Run full unit test suite."

Location: This points to the Python interpreter.

Workspace Directory: This points to the top-level project directory. As with the Unit Tests builder, this is specified using the Eclipse variable `${workspace_loc:/rsreader}`.

Arguments : The arguments to Python are `./setup.py test`.

The Common tab allows several options to be specified. It is shown in Figure 6-13, with the values described in the following list selected.



Name: Run full unit test suite

Save as
[✓ Local file]
Shared file:

Display in favorites menu
[✓ External Tools]

Console Encoding
[✓ Default (MacRoman)]
Other ISO-8859-1

Standard Input and Output
[✓ Allocate Console (necessary for input)]
File:
Append

[✓ Launch in background]

Figure 6-13. *The Common tab for the external tool*

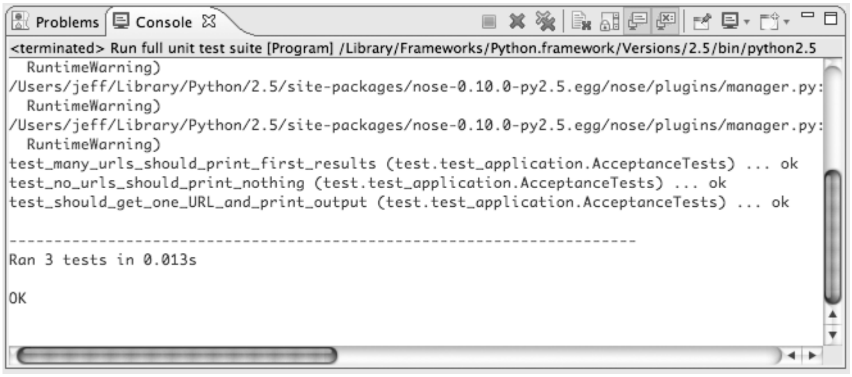
Save as: This section determines if the tool definition is stored in the local Eclipse configuration or in the project configuration. Unless your site has a common location for the Python binary, then I advise leaving it as “Local file.”

Console Encoding: This sets the console character encoding. As in Figure 6-13, the default should be left untouched.

Display in favorites menu: Checking the External Tools check box in this section places this definition into the External Tools menu. This is what we desire, so the check box is checked.

Standard Input and Output: This section chooses the input and output locations. The default allocates a console, and this is what we want.

Once the settings are chosen, you can click the Apply button, which creates the external tool entry. Nothing more remains to be done, so you can click the Close button, which will close the External Tools window and return focus to the workbench. The new task can be run from either the application menu or the external tools drop-down. On the main menu bar, the path is Run ➤ External Tools ➤ “1 Run full test suite,” and from the drop-down it is just “1 Run full test suite.” The output is shown in the console, as in Figure 6-14.



```
<terminated> Run full unit test suite [Program] /Library/Frameworks/Python.framework/Versions/2.5/bin/python2.5
  RuntimeWarning)
/Users/jeff/Library/Python/2.5/site-packages/nose-0.10.0-py2.5.egg/nose/plugins/manager.py:
  RuntimeWarning)
/Users/jeff/Library/Python/2.5/site-packages/nose-0.10.0-py2.5.egg/nose/plugins/manager.py:
  RuntimeWarning)
test_many_urls_should_print_first_results (test.test_application.AcceptanceTests) ... ok
test_no_urls_should_print_nothing (test.test_application.AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output (test.test_application.AcceptanceTests) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.013s

OK
```

Figure 6-14. *Output of "Run full test suite"*

Buildbot with Unit Tests

The fast build suite runs automatically before each update, and the developer can run the full test suite when needed. Buildbot needs to run the full suite every time it produces a build. As with the external build task just shown, Buildbot calls `python ./setup.py test`.

Conveniently, `setup.py test` returns a meaningful exit code. On UNIX systems, it returns a zero value for success and a nonzero value for failure. Success is defined as all unit tests passing, and failure is defined as one or more unit tests failing. The `ShellCommand` build steps interpret these exit codes in the same way. The change to `master.cfg` is one line.

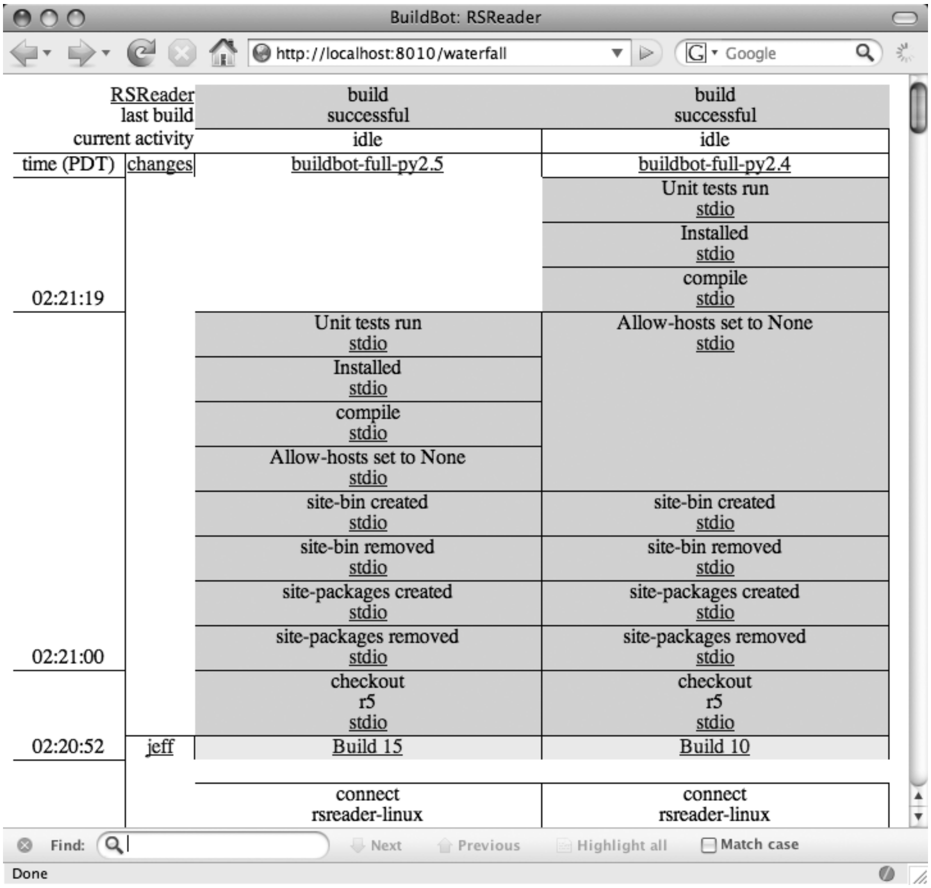
```
def pythonBuilder(version):
    python = python_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)
    ...
    f.addStep(ShellCommand,
              command=[python, "./setup.py",  "install",
                       "--install-scripts", site_bin],
              description="Installing",
              descriptionDone="Installed")
    f.addStep(ShellCommand,
              command=[python, "./setup.py", "test"],
              description="Running unit tests",
              descriptionDone="Unit tests run")
    return f
```

The change is saved, and the Buildbot master is reconfigured:

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

```
sending SIGHUP to process 786
...
Reconfiguration appears to have completed successfully.
```

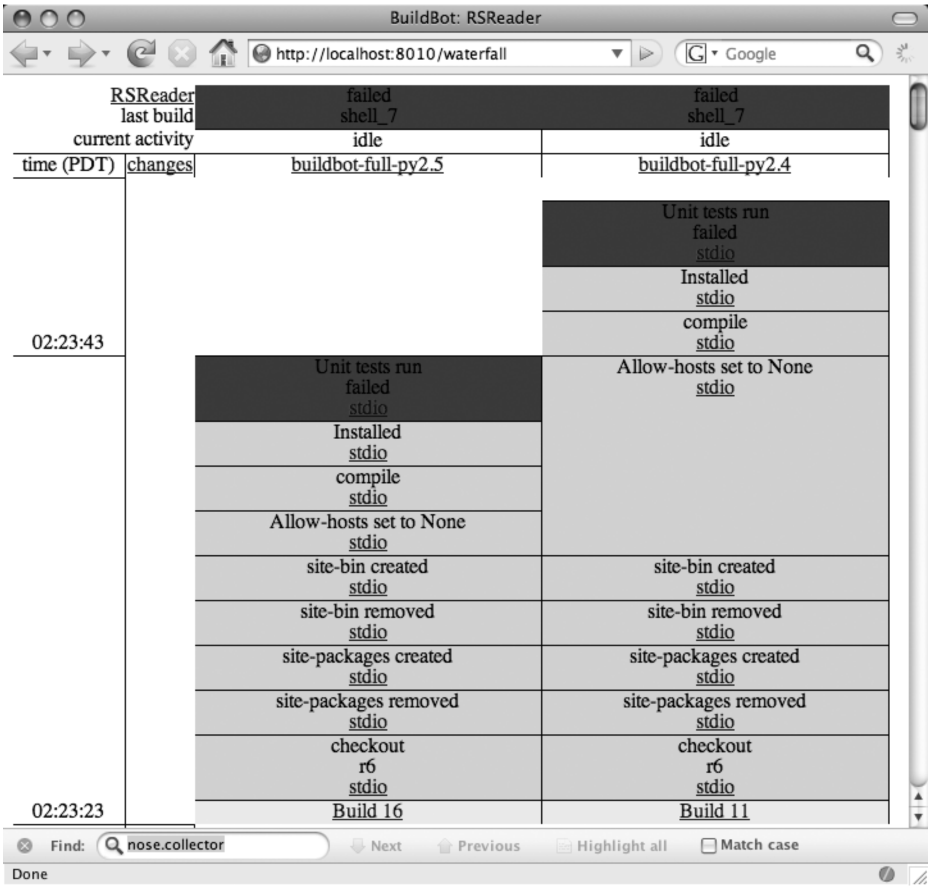
Committing the recent changes causes a build, or, if the changes have already been committed, then a build can be triggered from the command line. The resulting successful build is shown in Figure 6-15.



	RSReader last build	build successful	build successful
	current activity	idle	idle
time (PDT)	changes	buildbot-full-py2.5	buildbot-full-py2.4
			Unit tests run stdio
			Installed stdio
02:21:19			compile stdio
		Unit tests run stdio	Allow-hosts set to None stdio
		Installed stdio	
		compile stdio	
		Allow-hosts set to None stdio	
		site-bin created stdio	site-bin created stdio
		site-bin removed stdio	site-bin removed stdio
		site-packages created stdio	site-packages created stdio
02:21:00		site-packages removed stdio	site-packages removed stdio
		checkout r5 stdio	checkout r5 stdio
02:20:52	jeff	Build 15	Build 10
		connect rsreader-linux	connect rsreader-linux

Figure 6-15. *The test step succeeds when all tests succeed.*

The build succeeds, but it must be proven to fail. You can easily do this by adding a test that always fails. You commit the change, it triggers a build, and the build fails. The unsuccessful build is shown in Figure 6-16.



	RSReader last build	failed shell_7	failed shell_7
	current activity	idle	idle
time (PDT)	changes	buildbot-full-py2.5	buildbot-full-py2.4
			Unit tests run failed stdio
			Installed stdio
02:23:43			compile stdio
		Unit tests run failed stdio	Allow-hosts set to None stdio
		Installed stdio	
		compile stdio	
		Allow-hosts set to None stdio	
		site-bin created stdio	site-bin created stdio
		site-bin removed stdio	site-bin removed stdio
		site-packages created stdio	site-packages created stdio
		site-packages removed stdio	site-packages removed stdio
		checkout r6 stdio	checkout r6 stdio
02:23:23		Build 16	Build 11

Figure 6-16. *The test step fails when a unit test fails.*

Summary

Unit tests verify the behavior of the tested code. They ensure that the code works as the programmer expects. This distinguishes them from customer or acceptance tests that determine if the code works as the user expects. Unit tests are written by programmers and for programmers. They serve as living documentation that can be programmatically verified. Concrete unit-testing benefits include fewer bugs, less debugging, live documentation, increased confidence in refactoring, and better designs on small scales.

Test-driven-development (TDD) is a collection of self-reinforcing practices. Tests are written before the code. Tests and code are written in very short cycles—sometimes just a single line of code. Only enough code is written to make a test pass, and no code goes to production without associated tests. The unit tests are executed automatically, and the build fails unless all tests succeed. The code base is constantly refactored, and there are no restrictions on where those refactorings lead. Finally, when the product ships, the tests ship with it.

Designs should be driven by the customer. Minimalist communication methods should be used to specify designs, and interaction with the on-site customer should be emphasized.

Unit tests have a common structure. They are based around assertions that report whether a test has succeeded or failed. The tests themselves have four parts: fixture setup, execution, result verification, and fixture tear-down.

Two common Python frameworks for writing, executing, and reporting unit tests are unittest and Nose. unittest is a stock Python package modeled on Smalltalk's xUnit testing framework. Nose is very good at unit test discovery and execution. It is quite extensible, and it knows how to run unittest tests. It is the basis for automating unit test execution. In the development environment, it can be run after every change using Eclipse external builders to execute only fast-running tests. It takes only a single line to connect Nose to Setuptools's testing facility, and this in turn allows the tests to be run from Buildbot.

Attributes can be used to label test methods, and Nose can use those attributes to drive test execution. One very useful example is skipping slow tests. This is often done when running in the local development environment. Setting attributes can be simplified by using Python decorators.

The next chapter is very much a continuation of this one. It fleshes out the RSReader application using TDD techniques, and focuses on the process of writing a program using TDD and refactoring.