CHAPTER 10

Web Testing

The World Wide Web was the killer app for the Internet. In the course of less than a decade, it went from a simple document-sharing system for physicists to ubiquity. In 1994, if you'd said to someone that six years later billboards hawking milk would have URLs plastered on them, you would have been asked if you'd seen your psychiatrist recently and if she'd considered upping your dosage. Nevertheless, six years later there *were* URLs on billboards hawking all manner of consumer wares.

To say that the Web grew quickly is an understatement. It grew quickly, and it grew from the ground up. The technologies composing it were not planned out. They arose from need and circumstance. To put it more frankly, the Web as we know it today is a hodgepodge of different technologies that have been hacked together with the digital equivalent of bubblegum, spit, and baling wire. Afterward, standards bodies come through and codify the things that held together, but by that time everyone else has rushed on to the next set of problems.

Despite this madcap development, the Web has a very simple basis. Every application must use the same technologies to talk to the browser, so web applications have gross similarities in structure. These similarities give rise to repeated solutions to problems, which in turn means repeated testing methods. This is true of both unit testing and functional testing, but in this chapter I'll be demonstrating unit testing tools.

Really Simple Primer

At its simplest, the Web is a document format combined with a notation identifying these documents, and a protocol for using those identifiers to retrieve the documents.

- The document format is *HTML (Hypertext Markup Language)*.
- The document identifiers are called *URIs (Universal Resource Identifiers)*. When used to locate documents on a network, they are called *URLs (Universal Resource Locators)*.
- The network protocol is *HTTP (Hypertext Transfer Protocol)*.

All three are based on ASCII text rather than binary encodings. This makes them easy to manipulate with text-based tools such as text editors or telnet clients.

Web browsers are programs that retrieve documents via URLs, render the HTML, and then allow users to follow the URLs included in the retrieved documents. At its simplest, a browser follows a well-defined series of steps:

1. The user supplies a URL to the browser.
2. The browser uses the URL to locate the server containing the required document.
3. The browser requests the document from the server by HTTP.
4. The server returns the document to the browser.
5. The browser renders the document and presents it to the user.

This system allows for a limited amount of interaction from the user. The document may specify data to be retrieved from the user and a method for sending the results back to a server. This data is still sent back via HTTP. The result is yet another document.

HTML was originally intended to describe the content of a document, and not its formatting, but it was quickly forced into that role. *Cascading Style Sheets (CSS)* was created to restore this separation.

HTML is based on a document format called SGML (Standard Generalized Markup Language). SGML eventually begat a simpler markup language called XML (eXtensible Markup Language), which has become wildly successful for representing many kinds of data.

HTML

HTML is a text format. Ideally it describes the contents of a document, not how that document is to be rendered. In reality, this ideal is rarely met, and the elements of a form are often used for layout. Crucially, HTML documents also describe how they connect to other documents.

This is a simple HTML document:

```
<html>
   <head>
       <title>My Favorite Comics</title>
   <head>
   <body>
        I love XKCD and PVP.
    </body>
</html>
```

An HTML document can be viewed as a tree. The opening tag `<foo>` defines a node named "foo." It is terminated by the closing tag `</foo>`. The nodes are referred to as elements. Elements nest, but they do not interleave. If a tag contains no other elements, then the opening and closing can be combined, as in `<foo/>`.

Elements can also contain key/value pairs called *attributes*. Here the input tag has the type `text` and the name `comicname`: `< input type="text" name="comicname" />`.

SGML, from which HTML was derived, is a vast standard developed by committee. It was far from simple, and its parsers had to be very complete and strict in their interpretations.

HTML is a limited derivative of SGML with a very narrow problem domain: displaying simple documents on a network. HTML parsers were intended to be very forgiving so that slightly inaccurate documents created by relatively naive users could be successfully

presented. While this do-what-I-mean approach is in the spirit of Postel's Law,[1] it has introduced much ambiguity, and has resulted in a situation where no two web browsers render things in exactly the same way.

CSS

CSS describes how HTML documents are to be rendered. It is the result of an effort to remove formatting information from HTML documents. CSS binds HTML tags to formatting directives.

XML

The wild success of HTML and the relative failure of SGML gave birth to an effort to simplify SGML. This led to XML. The preceding description of HTML tells you most of what you need to know about XML syntax.

In the late '90s, XML was hyped beyond all belief. Vendors were suggesting that it would solve all data interchange problems, when clearly this was not the case. Despite this failure to deliver on the hype, I feel that it has been underappreciated for what it really does.

It provides a common syntax for structuring data, essentially doing for file formats what ASCII did for character sets. Having a common character representation vastly increased the portability of programs across computer systems, but it didn't solve all data interchange problems. It just allowed the focus to be raised to a new level of abstraction. XML does the same for data by supplying a universal syntax.

URI and URL

The URI format identifies documents unambiguously. Once obscure, it can now be seen even on billboards for toilet paper. A URI has four parts, organized as follows:

scheme **:** *hierarchical part [* **?** *query] [* **#** *fragment]*

The scheme identifies the kind of resource, and it determines how the other three parts are interpreted. Common schemes include the following:

- `http`, for web pages
- `https`, for encrypted web pages
- `file`, for files on the local system
- `mailto`, for e-mail addresses

The hierarchical part is separated from the scheme by a colon, and it is mandatory. A question mark separates the hierarchical portion from an optional query. It contains nonhierarchically organized information. The fragment is separated from these parts by a pound sign, and it serves as a secondary index into the identified resource. The following URI shows all the parts:

```
http://www.theblobshop.com/theguide?chapter=6times9#answer
```

1. Postel's Law is "Be conservative in what you do; be liberal in what you accept from others." See, for example, `http://ironick.typepad.com/ironick/2005/05/my_history_of_t.html`.

When a URI contains the information necessary to locate a resource, it is referred to as a URL. The two terms are often used interchangeably.

HTTP

HTTP is the network protocol that the Web is built on. It is defined in RFC 2616. In this protocol, a client initiates a connection to a server, sends a request, receives a response, and then disconnects. The server is not expected to maintain state between invocations.

A request consists of the following:

- A command
- The URI the command operates on
- A message describing the command

A response consists of the following:

- A numeric status code
- A message describing the response

The request and response messages share the same format. It is precisely the same format as that used to represent letters in e-mail, and it is defined in RFC 822. The message consists of a set of headers followed by a blank line and then an arbitrary number of data sections. There is one header per line, and each header is just a name/value pair. The name is on the left, the value is on the right, and they are separated by a colon.

JavaScript

JavaScript is not Java. It is not Java-light. It has nothing to do with Java. It is a dialect of a standardized language called ECMAScript, which is defined in the document ECMA-262.

JavaScript is a dynamically typed, object-oriented, prototype-based language. It has a C-based syntax, but its object model is much closer to that of Python, and Python programmers will find themselves at home.

JavaScript executes within the browser, and each browser has its own slightly different implementation. JavaScript programs manipulate a tree-shaped data structure representing the HTML document they reside in. Changes to this document are reflected on the screen. JavaScript programs can also send data back and forth to the server from which they were retrieved.

A display model that can be easily manipulated, combined with two-way network communications, has given rise to a programming paradigm called *Ajax (Asynchronous JavaScript and XML)*. You can use Ajax techniques to create web pages that behave much like local applications.

Web Servers and Web Applications

Web applications run on both the client that displays the pages and the server that delivers them, yet almost all applications start with the server. There is wide variation in how the applications are implemented.

At one end are simple scripts executed by the web server. The web server and scripts typically communicate using the Common Gateway Interface (CGI) defined by RFC 3875. At its heart, this standard defines a few more request headers describing the HTTP conversation. These are passed to your script, and the server expects your script to send back a few more headers. The new HTTP request is passed into your script via `stdin`, and the server reads the response message from your script's `stdout`.

The odds are that you will never deal with CGI at such a low level; all languages that I can think of provide libraries for handling these nuts and bolts. In Python, this library is named `cgi`.

At the other extreme are full stack applications. These implement everything from the web server to the application logic. They are often seen in shrink-wrapped applications, or with applications that act as platforms for other applications. One example in Python is the Plone content management system.

Between the two extremes are applications written with web application frameworks. These typically run on top of different web servers. These frameworks support writing complex applications, providing solutions for common problems. Typical features are

- Form validation and data conversion
- Session management
- Persistent data storage
- HTML templating

Common Python application servers include

- Zope
- Django
- Google App Engine
- Pylons
- Turbogears

These days, most applications of any appreciable size are written with web application frameworks. These frameworks run on top of some kind of a web server, such as Apache, IIS, or the Python-based Twisted.

Application frameworks typically have large startup costs connected to the extensive services they provide, so running them from CGI isn't feasible. The delay between the user's request and the application's response would be too long. Instead they connect to web servers through different mechanisms.

These mechanisms fall into two broad categories. In one, the application runs as part of the web server itself, and in the other, the application runs in a separate process and the web server forwards requests and responses to this process.

When an application framework runs as part of a web server's process, there is often little configuration to be done. The application often has direct access to the web server's internal state and its optimized services. The problem is that you're engaging directly with the web server's environment. This can lead to strange interactions, particularly when other applications are also running in the server's address space.

There are as many ways of doing this as there are web servers, since each different kind has its own extension interfaces. With Apache, this functionality is provided by the Apache plug-in `mod_python`.

THE PROBLEM WITH OCCUPYING ANOTHER'S SPACE

I once spent days trying to determine why a Python application was failing when running under `mod_python`, but succeeding from its test environment. It used the SQLObject object-relational mapping layer (see Chapter 9) in combination with the MySQLdb back end. The application would access the database layer, and then simply die without sending a response. There were no messages in the logs, there were no stack traces, and there were no core dumps.

Tracing the calls at the system level led to the discovery that PHP was loading a custom version of the dynamically linked MySQL client libraries. When MySQLdb attempted to load the client libraries, it was instead linked with the PHP version. The PHP version was incompatible at a very low level, and the calls to the database died silently.

Luckily, PHP was not required for the operation of the production system, and I was able to turn off the `mod_php` plug-in with impunity.

The alternative approach is running the application framework in another process. The web server passes requests and responses to and from the external process. Once again, there are multiple ways of accomplishing this, but in this case there is also a standard mechanism called FastCGI.

To make things worse, every application framework used to have its own method for interfacing to each web server. Even if two different frameworks both had FastCGI adapters, each was configured in a different way. Having m web server interfaces and n web servers leads to $m \times n$ combinations; or to put it more succinctly, it resulted in a big mess.

What happens when you want to connect multiple web applications to a single web server? What if you want to set up more than one application running under the same application framework? These used to be significant problems, but they've been solved within the last few years.

WSGI

The Web Server Gateway Interface (WSGI; pronounced *whiskey*) defines a simple interface between web servers and Python web applications. It is defined in PEP 333. Adapters are written from the web servers to WSGI, so applications only have to support a connection to WSGI. Over the last few years, WSGI has become ubiquitous. On the server side, it is supported by Apache, CherryPy, LightHTTPd, and Zope, among others. On the app server side, it is supported by CherryPy, Django, Pylons, Turbogears, TwistedWeb, and Webware, to name a few.

The interface is similar in concept to Java's Servlet interface. While servlets are designed for implementing any kind of network protocol, WSGI is focused on HTTP.

There are two parties in each WSGI conversation: the gateway and the application, with the gateway representing the web server. The application is a callable, and I'll refer to it as `application`. The interaction can be summarized as follows:

1. The gateway calls `application` passing an `environ` dictionary and a `start_response` callback. The dictionary `environ` contains the application's environment variables.
2. The application processes the request.
3. The application calls `start_response`, passing the response status and a set of response headers back to the gateway.
4. The application returns the response contents as an iterable object.

In the first step, the gateway calls `application(environ, start_response)`. The `application` object must be a callable, but it may be a function, a class, or an instance. The method the gateway calls for each of these is shown in Table 10-1.

Table 10-1. *Call Equivalents*

application Is a(n) . . .	application(environment, start_headers) Is Equivalent to . . .
Function or method	`application(environ, start_headers)`
Class	`application.__init__(self, environ, start_headers)`
Object	`application.__call__(self, environ, start_headers)`

In the third step, the `application` object calls `start_response(status, headers)` when it is ready to return HTTP results. This must be done before the last result is read from the iterator returned by `application(environ, start_headers)`.

In the fourth step, the returned sequence may be a collection, a generator, or even `self`, as long as the returned object implements the `__iter__` method.

Using the write Callback

Some underlying web servers read the application's results in a different way. They hand the `application` object an output stream, and instead of returning the results, the `application` object writes the results to this output stream. This stream is accessed through the `write(data)` callback, which is returned from `start_response(environment, headers)`. In this case, the calling sequence is as follows:

1. The gateway calls `application(environment, start_response)`.
2. The `application` object calls `write = start_response(status, headers)`.
3. The `application` object writes the results: `for x in results; write(x)`.
4. The `application` object returns empty results: `return [""]`.

WSGI Middleware

In this chapter, I will use the term *middleware* in the limited sense defined by WSGI. These components are both WSGI gateways and WSGI applications. They are shimmed between the web server and the application. They add functionality to the web server or application without needing to alter either. They perform duties such as the following:

- URL routing
- Session management
- Data encryption
- Logging traffic
- Injecting requests

The last two give an inkling of why WSGI middleware is important to testing. Middleware components provide a way of implementing testing spies and call recorders. These can be used to create functional tests. The underlying web server can also be completely replaced by a test harness that acts as a WSGI gateway. This bypasses the need to start a web server for many kinds of tests.

Testing Web Applications

Web testing breaks down into the two broad categories of unit testing and integration testing. Integration testing involves multiple components being tested in concert. It requires a more complicated testing infrastructure, it distances your tests from the origin of your errors, and it tends to take more time. It is an invaluable approach with web applications, since there are aspects of many programs that can't be performed in isolation, yet because of its shortcomings, it should be used judiciously.

This returns us to the idea of designing for testability. By restructuring your program, you can limit the number of places where you have to run integration tests, and this restructuring happens to result in more maintainable programs. There is a well-defined architecture called *model-view-controller (MVC)* that facilitates this.

MVC separates the input (controller) and output (view) from the computation and storage (model). Web programs receive sets of key/value pairs at distinct intervals as input. The computation is no different than with any other software. Both of these are easily tested with techniques you've already seen in previous chapters. The real differences reside in the view.

The views generate four distinct kinds of output:

- Graphics
- Marshalled/serialized objects in text form
- Markup
- Executable content

Each has a distinct set of testing strategies.

Graphics and Images

There are multiple levels of image testing. There are two basic strategies: one is to watch the image generation process, and the other is to examine the resulting image.

The first is accomplished with testing techniques that we've already examined. The drawing library is replaced with a fake or a mock, and the resulting instructions are verified. Common sequences of primitive drawing operations are combined into larger operations. These can be verified and then used as the blocks for instrumenting larger higher-level drawing operations.

The other approach employs additional techniques. At the simplest, you can check whether something was returned, and the basic characteristics are checked without regard for the contents at all. Image generation should produce results, and it should do so without raising an error. Verifying this may be enough for some problem domains.

The image can be validated through parsing. It is passed to the appropriate image library and rendered to an internal representation. The rendering process will fail if the image is not valid. Once rendered, your graphics library may supply enough data to verify certain image characteristics. These could include the image width and height, the image size, the number of bits in the color palette, or the range of colors.

In other cases, the contents of the images may need to be verified. The simplest cases are when a known image is generated. The resulting image may be compared byte for byte against a reference image. For other kinds of images, it may be sufficient to compare certain image properties such as the center of mass, average brightness, color spectrum, or autocorrelation results. These sorts of properties are generated using image-processing libraries. Each library has unique properties and should be chosen with regard to which properties must be measured.

Vector image formats often produce instructions that may already be text or that can be easily converted to text, and they may be treated as if it they were any other kind of text document.

It may also be possible to instrument the rendering library itself. The test subject is passed to the rendering library, and the calls that it produces are verified either through logs generated by test spies or by fakes and mocks.

Markup

The output from web applications isn't strictly limited to markup documents, but they form the vast bulk of the output you'll be testing. These can be analyzed through lexical, syntactic, and semantic tools. For the simplest cases, where you just want to verify that a word was included in otherwise tested results, lexical analysis may be sufficient. In these cases, the HTML output is just text, and the entire toolbox of Python string operators may be brought to bear. Regular expressions and `string.find` are very useful in these cases.

One of the primary drawbacks of lexical testing is that it doesn't verify that the document is well formed. However, this is easily done through syntactic testing techniques. In particular, the Python standard library includes `HTMLParser` for these simple cases.

At the syntactic level, it may be enough to verify that the output is valid HTML. This can be accomplished by passing the document through the standard library's `HTMLParser`. It allows you to quickly verify that a sequence of tags is included in a page, but it tells you little about the meaning of those tags—it's a very low-level tool.

More complete parsers produce a tree representing the parsed document. The structure and relationship between nodes is available for your tests' perusal. The elements are the nodes, and they are named. Attributes are attached to the element, as are the attribute values. Child and sibling nodes can be iterated for every element. This functionality is available through the standard library's `ElementTree` package.[2] Parsing a document with `ElementTree` is easy:

```
import xml.etree.ElementTree as et
...
doc = """
<html>
   <head>
      <title>Comic Feeds</title>
   </head>
   <body bgcolor="#ffffff">
      You are not subscribed to any feeds
   </body>
</html>
"""

parsed = et.XML(doc)
```

The parsed object is an `ElementTree` describing the document. Each node contains methods for navigating the subtrees.

```
def setup(self):
    self.root = et.XML(doc)
def test_get_tag_name(self):
    root = et.XML(doc)
    assert self.root.tag == 'html'

def test_get_children(self):
    children = self.root.getchildren()
    assert children[0].tag == 'head'
    assert children[1].tag == 'body'

def test_get_attributes_from_body_tag(self):
    body = self.root.getchildren()[1]
    assert body.item() == [('bgcolor', '#ffffff')]
```

The line between syntactic analysis and semantic analysis of HTML documents is fuzzy. When writing tests, you want to know the answer to questions such as the following:

---

2. `ElementTree` was added to the standard library in Python 2.5, so it is not present in earlier versions. It still exists as an external package, and you can install it with `easy_install`. It installs into a different namespace: `elementtree.ElementTree`. It is under active development, and there have been significant improvements since it was added to the standard libraries, so it may be worth installing it even if you are using Python 2.5. In this case, it happily coexists with the standard installation.

- Are the two links to my favorite comics included in this document?
- Is the table of contents included?
- Are there three links to xkcd?

These all involve searching for specific nodes within the parsed document. The overall test pattern is the same—the document is parsed, and then the element tree is searched for the relevant tags. `ElementTree` searches are done with the `find()`, `findtext()`, and `findall()` methods. The following code finds the `title` tag in the previous example:

```
def test_find_title_tag(self):
    title = self.root.find('.//title')
    assert title.tag == 'title'
```

The `findtext()` method returns the text contained in the found tag:

```
def test_find_title_text(self):
    title_text = self.root.findtext('.//title')
    assert title_text == 'Comic Feeds'
```

If the search expression matches more than one element, then these two methods will only return results for the first one. To return all matches, you must use the `findall()` method:

```
def test_find_all_top_of_roots_children(self):
    root_children = self.root.findall('*')
    assert len(root_children) == 2
    child_tags = [x.tag for x in root_children]
    assert child_tags == ['head', 'body']
```

You may be wondering about the strange query strings that the find operations use. These are XPath queries. *XPath* is a standard format for locating nodes within an XML document. XPath is somewhat like a directory path specification. It is a very rich query language, but the `ElementTree` implements only a small subset of the full specification. Despite its limitations, it's quite usable for many testing purposes.

A summary of query components can be found in Table 10-2. The full XPath specifications can be found on the World Wide Web Consortium (W3C) web site at `www.w3.org/TR/`. Although the current version is 2.0, most XPath packages still support only 1.0 or some variant thereof. More complete XPath implementations can be found in other Python packages such as PyXML.

Table 10-2. *The XPath Operations Supported by ElementTree*

Operator	Action
`foo`	Matches an element with the tag `foo`
*	Matches any child tag name
`.`	Specifies the current tag; mostly used at the top level
`/`	Separates levels within the tree
`//`	Finds the next pattern anywhere in the subtree

The other solution is the package named `BeautifulSoup`. It is downloaded via `easy_intall BeautifulSoup`. It makes free-form queries of HTML and XML documents. It possesses a wide set of parsers that allow it to work with a variety of web XML–related formats. Some of these parsers are very strict, and some are very permissive.

```
from BeautifulSoup import BeautifulSoup
...
class TestBeautifulSoup(TestCase):

    def setUp(self):
        self.soup = BeautifulSoup(doc)

    def test_find_title_element(self):
        title = self.soup.find(name='title')[0]
        assert title.name == 'title'

    def test_find_body_by_attributes(self):
        body = self.soup.find(attrs={'bgcolor': '#ffffff'})
        assert body.name == 'body'
        assert body.attrs = [(u'bgcolor', u'#ffffff')]

    def test_find_by_text(self):
        # must match entire text string
        text = self.soup.find(text='Comic Feeds')
        assert text == 'Comic Feeds'
```

When the find arguments are used in the same expression, they are anded together, restricting the set of returned elements.

The name and text search attributes aren't limited to strings. They can be replaced by regular expressions. This is particularly useful in combination with the `findAll()` method, which returns all matches, rather than just the first one.

```
import re
...
def test_find_all_elements_with_e(self):
    has_e = self.soup.findAll(name=re.compile('e'))
    element_names = [x.name for x in has_e]
    assert element_names == ['head', 'title']
```

Testing JavaScript

Testing JavaScript is far more involved than testing other kinds of content. It poses many of the same problems as testing Python. As with Python, there are tools for performing both unit and functional tests. I'll only be dealing with the former in this chapter.

In order to unit test JavaScript, you have to be able to run the JavaScript code. There are stand-alone interpreters for JavaScript, but these have shortcomings compared with browsers. First and foremost, each browser has a slightly different set of libraries. Emulating these differences in a stand-alone interpreter is a technical challenge that nobody has risen to yet, nor are

they likely to. Since these changes must be tested, we're left with the option of executing the code with the target browsers.

This is done with the software package JsUnit (`www.jsunit.net/`), written by Edward Hieatt of Pivitol Labs. It is not to be confused with the similarly named JsUnit package written by Jörg Schaible.

Using JsUnit

The first step in working with JsUnit is obtaining a copy. It can be downloaded from `http://downloads.sourceforge.net/jsunit`. As of this time, there are two ZIP files available. One is an Eclipse plug-in, and the other is JsUnit itself. Sadly, the Eclipse plug-in does not work with the most recent versions of Eclipse (3.2 as of this writing), so I won't discuss its use.

We'll be using JsUnit in conjunction with the RSReader project from previous chapters. This is the first non-Python tool in the project, so it fits in a different place. I tend to create a generic tools directory when the project is small. Only when a particular class of tools gets large enough do I create dedicated directory hierarchies.

```
$ cd /Users/jeff/Documents/ws/rsreader
$ ls
```

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

```
$ mkdir tools
$ cd tools
$ curl -L -o jsunit2.2alpha11.zip➥
    http://downloads.sourceforge.net/jsunit/jsunit2.2alpha11.zip
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 5968k  100 5968k    0     0   967k      0  0:00:06  0:00:06 --:--:-- 1169k
```

```
$ unzip jsunit2.2alpha11.zip
```

```
  inflating: jsunit/app/css/readme
  inflating: jsunit/app/emptyPage.html
...
  inflating: jsunit/tests/jsUnitUtilityTests.html
  inflating: jsunit/tests/jsUnitVersionCheckTests.html
```

```
$ ls
```

```
jsunit/                jsunit2.2alpha11.zip
```

```
$ rm jsunit2.2alpha11.zip
```

The JavaScript source and tests will be placed in their own directory trees:

```
$ cd ..
$ mkdir javascript
$ ls
```

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

```
$ cd javascript
$ mkdir src
$ mkdir test
$ ls
```

```
src     tests
```

Running a Test

You can run tests stand-alone or distributed. Stand-alone tests are suitable for developing the tests themselves or interactively testing small pieces of code, as they require the user to interact with a web browser. Distributed tests are run from within the build. They use a farm of web browsers that may reside on other machines.

To start with, I'll demonstrate stand-alone testing. Once you've gained an understanding of how to use JsUnit, I'll move on to using distributed tests, in order to tie them into the larger build for automatic execution.

The JsUnit test runner is a web page in your browser. Open the browser of your choice to the file `rsreader/tools/jsunit/app/testRunner.html`. On my system, this is `file:///Users/jeff/Documents/ws/rsreader/tools/jsunit/testRunner.html`. The test runner is shown in Figure 10-1.

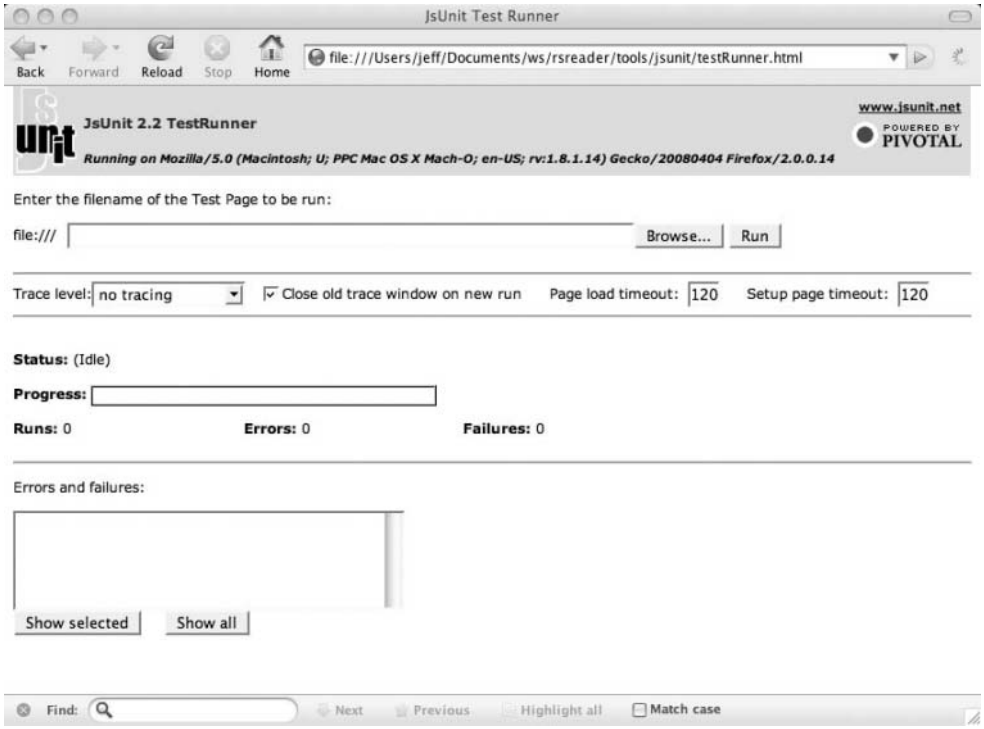

Figure 10-1. *The JsUnit stand-alone test runner*

The name of the file containing the tests to be run is put into the first text box. Clicking the Run button beside the text bar loads and runs the test.

A JsUnit test is an HTML file. JavaScript documents frequently manipulate the document structure, so it must be included as part of the test. HTML is the natural place to do this. The following file defines a function and a test for that function:

```
$ more test/lineTest.html
```

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript">

function line(m, x, b) {
    return m*x + b;
}
```

```
function testCalculationIsValid() {
    assertEquals("zero intercept", 10, line(5, 2, 0));
    assertEquals("zero slope", 5, line(0, 2, 5));
    assertEquals("at x = 10", 25, line(2, 10, 5));
}

        </script>
    </head>
    <body>
        This page tests line(m, x, b).
    </body>
</html>
```

First notice that all of the test code resides within the `<head>` tag. The first `<script>` tag is what makes this a JsUnit test:

```
<html>
...
        <script language="JavaScript"
                        src="../../tools/jsunit/app/jsUnitCore.js">
...
</html>
```

It loads all of the JsUnit test code that executes when the test page finishes loading. If anything goes wrong with this loading process, then you'll see the message shown in Figure 10-2.

■ **Note** I strongly advise you to adjust the "Page load timeout" and "Page setup timeout" to much smaller values. They are specified in seconds, and the defaults are 2 minutes (120 seconds). This is much too long when you're running tiny tests from the filesystem. Somewhere between 2 and 5 seconds is a reasonable value.

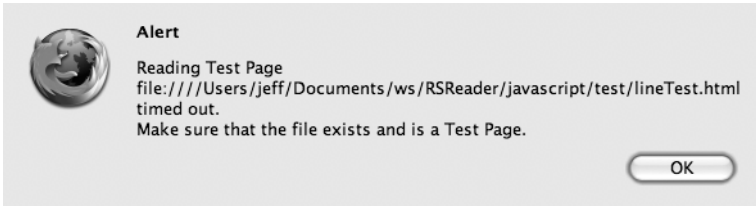

Figure 10-2. *The dreaded "Reading Test Page . . . timed out" alert*

The error in Figure 10-2 means one of two things. Either the file doesn't exist or the path to `jsUnitCore.js` is incorrect. You can check the former by trying to load the URL in a normal web browser. The latter is a bit trickier. Change to the test page's directory (in this case `/Users/jeff/Documents/ws/rsreader/javascript/test`), and then cut and paste the path in the script tag's `src` attribute.

This is the single most frustrating part of getting started with JsUnit.[3] The good news is that once it's ironed out, you won't have to deal with it again. If you're using Eclipse and Pydev, you should set up a template for these test pages so that nobody on your project will have to deal with this problem either.

A successful test run is shown in Figure 10-3. The progress bar is full and green, and the total test execution time is shown above it. Notice that this absolutely trivial test took over half a second to run.

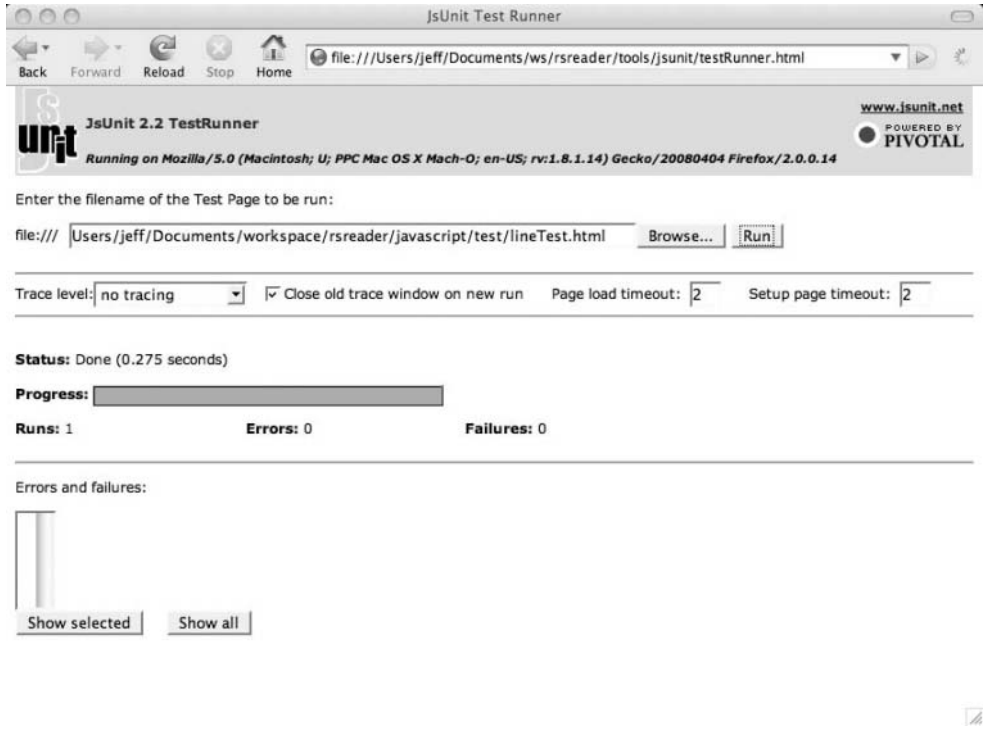

Figure 10-3. *The test runs and succeeds.*

JsUnit tests are *slow*. This highlights a theme with all JavaScript testing tools: write as few tests as possible. Don't do this by skimping on tests, though—do it by structuring your code so that you need to run as few tests as possible.

3. I suffer so you won't have to.

An excellent way of doing this is by depending on someone else to do the heavy lifting. There are wonderful JavaScript libraries available for free that implement the vast majority of things you'll want to do. MochiKit, script.aculo.us, and Ext JS are three of the most popular. Use them.

How It Works

Test cases are recognized by name. If a function begins with the string `test`, then it is treated as a test. JsUnit is a direct translation of the XUnit framework, as is Python's unittest. There is a one-to-one correspondence between most of the major concepts. This is shown in Table 10-3.

Table 10-3. *The Correspondence Between unittest and JsUnit*

unittest	JsUnit
`TestCase` classes	Test pages
Test methods	Test functions
Test suites	Test suite pages
Extend `unittest.TestCase`	Include `app/jsUnitCore.js`
Import subject code	Include subject code
`setUp()` and `tearDown()` methods	`setUp()` and `tearDown()` functions
IDE test runners	Web browser test runners

The correspondence between the two carries through to the test methods, too. These similarities are shown in Table 10-4.

Table 10-4. *The Correspondence Between unittest Test Methods and JsUnit Test Functions*

unittest Method	JsUnit Function
`assert_()`	`assert()`
`failUnless()`	`assertTrue()`
`assertTrue()`	`assertTrue()`
`failIf()`	`assertFalse()`
`assertFalse()`	`assertFalse()`
`assertEqual()`	`assertEquals()`
`failUnlessEqual()`	`assertEquals()`
`assertNotEqual()`	`assertNotEquals()`
`failIfEqual()`	`assertNotEquals()`
`failUnless(x is None)`	
`failIf(x is None)`	
`failUnless(x is None)`	`assertNull()`
`failIf(x is None)`	`asssertNotNull()`
	`assertNaN()`
	`assertNotNan()`

Connoisseur of the Undefined

If you're not familiar with JavaScript, there are a few methods that bear some explanation. Unlike Python, JavaScript draws a distinction between variable declaration and variable assignment.

In JavaScript, variables must be declared before they are used. Until you assign a value to that variable, its value is `undefined`. (It might have been clearer to call it *uninitialized*.) After you assign a value to the variable, it is no longer `undefined`. When you want to say that this variable has no value, you assign it the value `null`, which corresponds to Python's `None`.

As variables can be in two different indeterminate states, it is necessary to have two different sets of test methods. Since Python variables are created by the act of assignment, there is no such thing as a declared but unassigned variable, and there is no need for these test functions. The trade-off is that in Python, it is possible to create new variables accidentally by misspelling names. The following test function shows how `undefined` and `null` relate:

```
function testVariableInitializationStates() {
    var foo;
    assertUndefined(foo);
    assertNotNull(foo);

    foo = 0;
    assertNotUdefined(foo);
    assertNotNull(foo);

    foo = null;
    assertNotUndefined(foo);
    assertNull(foo);
}
```

The second feature visible in the test methods is the value `NaN`, which is short for *Not a Number*. JavaScript returns `NaN` from many arithmetic expressions that would raise exceptions in Python. Commonly, it also arises when a string-to-numeric conversion fails. The following Python test checks for just such a failure:

```
    def testNumericConversionFailure(self):
        self.failUnlessRaises(ValueError, int, 'foo')
```

It is equivalent to this JavaScript test:

```
    function testNumericConversionFailure() {
        assertNaN(parseInt('foo'));
    }
```

The value `NaN` is a valid JavaScript number, and it can participate in normal computations. JavaScript also has special values to represent positive and negative infinities. In general, where Python will generate an exception such as `DivisionByZero`, JavaScript will return a sensible but not terribly useful symbol representing the numeric construct.

Python also has a `nan`, but it appears in fewer places. While Python mixes exceptions and special symbolic representations, JavaScript is pleasantly consistent in its usage.

Adding a Little More Realism

Note that in Table 10-3, I mentioned including subject code in the test. However, I didn't do that in the simple test example, and the subject function `slope(m, x, b)` is declared within the test itself. To make the example a bit more realistic, I'll move it to the source directory in a file named `line.js`, and I'll reference that from the test. The subject code is shown in Listing 10-1 and the test is shown in Listing 10-2.

Listing 10-1. *The Subject Code in rsreader/javascript/src/line.js*

```
function line(m, x, b) {
   return m*x + b;
}
```

Listing 10-2. *The Test Code in rsreader/javascript/test/lineTest.html*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript" src="../src/line.js"></script>
        <script language="JavaScript">

function testCalculationIsValid() {
    assertEquals("zero intercept", 10, line(5, 2, 0));
    assertEquals("zero slope", 5, line(0, 2, 5));
    assertEquals("at x-axis", 25, line(2, 10, 5));
}

        </script>
    </head>
    <body>
        This a page tests line(m, x, b).
    </body>
</html>
```

With these changes in place, the test executes successfully, leaving you with a green bar in the test runner.

Manipulating the DOM

To do anything of interest, a JavaScript program must interact with the user, and to interact with the user, it must interact with the browser. This is done through the *Document Object Model (DOM)*. The DOM is a standard internal representation of the HTML document being presented to the user. It acts as the formal interface between the browser and the JavaScript interpreter.

Anything passing between the browser and your JavaScript code goes through the DOM. By embedding your tests within HTML pages, JsUnit gives them access to the DOM.

We're going to add a validation function to `line.js`. It will check an input field named `slope` in a form named `lineForm`. When there is an error, it will write an error message with a `<div>` tag with the ID `errorMsg`. The validation will be activated when someone clicks the `calculate` button. Listing 10-3 gives a skeleton test document that will be filled in as we walk through the process.

Listing 10-3. *The Test Skeleton for rsreader/javascript/test/testSlopeValidator.html*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript" src="../src/line.js"></script>
        <script language="JavaScript">
           // tests go here
        </script>
    </head>
    <body>
         // DOM elements for the test go here
    </body>
</html>
```

The input will require a form named `slopeForm`. Since the form is not being submitted, we don't need to include an action.

```
<form name="lineForm">
</form>
```

The form needs to include the input text field named `slope`:

```
<form name="lineForm">
    <input type="text" name="slope"/>
</form>
```

The validator is activated when the user clicks the calculate button:

```
<form name="lineForm">
    <input type="text" name="slope"/>
    <input type="button" value="Calculate" onclick="validateSlope()"/>
</form>
```

The calculate button will never be called by the tests. It serves as documentation, showing how the tested method should be used within a real document.

The error messages will be presented in a `<div>` tag. When the validator is run, it will change the inner contents of this element. The tag is split in two to emphasize that it is the contents that are important.

```
<div id="errorMsg"></div>
<form name="lineForm">
    <input type="text" name="slope"/>
    <input type="button" value="Calculate" onclick="validateSlope()"/>
</form>
```

The location of the error message isn't important. It could just as well be part of the form, but placing it outside underscores this fact.

For the first tests, you'll just want to verify that the included subject page defines the method you want to test:

```
function testThatValidateSlopeIsDefinedAndIncluded() {
    validateSlope();
}
```

The tests will be looking at the value of the `<div>` tag, so it should be cleared to a known value before every test. This is done with a `setUp()` function:

```
function setUp() {
     document.getElementById('errorMsg').innerHTML = "";
}

function testThatValidateSlopeIsDefinedAndIncluded() {
    validateSlope();
}
```

This preceding JavaScript and the DOM fixtures are placed into the test skeleton. The minimal test is shown in Listing 10-4.

Listing 10-4. *A Minimal Test in rsreader/javascript/test/testSlopeValidator.js*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript" src="../src/line.js"></script>
        <script language="JavaScript">

function setUp() {
     document.getElementById('errorMsg').innerHTML = "";
}

function testThatValidateSlopeIsDefinedAndIncluded() {
    validateSlope();
}

        </script>
    </head>
```

```
    <body>
        <div id="errorMsg"></div>
        <form name="lineForm">
            <input type="text" name="slope"/>
           <input type="button" value="Calculate" onclick="validateSlope()"/>
        </form>
    </body>
</html>
```

The `validateSlop()` function hasn't been defined yet, so when the test is run, you will see a failure, as shown in Figure 10-4.

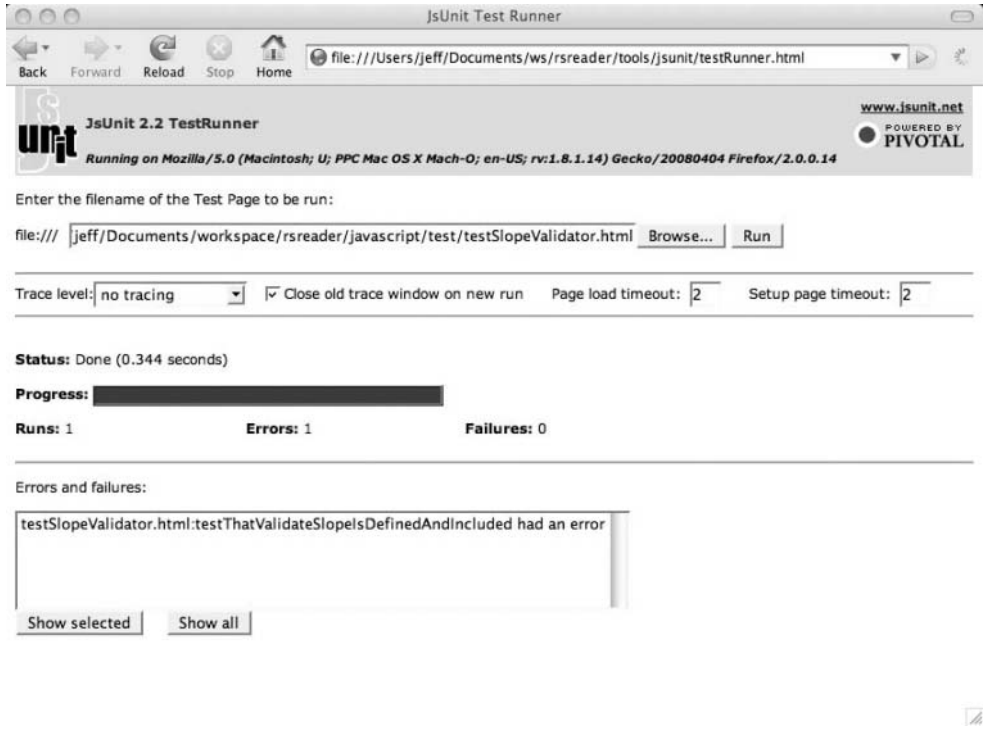

Figure 10-4. *The test dies because the subject hasn't been implemented.*

JsUnit, like unittest and Nose, distinguishes between failures and errors. Failures result from assertions failing, and errors result from the tests dying during execution. The combined failures and errors are shown in the "Errors and failures" panel.

Each line in the "Errors and failures" panel represents one unit test. Highlighting a test and clicking "Show selected" brings up the details for that test in an alert. Clicking "Show all" brings up a window with all the failure information. Failures will show detailed information about the failure, such as the expected value and the value produced. Errors will show a trace-back. (If you've worked with JavaScript much, you'll be drooling at this prospect.) This is shown in Figure 10-5.

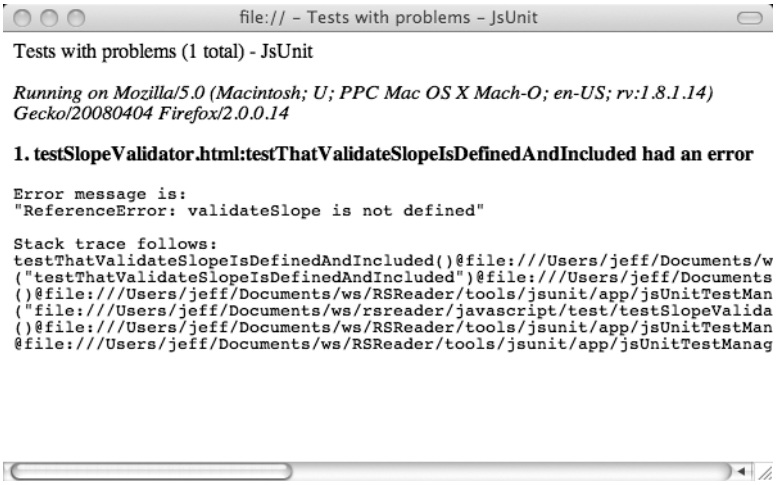

Figure 10-5. *The "Show all" errors window with a message and traceback*

Add a minimal definition to `line.js`. The file now reads as follows:

```
$ cat src/line.js
```

```
function validateSlope() {
}

function line(m, x, b) {
   return m*x + b;
}
```

Run the test again. This time it succeeds, as shown in Figure 10-6.

From this point forward, the process is very similar to developing with TDD in Python. You define a function that checks one kind of validation failure:

```
function testValidationFailsWhenEmpty() {
    document.lineForm.slope.value = '';
    validateSlope();
    var errorMsg = document.getElementById('errorMsg');
    assertEquals('You must define a slope', errorMsg.innerHTML);
}
```

This function demonstrates the classic XUnit test pattern. It sets the expectations, performs the action, and then checks the results, which are to be found within the `<div id="errorMsg">` tag in the test page.

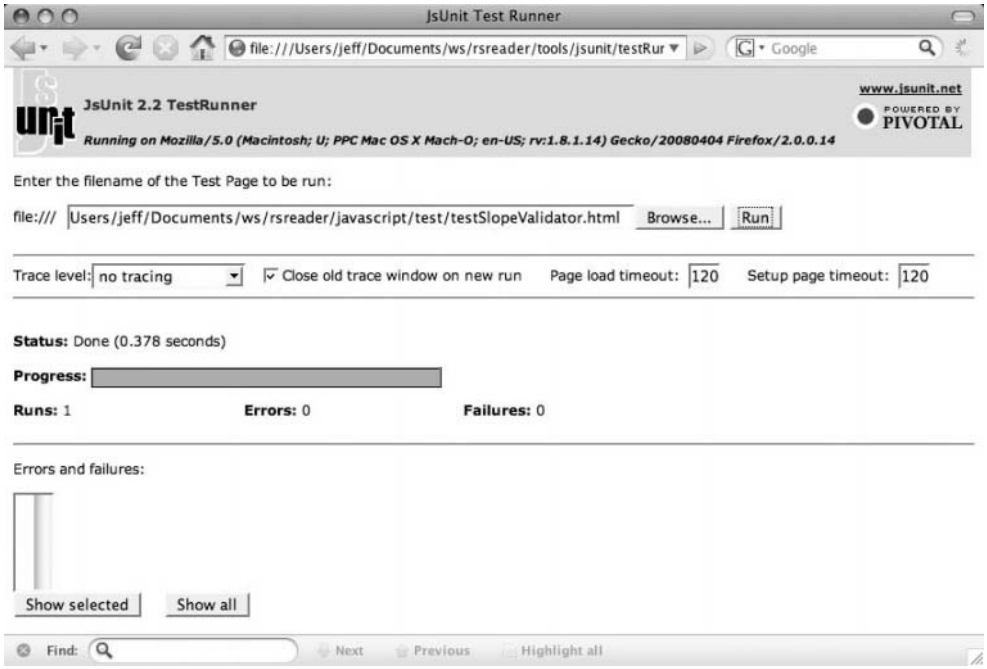

Figure 10-6. *The test runs successfully.*

You run the test and, as expected, it fails. You then add the corresponding logic to `validateSlope()`. The new definition is as follows:

```
function validateSlope() {
    var errorMsg = document.getElementById('errorMsg');
    errorMsg.innerHTML = "You must define a slope";
}
```

Now when you run the test, it succeeds. This back-and-forth process continues until the test cases are completed, as shown in Listings 10-5 and 10-6.

Listing 10-5. *The Subject Code in rsreader/javascript/src/line.js*

```
function line(m, x, b) {
   return m*x + b;
}

function validateSlope() {
    var slope  = document.lineForm.slope.value;
    var errorMsg = document.getElementById('errorMsg');
    if (!slope) {
        errorMsg.innerHTML = 'You must define a slope';
    } else if (isNaN(parseInt(slope))) {
        errorMsg.innerHTML = "The slope must be a number";
```

```
    } else {
        errorMsg.innerHTML = '';
    }
}
```

Listing 10-6. *The Test Code in rsreader/javascript/test/testSlopeValidator.html*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript" src="../src/line.js"></script>
        <script language="JavaScript">

function setUp() {
    // clear out any previous message
    errorMsg().innerHTML = '';
}

function testFieldIsBlankAfterSuccessfulValidation() {
    document.lineForm.slope.value = '0';
    errorMsg().innerHTML = 'an arbitrary message';
    pressCalculate();
    assertEquals('', errorMsg().innerHTML);
}

function testValidationFailsWhenEmpty() {
    document.lineForm.slope.value = '';
    pressCalculate();
    assertEquals('You must define a slope', errorMsg().innerHTML);
}

function testValidationFailsWhenNotANumber() {
    document.lineForm.slope.value = 'this is not a number';
    pressCalculate();
    expected = 'The slope must be a number';
    assertEquals(expected, errorMsg().innerHTML);
}

// separate tests from validation mechanism details
function pressCalculate() {
    validateSlope();
}
```

```
// make tests more concise
function errorMsg() {
    return document.getElementById('errorMsg');
}

        </script>
    </head>
    <body>
       <form name="lineForm">
           <input type="text" name="slope" value="0"/>
           <input type="button" value="Calculate" onclick="validateSlope()"/>
           <div id="errorMsg"></div>
       </form>
    </body>
</html>
```

There are a few things worth pointing out about the final tests:

- The original test function `testThatValidateSlopeIsDefinedAndIncluded()` no longer serves any function, so it has been removed.
- The call to the subject method `validateSlope()` has been extracted into the method `pressCalculate()`. This makes it clearer what user action is being tested. It also makes the test less dependent on the name of the subject function. This is more of a worry with JavaScript, since we don't have refactoring tools at our disposal.
- The common subexpression `document.getElementById('errorMsg')` has been extracted into a utility method. This makes the test code more concise and moves this dependency into one place, reducing the test's brittleness.

Aggregating Tests

JsUnit tests can be combined using test suite pages. Test suite pages don't define any test functions. Instead they define a single function called `suite()`. This function, which you create, must return a `jsUnitTestSuite()` object. Test pages are added using the methods `addTestPage(testPagePath)` and `addTestSuite(testSuite)`. The argument to the latter must be a `jsUnitTestSuite`.

At this point, two tests have been defined. They are `testLine.js` and `testSlopeValidator.html`. These are combined into the suite page shown in Listing 10-7.

Listing 10-7. *The line.js Test Suite rsreader/javascript/test/lineSuite.html*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript">
```

```
function suite() {
      var suite = new top.jsUnitTestSuite();
      suite.addTestPage("../../javascript/test/lineTest.html");
      suite.addTestPage("../../javascript/test/testSlopeValidator.html");
      return suite;
}
        </script>
    </head>
    <body>
        All tests for line.js.
    </body>
</html>
```

The trickiest part about making a test suite page is the getting the paths right. They are relative to `testRunner.html`, which in this example is in `rsreader/tools/jsunit/`.

Usually when you create test suites, you aggregate test pages or other suite pages. For the purposes of `addTestPage(testPagePath)`, there is no difference between a test page full of test functions and a test page that defines a suite. Accordingly, you will deal with test suite objects and `addTestSuite(testSuite)` infrequently. The following `suite()` function aggregates several test suite pages into a single suite:

```
function suite() {
    // the geometry suite
    var suite = new top.jsUnitTestSuite();
    suite.addTestPage("../../javascript/test/lineSuite.html");
    suite.addTestPage("../../javascript/test/circleSuite.html");
    return suite;
}
```

Running Tests by URL

Remember what I wrote about JsUnit being slow? The preceding test suite with four methods takes about 1.7 seconds on Firefox 2.0.0.14 on a dual-core 2.3 GHz processor.[4] Running an exhaustive set of JsUnit tests takes a long time. During development, you'll run tests frequently, so you'll want to break them up into meaningful and useful chunks that you can run as needed, and run quickly. Essentially, you'll use test suites as a means of optimizing your test runs. Frequently you'll run specific suites interactively.

You'll find yourself pasting, clicking, and running ad nauseam. Any means of making this process go faster will make your life easier and encourage you to use tests.[5] You can easily speed things up by supplying information as part of the test page URL.

---

4. In five years, we're both going to look back at this sentence and think, "Gee, that's not even as fast as the low-voltage CPU in my vacuum cleaner." I hate dating myself.

5. And that is what I want—I want you to test and test and test, simply because I like using software that isn't infested by bugs. I want you to write better software, and I want you to have more fun doing it, too.

Several parameters can be supplied as part of the test runner URL. The `testPage` parameter is the path to the test page that is run. The `autoRun` parameter tells the runner to immediately execute the test page. Instead of loading the test runner, typing in the test page, and then clicking Run, you could simply open the following URL to execute `lineSuite.html`:

```
file:///Users/jeff/Documents/ws/rsreader/tools/jsunit/testRunner.html?testPage=/➥
Users/jeff/Documents/ws/rsreader/javascript/test/lineSuite.html&autoRun=true
```

Values may also be passed to the test pages themselves. Arbitrary parameter/value pairs can be placed in the URL. When the page runs, these will be available to the tests through the object `top.jsUnitParmHash`. The parameters can be accessed as either `top.jsUnitParmHash.`*`parameterName`* or `top.jsUnitParmHash['`*`parameterName`*`']`.

Summary

The Web is a hodgepodge of rapidly evolving technologies, but at its core it is based on three things: a document format (HTML), a method of identifying documents (URIs/URLs), and a network protocol (HTTP) that retrieves those documents. Web browsers retrieve documents from web servers and present them to users. Those documents link to each other through embedded URIs, and browsers can follow those links.

HTML documents can act as forms. The user supplies the requested information, and it is sent back to a web server, which returns yet another document. Originally, all processing had to happen at the server, but this changed with the advent of JavaScript.

JavaScript is a powerful interpreted programming language that runs within the user's web browser. The programs are embedded within web pages; they can communicate across the network to the servers they were loaded from.

Today, most interesting web applications are based on web application frameworks of one sort or another. These offer the programmer a wide variety of services such as data persistence, HTML templating, and session management. Notable Python frameworks are Django, Google App Engine, Pylons, Turbogears, and Zope. These frameworks interface with the underlaying web servers using the Python protocol WSGI.

Web applications should be structured to facilitate unit testing. MVC is a common architectural choice that separates computation from output generation and input processing. In this way, each component can be tested independently of the others.

These components are often tested with different tools. Testing the computation or business logic is no different than with any other program. The input is also amenable to similar treatment. The real difference lies with the output. There are many specialized tools for dealing with the markup that most pages generate. Two useful Python libraries are `xml.etree.ElementTree` and `BeautifulSoup`.

JavaScript testing is the real challenge. It is a second programming environment, and it comes with its own tools. JavaScript programs are application code, so they should be developed with the same disciplines and practices as the rest of your programs. JavaScript implementations vary in important details from browser to browser, so it is necessary to test JavaScript within the target browsers. The most commonly used unit-testing tool is JsUnit. It operates in both stand-alone and distributed mode. Distributed mode has poor Python harness integration, so I only cover the stand-alone mode in this book.

Chapter 11 examines acceptance testing tools. These tools help to define the program's requirements, as well as ensure that the program behaves as expected. The tool I'll demonstrate is PyFit.