Chapter 13

Unit Testing

Unit testing is testing that focuses on small chunks (units) of code, typically individual methods or lines within methods. This is in contrast to most other forms of testing, which consider the system as a whole.

Why focus in so tightly? It's because ultimately all software is constructed in layers; code on one layer relies on the correct operation of the code in the layers below. If this underlying code turns out to contain bugs, then all higher layers are potentially affected. This is a big problem. Fred may write some code with a bug one week, and then you may end up calling it, indirectly, two months later. When your code generates incorrect results, it will take you a while to track down the problem in Fred's method. And when you ask Fred why he wrote it that way, the likely answer will be "I don't remember. That was months ago."

If instead Fred had unit tested his code when he wrote it, two things would have happened. First, he'd have found the bug while the code was still fresh in his mind. Second, because the unit test was only looking at the code he'd just written, when the bug *did* appear, he'd only have to look through a handful of lines of code to find it, rather than doing archaeology on the rest of the code base.

Unit testing helps developers write better code. It helps before the code is actually written, because thinking about testing leads you naturally to create better, more decoupled designs. It helps as you're writing the code, because it gives you instant feedback on how accurate your code is. And it helps after you've written code, both because it gives you the ability to check that the code still works and because it helps others understand how to use your code.

Unit testing is a Good Thing.

But why have a chapter on unit testing in the middle of a book on Ruby? Well, it's because unit testing and languages such as Ruby seem to go hand in hand. The flexibility of Ruby makes writing tests easy, and the tests make it easier to verify that your code is working. Once you get into the swing of it, you'll find yourself writing a little code, writing a test or two, verifying that everything is copacetic, and then writing some more code.

Unit testing is also pretty trivial—run a program that calls part of your application's code, get back some results, and then check the results are what you expected.

Let's say we're testing a Roman number class. So far the code is pretty simple: it just lets us create an object representing a certain number and display that object in Roman numerals:

Download **samples/unittesting_1.rb**

```
# NOTE:  This code has bugs!
class Roman
  MAX_ROMAN = 4999
  def initialize(value)
    if value <= 0 || value > MAX_ROMAN
      fail "Roman values must be > 0 and <= #{MAX_ROMAN}"
    end
    @value = value
  end
  FACTORS = [["m", 1000], ["cm", 900], ["d",  500], ["cd", 400],
             ["c",  100], ["xc",  90], ["l",   50], ["xl",  40],
             ["x",   10], ["ix",   9], ["v",    5], ["iv",   4],
             ["i",    1]]
  def to_s
    value = @value
    roman = ""
    for code, factor in FACTORS
      count, value = value.divmod(factor)
      roman << code unless count.zero?
    end
    roman
  end
end
```

We could test this code by writing another program, like this:

```
require 'roman'
r = Roman.new(1)
fail "'i' expected" unless r.to_s == "i"
r = Roman.new(9)
fail "'ix' expected" unless r.to_s == "ix"
```

However, as the number of tests in a project grows, this kind of ad hoc approach can start to get complicated to manage. Over the years, various unit testing frameworks have emerged to help structure the testing process. Ruby comes with one preinstalled. In Ruby 1.8, this used to be Nathaniel Talbott's Test::Unit framework. Ruby 1.9 instead comes with Ryan Davis' MiniTest.

1.9

MiniTest is largely compatible with Test::Unit but without a lot of bells and whistles (test-case runners, GUI support, and so on). However, because there are areas where it is different and because there are tens of thousands of tests out there that assume the Test::Unit API, Ryan has also added a compatibility layer to MiniTest. For a little bit more information on the differences between the two, see the sidebar on the following page. In this chapter, we'll be using the Test::Unit wrapper, because it automatically runs tests for us. But we'll also be using some of the new assertions available in MiniTest.

MiniTest::Unit vs. Test::Unit

Folks have been using Test::Unit with Ruby for a good number of years now. However, the core team decided to replace the testing framework that comes as standard with Ruby with something a little leaner. Ryan Davis and Eric Hodel wrote MiniTest::Unit as a partial drop-in replacement for Test::Unit.

Most of the assertions in MiniTest mirror those in Test::Unit::TestCase. The major differences are the absence of assert_not_raises and assert_not_throws and the renaming of all the negative assertions. Whereas in Test::Unit you'd say assert_not_nil(x) and assert_not(x), in MiniTest you'd use refute_nil(x) and refute(x).

MiniTest also drops most of the little-used features of Test::Unit, including test cases, GUI runners, and some assertions.

And, probably most significantly, MiniTest does not automatically invoke the test cases when you execute a file that contains them.

So, you have three basic options with this style of unit testing:

- require 'minitest/unit' and use the MiniTest functionality.
- require 'test/unit' and use Minitest with the Test::Unit compatibility layer. This adds in the assertions in Figure 13.2 on page 219 and reenables the autorun functionality.
- You can install the test-unit gem and get all the original Test::Unit functionality back.

The Testing Framework

The Ruby testing framework is basically three facilities wrapped into a neat package:

- It gives you a way of expressing individual tests.
- It provides a framework for structuring the tests.
- It gives you flexible ways of invoking the tests.

Assertions == Expected Results

Rather than have you write series of individual if statements in your tests, the testing framework provides a set of assertions that achieve the same thing. Although a number of different styles of assertion exist, they all follow basically the same pattern. Each assertion gives you a way of specifying a desired result or outcome and a way of passing in the actual outcome.

If the actual doesn't equal the expected, the assertion outputs a nice message and records the fact as a failure.

For example, we could rewrite our previous test of the Roman class using the testing framework. For now, ignore the scaffolding code at the start and end, and just look at the assert_equal methods:

Download **samples/unittesting_3.rb**

```
require 'roman'
require 'test/unit'
class TestRoman < MiniTest::Unit::TestCase
  def test_simple
    assert_equal("i",  Roman.new(1).to_s)
    assert_equal("ix", Roman.new(9).to_s)
  end
end
```

produces:

```
Loaded suite /tmp/prog
Started
.
Finished in 0.000499 seconds.

1 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

The first assertion says that we're expecting the Roman number string representation of 1 to be "i," and the second test says we expect 9 to be "ix." Luckily for us, both expectations are met, and the tracing reports that our tests pass. Let's add a few more tests:

Download **samples/unittesting_4.rb**

```
require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
  def test_simple
    assert_equal("i",   Roman.new(1).to_s)
    assert_equal("ii",  Roman.new(2).to_s)
    assert_equal("iii", Roman.new(3).to_s)
    assert_equal("iv",  Roman.new(4).to_s)
    assert_equal("ix",  Roman.new(9).to_s)
  end
end
```

produces:

```
Loaded suite /tmp/prog
Started
F
Finished in 0.000594 seconds.

  1) Failure:
<"ii"> expected but was
<"i">.
```

```
1 tests, 2 assertions, 1 failures, 0 errors, 0 skips
test_simple(TestRoman) [/tmp/prog.rb:8]:
```

Uh-oh! The second assertion failed. See how the error message uses the fact that the assert knows both the expected and actual values: it expected to get "ii" but instead got "i." Looking at our code, you can see a clear bug in to_s. If the count after dividing by the factor is greater than zero, then we should output that many Roman digits. The existing code outputs just one. The fix is easy:

Download **samples/unittesting_5.rb**

```
def to_s
  value = @value
  roman = ""
  for code, factor in FACTORS
    count, value = value.divmod(factor)
    roman << (code * count)
  end
  roman
end
```

Now let's run our tests again:

```
Loaded suite /tmp/prog
Started
.
Finished in 0.000462 seconds.

1 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

Looking good. We can now go a step further and remove some of that duplication:

Download **samples/unittesting_7.rb**

```
require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
  NUMBERS = [
    [ 1, "i" ],  [ 2, "ii" ],  [ 3, "iii" ],
    [ 4, "iv"],  [ 5, "v"  ],  [ 9, "ix"  ]
  ]
  def test_simple
    NUMBERS.each do |arabic, roman|
      r = Roman.new(arabic)
      assert_equal(roman, r.to_s)
    end
  end
end
```

produces:

```
Loaded suite /tmp/prog
Started
.
Finished in 0.000469 seconds.
```

```
1 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

What else can we test? Well, the constructor checks that the number we pass in can be represented as a Roman number, throwing an exception if it can't. Let's test the exception:

Download **samples/unittesting_8.rb**

```
require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
  def test_range
    # no exception for these two...
    Roman.new(1)
    Roman.new(4999)
    # but an exception for these
    assert_raises(RuntimeError) { Roman.new(0) }
    assert_raises(RuntimeError) { Roman.new(5000) }
  end
end
```

produces:

```
Loaded suite /tmp/prog
Started
.
Finished in 0.000583 seconds.

1 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

We could do a lot more testing on our Roman class, but let's move on to bigger and better things. Before we go, though, we should say that we've only scratched the surface of the set of assertions available inside the testing framework. For example, for every positive assertion, such as assert_equal, there's a negative refutation (in this case refute_equal). Figure 13.2 on page 219 lists the additional assertions you get if you load the Test::Unit shim (which we do in this chapter), and Figure 13.1 on page 218 gives a full list of the MiniTest assertions.

The final parameter to every assertion is a message that will be output before any failure message. This normally isn't needed, because the failure messages are normally pretty reasonable. The one exception is the test refute_nil (or assert_not_nil in Test::Unit), where the message "Expected nil to not be nil" doesn't help much. In that case, you may want to add some annotation of your own. (This code assumes the existence of some kind of User class.)

Download **samples/unittesting_9.rb**

```
require 'test/unit'
class ATestThatFails < Test::Unit::TestCase
  def test_user_created
    user = User.find(1)
    refute_nil(user, "User with ID=1 should exist")
  end
end
```

produces:

```
Loaded suite /tmp/prog
Started
F
Finished in 0.000568 seconds.

  1) Failure:
User with ID=1 should exist.
Expected nil to not be nil.

1 tests, 1 assertions, 1 failures, 0 errors, 0 skips
test_user_created(ATestThatFails) [/tmp/prog.rb:10]:
```

Structuring Tests

Earlier we asked you to ignore the scaffolding around our tests. Now it's time to look at it.

You include the testing framework facilities in your unit test with either this:

```
require 'test/unit'
```

or, for raw MiniTest, with this:

```
require 'minitest/unit'
```

Unit tests seem to fall quite naturally into high-level groupings, called *test cases*, and lower-level groupings, the test methods themselves. The test cases generally contain all the tests relating to a particular facility or feature. Our Roman number class is fairly simple, so all the tests for it will probably be in a single test case. Within the test case, you'll probably want to organize your assertions into a number of test methods, where each method contains the assertions for one type of test; one method could check regular number conversions, another could test error handling, and so on.

The classes that represent test cases must be subclasses of Test::Unit::TestCase. The methods that hold the assertions must have names that start with test. This is important: the testing framework uses reflection to find tests to run, and only methods whose names start with test are eligible.

Quite often you'll find all of the test methods within a test case start by setting up a particular scenario. Each test method then probes some aspect of that scenario. Finally, each method may then tidy up after itself. For example, we could be testing a class that extracts jukebox playlists from a database:

Download **samples/unittesting_12.rb**

```
require 'test/unit'
require 'dbi'
require 'playlist_builder'
class TestPlaylistBuilder < Test::Unit::TestCase
  def test_empty_playlist
    db = DBI.connect('DBI:mysql:playlists')
    pb = PlaylistBuilder.new(db)
    assert_empty(pb.playlist)
```

```
    db.disconnect
  end
  def test_artist_playlist
    db = DBI.connect('DBI:mysql:playlists')
    pb = PlaylistBuilder.new(db)
    pb.include_artist("krauss")
    refute_empty(pb.playlist, "Playlist shouldn't be empty")
    pb.playlist.each do |entry|
      assert_match(/krauss/i, entry.artist)
    end
    db.disconnect
  end
  def test_title_playlist
    db = DBI.connect('DBI:mysql:playlists')
    pb = PlaylistBuilder.new(db)
    pb.include_title("midnight")
    refute_empty(pb.playlist, "Playlist shouldn't be empty")
    pb.playlist.each do |entry|
      assert_match(/midnight/i, entry.title)
    end
    db.disconnect
  end
  # ...
end
```

produces:

```
Loaded suite /tmp/prog
Started
...
Finished in 0.000629 seconds.

3 tests, 46 assertions, 0 failures, 0 errors, 0 skips
```

Each test starts by connecting to the database and creating a new playlist builder. Each test ends by disconnecting from the database. (The idea of using a real database in unit tests is questionable, because unit tests are supposed to be fast running, context independent, and easy to set up, but it illustrates a point.)

We can extract all this common code into *setup* and *teardown* methods. Within a TestCase class, a method called setup will be run before each and every test method, and a method called teardown will be run after each test method finishes. Let's emphasize that: the setup and teardown methods bracket each test, rather than being run once per test case. Our test would then become this:

Download **samples/unittesting_13.rb**

```
require 'test/unit'
require 'dbi'
require 'playlist_builder'
class TestPlaylistBuilder < Test::Unit::TestCase
  def setup
    @db = DBI.connect('DBI:mysql:playlists')
```

```
    @pb = PlaylistBuilder.new(@db)
  end
  def teardown
    @db.disconnect
  end
  def test_empty_playlist
    assert_empty(@pb.playlist)
  end
  def test_artist_playlist
    @pb.include_artist("krauss")
    refute_empty(@pb.playlist, "Playlist shouldn't be empty")
    @pb.playlist.each do |entry|
      assert_match(/krauss/i, entry.artist)
    end
  end
  def test_title_playlist
    @pb.include_title("midnight")
    refute_empty(@pb.playlist, "Playlist shouldn't be empty")
    @pb.playlist.each do |entry|
      assert_match(/midnight/i, entry.title)
    end
  end
  # ...
end
```

produces:

```
Loaded suite /tmp/prog
Started
...
Finished in 0.000619 seconds.

3 tests, 46 assertions, 0 failures, 0 errors, 0 skips
```

Inside the teardown method, you can detect whether the preceding test succeeded with the passed? method.

Organizing and Running Tests

The test cases we've shown so far are all runnable Test::Unit programs. If, for example, the test case for the Roman class was in a file called test_roman.rb, we could run the tests from the command line using this:

```
% ruby test_roman.rb
Loaded suite test_roman
Started
..
Finished in 0.000883 seconds.

2 tests, 7 assertions, 0 failures, 0 errors, 0 skips
```

Test::Unit is clever enough to run the tests even though there's no main program. It collects all the test case classes and runs each in turn.

If we want, we can ask it to run just a particular test method:

```
% ruby test_roman.rb -n test_range
Loaded suite test_roman
Started
.
Finished in 0.000600 seconds.

1 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

or tests whose names match a regular expression:

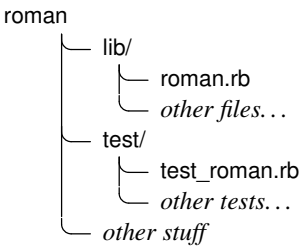
```
% ruby test_roman.rb -n /range/
Loaded suite test_roman
Started
.
Finished in 0.001036 seconds.

1 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

This last capability is a great way of grouping your tests. Use meaningful names, and you'll be able to run (for example) all the shopping-cart-related tests by simply running tests with names matching /cart/.

Where to Put Tests

Once you get into unit testing, you may well find yourself generating almost as much test code as production code. All of those tests have to live somewhere. The problem is that if you put them alongside your regular production code source files, your directories start to get bloated—effectively you end up with two files for every production source file.

A common solution is to have a `test/` directory where you place all your test source files. This directory is then placed parallel to the directory containing the code you're developing. For example, for our Roman numeral class, we may have this:



This works well as a way of organizing files but leaves you with a small problem: how do you tell Ruby where to find the library files to test? For example, if our TestRoman test code was in a `test/` subdirectory, how does Ruby know where to find the `roman.rb` source file, the thing we're trying to test?

An option that *doesn't* work reliably is to build the path into require statements in the test code and run the tests from the `test/` subdirectory:

```
require 'test/unit'
require '../lib/roman'
class TestRoman < Test::Unit::TestCase
  # ...
end
```

Why doesn't it work? Because our `roman.rb` file may itself require other source files in the library we're writing. It'll load them using require (without the leading ../lib/), and because they aren't in Ruby's $LOAD_PATH, they won't be found. Our test just won't run. A second, less immediate problem is that we won't be able to use these same tests to test our classes once installed on a target system, because then they'll be referenced simply using require 'roman'.

A better solution is to assume that your Ruby program is packaged according to the conventions we'll be discussing in Section 16 on page 251. In this arrangement, the top-level directory of your application is assumed to be in Ruby's load path by all other components of the application. Given that, your unit tests can assume that they can find the components they are testing using the path lib/xxx.rb.

Your test code would then be as follows:

```
require 'test/unit'
require 'lib/roman'
class TestRoman < Test::Unit::TestCase
  # ...
end
```

And you'd run it using this:

```
% ruby -I path/to/app path/to/app/test/test_roman.rb
```

The normal case, where you're already in the application's directory, would be as follows:

```
% ruby -I . test/test_roman.rb
```

This would be a good time to investigate using Rake to automate your testing....

Test Suites

After a while, you'll grow a decent collection of test cases for your application. You may well find that these tend to cluster: one group of cases tests a particular set of functions, and another group tests a different set of functions. If so, you can group those test cases together into *test suites*, letting you run them all as a group.

This is easy to do—just create a Ruby file that requires `test/unit` and then requires each of the files holding the test cases you want to group. This way, you build yourself a hierarchy of test material.

- You can run individual tests by name.
- You can run all the tests in a file by running that file.

- You can group a number of files into a test suite and run them as a unit.
- You can group test suites into other test suites.

This gives you the ability to run your unit tests at a level of granularity that you control, testing just one method or testing the entire application.

At this point, it's worthwhile to think about naming conventions. Nathaniel Talbott, the author of Test::Unit, uses the convention that test cases are in files named tc_*xxx* and test suites are in files named ts_*xxx*. Most people seem to use test_ as the test-case filename prefix:

```
# file ts_dbaccess.rb
require 'test/unit'
require 'test_connect'
require 'test_query'
require 'test_update'
require 'test_delete'
```

Now, if you run Ruby on the file ts_dbaccess.rb, you execute the test cases in the four files you've required.

RSpec and Shoulda

The built-in testing framework has a lot going for it. It is simple, and it is compatible in style with frameworks from other languages (such as JUnit for Java and NUnit for C#).

However, there's a growing movement in the Ruby community to use a different style of testing. So-called behavior-driven development encourages people to write tests in terms of your expectations of the program's behavior in a given set of circumstances. In many ways, this is like testing according to the content of *user stories*, a common requirements-gathering technique in agile methodologies. With these testing frameworks, the focus is not on assertions. Instead, you write expectations.

Although both RSpec and Shoulda allow this style of testing, they focus on different things. RSpec is very much concerned with driving the design side of things. You can write and execute specs with RSpec well before you've written a line of application code. These specs, when run, will output the user stories that describe your application. Then, as you fill in the code, the specs mutate into tests that validate that your code meets your expectations.

Shoulda, on the other hand, is really more focused on the testing side. Whereas RSpec is a complete framework, Shoulda works inside Test::Unit—you can even mix Shoulda tests with regular Test::Unit test methods.

Let's start with a simple example of RSpec in action.

Starting to Score Tennis Matches

The scoring system used in lawn tennis originated in the middle ages. As players win successive points, their scores are shown as 15, 30, and 40. The next point is a win unless your

opponent also has 40. If you're both tied at 40, then different rules apply—the first player with a clear two-point advantage is the winner.[1]

We're tasked with writing a class that handles this scoring system. Let's use RSpec specifications to drive the process. We install RSpec with gem install rspec. We'll then create our first specification file:

Download samples/unittesting_20.rb

```
describe "TennisScorer", "basic scoring" do
  it "should start with a score of 0-0"
  it "should be 15-0 if the server wins a point"
  it "should be 0-15 if the receiver wins a point"
  it "should be 15-15 after they both win a point"
  # ...
end
```

This file contains nothing more than a description of an aspect of the tennis scoring class (that we haven't yet written, by the way). It contains a description of the basic scoring system. Inside the description are a set of four expectations (it "should start..." and so on). We can run this specification using the spec command:

```
$ spec ts_spec.rb
```

produces:

```
****

Pending:

TennisScorer basic scoring should start with a score of 0-0 (Not Yet
 Implemented)
ts_spec.rb:2:in `block in <top (required)>'

TennisScorer basic scoring should be 15-0 if the server wins a point
 (Not Yet Implemented)
ts_spec.rb:3:in `block in <top (required)>'

TennisScorer basic scoring should be 0-15 if the receiver wins a point
 (Not Yet Implemented)
ts_spec.rb:4:in `block in <top (required)>'

TennisScorer basic scoring should be 15-15 after they both win a point
 (Not Yet Implemented)
ts_spec.rb:5:in `block in <top (required)>'

Finished in 0.038935 seconds

4 examples, 0 failures, 4 pending
```

1. Some say the 0, 15, 30, 40 system is a corruption of the fact that scoring used to be done using the quarters of a clock face. Me, I just think those medieval folks enjoyed a good joke.

That's pretty cool. Executing the tests echoes our expectations back at us, telling us that each has yet to be implemented. Coding, like life, is full of these disappointments. However, unlike life, fixing things is just a few keystrokes away. Let's start by meeting the first expectation—when a game starts, the score should be 0 to 0. We'll start by fleshing out the test:

Download **samples/unittesting_22.rb**

```
require "tennis_scorer"
describe TennisScorer do
  it "should start with a score of 0-0" do
    ts = TennisScorer.new
    ts.score.should == "0-0"
  end
  it "should be 15-0 if the server wins a point"
  it "should be 0-15 if the receiver wins a point"
  it "should be 15-15 after they both win a point"
end
```

Note that we've assumed we have a class TennisScorer in a file called tennis_scorer.rb. Our first expectation now has a code block associated with it. Inside that block, we create a TennisScorer and then use a funky RSpec syntax to validate that the score starts out at 0 to 0. This particular aspect of RSpec probably generates the most controversy—some people love it, others find it awkward. Either way, ts.score.should == "0-0" is basically the same as an assertion in Test::Unit.

We'll beef up our TennisScorer class, but only enough to let it satify this assertion:

Download **samples/unittesting_23.rb**

```
class TennisScorer
  def score
    "0-0"
  end
end
```

Well run our spec again:

```
$ spec ts_spec.rb
```

produces:

```
.***

Pending:

TennisScorer should be 15-0 if the server wins a point (Not Yet
 Implemented)
ts_spec.rb:9:in `block in <top (required)>'

TennisScorer should be 0-15 if the receiver wins a point (Not Yet
 Implemented)
ts_spec.rb:10:in `block in <top (required)>'
```

```
TennisScorer should be 15-15 after they both win a point (Not Yet
 Implemented)
ts_spec.rb:11:in `block in <top (required)>'

Finished in 0.015241 seconds

4 examples, 0 failures, 3 pending
```

Note that we now have three pending expectations; the first one has been satisfied.

Let's flesh out the next expectation:

Download **samples/unittesting_25.rb**

```
require "tennis_scorer"
describe TennisScorer, "basic scoring" do
  it "should start with a score of 0-0" do
    ts = TennisScorer.new
    ts.score.should == "0-0"
  end
  it "should be 15-0 if the server wins a point" do
    ts = TennisScorer.new
    ts.give_point_to(:server)
    ts.score.should == "15-0"
  end


  it "should be 0-15 if the receiver wins a point"
  it "should be 15-15 after they both win a point"
end
```

This won't run, because our TennisScorer class doesn't implement a give_point_to method. Let's rectify that. Our code isn't finished, but it lets the test pass:

Download **samples/unittesting_26.rb**

```
class TennisScorer
  OPPOSITE_SIDE_OF_NET = {
    :server => :receiver,
    :receiver => :server
  }
  def initialize
    @score = { :server => 0, :receiver => 0 }
  end
  def score
    "#{@score[:server]*15}-#{@score[:receiver]*15}"
  end
  def give_point_to(player)
    other = OPPOSITE_SIDE_OF_NET[player]
    fail "Unknown player #{player}" unless other
    @score[player] += 1
  end
end
```

Again, we'll run the specification:

```
$ spec ts_spec.rb
```

produces:

```
..**

Pending:

TennisScorer basic scoring should be 0-15 if the receiver wins a point
 (Not Yet Implemented)
ts_spec.rb:16:in `block in <top (required)>'

TennisScorer basic scoring should be 15-15 after they both win a point
 (Not Yet Implemented)
ts_spec.rb:17:in `block in <top (required)>'

Finished in 0.016136 seconds

4 examples, 0 failures, 2 pending
```

We're now meeting two of the four initial expectations. But, before we move on, note there's a bit of duplication in the specification: both our expections create a new TennisScorer object. We can fix that by using a before stanza in the specification. This works a bit like the setup method in Test::Unit, allowing us to run code before expecations are executed. Let's use this feature and, at the same time, build out the last two expectations:

Download **samples/unittesting_28.rb**

```
require "tennis_scorer"
describe TennisScorer, "basic scoring" do
  before(:each) do
    @ts = TennisScorer.new
  end
  it "should start with a score of 0-0" do
    @ts.score.should == "0-0"
  end
  it "should be 15-0 if the server wins a point" do
    @ts.give_point_to(:server)
    @ts.score.should == "15-0"
  end
  it "should be 0-15 if the receiver wins a point" do
    @ts.give_point_to(:receiver)
    @ts.score.should == "0-15"
  end
  it "should be 15-15 after they both win a point" do
    @ts.give_point_to(:receiver)
    @ts.give_point_to(:server)
    @ts.score.should == "15-15"
  end
end
```

Let's run it:

```
$ spec ts_spec.rb
```

produces:

```
....

Finished in 0.016362 seconds

4 examples, 0 failures
```

We're going to stop here, but I suggest that you might want to take this code and continue to develop it. Write expectations such as these:

Download **samples/unittesting_30.rb**

```
it "should be 40-0 after the server wins three points"
it "should be W-L after the server wins four points"
it "should be L-W after the receiver wins four points"
it "should be Deuce after each wins three points"
it "should be A-server after each wins three points and the server
gets one more"
it "should be A-receiver after each wins three points and the receiver
gets one more"
```

and so on. Note that none of these expectations is met by our current implementation.

RSpec has a lot more depth than just the description of expectations. In particular, it has an entire language for describing and running complete user stories. But that's beyond the scope of this book.

Anyone for Shoulda?

RSpec is testing with attitude. On the other hand, Shoulda takes many of the ideas from RSpec and humbly offers them to you for integration into your regular unit tests. For many developers, particularly those with existing Test::Unit tests, this is a good compromise. You get much of the descriptive power of RSpec-style expectations without having to commit to the full framework.

Install Shoulda using this:

```
% gem install thoughtbot-shoulda --source=http://gems.github.com
```

Then, unlike RSpec, write a regular Test::Unit test case. Inside it, though, you can use the Shoulda mini-language to describe your tests.

Let's recast our final RSpec tennis scoring tests using Shoulda:

Download **samples/unittesting_31.rb**

```
require 'rubygems'
require 'test/unit'
require 'shoulda'
require 'tennis_scorer.rb'
class TennisScorerTest < Test::Unit::TestCase
  def assert_score(target)
    assert_equal(target, @ts.score)
  end
  context "Tennis scores" do
    setup do
      @ts = TennisScorer.new
    end
    should "start with a score of 0-0" do
      assert_score("0-0")
    end
    should "be 15-0 if the server wins a point" do
      @ts.give_point_to(:server)
      assert_score("15-0")
    end
    should "be 0-15 if the receiver wins a point" do
      @ts.give_point_to(:receiver)
      assert_score("0-15")
    end
    should "be 15-15 after they both win a point" do
      @ts.give_point_to(:receiver)
      @ts.give_point_to(:server)
      assert_score("15-15")
    end
  end
end
```

```
$ ruby ts_spec.rb
```

produces:

```
Loaded suite ts_shoulda
Started
....
Finished in 0.000689 seconds.

4 tests, 4 assertions, 0 failures, 0 errors, 0 skips
```

Behind the scenes, Shoulda is creating Test::Unit test methods for each should block in your tests. This is why we can use regular test::Unit assertions in Shoulda code. But Shoulda also works hard to maintain the right context for our tests. For example, I can nest contexts and their setup blocks, allowing me to have some initialization that's common to all tests and some that's common to just a subset. We can apply this to our tennis example. We'll write nested contexts and put setup blocks at each level. When Shoulda executes our tests, it runs all the appropriate setup blocks for the should blocks.

Download **samples/unittesting_33.rb**

```
require 'rubygems'
require 'test/unit'
require 'shoulda'
require 'tennis_scorer.rb'
class TennisScorerTest < Test::Unit::TestCase
  def assert_score(target)
    assert_equal(target, @ts.score)
  end
  context "Tennis scores" do
    setup do
      @ts = TennisScorer.new
    end
    should "start with a score of 0-0" do
      assert_score("0-0")
    end
    context "where the server wins a point" do
      setup do
        @ts.give_point_to(:server)
      end
      should "be 15-0" do
        assert_score("15-0")
      end
      context "and the oponent wins a point" do
        setup do
          @ts.give_point_to(:receiver)
        end
        should "be 15-15" do
          assert_score("15-15")
        end
      end
    end
    should "be 0-15 if the receiver wins a point" do
      @ts.give_point_to(:receiver)
      assert_score("0-15")
    end
  end
end
```

Let's run it:

```
$ ruby ts_spec.rb
```

produces:

```
Loaded suite ts_shoulda_1
Started
....
Finished in 0.000806 seconds.

4 tests, 4 assertions, 0 failures, 0 errors, 0 skips
```

Would I use these nested contexts for this tennis scoring example? I probably wouldn't as it stands, because the linear form is easier to read. But I use them all the time when I have tests where I want to run through a complex and building scenario. This nesting lets me set up an environment, run some tests, then change the environment, run more tests, change it again, run even more tests, and so on. It ends up making tests far more compact and removes a lot of duplication.

Figure 13.1. Testing Framework Assertions

assert | refute(*boolean*, [*message*])
Fails if *boolean* is (is not) false or nil.

assert_block { *block* }
Expects the block to return true.

assert_ | refute_empty(*collection*, [*message*])
Expects empty? on *collection* to return true (false).

assert_ | refute_equal(*expected*, *actual*, [*message*])
Expects *actual* to equal/not equal *expected*, using ==.

assert_ | refute_in_delta(*expected_float*, *actual_float*, *delta*, [*message*])
Expects that the actual floating-point value is (is not) within *delta* of the expected value.

assert_ | refute_in_epsilon(*expected_float*, *actual_float*, *epsilon=0.001*, [*message*])
Calculates a delta value as *epsilon * min(expected, actual)*, then calls the _in_delta test.

assert_ | refute_includes(*collection*, *obj*, [*message*])
Expects include?(obj) on *collection* to return true (false).

assert_ | refute_instance_of(*klass*, *obj*, [*message*])
Expects *obj* to be (not to be) a instance of *klass*.

assert_ | refute_kind_of(*klass*, *obj*, [*message*])
Expects *obj* to be (not to be) a kind of *klass*.

assert_ | refute_match(*regexp*, *string*, [*message*])
Expects *string* to (not) match *regexp*.

assert_ | refute_nil(*obj*, [*message*])
Expects *obj* to be (not) nil.

assert_ | refute_operator(*obj1*, *operator*, *obj2*, [*message*])
Expects the result of sending the message *operator* to *obj1* with parameter *obj2* to be (not to be) true.

assert_raises(*Exception*, . . .) { *block* }
Expects the block to raise one of the listed exceptions.

assert_ | refute_respond_to(*obj*, *message*, [*message*])
Expects *obj* to respond to (not respond to) *message* (a symbol).

assert_ | refute_same(*expected*, *actual*, [*message*])
Expects *expected*.equal?(*actual*).

assert_send(*send_array*, [*message*])
Sends the message in *send_array[1]* to the receiver in *send_array[0]*, passing the rest of *send_array* as arguments. Expects the return value to be true.

assert_throws(*expected_symbol*, [*message*]) { *block* }
Expects the block to throw the given symbol.

flunk(*message*="Epic Fail!")
Always fails.

skip(*message*)
Indicates that a test is deliberately not run.

pass
Always passes.

Figure 13.2. Additional Test::Unit Assertions

assert_not_equal(*expected*, *actual*, [*message*])
: Expects *actual* not to equal *expected*, using ==. Like refute_equal.

assert_not_match(*regexp*, *string*, [*message*])
: Expects *string* not to match *regexp*. Like refute_match.

assert_not_nil(*obj*, [*message*])
: Expects *obj* not to be nil. Like refute_nil.

assert_not_same(*expected*, *actual*, [*message*])
: Expects !*expected*.equal?(*actual*). Like refute_same.

assert_nothing_raised(*Exception*, ...) { *block* }
: Expects the block not to raise one of the listed exceptions.

assert_nothing_thrown(*expected_symbol*, [*message*]) { *block* }
: Expects the block not to throw the given symbol.

assert_raise(*Exception*, ...) { *block* }
: Synonym for assert_raises.