

# Test-first programming with Ruby

Skill Level: Intermediate

Pat Eyer ([pat.eyer@gmail.com](mailto:pat.eyer@gmail.com))

Freelance Writer

Seattle Ruby Brigade

24 May 2005

This tutorial looks at building Ruby applications following test-first principles. It starts with an overview of the Test::Unit library that ships with Ruby, then moves on to writing tests from a specification and writing code to fulfill those tests. It touches on tools like ZenTest and unit\_diff, as well as on the process of refactoring, and is built around a single programming example.

## Section 1. Before you start

### About this tutorial

Test-first programming is a promising extension of agile programming methodologies that let you write code with more confidence, refactor easily, and readily adapt your code to new requirements. This tutorial gets you started with the test-first programming model using the Ruby programming language. After you've completed the tutorial, you should be well on your way to producing robust, functional code more quickly and with more confidence.

The code examples in this tutorial come from the r43 library, a wrapper around the 43 Things API (see [Resources](#)).

This tutorial is dedicated to the memory Masuru Ishii -- a Japanese Rubyist and the father of unit testing in Ruby. He died in the 25 April 2005 rail disaster in Amagasaki, Japan.

## Prerequisites

To complete this tutorial, you need:

- A working installation of Ruby (V1.8.2 is preferred).
- A basic understanding of the Ruby programming language, including the ability to write small programs.

If you're on a Linux® or Mac OS X machine, you're almost certainly covered for the first prerequisite. If your distribution doesn't ship with Ruby, it's easy to build from source (see [Resources](#)), but you should let the distributor know that you want it in the next release. If you're working on a Microsoft® Windows® system, it has a nice one-click installer (see [Resources](#)).

If you don't already know Ruby, [Resources](#) contains links to tutorials, books, and other literature to get you started. You can probably learn enough just by completing this tutorial, though. Ruby is very easy to read.

---

## Section 2. Test-first programming with Test::Unit

### How unit testing works

*Unit testing* is a method of testing your code in chunks -- typically methods. You write individual *tests*, which you collect into *test cases*. You bundle these test cases into a *test suite*. Because the tests are small and run quickly, you can run them frequently to ensure that your code works correctly. Most *test runners* (an interface for running your tests) even allow you to select a subset of your tests to run instead of running every test every time.

Basically, unit testing helps you write code more quickly and with more confidence. You write smaller units of code at a time, reducing the number of bugs you introduce. You see the bugs immediately as a failing test and know exactly where to look for them in the code.

It might seem backwards, but coding test first really does mean that you write your tests before you start writing code. Doing so gives you some boundaries for your coding: You know what to write and when to stop writing code. Test::Unit actually does most of the heavy lifting to make this process work. It creates the test runner and collects the tests into cases and suites for you.

## What should you test?

The key to good unit testing is to test enough of your code to catch problems. Writing too many tests pulls you into an unending cycle of writing and running tests -- you'll never get to code, and you don't want that.

A common answer to the question "What should I test?" is *anything that can fail*. A better rule of thumb might be: Test anything that's likely to fail. You want to write tests to ensure that your method does what it's supposed to do with normal input. You should check that invalid input will be handled correctly. Finally, test your boundary conditions -- the extreme edges of your expected input.

You should also make sure that you mine your bug reports for tests. Writing tests before fixing bugs confirms that you really understand -- and can repeat -- the bug. In addition, having the new test in your test suite ensures that you won't inadvertently reintroduce the same bug later.

Sometimes you find that you need to test dynamic data. Doing so can make repeatable results difficult to come by and testing nearly impossible. If you find yourself in this situation, pay particular attention to [Mock test fixtures](#) later in this tutorial.

## Build tests from your ideas

A helpful trick for building good tests is to reduce your ideas to writing. Whether you want to call these ideas *use cases* or *stories*, the informal specification you write will help you see which tests you need to write. For example, as I started writing `r43`, I saw that I'd need to have a `Response` object. I could write some initial stories like this:

- When a connection can't be made, the `Response` object should indicate the failure by making the `status_code` attribute *nil*.
- When the connection is made, but doesn't result in a valid response, the `status_code` attribute should be set to the HTTP response code.
- When the connection is made and the response is valid, the `status_code` attribute should be set to `200`, and the content attributes should be set with valid data.

Each of these stories derives cleanly from the original idea, and each one is easily testable. After you've reduced your idea to this level, you're ready to start writing tests -- then you can get on to writing code.

---

## Section 3. Setting up unit testing

### Build a test suite

Building a test suite is easy with `Test::Unit`. Listing 1 shows the initial suite of tests for `r43`, your sample application. (Note that the line numbers are not a part of Ruby and are for convenience only.) Put all this information into a file called `ts_r43.rb`.

#### Listing 1. r43 test suites

```
1 require 'test/unit'
2 require 'r43'

3 class TestR43 <
  Test::Unit::TestCase

4   def test_key
5     connection = R43.new('1234')
6
7     assert_equal('1234',connection.key)
8   end

8 end
```

Let's step through the example and see what needs to be done. Lines 1 and 2 pull in the libraries you need: `Test::Unit` (line 1) and `r43` (line 2). Lines 3-8 define the first `TestCase`, a class for testing `r43` objects. In line 3, you're making the new class a subclass of `Test::Unit::TestCase`. Lines 4-7 define your first test, the `test_key` method of the `TestR43` class.

The test itself consists of a bit of fixture code -- creating a `connection` object -- and an assertion (I talk more about assertions later). This assertion just says, "The result of calling the `key` method on the `connection` object is equal to the string `1234`."

### Run the tests

Because you're writing your application test first, you've written these tests before you've written any code. The test suite actually forms a Ruby program on its own. When you try to run it for the first time, it will fail:

```
$ ruby ts_43.rb
ts_r43.rb:2:in 'require': No such file to load -- r43 (LoadError)
from ts_r43.rb:2
```

```
$
```

Fortunately, this problem is easy to fix. Simply create an empty `r43.rb` file so you can load it when you run your tests. Running the test again gives:

```
$ touch r43.rb
$ ruby ts_r43.rb
Loaded suite ts_r43
Started
E
Finished in 0.00051 seconds.

1) Error:
test_key(TestR43):
NameError: uninitialized constant TestR43::R43
ts_r43.rb:6:in 'test_key'

1 tests, 0 assertions, 0 failures, 1 errors
$
```

Better, but still not quite what you want. This error message means that your test has no `R43` class to use. So, define the `R43` class as follows:

```
1 class R43
2 end
```

Running your test suite again gives:

```
Loaded suite ts_r43
Started
E
Finished in 0.000449 seconds.

1) Error:
test_key(TestR43):
ArgumentError: wrong number of arguments(1 for 0)
ts_r43.rb:6:in 'initialize'
ts_r43.rb:6:in 'new'
ts_r43.rb:6:in 'test_key'

1 tests, 0 assertions, 0 failures, 1 errors
$
```

To fix these errors, you're going to have to write some actual code. Before I get to that code, though, you need to understand the assertions that you can test with.

## Catalog of assertions

Now that you know what a test suite looks like and what kind of output you can expect from it, let's take a look at what kinds of assertions exist for you to test with. Nearly every assertion takes an optional message string, which is used to provide

more information in the case of test failure.

The first group of assertions covers the values returned by the method under test:

- `assert(boolean, [message])`
- `assert_equal(expected, actual, [message])`
- `assert_not_equal(expected, actual, [message])`
- `assert_in_delta(expected_float, actual_float, delta, [message])`
- `assert_match(pattern, string, [message])`
- `assert_no_match(regex, string, [message])`
- `assert_same(expected, actual, [message])`
- `assert_not_same(expected, actual, [message])`

The second group of assertions tests which type of object you're dealing with:

- `assert_nil(object, [message])`
- `assert_not_nil(object, [message])`
- `assert_instance_of(klass, object, [message])`
- `assert_kind_of(klass, object, [message])`

The third group of assertions covers an object's ability to respond to a method or operator:

- `assert_operator(object1, operator, object2, [message])`
- `assert_respond_to(object, method, [message])`
- `assert_send([obj, meth, *args], [message])`

These assertions all deal with exception handling:

- `assert_raise(*args) {|| ...}`
- `assert_nothing_raised(*args) {|| ...}`
- `assert_throws(expected_symbol, [message], &proc)`
- `assert_nothing_thrown([message], &proc)`

Because it is so easy to extend existing Ruby classes, some libraries even provide

additional assertions. A great example is the Rails-specific assertions distributed with the Ruby on Rails framework (see [Resources](#)).

---

## Section 4. Writing Ruby code

### Correct errors from the initial test

All right, you've figured out how to build test suites and how to write the correct tests based on your specifications, so let's get back to your initial test suite. When you stepped away from the test suite, it still generated errors because there wasn't a real *constructor*. You can fix that by writing an initializer (or constructor) for the object.

```
1 class R43
2   def initialize(key)
3     @key = key
4   end
5 end
```

Lines 2 and 3 make up the initializer method. This method takes an argument (called `key`) and sets an instance variable (called `@key`) to its value. When you run the test suite again, you get:

```
$ ruby ts_r43.rb
Loaded suite ts_r43
Started
E
Finished in 0.000527 seconds.

1) Error:
test_key(TestR43):
NoMethodError: undefined method \
'key' for #<R43:0xb73e8ba4 @key="1234">
ts_r43.rb:7:in 'test_key'

1 tests, 0 assertions, 0 failures, 1 errors
$
```

You're almost there. The next step is to create the `getter` method `key`. Ruby makes this process easy. Your whole `getter` is shown on line 2:

```
1 class R43
2   attr_reader :key
```

```
3 def initialize(key)
4   @key = key
5 end
6 end
```

You've written just enough code to pass your first test. The `R43` class can be initialized (with a key) and has a `getter` method that returns the key. Now when you run your test suite, you see:

```
$ ruby ts_r43.rb
Loaded suite ts_r43
Started
.
Finished in 0.000457 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
$
```

Everything passed! (Not really a surprise at this point, but it's a nice feeling anyway.) Check your tests and code into revision control.

## Write additional tests

Now that you've got some working code (it passes all the tests), consider writing some additional tests to make sure your code doesn't fail. This method is simple, but what happens if you pass in a key that's not a `String` object? If it's a `Fixnum` object, you can just convert it to a `String` object. If it's anything else, your constructor should return `nil`. (The right way to handle this situation is probably by raising an exception, but that's beyond the scope of this tutorial.) Here's what your new tests should look like:

```
1 def test_key
2   connection = R43.new('1234')
3   assert_equal('1234',connection.key)
4   connection = R43.new(1234)
5   assert_equal('1234',connection.key)
6   connection = R43.new([1234])
7   assert_nil(connection.key)
8 end
```

You know that this code will fail, but run it anyway. You should see the following output:

```
$ ruby ts_r43.rb
Loaded suite ts_r43
Started
F
Finished in 0.019398 seconds.
```

```
1) Failure:
test_key(TestR43) [ts_r43.rb:9]:
<"1234"> expected but was
<1234>.

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

Now that you've got failing tests again (or are getting a *red light*, in unit-testing jargon), you can start writing code.

## Failures vs. errors

The report from your last test run points out some new information. First of all, you get to see what a failure looks like, instead of an error. This failure is significant because it means that your test and code work, but not correctly. You've got a bug. Second, you see that the report shows only two assertions, even though you wrote three. The report shows only two assertions because `Test::Unit` won't continue past the first failure. (That way, you can focus on getting your code right one test at a time.)

To fix the first failure, ensure that `Fixnum` objects are converted to `String` objects by changing your code as follows:

```
1 def initialize(key)
2   @key = key
3   if @key.class == Fixnum then
4     @key = @key.to_s
5   end
6 end
```

Output generated by re-running your test suite indicates that you've fixed the first of two bugs:

```
$ ruby ts_r43.rb
Loaded suite ts_r43.rb
Started
F
Finished in 0.007873 seconds.

1) Failure:
test_key(TestR43) [ts_r43.rb:11]:
<"1234"> expected but was
<[1234]>.

1 tests, 3 assertions, 1 failures, 0 errors
$
```

Now you can fix the next failure with the code by converting arrays to strings with the `to_s` method:

```
1 def initialize(key)
2   @key = key
3   if @key.class == Fixnum then
4     @key = @key.to_s
5   end
6   if @key.class != String then
7     @key = nil
8   end
9 end
```

Now your code passes all your tests:

```
$ ruby ts_r43.rb
Loaded suite ts_r43.rb
Started
.
Finished in 0.000455 seconds.

1 tests, 3 assertions, 0 failures, 0 errors
$
```

## Add a method to the R43 class

After basking in the green glow of your passing tests, it's time to take a bigger step. The simplest method in the 43 Things API is `echo`. When you call the `echo` method, you get a response containing your API key, action (`echo`), and the controller. You can translate this response into a new test method in `ts_r43.rb`, as shown here:

```
1 def test_echo
2   connection = r43.new('1234')
3   assert_equal({"api_key"    => '1234',
4                "action"     => 'echo',
5                "controller" => 'service'},
6               connection.echo("/service/echo"))
7 end
```

When you run your test suite, you see one error:

```
$ ruby ts_r43.rb
ruby ts_r43.rb
Loaded suite ts_r43
Started
E.
Finished in 0.008784 seconds.

1) Error:
test_echo(TestR43):
NoMethodError: undefined method 'echo' for #<R43:0x4022c5e4 @key="1234">
ts_r43.rb:22:in 'test_echo'

2 tests, 3 assertions, 0 failures, 1 errors
$
```

This error means you can start writing code. You're going to be contacting a server over HTTP, so you want to use the `Net::HTTP` library. And, because the response from the server will be XML, you need to require the `REXML::Document` library, as well. You should end up with a method like the one shown in Listing 2.

### Listing 2. The echo method

```
1 def echo()
2   Net::HTTP.start('www.43things.com') do |http|
3     response = http.get("/service/#{url}")
4     if response.code == "200" then
5       xml = REXML::Document.new(response.body)
6     else
7       return response.code
8     end
9     response = {
10      "api_key" =>
11        xml.elements["parameters"].elements["api_key"].text,
12      "action" =>
13        xml.elements["parameters"].elements["action"].text,
14      "controller" =>
15        xml.elements["parameters"].elements["controller"].text
16    }
17  end
18 end
```

After you write your `echo` method, you can test it to ensure that it works as expected. If your first attempt at writing the method doesn't work, you have a very small set of code to rewrite (and, we hope, some good feedback from your tests so you know where to start looking). This time, run the test with the `-n` switch, which runs only the named test you pass in:

```
$ ruby ts_r43.rb -n test_echo
Loaded suite ts_r43
Started
.
Finished in 0.014962 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
$
```

Once again, you've got a solid new method and good test coverage, so it's time to check things back into revision control.

---

## Section 5. Refactoring

## Refactor your tests

If you're going to write a program test first, you should refactor test first, too. Typically, this means you shouldn't refactor anything that doesn't have a solid test suite around it. You don't have much functionality to refactor yet, but you can clean up your tests in some ways.

Nearly every test you write for `r43` uses the `connection` object. Instead of creating the object each time, you can use a `setup` method to do it for you. `Test::Unit` has two special methods: `setup` and `tear_down`. You run `setup` before each test you've written because that's the spot where you want to put the things your tests need to do frequently, like building a standard `r43` object. Listing 3 shows your test case after refactoring.

### Listing 3. Adding a setup method to your tests

```
1 def setup
2   @connection = R43.new('1234')
3 end

4 def test_key
5
6   assert_equal('1234',@connection.key)
7   # additional functionality
8   removed for brevity
9 end
```

In this example, you've created an instance variable called `@connection`, which is available to all the tests. If this variable needs to be set to a specific value before each test, the fact that the `setup` method creates it for each test ensures that the variable will have the correct starting value each time.

## Simple refactoring

The next method you add to the `R43` class duplicates the HTTP connection portion of the `echo` method. Use your test suite as a safety net so you can perform the *extract method* refactoring. Because this method is private, you won't write tests for it; your existing test suite provides coverage.

The extraction itself is simple. You first create a new object -- call it `_get_response` (the leading underscore is a nice convention for private methods) -- whose sole purpose is handling the HTTP requests. The object should look like the code in Listing 4. Then you modify the `echo` method to use your new private method, instead of calling `Net::HTTP` directly.

### Listing 4. The `_get_response` method

```
1 def _get_response(url)
2   Net::HTTP.start('www.43things.com') do |http|
3     response = http.get("/service/#{url}")
4     xml = REXML::Document.new(response.body)
5   else
6     return response.code
7   end
8   end
9 end
10 end

11 def echo()
12   xml = _get_response("echo?api_key=#{@key}")
13   response = {
14     "api_key" =>
15     xml.elements["parameters"].elements["api_key"].text,
16     "action" =>
17     xml.elements["parameters"].elements["action"].text,
18     "controller" =>
19     xml.elements["parameters"].elements["controller"].text
20   }
21 end
```

Check your test suite again. All your tests pass, and you know that your refactoring worked:

```
$ ruby ts_r43.rb -n test_echo
Loaded suite ts_r43
Started
.
Finished in 0.01465 seconds.

2 tests, 4 assertions, 0 failures, 0 errors
$
```

---

## Section 6. Testing tools

### Use ZenTest to go faster

Because you've been writing this code from scratch, you know that you have pretty good test coverage. What if you'd taken this project over midstream? You need a tool like ZenTest (written in Ruby) to help you.

ZenTest loads the library you're working with and writes test stubs for every method that needs testing. ZenTest can also work in reverse, loading a test suite and writing code stubs for every test. Both of these uses can save you some significant work.

The code in Listing 5 contains silly.rb, a trivial (and silly) Ruby library.

## Listing 5. silly.rb

```
1 class Foo
2   attr_accessor :foo, :bar, :baz

3   def initialize(foo = "foo",
bar = "bar", baz = "baz")
4     @foo = foo.to_s
5     @bar = bar.to_s
6     @baz = baz.to_s
7   end

8   def to_s
9     "#{@foo}:#{@bar}:#{@baz}"
10  end

11  def empty!
12    @foo = ""
13    @bar = ""
14    @baz = ""
15  end
16 end
```

Use the command `ZenTest silly.rb > ts_silly.rb` to run ZenTest on the code in Listing 5. The output looks like this:

```
1 require 'test/unit' unless defined? $ZENTEST and $ZENTEST

2 class TestFoo < Test::Unit::TestCase
3   def test_bar
4     raise NotImplementedError, 'Need to write test_bar'
5   end

6   def test_bar_equals
7     raise NotImplementedError, 'Need to write test_bar_equals'
8   end

9   def test_baz
10    raise NotImplementedError, 'Need to write test_baz'
11  end

12  def test_baz_equals
13    raise NotImplementedError, 'Need to write test_baz_equals'
14  end

15  def test_empty_bang
16    raise NotImplementedError, 'Need to write test_empty_bang'
17  end

18  def test_foo
19    raise NotImplementedError, 'Need to write test_foo'
20  end

21  def test_foo_equals
22    raise NotImplementedError, 'Need to write test_foo_equals'
23  end

24 end
```

Letting ZenTest write code stubs for you works the same way. If you were to run the tool against your existing `ts_r43.rb` like `ZenTest.rb ts_r43.rb`, it would generate

the output shown here:

```
1 class R43
2   def key
3     raise NotImplementedError, 'Need to write key'
4   end
5
6   def echo
7     raise NotImplementedError, 'Need to write echo'
8   end
9 end
```

In either case, you still have to write the code, but you know you've got good coverage of your methods when you're done. For information about getting and installing ZenTest, see [Resources](#).

## Use `unit_diff` to make test output cleaner

Sometimes `Test::Unit` reports are difficult to read. They can throw a lot of extra information at you, obscuring the important bits. The writers of ZenTest also developed a small utility called `unit_diff`, which is distributed with ZenTest. It's worth the time to download and install the package all by itself.

The `unit_diff` utility is a filter for `Test::Unit` output and redisplayes the output as though it were run through this command:

```
diff expected_output actual_output
```

This behavior makes your errors easy to identify -- and, so, easier to fix. To use `unit_diff`, use this command:

```
$ ts_bigfailure.rb | unit_diff.rb
```

Here's some output from `Test::Unit` showing the same results before and after being run through `unit_diff`:

```
$ ts_bigfailure.rb | unit_diff.rb
Listing 14. Test::Unit output
$ ruby ts_long.rb
Loaded suite ts_long
Started
F
Finished in 0.02506 seconds.

1) Failure:
test_long(TestLong) [ts_long.rb:33]:
<"If you use open source software, you're sure to
```

```

brush up against some of its rough edges. It may be a
bug or a place where the documentation isn't clear (or doesn't exist),
or maybe it's not available in your language.\n\nIf you want to
give back to the community, you can work on any of these problems,
even if you're not a programmer.\n\nYou could write a bug report
for the developers (and submit a patch if you're a programmer).\n\nYou
could write or extend existing documentation.\n\nIf you're really up
for a challenge. You could even work on translating the application
and/or documentation.\n\nHowever you choose to get involved, you
should start by contacting the developers."> expected but was
<"If you use open source software, you're sure to brush up
against some of its rough edges. It may be a bug, or a place
where the documentation isn't clear (or doesn't exist), or maybe
it's not available in your language.\n\nIf you want to give back
to the community you can work on any of these problems, even if
you're not a programmer.\n\nYou could write a bug report for the
developers (and submit a patch if you're a programmer).\n\nYou
could write or extend existing documentation.\n\nIf you're really
up for a challenge, you could even work on translating the
application and/or documentation.\n\nHowever you choose to get
involved, you should start by contacting the developers.">.

1 tests, 1 assertions, 1 failures, 0 errors
$ ruby ts_long.rb | unit_diff.rb
Loaded suite ts_long
Started
F
Finished in 0.017676 seconds.

1) Failure:
test_long(TestLong) [ts_long.rb:33]:
3c3
< If you want to give back to the community, you can work on
any of these problems, even if you're not a programmer.
---
> If you want to give back to the community you can work on
any of these problems, even if you're not a programmer.
1 tests, 1 assertions, 1 failures, 0 errors
$

```

Anytime your failures might include long blocks of output, `unit_diff` is going to be a lifesaver.

## Mock test fixtures

At times, you'll find that you don't want to test against a real data source -- for example, a database you don't want to insert test data into. The data source may require too much time to start up and shut down. It may be a system you don't always have access to or that returns hard-to-predict dynamic data. Whatever your reason, there's a way out: *mock objects*.

Mock objects allow you to replace a test fixture with a programmatic representation. For example, in your `echo` method from earlier, you have to be able to connect to the 43 Things server to get a passing test. You could mock the connection by creating a subclass of `R43`, as shown here:

```
1 require 'stringio'
```

```
2 class FakeR43 < R43
3   @@data = eval(DATA.read)

4   def _get_response(url)
5     raise "no data for #{url.inspect}" unless @@data.has_key? url
6     return REXML::Document.new(@@data[url])
7   end
8 end
```

You can use this class to replace the `R43` class when building connections in your tests. This way, instead of connecting to the actual 43 Things Web servers, you just use the XML responses you've stored in the `DATA` portion of your `ts_r43.rb` file. Your tests will be faster, more reliable, and repeatable.

---

## Section 7. Summary

In this tutorial, you started building a Ruby library test first. In addition to writing tests and code, you learned to apply these principles to refactoring to improve the code you'd written. You also learned about the `Test::Unit` library, `ZenTest`, `unit_diff`, and mock objects.

By applying these ideas in your day-to-day Ruby programming, you'll be able to write code faster. Your code will also be better tested, more robust, and easier to debug, optimize, and extend.

# Resources

## Learn

- Get more information about the [Ruby programming language](#).
- Another great site is [RubyCentral](#), maintained by Dave Thomas and Andy Hunt (the Pragmatic Programmers).
- [Ruby on Rails](#) is a Web development framework for Ruby.
- Local Ruby Brigades (like the [Seattle.rb](#)) are great places to learn more about Ruby. You can find more information at the RubyGarden wiki on [RubyUserGroups](#).
- For refactoring information, read *Refactoring: Improving the Design of Existing Code* by Martin Fowler, et al., and see the book's [supporting Web page](#).
- If you'd like a better look at r43, you can see the whole thing at [RubyForge](#).
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

## Get products and technologies

- ZenTest and unit\_diff are hosted at RubyForge. You can get them from [RubyForge](#).
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

## Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

# About the author

Pat Eyer

Pat Eyer is a co-founder of the [Seattle Ruby Brigade](#) and has been an active member of the Ruby community for nearly five years. He currently maintains the r43 library. He has written about free software, Linux, Ruby, and networking for several print- and Web-based publishers.