Extracted from:

Continuous Testing

with Ruby, Rails, and JavaScript

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Continuous Testing

with Ruby, Rails, and JavaScript

Ben Rady and Rod Coffin

Edited by Jacquelyn Carter

Continuous Testing

with Ruby, Rails, and JavaScript

Ben Rady
Rod Coffin

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

Pragmatic Bookshelf

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-70-8
Printed on acid-free paper.
Book version: P1.0—June 2011

Reading code is good, but nothing beats experimentation. However well you think you know a language or library, the cold, hard reality of tested behavior trumps all.

That's why we really like to run code and see what it does with exploratory tests, rather than speculate about what it might or might not do. We generally prefer this method to looking up behavior in documentation because documentation can be incorrect. Better still, an exploratory test can ensure that if the behavior provided by the library changes in the future, we'll have an automated way to find out about it. We don't leave those tests in our code all the time—only when there's a specific risk or problem that we're trying to mitigate—but when we need them, boy, are they helpful.

In this chapter, we're going to discuss how interacting with your code, rather than merely reading or executing it, can be a powerful technique that is made much easier with continuous testing.

4.1 Understanding Code by Changing It

When running tests continuously, the easiest way to learn what code does is often just to change it and see what happens. In contrast to running an external tool like a debugger or only reading through the code, when you test it continuously you can modify it in a way that proves (or disproves) your theory about what is going on and let the test runner react to your change. In the previous chapter, for example, we looked at how temporarily deleting code can help us determine if it is unnecessary. Let's look at how this technique can help us answer other kinds of questions we may have about our code.

Are We Testing This?

How can we check whether particular conditions in our code are tested? How can we be sure, for example, that we're testing a method like `zip_code()` with a nil address? Because RSpec adds `should()` and `should_not()` methods to Object when running specs, we can temporarily add assertions to our production code to check whether or not particular cases are being tested. So we can answer our question by making a small change to `zip_code()`:

Download ruby/twits/lib/revisions/user3.1.rb

```
def zip_code
  @address_text.should_not == nil
  Locales.current.parse_postal_code(@address_text)
end
```

If all our tests pass after adding such an assertion, we know that case is *not* being tested. After saving this change, Autotest immediately runs our tests and tell us that, in fact, we never test zip_code() with a nil value. To cover this condition, we can leave the assertion in place and then add a test that we expect to fail at that point:

Download ruby/twits/spec/revisions/user3.1_spec_fail.rb

```
it "should treat missing addresses like missing zip code" do
  user = User.new
  user.zip_code.should == nil
end
```

A quick save, and Autotest reveals that we're now covering that condition:

```
F

Failures:

  1) Twitter User should treat missing addresses like missing zip code
     Failure/Error: user.zip_code.should be nil
       expected not: == nil
                got:    nil
     # ./code/ruby/twits/lib/revisions/user3.1.rb:8:in `zip_code'
     # ./code/ruby/twits/spec/revisions/user3.1_spec_fail.rb:8:in
`block (2 levels) in <top (required)>'

Finished in 0.00105 seconds
1 example, 1 failure
```

Now we can remove the assertion in User and let the example run normally:

```
8 examples, 0 failures
```

So it turns out that case works without any additional changes. If the test had failed, we would know that the current implementation couldn't handle a nil address, and we could change it as necessary. Also note that before we added this example, the zip_code() method was completely covered. Unlike the previous example, where the code was merely being executed but not tested, in this case all the existing code is being tested. What is missing is a particular usage of that code (and what the expected behavior should be).

We refer to these kinds of assertions, added to production code rather than tests, as *inline assertions*. By temporarily adding these kinds of assertions to our code, we get an immediate, reliable verification of whether or not this condition is tested. While reading through specs is a great way to learn what your system does (and doesn't do), making them fail is even better.

How Did I Get to This Method?

Ruby's Kernel#caller() method returns a trace of the current call stack. This information can be really useful when trying to figure out how a particular method is used. Say you're staring at a piece of code (for example, the zip_code() method in our user class) and you want to know which tests it's covered by, you can simply add puts caller.first, save the change, and Autotest will report the callers like this:

```
../lib/revisions/user_spec.rb:27
../lib/revisions/user_spec.rb:34
../lib/revisions/user_spec.rb:41
../lib/revisions/user_spec.rb:48
```

These refer to the four lines in our spec where we're calling zip_code(). If this method was being tested indirectly, then we would see entries in this list that weren't specs. In that case, we might want to look three calls back using something like puts caller[0..3].

What Can I Do with This Object?

In a statically typed language, behavior and type are nearly inseparable. If you know an object's type, you know everything that it's capable of doing. In a dynamic language like Ruby, that's not really true. Methods can be added, changed, or removed at runtime,

Inline Assertions vs. Assert Keywords

Many languages support an assert keyword that lets you check for particular conditions in your code and raise an error if one of the checks fails. Typically, these kinds of assertions are enabled during testing but turned off in production.

We've never really understood the rationale behind this. Either something is broken or it isn't. Most of the time, if something is broken, we want our code to stop executing as quickly as possible so we don't corrupt data or give the user an erroneous result. The remainder of the time, we want to recover gracefully. That graceful recovery is expected behavior that we want to verify. Failing loud and fast in "test mode" while silently ignoring problems in production just seems crazy.

The intent behind inline assertions differs from these kinds of language-supported assert keywords. Inline assertions are temporary and only used to drive out missing cases in your tests. Once those cases are tested, we remove the inline assertions. Inline assertions may also check for cases that are legitimate—even expected—while assert keywords are only used to check for things that indicate an error state.

so simply knowing an object's type doesn't necessarily tell you what behaviors it has.

The problem is, we don't always know how to invoke the exact behavior we're looking for. You might know you need to split a string, but what are your options for providing a delimiter? Maybe you're not even sure what the behavior is, but you have a general idea and you just want to know what your options are. No problem, just make a simple call:

```
puts user.zip_code.methods
```

This tells us exactly what the object can do. Just like the caller information in the previous example, by temporarily adding this to our code, we can get Autotest to print the list of available methods for each object instance driven out by a particular example. This includes methods dynamically mixed in with Ruby's extend method. If we want to be more discerning about the type of methods we're looking for, we filter the results with a regular expression:

```
puts user.zip_code.methods.grep(/each/)
```

This would tell us all the method names that contain the word each, such as each(), each_byte(), and reverse_each().

Adding Diagnostic Methods

Being a dynamic language with rich metaprogramming support, Ruby allows us to change almost anything in our environment. Not only can we dynamically create new methods and classes, but we can also change any existing method or class based on the particular context in which we're using the system.

We can take advantage of this to add additional diagnostic tools that are available when running specs from Autotest. This gives us a lot of power and control over the kinds of information we can get from our code without cluttering it up with a bunch of logging statements that are only useful when running tests.

In the previous section, we took advantage of Ruby's built-in methods and the assertion methods mixed in by RSpec to either print information to the console or to intentionally fail a test to indicate what the code was doing. We chose to use these methods mostly because they were already there and we could make use of them.

However, there's no reason to limit ourselves to what's provided. With Ruby, we can add whatever methods we want to whatever classes we want. Let's take our original example:

```
puts user.zip_code.methods.grep(/each/)
```

This is handy but a little more verbose than we'd like, especially considering we're going to delete this code as soon as we find the method we're looking for. What if we could get the same effect by doing this:

```
zip_code.put_methods /each/
```

We can add this method to every object when (and only when) our tests are run just by using Ruby's metaprogramming facilities. First, let's open the spec_helper.rb file in the spec and add our put_methods() method to Ruby's Object:

Download ruby/twits/spec/spec_helper.rb

```
class Object
  def put_methods(regex=/.*/)
```

Joe asks:
What Is Metaprogramming?

Metaprogramming is writing code that modifies code. In Ruby this can be done in a number of different ways, ranging from calling the extend() or include() methods to add methods to a class or object to simply defining (or redefining) a method for a third-party class in your source code. As one might expect, metaprogramming can cause some rather counterintuitive behavior if not used carefully, but it is a very effective way to change the behavior of existing code without modifying the source itself.

```
    puts self.methods.grep(regex)
  end
end
```

What we're doing here is dynamically adding a method (put_methods()) to Ruby's base object (Object). This means put_methods() will be available on every object instance in our system whenever this code is loaded in the runtime environment. Now, if we want to know what looping methods are available on a user's zip code, we can find out like so:

```
user.zip_code.put_methods /each/
```

This results in the following output from Autotest:

```
each_cons
each_with_object
each_with_index
each_line
each
each_byte
reverse_each
each_char
each_slice
```

There are many other places we could take this. We could use this technique to look at state within our domain objects by creating custom inspectors for more complex types. This is sometimes preferable to overriding `to_s()` in our classes when the only motivation for doing so is inspection while running tests.