THE Minitest Cookbook

Testing Tactics for the Pragmatic Rubyist

Chris Kottom

Table of contents

Introduction

It's great to be a software developer today.

Give it a moment's thought. There's a whole universe of innovative businesses and awesome projects that have shown us what's possible with enough ingenuity and a little technical know-how. Hardware and hosting services just keep getting cheaper and more consumer-friendly, and deploying your latest and greatest app or pet project to millions of eager users can be done in minutes and practically for free. And of course, we've got an unbelievable set of tools at our disposal, that have been invented to help us bring our ideas into reality - programming languages, databases, application development frameworks, operating systems, and of course, testing tools. Most of them totally free, all of them totally awesome.

It wasn't always this way. Twenty years ago, there were no flame wars about whether to use Minitest or RSpec on the new project, and not just because neither of them had been invented yet. Instead, developers would be trying to decide whether or not to to invite that guy from QA to go to lunch because he's always breaking their code and kind of a downer. Back then, there were programmers and there were testers with major differences in culture and status between the two roles.

Today though, the separation between development and testing has largely disappeared - at least in the universe that most Ruby and Rails programmers occupy. In many if not most cases now, the one writing the code is also responsible for producing *automated tests* that cover the work done. And while you're probably sick of hearing it, this is a good thing for all kinds of reasons. Why?

- Tests demonstrate that your code actually works.
- The pattern of thinking needed to write tests for code is very close to that needed to design it.
- Testing and developing in parallel tends to surface more bugs early in the development process when fixing them is cheap and easy.
- Well-tested code tends to be better designed with reduced coupling and greater cohesion.
- A good test suite acts as a detailed specification.
- Writing tests during development increases programmer engagement and efficiency.
- A test suite with good coverage aids in maintenance, refactoring, and upgrades with reduced risk of breakage and regression.
- It's faster to write code with tests than without.
- Having automated tests reduces or removes the need for manual testing.

I'm not saying you need buy into every one of these statements. In fact, I've probably only seen evidence for about half of them myself, and by that I mean: "Works for me." But even if only one or two of these turns out to be true, it would make time spent writing tests very worthwhile indeed.

Rubyists tend to take the benefits of testing as an article of faith, but most don't spend quite enough time thinking about what makes tests good or effective. I'd offer that the best tests will have a few important characteristics in common:

- Clarity: The name of each test suggests what it's about at a glance.

- Purpose: The intent and meaning of the test is obvious and unambiguous from the testing logic.
- Eloquence: The test logic is expressed through fluent use of the language and the testing framework.
- Readability: Tests are written and formatted in a way that promotes rapid discovery and comprehension.
- Efficiency: All other things being equal, automated tests should use the minimum possible system resources.

That's not by any means a complete list, but let's take it as a good place to start. We'll refer to these criteria as we look at different ways of using Minitest and make choices about how we'll test. Because at the end of all this, whether you're already 100% devoted to the test-first lifestyle or just getting started, all of us are here to get better. If that sounds like a good use of your time, read on.

Why a Cookbook?

Confession time: by all accounts, I'm a pretty mediocre cook. I love good food, and like a lot of people, I've got a handful of things that I can prepare to a passable level. (Except for my chili which is, I dare say, friggin' awesome.) But aside from a few items, I'm not expecting to win awards any time soon.

Hand me a recipe though, and it's a completely different story. Suddenly, I'm chopping and slicing and stirring and frying like a champ. One minute, I'm all confused and not sure which end of the knife to hold, and the next, I'm Jamie Oliver. Julia Child. The Swedish Chef.

And that right there, my friend, is the miracle of the recipe. The instructions help, but **what a recipe gives us is the confidence we need to turn on the stove**. That's not the same as a tasty finished meal by any means, but for most people, it's enough to help get them moving toward that goal.

Each of the recipes in *The Minitest Cookbook* addresses a specific question or problem that developers commonly encounter when setting up and writing their test suites - based on Minitest, of course. For the most part, I've tried to keep the chapters short and to the point so that the book can be used as a reference as you encounter questions or problems related to testing your code.

Cooking also occurs within a context, though, that's determined by your needs as a chef and as an eater of food. So there will be some cases when you're looking for ideas for soups or side dishes, but other times you might need to incorporate specific ingredients like potatoes or bananas or that rack of lamb that's been taking up space in your freezer for the past two months. That's why most cookbooks impose an organizational structure on the recipes in their collection - one that hopefully makes some sense and respects the way that a real cook is likely to use it. So rather than a completely random list of recipes or sorting them alphabetically, it's normal to have sections organized by course, for example, or style of cuisine.

The Minitest Cookbook is designed to be a guided tour through a broad range of general testing and Minitest-specific topics. It's organized into sections that progress through problems stretching from the very basic, generally useful fundamentals through more advanced and situational techniques. So you can start from the beginning and read through to the end, or browse until you find a level that suits you.

What to Expect from Reading This Book

For the beginner with no experience in writing and running automated tests, or at least no experience using Minitest, this book provides a gentle but thorough introduction. Newbies will be guided through the setup of a basic automated testing environment as well as the writing and running of their first few tests. The book will also look at how to maintain and organize your tests as they grow with your application.

Intermediate developers will benefit from detailed information on a whole range of practical questions and problems they'll encounter when attempting to grow test suites for their applications including:

- How can I get more useful and readable information out of my test reports?
- What are mocks and stubs, and how should I use them in my code?
- How do I test a module that's included in lots of different classes?
- What techniques can I use to cut down on duplication and share code among tests?

Rails developers won't be left out either. The book includes a whole section of recipes that look at the specific issues involved in using Minitest with Rails applications - from setting up your testing stack and managing your test data to writing and running tests that exercise all the key parts of your application. When you're done, you'll be able to develop with confidence and know that you're testing your Rails apps the right way.

Unlike a lot of books on testing, this one won't dwell on the mechanics of test-driven development. TDD has become so prevalent and popular among the Ruby and Rails development community that you'd be hard pressed to find a book on testing or development that doesn't take it as a starting point for everything taught. But TDD is primarily about development and only incidentally about testing, and it often treats the tests that fall out of it as a by-product rather than as first-class citizens of your project. That tends to result in test suites that are neglected after they've served the purpose of driving out features.

While I do work in a way that maintains a very tight loop between production code, I don't practice rigorous test-first TDD, and so this book remains agnostic on the subject. In either case, it's tangential to the main objective: writing more effective tests. To the extent that you're already using TDD successfully in your own development practice, there's nothing here that will get in the way of that. And if you're not already practicing TDD

and want to find out more about it, I'd encourage you to pick up one of the following classics on the subject:

- *Test Driven Development: By Example* by Kent Beck
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman

Source Code

Selections of the source code for the recipes in this book have been provided to you in the zip archive delivered with your copy of the book. I'll let you know which recipes have source and where you can find it for those that do.

To make things more interesting, I've also set up a repository with the same source on GitHub. If you find a problem with any of the source in the book or in the archive, feel free to send a pull request. If you have a question or a comment, open an issue. I'd like this to be a place for discussing and improving the techniques taught here.

https://github.com/chriskottom/minitest_cookbook_source

How Minitest Works

Minitest fans tend to use a lot of superlatives and ecstatic language when talking about the framework in blog posts and tweets. You've probably already heard things like this:

- "So *easy*, so *fast*, so *readable*."
- "This is just so *simple* and *clean*."
- "Seriously, figuring out minitest.stub == *omg tests are so much awesome*."
- "Minitest is such a treasure - *limited in scope* but practically *infinitely extensible*."
- "minitest and minitest-rails is *awesome*. check it out if you haven't yet. *lightweight*, *flexible* testing code."

People get pretty worked up about Minitest, and with good reason. After wrestling with testing for years, it's really a pleasure to find tools that don't require so much... coercion, I suppose.

But even if Minitest is the greatest thing since peanut butter met jelly, it would be great to find some words that describe the framework more objectively and without cheerleading.

Let's start by looking at the code itself. Source isn't open to interpretation. (Well, unless you happen to be a Ruby interpreter, of course. In that case, keep on interpreting, and thanks for all you do.)

For a moment let's remove emotions from the equation and just take a look at Minitest, the project. What sorts of words that mean something apply when speaking about it?

- **Fact:** The entire framework weighs in at less than 1600 lines of code. RSpec is almost 8 times as large. With a codebase that size, the source practically becomes its own documentation.
- **Fact:** Minitest has been singled out as a very readable project because it's written in plain Ruby that developers of all experience levels can dig into and understand.
- **Fact:** The project has remained small and simple because of conscious decisions to keep it that way in spite of frequent requests for expanded features.
- **Fact:** The source code showcases Ruby's power and elegance with great uses of closures, metaprogramming, concurrent programming, and others.
- **Fact:** Since the Minitest framework also happens to be tested with Minitest, it includes some exceptional practical examples illustrating good testing technique.

The goal for this section is to use a high-level reading of the Minitest source code to give you a basic understanding of how the framework does what it does. I'm hoping you'll add a few elegant bits of Ruby to your personal snippet collection in the process.

Do you absolutely need that in order to be able to write better tests? Of course not. So if that sounds a little deeper than you'd like to dive just now, skip over it and dig right into the testing recipes in the sections that follow. If you find later that you're curious about the plumbing that makes all this possible, you can always return for a quick summary of the framework's core concepts. For now though, I'll assume that you're sticking with me.

To really get comfortable with Minitest's internals, there are four basic abstractions that you'll need to understand: plugins, reporters, runnables and the Minitest runner. Once you understand these, you'll know what the framework is doing during every step of the testing cycle, and that in turn will help you to use it more effectively and write your own extensions.

Plugins

Let's kick things off with a nice paradox: the success of Minitest is *because of* and *in spite of* its stripped down approach to testing. Discuss.

It's not a surprise that simplicity leads a lot of developers to try Minitest for the first time. That might even be the reason you decided to pick up this book. Even so, a lot of the developers using Minitest are using open source or custom extensions to enhance some aspect of it. In many cases, the goal is to make Minitest more closely match behavior or syntax that they came to rely on in RSpec or another testing framework. So yes, give us simplicity, but give it to us **our way**.

Fortunately, Minitest supports a simple plugin architecture that developers have used to release a large and growing pool of extensions intended to enhance and modify the standard behavior to fit a whole range of needs and preferences. That includes:

- Test runner behavior
- Syntax for defining tests
- The format and channel used to report test results
- Error and failure handling - e.g. firing up a debugger, console, etc.
- Adding supplemental tools for acceptance testing, mocking and stubbing, etc.
- Integrating with 3rd party software like CI systems, parallel execution libraries, etc.

Minitest comes packaged with Minitest::Pride which, in addition to adding a touch of *faaaaabulous* to your test runs, provides a nice, barebones

example of how to implement a plugin. We'll use it for just that purpose here.

Minitest plugins often come packaged as gems, and they need to implement a simple framework-defined contract in order to be loaded and initialized. For starters, every plugin must include a loader file that follows the Minitest naming convention - `minitest/foo_plugin.rb` where `foo` is the name of your plugin. Minitest dynamically requires this file at the very beginning of the test run, and so it's where most plugins load additional supporting code and patch existing Minitest classes if required.

A plugin can also include an optional initialization hook which should be implemented as a method monkeypatched directly into the Minitest module and named according to the standard naming convention (ex: `Minitest.plugin_foo_init`). This is usually defined directly in the previously described loader file. It needs to be part of the top-level namespace like this to have access to the module attribute accessors that are exposed specifically with plugins in mind. They include:

- Backtrace filter - keeps backtraces readable
- Extensions list - register of known extension names
- Parallel executor - maintains thread pool for test execution
- Reporters - test output printing and formatting

The only way to access these without resorting to nasty hack-of-doom tactics is at plugin initialization time, so extensions that need these must be implemented as plugins.

As an example, Minitest::Pride changes reporter behavior with its own initialization method by swapping out the standard output stream with a more colorful, prideful equivalent that wraps it.

```
def self.plugin_pride_init options # :nodoc:
  if PrideIO.pride? then
    klass = ENV["TERM"] =~ /^xterm|-256color$/ ? PrideLOL : PrideIO
```

```
    io    = klass.new options[:io]

    self.reporter.reporters.grep(Minitest::Reporter).each do |rep|
      rep.io = io if rep.io.tty?
    end
  end
end
```

A Minitest plugin can also include an optional hook for processing command line arguments. This hook is also implemented as a class method patched into the Minitest module with a name in the form of `Minitest.plugin_foo_options`, and it must accept a Ruby OptionParser and a Hash of parsed options as arguments as shown in the Minitest::Pride example.

```
def self.plugin_pride_options opts, _options # :nodoc:
  opts.on "-p", "--pride", "Pride. Show your testing pride!" do
    PrideIO.pride!
  end
end
```

It's worth pointing out here that most of your favorite Minitest extensions probably don't use the plugin architecture described here. Why is that? As we just said, writing your extension as a plugin imposes certain responsibilities on the developer (implementing the methods in the contract) while offering certain limited rights in return.

- Automatic loading during the Minitest runner bootstrap process without an explicit `require` by the developer
- Easy access to reporters and other Minitest module attributes
- The ability to accept and use command line arguments

Many extensions don't need any of these things. Take the minitest-rails gem as an example. It runs on top of Minitest and provides defaults, generators, and syntactic sugar for Rails application testing. But it doesn't

accept options and doesn't change the reporting format, so it isn't implemented as a plugin. Likewise, if you find yourself writing an extension that doesn't need what plugins offer, then it’s perfectly reasonable to implement it without sticking to the plugin contract. Just make sure your decision is based on a good understanding of Minitest internals and especially the Minitest runner, which we'll be looking at more closely later in this section.

Reporters

Your test suite is a tool for directing development effort to the areas of your codebase that need it most. In that respect, the results reported by your test suite become its user interface and act as an indicator that shows how well (or how poorly) your code is working.

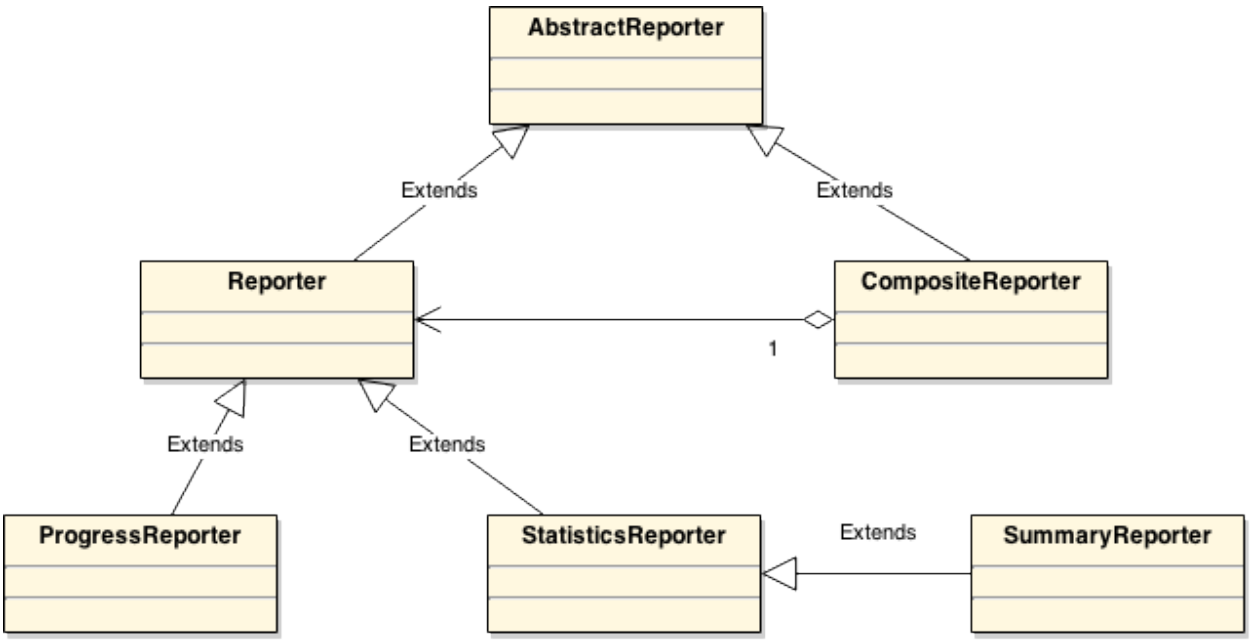
Each test that Minitest runs passes a result to a Reporter object responsible for handling it. Depending on the Reporter implementation, it might:

- Display information to the console.
- Store the result for later processing.
- Increment counters or compile statistics.
- Send the result to another system - ex: a CI, a database, etc.

The plain and simple version: a Reporter is just an object responsible for accepting and operating with test results according to a defined interface with four methods:

- `#start` - called before the first test is run
- `#record` - accepts and processes a single test result
- `#report` - delivers a detailed report after the test run
- `#passed?` - indicates passed/failed/errored/skipped tests

Minitest implements a hierarchy of Reporter classes for your formatting pleasure.



It's not important for you to commit the diagram to memory, but it is helpful to understand the relationships and dependencies between the various classes.

- **AbstractReporter** is the common parent for all other Reporter classes. It has empty implementations of all four interface methods described above and ensures thread safety for all other classes in the hierarchy.
- **Reporter** is a simple subclass of AbstractReporter that can be instantiated.
- **ProgressReporter** implements a simple `#record` method so that it spits out one character for each test result received - the familiar *dot-notation*.
- **StatisticsReporter** quietly maintains timing information for the whole test run and increments counters for assertions, errors, tests, failures, etc.
- **SummaryReporter** inherits from StatisticsReporter and layers on output to the console both before the first test and after the last one.

- **CompositeReporter** acts as a top-level proxy to a collection of other Reporters and delegates any calls it receives to the four common reporting event methods to them.

One more thing before we move on: in the previous section, we said that Minitest exposes the Reporter instance - a CompositeReporter instance, to be exact - to its plugins during initialization as an attribute on the Minitest module. After that though, it resets the attribute reference to `nil` so that test classes have no direct access to it; only the local reference to the object remains to be passed around by the Minitest runner.

What does that mean for you? Simply, it means that any changes you want to make to the Reporters that Minitest uses when actually running the tests will, nasty monkeypatches aside, need to be done by developing a plugin that implements the Minitest contract as described in **Plugins**. We'll look at how to implement that later in the book.

Runnables

Every type of test that Minitest can run is a descendant of Runnable. That includes the Test class, which inherits directly from Runnable, and the Spec and Benchmark classes, which inherit from Test. If you've ever written a test in Minitest, you're already familiar with that concept whether you know it or not. Your own tests also fall into the same hierarchy since they also extend Test, or Spec, depending one which way you lean.

```
class TpsReportTest < Minitest::Test
  def setup
    @tps_report = TpsReport.new
  end

  def test_must_have_cover_sheet
    refute_nil @tps_report.cover_sheet
  end

  def test_should_be_finished
```

```
    assert @tps_report.finished?
  end
end
```

At this point, you might be saying to yourself: "Well, sure, I've written a few tests, but that doesn't mean I know anything about what happens under the covers." Don't be so sure of that. There are a bunch of important observations that you could make solely based on having used the framework in the most basic ways.

- Test cases are subclasses of Minitest::Test. Assert-style tests usually explicitly subclass Minitest::Test, while spec-style tests usually subclass Minitest::Spec by way of a `describe` block (which is just syntactic sugar for subclassing).
- Tests are defined as methods on the test case. In assert-style tests, they're public instance methods that begin with the string `"test_"`. In spec-style test cases, Minitest provides some syntactic sugar for defining test methods via the `it` block, but behind the scenes, it's doing the same thing.
- Minitest detects your test classes and sniffs out the tests of each without the need for you to maintain a configuration file or other manifest.
- The result of each test that's run is reported in real time and also aggregated with the results of other tests in the test run.

All of these features are brought to you by the classes of the Runnable class hierarchy and some clever Ruby metaprogramming. Let's take a look at some of the major moving parts.

Runnable.inherited

Runnable maintains a registry of all of its subclasses by implementing the `inherited` callback method that it inherits from Class.

```
def self.inherited klass # :nodoc:
  self.runnables << klass
  super
end
```

Every time the interpreter loads a Runnable subclass, the Class object of that Runnable is added to this list of Runnable descendants. Clever, isn't it? Runnable also provides a class-level accessor that exposes the list of Runnable subclasses in the `Runnable.runnables` method.

Runnable.runnable_methods

In order to run itself, every Runnable subclass must be able to produce a list of its test methods. The conventions used for defining a test will vary from one implementation to another, but by way of example:

- Test and Spec assume that all public instance methods matching `/^test_/` are runnable methods.
- Benchmarks do something similar, but instead search for public instance methods matching `/^bench_/`.

You'll read the term "runnable method" from time to time though the rest of the book whenever it seems important to call attention to that aspect of it. When you do read that, you should just replace it with "test" if that helps you.

Runnable.run and Runnable.run_one_method

Given that Runnable knows its subclasses (a.k.a. all the test cases), it needs a way of running every test method for a given test class - each of which represents a single test - and then passing the result of the test to the Reporter for display, processing, collecting, whatever.

The first part is handled by the `Runnable.run` class method. It takes the collection returned by `runnable_methods`, filters it based on any optional parameters provided by the user, and passes each remaining element as an argument to the next link in the chain: `Runnable.run_one_method`.

```
##
# Responsible for running all runnable methods in a given class,
# each in its own instance. Each instance is passed to the
# reporter to record.

def self.run reporter, options = {}
  filter = options[:filter] || "/./"
  filter = Regexp.new $1 if filter =~ %r%/(.*)/%

  filtered_methods = self.runnable_methods.find_all { |m|
    filter === m || filter === "#{self}##{m}"
  }

  return if filtered_methods.empty?

  with_info_handler reporter do
    filtered_methods.each do |method_name|
      run_one_method self, method_name, reporter
    end
  end
end
```

The `Runnable.run_one_method` class method delegates to the `Minitest.run_one_method` module method and passes the result back to the Reporter instance that's been passed down the chain up until now:

```
def self.run_one_method klass, method_name, reporter
  reporter.record Minitest.run_one_method(klass, method_name)
end
```

`Minitest.run_one_method` is called with two arguments: a Class and the name of a public instance method on that class. It creates a new instance of the class, passing the name of of the instance method as a parameter, and calls `#run` on the new instance.

```
def self.run_one_method klass, method_name # :nodoc:
  result = klass.new(method_name).run
  raise "#{klass}#run _must_ return self" unless klass === result
  result
end
```

Just to be clear about what's happening here: *every test you write is run in a separate instance of your test class.*

Why do it that way? Simply put, creating a new object instance clears all of the system state that's easy to clear - local variables, instance variables, and so on. That means that your tests run within as clean a context as possible.

Runnable#run

This instance method represents the bottom of the stack, at least from the framework's perspective. It calls the test method for which this particular instance of the Runnable class is responsible. Have a look at the source code to see what it's all about:

```
TEARDOWN_METHODS = %w[ before_teardown teardown after_teardown ] # :nodoc:

##
# Runs a single test with setup/teardown hooks.

def run
  with_info_handler do
    time_it do
      capture_exceptions do
        before_setup; setup; after_setup

        self.send self.name
      end

      TEARDOWN_METHODS.each do |hook|
        capture_exceptions do
          self.send hook
        end
```

```
      end
    end
  end

  self # per contract
end
```

Minitest creates one new instance of the test class for each of its runnable methods, and the `#run` method is responsible for dynamically executing the method assigned to each instance.

In the case of the `Minitest::Test#run` implementation shown above, you can also see some of the other framework-provided features in action:

- Setup and teardown logic (`setup` and `teardown`)
- before and after hooks (`before_setup`, `before_teardown`, `after_setup`, `after_teardown`)
- Test method exception handling (`capture_exceptions` block)
- Test timing (`time_it` block)

Result Query Methods

Each instance of a Runnable also acts as the result for its assigned test which is why the `#run` method must always return `self`. If it didn't, the result couldn't be passed to the Reporter.

Each Runnable implements a collection of instance methods that allow Minitest, and specifically the Reporter, to easily understand how the test turned out.

- `#error?` - returns `true` in case the test errors out
- `#passed?` - returns `true` if the test does not fail or error out
- `#skipped?` - returns `true` if the test is skipped
- `#result_code` - returns a single-character code denoting the final result, e.g. `.`, `E`, or `F`

- `#location` - returns a string indicating the class, test, and line number where a failure occurred (based on the stack trace of the failed assertion)

The Minitest Runner

Think of the Minitest runner as a *virtual component* baked into the Minitest module. There's no Runner class definition, so don't bother looking for one, but it's still a real thing. It's the context that ties together all of the concepts that we've discussed so far and gets them interacting with one another during the test run.

If you're the sort of person who needs a visual, imagine the runner as a big onion. Each layer of the onion is a Ruby block or method that wraps other blocks and methods inside it. Each layer builds on the one that contains it and takes you a little closer to the center where your tests are finally run. In the last section we talked about some of the innermost layers which are implemented by Runnable and friends. We can now look at the outer layers handled by the runner.

minitest/autorun.rb

If you've used Minitest before, you're probably used to requiring this file in your test helper. Now you get to find out why.

The Minitest runner is activated when you require `minitest/autorun` in your code. It's responsible for loading the Ruby code needed to run Minitest and kicking off the test run with the `Minitest.autorun` method.

```
begin
  require "rubygems"
  gem "minitest"
rescue Gem::LoadError
  # do nothing
end

require "minitest"
require "minitest/spec"
```

```
require "minitest/mock"

Minitest.autorun
```

Minitest.autorun

Minitest uses an `at_exit` hook to call `Minitest.run` just before the interpreter exits. What the heck is an `at_exit` hook, you ask?

`Kernel.at_exit` is part of Ruby core. It's not the sort of thing that gets a lot of use in most application code, but it's super handy if you happen to be coding a testing framework or a daemonized server. Programmers can define blocks of code that should be run after the rest of the program has completed and before Ruby shuts down - sort of a like telling the interpreter, "One more thing..." It's particularly useful in cases where users of a library like Minitest need to load their own code (tests, plugins, extensions, etc.) at runtime before the framework swings into action while keeping the burden on the developer minimal.

```
##
# Registers Minitest to run at process exit

def self.autorun
  at_exit {
    next if $! and not ($!.kind_of? SystemExit and $!.success?)

    exit_code = nil

    at_exit {
      @@after_run.reverse_each(&:call)
      exit exit_code || false
    }

    exit_code = Minitest.run ARGV
  } unless @@installed_at_exit
  @@installed_at_exit = true
end
```

Minitest.run

Dropping down one more layer, we find Minitest setting up the environment for the test run along with all the necessary supporting objects. All the framework's major responsibilities are handled right here as it:

- Parses the command line arguments.
- Loads and initializes all detected Minitest plugins.
- Instantiates and runs the reporters.
- Ensures that parallel worker threads are shut down gracefully.
- Runs tests by passing control on to the next layer.

At the 10,000-foot level, this is a list of the framework's major responsibilities during the test run.

```
def self.run args = []
  self.load_plugins

  options = process_args args

  reporter = CompositeReporter.new
  reporter << SummaryReporter.new(options[:io], options)
  reporter << ProgressReporter.new(options[:io], options)

  self.reporter = reporter # this makes it available to plugins
  self.init_plugins options
  self.reporter = nil # runnables shouldn't depend on the reporter, ever

  self.parallel_executor.start if parallel_executor.respond_to?(:start)
  reporter.start
  begin
    __run reporter, options
  rescue Interrupt
    warn "Interrupted. Exiting..."
  end
  self.parallel_executor.shutdown
  reporter.report
```

```
  reporter.passed?
end
```

Also worth a mention here: notice how the `:reporter` attribute is initialized with a CompositeReporter and then reset to `nil` just after the plugins are initialized? As we said before in the **Plugins** and **Reporters** sections, this gives any loaded Minitest plugins their one and only opportunity to add, remove, or swap out Reporters. Keep that in mind if you're developing an extension that touches reporting.

Minitest.__run

Did you know that Minitest can run your tests over multiple threads? The runner is able to spin up a configurable number of worker threads that can be used to execute tests concurrently which can help developers locate code that's not threadsafe and in some situations speed up test suite execution. While this is a powerful feature, it's not necessarily one that can be used in every case. For the moment, it's enough that we understand two basic principles:

1. Minitest lets developers specify, either globally or individually, test cases that may be run in parallel or, alternately, those which must run serially.
2. In order to keep from mistakenly running serial tests side-by-side with parallel tests, serial tests must run and complete first.

This is the crux of what `Minitest.__run` does. It gets the collection of all test cases from our old friend `Runnable.runnables` and splits them according to this serial-parallel distinction.

```
##
# Internal run method. Responsible for telling all Runnable
# sub-classes to run.
#
# NOTE: this method is redefined in parallel_each.rb, which is
```

```
# loaded if a Runnable calls parallelize_me!.

def self.__run reporter, options
  suites = Runnable.runnables.shuffle
  parallel, serial = suites.partition { |s| s.test_order == :parallel }

  # If we run the parallel tests before the serial tests, the parallel tests
  # could run in parallel with the serial tests. This would be bad because
  # the serial tests won't lock around Reporter#record. Run the serial tests
  # first, so that after they complete, the parallel tests will lock when
  # recording results.
  serial.map { |suite| suite.run reporter, options } +
    parallel.map { |suite| suite.run reporter, options }
end
```

On to the Runnables!

From `Minitest#__run`, we're now getting to familiar territory. Since each `suite` above is one of our Runnable classes (tests and benchmarks and such), calling the `run` class method on one of them follows the same sequence of events we outlined in Runnables:

- `Runnable.run` gets the list of `runnable_methods` for its class and passes that along with itself to...
- `Runnable.run_one_method`, which hands the shared Reporter the result of the test that it receives from...
- `Minitest.run_one_method`, which creates a new instance of the Runnable class initialized with the name of the one method it will be responsible for and calls...
- `Runnable#run` on it to actually run the test method.

I said it before, Minitest is just like an onion, and like all onions, you can peel it - just not without a few tears.

Wrap-Up

The good news is that your basic introduction to Minitest is now complete. In this section, you've seen all the major moving parts of the framework

and how they come together to run your tests in a way that's pretty elegant. Hopefully, you've also seen a bit of Ruby that you didn't know about before as well.

OK, now the bad news... No, just kidding. I love that joke. There is no bad news, just more good news.

The second piece of good news is that we're a heck of a long way from being finished learning about Minitest internals. Minitest really is a testing framework for Do It Yourself-ers, and what you've got so far is *just enough* knowledge to really get started hacking on it. As we progress through some of the more advanced recipes in the later sections of the book, what you've learned here will definitely become more and more valuable.

And of course, you're always encouraged to RTFS - Read The Freakin' Source. Crack open the code, and find out for yourself. The answer to any question you might have is only ever a visit to GitHub away.

Basic Recipes

> *A journey of a thousand miles begins with a single step.*
>
> ***- Lao-tzu, The Way of Lao-tzu***

Everyone's got to start somewhere, and the recipes in this section are designed to cover some of the most fundamental tasks that you'll perform when working with Minitest. We'll tackle the basics of setting up and using your test suite as well as how to write and organize basic tests. Even experienced old-dog developers might be surprised to find a few new tricks worth adding to their coding routines in this section.

To illustrate some of the concepts in this section's recipes, we'll be using a really simple implementation of FizzBuzz, a coding exercise that is often given as an initial test to programming job applicants. The rules of FizzBuzz are simple:

1. FizzBuzz accepts a number as input.
2. If the number is a multiple of 3, it responds with "Fizz".
3. If the number is a multiple of 5, it responds with "Buzz".
4. If the number is a multiple of both 3 and 5, it responds with "FizzBuzz".

5. If the number is neither a multiple of 3 or 5, it responds with the original number.

Plenty of blog posts and mailing list threads have already been devoted to solving this simple problem. My implementation is basic and obvious, but it works.

```
class FizzBuzz
  def convert(number)
    if number % 15 == 0
      "FizzBuzz"
    elsif number % 5 == 0
      "Buzz"
    elsif number % 3 == 0
      "Fizz"
    else
      number.to_s
    end
  end
end
```

You can expect to be seeing this and variants of it through the rest of this section.

Code for all FizzBuzz examples throughout this section can be found in the `fizzbuzz/` directory of the source code archive or in the **GitHub repo**.

Writing Tests

Problem

You're ready to write your first tests for code that you have written or are just about to write, and good sense and good taste has convinced you to use Minitest for the job. Your text editor is open, and you're ready to start.

Yep, just need to start at the beginning. No problem. Right after you check your Twitter feed. And maybe have a snack.

A lot of developers work through a handful of Ruby and Rails tutorials that include testing but find themselves uncertain where to start when it comes time to write tests for their own code. You've probably worked through tutorials that included testing before, but this time, let's try a tutorial that keeps the code super-simple and instead focuses on the tests.

Solution

All the tests that we write from this point forward will follow the same basic four-phase structure.

1. **Setup** the inputs and data objects prior to running the test.
2. **Exercise** the logic under test.
3. **Verify** that the tested code produces the expected results.
4. **Teardown** or reset application state before running the next test.

It's the framework's job to ensure that your tests run according to this process as long as you use the hooks and features it provides. This recipe will show you how to do that and how to test a basic program starting from zero.

We'll be writing a first test for the simple FizzBuzz implementation that we outlined at the beginning of this section. To get started, we only need to create a FizzBuzzTest class in the `test/` directory that inherits from Minitest::Test:

```
require 'test_helper'
require 'fizz_buzz'

class FizzBuzzTest < Minitest::Test
end
```

It's not doing much of anything, but what we have here is, in fact, a full-fledged test case. We could execute our test suite right now, and the framework would run it.

At first glance, all we see is a normal class - no domain-specific language (DSL) to learn, just plain Ruby. The only discernible features are two `require` statements - one that loads the `test_helper.rb` file that we created when we set up the FizzBuzz project in **Add Minitest to Your Ruby Project** and one that that refers to the FizzBuzz class itself, which is what we'll end up testing.

The test class inherits from Minitest::Test which is part of Minitest's hierarchy of Runnables. The framework treats all Runnables as test cases when it starts a test run.

Suites, Cases, and Tests

The terminology surrounding testing has been rendered almost meaninglessness through years of misuse by well-meaning tech writers, but we're going to try to maintain some level of consistency here. Let's agree on the following definitions.

An **assertion** is a single verifiable statement about the expected state or behavior of the system under test. Minitest provides two varieties of these - assertions for assert-style testing and **expectations** for spec-style testing. Throughout the book, we'll try to make it clear, at least based on context, which type of assertion is intended.

A **test** refers to a collection of assertions that are executed as a unit and return a single result to the runner. In Minitest, a test corresponds to a single runnable method - whether it's defined using a standard method definition or a spec-style `it` block.

A **test case** is a collection of tests that all relate to a similar class, unit, subsystem, system, etc. Usually test cases are defined in a single file, but more precisely for Minitest, they're individual Runnable subclasses. You might also see terms like "test file" and "test class" used as synonyms.

A **test suite** refers to a collection of test cases that can be run as a set. In general, this term will only ever be applied to the set of test cases for a project.

In Minitest assert-style testing, every public instance method of a test class that begins with the pattern `test_` is treated as a test. So let's say that we want to add four tests to the test case that map to the known behaviors that our FizzBuzz class follows:

- Given input divisible by 15, respond with "FizzBuzz".
- Given input divisible by 5, respond with "Buzz".
- Given input divisible by 3, respond with "Fizz".
- Given any other input, respond with the original input.

We choose descriptive names for each and stub out empty methods that will be filled in shortly.

```
class FizzBuzzTest < Minitest::Test
  def test_converts_multiples_of_fifteen_to_fizzbuzz
  end

  def test_converts_multiples_of_five_to_buzz
  end
```

```
  def test_converts_multiples_of_three_to_fizz
  end

  def test_returns_same_number_for_other_numbers
  end
end
```

Rails supports an alternate block syntax for defining tests that makes the tests read a little more naturally.

```
class ArticleTest < ActiveSupport::TestCase
  test "should not save article without title" do
    article = Article.new
    assert_not article.save
  end
end
```

Rails provides this as syntactic sugar for defining tests. Under the covers though, it's doing exactly the same thing that FizzBuzzTest is doing explicitly - defining methods.

Running the test suite now with `rake`, you can see that Minitest runs these four empty tests.

```
$ rake
Run options: --seed 36226

# Running:

....

Finished in 0.001038s, 3852.3398 runs/s, 11557.0194 assertions/s.

4 runs, 12 assertions, 0 failures, 0 errors, 0 skips
```

As Minitest executes your tests, it outputs a text-based progress bar to the terminal with each character signifying the result of a completed test. Each test run by Minitest must finish in one of four possible states:

- Skipped - The test was explicitly skipped using the `skip` method (`S`).
- Error - Running the test raised an uncaught error (`E`).
- Failed - One of the test's assertions failed (`F`).
- Passed - The test didn't end in any other state (`.`).

All these tests pass simply because we haven't told Minitest to skip them and because there's nothing in them to either fail or raise an error. As tempting as it might be to declare victory and call it a day, let's push on instead and see how we can add to these.

Minitest provides a set of basic assertions out of the box which allow us to compare values computed by the system under test with the values that we expect given the inputs we provide. *Refutations* are the opposite of assertions and are used to check the opposite of an assertion. Out of the box, there are 19 standard assertions and 14 standard refutations, but in the course of regular testing, you'll probably find yourself using fewer than half of them. The most commonly used assertions and refutations are described in the table below, but there's a complete reference in **Appendix A: Minitest::Test Reference**.

Assertion	Refutation	Example
`assert`	`refute`	`assert @admin.admin?, 'not an administrator'`
`assert_empty`	`refute_empty`	`assert_empty @menu.items`
`assert_equal`	`refute_equal`	`assert_equal 'admin', @admin.username`
`assert_instance_of`	`refute_instance_of`	`assert_instance_of User, @admin`
`assert_includes`	`refute_includes`	`assert_includes @menu.items, 'Chunky Bacon'`

Assertion	Refutation	Example
`assert_match`	`refute_match`	`assert_match @menu.items.first, /Bacon/`
`assert_nil`	`refute_nil`	`assert_nil @admin.blocked_at`
`assert_raises`		`assert_raises(FormatError) { @admin.email = 'admin' }`

Every one of these, in addition to its own specific parameters, also takes an optional message that's displayed in case the assertion fails. Minitest generally does a decent job formatting informative messages. The one exception to this is `assert` which always fails with `Failed assertion, no message given.` For the sake of clarity and your own sanity, **always** specify your own failure message when using `assert` or `refute`.

Let's apply this new information to FizzBuzz and fill in the tests with some well-selected assertions. In each case, we'll check a selection of input values against the expected results when passed to a FizzBuzz object.

```
class FizzBuzzTest < Minitest::Test
  def test_converts_multiples_of_fifteen_to_fizzbuzz
    fb = FizzBuzz.new

    assert_equal 'FizzBuzz', fb.convert(15)
    assert_equal 'FizzBuzz', fb.convert(45)
    assert_equal 'FizzBuzz', fb.convert(90)
  end

  def test_converts_multiples_of_five_to_buzz
    fb = FizzBuzz.new

    assert_equal 'Buzz', fb.convert(5)
    assert_equal 'Buzz', fb.convert(20)
    assert_equal 'Buzz', fb.convert(100)
  end

  def test_converts_multiples_of_three_to_fizz
```

```
    fb = FizzBuzz.new

    assert_equal 'Fizz', fb.convert(3)
    assert_equal 'Fizz', fb.convert(18)
    assert_equal 'Fizz', fb.convert(42)
  end

  def test_returns_same_number_for_other_numbers
    fb = FizzBuzz.new

    assert_equal '1', fb.convert(1)
    assert_equal '101', fb.convert(101)
    assert_equal '2014', fb.convert(2014)
  end
end
```

`assert_equal` is the best choice for what we want to do here because it most closely aligns with the purpose of the test. Every test could, of course, be written using only `assert` statements (e.g. `assert 'FizzBuzz' == fb.convert(15)`, etc.) but that would obscure the intent.

Minitest expects you to pass parameters to `assert_equal` and most of its other standard assertions with the expected value first followed by the computed value. `assert_equal` will pass or fail exactly the same if the parameters are swapped, but in case of a failure, the message displayed by Minitest might not be as easily understandable. This same general expected-first rule applies to other assertions and refutations where two values are being compared, but go ahead and browse the **Minitest::Test Reference** at the end of the book for a complete reference to all assertion and refutation methods.

Any failing assertion, error, or `skip` statement stops a test immediately, and any further assertions or logic will not be executed. That's why many developers follow a strict *one assertion per test* policy - so that each assertion has exactly one chance to succeed or fail. It's true that your tests will be better and more maintainable when **each test verifies a single**

behavior of the system, but writing a single test for each of the assertions above seems like overkill to me since they're all testing the same general behavior. Also, what we consider to be a single behavior is going to vary in complex systems depending on the level of abstraction our tests are designed to exercise. In the end, you'll benefit from keeping the number of assertions per test to a minimum, but don't be afraid to use more than one assertion where it makes sense to do so.

The tests are working just fine now, but you'll notice that there's an awful lot of code repeated between them. If you've been around the Ruby world, or programming in general for any length of time, you're probably familiar with the principle *DRY (Don't Repeat Yourself)* - the notion that each piece of logic in a system should have a single location. *DRY* is an important principle for building maintainable systems, and Minitest::Test includes two lifecycle methods that we can use to cut down on repeated code: `setup`, which runs before each test, and `teardown`, which runs after each test. The `teardown` method is used sparingly because of the way the Minitest runner (and Rails fixtures, when in use) work, but we can use the `setup` method to DRY up some of the shared logic and move it out of our tests so that their real purpose can shine through.

```
class FizzBuzzTest < Minitest::Test
  def setup
    @fb = FizzBuzz.new
  end

  def test_converts_multiples_of_fifteen_to_fizzbuzz
    assert_equal 'FizzBuzz', @fb.convert(15)
    assert_equal 'FizzBuzz', @fb.convert(45)
    assert_equal 'FizzBuzz', @fb.convert(90)
  end

  def test_converts_multiples_of_five_to_buzz
    assert_equal 'Buzz', @fb.convert(5)
    assert_equal 'Buzz', @fb.convert(20)
    assert_equal 'Buzz', @fb.convert(100)
```

```
  end

  def test_converts_multiples_of_three_to_fizz
    assert_equal 'Fizz', @fb.convert(3)
    assert_equal 'Fizz', @fb.convert(18)
    assert_equal 'Fizz', @fb.convert(42)
  end

  def test_returns_same_number_for_other_numbers
    assert_equal '1', @fb.convert(1)
    assert_equal '101', @fb.convert(101)
    assert_equal '2014', @fb.convert(2014)
  end
end
```

That's a small improvement. If we wanted to, we could *DRY* up our test code even further by reducing the duplicated assertions in each test and instead just calling `assert_equal` once within an enumerator loop like so:

```
class FizzBuzzTest < Minitest::Test
  def setup
    @fb = FizzBuzz.new
  end

  def test_converts_multiples_of_fifteen_to_fizzbuzz
    [15, 45, 90].each do |i|
      assert_equal 'FizzBuzz', @fb.convert(i)
    end
  end

  def test_converts_multiples_of_five_to_buzz
    [5, 20, 100].each do |i|
      assert_equal 'Buzz', @fb.convert(i)
    end
  end

  def test_converts_multiples_of_three_to_fizz
    [3, 18, 42].each do |i|
      assert_equal 'Fizz', @fb.convert(i)
```

```
    end
  end

  def test_returns_same_number_for_other_numbers
    [1, 101, 2014].each do |i|
      assert_equal i.to_s, @fb.convert(i)
    end
  end
end
```

This code is certainly more *DRY*, but is it better? It's still pretty clear what each test is meant to do, and there's no change in the testing logic. I would argue that these tests are a little more *maintainable* than the previous ones are now a little less *readable* than they were, since now we need to think through the logic of the iterator block rather than just reading assertions. Both of these are valuable, but with tests especially, we can expect to read them many times more than we will change them. So it's generally a good idea to **favor readability over DRY-ness in test code** while keeping in mind that what each of those words means will vary for each team and each developer.

Finally, it's also possible to explicitly cause a test to be skipped or failed using the `skip` and `flunk` methods, respectively. It's rare that you'll find a real-world reason to use `flunk`, but `skip` can be useful in situations where you want to write a test that specs out some future work that you're not quite ready to code just yet. For example, FizzBuzzTest doesn't currently check to see what happens when we pass FizzBuzz some unexpected input. You might expect that FizzBuzz would raise an ArgumentError, but for the moment, the behavior isn't well-defined in the requirements. In this case, you could add the test as:

```
def test_raises_argument_error_for_bad_argument
  skip 'not yet implemented'
  assert_raises(ArgumentError) { @fb.convert(-1) }
  assert_raises(ArgumentError) { @fb.convert(0) }
```

```
  assert_raises(ArgumentError) { @fb.convert(1.0) }
  assert_raises(ArgumentError) { @fb.convert('foo') }
  assert_raises(ArgumentError) { @fb.convert(nil) }
end
```

The declarative syntax of `skip` is better than a code comment, and Minitest flags the skipped test in my console output.

```
$ rake
Run options: --seed 13108

# Running:

.S...

Finished in 0.001210s, 4133.1981 runs/s, 9919.6754 assertions/s.

5 runs, 12 assertions, 0 failures, 0 errors, 1 skips

You have skipped tests. Run with --verbose for details.
```

We'll use and extend what we've learned here in further recipes, but for the time being, congratulate yourself on writing your first readable, well-designed test case.

Takeaways

- Assert-style test cases are classes that inherit from Minitest::Test.
- Public instance methods of those classes whose names begin with `test_` are treated as tests by the runner.
- Minitest provides a small set of assertions out of the box, and of those, about half are used frequently.
- Override the `setup` and `teardown` methods to include code that should be executed before or, respectively, after each test is executed.

- If you have to choose between readability and conciseness in your tests, you should almost always choose readability.

Additional Resources

- **Minitest Quick Reference**