CHAPTER 6

When Things Go Wrong

Unfortunately, neither this book nor a lifetime of practice can cause you to attain Ruby programming perfection. However, a good substitute for never making a mistake is knowing how to fix your problems as they arise. The purpose of this chapter is to provide you with the necessary tools and techniques to prepare you for Ruby search-and-rescue missions.

We will start by walking through a simple but real bug-hunting session to get a basic outline of how to investigate issues in your Ruby projects. We'll then dive into some more specific tools and techniques for helping refine this process. What may surprise you is that we'll do all of this without ever talking about using a debugger. This is mainly because most Rubyists can and do get away without the use of a formal debugging tool, via various lightweight techniques that we'll discuss here.

One skill set you will need in order to make the most out of what we'll discuss here is a decent understanding of how Ruby's built-in unit testing framework works. That means if you haven't yet read Chapter 1, *Driving Code Through Tests*, you may want to go ahead and do that now.

What you will notice about this chapter is that it is much more about the process of problem solving in the context of Ruby than it is about solving any particular problem. If you keep this goal in mind while reading through the examples, you'll make the most out of what we'll discuss here.

Now that you know what to expect, let's start fixing some stuff.

A Process for Debugging Ruby Code

Part of becoming masterful at anything is learning from your mistakes. Because Ruby programming is no exception, I want to share one of my embarrassing moments so that others can benefit from it. If the problems with the code that I am about to show are immediately obvious to you, don't worry about that. Instead, focus on the problem-solving strategies used, as that's what is most important here.

We're going to look at a simplified version of a real problem I ran into in my day-to-day work. One of my Rails gigs involved building a system for processing scholarship applications online. After users have filled out an application once, whether it was accepted or rejected, they are presented with a somewhat different application form upon renewal. Although it deviates a bit from our real-world application, here's some simple code that illustrates that process:

```
if gregory.can_renew?
  puts "Start the application renewal process"
else
  puts "Edit a pending application or submit a new one"
end
```

At first, I thought the logic for this was simple. As long as all of the user's applications had a status of either accepted or rejected, it was safe to say that they could renew their application. The following code provides a rough model that implements this requirement:

```
Application = Struct.new(:state)

class User
  def initialize
    @applications = []
  end

  attr_reader :applications

  def can_renew?
    applications.all? { |e| [:accepted, :rejected].include?(e.state) }
  end
end
```

Using this model, we can see that the output of the following code is `Start the application renewal process`:

```
gregory = User.new
gregory.applications << Application.new(:accepted)
gregory.applications << Application.new(:rejected)

if gregory.can_renew?
  puts "Start the application renewal process"
else
  puts "Edit a pending application or submit a new one"
end
```

If we add a pending application into the mix, we see that the other case is triggered, outputting `Edit a pending application or submit a new one`:

```
gregory = User.new
gregory.applications << Application.new(:accepted)
gregory.applications << Application.new(:rejected)
gregory.applications << Application.new(:pending)

if gregory.can_renew?
  puts "Start the application renewal process"
```

```
else
  puts "Edit a pending application or submit a new one"
end
```

So far everything has been going fine, but the next bit of code exposes a nasty edge case:

```
gregory = User.new

if gregory.can_renew?
  puts "Start the application renewal process"
else
  puts "Edit a pending application or submit a new one"
end
```

I fully expected this to print out `Edit a pending application or submit a new one`, but it managed to print the other message instead!

Popping open *irb*, I tracked down the root of the problem:

```
>> gregory = User.new
=> #<User:0x2618bc @applications=[]>
>> gregory.can_renew?
=> true

>> gregory.applications
=> []
>> gregory.applications.all? { false }
=> true
```

Of course, the trouble here was due to an incorrect use of the `Enumerable#all?` method. I had been relying on Ruby to do what I meant rather than what I actually asked it to do, which is usually a bad idea. For some reason I thought that calling `all?` on an empty array would return `nil` or `false`, but instead, it returned `true`. To fix it, I'd need to rethink `can_renew?` a little bit.

I could have fixed the issue immediately by adding a special case involving `applications.empty?`, but I wanted to be sure this bug wouldn't have a chance to crop up again. The easiest way to do this was to write some tests, which I probably should have done in the first place.

The following simple test case clearly specified the behavior I expected, splitting it up into three cases as we did before:

```
require "test/unit"

class UserTest < Test::Unit::TestCase
  def setup
    @gregory = User.new
  end

  def test_a_new_applicant_cannot_renew
    assert_block("Expected User#can_renew? to be false for a new applicant") do
      not @gregory.can_renew?
    end
  end
  def test_a_user_with_pending_applications_cannot_renew
    @gregory.applications << app(:accepted) << app(:pending)

    msg = "Expected User#can_renew? to be false when user has pending applications"
    assert_block(msg) do
      not @gregory.can_renew?
    end
  end

  def test_a_user_with_only_accepted_and_rejected_applications_can_renew
    @gregory.applications << app(:accepted) << app(:rejected) << app(:accepted)
    msg = "Expected User#can_renew? to be true when all applications " +
          "are accepted or rejected"
    assert_block(msg) { @gregory.can_renew? }
  end

  private

  def app(name)
    Application.new(name)
  end

end
```

When we run the tests, we can clearly see the failure that we investigated manually a little earlier:

```
  1) Failure:
test_a_new_applicant_cannot_renew(UserTest) [foo.rb:24]:
Expected User#can_renew? to be false for a new applicant

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

Now that we've successfully captured the essence of the bug, we can go about fixing it. As you may suspect, the solution is simple:

```
def can_renew?
  return false if applications.empty?
  applications.all? { |e| [:accepted, :rejected].include?(e.state) }
end
```

Running the tests again, we see that everything passes:

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

If we went back and ran our original examples that print some messages to the screen, we'd see that those now work as expected as well. We could have used those on their own to test our attempted fix, but by writing automated tests, we have a safety net against regressions, which may be one of the main benefits of unit tests.

Though the particular bug we squashed may be a bit boring, what we have shown is a repeatable procedure for bug hunting, without ever firing up a debugger or combing through logfiles. To recap, here's the general plan for how things should play out:

1. First, identify the different scenarios that apply to a given feature.
2. Enumerate over these scenarios to identify which ones are affected by defects and which ones work as expected. This can be done in many ways, ranging from printing debugging messages on the command line to logfile analysis and live application testing. The important thing is to identify and isolate the cases affected by the bug.
3. Hop into *irb* if possible and take a look at what your objects actually look like under the hood. Experiment with the failing scenarios in a step-by-step fashion to try to dig down and uncover the root cause of problems.
4. Write tests to reproduce the problems you are having, along with what you expect to happen when the issue is resolved.
5. Implement a fix that passes the tests, and then repeat the process until all issues are resolved.

Sometimes, it's possible to condense this process into two steps simply by writing a test that reproduces the bug and then introducing a fix that passes the tests. However, most of the time the extra legwork will pay off, as understanding the root cause of the problem will allow you to treat your application's disease all at once rather than addressing its symptoms one by one.

Given this basic outline of how to isolate and resolve issues within our code, we can now focus on some specific tools and techniques that will help improve the process for us.

Capturing the Essence of a Defect

Before you can begin to hunt down a bug, you need to be able to reproduce it in isolation. The main idea is that if you remove all the extraneous code that is unrelated to the issue, it will be easier to see what is really going on. As you continue to investigate an issue, you may discover that you can reduce the example more and more based on what you learn. Because I have a real example handy from one of my projects, we can look at this process in action to see how it plays out.

What follows is some Prawn code that was submitted as a bug report. The problem it's supposed to show is that every text `span()` resulted in a page break happening, when it wasn't supposed to:

```
Prawn::Document.generate("span.pdf") do

  span(350, :position => :center) do
    text "Here's some centered text in a 350 point column. " * 100
  end

  text "Here's my sentence."

  bounding_box([50,300], :width => 400) do
    text "Here's some default bounding box text. " * 10
    span(bounds.width,
      :position => bounds.absolute_left - margin_box.absolute_left) do
      text "The rain in Spain falls mainly on the plains. " * 300
    end
  end

  text "Here's my second sentence."

end
```

Without a strong knowledge of Prawn, this example may already seem fairly reduced. After all, the text represents a sort of abstract problem definition rather than some code that was ripped out of an application, and that is a good start. But upon running this code, I noticed that the defect was present whenever a `span()` call was made. This allowed me to reduce the example substantially:

```
Prawn::Document.generate("span.pdf") do

  span(350) do
    text "Here's some text in a 350pt wide column. " * 20
  end

  text "This text should appear on the same page as the spanning text"

end
```

Whether or not you have any practical experience in Prawn, the issue stands out better in this revised example, simply because there is less code to consider. The code is also a bit more self-documenting, which makes buggy output harder to miss. Many bug reports can be reduced in a similar fashion. Of course, not everything compacts so well, but every little bit of simplification helps.

Most bugs aren't going to show up in the first place you look. Instead, they'll often be hidden farther down the chain, stashed away in some low-level helper method or in some other code that your feature depends on. As this is so common, I've developed the habit of mentally tracing the execution path that my example code follows, in hopes of finding some obvious mistake along the way. If I notice anything suspicious, I start the next iteration of bug reproduction.

Using this approach, I found that the problem with `span()` wasn't actually in `span()` at all. Although the details aren't important, it turns out that the core problem was in a lower-level function called `canvas()`, which `span()` relies on. This method was incorrectly setting the text cursor on the page to the very bottom of the page after executing its block argument. I used the following example to confirm this was the case:

```
Prawn::Document.generate("canvas_sets_y_to_0.pdf") do
  canvas { text "Some text at the absolute top left of the page" }

  text "This text should not be after a pagebreak"
end
```

When I saw that I was able to reproduce the problem, I went on to formally specify what was wrong in the form of tests, feeling reasonably confident that this was the root defect.

Whenever you are hunting for bugs, the practice of reducing your area of interest first will help you avoid dead ends and limit the number of possible places in which you'll need to look for problems. Before doing any formal investigation, it's a good idea to check for obvious problems so that you can get a sense of where the real source of your defect is. Some bugs are harder to catch on sight than others, but there is no need to overthink the easy ones.

If a defect can be reproduced in isolation, you can usually narrow it down to a specific deviation from what you expected to happen. We'll now take a look at how to go from an example that reproduces a bug to a failing test that fully categorizes it.

The main benefit of an automated test is that it will explode when your code fails to act as expected. It is important to keep in mind that even if you have an existing test suite, when you encounter a bug that does not cause any failures, you need to update your tests. This helps prevent regressions, allowing you to fix a bug once and forget about it.

Continuing with our example, here is a simple but sufficient test to corner the bug:

```
class CanvasTest < Test::Unit::TestCase

  def setup
    @pdf = Prawn::Document.new
  end

  def test_canvas_should_not_reset_y_to_zero
    after_text_position = nil

    @pdf.canvas do
      @pdf.text "Hello World"
      after_text_position = @pdf.y
    end

    assert_equal after_text_position, @pdf.y
  end
end
```

Here, we expect the *y* coordinate after the `canvas` block is executed to be the same as it was just after the text was rendered to the page. Running this test reproduces the problem we created an example for earlier:

```
  1) Failure:test_canvas_should_not_reset_y_to_zero(CanvasTest) [---]
<778.128> expected but was
<0.0>.
```

Here, we have converted our simplified example into something that can become a part of our automated test suite. The simpler an example is, the easier this is to do. More complicated examples may need to be broken into several chunks, but this process is straightforward more often than not.

Once we write a test that reproduces our problem, the way we fix it is to get our tests passing again. If other tests end up breaking in order to get our new test to pass, we know that something is still wrong. If for some reason our problem isn't solved when we get all the tests passing again, it means that our reduced example probably didn't cover the entirety of the problem, so we need to go back to the drawing board in those cases. Even still, not all is lost. Each test serves as a significant reduction of your problem space. Every passing assertion eliminates the possibility of that particular issue from being the root of your problem. Sooner or later, there won't be any place left for your bugs to hide.

For those who need a recap, here are the keys to producing a good reduced example:

- Remove as much extraneous code as possible from your example, and the bug will be clearer to see.
- Try to make your example self-describing, so that even someone unfamiliar with the core issue can see at a glance whether something is wrong. This helps others report regressions even if they don't fully understand the internals of your project.
- Continue to revise your examples until they reach the root cause of the problem. Don't throw away any of the higher-level examples until you verify that fixing a general problem solves the specific issue that you ran into as well.
- When you understand the root cause of your problem, code up a failing test that demonstrates how the code should work. When it passes, the bug should be gone. If it fails again, you'll know there has been a regression.

Scrutinizing Your Code

When things aren't working the way you expect them to, you obviously need to find out why. There are certain tricks that can make this task a lot easier on you, and you can use them without ever needing to fire up the debugger.

Utilizing Reflection

Many bugs come from using an object in a different way than you're supposed to, or from some internal state deviating from your expectations. To be able to detect and fix these bugs, you need to be able to get a clear picture of what is going on under the hood in the objects you're working with.

I'll assume that you already know that `Kernel#p` and `Object#inspect` exist, and how to use them for basic needs. However, when left to their default behaviors, using these tools to debug complex objects can be too painful to be practical. We can take an unadorned `Prawn::Document`'s `inspect` output for an example:

```
#<Prawn::Document:0x12cf17c @page_content=#<Prawn::Reference:0x12cecf4
@data={:Length=>0}, @gen=0, @identifier=4, @stream="0.000 0.000 0.000 r
g\n0.000 0.000 0.000 RG\nq\n", @compressed=false>, @info=
#<Prawn::Reference:0x12cf0c8  @data={:Creator=>"Prawn", :Producer=>"Prawn"},
@gen=0, @identifier=1, @compressed=false>
, @root=#<Prawn::Reference:0x12cf064 @data={:Type=>:Catalog, :Pages=>
#<Prawn::Reference:0x12cf08c @data={:Count=>1, :Kids=>[#<Prawn::Reference:0x12ceca4
@data={:Contents=>#<Prawn::Reference:0x12cecf4
@data={:Length=>0}, @gen=0, @identifier=4, @stream="0.000 0.000 0.000 rg\n0.000
0.000 0.000 RG\nq\n",

<< ABOUT 50 MORE LINES LIKE THIS >>

#<Prawn::Reference:0x12cf08c @data={:Count=>1, :Kids=>[#<Prawn::Reference:0x12ceca4
...>], :Type=>:Pages}, @gen=0, @identifier=2, @compressed=false>,
:MediaBox=>[0, 0, 612.0, 792.0]}, @gen=0, @identifier=5, @compressed=false>],
@margin_box=#<Prawn::Document::BoundingBox:0x12ced30 @width=540.0,
@y=756.0, @x=36, @parent=#<Prawn::Document:0x12cf17c ...>, @height=720.0>,
@fill_color="000000", @current_page=#<Prawn::Reference:0x12ceca4 @data={:Contents=>
#<Prawn::Reference:0x12cecf4  @data={:Length=>0}, @gen=0, @identifier=4,
@stream="0.000 0.000 0.000 rg\n0.000 0.000 0.000 RG\nq\n",  @compressed=false>,
:Type=>:Page, :Parent=>#<Prawn::Reference:0x12cf08c  @data={:Count=>1,
:Kids=>[#<Prawn::Reference:0x12ceca4 ...>], :Type=>:Pages},
@gen=0, @identifier=2, @compressed=false>,  :MediaBox=>[0, 0, 612.0, 792.0]},
@gen=0, @identifier=5, @compressed=false>, @skip_encoding=nil,
@bounding_box=#<Prawn::Document::BoundingBox:0x12ced30 @width=540.0, @y=756.0, @x=36,
@parent=#<Prawn::Document:0x12cf17c ...>, @height=720.0>, @page_size="LETTER",
@stroke_color="000000" , @text_options={}, @compress=false, @margins={:top=>36,
:left=>36, :bottom=>36, :right=>36}>
```

Although this information sure is thorough, it probably won't help us quickly identify what page layout is being used or what the dimensions of the margins are. If we aren't familiar with the internals of this object, such verbose output is borderline useless. Of course, this doesn't mean we're simply out of luck. In situations like this, we can infer a lot about an object by using Ruby's reflective capabilities:

```
>> pdf.class
=> Prawn::Document

>> pdf.instance_variables
=> [:@objects, :@info, :@pages, :@root, :@page_size, :@page_layout, :@compress,
:@skip_encoding, :@background, :@font_size, :@text_options, :@margins, :@margin_box,
:@bounding_box, :@page_content, :@current_page, :@fill_color, :@stroke_color, :@y]

>> Prawn::Document.instance_methods(inherited_methods=false).sort
=> [:bounding_box, :bounds, :bounds=, :canvas, :compression_enabled?, :cursor,
:find_font, :font, :font_families, :font_registry, :font_size, :font_size=,
:margin_box, :margin_box=, :margins, :mask, :move_down, :move_up, :pad, :pad_bottom,
:pad_top, :page_count, :page_layout, :page_size, :render, :render_file, :save_font,
:set_font, :span, :start_new_page, :text_box, :width_of, :y, :y=]

>> pdf.private_methods(inherited_methods=false)
=> [:init_bounding_box, :initialize, :build_new_page_content, :generate_margin_box]
```

Now, even if we haven't worked with this particular object before, we have a sense of what is available, and it makes queries like the ones mentioned in the previous paragraph much easier:

```
>> pdf.margins
=> {:left=>36, :right=>36, :top=>36, :bottom=>36}

>> pdf.page_layout
=> :portrait
```

If we want to look at some lower-level details, such as the contents of some instance variables, we can do so via `instance_variable_get`:

```
>> pdf.instance_variable_get(:@current_page)
=> #<Prawn::Reference:0x4e5750 @identifier=5, @gen=0, @data={:Type=>:Page,
:Parent=>#<Prawn::Reference:0x4e5b60 @identifier=2, @gen=0, @data={:Type=>:Pages,
:Count=>1, :Kids=>[#<Prawn::Reference:0x4e5750 ...>]}, @compressed=false,
@on_encode=nil>, :MediaBox=>[0, 0, 612.0, 792.0],
:Contents=>#<Prawn::Reference:0x4e57a0 @identifier=4, @gen=0, @data={:Length=>0},
@compressed=false, @on_encode=nil,
@stream="0.000 0.000 0.000 rg\n0.000 0.000 0.000 RG\nq\n">}, @compressed=false,
@on_encode=nil>
```

Using these tricks, we can easily determine whether we've accidentally got the name of a variable or method wrong. We can also see what the underlying structure of our objects are, and repeat this process to drill down and investigate potential problems.

Improving inspect Output

Of course, the whole situation here would be better if we had easier-to-read `inspect` output. There is actually a standard library called *pp* that improves the formatting of `inspect` while operating in a very similar fashion. I wrote a whole section in Appendix B about this library, including some of its advanced capabilities. You should definitely read up on what `pp` offers you when you get the chance, but here I'd like to cover some alternative approaches that can also come in handy.

As it turns out, the output of `Kernel#p` can be improved on an object-by-object basis. This may be obvious if you have used `Object#inspect` before, but it is also a severely underused feature of Ruby. This feature can be used to turn the mess we saw in the previous section into beautiful debugging output:

```
>> pdf = Prawn::Document.new
=> < Prawn::Document:0x27df8a:
      @background: nil
      @compress: false
      @fill_color: "000000"
      @font_size: 12
      @margins: {:left=>36, :right=>36, :top=>36, :bottom=>36}
      @page_layout: :portrait
      @page_size: "LETTER"
      @skip_encoding: nil
      @stroke_color: "000000"
      @text_options: {}
      @y: 756.0

      @bounding_box -> Prawn::Document::BoundingBox:0x27dd64
      @current_page -> Prawn::Reference:0x27dd1e
      @info -> Prawn::Reference:0x27df44
      @margin_box -> Prawn::Document::BoundingBox:0x27dd64
      @objects -> Array:0x27df6c
      @page_content -> Prawn::Reference:0x27dd46
      @pages -> Prawn::Reference:0x27df26
      @root -> Prawn::Reference:0x27df12 >
```

I think you'll agree that this looks substantially easier to follow than the default `inspect` output. To accomplish this, I put together a template that allows you to pass in a couple of arrays of symbols that point at instance variables:

```
module InspectTemplate

  def __inspect_template(objs, refs)
    obj_output = objs.sort.each_with_object("") do |v,out|
       out << "\n      #{v}: #{instance_variable_get(v).inspect}"
    end

    ref_output = refs.sort.each_with_object("") do |v,out|
      ref = instance_variable_get(v)
      out << "\n      #{v} -> #{__inspect_object_tag(ref)}"
    end

    "< #{__inspect_object_tag(self)}: #{obj_output}\n#{ref_output} >"
  end

  def __inspect_object_tag(obj)
    "#{obj.class}:0x#{obj.object_id.to_s(16)}"
  end

end
```

After mixing this into `Prawn::Document`, I need only to specify which variables I want to display the entire contents of, and which I want to just show as references. Then, it is as easy as calling `__inspect_template` with these values:

```
class Prawn::Document

  include InspectTemplate

  def inspect
    objs = [ :@page_size, :@page_layout, :@margins, :@font_size, :@background,
             :@stroke_color, :@fill_color, :@text_options, :@y, :@compress,
             :@skip_encoding ]
    refs = [ :@objects, :@info, :@pages, :@bounding_box, :@margin_box,
             :@page_content, :@current_page, :@root]
    __inspect_template(objs,refs)
  end
end
```

Once we provide a customized `inspect` method that returns a string, both `Kernel#p` and *irb* will pick up on it, yielding the nice results shown earlier.

Although my `InspectTemplate` can easily be reused, it carries the major caveat that you become 100% responsible for exposing your variables for debugging output. Anything not explicitly passed to `__inspect_template` will not be rendered. However, there is a middle-of-the-road solution that is far more automatic.

The *yaml* data serialization standard library has the nice side effect of producing highly readable representations of Ruby objects. Because of this, it actually provides a `Kernel#y` method that can be used as a stand-in replacement for `p`. Although this may be a bit strange, if you look at it in action, you'll see that it has some benefits:

```
>> require "yaml"
=> true

>> y Prawn::Document.new
--- &id007 !ruby/object:Prawn::Document
background:
bounding_box: &id002 !ruby/object:Prawn::Document::BoundingBox
  height: 720.0
  parent: *id007
  width: 540.0
  x: 36
  y: 756.0
compress: false
info: &id003 !ruby/object:Prawn::Reference
  compressed: false
  data:
    :Creator: Prawn
    :Producer: Prawn
  gen: 0
  identifier: 1
  on_encode:
margin_box: *id002
margins:
  :left: 36
  :right: 36
  :top: 36
  :bottom: 36
page_content: *id005
page_layout: :portrait
page_size: LETTER
pages: *id004
root: *id006
skip_encoding:
stroke_color: "000000"
text_options: {}

y: 756.0
=> nil
```

I truncated this file somewhat, but the basic structure shines through. You can see that YAML nicely shows nested object relations, and generally looks neat and tidy. Interestingly enough, YAML automatically truncates repeated object references by referring to them by ID only. This turns out to be especially good for tracking down a certain kind of Ruby bug:

```
>> a = Array.new(6)
=> [nil, nil, nil, nil, nil, nil]
>> a = Array.new(6,[])
=> [[], [], [], [], [], []]
>> a[0] << "foo"
=> ["foo"]
>> a
=> [["foo"], ["foo"], ["foo"], ["foo"], ["foo"], ["foo"]]
>> y a
---
- &id001
  - foo
- *id001
- *id001
- *id001
- *id001
- *id001
```

Here, it's easy to see that the six subarrays that make up our main array are actually just six references to the same object. And in case that wasn't the goal, we can see the difference when we have six distinct objects very clearly in YAML:

```
>> a = Array.new(6) { [] }
=> [[], [], [], [], [], []]
>> a[0] << "foo"
=> ["foo"]
>> a
=> [["foo"], [], [], [], [], []]
>> y a
---
- - foo
- []

- []

- []

- []

- []
```

Although this may not be a problem you run into every day, it's relatively easy to forget to deep-copy a structure from time to time, or to accidentally create many copies of a reference to the same object when you're trying to set default values. When that happens, a quick call to `y` will make a long series of references to the same object appear very clearly.

Of course, the YAML output will come in handy when you encounter this problem by accident or if it is part of some sort of deeply nested structure. If you already know exactly where to look and can easily get at it, using pure Ruby works fine as well:

```
>> a = Array.new(6) { [] }
=> [[], [], [], [], [], []]
>> a.map { |e| e.object_id }
=> [3423870, 3423860, 3423850, 3423840, 3423830, 3423820]
>> b = Array.new(6,[])
=> [[], [], [], [], [], []]
>> b.map { |e| e.object_id }
=> [3431570, 3431570, 3431570, 3431570, 3431570, 3431570]
```

So far, we've been focusing very heavily on how to inspect your objects. This is mostly because a great deal of Ruby bugs can be solved by simply getting a sense of what objects are being passed around and what data they really contain. But this is, of course, not the full extent of the problem; we also need to be able to work with code that has been set in motion.

Finding Needles in a Haystack

Sometimes it's impossible to pull up a defective object easily to directly inspect it. Consider, for example, a large dataset that has some occasional anomalies in it. If you're dealing with tens or hundreds of thousands of records, an error like this won't be very helpful after your script churns for a while and then goes right off the tracks:

```
>> @data.map { |e|Integer(e[:amount]) }
ArgumentError: invalid value for Integer: "157,000"
        from (irb):10:in 'Integer'
        from (irb):10
        from (irb):10:in 'inject'
        from (irb):10:in 'each'
        from (irb):10:in 'inject'
        from (irb):10
        from :0
```

This error tells you virtually nothing about what has happened, except that somewhere in your giant dataset, there is an invalidly formatted integer. Let's explore how to deal with situations like this, by creating some data and introducing a few problems into it.

When it comes to generating fake data for testing, you can't get easier than the *faker* gem. Here's a sample of creating an array of hash records containing 5,000 names, phone numbers, and payments:

```
>> data = 5000.times.map do
?>   { name: Faker::Name.name, phone_number: Faker::PhoneNumber.phone_number,
?>     payment: rand(10000).to_s }
>> end

>> data.length
=> 5000
>> data[0..2]
```

```
=> [{:name=>"Joshuah Wyman", :phone_number=>"393-258-6420", :payment=>"6347"},
    {:name=>"Kraig Jacobi", :phone_number=>"779-295-0532", :payment=>"9186"},
    {:name=>"Jevon Harris", :phone_number=>"985.169.0519", :payment=>"213"}]
```

Now, we can randomly corrupt a handful of records, to give us a basis for this example. Keep in mind that the purpose of this demonstration is to show how to respond to unanticipated problems, rather than a known issue with your data.

```
5.times { data[rand(data.length)][:payment] << ".25" }
```

Now if we ask a simple question such as which records have an amount over 1,000, we get our familiar and useless error:

```
>> data.select { |e| Integer(e[:payment]) > 1000 }
ArgumentError: invalid value for Integer: "1991.25"
```

At this point, we'd like to get some more information about where this problem is actually located in our data, and what the individual record looks like. Because we presumably have no idea how many of these records there are, we might start by rescuing a single failure and then reraising the error after printing some of this data to the screen. We'll use a `begin...rescue` construct here as well as `Enumerable#with_index`:

```
>> data.select.with_index do |e,i|
?>   begin
?>     Integer(e[:payment]) > 1000
>>   rescue ArgumentError
>>      p [e,i]
>>      raise
>>   end
>> end
[{:name=>"Mr. Clotilde Baumbach", :phone_number=>"(608)779-7942",
:payment=>"1991.25"}, 91]
ArgumentError: invalid value for Integer: "1991.25"
        from (irb):67:in 'Integer'
        from (irb):67:in 'block in irb_binding'
        from (irb):65:in 'select'
        from (irb):65:in 'with_index'
        from (irb):65
        from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
```

So now we've pinpointed where the problem is coming from, and we know what the actual record looks like. Aside from the payment being a string representation of a `Float` instead of an `Integer`, it's not immediately clear that there is anything else wrong with this record. If we drop the line that reraises the error, we can get a full report of records with this issue:

```
>> data.select.with_index do |e,i|
?>   begin
?>      Integer(e[:payment]) > 1000
>>   rescue ArgumentError
>>      p [e,i]
>>   end
>> end; nil
[{:name=>"Mr. Clotilde Baumbach", :phone_number=>"(608)779-7942",
:payment=>"1991.25"}, 91]
```

```
[{:name=>"Oceane Cormier", :phone_number=>"658.016.1612", :payment=>"7361.25"}, 766]
[{:name=>"Imogene Bergnaum", :phone_number=>"(573)402-6508",
:payment=>"1073.25"}, 1368]
[{:name=>"Jeramy Prohaska", :phone_number=>"928.266.5508 x97173",
:payment=>"6109.25"}, 2398]
[{:name=>"Betty Gerhold", :phone_number=>"250-149-3161", :payment=>"8668.25"}, 2399]
=> nil
```

As you can see, this change recovered all the rows with this issue. Based on this information, we could probably make a decision about what to do to fix the issue. But because we're just interested in the process here, the actual solution doesn't matter that much. Instead, the real point to remember is that when faced with an opaque error after iterating across a large dataset, you can go back and temporarily rework things to allow you to analyze the problematic data records.

You'll see variants on this theme later on in the chapter, but for now, let's recap what to remember when you are looking at your code under the microscope:

- Don't rely on giant, ugly `inspect` statements if you can avoid it. Instead, use introspection to narrow your search down to the specific relevant objects.
- Writing your own `#inspect` method allows customized output from `Kernel#p` and within *irb*. However, this means that you are responsible for adding new state to the debugging output as your objects evolve.
- YAML provides a nice `Kernel#y` method that provides a structured, easy-to-read representation of Ruby objects. This is also useful for spotting accidental reference duplication bugs.
- Sometimes stack traces aren't enough. You can `rescue` and then reraise an error after printing some debugging output to help you find the root cause of your problems.

So far, we've talked about solutions that work well as part of the active debugging process. However, in many cases it is also important to passively collect error feedback for dealing with later. It is possible to do this with Ruby's logging system, so let's shift gears a bit and check it out.

Working with Logger

I'm not generally a big fan of logfiles. I much prefer the immediacy of seeing problems directly reported on the screen as soon as they happen. If possible, I actually want to be thrown directly into my problematic code so I can take a look around. However, this isn't always an option, and in certain cases, having an audit trail in the form of logfiles is as good as it's going to get.

Ruby's standard library *logger* is fairly full-featured, allowing you to log many different kinds of messages and filter them based on their severity. The API is reasonably well documented, so I won't be spending a ton of time here going over a feature-by-feature summary of what this library offers. Instead, I'll show you how to replicate a bit of functionality that is especially common in Ruby's web frameworks: comprehensive error logging.

If I pull up a log from one of my Rails applications, I can easily show what I'm talking about. The following is just a small section of a logfile, in which a full request and the error it ran into have been recorded:

```
Processing ManagerController#call_in_sheet (for 127.0.0.1 at 2009-02-13 16:38:42)
  [POST] Session ID: BAh7CCIJdXNlcmkiOg5yZXR1cm5fdG8wIgpmbGFzaElDOidBY3Rpb25Db250
  %0Acm9sbGVyOjpGbGFzaDo6Rmxhc2hIYXNoewAGOgpAdXNlZHsA--2f1d03dee418f4c9751925da42
  1ae4730f9b55dd Parameters: {"period"=>"01/19/2009", "commit"=>"Select",
  "action"=>"call_in_sheet", "controller"=>"manager"}


NameError (undefined local variable or method 'lunch' for
  #<CallInAggregator:0x2589240>):
    /lib/reports.rb:368:in 'employee_record'
    /lib/reports.rb:306:in 'to_grouping'
    /lib/reports.rb:305:in 'each'
    /lib/reports.rb:305:in 'to_grouping'
    /usr/local/lib/ruby/gems/1.8/gems/ruport-1.4.0/lib/ruport/data/table.rb:169:in
        'initialize'
    /usr/local/lib/ruby/gems/1.8/gems/ruport-1.4.0/lib/ruport/data/table.rb:809:in
        'new'
    /usr/local/lib/ruby/gems/1.8/gems/ruport-1.4.0/lib/ruport/data/table.rb:809:in
        'Table'
    /lib/reports.rb:304:in 'to_grouping'
    /lib/reports.rb:170:in 'CallInAggregator'
    /lib/reports.rb:129:in 'setup'
    /usr/local/lib/ruby/gems/1.8/gems/ruport-1.4.0/lib/ruport/renderer.rb:337:in
        'render'
    /usr/local/lib/ruby/gems/1.8/gems/ruport-1.4.0/lib/ruport/renderer.rb:379:in
        'build'
    /usr/local/lib/ruby/gems/1.8/gems/ruport-1.4.0/lib/ruport/renderer.rb:335:in
        'render'
    /usr/local/lib/ruby/gems/1.8/gems/ruport-1.4.0/lib/ruport/renderer.rb:451:in
        'method_missing'
    /app/controllers/manager_controller.rb:111:in 'call_in_sheet'
    /app/controllers/application.rb:62:in 'on'
    /app/controllers/manager_controller.rb:110:in 'call_in_sheet'
    /vendor/rails/actionpack/lib/action_controller/base.rb:1104:in 'send'
    /vendor/rails/actionpack/lib/action_controller/base.rb:1104:in
        'perform_action_wit
```

Although the production application would display a rather boring "We're sorry, something went wrong" message upon triggering an error, our backend logs tell us exactly what request triggered the error and when it occurred. It also gives us information about the actual request, to aid in debugging. Though this particular bug is fairly boring, as it looks like it was just a typo that snuck through the cracks, logging each error that occurs along with its full stack trace provides essentially the same information that you'd get if you were running a script locally and ran into an error.

It's nice that some libraries and frameworks have logging built in, but sometimes we'll need to roll our own. To demonstrate this, we'll be walking through a `TCPServer` that does simple arithmetic operations in prefix notation. We'll start by taking a look at it without any logging or error-handling support:

```
require "socket"

class Server

  def initialize
    @server   = TCPServer.new('localhost',port=3333)
  end

  def *(x, y)
    "#{Float(x) * Float(y)}"
  end

  def /(x, y)
    "#{Float(x) / Float(y)}"
  end

  def handle_request(session)
    action, *args = session.gets.split(/\s/)
    if ["*", "/"].include?(action)
      session.puts(send(action, *args))
    else
      session.puts("Invalid command")
    end
  end

  def run
    while session = @server.accept
      handle_request(session)
    end
  end
end
```

We can use the following fairly generic client to interact with the server, which is similar to the one we used in Chapter 2, *Designing Beautiful APIs*:

```
require "socket"

class Client

  def initialize(ip="localhost",port=3333)
    @ip, @port = ip, port
  end

  def send_message(msg)
    socket = TCPSocket.new(@ip,@port)
    socket.puts(msg)
    response = socket.gets
    socket.close
    return response
  end
  def receive_message
    socket = TCPSocket.new(@ip,@port)
    response = socket.read
    socket.close
    return response
  end

end
```

Without any error handling, we end up with something like this on the client side:

```
client = Client.new

response = client.send_message("* 5 10")
puts response

response = client.send_message("/ 4 3")
puts response

response = client.send_message("/ 3 foo")
puts response

response = client.send_message("* 5 7.2")
puts response

## OUTPUTS ##

50.0
1.33333333333333
nil
client.rb:8:in 'initialize': Connection refused - connect(2) (Errno::ECONNREFUSED)
        from client.rb:8:in 'new'
        from client.rb:8:in 'send_message'
        from client.rb:35
```

When we send the erroneous third message, the server never responds, resulting in a `nil` response. But when we try to send a fourth message, which would ordinarily be valid, we see that our connection was refused. If we take a look server-side, we see that a single uncaught exception caused it to crash immediately:

```
server_logging_initial.rb:15:in 'Float':
invalid value for Float(): "foo" (ArgumentError)
        from server_logging_initial.rb:15:in '/'
        from server_logging_initial.rb:20:in 'send'
        from server_logging_initial.rb:20:in 'handle_request'
        from server_logging_initial.rb:25:in 'run'
        from server_logging_initial.rb:31
```

Though this does give us a sense of what happened, it doesn't give us much insight into when and why. It also seems just a bit fragile to have a whole server come crashing down on the account of a single bad request. With a little more effort, we can add logging and error handling and make things behave much better:

```
require "socket"
require "logger"

class StandardError
  def report
    %{#{self.class}: #{message}\n#{backtrace.join("\n")}}
  end
end

class Server

  def initialize(logger)
    @logger   = logger
    @server   = TCPServer.new('localhost',port=3333)
  end

  def *(x, y)
    "#{Float(x) * Float(y)}"
  end

  def /(x, y)
    "#{Float(x) / Float(y)}"
  end

  def handle_request(session)
    action, *args = session.gets.split(/\s/)
    if ["*", "/"].include?(action)
      @logger.info "executing: '#{action}' with #{args.inspect}"
      session.puts(send(action, *args))
    else
      session.puts("Invalid command")
    end
  rescue StandardError => e
    @logger.error(e.report)
    session.puts "Sorry, something went wrong."
  end

  def run
    while session = @server.accept
      handle_request(session)
    end
  end
end

begin
  logger = Logger.new("development.log")
  host   = Server.new(logger)

  host.run
rescue StandardError => e
  logger.fatal(e.report)
  puts "Something seriously bad just happened, exiting"
end
```

We'll go over the details in just a minute, but first, let's take a look at the output on the client side running the identical code from earlier:

```
client = Client.new

response = client.send_message("* 5 10")
puts response

response = client.send_message("/ 4 3")
puts response

response = client.send_message("/ 3 foo")
puts response

response = client.send_message("* 5 7.2")
puts response

## OUTPUTS ##

50.0
1.33333333333333
Sorry, something went wrong.
36.0
```

We see that the third message is caught as an error and an apology is promptly sent to the client. But the interesting bit is that the fourth example continues to run normally, indicating that the server did not crash this time around.

Of course, if we swallowed all errors and just returned "We're sorry" every time something happened without creating a proper paper trail for debugging, that'd be a terrible idea. Upon inspecting the server logs, we can see that we haven't forgotten to keep ourselves covered:

```
# Logfile created on Sat Feb 21 07:07:49 -0500 2009 by /
I, [2009-02-21T07:08:54.335294 #39662]  INFO -- : executing: '*' with ["5", "10"]
I, [2009-02-21T07:08:54.335797 #39662]  INFO -- : executing: '/' with ["4", "3"]
I, [2009-02-21T07:08:54.336163 #39662]  INFO -- : executing: '/' with ["3", "foo"]
E, [2009-02-21T07:08:54.336243 #39662] ERROR -- :
ArgumentError: invalid value for Float(): "foo"
server_logging.rb:22:in 'Float'
server_logging.rb:22:in '/'
server_logging.rb:28:in 'send'
server_logging.rb:28:in 'handle_request'
server_logging.rb:36:in 'run'
server_logging.rb:45
I, [2009-02-21T07:08:54.336573 #39662]  INFO -- : executing: '*' with ["5", "7.2"]
```

Here we see two different levels of logging going on, `INFO` and `ERROR`. The purpose of our `INFO` logs is simply to document requests as parsed by our server. This is to ensure that the messages and their parameters are being processed as we expect. Our `ERROR` logs document the actual errors we run into while processing things, and you can see in this example that the stack trace written to the logfile is nearly identical to the one that was produced when our more fragile version of the server crashed.

Although the format is a little different, like the Rails logs, this provides us with everything we need for debugging. A time and date of the issue, a record of the actual request, and a trace that shows where the error originated. Now that we've seen it in action, let's take a look at how it all comes together.

We'll start with the small extension to `StandardError`:

```
class StandardError
  def report
    %{#{self.class}: #{message}\n#{backtrace.join("\n")}}
  end
end
```

This convenience method allows us to produce error reports that look similar to the ones you'll find on the command line when an exception is raised. Although `StandardError` objects provide all the same information, they do not have a single public method that provides the same report data that Ruby does, so we need to assemble it on our own.

We can see how this error report is used in the main `handle_request` method. Notice that the server is passed a `Logger` instance, which is used as `@logger` in the following code:

```
def handle_request(session)
  action, *args = session.gets.split(/\s/)
  if ["*", "/"].include?(action)
    @logger.info "executing: '#{action}' with #{args.inspect}"
    session.puts(send(action, *args))
  else
    session.puts("Invalid command")
  end
rescue StandardError => e
  @logger.error(e.report)
  session.puts "Sorry, something went wrong."
end
```

Here, we see where the messages in our logfile actually came from. Before the server attempts to actually execute a command, it records what it has parsed out using `@logger.info`. Then, it attempts to send the message along with its parameters to the object itself, printing its return value to the client end of the socket. If this fails for any reason, the relevant error is captured into `e` through `rescue`. This will catch all descendants of `StandardError`, which includes virtually all exceptions Ruby can throw. Once it is captured, we utilize the custom `StandardError#report` extension to generate an error report string, which is then logged as an error in the logfile. The apology is sent along to the client, thus completing the cycle.

That covers what we've seen in the logfile so far, but there is an additional measure for error handling in this application. We see this in the code that actually gets everything up and running:

```
begin
  logger = Logger.new("development.log")
  host   = Server.new(logger)

  host.run
rescue StandardError => e
  logger.fatal(e.report)
  puts "Something seriously bad just happened, exiting"
end
```

Although our response-handling code is pretty well insulated from errors, we still want to track in our logfile any server crashes that may happen. Rather than using `ERROR` as our designation, we instead use `FATAL`, indicating that our server has no intention of recovering from errors that bubble up to this level. I'll leave it up to you to figure out how to crash the server once it is running, but this technique also allows us to log things such as misspelled variable and method names among other issues within the `Server` class. To illustrate this, replace the `run` method with the following code:

```
def run
  while session = @server.accept
    handle_request(sessions)
  end
end
```

You'll end up crashing the server and producing the following log message:

```
F, [2009-02-21T07:39:40.592569 #39789] FATAL -- : NameError: undefined local
variable or method 'sessions' for #<Server:0x20c970>
server_logging.rb:36:in 'run'
server_logging.rb:45
```

This can be helpful if you're deploying code remotely and have some code that runs locally but not on the remote host, among other things.

Although we have not covered `Logger` in depth by any means, we've walked through an example that can be used as a template for more general needs. Most of the time, logging makes the most sense when you don't have easy, immediate access to the running code, and can be overkill in other places. If you're considering adding logging code to your applications, there are a few things to keep in mind:

- Error logging is essential for long-running server processes, during which you may not physically be watching the application moment by moment.
- If you are working in a multiprocessing environment, be sure to use a separate logfile for each process, as otherwise there will be clashes.
- `Logger` is powerful and includes a ton of features not covered here, including built-in logfile rotation.
- See the template for `StandardError#report` if you want to include error reports in your logs that look similar to the ones that Ruby generates on the command line.
- When it comes to logging error messages, `FATAL` should represent a bug from which your code has no intention of recovering; `ERROR` is more open-ended.

Depending on the kind of work you do, you may end up using `Logger` every day or not at all. If it's the former case, be sure to check out the API documentation for many of the features not covered here.

And with that, we've reached the end of another chapter. I'll just wrap up with some closing remarks, and then we can move on to more upbeat topics.

Conclusions

Dealing with defective code is something we all need to do from time to time. If we approach these issues in a relatively disciplined way, we can methodically corner and squash pretty much any bug that can be imagined. Debugging Ruby code tends to be a fluid process, starting with a good specification of how things should actually work, and then exercising various investigative tactics until a fix can be found. We don't necessarily need a debugger to track down issues in our code, but we do need to use Ruby's introspective features as much as possible, as they have the power to reveal to us exactly what is going on under the hood.

Once you get into a comfortable workflow for resolving issues in your Ruby code, it becomes more and more straightforward. If you find yourself lost while hunting down some bug, take the time to slow down and utilize the strategies we've gone over in this chapter. Once you get the hang of them, the tighter feedback loop will kick in and make your job much easier.