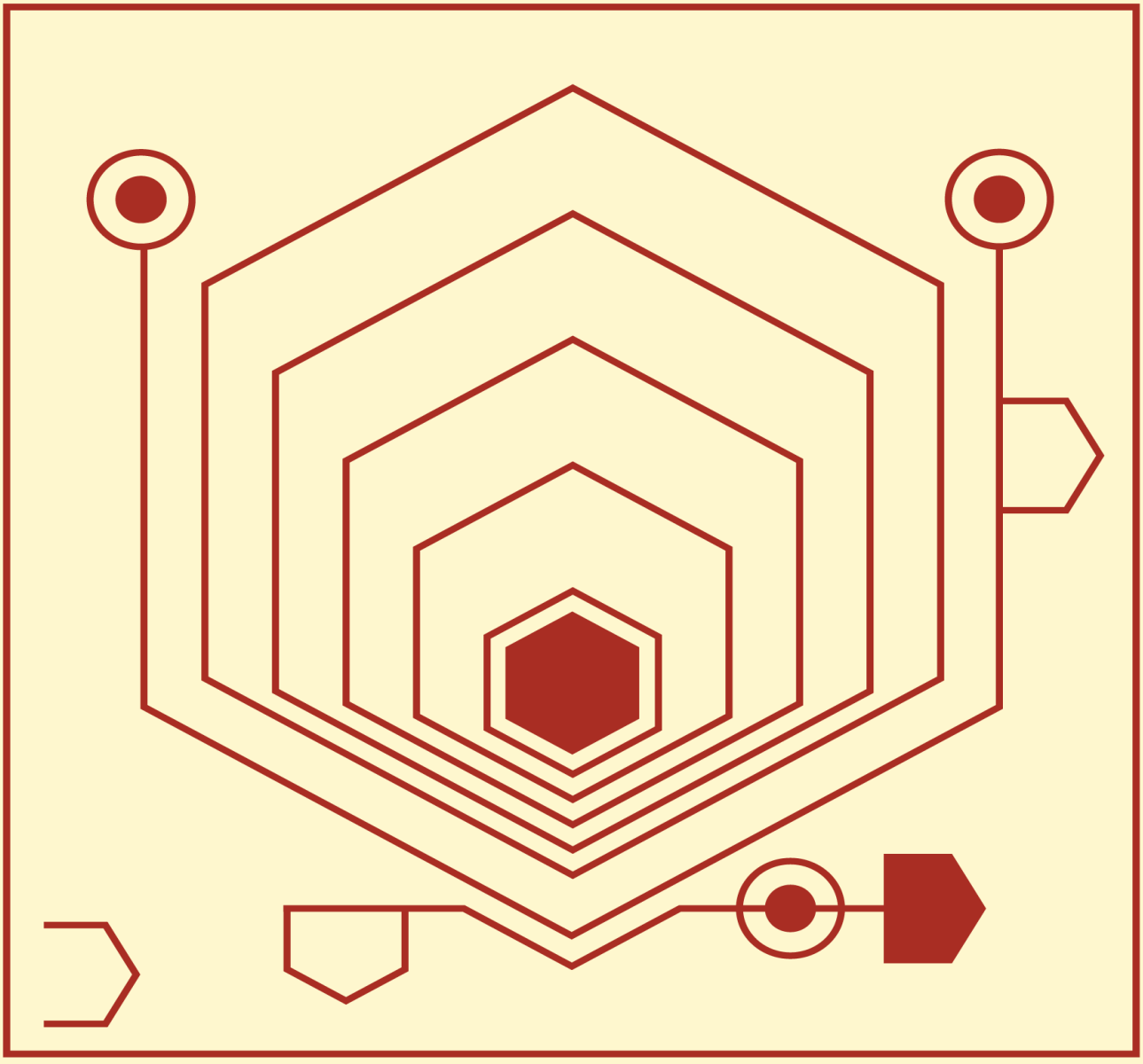

Confident Ruby

by Avdi Grimm

Table of Contents

Chapter 1

Introduction

> Ruby is designed to make programmers happy.
>
> – Yukhiro "Matz" Matsumoto

This is a book about Ruby, and about joy.

If you are anything like me, the day you first discovered the expressive power of Ruby was a very happy day. For me, I think it was probably a code example like this which made it "click":

```ruby
3.times do
  puts "Hello, Ruby world!"
end
```

To this day, that's still the most succinct, straightforward way of saying "do this three times" I've seen in any programming language. Not to mention that after

using several supposedly object-oriented languages, this was the first one I'd seen where everything, including the number "3", really was an object. Bliss!

Ruby meets the real world

Programming in Ruby was something very close to a realization of the old dream of programming in pseudocode. Programming in short, clear, intent-revealing stanzas. No tedious boilerplate; no cluttered thickets of syntax. The logic I saw in my head, transferred into program logic with a minimum of interference.

But my programs grew, and as they grew, they started to change. The real world poked its ugly head in, in the form of failure modes and edge cases. Little by little, my code began to lose its beauty. Sections became overgrown with complicated nested `if/then/else` logic and && conditionals. Objects stopped feeling like entities accepting messages, and started to feel more like big bags of attributes. `begin/rescue/end` blocks started sprouting up willy-nilly, complicating once obvious logic with necessary failure-handling. My tests, too, became more and more convoluted.

I wasn't as happy as I once had been.

Confident code

If you've written applications of substantial size in Ruby, you've probably experienced this progression from idealistic beginnings to somewhat less satisfying daily reality. You've noticed a steady decline in how much fun a project is the larger and older it becomes. You may have even come to accept it as the inevitable trajectory of any software project.

In the following pages I introduce an approach to writing Ruby code which, when practiced dilligently, can help reverse this downward spiral. It is not a brand new set

of practices. Rather, it is a collection of time-tested techniques and patterns, tied together by a common theme: *self confidence*.

This book's focus is on where the rubber meets the road in object-oriented programming: the individual method. I'll seek to equip you with tools to write methods that tell compelling stories, without getting lost down special-case rabbit holes or hung up on tedious type-checking. Our quest to write better methods will sometimes lead us to make improvements to our overall object design. But we'll continually return to the principal task at hand: writing clear, uncluttered methods.

So what, exactly do I mean when I say that our goal is to write methods which *tell a story*? Well, let me start with an example of a story that *isn't* told very well.

A good story, poorly told

Have you ever read one of those "choose your own adventure" books? Every page would end with a question like this:

> If you fight the angry troll with your bare hands, turn to page 137.
>
> If you try to reason with the troll, turn to page 29.
>
> If you don your invisibility cloak, turn to page 6.

You'd pick one option, turn to the indicated page, and the story would continue.

Did you ever try to read one of those books from front to back? It's a surreal experience. The story jumps forward and back in time. Characters appear out of nowhere. One page you're crushed by the fist of an angry troll, and on the next you're just entering the troll's realm for the first time.

What if *each individual page* was this kind of mish-mash? What if every page read like this:

> You exit the passageway into a large cavern. Unless you came from page 59, in which case you fall down the sinkhole into a large cavern. A huge troll, or possibly a badger (if you already visited Queen Pelican), blocks your path. Unless you threw a button down the wishing well on page 8, in which case there nothing blocking your way. The [troll or badger or nothing at all] does not look happy to see you.
>
> If you came here from chapter 7 (the Pool of Time), go back to the top of the page and read it again, only imagine you are watching the events happen to someone else.
>
> If you already received the invisibility cloak from the aged lighthouse-keeper, and you want to use it now, go to page 67. Otherwise, forget you read anything about an invisibility cloak.
>
> If you are facing a troll (see above), and you choose to run away, turn to page 84.
>
> If you are facing a badger (see above), and you choose to run away, turn to page 93...

Not the most compelling narrative, is it? The story asks you to carry so much mental baggage for it that just getting through a page is exhausting.

Code as narrative

What does this have to do with software? Well, code can tell a story as well. It might not be a tale of high adventure and intrigue. But it's a story nonetheless; one about

a problem that needed to be solved, and the path the developer(s) chose to accomplish that task.

A single method is like a page in that story. And unfortunately, a lot of methods are just as convoluted, equivocal, and confusing as that made-up page above.

In this book, we'll take a look at many examples of the kind of code that unnecessarily obscures the storyline of a method. We'll also explore a number of techniques for minimizing distractions and writing methods that straightforwardly convey their intent.

The four parts of a method

I believe that if we take a look at any given line of code in a method, we can nearly always categorize it as serving one of the following roles:

1. Collecting input
2. Performing work
3. Delivering output
4. Handling failures

(There are two other categories that sometimes appear: "diagnostics", and "cleanup". But these are less common.)

Let's test this assertion. Here's a method taken from the MetricFu project.

```
  def location(item, value)
    sub_table = get_sub_table(item, value)
    if(sub_table.length==0)
      raise MetricFu::AnalysisError, "The #{item.to_s}
'#{value.to_s}' "\
      "does not have any rows in the analysis table"
    else
      first_row = sub_table[0]
      case item
      when :class
        MetricFu::Location.get(first_row.file_path,
first_row.class_name, nil)
      when :method
        MetricFu::Location.get(first_row.file_path,
first_row.class_name, first_row.method_name)
      when :file
        MetricFu::Location.get(first_row.file_path, nil, nil)
      else
        raise ArgumentError, "Item must be :class, :method, or :file"
      end
    end
  end
```

Don't worry too much right now about what this method is supposed to do. Instead, let's see if we can break the method down according to our four categories.

First, it gathers some input:

```
  sub_table = get_sub_table(item, value)
```

Immediately, there is a digression to deal with an error case, when `sub_table` has no data.

```
  if(sub_table.length==0)
    raise MetricFu::AnalysisError, "The #{item.to_s} '#{value.to_s}'
  "\
    "does not have any rows in the analysis table"
```

Then it returns briefly to input gathering:

```
  else
    first_row = sub_table[0]
```

Before launching into the "meat" of the method.

```
  when :class
    MetricFu::Location.get(first_row.file_path, first_row.class_name,
  nil)
  when :method
    MetricFu::Location.get(first_row.file_path, first_row.class_name,
  first_row.method_name)
  when :file
    MetricFu::Location.get(first_row.file_path, nil, nil)
```

The method ends with code dedicated to another failure mode:

```
  else
      raise ArgumentError, "Item must be :class, :method, or :file"
    end
  end
```

Let's represent this breakdown visually, using different colors to represent the different parts of a method.

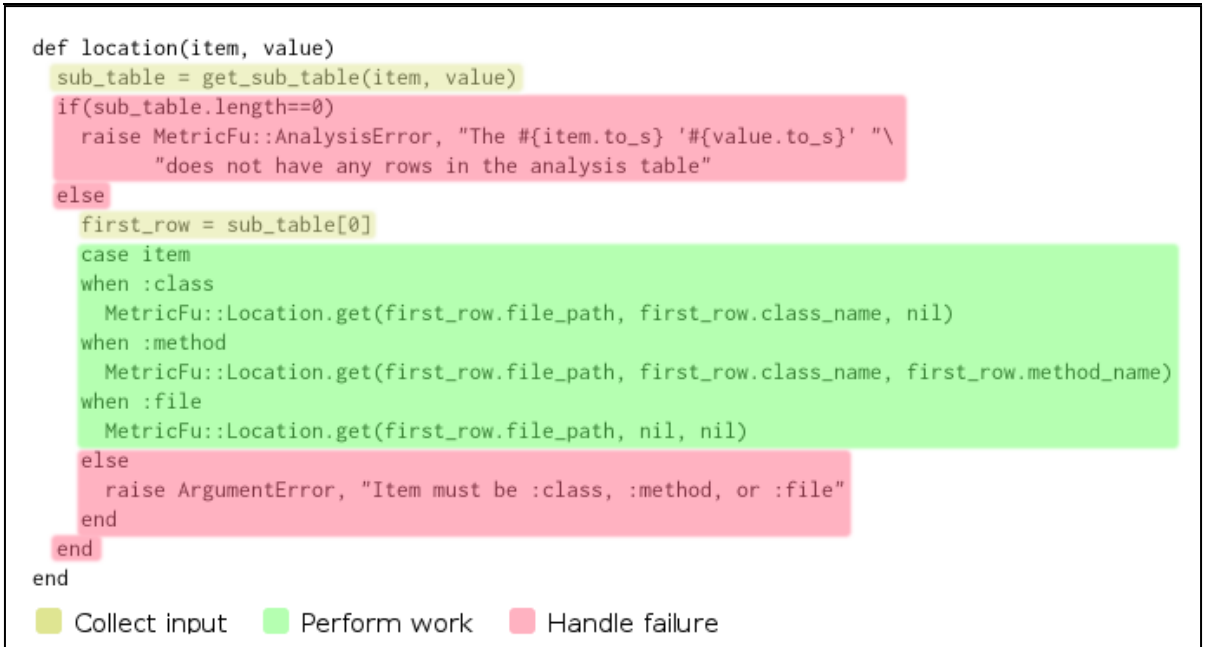

Figure 1: The #location method, annotated

This method has no lines dedicated to delivering output, so we haven't included that in the markup. Also, note that we mark up the top-level `else...end` delimiters as "handling failure". This is because they wouldn't exist without the preceding `if` block, which detects and deals with a failure case.

The point I want to draw your attention to in breaking down the method in this way is that the different parts of the method are mixed up. Some input is collected; then some error handling; then some more input collection; then work is done; and so on.

This is a defining characteristic of "un-confident", or as I think of it, "timid code": the haphazard mixing of the parts of a method. Just like the confused adventure story earlier, code like this puts an extra cognitive burden on the reader as they

unconsciously try to keep up with it. And because its responsibilities are disorganized, this kind of code is often difficult to refactor and rearrange without breaking it.

In my experience (and opinion), methods that tell a good story lump these four parts of a method together in distinct "stripes", rather than mixing them will-nilly. But not only that, they do it in the order I listed above: First, collect input. Then perform work. Then deliver output. Finally, handle failure, if necessary.

(By the way, we'll revisit this method again in the last section of the book, and refactor it to tell a better story.)

How this book is structured

This book is a patterns catalog at heart. The patterns here deal with what Steve McConnell calls "code construction" in his book Code Complete. They are "implementation patterns", to use Kent Beck's terminology. That means that unlike the patterns in books like Design Patterns or Patterns of Enterprise Application Architecture, most of these patterns are "little". They are not architectural. They deal primarily with how to structure individual methods. Some of them may seem more like idioms or stylistic guidelines than heavyweight patterns.

The material I present here is intended to help you write straightforward methods that follow this four-part narrative flow. I've broken it down into six parts:

- First, a discussion of writing methods in terms of *messages* and *roles*.
- Next, a chapter on "Performing Work". While it may seem out of order based on the "parts of a method" I laid out above, this chapter will equip you with a way of thinking through the design of your methods which will set the stage for the patterns to come.

After that comes the real "meat" of the book, the patterns. Each pattern is written in five parts:

1. A concise statement of the **indications** for a pattern. Like the indications label on a bottle of medicine, this section is a quick hint about the situation the pattern applies to. Sometimes the indications may be a particular problem or code smell the pattern can be used to address. In other cases the indications may simply be a statement of the style of code the pattern can help you achieve.

2. A **synopsis** that briefly sums up the pattern. This part is intended to be most helpful when you are trying to remember a particular pattern, but can't recall its name.

3. A **rationale** explaining why you might want to use the pattern.

4. A **worked example** which uses a concrete scenario (or two) to explain the motivation for the pattern and demonstrates how to implement it.

5. A **conclusion**, summing up the example and reflecting on the value (and sometimes, the potential pitfalls and drawbacks) of the pattern.

The patterns are divided into three sections, based on the part of a method they apply to:

- A section on patterns for *collecting input*.

- A set of patterns for delivering results such that code calling your methods can also be confident.

- Finally, a few techniques for dealing with failures without obfuscating your methods.

- After the patterns, there is another chapter, "Refactoring for Confidence", which contains some longer examples of applying the patterns from this book to some Open-Source Ruby projects.

```
3.times { rejoice! }
```

I could tell you that writing code in a confident style will reduce the number of bugs in the code. I could tell you that it will make the code easier to understand and maintain. I could tell that it will make the code more flexible in the face of changing requirements.

I could tell you all of those things, and they would be true. But that's not why I think you should read on. My biggest goal in this book is to help bring back the *joy* that you felt when you first learned to write Ruby. I want to help you write code that makes you *grin*. I want you to come away with habits which will enable you to write large-scale, real-world-ready code with the same satisfying crispness as the first few Ruby examples you learned.

Sound good? Let's get started.

Chapter 2

Collecting Input

2.1 Conditionally call conversion methods

Indications

You want to provide support for transforming inputs using conversion protocols, without forcing all inputs to understand those protocols. For instance, you are writing a method which deals with filenames, and you want to provide for the possibility of non-filename inputs which can be implicitly converted to filenames.

Synopsis

Make the call to a conversion method conditional on whether the object can respond to that method.

Rationale

By optionally supporting conversion protocols, we can broaden the range of inputs our methods can accept.

Example: opening files

Earlier, we said that `File.open` calls `#to_path` on it's `filename` argument. But `String` does not respond to `#to_path`, yet it is still a valid argument to `File.open`.

```
"/home/avdi/.gitconfig".respond_to?(:to_path) # => false
File.open("/home/avdi/.gitconfig")
# => #<File:/home/avdi/.gitconfig>
```

Why does this work? Let's take a look at the code. Here's the relevant section from MRI's `file.c`:

```
CONST_ID(to_path, "to_path");
tmp = rb_check_funcall(obj, to_path, 0, 0);
if (tmp == Qundef) {
  tmp = obj;
 }
StringValue(tmp);
```

This code says, in effect: "see if the passed object responds to `#to_path`. If it does, use the result of calling that method. Otherwise, use the original object. Either way, ensure the resulting value is a `String` by calling `#to_str`"

Here's what it would look like in Ruby:

```
if filename.respond_to?(:to_path)
  filename = filename.to_path
end

unless filename.is_a?(String)
  filename = filename.to_str
end
```

Conditionally calling conversion methods is a great way to provide flexibility to client code: code can *either* supply the expected type, *or* pass something which responds to a documented conversion method. In the first case the target object isn't forced understand the conversion method, so you can safely use conversion methods like `#to_path` which are context-specific. Of course, for client code to take advantage of this flexibility the optional conversion should be noted in the class and/or method documentation.

This is especially useful when we are defining our own conversion methods, which we'll talk more about in the next section.

Violating duck typing, just this once

But wait... didn't we say that calling `#respond_to?` violates duck-typing principles? How is this exception justified?

To examine this objection, let's define our own `File.open` wrapper, which does some extra cleanup before opening the file.

```
def my_open(filename)
  filename.strip!
  filename.gsub!(/^~/,ENV['HOME'])
  File.open(filename)
end

my_open("  ~/.gitconfig   ")     # => #<File:/home/avdi/.gitconfig>
```

Let's assume for the sake of example that we want to make very sure we have the right sort of input *before* the logic of the method is executed. Let's look at all of our options for making sure this method gets the inputs it needs.

We could explicitly check the type:

```
def my_open(filename)
  raise TypeError unless filename.is_a?(String)
  # ...
end
```

I probably don't have to explain why this is silly. It puts an arbitrary and needlessly strict constraint on the class of the input.

We could check that the object responds to every message we will send to it:

```ruby
def my_open(filename)
  unless %w[strip! gsub!].all?{|m| filename.respond_to?(m)}
    raise TypeError, "Protocol not supported"
  end
  # ...
end
```

No arbitrary class constraint this time, but in some ways this version is even worse. The protocol check at the beginning is terribly brittle, needing to be kept in sync with the list of methods that will actually be called. And `Pathname` arguments are still not supported.

We could call `#to_str` on the input.

```ruby
def my_open(filename)
  filename = filename.to_str
  # ...
end
```

But this still doesn't work for `Pathname` objects, which define `#to_path` but not `#to_str`

We could call `#to_s` on the input:

```ruby
def my_open(filename)
  filename = filename.to_s
  # ...
end
```

This would permit both `String` and `Pathname` objects to be passed. But it would also allow invalid inputs, such as `nil`, which are passed in error.

We could call `to_path` on every input, and alter `String` so it responds to `#to_path`:

```
class String
  def to_path
    self
  end
end

def my_open(filename)
  filename = filename.to_path
  # ...
end
```

This kind of thing could quickly get out of hand, cluttering up core classes with dozens of monkey-patched conversion methods for specific contexts. Yuck.

Finally, we could conditionally call `#to_path`, followed by a `#to_str` conversion, just as `File.open` does:

```
def my_open(filename)
  filename = filename.to_path if filename.respond_to?(:to_path)
  filename = filename.to_str
  # ...
end
```

Of all the strategies for checking and converting inputs we've looked at, this one is the most flexible. We can pass `String`, `Pathname`, or any other object which defines a conversion to a path-type `String` with the `#to_path` method. But other objects passed in by mistake, such as `nil` or a `Hash`, will be rejected early.

This use of `#respond_to?` is different from most type-checking in a subtle but important way. It doesn't ask "are you the kind of object I need?". Instead, it says "can you give me the kind of object I need?" As such, it strikes a useful balance. Inputs are checked, but in a way that is *open for extension*.

Conclusion

Sometimes we want to have our cake and eat it too: a method that can take input either as a core type (such as a `String`), or as a user-defined class which is convertible to that type. Conditionally calling a conversion method (such as `#to_path`) only if it exists is a way to provide this level of flexibility to our callers, while still retaining confidence that the input object we finally end up with is of the expected type.

2.2 Represent special cases as objects

> If it's possible to for a variable to be null, you have to remember to surround it with null test code so you'll do the right thing if a null is present. Often the right thing is the same in many contexts, so you end up writing similar code in lots of places—committing the sin of code duplication.
>
> — Martin Fowler, Patterns of Enterprise Application Architecture

Indications

There is a special case which must be taken into account in many different parts of the program. For example, a web application may need to behave differently if the current user is not logged in.

Synopsis

Represent the special case as a unique type of object. Rely on polymorphism to handle the special case correctly wherever it is found.

Rationale

Using polymorphic method dispatch to handle special cases eliminate dozens of repetitive conditional clauses.

Example: A guest user

In many multi-user systems, particularly web applications, it's common to have functionality which is available only to logged-in users, as well as a public-facing subset of functions which are available to anyone. In the context of a given human/computer interaction, the logged-in status of the current user is often represented

as an optional variable in a "session" object. For instance, here's a typical implementation of a `#current_user` method in a Ruby on Rails application:

```
def current_user
  if session[:user_id]
    User.find(session[:user_id])
  end
end
```

This code searches the current session (typically stored in the user's browser cookies) for a `:user_id` key. If found, the value of the key is used to find the current `User` object in the database. Otherwise, the method returns `nil` (the implicit default return value of an `if` when the test fails and there is no `else`).

A typical use of the `#current_user` method would have the program testing the result of `#current_user`, and using it if non-nil. Otherwise, the program inserts a placeholder value:

```
def greeting
  "Hello, " +
    current_user ? current_user.name : "Anonymous" +
    ", how are you today?"
end
```

In other cases, the program may need to switch between two different paths depending on the logged-in status of the user.

```
if current_user
  render_logout_button
else
  render_login_button
end
```

In still other cases, the program may need to ask the user if it has certain privileges:

```
if current_user && current_user.has_role?(:admin)
  render_admin_panel
end
```

Some of the code may use the `#current_user`, if one exists, to get at associations of the `User` and use them to customize the information displayed.

```
if current_user
  @listings = current_user.visible_listings
else
  @listings = Listing.publicly_visible
end
# ...
```

The application code may modify attributes of the current user:

```
if current_user
  current_user.last_seen_online = Time.now
end
```

Finally, some program code may update associations of the current user.

```
cart = if current_user
   current_user.cart
       else
   SessionCart.new(session)
       end
cart.add_item(some_item, 1)
```

All of these examples share one thing in common: uncertainty about whether `#current_user` will return a `User` object, or `nil`. As a result, the test for `nil` is repeated over and over again.

Representing current user as a special case object

Instead of representing an anonymous session as a `nil` value, let's write a class to represent that case. We'll call it `GuestUser`.

```
class GuestUser
  def initialize(session)
    @session = session
  end
end
```

We rewrite `#current_user` to return an instance of this class when there is no `:user_id` recorded.

```
def current_user
  if session[:user_id]
    User.find(session[:user_id])
  else
    GuestUser.new(session)
  end
end
```

For the code that used the `#name` attribute of `User`, we add a matching `#name` attribute to `GuestUser`.

```
class GuestUser
  # ...
  def name
    "Anonymous"
  end
end
```

This simplifies the greeting code nicely.

```
def greeting
  "Hello, #{current_user.name}, how are you today?"
end
```

For the case that chose between rendering "Log in" or "Log out" buttons, we can't get rid of the conditional. Instead, we add `#authenticated?` predicate methods to both `User` and `GuestUser`.

```
class User
  def authenticated?
    true
  end
  # ...
end

class GuestUser
  # ...
  def authenticated?
    false
  end
end
```

Using the predicate makes the conditional state its intent more clearly:

```
if current_user.authenticated?
  render_logout_button
else
  render_login_button
end
```

We turn our attention next to the case where we check if the user has admin privileges. We add an implementation of `#has_role?` to `GuestUser`. Since an anonymous user has no special privileges, we make it return `false` for any role given.

```ruby
class GuestUser
  # ...
  def has_role?(role)
    false
  end
end
```

This simplifies the role-checking code.

```ruby
if current_user.has_role?(:admin)
  render_admin_panel
end
```

Next up, the example of code that customizes a `@listings` result set based on whether the user is logged in. We implement a `#visible_listings` method on `GuestUser` which simply returns the publicly-visible result set.

```ruby
class GuestUser
  # ...
  def visible_listings
    Listing.publicly_visible
  end
end
```

This reduces the previous code to a one-liner.

```ruby
@listings = current_user.visible_listings
```

In order to allow the application code to treat `GuestUser` like any other user, we implement attribute setter methods as no-ops.

```
class GuestUser
  # ...
  def last_seen_online=(time)
    # NOOP
  end
end
```

This eliminates another conditional.

```
current_user.last_seen_online = Time.now
```

One special case object may link to other special case objects. In order to implement a shopping cart for users who haven't yet logged in, we make the `GuestUser`'s `cart` attribute return an instance of the `SessionCart` type that we referenced earlier.

```
class GuestUser
  # ...
  def cart
    SessionCart.new(@session)
  end
end
```

With this change, the code for adding an item to the cart also becomes a one-liner.

```
current_user.cart.add_item(some_item, 1)
```

Here's the final `GuestUser` class:

```
class GuestUser
  def initialize(session)
    @session = session
  end

  def name
    "Anonymous"
  end

  def authenticated?
    false
  end

  def has_role?(role)
    false
  end

  def visible_listings
    Listing.publicly_visible
  end

  def last_seen_online=(time)
    # NOOP
  end

 def cart
    SessionCart.new(@session)
  end
end
```

Making the change incrementally

In this example we constructed a Special Case object which fully represents the case of "no logged-in user". This object functions as a working stand-in for a real `User`

object anywhere that code might reasonably have to deal with both logged-in and not-logged-in cases. It even supplies related special case associated objects (like the `SessionCart`) when asked.

Now, this part of the book is about handling input to methods. But we've just stepped through highlights of a major redesign, one with an impact on the implementation of many different methods. Isn't this a bit out of scope?

Method construction and object design are not two independent disciplines. They are more like a dance, where each partner's movements influence the other's. The system's object design is reflected down into methods, and method construction in turn can be reflected up to the larger design.

In this case, we identified a common role in the inputs passed to numerous methods: "user". We realized that the absence of a logged-in user doesn't mean that there is *no* user; only that we are dealing with a special kind of *anonymous* user. This realization enabled us to "push back" against the design of the system from the method construction level. We pushed the differences between authenticated and guest users out of the individual methods, and into the class hierarchy. By starting from the point of view of the code we *wanted* to write at the method level, we arrived at a different, and likely better, object model of the business domain.

However, changes like this don't always have to be made all at once. We could have made this change in a single method, and then propagated it further as time allowed or as new features gave us a reason to touch other areas of the codebase. Let's look at how we might go about that.

We'll use the example of the `#greeting` method. Here's the starting code:

```
def greeting
  "Hello, " +
    current_user ? current_user.name : "Anonymous" +
    ", how are you today?"
end
```

We know the role we want to deal with (a user, whether logged in or not). We don't want to compromise on the clarity and confidence we can achieve by writing this method in terms of that role. But we're not ready to pick through the whole codebase switching `nil` tests to use the new `GuestUser` type. Instead, we introduce the use of that new class in only one place. Here's the code with the `GuestUser` introduced internally to the method:

```
def greeting
  user = current_user || GuestUser.new(session)
  "Hello, #{user.name}, how are you today?"
end
```

(In his book Working Effectively with Legacy Code, Michael Feathers calls this technique for introducing a new class *sprouting a class*.)

`GuestUser` now has a foothold. `#greeting` is now a "pilot program" for this redesign. If we like the way it plays out inside this one method, we can then proceed to try the same code in others. Eventually, we can move the creation of `GuestUser` into the `#current_user` method, as shown previously, and then eliminate the piecemeal creation of `GuestUser` instances in other methods.

Keeping the special case synchronized

Note that Special Case is not without drawbacks. If a Special Case object is to work anywhere the "normal case" object is used, their interfaces need to be kept in sync.

For simple interfaces it may simply be a matter of being diligent in updating the Special Case class, along with integration tests that exercise both typical and special-case code paths.

For more complex interfaces, it may be a good idea to have a shared test suite that is run against both the normal-case and special-case classes to verify that they both respond to the same set of methods. In codebases that use RSpec, a shared example group is one way to capture the shared interface in test form.

```
shared_examples_for 'a user' do
  it { should respond_to(:name) }
  it { should respond_to(:authenticated?) }
  it { should respond_to(:has_role?) }
  it { should respond_to(:visible_listings) }
  it { should respond_to(:last_seen_online=) }
  it { should respond_to(:cart) }
end

describe GuestUser do
  subject { GuestUser.new(stub('session')) }
  it_should_behave_like 'a user'
end

describe User do
  subject { User.new }
  it_should_behave_like 'a user'
end
```

Obviously this doesn't capture all the expected semantics of each method, but it functions as a reminder if we accidentally omit a method from one class or the other.

Conclusion

When a special case must be taken into account at many points in a program, it can lead to the same `nil` check over and over again. These endlessly repeated tests for object existence clutter up code. And it's all too easy to introduce defects by missing a case where we should have used another `nil` test.

By using a Special Case object, we isolate the differences between the typical case and the special case to a single location in the code, and let polymorphism ensure that the right code gets executed. The end product is code that reads more cleanly and succinctly, and which has better partitioning of responsibilities.

Control statements that switch on whether an input is `nil` are red flags for situations where a Special Case object may be a better solution. To avoid the conditional, we can introduce a class to represent the special case, and instantiate it within the method we are presently working on. Once we've established the Special Case class and determined that it improves the flow and organization of our code, we can refactor more methods to use the instances of it instead of conditionals.

Chapter 3

Delivering Results

3.1 Represent failure with a benign value

> The system might replace the erroneous value with a phony value that it knows to have a benign effect on the rest of the system.
>
> — Steve McConnell, Code Complete

Indications

A method returns a nonessential value. Sometimes the value is `nil`.

Synopsis

Return a default value, such as an empty string, which will not interfere with the normal operation of the caller.

Rationale

Unlike `nil`, a benign value does not require special checking code to prevent `NoMethodError` being raised.

Example: Rendering tweets in a sidebar

Let's say we're constructing a company home page, and as part of the page we want to include the last few tweets from the corporate Twitter account.

```
def render_sidebar
  html = ""
  html << "<h4>What we're thinking about...</h4>"
  html << "<div id='tweets'>"
  html << latest_tweets(3) || ""
  html << "</div>"
end
```

See that conditional when calling out to the `#latest_tweets` helper method? That's because sometimes our requests to the Twitter API fail, and when they do the method returns `nil`.

```
def latest_tweets(number)
  # ...fetch tweets and construct HTML...
rescue Net::HTTPError
  nil
end
```

Feeding `nil` to `String#<<` gives rise to a `TypeError`, necessitating the special handling of the `#latest_tweets` return value.

Do we really need to represent the error case with `nil` here, forcing callers to check for that possibility? In this case returning the empty string on failure would probably be a more humane interface.

```
def latest_tweets(number)
  # ...fetch tweets...
rescue Net::HTTPError
  ""
end
```

Now we can construct HTML without a nil-checking `||`:

```
html << latest_tweets(3)
```

If callers *really* need to find out if the request succeeded, they can always check to see if the returned string is empty.

```
tweet_html = latest_tweets(3)
if tweet_html.empty?
  html << '(unavailable)'
else
  html << tweet_html
end
```

Conclusion

`nil` is the worst possible representation of a failure: it carries no meaning but can still break things. An exception is more meaningful, but some failure cases aren't really exceptional. When a return value is used but non-essential, a workable but semantically blank object—such as an empty string—may be the most appropriate result.

Chapter 4

Handling Failure

4.1 Use bouncer methods

Indications

An error is indicated by program state rather than by an exception. For instance, a failed shell command sets the `$?` variable to an error status.

Synopsis

Write a method to check for the error state and raise an exception.

Rationale

Like checked methods, bouncer methods DRY up common logic, and keep higher-level logic free from digressions into low-level error-checking.

Example: Checking for child process status

In the last section, we looked at a method for filtering a message through a shell command. The method uses `IO.popen` to execute the shell command.

```
def filter_through_pipe(command, message)
  checked_popen(command, "w+", ->{message}) do |process|
    process.write(message)
    process.close_write
    process.read
  end
end
```

When the shell command finishes, `IO.popen` sets the `$?` variable to a `Process::Status` object containing, among other things, the command's exit

status. This is an integer indicating whether the command succeeded or not. A value of `0` means success; any other value typically means it failed.

In order to verify that the command succeeded, we have to interrupt the flow of the method with some code that checks the process exit status and raises an error it indicates an error. This is nearly as distracting as a `begin/rescue/end` block.

```
def filter_through_pipe(command, message)
  result = checked_popen(command, "w+", ->{message}) do |process|
    process.write(message)
    process.close_write
    process.read
  end
  unless $?.success?
    raise ArgumentError,
    "Command exited with status "\
    "#{$?.exitstatus}"
  end
  result
end
```

Enter the Bouncer Method. A Bouncer Method is a method whose sole job is to raise an exception if it detects an error condition. In Ruby, we can write a bouncer method which takes a block containing the code that may generate the error condition. The bouncer method below encapsulates the child process status-checking logic seen above.

```
def check_child_exit_status
  unless $?.success?
    raise ArgumentError,
    "Command exited with status "\
    "#{$?.exitstatus}"
  end
end
```

We can add a call the bouncer method after the #popen call is finished, and it will ensure that a failed command is turned into an exception with a minimum of disruption to the method flow.

```
def filter_through_pipe(command, message)
  result = checked_popen(command, "w+", ->{message}) do |process|
    process.write(message)
    process.close_write
    process.read
  end
  check_child_exit_status
  result
end
```

An alternative version has the code to be "guarded" by the bouncer executed inside a block passed to the bouncer method.

```
def check_child_exit_status
  result = yield
  unless $?.success?
    raise ArgumentError,
    "Command exited with status "\
    "#{$?.exitstatus}"
  end
  result
end

def filter_through_pipe(command, message)
  check_child_exit_status do
    checked_popen(command, "w+", ->{message}) do |process|
      process.write(message)
      process.close_write
      process.read
    end
  end
end
```

This eliminates the need to save a local `result` variable, since the bouncer method is written to return the return value of the block. But it imposes awareness of the exit status checking at the very top of the method. I'm honestly not sure which of these styles I prefer.

Conclusion

Checking for exceptional circumstances can be almost as disruptive to the narrative flow of code as a `begin/rescue/end` block. What's worse, it can often take some deciphering to determine what, exactly the error checking code is looking for. And the checking code is prone to being duplicated anywhere the same error can crop up.

A bouncer method minimizes the interruption incurred by error-checking code. When given an intention-revealing name, it can clearly and concisely reveal to the reader exactly what potential failure is being detected. And it can DRY up identical error-checking code in other parts of the program.