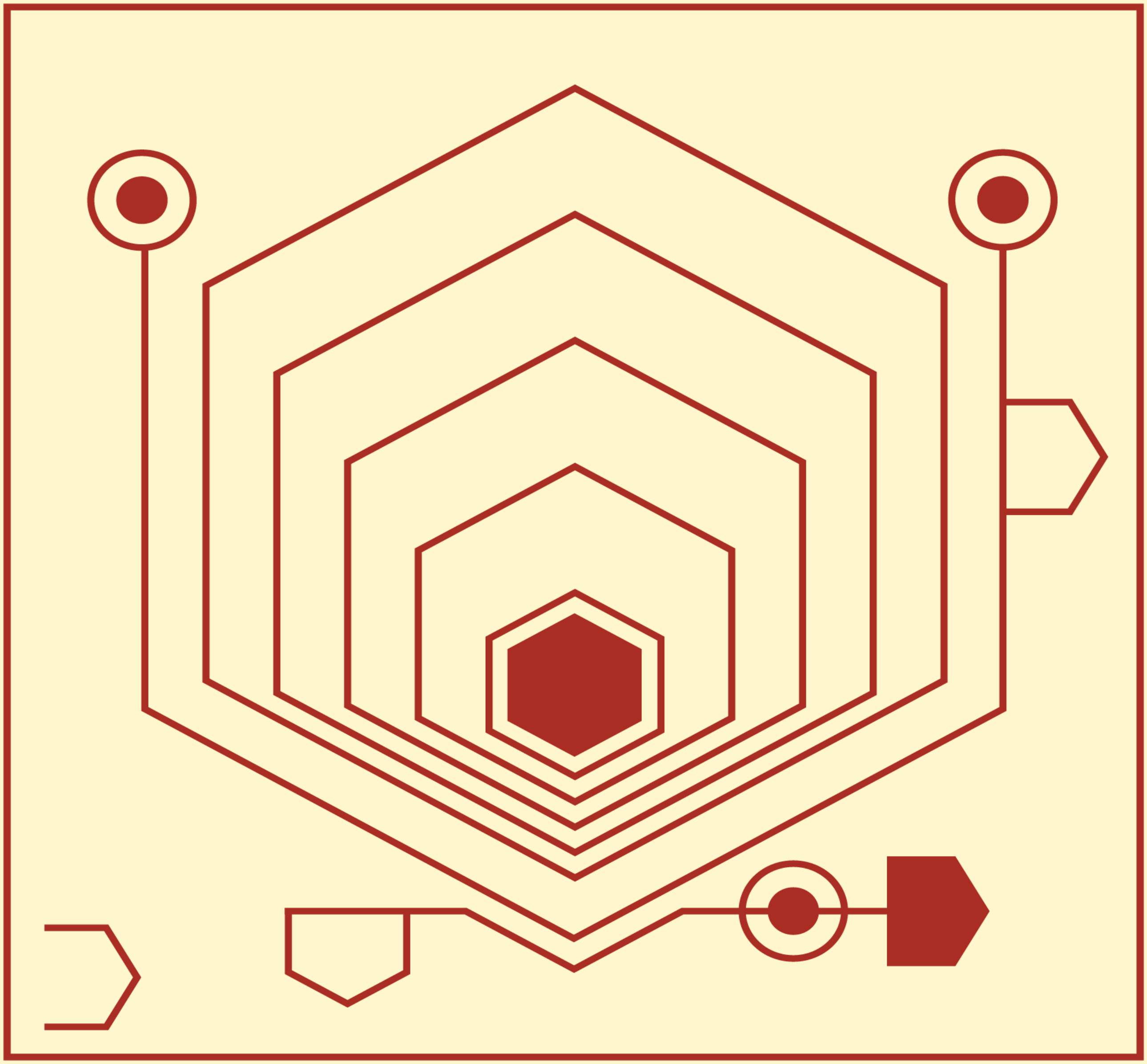

Confident Ruby

by Avdi Grimm

4.17 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:

```ruby
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:

```ruby
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`.

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

This simplifies the greeting code nicely.

```ruby
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`.

```ruby
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:

```ruby
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.