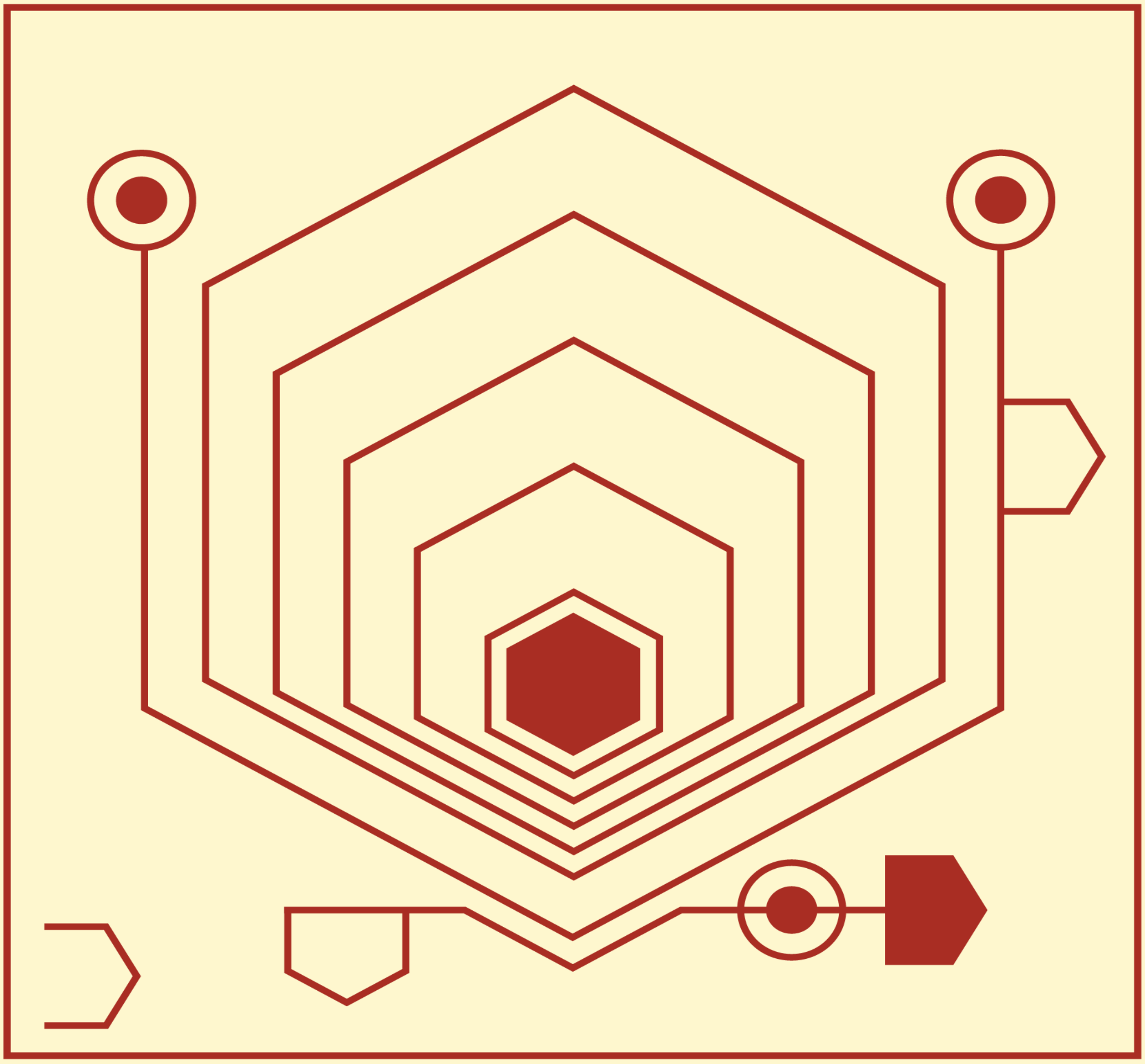

Confident Ruby

by Avdi Grimm

5.3 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.

6.3 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.

```ruby
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.

```ruby
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.