RATFT

Anthony Eden
@aeden
DNSimple

It’s cute and its cuddly. It gives you dumb looks, sticks out its tongue and walks funny.

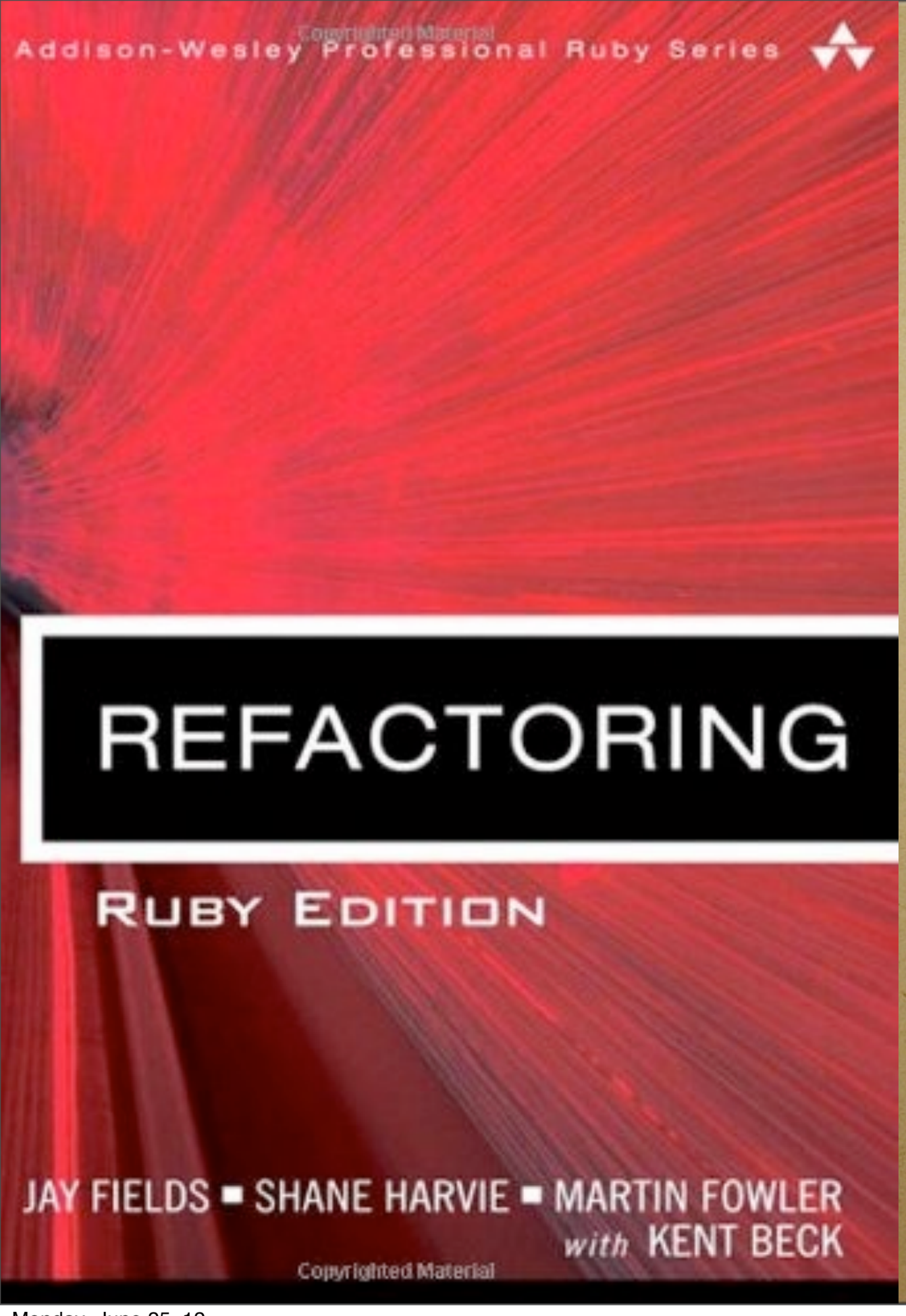
Eventually it grows up and can take an arm off. Software: same thing.

According to Kent Beck
What makes programs hard to work with?
Programs that are hard to read are hard to modify.
Programs that have duplicated logic are hard to modify.
Programs that require additional behavior that requires you to change running code are hard to modify.
Programs with complex conditional logic are hard to modify.

Taming the Beast

Why should you refactor?
Martin Fowler's thoughts:

Refactoring improves the design of software.
Refactoring makes software easier to understand.
Refactoring helps you find bugs.
Refactoring helps you program faster.
Refactoring helps you develop software more rapidly, because it stops the design of the system from decaying.



How-to Guide

Describes a variety of code smells.
Provides a catalog of refactorings with examples.

Tests for Safety

The only way to refactor is to have a baseline set of tests for what is being refactored. Since refactoring is not supposed to change the behavior you need to have the behavior defined first.

I can haz code?

```ruby
class Domain
  # ...methods
  def self.check(*name_or_names)
    name_or_names = *name_or_names
    query = base_query.merge('Command' => 'Check')

    if name_or_names.is_a?(Array)
      query = query.merge('DomainList' => name_or_names.join(","))
    else
      sld, tld = parse(name_or_names)
      query = query.merge('SLD' => sld, 'TLD' => tld)
    end
    response = execute_command(query)

    results = {}

    if response['DomainCount']
      1.upto(response['DomainCount'].to_i) do |i|
        results[response["Domain#{i}"]] = response["RRPCode#{i}"] == '210'
      end
    else
      results[response['DomainName']] = response['RRPCode'] == '210'
    end

    results
  end
  # ...yet more methods
end
```


The original code, no tests

```ruby
require 'rspec'
require './domain'

describe Domain do
  it "works" do
    result = Domain.check
    assert_equal({}, result)
  end
end
```


Create a basic test to discover what the code is doing

```ruby
describe Domain do
  it "works" do
    Domain.stub(:base_query).and_return({})
    result = Domain.check
    assert_equal({}, result)
  end
end
```


base_query is used somehow, it looks like a Hash, so let's start with that.

```ruby
describe Domain do
  it "works" do
    Domain.stub(:base_query).and_return({})
    Domain.stub(:execute_command).and_return({})
    result = Domain.check
    assert_equal({}, result)
  end
end
```


The response looks like a Hash as well...

```ruby
describe Domain do
  it "works" do
    Domain.stub(:base_query).and_return({})
    Domain.stub(:execute_command).and_return({
      'DomainCount' => 1,
      'Domain1' => 'example.com',
      'RRPCode1' => '210'
    })
    results = Domain.check("example.com")
    results.should eq({"example.com" => true})
  end
end
```


We are obliged to make some guesses about the response based on what's in the code. It looks like the hash has some keys that are interesting and that the results may vary.

In one version there may be a DomainCount which is a number. This number is then used to iterate through the response data to retrieve a domain and an RRPCode. There also appears to be a magic value of 210 which indicates "success"

```ruby
describe Domain do
  context "with a domain count" do
    it "works" do
      # snip
    end
  end

  context "without a domain count" do
    it "works" do
      Domain.stub(:base_query).and_return({})
      Domain.stub(:execute_command).and_return({
        'DomainName' => 'example.com',
        'RRPCode' => '210'})
      results = Domain.check("example.com")
      results.should eq({"example.com" => true})
    end
  end
end
```


There is also another branch in the case where DomainCount doesn't exist.

```
context "with an Array of names" do
  it "works" do
    Domain.stub(:base_query).and_return({})
    Domain.stub(:execute_command).and_return({
      'DomainCount' => 2,
      'Domain1' => 'example.com',
      'RRPCode1' => '210',
      'Domain2' => 'example2.com',
      'RRPCode2' => '210'
    })
    results = Domain.check("example.com", "example2.com")
    results.should eq({
      "example.com" => true, 'example2.com' => true
    })
  end
end
```


There’s another if/else branch as well: If name or names is an Array then do something otherwise do something else
Type checking isn't the best thing to do here, but we can address that later. For now let's get tests for the branching.

```
class CheckDomain

end
```


First let's extract the method in its entirety using Create Method Object refactoring.

```ruby
class Domain
  # ...methods
  def self.check(*name_or_names)
```


The method object will be initialized with the name_or_names as that argument is originally passed into the .check method.

```ruby
class CheckDomain
  attr_reader :name_or_names

  def initialize(name_or_names)
    @name_or_names = *name_or_names
  end

  def call
    # Move code here
  end
end
```


The Method Object will need to be invoked. I'll use call() for that.

```
query = base_query.merge('Command' => 'Check')

response = execute_command(query)
```


We also need two elements from the Domain class: the results of the base_query method call and an execute_command method.

```
def call(base_query)
```


The base_query seems to be a "query" method on the Domain class, constructing some sort of a base Hash that houses some common key/value pairs for queries, so I pass that in to the call() method directly.

```
def call(base_query, command_executor)
```


The execute_command method on the other hand appears to be an "action" method on the Domain class. In this case I pass in a reference to the Domain class itself calling in the command_executor.

```ruby
def call(base_query, command_executor)
  query = base_query.merge('Command' => 'Check')

  if name_or_names.is_a?(Array)
    query = query.merge('DomainList' => name_or_names.join(","))
  else
    sld, tld = parse(name_or_names)
    query = query.merge('SLD' => sld, 'TLD' => tld)
  end
  response = command_executor.execute_command(query)

  results = {}

  if response['DomainCount']
    1.upto(response['DomainCount'].to_i) do |i|
      results[response["Domain#{i}"]] = response["RRPCode#{i}"] == '210'
    end
  else
    results[response['DomainName']] = response['RRPCode'] == '210'
  end

  results
end
```


Finally the body gets copied in. To body is changed slightly to invoke execute_command on the command executor class passed in.

Note that this still allows the specs to pass, which is the goal. We are not changing the specs.

```ruby
class Domain
  def self.check(*name_or_names)
    CheckDomain.new(name_or_names).call(base_query, self)
  end
end
```


The Domain.check method now looks like this

- Builds a query which is a Hash.
- Executes that query which returns a Hash.
- Determines success or failure.
- Builds and returns a result Hash.

Monday, June 25, 12

Let’s look at the body of the call method now.

* Builds a query which is a Hash.
* Executes that query which returns a Hash response.
* Checks the results of the response to determine what should be returned as the result.
* Builds and returns a result Hash.

```ruby
def call(base_query, command_executor)
  query = base_query.merge('Command' => 'Check')

  if name_or_names.is_a?(Array)
    query = query.merge('DomainList' => name_or_names.join(","))
  else
    sld, tld = parse(name_or_names)
    query = query.merge('SLD' => sld, 'TLD' => tld)
  end
  response = command_executor.execute_command(query)

  results = {}

  if response['DomainCount']
    1.upto(response['DomainCount'].to_i) do |i|
      results[response["Domain#{i}"]] = response["RRPCode#{i}"] == '210'
    end
  else
    results[response['DomainName']] = response['RRPCode'] == '210'
  end

  results
end
```


Builds a query which is a Hash.

```ruby
def call(base_query, command_executor)
  query = base_query.merge('Command' => 'Check')

  if name_or_names.is_a?(Array)
    query = query.merge('DomainList' => name_or_names.join(","))
  else
    sld, tld = parse(name_or_names)
    query = query.merge('SLD' => sld, 'TLD' => tld)
  end
  response = command_executor.execute_command(query)

  results = {}

  if response['DomainCount']
    1.upto(response['DomainCount'].to_i) do |i|
      results[response["Domain#{i}"]] = response["RRPCode#{i}"] == '210'
    end
  else
    results[response['DomainName']] = response['RRPCode'] == '210'
  end

  results
end
```


Executes that query which returns a Hash response.

```ruby
def call(base_query, command_executor)
  query = base_query.merge('Command' => 'Check')

  if name_or_names.is_a?(Array)
    query = query.merge('DomainList' => name_or_names.join(","))
  else
    sld, tld = parse(name_or_names)
    query = query.merge('SLD' => sld, 'TLD' => tld)
  end
  response = command_executor.execute_command(query)

  results = {}

  if response['DomainCount']
    1.upto(response['DomainCount'].to_i) do |i|
      results[response["Domain#{i}"]] = response["RRPCode#{i}"] == '210'
    end
  else
    results[response['DomainName']] = response['RRPCode'] == '210'
  end

  results
end
```


* Checks the results of the response to determine what should be returned as the result.
* Builds and returns a result Hash.

```ruby
private
def query(base_query)
  query = base_query.merge('Command' => 'Check')
  if name_or_names.is_a?(Array)
    query = query.merge('DomainList' => name_or_names.join(","))
  else
    sld, tld = parse(name_or_names)
    query = query.merge('SLD' => sld, 'TLD' => tld)
  end
  query
end
```


The first thing is to isolate it's pieces, starting with the query building code. Here we use Extract Method (note that when I'm extracting to an internal method I set the access level at private. It is good practice to start with the most restrictive level of visibility and then only expose methods that are purposefully meant to be the API of the class.

The base_query is passed in as it is local to the call method and needs to be made available to the query builder.

```
def call(base_query, command_executor)
  response = command_executor.execute_command(
    query(base_query))

  # ...
end
```


The call method is now updated to call the query method.

```ruby
def results(response)
  results = {}
  if response['DomainCount']
    1.upto(response['DomainCount'].to_i) do |i|
      results[response["Domain#{i}"]] = response["RRPCode#{i}"] == '210'
    end
  else
    results[response['DomainName']] = response['RRPCode'] == '210'
  end
  results
end
```


Next extract the code to process the response from the execute_command method, also using the Extract Method refactoring

```
def call(base_query, command_executor)
  response = command_executor.execute_command(
    query(base_query))
  results(response)
end
```


The logic in the call is now only two lines

```
def call(base_query, command_executor)
  results(command_executor.execute_command(query(base_query)))
end
```


Use the Remove Temp Variable refactoring to clear out the unnecessary temp variable.

```
def call(base_query, command_executor)
  handle_response(command_executor.execute_command(
    build_query(base_query)))
end
```


The call() body now appears quite like a functional implementation, but the names of the internal methods seem a bit *too* general: they’re not querying, they’re doing. Let's use the Rename Method refactoring.

```ruby
SUCCESSFUL_RRP_CODE = '210'
FIRST_INDEX = 1
def handle_response(response)
  results = {}
  if response['DomainCount']
    FIRST_INDEX.upto(response['DomainCount'].to_i) do |i|
      results[response["Domain#{i}"]] = response["RRPCode#{i}"] ==
SUCCESSFUL_RRP_CODE
    end
  else
    results[response['DomainName']] = response['RRPCode'] ==
SUCCESSFUL_RRP_CODE
  end
  results
end
```


I could stop here, but there are a lot of magic values being thrown around, maybe I should do something about that. Duplication is a code smell as are magic values. Let's get that '210' into something more descriptive and define something to describe the magic value of 1.

```ruby
class CheckDomainResponse
  def initialize(response)
    @response = response
  end

  def [](attribute_name)
    @response[attribute_name]
  end
end
```


That's ok, but there seems to be a lot of behavior associated with the response, so maybe it should be its own class.

```ruby
class CheckDomainResponse
  def initialize(response)
    @response = response
  end

  def [](attribute_name)
    @response[attribute_name]
  end

  def multiple_domains?
    !@response['DomainCount'].nil?
  end

  def domain_count
    @response['DomainCount'].to_i
  end
end
```


Next we will add methods to replace those magic strings

```ruby
def handle_response(response)
  results = {}
  if response.multiple_domains?
    FIRST_INDEX.upto(response.domain_count) do |i|
      results[response["Domain#{i}"]] = response["RRPCode#{i}"] == SUCCESSFUL_RRP_CODE
    end
  else
    results[response['DomainName']] = response['RRPCode'] == SUCCESSFUL_RRP_CODE
  end
  results
end
```


The updated handle_response method

```ruby
class CheckDomainResponse
  def initialize(response)
    @response = response
  end

  SUCCESSFUL_RRP_CODE = '210'
  def results
    # move over from handle_response
  end

  private
  def multiple_domains?
    !@response['DomainCount'].nil?
  end

  def domain_count
    @response['DomainCount'].to_i
  end
end
```


The next step is to handle the single vs. multiple domain cases cleanly. First move the handle_response logic out of the check domain class and into the check domain response. All of the query methods can be made private now as well and since nothing is accessing the Hash accessor that can be removed.

```ruby
def call(base_query, command_executor)
  CheckDomainResponse.new(
    command_executor.execute_command(build_query(base_query))
  ).results
end
```


Here is the CheckDomain call method now.

```
def results
  if multiple_domains?
    multiple_domains_results
  else
    single_domain_results
  end
end
```


In the CheckDomainResponse I am now left with an implementation that seems to indicate polymorphism is appropriate, however I don't know if I really want that for just two choices. First I'll extract the result Hash building for each type into its own method:

```ruby
SUCCESSFUL_RRP_CODE = '210'
def multiple_domains_results
  results = {}
  FIRST_INDEX.upto(domain_count) do |i|
    results[@response["Domain#{i}"]] = @response["RRPCode#{i}"] == SUCCESSFUL_RRP_CODE
  end
  results
end

def single_domain_results
  {@response['DomainName'] => @response['RRPCode'] == SUCCESSFUL_RRP_CODE}
end
```

```ruby
FIRST_INDEX = 1
def multiple_domains_results
  results = {}
  FIRST_INDEX.upto(domain_count) do |i|
    results[@response["Domain#{i}"]] = successful?(@response["RRPCode#{i}"])
  end
  results
end

def single_domain_results
  {@response['DomainName'] => successful?(@response['RRPCode'])}
end

def successful?(rrp_code)
  rrp_code == SUCCESSFUL_RRP_CODE
end
```


Next I remove duplication around the RRP code check.

```ruby
def domain_key(index=nil)
  index ? "Domain#{index}" : 'DomainName'
end

def rrp_code_key(index=nil)
  index ? "RRPCode#{index}" : 'RRPCode'
end
```


There's something that's really bothering me about the keys used. I notice that they are different depending on whether the response is singular or multiple. Ugh. It turns out that the API documentation for the service provider is a bit unclear about this so I want to isolate those keys since I have a feeling the provider might change them in the future plus it seems like determining the keys is not something that should be done while checking the response.

```ruby
SUCCESSFUL_RRP_CODE = '210'
FIRST_INDEX = 1
def multiple_domains_results
  results = {}
  FIRST_INDEX.upto(domain_count) do |i|
    results[@response[domain_key(i)]] = successful?(@response[rrp_code_key(i)])
  end
  results
end

def single_domain_results
  {@response[domain_key] => successful?(@response[rrp_code_key])}
end
```


Here is how the rrp_code_key method is used.

```ruby
def build_query(base_query)
  query = base_query.merge('Command' => 'Check')
  if name_or_names.is_a?(Array)
    query = query.merge('DomainList' => name_or_names.join(","))
  else
    sld, tld = parse(name_or_names)
    query = query.merge('SLD' => sld, 'TLD' => tld)
  end
  query
end
```


At this point I feel like the next step would be refactoring the condition to polymorphism, however I have reservations about it because I don't know if its necessary. Instead I'm going to go back and look at the query builder.

```ruby
class CheckDomainQuery
  attr_reader :name_or_names
  def initialize(*name_or_names)
    @name_or_names = *name_or_names
  end
  def to_hash(base_query)
    query = base_query.merge('Command' => 'Check')
    if name_or_names.is_a?(Array)
      query = query.merge('DomainList' => name_or_names.join(","))
    else
      sld, tld = parse(name_or_names)
      query = query.merge('SLD' => sld, 'TLD' => tld)
    end
    query
  end
end
```


First extract the method body into a new class.

```ruby
def build_query(base_query)
  CheckDomainQuery.new(name_or_names).to_hash(base_query)
end
```


The build query method becomes this.

```ruby
def initialize(*name_or_names)
   @name_or_names = *name_or_names
end
```


CheckDomainQuery sure seems to want to be polymorphic, but there's something that's bothering me for quite some time: what is this about?

The first splat constructs an array with the method parameters. The second splat is not really doing anything. It’s converting an Array into a collection of arguments that is one Array and assigning the first argument to the name_or_names local variable.

Monday, June 25, 12

WTF

```ruby
class CheckDomainQuery
  attr_reader :name_or_names
  def initialize(*name_or_names)
    @name_or_names = *name_or_names
  end
  def to_hash(base_query)
    query = base_query.merge('Command' => 'Check')
    query = query.merge('DomainList' => name_or_names.join(","))
    query
  end
end
```


Is there *ever* a case where the result will be anything other than an Array? The answer is no. So the CheckDomainQuery can remain a single class without the need for polymorphism.

I'll stop here. However I would like to show you how the code currently looks...

```ruby
class CheckDomain
  RRP_CHECK_SUCCESS_CODE = '210'

  attr_reader :names

  def initialize(*name_or_names)
    @names = *name_or_names
  end

  def call(base_query, command_executor)
    query = base_query.merge('Command' => 'Check')
    query = query.merge('DomainList' => names.join(","))
    results(command_executor.execute_command(query))
  end
```


Here is the public API.

```ruby
private
def results(response)
  if multiple_domains?(response)
    results = {}
    1.upto(domain_count(response)) do |i|
      results[response[domain_key(i)]] = success?(response[rrp_code_key(i)])
    end
    results
  else
    {response[domain_key] => success?(response[rrp_code_key])}
  end
end

def domain_count(response)
  response['DomainCount'].to_i
end

def multiple_domains?(response)
  !response['DomainCount'].nil?
end

def domain_key(index=nil)
  index ? "Domain#{index}" : "DomainName"
end

def rrp_code_key(index=nil)
  index ? "RRPCode#{index}" : "RRPCode"
end

def success?(rrp_code)
  rrp_code == RRP_CHECK_SUCCESS_CODE
end
```


Here is the internal API.

Refactorings do not change the behavior of a program.

The purpose of refactoring is to make the software easier to understand and modify.

Tests are required.

Make small changes.

Don’t be afraid to back out of a refactoring.

Become fluent in the vocabulary of refactoring.

- Refactor one of your own classes.
- Refactor a class created by someone else.
- If you use other languages think about how refactoring applies to them as well.

RATFT

Anthony Eden
@aeden
DNSimple

flickr.com/photos/play4smee/795198106/
flickr.com/photos/rentman1225/2150085387/
flickr.com/photos/inannabintali/2858023183
flickr.com/photos/tyrian123/507139524/