

# ruby\_cool\_kid.rb — Meta Programming series: Ghost Methods

*Pablo Adell*

7-9 minutes

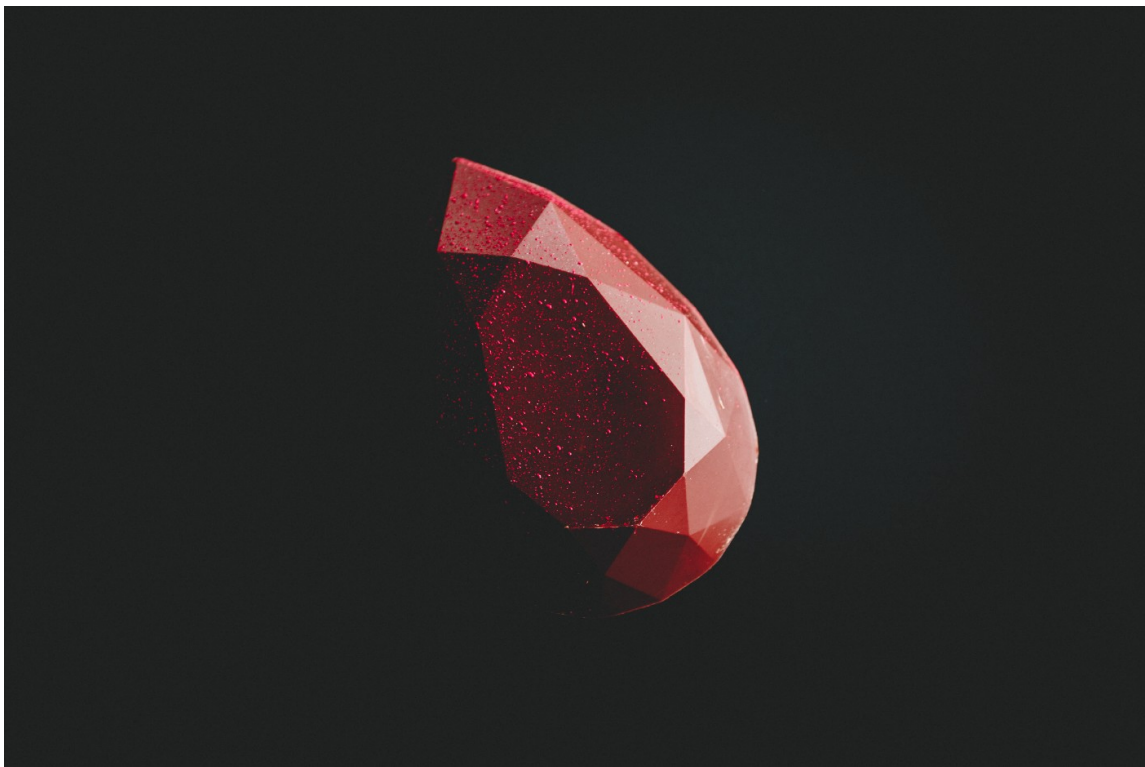


Photo by [Joshua Fuller](#) on [Unsplash](#)

Following my ruby series, here is the next chapter, where I will be diving into Ruby's `method_missing` and its pros and cons.

The other day, I was thinking about writing this story when one of my colleagues heard me talking about Paolo Perrotta. He sent me one of his conferences from 2012 ([reference](#)). On it, Perrotta talked about the advantages of using `method_missing` to

avoid code duplication on a Ruby app. However, *with every great power, comes great responsibility* (deep hero voice 🦸)

In a previous story, we talked about [dynamic dispatching](#) and the fact that **Ruby, as a language, is interpreted and not compiled, allowing** us to do things out of normal. For example, **defining a method on runtime** when needed. However, that is not the only way of getting rid of duplicated code.

In this story, following Perrotta's talk, we are going to talk about *Ghost methods* and the use of `method_missing`. Let's take the initial code we had on a previous example

```
class BirthdayCongratulations
  def initialize(name)
    @name = name
  end

  def congratulate
    __send__("congratulate_#{@name}")
  end

  def congratulate_mom
    "Happy Birthday Mom!! I love you"
  end

  def congratulate_dad
    "Happy Birthday Dad!! I love you"
  end
end

birthday_congratulations = BirthdayCongratulations.new('mom')
puts birthday_congratulations.congratulate # Happy Birthday Mom!! I love you
```

Base code case

First, in order to avoid code duplication, we can try and extract the person to be congratulated to another class, let's say **BirthdayPerson**. Additionally, let's try to up the notch a bit and suppose we have a file containing all the people we can congratulate for their birthday, **BirthdayDatabase**.

Let's define **BirthdayPerson**:

```
class BirthdayPerson

  def initialize
    allowed_people.each { |person| BirthdayPerson.create_birthday person }
  end

  private

  def self.create_birthday(person)
    define_method "congratulate_#{person.to_s}" do
      "Happy birthday #{person.upcase}!! I love you <3"
    end
  end

  def allowed_people
    # [:mom, :dad, :grandma, :sister] --> Data coming from our Database
    @birthdays ||= BirthdayDatabase.new.available_birthdays
  end
end
```

### BirthdayPerson class

Here we have a couple of interesting points to note. First, take a look at the initialisation method.

**allowed\_people** returns all the different people we have in our “database” that we could want to congratulate, each one of them being returned as a symbol.

Then by calling **create\_birthday** we will define each one of the available methods to congratulate the people in our “database”.

As the reader can observe, the way of defining the methods is quite different from the method we used in a [previous story](#). The reason for that change lays in the fact that we are defining methods at the instance level, rather than the singleton level, so that all instances of the class share the same methods.

A **singleton method** is a method only available for a “single” instance of the object

Thus, `define_method` ([reference](#)) will define the methods resulting from the people in our database.

We have now an interface to map the people in our database to methods that can be called or “sent” 😊

It is time then to refactor our base code and remove code duplication.

But first, a small note on Ruby runtime.

In Ruby, there is no compiler that enforces method calls due to its dynamic nature. This means that you could call a method that doesn't exist

But, if that is the case, ***what is happening when a non-existent method is called? What is returning `NoMethodError: undefined method 'method'`? Where do all those send methods go, spam folder? (Sorry for the bad joke 😊)*** .Too many questions, very few answers.

Well, **what starts in Ruby ends in Ruby**. Let's bring `method_missing` and [ancestors](#) into the stage.

When Ruby does **not find a method** in a **Class** or any of its **ancestors**, it **calls `method_missing`** . However, **it is just another method** that, as we saw in our [ancestors story](#), is called on each element of the `ancestors'` chain until responded. It is like that time you want to go for a run with your friends, you send the message hoping for it to be answered.

In this case, **`method_missing` will always be answered, if not by the closer ancestors, by `Object` and then `BasicObject`** , returning that sweet `NoMethodError` message. However, knowing what we know about Ruby, where method names are sent on the `ancestors'` chain and the

method named is sent from bottom to top, *how could we use this to our advantage and duplicate code?*

## Ghost methods

Knowing what we know on `method_missing` and our knowledge on Ruby ancestors we could try to play a little with it.

Let's imagine the first piece of code I showed you in this article:

```
class BirthdayCongratulations
  def initialize(name)
    @name = name
  end

  def congratulate
    __send__("congratulate_#{@name}")
  end

  def congratulate_mom
    "Happy Birthday Mom!! I love you"
  end

  def congratulate_dad
    "Happy Birthday Dad!! I love you"
  end
end

birthday_congratulations = BirthdayCongratulations.new("mom")
puts birthday_congratulations.congratulate # "Happy Birthday Mom!! I love you"
```

And refactor it to use our abstraction to **BirthdayPerson**:

```
require './birthday_person'

class BirthdayCongratulations
  def initialize(name)
    @name = name
    @congratulate = BirthdayPerson.new
  end

  def congratulate
    @congratulate.send("congratulate_#{@name}")
  end
end

birthday_congratulations = BirthdayCongratulations.new('dad')
birthday_congratulations.congratulate # => "Happy birthday Dad!! I love you <3"
```

Looks way better and simpler, right? We are just delegating the method to BirthdayPerson by sending it the method name.

However, we still have to define the congratulate method.

So, would it be possible to refactor the code in any way that no new method is defined? 🤔

What about using method\_missing and Ruby's ancestors?

That's it!

```
require './birthday_person'

class BirthdayCongratulations
  def initialize(name)
    @name = name
    @congratulate = BirthdayPerson.new
  end

  def method_missing(name, *args)
    @congratulate.send("congratulate_#{@name}")
  end
end

birthday_congratulations = BirthdayCongratulations.new('dad')
birthday_congratulations.congratulate # => "Happy birthday Dad!! I love you <3"
```

Using method missing to call method on BirthdayPerson

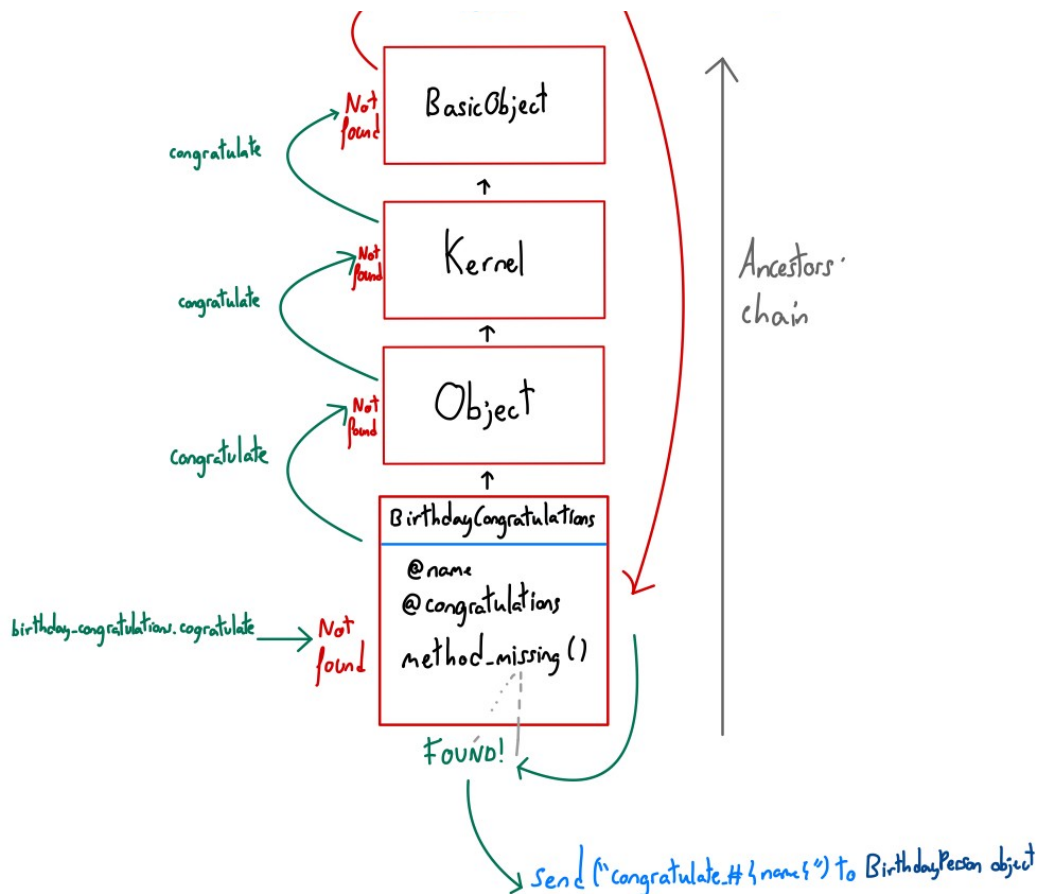
Boom!! If you take a look, you'd see that congratulate has been replaced by method\_missing, which will be the one sending the method name to our BirthdayPerson object.

Thus, saving us from defining a new method by taking advantage of Ruby's ancestors' chain.

But, how is this happening under the hood?

A drawing may help, right?





Schema on how method\_missing worked for the piece of code


The drawing above tries to explain the path the Interpreter of Ruby took to find the method sent. First, it tried to find it on the current object it was evaluating, an instance of BirthdayCongratulations. As it did not find it there, it went up on the ancestors' chain performing method\_lookup until the method was found.

When it got to the top of the chain without any match, the Interpreter called method\_missing on the first element on the chain, where the original call to the method was sent. Finding a match for method\_missing at that level, it proceeds to execute the code inside.

In this case, the code inside was a call on an instance of BirthdayPerson that will execute the desired code.

Such scenario and availability for a method that is not within the

object it gets called on, nor any of its ancestors, is called a Ghost method. It is there for the interpreter, but it is not really there. If we called `respond_to?` on `birthday_congratulations` the output would be:

A screenshot of a Ruby IRB session. The prompt is `irb(main):003:0>`. The user enters `birthday_congratulations.respond_to? :congratulate`. The output is `=> false`. The background is a dark grey terminal window with three colored window control buttons (red, yellow, green) in the top left corner.

```
irb(main):003:0> birthday_congratulations.respond_to? :congratulate
=> false
```

So 🤔 The method call works on the object, but the object does not `respond_to` the method? Looks like something that makes sound but is not there. A ghost!!

The concept of Ghost method is really interesting and can be useful when developing things like gems or frameworks, but it still can cause some problems for implementations that expect objects to `respond_to` the method called upon. This is not a huge deal normally but is better to be aware of. Also, of course, there is a solution 😊

## Respond\_to\_missing?

As the reader may have noticed by now, in Ruby nothing is what it seems.

So, why would be `respond_to??` If we inspect the method ([reference](#)), we would see that it calls `respond_to_missing?` if it can't find the method on the ancestors' chain of the class. Notice a behaviour similar to the one happening when the Interpreter can't find the method on the object.

Bearing this in mind, implementing `respond_to_missing?` is



key when applying the Ghost method pattern. If not implemented, again, it is not a big deal, but can cause problems if the object is used in another programs or in ways we did not anticipate. Better safe than sorry 😊

```
require './birthday_person'

class BirthdayCongratulations
  def initialize(name)
    @name = name
    @congratulate = BirthdayPerson.new
  end

  def method_missing(name, *args)
    @congratulate.send("congratulate_#{@name}")
  end

  def respond_to_missing?(name, *args)
    @congratulate.respond_to?("name_#{@name}") # Call respond to? to avoid method lies
  end
end

birthday_congratulations = BirthdayCongratulations.new('dad')
puts birthday_congratulations.respond_to? :congratulate # => true
```

Implementing respond\_to\_missing?

As of right now, you are almost a gurú on Ghost methods but there are still some nuances worth mentioning. But, let's leave it for another story as I think this one is dense enough.

I hope you enjoyed today's story. Don't be scared of ghosts 👻



See you around!!

Pablo