A Ruby Refactor: Dependency Injection Options | Brandon Hilkert

I recently wrote some code to interface with Stripe's webhooks. After looking at the code and tests, I decided I needed to do something to make it easier to test *all* pricing tiers— something I wasn't able to easily do from the start.

Dependency injection was a necessary piece of that puzzle. I've always been curious about the various forms of dependency injection and the effects each would have on the code. Below I explore 2 options (constructor injection and setter injection).

In the end, setter injection felt for more natural for this case and it didn't interfere with the classes argument list and felt ancillary to the responsibility of the code. While the change in code was small, it has a huge impact on my confidence in the code and associated tests.

The Code

The class below is responsible for handling Stripe's `invoice.created` webhook. Prior to a customer being billed monthly subscription, Stripe will ping your application (if configured) — giving you the opportunity to add additional line items (think metered billing…). It could be additional services, or perhaps the entire bill itself (this use case). Nevertheless, the responsibility of the class is to create an invoice item based on the customer's usage during the previous period.

```
module StripeEvent
  class InvoiceCreated
    attr_reader :payload

    def initialize(payload)
      @payload = payload
    end

    def perform
      Stripe::InvoiceItem.create(
        customer: user.stripe_id,
        amount: additional_charges_in_cents,
        currency: "usd",
        description: "Usage charges"
      )
    end

    private
```

```
    def additional_charges_in_cents
      Billing::Tier.new(usage).additional_charges_in_cents
    end

    def usage
      Billing::Usage.new(user).last_30_days
    end

    def user
      @user ||= User.find_by(stripe_id: payload["data"]["object"]["customer"])
    end
  end
end
```

I wrote this code pretty quickly and felt pretty good about it. The responsibility of determining the pricing tier had been broken out in to a separate class, as well as determining the customer's actual usage. At least I thought they were…

So what about the tests?

```
require 'test_helper'

class InvoiceCreatedTest < ActiveSupport::TestCase
  def setup
    @payload = {
      "data" => {
        "object" => {
          "customer" => "stripe_brandon"
        }
      }
    }
  end

  test 'adds invoice item based on usage' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 1900,
      currency: "usd",
      description: "Usage charges"
    ).returns(true)
    StripeEvent::InvoiceCreated.new(@payload).perform
  end
```

```
  test 'adds next level charge for usage' do
    Stat.create!(user: users(:brandon), step: steps(:nav_one), impressions: 3_000, date:
5.days.ago)

    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 4900,
      currency: "usd",
      description: "Usage charges"
    ).returns(true)
    StripeEvent::InvoiceCreated.new(@payload).perform
  end
end
```

The first thing I noticed with this setup was the detailed usage of `Stripe::InvoiceItem.expects`. I wasn't sure if this was necessarily a bad thing because it was a third-party service and it seemed like reasonable boundary of the application.

Aside from the mock, another thing that bothered me was the difficulty simulating different pricing tiers and customer usage. You probably noticed the `Stat.create!…` in the last test. I could've duplicated `Stat` entries until I reached some arbitrary level of usage that bumped this user to the next pricing tier. But that felt risky and very dependent on knowing the actual value of the subsequent tier.

What if I wanted to change the ceiling of that tier next month? I'd have to come in here and adjust the stats being created until it totaled something above the adjustment. It just felt weird…

What if we had a way to easily swap in implementations of the `Billing::Usage`? It would then allow me to concoct any combination of usage and mock the expected values sent to Stripe.

Setter Injection

In a few other articles, I've heard this termed "accessors as collaborators". Whatever the name, it was surprising how such a little a change could produce so much flexibility in my tests. And with that additional flexibility came confidence because it allowed me to test the edge cases with minimal overhead.

```
module StripeEvent
  class InvoiceCreated
    attr_writer :usage_service
    attr_reader :payload

    def initialize(payload)
      @payload = payload
    end
```

```
    def perform
      if user.created_at < 14.days.ago
        Stripe::InvoiceItem.create(
          customer: user.stripe_id,
          amount: additional_charges_in_cents,
          currency: "usd",
          description: "Usage charges"
        )
      end
    end

    private

    def additional_charges_in_cents
      Billing::Tier.new(usage).additional_charges_in_cents
    end

    def usage
      usage_service.last_30_days
    end

    def usage_service
      @usage_service ||= Billing::Usage.new(user)
    end

    def user
      @user ||= User.find_by(stripe_id: payload["data"]["object"]["customer"])
    end
  end
end
```

A couple things changed:

1. `usage_service` was created to extract the code to calculate customer usage
2. The `usage` method now calls the `last_30_days` method on `usage_service`

   This is interesting because you'll notice now that the only important idea about `usage_service` is the fact that is has a `last_30_days` method. We can now take comfort in the idea that `usage_service` could be anything really, as long as it implements the `last_30_days` method.

3. `attr_writer :usage_service` was added to allow for other implementations of the usage class

This allows us to inject other forms of the `usage_service` to simulate more or less customer usage:

```
require 'test_helper'

class InvoiceCreatedTest < ActiveSupport::TestCase
  def setup
    @payload = {
      "data" => {
        "object" => {
          "customer" => "stripe_brandon"
        }
      }
    }
  end

  test 'adds invoice item based on usage' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 1900,
      currency: "usd",
      description: "Usage charges"
    )
    StripeEvent::InvoiceCreated.new(@payload).perform
  end

  test 'adds next level charge for usage' do
    Stat.create!(user: users(:brandon), step: steps(:nav_one), impressions: 3_000, date:
5.days.ago)

    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 4900,
      currency: "usd",
      description: "Usage charges"
    )
    StripeEvent::InvoiceCreated.new(@payload).perform
  end

  test 'adds highest tier' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 49900,
      currency: "usd",
      description: "Usage charges"
    )
    inv = StripeEvent::InvoiceCreated.new(@payload)
```

```
  inv.usage_service = Level5Usage.new
  inv.perform
end

test 'adds 2nd highest tier' do
  Stripe::InvoiceItem.expects(:create).with(
    customer: "stripe_brandon",
    amount: 24900,
    currency: "usd",
    description: "Usage charges"
  )
  inv = StripeEvent::InvoiceCreated.new(@payload)
  inv.usage_service = Level4Usage.new
  inv.perform
end

test 'adds middle tier' do
  Stripe::InvoiceItem.expects(:create).with(
    customer: "stripe_brandon",
    amount: 12900,
    currency: "usd",
    description: "Usage charges"
  )
  inv = StripeEvent::InvoiceCreated.new(@payload)
  inv.usage_service = Level3Usage.new
  inv.perform
end

test 'adds 2nd tier' do
  Stripe::InvoiceItem.expects(:create).with(
    customer: "stripe_brandon",
    amount: 4900,
    currency: "usd",
    description: "Usage charges"
  )
  inv = StripeEvent::InvoiceCreated.new(@payload)
  inv.usage_service = Level2Usage.new
  inv.perform
end

test 'adds 1st tier' do
  Stripe::InvoiceItem.expects(:create).with(
    customer: "stripe_brandon",
    amount: 1900,
```

```
      currency: "usd",
      description: "Usage charges"
    )
    inv = StripeEvent::InvoiceCreated.new(@payload)
    inv.usage_service = Level1Usage.new
    inv.perform
  end

  private

  Level5Usage = Class.new { def last_30_days; 2_000_000; end }
  Level4Usage = Class.new { def last_30_days; 900_000; end }
  Level3Usage = Class.new { def last_30_days; 190_000; end }
  Level2Usage = Class.new { def last_30_days; 19_000; end }
  Level1Usage = Class.new { def last_30_days; 1_900; end }
```

I've created classes for each usage tier that implement the `last_30_days` method. In real life, this usage service is more complex, but we can test the complexity of it alone through unit tests. The responsibility of this class is to ensure invoice items are added to Stripe correctly, so removing the complexity of `Billing::Usage` form this test allows us to maximize this test's value and keep us isolated from the implementation of `Billing::Usage` — assuming it implements the `last_30_days` method.

Constructor Injection

Most dependency injection posts focus on constructor injection. The idea being that an implementation can be supplied. If not, a reasonable default will be provided. How might that change this scenario?

```
module StripeEvent
  class InvoiceCreated
    attr_reader :payload

    def initialize(payload, usage_service = Billing::Usage)
      @payload = payload
      @usage_service = usage_service
    end

    def perform
      if user.created_at < 14.days.ago
        Stripe::InvoiceItem.create(
          customer: user.stripe_id,
          amount: additional_charges_in_cents,
          currency: "usd",
          description: "Usage charges"
```

```
        )
      end
    end

    private

    def additional_charges_in_cents
      Billing::Tier.new(usage).additional_charges_in_cents
    end

    def usage
      @usage_service.new(user).last_30_days
    end

    def user
      @user ||= User.find_by(stripe_id: payload["data"]["object"]["customer"])
    end
  end
end
```

Because the `usage` method requires instantiation from within the class, I had to update the fake test Usage classes to accept `user` as an argument during instantiation:

```
require 'test_helper'

class InvoiceCreatedTest < ActiveSupport::TestCase
  def setup
    @payload = {
      "data" => {
        "object" => {
          "customer" => "stripe_brandon"
        }
      }
    }
  end

  test 'adds invoice item based on usage' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 1900,
      currency: "usd",
      description: "Usage charges"
    )
    StripeEvent::InvoiceCreated.new(@payload).perform
```

```
  end

  test 'adds next level charge for usage' do
    Stat.create!(user: users(:brandon), step: steps(:nav_one), impressions: 3_000, date:
5.days.ago)

    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 4900,
      currency: "usd",
      description: "Usage charges"
    )
    StripeEvent::InvoiceCreated.new(@payload).perform
  end

  test 'adds highest tier' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 49900,
      currency: "usd",
      description: "Usage charges"
    )
    inv = StripeEvent::InvoiceCreated.new(@payload, Level5Usage)
    inv.perform
  end

  test 'adds 2nd highest tier' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 24900,
      currency: "usd",
      description: "Usage charges"
    )
    inv = StripeEvent::InvoiceCreated.new(@payload, Level4Usage)
    inv.perform
  end

  test 'adds middle tier' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 12900,
      currency: "usd",
      description: "Usage charges"
    )
```

```
    inv = StripeEvent::InvoiceCreated.new(@payload, Level3Usage)
    inv.perform
  end

  test 'adds 2nd tier' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 4900,
      currency: "usd",
      description: "Usage charges"
    )
    inv = StripeEvent::InvoiceCreated.new(@payload, Level2Usage)
    inv.perform
  end

  test 'adds 1st tier' do
    Stripe::InvoiceItem.expects(:create).with(
      customer: "stripe_brandon",
      amount: 1900,
      currency: "usd",
      description: "Usage charges"
    )
    inv = StripeEvent::InvoiceCreated.new(@payload, Level1Usage)
    inv.perform
  end

  private

  Level5Usage = Struct.new(:user) { def last_30_days; 2_000_000; end }
  Level4Usage = Struct.new(:user) { def last_30_days; 900_000; end }
  Level3Usage = Struct.new(:user) { def last_30_days; 190_000; end }
  Level2Usage = Struct.new(:user) { def last_30_days; 19_000; end }
  Level1Usage = Struct.new(:user) { def last_30_days; 1_900; end }
```

The resulting test classes seem overly complex and sprinkled with details that aren't particularly relevant to its responsibility. If we were to pass in an already instantiated usage class as an argument, it means we would have to already know the user before-hand, which means we'd have to parse `@user ||= User.find_by(stripe_id: payload["data"]["object"]["customer”])` outside of this class. I don't love that solution — the parent that calls this `InvoiceCreated` class is pretty minimal and I wanted to keep it that way.

Another option would be to provide `user` as an argument to the to the `last_30_days` method:

```
def usage
```

```
  @usage_service.new.last_30_days(user)
end
```

We could then change our fake test usage classes back to:

```
Level1Usage = Class.new { def last_30_days; 1_900; end }
```

Notice the lack of `Struct` with an argument…

Summary

Of the two options, I prefer the setter injector in this case. There's something about changing the signature of a class just for tests that didn't feel natural.

An accessor (or writer…), in this case, provided the same flexibility without changing the signature. I like being able to quickly look at the argument list of `initialize` and clearly understand its roles and responsibilities within the system.

Which do you prefer?