What is Dependency Injection?

Posted on January 5, 2015 by gosukiwi in php, rails, ruby, webdev

If you are a developer, chances are you know what Object Oriented code is. You might have also heard about Object Oriented Design Patterns; Things like *single responsibilities*,*Dependency Injection* (DI) and *decoupled* code.

All those things help you write code that is easier to change and test. Beeing a quite abstract concept, they apply to a lot of programming languages, such as Java, Ruby and PHP. Each language has it's own way to do things, thus each language implements the concepts in a different way. Some implementations are similar, others not so much. In this article I'll talk about Dependency Injection, what it is, and especifically Ruby's implementation; I'll compare it with PHP's view on DI.

Can you tell me what DI is already?

Dependency Injection can be tricky because it is a very simple concept. For that reason it can get abstract pretty quickly. Let's demonstrate with an example:

```
1  class MyPORO
2    def greet(name)
3      Greeter.new.say_hi(name)
4    end
5  end
6
```

Right now, the `MyPORO` class has several dependencies. According to Practical Object Oriented Design In Ruby, an object depends on another when:

- It knows the name of another class
- It knows the name of a method of another class
- It knows the arguments a method requires
- It knows the order of those arguments

Now that we know how to identify dependencies, let's see how many our little PORO (Plain Old Ruby Object) has. Following the rules we can see it has **four** dependencies. We know the name of the class, two methods and an argument. This can't be good, ideally we want to avoid dependencies, that way our object is *decoupled* from the rest.

Here's when DI comes in. Let's inject the dependency in the constructor:

```ruby
class MyPoro
  attr_reader :greeter
  def initialize(greeter)
    @greeter = greeter
  end

  def greet(name)
    greeter.say_hi(name)
  end
end

```

By passing the dependency to our object, thus *injecting* it, we reduced the ammount of dependencies from four to two. All we know is that the `greeter` object has a `say_hi` method, and it takes one argument.

That's it, that's dependency injection! You can also inject dependencies in setter methods instead of the constructor, it still counts as DI. This simple implementation is shared across multiple programming languages pretty much unchanged.

Dependency Injection Containers

You might have noticed in the example above, we are in fact removing a lot of dependencies, but where do they go? They don't magically dissapear. Truth is, another object must have that knowledge. The idea of DI containers is simple: Use an object who knows everything about other objects and their relations. This object is in charge of all your dependencies. Ask this guy for any *service* and he'll know how to instantiate it and give it to you (in the DIC world, they like calling

objects services).

Some languages — like Ruby, discourage the usage of DI containers. Others, like Java, embrace it. Before diving deeper, let's see an example of how a DI container might work. I've always liked Fabien Potencer (creator of PHP's Symfony framework) work, and not only his code, his articles onService Containers really helped me understand that concept.

In that series he explains a lot on DI and gives pretty good examples. Quoting from his writing, we can see a perfect example of a setter dependency injection:

```
$transport = new Zend_Mail_Transport_Smtp('smtp.gmail.com', array(
  'auth'     => 'login',
  'username' => 'foo',
  'password' => 'bar',
  'ssl'      => 'ssl',
  'port'     => 465,
));

$mailer = new Zend_Mail();
$mailer->setDefaultTransport($transport);

```

The same code using a DIC looks like this:

```
$container = new Container();
$mailer = $container->getMailer();

```

As you can see, the responsibility for getting services has been delegated to the container. If you are interested in DICs check out Fabien's series, they are really good and guide you though a basic implementation of a DIC in PHP.

The first thing I noticed when coming from PHP to Ruby was that they don't use DI containers. I learnt this is mostly because they embrace dynamic types over static ones. David Heinemeier Hansson, Rails creator and a **much** better Ruby developer than myself, explains in his blog post whyDependency injection is not a virtue in Ruby.

I'd say one of the points he makes the more emphasis on why a DIC is a bad idea in Ruby is that the code is already easy to test, there's no need to add complexity. Consider David's example, `publish!` has a dependency on the `Time` class, and a second one on the `now` method. His point is that it's not a big deal, it's still easy to test:

```
1  Time.stub(:now) { Time.new(2012, 12, 24) }
2  article.publish!
3  assert_equal 24, article.published_at.day
4
```

The ways in which Ruby can bend makes writing tests very easy. We know it's easy to test, but is it easy to change?

Writing code that's easy to change

We said before that the main objective of Object Oriented Design is writing code that is easy to test and **change**. DI containers make things easy to change, as every time your dependencies change, you'll only need to change your code in one place, the DIC. If you had hard-coded dependencies all over the place, you'd have to change your code in many places, which is never a good thing.

How can we write easy to change code without using a DIC? Well, one solution Ruby developers use is by wrapping your dependencies in methods:

```
1  class MyPORO
2    def greeter
3      @greeter ||= Greeter.new
4    end
5
6    def greet(name)
7      greeter.say_hi(name)
8    end
9  end
10
```

The code above still has a dependency on `Greeter`, but it's wrapped in the `greeter` method. If we change the name of the class, thus affecting the class

name dependency, we only have to change the class in one place. The same applies for the `say_hi` method, if we ever change the name or arguments it takes, we'd only have to change our code in one place: In the `greet` method.

This might not stop you from having to modify your code in many places when something changes, but it makes changes trivial.

Organizing Dependencies

Another thing you need to consider in Ruby is where you put your dependencies. Well organized dependencies allow you to make even easier changes, as less files will need to be changed.

A good rule of thumb is that dependencies should be specified in the object which **changes less**. Consider this scenario: You are writing an MVC application. Your controller depends on class A. Class A expects to be injected B as a dependency.

```
# Inside the controller
injected_service = B.new
another_service  = A.new injected_service

```

In this case, the controller is much likely to change than both services. It might be a better idea to change the order of the dependencies:

```
# Inside the controller
service = A.new

# Inside Class A
class A
  def service
    @service ||= B.new
  end

  # some more code...
end

```

This way the controller doesn't care about A's dependencies. Because A is less likely to change, one could say this code is more efficient.

Conclusion

Ruby gives the developer a bit more of responsibility, it's you who decide how to manage your dependencies. In exchange, you get simpler code and more freedom.

To briefly illustrate what I mean, consider Symfony and Rails. Both are awesome frameworks. In Symfony, you have to learn how to use the DIC and update your dependencies file whenever you want to add a new service. Adding a service certainly involves some ceremony, but fetching it will be easy. On the other hand, in Rails you simply add it to your Gemfile and start using the service directly. No learning curve, less ceremony, but you'll have to manage dependencies on your own.

As everything in life, both have their pros and cons. I'll just say, when in Rome, do as romans do. Ruby is quite mature by now and there's a reason Ruby developers do it this way. Try it and then judge by yourself.

Dependency injection is awesome. It's so powerful for such a simple concept! It's the base of many more-advanced Object Oriented Design principles and patterns. Understanding it will only do good things for you.

If you are interested in Ruby's OO design, I highly recommendPractical Object Oriented Design in Ruby. It really helped me think about OO code the Ruby way.