Ruby and dependency injection in a dynamic world

It's been many years I've been teaching Java and advocating dependency injection. It makes me design more loosely coupled modules/classes and generally leads to more extensible code.

But, while programming in Ruby and other dynamic languages, the different mindset always intrigued me. Why the reasons that made me love dependency injection in Java and C# don't seem to apply to dynamic languages? Why am I not using DI frameworks (or writing simple wiring code as I did many times before) in my Ruby projects?

I've been thinking about it for a long time and I don't believe I'm alone.

> **DI frameworks are unnecessary**. In more rigid environments, they have value. In agile environments like Ruby, not so much. The patterns themselves may still be applicable, but beware of falling into the trap of thinking you need a special tool for everything. Ruby is Play-Doh, remember! Let's keep it that way.
>
> — Jamis Buck

Even being a huge fan of DI frameworks in the Java/C# world (particularly the lightweights), I completely agree with Jamis Buck (read his post, it's very good!). This is a recurrent subject in talks with really smart programmers I know and I've tried to explain my point many times. I guess it's time to write it down.

The whole point about *Dependency Injection* is that it is one of the best ways to achieve *Inversion of Control* for object dependencies. Take the `Carpenter` *object*: his *responsibility* is to make and repair wooden structures.

```
class Carpenter {
    public void repair(WoodenStructure structure) {
        // ...
    }
    public WoodenStructure make(Specification spec) {
        // ...
    }
}
```

Because of his woodwork, the carpenter often needs a saw (*dependency*). Everytime the carpenter needs the saw, he may go after it (*objects who take care of their dependencies on their own*).

```
class Carpenter {
    public void repair(WoodenStructure structure) {
        PowerSource powerSource = findPowerSourceInsideRoom(this.myRoom);
        Saw saw = new ElectricalSaw(powerSource);
        // ok, now I can *finally* do my real job...
    }
}
```

The problem is that saws could be complicated to get, to assemble, or even to build. The saw itself may have some dependencies (`PowerSource`), which in turn may also have some dependencies… Everyone who needs a saw must know where to find it, and potencially how to assemble it. What if saw technology changes and some would rather to use hydraulic saws? Every object who needs saws must then be changed.

When some change requires you to mess with code everywhere, clearly **it's a smell**.

Have you also noticed that the carpenter has spent a lot of time doing stuff which isn't his actual job? Finding power sources and assembling saws aren't his responsibilities. His job is to repair wooden structures! (*Separation of Concerns*)

One of the possible solutions is the *Inversion of Control* principle. Instead of going after their dependencies, objects **receive them** somehow. Dependency Injection is the most common way:

```
class Carpenter {
    private Saw saw;
    public Carpenter(Saw saw) {
        this.saw = saw;
    }
    public void repair(WoodenStructure structure) {
        // I can focus on my job!
    }
}
```

Now, the code is more testable. In unit tests where you don't care about testing saws, but only about testing the carpenter, a mock of the saw can be provided (injected in) to the carpenter.

In the application code, you can also centralize and isolate code which decides what saw implementation to use. In other words, you now may be able to give the responsibility to do wiring to someone else (and only to him), so that objects can focus on their actual responsibilities (again *Separation of Concerns*). DI framework configuration is a good example of wiring centralized somewhere. If the saw implementation changes somehow you have just one place to modify (Factories help but don't actually solve the problem – I'll leave this discussion for another post).

```
class GuyWithResponsibilityToDoWiring {
    public Saw wireSaw() {
        PowerSource powerSource = getPowerSource();
        return new Saw(powerSource);
    }
    public PowerSource getPowerSource() { ... }
}
```

Ok. *Plain Old Java Dependency Injection* so far. But what about Ruby?

Sure you might do dependency injection in Ruby (and its dynamic friends) exactly the same way as we did in Java. But, it took some time for me to realize that there are other ways of doing Inversion of Control in Ruby.

That's why I never needed a DI framework.

First, let's use a more realistic example: repositories that need database connections:

```
class Repository {
    private Connection connection;
    public Repository(Connection connection) {
        this.connection = connection;
    }
    public Something find(Integer id) {
        return this.connection.execute("SELECT ...");
    }
}
```

Our problem is that many places need a database connection. Then, we inject the database connection as a dependency in everywhere it is needed. This way, we avoid replicating database-connection-retrieval code in such places and we may keep the wiring code centralized somewhere.

In Ruby, there is another way to isolate and centralize common behavior which is needed in many other places. **Modules**!

```
module ConnectionProvider
  def connection
    # open a database connection and return it
  end
end
```

This module can now be mixed, or injected in other classes, providing them its functionality (*wiring*). The "module injection" process is to some degree similar to what we did with dependency injection, because when a dependency is injected, it is providing some functionality to the class too.

Together with Ruby open classes, we may be able to centralize/isolate the injection of this module (perhaps even in the same file it is defined):

```
# connection_provider.rb
module ConnectionProvider
  def connection
    # open a database connection and return it
  end
end# reopening the class to mix the module in
class Repository
  include ConnectionProvider
end
```

The effect inside the `Repository` class is very similar to what we did with dependency injection before. Repositories are able to simply use database connections without worrying about how to get or to build them.

Now, the method that provides database connections exists because the `ConnectionProvider` module was mixed in this class. This is the same as if the dependency was injected (compare with the Java code for this Repository class):

```
# repository.rb
class Repository
  def find(id)
    connection.execute("SELECT ...")
  end
end
```

The code is also very testable. This is due to the fact that Ruby is a dynamic language and the connection method of Repository objects can be overridden anywhere. Particularly inside tests or specs, the connection method can be easily overridden to return a mock of the database connection.

I see it as a different way of doing Inversion of Control. Of course it has its pros and cons. I can tell that it's simpler (modules are a feature of the Ruby language), but it may give you headaches if you have a multithreaded application and must use different implementations of database connections in different places/threads, i.e. to inject different implementations of the dependency, depending on where and when it's being injected.

Opening classes and including modules inside them is a global operation and isn't thread safe (think of many threads trying to include different versions of the module inside the class). I'm not seeing this issue out there because most of Ruby applications aren't multithreaded and run on many processes instead; mainly because of Rails.

Regular dependency injection still might have its place in Ruby for these cases where plain modules aren't enough.

IMO, it's just much harder to justify. In my experience modules and metaprogramming are usually just enough.