Subject: (re-)introducing Dependency Injection
Posted by DannyBoyPoker on Sat, 06 Apr 2013 03:03:46 GMT
View Forum Message <> Reply to Message

I've seen this:
http://www.tonymarston.net/php-mysql/dependency-injection-is -evil.html

What does "Dependency" mean? etc. And: 'Using terms such as "dependency", "coupling" and "cohesion" can be confusing unless you identify exactly what they mean.'

Here I learn, that 'coupling' describes how modules interact. Lower coupling is better. And:
--
Tightly coupled systems tend to exhibit the following developmental characteristics, which are often seen as disadvantages:
A change in one module usually forces a ripple effect of changes in other modules. Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
A particular module might be harder to reuse and/or test because dependent modules must be included.
--

And, I learn of 'too many dunderheads out there', etc. 'And these people call me crazy!' And, I've been there

You quote an article, that has this line: 'Consider the below example we have a customer class which contains an address class object etc.'

And, your impatience w/this is pretty boundless. So, of this: 'Customer class is aware of the address class type. So if we add new address types like home address, office address it will lead to changes in the customer class also as customer class is exposed to the actual address implementation'.

You offer here, that: This is a meaningless accusation which describes a non-problem. It is a simple fact that the customer object must communicate with the address object in order to obtain the customer's address, so the customer object *MUST* know that the address object exists, and it *MUST* know which method to use to get the data it wants. It is simply not possible to write code which effectively says "get me some data from an unknown object using an unknown method" - unless, of course, you live in cloud cuckoo land.

I don't gather that the author of this article lives in cloud cuckoo land.

It may be a simple fact that etc., as you say, but this is not a meaningless accusation that describes a non-problem. I suppose that it actually is a non-problem if it can't be solved. Then, is it possible to write code which effectively says "get me some data from an unknown object using an unknown method"? Given that he lists various types of DI to implement what he wants (specifically, he mentions four), and how 'we can actually implement these', and here, he talks about using factory, and about a container, I'll leave this aside.

The point I want to get to, is, are you addressing, here, his two points? Which are, first, that main

classes aggregating other classes should not depend on the direct implementation of the aggregated classes. What then? Well, both the classes should, putatively, depend on abstraction. And, second, abstraction should not depend on details (rather, details should depend on abstraction). This is what he calls tight coupling, which is what he calls the problem.

And, you ask: 'all the "problems" identified in that article do not exist in my application, so if the problems do not exist then can you please explain to me in words on one syllable what benefits I would obtain by implementing this solution?'

You also cite an article:
http://ralphschindler.com/2011/05/18/learning-about-dependen cy-injection-and-php

And you say: 'I therefore consider the article to be a waste of space and unworthy of serious consideration, especially from an OO heretic such as me.'

Okay, so, um, okay, I'm not picking on you--this is a cry for help. You asked. I note, that this article says: 'This article is not about the intricacies and implementation details of DI containers and DI frameworks.'

I kind of gather a comportment from you on this, maybe it's possible to caricature, that you're saying: Decouple A from B? Why? A needs B!

To this general point, I have a general answer, that therein lies the problem. If I have a plug, and then I have an outlet, what is the reasonable response? However, software systems, so I am told, become unmaintainable because we let them grow on their own, without keeping a keen eye on how that growth is proceeding, until, one day, we realize that our software is a tangled mess of copied code and interwoven dependencies. And we have no one to blame but ourselves. Which I write, for my own benefit, I'm experienced, with taking action, without considering the reasons or implications.

More specifically, I see this: http://www.radicore.org/whatisradicore.php

8. Developers are spared the chore of designing and coding their hierarchy of menus as RADICORE comes supplied with a pre-built MENU system which allows menu pages to be constructed and maintained using a standard set of online screens.

9. Developers are spared the chore of designing and building a security system as RADICORE comes supplied with a pre-built RBAC system which allows access control lists to be constructed and maintained using a standard set of online screens.

However, I notice, that, in fact, the 'menu' subsystem, is really a menu and security system. Instead of there being a separate, global security system, security is handled by the menu subsystem. "Menu", in fact, maintains a list of the subsystems, and has a pointer to itself, as one of them. The users, and roles, tasks, also, are maintained, by the menu subsystem. Nothing runs w/out the menu subsystem, then. You can't not have a menu.

It occurs to me, that one might want to swap out the menu system, for another. This wouldn't be a simple as dropping in another menu subsystem, as a replacement. You can't delete the

menu subsystem. I might have supposed, that menu systems, and, more generally, structure & navigation,
would be areas in which one might want to mix and match various components. One might have a few menu
components, and display them both on a page (which is more the rule, than the exception).

One might, similarly, take this view of access & security. Furthermore, these are maybe two different things. Authentication would be a matter of site access, for example. One might describe password management, backup, site monitoring, as security.

How to accomplish this?

Subject: Re: (re-)introducing Dependency Injection
Posted by AJM on Sat, 06 Apr 2013 13:33:08 GMT
View Forum Message <> Reply to Message

Follow these links for definitions of dependency, cohesion and coupling.

One reason that I don't use DI is that I prefer to instantiate an object only when I need it and not before. Sometimes my CUSTOMER object may need to access the ADDRESS object, and sometimes it may not. Why instantiate an object that is not going to be used? In some cases there may be a choice of possible classes, and I don't know which one I want until I come to access it. There may also be arguments that I need to supply to the constructor which are only known at the last moment.

The main reason I don't use DI is that it requires too much effort for too little reward, and my code is easier to write without it. It is also easier to read and maintain, especially by others, and that is *FAR* more important than following some stupid principle devised by some DIC head (do you see the play on words there?)

Your point that "main classes aggregating other classes should not depend on the direct implementation of the aggregated classes" is just an opinion as far as I am concerned and not an absolute rule. Unless you can prove that breaking this rule has unwanted consequences then I shall continue breaking it until the cows come home.

Another point which falls into the same category is "both the classes should depend on abstractions". What on earth does that mean? A class is the result of performing an abstraction, so what does "abstraction" mean in that statement? What does the statement "abstractions should not depend on details, details should depend on abstractions" supposed to mean? This sounds like pure gobbledegook to me.

You statement about the menu system being intertwined with the security system is also pointless. There is a menu system and there is a security system as they each have their own separate sets of maintenance screens. They interact at runtime when a list of menu options is filtered to remove those to which the user does *not* have access, thus producing a list which the

user *can* access. They are separate, yet they work together to produce a result.

The idea that you may want to swap out one menu system for another is pointless. Radicore has a single built-in menu system which cannot be changed for another. If you don't want to use the Radicore menu system then don't use Radicore.

Finally, you ask the question "how to accomplish this?" How to accomplish what exactly? Your post is so full of numerous meandering statements it is difficult to see where there is a real question.

Subject: Re: (re-)introducing Dependency Injection
Posted by DannyBoyPoker on Sun, 07 Apr 2013 01:01:27 GMT
View Forum Message <> Reply to Message

First you say 'The idea that you may want to swap out one menu system for another is pointless.' And then you say 'Finally, you ask the question "how to accomplish this?" How to accomplish what exactly?' It's not really that I don't want to use the Radicore menu system, but rather, that I may want, generally, to pick and choose from various components that all work from a common database. Here's the pitch:

Each module builds upon the others to provide additional functionality. These intelligent modules can even detect their peers and may offer additional functionality depending on what other modules are present. I have spent countless hours developing modules that perform specific functions for a web site on their own but because they all interconnect, the more modules you have, the more functionality each may provide.

Now, in such a case, User Management, and security, could be separate modules. You say: 'Radicore has a single built-in menu system which cannot be changed for another. If you don't want to use the Radicore menu system then don't use Radicore.' And, to this, I ask, what is 'core', in Radicore? I'm fine with it, if the basic functionality provided is fairly substantial, that's a good thing. And it is. But building on top of it, would mean, could mean various things, expanding the user interactive stuff, or the general information stuff, or business management stuff. eCommerce stuff.

And, the point is to discuss the problem a dependency. And, the the Dependency Injection pattern is only applicable in software systems consisting of separated components. The idea is that outer components inject dependencies into smaller components. Along with definitions of dependency, cohesion and coupling, we might try to find agreement, on what are the attributes to make something a component. I don't think it's true, that this doesn't matter to you, you are offering, in Radicore, components that in the general case have a limited scope of interest. And, these component instances cannot access properties of other components, unless you're providing a way to do so. And, your components are reusable and might be used in different systems in a way that the using system does not have to change code on the imported component to make it work.

If I say something like that in Radicore, maybe there are not enough components, then this only to ask for more Radicore, in Radicore. The idea is to not make it hard to write reusable software and test these in isolation, right? You give reasons why you don't use DI and that is fine (actually you do, to a great extent, I would have said, when I actually look at the code), as the point is not that couldn't possibly be a cleaner solution for dissolving dependencies. If you identify your dependencies, then refactor to components, as you have, largely, then you can proceed to keep your components dumb, and the DI pattern is just something that supposedly makes it easy.

Subject: Re: (re-)introducing Dependency Injection
Posted by AJM on Sun, 07 Apr 2013 10:15:22 GMT
View Forum Message <> Reply to Message

It is quite clear that you do not understand what RADICORE is. It is *NOT* a collection of libraries from which you can pick and choose, it is a fully-fledged framework which you either use in its entirety or you do not use at all. RADICORE has a single menu system and a single security system which are intertwined. You can customise the way it looks by replacing the CSS files but you cannot change the way it works. You can swap the password authentication by using either a RADIUS or an LDAP server, but you cannot change how you allow/disallow a user's access to a specific task.

There is an in-built audit logging system which you can turn off should you choose, but you cannot swap it for another, nor can you extract it to use without the rest of RADICORE.

There is an in-built workflow system which you can turn on should you choose, but you cannot swap it for another, nor can you extract it to use without the rest of RADICORE.

There is an in-built data dictionary and collection of transaction patterns which you use to generate your application components, but you cannot swap it for another, nor can you extract it to use without the rest of RADICORE.

You do not use RADICORE on its own, you use it to build your own back-end application. There are several prototype applications included in the download which you can play with and examine. Your application will have numerous database tables to maintain and numerous user transactions which implement your business logic. You can build the basic CRUD screens for each of your database tables very quickly, then all you do is add the code to process your business rules.

Although RADICORE's presentation layer is not suitable for customer-facing front-end websites (it was designed specifically for members of staff who administer the site) it's implementation of the 3 Tier Architecture means that you can easily create an alternative presentation layer using whatever tools you like, but which shares the business and data access layers already within RADICORE.

I have used RADICORE to build an order processing application called TRANSIX which is currently used by businesses as varied as custom jewellery, baby clothes and photographic prints. They all share the same back-end (although they have their own private instances of the databases) but they each have their own unique front-end websites. Because they require nothing more than a different presentation layer they are quick and easy to build.

When you build your own application using RADICORE you have instant access to all the features within RADICORE, and you are free to write your own business logic and use whatever external libraries that take your fancy.

I don't use Dependency Injection within RADICORE because it has enormous costs and zero benefits. It has benefits in only a limited set of circumstances, and those circumstances do not exist within RADICORE. I will *NEVER* refactor RADICORE to use DI, so telling me that, in your opinion, DI is a good idea is just a waste of time.
