Are dependency injection frameworks worth the extra indirection layer?

Asked 15 years, 10 months ago Modified 13 years, 3 months ago

Viewed 3k times

11

Do you find that dependency injection frameworks make code more difficult to follow? Does the indirection outweigh the benefit?

dependency-injection

Share

Improve this question

Follow

asked Feb 16, 2009 at 22:40

BC.

24.8k ● 12 ● 48 ● 63

Other students of DI may be interested to watch the Google tech talk on the subject of DI and testability: misko.hevery.com/2008/11/11/… – BC. Feb 16, 2009 at 23:04

9 Answers

Sorted by: Highest score (default)

Yes, your code becomes much more decoupled and much more testable. This in particular becomes handy

14

when you have lots of tests and each test requires a heavy object such as database layers.

If you use dependency injection, you can simply create so called 'mock' objects or stubs and use those to let your tests run more quickly and have less side effects (database state).

It is true that you cannot see directly which implementation is used by looking at the code. You will see a reference to the interface. A good IDE might have functionallity to view all implementations for a particular interface, so use that to your advantage.

Share Improve this answer Follow

edited Sep 12, 2011 at 5:01
ocodo
30.2k ● 19 ● 110 ● 125

answered Feb 16, 2009 at 22:44
TomHastjarjanto
5,408 ● 1 ● 30 ● 41

3 One other thing to consider is your language. Java for instance, it's totally worth it. On the other hand, for ruby, not so much – pope Feb 16, 2009 at 22:47

My bad! Given that this site is so crowded with C# questions, I sometimes forget that there are users of other languages :) – TomHastjarjanto Feb 16, 2009 at 22:54

1 You don't need to use interfaces to use dependency injection and gain testability. I would argue that if you're almost positive you'll never have two implementations of such an interface, it's a waste of time to make the interface at all. – CorayThan Aug 26, 2013 at 21:41

This answer is all about interfaces, not DI. All benefits mentioned are achieved by coding to interfaces contracts and those mocks frameworks use the interfaces, not the DI. – user6490459 Jun 2, 2018 at 10:40

Actually your code does not become more decoupled by using DI. A dependency on a specific interface is still a dependency and if no implementation is injected the program will crash at runtime. A decoupled behaviour is when two parts of your software can work together but are not required to be present at all. With DI, you do get the benbefits of easier testability thou. – Christian Flodihn Nov 25, 2019 at 10:03

6

I have found that a custom static HashTable factory decouples my dependencies fine and meets my needs. I have tried a few times to use a full-blow IOC container and each time I am taken aback by the learning curve (and all the config) that the rest of my team has to put up with ... and all that for little or no added features over my vanilla.

So, I guess the bigger problem with dependency injection is not in the pattern itself but with the Fad it currently generates in the developer community. It sounds cool, so there's pressure to use it, even where the engineering is not driven by a commensurate requirement.

We tend to take a big gun to a mosquito because the gun looks cool.

P

Share Improve this answer

Follow

answered Feb 17, 2009 at 0:33

Pita.O

1,837 ● 1 ● 20 ● 27

5

For non-trivial "enterprisey" apps, yes it's worth it. Before DI frameworks, every shop implemented its own fancy *"ServiceLocator"* class in some internal library that their other projects used. So you had calls to that thing littered throughout the codebase.

Those calls represented the objects' need to discover/configure their own dependencies. The DI framework eliminates all that code, so your objects become simpler and therefore, easier to test.

Now, it follows that if you don't have a lot of variability in your objects' dependencies, the value of the indirection (the centralized configuration) is less for you.

For more details contrasting DI to ServiceLocator, see Fowler's [Inversion of Control Containers and the Dependency Injection pattern]

Share Improve this answer Follow

answered Feb 16, 2009 at 22:58

jcrossley3
11.8k ● 4 ● 32 ● 33

1 Great answer and it really address the DI or not DI subject. Not like the accepted answer that confuses DI with programming to interfaces. – user6490459 Jun 2, 2018 at 10:44

DI benefits are difficult to grasp at first sight. Testability is the most obvious. I will try to coin a very short example to

3

illustrate these benefits. Imagine your application is using certain database provider. If you keep all your code see only the DB provider interface, you will easily switch from one implementation to another. Now, if you make enterprise applications, not depending on particular DB provider can be extremely desirable, like when client has already purchased DB license. This is actually benefit of depending on interface and not implementation. So far so good. Now, I need somehow to get hold of my concrete object. My concrete object can be an instance of concrete provider only. How do I do it?

1. Use new to create object.
   I will have to recompile the project in order to distribute it. A huge downside: will have to test, deploy, maintain new code branch etc.
2. Use Factory.
   My code will be sprinkled with code that gets hold of factory and obtains the instance. Also, I will have to return a generic instance and cast it to the type of object I expect. If not, I will have to change the Factory interface each time I add new object created by factory.
3. Use Service Locator.
   Similar as 2. Only, depending on the Service Locator might not always be around, like Java JNDI.

DI externalizes this and takes approach that injection is separate concern. Your object should be concerned with the domain and not with finding appropriate collaborators.

Share Improve this answer Follow

edited Feb 20, 2009 at 19:42

answered Feb 20, 2009 at 17:06

Dan
11.2k ● 21 ● 86 ● 121

Great example. Sometimes loose coupling is a requirement.
– BC. Feb 20, 2009 at 19:13

2

I initially liked the idea of a dependency injection framework, but other than supporting unit testing I am still unconvinced of the benefits of using one. It means one more framework/API/technique for someone who takes over my project to learn and can actually be more verbose in some cases. You can find an excellent back and forth on the pros and cons on these two competing blog entries

For DI Framework

Against DI Framework

The bottom line comes down to does it *really* reduce coupling and increase cohesion or does it just push the problems under the surface. On my project right now I don't see much need for it, even to support unit testing. If you were going to pick one up for C# though, I would highly recommend Ninject. It's lightweight, simple and completely fluently configured ... no XML! Sweeet :)

Share Improve this answer Follow

answered Feb 17, 2009 at 1:09

Jeffrey Cameron
10.3k ● 11 ● 47 ● 77

2

You can design your classes to have their dependencies injected (via a constructor or properties) without having to use a dependency injection framework. Just instantiate the dependency yourself and pass it in or grab it from a service locator or registry class that you toss together. But instead of having the class itself resolve its dependencies by calling the service locator, have the class that's instantiating the class resolve the dependencies for it. You maintain a testable class design without the overhead and complexity of another library and framework.

I for one know that whenever I've tried using a DI framework, that's essentially all that I really ended up using it for. I've also seen cases where people wrap their DI container in a static IoC class to build up objects way down in the hierarchy, and in my mind this sort of defeats the purpose; isn't it just right back to being a service locator at that point? I guess I don't quite get the difference in practical usage. I say shenanigans, you're using reflection and taking a big start-up hit to do the same thing.

You can't eliminate the dependencies, but you sure can obfuscate the heck out of them in an XML configuration file. At some point *something* is going to call `new`. Are you really going to swap out an implementation of an interface without recompiling or retesting your application anyway? If not, keep it simple. It's nice to click "Find Definition" and see something actually getting instantiated in a service locator class with explicit singleton or `ThreadStatic` or some other behavior.

Just my two cents--I'm still pretty new to DI frameworks, but that's my current train of thought: inversion of control is useful, but the actual frameworks themselves probably only for very large projects.

Share Improve this answer Follow

answered Feb 17, 2009 at 1:11

Nicholas Piasecki
25.6k ● 5 ● 82 ● 92

1

In my opinion dependency- injection frameworks don't make the code more complicated to read. The fact that you see only references to interfaces and not references to the concrete implementations is good - because you don't have to care about how the concrete implementation works.

If navigating through the source code is too difficult, get Resharper and everything is fine.

Share Improve this answer Follow

answered Feb 16, 2009 at 22:46

ollifant
8,606 ● 10 ● 37 ● 45

1

Dependency injection is a fantastic idea; although a big part of the *win* is in the fact that you can code purely to interfaces. It forces (or encourages) you to separate concerns in your program into multiple, collaborating service-type instances which do not know *how* each other are implemented. This makes it much less likely that you'll introduce unwanted and unnecesary dependencies between classes.

However, Inversion of Control is not the only way to go here. A colleague has achieved this via writing his own implementation of `JNDI`. It took quite a while to win me around but it is fantastic and has a fraction of the configuration of a typical Spring project.

Share Improve this answer Follow

answered Feb 16, 2009 at 23:01

oxbow_lakes
134k ● 56 ● 320 ● 450

Is it the DI Framework that provides the advantage here or coding to interfaces? I think using interfaces is a no-brainer, but whether one needs a DI Framework in support I'm still a little unsure of – Jeffrey Cameron Feb 17, 2009 at 1:11

Well, it's coding to interfaces. But DI (in Java) makes coding to interfaces much less hassle. I think that the difference between dependency injection and dependency lookup is really not that great, however. I have seen the light! – oxbow_lakes Feb 17, 2009 at 8:02

I think the benefit depends on how you use it.

0

For example, on my current project, we use dep inj to inject different items depending on the app configuration, specifically unit testing. A good example would be having a production system use a real "login" implementation, but having the unit tests use a mock "login" that always returns true/valid for a knows login, but no others.

Share Improve this answer Follow

answered Feb 16, 2009 at 22:44

CodingWithSpike
43.7k ● 18 ● 105 ● 139