What are the downsides to using dependency injection? [closed]

Asked 14 years, 9 months ago Modified 1 year, 11 months ago

Viewed 98k times

353

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 13 years ago.

I'm trying to introduce dependency injection (DI) as a pattern here at work and one of our lead developers would like to know: What, if any, are the **downsides** to using the dependency injection pattern?

Note I'm looking here for an, if possible, exhaustive list, not a subjective discussion on the topic.

Clarification: I'm talking about the dependency injection *pattern* (see this article by Martin Fowler), *not* a specific framework, whether XML-based (such as Spring) or code-based (such as Guice), or "self-rolled".

Some great further discussion / ranting / debate is going on the [Reddit's subreddit */r/programming*](#).

design-patterns dependency-injection

Share Follow

edited Jan 15, 2023 at 23:39

Peter Mortensen
31.6k ● 22 ● 109 ● 133

asked Mar 9, 2010 at 8:24

Epaga
39k ● 59 ● 162 ● 246

It might be a good idea to specify whether we're supposed to be discussing DI itself or a specific type of tool that supports it (XML-based or not). – Jesse Millikan Mar 9, 2010 at 8:37

---

5 the difference is that I'm NOT looking for answers that only apply to specific frameworks, such as "XML sucks". :) I'm looking for answers that apply to the concept of DI as outlined by Fowler here: [martinfowler.com/articles/injection.html](#) – Epaga Mar 9, 2010 at 12:08

---

3 I see no downsides at all to DI in general (constructor, setter or method). It decouples and simplifies with no overhead. – Kissaki Feb 10, 2011 at 10:16

---

2 @kissaki well apart from the setter inject stuff. Best to make objects that are usable at construction. If you don't believe me, just try adding another collaborator to an object with 'setter' injection, you will get a NPE for sure.... – time4tea Feb 10, 2011 at 17:05

2 In Defense of Service Locator – ZhongYu Jan 6, 2016 at 13:15

19 Answers

Sorted by: Highest score (default)

229

A couple of points:

- DI increases complexity, usually by increasing the number of classes since responsibilities are separated more, which is not always beneficial
- Your code will be (somewhat) coupled to the dependency injection framework you use (or more generally how you decide to implement the DI pattern)
- DI containers or approaches that perform type resolving generally incur a slight runtime penalty (very negligible, but it's there)

Generally, the benefit of decoupling makes each task simpler to read and understand, but increases the complexity of orchestrating the more complex tasks.

Share Follow

edited Feb 10, 2011 at 19:08

Joe
47.6k ● 35 ● 163 ● 257

answered Mar 9, 2010 at 8:34

Håvard S
23.9k ● 8 ● 63 ● 73

79 - Separation of classes reduces complexity. Many classes don't make an application complex. - Your should only have a dependency at to your DI framework at the application-root. – Robert Mar 9, 2010 at 9:04

110 We're humans; our short term memory is limited and cannot handle many <xxx> simultaneously. It's the same for classes as it is for methods, functions, files, or any construct you use to develop your program. – Håvard S Mar 9, 2010 at 9:20

54 @Havard S: Yes, but 5 really complex classes is not simpler than 15 simple ones. The way to decrease complexity is to reduce the working set required by the programmer working on the code - complex, highly interdependent classes do not accomplish that. – kyoryu Mar 9, 2010 at 10:01

39 @kyoryu I think we all agree on that. Keep in mind that *complexity is not the same as coupling*. Decreasing coupling can increase complexity. I'm really not saying that DI is a bad thing because it increases the number of classes, I'm highlighting the potential downsides associated with it. :) – Håvard S Mar 9, 2010 at 10:31

114 +1 for "DI increases complexity" That's true in DI and beyond. (Almost) any time we increase flexibility, we're going to increase complexity. It's all about balance, which begins with knowing the upsides and downsides. When people say, 'there are no downsides,' it's a sure indicator that they haven't fully understood the thing yet. – Don Branson Mar 9, 2010 at 18:37

The same basic problem you often get with object oriented programming, style rules and just about everything else. It's possible - very common, in fact - to

203

do too much abstraction, and to add too much indirection, and to generally apply good techniques excessively and in the wrong places.

Every pattern or other construct you apply brings complexity. Abstraction and indirection scatter information around, sometimes moving irrelevant detail out of the way, but equally sometimes making it harder to understand exactly what's happening. Every rule you apply brings inflexibility, ruling out options that might just be the best approach.

The point is to write code that does the job and is robust, readable and maintainable. You are a software developer - not an ivory tower builder.

Relevant Links

The Inner-Platform Effect

Don’t Let Architecture Astronauts Scare You

Probably the simplest form of dependency injection (don't laugh) is a parameter. The dependent code is dependent on data, and that data is injected in by the means of passing the parameter.

Yes, it's silly and it doesn't address the object-oriented point of dependency injection, but a functional programmer will tell you that (if you have first class functions) this is the only kind of dependency injection you need. The point here is to take a trivial example, and show the potential problems.

Let’s take this simple traditional function. C++ syntax isn't significant here, but I have to spell it somehow...

```
void Say_Hello_World ()
{
  std::cout << "Hello World" << std::endl;
}
```

I have a dependency I want to extract out and inject - the text "Hello World". Easy enough...

```
void Say_Something (const char *p_text)
{
  std::cout << p_text << std::endl;
}
```

How is that more inflexible than the original? Well, what if I decide that the output should be Unicode. I probably want to switch from *std::cout* to *std::wcout*. But that means my strings then have to be of *wchar_*t, not of *char*. Either every caller has to be changed, or (more reasonably), the old implementation gets replaced with an adaptor that translates the string and calls the new implementation.

That's maintenance work right there that wouldn't be needed if we'd kept the original.

And if it seems trivial, take a look at this real-world function from the Win32 API...

[*CreateWindowExA function (winuser.h)*]

That's 12 "dependencies" to deal with. For example, if screen resolutions get really huge, maybe we'll need 64-bit co-ordinate values - and another version of CreateWindowEx. And yes, there's already an older version still hanging around, that presumably gets mapped to the newer version behind the scenes...

[*CreateWindowA macro (winuser.h)*]

Those "dependencies" aren't just a problem for the original developer - everyone who uses that interface has to look up what the dependencies are, how they are specified, and what they mean, and work out what to do for their application. This is where the words "sensible defaults" can make life much simpler.

Object-oriented dependency injection isn't any different in principle. Writing a class is an overhead, both in source-code text and in developer time, and if that class is written to supply dependencies according to some dependent objects specifications, then the dependent object is locked into supporting that interface, even if there's a need to replace the implementation of that object.

None of this should be read as claiming that dependency injection is bad - far from it. But any good technique can be applied excessively and in the wrong place. Just as not every string needs to be extracted out and turned into a parameter, not every low-level behaviour needs to be extracted out from high-level objects and turned into an injectable dependency.

Share Follow

edited Jan 15, 2023 at 23:43
Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Mar 9, 2010 at 10:22
user180247

3 Dependency injection does not have any of those drawbacks. It only changes the way your pass dependencies to objects, which does not add complexity nor inflexibility. Quite the contrary. – Kissaki Feb 10, 2011 at 10:13

28 @Kissaki - yes, it changes the way you pass your dependencies. And you have to write code to handle that passing of dependencies. And before doing that, you have to work out which dependencies to pass, define them in a way that makes sense to someone who didn't code the dependent code, etc. You can avoid run-time impact (e.g. using policy parameters to templates in C++), but that's still code you have to write and maintain. If it's justified, it's a very small price for a big reward, but if you pretend there's no price to pay that just means you'll be paying that price when there's no gain. – user180247 Feb 10, 2011 at 17:30

6 @Kissaki - as for inflexibility, once you have specified what your dependencies are, you are locked into that - other people have written dependencies to inject according to that spec. If you need a new implementation for the dependent code which needs slightly different dependencies - tough, you're locked in. Time to start writing some adaptors for those dependencies instead (and a bit more overhead and another layer of abstraction). – user180247 Feb 10, 2011 at 17:59

2 @ingredient_15939 - yes, this is a simplified toy example. If you were really doing this, you'd just use overloading rather than an adapter as the cost of duplicating the code is less than the cost of the adapter. But note that duplicating code is generally a bad thing, and using adapters to avoid duplicating code is another good technique (that can sometimes be used too much and in the wrong places). – user180247 Mar 24, 2014 at 2:45

2 @FrancescoPasa - The point isn't to advocate another form of dependency management - the point is to not have dependency management issues just because "DI is awesome!!!", but only if needed. If DI is provided even though it's unnecessary, it's a failure of encapulation - what should be a fully encapsulated implementation detail has been exposed to the caller in the form of that dependency, so the caller can be broken if it changes. A fully encapsulated implementation detail cannot cause that breakage by definition. The trade-off - encapsulated internals also can't be controlled by the caller. – user180247 Jul 25, 2018 at 7:14

82

Here's my own initial reaction: Basically the same downsides of any pattern.

- it takes time to learn
- if misunderstood it can lead to more harm than good
- if taken to an extreme it can be more work than would justify the benefit

Share Follow

edited Aug 19, 2012 at 18:03

Chris Laplante
29.7k ● 18 ● 109 ● 136

answered Mar 9, 2010 at 8:26

Epaga
39k ● 59 ● 162 ● 246

2 Why it takes time to learn? I thought it makes things simple compared with ejb. – fastcodejava Mar 9, 2010 at 8:31

9 This seems hand-wavy considering you don't want a subjective discussion. I suggest building (collectively, if necessary) a realistic example for examination. Building a sample *without* DI would be a good starting point. Then we could challenge it w.r.t. to requirement changes and examine the impact. (This is extremely convenient for me to suggest, by the way, as I'm about to go to bed.) – Jesse Millikan Mar 9, 2010 at 8:41

5 This really needs some examples to back it up, and having taught dozens of people DI I can tell you that it really is a very simple concept to learn and to start putting into practice. – chillitom Mar 9, 2010 at 9:18

4 I don't really consider DI a pattern. It (coupled with IoC) are really more of a programming model - if you fully follow them, your code looks a lot more Actor-like than "typical" pseudo-procedural OO. – kyoryu Mar 9, 2010 at 10:04

2 +1 I'm using Symfony 2, the DI is all over the user code, is exausting just using it. – MGP Feb 22, 2013 at 15:23

48

The biggest "downside" to Inversion of Control (not quite DI, but close enough) is that it tends to remove having a single point to look at an overview of an algorithm. That's basically what happens when you have decoupled code, though - the ability to look in one place is an artifact of tight coupling.

Share Follow

answered Mar 9, 2010 at 9:13

kyoryu
13k ● 2 ● 30 ● 33

3 But surely that "downside" is because of the nature of the problem we are solving, decoupling it so we can easily change the implementation means there is no one place to look at, and what relevance is it being able to look at it? The only case I can think is in debugging and the debugging environment should be able to step into the implementation. – vickirk Mar 11, 2010 at 12:55

3 This is why the word "downside" was in quotes :) Loose coupling and strong encapsulation preclude "one place to see everything," kind of by definition. See my comments on Havard S's response, if you get the feeling that I'm against DI/IoC. – kyoryu Mar 12, 2010 at 2:37

1 I agree (I even voted you up). I was just making the point. – vickirk Mar 12, 2010 at 8:20

1 @vickirk: The downside I guess is that it is hard for a human to comprehend. Decoupling things increases complexity in the sense that it is harder for humans to understand, and take longer to comprehend fully. – Bjarke Freund-Hansen Dec 30, 2010 at 9:42

2 On the contrary, one advantage of DI is that you can look at the constructor of an individual class, and work out instantly what classes it depends on (i.e. what classes needs to do its work). This is much harder for other code, as the code can create classes willy-nilly. – Contango Sep 26, 2013 at 13:38

I have been using Guice (Java DI framework) extensively for the past 6 months. While overall I think it is great

46

(especially from a testing perspective), there are certain downsides. Most notably:

- **Code can become harder to understand.** Dependency injection can be used in very... creative... ways. For example I just came across some code that used a custom annotation to inject a certain IOStreams (eg: @Server1Stream, @Server2Stream). While this does work, and I'll admit has a certain elegance, it makes understanding the Guice injections a prerequisite to understanding the code.
- **Higher learning curve when learning project.** This is related to point 1. In order to understand how a project that uses dependency injection works, you need to understand both the dependency injection pattern and the specific framework. When I started at my current job I spent quite a few confused hours groking what Guice was doing behind the scenes.
- **Constructors become large.** Although this can be largely resolved with a default constructor or a factory.
- **Errors can be obfuscated.** My most recent example of this was I had a collision on 2 flag names. Guice swallowed the error silently and one of my flags wasn't initialized.
- **Errors are pushed to run-time.** If you configure your Guice module incorrectly (circular reference, bad binding, ...) most of the errors are not uncovered

during compile-time. Instead, the errors are exposed when the program is actually run.

Now that I've complained. Let me say that I will continue to (willingly) use Guice in my current project and most likely my next. Dependency injection is a great and incredibly powerful pattern. But it definitely can be confusing and you will almost certainly spend some time cursing at whatever dependency injection framework you choose.

Also, I agree with other posters that dependency injection can be overused.

Share Follow

answered Feb 14, 2011 at 2:25

Andy Peck
461 ● 4 ● 2

20 I don't understand why people don't make a bigger deal of "Errors are pushed to run-time" for me that's a deal breaker, static typing and compile time errors are the greatest gift given to developers, I wouldn't throw them away for anything – Richard Tingle Jul 23, 2015 at 14:06

2 @RichardTingle , IMO during the app start up DI modules will get initialized first so any misconfiguration in module will be visible as soon as the app starts rather than after some days or time. It is also possible to load modules incrementally but if we stick to the spirit of DI by limiting the module loading before initializing the application logic, we can successfully isolate the bad bindings to the start of the app. But if we configure it as service locator anti pattern then those bad bindings will surely be a surprise. – Kavin Eswaramoorthy Dec 22, 2016 at 13:24

@RichardTingle I understand that it will never be similar to the safety net provided by the compiler but DI as a tool used correctly as told in the box then those runtime errors are limited to the app initialization. Then we can look the app initialization as a sort of compilation phase for DI modules. In my experience most of the time if the app starts then there wont be bad bindings or incorrect references in it. PS - I've been using NInject with C# – Kavin Eswaramoorthy Dec 22, 2016 at 13:29

@RichardTingle - I agree with you, but it's a trade-off in order to obtain loosely coupled, thus testable code. And as k4vin said, missing dependencies are found at initialization, and using interfaces will still help with compile time errors.
– Andrei Epure Feb 28, 2017 at 11:52

1 I would add "Hard to keep code clean" in your list. You never know if you can delete a registration before you extensively test the code without it. – fernacolo Sep 21, 2018 at 0:26

42

I don't think such a list exists, however try to read those articles:

- [DI can obscure the code (if you're not working with a good IDE)]
- [Misusing IoC can lead to bad code according to Uncle Bob.]
- [Need to look out for over-engineering and creating unnecessary versatility.]

Share Follow

edited Jul 25, 2011 at 13:57

vgru
51.1k ● 17 ● 125 ● 209

answered Mar 9, 2010 at 8:47

Gabriel Ščerbák
18.6k ● 9 ● 42 ● 53

7 I am curious, when someone asks for wonsides, why answers in a spirit "there aren't any" get upvoted and those which contain some information related to the question not? – Gabriel Ščerbák Mar 9, 2010 at 9:21

13 -1 because I'm not looking for a link collection, I'm looking for actual answers: how about summarizing each article's points? Then I'd upvote instead. Note that the articles themselves are quite interesting though it seems the first two are actually debunking negatives of DI? – Epaga Mar 9, 2010 at 9:51

5 I wonder if the most upvoted answer got downvoted as well, because it really doesn't answer the question at all imho:) Sure I know what you asked and what I answered, I am not asking for upvotes, I am just confused why my answer isn't helping whilst apparently saying DI downside is that it is too cool is helping much. – Gabriel Ščerbák Mar 9, 2010 at 10:06

30

Code without any DI runs the well-known risk of getting tangled up into Spaghetti code - some symptoms are that the classes and methods are too large, do too much and cannot be easy changed, broken down, refactored, or tested.

Code with DI used a lot can be Ravioli code where each small class is like an individual ravioli nugget - it does one small thing and the single responsibility principle is adhered to, which is good. But looking at classes on their own it's hard to see what the system as a whole does, since this depends on how all these many small parts fit together, which is hard to see. It just looks like a big pile of small things.

By avoiding the spaghetti complexity of big bits of coupled code within a large class, you run the risk of another kind of complexity, where there are lots of simple little classes and the interactions between them are complex.

I don't think that this is a fatal downside - DI is still very much worthwhile. Some degree of ravioli style with small classes that do just one thing is probably good. Even in excess, I don't think that it is bad as spaghetti code. But being aware that it can be taken too far is the first step to avoiding it. Follow the links for discussion of how to avoid it.

Share Follow

edited Feb 11, 2011 at 21:55

community wiki
4 revs
Anthony

1 Yes, I like that term "ravioli code"; I express it more long-windedly when I talk about the downside of DI. Still, I can't imagine developing any real framework or application in Java without DI. – Howard M. Lewis Ship Feb 11, 2011 at 18:57

The trick with ravioli code is understanding that, unlike in "spaghetti code", the main code flow doesn't exist in a function - it exists in whatever is wiring together the objects. That's a thing to *learn*, but once you're accustomed to looking for it, it gets a lot easier. – kyoryu Jun 21, 2022 at 20:27

14

The illusion that you've decoupled your code just by implementing dependency injection without *actually* decoupling it. I think that's the most dangerous thing about DI.

Share Follow

edited Jan 15, 2023 at 23:36
Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Feb 10, 2011 at 12:37
Joey Guerra
596 ● 6 ● 12

13

If you have a home-grown solution, the dependencies are right in your face in the constructor. Or maybe as method parameters which again is not too hard to spot. Though framework managed dependencies, if taken to the extremes, can begin to appear like magic.

However, having too many dependencies in too many classes is a clear sign that your class structure is screwed up. So in a way dependency injection (home-grown or framework managed) can help bring glaring design issues out that might otherwise be hidden lurking in the dark.

To illustrate the second point better, here's an excerpt from this article (original source) which I whole heartedly believe is the fundamental problem in building any system, not just computer systems.

> Suppose you want to design a college campus. You must delegate some of the design to the students and professors, otherwise the Physics building won't work well for the physics people. No architect knows enough about about what physics people need to do it all themselves. But you can't delegate the design of every room to its occupants, because then you'll get a giant pile of rubble.
>
> How can you distribute responsibility for design through all levels of a large hierarchy, while still maintaining consistency and harmony of overall design? This is the architectural design problem Alexander is trying to solve, but it's also a fundamental problem of computer systems development.

Does DI solve this problem? **No**. But it does help you see clearly if you're trying to delegate the responsibility of designing every room to its occupants.

Share Follow

edited Jan 15, 2023 at 23:23
Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Mar 9, 2010 at 8:36
Anurag
142k ● 37 ● 222 ● 261

13

One thing that makes me squirm a little with DI is the assumption that all injected objects are *cheap to instantiate* and *produce no side effects* -OR- the dependency is used so frequently that it outweighs any associated instantiation cost.

Where this is can be significant is when a dependency is not *frequently* used within a consuming class; such as something like an `IExceptionLogHandlerService`. Obviously, a service like this is invoked (hopefully :)) rarely within the class - presumably only on exceptions needing to be logged; yet the *canonical constructor-injection pattern*...

```
Public Class MyClass
    Private ReadOnly mExLogHandlerService As IExceptio
    
    Public Sub New(exLogHandlerService As IExceptionLo
        Me.mExLogHandlerService = exLogHandlerService
    End Sub
    
    ' ...
End Class
```

...requires that a "live" instance of this service be provided, damned the cost/side-effects required to get it there. Not that it likely would, but what if constructing this dependency instance involved a service/database hit, or configuration file look-ups, or locked a resource until disposed of? If this service was instead constructed as-needed, service-located, or factory-generated (all having problems their own), then you would be taking the construction cost only when necessary.

Now, it is a generally accepted software design principle that constructing an object *is* cheap and *doesn't* produce side-effects. And while that's a nice notion, it isn't always the case. Using typical constructor-injection however basically demands that this is the case. Meaning when you create an implementation of a dependency, you have to design it with DI in mind. Maybe you would have made object-construction more costly to obtain benefits elsewhere, but if this implementation is going to be injected, it will likely force you to reconsider that design.

By the way, certain techniques can mitigate this exact issue by allowing lazy-loading of injected dependencies, e.g., providing a class a `Lazy<IService>` instance as the dependency. This would change your dependent objects' constructors and make then even more cognizant of implementation details such as object construction expense, which is arguably not desirable either.

Share Follow

edited Jan 15, 2023 at 23:54

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 6, 2011 at 1:25

ckittel

6,646 ● 4 ● 42 ● 71

1 I understand what you're saying, but I don't think its fair to say "when you create an implementation of a dependency, you have to design it with DI in mind" - I think a more accurate statement would be "DI works best when used with classes that are implemented with best practices regarding instantiation cost and a lack of side effects or failure modes during construction". Worst case, you could always inject a lazy proxy implementation that defers allocation of the real object until first use. – Jolly Roger Nov 7, 2011 at 20:28

1 Any modern IoC container will allow you to specify an object's lifetime for a specific abstract type/interface (always unique, singleton, http scoped, etc). You can also provide a factory method/delegate to instantiate it lazily using Func<T> or Lazy<T>. – Dmitry S. Dec 14, 2011 at 6:47

Spot on. Instantiating dependencies unnecessarily costs memory. I normally utilize "lazy-loading" with getters that only instantiate a class when called upon. You can provide default a constructor without parameters as well as subsequent constructors that accept instantiated dependencies. In the first case, a class implementing IErrorLogger, for example, is only instantiated with `this.errorLogger.WriteError(ex)` when an error occurs in a try/catch statement. – John Bonfardeci Feb 23, 2018 at 18:07

This is more of a nitpick. But one of the downsides of dependency injection is that it makes it a little harder for

12

development tools to reason about and navigate code.

Specifically, if you Control-Click/Command-Click on a method invocation in code, it'll take you to the method declaration on an interface instead of the concrete implementation.

This is really more of a downside of loosely coupled code (code that's designed by interface), and applies even if you don't use dependency injection (i.e., even if you simply use factories). But the advent of dependency injection is what really encouraged loosely coupled code to the masses, so I thought I'd mention it.

Also, the benefits of loosely coupled code far outweigh this, thus I call it a nitpick. Though I've worked long enough to know that this is the sort of push-back you may get if you attempt to introduce dependency injection.

In fact, I'd venture to guess that for every "downside" you can find for dependency injection, you'll find many upsides that far outweigh it.

Share Follow

edited Mar 9, 2010 at 10:49

answered Mar 9, 2010 at 10:36

Jack Leow
22.5k ● 4 ● 53 ● 59

5 Ctrl-shift-B with resharper takes you the implementation – adrianm Mar 9, 2010 at 10:50

1 But then again would we not (in an ideal world) have at least 2 implementations of everything, i.e. at least one real implementation and a mock one for unit testing ;-) – vickirk Mar 11, 2010 at 12:48

1 In Eclipse if you hold Ctrl and hover over method invocation code it will show you a menu to open either the Declaration or Implementation. – Sam Hasler Feb 10, 2011 at 9:32

If you're at the interface definition, highlight the method name and strike Ctrl+T to bring up the type hierarchy for that method--you'll see every implementor of that method in a type hierarchy tree. – qualidafial Feb 11, 2011 at 20:12

12

Constructor-based dependency injection (without the aid of magical "frameworks") is a clean and beneficial way to structure OO code. In the best codebases I've seen, over years spent with other ex-colleagues of Martin Fowler, I started to notice that most good classes written this way end up having a single `doSomething` method.

The major downside, then, is that once you realise it's all just a kludgy long-handed OO way of writing closures as classes in order to get the benefits of functional programming, your motivation to write OO code can quickly evaporate.

Share Follow edited Feb 10, 2011 at 10:21

answered Feb 10, 2011 at 10:15

sanityinc
15.2k ● 2 ● 51 ● 43

1 Problem with constructor-based is that you'll always need to add more constructor args... – Miguel Ping Feb 10, 2011 at 11:24

1 Haha. If you do that too much you will soon see that your code is crappy and you will refactor it. Besides, ctrl-f6 ain't that hard. – time4tea Feb 10, 2011 at 17:10

1 And of course the reverse is also true: it's all just a kludgy long-handed functional way of writing classes as closures in order to get the benefits of OO programming – CurtainDog Feb 11, 2011 at 9:32

Many years ago I built an object system using closures in Perl, so I get what you're saying, but encapsulating data isn't a *unique* benefit of OO programming, so it's not clear what these beneficial features of OO are which would be comparably kludgy and long-handed to obtain in a functional language. – sanityinc Feb 11, 2011 at 13:34

10

I find that constructor injection can lead to big ugly constructors, (and I use it throughout my codebase - perhaps my objects are too granular?). Also, sometimes with constructor injection I end up with horrible circular dependencies (although this is very rare), so you may find yourself having to have some kind of ready state lifecycle with several rounds of dependency injection in a more complex system.

However, I favour construtor injection over setter injection because once my object is constructed, then I know without a doubt what state it is in, whether it is in a unit test environment or loaded up in some IOC container. Which, in a roundabout sort of way, is saying what I feel is the main drawback with setter injection.

(as a sidenote, I do find the whole topic quite "religious", but your mileage will vary with the level of technical zealotry in your dev team!)

Share Follow

answered Mar 9, 2010 at 9:36
James B
3,742 ● 1 ● 27 ● 34

8 If you have big ugly constructors, may be your classes are to big and you just have to much dependencies? – Robert Mar 9, 2010 at 9:41

5 That is certainly a possibility I'm willing to entertain!...Sadly, I don't have a team I can do peer reviews with though. *violins play softly in the background, and then screen fades to black...* – James B Mar 9, 2010 at 9:48

1 As Mark Seeman sad above "It can be dangerous for your career" ... – Robert Mar 9, 2010 at 9:56

I had no clue background narrations could be so expressive here :P – Anurag Mar 9, 2010 at 10:10

@Robert I'm trying to find the quote, where above did he say that? – liang Jun 2, 2015 at 14:43

If you're using DI without an IoC container, the biggest downside is you quickly see how many dependencies your code actually has and how tightly coupled everything

8

really is. ("But I thought it was a good design!") The natural progression is to move towards an IoC container which can take a little bit of time to learn and implement (not nearly as bad as the WPF learning curve, but it's not free either). The final downside is some developers will begin to write honest to goodness unit tests and it will take them time to figure it out. Devs who could previously crank something out in half a day will suddenly spend two days trying to figure out how to mock all of their dependencies.

Similar to Mark Seemann's answer (now deleted; only visible with more than 10,000 reputation points), the bottom line is that you spend time becoming a better developer rather than hacking bits of code together and tossing it out the door/into production. Which would your business rather have? Only you can answer that.

Share Follow

edited Jan 15, 2023 at 23:26
Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Mar 9, 2010 at 9:05
Mike Post
6,450 ● 3 ● 41 ● 49

Good point. That is poor quality/fast delivery vs good quality/slow delivery. The downside? DI is not magic, expecting it to be is the downside. – liang Jul 18, 2013 at 15:33

5

Code readability. You'll not be able to easily figure out the code flow since the dependencies are hidden in XML files.

Share Follow

answered Mar 9, 2010 at 8:33

Rahul
13k ● 14 ● 59 ● 64

10 You don't need to use XML configuration files for dependency injection - choose a DI container which supports configuration in code. – Håvard S Mar 9, 2010 at 8:35

9 Dependency injection doesn't necessarily imply XML files. It seems like this may still be the case in Java, but in .NET we abandonded this coupling years ago. – Mark Seemann Mar 9, 2010 at 8:35

8 Or don't use a DI container at all. – Jesse Millikan Mar 9, 2010 at 8:36

1 if you know the code is using DI, you can easily assume who sets the dependencies. – Bozho Mar 9, 2010 at 8:49

1 In Java, Google's Guice doesn't need XML files for example. – Epaga Feb 14, 2011 at 7:54

5

DI is a technique or a pattern and not related to any framework. You can wire up your dependencies manually. DI helps you with SR (Single responsibility) and SoC (separation of concerns). DI leads to a better design. From my point of view and experience **there are no downsides**. Like with any other pattern you can get it wrong or misuse it (but what is in the case of DI quite hard).

If you introduce DI as principle to a legacy application, using a framework - the single biggest mistake you can do is to misuse it as a Service-Locater. DI+Framework itself is great and just made things better everywhere I saw it! From organizational standpoint, there are the common problems with every new process, technique, pattern, ...:

- You have to train you team
- You have to change your application (which include risks)

In general you have to **invest time and money**, beside that, there a no downsides, really!

Share Follow

edited Mar 9, 2010 at 18:28

answered Mar 9, 2010 at 9:37

Robert
1,466 ● 10 ● 25

3

It seems like the supposed benefits of a statically-typed language diminish significantly when you're constantly employing techniques to work around the static typing. One large Java shop I just interviewed at was mapping out their build dependencies with static code analysis...which had to parse all the Spring files in order to be effective.

Share Follow

answered Feb 10, 2011 at 12:40

Chris D
107 ● 1 ● 4

2

Two things:

- They require extra tool support to check that the configuration is valid.

For example, IntelliJ (commercial edition) has support for checking the validity of a Spring configuration, and will flag up errors such as type violations in the configuration. Without that kind of tool support, you can't check whether the configuration is valid before running tests.

This is one reason why the 'cake' pattern (as it's known to the Scala community) is a good idea: the wiring between components can be checked by the type checker. You don't have that benefit with annotations or XML.

- It makes global static analysis of programs very difficult.

Frameworks like Spring or Guice make it difficult to determine statically what the object graph created by the container will look like. Although they create an object graph when the container is started up, they don't provide useful APIs that describe the object graph that /would/ be created.

Share Follow

answered Feb 10, 2011 at 10:41

Martin Ellis
9,651 ● 43 ● 53

1 This is absolute bull. Dependency injection is a concept, it does not require a fricken framework. All you need is new. Ditch spring and all that crap, then your tools will work just fine, plus you will be able to refactor your code much more better. – time4tea Feb 10, 2011 at 17:02

True, good point. I should've been more clear that I was talking about problems with DI frameworks and not the pattern. It's possible I missed the clarification to the question when I answered (assuming it was there at the time). – Martin Ellis Feb 14, 2011 at 17:39

Ah. In which case I happily retract my annoyance. apologies. – time4tea Feb 14, 2011 at 17:59

1

It can increase app startup time because IoC container should resolve dependencies in a proper way and it sometimes requires to make several iterations.

Share Follow

edited Mar 9, 2010 at 8:55

Ian Nelson
58.6k ● 20 ● 78 ● 104

answered Mar 9, 2010 at 8:54

Roman
66.1k ● 93 ● 244 ● 339

3 Just to give a figure, a DI container should resolve multiple thousands dependencies per second. (codinginstinct.com/2008/05/…) DI containers allow deferred instantiation. Performance should (even in huge applications) not be a problem. Or at least performance problem should be addressable and not be a reason to decide against IoC and its frameworks. – Robert Mar 9, 2010 at 9:12