When do you use dependency injection?

Asked 16 years, 3 months ago Modified 12 years, 2 months ago

Viewed 2k times

16

I've been using StructureMap recently and have enjoyed the experience thoroughly. However, I can see how one can easily get carried away with interfacing everything out and end up with classes that take in a boatload of interfaces into their constructors. Even though that really isn't a huge problem when you're using a dependency injection framework, it still feels that there are certain properties that really don't need to be interfaced out just for the sake of interfacing them.

Where do you draw the line on what to interface out vs just adding a property to the class?

dependency-injection

Share

Improve this question

Follow

asked Aug 26, 2008 at 15:55

Kevin Pang

41.4k ● 38 ● 122 ● 173

9 Answers Sorted by: Highest score (default)

22

The main problem with dependency injection is that, while it gives the appearance of a loosely coupled architecture, it really doesn't.

What you're really doing is moving that coupling from the compile time to the runtime, but still if class A needs some interface B to work, an instance of a class which implements interface B needs still to be provided.

Dependency injection should only be used for the parts of the application that need to be changed dynamically without recompiling the base code.

Uses that I've seen useful for an Inversion of Control pattern:

- A plugin architecture. So by making the right entry points you can define the contract for the service that must be provided.
- Workflow-like architecture. Where you can connect several components dynamically connecting the output of a component to the input of another one.
- Per-client application. Let's say you have various clients which pays for a set of "features" of your project. By using dependency injection you can easily provide just the core components and some "added" components which provide just the features the client have paid.
- Translation. Although this is not usually done for translation purposes, you can "inject" different

language files as needed by the application. That includes RTL or LTR user interfaces as needed.

Share Improve this answer Follow

edited Oct 12, 2012 at 5:02
rism
12.1k ● 16 ● 78 ● 121

answered Aug 26, 2008 at 17:58
Jorge Córdoba
52.1k ● 11 ● 82 ● 130

1 I've run out of votes today so I can't vote you up but I like your answer. You're right - just because you use DI doesn't mean you code is defacto loosely coupled. – David Robbins Oct 5, 2008 at 16:47

I think I disagree with this answer. To my knowledge, the biggest and most important benefit of DInjected code is testability. You literally cannot read a single book on Dependency Injection nor Test-Driven Development, without one referring to the other. They go together like oatmeal and raisins. – Breealzibub Jan 24, 2013 at 13:35

Dinjected code is easier to mock which makes it easier to create tests for things that use that injected code. Still I think that testing alone shouldn't be a reason to use dependency injection. Dependency injection adds complexity (more interfaces, less straightforward architecture) so if it doesn't reduce it considerably in other ways (other than testing) or solves an specific problem that you have, it's not worth adding it. – Jorge Córdoba Jan 24, 2013 at 13:45

Thank you for this answer. I've been looking everywhere to see *when* I would really want to use it. – johnny Mar 27, 2014 at 21:10

9

✓

Think about your design. DI allows you to change how your code functions via configuration changes. It also allows you to break dependencies between classes so that you can isolate and test objects easier. You have to determine where this makes sense and where it doesn't. There's no pat answer.

A good rule of thumb is that if its too hard to test, you've got some issues with single responsibility and static dependencies. Isolate code that performs a single function into a class and break that static dependency by extracting an interface and using a DI framework to inject the correct instance at runtime. By doing this, you make it trivial to test the two parts separately.

Share Improve this answer Follow

answered Aug 26, 2008 at 16:03

user1228

8

> Dependency injection should only be used for the parts of the application that need to be changed dynamically without recompiling the base code

DI should be used to isolate your code from external resources (databases, webservices, xml files, plugin architecture). The amount of time it would take to test your logic in code would almost be prohibitive at a lot of companies if you are testing components that DEPEND on a database.

In most applications the database isn't going to change dynamically (although it could) but generally speaking it's almost always good practice to NOT bind your application to a particular external resource. The amount involve in changing resources should be low (data access classes should rarely have a cyclomatic complexity above one in it's methods).

Share Improve this answer Follow

edited May 16, 2012 at 5:16
user
5,401 ● 7 ● 52 ● 65

answered Oct 5, 2008 at 16:32
Korbin
1,808 ● 2 ● 19 ● 29

2

What do you mean by "just adding a property to a class?"

My rule of thumb is to make the class unit testable. If your class relies on the implementation details of another class, that needs to be refactored/abstracted to the point that the classes can be tested in isolation.

EDIT: You mention a boatload of interfaces in the constructor. I would advise using setters/getters instead. I find that it makes things much easier to maintain in the long run.

answered Aug 26, 2008 at 15:59
Chris Marasti-Georg
34.6k ● 17 ● 93 ● 137

2

I do it only when it helps with separation of concerns.

Like maybe cross-project I would provide an interface for implementers in one of my library project and the implementing project would inject whatever specific implementation they want in.

But that's about it... all the other cases it'd just make the system unnecessarily complex

answered Aug 26, 2008 at 15:59
chakrit
61.5k ● 25 ● 136 ● 163

1

Even with all the facts and processes in the world.. every decision boils down to a judgment call - Forgot where I read that
I think it's more of a experience / flight time call. Basically if you see the dependency as a candidate object that may be replaced in the near future, use dependency injection. If I see 'classA and its dependencies' as one block for substitution, then I probably won't use DI for A's deps.

Share Improve this answer Follow

answered Aug 26, 2008 at 16:05
Gishu
137k ● 47 ● 226 ● 311

1

The biggest benefit is that it will help you understand or even uncover the architecture of your application. You'll be able to see very clearly how your dependency chains work and be able to make changes to individual parts without requiring you to change things that are unrelated. You'll end up with a loosely coupled application. This will push you into a better design and you'll be surprised when you can keep making improvements because your design will help you keep separating and organizing code going forward. It can also facilitate unit testing because you now have a natural way to substitute implementations of particular interfaces.

There are some applications that are just throwaway but if there's a doubt I would go ahead and create the interfaces. After some practice it's not much of a burden.

Share Improve this answer Follow

edited Aug 26, 2008 at 17:48

answered Aug 26, 2008 at 16:11

Harpreet
503 ● 5 ● 6

0

Another item I wrestle with is *where should I use dependency injection?* Where do you take your dependency on StructureMap? Only in the startup application? Does that mean all the implementations have to be handed all the way down from the top-most layer to the bottom-most layer?

Share Improve this answer Follow

answered Aug 26, 2008 at 17:04

flipdoubt
14.4k ● 16 ● 66 ● 98

0

I use Castle Windsor/Microkernel, I have no experience with anything else but I like it a lot.

As for how do you decide what to inject? So far the following rule of thumb has served me well: If the class is so simple that it doesn't need unit tests, you can feel free to instantiate it in class, otherwise you probably want to have a dependency through the constructor.

As for whether you should create an interface vs just making your methods and properties virtual I think you should go the interface route either if you either a) can see the class have some level of reusability in a different application (i.e. a logger) or b) if either because of the amount of constructor parameters or because there is a significant amount of logic in the constructor, the class is otherwise difficult to mock.

Share Improve this answer Follow

answered Oct 5, 2008 at 16:43

George Mauer

122k ● 139 ● 395 ● 626