Service Locator is an Anti-Pattern by Mark Seemann

Service Locator is a well-known pattern, and since it was described by Martin Fowler (http://martinfowler.com/articles/injection.html), it must be good, right?

No, it's actually an **anti-pattern** and should be avoided.

Let's examine why this is so. In short, the problem with Service Locator is that it hides a class' dependencies, causing run-time errors instead of compile-time errors, as well as making the code more difficult to maintain because it becomes unclear when you would be introducing a breaking change.

OrderProcessor example

As an example, let's pick a hot topic in DI these days: an OrderProcessor. To process an order, the OrderProcessor must validate the order and ship it if valid. Here's an example using a static Service Locator:

```
public class OrderProcessor : IOrderProcessor
{
    public void Process(Order order)
    {
        var validator = Locator.Resolve<IOrderValidator>();
        if (validator.Validate(order))
        {
            var shipper = Locator.Resolve<IOrderShipper>();
            shipper.Ship(order);
        }
    }
}
```

The Service Locator is used as a replacement for the *new* operator. It looks like this:

```
public static class Locator
{
    private readonly static Dictionary<Type, Func<object>>
        services = new Dictionary<Type, Func<object>>();

    public static void Register<T>(Func<T> resolver)
    {
        Locator.services[typeof(T)] = () => resolver();
    }

    public static T Resolve<T>()
    {
        return (T)Locator.services[typeof(T)]();
    }

    public static void Reset()
    {
        Locator.services.Clear();
    }
}
```

We can configure the Locator using the Register method. A ‘real' Service Locator implementation would be much more advanced than this, but this example captures the gist of it.

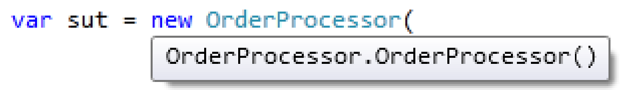
This is flexible and extensible, and it even supports replacing services with Test Doubles, as we will see shortly.

Given that, then what could be the problem?

API usage issues

Let's assume for a moment that we are simply consumers of the OrderProcessor class. We didn't write it ourselves, it was given to us in an assembly by a third party, and we have yet to look at it in Reflector.

This is what we get from IntelliSense in Visual Studio:



```
var sut = new OrderProcessor(
```

Okay, so the class has a default constructor. That means I can simply create a new instance of it and invoke the Process method right away:

```
var order = new Order();
var sut = new OrderProcessor();
sut.Process(order);
```

Alas, running this code surprisingly throws a KeyNotFoundException because the IOrderValidator was never registered with Locator. This is not only surprising, it may be quite baffling if we don't have access to the source code.

By perusing the source code (or using Reflector) or consulting the documentation (ick!) we may finally discover that we need to register an IOrderValidator instance with Locator (a completely unrelated static class) before this will work.

In a unit test test, this can be done like this:

```
var validatorStub = new Mock<IOrderValidator>();
validatorStub.Setup(v => v.Validate(order)).Returns(false);
Locator.Register(() => validatorStub.Object);
```

What is even more annoying is that because the Locator's internal store is static, we need to invoke the Reset method after each unit test, but granted: that is mainly a unit testing issue.

All in all, however, we can't reasonably claim that this sort of API provides a positive developer experience.

Maintenance issues

While this use of Service Locator is problematic from the consumer's point of view, what seems easy soon becomes an issue for the maintenance developer as well.

Let's say that we need to expand the behavior of OrderProcessor to also invoke the IOrderCollector.Collect method (/2010/02/02/RefactoringtoAggregateServices). This is easily done, or is it?

```
public void Process(Order order)
{
    var validator = Locator.Resolve<IOrderValidator>();
    if (validator.Validate(order))
    {
        var collector = Locator.Resolve<IOrderCollector>();
        collector.Collect(order);
        var shipper = Locator.Resolve<IOrderShipper>();
        shipper.Ship(order);
    }
}
```

From a pure mechanistic point of view, that was easy - we simply added a new call to Locator.Resolve and invoke IOrderCollector.Collect.

Was this a breaking change?

This can be surprisingly hard to answer. It certainly compiled fine, but broke one of my unit tests. What happens in a production application? The IOrderCollector interface may already be registered with the Service Locator because it is already in use by other components, in which case it will work without a hitch. On the other hand, this may not be the case.

The bottom line is that it becomes a lot harder to tell whether you are introducing a breaking change or not. You need to understand the *entire* application in which the Service Locator is being used, and the compiler is not going to help you.

Variation: Concrete Service Locator

Can we fix these issues in some way?

One variation commonly encountered is to make the Service Locator a concrete class, used like this:

```
public void Process(Order order)
{
    var locator = new Locator();
    var validator = locator.Resolve<IOrderValidator>();
    if (validator.Validate(order))
    {
        var shipper = locator.Resolve<IOrderShipper>();
        shipper.Ship(order);
    }
}
```

However, to be configured, it still needs a static in-memory store:

```
public class Locator
{
    private readonly static Dictionary<Type, Func<object>>
        services = new Dictionary<Type, Func<object>>();

    public static void Register<T>(Func<T> resolver)
    {
        Locator.services[typeof(T)] = () => resolver();
    }

    public T Resolve<T>()
    {
        return (T)Locator.services[typeof(T)]();
    }

    public static void Reset()
    {
        Locator.services.Clear();
    }
}
```

In other words: there's no structural difference between the concrete Service Locator and the static Service Locator we already reviewed. It has the same issues and solves nothing.

Variation: Abstract Service Locator

A different variation seems more in line with true DI: the Service Locator is a concrete class implementing an interface.

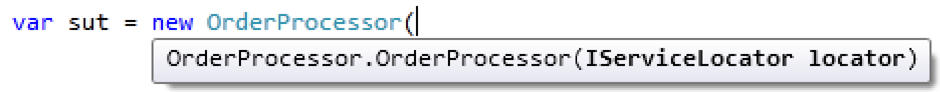
```
public interface IServiceLocator
{
    T Resolve<T>();
}

public class Locator : IServiceLocator
{
    private readonly Dictionary<Type, Func<object>> services;

    public Locator()
    {
        this.services = new Dictionary<Type, Func<object>>();
    }

    public void Register<T>(Func<T> resolver)
    {
        this.services[typeof(T)] = () => resolver();
    }

    public T Resolve<T>()
    {
        return (T)this.services[typeof(T)]();
    }
}
```

With this variation it becomes necessary to inject the Service Locator into the consumer. **Constructor Injection** is always a good choice for injecting dependencies, so OrderProcessor morphs into this implementation:

```
public class OrderProcessor : IOrderProcessor
{
    private readonly IServiceLocator locator;

    public OrderProcessor(IServiceLocator locator)
    {
        if (locator == null)
        {
            throw new ArgumentNullException("locator");
        }

        this.locator = locator;
    }

    public void Process(Order order)
    {
        var validator =
            this.locator.Resolve<IOrderValidator>();
        if (validator.Validate(order))
        {
            var shipper =
                this.locator.Resolve<IOrderShipper>();
            shipper.Ship(order);
        }
    }
}
```

Is this good, then?

From a developer perspective, we now get a bit of help from IntelliSense:



```
var sut = new OrderProcessor(
```

```
OrderProcessor.OrderProcessor(IServiceLocator locator)
```

What does this tell us? Nothing much, really. Okay, so OrderProcessor needs a ServiceLocator - that's a bit more information than before, but it still doesn't tell us *which services* are needed. The following code compiles, but crashes with the same KeyNotFoundException as before:

```
var order = new Order();
var locator = new Locator();
var sut = new OrderProcessor(locator);
sut.Process(order);
```

From the maintenance developer's point of view, things don't improve much either. We still get no help if we need to add a new dependency: is it a breaking change or not? Just as hard to tell as before.

Summary

The problem with using a Service Locator isn't that you take a dependency on a particular Service Locator implementation (although that may be a problem as well), but that it's a bona-fide **anti-pattern**. It will give consumers of your API a horrible developer experience, and it will make your life as a maintenance developer worse because you will need to use considerable amounts of brain power to grasp the implications of every change you make.

The compiler can offer both consumers and producers so much help when **Constructor Injection** is used, but none of that assistance is available for APIs that rely on Service Locator.

You can read more about DI patterns and anti-patterns in my book (http://amzn.to/12p90MG).

Update 2014-05-20: Another way to explain the negative aspects of Service Locator is that it violates SOLID (/2014/05/15/service-locator-violates-solid).

Update 2015-10-26: The fundamental problem with Service Locator is that it violates encapsulation (/2015/10/26/service-locator-violates-encapsulation).

Comments

Janus (http://www.janusknudsen.dk)
I couldn't agree more on this :), thank you for different examples.

I would like to hear your opinion, what do you think about a service locator that operates with dependency injection? So when you call the service locator it returns the requested type that was specified in the di wiring?
For instance.. many DI framework have trouble working without constructors as in .aspx, .asmx, .svc, how would you solve such scenario?
2010-02-03 23:17 UTC

FZelle
Sure, if you don't understand how something was meant to be used you can missuse everything.
A hammer is only aprobiate when you need to hit on something.

Servicelocator is only useable if you need to create dynamic objects inside your class,
otherwise it really is an antipattern.

The right thing to do is use Injection where the dependency tree is fixed and ServiceLocator
where you are sure you need to dynamicaly create something new (in a factory for ex.).
2010-02-04 09:54 UTC

Will (http://www.humblecoder.co.uk)
I always struggle to do IoC without service locator for the exact reasons you stated, how do you get rid of all pesky little new operators. Much of the advice I've read is that passing around a container is a bad idea.

Lately, I've been using AutoFac more and more which, I feel, helps with these type of problems by using delegate factories. Be interested to hear your thoughts on if this is a good compromise?

BTW, nice pimp of your book, I might buy it if it's all this thought provoking :)

2010-02-04 10:00 UTC

Mark Seemann (/)
It's true that frameworks such as ASP.NET, PowerShell and the MMC SDK (but *not* WCF) are inherently DI-unfriendly because they insist on managing the lifetime of important objects. ASP.NET Page objects are the most well-known example.

In such cases you really have no recourse but to move the **Composition Root** into each object (e.g. Page) and let your DI Container wire up your dependencies from there. This may look like the Service Locator anti-pattern, but it isn't because you still keep container usage at an absolute minimum. The difference is that the 'absolute minimum' in this case is inside the constructor or Initialize method of each and every object (e.g. Page) created by the framework (e.g. ASP.NET).

At that point, we need to ask our DI Container to wire up the entire object graph for further use. This also means that we should treat the object where we do this as a **Humble Object** whose only responsibility is to act as an Adapter between the framework and our own code that uses proper DI patterns.

In ASP.NET, we can pull the container from the Application object so that we can have a single, shared container that can track long-lived (i.e. Singleton-scoped) dependencies without having to resort to a static container. This is by far the best option because there will be no 'virtual new operator' available further down the call stack - you can't call Locator.Resolve<IMyDependency>() because there will be no static container.

The Hollywood principle still applies to the rest of the application: **Don't call the container, it'll call you**. Instead of bootsrapping the application in one

go from its entry point (as we can do in ASP.NET MVC) we need to bootstrap each created object individually, but *from there on down*, there will be no Service Locator available.
2010-02-04 10:18 UTC

Mark Seemann (/)
@FZelle: I don't agree that Service Locator is ever appropriate. The standard solution if you need short-lived or dynamically created objects is to use an injected **Abstract Factory**.
2010-02-04 10:32 UTC

Mark Seemann (/)
@Will: Autofac is just one among several DI Containers. I'm not yet that familiar with it, but if Delegate Factories are comparable to Windsor's Typed Factory Facility, it sounds like a good approach. In general **Constructor Injection** is preferrable, but there are many cases where you need a short-lived object.

This is where **Abstract Factories** bridge the gap. What is better about an Abstract Factory compared to a Service Locator is that it is strongly typed: you can't just ask it for any dependency, but only for instances of specific types.
2010-02-04 10:46 UTC

Huy Nguyen
I just cannot agree. The problem with IntelliSense or Compiler doesn't make sense to reject the use of service locator. IMHO, the only problem for the misuse here is that the developer (or maintenance developer) does not know what service locator is and how it works. What about using Convention Over Configuration that we don't need to include the configuration file but specifying all the contracts and default implementations then initialize them at bootstrapper. Also, if you think the error is only found at runtime, we can do unit test to make sure all the dependencies are configured correctly before the component is delivered.
2010-02-05 04:03 UTC

Mark Seemann (/)
Huy Nguyen

Thank you for your comment.

Missing IntelliSense and compiler support are just two symptoms of a poorly modeled object model. You are in your right to disagree, but I consider good API design to be as explicit as possible and adhere to the Principle of Least Astonishment (http://en.wikipedia.org/wiki/Principle_of_least_astonishment). It really has nothing to do with whether I, as a developer, understand the Service Locator (anti-)pattern or not.

A good API should follow design principles for reusable libraries. It doesn't really matter whether you are building a true reusable library, or just a single application for internal use. If you want it to be maintainable, it must behave in a non-surprising and consistent manner, and not require you to have intimate knowledge of the inner workings of each class. This is one of the driving forces behind Domain-Driven Design (http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215), as well as such principles as Command-Query Separation (http://en.wikipedia.org/wiki/Command-query_separation).

Any class that internally uses a Service Locator isn't Clean Code because it doesn't communicate its intent or requirements.

Unit tests may help, but it would be symptomatic treatment instead of driving to the core of the problem. The benefit of unit testing is that it provides us with faster feedback than integration testing or systems testing would do. That's all fine, but the compiler provides even faster feedback, so why should I resort to unit testing if the compiler can give me feedback sooner?
2010-02-05 10:13 UTC

Arnis L.
It's not an anti-pattern.

Actually - `anti-pattern` is not an antonym to `pattern`. Pattern does NOT say that something is cool and good per se in every imaginable situation.

Conclusion:

while these are definitely correct and good points you make and everyone should be aware of them, we return back to those boring phrases like =>

"use right tool for the job"

and

"it depends..."
2010-02-06 21:19 UTC

Mark Seemann (/)
Arnis L.

Thank you for writing.

At the risk of taking the discussion in the wrong direction, "AntiPatterns" [Brown et al., Wiley, 1998] define an anti-pattern as a "commonly ocurring solution to a problem that generates decidedly negative consequences" (p. 7). According to that definition, Service Locator is an anti-pattern.

The reason I insist on protracting this semantic debate is that I have yet to see a valid case *for* Service Locator. There may be occasions where we need to invoke container.Resolve from deeper in the application than we would ideally have liked (the ASP.NET case described above in point), but that isn't the Service Locator anti-pattern in play, but rather a set of distributed **Composition Roots**.

To me, Service Locator is when you use the Resolve method as a sort of 'virtual new operator', and that is just never the right tool for the job.
2010-02-06 21:45 UTC

Arnis L.
People don't act accordingly to mentioned definition. That's their nature. Anyway - direction is surely wrong.
What's interesting - i started to feel like you - can't find reason for service locator.
Good thing is - you gave a great push. At least - for me. Currently - clearing out confusion about IoC in my head.
Bad thing is - learning process is still in progress and i haven't found feeling that 'i know' to wholeheartedly agree with you.
Must confess that I've used related tools and techniques without necessary knowledge. :)
2010-02-06 23:11 UTC

Mark Seemann (/)
To be fair, I must admit that I'm puposefully being rigid and unrelenting in my tone to get the point across :)

Here's an interesting confession: There are a few places in Safewhere's production code where we call Resolve() pretty deep within the bowels of the application. You could well argue that we apply Service Locator there. The point, however, is that I still don't consider those few applications valid, but rather as failures on my part to correctly model our abstractions. One day, I still hope to get them right so I can get rid of those last pieces, and I actually managed to remove one dirty part of that particular API just this week.

We are all only fallible humans, but that doesn't stop me from striving towards perfection :)
2010-02-06 23:25 UTC

FZelle
@Mark:
And that abstract Factory is useing new xyz()?
No that Factory than needs the Servicelocator.
2010-02-08 09:27 UTC

Mark Seemann (/)

@FZelle: The Abstract Factory needs no dependencies since it's an abstraction (interface/abstract base class).

Concrete implementations of an Abstract Factory may very well need specific dependencies, which it can request via **Constructor Injection** - just like any other service.

The DI Container will then wire up the concrete factory with its dependencies just like it wires up any other service. No Service Locator is ever needed.

Here's (http://stackoverflow.com/questions/1926826/cant-combine-factory-di/1927167#1927167) just one example demonstrating what I mean.
2010-02-08 14:25 UTC

Andrei Alecu
In some advanced usage scenarios, dependencies are not always set in stone and known at design time, or even at application startup time. They could even change several times throughout the run-time of the application, so, having a service locator which can manage this reconfiguration aspect from a single place can be preferred.

This is similar to an abstract factory pattern, but it's much more feasible to implement when you have a ton of dependencies.

Additionally, if you consider a service oriented architecture where the services are well known and are always guaranteed to exist, but you just don't know where (could be on a separate server via WCF), then the service locator pattern solves a lot of problems.

Passing a ton of dependencies through dependency injection every time leads to unreadable code in my opinion. If by convention, the service is guaranteed to always exist, then Service Locator is a valuable pattern.

I would consider this to be an application of the Convention over Configuration paradigm.

However, it is a pattern that should only be used where it makes sense, I can see how abusing it would create more problems than it solves.
2010-05-11 14:14 UTC

Krzysztof Kozmic (http://kozmic.pl)
@Andrei Alecu

Modern IoC containers let you resolve dependencies at runtime without using service locator. Windsor for example has several places where you can plug in to provide the dependencies, DynamicParameters method or OnCreated are just two examples from the top of my head. More generic solutions like handler selectors or subdependencyresolvers also exist.

In cases where you do need to trigger resolution of a dependency from the call site many containers provide autogenerated abstract factories (Windsor has TypedFactoryFacility which creates interface based factories and in trunk (v2.5) also delegate based factories).

So in my opinion using SL is just an excuse for being lazy. The only place where I'd use it (temporarily) is when migrating old, legacy code that can't be easily changed, as an intermediate step.
2010-06-11 03:52 UTC

Iran Hutchinson
Just came across this discussion. My opinion is whether you use ServiceLocator or not depends on your application requirements. What if you have to run inside or outside of a container environment? A service locator does make that situation much easier. What about SOA [implementation, not theory :)]? SOA generally involves some service registry and service lookup. You can definitely look at that as a ServiceLocator implementation. There are good SOA design that have great merit, especially in a distributed environment where services are not guaranteed to be local.

The strength or weakness of your choice and use of patterns depends on how they meet your requirements. There is no one golden way. Dependency Injections is not a cure all nor was service locator when it became widely used. I use both and many others.

And how many patterns can you fit into the definition of "Anti-Pattern" if it is misused / abused?
2010-07-21 23:54 UTC

Mark Seemann (/)
What do you mean by running "inside or outside of a container environment"? If you design your software appropriately, the presence of a DI container has no impact on the overall dependency usage. A DI container should resolve the appropriate object graph and get out of the way. This is what I call the **Hollywood Principle for DI Containers**: *Don't call the container - it will call you*.

In my opinion the SOA discussion is completely orthogonal to the discussion about the Service Locator (anti-)pattern. We shouldn't be misled by the similarity of names. The *services* located by a Service Locator are object services that may or may not encapsulate or represent an external resource. That external resource may be a web service, but could be something entirely different. When we discuss the Service Locator (anti-)pattern, we discuss it in the context of object-oriented design. Whether or not the underlying implementation is a web service is irrelevant at that level.

That said, I'm fully aware that many SOA environments work with a service registry. UDDI was the past attempt at making this work, while today we have protocols such as WS-Discovery. These can still be considered implementations of 'design patterns', but they certainly operate on a different architectural level. They have nothing to do with the Service Locator (anti-)pattern.

Nothing prevents us from combining proper Dependency Injection with service registries. They are unrelated because they operate at different levels, so they can be combined with exact the same flexibility as data access and UI technologies.

If we go back to the (original?) definition of the term *anti-pattern* from the book (http://www.amazon.com/AntiPatterns-Refactoring-Software-

Architectures-Projects/dp/0471197130), an anti-pattern is characterized by a "commonly occurring solution to a problem that generates decidedly negative consequences". By that definition, Service Locator is an anti-pattern - even if you can come up with niche scenarios where it makes sense. So far, I've never been presented with a case for Service Locator where I haven't been able to come up with a better design that uses proper DI.
2010-07-24 12:29 UTC

Karl Shifflett (http://karlshifflett.wordpress.com/)
Sauce béarnaise is a great way to start the book.

I first learned how to prepare sauce béarnaise when I started making Sole Wellington.

Reading book now, nice.

Cheers,

Karl
2010-07-31 17:04 UTC

Iran Hutchinson
I have multi-language / platform responsibilities which include .NET and Java Prior to DI being popular in Java many of us had requirements to run inside container (EJB, Spring, Pico, etc.) and outside of containers (just what was in the Virtual Machine libraries [prior to JDK version 1.5]). There was no DI available for free (unless you wrote it yourself). This is true still today in a number of environments.

If I go directly the posted definition of anti-pattern, which most can agree with, I can put a number of patterns that are not always anti-patterns in this category. I tend to emphasize proper pattern usage. Patterns are just tools that can be correctly or incorrectly. I have seen design-by-contract (interface pattern) used in a manner that fits the anti-pattern definition but I don't consider it an anti-pattern. I consider the use of pattern poor or inappropriate for that situation.

I consider SOA != Web services + UDDI + etc,. Web services and its relative technology stack can be used to implement an SOA. Therefore, from my perspective it is relative to this discussion and not orthogonal. In fact we had a discussion relative to these to these concepts last week in one of my current projects. The goal of that discussion was the abstraction of service implementation + location in a high-performance low-latency SOA design.

I am not a proponent of Service Locator or any other pattern. I us it and others with/without DI where appropriate. The perceived / actual negative impact of a pattern usage in one scenario does not apply to all scenarios. There are definitely patterns where one can get a majority agreement on them being anti-patterns. I just don't think Service Locator is one. However, the pursuit of try to qualify the validity of a pattern's usage or existence does generally lead to other solutions or ideas. Which I am a proponent of.

Cheers,

Iran
2010-08-01 00:42 UTC

Al
You have missed totally the point with service locator pattern. Probably you have done this intentionally, just to gain attention to try to sell a bunch of books. All you complaints can be equally applied to most DI implementations. In fact, many DI frameworks (Spring and Seam came into my mind), use a service locator internally and even let you to use it directly.

Service locator and DI, have essentially the same main downsides, and the only real difference is whether you prefer to retrieve the component from your code of to have it injected externally. In fact, and at least in the Java implementations i have worked with, you get less compiler support with DI variants, and also have a harder time to debug because of faulty injections. I am still amazed to see people arguing about how you get a dependency with a particular service locator implementation, when you a) can have a light wrapper to decouple from it and b) you get the same dependency with your particular DI implementation, you still have to tell de container what, how and when you need the injection, and thats a change too.

With you service locator example, you have only proved that it is also possible to write a crappy service locator implementation. And your moans are about that, not the pattern.
2010-09-10 08:57 UTC

Phil
Sure, Service locator is not the best way to implement kind of IoC. Your argument about hidden dependencies is certainly a good point.

Though, how many developers with good will and at least, average knowledge are trying to use IoC just to fail to understand how to set up their root components and bootstrap correctly their dependencies?

Now, imagine those who understand well those principles and techniques, trying to explain, guide, support an entire team of developers about those concepts...I guess you see (or have seen) like me those astonished developers faces trying to understand and not mess this magic behind dependency injection.

Ok, I'll admit it is not so hard. But still, my point is it is much harder to understand how those dependencies are injected and how not to mess and miss the point than using a simple Service locator instead of a "new" keyword.

And as someone else mention before, it is mandatory to use a Service locator with good convention-over-configuration to have fun with it. But it is the same with Dependency injection.

I see service locator as a first step for a team into IoC. After the concept and benefits are known by the team, I think it is much easier to turn the ship towards the Dependency injection final destination.

Good article and back and forth discussions here
Thanks
Phil
2011-05-09 16:01 UTC

Alex Brina
What every article I read fail to show is how to use DI without some sort of container, or factory or locator or whatever is the name of the "thing" responsible for objects instantiation. It's no big deal moving all "new" keywords out of my "OrderProcessor" class, preparing them for injection via constructor or setter methods, but what about the calling object? Is it now responsible for "injecting" the dependencies? Should it instantiate them and pass to "OrderProcessor"? where is the value of this thing I may call "Inversion of Dependency"? Please show me where dependencies come from? Should the calling object call the DIC to instantiate "OrderProcessor" fulfilling its dependencies based on some configuration? No, because it is masking its dependency on "OrderProcessor", your calling object should have received the "OrderProcessor" beforehand. Allright, but now it's the calling object of the object who's responsible for passing all this chain of dependencies. I just don't get it, and I would love to understand this DI without "Container" thing!
2011-05-28 01:03 UTC

Mark Seemann (/)
With DI, all "new" keywords are being pushed to the entry point of the application. This is what is called the *Composition Root*. In this example (/2010/04/07/DependencyInjectionisLooseCoupling) I use Poor Man's DI towards the end of the post.

FWIW the first nine chapters of my book (http://amzn.to/12p90MG) discuss DI mainly in the context of Poor Man's DI (that is, without containers).
2011-05-28 21:20 UTC

Alex Brina
Hello Mark,

thanks for you answer, I was subconsciously avoiding this concept of Composition Root, although I didn't know why. But today I came to some code that shed some light on it, and the driving force against using the entry point for dependecies instantiation was my understanding of "encapsulation". As I see it, some dependencies should not be known outside the object. The code I mentioned, for instance, is a class named "Syncronizer" which defines a sequency of "Steps" (each step is an object), the sequence and steps are defined internally and it is the Syncronizer's sole responsibility, chosen steps are "instantiated" internally (say 10 of 30 available), The entry point isn't aware of which Steps would be needed. AS I see, there is something in DI that goes against "encapsulation". Would love to know your thoughts about it.

Alex Brina
2011-06-01 15:35 UTC

Mark Seemann (/)
DI doesn't go against encapsulation, which is one of the most misunderstood concepts in OOP (/2011/05/24/Poka-yokeDesignFromSmelltoFragrance).
2011-06-01 18:49 UTC

Patrick Sears
Mark - I love this post. I'm diving into IoC and planning to use the Unity framework and one of the first and immediate problems I ran into was the need to resolve types deep within the object hierarchy. So of course, first thing I did was search for common ways to do this.

1. Passing around the container was obviously the wrong approach
2. A global, static container also "felt" like the wrong approach... the business code shouldn't need to know that much about the application domain.
3. A very common solution around the web is the ServiceLocator pattern, but - the reason I continued looking is that pattern just "felt" wrong to me in an intuitive sense - it's essentially no better than (2) and as you say, the container begins to invade the business logic. Now, the entire architecture has to know about the container - which completely defeats the purpose of DI and IoC.

So then I stumbled upon your post and it summed up EXACTLY why those approaches "felt" wrong - and also helped me re-orient my thinking with regard to where to create the containers.

I don't NEED a single container for the whole application, I only need them within the scope where they're required. If I have a type that is only ever instantiated 4 levels deep in the object hierarchy, it makes no difference whether I register the type at the root of the application or scoped to the code where that type is relevant. Thus, in order to prevent the container from invading the application in the ways you describe, it makes perfect sense to create the container at a lower level (what you call the Composition Root?).

Really like your blog. I appreciate the dedication to design principles that seem so easily thrown away when pragmatism requires.
2011-07-11 08:14 UTC

Danil
I constantly get this when attempting to post a comment:

An error has been encountered while processing the page. We have logged the error condition and are working to correct the problem. We apologize for any inconvenience.

Page rendered at Wednesday, July 13, 2011 8:02:11 AM (Romance Daylight Time, UTC+02:00)
2011-07-13 08:03 UTC

Mark Seemann (/)
Yes, the software hosting this blog is crap, and migration is on my to-do list. The error usually has something to do with the use of angle brackets.

Sorry about the inconvenience.
2011-07-13 09:44 UTC

Gary McLean Hall (http://garymcleanhall.wordpress.com/)
Hi Mark,

Apologies for the necromancy, but I googled for 'Service Locator anti pattern' and this post is one of the top-five hits.

I totally agree that Service Locator is an anti-pattern. It's the use of the static class/property that makes it so insidious. It's an example of a Skyhook (http://garymcleanhall.wordpress.com/2011/07/24/skyhooks-vs-cranes/) as opposed to DI, which is a Crane.

Service Locator seems to be preferred because the alternative is sometimes to write a constructor to inject half a dozen (or more) dependencies, ie: it smells like the class has too many responsibilities.
2011-10-05 00:20 UTC

Mark Seemann (/)
Agreed. The important thing is to realize that the smell comes not from Constructor Injection, but from violating the Single Responsibility Principle (/2010/01/20/RebuttalConstructorover-injectionanti-pattern).
2011-10-05 09:39 UTC

Mohammed
A google Tech Talk that addressed pretty much the same issue in 2008 can be seen here:
http://googletesting.blogspot.com/2008/11/clean-code-talks-dependency-injection.html

2011-10-08 16:19 UTC

KT

I actually don't see a problem with the user of the service locator. In the case of both IOC and the service locator, you have the same problems. The only question is how much indirection you want shown the client of the service.

API Usage Issue:
In the case of both IOC and Service locator, there is an independent class that is responsible for making sure that the appopriate service (IOrderValidator or IOrderShipper) is found by the client. The only difference is whether the client automatically receives the information or if the client makes an explicit call to a 3rd party class (Locator) to find it. It is more a semantic argument than anything else.

If the configuration of the IOC is done incorrectly, you will also receive an exception. The problem is with the management of the configuration as opposed to the service locator itself.

Maintenance Issues:
Same problem as above. This is a problem with configuration of the service as opposed to the way in which IOC or service locator is used.

With both service locator and IOC, someone is going to have to understand all the dependencies in order to configure the services correctly and ensure that the correct service is returned when called upon. This configuration is part of the setup of the service contract. Whether or not this configuration is done through an IOC container or through a service locator seems immaterial.
2011-10-17 19:00 UTC

Vlad
=== Service Locator is NOT an anti-pattern ===

Sorry, I cannot agree with author’s statements from practical standpoint. If you discuss pure theory and your team has unlimited resources for your projects - you can use all kind of nice patterns that require more work and give some advantage.

Let's compare advantages and disadvantages and not just talk about pros on one side.

I talk from practical large scale software Dev experience when your product is not one-man show but you have mature product with a lot of components, several teams, and engineers coming and leaving. Is it what we all try to get to? If you are talking about one-man show project or small-highly-skilled team project and you work on v1 or v1.1 of your product - sure you can use any pattern and dependency injection through constructor parameters would probably work for you.

==========================

== Service Locator advantages ==

1. Much smaller production code. No need to declare variables, parameters to pass dependencies, properties for stateful objects, etc.

Every line of code/symbol has associated Dev/QA/support cost. Smaller code is easier to understand and update.

2. Much smaller unit test code. No need to instantiate and pass dependencies - you can set all common dependencies in TestSetup and reset them in TestCleanup. Again, every line of code/symbol has associated cost.

3. Much easier to introduce dependencies. Yes, this is an advantage, because you need to write much less code for this if you use Service Locator. And again every line of code....

Imagine you need to add SQL performance logger deep in the code and of course you want all your DB access methods use the same logger instance no matter how you get there from business logic layer. You have to change all stack of callers if you do not use Service Locator.

And if you have to change any caller because of your method change -this can be a huge problem in mature product. There will be no way you can justify expenses for a new release of the calling component to you management.

== Service Locator disadvantages ==

1. Introduced dependencies are not captured at compile time.

How to fix this? The answer is - test your product (with automatic integration or manual tests). But you have to test only changed functionality to make sure that feature that uses new introduced dependency works correctly. You need this anyway and this is already in the plan/budget for your project/change anyway.

Did I miss anything?

==========================

We've tried to use dependency injection and switch to Service Locator turned out to be such a huge relief
that we would never even consider switching back.

== My conclusion ==

If you are working on long-term project and you have to implement cost-efficient solution, maintainable by the team with various skill levels - Service Locator is significant cost saver for you with small to none down sides.

If quality is your first priority and budget/time-to-market is not an issue and you have skilled team - do not use Service Locator and rely as much as possible on compile time checks.
2011-10-28 09:47 UTC

Arialdo Martini (http://arialdomartini.wordpress.com)
I couldn't agree more.

We could say the container itself should not be a dependency, and hance should not be passed around, as stated in this StackOverflow question.
http://stackoverflow.com/questions/4806918/should-i-be-passing-a-unity-container-in-to-my-dependencies

To me, Service Locator is the exact opposite of Inversion of Control.

http://stackoverflow.com/questions/2386487/is-it-better-to-create-a-singleton-to-access-unity-container-or-pass-it-through is another great StackOverflow question about this topic (no surpsise the correct answer is by Mark Seemann himself.
2011-11-24 13:59 UTC

Barley N. Hopps
The problem with your argument is that you state "I don't agree that Service Locator is ever appropriate", then provide one contrived example where Service Locator is implemented poorly. All you have proven is Service Locator might not be the best pattern to solve this problem (although you don't prove that either by defining a better solution). It seems that you started with the conclusion and derived the premise.
2011-12-15 22:02 UTC

Chris Marisic (http://www.marisic.net)
I vehemently disagree with your premise that service locator itself is an anti-pattern. The service locator is an invaluable pattern when it is needed, and it fulfills roles that there are no other solutions to, other than poor implementations of the service locator pattern.

Do people apply it in less then optimal scenarios? Absolutely. Does that make a pattern, an anti-pattern? Absolutely not.
2012-01-31 13:54 UTC

Zach
Hi!
(I've had to remove angle brackets so I hope the code still makes sense.. where you see a T think generics T with angle brackets:))

All you are doing with your proposal is turning the abstract factory into an "anti-pattern". Why go through the creation of the factory only to create a new level of indirection just to (essentiall)resolve 1 object that needs resolution in 1 one. I know you can reuse this everywhere. Why not create an object

registry interface eg IMyObjectRegistry
eg:
public interface IMyObjectRegistry { T get_for T ();}

public class MyObjectRegistry : IMyObjectRegistry {
public T get_for T (){
return Resolver.Get T ();
}
}

IMyObjectRegistry get passed around. You could also further constrain these to only work with certain types.
You register the "Registry" with the DI container. Now all objects have a way of finding dependencies with the ServiceLocator. Its still very testable you can Mock/Stub/Fake IMyObjectRegistry. You don't even need a DI implementation in your tests.

2012-02-06 10:26 UTC

Adrian
I'm using the service locator pattern in my Zend Framework (ZF) based PHP application. The upcoming ZF2 seems to promote DI and I'm thinking about refactoring with constructor DI and a DI container. However, there's a proposal for DI-enabled service locators (http://framework.zend.com/wiki/display/ZFDEV2/Proposal+for+ServiceLocator+and+DependencyInjector). Doesn't this completely go against the spirit of IoC/DI? Would you consider a DI-enabled service locator an anti-pattern, too?
2012-02-20 16:23 UTC

Mark Seemann (/)
In my opinion, there's no such thing as a "DI-enabled Service Locator". Service Locator and DI are two mutually exclusive concepts (http://www.infoq.com/articles/Succeeding-Dependency-Injection).
2012-02-22 15:56 UTC

Grasp
Did not read the everyone's comments, but dont agree with the post.

It is the Service Locator's responsibilty to RESOLVE the interface implementation, if the interface is not registered, instead of doing NOTHING which bubble's KeyNotFoundException up, it should raise its own exception to notify consumer that the interface has NOT been registered yet.
2012-03-07 23:38 UTC

Justin
@Grasp

I think you fail to understand the argument against ServiceLocator. No one is debating what ServiceLocator's responiblities are.
This problem with ServiceLocator is the problems/headaches it introduces. The biggest one being that it hides a type's dependencies.
It's difficult to write unit tests for a type when it isn't clear from the type's contract what type of dependencies it has. This can make writing unit tests extremely painful and discouraging. Secondly, it make detecting breaking changes a PITA. That fact that your unit test fails because some hidden dependency hasn't been resolved is a side-effect of ServiceLocator.

Unfortunately, when some developers hear the word 'pattern' they automatically assume it's good/best practice. This is hardly the case with ServiceLocator. A pattern that makes your life harder or encourages bad design practices is an anti-pattern.
2012-03-12 14:03 UTC

Hannes Kochniß
I know the arguments so far, but there is _one_ issue that comes up constantly, and even some TDD veterans agree that SL is ok: a Logger. I'm not always in "register the service locator with the framework" MVC wonderland, and AOP doesn't cover it, I don't want to log only on exceptions or method boundaries. Assuming I understand the capabilities of AOP frameworks like PostSharp correctly (I know IL weaving / dynamic proxy is a big difference, but anyway).

So your normal answer is "abstract factory", but that still would mean an additional parameter. Should I inject an ILogger or ILoggerAbstractFactory into EVERY class? That.. would be insane.

So what's your solution there?
2012-05-10 22:05 UTC

Hannes Kochniß
just to change the tone of my post: "insane" is too harsh. :)

As you well argue, "expressing intention" in design is amongst the main points of DI. But ILogger is an cross-cutting concern, showing intention to log in a class is 0% Info to anyone (as is tracing). I know these are the most used examples for AOP, but I repeat myself: AOP doesn't solve logging additional/debug info, just exception logging. Or I'm misunderstanding AOP capabilities.

2012-05-10 22:09 UTC

Mark Seemann (/)
Yes, logging is a cross-cutting concern, which is why I'd strongly recommend not injecting it all. AOP is much preferred (http://stackoverflow.com/questions/7905110/logging-aspect-oriented-programming-and-dependency-injection-trying-to-make/7906547#7906547). It addresses fine-grained instrumentation too, with the caveat that (as you imply) you can only log at method boundaries. However, before you dismiss this as a solution, consider this: why is that not enough? If you need to log from within a single method, it most like means that this method is too big.
2012-05-11 03:55 UTC

Mark Seemann (/)
BTW, even if you're not convinced by the arguments above, or if you have legacy code with long methods, Service Locator is not a good solution. If you don't like to inject an Abstract Factory, you could instead use an Ambient Context. That's not my first choice, but still way better than a Service Locator.
2012-05-11 04:04 UTC

Dima
Hi Mark! I read your post about Ambient Context, but it doesn't look like full-fledged pattern. It has two problems:
1) it doesn't know its scope (app domain, thread, call context, or custom context)
2) it doesn't know when to dispose itself. Both problems come from the fact that it's the root application that should care about those problems.
Again it comes down to Service Locator.
2012-05-14 18:24 UTC

Mark Seemann (/)
Which post?

Did you read the Ambient Context pattern description in my book (http://amzn.to/12p90MG)?

Most loggers already come with an Ambient Context implementation. Logger.Log("foo") is, essentially, an Ambient Context. It's true that it's not particularly clear when an Ambient Context should dispose itself, but then again: why would you need to dispose of something which is a cross-cutting concern? I'd prefer such a thing to stick around (for performance reasons).

Still, if you don't like Ambient Context (I don't), you can always use an Abstract Factory to get your ILogger. Service Locator is not required.
2012-05-14 18:33 UTC

Dima
I read about them in your book and in your old msdn blog.
Dispose problems come with asp.net doing threads reuse. Though CallContext can be used as workaround, it feels a bit hacky.
It's easier with loggers as they usually can be just static. But if I need something to be bound to a HttpContext or a session?

Abstract factory doesn't solve the problem this all started with: extra reference in a constructor. IoC looks very nice when my classes have parameters they really need to do their work. But secondary parameters like ILogger or ILoggerFactory don't make much sense. If I went this way I would prefer something like IInfrastructure and let it have all ILogger properties and other cross cutting stuff.

I still don't feel satisfied with all solutions. I don't like that SL hides contracts, but I also don't like that DI makes them over complicated.
2012-05-14 19:28 UTC

Mark Seemann (/)
Oh, that blog post... I'd completely forgotten about that one.

I agree that injecting an ILoggerFactory could be considered as parameter pollution of the constructor, but remember that (IMO) we are only talking about this as a solution when dealing with legacy code. Properly factored code should have methods that are short enough that a Decorator or Around Advice should be more than enough for logging and instrumentation purposes.

So once again: Service Locator is never a solution for anything. The solution is to refactor the code so that you don't need to inject an ILogger at all.
2012-05-14 20:03 UTC

Tarriq Ferrose Khan
Dear Mark,

am desparately in need of a help, I tried to post my complete content (with code) but it was always throwing an error, Have sent you an email, request you to please review and share your thoughts.

Thanks,
Tarriq
2012-06-30 01:31 UTC

Vaseem
Did you mention anything about how to do DI over web forms in your book ?
2012-07-28 10:13 UTC

Mark Seemann (/)
That's section 7.5, but to be fair, it's only 6 pages, so it's not a major part of the book in any way.
2012-07-28 10:24 UTC

Michael Bui
It appears that this is more an argument about developing using Interfaces versus using concrete implementation. It is clear if you change an interface you will have dependencies that you will affect. In general, it can be assumed that breaking interfaces (changing) them is not a good practice. I think you can resolve your issues by not breaking the interfaces but defined new ones and having the implementation use the new version. This is some of the core concepts underlying component programming. It isnt an issue about the service locator, it seeks to achieve to solve its own problem. The problem here is about how your deal with changes to behavior or structure on a already published contract. In using service locator, knowning your implemenation dependency is a must. Try solution wide refactoring tools.
2012-10-11 17:49 UTC

Michael
So, we've got code 1:

```
public class MyService
{
public MyService (IRepository1 repository, IRepository2 repositor2)
...
public void DoStuff ()
{
repository.Find...
repository2.QueryAllWithCriteria...
...
}
}
```

And code 2:

```
public class MyService
{
public void DoStuff ()
{
```

```
// used square brakets instead of angle
// cause of comment posting error
ServiceLocator.Resolve[IRepository1].Find...
ServiceLocator.Resolve[IRepository2].QueryAllWithCriteria...
...
}
}
```

Since when injecting dependencies (code 1) is helping understand better on on which code MyService really depends compared to code 2? I mean real world repositories usually have a lot of different methods. Just because you pass it around does not help user understand which method inside that repository the service use. For example, if I'm going to write unit test it won't help my at all - unless I got some really simple/generic repository I can't just mock all repository methods just to test my service. So I still end up looking into the code or catching first run test failures to see what I need to mock inside that repository. And no - SRP has nothing to do with it, unless you plan to build separate repository for every single query/operation.

Why then I should clutter my client code with pointless stuff?

For example, I develop a library for internal usage between many projects. It exports some services, command and queries. It does not work with some external repositories - so why should I make all users to pass my repositories to my methods? I could just make them call one single bootstrapper static method during app startup which will register all my necessary components for my library to work.

Service locator is ideal thing for allowing loose coupling of internal components (for ability to unit test them).
2012-11-26 09:54 UTC

Mark Seemann (/)
It's no so much the SRP as it's the ISP which is in play here. I'd strongly recommend building a Role Interface (http://martinfowler.com/bliki/RoleInterface.html) for each query or command instead of the sort of Header Interface (http://martinfowler.com/bliki/HeaderInterface.html) you seem to prefer.
2012-11-26 10:01 UTC

Michael
Hm.. Isn't that means that I'll end up with separate role interface for each method inside repository?

And the second question - why do client (consumer of my library) have to deal with my internals? I mean why does he had to provide my service my repositories?

I think the whole misconception and negativism toward service locator comes from trying to use it everywhere and thus leaking it outside of your domain. Look at those many good libraries - they don't ask us to provide instances of their internals. In order to use NHibernate you have to configure it once and not feeding every session creation bunch of interface instances which it will use. NHibernate will use configuration to get instance of internal object when he needs to - that's basically the service locator with different name.

I think that real anti-patterns are:
1. Usage of constructor or property injection for external interfaces of your domain, that is expecting user to provide instances of interfaces that your service need in order to operate. Though it's good to provide a way for user to override some part implementaion (like NHibernate, MVC does).

2. Usage of service locator for external resources: if your service mean to operate on some external user-provided source you should never expect it to be provided via service locator - use constructor/properties/parameters injection instead.
2012-11-26 10:50 UTC

Mansoor Omrani
I'm agree with the problems you mentioned for the Service Locator pattern. But I think this pattern can be a last resort in scenarios where you have a deployed component on which many clients are depending. In such a case modifying constructor or methods signatures will break those dependent apps, whereas service locator provides -at least- a little inversion of control. But in redesign scenarios or when are designing from the ground up I agree that service locator is a bad choice.

I also suggest a way to amend service locator pattern to overcome dependency opacity in the clients of a locator. I apologize for the code might be a little long. But I'm sure you'll instantly get the idea behind it with in a glance.

best regards

p.s.: I wasn't able to submit the comment because the blog-engine takes angle brackets as xss attack. So I've changed all "lower than" and "greater than" characters into "dollar sign" ($) and "pound sign" (#) respectively. If you perform a replace-all in the code against these characters in reverse direction, the code turns back to normal.

```
public interface IDependency1 { }
public interface IDependency2 { }

public interface IClient
{
KeyValuePair$Type, bool#[] GetDependencies();
bool IsReady();
}
```

```
public interface IServiceLocator
{
void Register$T#(Func$T# resolver);
T Resolve$T#();
bool Contains$T#();
}
public class ServiceLocator: IServiceLocator
{
private readonly Dictionary$Type, Func$object## services = new Dictionary$Type, Func$object##();
public void Register$T#(Func$T# resolver)
{
services[typeof(T)] = () =# resolver();
}
public T Resolve$T#()
{
return (T)services[typeof(T)]();
}
public bool Contains$T#()
{
return services.Keys.Contains(typeof(T));
}
}
public class MyService: IClient
{
private bool hasIDependency1;
private bool hasIDependency2;

public MyService() { }
public MyService(IServiceLocator servicelocator, bool checkDependencies = false)
{
hasIDependency1 = servicelocator.Contains$IDependency1#();
hasIDependency2 = servicelocator.Contains$IDependency2#();

if (checkDependencies)
{
if (!hasIDependency1)
throw new ArgumentException("ServiceLocator doesn't resolve dependency: IDependency1.");
if (!hasIDependency2)
throw new ArgumentException("ServiceLocator doesn't resolve dependency: IDependency2.");
}
}
public KeyValuePair$Type, bool#[] GetDependencies()
{
var result = new KeyValuePair$Type, bool#[]
{
new KeyValuePair$Type, bool#(typeof(IDependency1),hasIDependency1),
new KeyValuePair$Type, bool#(typeof(IDependency2),hasIDependency2)
};
return result;
}
public bool IsReady()
{
return hasIDependency1 && hasIDependency2;
}
}
public class Test
{
public static void Main()
{
var ms = new MyService();

Console.WriteLine("Dependencies of 'MyService':");
Console.WriteLine(" Dependency Resolved");
Console.WriteLine("--------------------------------");

foreach(var item in ms.GetDependencies())
{
Console.WriteLine(item.Key + " " + item.Value);
}

Console.ReadKey();
}
}
/*
```

Dependencies of 'MyService':
Dependency Resolved

Dummy.IDependency1 False
Dummy.IDependency2 False
*/
2013-01-24 08:19 UTC

Mark Seemann (/)
Mansoor, what you describe is already covered in the original post under the heading *Abstract Service Locator*.
2013-01-27 10:55 UTC

Mansoor Omrani
I disagree respectfully. My point is not about dependency injection. I suggested to "amend" service locator pattern. In fact based on your explanation in "Abstract Service Locator" which I'm agree with ...

[blockquote cite="Scott"]
...
What does this tell us? Nothing much, really. Okay, so OrderProcessor needs a ServiceLocator – that's a bit more information than before, but it still doesn't tell us which services are needed.
[/blockquote]

... I suggested a solution by which a class can "report" if the given locator "contains" its required dependencies or not. So this way the OrderProcessor can tell us exactly which services are needed.
2013-01-28 09:49 UTC

Buddy James (http://www.refactorthis.net)
Mark,

I agree with you completely. I've read this article and others like it concerning the Service Locator Anti-pattern. I've also read your book on dependency injection (I read it in a night, great book!). There is one topic that I would love to get your input on and that's the way in which Microsoft's Prism passes the Unity container around to each module for registering it's dependencies. For those who aren't familiar with Prism, it's a library from the Microsoft patterns and practices camp and it facilitates writing loosely coupled applications.
In prism your application is made up of independent modules that implement the IModule interface. Modules can be configured fluidly in code, defined in the application configuration file, and even discovered on demand from the file system. The IModule implementation allows for passing the

IUnityContainer used throughout the application in each module's constructor to allow the module to register it's dependencies. Though I love the flexibility of the Prism Modules, I can't help but feel dirty about the way that the container is passed around sporadically to each module. I as a developer don't know what the developer of the next module to be initialized may do to the container. To me, this mystery aspect of the way that the container is used is strikingly similar to the problems of the Service locator anti-pattern. I feel that I need an "intimate knowledge" of the other modules if I want to be sure about the container. I'm curious about your thoughts on this subject considering you wrote the book :)
Thanks for your time, and keep up the great work!
Buddy James
2013-01-30 20:10 UTC

Mark Seemann (/)
Mansoor, in which way does your proposed amendment of Service Locator change the feedback mechanism? While you propose a more explicit API for querying a Service Locator, it doesn't change the fundamental situation. Even without an explicit query mechanism, you could wrap your Service Locator calls in a try/catch block or check for null return values... so even without an explicit query mechanism, you can get the exact same information. Now you have that information at run-time. What will you do with it?

One major problem with Service Locator is that it doesn't provide feedback until at run-time. It's much better if we can get feedback already at compile time (/2011/04/29/Feedbackmechanismsandtradeoffs), and that's possible with Dependency Injection. It 'reports' on its requirements at compile time. It also enables tools (such as DI Containers) to walk up to the type information and start querying about its dependencies.

The proposed fix for Service Locator doesn't change any of that. What benefits does it provide?
2013-02-04 15:38 UTC

Mark Seemann (/)
Buddy, I think my opinion about passing the container around is sufficiently covered in the subsection of this post entitled *Variation: Abstract Service Locator*. That's exactly what it describes :)
2013-02-04 15:43 UTC

Bruce Pierson
Interesting. I've changed my service locator to use "reasonable defaults", like this:

--In IIocAdapter:

T TryGet[T](T defaultInstance, bool register = true);

--In AutofacIoc implementation:

```
public T TryGet[T]( T defaultInstance, bool register = true )
{
var instance = TryGet[T]();
if( null == instance )
{
instance = defaultInstance;
if( null != instance && register )
{
ContainerBuilder builder = new ContainerBuilder();
builder.RegisterType( instance.GetType() ).As( typeof( T ) );
builder.Update( m_Service );
}
}

return instance;
}
```

--And in the static Ioc "Service locator":

```
public static T TryGet[T]( T defaultInstance, bool register = true )
{
if( IsInitialized() )
return Adapter.TryGet[T]( defaultInstance, register );

return defaultInstance;
}
```

--From code:

```
public virtual IRuleContainer GetRuleContainer()
{
if( null == m_RuleContainer )
{
lock( LOCK )
{
if( null == m_RuleContainer )
```

m_RuleContainer = Ioc.TryGet[IRuleContainer](new DefaultRuleContainer());
}
}

return m_RuleContainer;
}

Very flexible. I can use IoC or inheritance.

I love it. It lets me create good default behavior that can be very simply replaced. It also makes the principle of "prefer containment over inheritance" much easier to achieve. I guess sometimes we need to use "anti-patterns" to enhance the implementation of other patterns.
2013-02-07 20:03 UTC

Mansoor Omrani
I got the point and I am convinced. :)

So I conclude it this way:

Inability to report on compile-time is one other deficiency to Service Locator pattern among its other deficiencies.

While my suggestion may ease finding the requirements of the holder class a little, easier than a try-and-error approach such as using a try/catch, but the need to "execute" the app and obtaining the information at runtime is never removable.

Perhaps the only benefit for adding query API to a service locator is in those scenarios where you are extending an app whose structure, libraries and API is already fixed (perhaps worked by a previous team) and you can't modify those classes and method signatures.

Thank you for time on reviewing my code.
2013-02-08 04:29 UTC

Buddy James (http://www.refactorthis.net)
@Mark I've read the entire article, and I don't think that the subsection that you specify does cover my question. First and foremost, I'm 100% with why you would not want to use the service locator anti-pattern. I agree that it's bad and shouldn't be used. My question is regarding Prism. In the Prism library, the Unity container is passed around as modules initialize.. So that way each module can be decoupled from the other modules, and they can each register their required dependencies. This process *most of the time* occurs at the composition root and you proceed to use Resolve only once to resolve your start up Window (for WPF) or whatever you are starting with. So at the point of resolve.. you call it once.. you call it at the composition root.. and the one call to resolve should resolve your entire "tree of dependencies" or graph so to speak.. This part is correct.. the part that is incorrect..

or rather, I'm fickled about, is the fact that you are passing 1 container around to multiple modules that have no knowledge of what the others may.. or may not have registered.. So with Unity.. this makes it possible for your to register a dependency, only to have the dependency overwritten by a module that someone else wrote that you aren't aware of. This may not happen frequently, however, it's still possible and that's where I have a problem :) . I'm wondering what would be some better alternatives to allow multiple modules developed by different developers to register dependencies and to then resolve all dependencies at the app start up or "composition root", without passing one unity container around. You could have a container for each module.. however.. you wouldn't have a single resolve that "unravels" your entire object graph. Any ideas anyone?? Thanks for the response!
2013-02-15 08:40 UTC

Mark Seemann (/)
Buddy, thanks for clarifying. While I've never looked at Prism, I think I understand how it works from what you've described. It's not that uncommon a 'pattern', but I can't particularly endorse it, as I think that it tightly couples a framework to a particular container (in this case Unity), and in any case there are better, less invasive alternatives.

The way MEF works comes to mind, but you can do the same as MEF does with convention-based wiring with any DI Container worth its salt. However, I think I need to do a write-up of this one day...

The fundamental thing to be aware of, however, is that if you are building an application with a true add-in architecture, you'll need to define your interfaces in such a way that any dependency can be implemented by zero to many classes. The cardinality of dependencies in an add-in architecture is never 1, it's always 0+. Doing this elegantly addresses the issue of 'what if one module overwrites the other?' They mustn't be allowed to do that. All modules have equal priority, and so must their dependencies. This also makes a lot of sense if you think about what an add-in is: usually, it's just a file you drop in a folder somewhere, so there's no explicit order or priority implied.
2013-02-15 09:53 UTC

Buddy James (http://www.refactorthis.net)
Mark, Thanks for the input. Just to be clear, Prism also supports a MEF bootstrapper to load the modules as well so you aren't forced to use Unity.. it's just a popular option. Anyway, it's been a pleasure and I loved your book. I'd be honored if you would stop by my blog and check out my articles on Unity and DI . it's refactorthis.net Thanks again!
2013-02-16 05:23 UTC

Danyil
Hi, Mark! What do you think of the Context IoC (http://sonymathew.blogspot.nl/2009/11/context-ioc-revisited-i-wrote-about.html) pattern? Would you consider it distinct from the Service Locator?
2015-10-07 14:54 UTC

Mark Seemann (/)

Danyil, thank you for writing. Context IoC, as described in that article, isn't service location (/2010/11/01/PatternRecognitionAbstractFactoryorServiceLocator), because each context injected is only a limited, well-known interface. A Service Locator, on the other hand, exposes an infinite set of services (/2014/05/15/service-locator-violates-solid).

That said, how does Context IoC solve anything that Pure DI (/2014/06/10/pure-di) doesn't solve in a simpler way?

2015-10-07 15:37 UTC

Danyil
To give my question above some context, I've run into online discussion threads where the participants equated the two, which I thought unfairly ignored the Context IoC pattern's better static guarantees. I am glad to hear you disagree with them on the matter of classification.

As for Pure DI, I will have to investigate it further. Thanks for your reply!

2015-10-07 16:55 UTC

Mark Seemann (/)
Danyil, thank you for writing. That Pure DI link may not give you the most succinct picture of what I had in mind. Using the blue/red example from the Context IoC article, you can rewrite MyService much simpler using Constructor Injection:

```
public class MyService : ISomeService
{
    private readonly ICommonService blue;
    private readonly ICommonService red;

    public MyService(ICommonService blue, ICommonService red)
    {
        if (blue == null)
            throw new ArgumentNullException(nameof(blue));
        if (red == null)
            throw new ArgumentNullException(nameof(red));

        this.blue = blue;
        this.red = red;
    }

    // Members that DO something can go here...
}
```

If you want to add a *green* dependency to BlueService, you can do that in the same manner:

```
public class BlueService : ICommonService
{
    private readonly ICommonService green;

    public BlueService(ICommonService green)
    {
        this.green = green;
    }

    // Members that DO something can go here...
}
```

You can compose desired object graphs (/2011/03/04/Composeobjectgraphswithconfidence) in your Composition Root (/2011/07/28/CompositionRoot):

```
public ISomeService Main()
{
    return new MyService(
        new BlueService(
            new GreenService()),
        new RedService());
}
```

Notice how the shape of the object graph is visibly present due to the (standard) indentation. That makes it easy to gauge the depth and complexity of most object graphs.

Does ContextIoC solve anything that this doesn't address in a simpler, less convoluted way with fewer moving parts?

2015-10-08 12:06 UTC

Published: Wednesday, 03 February 2010 21:49:39 UTC