

Paul Hammant's guide to killing singletons and heading to Dependency Injection, by way of Service Locator.

v2.0: March 2013.

Note: This is a refresh of an article I did for InfoQ in April of 2008 – <http://www.infoq.com/articles/drinking-your-guice-too-quickly> which was a write up of part of 2005 mission since detailed in <http://paulhammant.com/2013/03/11/legacy-app-rejuvenation>

We are at the 9 year mark for “Dependency Injection” the term and the technologies and practice is about a 10/11 years old now. formal Inversion of Control (IoC) is now 15 years on or so.

Overview

This document outlines how to move to Dependency Injection (DI) from a nest-of-singletons starting point.

This can prove to be difficult. Attacking one singleton at a time, without dragging in most of the source-base is highly desirable, but hard. This following approach outlines a safe and methodical technique, that allows for the ‘hair-ball’ to be attacked in a number of smaller commits.

Singletons happen

Singletons and static mutable state have been accused of being problematic for many years.

There is no doubt that an application built according to the principals of DI is a more testable and a cleaner architecture than the sprawling singleton hair-ball.

It is often the case, though, under time pressure, that simple

Singletons make it easy to bridge large gaps between components. What starts with one or two singletons, can end up being tens or even hundreds, until the entire team declares that the code is unmaintainable. The boss is not about to commission a rewrite, but likes the promise of DI, so it seems pragmatic to refactor the current codebase towards it.



*A metaphor for
entanglement*

Once you have decided to change direction towards that DI nirvana, you have to know which path to take. Sure, you could stop feature development and bug fixing and just go for it, but that is not the right way. Instead, you would want to deliver new functionality **at the same time**, and experience tells us that a promise to put in The Spring Framework (Spring henceforth) in a week or two ends up being many months later. Even if the DI work is performed on a branch that is divergent to the one where features and bug fixes are still being coded, there is a risk of merge-hell when the DI refactor is finished.

It is also not clear where to start. Do you put a DI container in 'main' method (if it is pertinent to your app) with no registered components yet and commit that change first? Or do you start at the web-tier and move towards DI there? Also does the DI container get populated with the results of singleton lookups initially, with 'TODO' comments promising a revisit later? Either of these can be messy and unsavory.

'Service Locator' as a stepping stone to Dependency Injection

Martin Fowler wrote the [definitive article on Dependency Injection](#) in 2003 (Paul gets a credit in that article). The formal field for DI was young at the time, and Martin discussed Service Locator as a worthy alternative choice. Of course, it means different things to different people, but for now assume it means a single class (that is a singleton itself) with a method on it like so...

```
public Object getService(String serviceName) {  
    // etc  
}
```

Or...

```
// yay generics!  
public T getService(Class serviceType) {  
    // etc  
}
```

The idea is that in a boot-like place (the main method?) it is populated with appropriate instances for the service names/classes, instances. In a build phase, unit tests can alternatively program the service locator with a mix of real and mock instances making for streamlined setup of tests. Wherever there are singleton lookups in the legacy codebase, a small change is made to lookup the same component via the service locator instead.

```
ZipCodeService z = ZipCodeService.getInstance();
```

Becomes...

```
ZipCodeService z = ServiceLocator.getInstance()  
    .getService(ZipCodeService.class);
```

This is only good for 'application scoped' services/components. Modern web frameworks have session and request-scoped components and handle the dependency injection for them as those scopes come into life. If you are heading towards DI **today** in an existing codebase, then it is likely that you do not yet have one of the modern web frameworks in the application, so live with what

you have for now. Specifically, try to this DI work on it's own, without perhaps concurrently tackling a retooling to a new web framework.

Back to specifics.

The service locator needs to be populated in the primordial boot place...

```
public void setService(String serviceName,
                       Object implementation) {
    // etc
}
```

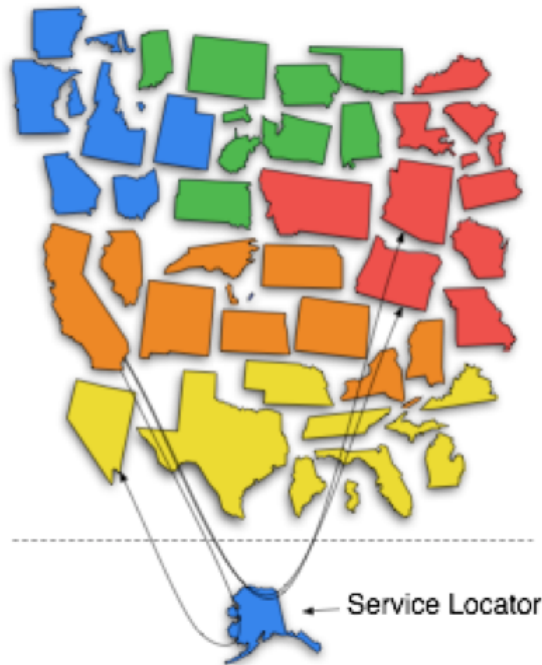
To make this safe, you may want to have a mechanism to lock the service locator and make it read only after that moment. That lock() method would be called at the end of the main method before any "start" lifecycle were invoked and would very effectively prevent misuse.


Putting in your service locator is the just the start. What follows is a series of small commits where you are replacing one singleton's Xyz.getInstance() method with getService(Xyz.class) on the service locator instance. In effect these singletons have now become 'managed single instances'. You might find it a good time to interface/impl separate the component in question. One reason you might decide to do that is to facilitate mocking (Mockito is the best library for that) because you obviously want to simultaneously increase the test coverage for the application.

When you have exhausted the list of Singletons to eliminate, you can revisit the code fragments that use the service locator. It might be that a getService() invocation is done wherever the component/service was needed in a class. It may even be done multiple times in the same class (garbage collected each time when de-scoped). In that case, doing the getService() invocation once in a constructor and storing the result as a member variable would be smart. As mentioned, that could/should be a follow up. These are refactoring operations and should be safe.

Following that you should ripple through the codebase again, moving from the in-constructor getService() invocation and

assignment of member variable, to injection via the constructor arguments. Thus the service locator lookup moves to the class & method that would instantiate the class. If you do that again and again, you'll move all the service locator lookups to towards a primordial place. We'll come back to that later.



 *Our metaphor for entanglement benefiting from an initial use of service locator..*

California (a component responsible for agriculture and high-tech) needs Nevada to provide gaming machine functionality. Oregon (Hazelnuts) and Arizona (chemicals). All via service locator now, and much more visible because of that.

As mentioned, it is desirable to do the changes in a series of small commits. Commits that will be easy for other developers to merge in to their working copy. Those commits should happen in addition to a team's other commitments for an iteration (functional improvements and bug fixes etc). The aim is to minimize the chance of introducing defects, by being methodical. You should go even further towards lowering that risk, by taking the opportunity to add small unit tests (and appropriate mocking; queue 2nd plug of Mockito) to these newly separated components.

Least Depending, Most Depended on first

This is a critical piece...

The first component to move towards the service locator design, and away from its singleton origins, is the one that depends on no

other singletons, yet may be depended by other singletons. Of course it can be depended on by any amount of non-singletons.

It's the lowest hanging fruit – the least depending and most depended on.

It is also going to be the one that is easiest to get high coverage for with small unit tests. Perhaps this means some work with your favorite mocking library. As you process one and commit, another will qualify as "least depending, most depended on".

While at Google I outlined a “Singleton Detector” that David Rubel wrote: <http://googlecode.blogspot.com/2007/07/google-singleton-detector-released.html>

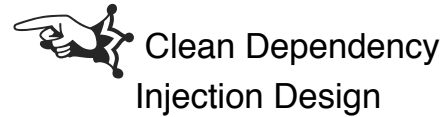
It will represent in graph form, which classes are singletons, and which classes use them. It's a visual guide to where the least depending, most dependent singletons are. It's a bit tricky to use, but worth it.

Spring after a short delay

Now that you have an application that is comprised of many components accessed via a single service locator, it is time to put in Spring and start moving components out of the Service Locator and into the XML context, or Java compositional logic.

As mentioned, the most methodical way of doing this is to find one of the service locator lookups in a constructor and push it to the class instantiating it. Change it to a constructor argument at the same time, and make the caller have a member variable for that same dependency. If you keep pushing them up, sooner or later your going to get them to the main method. At that time they can safely become Spring managed.

When the DI container manages everything, the service locator can be thanked for its good work and deleted. The unit tests, will directly instantiate the class being tested, and directly inject mocks as applicable.



Some side effects are going to be long argument lists for some of the classes/components. These are likely to be clues that the design for the application could do with some work. Making a facade at that point can often be the right thing.

Many companies have had some success with this methodical approach. It works for situations where there are hundred of components with which started out as a nest of singletons, and are now lightweight Dependency Injection. It works too if EJB 3.0 (or above) is your destination. It is also the experience that roll-out can happen concurrently with normal coding, with no code-freeze to facilitate merges at all.

Footnote

Dependency Injection is just one part of Inversion of Control (now fourteen years old as a pattern/practice). The other two aspects are configuration and lifecycle. The implication is that classes should be given their configuration (more injection) and lifecycle state-changes similarly controlled from outside. They should not get their own configuration, spawn threads or listen on sockets. Get/spawn/listen was tempting to do in their constructors, or worse still static initializers, but those should disappear as you start to do Dependency Injection and Inversion of Control properly.

Further Reading

Foote/Yoder used a ball of mud metaphor previously for general entanglement: <http://www.laputan.org/mud>. I'm fond of "hairball" as you can see above.

Martin's 2004 article on Dependency Injection is still where you'd start your journey to understanding it: <http://martinfowler.com/articles/injection.html>