



English



French ▼

[Refactorisation](#)[Agile](#)[Architecture](#)[À propos de](#)[Œuvres de pensée](#)

Inversion des conteneurs de contrôle et du modèle d'injection de dépendance

Dans la communauté Java, il y a eu une ruée vers des conteneurs légers qui aident à assembler des composants de différents projets dans une application cohérente. Sous-jacent à ces conteneurs se trouve un modèle commun à la façon dont ils effectuent le câblage, un concept auquel ils se réfèrent sous le nom très générique de "Inversion de contrôle". Dans cet article, j'examine le fonctionnement de ce modèle, sous le nom plus spécifique de "injection de dépendance", et je le compare à l'alternative Service Locator. Le choix entre eux est moins important que le principe de séparation de la configuration de l'utilisation.

23 janvier 2004



Martin Fowler

POPULAIRE

MOTIF

CONCEPTION DE COLLABORATION D'OBJETS

ARCHITECTURE DES APPLICATIONS

TRADUCTIONS : chinois

Table des matières

CONTENU

Composants et services

Un exemple naïf

Inversion de contrôle

Formes d'injection de dépendance

Injection de constructeur avec PicoContainer

Setter Injection avec ressort

Injection d'interface

Utilisation d'un localisateur de service

Utilisation d'une interface séparée pour le localisateur

Un localisateur de service dynamique

Utilisation à la fois d'un localisateur et d'une injection avec Avalon

Décider quelle option utiliser

Localisateur de service vs injection de dépendance

Constructeur contre Setter Injection

Fichiers de code ou de configuration

Séparer la configuration de l'utilisation

Quelques problèmes supplémentaires

Pensées finales

L'une des choses les plus divertissantes dans le monde Java d'entreprise est l'énorme quantité d'activités dans la construction d'alternatives aux technologies J2EE traditionnelles, la plupart se déroulant en open source. Une grande partie de cela est une réaction à la lourde complexité du monde J2EE grand public, mais une grande partie consiste également à explorer des alternatives et à proposer des idées créatives. Un problème courant à résoudre est de savoir comment connecter différents éléments : comment assembler cette architecture de contrôleur Web avec cette interface de base de données lorsqu'ils ont été construits par différentes équipes avec peu de connaissances les unes sur les autres. Un certain nombre de frameworks ont tenté de résoudre ce problème, et plusieurs se diversifient pour fournir une capacité générale d'assemblage de composants à partir de différentes couches. Ceux-ci sont souvent appelés conteneurs légers, par exemple PicoContainer et Spring .

Ces conteneurs reposent sur un certain nombre de principes de conception intéressants, des choses qui vont au-delà de ces conteneurs

spécifiques et même de la plate-forme Java. Ici, je veux commencer à explorer certains de ces principes. Les exemples que j'utilise sont en Java, mais comme la plupart de mes écrits, les principes sont également applicables à d'autres environnements OO, en particulier .NET.

Composants et services

Le sujet du câblage des éléments m'entraîne presque immédiatement dans les problèmes terminologiques épineux qui entourent les termes service et composant. Vous trouverez facilement des articles longs et contradictoires sur la définition de ces choses. Pour mes besoins, voici mes utilisations actuelles de ces termes surchargés.

J'utilise un composant pour désigner un ensemble de logiciels destinés à être utilisés, sans changement, par une application hors du contrôle des auteurs du composant. Par "sans changement", je veux dire que l'application utilisatrice ne modifie pas le code source des composants, bien qu'elle puisse modifier le comportement du composant en l'étendant de la manière autorisée par les auteurs du composant.

Un service est similaire à un composant en ce sens qu'il est utilisé par des applications étrangères. La principale différence est que je m'attends à ce qu'un composant soit utilisé localement (pensez à un fichier jar, un assemblage, une dll ou une importation de source). Un service sera utilisé à distance via une interface distante, synchrone ou asynchrone (par exemple, service Web, système de messagerie, RPC ou socket.)

J'utilise principalement le service dans cet article, mais une grande partie de la même logique peut également être appliquée aux composants locaux. En effet, vous avez souvent besoin d'une sorte de framework de

composants locaux pour accéder facilement à un service distant. Mais écrire "composant ou service" est fatigant à lire et à écrire, et les services sont beaucoup plus à la mode en ce moment.

Un exemple naïf

Pour aider à rendre tout cela plus concret, je vais utiliser un exemple courant pour parler de tout cela. Comme tous mes exemples, c'est l'un de ces exemples super simples; assez petit pour être irréal, mais suffisamment, espérons-le, pour que vous puissiez visualiser ce qui se passe sans tomber dans le marais d'un exemple réel.

Dans cet exemple, j'écris un composant qui fournit une liste de films réalisés par un réalisateur particulier. Cette fonction incroyablement utile est implémentée par une seule méthode.

classe MovieLister...

```
public Movie[] moviesDirectedBy(String arg) {  
    List allMovies = finder.findAll();  
    pour (Iterator it = allMovies.iterator(); it.hasNext();) {  
        Film film = (Film) it.next();  
        if (!movie.getDirector().equals(arg)) it.remove();  
    }  
    return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);  
}
```

L'implémentation de cette fonction est naïve à l'extrême, elle demande à un objet finder (auquel nous reviendrons dans un instant) de renvoyer chaque film dont il a connaissance. Ensuite, il chasse simplement dans cette liste pour renvoyer ceux dirigés par un réalisateur particulier. Ce

morceau particulier de naïveté que je ne vais pas réparer, car ce n'est que l'échafaudage pour le vrai point de cet article.

Le vrai point de cet article est cet objet `finder`, ou en particulier comment nous connectons l'objet `lister` avec un objet `finder` particulier. La raison pour laquelle c'est intéressant est que je veux que ma merveilleuse `moviesDirectedBy` méthode soit complètement indépendante de la façon dont tous les films sont stockés. Donc, tout ce que fait la méthode, c'est se référer à un chercheur, et tout ce que fait ce chercheur, c'est savoir comment répondre à la `findAll` méthode. Je peux faire ressortir cela en définissant une interface pour le `finder`.

```
interface publique MovieFinder {  
    Liste trouveTout();  
}
```

Maintenant, tout cela est très bien découplé, mais à un moment donné, je dois proposer une classe concrète pour réellement proposer les films. Dans ce cas, j'ai mis le code correspondant dans le constructeur de ma classe `Lister`.

```
classe MovieLister...  
    chercheur privé de MovieFinder ;  
    public MovieLister() {  
        finder = new ColonDelimitedMovieFinder("movies1.txt");  
    }
```

Le nom de la classe d'implémentation vient du fait que je reçois ma liste à partir d'un fichier texte délimité par deux-points. Je vais vous épargner les détails, après tout, le point est juste qu'il y a une mise en œuvre.

Maintenant, si j'utilise cette classe pour moi-même, tout va bien. Mais que se passe-t-il lorsque mes amis sont submergés par le désir de cette merveilleuse fonctionnalité et souhaitent une copie de mon programme ? S'ils stockent également leurs listes de films dans un fichier texte délimité par deux-points appelé "movies1.txt", alors tout est merveilleux.

S'ils ont un nom différent pour leur fichier vidéo, il est facile de mettre le nom du fichier dans un fichier de propriétés. Mais que se passe-t-il s'ils ont une forme complètement différente de stockage de leur liste de films : une base de données SQL, un fichier XML, un service Web ou simplement un autre format de fichier texte ? Dans ce cas, nous avons besoin d'une classe différente pour récupérer ces données. Maintenant que j'ai défini une `MovieFinderinterface`, cela ne modifiera pas `monmoviesDirectedBymethode`. Mais j'ai encore besoin d'un moyen de mettre en place une instance de la bonne implémentation du finder.

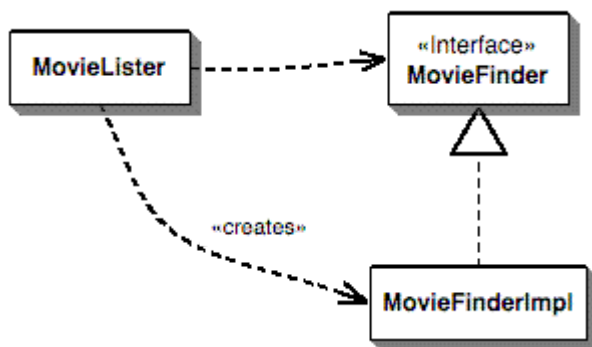


Figure 1 : Les dépendances à l'aide d'une simple création dans la classe lister

La figure 1 montre les dépendances pour cette situation. La `MovieLister` classe dépend à la fois de l' `MovieFinderinterface` et de l'implémentation. Nous préférierions que cela ne dépende que de l'interface, mais alors comment créer une instance avec laquelle travailler ?

Dans mon livre P d'EAA , nous avons décrit cette situation comme un Plugin . La classe d'implémentation du finder n'est pas liée au programme au moment de la compilation, car je ne sais pas ce que mes amis vont utiliser. Au lieu de cela, nous voulons que mon listeur fonctionne avec n'importe quelle implémentation, et que cette implémentation soit branchée ultérieurement, hors de mes mains. Le problème est de savoir comment puis-je faire ce lien pour que ma classe de listeur ignore la classe d'implémentation, mais puisse toujours parler à une instance pour faire son travail.

En développant cela dans un système réel, nous pourrions avoir des dizaines de services et de composants de ce type. Dans chaque cas, nous pouvons résumer notre utilisation de ces composants en leur parlant via une interface (et en utilisant un adaptateur si le composant n'est pas conçu avec une interface à l'esprit). Mais si nous souhaitons déployer ce système de différentes manières, nous devons utiliser des plugins pour gérer l'interaction avec ces services afin que nous puissions utiliser différentes implémentations dans différents déploiements.

Le problème principal est donc de savoir comment assembler ces plugins dans une application ? C'est l'un des principaux problèmes auxquels cette nouvelle génération de conteneurs légers est confrontée, et universellement, ils le font tous en utilisant l'inversion de contrôle.



Inversion de contrôle

Lorsque ces conteneurs expliquent à quel point ils sont si utiles parce qu'ils implémentent "l'inversion du contrôle", je finis par être très perplexe. L'inversion de contrôle est une caractéristique commune des cadres, donc dire que ces conteneurs légers sont spéciaux parce qu'ils utilisent l'inversion de contrôle revient à dire que ma voiture est spéciale parce qu'elle a des roues.

La question est : "quel aspect du contrôle inversent-ils ?" Lorsque j'ai rencontré l'inversion de contrôle pour la première fois, c'était dans le contrôle principal d'une interface utilisateur. Les premières interfaces utilisateur étaient contrôlées par le programme d'application. Vous auriez une séquence de commandes comme "Entrez le nom", "Entrez l'adresse" ; votre programme piloterait les invites et répondrait à chacune d'entre

elles. Avec des interfaces utilisateur graphiques (ou même basées sur écran), le cadre de l'interface utilisateur contiendrait cette boucle principale et votre programme fournirait à la place des gestionnaires d'événements pour les différents champs à l'écran. Le contrôle principal du programme a été inversé, éloigné de vous vers le cadre.

Pour cette nouvelle race de conteneurs, l'inversion concerne la façon dont ils recherchent une implémentation de plugin. Dans mon exemple naïf, le listeur a recherché l'implémentation du finder en l'instanciant directement. Cela empêche le finder d'être un plugin. L'approche que ces conteneurs utilisent est de s'assurer que tout utilisateur d'un plugin suit une convention qui permet à un module assembleur séparé d'injecter l'implémentation dans le lister.

En conséquence, je pense que nous avons besoin d'un nom plus spécifique pour ce modèle. L'inversion du contrôle est un terme trop générique, et donc les gens le trouvent déroutant. En conséquence, après de nombreuses discussions avec divers défenseurs d'IoC, nous avons opté pour le nom *Dependency Injection*.

Je vais commencer par parler des différentes formes d'injection de dépendances, mais je soulignerai maintenant que ce n'est pas le seul moyen de supprimer la dépendance de la classe d'application à l'implémentation du plugin. L'autre modèle que vous pouvez utiliser pour ce faire est Service Locator, et j'en discuterai une fois que j'aurai fini d'expliquer l'injection de dépendance.



Formes d'injection de dépendance

L'idée de base de l'injection de dépendance est d'avoir un objet séparé, un assembleur, qui remplit un champ dans la classe lister avec une implémentation appropriée pour l'interface de recherche, ce qui donne un diagramme de dépendance dans le sens de la figure 2.

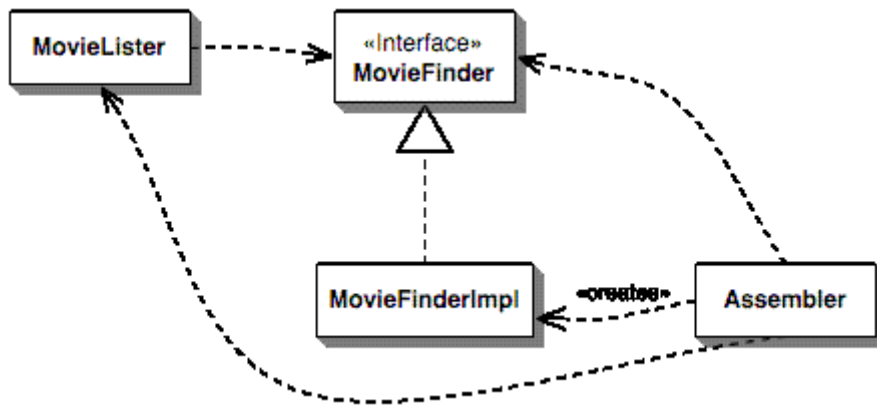


Figure 2 : Les dépendances d'un injecteur de dépendances

There are three main styles of dependency injection. The names I'm using for them are Constructor Injection, Setter Injection, and Interface Injection. If you read about this stuff in the current discussions about Inversion of Control you'll hear these referred to as type 1 IoC (interface injection), type 2 IoC (setter injection) and type 3 IoC (constructor injection). I find numeric names rather hard to remember, which is why I've used the names I have here.

Constructor Injection with PicoContainer

I'll start with showing how this injection is done using a lightweight container called PicoContainer. I'm starting here primarily because several of my colleagues at Thoughtworks are very active in the development of PicoContainer (yes, it's a sort of corporate nepotism.)

PicoContainer uses a constructor to decide how to inject a finder implementation into the lister class. For this to work, the movie lister class needs to declare a constructor that includes everything it needs injected.

```
class MovieLister...
```

```
public MovieLister(MovieFinder finder) {  
    this.finder = finder;  
}
```

The finder itself will also be managed by the pico container, and as such will have the filename of the text file injected into it by the container.

```
class ColonMovieFinder...  
public ColonMovieFinder(String filename) {  
    this.filename = filename;  
}
```

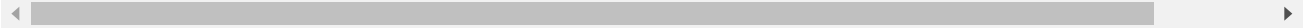
The pico container then needs to be told which implementation class to associate with each interface, and which string to inject into the finder.

```
private MutablePicoContainer configureContainer() {  
    MutablePicoContainer pico = new DefaultPicoContainer();  
    Parameter[] finderParams = {new ConstantParameter("movies1.txt")};  
    pico.registerComponentImplementation(MovieFinder.class, ColonMovieFinder.class);  
    pico.registerComponentImplementation(MovieLister.class);  
    return pico;  
}
```

This configuration code is typically set up in a different class. For our example, each friend who uses my lister might write the appropriate configuration code in some setup class of their own. Of course it's common to hold this kind of configuration information in separate config files. You can write a class to read a config file and set up the container appropriately. Although PicoContainer doesn't contain this functionality itself, there is a closely related project called NanoContainer that provides the appropriate wrappers to allow you to have XML configuration files. Such a nano container will parse the XML and then configure an underlying pico container. The philosophy of the project is to separate the config file format from the underlying mechanism.

To use the container you write code something like this.

```
public void testWithPico() {  
    MutablePicoContainer pico = configureContainer();  
    MovieLister lister = (MovieLister) pico.getComponentInstance(MovieLister  
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");  
    assertEquals("Once Upon a Time in the West", movies[0].getTitle());  
}
```



Although in this example I've used constructor injection, PicoContainer also supports setter injection, although its developers do prefer constructor injection.

Setter Injection with Spring

The Spring framework is a wide ranging framework for enterprise Java development. It includes abstraction layers for transactions, persistence frameworks, web application development and JDBC. Like PicoContainer it supports both constructor and setter injection, but its developers tend to prefer setter injection - which makes it an appropriate choice for this example.

To get my movie lister to accept the injection I define a setting method for that service

```
class MovieLister...  
    private MovieFinder finder;  
    public void setFinder(MovieFinder finder) {  
        this.finder = finder;  
    }
```

Similarly I define a setter for the filename.

```
class ColonMovieFinder...  
    public void setFilename(String filename) {  
        this.filename = filename;  
    }
```

The third step is to set up the configuration for the files. Spring supports configuration through XML files and also through code, but XML is the expected way to do it.

```
<beans>
  <bean id="MovieLister" class="spring.MovieLister">
    <property name="finder">
      <ref local="MovieFinder"/>
    </property>
  </bean>
  <bean id="MovieFinder" class="spring.ColonMovieFinder">
    <property name="filename">
      <value>movies1.txt</value>
    </property>
  </bean>
</beans>
```

The test then looks like this.

```
public void testWithSpring() throws Exception {
  ApplicationContext ctx = new FileSystemXmlApplicationContext("spring.xml");
  MovieLister lister = (MovieLister) ctx.getBean("MovieLister");
  Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
  assertEquals("Once Upon a Time in the West", movies[0].getTitle());
}
```



Interface Injection

The third injection technique is to define and use interfaces for the injection. Avalon is an example of a framework that uses this technique in places. I'll talk a bit more about that later, but in this case I'm going to use it with some simple sample code.

With this technique I begin by defining an interface that I'll use to perform the injection through. Here's the interface for injecting a movie finder into an object.

```
public interface InjectFinder {  
    void injectFinder(MovieFinder finder);  
}
```

This interface would be defined by whoever provides the MovieFinder interface. It needs to be implemented by any class that wants to use a finder, such as the lister.

```
class MovieLister implements InjectFinder  
public void injectFinder(MovieFinder finder) {  
    this.finder = finder;  
}
```

I use a similar approach to inject the filename into the finder implementation.

```
public interface InjectFinderFilename {  
    void injectFilename (String filename);  
}
```

```
class ColonMovieFinder implements MovieFinder, InjectFinderFilename...  
public void injectFilename(String filename) {  
    this.filename = filename;  
}
```

Then, as usual, I need some configuration code to wire up the implementations. For simplicity's sake I'll do it in code.

```
class Tester...  
private Container container;  
  
private void configureContainer() {  
    container = new Container();  
    registerComponents();  
    registerInjectors();  
    container.start();  
}
```

This configuration has two stages, registering components through lookup keys is pretty similar to the other examples.

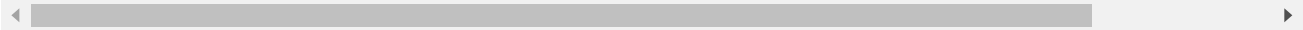
class Tester...

```
private void registerComponents() {  
    container.registerComponent("MovieLister", MovieLister.class);  
    container.registerComponent("MovieFinder", ColonMovieFinder.class);  
}
```

A new step is to register the injectors that will inject the dependent components. Each injection interface needs some code to inject the dependent object. Here I do this by registering injector objects with the container. Each injector object implements the injector interface.

class Tester...

```
private void registerInjectors() {  
    container.registerInjector(InjectFinder.class, container.lookup("MovieFi  
    container.registerInjector(InjectFinderFilename.class, new FinderFilenam  
}
```



```
public interface Injector {  
    public void inject(Object target);  
  
}
```

When the dependent is a class written for this container, it makes sense for the component to implement the injector interface itself, as I do here with the movie finder. For generic classes, such as the string, I use an inner class within the configuration code.

class ColonMovieFinder implements Injector...

```
public void inject(Object target) {  
    ((InjectFinder) target).injectFinder(this);  
}
```

class Tester...


```
public static class FinderFilenameInjector implements Injector {  
    public void inject(Object target) {  
        ((InjectFinderFilename)target).injectFilename("movies1.txt");  
    }  
}
```

The tests then use the container.

class Tester...

```
public void testIface() {  
    configureContainer();  
    MovieLister lister = (MovieLister)container.lookup("MovieLister");  
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");  
    assertEquals("Once Upon a Time in the West", movies[0].getTitle());  
}
```

The container uses the declared injection interfaces to figure out the dependencies and the injectors to inject the correct dependents. (The specific container implementation I did here isn't important to the technique, and I won't show it because you'd only laugh.)



Using a Service Locator

The key benefit of a Dependency Injector is that it removes the dependency that the `MovieLister` class has on the concrete `MovieFinder` implementation. This allows me to give listers to friends and for them to plug in a suitable implementation for their own environment. Injection isn't the only way to break this dependency, another is to use a service locator.

The basic idea behind a service locator is to have an object that knows how to get hold of all of the services that an application might need. So a service locator for this application would have a method that returns a movie finder when one is needed. Of course this just shifts the burden a tad, we still have to get the locator into the lister, resulting in the dependencies of Figure 3

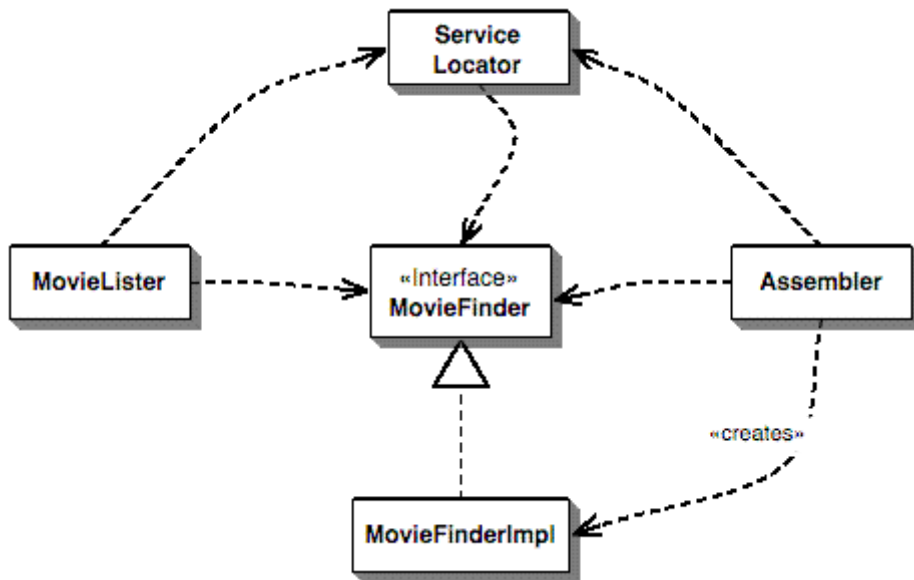


Figure 3: The dependencies for a Service Locator

In this case I'll use the ServiceLocator as a singleton Registry. The lister can then use that to get the finder when it's instantiated.

```
class MovieLister...
```

```
    MovieFinder finder = ServiceLocator.movieFinder();
```

```
class ServiceLocator...
```

```
    public static MovieFinder movieFinder() {
        return soleInstance.movieFinder;
    }
    private static ServiceLocator soleInstance;
    private MovieFinder movieFinder;
```

As with the injection approach, we have to configure the service locator. Here I'm doing it in code, but it's not hard to use a mechanism that would read the appropriate data from a configuration file.

```
class Tester...
```

```
private void configure() {  
    ServiceLocator.load(new ServiceLocator(new ColonMovieFinder("movies1.t  
}
```



```
class ServiceLocator...
```

```
public static void load(ServiceLocator arg) {  
    soleInstance = arg;  
}  
  
public ServiceLocator(MovieFinder movieFinder) {  
    this.movieFinder = movieFinder;  
}
```

Here's the test code.

```
class Tester...
```

```
public void testSimple() {  
    configure();  
    MovieLister lister = new MovieLister();  
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");  
    assertEquals("Once Upon a Time in the West", movies[0].getTitle());  
}
```

I've often heard the complaint that these kinds of service locators are a bad thing because they aren't testable because you can't substitute implementations for them. Certainly you can design them badly to get into this kind of trouble, but you don't have to. In this case the service locator instance is just a simple data holder. I can easily create the locator with test implementations of my services.

For a more sophisticated locator I can subclass service locator and pass that subclass into the registry's class variable. I can change the static methods to call a method on the instance rather than accessing instance variables directly. I can provide thread-specific locators by using thread-specific storage. All of this can be done without changing clients of service locator.

A way to think of this is that service locator is a registry not a singleton. A singleton provides a simple way of implementing a registry, but that implementation decision is easily changed.

Using a Segregated Interface for the Locator

One of the issues with the simple approach above, is that the MovieLister is dependent on the full service locator class, even though it only uses one service. We can reduce this by using a role interface. That way, instead of using the full service locator interface, the lister can declare just the bit of interface it needs.

In this situation the provider of the lister would also provide a locator interface which it needs to get hold of the finder.

```
public interface MovieFinderLocator {  
    public MovieFinder movieFinder();  
}
```

The locator then needs to implement this interface to provide access to a finder.

```
MovieFinderLocator locator = ServiceLocator.locator();  
MovieFinder finder = locator.movieFinder();
```

```
public static ServiceLocator locator() {  
    return soleInstance;  
}  
  
public MovieFinder movieFinder() {  
    return movieFinder;  
}  
  
private static ServiceLocator soleInstance;  
private MovieFinder movieFinder;
```

You'll notice that since we want to use an interface, we can't just access the services through static methods any more. We have to use the class to get a locator instance and then use that to get what we need.

A Dynamic Service Locator

The above example was static, in that the service locator class has methods for each of the services that you need. This isn't the only way of doing it, you can also make a dynamic service locator that allows you to stash any service you need into it and make your choices at runtime.

In this case, the service locator uses a map instead of fields for each of the services, and provides generic methods to get and load services.

class ServiceLocator...

```
private static ServiceLocator soleInstance;
public static void load(ServiceLocator arg) {
    soleInstance = arg;
}
private Map services = new HashMap();
public static Object getService(String key){
    return soleInstance.services.get(key);
}
public void loadService (String key, Object service) {
    services.put(key, service);
}
```

Configuring involves loading a service with an appropriate key.

class Tester...

```
private void configure() {
    ServiceLocator locator = new ServiceLocator();
    locator.loadService("MovieFinder", new ColonMovieFinder("movies1.txt"));
    ServiceLocator.load(locator);
}
```

I use the service by using the same key string.

class MovieLister...

```
MovieFinder finder = (MovieFinder) ServiceLocator.getService("MovieFinder")
```

On the whole I dislike this approach. Although it's certainly flexible, it's not very explicit. The only way I can find out how to reach a service is through textual keys. I prefer explicit methods because it's easier to find where they are by looking at the interface definitions.

Using both a locator and injection with Avalon

Dependency injection and a service locator aren't necessarily mutually exclusive concepts. A good example of using both together is the Avalon framework. Avalon uses a service locator, but uses injection to tell components where to find the locator.

Berin Loritsch sent me this simple version of my running example using Avalon.

```
public class MyMovieLister implements MovieLister, Serviceable {  
    private MovieFinder finder;  
  
    public void service( ServiceManager manager ) throws ServiceException {  
        finder = (MovieFinder)manager.lookup("finder");  
    }  
}
```

The service method is an example of interface injection, allowing the container to inject a service manager into MyMovieLister. The service manager is an example of a service locator. In this example the lister doesn't store the manager in a field, instead it immediately uses it to lookup the finder, which it does store.



Deciding which option to use

So far I've concentrated on explaining how I see these patterns and their variations. Now I can start talking about their pros and cons to help figure out which ones to use and when.

Service Locator vs Dependency Injection

The fundamental choice is between Service Locator and Dependency Injection. The first point is that both implementations provide the fundamental decoupling that's missing in the naive example - in both cases application code is independent of the concrete implementation of the service interface. The important difference between the two patterns is about how that implementation is provided to the application class. With service locator the application class asks for it explicitly by a message to the locator. With injection there is no explicit request, the service appears in the application class - hence the inversion of control.

Inversion of control is a common feature of frameworks, but it's something that comes at a price. It tends to be hard to understand and leads to problems when you are trying to debug. So on the whole I prefer to avoid it unless I need it. This isn't to say it's a bad thing, just that I think it needs to justify itself over the more straightforward alternative.

The key difference is that with a Service Locator every user of a service has a dependency to the locator. The locator can hide dependencies to other implementations, but you do need to see the locator. So the decision between locator and injector depends on whether that dependency is a problem.

Using dependency injection can help make it easier to see what the component dependencies are. With dependency injector you can just look at the injection mechanism, such as the constructor, and see the dependencies. With the service locator you have to search the source code for calls to the locator. Modern IDEs with a find references feature make this easier, but it's still not as easy as looking at the constructor or setting methods.

A lot of this depends on the nature of the user of the service. If you are building an application with various classes that use a service, then a dependency from the application classes to the locator isn't a big deal. In my example of giving a Movie Lister to my friends, then using a service locator works quite well. All they need to do is to configure the locator to hook in the right service implementations, either through some configuration code or through a configuration file. In this kind of scenario I don't see the injector's inversion as providing anything compelling.

The difference comes if the lister is a component that I'm providing to an application that other people are writing. In this case I don't know much about the APIs of the service locators that my customers are going to use. Each customer might have their own incompatible service locators. I can get around some of this by using the segregated interface. Each customer can write an adapter that matches my interface to their locator, but in any case I still need to see the first locator to lookup my specific interface. And once the adapter appears then the simplicity of the direct connection to a locator is beginning to slip.

Since with an injector you don't have a dependency from a component to the injector, the component cannot obtain further services from the injector once it's been configured.

A common reason people give for preferring dependency injection is that it makes testing easier. The point here is that to do testing, you need to easily replace real service implementations with stubs or mocks.

However there is really no difference here between dependency injection and service locator: both are very amenable to stubbing. I suspect this observation comes from projects where people don't make the effort to ensure that their service locator can be easily substituted. This is where continual testing helps, if you can't easily stub services for testing, then this implies a serious problem with your design.

Of course the testing problem is exacerbated by component environments that are very intrusive, such as Java's EJB framework. My view is that these kinds of frameworks should minimize their impact upon application code, and particularly should not do things that slow down the edit-execute cycle. Using plugins to substitute heavyweight components does a lot to help this process, which is vital for practices such as Test Driven Development.

So the primary issue is for people who are writing code that expects to be used in applications outside of the control of the writer. In these cases even a minimal assumption about a Service Locator is a problem.

Constructor versus Setter Injection

For service combination, you always have to have some convention in order to wire things together. The advantage of injection is primarily that it requires very simple conventions - at least for the constructor and setter injections. You don't have to do anything odd in your component and it's fairly straightforward for an injector to get everything configured.

Interface injection is more invasive since you have to write a lot of interfaces to get things all sorted out. For a small set of interfaces required by the container, such as in Avalon's approach, this isn't too bad. But it's a lot of work for assembling components and dependencies, which is why the current crop of lightweight containers go with setter and constructor injection.

The choice between setter and constructor injection is interesting as it mirrors a more general issue with object-oriented programming - should you fill fields in a constructor or with setters.

My long running default with objects is as much as possible, to create valid objects at construction time. This advice goes back to Kent Beck's Smalltalk Best Practice Patterns: Constructor Method and Constructor Parameter Method. Constructors with parameters give you a clear

statement of what it means to create a valid object in an obvious place. If there's more than one way to do it, create multiple constructors that show the different combinations.

Another advantage with constructor initialization is that it allows you to clearly hide any fields that are immutable by simply not providing a setter. I think this is important - if something shouldn't change then the lack of a setter communicates this very well. If you use setters for initialization, then this can become a pain. (Indeed in these situations I prefer to avoid the usual setting convention, I'd prefer a method like `initFoo`, to stress that it's something you should only do at birth.)

But with any situation there are exceptions. If you have a lot of constructor parameters things can look messy, particularly in languages without keyword parameters. It's true that a long constructor is often a sign of an over-busy object that should be split, but there are cases when that's what you need.

If you have multiple ways to construct a valid object, it can be hard to show this through constructors, since constructors can only vary on the number and type of parameters. This is when Factory Methods come into play, these can use a combination of private constructors and setters to implement their work. The problem with classic Factory Methods for components assembly is that they are usually seen as static methods, and you can't have those on interfaces. You can make a factory class, but then that just becomes another service instance. A factory service is often a good tactic, but you still have to instantiate the factory using one of the techniques here.

Constructors also suffer if you have simple parameters such as strings. With setter injection you can give each setter a name to indicate what the string is supposed to do. With constructors you are just relying on the position, which is harder to follow.

If you have multiple constructors and inheritance, then things can get particularly awkward. In order to initialize everything you have to provide

constructors to forward to each superclass constructor, while also adding you own arguments. This can lead to an even bigger explosion of constructors.

Despite the disadvantages my preference is to start with constructor injection, but be ready to switch to setter injection as soon as the problems I've outlined above start to become a problem.

This issue has led to a lot of debate between the various teams who provide dependency injectors as part of their frameworks. However it seems that most people who build these frameworks have realized that it's important to support both mechanisms, even if there's a preference for one of them.

Code or configuration files

A separate but often conflated issue is whether to use configuration files or code on an API to wire up services. For most applications that are likely to be deployed in many places, a separate configuration file usually makes most sense. Almost all the time this will be an XML file, and this makes sense. However there are cases where it's easier to use program code to do the assembly. One case is where you have a simple application that's not got a lot of deployment variation. In this case a bit of code can be clearer than a separate XML file.

A contrasting case is where the assembly is quite complex, involving conditional steps. Once you start getting close to programming language then XML starts breaking down and it's better to use a real language that has all the syntax to write a clear program. You then write a builder class that does the assembly. If you have distinct builder scenarios you can provide several builder classes and use a simple configuration file to select between them.

I often think that people are over-eager to define configuration files. Often a programming language makes a straightforward and powerful

configuration mechanism. Modern languages can easily compile small assemblers that can be used to assemble plugins for larger systems. If compilation is a pain, then there are scripting languages that can work well also.

It's often said that configuration files shouldn't use a programming language because they need to be edited by non-programmers. But how often is this the case? Do people really expect non-programmers to alter the transaction isolation levels of a complex server-side application? Non-language configuration files work well only to the extent they are simple. If they become complex then it's time to think about using a proper programming language.

One thing we're seeing in the Java world at the moment is a cacophony of configuration files, where every component has its own configuration files which are different to everyone else's. If you use a dozen of these components, you can easily end up with a dozen configuration files to keep in sync.

My advice here is to always provide a way to do all configuration easily with a programmatic interface, and then treat a separate configuration file as an optional feature. You can easily build configuration file handling to use the programmatic interface. If you are writing a component you then leave it up to your user whether to use the programmatic interface, your configuration file format, or to write their own custom configuration file format and tie it into the programmatic interface

Separating Configuration from Use

The important issue in all of this is to ensure that the configuration of services is separated from their use. Indeed this is a fundamental design principle that sits with the separation of interfaces from implementation. It's something we see within an object-oriented program when conditional logic decides which class to instantiate, and then future

evaluations of that conditional are done through polymorphism rather than through duplicated conditional code.

If this separation is useful within a single code base, it's especially vital when you're using foreign elements such as components and services. The first question is whether you wish to defer the choice of implementation class to particular deployments. If so you need to use some implementation of plugin. Once you are using plugins then it's essential that the assembly of the plugins is done separately from the rest of the application so that you can substitute different configurations easily for different deployments. How you achieve this is secondary. This configuration mechanism can either configure a service locator, or use injection to configure objects directly.



Some further issues

In this article, I've concentrated on the basic issues of service configuration using Dependency Injection and Service Locator. There are some more topics that play into this which also deserve attention, but I haven't had time yet to dig into. In particular there is the issue of life-cycle behavior. Some components have distinct life-cycle events: stop and starts for instance. Another issue is the growing interest in using aspect oriented ideas with these containers. Although I haven't considered this material in the article at the moment, I do hope to write more about this either by extending this article or by writing another.

You can find out a lot more about these ideas by looking at the web sites devoted to the lightweight containers. Surfing from the [picocontainer](#)

and spring web sites will lead to you into much more discussion of these issues and a start on some of the further issues.



Concluding Thoughts

The current rush of lightweight containers all have a common underlying pattern to how they do service assembly - the dependency injector pattern. Dependency Injection is a useful alternative to Service Locator. When building application classes the two are roughly equivalent, but I think Service Locator has a slight edge due to its more straightforward behavior. However if you are building classes to be used in multiple applications then Dependency Injection is a better choice.

If you use Dependency Injection there are a number of styles to choose between. I would suggest you follow constructor injection unless you run into one of the specific problems with that approach, in which case switch to setter injection. If you are choosing to build or obtain a container, look for one that supports both constructor and setter injection.

The choice between Service Locator and Dependency Injection is less important than the principle of separating service configuration from the use of services within an application.



Acknowledgments

My sincere thanks to the many people who've helped me with this article. Rod Johnson, Paul Hammant, Joe Walnes, Aslak Hellesøy, Jon Tirsén and Bill Caputo helped me get to grips with these concepts and commented on the early drafts of this article. Berin Loritsch and Hamilton Verissimo de Oliveira provided some very helpful advice on how Avalon fits in. Dave W Smith persisted in asking questions about my initial interface injection configuration code and thus made me confront the fact that it was stupid. Gerry Lowry sent me lots of typo fixes - enough to cross the thanks threshold.

► Significant Revisions



© Martin Fowler | Politique de confidentialité | Divulgations