



Conteneurs d'Inversion de Contrôle et modèle d'Injection de Dépendance

par Martin Fowler (traduit par Arnaud Bely : arnaudbely@free.fr)

Introduction

La communauté Java montre un intérêt grandissant sur le sujet des conteneurs légers, qui aident à assembler des composants de différents projets dans une application cohésive. A l'origine de ces conteneurs on trouve un modèle commun leur permettant d'exécuter le câblage, un concept auquel on se réfère sous le nom très générique d'« Inversion de Contrôle ». Dans cet article j'explore la manière dont ce modèle fonctionne, sous le nom plus spécifique d'« Injection de Dépendance », et je le mets en contraste avec l'alternative du Localisateur de Service. Le choix entre eux est moins important que le principe de séparation entre configuration et utilisation.

Dernière mise à jour significative : 20/01/04

Origine : www.martinfowler.com

Une des choses amusantes que l'on rencontre dans le monde Java pour l'entreprise est la somme énorme d'activités lancées dans la construction d'alternatives au courant dominant des technologies J2EE, la plupart se déroulant en open source. Elles sont généralement associées à une réaction face à la complexité pesante du courant dominant du monde J2EE, mais beaucoup explorent aussi des alternatives et proposent des idées créatrices. Une réponse classique pour s'en sortir consiste à combiner ensemble des éléments différents : comment adaptez-vous cette architecture de contrôleur Web avec ce support d'interface de base de données quand ils ont été construits par des équipes différentes sans relation l'une avec l'autre. Un certain nombre de frameworks ont pris en compte cette problématique et plusieurs fournissent une capacité générale à assembler des composants de couches différentes. Ceux-ci sont souvent présentés comme des conteneurs légers, les exemples incluant PicoContainer et Spring.

A la base de ces conteneurs on trouve un certain nombre de principes de conception intéressants, principes qui vont au-delà de ces conteneurs spécifiques et de la plate-forme Java. Ici je souhaite commencer à explorer certains de ces principes. Les exemples que je donne sont en Java, mais comme pour la plupart de mes publications les principes sont également applicables à d'autres environnements OO, particulièrement .NET.

Conteneurs d'Inversion du Contrôle et le modèle d'Injection de Dépendance.....	1
1. Composants et Services.....	3
2. Un Exemple Naïf	3
3. Inversion du Contrôle.....	5
4. Les formes d'Injection de Dépendance.....	5
Figure 2 : les dépendances pour un Injecteur de Dépendance.....	5
4.1. Injection de Constructeur avec PicoContainer	6
4.2. Injection de Mutateur avec Spring	7
4.3. Injection d'Interface.....	7
5. Utilisation d'un Localisateur de Service.....	9
5.1. Utilisation d'une Interface Isolée pour le Localisateur	11
5.2. Un Localisateur de Service Dynamique.....	11
5.3. Utilisation simultanée d'un localisateur et d'une injection avec Avalon.....	12
6. Décider qu'elle option utiliser.....	12
6.1. Localisateur de Service contre Injection de Dépendance	12
6.2. Injection de Constructeur contre Injection de Mutateur	14
6.3. Code ou fichiers de configuration	15
6.4. Séparer la Configuration de l'Utilisation	15
7. Quelques nouvelles questions	16

8.	Conclusion	16
9.	Remerciements.....	16
10.	Révisions Significatives.....	17

1. Composants et services

Le thème de la communication entre éléments m'entraîne presque immédiatement dans des problèmes de terminologie difficiles qui tournent autour des termes service et composant. On trouve sans problème des articles longs et contradictoires sur leur définition. Pour mes propres buts voici mes définitions du moment de ces termes surchargés.

J'utilise le mot composant pour désigner un module de logiciel qui est destiné à être utilisé, tel quel, par une application qui est hors du contrôle des auteurs du composant. Par « tel quel » je veux dire que l'application utilisatrice ne change pas le code source des composants, bien qu'il soit possible de modifier le comportement du composant en l'étendant par des moyens autorisés par ses auteurs.

Un service est semblable à un composant dans le sens où il est utilisé par des applications étrangères. La différence principale est que je m'attends à ce qu'un composant soit utilisé localement (pensez à un fichier jar, assembly, dll, ou une importation de code source). Un service sera utilisé à distance par une quelconque interface distante, synchrone ou asynchrone (service web, système de messagerie, RPC, ou socket).

J'utilise surtout le mot service dans cet article, mais la même logique peut être appliquée aux composants locaux également. En effet on a souvent besoin d'une sorte de composant local d'un framework pour avoir facilement accès 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 à l'heure actuelle.

2. Un exemple naïf

Pour aider à rendre tout ceci plus réaliste j'utiliserai un exemple concret pour parler de tout cela. Comme tous mes exemples il est très simple ; assez petit pour être irréal, mais suffisant pour vous aider à visualiser ce qui se passe sans tomber dans le marasme d'un exemple réel.

Dans cet exemple j'écris un composant qui fournit une liste de films dirigés par un directeur particulier. Cette fonction éminemment utile est implantée par une seule méthode.

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

L'implantation de cette fonction est naïve à l'extrême, elle nécessite un objet finder (que nous verrons dans un moment) pour renvoyer chaque film connu. Elle parcourt simplement cette liste pour renvoyer ceux dirigés par un directeur particulier. Je ne vais pas m'étendre sur cette partie particulièrement naïve, puisque c'est juste un échafaudage pour amener à l'aspect important de cet article.

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

```
public interface MovieFinder {
    List findAll();
}
```

Maintenant tout ceci est très bien découplé, mais à un certain moment je dois faire appel à une classe concrète pour récupérer les films. Dans ce cas je place le code pour le faire dans le constructeur de ma classe lister.

```
class MovieLister...
    private MovieFinder finder;
    public MovieLister() {
        finder = new ColonDelimitedMovieFinder("movies1.txt");
    }
}
```

Le nom de la classe implantée vient du fait que j'obtiens ma liste par un fichier texte délimité en colonnes. Je vous épargnerai les détails, après tout l'important est simplement qu'il y ait une implantation.

Maintenant si j'utilise cette classe juste pour moi-même, tout cela est bel et bon. Mais que se passe-t-il quand mes amis sont dévorés par l'envie d'utiliser cette merveilleuse fonctionnalité et voudraient une copie de mon programme ? S'ils stockent aussi leur liste de films dans un fichier texte délimité en colonnes nommé "movies1.txt" alors tout se passe bien. S'ils disposent d'un nom différent pour leur fichier de films, alors il est facile de mettre ce nom de fichier dans un fichier de propriétés. Mais que faire s'ils ont une manière complètement différente de stocker leur liste de films : une base de données SQL, un fichier XML, un service web, ou juste un autre format de fichier texte ? Dans ce cas nous avons besoin d'une classe différente pour saisir ces données. Maintenant que j'ai défini une interface `MovieFinder`, cela n'aura pas d'impact sur ma méthode `moviesDirectedBy`. Mais j'ai toujours besoin de disposer d'une certaine façon d'obtenir la bonne implantation du finder en place.

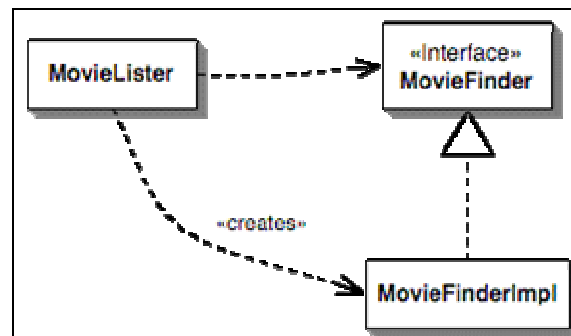


Figure 1 : les dépendances utilisant une simple création dans la classe lister

La figure 1 montre les dépendances de cette situation. La classe `MovieLister` dépend aussi bien de l'interface `MovieFinder` que de l'implantation. Nous préférons qu'elle dépende seulement de l'interface, mais alors comment en créer une instance pour nous en servir ?

Dans mon livre « P of EAA »¹, j'ai décrit cette situation comme un Plugin². La classe implantée pour le finder n'est pas liée dans le programme à la compilation, puisque je ne sais pas ce que mes amis vont utiliser. Au lieu de cela nous voulons que mon lister travaille avec n'importe quelle implantation et pour que cette implantation puisse être pluggé à un moment ultérieur, hors de mon contrôle. Le problème est de savoir comment je puis faire ce lien pour que ma classe lister soit ignorante de la classe d'implantation, mais puisse toujours parler à une instance pour faire son travail.

En étendant cela à un système réel, nous pourrions avoir des douzaines de services et composants de ce style. Dans chaque cas nous pouvons extraire notre utilisation de ces composants en dialoguant avec eux à travers une interface (et utiliser un adaptateur si le composant n'est pas conçu avec une interface en mémoire). Mais si nous voulons déployer ce système de différentes façons, nous devons utiliser des plugins pour manipuler l'interaction avec ces services de manière à ce que nous puissions utiliser des implantations différentes dans des déploiements différents.

¹ <http://www.martinfowler.com/books.html#eaa>

² <http://martinfowler.com/eaaCatalog/plugin.html>

Donc le problème fondamental est : comment assembler ces plugins dans une application ? C'est l'un des problèmes fondamentaux rencontrés par cette nouvelle race de conteneurs légers et universellement ils le font tous en se servant de l'Inversion de Contrôle.

3. Inversion de Contrôle

Quand on explique que ces conteneurs sont très utiles car qu'ils implantent « l'Inversion de Contrôle » je me sens très perplexe. L'inversion de contrôle est une caractéristique commune des frameworks, 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 véritable question est : quel aspect du contrôle inversent-ils ? Quand je me suis d'abord heurté à l'inversion de contrôle, c'était dans le contrôle principal d'une interface utilisateur. Les premières interfaces utilisateur étaient contrôlées par l'application. Vous trouviez un ordre de commandes comme "Entrer un nom", "Entrer l'adresse" ; votre programme faisait suivre la saisie et fournissait une réponse à chaque ordre. Avec les UI graphiques (ou même basées sur écran) le framework UI devait contenir cette boucle principale et votre programme fournissait des gestionnaires d'événement pour les différents champs de l'écran. Le contrôle principal du programme était inversé, éloigné de vous vers le framework.

Pour cette nouvelle race de conteneurs, l'inversion concerne la manière dont ils consultent l'implantation d'un plugin. Dans mon exemple naïf le lister cherchait l'implantation du finder en l'instantiant directement. Ceci interdit au finder d'être un plugin. L'approche utilisée dans ces conteneurs consiste à s'assurer que n'importe quel utilisateur d'un plugin suit une certaine convention qui permet à un module d'assemblage séparé d'injecter l'implantation dans le lister.

En conséquence je pense que nous avons besoin d'un nom plus spécifique pour ce modèle. L'Inversion de Contrôle (IoC) est un terme trop générique et nombreux sont ceux qui le trouvent confus. Suite à de nombreuses discussions avec divers préconisateurs de l'IoC nous nous sommes rejoint sur le nom d'Injection de Dépendance (Injection de code).

Je vais commencer en parlant des diverses formes d'injection de dépendance, mais en indiquant dès maintenant que ce n'est pas la seule façon de supprimer la dépendance entre la classe d'application et l'implantation d'un plugin. L'autre modèle que vous pouvez utiliser est le Localisateur de Service et j'en discuterai après en avoir fini avec l'explication de l'Injection de Dépendance.

4. Les formes d'Injection de Dépendance

L'idée à la base de l'Injection de Dépendance est d'avoir un objet séparé, un assembleur, qui peuple un champ dans la classe lister avec une implantation appropriée à l'interface du finder, aboutissant à un diagramme de dépendance suivant les lignes de la Figure 2.

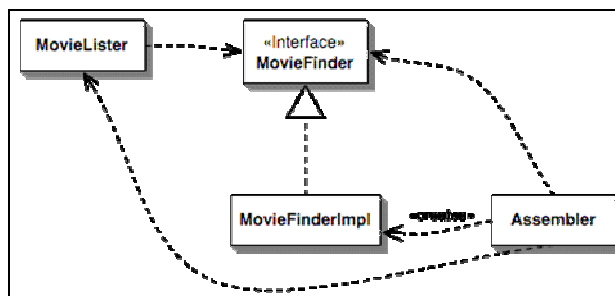


Figure 2 : les dépendances pour un Injecteur de Dépendance

Il y a trois formes principales d'injection de dépendance. Les noms que j'utilise pour les désigner sont l'Injection par Constructeur, l'Injection par Mutateur et l'Injection par Interface. Si vous lisez à ce sujet les discussions actuelles concernant l'Inversion de Contrôle vous en entendrez parler comme

mentionnés par le type 1 IoC (injection par interface), le type 2 IoC (injection par mutateur) et le type 3 IoC (injection par constructeur). Je trouve que les noms numérotés sont plutôt difficile à garder en mémoire, c'est pourquoi j'ai utilisé les noms définis ici.

4.1. Injection par Constructeur avec PicoContainer

Je commencerai par montrer de quelle manière est faite cette injection en utilisant un conteneur léger appelé PicoContainer³. Je commence ici principalement parce que plusieurs de mes collègues à ThoughtWorks sont très actifs dans le développement de PicoContainer (oui, c'est une sorte de népotisme d'entreprise).

PicoContainer utilise un constructeur pour décider de la manière d'injecter une implantation de finder dans la classe lister. Pour que cela fonctionne, la classe qui liste les films doit déclarer un constructeur qui intègre tout ce qui doit être injecté.

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

Le finder devra être contrôlé par le pico conteneur, et le nom du fichier texte devra également être injecté dedans par le conteneur.

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

Le pico conteneur a alors besoin qu'on lui spécifie l'implantation de classe à associer avec chaque interface et quelle chaîne de caractère injecter dans le finder.

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

Ce code de configuration est typiquement placé dans une classe différente. Dans notre exemple, chaque ami qui utilise mon lister pourrait écrire le code de configuration approprié dans une certaine classe d'installation qui lui est propre. Bien sûr il est classique de mettre ce type d'information de configuration dans des fichiers config séparés. Vous pouvez écrire une classe pour lire un fichier config et initialiser le conteneur convenablement. Bien que PicoContainer ne contienne pas cette fonctionnalité lui-même, il y a un projet étroitement lié appelé NanoContainer qui fournit les wrappers appropriés pour vous permettre d'avoir des fichiers de configuration XML. Un tel nano conteneur fera l'analyse syntaxique XML et configurera ensuite un pico conteneur sous-jacent. La philosophie de ce projet est de séparer le format de fichier config du mécanisme sous-jacent.

Pour utiliser le conteneur vous écrivez le code à peu près comme cela :

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

³ <http://www.picocontainer.org>

Bien que dans cet exemple j'ai utilisé l'injection de constructeur, PicoContainer supporte également l'injection de mutateur, même si ses développeurs préfèrent largement l'injection de constructeur.

4.2. Injection par Mutateur avec Spring

La framework Spring⁴ est un vaste Framework pour le développement Java d'entreprise. Il inclut des couches d'abstraction pour les transactions, les frameworks de persistance, le développement d'applications web et JDBC. Comme PicoContainer il supporte à la fois l'injection par constructeur et par mutateur, mais ses développeurs ont tendance à préférer l'injection par mutateur – ce qui en fait un choix approprié pour cet exemple.

Pour faire que mon lister de films accepte l'injection je définis une méthode de mutation pour ce service.

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

De la même façon je définis un mutateur pour la chaîne de caractère du finder.

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

La troisième étape consiste à mettre la configuration pour les fichiers. Spring supporte la configuration via des fichiers XML et aussi par le code, mais XML est le moyen dédié pour le faire.

```
<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>
```

Le test ressemble alors à cela :

```
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());
}
```

4.3. Injection par Interface

La troisième technique d'injection consiste à définir et utiliser des interfaces pour l'injection. Avalon⁵ est un exemple d'un framework qui utilise cette technique. J'en parlerai plus en détail plus tard, mais avant je vais l'utiliser avec un exemple de code simple.

⁴ <http://www.springframework.org>

⁵ <http://avalon.apache.org>

Avec cette technique je commence en définissant une interface que j'utiliserai pour exécuter l'injection à travers elle. Voici l'interface pour injecter un finder de films dans un objet :

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

Cette interface devra être définie par quiconque fournit l'interface `MovieFinder`. Elle doit être implantée par toute classe qui souhaite utiliser un finder, comme le lister.

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

J'utilise une approche similaire pour injecter le nom de fichier dans l'implantation du finder.

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

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

Alors, comme d'habitude, j'ai besoin d'un code de configuration pour dialoguer avec les implantations. Par amour de la simplicité, je le ferai dans le code :

```
class Tester...
    private Container container;

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

Cette configuration à deux niveaux, enregistrant les composants par des clés de consultation, est assez semblable aux autres exemples.

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

Une nouvelle étape consiste à enregistrer les injecteurs qui injecteront les composants dépendants. Chaque interface d'injection a besoin d'un certain code pour injecter l'objet dépendant. Ici je le fais en enregistrant des objets d'injecteur avec le conteneur. Chaque objet d'injecteur implante l'interface d'injecteur.

```
class Tester...
    private void registerInjectors() {
        container.registerInjector(InjectFinder.class,
container.lookup ("MovieFinder"));
        container.registerInjector(InjectFinderFilename.class, new
FinderFilenameInjector());
    }
}

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

Quand l'objet dépendant est une classe écrite pour ce conteneur, cela signifie quelque chose pour le composant d'implanter l'interface d'injecteur lui-même, comme je le fais ici avec le finder de films.

Pour des classes génériques, comme avec String, j'utilise une classe interne dans le code de configuration.

```
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");
        }
    }
}
```

Les tests utilisent alors le conteneur.

```
class FinderFilenameInjector...
    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());
    }
}
```

Le conteneur utilise les interfaces d'injection déclarées pour déterminer les dépendances et les injecteurs pour injecter les dépendants corrects (l'implantation spécifique du conteneur que j'ai faite ici n'est pas importante pour la technique et je ne le montrerai pas parce que vous ne feriez qu'en rire).

5. Utilisation d'un Localisateur de Service

L'avantage clef d'un Injecteur de Dépendance est qu'il supprime la dépendance que la classe `MovieLister` a sur l'implantation concrète du `MovieFinder`. Cela me permet de fournir des listers à des amis et leur permet de plugguer une implantation appropriée à leur propre environnement. L'injection n'est pas la seule façon de casser cette dépendance, une autre méthode consiste à utiliser un Localisateur de Service⁶.

L'idée de base qui se cache derrière un Localisateur de Service est d'avoir un objet qui sait comment se saisir de tous les services dont une application pourrait avoir besoin. Donc un Localisateur de Service pour cette application aurait une méthode qui renvoie un finder de films quand c'est nécessaire. Bien sûr cela n'englobe pas tout, nous devons toujours obtenir le localisateur au sein du lister, aboutissant aux dépendances de la Figure 3.

⁶ <http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html>

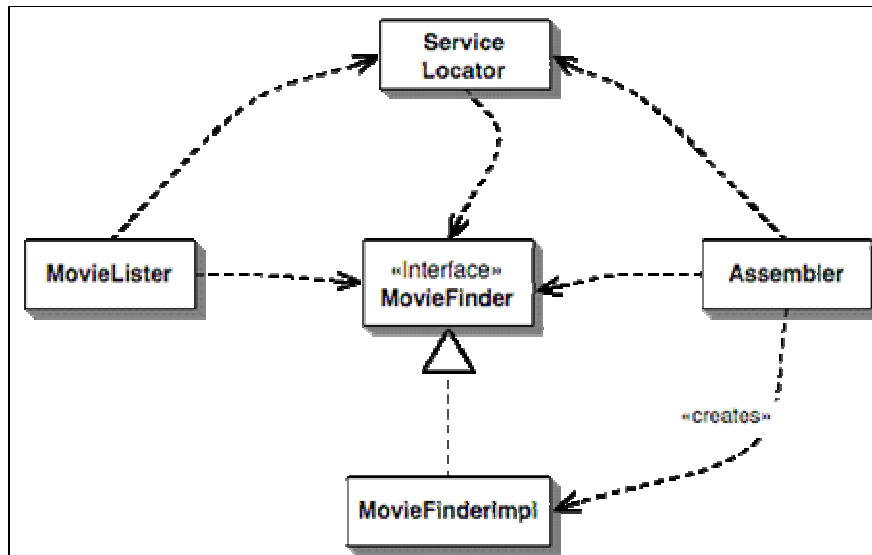


Figure 3 : dépendances pour un Localisateur de Service

Dans ce cas j'utiliserai le `ServiceLocator` comme un `Registre`⁷ singleton. Le lister peut alors l'utiliser pour obtenir le finder quand il est instancié.

```

class MovieLister...
    MovieFinder finder = ServiceLocator.movieFinder();
class ServiceLocator...
    public static MovieFinder movieFinder() {
        return soleInstance.movieFinder;
    }
    private static ServiceLocator soleInstance;
    private MovieFinder movieFinder;
  
```

Comme lors de l'approche de l'injection, nous devons configurer le Localisateur de Service. Ici je le fais dans le code, mais il n'est pas difficile d'utiliser un mécanisme qui lirait les données appropriées à partir d'un fichier de configuration.

```

class Tester...
    private void configure() {
        ServiceLocator.load(new ServiceLocator(new
        ColonMovieFinder("movies1.txt")));
    }
class ServiceLocator...
    public static void load(ServiceLocator arg) {
        soleInstance = arg;
    }

    public ServiceLocator(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
  
```

Voici le code de test.

```

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());
    }
  
```

⁷ <http://martinfowler.com/eaCatalog/registry.html>

J'ai souvent entendu comme remarque que ces types de localisateurs de service sont une mauvaise chose parce qu'ils ne sont pas testables car on ne peut pas leur substituer d'implantations. On peut certainement mal les concevoir pour avoir ce genre d'ennui, mais ce n'est pas obligatoire. Dans ce cas l'instance de Localisateur de Service est uniquement un simple détenteur de données. Je peux facilement créer le localisateur avec des implantations de test de mes services.

Pour un localisateur plus perfectionné je peux sous-classer le Localisateur de Service et passer cette sous-classe dans la variable de classe d'enregistrement. Je peux modifier les méthodes statiques pour appeler une méthode sur l'instance ayant plutôt accès directement aux variables d'instance. Je peux fournir des processus (thread) de localisateurs spécifiques en utilisant un processus de stockage spécifique. Tous cela peut être fait sans changer les clients du Localisateur de Service.

Une manière de voir cela est que le Localisateur de Service est un Registre, pas un singleton. Un singleton fournit une façon simple d'implanter un Registre, mais cette décision d'implantation est facilement modifiable.

5.1. Utilisation d'une Interface Isolée pour le Localisateur

Une des questions qui ressort de l'approche simple ci-dessus, est que le `MovieLister` est dépendant de la totalité de la classe du localisateur, bien qu'il n'utilise seulement qu'un service. Nous pouvons limiter cela en utilisant une interface isolée. Par ce biais, au lieu d'utiliser l'interface complète du localisateur, le lister peut déclarer uniquement le morceau d'interface dont il a besoin.

Dans cette situation le fournisseur du lister fournirait également une interface de localisateur dont il doit se saisir pour le finder.

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

Le localisateur a alors besoin d'implanter cette interface pour fournir l'accès au 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;
```

Vous remarquerez que puisque nous voulons utiliser une interface, nous ne pouvons pas désormais avoir uniquement accès aux services par des méthodes statiques. Nous devons utiliser la classe pour obtenir une instance de localisateur et l'utiliser ensuite pour obtenir ce dont nous avons besoin.

5.2. Un Localisateur de Service Dynamique

L'exemple précédent était statique, en cela que la classe de Localisateur de Service a des méthodes pour chacun des services dont vous avez besoin. Ce n'est pas la seule manière de procéder, vous pouvez également faire un Localisateur de Service Dynamique qui vous permet de masquer n'importe quel service dont vous avez besoin et faire vos choix lors de l'exécution.

Dans ce cas, le Localisateur de Service utilise une Map à la place d'attributs pour chacun des services et fournit des méthodes génériques pour obtenir et charger des 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);
}

```

La configuration implique le chargement d'un service avec une clé appropriée.

```

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

```

Je me sert du service en utilisant la même chaîne clef.

```

class MovieLister...
    MovieFinder finder = (MovieFinder) ServiceLocator.getService("MovieFinder");

```

Dans l'ensemble je n'aime pas cette approche. Bien que ce soit certainement flexible, ce n'est pas très explicite. La seule manière qui me permette de découvrir comment étendre un service passe par des clés textuelles. Je préfère des méthodes explicites parce qu'il est plus facile de trouver où ils se trouvent en regardant les définitions d'interface.

5.3. Utilisation simultanée d'un localisateur et d'une injection avec Avalon

L'injection de dépendance et un Localisateur de Service ne sont pas nécessairement des concepts mutuellement exclusifs. Un bon exemple d'utilisation des deux est le framework Avalon. Avalon utilise un Localisateur de Service, mais utilise l'injection pour dire aux composants où trouver le localisateur.

Berlin Loritsch m'a envoyé cette version simple de mon exemple fonctionnel en utilisant Avalon.

```

public class MyMovieLister implements MovieLister, Serviceable {
    private MovieFinder finder;

    public void service( ServiceManager manager ) throws ServiceException {
        finder = (MovieFinder) manager.lookup("finder");
    }
}

```

La méthode du service est à un exemple d'injection d'interface, permettant au conteneur d'injecter un manager de service dans `MyMovieLister`. Le manager de service est un exemple d'un Localisateur de Service. Dans cet exemple le lister ne stocke pas le manager dans un attribut, au lieu de cela il l'utilise immédiatement pour la consultation du finder, qu'il stocke vraiment .

6. Choix de l'option à utiliser

Jusqu'ici je me suis concentré à expliquer comment je vois ces modèles et leurs variations. Maintenant je peux commencer à parler du pour et du contre pour aider à envisager ceux qu'il faut utiliser et quand.

6.1. Localisateur de Service contre Injection de Dépendance

Le choix fondamental se situe entre Localisateur de Service et Injection de Dépendance. Le premier point est que ces deux implantations fournissent le découplage fondamental qui manque dans l'exemple naïf - dans les deux cas le code de l'application est indépendant de l'implantation concrète de l'interface de service. L'importante différence entre les deux modèles est la manière dont on fournit cette implantation à la classe de l'application. Avec le Localisateur de Service la classe applicative le

demande explicitement grâce à un message au localisateur. Avec l'injection il n'y a aucune demande explicite, le service apparaît dans la classe applicative – d'où l'inversion de contrôle.

L'inversion de contrôle est un dispositif classique des frameworks, mais c'est une démarche qui à un coût. Elle a tendance à être difficile à comprendre et amène des problèmes quand vous essayez de debugger. Donc dans l'ensemble je préfère l'éviter à moins d'en avoir besoin. Cela ne doit pas faire penser que c'est une mauvaise chose, seulement je pense qu'elle doit se justifier en regard de l'alternative plus directe.

La différence clef est qu'avec un Localisateur de Service chaque utilisateur d'un service a une dépendance au localisateur. Le localisateur peut masquer des dépendances à d'autres implantations, mais vous devez vraiment voir le localisateur. Donc le choix entre le localisateur et l'injecteur dépend de la manière dont cette dépendance pose problème.

L'utilisation de l'injection de dépendance peut permettre de plus facilement voir quelles sont les dépendances de composants. Avec l'injecteur de dépendance vous pouvez uniquement regarder le mécanisme d'injection, comme le constructeur, et voir les dépendances. Avec le Localisateur de Service vous devez chercher dans le code source les appels au localisateur. Les IDEs modernes possédant une fonction de recherche de références rend cela plus facile, mais ce n'est pas encore aussi facile que de regarder le constructeur ou les mutateurs.

Une bonne part de ceci dépend de la nature de l'utilisateur du service. Si vous construisez une application avec diverses classes qui utilisent un service, alors une dépendance des classes de l'application vers le localisateur n'est pas une bonne chose. Dans mon exemple où je donne un MovieLister à mes amis, là l'utilisation d'un Localisateur de Service marche tout à fait bien. Tout ce qu'ils ont à faire est de configurer le localisateur pour accrocher les bonnes implantations de service, soit via un certain code de configuration ou par un fichier de configuration. Dans ce type de scénario je ne vois pas ce que l'inversion d'injecteur fournisse qui soit intéressant.

La différence se ressent si le lister est un composant que je fournis à une application que d'autres personnes ont écrit. Dans ce cas je ne sais presque rien des API des localisateurs de service que mes clients vont utiliser. Chacun clients pourraient avoir leurs propres localisateurs de service incompatibles entre eux. Je peux contourner en partie ce problème en utilisant l'interface isolée. Chaque client peut écrire un adaptateur qui fait correspondre mon interface à son localisateur, mais quoi qu'il arrive je dois toujours voir le premier localisateur pour consulter mon interface spécifique. Et une fois que l'adaptateur apparaît alors la simplicité de connexion directe à un localisateur commence à dévier.

Puisque avec un injecteur vous n'avez pas de dépendance du composant vers l'injecteur, le composant peut ne pas obtenir de nouveaux services de l'injecteur une fois qu'il a été configuré.

Une raison souvent avancée pour préférer l'injection de dépendance est la simplification des tests. L'idée envisagée ici est que pour faire les tests, vous avez besoin de remplacer facilement des implantations de service réelles par des bouchons (mocks) ou des souches (stubs). Cependant il n'y a ici vraiment aucune différence entre l'injection de dépendance et le Localisateur de Service : les deux sont très susceptibles de stubbing. Je soupçonne que cette observation provienne de projets où les gens ne font pas l'effort de s'assurer que leur Localisateur de Service peut être facilement substitué. C'est là où des tests continus sont appréciables, si vous ne pouvez pas facilement stubber des services pour le test, cela implique alors un sérieux problème de conception.

Bien sûr le problème des tests est renforcé dans les environnements à composants qui sont très intrusifs, comme le framework des EJB Java. Mon avis à ce sujet est que ce type de frameworks devraient réduire au minimum leur impact sur le code de l'application et en particulier ne devraient pas faire les choses qui ralentissent le cycle d'édition-exécution. L'utilisation de plug-in pour se substituer à des composants lourds aide vraiment beaucoup ce processus, qui est essentiel pour des pratiques comme le Développement Piloté par les Tests.

Donc la question première concerne les personnes qui écrivent du code qui doit être utilisé dans des applications situées hors du contrôle de l'auteur. Dans ce cas, le simple fait d'envisager un Localisateur de Service est un problème.

6.2. Injection par Constructeur contre Injection par Mutateur

Pour la combinaison de service, vous devez toujours avoir une certaine convention pour faire dialoguer des choses ensemble. Le principal avantage de l'injection est qu'elle nécessite des conventions très simples - au moins pour les injections par constructeur et par mutateur. Vous ne devez faire rien d'étrange dans votre composant et il est assez direct pour un injecteur de tout configurer.

L'injection par interface est plus intrusive puisque vous devez écrire de nombreuses interfaces pour faire le tri de tout. Pour un petit jeu d'interfaces requises par le conteneur, comme dans l'approche d'Avalon, ce n'est pas trop mal. Mais il faut plus de travail pour assembler des composants et des dépendances, ce qui explique que la tendance actuelle des conteneurs légers se porte sur l'injection par constructeur et par mutateur.

Le choix entre l'injection par constructeur ou par mutateur est intéressant car il relève de questions plus générales de la programmation orientée objet – devez-vous remplir les attributs par un constructeur ou par des mutateurs.

Mon défaut habituel avec les objets est, autant que possible, de créer des objets viables dès la construction. Ce conseil est tiré du livre de Kent Beck *Smalltalk Best Practice Patterns*⁸ : *Constructor Method and Constructor Parameter Method*. Les constructeurs avec des paramètres vous donnent une déclaration claire de ce que signifie créer un objet viable à tout moment. S'il y a plusieurs façons de s'y prendre, créez de multiples constructeurs qui montrent les différentes combinaisons.

Un autre avantage de l'initialisation par constructeur consiste en ce qu'elle vous permet de cacher explicitement n'importe quel attribut qui soit immuable simplement en ne lui fournissant pas de mutateur. Je pense que c'est important - si quelque chose ne devrait pas changer alors l'absence d'un mutateur l'exprime très bien. Si vous utilisez des mutateurs pour l'initialisation, cela peut alors devenir douloureux (en effet dans ces situations je préfère éviter la convention habituelle de mutation, je préférerais une méthode `initFoo`, pour souligner que c'est une chose que vous ne devriez faire qu'à la création).

Mais dans toute situation il y a des exceptions. Si vous avez beaucoup de paramètres dans vos constructeurs les choses peuvent sembler inappropriées, particulièrement avec des langages sans paramètres clé. Il est vrai qu'un constructeur long est souvent le signe d'un objet surchargé qui devrait être segmenté, mais il y a des cas où c'est ce dont vous avez besoin.

Si vous disposez de multiples moyens pour construire un objet viable, il peut être difficile de le montrer par des constructeurs, puisque les constructeurs peuvent uniquement varier sur le nombre et le type des paramètres. C'est lorsque les Factory Methods entrent en jeu, celles-ci peuvent utiliser une combinaison de constructeurs privés et de mutateurs pour mettre en œuvre leur travail. Le problème avec les Factory Methods classiques pour l'assemblage de composants est qu'on les voit d'habitude comme des méthodes statiques et qu'on ne peut les avoir par des interfaces. Vous pouvez faire une classe factory, mais alors cela devient juste une autre instance de service. Un service factory est souvent une bonne tactique, mais vous avez toujours à instancier la factory utilisant une de ces techniques.

Les constructeurs s'avèrent également limités si vous avez des paramètres simples comme des String. Avec l'injection de mutateur vous pouvez donner à chaque mutateur un nom pour indiquer ce que la String est supposée faire. Avec des constructeurs vous comptez uniquement sur la position, qui est plus difficile à suivre.

Si vous avez de multiples constructeurs et de l'héritage, les choses peuvent alors devenir particulièrement maladroites. Pour tout initialiser vous devez fournir aux constructeurs l'appel à chaque superclasse de constructeur, tout en ajoutant vos arguments. Cela peut mener à une explosion encore plus importante du nombre de constructeurs.

⁸ <http://www.amazon.com/exec/obidos/ASIN/013476904X/103-9242494-0043817>

Malgré ces inconvénients ma préférence va par commencer à l'injection par constructeur, mais l'injection par mutateur devient plus pertinente lorsque les problèmes que j'ai décrits précédemment deviennent gênants.

Cette question est à l'origine de beaucoup de débats entre diverses équipes qui fournissent des injecteurs de dépendance au sein de leurs frameworks. Cependant il semble que la plupart des personnes qui construisent ces frameworks se sont rendues compte qu'il est important de supporter les deux mécanismes, même s'il existe une préférence pour l'un d'entre eux.

6.3. Code ou fichiers de configuration

Une question à part mais souvent associée concerne le choix entre utiliser des fichiers de configuration ou le code d'une API pour dialoguer avec les services. Pour la plupart des applications qui doivent être déployées dans de nombreux endroits différents, un fichier de configuration séparé a généralement plus d'intérêt. Presque tout le temps ce sera un fichier XML. Cependant il y a des cas où c'est le code du programme qui est plus facile à utiliser pour faire l'assemblage. Un cas à considérer est celui où vous avez une application simple qui ne possède pas beaucoup de variation de déploiement. Dans ce cas un morceau de code peut être plus clair qu'un fichier XML séparé.

Un cas contrastant est celui où l'assemblage est particulièrement complexe, impliquant des étapes optionnelles. Une fois que vous commencez à être habitué au langage de programmation alors XML s'avère moins intéressant et il est préférable d'utiliser un vrai langage qui possède toute la syntaxe permettant d'écrire un programme clair. Vous écrivez alors une classe de montage qui fait l'assemblage. Si vous avez des scénarios de montage distincts vous pouvez fournir plusieurs classes de montage et utiliser un simple fichier de configuration pour faire le choix entre eux.

Je pense souvent que les gens sont trop empressés de définir des fichiers de configuration. Souvent un langage de programmation fournit un mécanisme de configuration direct et puissant. Les langages modernes peuvent facilement compiler de petits assembleurs qui peuvent être utilisés pour assembler des plug-in pour de plus grands systèmes. Si la compilation est douloureuse, il existe alors des langages de script qui peuvent fonctionner aussi bien.

Il est souvent dit que les fichiers de configuration ne devraient pas utiliser un langage de programmation parce qu'ils doivent être édités par des non programmeurs. Mais combien de fois est-ce le cas ? Les gens s'attendent-ils vraiment à ce que des non programmeurs modifient les niveaux d'isolation des transactions des applications complexes côté serveur ? Les fichiers de configuration non-langagiers fonctionnent bien seulement à la condition qu'ils soient simples. S'ils deviennent complexes alors il est temps de penser à l'utilisation d'un langage de programmation approprié.

Une chose que nous voyons dans le monde Java à l'heure actuelle est une cacophonie de fichiers de configuration, où chaque composant a ses propres fichiers de configuration différents des autres. Si vous utilisez une douzaine de ces composants, vous pouvez facilement vous retrouver avec une douzaine de fichiers de configuration pour être synchrone.

Mon conseil à ce propos est de toujours fournir une façon de réaliser toute configuration facilement avec une interface de programmation et ensuite d'avoir un fichier de configuration séparé facultatif. Vous pouvez facilement construire le traitement de fichier de configuration pour utiliser l'interface de programmation. Si vous écrivez un composant vous pouvez alors le laisser à votre utilisateur pour qu'il se serve soit d'une interface de programmation, soit de votre format de fichier de configuration, soit qu'il écrive son propre format de fichier de configuration personnalisé et le lie à l'interface de programmation.

6.4. Séparer la configuration de l'utilisation

Le point important dans tout cela est d'assurer la séparation entre la configuration des services et leur utilisation. C'est en effet un principe de conception fondamental qui est défini par la séparation des interfaces de l'implantation. C'est quelque chose que nous voyons dans un programme orienté objet

quand la logique conditionnelle décide quelle classe instantier et que les évaluations ultérieures de ces conditions sont faites par le polymorphisme plutôt que par un code conditionnel dupliqué.

Si cette séparation est utile dans un simple code de base, elle est particulièrement essentielle quand vous utilisez des éléments étrangers comme des composants et des services. La première question à vous poser est de savoir si vous voulez reporter le choix de la classe d'implantation à des déploiements particuliers. Si c'est le cas vous devez utiliser une implantation de plug-in. Une fois que vous utilisez des plug-in il est alors essentiel que leur assemblage soit réalisé séparément du reste de l'application pour que vous puissiez facilement substituer des configurations différentes à des déploiements différents. La manière de le faire est secondaire. Ce mécanisme de configuration peut soit configurer un Localisateur de Service, soit utiliser l'injection pour configurer des objets directement.

7. Quelques nouvelles questions

Dans cet article, je me suis concentré sur les questions de base de la configuration de service en utilisant l'Injection de Dépendance et le Localisateur de Service. Il y a d'autres sujets du même style qui méritent aussi l'attention, mais je n'ai pas encore eu le temps de les explorer. En particulier il y a la question du comportement lors du cycle de vie. Certains composants ont des événements de cycle de vie distincts : arrêt et démarrage par exemple. Une autre question porte sur l'intérêt croissant de l'orientée aspect pour implémenter les services techniques de ces conteneurs. Bien que je n'aie pas considéré ces notions dans cet article à l'heure actuelle, j'espère vraiment en écrire plus en le prolongeant ou en en écrivant un autre.

Vous pouvez découvrir beaucoup plus d'éléments sur ces idées en regardant les sites Web consacrés aux conteneurs légers. La navigation sur les sites de Picocontainer et de Spring vous mènera à de nombreuses réflexions sur ces questions et à des ébauches sur certaines de ces nouvelles questions.

8. Conclusion

L'intérêt actuel pour les conteneurs légers est dû au fait qu'ils ont tous un modèle commun sous-jacent concernant la manière de faire l'assemblage de services - le modèle d'injecteur de dépendance. L'Injection de Dépendance est une alternative utile au Localisateur de Service. En construisant les classes d'application les deux sont grossièrement équivalent, mais je pense que le Localisateur de Service a un avantage certain en raison de son comportement plus direct. Cependant si vous construisez des classes qui doivent servir dans de multiples applications alors l'Injection de Dépendance est un meilleur choix.

Si vous utilisez l'Injection de Dépendance il y a un certain nombre de styles parmi lesquels choisir. Je suggérerais que vous suiviez l'injection par constructeur à moins que vous ne vous heurtiez à des problèmes spécifiques avec cette approche, dans ce cas passez à l'injection par mutateur. Si vous voulez construire ou obtenir un conteneur, cherchez celui qui supporte à la fois l'injection par mutateur et par constructeur.

Le choix entre le Localisateur de Service et l'Injection de Dépendance est moins important que le principe consistant à séparer dans une application la configuration des services de leur utilisation.

9. Remerciements

Mes sincères remerciements aux nombreuses personnes qui m'ont aidées pour cet article. Rod Johnson, Paul Hammant, Joe Walnes , Aslak Helleoy, Jon Tirsén et Bill Caputo m'ont aidé à en venir aux mains avec ces concepts et ont commentés les premiers projets de cet article. Berin Loritsch et Hamilton Verissimo d'Oliveira ont fourni certains conseils très utiles sur le fonctionnement d'Avalon. Dave W smith a persisté à poser des questions sur mon code de configuration d'injection d'interface initial et m'a ainsi convaincu qu'il était stupide.

10. Révisions Significatives

23 Jan 04 : Le code de configuration de l'exemple d'injection d'interface a été refait.

16 Jan 04 : Un exemple court ajouté sur l'utilisation simultanée du localisateur et de l'injection avec Avalon.

14 Jan 04 : Première Publication