

L'implémentation dynamique en Java (classe Proxy et InvocationHandler)

Par Alain Defrance  

Date de publication : 7 juin 2009

Dernière mise à jour : 16 juillet 2009

L'implémentation dynamique est une façon de construire dynamiquement une instance. Cette pratique peut être difficile à appréhender mais nous allons voir comment l'utiliser, et dans quels types de projets elle se révèle très utile.

I - Introduction.....	3
II - Mise en place basique d'une implémentation dynamique.....	3
II-A - Le principe.....	3
II-B - Pourquoi passer par de l'implémentation dynamique ?.....	3
II-C - L'interface que nous allons dynamiquement implémenter.....	4
II-D - L'interface InvocationHandler.....	4
II-E - La classe Proxy.....	4
III - L'implémentation dynamique dans un conteneur EJB.....	4
III-A - Objectifs du container.....	4
III-B - Gestion des invocations.....	5
IV - Notre propre mini container.....	5
IV-A - Les interceptors.....	5
IV-A-1 - LoggingInterceptor.....	6
IV-A-2 - BeanInterceptor.....	7
IV-B - La classe Invocation.....	8
IV-C - L'implémentation d'InvocationHandler.....	9
IV-D - Notre micro container.....	9
IV-E - L'application utilisant le conteneur.....	10
IV-E-1 - Création de nos beans.....	10
IV-E-2 - Notre application.....	11
IV-F - Améliorations.....	11
IV-F-1 - Notre nouveau conteneur.....	12
IV-F-2 - Utiliser notre nouveau container.....	12
V - Conclusion.....	13
VI - Remerciements.....	13

I - Introduction

Lorsque nous écrivons une classe et son implémentation, nous définissons un comportement à la compilation. Parfois nous n'avons pas toute l'information nécessaire pour écrire l'implémentation à la compilation car cette dernière dépendra d'un contexte applicatif particulier. C'est pourquoi il est possible de différer cette écriture, c'est-à-dire détyper l'invocation. Attention cependant il n'est pas question de générer du code proprement dit, mais simplement de rediriger les appels vers divers processus.

L'implémentation dynamique est beaucoup utilisée par les conteneurs EJB (comme JBoss, Glassfish, WebSphere, etc ...). En effet, les beans sont écrits par le développeur, mais le conteneur doit modifier cette implémentation de manière transparente pour ne pas affecter l'écriture du développeur. Un session bean par exemple, s'il est utilisé à distance (@Remote), utilisera le RMI pour les appels à distance. Cette couche RMI n'est pas gérée par le développeur, et ça sera donc la tâche du conteneur. Ce sera lui qui implémentera le stub et le skeleton. Le stub utilisera RMI pour être lié à son skeleton, puis le skeleton devra déléguer l'appel au session bean implémenté par le développeur. Le conteneur EJB implémentera ces interfaces dynamiquement.

Il est également intéressant de se servir de l'implémentation dynamique afin d'affiner le comportement d'une méthode. Certains modèles de conception permettent cela (comme le Decorator), mais grâce à l'implémentation dynamique nous pourrions aller plus loin.

Afin de bien assimiler l'utilité de ces implémentations dynamiques, nous allons créer un mini conteneur EJB. Bien évidemment, on sera très loin d'un vrai conteneur EJB, mais nous respecterons la philosophie et cela sera largement suffisant pour assimiler l'implémentation dynamique.

La partie dédiée au conteneur EJB a été grandement inspirée par des sources de Julien Viet, ancien développeur à JBoss, qui expliquait comment un conteneur EJB pouvait fonctionner.

II - Mise en place basique d'une implémentation dynamique

II-A - Le principe

Le principe de l'implémentation dynamique n'est pas compliqué, il s'agit de découpler l'interface de son implémentation. Tous les appels seront redirigés vers une méthode, et c'est cette méthode qui va invoquer la méthode écrite par le développeur du bean, et éventuellement, gérer des listes de procédures à exécuter.

La méthode qui permet d'aiguiller et composer l'appel est invoke() et fait partie de l'interface InvocationHandler. Il va donc falloir implémenter cette interface qui sera utilisée à l'allocation de l'instance dynamiquement implémentée.

Ce sera enfin la classe Proxy possédant une méthode statique newProxyInstance() qui construira réellement l'instance dynamiquement à partir de l'InvocationHandler, d'un tableau d'interfaces (celles que l'InvocationHandler doit implémenter), et le ClassLoader ayant chargé l'interface que l'on veut implémenter.

II-B - Pourquoi passer par de l'implémentation dynamique ?

Il n'est pas rare d'entendre des personnes dire : "ça sert à rien" ou "on peut s'en passer". Bien évidemment l'implémentation dynamique n'est pas une solution exclusive à certains problèmes, cependant c'est parfois la façon la plus élégante pour répondre à un besoin précis.

Il ne faut pas chercher à l'utiliser à tout va, on perdrait en lisibilité. Nous allons, pour mieux appréhender son utilisation, se baser sur un exemple très simple et nous focaliser sur son fonctionnement, puis plus tard nous verrons un cas plus concret où l'implémentation se justifie aisément.

Typiquement, l'implémentation dynamique est utile dans plusieurs cas :

- L'implémentation nécessite certaines données que l'on ne peut pas avoir à la compilation
- On veut modifier le comportement d'un appel, rajouter des responsabilités
- Mise en place d'un lazyloading (chargement de ressources uniquement au besoin)
- Faire du failover (gestion d'une liste de serveurs interrogeables en cas de panne de l'un d'entre eux)

II-C - L'interface que nous allons dynamiquement implémenter

Interface à implémenter : Customer.java

```
public interface Customer {  
    public Integer getRand();  
}
```

II-D - L'interface InvocationHandler

L'InvocationHandler est une interface qui permettra l'implémentation dynamique. Elle possède une méthode invoke(Object proxy, Method method, Object[] args). Le proxy est l'instance ayant provoqué l'appel, method est la méthode invoquée, et args sont les arguments de l'appel. Cette méthode renvoie un Object qui est la valeur de retour de la méthode ayant provoqué l'invocation.

L'implémentation de l'InvocationHandler serait la suivante :

Implémentation de l'InvocationHandler : CustomHandler.java

```
public class CustomHandler implements InvocationHandler {  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        System.out.println("invocation");  
        return new Random().nextInt();  
    }  
}
```

II-E - La classe Proxy

La classe Proxy ne servira qu'à assembler les éléments nécessaires à la création de l'instance, puis réalisera l'allocation de la nouvelle instance. Son utilisation est la suivante :

Implémentation du programme principal avec la classe Proxy : Main.java

```
public class Main {  
    public static void Main(String[] argv) {  
        Customer instance = (Customer) Proxy.newProxyInstance(  
            Customer.class.getClassLoader(),  
            new Class[] {Customer.class},  
            new CustomHandler()  
        );  
        System.out.println(instance.getRand());  
        System.out.println(instance.getRand());  
    }  
}
```

Sortie standard

```
invocation  
605842933  
invocation  
-2016932835
```

III - L'implémentation dynamique dans un conteneur EJB

III-A - Objectifs du container

La raison d'être d'un conteneur EJB est de gérer des beans. C'est lui qui gère les pools, les mises en cache, et les allocations de ces beans.

Un des objectifs de ces conteneurs est qu'il faut imposer le moins de contraintes possibles au développeur d'EJB. Une des solutions est de réimplémenter les beans au travers de l'implémentation dynamique. On pourra alors rajouter aux implémentations fournies par le développeur d'EJB des responsabilités spécifiques au conteneur.

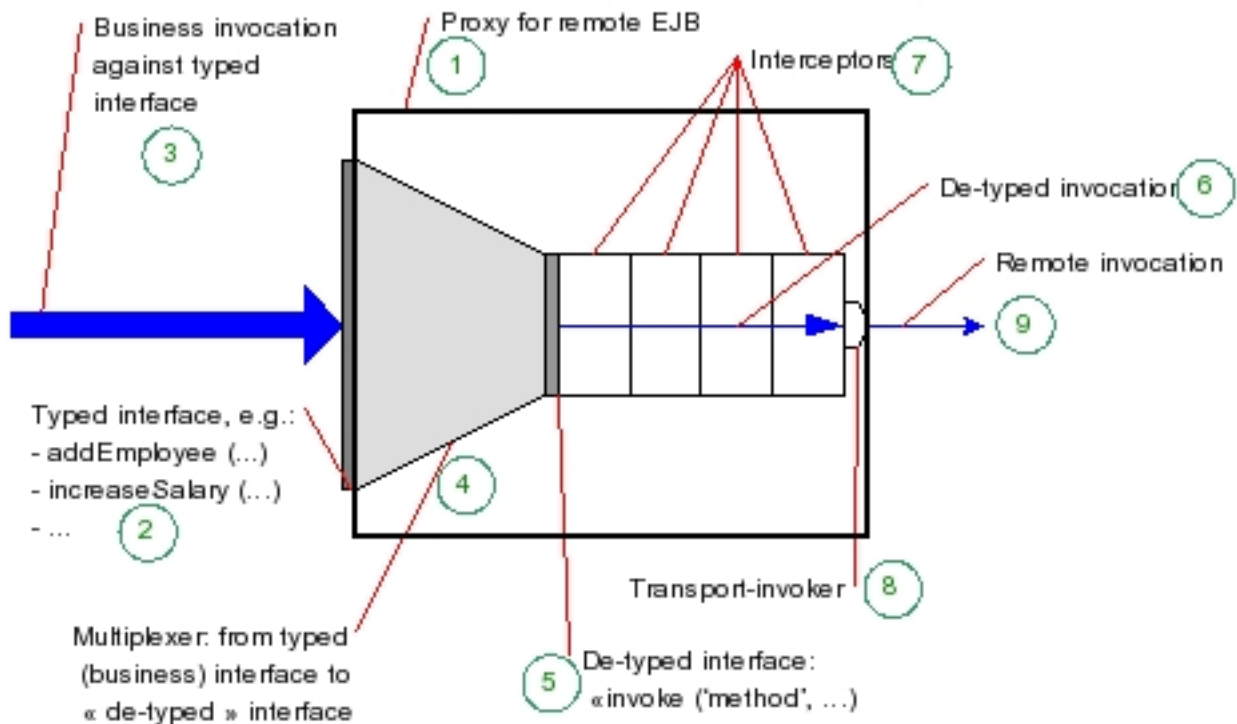
Ceci peut être le traitement d'annotations, du logging, et à peu près tout ce qui peut être utile.

III-B - Gestion des invocations

Afin de gérer proprement les ajouts de responsabilités, nous allons créer une interface `Interceptor` qui représentera une responsabilité que nous allons ajouter à la méthode invoquée. Ainsi il nous suffira de gérer des chaînes d'interceptors pour appliquer toutes modifications désirées.

Nous finirons par créer une classe `Invocation` qui se chargera du déroulement de la chaîne.

IV - Notre propre mini container



Shéma d'une implémentation dynamique par un conteneur EJB

C'est le moment de passer à l'action, nous allons implémenter un micro container utilisant l'implémentation dynamique. Rien ne sera compliqué, il ne s'agit, comme bien souvent, que d'un problème d'organisation.

Nos objectifs sont les suivants :

- Avoir une classe gérant l'implémentation à l'instanciation (`MicroContainer`)
- Logger tous les appels sous deux formes différentes précisées par annotation (logueur ou sortie standard)

IV-A - Les interceptors

Comme nous l'avons introduit plus tôt, nous allons utiliser une chaîne d'interceptors, qui en fait sont tous des traitements à appliquer au cours de l'implémentation de l'instance, afin de séparer les types d'ajouts. Nous aurons au moins un `Interceptor` qui délèguera l'appel à la méthode écrite par le développeur. Dans notre cas nous allons en ajouter un autre : un invocateur de logging.

Nous aurons donc une interface `Interceptor` qui sera implémentée par tous les invocateurs.


Interceptor : `Interceptor.java`

```
package com.developpez.dynamic.invoke;

/**
 * @author Alain Defrance
 */
public interface Interceptor {
```

Interceptor : Interceptor.java

```
/**
 * @param invocation invocation qui sera utilisée pour appliquer la chaîne d'interceptors
 * @return retour de l'appel de l'invocation
 */
public Object invoke(Invocation invocation);
```

 Nous n'avons pas encore décrit le type *Invocation*, mais nous reviendrons dessus un peu plus tard, c'est une instance qui fera le lien entre les différentes interceptions et guidera la chaîne. Ce que l'on a besoin de savoir pour le moment sur *Invocation*, c'est que cette classe connaît la méthode invoquée, et ses arguments.

IV-A-1 - LoggingInterceptor

Il va nous falloir créer un Interceptor concret qui nous permettra de traiter le logging des invocations. Pour chaque appel fait sur le bean, on voudra journaliser le nom de la méthode qui a été appelée en fonction du type d'annotation `@Log` (que nous allons créer).

Avant toutes choses voici à quoi ressemble notre annotation :

Notre annotation : Log.java

```
package com.developpez.dynamic.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Annotation décrivant le type de log à faire générer par le conteneur
 * @author Alain Defrance
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Log {
    static enum LogType {
        STD,
        LOG
    }
    LogType logtype();
}
```

Notre Interceptor va donc regarder si cette annotation est présente sur la méthode à implémenter, et si c'est le cas, alors on effectuera les opérations nécessaires en fonction du type de log demandé.

Interceptor de log : LoggingInterceptor.java

```
package com.developpez.dynamic.invoke;

import com.developpez.dynamic.annotation.Log;
import java.lang.reflect.Method;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * @author Alain Defrance
 */
public class LoggingInterceptor implements Interceptor {
    /**
     * Intercepte l'appel pour créer le log du bon type
     * @param invocation Invocation qui sera utilisée pour appliquer la chaîne d'intercepteurs
     * @return retour de l'appel de l'invocation
     */
    public Object invoke(Invocation invocation) {
        Method method = invocation.getMethod();
        if (method.isAnnotationPresent(Log.class)) {
            Log.LogType logType = method.getAnnotation(Log.class).logtype();
```

Interceptor de log : LoggingInterceptor.java

```

        switch (logType) {
            case LOG:
                Logger.getLogger(method.getName()).log(Level.INFO, "Invocation de " +
method.getName());
                break;
            case STD:
                System.out.println("Invocation de " + method.getName());
                break;
        }
        return invocation.nextInterceptor();
    }
}

```

IV-A-2 - BeanInterceptor

Il est maintenant temps de lier l'exécution au bean implémenté par le développeur. Cet intercepteur utilisera l'invocation afin de récupérer le bean originel, la méthode appelée, et les arguments pour finalement déléguer l'appel à l'implémentation du développeur.

Interceptor de bean : BeanInterceptor.java

```

package com.developpez.dynamic.invoke;

import java.lang.reflect.Method;

/**
 * Intercepteur redirigeant l'appel vers l'implémentation originelle
 * @author Alain Defrance
 */
public class BeanInterceptor implements Interceptor {

    /**
     * Intercepte l'appel pour invoquer la méthode implémentée par le développeur
     * @param invocation Invocation qui sera utilisée pour appliquer la chaîne d'intercepteurs
     * @return retour de l'appel de l'invocation
     */
    public Object invoke(Invocation invocation) {
        try {
            Object bean = invocation.getBean();
            Method method = invocation.getMethod();
            Object[] args = invocation.getArgs();
            return method.invoke(bean, args);
        } catch (Exception e) {
            throw new InvocationException(e.getMessage());
        }
    }
}

```

Nous avons utilisé une exception `InvocationException`. Celle-ci n'existe pas nativement.



Lorsque comme ici nous utilisons l'introspection pour invoquer une méthode, nous nous exposons à une exception. Pour notre test, nous avons regroupé ces exceptions dans une seule créée pour ceci : `InvocationException`.

Interceptor de bean : BeanInterceptor.java

```

package com.developpez.dynamic.invoke;


/**
 * Propagée lors d'une exception survenue pendant une invocation
 * @author Alain Defrance
 */
public class InvocationException extends RuntimeException {

    public InvocationException(String message) {
        super(message);
    }
}

```

Interceptor de bean : BeanInterceptor.java

```
}
```

 *Un intercepteur peut servir à toute chose, en passant par le logging jusqu'à la construction d'aspects en AOP.*

Une particularité de cette classe est qu'elle devra posséder une méthode (nous l'appellerons `nextInterceptor()`) qui déroulera un à un la chaîne d'intercepteurs.

Invocation.java

```
package com.developpez.dynamic.invoke;

import java.lang.reflect.Method;

/**
 * Permet de gérer l'exécution successive des intercepteurs
 * @author Alain Defrance
 */
public class Invocation {
    private Object bean;
    private Interceptor[] interceptors;
    private Method method;
    private Object[] args;
    private int index;


    public Invocation(Object bean, Interceptor[] interceptors, Method method, Object[] args) {
        this.bean = bean;
        this.interceptors = interceptors;
        this.method = method;
        this.args = args;
    }

    public Object getBean() {
        return bean;
    }

    public Method getMethod() {
        return method;
    }

    public Object[] getArgs() {
        return args;
    }

    public Object nextInterceptor() {
        try {
            return interceptors[index++].invoke(this);
        } finally {
            index--;
        }
    }
}
```

 *Invocation lance l'invocation de l'intercepteur en se passant en paramètre afin que l'intercepteur puisse avoir toutes les informations utiles à l'invocation (bean, méthode, arguments).*

IV-B - La classe Invocation

Nous allons maintenant nous occuper du chaînage des invocations. C'est grâce à la classe `Invocation` que nous allons le faire. Elle devra posséder un constructeur acceptant :

- Le bean concerné par l'invocation

- Les intercepteurs qui seront appliqués à l'invocation
- La méthode invoquée
- Les arguments



Il ne nous faudra pas oublier les getter qui seront utiles aux intercepteurs.

IV-C - L'implémentation d'InvocationHandler

Le plus compliqué est fait, pour ce qui reste à faire, nous l'avons déjà vu dans l'implémentation classique. Nous allons créer un handler qui permettra au container de créer une instance, avec la classe Proxy, utilisant les divers invokeurs que nous venons de créer. Ce handler recevra de notre container l'instance source ainsi que la liste des invocations, il n'y aura alors plus qu'à créer une instance d'Invocation, puis appeler sa méthode nextInterceptor() pour traiter l'implémentation.

Le handler BeanInvocationHandler.java

```
package com.developpez.dynamic.handler;

import com.developpez.dynamic.invoke.Interceptor;
import com.developpez.dynamic.invoke.Invocation;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

/**
 * Appel l'invocation qui implémentera dynamiquement l'instance grâce aux intercepteurs
 * @author Alain Defrance
 */
public class BeanInvocationHandler implements InvocationHandler {
    private Object bean;
    private Interceptor[] interceptors;

    public BeanInvocationHandler(Object bean, Interceptor[] interceptors) {
        this.bean = bean;
        this.interceptors = interceptors;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Invocation invocation = new Invocation(bean, interceptors, method, args);
        return invocation.nextInterceptor();
    }
}
```

IV-D - Notre micro container

La classe qui représentera notre conteneur devra avoir une implémentation initiale (celle du développeur), et être conforme à une interface qui sera utilisée lors de l'implémentation dynamique.

Son constructeur réclamera donc ces deux types. Nous écrirons enfin la méthode createBean() qui se chargera d'utiliser la classe Proxy pour créer la nouvelle instance et la retourner au développeur. L'implémentation de cette classe, comme pour le handler, reste semblable à notre premier exemple.

Notre container MiniContainer.java

```
package com.developpez.dynamic.container;

import com.developpez.dynamic.handler.BeanInvocationHandler;
import com.developpez.dynamic.invoke.BeanInterceptor;
import com.developpez.dynamic.invoke.Interceptor;
import com.developpez.dynamic.invoke.LoggingInterceptor;
import java.lang.reflect.Proxy;


/**
 * Implémentation de notre mini container
 * @author Alain Defrance
 */
```

Notre container MiniContainer.java

```
public class MiniContainer<T> {
    private Class<? extends T> beanClass;
    private Class<T> beanInterface;
    private Interceptor[] interceptors;

    public MiniContainer(Class<? extends T> beanClass, Class<T> beanInterface) {
        this.beanClass = beanClass;
        this.beanInterface = beanInterface;
        interceptors = new Interceptor[] {
            new LoggingInterceptor(),
            new BeanInterceptor()
        };
    }

    public T createBean() {
        try {
            BeanInvocationHandler handler = new BeanInvocationHandler(beanClass.newInstance(),
interceptors);
            return (T) Proxy.newProxyInstance(
                Thread.currentThread().getContextClassLoader(),
                new Class[] {beanInterface},
                handler);
        } catch (Exception e) {
            throw new ContainerException(e.getMessage());
        }
    }
}
```

 Comme pour l'invocation, nous avons une exception `ContainerException` qui nous permet de ne pas nous soucier des exceptions pour notre exemple.

ContainerException.java

```
package com.developpez.dynamic.container;

/**
 * Survient lors d'une exception à l'instanciation dans le container
 * @author Alain Defrance
 */
public class ContainerException extends RuntimeException {
    public ContainerException(String message) {
        super(message);
    }
}
```

IV-E - L'application utilisant le conteneur

IV-E-1 - Création de nos beans

Nous allons maintenant profiter de ce que nous venons de développer et jouer le rôle du programme réutilisant notre mini container.

Il faut premièrement créer une interface pour notre beans, et son implémentation.

Interface de notre bean : Customer.java

```
package com.developpez.dynamic.bean;

import com.developpez.dynamic.annotation.Log;

/**
 * Interface de notre bean
 * @author Alain Defrance
 */
public interface Customer {
    @Log(logtype=Log.LogType.STD)
    public int getValue();
}
```

Interface de notre bean : Customer.java

```
@Log(logtype=Log.LogType.LOG)
public void setValue(int value);
}
```

Il nous reste à implémenter simplement notre interface :

Implémentation de notre bean : CustomerBean.java

```
package com.developpez.dynamic.bean;

/**
 * Implémentation de notre bean
 * @author Alain Defrance
 */
public class CustomerBean implements Customer {
    private int value;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

IV-E-2 - Notre application

Nous allons à présent créer notre application principale qui utilisera le MicroContainer, et constater que ce dernier ré-implémentera dynamiquement notre bean.

Implémentation de notre bean : CustomerBean.java

```
package com.developpez.dynamic.run;

import com.developpez.dynamic.bean.Customer;
import com.developpez.dynamic.bean.CustomerBean;
import com.developpez.dynamic.container.MiniContainer;

/**
 * Programme principal
 * @author Alain Defrance
 */
public class Main {
    public static void main(String[] argv) {
        MiniContainer mc = new MiniContainer(CustomerBean.class, Customer.class);
        Customer c = mc.createBean();
        c.setValue(42);
        System.out.println(c.getValue());
    }
}
```

Sortie standard

```
run:
Jun 10, 2009 10:05:18 PM com.developpez.dynamic.invoke.LoggingInterceptor invoke
INFO: Invocation de setValue
Invocation de getValue
42
BUILD SUCCESSFUL (total time: 0 seconds)
```

IV-F - Améliorations

Il est possible d'améliorer un peu notre conteneur. En effet, il n'est pas très élégant d'avoir à instancier un conteneur par bean à gérer. Nous allons donc améliorer notre conteneur afin qu'il puisse gérer plusieurs beans.

IV-F-1 - Notre nouveau conteneur

Nous allons rajouter un registre qui nous permettra d'avoir à disposition une map contenant les types de beans gérés par le container. Puisque le conteneur n'est pas focalisé sur un seul bean, passer l'interface et l'implémentation au constructeur n'a plus de sens. Ces paramètres vont donc disparaître au profit d'une nouvelle méthode nous permettant de remplir notre registre. Cette méthode renverra this afin de permettre l'appel en chaîne. Pour finir, nous préciserons à la méthode createBean() le type de bean à créer (présent dans la map).

Nouveau conteneur : MiniContainer.java

```
package com.developpez.dynamic.container;

import com.developpez.dynamic.handler.BeanInvocationHandler;
import com.developpez.dynamic.invoke.BeanInterceptor;
import com.developpez.dynamic.invoke.Interceptor;
import com.developpez.dynamic.invoke.LoggingInterceptor;
import java.lang.reflect.Proxy;
import java.util.HashMap;
import java.util.Map;

/**
 * Implémentation de notre mini container supportant plusieurs beans
 * @author Alain Defrance
 */
public class MiniContainer {
    private Map<Class<?>, Class<?>> registry;
    private Interceptor[] interceptors;

    public MiniContainer() {
        registry = new HashMap<Class<?>, Class<?>>();
        interceptors = new Interceptor[] {
            new LoggingInterceptor(),
            new BeanInterceptor()
        };
    }

    public <T> MiniContainer register(Class<? extends T> impl, Class<T> iface) {
        registry.put(iface, impl);
        return this;
    }

    public <T> T createBean(Class<T> iface) {
        try {
            Class<? extends T> impl = (Class<? extends T>) registry.get(iface);
            BeanInvocationHandler handler = new BeanInvocationHandler(
                impl.newInstance(), interceptors);
            return (T) Proxy.newProxyInstance(
                Thread.currentThread().getContextClassLoader(),
                new Class[] { iface },
                handler);
        } catch (Exception e) {
            throw new ContainerException(e.getMessage());
        }
    }
}
```

IV-F-2 - Utiliser notre nouveau container

Nous devons donc modifier légèrement notre utilisation afin de respecter le nouveau fonctionnement.

Utilisation du nouveau conteneur : Main.java

```
package com.developpez.dynamic.run;

import com.developpez.dynamic.bean.Customer;
import com.developpez.dynamic.bean.CustomerBean;
import com.developpez.dynamic.container.MiniContainer;
```

Utilisation du nouveau conteneur : Main.java

```
/**
 * Programme principal
 * @author Alain Defrance
 */
public class Main {
    public static void main(String[] argv) {
        MiniContainer mc = new MiniContainer();
        // Ici on enregistre qu'un bean
        mc.register(CustomerBean.class, Customer.class);
        // Plus de cast nécessaire
        Customer c = mc.createBean(Customer.class);
        c.setValue(42);
        System.out.println(c.getValue());
    }
}
```

V - Conclusion

Nous avons vu comment exploiter raisonnablement l'implémentation dynamique pour créer un petit conteneur EJB. Sans cette dernière, il aurait été beaucoup plus difficile de réaliser ce projet, et le résultat aurait certainement été moins élégant. Bien sûr cette technique est typique de Java, et ne pourra pas faire partie de la conception d'une application qui devra être portée plus tard dans un autre langage, contrairement aux modèles de conceptions. Mais puisque nous développons en Java, autant profiter de ses avantages :).

Ce type d'architecture est inspirée de celle du conteneur JBoss 3.x. L'architecture de ce conteneur EJB3 inspire également **eXo Platform** qui fonctionne de la même façon.

VI - Remerciements

Un grand merci à **Julien Viet** pour sa correction ainsi que ses sources. Cet article n'aurait pas pu avoir l'intérêt qu'il porte aujourd'hui sans son aide et les précieuses explications qu'il a pu me fournir.

Merci également à **djo.mos** pour sa relecture technique et ses multiples conseils.

Pour finir merci à **ced** pour sa relecture orthographique.