VEGAS - Design Pattern d'inversion de contrôle : Généralités

par Marc Alcaraz (Mon site)

Date de publication : 11/09/2009

Dernière mise à jour :

I - Introduction

IoC ? Injection de dépendance ? Conteneur Léger ... depuis quelques temps maintenant nous entendons parler de ce **Design Pattern** très puissant qui révolutionne la construction des applications orientées objets. Ce **Design Pattern** est vraiment pour moi indispensable une fois bien maitrisé et permet d'aller loin, très loin !

Depuis plusieurs mois maintenant je me suis lancé dans la construction de ma propre implémentation de ce pattern en m'inspirant d'une part de ce qu'il se fait de mieux à l'heure actuelle dans ce domaine (Spring, Parsley, etc..) mais surtout des problèmes et besoins que je peux rencontrer chaque jour dans mes productions. Je remercie d'ailleurs tous ceux qui ont pris le temps de me supporter et de prendre le temps de poser un oeil sur mon code et au final qui m'ont aidé à aller au bout d'une version 1 finale (ou presque.. je finiolle :D) de VEGAS et ses extensions.

La version **AS3** de **VEGAS** est composée de plusieurs extensions très solides et spécialisées. Mais **AndromedAS** avec ses outils d'**IoC**, etc. est devenu le noyau dure du framework pour créer avec facilité une RIA moderne et fonctionnelle aussi bien dans Flash, Flex, ou dans une application **AIR**.

Je commence donc ici par un premier chapitre de mon énorme article basé sur le **Design Pattern IoC** dans **VEGAS** qui va se découper en plusieurs parties ici sur ce blog et qui finira au bout du compte en livre opensource disponible très prochainement sur les pages officielles du projet et aussi sur Google Documents. J'ai encore pas mal de fonctionnalités à documenter car la bibliothèque **IoC** s'est énormément étoffée ces derniers mois (pour la bonne cause).

J'espère que cette introduction, vous permettra de vous faire une idée clair et précise des bases du pattern **IoC** mais surtout de l'orientation que j'ai pu prendre pour l'implémenter. Cet article reste ouvert à tous commentaires et s'accompagnera très bientôt d'une seconde partie qui précisera le DOM très complet des définitions d'objets utilisée par le conteneur léger **IoC** pour créer des objets dans une application.

Comme toujours, je suis toute personne pouvant m'aider à accélérer le processus de traduction de cet article en anglais.

A noter que vous trouverez les sources des exemples ci-dessous dans le repository SVN du projet :

- **http://vegas.googlecode.com/svn/AS3/trunk/bin/test/andromeda/ioc/**
- **http://vegas.googlecode.com/svn/AS3/trunk/bin/test/asgard/net/** (3 exemples d'utilisation de la classe ECMAObjectLoader)

Pour ceux qui débarquent dans l'aventure, et qui ne savent pas encore ce que représente VEGAS, comment l'installer etc.. Je vous propose d'aller faire un tour rapide sur le Google Code du projet :

- **http://code.google.com/p/vegas/**
- **http://code.google.com/p/vegas/wiki/InstallVEGASwithSVN**

Pour toutes vos questions sur VEGAS et ses extensions n'hésitez pas à vous inscrire sur le Google Groups :

- VEGASoS : **http://groups.google.com/group/vegasos**

Enfin pour suivre le projet au mieux je peux vous proposer de consulter la page officielle de VEGAS sur Ohloh :

- **http://www.ohloh.net/projects/vegas**

Il est temps maintenant de démarrer cet article sur le **Design Pattern d'IoC** et je vous promets la suite de cet article d'ici peu de temps

II - Définitions

II-A - Injection de dépendance (Dependency Injection)

Le design pattern "**Dependency injection (DI)**" se réfère à la pratique de l'injection de code dans une application. Les classes ne sont pas responsables dans l'instanciation des autres classes dont elles ont besoin. Les objets sont "injectés" au moment de la construction ou de l'initialisation de l'instance qui les utilise.

Ce modèle de conception consiste en plusieurs classes ou objets qui collaborent les uns avec les autres. Nous pouvons aussi parler de la notion de dépendance entre les objets.

Il existe plusieurs stratégies pour appliquer ce modèle de conception. Voyons donc de plus prêt les formes d'injection de dépendance existantes en général dans un code orienté objet.

II-A-1 - Hard-coding dependencies :

La classe crée en interne ses propres dépendances. Cette technique est la plus simple pour gérer une dépendance mais elle reste la moins souple de toutes.

Exemple :

```
package
{
    public class Writer
    {

        public function Writer()
        {
        }

        public function write( message:String ):void
        {
            ( new Pen() ).write( message ) ;
        }
    }
}
```

Dans l'exemple précédent la classe **Writer** utilise dans sa méthode **write(message:String)** un objet de type **Pen**. Cette méthode est donc dépendante d'un objet de type **Pen** pour fonctionner.

Le code est en effet très simple mais le fait de créer directement l'objet de type Pen dans la méthode limite très vite le champ d'action de la classe et toutes évolutions de celle-ci par la suite. En cas de changement il faudra modifier directement le code dans la classe ce qui n'est pas forcément très pratique.

II-A-2 - Looking up dependencies :

Cette stratégie nécessite un contexte. La classe appelle un objet et utilise ses méthodes et attributs selon ses besoins. Le contexte est un objet externe qui contient donc toutes les dépendances.

Voici un petit exemple pour illustrer cette implémentation :

```
package
{
    public class Writer
    {
```

```
        public function Writer()
        {
        }

        public function write( message:String ):void
        {
            var context:Tools = new Tools() ;
            context.getPen().write( message ) ;
        }
    }
}
```

II-A-3 - Constructor injection :

L'injection par constructeur est une approche simple et classique. Cette approche du modèle utilise une classe qui possède une fonction constructeur avec des paramètres pour chaque dépendance que l'on souhaite définir.

Les dépendances sont transmises au moment de l'instanciation de la classe.

Exemple :

```
package
{
    public class Writer
    {

        public function Writer( pen:Pen )
        {
            _pen = pen ;
        }

        private var _pen:Pen ;

        public function write( message:String ):void
        {
            _pen.write( message ) ;
        }
    }
}
```

Dans cet exemple la fonction constructeur **Writer(pen:Pen)** permet d'injecter au moment de la création d'une instance de type **Writer** la référence d'un objet de type **Pen** qui permettra à la méthode **write()** de fonctionner correctement.

II-A-4 - Setter Injection :

Dans cette approche la classe possède des propriétés qui lui permettent de définir chacune de ses dépendances (attributs ou méthodes publiques).

Les dépendances sont définies après la création d'une instance de cette classe.

Exemple 1 : *Avec une méthode ou un attribut publique de base.*

```
package
{
    public class Writer
    {

        public function Writer()
```

```
        {
        }

        public var pen:Pen ;

        public function setPen( p:Pen ):void
        {
            this.pen = p ;
        }

        public function write( message:String ):void
        {
            pen.write( message ) ;
        }
    }
}
```

Dans cet exemple la méthode '**setPen(p:Pen)**' permet d'injecter dans les instance de la classe **Writer** tout objet de type **Pen** nécessaire au bon fonctionnement de la méthode **write()**.

Exemple 2 : *Avec un attribut vituel (setter).*

```
package
{
    public class Writer
    {

        public function Writer()
        {
        }

        private var _pen:Pen ;

        public function set pen( p:Pen ):void
        {
            _pen = p ;
        }

        public function write( message:String ):void
        {
            _pen.write( message ) ;
        }
    }
}
```

Dans cet exemple la propriété virtuelle '**pen**' permet d'injecter dans les instance de la classe **Writer** tout objet de type Pen nécessaire au bon fonctionnement de la méthode **write()**.

II-A-5 - Interface Injection :

Cette méthode s'utilise de la même manière que les principes de **Setter Injection** ou de **Constructor** injection mais en typant la dépendance (propriétés, arguments) avec une interface spécifique :

Exemple :

```
package
{
    public class Writer
    {

        public function Writer()
            {
```

```
        }

        private var _pen:IPen ;

        public function get pen():IPen
        {
            return _pen ;
        }

        public function set pen( p:IPen ):void
        {
            _pen = p ;
        }

        public function write( message:String ):void
        {
            (_pen as IPen).write( message ) ;
        }
    }
}
```

II-B - Inversion de contrôle

Le modèle de conception d'Inversion de contrôle, ou plus simplement appelé IoC (L'autre nom du modèle **Dependency injection**) définit un ensemble de techniques de programmation dans lesquels le flux de contrôle du système et de ses objets est inversé par rapport à une technique traditionnelle de création d'une application et de ses interactions.

Le modèle de conception d'inversion de contrôle se base sur le "Hollywood Principle" : "don't call us, we will call you" : « **Ne nous appelez pas, c'est nous qui vous appellerons.** »

Le principe est simple, il indique un framework d'application qui n'a pas besoin d'être appelé pour faire fonctionner l'application, c'est le framework lui même qui s'occupe de créer les interactions entre les objets (dans la limite du possible).

Ce modèle favorise la programmation en "**couches**". Cette programmation permet de séparer chaque élément d'une application et de travailler sur chacun de façon indépendante et ensuite de les lier les uns aux autres très simplement en ayant le moins de dépendances possible au départ. C'est donc au framework de lier chaque couche les unes avec les autres le moment venu en injectant toutes les dépendances dans l'objet qui en aura besoin.

II-C - Conteneurs basés sur le modèle d'inversion de contrôle.

Un conteneur d'inversion de contrôle est une infrastructure d'application qui fournit de nombreux services indispensables pour créer et faire vivre une application. Voici les principaux services d'un conteneur **IoC** classique :

II-C-1 - Lookup :

Le conteneur permet de stocker ou créer des références pour tous les objets de l'application. Le conteneur est donc également considéré comme une "fabrique" (factory) d'application.

II-C-2 - Lifecycle management :

Le cycle de vie des objets dans une application est géré par le conteneur. Le conteneur est capable de créer des nouveaux objets, d'injecter automatiquement des valeurs prédéfinies dans les propriétés d'une instance au moment de sa création, d'invoquer des méthodes spécifiques au moment de l'initialisation et de la destruction des objets, etc.

II-C-3 - Configuration :

Les objets de l'application peuvent être configurés simplement via des données externes sans avoir à recompiler l'application; Il est donc possible d'utiliser des données au format **eden**, **JSON** ou **XML** pour créer la fabrique et initialiser l'application.

II-C-4 - Dependency resolution :

Le conteneur ne se contente pas de gérer et configurer les objets de l'application avec des types simples, il permet aussi de gérer les relations et dépendances que peuvent avoir les objets les uns avec les autres.

II-D - Définition d'objet dans un conteneur IoC.

Une **définition d'objet** est un objet qui va permettre de créer un nouvel objet dans une application avec la **fabrique IoC**. Cette définition d'objet peut être considérée comme un "**mode d'emploi**" défini au préalable par les développeurs de l'application qui utilise ce conteneur.

Ce mode d'emploi contient entre autre :

- un identifiant qui permet d'enregistrer la définition dans un conteneur léger et par la suite de créer des objets,
- le type de l'objet que l'on souhaite créer,
- les paramètres que l'on souhaite passer dans le constructeur,
- les valeurs par défaut que l'on souhaite appliquer sur l'objet (propriétés et méthodes),
- le nom de la méthode que l'on souhaite exécuter une fois l'objet créé pour l'initialiser le rendre actif dans l'application,
- injecter les dépendances avec les autres objets définis eux aussi dans la fabrique,
- etc.

La définition d'objet peut donner d'autres informations sur la stratégie utilisée par la fabrique pour créer l'objet ou sur la nature de cet objet dans le cycle de vie de l'application (singleton, prototype..).

Si l'on devait réduire à sa plus simple expression la définition d'un **conteneur IoC**, nous pourrions dire qu'il se compose d'une collection de **définitions d'objet** qui permettent de structurer efficacement et de façon solide une application tout au long de son cycle de vie.

III - Configuration d'un conteneur IoC.

Pour créer et surtout remplir convenablement un conteneur **IoC**, il est possible d'utiliser tout simplement du code classique, mais sur des applications de moyenne ou grande taille cette technique peut s'avérer rapidement très fastidieuse et peu souple. Surtout qu'à chaque ajout dans le conteneur il faudra la plupart du temps relancer la compilation du programme.

Le modèle de conception d'inversion de contrôle à pour but de proposer des outils simples et d'optimiser efficacement le flux de production d'une application.

Il devient donc rapidement indispensable de compléter celui-ci par une gestion dynamique de son contenu. Dans la plupart des frameworks utilisant un design pattern **IoC** nous retrouvons un déploiement des applications au runtime (après compilation) via des fichiers de configuration externes souvent au format **XML** (exemple avec le framework **Spring.NET** ou **Spring JAVA**).

L'utilisation d'un ou plusieurs fichiers externes de configuration pour gérer une application reste la solution la plus souple. En intervenant après le lancement de l'application, il est possible de modifier le contenu du conteneur sans avoir à recompiler à chaque fois l'application. Nous parlerons alors d'un **"conteneur léger".**

De plus, il est possible avec un moteur de configuration avancé de faire évoluer l'application dynamiquement en fournissant au conteneur uniquement les définitions d'objets nécessaires et en alimentant régulièrement le conteneur avec de nouvelles définitions d'objets au fil du temps selon les besoins.

III-A - Fichier de configuration basé sur le format de données "eden".

Le modèle de conception **IoC AndromedAS** propose une fabrique **IoC** basée sur des fichiers de configuration au format **eden** : **ECMAScript Data Exchange Notation**.

eden est à la base un outil de **sérialisation/désérialisation** d'objets via des données au format texte et sur une notation basée sur la norme **ECMAScript** écrit par **Zwetan Kjukov**. Cette notation permet l'échange de données entre un client et un serveur mais aussi la mise en place de configurations simples ou complexes dans nos applications.

Une des grandes forces du format **eden** est de proposer une structure de données basée sur un format texte très simple à utiliser et que l'on peut *copier/coller* très simplement pour l'utiliser directement dans un code **ActionScript** ou **JavaScript** sans avoir à traiter la chaine de caractère récupérée dans un fichier de texte externe par exemple.

Il peut s'avérer facile de comparer après un bref coup d'□il le format **eden** avec le format **http://www.json.org/jsonfr.html** mais ce dernier reste vraiment pauvre par rapport à toutes les fonctionnalités que propose **eden**.

Le format **eden** accepte de nombreux types :

Array

```
//eden

    a = [ 1, 2, 3 ] ;
    b = new Array() ;
```

Boolean

```
//eden

      b = true;
      c = false;
      d = new Boolean(); // d = false
```

Date

```
//eden

      d1 = new Date();
      d2 = new Date( 1974, 2, 30 );
```

Null

```
//eden

      x1 =null ;// x1 = null
      x2 ;  //x2 = undefined
```

Number

```
//eden
```

```
n1 = 123;
n2 = 1.23;
n3 = 1e5;
n4 = 0xff;
n5 = -123;
n6 = -n2; // n6 = -1.23
n7 = new Number( 100 ); // n7 = 100
```

Object

```
//eden

o1 = { a:1, b:2, c:3 };
o2 = new Object(5); // o2 = 5
o3 = new Object();
```

String

```
//eden

s1 = "hello world";
s2 = "unicode supported ????";
s3 = new String( "hello world" ); // s3 = "hello world"
```

Types customs définis par l'utilisateur

```
//eden

v1 = new Version( 1, 2, 3, 4 );
v2 = new system.Version( 1, 2, 3, 4 );
```

Il permet également les fonctionnalités suivantes :

- Fonctionne avec n'importe quel client ou serveur basé sur la norme **ECMA-262** (**JavaScript**, **ActionScript**, **JScript**, Flash Media Server__, etc.)
- Fonctionne avec la plupart des serveurs de données existant qui manipulent les chaînes de caractères.
- Appel de fonctions et de méthodes sur les objets.
- Gestion des données sécurisées avec isolation des scopes des objets et "white list" d'autorisation pour les types d'objets customs (types d'objets créés par le développeur dans son application).
- Possibilité de définir plusieurs valeurs dans une même chaine de caractère.
- Supporte les commentaires sur une ligne avec le séparateur **//** ou les commentaires multilignes avec les séparateurs **/* */**.

Il serait possible de créer des fichiers de configuration au format **XML** peut être plus classiques et plus "*standard*" si l'on compare les différents **frameworks IoC** actuels. Maintenant je trouve personnellement trop "*verbeux*" et très limitée la notation **XML** qui propose des fichiers de configurations complexes et qui entraine un énorme travail de **parsing** au niveau de l'application pour reproduire une partie infime des possibilités du parseur **eden**. La grande force de **eden** est de conserver dans le document de configuration, le type de tous les objets sans avoir à définir à chaque fois celui-ci.

Voyons par exemple rapidement la différence entre une même définition d'objet définie par un schéma **XML** et un objet au format **eden**.

Prenons une classe simple pour illustrer notre exemple une classe **test.User** très simple écrite en **ActionScript 3** :

```
package test
```

```
{

    /**
     * The User class.
     */
    public class User
    {

        /**
         * Creates a new User instance.
         */
        public function User( pseudo:String = null, name:String = null , address:Address = null )
        {
            this.pseudo  = pseudo ;
            this.name    = name ;
            this.address = address ;
        }

        /**
         * The Address reference of this object.
         */
        public var address:Address ;

        /**
         * The age of the user.
         */
        public var age:Number ;

        /**
          * The city of the user.
          */
        public var city:String ;

        /**
         * The name of the User.
         */
        public var name:String ;

        /**
         * Initialize the User.
         */
        public function initialize():void
        {
            trace( this + " initialize.") ;
        }

    }

}
```

Voyons tout d'abord un exemple de définition d'objet basé sur un schéma **XML** classique proche du DOM **Spring.NET** :

```
<objects>

    <object name="address"  type="test.Address" >
        <constructor-arg index="0" value="34 xxx street"/>
    </object>

    <object
         name        = "user"
         type        = "myPackage.User"
        init-method = "initialize"
    >

        <constructor-arg index="0" value = "ekameleon"/>
        <constructor-arg index="1" value = "ALCARAZ"  />
        <constructor-arg index="2" ref   = "address"  />

        <property name="age"  value="31" />
```

```
        <property name="city"  value="marseille" />

    </object>

</objects>
```

Voyons le même exemple (ou presque) avec une notation **eden** :

```
objects =
[

    {
        id          : "user" ,
        type        : "myPackage.User" ,
        init        : "initialize" ,
        arguments   :
        [
             { value : "ekameleon"} ,
             { value : "ALCARAZ"} ,
             { value : new test.Address("34 xxx street") }
        ]
        ,
        properties :
        [
           { name : "age" , value : 31 } ,
           { name : "city" , value : "marseille" }
        ]
    }

] ;
```

A noter qu'il est possible d'utiliser des objets directement dans le code de vos applications pour alimenter le **conteneur IoC** sans passer par un fichier de configuration au format texte. Il est également possible de créer cet objet via un service web **PHP**, **JAVA**, **Pyhon** et de le renvoyer vers le client via le protocole texte classique (**eden** ou **JSON**) ou le **protocol AMF** (**Action Message Format - Flash Remoting**).

III-B - Autres alternatives avec JSON ?

Même si je vous conseille vivement d'utiliser le format **eden** pour créer vos fichiers de configuration, si vous désirez absolument utiliser un format **JSON** pour alimenter le conteneur **IoC**, dans **VEGAS**, j'ai implémenté une adaptation de la classe **JSON officielle**. **La classe *vegas.string.JSON* offre en plus des fonctionnalités classiques, la possibilité d'utiliser des nombres hexadécimaux *(0xFF...)* et surtout elle permet de désérialiser** sans problème des expressions contenant des objets génériques définis avec ou sans **double-quotes** ou même avec des **simple-quotes** :

```
import vegas.string.JSON ;

JSON.deserialize( { "prop" : 0xFF0000 } ) ;
JSON.deserialize( { 'prop' : 0x00FF00 } ) ;
JSON.deserialize( { prop : 0x0000FF } ) ;
```

Vous pouvez consulter pour plus d'information la documentation de la classe **vegas.string.JSON** dans la référence **AS3** de **VEGAS** et ses extensions.

III-C - Structure et DOM général des fichiers de configuration basés sur le format de donnée "eden".

Voici la structure principale objet d'une configuration pour une fabrique **IoC** dans le **framework AndromedAS** (extension de **VEGAS**) :

```
{

    configuration : // only in the first configuration file
    {
         // ..
    }
    ,
    imports :
    [
        //..
    ]
    ,
    objects :
    [
        //..
    ]
}
```

Il est aussi possible d'écrire plus simplement la structure précédente avec une chaine de caractère au format **eden** avec la structure suivante :

```
configuration =
{
        // only in the first file
} ;

imports =
[
    //..
] ;

objects =
[
    //..
] ;
```

L'attribut "**objects**" est le plus important car il représente la collection (**Array**) de toutes les **définitions d'objets** que l'on souhaite insérer dans le **conteneur IoC**.

Les attributs "**configuration**" et "**imports**" sont facultatifs et sont utilisés via une classe spéciale mais non obligatoire pour utiliser les fonctionnalités de base du conteneur. L'attribut "**configuration**" sera utilisé uniquement dans le fichier principal de configuration utilisé pour lancer l'initialisation de la fabrique dans l'application.

Pour en savoir plus sur ces attributs, vous pouvez consulter l'article "Configuration d'une fabrique IoC".

IV - La classe andromeda.ioc.factory.ECMAObjectFactory

IV-A - Description

La classe ECMAObjectFactory est la classe principale du moteur d'injection de dépendance définie dans le package andromeda.ioc.*.

Elle définit, à elle seule, le conteneur et la fabrique IOC qui sont utilisés par le framework pour gérer le contenu d'une application avec le Design Pattern d'Inversion de Contrôle.

Cette classe possède un héritage riche, avec plusieurs niveaux d'héritage, qui lui procure plusieurs fonctionnalités importantes :

- **ECMAObjectFactory** (Classe principale du framework **IoC** basée sur des **définitions d'objet** au format **ECMAScript**)
- **ObjectFactory**
- **ObjectDefinitionContainer**
- **Action**
- **SimpleAction**
- **CoreEventDispatcher**
- **CoreObject**
- **Object**

Cette classe implémente les interfaces :

- **IObjectDefinitionContainer** : Définit l'ensemble des méthodes permettant de gérer les définitions d'objets dans le conteneur **IoC** (voir classe **ObjectDefinitionContainer**).
- **IObjectFactory** : Définit l'ensemble des méthodes qui permettent la gestion et récupération des objets contenus dans la fabrique **IoC** (voir classe **ObjectFactory**).
- **IFactory** : Définit la méthode create() qui permet de générer un ensemble de définition d'objets nécessaires pour une application donnée (voir classe **ECMAObjectFactory**).
- **IAction** : Définit une notion de "process" sur la fabrique. La classe notifie un évènement au début et à la fin de l'initialisation complète du conteneur.
- **IRunnable** : Définit que le conteneur peut être initialisé en exécutant simplement l'objet comme une commande avec la méthode **run()**.

IV-B - Exemple d'utilisation en ActionScript ("hello world").

Je vais illustrer l'utilisation de la fabrique **IoC "ECMAObjectFactory"** avec une initialisation simple basée sur simple objet de type **Array** qui contient un ensemble d'objets génériques. Ces objets génériques permettent l'initialisation et l'injection de plusieurs **définitions d'objets** et la création automatique de certains objets "**singleton**" pendant l'initialisation du conteneur léger avec certaines **définitions d'objets**.

L'exemple qui va suivre peut être testé aussi bien dans **Flash CS3** que dans une application **Flex** ou même une application basée sur la technologie **AIR**. Cet exemple se base sur un bout de code saisi dans un calque sur la scène principale d'une animation dans **Flash** mais il est très simple d'adapter ce bout de code pour l'utiliser dans un projet **AS3** basé sur une classe "**main**" d'application.

```
import andromeda.ioc.factory.ECMAObjectFactory ;

var objects:Array =
[
    {
        id         : "my_format" ,
        type       : "flash.text.TextFormat" ,
        arguments : [ { value:"arial" } , { value:24 } , { value:0xFEF292 } , { value:true } ]
    }
    ,
    {
        id          : "my_field" ,
        type        : "flash.text.TextField"  ,
        properties :
        [
            { name : "autoSize"          , value  : "left"            } ,
            { name : "defaultTextFormat" , ref    : "my_format"       } ,
            { name : "text"              , value  : "HELLO WORLD"     } ,
            { name : "x"                 , value  : 10                } ,
            { name : "y"                 , value  : 10                }
        ]
    }
    ,
    {
        id              : "root" ,
        type            : "flash.display.MovieClip"  ,
```

```
        factoryReference : "#root" ,
        singleton        : true ,
        properties       :
        [
            { name : "addChild" , arguments : [ { ref:"my_field" } ]}
        ]
    }
] ;


var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.config.root = this ;

factory.create( objects ) ;
```

L'objet "**objects**" de type **Array** de configuration ci-dessus permet de définir 3 **définitions d'objet** :

- "**my_format**" : cette définition permet de créer des instances de types **flash.text.TextFormat**, cette définition injecte des dépendances via la fonction constructeur de la classe pour initialiser l'objet.
- "**my_field**" : cette définition permet de créer des champs de texte dynamiques de type **flash.text.TextField** avec une initialisation de certains de ces attributs avec des injections de dépendance via l'attribut "**properties**".
- "**root**" : cette définition permet de créer une référence **singleton** qui cible la scène principale de l'application définie dans la configuration de la fabrique **IoC** (voir code ci-dessous) et d'attacher dessus le champ de texte dynamique "**my_field**".

A noter que la définition d'objet "**root**" utilise une stratégie spéciale de la **fabrique IoC** qui lui permet de cibler dans une définition d'objet une référence de la scène principale définie dans le code ci-dessus avec la propriété **root** de l'objet de config de la fabrique :

```
factory.config.root = this ;
```

*La **fabrique IoC** possède plusieurs types de stratégies pour créer un objet dans l'application avec une **définition d'objet**. Voir l'article consacré aux définitions d'objets.*

L'exemple ci-dessus permet d'afficher très rapidement un champ de texte sur la scène d'une animation. Cet exemple nous montre bien l'utilisation d'une "**programmation en couches**" qui permet d'isoler chaque objet d'une application et de travailler tranquillement sur chacun d'eux sans se soucier de leurs dépendances les uns avec les autres. C'est le conteneur léger et le framework qui se chargent de créer les objets et de les mettre en relations les uns avec les autres

Nous reprendrons dans le prochain chapitre cet exemple pour illustrer la technique utilisée pour chargée ce contexte de configuration avec un fichier au format texte externe (au format **eden**).

IV-C - Inspection des méthodes de la classe ObjectDefinitionContainer, base du conteneur IOC (interface IObjectDefinitionContainer)

Voici l'ensemble des méthodes et propriétés définies par l'interface **IObjectDefinitionContainer** implémentée par la classe **EdenObjectFactory**.

Voici un exemple simple d'utilisation des méthodes définies ci-dessus :

```
import flash.text.TextField ;
import flash.text.TextFormat ;

import andromeda.ioc.core.ObjectDefinition ;
import andromeda.ioc.core.ObjectDefinitionContainer ;
import andromeda.ioc.factory.ObjectFactory ;
```

```
var container:ObjectDefinitionContainer = new ObjectFactory();

var context:Object =
{
    id         : "my_field" ,
    type       : "flash.text.TextField" ,
    properties :
    [
        { name : "defaultTextFormat" , value : new TextFormat("verdana", 11) } ,
        { name : "selectable"        , value : false                          } ,
        { name : "text"              , value : "hello world"                  } ,
        { name : "textColor"         , value : 0xF7F744                       } ,
        { name : "x"                 , value : 100                            } ,
        { name : "y"                 , value : 100                            }
    ]
}

var definition:ObjectDefinition = ObjectDefinition.create( context ) ;

container.addObjectDefinition( definition );

trace( container.containsObjectDefinition( "my_field" ) ) ; // true
trace( container.getObjectDefinition( "my_field" ) ) ; // [ObjectDefinition]
trace( container.sizeObjectDefinition() ) ; // 1

var field:TextField = (container as ObjectFactory).getObject("my_field") as TextField ;

addChild(field) ;
```

Dans cet exemple nous initialisons une **définition d'objet** avec un objet générique simple qui permettra de générer avec la fabrique une instance de la classe **TextField** spécifique qu'il sera très simple ensuite d'attacher sur la scène. Cette technique simple permet d'injecter à tout moment des nouvelles définitions d'objets à la main dans la fabrique avec la méthode **addObjectDefinition()**.

IV-D - Inspection des propriétés et méthodes de la classe ObjectFactory, base de la fabrique IOC (interface IObjectFactory)

Voici l'ensemble des méthodes et propriétés définies par l'interface **IObjectFactory** implémentée par la classe **EdenObjectFactory** (hérite de la classe **ObjectFactory**). La classe ObjectFactory reste de plus bas niveau et permet à ceux qui le désirent de garder le moteur **IoC** de **AndromedAS** en utilisant un parseur **XML** ou autre pour remplir la fabrique.

IV-E - Inspection des propriétés et méthodes de la classe ECMAObjectFactory.

Voici l'ensemble des méthodes statiques définies dans la classe **EdenObjectFactory**.

Cette classe contient le moteur d'initialisation de la fabrique. Le conteneur interprète une collection de type **Array** contenant des **objets génériques** simples (format **ECMAScript**) et peut ainsi créer ses propres **définitions d'objets** de type **ObjectDefinition**.

La classe **ECMAObjectFactory** contient aussi un ensemble de méthodes statiques (voir tableau ci-dessus) qui permettent de créer plusieurs instances globales dans une application.

IV-F - Plusieurs fabriques IoC dans vos application.

La classe **EdenObjectFactory** peut être instanciée tout simplement avec le mot clé **new** mais si nous regardons de plus prêt nous pouvons oberver dans sa signature qu'elle implémente l'interface **vegas.core.Identifiable**. Cette interface définie l'existance d'un attribut "**id**" qu'il est possible de définir sur toutes les instances de la classe.

De façon générale, ce petit attribut peut être pratique mais il trouve tout son intérêt si l'on utilise la fonctionnalité "**multi-singleton**" de la classe. Cette notion permet de définir plusieurs références globales uniques dans l'application de type **ECMAObjectFactory**. On peut ainsi utiliser dans une même application plusieurs modules définis par des fabriques **IoC** totalement indépendantes.

La classe **EdenObjectFactory** contient donc une méthode statique **getInstance(id:String=null)** qui permet de créer et renvoyer des références uniques de la classe en fonction d'un identifiant spécifique et lui aussi unique. La classe contient d'autres méthodes statiques permettant de supprimer complètement un singleton de la classe en mémoire mais aussi de vérifier si pour un identifiant donné, un **singleton** de la classe existe.

Exemple :

```
import andromeda.ioc.factory.ECMAObjectFactory;

var factory:ECMAObjectFactory;

factory = ECMAObjectFactory.getInstance() ;

trace( "ECMAObjectFactory.getInstance() : " + factory + " with the default id : " + factory.id ) ;

factory = ECMAObjectFactory.getInstance("factory_one") ;

trace( "ECMAObjectFactory.getInstance('factory_one')      : " + factory + " with the default id : " +
 factory.id ) ;

trace( "ECMAObjectFactory.containsInstance('factory_one') : " +
 ECMAObjectFactory.containsInstance( "factory_one" ) ) ;

trace( "ECMAObjectFactory.removeInstance('factory_one')   : " +
 ECMAObjectFactory.removeInstance( "factory_one" ) ) ;

trace( "ECMAObjectFactory.containsInstance('factory_one') : " +
 ECMAObjectFactory.containsInstance( "factory_one" ) ) ;
```

V - La classe asgard.net.ECMAObjectLoader.

V-A - Définition et exemple de base.

La classe **ECMAObjectLoader** est une classe un peu particulière car elle ne se trouve pas directement dans l'extension **AndromedAS**. En effet pour utiliser cette classe il faut la cibler dans le package **asgard.net.*** de l'extension **ASGard** de **VEGAS**.

Cette classe est utilisée pour charger un ou plusieurs fichiers externes qui permettront d'initialiser la fabrique **IoC**.

Exemple :

```
import asgard.net.ECMAObjectLoader ;

var loader:ECMAObjectLoader = new ECMAObjectLoader( "application.eden" , "context/" ) ;

loader.run() ;
```

Toute instance de la classe **ECMAObjectLoader** contient par défaut une référence vers la référence **singleton** définie par défaut de la classe **ECMAObjectFactory** mais il est possible de modifier cette référence avec tout autre objet du même type.

La classe **ECMAObjectLoader** utilise également par défaut un chargeur de données externe basé sur un parsing **eden** avec la classe **asgard.net.EdenLoader**. Vous pouvez en cas de besoin modifier ce chargeur pas un autre.

Par exemple dans le cas d'un chargement de données via un fichier texte au format **JSON** vous pouvez utiliser la classe **asgard.net.JSONLoader**.

Vous pouvez créer vos propres **chargeurs/parseurs/désérialisations** de données avec un format **XML** ou autre en créant votre propre classe de chargement qui héritera de la classe **asgard.net.ParserLoader**.

La classe **ECMAObjectLoader** est un objet basé sur le modèle de commandes et d'actions de **AndromedAS**, elle implémente l'interface **andromeda.process.IAction**. Une fois les fichiers de configurations chargées la classe **ECMAObjectLoader** remplit le conteneur d'**Inversion de Contrôle**.

V-B - Exemple d'utilisation : "hello world".

Illustrons l'utilisation de la fabrique IoC "**ECMAObjectFactory**" avec son chargeur de configurations externes "**ECMAObjectLoader**" avec un exemple basique, en créant dynamiquement un champ de texte dans une animation au format **SWF**.

Tout d'abord, nous allons créer un fichier au format texte (**UTF8**) externe de configuration avec la chaîne de caractères (format **eden**) suivante :

```
objects =
[
    {
        id         : "my_format" ,
        type       : "flash.text.TextFormat" ,
        arguments  : [ { value:"arial" } , { value:24 } , { value:0xFEF292 } , { value:true } ]
    }
    ,
    {
        id          : "my_field" ,
        type        : "flash.text.TextField"  ,
        properties :
        [
            { name : "autoSize"          , value  : "left"            } ,
            { name : "defaultTextFormat" , ref    : "my_format"       } ,
            { name : "text"              , value  : "HELLO WORLD"     } ,
            { name : "x"                 , value  : 10                } ,
            { name : "y"                 , value  : 10                }
        ]
    }
    ,
    {
        id               : "root" ,
        type             : "flash.display.MovieClip"  ,
        factoryReference : "#root" ,
        singleton        : true ,
        properties       :
        [
            { name : "addChild" , arguments  : [ { ref:"my_field" } ]}
        ]
    }
] ;
```

Le fichier de configuration externe permet de définir 3 définitions d'objet :

- "**my_format**" : cette définition permet de créer des instances de types **flash.text.TextFormat,** cette définition injecte des dépendances via la fonction constructeur de la classe pour initialiser l'objet.
- "**my_field**" : cette définition permet de créer des champs de texte dynamiques de type **flash.text.TextField** avec une initialisation de certains de ces attributs avec des injections de dépendance via l'attribut "**properties**".

- "**root**" : cette définition permet de créer une référence **singleton** qui cible la scène principale de l'application définie dans la configuration de la fabrique **IoC** (voir code ci-dessous) et d'attacher dessus le champ de texte dynamique "**my_field**".

*le nom d'extension **".eden"** n'est pas obligatoire, il suffit que le fichier externe contienne juste du texte au format eden pour que le chargeur fasse correctement son travail. Il est donc possible d'utiliser des fichiers portant l'extension **.txt** par exemple, même si il est tout de même plus simple d'utiliser une extension différente pour distinguer correctement vos fichiers de configuration au format **eden** par rapport à d'autres.*

Il ne reste plus qu'à charger ce fichier de configuration et de remplir un conteneur IoC avec la classe **ECMAObjectLoader** :

```
import andromeda.events.ActionEvent;

import asgard.net.ECMAObjectLoader ;

var debug:Function = function( e:Event ):void
{
     trace( e ) ;
}

var loader:ECMAObjectLoader = new ECMAObjectLoader( "hello_world.eden" , "context/" ) ;

loader.root = this ; // to use the "#root" expression in the 'ref' attribute.

loader.addEventListener( ActionEvent.START  , debug ) ;
loader.addEventListener( ActionEvent.FINISH , debug ) ;

loader.run() ;
```

L'exemple ci-dessus illustre :

- L'utilisation de plusieurs définitions d'objets chargées dans un fichier externe.
- La notion de dépendance entre une définition d'objet et une autre
- La notion de dépendance avec une référence "magique" (#root) qui permet de cibler la référence d'un objet spécial défini dans l'application.
- La méthode pour créer un objet visuel (ici un champ de texte) directement avec la fabrique sans appeler sa définition dans le code ActionScript.

Cet exemple très basique dévoile quelques fonctionnalités disponibles dans la moteur **IoC** de **AndromedAS**. Nous chargeons un fichier texte encodé en **UTF8** "**hello_world.eden**" défini dans un répertoire "**context/**" situé au même niveau que le **swf** de l'application de test.

Le constructeur de la classe **ECMAObjectLoader** permet de définir le nom complet du fichier de configuration que l'on souhaite charger mais aussi le répertoire principal dans lequel il se trouve. Si aucun paramètre n'est défini dans la fonction constructeur de la classe alors le chargeur essaie de charger un fichier "**application.eden**" situé au même niveau que le **swf** de l'application.

Ce fichier texte est ensuite parsé par le moteur de désérialization **eden** puis injecté automatiquement dans le fabrique IoC principale de l'application définie dans la classe **ECMAObjectLoader** par défaut :

```
import andromeda.ioc.factory.ECMAObjectFactory ;
import andromeda.ioc.factory.IObjectFactory ;

import asgard.net.ECMAObjectLoader ;

var loader:ECMAObjectLoader = new ECMAObjectLoader() ;
```

```
var factory:IObjectFactory = loader.factory ;

trace( factory == ECMAObjectFactory.getInstance() ) ; // true
```

Il est bien entendu possible de modifier cette référence par une autre de type **IObjectFactory** à tout moment dans l'application mais il est tout de même préférable de tenter ce genre d'opérations en maitrisant parfaitement le moteur **IoC** de **AndromedAS**.

La classe **ECMAObjectLoader** possède un modèle évènementiel simple basée sur le moteur d'**Action** et process de **AndromedAS** qui lui permet de notifier l'utilisateur du début et de la fin de l'initialisation du conteneur léger en passant par un "**helper**" de type **ECMAObjectLoader**.

```
import andromeda.events.ActionEvent;

import asgard.net.ECMAObjectLoader ;

var debug:Function = function( e:Event ):void
{
     trace( e ) ;
}

var loader:ECMAObjectLoader = new ECMAObjectLoader( "hello_world.eden" , "context/" ) ;

loader.addEventListener( ActionEvent.START  , debug ) ;
loader.addEventListener( ActionEvent.FINISH , debug ) ;
```

Il existe une multitude de fonctionnalités disponibles dans les définitions d'objets des conteneurs léger définis dans AndromedAS. Nous allons détailler ces fonctions, stratégies et options dans la prochaine grosse partie de cet article sur le **Design Pattern d'injection de contrôle** implémenté dans l'extension **AndromedAS** de **VEGAS**.