VEGAS - Design Pattern d'inversion de contrôle : Définitions d'objets

par Marc Alcaraz (Mon site)

Date de publication : 11/09/2009

Dernière mise à jour :

Ce nouvel article essaiera de dévoiler une part importante des secrets que renferme l'implémentation du moteur d'IoC de **VEGAS**.

Je vais vous y parler tout particulièrement des **définitions d'objets** et surtout de tous les attributs qui vont permettre de les configurer. Une bonne maitrise des définitions d'objets permet d'aller très loin dans la conception d'une RIA basée sur un moteur **IoC**.

Pour ceux qui découvrent le monde merveilleux du design pattern d'**Inversion de contrôle** avec **VEGAS** et pour les autres aussi, je vous invite à lire attentivement (si ce n'est pas encore fait) le premier article sur le design pattern IoC : **VEGAS - Design Pattern d'inversion de contrôle : Généralités**

I - Introduction....3
I-A - Résumé des différentes parties de ce chapitre et des attributs qui seront traités à l'interieur.... 4
I-A-1 - Les attributs de base d'une définition d'objet....4
I-A-2 - Cycle de vie d'un objet configuré avec une définition d'objet.... 4
I-A-3 - Les différentes stratégies utilisées par la fabrique pour créer un objet avec une définition d'objet....4
I-A-4 - L'attribut "evaluators"....4
I-B - Exemple général de cet article....4
II - Description des attributs de base d'une définition d'objet.... 8
II-A - L'attribut "id".... 8
II-B - L'attribut "type".... 9
II-C - L'attribut "arguments"....10
II-C-1 - L'attribut "value"....12
II-C-2 - L'attribut "ref".... 12
II-C-3 - L'attribut "config"....13
II-C-4 - L'attribut "locale"....13
II-C-5 - L'attribut "properties".... 14
II-D - L'attribut "properties"....14
II-D-1 - Utilisation de l'attribut "value"....14
II-D-2 - Utilisation de l'attribut "ref".... 14
II-D-3 - Utilisation de l'attribut "config"....15
II-D-4 - Utilisation de l'attribut "locale"....15
II-D-5 - Définition d'une propriété de type Function et utilisation de l'attribut "arguments"....15
II-D-6 - L'attribut "evaluators"....16
II-E - Les attributs "scope", "singleton"et "lazyInit".... 16
II-E-1 - L'attribut "scope"....16
II-E-2 - L'attribut "singleton".... 17
II-E-3 - L'attribut "lazyInit".... 18
II-F - L'attribut "listeners"....18
III - Cycle de vie d'un objet configuré avec une définition d'objet....21
III-A - Les attributs "init" et "destroy".... 21
III-B - L'attribut "identify"....22
III-C - L'attribut "lock".... 23
III-D - Les attributs "dependsOn" et "generates".... 28
IV - Les différentes stratégies utilisées par la fabrique pour créer un objet avec une définition d'objet.... 30
Introduction.... 30
IV-B - Utilisation de l'attribut "factoryMethod".... 31
IV-C - Utilisation de l'attribut "factoryProperty".... 33
IV-D - Utilisation de l'attribut "factoryReference".... 35
IV-E - Utilisation de l'attribut "factoryValue".... 37
IV-F - Utilisation de l'attribut "staticFactoryMethod"....37
IV-G - Utilisation de l'attribut "staticFactoryProperty"....39
V - Les attributs "evaluators"....40
V-A - Description et utilisation de l'interface Evaluable.... 40
V-A-1 - La classe system.evaluators.DateEvaluator.... 41
V-A-2 - La classe system.evaluators.EdenEvaluator.... 41
V-A-3 - La classe system.evaluators.MathEvaluator....41
V-A-4 - La classe system.evaluators.MultiEvaluator.... 41
V-A-5 - La classe system.evaluators.PropertyEvaluator.... 42
V-B - Les implémentations de l'interface Evaluable dans le package andromeda.ioc.*.... 42
V-B-1 - La classe ConfigEvaluator.... 43
V-B-2 - La classe LocaleEvaluator....43
V-B-3 - La classe ReferenceEvaluator.... 44
V-B-4 - La classe TypeEvaluator....45
V-C - Utilisation des objets Evaluable dans les définitions d'objets.... 46
V-C-1 - Évaluation dans une fonction constructeur....48
V-C-2 - Évaluation dans l'attribut "properties".... 48
V-C-3 - Évaluation dans les attributs "factoryMethod" et "staticFactoryMethod"....50

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I - Introduction

Nous avons vu dans l'**article précédent** que les conteneurs **IoC** contiennent des **définitions d'objet** qui sont des "**modes d'emploi**" qui permettent de créer des objets dans une application. Elles définissent aussi les relations de dépendances que peuvent avoir les objets et les classes les uns avec les autres.

Dans le **modèle IoC** de **AndromedAS**, Les **définitions d'objets** sont représentées par des objets qui implémentent l'interface IObjectDefinition, cette interface est implémentée par la classe ObjectDefinition.

L'interface **IObjectDefinition** possède les caractéristiques suivantes :

- Un identifiant unique pour chaque **définition d'objet** (obligatoire).
- Une description du type de l'objet qui sera renvoyé ou créé avec la **définition d'objet** (obligatoire).
- Les références vers les **dépendances** (ou **collaborateurs**) de l'objet.
- Tous les éléments qui permettront d'initialiser l'objet au moment de sa création.
- Tous les éléments qui permettront de contrôler le comportement d'un objet dans l'application : scope de l'objet, méthodes invoquées tout au long de son cycle de vie, etc.

Les concepts énumérés ci-dessus correspondent à un ensemble d'éléments contenus dans des **objets génériques** simples qui permettent de créer et d'initialiser les **définitions d'objet** d'un conteneur **IoC**.

La classe **ObjectDefinition** contient une méthode statique **create(init:Object)** qui lui permet de générer des nouvelles instances très simplement en passant en paramètre un **objet générique** d'initialisation compatible avec les attributs qui seront définis par la suite dans ce chapitre. En général il sera toujours plus simple d'utiliser cette méthode que d'initialiser à la main toutes les instances de types **ObjectDefinition** d'un conteneur **IoC**.

Exemple : Créer une définition d'objet avec un objet d'initialisation dynamique simple

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

import andromeda.ioc.core.ObjectDefinition ;

var init:Object =
{
    id         : "my_field" ,
    type       : "flash.text.TextField" ,
    singleton  : true ,
    properties :
    [
        { name : "defaultTextFormat" , value : new TextFormat("verdana", 11) } ,
        { name : "selectable"        , value : false                          } ,
        { name : "text"              , value : "hello world"                  }
    ]
}

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

trace( "definition.id              : " + definition.id ) ;
trace( "definition.getType()       : " + definition.getType() ) ;
trace( "definition.getScope()      : " + definition.getScope() ) ;
trace( "definition.isSingleton()   : " + definition.isSingleton() ) ;
trace( "definition.isLazyInit()    : " + definition.isLazyInit() ) ;
trace( "definition.getProperties() : " + definition.getProperties() )
```

Ces **objets génériques** contiennent différents attributs. Nous allons étudier ces attributs dans 4 parties distinctes.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I-A - Résumé des différentes parties de ce chapitre et des attributs qui seront traités à l'interieur.

I-A-1 - Les attributs de base d'une définition d'objet

Aller directement à cette partie

I-A-2 - Cycle de vie d'un objet configuré avec une définition d'objet

Aller directement à cette partie

I-A-3 - Les différentes stratégies utilisées par la fabrique pour créer un objet avec une définition d'objet

Aller directement à cette partie

I-A-4 - L'attribut "evaluators"

Aller directement à cette partie

I-B - Exemple général de cet article

Nous allons définir pour la suite de l'article un exemple de base qui illustrera comment définir une collection (**Array**) d'objets génériques qui servirons à initialiser une fabrique **IoC**. Cette collection pourra par la suite être stockée dans un fichier texte externe au format **eden** ou directement dans le code source **ActionScript** de l'application.

Prenons le temps de créer une petite classe **test.User**, cette classe permet d'illustrer les dépendances que peut avoir une classe avec d'autres classes.

La classe test.User

```
package test
{
    import system.Reflection;

    /**
     * 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.
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La classe test.User

```
         */
        public var age:Number ;

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

        /**
         * The job of this User.
         */
        public var job:Job ;

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

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

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

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

        /**
         * Destroy the user.
         */
        public function destroy():void
        {
            trace( this + " destroy.") ;
        }

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

        /**
         * Sets the mail value of this user.
         */
        public function setMail( sMail:String ):void
        {
            mail = sMail ;
        }

        /**
         * Returns the string representation of the object.
         * @return the string representation of the object.
         */
        public override function toString():String
        {
            return "[" + Reflection.getClassName(this) + ( pseudo != null ? " " + pseudo : "" ) + "]" ;

        }
    }

}
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Nous observons dans la classe **test.User** les notions de dépendances expliquées dans le chapitre précédent au niveau de sa fonction constructeur, ses attributs et ses méthodes.

La classe **test.User** possède de nombreuses dépendances, nous pourront noter en particulier une dépendance avec les classes **test.Address** et **test.Job** définies ci-dessous :

La classe test.Address

```
package test
{

    /**
     * The Address class
     */
    public class Address
    {

        /**
         * Creates a new Address instance.
         */
        public function Address( city:String = null , street:String = null , zip:Number = NaN )
        {
            this.city    = city ;
            this.street  = street ;
            this.zip     = zip    ;
        }

        /**
         * The city of this address.
         */
        public var city:String ;

        /**
         * The street of this address.
         */
        public var street:String ;

        /**
         * The zip code of this address.
         */
        public var zip:Number ;

    }

}
package test
{
    import system.Reflection;

    /**
     * The Job class.
     */
    public class Job
    {

        /**
         * Creates a new Job instance.
         */
        public function Job( name:String = null )
        {
            this.name = name ;
        }

        /**
         * The name of this Job.
         */
        public var name:String ;

        /**
         * Returns the string representation of this object.
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La classe test.Address

```
         * @return the string representation of this object.
         */
        public override function toString():String
        {
            return "[" + Reflection.getClassName(this) + " " + name + "]" ;
        }

    }

}
```

Voyons maintenant un exemple de configuration **IoC** contenant l'ensemble des **définitions d'objet** au format **ECMAScript** (objets génériques) qui permettra ensuite d'initialiser un conteneur léger **IoC** et de créer un objet de type **test.User** :

```
objects =
[
    {
        id            : "address"  ,
        type          : "test.Address" ,
        properties    :
        [
            { name : "city"   , value : "Marseille" } ,
            { name : "street" , value : "xx xxx xxxxxxxxxxx" } ,
            { name : "zip"    , value : 13001 }
        ]
    }
    ,
    {
        id         : "job_dev"   ,
        type       : "test.Job"  ,
        scope      : "prototype" ,
        properties : [ { name:"name" , value:"AS Developper" } ]
    }
    ,
    {
        id         : "user" ,
        type       : "test.User" ,
        arguments    :
        [
            { value : "eKameleon" } ,
            { value : "ALCARAZ"   } ,
            { ref   : "address"   }
        ] ,
        lazyInit      : true ,
        singleton     : true ,
        destroy       : "destroy" ,
        init          : "initialize" ,
        properties    :
        [
             { name : "age"       , value : 31        } ,
             { name : "firstName" , value : "Marc"    } ,
             { name : "job"       , ref   : "job_dev" } ,
             { name : "url"       , value : "http://www.ekameleon.net" }

             { name : "setMail" , arguments : [ { value : "ekameleon [at] gmail.com" } ] }
        ]
    }
] ;
```

L'exemple précédent n'utilise pas tous les attributs définis au début de ce chapitre mais illustre les fonctionnalités les plus importantes d'une **définition d'objet.**

Il ne reste plus qu'à initialiser la fabrique **IoC** et récupérer l'objet de type **test.User** défini dans celle-ci dans un bout de code **ActionScript** très simple :

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

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

import test.User ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( objects ) ;

var user:User = factory.getObject( "user" ) as User ;

trace( user ) ;
```

Nous utiliserons en grande partie cet exemple dans les descriptions des attributs qui vont suivre même si, parfois, il nous faudra sortir des sentiers battus pour expliquer avec d'autres exemples certaines notions importantes.

II - Description des attributs de base d'une définition d'objet.

II-A - L'attribut "id"

Cet identifiant est une **clé unique** obligatoire utilisée pour obtenir la référence d'un objet défini dans le conteneur **IoC**.

Exemple

```
id  : "user" ,
```

Regardons maintenant comment utiliser l'identifiant unique d'une **définition d'objet** pour créer et obtenir avec la fabrique l'objet correspondant :

```
var user:User = factory.getObject( "user" ) ;

trace( user ) ;
```

L'attribut **"id"** est indispensable pour créer un objet avec la méthode **getObject()** mais aussi pour abonner une nouvelle **définition d'objet** dans la fabrique.

Petite astuce :

J'utilise souvent dans mes projets des classes contenant des énumérations sous forme de constantes statiques pour lister l'ensemble des identifiants importants utilisés par mes **définitions d'objets** dans mes fabriques **IoC**. J'utilise cette technique surtout pour les **définitions d'objets** ayant un scope **singleton**.

Par exemple, nous pouvons créer une énumération des vues principales d'une application en créant rapidement une petite classe **DisplayList** :

```
package test
{

    public class DisplayList
    {

         public static const BACKGROUND:String = "background" ;

         public static const BODY:String       = "body" ;

         public static const MENU:String       = "menu" ;

     }
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
}
```

Ensuite nous pouvons créer un contexte de configuration **IoC** avec les **définitions d'objets** suivantes :

```
objects =
[

    {
        id        : "background"  ,
        type      : "flash.dislay.Sprite" ,
        singleton : true
    }
    ,
    {
        id        : "body"  ,
        type      : "flash.dislay.Sprite" ,
        singleton : true
    }
    ,
    {
        id        : "menu"  ,
        type      : "flash.dislay.Sprite" ,
        singleton : true
    }
] ;
```

Pour appeler, à tout moment, un des **singletons** définis ci-dessus, il suffira de taper le code suivant :

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

import flash.display.Sprite ;

import test.DisplayList ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( objects ) ;

var background:Sprite = factory.getObject( DisplayList.BACKGROUND ) as Sprite ;
```

II-B - L'attribut "type"

Cet attribut est obligatoire et indique en général le nom complet de la **classe** (ou **type**) de l'objet que l'on souhaite créer ou cibler dans la fabrique (cas d'un objet "**singleton**" par exemple).

```
type  : "test.User" ,
```

Il faut s'assurer que la classe existe dans le fichier principal de l'application. Si cette classe n'existe pas, il est possible d'inclure cette classe (ou les éléments graphiques, les sons, etc.) en utilisant le moteur de chargement de ressources externes de la **fabrique IoC** courante. Sinon il faudra forcer l'existence de cette classe en l'appelant au moins une fois dans le code principal de l'application.

A noter que l'expression (**String**) de cet attribut peut être soumise à des filtrages et formatages en utilisant des outils câblés dans la configuration de la fabrique, voir à ce sujet le chapitre C-1.7. Ces évaluations et filtrages permettent de simplifier grandement l'écriture de vos **définitions d'objets**.

Remarque :

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Si nous utilisons dans une **définition d'objet** une **stratégie de création** différente de celle utilisée par défaut dans le conteneur léger, l'objet généré à partir de cette **définition d'objet** peut avoir un type "**différent**" de celui défini dans la **définition d'objet** mais devra avoir un "**casting**" compatible avec ce type établi avec l'attribut "**type**".

Il faut donc que l'objet renvoyé par la fabrique hérite ou implémente le type défini dans sa **définition d'objet**. Le mot clé "**as**" est utilisé dans la fabrique juste après la création d'un nouvel objet et il assure que l'objet généré possède bien le bon type. En cas de problème et d'incompatibilité sur le typage défini dans la **définition d'objet** avec le type de l'objet créé par la fabrique, le conteneur renverra un objet "**null**".

II-C - L'attribut "arguments"

Dans les **définitions d'objets** l'attribut "**arguments**" est une collection (**Array**) d'**objets génériques** qui permet de définir les paramètres que l'on souhaite passer dans la fonction constructeur d'une classe ou dans une méthode.

Cet attribut peut être utilisé à la racine d'une **définition d'objet**, alors il correspond à la liste des paramètres de la fonction constructeur de la classe utilisée pour créer l'objet, sinon (exemple dans l'attribut "**properties**") il correspondra à la liste des arguments que l'on souhaite passer dans une méthode.

Cet attribut n'est pas obligatoire, toute fonction constructeur ou toute méthode définie sans cet attribut sera invoquée sans paramètre.

Dans **AndromedAS**, cet attribut permet de créer et d'initialiser des objets de type **andromeda.ioc.core.ObjectArgument**.

Cette classe contient essentiellement les attributs suivants :

- **value** : Correspond à la valeur ou référence de l'argument. Cet attribut peut être soit directement la valeur de l'argument, soit une expression (**String**) qui sera interprété par la fabrique en fonction de la valeur de l'attribut "**policy**" de l'instance.
- **policy** : Permet de définir la stratégie utilisée par la fabrique pour créer l'argument, il détermine si l'argument est défini avec une valeur simple, une référence vers une autre **définition d'objet**, etc.
- **evaluators** : (optionnel) Permet de définir dans un cadre bien particulier une collection d'objects d'évaluation qui sera appliquée pour filtrer la valeur ou référence définie dans l'objet de type **ObjectArgument**.

La classe **ObjectArgument** contient comme la classe **ObjectDefinition** une méthode statique **create(init:Object)** qui lui permet de générer simplement une liste (**Array**) d'objet de type **ObjectArgument** en l'initialisant avec liste (**Array**) d'**objets génériques**. Tous les **objets génériques** seront transformés en instance de type **ObjectArgument**.

Exemple

```
import andromeda.ioc.core.ObjectArgument ;
import andromeda.ioc.core.ObjectAttribute ;

var args:Array =
[
    { value  : "hello world"  } ,
    { ref    : "my_id"        } ,
    { config : "config.value" } ,
    { locale : "locale.value" } ,

    { value  : "hello world" , evaluators:["myEvaluator"] }
] ;

args = ObjectArgument.create( args ) ; // transform all arguments with the ObjectArgument factory.

trace( args[0] is ObjectArgument ) ; // true

trace( args[0].policy == ObjectAttribute.VALUE     ) ; // true
trace( args[1].policy == ObjectAttribute.REFERENCE ) ; // true
trace( args[2].policy == ObjectAttribute.CONFIG    ) ; // true
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Exemple

```
trace( args[3].policy == ObjectAttribute.LOCALE    ) ; // true

trace( args[4].evaluators ) ; // myEvaluator
```

Les **objets génériques** doivent absolument être définis avec un attribut obligatoire spécifique pour être transformé en objet de type **ObjectArgument** sinon il ne sera pas pris en compte dans la méthode statique **create()** et sera remplacé par un objet **null**.

Nous pouvons observer dans l'exemple ci-dessus que tous les **objets génériques** sont définis avec un des attribut ci-dessous :

- **value**
- **ref**
- **config**
- **locale**

Ces attributs doivent correspondrent aux valeurs possibles de l'attribut **policy** de la classe **ObjectArgument** définies dans la classe **ObjectAttribute** avec les valeurs suivantes :

- **ObjectAttribute.VALUE : "value"**
- **ObjectAttribute.REFERENCE : "ref"**
- **ObjectAttribute.CONFIG : "config"**
- **ObjectAttribute.LOCALE : "locale"**

Chacun des types ci-dessus donnera l'information qu'il faut au conteneur pour générer l'argument en utilisant la stratégie nécessaire. Ces types sont stockés dans la propriété **policy** de chaque objet de type **ObjectArgument** renvoyé par la méthode statique **create()** de la classe.

Reprenons l'exemple principal de ce chapitre en isolant dans la **définition d'objet** "**user**" l'attribut principal "**arguments**" :

```
{

    id          : "user" ,
    type        : "test.User" ,
    arguments   :
    [
        { value : "eKameleon" } ,
        { value : "ALCARAZ"   } ,
        { ref   : "address"   }
    ]
}
```

Comme nous le voyons ci-dessus, les arguments sont définis par des objets génériques qui contiennent en général un attribut "**value**" ou un attribut "**ref**". Un seul de ces attributs peut être utilisé pour un même objet générique. Ces attributs permettent de définir la valeur d'un argument ou permettent de cibler une référence en ciblant une autre **définition d'objet** dans le conteneur **IoC**.

L'exemple précédent correspond au code **ActionScript** suivant :

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

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

var user:User = new User( "eKameleon" , "ALCARAZ" , factory.getObject("address") ) ;
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Utilisé dans l'attribut "**properties**", l'attribut "**arguments**" sera pris en compte par le conteneur uniquement si la propriété définie dans un des **objets génériques** de cette collection sera de type "**Function**" (méthode de l'objet défini dans la fabrique).

Exemple

```
properties :
[
     { name : "setMail" , arguments : [ { value : "ekameleon [at] gmail.com" } ] }
]
```

Cet exemple très simple permet de définir dans l'attribut "**properties**" une méthode "**setMail()**" qui sera invoquée lors de l'initialisation de l'objet.

*A noter que nous pouvons utiliser à la place des attributs classiques "**value**" et "**ref**", deux autres attributs "spéciaux" : "**config**" (voir chapitre B-1.3.3) et "**locale**" (voir chapitre B-1.3.4). Ces attributs sont très spécifiques car ils entrainent une évaluation de la valeur passée en argument pour récupérer une valeur "**localisée**" ou "**configurée**" dans l'application.*

II-C-1 - L'attribut "value"

Cet attribut utilisé dans les définitions d'un argument est le plus simple :

```
{ value : "eKameleon" }
```

Cette solution permet de définir un argument avec une simple valeur de n'importe quel type.

II-C-2 - L'attribut "ref"

Cet attribut permet de définir une expression (**String**) qui correspondra à l'identifiant d'une **définition d'objet** enregistrée dans le conteneur.

```
{ ref : "address" }
```

Cet attribut permet de définir les dépendances entre les objets définis dans le conteneur **IoC**.

Dans l'exemple ci-dessus, la référence "**address**" sera utilisée au moment de créer l'objet et la fabrique invoquera la méthode **getObject("address")** pour récupérer (ou créer) la référence de l'objet dont elle dépend.

Tous les attributs "**ref**" utilisés dans les **définitions d'objets** ont le même mode de fonctionnement. Je vais donc essayer de généraliser ici l'utilisation de cet attribut que nous pourrons retrouver par exemple dans les attributs "**properties**", etc.

Cet attribut possède donc une expression évaluée dans la fabrique par un objet de type **andromeda.ioc.evaluators.ReferenceEvaluator**.

La classe **ReferenceEvaluator** cible une fabrique **IoC** et analyse une chaine de caractère. Cette classe évalue donc cette expression et essai en priorité d'identifier une **définition d'objet** contenue dans la fabrique **IoC** pour renvoyer tout de suite la référence de l'objet correspondant.

Si l'expression est définie avec une **notation pointée** il sera alors possible de cibler un objet dans la fabrique et de récupérer la valeur d'un de ses attributs.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Exemple

```
{ ref : "address.city"
```

L'argument défini avec la notation ci-dessus aura pour valeur "**Marseille**". A noter qu'il est possible d'utiliser une notation pointée plus profonde pour chercher une valeur définie plus plus loin dans les attributs de l'objet, exemple "**a.b.c.d....**".

L'attribut "**ref**" peut être défini également avec des "**expressions magiques**", ces chaines de caractères spéciales permettent de cibler par référence certains objets importants définis au préalable dans la configuration de la fabrique **IoC**.

- **#this** : Permet de cibler la référence de la fabrique locale utilisée par la définition d'objet courante.
- **#root** : Permet de cibler la référence de la scène principale de l'animation courante.
- **#stage** : Permet de cibler la référence de l'objet **stage** de l'application.

Note :

Pour définir convenablement les **expressions magiques #root** et **#stage**, il faut définir manuellement la propriété "**root**" de l'objet de configuration de la fabrique **IoC** courante en ciblant si possible la référence de plus bas niveau de l'application principale, nous pourrons par exemple écrire le code suivant dans la fonction constructeur de la classe principale de l'application :

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

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.config.root = this ;
```

Voir à ce sujet des exemples dans les chapitre A.3-2 et C.1-6.

II-C-3 - L'attribut "config".

```
{ config : "author" }
```

Cet attribut spécial peut être défini à la place de l'attribut "**value**" ou de l'attribut "**ref**" pour cibler une valeur définie dans l'objet de configuration de la fabrique (voir chapitre C-1-1).

II-C-4 - L'attribut "locale".

```
{ locale : "message" }
```

Cet attribut spécial peut être défini à la place de l'attribut "**value**" ou de l'attribut "**ref**" pour affecter sur la propriété une valeur définie dans l'objet de localisation courante de la fabrique (voir chapitre C-1-4).

Cet objet est très important car dans une utilisation avancée des fichiers de configurations du conteneur **IoC**, il est possible de charger des données localisées et de les utiliser le moment voulu par propagation évènementielle ou plus simplement avec l'attribut "**locale**" dans les arguments d'une **définition d'objet**. En fonction de la langue courante choisie dans l'application l'attribut locale renverra une valeur spécifique et localisée.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II-C-5 - L'attribut "properties"

L'attribut "**evaluators**" est un attribut optionnel dans tous les objets génériques définis dans l'attribut "**arguments**" d'une **définition d'objet**. Plus d'informations à ce sujet à la fin de ce chapitre (cf. Partie 4 de ce chapitre).

II-D - L'attribut "properties"

Dans l'objet de configuration d'une fabrique **IoC**, l'attribut "**properties**" est une collection (**Array**) d'**objets génériques** simples qui permet d"initialiser un objet en définissant les valeurs de certains de ses attributs et en d'invoquant certaines de ses méthodes.

Dans l'objet de configuration une définition d'objet il est possible de définir chacun des attributs que l'on souhaite initialiser avec un object générique simple :

```
properties :
[
    { name:"age"  , value : 31        } ,
    { name:"job"  , ref   : "job_dev" }
]
```

Les objets génériques définis dans la collection ci-dessus nécessitent un attribut "**name**" qui indique quel attribut nous souhaitons initialiser.

Ensuite nous pourrons définir la valeur d'un attribut en utilisant les attributs "**value**", "**ref**", "**config**" ou "**locale**" et dans le cas d'une méthode nous pourrons utiliser un attribut "arguments" pour pour définir en cas de besoin ses paramètres.

II-D-1 - Utilisation de l'attribut "value".

```
{ name:"age"  , value : 31  }
```

Cet attribut permet d'initialiser l'attribut d'un objet avec une valeur de n'importe quel type.

II-D-2 - Utilisation de l'attribut "ref".

```
{ name : "job"  , ref : "job_dev" }
```

L'attribut "**ref**" est une une chaine de caractère (String) qui cible principalement une autre **définition d'objet** dans le **conteneur IoC** courant.

Comme dans l'attribut "**arguments**", il est possible d'aller plus loin au moment de définir l'expression de l'attribut "**ref**". Cette expression est évaluée par la fabrique avec la classe ReferenceEvaluator.

Exemple

```
{
    id         : "job_dev"   ,
    type       : "test.Job"  ,
    properties : [ { name:"name" , value:"AS Developper" } ]
}
,
{
    id         : "field"   ,
    type       : "flash.text.TextField"  ,
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Exemple

```
    properties :
    [
        { name:"autoSize" , value : "left"         } ,
        { name:"text"     , ref   : "job_dev.name" }
    ]
}
```

Cet exemple très simple affiche le nom du métier défini dans la **définition d'objet** "**job_dev**" dans un champ de texte dynamique simple. L'attribut "**ref**" prend pour valeur une expression avec une notation pointée, le premier élément dans cette expression correspond à l'identifiant d'une **définition d'objet** dans la fabrique et une fois cette référence identifiée dans la fabrique la propriété "text" du champ de texte prend la valeur de la propriété name de l'objet de type **test.Job**.

Je vous conseille de consulter le descriptif plus complet du chapitre B.1-3-2 précédent (sur l'attribut "arguments") pour en savoir plus.

II-D-3 - Utilisation de l'attribut "config".

```
{ name : "name"  , config : "author" }
```

Cet attribut spécial peut être défini à la place de l'attribut "**value**" ou de l'attribut "**ref**" pour cibler une valeur définie dans l'objet de configuration de la fabrique (voir chapitre C-1-1).

II-D-4 - Utilisation de l'attribut "locale".

```
{ name : "name"  , locale : "message" }
```

Cet attribut spécial peut être défini à la place de l'attribut "**value**" ou de l'attribut "**ref**" pour affecter sur la propriété une valeur définie dans l'objet de localisation courante de la fabrique (voir chapitre C-1-4).

II-D-5 - Définition d'une propriété de type Function et utilisation de l'attribut "arguments"

L'attribut "**properties**" permet également d'invoquer une ou plusieurs méthodes pendant l'initialisation de l'objet.

```
properties :
[
    { name : "setMail" , arguments : [ { value : "ekameleon [at] gmail.com" } ] }
]
```

Dans le cas de figure où nous souhaitons invoquer une méthode, nous pouvons définir un **objet générique** identique à ceux utilisés pour les attributs avec un attribut "**name**" indispensable pour définir le nom de la méthode que l'on souhaite invoquer. Il est possible aussi de déclarer un attribut "**arguments**" facultatif.

- L'attribut "**name**" représente tout simplement le nom de la méthode à invoquer.
- L'attribut "**arguments**" est une liste (**Array**) des arguments passés dans la méthode.

L'attribut "**arguments**" possède le même mode de fonctionnement que celui utilisé pour initialiser les paramètres de la fonction constructeur d'une **définition d'objet** (voir chapitre B.1-3).

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II-D-6 - L'attribut "evaluators".

L'attribut "**evaluators**" est un attribut optionnel dans tous les objets génériques définis dans l'attribut "**properties**" d'une définition d'objet. Plus d'informations à ce sujet à la fin de ce chapitre (chapitre B-4).

II-E - Les attributs "scope", "singleton"et "lazyInit".

Quand nous créons une **définition d'objet** nous mettons en place en quelque sorte une "**recette**" qui permet de créer un objet dans l'application.

L'idée de "**recette**" est importante, elle indique qu'une **définition d'objet** permet de créer un grand nombres d'instances dans l'application en utilisant des stratégies plus ou moins complexes de création et d'initialisation. Il est donc possible de créer plusieurs objets avec une **définition d'objet** unique.

Ces objets auront des cycles de vies particuliers et parfois devront avoir une représentation unique ou multiple dans l'application.

La notion de **scope** est importante dans un **conteneur IoC** car elle indique à la fabrique comment créer les objets de l'application et surtout comment les référencer par la suite.

II-E-1 - L'attribut "scope"

L'attribut "**scope**" n'est pas obligatoire dans une **définition d'objet** et prend pour valeur une chaine de caractère (**String**).

```
scope : "prototype"
```

Voici les différentes valeurs de l'attribut "**scope**" dans les **définitions d'objet** :

- "**prototype**"
- "**singleton**"

Le scope "**prototype**" indique que la fabrique **IoC** peut créer avec la même **définition d'objet** un nombre indéfini d'instances.

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

import test.User ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

var user1:User = factory.getObject("user") as User ;

var user2:User = factory.getObject("user") as User ;

trace( user1 == user2 ) ; // false
```

Le scope "**singleton**" indique que la **définition d'objet** peut créer une instance unique (singleton) et ensuite la mémorise dans le conteneur **IoC**. Par la suite chaque appel de cette même **définition d'objet** reverra une simple référence du singleton enregistré dans le cache de la fabrique.

Par défaut, toutes les **définitions d'objets** ont le scope "**prototype**" mais il est possible de modifier cette valeur avec les valeurs définies ci-dessus.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Vous pouvez retrouver dans **AndromedAS** une classe statique qui énumère les différents types de "**scope**" existant avec la classe andromeda.ioc.core.ObjectScope :

- **ObjectScope.PROTOTYPE**
- **ObjectScope.SINGLETON**

A noter que l'attribut "**scope**" dans une **définition d'objet** écrase toujours la valeur de l'attribut "**singleton**" dans cette même définition d'objet si il est utilisé. (voir chapitre B-1.5.2)

II-E-2 - L'attribut "singleton"

Les **objets génériques** d'initialisation des **définitions d'objets** peuvent contenir un attribut "**singleton**" booléen. Cet attribut permet de modifier rapidement l'état par défaut de la propriété **scope** d'une définition d'objet en lui attribuant directement la valeur "**singleton**". Cet attribut est donc suffisant pour transformer rapidement la nature par défaut d'une **définition d'objet** dans un **conteneur léger**.

```
singleton : true
```

Si la définition d'objet est définie avec un scope "**singleton**" et que cet attribut possède la valeur "**true**" alors l'instance d'objet créé via la fabrique est sauvegardée dans le cache du conteneur et sera utilisée par la suite à chaque appel de la même **définition d'objet**.

Si pour un identifiant dans la fabrique **IoC** une **définition d'objet** est initialisée comme "**singleton**" alors à chaque appel de l'identifiant id dans la méthode **getObject()** de la fabrique renverra une référence singleton unique. La première fois où la méthode **getObject()** sera invoquée, la référence unique sera créée, initialisée et stockée en mémoire dans la fabrique.

Voyons un petit exemple de ce scope avec une **définition d'objet** singleton :

```
{
    id          : "user" ,
    type        : "test.User" ,
    arguments   :
    [
        { value : "eKameleon" } ,
        { value : "ALCARAZ"   } ,
        { ref   : "address"   }
    ] ,
    singleton   : true
}
```

Voyons ensuite dans un script **ActionScript** comment récupérer plusieurs fois une même référence de l'objet défini avec l'identifiant "**user**" dans la fabrique :

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

import test.User ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

var user1:User = factory.getObject("user") as User ;

var user2:User = factory.getObject("user") as User ;

trace( user1 == user2 ) ; // true
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II-E-3 - L'attribut "lazyInit"

Cet attribut est un booléen optionnel dans les définitions d'objets, il prend la valeur false par défaut.

```
lazyInit : true , // only if this object is a singleton
```

Cet attribut s'applique uniquement sur les **définitions d'objets** ayant un scope "**singleton**". Si vous utilisez cet attribut sur une **définition d'objet** non singleton il sera tout simplement ignoré.

Si cet attribut est **false** ou non défini, l'objet singleton sera automatiquement instancié pendant l'initialisation de la fabrique.

Si cet attribut est **true** alors l'objet singleton ne sera pas instancié automatiquement et l'objet singleton sera créé uniquement lors du premier appel de la méthode **getObject()**.

Dans le cycle de vie d'une application, il peut s'avérer très pratique de créer les objets singletons au lancement de l'application, car ces objets peuvent lancer certains processus automatiquement sans avoir à utiliser un code en particulier dans l'application. Cette fonctionnalité permettra aussi de placer en mémoire certains objets qui seront disponibles beaucoup plus rapidement par la suite.

Malgré tout, l'attribut "**lazyInit**" peut s'avérer fort important si un objet singleton défini dans la fabrique ne doit absolument pas être instancié au lancement de l'application. Nous pouvons à ce sujet noter les raisons suivantes :

- Ne pas surcharger la mémoire avec un singleton non nécessaire tout de suite au lancement de l'application.
- Ne pas créer tout de suite le singleton car certaines de ses dépendances n'existent pas ou ne peuvent pas être instanciées dans le conteneur **IoC** pour le moment.

II-F - L'attribut "listeners"

L'attribut "**listeners**" d'une **définition d'objet** est une collection (Array) d'**objets génériques** qui permettent d'initialiser un objet pour que certaines de ses méthodes servent de fonctions écouteurs dans le modèle évènementiel de l'application.

Avec cet attribut l'objet est automatiquement abonné pour recevoir des évènements diffusés par un ou plusieurs diffuseurs d'évènements définis eux aussi dans la fabrique IoC.

A noter que si l'objet peut lui même diffuser des évènements (si il implémente l'interace **flash.events.IEventDispatcher**) il pourra s'abonner à lui même pour écouter la propagation de ses propres évènements .

Voyons maintenant un exemple concret d'utilisation de cet attribut avec une petite classe **test.Listener** qui possède une méthode **change(e:Event)**. Cette méthode peut intercepter des évènements de type **flash.events.Event**.

```
package test
{

    import flash.events.Event;

    public class Listener
    {

        public function Listener()
        {
            //
        }
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        public function change( e:Event ):void
        {
            trace( this + " change" + e ) ;
        }

    }
}
```

Voyons maintenant comment câbler un diffuseur d'évènement et son écouteur dans les **définitions d'objets** d'un conteneur **IoC**.

```
var context:Object =
[
    {
        id        : "dispatcher" ,
        type      : "flash.events.EventDispatcher" ,
        singleton : true
    }
    ,
    {
        id        : "listener"   ,
        type      : "test.Listener" ,
        singleton : true ,
        listeners :
        [
            { dispatcher:"dispatcher" , type:"change" , method:"change" , useCapture:false,
 priority:0 , useWeakReference:true }
        ]
    }
] ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( context ) ;

var dispatcher:EventDispatcher = factory.getObject("dispatcher") ;

dispatcher.dispatchEvent( new Event( "change"
 ) ) ; // [object Listener] change [Event type="change" bubbles=false cancelable=false eventPhase=2]
```

Les 2 **définitions d'objets** ci-dessus ont un scope "**singleton**", les objets définis dans la fabrique sont automatiquement instanciés, initialisés et placés dans le cache de la fabrique **IoC**. Le cablage entre l'objet écouteur "**listener**" et le diffuseur d'évènement "**dispatcher**" est donc automatique.

Dans la **définition d'objet** dans l'attribut "**listener**" nous pouvons observer un objet générique qui contient les attributs suivants :

- **dispatcher** *(obligatoire)* : Correspond à l'identifiant d'une **définition d'objet** enregistrée dans le conteneur. Cet attribut fonctionne comme l'attribut "**ref**" défini dans les attributs "**arguments**" et "**properties**" vus plus haut.
- **type** *(obligatoire)* : Correspond au type de l'évènement utilisé pour abonner l'écouteur sur le diffuseur d'évènement.
- **method** : Correspond au nom de la méthode qui va servir d'écouteur. *(obligatoire si l'objet n'implémente pas l'interface vegas.events.EventListener)*
- **useCapture** : Permet de définir si l'écouteur sera notifié par un évènement pendant la phase de capture lors d'une propagation évènementielle lancée par un diffuseur d'évènement dans l'application. *(optionnel)*
- **priority** : Permet de définir l'ordre de priorité de l'écouteur dans la collection de diffusion définie par le diffuseur d'évènement pour un type d'évènement donné. *(optionnel)*
- **useWeakReference** : Active le mode **useWeakReference** de la méthode **addEventListener** du diffuseur d'évènement. *(optionnel)*

L'exemple précédent pourrait être défini en **ActionScript** avec le code suivant :

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
dispatcher.addEventListener("change", listener.change , false, 0 , true ) ;
```

Le **câblage en couche** des objets renvoyés par la fabrique **IoC** permet un contrôle complet sur tous les arguments de la méthode **addEventListener()** définie dans l'interface **flash.events.IEventDispatcher**.

Voyons maintenant une spécificité au niveau des objets génériques définis dans l'attribut "**listeners**" d'une **définition d'objet**. Nous avons pu voir précédemment que, de façon générale, il faudra définir une référence vers une autre **définition d'objet** dans le conteneur **IoC** et ensuite attacher sur le **diffuseur d'évènements** une méthode appartenant à l'objet écouteur pour un type d'évènement donné. Pour cibler convenablement la méthode désirée nous utilisons l'attribut "**method**" qui représente le nom de la méthode qui recevra les évènements (à condition que l'objet écouteur définisse bien une méthode du même nom).

Cette fonctionnalité est prioritaire. Dès que l'attribut "**method**" est défini dans une définition d'un écouteur alors c'est cette méthode qui sera utilisée.

Maintenant, il est possible d'aller un peu plus loin en oubliant délibérément de déclarer l'attribut "**method**" dans un objet générique défini dans l'attribut "**listeners**". Dans ce cas, il faudra absolument que l'objet généré par la **définition d'objet** implémente l'interface vegas.events.EventListener qui lui impose l'existence d'une méthode **handleEvent(e:Event)** comme défini dans le DOM 2/3 du modèle évènementiel du W3C. Cette fonction servira automatiquement de fonction écouteur dans l'objet de la **définition d'objet** courante.

Pour illustrer cette spécificité, je reprends l'exemple précédent en redéfinissant la classe **test.Listene**r qui implémente maintenant l'interface vegas.events.EventListener.

```
package test
{

    import flash.events.Event;

    import vegas.events.EventListener ;

    public class Listener implements EventListener
    {

        public function Listener()
        {
            //
        }

        public function handleEvent( e:Event ):void
        {
            trace( this + " handleEvent" + e ) ;
        }

    }
}
```

Voyons maintenant comment utiliser très simplement cette classe dans une **définition d'objet** avec l'attribut "**listeners**" :

```
var context:Object =
[
    {
        id        : "dispatcher" ,
        type      : "flash.events.EventDispatcher" ,
        singleton : true
    }
    ,
    {
        id        : "listener"   ,
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        type      : "test.Listener" ,
        singleton : true ,
        listeners :
        [
            { dispatcher:"dispatcher" , type:"change"  }
        ]
    }
] ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( context ) ;

var dispatcher:EventDispatcher = factory.getObject("dispatcher") ;

dispatcher.dispatchEvent( new Event( "change" ) ) ;
// [object Listener] handleEvent [Event type="change" bubbles=false cancelable=false eventPhase=2]
```

Plus besoin de déclarer l'attribut "**method**" dans ce cas de figure, la méthode **handleEvent** est utilisée automatiquement pour intercepter les évènements.

Il ne faut pas oublier qu'il est toujours possible de cibler avec l'attribut "**method**" une autre méthode définie dans la classe qui prendra la place de la méthode **handleEvent()** comme fonction écouteur.

Remarque :

Les **objets génériques** définis dans la collection (**Array**) de l'attribut "**listeners**" sont filtrés dans la fabrique et sont transformés en objet de type andromeda.ioc.core.ObjectListener. Cette classe contient une méthode statique "**create()**" qui prend en argument un **Array** d'**objets génériques** et qui renvoie un nouveau Array d'objets de type **ObjectListener**.

Exemple

```
import andromeda.ioc.core.ObjectListener ;

var init:Array =
[
    { dispatcher:"dispatcher1" , type:"click"  , method:"handleEvent" } ,
    { dispatcher:"dispatcher2" , type:"change" , method:"handleEvent" }
] ;

var listeners:Array = ObjectListener.create( init ) ;

trace( "listeners[0] : " + listeners[0] ) ;

// listeners[0] : [ObjectListener dispatcher:"dispatcher1" type:"click" method:"handleEvent"]

trace( "listeners[1] : " + listeners[1] ) ;

// listeners[1] : [ObjectListener dispatcher:"dispatcher2" type:"change" method:"handleEvent"]
```

III - Cycle de vie d'un objet configuré avec une définition d'objet.

III-A - Les attributs "init" et "destroy".

Il est possible de définir dans une **définition d'objet** une méthode qui sera invoquée à la fin de l'initialisation d'un objet une fois généré via le conteneur **IoC**. Cette méthode devra être définie dans la classe de l'objet renvoyé et sera définie sans paramètre. Cette méthode sera invoquée à la fin de l'initialisation de l'objet dans la méthode **getObject()** du conteneur.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Pour définir cette méthode il suffira d'utiliser l'attribut "**init**" dans une **définition d'objet**, cet attribut est de type **String** et prend pour valeur le nom de la méthode que l'on souhaite invoquer.

A noter qu'il est également possible de définir une méthode qui sera invoquée lorsque un objet renvoyé par la fabrique sera détruit par cette même fabrique. Il faut dans ce cas que l'objet ait un scope "**singleton**" et soit placé en mémoire dans la fabrique au moment de sa création avec la méthode **getObject()**. Pour définir le nom de la méthode utilisée à ce moment là nous utiliserons l'attribut "**destroy**" dans les **définitions d'objets**.

Exemple

```
{
   id          : "user" ,
   type        : "test.User" ,
   singleton   : true ,
   destroy     : "destroy" ,
   init        : "initialize"
}
```

Dans l'exemple précédent la méthode **initialize()** du singleton de type **test.User** sera invoquée au moment de sa création avec la méthode **factory.getObject("user")** du conteneur.

L'instance de type **test.User** est définie avec un scope "**singleton**" dans la fabrique et il sera donc possible en cas de nécessité de le supprimer dans le cache interne de la fabrique avec la méthode "**removeSingleton(id)**" :

```
ECMAObjectFactory.removeSingleton( "user" ) ; // the destroy method is invoked
```

La méthode **removeSingleton()** se charge d'invoquer la méthode définie avec l'attribut "**destroy**" avant de la supprimer la référence dans le cache de la fabrique.

*Il est possible de généraliser dans toutes les **définitions d'objets** d'une fabrique l'appel automatique des méthodes d'initialisation et de destruction (si elles existent dans la classe de l'objet défini) en configurant la fabrique convenablement. Vous pouvez consulter à ce sujet le chapitre C-1-2 sur les attributs de configuration "**defaultInitMethod**" et "**defaultDestroyMethod**".*

III-B - L'attribut "identify".

Cet attribut de type **Boolean** doit être utilisé uniquement avec une **définition d'objet** avec un scope "**singleton**" et si le type de l'objet défini implémente l'interface **vegas.core.Identifiable**.

L'interface vegas.core.Identifiable définie simplement un objet avec un identifiant unique "**id**". Cette interface est implémentée par de nombreux objets dans les différentes classes de **VEGAS** et ses extensions.

Quand l'attribut **identify** est vrai (true), le nouveau objet Identifiable créé par la fabrique est automatiquement identifié avec l'identifiant de sa **définition d'objet**.

Exemple

```
objects =
[
    {
        id          : "vo_01"  ,
        type        : "andromeda.vo.NetServerInfoVO" ,
        singleton   : true ,
        identify    : true ,
        arguments   :
        [
            value :
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Exemple

```
            {
                application : "local" ,
                level       : "error" ,
                code        : "application_error" ,
                description : "An error is invoqued in the application" ,
                line        : 3 ,
                methodName  : "noMethod" ,
                serviceName : "noService"
            }
        ]
    }
] ;
```

Dans cet exemple la classe andromeda.vo.NetServerInfoVO implémente l'interface vegas.core.Identifiable.

Une fois la fabrique initialisée avec cette **définition d'objet**, nous pouvons tester celle ci avec le code suivant :

```
var info:NetServerInfoVO = factory.getObject( "vo_01" ) ;

// The id of the vo is auto initialized if the identify property of the ObjectDefinition of this object is true.

trace( "info.id : " + info.id ) ; // info.id : vo_01
```

La valeur de l'attribut "**id**" de l'instance de type **NetServerInfoVO** est automatiquement définie avec la valeur "**vo_01**". L'identifiant dans la fabrique de la **définition de l'objet** est bien le même que celle du singleton récupéré dans la fabrique.

*Cet attribut est prioritaire sur un attribut du même nom défini dans la configuration de la fabrique, la référence de type **ObjectConfig** dans la classe **ObjectFactory** contient elle aussi une propriété **identify** booléenne qui permet de généraliser l'action d'initialisation automatique des singletons identifiables définis dans la fabrique (voir chapitre C-1-3)*

III-C - L'attribut "lock".

Cet attribut de type **Boolean** doit être utilisé uniquement avec une **définition d'objet** qui permet de créer un objet qui implémente l'interface vegas.core.ILockable.

Cet attribut peut prendre les valeurs suivantes :

- **null** : L'objet défini dans le conteneur **IoC** n'est pas protégé par défaut au moment de son initialisation sauf si la propriété "**lock**" de la configuration de la fabrique vaut **true**.
- **true** : L'objet défini dans la fabrique est protégé durant son initialisation (cet attribut est prioritaire sur l'attribut "**lock**" défini dans la configuration du conteneur **IoC**).
- **false** : L'objet défini dans la fabrique ne sera pas protégé durant son initialisation même si la propriété "**lock**" de la configuration du conteneur vaut **true**.

De façon général l'interface **ILockable** définie dans une classe qui l'implémente les trois méthodes **lock()**, **unlock()** et **isLocked()**.

Cette interface minimaliste peut être implémentée par toute classe qui nécessite dans son cycle de vie de nombreuses mises à jours ou de nombreuses actions répétitives qui pourraient nuire en diminuant les performances de l'ensemble de l'application.

Prenons par exemple une classe d'un composant qui possède plusieurs propriétés virtuelles (**get/set**). Ce composant utilise des propriétés pour mettre à jour ses couleurs, la taille de son texte, etc. Imaginons ensuite que toutes ses propriétés une fois modifiées exécutent à chaque fois une méthode qui permet de mettre à jour la vue du composant.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Si nous cherchons à créer plusieurs instances de cette même classe et au même moment à initialiser plusieurs propriétés de ces instances, nous pouvons rapidement avoir une surchage du **CPU** avec une relance inutile de la méthode d'initialisation de tous les composants.

D'où l'intérêt dans ce cas de figure d'implémenter l'interface **ILockable** et de définir les 3 méthodes de cette interface pour ensuite créer des sécurités dans le code et ainsi avoir la possibilité de contrôler plus finement les mises à jour de chaque composant le moment voulu.

Pour illustrer ce concept je vais créer une classe visuelle **test.display.RectangleSprite** qui permet de créer des symboles graphiques dynamiques rectangulaires. Cette classe implémente l'interface **ILockable**.

```
package test.display
{
    import flash.display.Sprite;

    import vegas.core.ILockable;

    /**
     * This sprite draw a rectangle with a virtual width and height value.
     */
    public class RectangleSprite extends Sprite implements ILockable
    {

        /**
         * Creates a new RectangleSprite instance.
         */
        public function RectangleSprite()
        {
            super() ;
        }

        /**
         * The color of the rectangle shape.
         */
        public function get color():uint
        {
            return _color ;
        }

        /**
         * @private
         */
        public function set color( value:uint ):void
        {
            _color = value ;
            update() ;
        }

        /**
         * Determinates the virtual height value of this sprite.
         */
        public function get h():Number
        {
            return _h ;
        }

        /**
         * @private
         */
        public function set h( n:Number ):void
        {
            _h = isNaN(n) ? 0 : n ;
            update() ;
        }

        /**
         * Determinates the virtual height value of this sprite.
         */
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        public function get w():Number
        {
            return _w ;
        }

        /**
         * @private
         */
        public function set w( n:Number ):void
        {
            _w = isNaN(n) ? 0 : n ;
            update() ;
        }

        /**
         * Draw the display.
         */
        public function draw():void
        {
            graphics.clear() ;
            graphics.beginFill( color , 1 ) ;
            graphics.drawRect( 0, 0, w, h ) ;
        }

        /**
         * Returns true if the object is locked.
         * @return true if the object is locked.
         */
        public function isLocked():Boolean
        {
            return ___isLock___ == true ;
        }

        /**
         * Locks the object.
         */
        public function lock():void
        {
            ___isLock___ = true ;
        }

        /**
         * Unlocks the display.
         */
        public function unlock():void
        {
            ___isLock___ = false ;
        }

        /**
         * Update the display.
         */
        public function update():void
        {
            if ( isLocked() )
            {
                return ;
            }
            trace(this + " update") ;
            draw() ;
        }

        /**
         * @private
         */
        protected var _color:uint = 0xFF0000 ;

        /**
         * @private
         */
        protected var _h:Number = 0 ;
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        /**
         * @private
         */
        protected var _w:Number = 0 ;

        /**
         * The internal flag to indicates if the display is locked or not.
         * @private
         */
        private var ___isLock___:Boolean ;

    }
}
```

La classe **RectangleSprite** possède les propriétés virtuelles **color**, **h** et **w** qui permettent de définir respectivement la couleur, la hauteur et la largeur du rectangle que l'on souhaite dessiner dynamiquement et afficher. Chaque appel d'une de ces propriétés virtuelles lance automatiquement la méthode **update()** pour mettre à jour la vue du **Sprite** avec la méthode **draw()** définie dans la classe.

La classe **RectangleSprite** implémente l'interface **ILockable** et son implémentation reste très simple avec un simple attribut booléen qui peut permuter entre une valeur **true** ou **false** avec les méthodes **lock()** et **unlock()**. Nous utilisons donc un point d'arrêt avec la méthode **isLocked()** dans la méthode **update()** de la classe :

```
/**
 * Update the display.
 */
public function update():void
{
   if ( isLocked() )
   {
       return ; // Point d'arrêt ici si isLocked() renvoi "true".
   }
   trace( this + " update" ) ;
   draw() ;
}
```

Ce point d'arrêt avec une instruction **return** sera invoquée si la méthode **isLocked()** renvoie true. Cette méthode très simple peut être déployée dans n'importe quelle classe, il suffira d'implémenter l'interface **ILockable** et de définir en gros ses méthodes comme dans la classe **RectangleSprite** ci-dessus.

A noter que dans **VEGAS** et toutes ses extensions de nombreuses classes implémentent déjà cette interface et il sera assez simple de s'inspirer de ces classes pour aller un peu plus loin dans vos propres applications.

Voyons maintenant un petit bout de code **ActionScript** pour illustrer l'utilisation de cette implémentation :

```
import test.display.RectangleSprite ;

var sprite:RectangleSprite = new RectangleSprite() ;

sprite.color = 0xFF00FF
sprite.w     = 200 ;
sprite.h     = 180 ;

sprite.x     = 25 ;
sprite.y     = 25 ;

addChild( sprite ) ;
```

Je copie le code précédent dans une petite animation **Flash** et je lance la compilation, j'obtiens un rectangle fushia sur la scène et dans le panneau de sortie les **trace** suivantes :

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
Je copie le code précédent dans une petite animation Flash et je lance la compilation,
 j'obtiens un rectangle fushia sur la scène et dans le panneau de sortie les trace suivantes :
```

La méthode **update()** est donc utilisée 3 fois de suite lors de l'initialisation de l'instance avec les propriétés **w**, **h** et **color**.

Dans un petit bout de code comme celui-ci, la notion d'optimisation n'est pas très importante, mais à plus grande échelle si nous cherchons à instancier plusieurs objets nous observerons un nombre incalculable d'appels inutiles de la méthode **update()**.

Voyons maintenant comment utiliser l'interface **ILockable** :

```
import test.display.RectangleSprite ;

var sprite:RectangleSprite = new RectangleSprite() ;

sprite.lock() ;

sprite.color = 0xFF00FF
sprite.w     = 200 ;
sprite.h     = 180 ;

sprite.unlock() ;

sprite.update() ;

sprite.x     = 25 ;
sprite.y     = 25 ;

addChild( sprite ) ;
```

Nous obtenons dans le panneau de sortie un seul **trace** :

```
[object RectangleSprite] update
```

La méthode **update()** est donc solicitée qu'une seule fois après la méthode **unlock()**. En **AS3**, nous retrouvons cette implémentation (sans l'interface **ILockable**) dans certains objets comme la classe native **flash.display.BitmapData**.

Voyons maintenant la **définition d'objet** correspondante au code **ActionScript** précédent en utilisant l'attribut "**lock**" :

```
objects =
[
    {
        id          : "my_rectangle"  ,
        type        : "test.display.RectangleSprite" ,
        lock        : true     , // can be 'true', 'false' or 'null'
        init        : "update" ,
        properties  :
        [
            { name : "color" , value : 0xFF00FF } ,
            { name : "w"     , value : 200      } ,
            { name : "h"     , value : 180      } ,
            { name : "x"     , value :  20      } ,
            { name : "y"     , value :  20      }
        ]
    }
] ;
```

Voyons comment utiliser cette définition d'objet dans un conteneur léger **IoC** :

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

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

import flash.display.StageAlign ;
import flash.display.StageScaleMode ;

import test.display.RectangleSprite ;

stage.align     = StageAlign.TOP_LEFT ;
stage.scaleMode = StageScaleMode.NO_SCALE ;

var objects:Array =
[
    {
        id          : "my_rectangle"  ,
        type        : "test.display.RectangleSprite" ,
        lock        : true     ,
        init        : "update" ,
        properties  :
        [
            { name : "color" , value : 0xFF00FF } , // launch the update method
            { name : "w"     , value : 200      } , // launch the update method
            { name : "h"     , value : 180      } , // launch the update method
            { name : "x"     , value : 20       } ,
            { name : "y"     , value : 20       }
        ]
    }
] ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

// if true all ILockable object are locked during the initialization of the properties and methods of the object
// factory.config.lock = true ;

factory.create( objects ) ;

var rec:RectangleSprite = factory.getObject("my_rectangle") ;

addChild( rec ) ;
```

La **définition d'objet** insérée dans la fabrique **IoC** permet de créer avec la méthode **factory.getObject()** une instance de la classe **test.display.RectangleSprite**, elle est attachée ensuite sur la scène principale avec la méthode **addChild()**.

La méthode **lock()** est automatiquement utilisée sur l'instance pendant l'initialisation des propriétés déclarées dans la **définition d'objet**.

La méthode **unlock()** est invoquée ensuite juste avant l'exécution de la méthode **update()** définie comme méthode d'initialisation avec l'attribut "**init**" dans la **définition d'objet**.

A noter dans l'exemple précédent le petit commentaire :

```
// if true all ILockable object are locked during the initialization of the properties and methods of the object
// factory.config.lock = true ;
```

Il est donc possible de définir de façon plus générale un attribut "**lock**" dans la configuration de la fabrique **IoC** pour que tous les objets **ILockable** instanciés dans le conteneur **IoC** soient protégées avec les méthodes **lock()** et **unlock()** au moment de leur initialisation. Il suffit pour cela de désactiver le commentaire de la ligne "**factory.config.lock = true**" pour tester cet attribut global de configuration de la fabrique (voir explication plus complète dans le chapitre C-1-3).

III-D - Les attributs "dependsOn" et "generates".

Les attributs "**dependsOn**" et "**generates**" permettent de lier une **définition objet** avec d'autres **définitions d'objet** enregistrées dans une même fabrique **IoC**.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ces attributs sont définis par un objet de type **Array** qui contient un ensemble d'identifiants (**String**). Chaque identifiant correspond à une **définition d'objet** dans la fabrique **IoC**.

A chaque création d'un nouveau objet via la fabrique, la fabrique pourra initialiser et créer certaines de ses dépendances automatiquement.

Il n'est pas toujours possible d'initialiser les dépendances d'un objet en lui passant une référence de celle ci dans une de ses méthode, ou en lui passant en paramètre de son constructeur, etc. Il faudra donc utiliser les attributs "**dependsOn**" et "**generates**" pour créer certaines objets et assurer le bon fonctionnement de l'objet appelé avec la méthode **getObject()** dans la fabrique.

Les objets définis dans l'attribut "**dependsOn**" seront créés ou initialisés avant la phase d'initialisation du nouveau objet appelé par référence dans la fabrique ou directement dans le code de l'application. Nous utiliserons donc cet attribut pour créer des objets qui n'existent pas pour le moment dans l'application mais qui sont indispensables pour créer et initialiser correctement le nouveau objet. Nous pouvons donc dire que l'objet dépend d'autres objets et ses dépendances doivent absolument exister avant d'initialiser l'objet principal.

Les objets définis dans l'attribut "**generates**" seront créés ou initialisés après la phase d'initialisation du nouveau objet appelé dans la fabrique avec la méthode **getObject(id)**. Nous pourrons dire que les objets définis dans cet attribut dépendent de l'objet crée avec la fabrique sous réserve qu'il soit totalement initialisé avant que le conteneur **IoC** les fabrique.

Exemple d'utilisation des attributs dependsOn et generates :

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

import flash.events.Event ;

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

var context:Object =
[
     {
        id          : "my_button" ,
        type        : "MyButton"  , // a symbol in the library of the application
        singleton   : true        ,
        dependsOn   : [ "root_before" ] ,
        properties :
        [
            { name : "buttonMode" , value : true } ,
            { name : "x"          , value : 50   } ,
            { name : "y"          , value : 50   }
        ]
        ,
        generates   : [ "root_after"  ]
    }
    ,
    {
        id                : "root_before" ,
        type              : "flash.display.MovieClip" ,
        factoryReference : "#root" ,
        listeners         :
        [
            { dispatcher:"my_button" , type:"click" , method:"click" }
        ]
    }
    ,
    {
        id                : "root_after" ,
        type              : "flash.display.MovieClip" ,
        factoryReference : "#root" ,
        properties        :
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Exemple d'utilisation des attributs dependsOn et generates :

```
            [
                { name:"addChild" , arguments:[ { ref:"my_button" } ] }
            ]
        }
] ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.config.root = this ;

factory.create( context ) ;
```

Sans notion de dépendance, cette fonctionnalité permet également de créer de nombreux objets à un moment précis dans l'application en utilisant pour cela un seul objet qui servira de déclencheur pour tous les autres. Cette fonctionnalité est utilisée souvant avec des objets singleton mais ce n'est pas obligatoire.

IV - Les différentes stratégies utilisées par la fabrique pour créer un objet avec une définition d'objet.

Introduction

Dans les **définitions d'objet** il est possible d'utiliser des attributs un peu spéciaux, les "factory attributes".

Ces attributs permettent de définir une stratégie spéciale pour créer un objet en utilisant une technique différente de celle utilisée de façon classique dans la fabrique.

Nous pouvons résumer la stratégie utilisée par défaut dans une **définition d'objet** avec l'exemple suivant :

```
var objects:Array =
[
    { id : "my_object" , type : "Object" }
] ;
```

Pour récupérer une instance de l'objet défini dans la **définition d'objet** précédente, il suffit d'utiliser la méthode **getObject()** :

```
var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( objects ) ;

var o:Object = factory.getObject( "my_object" ) ;
```

Dans la pratique, le code ci-dessus est très simple et équivalent au code suivant :

```
var o:Object = new Object() ;
```

Le conteneur **IoC** utilise donc dans la méthode **getObject()** tout simplement le mot clé "**new**" pour créer des instances ou des singletons avec des **définitions d'objet** classiques.

Il est donc possible d'aller plus loin en utilisant d'autres stratégies pour créer les objets avec une **définition d'objet**. Elles permettent par exemple d'utiliser dans une **définition d'objet** un objet déjà existant dans l'application ou même dans la fabrique, ou d'utiliser des outils externes pour créer, filtrer, référencer le nouvel objet.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ces attributs sont basés sur un moteur défini par des classes situées dans le package andromeda.ioc.factory.strategy. Toutes les classes de ce package implémentent l'interface IObjectFactoryStrategy. Cette interface ne contient pas de méthode en particulier mais permet avant tout de regrouper certaines classes qui seront utilisées dans la fabrique pour créer correctement les objets et leurs dépendances.

Regardons de plus prêt une liste des attributs que nous pouvons utiliser dans une **définition d'objet** pour modifier la stratégie utilisée pour créer ou cibler un objet, chacun de ces attributs correspond à une classe spécifique et surtout à une stratégie de création ou récupération de l'objet différente :

L'attribut factoryMethod

Basé sur la classe **ObjectFactoryMethod**, cet attribut permet de créer un objet en utilisant la méthode d'une référence définie dans une autre **définition d'objet**.

L'attribut factoryProperty

Basé sur la classe **ObjectFactoryProperty,** cet attribut permet de définir un objet en utilisant la valeur d'un simple attribut défini dans une référence d'une autre **définition d'objet**.

L'attribut factoryReference

Basé sur la classe **ObjectFactoryReference**, cet attribut permet de cibler tout simplement une référence crée via une autre **définition d'objet**.

L'attribut factoryValue

Basé sur la classe **ObjectFactoryValue**, cet attribut permet d'utiliser n'importe quelle valeur définie directement dans le contexte de configuration de la fabrique **IoC**.

L'attribut staticFactoryMethod

Basé sur la classe **ObjectStaticFactoryMethod**, cet attribut permet de créer un objet dans la fabrique en utilisant une méthode statique définie dans une classe de l'application.

L'attribut staticFactoryProperty

Basé sur la classe **ObjectStaticFactoryProperty**, cet attribut permet de créer un objet dans la fabrique en utilisant la valeur ou référence d'un attribut statique ou d'une constante d'une classe définie dans l'application.

*Il est important d'utiliser un et un seul de ces attributs pour une même **définition d'objet** et de ne surtout pas essayer d'utiliser plusieurs attributs en même temps.*

IV-B - Utilisation de l'attribut "factoryMethod"

Cet attribut définit une stratégie qui permet :

- de récupérer la référence d'un objet définie dans le conteneur **IoC**
- puis d'invoquer une méthode sur cette référence pour fabriquer et retourner l'objet de la définition d'objet courante.

Pour illustrer l'utilisation de cet attribut nous commençons par définir une simple classe **test.User** :

```
package test
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
{

    public class User
    {

        public function User( name:String )
        {
            this.name = name ;
        }

        public var name:String ;

        public override function toString():String
        {
            return "[User" + ( name != null ? ( " " + name ) : "" ) + "]" ;
        }

    }

}
```

Ensuite nous allons définir une classe qui permet de créer des instances de type **test.User** en se servant d'un filtrage via une liste noire (black list).

```
package test.factory
{

    /**
     * This factory creates User instances but use a filter with a black list array.
     * @author eKameleon
     */
    public class UserFilterFactory
    {

        /**
         * Creates a new UserFilterFactory instance.
         * @param blackList The optional array of string pseudo to banish in this factory.
         */
        public function UserFilterFactory( blackList:Array )
        {
            this.blackList = [] ;
        }

        /**
         * The blackList of this factory.
         */
        public var blackList:Array ;

        /**
         * Creates a new User instance with the specified pseudo.
         * @param pseudo The pseudo String representation of the new User.
         */
        public function build( pseudo:String ):User
        {
            if ( blackList.indexOf( pseudo ) > -1 )
            {
                 throw new Error(this + " build failed, the pseudo '" +
 pseudo + "' is register in the black list of the factory." ) ;
            }
            return new User( pseudo ) ;
        }

    }

}
```

Voyons maintenant comment utiliser l'attribut "**factoryMethod**" dans une **définition d'objet** avec les classes précédentes.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
{
    id          : "user_factory"     ,
    type        : "test.UserFilterFactory" ,
    arguments   :
    [
        { value : [ "lunas" , "pegas" ] }
    ]
}
,
{
    id             : "my_user"    ,
    type           : "test.User" ,
    factoryMethod : { factory:"user_factory" , name:"build" , arguments: [ { value : "vegas" } ]  }
}
,
{
    id             : "your_user"    ,
    type           : "test.User" ,
    factoryMethod : { factory:"user_factory" , name:"build" , arguments: [ { value : "pegas" } ]  }
}
```

L'attribut est défini par un objet générique ayant les attributs suivants :

- **factory** : Une chaine de caractère qui représente l'identifiant d'une référence d'objet défini dans le conteneur **IoC**.
- **name** : Le nom de la méthode invoquée sur cette référence.
- **arguments** : Cet attribut optionnel représente un **Array** de tous les arguments que l'on souhaite passer dans la fonction (voir chapitre B.1-3 **L'attribut "arguments"**).

Finissons cette démonstration avec un petit test **ActionScript** qui illustrera l'utilisation des **définitions d'objets** définies ci-dessus :

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

import test.User ;
import test.factory.UserFilterFactory ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

trace ( factory.getObject( "my_user" ) ) ;
//output: [User vegas]

trace ( factory.getObject( "your_user" ) ) ;
//output: [UserFilterFactory] build failed, the pseudo 'pegas' is register in the black list of the factory.
```

IV-C - Utilisation de l'attribut "factoryProperty"

Cet attribut définit une stratégie qui permet de :

- récupérer la référence d'un objet défini dans une **définition d'objet** du conteneur **IoC**
- puis de cibler une valeur ou une référence d'objet définie dans un des attribut de l'objet défini dans la fabrique.

L'attribut "**factoryProperty**" change donc la stratégie de création naturelle d'une **définition d'objet** en lui permettant de cibler rapidement une propriété d'un objet défini ailleurs dans la fabrique **IoC**.

Pour illustrer l'utilisation de cet attribut nous pouvons définir une classe **test.User** avec une simple attribut publique "**name**" :

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
package test
{

    public class User
    {

        public function User( name:String )
        {
            this.name = name ;
        }

        public var name:String ;

        public override function toString():String
        {
            return "[User" + ( name != null ? ( " " + name ) : "" ) + "]" ;
        }

    }

}
```

Voyons maintenant comment utiliser l'attribut "**factoryProperty**" dans une **définition d'objet** en cherchant à cibler l'attribut **name** d'un objet de type **test.User** défini dans le conteneur **IoC**.

```
{
    id        : "my_user"   ,
    type      : "test.User" ,
    arguments : [ { value:  "JohnDoe" } ]
}
,
{
    id              : "my_name"   ,
    type            : "String" ,
    factoryProperty : { factory:"my_user" , name:"name" }
}
```

L'attribut est défini par un objet générique ayant les attributs suivants :

- **factory** : Une chaine de caractère qui représente l'identifiant d'un objet défini dans le conteneur **IoC**.
- **name** : Le nom de la propriété que l'on souhaite cibler pour définir la valeur ou référence de l'objet défini dans la fabrique.

Finissons ce petit exemple avec un petit script **ActionScript** :

```
var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

var name:String = factory.getObject("my_name") as String ;

trace( name ) ; // JohnDoe
```

Dans la **définition d'objet** nous avons défini un objet de type **String** qui prend sa valeur dans la définition d'objet "**my_user**" en ciblant la propriété "**name**" de l'instance crée via la fabrique.

A noter qu'il ne faut pas oublier que cette stratégie permet de cibler aussi bien des attributs que des méthodes (objet de type **Function**) dans une instance sans les invoquer. J'utilise beaucoup cette stratégie qui permet de créer une copie par valeur d'une fonction ou méthode, par exemple dans un cas de figure de câblage évènementiel avec un objet de type **flash.events.IEventDispatcher** qui cherche à abonner une fonction écouteur avec sa méthode **addEventListener** :

```
{
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    id        : "my_listener"     ,
    singleton : true              ,
    type      : "test.MyListener"
}
,
{
    id               : "my_listener_handler" ,
    type             : "Function" ,
    factoryProperty : { factory:"my_listener" , name:"handleEvent" }
}
,
{
    id         : "my_dispatcher"   ,
    type       : "flash.events.EventDispatcher" ,
    properties :
    [
       { name:"addEventListener" , arguments:[ { value:"change" } , { ref:"my_listener_handler" } ] }
    ]
}
```

Dans l'exemple précédent la classe **test.MyListener** implémente l'interface vegas.events.EventListener qui impose la méthode **handleEvent(e:Event)** dans une classe qui l'implémente. La **définition d'objet** "**my_listener_handler**" permet de récupérer une copie par valeur de la fonction qui doit être enregistrée comme écouteur au niveau de la **définition d'objet** "**my_dispatcher**" et de sa méthode **addEventListener**.

Remarque : Il est possible de simplifier l'exemple précédent en utilisant dans l'attribut "**ref**" du second argument de la méthode **addEventListener** avec la fonctionnalité avancée d'évaluation d'une référence avec la notation pointée (voir chapitre B-1-3-2) :

```
{
    id        : "listener"     ,
    singleton : true              ,
    type      : "test.MyListener"
}
,
{
    id         : "dispatcher"   ,
    type       : "flash.events.EventDispatcher" ,
    properties :
    [
       { name:"addEventListener" , arguments:[ { value:"change" } , { ref:"listener.handleEvent" } ] }
    ]
}
```

Il faudra donc bien réfléchir avant d'utiliser l'attribut **factoryProperty** et bien cerner si il est indispensable de l'utiliser ou non.

A noter que l'exemple précédent peut être modifié également en utilisant l'attribut "**listeners**" (chapitre B.1-7) qui inverse la dépendance entre l'objet qui diffuse les évènements et celui qui les écoute. Malgré tout je trouve très intéressant d'étudier les différentes solutions offertes par la fabrique **IoC** pour câbler convenablement les objets entre eux dans le contexte d'une programmation orientée objet basée sur le **modèle évènementiel AS3** (basé DOM2/3 du W3C).

IV-D - Utilisation de l'attribut "factoryReference"

L'attribut "**factoryReference**" définit une stratégie qui permet de créer un objet en ciblant une autre **définition d'objet** enregistrée dans le conteneur **IoC**. Cet attribut a pour valeur une simple chaine de caractère (**String**) qui correspond à l'identifiant d'une autre **définition d'objet** dans le conteneur **IoC**, celle-ci servira de base pour créer l'objet de la définition courante.

Exemple

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Exemple

```
{
    id            : "user"       ,
    type          : "test.User"  ,
    properties    :
    [
        { name : "city" , value : "Marseille" }
    ]
}
,
{
    id               : "my_user"   ,
    type             : "test.User" ,
    factoryReference : "user"      ,
    singleton        : true        ,
    lazyInit         : true        ,
    properties       :
    [
        { name : "name"      , value : "Doe"  } ,
        { name : "firstname" , value : "John" }
    ]
}
```

Dans l'exemple précédent nous observons que la **définition d'objet** portant l'identifiant "**user**" sert de base pour créer un objet singleton dans la définition d'objet "**my_user**". Voyons maintenant dans un code **ActionScript** comment utiliser ces **définitions d'objet** :

```
import test.User ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

var user:User   = factory.getObject( "my_user" ) as User   ;
var city:String = factory.getObject( "my_city" ) as String ;

trace( "user.name : " + user.firstname ) ; // John
trace( "user.city : " + user.city ) ; // Marseille
```

Cette stratégie peut être très utile pour définir une sorte de **template** plus ou moins avancé qui permettra de créer plusieurs instances dans la fabrique basées sur ce **template** mais ayant ensuite chacune leurs particularités et différences.

A noter que la définition d'objet "my_city" définie plus haut met en avant l'utilisation de la classe **andromeda.ioc.evaluators.ReferenceEvaluator**. Cette classe est utilisée dans toutes les **définitions d'objet** au moment de créer ou d'initialiser un objet au niveau des attributs "**ref**", "**factoryReference**", "**dispatcher**" (dans les attributs "**listeners**") et elle permet d'évaluer une expression (**String**) en retournant en priorité la référence d'un objet défini dans la fabrique **IoC** avec un identifant correspondant à l'expression évaluée. Il est possible d'aller un peu plus loin en utilisant une syntaxe pointée pour récupérer en profondeur dans l'objet la valeur d'un de ses attributs (plus d'explication dans le chapitre sur l'attribut "**evaluators**") :

```
{
    id            : "user"       ,
    type          : "test.User"  ,
    properties    :
    [
        { name : "city" , value : "Marseille" }
    ]
}
,
{
    id               : "my_city"    ,
    type             : "String"     ,
    factoryReference : "user.city"
}
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Dans l'exemple ci-dessus, l'attribut **factoryReference** permet de cibler l'attribut "**city**" de la référence "**user**" définie dans le conteneur **IoC** :

```
import test.User ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

var city:String = factory.getObject("my_city") as String ;

trace( "city      : " + city      ) ; // Marseille
```

IV-E - Utilisation de l'attribut "factoryValue"

Cette stratégie est la plus simple de toute car elle permet de définir directement une valeur simple ou "custom" dans la **définition d'objet**. Celle-ci servira de valeur ou référence directement renvoyée par la méthode **getObject()** du conteneur.

```
{
    id           : "my_string"    ,
    type         : "String"       ,
    factoryValue : "hello world"
}
,
{
    id           : "my_date" ,
    type         : "Date"    ,
    factoryValue : new Date(1977,2,22)
}
,
{
    id           : "my_object" ,
    type         : "Object"    ,
    properties   :
    [
        { name : "message" , ref:"my_string" } ,
        { name : "date"    , ref:"my_date"   }
    ]
}
```

Les **définitions d'objet** ci-dessus servent en quelque sorte de **variables** dans la fabrique si nous les utilisons avec un passage par référence par la suite. C'est surtout avec l'écriture au format **eden** des fichiers de configuration externes du conteneur que cet attribut prend tout son intérêt en déclarant un objet typé (custom ou non) en profitant pleinement des possibilités étendues du parseur **eden**.

IV-F - Utilisation de l'attribut "staticFactoryMethod"

Pour illustrer l'utilisation de cet attribut nous allons commencer par déclarer une petite classe simplifiée **test.User** :

```
package test
{

    public class User
    {

        public function User( name:String )
        {
            this.name = name ;
        }

        public var name:String ;
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    }

}
```

Nous définissons maintenant une fabrique dans notre application avec une classe qui contiendra une méthode statique :

```
package test
{

    public class UserFactory
    {

        public static function create( name:String ):User
        {
            return new User(name) ;
        }

    }
}
```

Voyons pour finir comment utiliser cette classe et sa méthode statique dans la définition d'un objet de notre conteneur **IoC**.

```
{
    id                  : "my_user"   ,
    type                : "test.User" ,
    staticFactoryMethod :
    {
        type      : "test.UserFactory" ,
        name      : "create"           ,
        arguments : [ { value : "ekameleon" } ]
    }
}
```

A noter, que dans **VEGAS** et ses extensions de nombreuses classes utilisent des méthodes statiques pour générer des singletons ou des références globales. Nous pouvons donc ajouter un autre exemple simple avec la classe **vegas.events.FrontController** qui peut gérer des références globales via une gestion de "**singletonsmulti-channels**" avec sa méthode **getInstance(channel:String = null)** (fonctionne comme la classe **ECMAObjectFactory**, voir chapitre A.3-6).

Exemple

```
{
    id                  : "front_controller"   ,
    type                : "vegas.events.FrontController" ,
    singleton           : true ,
    staticFactoryMethod :
    {
        type      : "vegas.events.FrontController" ,
        name      : "getInstance" ,
        arguments : [ { value : "my_channel" } ]
    }
}
```

Cette stratégie est donc fortement conseillée avec de nombreuses classes implémentant des méthodes statiques dans **VEGAS** et ses extensions ou dans vos propres classes.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

IV-G - Utilisation de l'attribut "staticFactoryProperty"

Cet attribut permet de récupérer une référence ou une valeur située dans un attribut statique ou dans une constante définie dans une classe de l'application.

Par exemple nous pouvons créer rapidement une classe d'énumération statique **test.Civility** qui contiendra un ensemble de constantes de type **String** :

```
package test
{

    /**
     * The civility static enumeration list.
     */
    public class Civility
    {

        /**
         * Determinates the "man" civility.
         */
        public static const MAN:String = "man" ;

        /**
         * Determinates the "woman" civility.
         */
        public static const WOMAN:String = "woman" ;

    }
}
```

Dans une **définition d'objet** de la configuration du conteneur **IoC** nous pouvons alors utiliser la stratégie définie par l'attribut "**staticFactoryProperty**" pour cibler une des constantes de la classe ci-dessus :

```
objects :
{
    {
        id                    : "man"    ,
        type                  : "String" ,
        staticFactoryProperty : { type:"test.Civility" , name:"MAN" }
    }
}
```

L'attribut **staticFactoryProperty** a pour valeur un simple objet générique avec les 2 attributs :

- **type** : le nom complet de la classe que l'on souhaite cible.
- **name** : le nom de la propriété statique ou de la constante définie dans la classe.

```
var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

var civility:String = factory.getObject("man") ;

trace( civility ) ; // man
```

L'exemple précédent est vraiment basique, mais il faut bien comprendre que la statégie de l'attribut "**staticFactoryProperty**" permet d'aller plus loin. Elle permet par exemple de récupérer certains objets définis dans l'application et de les mettre à disposition simplement dans le conteneur **IoC** en appliquant au passage certaines modifications et initialisations avant utilisation.

Par exemple dans une application **MXML** basée sur le **Flex SDK** nous pouvons retrouver facilement dans une fabrique **IoC** la référence singleton de l'application principale en

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

ciblant l'attribut statique **http://livedocs.adobe.com/flex/201/html/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Book_Parts&file=app_container_064_10.html** :

```
objects :
{
    {
        id   : "myPanel" ,
        type : "mx.controls.Panel"
    }
    ,
    {
        id                    : "application"    ,
        type                  : "mx.core.Application" ,
        singleton             : true ,
        lazyInit              : true ,
        staticFactoryProperty : { type:"mx.core.Application" , name:"application" } ,
        properties            :
        [
            { name:"addChild" , arguments:[ { ref:"myPanel" } ] }
        ]
    }
}
```

La définition d'objet précédente nous donne donc l'accès dans le conteneur **IoC** vers une référence de l'application principale de l'application.

Cette **définition d'objet** est **singleton** et utilise l'attribut **lazyInit**. Quand nous appellerons pour la première fois la méthode **getObject()** du conteneur IoC avec l'identifiant "**application**", la méthode **addChild** sera automatiquement invoquée sur la scène principale de l'application pour attacher des composants **MXML** (customs ou non).

V - Les attributs "evaluators"

Cet attribut définit une fonctionnalité un peu spéciale dans les **définitions d'objets**, il peut être utilisé essentiellement dans dans les objets génériques définis dans les attributs "**properties**" et "**arguments**".

Cet attribut est une collection (**Array**) contenant soit des objets dont la classe implémente l'interface system.evaluators.Evaluable, soit des expressions (**String**) qui permettront de cibler dans la fabrique une référence d'un objet qui implémentera lui aussi l'interface **system.evaluators.Evaluable**.

V-A - Description et utilisation de l'interface Evaluable

L'interface **system.evaluators.Evaluable** est définie dans le package **system.evaluators.*** du **framework ES4a (Maashaack)** développé par **Zwetan** et moi même dans le cadre d'un projet de framework standard et bas niveau défini pour les langages basés sur les normes **ECMASCript 4**.

L'interface **system.evaluators.Evaluable** est très simple, elle définit l'existence d'une méthode **eval(o:*):*** dans une classe. Cette interface permet à tout moment de créer des classes concrètes qui permettront en fonction des problèmes et des besoins d'une application de filtrer ou de transformer certains objets. Il ne faut pas hésiter à utiliser cette interface dans vos applications.

```
package system.evaluators
{
    public interface Evaluable
    {
        function eval( o:* ):* ;
    }
}
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Cette interface possède quelques classes concrètes intéressante contenues dans le package **system.evaluators.***.

V-A-1 - La classe system.evaluators.DateEvaluator

Elle permet d'évaluer un objet de type **Date** et de renvoyer une chaine de caractère formatée en conséquence.

```
import system.evaluators.DateEvaluator ;

var evaluator:DateEvaluator = new DateEvaluator() ;

evaluator.pattern = "hh 'h' nn 'mn' ss 's' TT" ;

trace( evaluator.eval( new Date(2008,1,21,14,15,0,0) ) ) ; // 02 h 15 mn 00 s PM
```

*Je vous conseille de lire attentivement la documentation de la classe **system.formatters.DateFormatter** pour en savoir plus sur le pattern qu'il est possible d'utiliser pour évaluer l'objet de type **Date** avec un **DateEvaluator**.*

V-A-2 - La classe system.evaluators.EdenEvaluator

Elle permet d'évaluer une chaine de caractère au format **eden** et de renvoyer un objet désérialisé ou sérialisé correspondant.

```
import system.evaluators.EdenEvaluator ;

var e:EdenEvaluator = new EdenEvaluator( false ) ;

var result:Date = e.eval( "new Date(2007,5,12,22,10,5)" ) ;

trace( result ) ; // Tue Jun 12 22:10:05 GMT+0200 2007
```

V-A-3 - La classe system.evaluators.MathEvaluator

Elle permet d'évaluer une chaine de caractère contenant une expression mathématiques plus ou moins complexe et de renvoyer le résultat de cette expression.

```
import system.evaluators.MathEvaluator ;

var init:Object =
{
    x      : 100 ,
    square : function(a:Number):Number {return a*a;}
}

var e:MathEvaluator = new MathEvaluator( init ) ;

var result:Number = e.eval( "( 2 + square(x) ) / 2" ) ;

trace( result ) ; // 5001
```

V-A-4 - La classe system.evaluators.MultiEvaluator

Elle permet de chainer plusieurs objets de type **Evaluable** pour lancer une évaluation successive d'un objet et le transformer en un autre objet en combinant tous les objets **Evaluable** enregistrés.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
import system.evaluators.* ;

var e1:EdenEvaluator = new EdenEvaluator(false) ;
var e2:DateEvaluator = new DateEvaluator() ;

var result:* ;

var me:MultiEvaluator = new MultiEvaluator( e1, e2 ) ;

result = me.eval( "new Date(2007,5,12,22,10,5)" ) ;

trace("evaluate new Date(2007,5,12,22,10,5) : " + result ) ;
// output: evaluate new Date(2007,5,12,22,10,5) : 12.06.2007 22:10:05

me.clear() ; // clear all evaluators

me.add( e1 , e2 ) ;

result = me.eval("new Date(2007,5,12,22,10,5)") ;

trace("evaluate new Date(2007,5,12,22,10,5) : " + result ) ;
// output: evaluate new Date(2007,5,12,22,10,5) : 12.06.2007 22:10:05

me.remove( e2 ) ;

result = me.eval("new Date(2007,5,12,22,10,5)") ;

trace("evaluate new Date(2007,5,12,22,10,5) : " + result ) ;
// output: evaluate new Date(2007,5,12,22,10,5) : Tue Jun 12 22:10:05 GMT+0200 2007
```

V-A-5 - La classe system.evaluators.PropertyEvaluator

Cette classe évalue une expression (**String**) et renvoie la valeur d'une propriété définie dans un objet. Cette expression peut être définie avec une syntaxe pointée sur plusieurs niveaux pour cibler précisément en profondeur dans l'objet une valeur en particulier.

```
import system.evaluators.PropertyEvaluator ;

var obj:Object =
{
     message : "hello world" ,
     title   : "my title"    ,
     menu    :
     {
         title : "my menu title" ,
         label : "my label"
     }
}

var evaluator:PropertyEvaluator = new PropertyEvaluator( obj ) ;

trace( evaluator.eval( "test"       ) ) ; // null
trace( evaluator.eval( "message"    ) ) ; // hello world
trace( evaluator.eval( "title"      ) ) ; // my title
trace( evaluator.eval( "menu.title" ) ) ; // my menu title
trace( evaluator.eval( "menu.label" ) ) ; // my label
```

Cette classe peut être utilisée seule ou par héritage, nous verrons par exemple dans la suite de ce chapitre les classes **ConfigEvaluator** et **LocaleEvaluator** qui héritent de la classe PropertyEvaluator.

V-B - Les implémentations de l'interface Evaluable dans le package andromeda.ioc.*

Dans **AndromedAS** le moteur **IoC** utilise lui aussi pour simplifier certains processus des objets qui implémentent l'interface **Evaluable**, nous pouvons donc regarder de plus prêt dans le package **andromeda.ioc.evaluators**.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

V-B-1 - La classe ConfigEvaluator

Cette classe hérite de la classe **system.evaluators.PropertyEvaluator**, et cible un objet de type **ObjectConfig** qui contient la configuration interne d'une fabrique **IoC**.

Dans les objets de type **ObjectConfig** il existe un attribut "**config**" dynamique qui permet de définir des variables qui seront utilisées le moment voulu dans l'application ou dans les **définitions d'objets** du conteneur léger.

Les objets de type **ConfigEvaluator** pourront donc évaluer une expression (**String**) et renvoyer la valeur d'une variable définie dans l'objet de configuration dynamique de la fabrique :

```
import andromeda.ioc.evaluators.ConfigEvaluator;

import andromeda.ioc.factory.ObjectConfig ;

var init:Object =
{
    message : "hello world" ,
    menu    :
    {
        title : "my title" ,
        count : 10 ,
        data  : [ "item1" , "item2", "item3" ]
    }
}

var configurator:ObjectConfig = new ObjectConfig() ;

configurator.config = init ;

var evaluator:ConfigEvaluator = new ConfigEvaluator( configurator ) ;

trace( evaluator.eval( "test"       ) ) ; // null
trace( evaluator.eval( "message"    ) ) ; // hello world
trace( evaluator.eval( "menu"       ) ) ; // [object Object]
trace( evaluator.eval( "menu.title" ) ) ; // my title
trace( evaluator.eval( "menu.count" ) ) ; // 10
trace( evaluator.eval( "menu.data"  ) ) ; // item1,item2,item3
trace( evaluator.eval( "menu.test"  ) ) ; // null
```

Nous traiterons en détail cette fonctionnalité dans le prochain chapitre concernant l'attribut "**configuration**" du contexte de configuration d'une fabrique **IoC**.

V-B-2 - La classe LocaleEvaluator

Cette classe hérite de la classe **system.evaluators.PropertyEvaluator** et cible un objet de type **ObjectConfig** qui contient la configuration interne d'une fabrique **IoC**.

Cette classe fonctionne comme la classe **ConfigEvaluator** définie ci-dessus sauf qu'elle évalue une expression (**String**) et renvoie une valeur définie dans l'objet "**locale**" de l'objet de type **ObjectConfig**.

```
import andromeda.ioc.evaluators.LocaleEvaluator;
import andromeda.ioc.factory.ObjectConfig ;

var init:Object =
{
     message : "hello world" ,
     title   : "my title"    ,
     menu    :
     {
         title : "my menu title" ,
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        label : "my label"
    }
}

var configurator:ObjectConfig = new ObjectConfig() ;

configurator.locale = init ;

var evaluator:LocaleEvaluator= new LocaleEvaluator( configurator ) ;

trace( evaluator.eval( "test"       ) ) ; // null
trace( evaluator.eval( "message"    ) ) ; // hello world
trace( evaluator.eval( "title"      ) ) ; // my title
trace( evaluator.eval( "menu.title" ) ) ; // my menu title
trace( evaluator.eval( "menu.label" ) ) ; // my label
```

Nous traiterons également en détail cette fonctionnalité dans le prochain chapitre concernant l'attribut "**configuration**" du contexte de configuration d'une fabrique **IoC**.

V-B-3 - La classe ReferenceEvaluator

Cette classe est utilisée dans les **définitions d'objets**, elle permet d'évaluer une expression (**String**) et de renvoyer une référence d'un objet défini dans la fabrique ou plus finement la valeur d'un attribut spécifique d'une de ces références.

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

import andromeda.vo.FilterVO ;

var linkages:Array = [ FilterVO ] ;

var objects:Array =
[
    {
        id         : "VIDEO" ,
        type       : "andromeda.vo.FilterVO" ,
        properties : [ { name : "filter" , value : 1 } ]
    }
    ,
    {
        id         : "MP3" ,
        type       : "andromeda.vo.FilterVO" ,
        properties : [ { name : "filter" , value : 2 } ]
    }
    ,
    {
        id         : "my_filter" ,
        type       : "andromeda.vo.FilterVO" ,
        properties :
        [
            { name:"toggleFilter" , arguments:[ { ref : "VIDEO.filter" } , { value : true } ] } ,
            { name:"toggleFilter" , arguments:[ { ref : "MP3.filter"   } , { value : true } ] }
        ]
    }
];

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( objects ) ;

var evaluator:ReferenceEvaluator = new ReferenceEvaluator( factory ) ;

// test evaluator

trace( evaluator.eval( "DISK" ) ) ; // null
trace( evaluator.eval( "VIDEO" ) ) ; // [FilterVO:1]
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
trace( evaluator.eval( "VIDEO.filter" ) ) ; // 1

// test in the factory

trace( factory.getObject( "my_filter" ) ) ; // [FilterVO:3]
```

Dans l'exemple ci-dessus, j'utilise la classe **andromeda.vo.FilterVO** (un simple "**value object**") qui permet de stocker une valeur binaire via un attribut numérique "**filter**". Cette classe est très pratique dans une application pour gérer des filtrages binaires sans avoir à passer par une multitude d'objets de type **Boolean**. L'exemple précédent illustre d'ailleurs bien la possibilité de créer des énumérations de valeurs uniques binaires et de combiner selon les besoins les filtres de cette énumération avec la classe **FilterVO**.

En **ActionScript**, nous pouvons par exemple observer un comportement similaire avec les filtres définis dans la classe **Array** avec les constantes statiques : **Array.NUMERIC**, **Array.DESCENDING**, etc. Ces filtres permettent de définir les **options** que nous pouvons utiliser lors du tri d'un **Array** avec les méthodes **sort()** ou **sortOn()**.

V-B-4 - La classe TypeEvaluator

Cette classe est utilisée pour évaluer l'attribut "**type**" dans une **définition d'objet**.

L'attribut "**type**" contient une expression (**String**) qui de façon général représente le nom complet d'une classe définie en natif dans le **FlashPlayer** ou directement dans le code source de votre application (voir chapitre B-1-2). Cette expression est donc directement évaluée et si une référence de la classe concernée existe dans l'animation, la fabrique pourra en cas de besoin générer une instance de cette classe.

L'évaluation du nom complet de classe n'est pas la seule fonctionnalité des objets de type **andromeda.ioc.evaluators.TypeEvaluator**. Cette classe permet de simplifier les expressions définies dans les attributs "**type**" d'une **définition d'objet** avec la possibilité d'utiliser 2 stratégies simples basées sur un filtrage de l'expression (**String**) avec :

- Une collection d'**alias** simple qui permet de simplifier la définition du type d'une **définition d'objet** selon les besoins.
- Un dictionnaire d'expressions qui permettra de formater en cas de besoin le type défini dans une **définition d'objet**.

Ces stratégies peuvent être utilisées en même temps ou séparément selon les besoins d'optimisation d'un contexte externe de configuration d'une fabrique **IoC**.

Voyons maintenant un exemple d'utilisation simplifié de la classe **TypeEvaluator** :

```
import andromeda.ioc.core.TypePolicy ;
import andromeda.ioc.evaluators.TypeEvaluator ;
import andromeda.ioc.factory.ObjectConfig ;

var conf:ObjectConfig = new ObjectConfig() ;

conf.typePolicy       =
 TypePolicy.ALL ; // or: TypePolicy.NONE, TypePolicy.ALIAS, TypePolicy.EXPRESSION

conf.typeAliases      =
[
     { alias:"CoreObject" , type:"vegas.core.CoreObject" }
] ;

conf.typeExpression   =
[
    { name:"map"     , value:"vegas.data.map" } ,
    { name:"HashMap" , value:"{map}.HashMap"  }
] ;
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
var evaluator:TypeEvaluator = new TypeEvaluator( conf ) ;

trace( evaluator.eval( "vegas.core.CoreObject" ) ) ; // [class CoreObject]
trace( evaluator.eval( "CoreObject"  ) ) ; // [class CoreObject]
trace( evaluator.eval( "{HashMap}"   ) ) ; // [class HashMap]
trace( evaluator.eval( "test"        ) ) ; // null
trace( evaluator.eval( "{map}.Test"  ) ) ; // null
```

L'exemple ci-dessus illustre le lien que peuvent avoir la configuration d'une fabrique **IoC** et un objet de type **TypeEvaluator**. Cette évaluation est bien entendu totalement transparente une fois utilisée dans une fabrique basée sur le modèle **IoC** implémenté dans **AndromedAS**.

Nous verrons par la suite un descriptif complet de l'utilisation de la configuration d'un conteneur léger **IoC** avec les attributs **typePolicy**, **typeAliases** et **typeExpression** (voir chapitre C-1-7 - Les attributs de configuration des types dans les définitions d'objets).

V-C - Utilisation des objets Evaluable dans les définitions d'objets

Maintenant que nous avons pu voir comment fonctionne les objets de type **Evaluable** dans une application, il va être beaucoup intéressant d'observer l'utilisations de ces objets dans les **définitions d'objets** d'une fabrique **IoC**.

Dans les **définitions d'objets** nous pouvons donc définir un attribut "**evaluators**" dans certains cas de figure. Cet attribut sera toujours défini avec une collection (**Array**) d'objets de type **Evaluable** ou d'identifiants de type **String**.

Tous les identifiants définis dans la collection "evaluators" permettent de cibler des références d'autres **définitions d'objets** dans la fabrique courante sous condition que ces **définitions d'objets** soient définies avec un "**type**" qui implémente l'interface **Evaluable**. Tout autre objet défini dans cette collection (**Array**) sera ignoré.

Nous pouvons trouver l'attribut "**evaluators**" dans les **définitions d'objets** dans les attributs : "**properties**" et "**arguments**".

Nous avons vu précédement dans le chapite B-4-1-4 l'utilisation de la classe **MultiEvaluator**. Il faut donc considérer que c'est cette classe qui permet le filtrage des valeurs dans les **propriétés** et **arguments** définis dans les **définitions d'objets** pendant qu'un objet et ses dépendances sont créés.

Pour illustrer de façon complète l'utilisation de cette fonctionnalité nous allons réaliser un petit exemple qui utilisera :

- 2 classes simples **test.Appointment** et **test.factory.AppointmentFactory** (définies un peu plus bas).
- Les classes **system.evaluators.DateEvaluator** et **system.evaluators.EdenEvaluator** définies un peu plus haut dans ce chapitre.

Pour commencer nous allons définir la classe **test.Appointment** :

```
package test
{

    /**
     * Represents a commitment representing a time interval with start and end times and a duration.
     */
    public class Appointment
    {

        /**
         * Creates a new Appointment instance.
         */
        public function Appointment( scheduledStart:String = null , scheduledEnd:String = null )
        {
            this.scheduledEnd   = scheduledEnd   ;
            this.scheduledStart = scheduledStart ;
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        }

        /**
         * Indicates the scheduled end time of the appointment.
         */
        public var scheduledEnd:String ;

        /**
         * Indicates the scheduled start time of the appointment.
         */
        public var scheduledStart:String ;

        /**
         * The subject of the appointment.
         */
        public var subject:String ;

        /**
         * Defines the start and the end of the appointment.
         */
        public function setShedule( start:String , end:String ):void
        {
            scheduledStart = start ;
            scheduledEnd   = end ;
        }

    }

}
```

Cette classe va nous permettre d'illustrer l'utilisation de l'attribut "**evaluators**" dans les attributs "**properties**" et "**arguments**" d'une **définition d'objet** aussi bien dans la fonction constructeur que dans les différentes propriétés (attributs et méthodes) définies pour initialiser l'objet au moment de sa création.

Ensuite nous pouvons définir la classe **test.factory.AppointmentFactory** :

```
package test.factory
{
    import test.Appointment;

    /**
     * This factory creates Appointment instances.
     */
    public class AppointmentFactory
    {

        /**
         * Creates a new Appointment instance.
         */
        public function build( scheduledStart:String = null , scheduledEnd:String = null ):Appointment
        {
            return new Appointment( scheduledStart, scheduledEnd ) ;
        }

        /**
         * Creates a new Appointment instance (static method).
         */
        public static function create( scheduledStart:String = null , scheduledEnd:String = null
 ):Appointment
        {
            return new Appointment( scheduledStart, scheduledEnd ) ;
        }

    }
}
```

Cette classe contient une méthode classique **build()** et une méthode statique **create()**.

Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Cette classe va nous permettre d'illustrer l'utilisation de l'attribut "**evaluators**" dans le cadre des stratégies basées sur les attributs "**factoryMethod**" et "**staticFactoryMethod**".

Nous finissons cet exemple plusieurs petits bout de code qui permet de définir selon les cas plusieurs stratégies avec à chaque fois une évaluation d'une expression (**String**) contenant une représentation **eden** d'un objet de type **Date** qui sera évalué tout d'abord avec un objet de type **EdenEvaluator** et ensuite avec un objet de type **DateEvaluator**.

V-C-1 - Évaluation dans une fonction constructeur

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

import system.evaluators.* ;

var objects:Object =
[
    {
        id          : "my_date"  ,
        type        : "String" ,
        arguments   :
        [
            {
                value      : "new Date(2007,11,5,10,22,33)" ,
                evaluators :
                [
                    new system.evaluators.EdenEvaluator(false) ,
                    new system.evaluators.DateEvaluator()
                ]
            }
        ]
    }
] ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( objects ) ;

trace( "---- test evaluators in the constructor arguments" ) ;

trace( "my date : " + factory.getObject("my_date") ) ;
//output: my_date : 05.12.2007 10:22:33
```

V-C-2 - Évaluation dans l'attribut "properties"

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

import test.Appointment ;

import system.evaluators.* ;

var objects:Object =
[

    // evaluators

    {
        id        : "eden" ,
        type      : "system.evaluators.EdenEvaluator" ,
        singleton : true ,
        arguments : [ { value:false } ]
    }
    ,
    {
        id        : "date" ,
        type      : "system.evaluators.DateEvaluator" ,
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        singleton : true
    }

    ,

    // test attributes ("properties")

    {
        id         : "appointment_01"   ,
        type       : "test.Appointment" ,
        properties :
        [
            {
                name       : "scheduledStart" ,
                value      : "new Date(2008,5,30,10,30,00)" ,
                evaluators : [ "eden" , "date" ]
            }
            ,
            {
                name       : "scheduledEnd" ,
                value      : "new Date(2008,5,30,12,40,00)" ,
                evaluators :
                [
                    "eden" ,
                    new system.evaluators.DateEvaluator()
                ]
            }
        ]
    }

    ,

    // test methods ("properties")

    {
        id         : "appointment_02"   ,
        type       : "test.Appointment" ,
        properties :
        [
            {
                name       : "setShedule" ,
                arguments  :
                [
                    {
                        value      : "new Date(2008,5,30,14,00,00)" ,
                        evaluators :
                        [
                            new system.evaluators.EdenEvaluator(false) ,
                            "date"
                        ]
                    }
                    ,
                    {
                        value      : "new Date(2008,5,30,16,30,00)" ,
                        evaluators : [ "eden" , "date" ]
                    }
                ]
            }
        ]
    }

] ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( objects ) ;

var app:Appointment ;

trace( "---- test evaluators in attributes" ) ;

app = factory.getObject("appointment_01") as Appointment ;
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
trace( "> start  : " +  app.scheduledStart ) ;
//output: > start  : 30.06.2008 10:30:00

trace( "> finish : " +  app.scheduledEnd ) ;
//output: > finish : 30.06.2008 12:40:00

trace( "---- test evaluators in methods" ) ;

app = factory.getObject("appointment_02") as Appointment ;

trace( "> start  : " +  app.scheduledStart ) ;
//output: > start  : 30.06.2008 14:00:00

trace( "> finish : " +  app.scheduledEnd ) ;
//output: > finish : 30.06.2008 16:30:00
```

L'exemple ci-dessus illustre l'utilisation de l'attribut "**evaluators**" avec des identifiants permettant de cibler d'autres **définitions d'objets** dans la fabrique ou directement des objets **Evaluable** définis dans le contexte de la fabrique.

V-C-3 - Évaluation dans les attributs "factoryMethod" et "staticFactoryMethod"

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

import test.Appointment ;
import test.factory.AppointmentFactory ;

import system.evaluators.* ;

var linkageEnforcer:Array =
[
    AppointmentFactory
] ;

var objects:Object =
[

    // evaluators

    {
        id        : "eden" ,
        type      : "system.evaluators.EdenEvaluator" ,
        singleton : true ,
        arguments : [ { value:false } ]
    }
    ,
    {
        id        : "date" ,
        type      : "system.evaluators.DateEvaluator" ,
        singleton : true
    }

    ,

    // factory method

    {
        id   : "appointment_factory"   ,
        type : "test.factory.AppointmentFactory"
    }
    ,
    {
        id            : "appointment_03"   ,
        type          : "test.Appointment" ,
        factoryMethod :
        {
            factory    : "appointment_factory" ,
            name       : "build" ,
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
            arguments  :
            [
                {
                    value      : "new Date(2008,6,30,14,00,00)" ,
                    evaluators : [ "eden" , "date" ]
                }
                ,
                {
                    value      : "new Date(2008,6,30,16,30,00)" ,
                    evaluators : [ "eden" , "date" ]
                }
            ]
        }
    }

    ,

    // static factory method

    {
        id                  : "appointment_04"   ,
        type                : "test.Appointment" ,
        staticFactoryMethod :
        {
            type       : "test.factory.AppointmentFactory" ,
            name       : "create" ,
            arguments  :
            [
                {
                    value      : "new Date(2008,7,30,16,00,00);" ,
                    evaluators : [ "eden" , "date" ]
                }
                ,
                {
                    value      : "new Date(2008,7,30,17,00,00);" ,
                    evaluators : [ "eden" , "date" ]
                }
            ]
        }
    }

] ;

var factory:ECMAObjectFactory = ECMAObjectFactory.getInstance() ;

factory.create( objects ) ;

var app:Appointment ;

trace( "---- test evaluators in factory method strategy" ) ;

app = factory.getObject("appointment_03") as Appointment ;

trace( "> start  : " +  app.scheduledStart ) ;
//output: > start  : 30.07.2008 14:00:00

trace( "> finish : " +  app.scheduledEnd   ) ;
//output: > finish : 30.07.2008 16:30:00

trace( "---- test evaluators in static factory method strategy" ) ;

app = factory.getObject("appointment_04") as Appointment ;

trace( "> start  : " +  app.scheduledStart ) ;
//output: > start  : 30.08.2008 16:00:00

trace( "> finish : " +  app.scheduledEnd   ) ;
//output: > finish : 30.08.2008 17:00:00
```


Copyright © 2009 - test. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.