

Réaliser un injecteur de dépendances, en utilisant de bonnes pratiques logicielles

Un injecteur de dépendances ? Peut être cela ne vous parle il pas vraiment. Tant mieux, l'objectif de cet article est d'éclaircir ces termes, et de présenter une implémentation que j'ai eu l'occasion de mettre en place, en m'appuyant sur de bonnes pratiques logicielles.

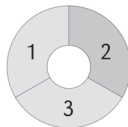
Cet article explique :

- Ce qu'est l'injection de dépendances, et l'inversion de contrôle.
- Quelques bonnes pratiques de développement logiciel.

Ce qu'il faut savoir :

- Utiliser les concepts orienté objet de PHP 5.

Niveau de difficulté



Dans cet article nous parlerons de l'injecteur de dépendances de Spiral, un framework maison réalisé avec quelques amis, et dont l'objectif principal est de découvrir les rouages des frameworks, ainsi que de nous initier aux bonnes pratiques logicielles. Alors que nous travaillions sur ce projet, notre principal but était de réellement comprendre, dans le détail, comment un injecteur de dépendances pouvait fonctionner. *Ré-inventer la roue*, pour mieux comprendre comment une roue fonctionne, en quelque sorte. Aussi, l'objectif de cet article n'est pas de fournir une documentation sur l'utilisation du composant, mais bien d'expliquer *comment* il à été réalisé.

L'injecteur de dépendances est disponible dans une version intégrée à Spiral ou dans une version *standalone*. Vous pouvez trouver le code sur le dépôt mercurial associé au projet. À l'heure où j'écris ces lignes, l'injecteur de dépendances de spiral n'est pas encore terminé, mais est dans un état avancé, et devrait être disponible en février 2010. L'ensemble des exemples de ce document sont en PHP, mais les concepts discutés ici peuvent être (et sont) implémentés dans d'autres langages.

Comment gérons-nous nos objets ?

Avant toute chose, il est indispensable de bien comprendre ce qu'est l'inversion de contrôle. Lorsque nous réalisons des logiciels en utilisant le *paradigme* orienté objet, nous travaillons avec des classes, et la majeure partie du temps, nous faisons interagir ces classes entre elles. En pratique, certaines classes sont dépendantes d'autres classes.

Pour mettre un exemple derrière ces concepts, tout au long de ce document, nous allons nous mettre dans la peau d'Alice, une jeune fille qui adore manger des glaces, spécialement celles à la fraise ! Certains disent même qu'Alice est dépendante de la glace à la fraise, comme le souligne le Listing 1.

Il est clair, au regard de cette implémentation, qu'à chaque fois qu'Alice mange une glace, il s'agit d'une glace à la fraise. Génial, mais un jour, la mère d'Alice souhaite lui faire découvrir d'autres parfums...

En réalité, avec cette implémentation, il est impossible de changer la glace qu'Alice va manger.

L'inversion de Contrôle (IoC)

Il apparaît nécessaire de supprimer les dépendances entre nos deux classes, pour permettre à Alice de goûter de nouveaux parfums. Comment ? C'est assez simple, regardez donc le code :

```
class Alice {
    public function mangerGlace(Glace $glace){
        $glace->manger();
    }
}
```

Quand Alice mange une glace (via la méthode `mangerGlace`), nous devons lui passer la glace, ce n'est plus elle qui choisit, nous le faisons à sa place. Ce principe est connu comme étant le principe d'Hollywood : *Ne nous appelez pas, nous vous appellerons*. En d'autres termes, n'utilisez pas l'opérateur `new` dans vos classes, mais préférez passer (ou qu'on vous passe) les objets par référence.

Alice peut faire d'autres choses avec sa glace, la laisser tomber par terre par exemple, grâce à la méthode `lacherGlace()`. Nous pouvons alors choisir de passer la glace à cette méthode également, ou choisir de la donner directement à Alice, la laissant s'occuper du reste, et évitant de lui passer une glace pour chaque action qui en nécessite une.

Le Listing 2 présente la classe `Alice`, faisant usage de l'inversion de contrôle. Il est

Listing 1. Une classe avec des dépendances.

```
class Alice {
    public function mangerGlace(){
        $glace = new GlaceALaFraise();
        $glace->manger();
    }
}
```

bien plus facile maintenant de choisir la glace à donner à *Alice*, et ainsi de contrôler les dépendances d'*Alice* vis à vis de la glace. Ici, il subsiste des dépendances dans le code. Il s'agit de dépendances vis à vis de contrats (interfaces) et non d'implémentations données (classes), puisque j'ai choisi d'utiliser le paradigme de programmation par contrats.

Et c'est tout pour le principe d'inversion de contrôle (pas les glaces à la fraise, contrôler les dépendances) ! Il s'agit *simplement* du fait d'inverser le flux de contrôle de vos application, en déléguant à un plus haut niveau la création des objets.

Injection de dépendances

Maintenant que le concept d'inversion de contrôle est clair, expliquons ce qu'est l'injection de dépendances. Les deux concepts sont assez proches, et souvent utilisés de pair, mais il est important de bien saisir la frontière entre les deux.

Dans la méthode `mangerGlace`, nous considérons que la glace en question est déjà donnée à *Alice*. C'est un comportement vraiment utile : Nous n'avons plus à nous occuper de la manière dont la glace est arrivée là, nous l'avons déjà (dans une propriété privée par exemple). Dans la section précédente, *Alice* était *dépendante* de sa glace. En inversant le flux de contrôle, le comportement d'*Alice* vis à vis des glaces est plus facilement contrôlable, et testable (utiliser des *mocks*, ou *bouillons de tests* est aussi facile que de régler une propriété, nous parlerons de tests plus tard).

Notre travail (celui de la mère d'*Alice*), est de créer les objets et de les passer à *Alice*. Les *injecter* est le bon mot. En utilisant des mutateurs, ou en utilisant le constructeur, injectant les objets nécessaires. Allons-y :

```
$alice = new Alice();
$glaceAuPaté = new GlaceAuPaté();
$alice->setGlace($glaceAuPaté);
```

L'inversion de contrôle est donc le fait d'exposer des méthodes publiques (ou des constructeurs) pour régler certaines propriétés,

et l'injection de dépendances est le fait de, justement, injecter ces dépendances, utiliser ces méthodes et constructeurs.

Un conteneur ?

L'exemple utilisé jusqu'ici est volontairement simple, et il a été choisi afin d'expliquer les concepts le plus clairement possible : *nous avons uniquement deux classes, et une dépendance*. En pratique, vous serez sûrement d'accord pour dire qu'un projet est rarement aussi simple. Aussi, dans les projets importants, la gestion du cycle de vie des objets et de l'injection de leurs propriété peut rapidement devenir un vrai casse tête. L'idéal est alors d'automatiser le processus de création, d'injection et de gestion de ces cycles de vie. Le conteneur fait exactement ça.

Pourquoi *conteneur* ? Parce que la création automatique et l'injection est effectuée grâce à un objet, qui se charge de contenir toutes les informations sur les dépendances. Une fois les objets créés, le conteneur garde une référence vers ces derniers au cas où nous en aurions encore besoin (voir la définition de portée d'un service - les *scopes* - plus loin).

Le conteneur va donc se charger d'injecter les objets pour nous, en quelque sorte, il fait le travail de la Mère d'*Alice* à sa place. Nous souhaitons donc que lorsque nous appelons *Alice*, via le conteneur, elle nous soit retournée avec une glace prête à être mangée !

```
$alice = $container-
>getService('Alice');
$alice->mangerGlace();
```

Ici, le conteneur a injecté la bonne glace à *Alice* (peu importe laquelle d'ailleurs, nous souhaitons juste avoir une glace). Si la glace elle même avait été dépendante d'autres objets (disons, des noix de coco par exemple), c'est le rôle du conteneur que de résoudre l'ensemble des dépendances, dans le bon ordre, simplifiant au maximum la tâche de gestion des dépendances entre les objets et les classes, laissant la tâche simple pour le développeur.

Concepts logiciels

Maintenant que les concepts d'inversion de contrôle et d'injection de dépendances sont clairs (enfin, j'espère), nous pouvons commencer à parler de *comment* nous avons réalisé cette bibliothèque. Les concepts discutés ici sont des concepts assez simples, dont le principal objectif est de fournir une structure solide aux composants. Chaque composant a ainsi un rôle et un emplacement précis au sein de notre architecture.

Le Schéma

Dans le schéma, et dans l'injecteur de dépendances en général, un service est un objet qui est géré par le conteneur. Le schéma représente les liens entre les différents services. Il décrit les dépendances de nos objets. Si vous connaissez le patron de conception de *fabrique abstraite*, vous pouvez vous représenter le schéma comme une configuration alors que le conteneur serait la fabrique elle même (ou quelque chose d'approchant).

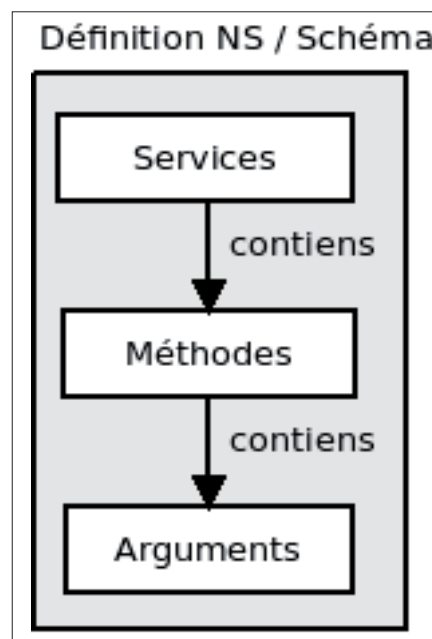


Figure 1. Le schéma représente les liaisons de nos services

Le schéma contient toutes les informations sur les méthodes qui doivent être appelées pour injecter les objets, le type des arguments qui doivent être passés, et tout autre type d'information potentiellement utile au moment de l'injection. Pour en revenir à notre exemple, le schéma contiendrait des informations sur le type de glace qui doit être passée à *Alice* (une glace à la fraise bien sûr), et sur la manière de donner cette glace à *Alice* (via la méthode `setGlace()`). Jusqu'à maintenant, nous avons parlé de dépendances simples, mais le schéma peut aussi gérer d'autres types de services, méthodes et arguments. Tout est décrit dans les sections suivantes : *Services*, *Methods* et *Attributes*.

Listing 2. Une classe qui utilise l'inversion de contrôle.

```
class Alice {
    protected $_glace = null;

    public function setGlace(Glace $glace){
        $this->_glace = $glace;
    }

    public function mangerGlace(){
        $this->_glace->manger();
    }

    public function lacherGlace(){
        $this->_glace->lacher();
    }
}
```

Tableau 1. Liste des types de services

Type	Description
Défaut	Un service <i>simple</i> , composé de méthodes, et qui peut être construit comme un simple objet
Alias	Un alias vers un autre service. Seul le nom est différent. Ce type de service permet de gérer facilement les dépendances dans le temps. «Pour le moment, il s'agit d'un alias, mais peut être qu'un jour nous aurons besoin d'un autre type de service»
Héritage de services	Plutôt que de se répéter maintes et maintes fois lors de la description de services qui se ressemblent, il est possible d'utiliser l'héritage. Cela ressemble grandement à l'héritage de classes : les méthodes que vous redéfinissez ou ajoutez dans les services enfants écraseront ceux des parents.

Services

Un service représente un objet. Dans notre exemple, la *Glace* et *Alice* sont des services. Un service se compose de :

- un nom,
- un ensemble de méthodes,
- une manière de se construire,
- une portée (scope).

La portée d'un service définit comment la durée de vie des services doit être gérée par le conteneur : Est-ce que le service doit rester dans le conteneur pendant toute la durée du script (singleton), ou doit-il être systématiquement supprimé après avoir été construit (prototype) ? L'instance de l'objet courant peut être la même pour l'ensemble des services si la portée du service est définie comme étant un *singleton*, ou être à chaque fois différente si la portée est définie comme *prototype*.

D'autres types de portées peuvent être imaginées comme une portée de «session», qui retournerait la même instance durant une session unique, ou une sorte de portée «immortelle», qui retournerait toujours le même objet, en faisant

persister cet objet à travers différentes sessions. L'injecteur de dépendances est fourni avec les types de service décrits dans le Tableau 1.

Méthodes

Chaque service contient des méthodes. Une méthode permet d'injecter certains paramètres dans nos services, ou de définir certaines ressources qui doivent être appelées au moment de la construction des services. Dans le cas d'Alice, `setGlace()` est une méthode. Une méthode est composée d' :

- un nom,
- optionnellement, un nom de classe,
- une liste d'arguments,
- une information disant si la méthode est statique ou non.

Vous pouvez trouver les différents types de méthodes actuellement implémentées dans le Tableau 2.

Arguments

Les méthodes contiennent donc des arguments, et il existe plusieurs types d'argu-

ments également. Les arguments sont le bout de la chaîne services / méthodes / arguments.

Stratégies de construction

Maintenant que nous avons un schéma qui représente les relations entre nos services, nous allons nous occuper de la construction de ces services. Nous avons choisi de séparer complètement les logiques de construction et de définition, pour permettre de favoriser un maximum d'usages possibles pour l'un et l'autre des composants.

Chaque type, dans le schéma, peut être lié à un type de stratégie pour se construire. Il y a donc plusieurs stratégies de construction pour les services, les méthodes et les arguments. L'intérêt d'utiliser des stratégies de construction est de permettre à chacun de nos types, dans le schéma, de se construire *eux mêmes*, en utilisant leur méthode `build()`, qui va elle déléguer la tâche de construction aux stratégies. En interne, il est possible d'utiliser des stratégies de construction différentes, et d'en changer à tout moment. Ce comportement suit, en fait, le patron de conception stratégie.

Tableau 2. Liste des types de méthodes

Type	Description
Défaut	Une simple méthode, avec des arguments. Peut être une méthode statique
Attributs	Utilisé pour régler directement les propriétés en utilisant les attributs publics de l'objet (<code>\$service->attribut = \$valeur</code>). Ce type de méthode peut contenir uniquement un argument. Il peut paraître étrange de gérer les attributs comme des méthodes. En réalité, il est important de comprendre la différence entre une méthode et un argument. Alors qu'un argument représente une valeur, une méthode représente une manière d'utiliser ces arguments. Dès lors, il paraît plus logique de gérer les attributs comme des méthodes que comme des arguments.
Rappels (Callback)	Plutôt que de se répéter maintes et maintes fois lors de la description de services qui se ressemblent, il est possible d'utiliser l'héritage. Cela ressemble grandement à l'héritage de classes : les méthodes que vous redéfinissez ou ajoutez dans les services enfants écraseront ceux des parents.

Tableau 2. Liste des types de méthodes

Type	Description
Défaut	Types PHP natifs (int, string, float etc)
Conteneur	Il est possible d'injecter directement le conteneur. Ce type d'argument n'est utilisé que par les services qui nécessitent d'utiliser le conteneur. Ils sont appelés services «ContainerAware».
Service courant	Il est possible d'injecter le service en cours, et de l'utiliser comme argument. En pratique, ceci est uniquement utile pour les méthodes de rappel (callback)
Argument vide	Il s'agit d'un type d'argument qui n'a pas de valeur. L'argument «conteneur» et «service courant» étendent ce type. Attention, l'argument vide est différent de null.
Référence à un service	C'est un des types d'argument le plus utilisé, il représente un autre service.
Argument résolu grâce aux services	Parfois, il est utile d'utiliser un service pour récupérer un argument, je pense à la configuration entre autres. Ce type d'argument utilise donc une méthode spécifique d'un autre service pour être résolu.

Listing 3. Exemple d'utilisation des monteurs

```
// utilisation du moniteur XML pour construire le schema
$builder = new XmlBuilder();
$schema = $builder->build('schema.xml');
// et une fois le schéma construit, passons le au conteneur !
$container = new DefaultContainer($schema);
```

Builders / Monteurs

Puisque nous parlons de patrons de conception (*design patterns*), parlons du motif *Monteur*. Vous serez sans doute d'accord avec moi pour dire qu'écrire un schéma entièrement à la main, en utilisant les classes dont nous avons parlé un peu plus haut peut s'avérer rapidement assez pénible. En tout cas, pour l'avoir expérimenté lors de l'écriture des tests, je peux dire qu'il ne s'agit pas d'un gain de temps, loin de là.

Une solution pratique consiste à utiliser le motif *Monteur*. L'idée est d'écrire le schéma sous une forme sympathique et facile à écrire pour nous, développeurs, et d'utiliser une classe intermédiaire pour transformer notre représentation du schéma dans la représentation compréhensible par notre composant. Cette classe intermédiaire *monte* donc notre schéma, en déchiffrant une autre structure.

Le premier type de *monteur* qui me vient à l'esprit (le plus pratique, en fait), est le *monteur XML*. Il est capable de lire un schéma, décrit au format XML, et de construire le schéma en utilisant les objets de notre bibliothèque. L'écriture du schéma XML à plusieurs avantages : il est facile à écrire, permet d'utiliser des outils extérieurs pour l'éditer facilement, et bénéficie, grâce à XML Schema, d'une auto-complétion et d'une vérification à la volée, lors de l'écriture.

Le Listing 3 montre un exemple d'utilisation du moniteur XML. Le fichier XML n'est volontairement pas présent ici, pour des raisons de place. Il est possible de trouver un exemple d'utilisation en ligne, sur le dépôt de spiral. Les injecteurs de dépendances *Google Juice* et *Spring* permettent l'utilisation des annotations directement dans le code, pour définir les règles d'injection (le schéma pour nous).

Bien qu'il ne s'agisse pas d'un comportement recommandé (les annotations ne sont exploitables que par un type d'injecteur, même si une spécification est actuellement en cours), il est possible d'utiliser la réflexion sur un projet, et de la combiner à l'utilisation d'annotations pour déduire facilement la structure de notre schéma, pour le remplir ensuite à notre guise. Ce composant est également un *monteur*. Les monteurs suivants sont fournis de base :

- Le moniteur XML.
- Le moniteur PHP, qui utilise une interface fluide, pour permettre des confi-

gurations de ce type : `$moniteur->addService()->withMethod()`.

- Le moniteur Réflexion (utilise la réflexion sur nos classes pour construire un schéma).

Dumpers

Un *dumper* est un objet qui copie des données d'un type de format vers un autre. Effectivement, il peut s'avérer utile d'avoir une manière simple de se représenter un schéma déjà défini.

Les dumpers permettent par exemple de représenter un schéma sous une forme graphique, ou bien sous une forme plus compréhensible pour nous, avec un simple texte par exemple. Il est donc vraiment facile de montrer les dépendances de vos projets, en utilisant simplement le dumper Dot (qui est le format utilisé par *graphviz*) par exemple. Voici la liste des dumpers :

- Le dumper texte.
- Le dumper Dot (*graphviz*).
- Le dumper XML.
- Le dumper PHP.

Ces composants laissent entrevoir des pistes intéressantes : il est ainsi possible d'écrire ses classes, puis de générer un schéma partiel grâce au moniteur réflexion, de le compléter à la main (avec de l'auto-complétion), et de le monter à nouveau, grâce au moniteur XML.

Implémentation

Voici quelques règles que nous avons suivies lors du développement en lui-même.

Espaces de noms / PHP 5.3

Alors que nous nous penchions sur ce projet, PHP 5.3 n'était pas encore sorti, mais puisque cette version apportait des fonctionnalités vraiment intéressantes (late static binding, espaces de noms et closures), nous avons choisi d'utiliser alors la version en cours de développement de PHP 5.3.

Maintenant, PHP 5.3 est disponible en version stable, et permet de faire fonctionner notre projet. Notre bibliothèque se sépare selon les espaces de noms suivants :

- L'espace de nom `Construction`, qui contient toutes les classes liées au concept de construction (les stratégies de construction).

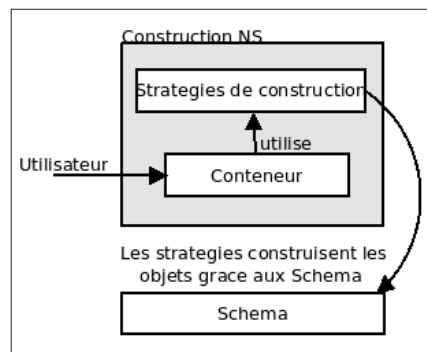


Figure 2. Les stratégies de construction

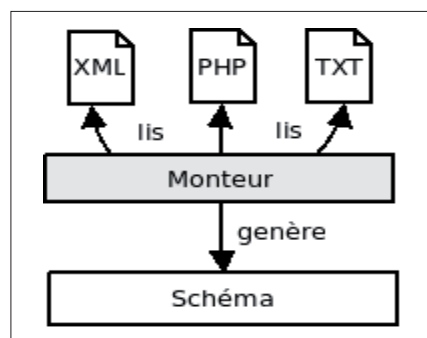


Figure 3. Les monteurs

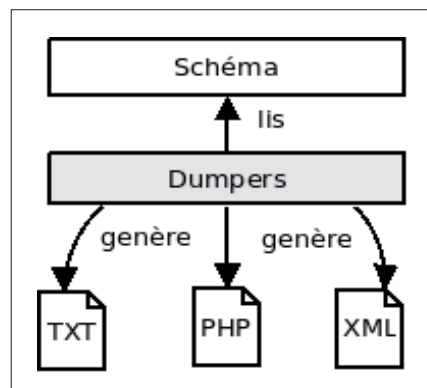


Figure 3. Les dumpers

- L'espace de nom `Definition`, qui contient le schéma.
- L'espace de nom `Transformation` qui contient les Dumpers et les Monteurs.

Développement piloté par les tests (TDD)

Ce projet fut également l'occasion d'écrire nos premiers tests, pour finir par utiliser une approche pilotée par les tests. Le développement piloté par les tests préconise de réaliser ses tests avant d'écrire ses classes. Au début, ça chatouille un peu, mais on comprend rapidement l'intérêt de cette méthodologie, qui est une vraie bonne pratique.

Écrire ses tests avant d'avoir codé la classe nous oblige à la fois à privilégier une utilisation logique de nos composants, et à fixer les interfaces. Le code produit est réellement comme on souhaite l'utiliser, et non pas comme il est plus facile de l'implémenter. Écrire

Listing 4. Les dépendances sont décrites dans un fichier «schema.xml»

```
<?xml version="1.0" encoding="UTF-8"?>
<services [...]>
  <service id="alice" class="Alice">
    <method name="setGlace">
      <argument ref="glace" />
    </method>
  </service>
  <service id="glace" class="GlaceAuPate">
    <constructor>
      <argument type="string" value="gelatine" />
    </constructor>
  </service></services>
```

Listing 5. Les dépendances sont décrites dans un fichier XML

```
interface Glace{
    function manger();
    function lacher();
}
class GlaceAuPate implements Glace{
    protected $_additif;
    public function __construct($additif){
        $this->_additif = $additif;
    }
    public function manger(){
        echo 'Et une glace à la '.$this->_additif.', une !';
    }
    public function lacher(){
        echo 'oops!';
    }
}
// on récupère le schema grace au monteur XML
$dumper = new XmlDumper();
$schema = $dumper->dump('schema.xml');
$container = new DefaultContainer($schema);
/* puis on peut utiliser directement le conteneur, qui nous retourne des
service déjà configurés, et prêts à être utilisés ! */
$alice = $container->alice;
// ou $container->getService('alice');
/* à ce moment, l'injecteur de dépendances, à, tout seul, appelé la
méthode « setGlace » de l'objet $alice */
$alice->mangerGlace();
// retourne « Et une glace à la gélatine, une ! »
```

des tests, c'est aussi penser à l'ensemble des scénarios d'utilisation de ces classes, même les plus farfelus. Cela nous oblige à réfléchir à tous ces cas d'utilisation, et ça fait le plus grand bien !

Pour revenir aux tests, ils permettent de tester que notre application se comporte bien comme elle le devrait, mais cela permet aussi de détecter rapidement des régressions que de nouvelles fonctionnalités peuvent apporter. Rapidement, on écrit des tests pour tout : bugs, idées, etc. Ça favorise vraiment le développement d'une application. Un peu plus haut, je parlais de Mock objets (ou objets bouchon, en français). Je vous laisse consulter l'article wikipédia sur les mocks pour

vous faire une idée plus précise, mais il s'agit, rapidement, d'objets qui permettent de simuler le comportement d'autres objets, ces derniers pouvant communiquer avec la suite de tests.

Interfaces

Dans l'ensemble de nos classes, nous essayons d'utiliser des interfaces plutôt que des implémentations particulières. Pourquoi ? Parce que travailler avec des interfaces nous permet de changer à tout moment d'implémentation !

Dans le cas d'Alice, elle n'est pas dépendante d'un type particulier de glace (celle à la fraise), mais simplement aux glaces, à l'interface

Glace, pour être exact. Chacune des interfaces ci-dessous représente un comportement décrit plus haut :

- Schema.
- Service.
- Method.
- Argument.
- Container.
- Dumper.
- Builder.

L'écriture des classes

Pour écrire nos classes, et parce que nous souhaitons fournir un système facilement extensible, nous fournissons quasi systématiquement une interface, et une classe abstraite, pour que chaque concept puisse être étendu facilement. D'ailleurs, l'écriture des classes en elle-même est assez simple, une fois que tous les concepts ont été décrits et sont clairs.

Vous pouvez regarder le code sur le dépôt mercurial de spiral. Je ne vois pas grand chose à ajouter à propos de l'implémentation, si ce n'est, peut-être, qu'il est indispensable de commenter votre code : cela permet aux potentiels futurs contributeurs de s'y retrouver facilement, et de comprendre le détail des opérations !

Reprenons notre exemple

Et Alice dans l'histoire ? Le listing 4 fournit un fichier XML de description du schema de dépendances d'Alice, et le listing 5 décrit comment utiliser notre injecteur dans ce cas précis. Il faut bien sûr garder à l'esprit que l'utilisation d'un injecteur de dépendances est surtout utile sur des projets d'envergure. Pour Alice, il semble bien plus simple d'injecter les dépendances à la main.

Conclusion

J'espère que cet article vous aura intéressé, en tout cas j'ai pris beaucoup de plaisir à l'écrire, et vous aurez au moins appris comment nous avons choisi d'implémenter un injecteur de dépendances en utilisant quelques bonnes pratiques logicielles ! Si vous êtes intéressés pour discuter à ce propos, ou à propos d'autres concepts logiciels, vous pouvez me contacter, je serais ravi d'échanger avec vous.

ALEXIS MÉTAIREAU

L'auteur est actuellement en 4ème année au sein de l'ESI SUPINFO. Il travaille actuellement en alternance chez Makina Corpus, au sein du pôle django/python. Parallèlement, il publie de temps à autre des réflexions sur l'architecture logicielle, sur son blog personnel : <http://www.notmyidea.org>. Vous pouvez contacter l'auteur en utilisant l'adresse alexis@spiral-project.org.

Sur Internet

- <http://hg.spiral-project.org/> – Le dépôt de spiral,
- <http://www.spiral-project.org/> – Le site de spiral,
- http://fr.wikipedia.org/wiki/Programmation_par_contrat – La programmation par contrats.
- [http://fr.wikipedia.org/wiki/Strat%C3%A9gie_\(patron_de_conception\)](http://fr.wikipedia.org/wiki/Strat%C3%A9gie_(patron_de_conception)) – Le patron de conception stratégique.