

Philippe PRADOS
site@philippe.prados.name

JSR269



*Préservez l'environnement,
n'imprimez pas ce document*

TABLE DES MATIÈRES

Table des matières.....	2
Table des matières.....	2
1 Les annotations.....	3
2 Utilisation au runtime.....	4
3 JSR268.....	4
4 Génération d'une ressource.....	4
5 Génération d'une classe.....	6
6 Eclipse.....	8
7 Autres solutions.....	9

Avant-propos

Les annotations sont une nouveauté de la version 5 de Java. Elles permettent d'ajouter des méta-informations aux éléments du langage (classe, attribut, méthode, paramètre, variables). Ces données peuvent être exploitées à l'exécution, par introspection des classes, mais elles peuvent également être utilisées lors de la compilation. Le JSR269 propose une architecture permettant la prise en compte des annotations lors de la compilation pour générer des fichiers de paramètres ou de nouvelles classes.

1 LES ANNOTATIONS

Les frameworks ont souvent besoins d'enrichir d'informations les classes afin de faciliter leur travail. Par exemple, pour convertir un objet en XML, il faut déclarer un mapping entre les attributs et la syntaxe XML à produire. Cette déclaration peut être effectuée en parallèle de la classe, dans un fichier de paramètre par exemple, mais cela présente le risque d'une désynchronisation entre le source java et le mapping XML. Parfois, des classes doivent être modifiées en parallèles, car elles présentent une cohérence globale. Par exemple, un BeanInfo est une classe qui décrit une autre classe. Cette dernière doit être conforme aux services proposés par la classe qu'il représente.

Pour permettre d'enrichir les classes, sans remettre en cause leurs sémantiques, Java 5+ propose d'utiliser des annotations. Il s'agit de modificateur comme le sont `public`, `private` ou `protected`, mais dont le nom et les valeurs associées sont à la charge du développeur.

Une annotation se déclare pratiquement comme une interface. Il suffit d'ajouter un arobase au mot clef `interface` pour en faire une annotation. Les annotations peuvent avoir des valeurs constantes de type primitif, `String`, `Class` ou `enums`. Ces valeurs sont identifiées par les méthodes de l'interface. Elles peuvent également avoir une valeur par défaut, en utilisant une syntaxe particulière.

```
public @interface MonAnnotation
{
    int id();
    Class cl();
    String value() default "hello";
}
```

Pour utiliser une annotation, il faut utiliser son nom, préfixé par un arobase. Entre parenthèse, il est possible d'indiquer une valeur constante pour chaque attribut, via une écriture du type `<nom>=<valeur>`. Si une seule valeur est présente, il est recommandé de l'appeler `value`, car elle peut ainsi être valorisée directement. Les écritures suivantes sont valides pour l'annotation que nous venons de créer.

```
@MonAnnotation("Bonjour")
@MonAnnotation(id=123,cl=String.class,value="hey")
```

Les annotations doivent être placée juste avant une déclaration du langage. C'est-à-dire devant un package, une classe, une interface, une méthode, un attribut, un paramètre d'une méthode et même devant une variable locale à une méthode.

```
@MonAnnotation
class MaClass
{
    @MonAnnotation
    int monAttribut;
    @MonAnnotation
    void maMethode(@MonAnnotation int param)
    {
        @MonAnnotation
        int val;
    }
}
```

Pour déclarer une annotation pour un package, il faut créer un fichier `package-info.java` avec uniquement la déclaration du package, préfixé par une ou plusieurs annotations.

```
@MonAnnotation
package mon.package;
```

Les annotations peuvent également être annotés. On parle alors de méta-annotation. Certaines méta-annotations sont traitées par le compilateur. Par exemple, les annotations peuvent être visible uniquement lors de la compilation, présentent dans les classes, ou également présentent à l'exécution. Il est également possible d'indiquer que certaines annotations ne peuvent être utilisées que devant certaines structures de code. Uniquement devant une classe par exemple.

L'extrait suivant indique que l'annotation `@MonAnnotation` ne peut être utilisée que devant une classe ou une méthode (`@Target`), qu'elle doit être documenté lors de la génération de la Javadoc (`@Documented`), qu'elle est héritée par les sous-classes (`@Inherited`) et qu'elle est visible à l'exécution (`@Retention`).

```
@Target({ElementType.TYPE,ElementType.METHOD})
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MonAnnotation
{
    String value() default "hello";
}
```

Le compilateur sait gérer d'autres annotations. `@Deprecated` est une autre approche pour signaler qu'une classe ou une méthode est obsolète. `@SuppressWarnings` permet de contrôler les messages d'erreurs émit par le compilateur. `@Override` permet de signaler qu'une

méthode en surcharge une autre. Le seul intérêt de cette annotation est de générer une erreur si ce n'est pas le cas. Cela permet d'identifier des erreurs dans la signature d'une méthode ou lors de l'évolution de la super classe. C'est un élément de qualité important. Toute méthode surchargée devrait avoir cette annotation.

2 UTILISATION AU RUNTIME

Si les annotations sont déclarées comme étant visible à l'exécution (`@Retention(RetentionPolicy.RUNTIME)`) il est possible de les identifier en invoquant les API d'introspections. Cette approche peut être utilisée pour transformer une implémentation d'une interface par une annotation.

```
public interface CouleurPreferee
{
    String getCouleur();
}

public @interface Couleur
{
    String value();
}

public abstract class AbstactCouleurPreferee implements CouleurPreferee
{
    @Override
    public String getCouleur()
    {
        return getClass().getAnnotation(Couleur.class).value();
    }
}

@Couleur("rouge")
public class CouleurPrefereeImp extends AbstactCouleurPreferee
{
}
```

La classe `AbstactCouleurPreferee` propose une implémentation de la méthode `getCouleur()` s'appuyant sur l'annotation de la classe. La classe `CouleurPrefereeImp` ne surcharge pas de méthode. Elle se contente d'une annotation.

Attention, les annotations ne sont pas hérité. Des API ouvertesⁱ sont proposées pour contourner cela.

Mais, souvent, les annotations servent à découvrir des paramètres. Il n'est donc pas possible de connaître à l'avance, les classes annotées. Il faut dans ce cas parcourir toutes les classes du projet, analyser les fichiers `.class` sans les installer dans la JVM, pour découvrir éventuellement des annotations pertinentes. C'est ce que font les serveurs d'applicationsⁱⁱ pour découvrir les annotations des spécifications Servlet 3.0. Donc, plus il y a de classes dans le projets, plus ce processus est long et couteux.

Le problème de cette approche est qu'elle n'est pas généralisée et supportée par une API. Ainsi, chaque framework doit parcourir et analyser lui-même toutes les classes pour identifier les candidats portant les annotations convoitées.

Pour des raisons d'optimisations, l'analyse des classes est grossière. Seule la présence d'une chaîne de caractère dans le pool de constante des classes, correspondant a priori à une annotation est détectée. Une classe sans annotations, mais ayant la bonne chaîne de caractère peut être identifié à tort. Par exemple, une classe possédant une chaîne de caractère « `Ljavax/servlet/annotation/WebServlet;` » sera considérée comme une candidate à l'annotation.

3 JSR268

Depuis la version 6 de la JVM, il est possible d'intervenir lors de la compilation des sources Java, pour générer de nouvelles classes ou des ressources déduites des annotations des classes.

Lors de la compilation d'un fichier source Java, le compilateur recherche des « Processors » associés à différentes annotations. Un processeur est une classe java répondant à des interfaces, invoqué par le compilateur après l'analyse syntaxique des classes à compiler. Cette dernière peut alors analyser la ou les classes en cours d'analyse, générer d'autres classes ou des fichiers de configurations, et laisser ensuite le compilateur produire les classes.

En fait, le mécanisme suit des cycles, dirigés par les annotations. Une analyse de toutes les classes est effectuées. Il en ressort une liste d'annotations découvertes et une liste de classes à compiler. Puis, le compilateur recherche parmi tous les processeurs, ceux étant en charge de ces annotations. Un processeur peut déclarer être en charge de toutes les annotations. Chaque processeur est alors invoqué. Ces derniers peuvent interroger l'état du compilateur pour, par exemple, récupérer la liste de toutes les méthodes, les annotations associées, etc. De ces informations, les processeurs peuvent se contenter d'afficher une alerte à l'utilisateur en cas d'usage jugé malheureux, ou générer de nouvelles classes en source `.java` ou directement compilé en `.class`. Il est également possible de générer une ressource. Un fichier `web.xml` par exemple.

Le cycle étant terminé, les classes initiales sont compilées. Les listes des annotations et des sources restants sont adaptées. Une nouvelle invocation des processeurs est effectués. Le cycle recommence jusqu'à ce qu'il ne reste plus aucun source à compiler.

Pour pouvoir proposer un processeur, il faut utiliser une archive Java proposant le service. Pour cela, il faut créer un fichier `META-INF/services/javax.annotation.processing.Processor` avec une ligne en UTF-8, indiquant le nom de la classe implé-

mentant l'interface correspondante. Ainsi, la compilation avec cette archive dans le CLASSPATH permet d'utiliser ce processeur lors de la compilation de nouvelles classes.

C'est ce qui est étrange avec java, compiler une classe avec une archive entraîne une modification de la compilation elle-même ! Mais c'est également ce qui fait son charme, pouvoir intervenir profondément pour réduire le nombre de fichier de paramètres et autres scories.

4 GÉNÉRATION D'UNE RESSOURCE

Par exemple, les spécifications 3.0 des servlets proposent d'annoter les servlets avec une annotation `@WebServlet`.

```
@WebServlet(  
    name = "SimpleServlet",  
    urlPatterns="/myservlet"  
)  
  
public class MyServlet extends HttpServlet  
{  
    ...  
}
```

Nous pouvons proposer un processeur pour générer le fichier `web.xml` à partir de ces informations.

```
@SupportedAnnotationTypes("javax.servlet.annotation.WebServlet")  
@SupportedSourceVersion(SourceVersion.RELEASE_6)  
public class MyProcessor extends AbstractProcessor  
{  
    public boolean process(final Set<? extends TypeElement> annotations,  
                           final RoundEnvironment env)  
    {  
        final File file = processingEnv.getFile();  
        final Messenger messenger = processingEnv.getMessager();  
  
        if (!env.processingOver())  
        {  
            try  
            {  
                FileObject jfo=file.createResource(StandardLocation.CLASS_OUTPUT, "",  
                                                    "web.xml", (Element)null);  
                messenger.printMessage(Kind.NOTE, "Writing web.xml");  
                PrintWriter pw = new PrintWriter(jfo.openOutputStream());  
                pw.write(  
                    "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>\n"+  
                    "<web-app id=\"sample\"\n"+  
                    "    version=\"2.4\"\n"+  
                    "    xmlns=\"http://java.sun.com/xml/ns/j2ee\"\n"+  
                    "    xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"\n"+  
                    ");  
                for (TypeElement elm:ElementFilter.typesIn(  
                    env.getElementsAnnotatedWith(WebServlet.class)))  
                {  
                    WebServlet annotation=elm.getAnnotation(WebServlet.class);  
                    pw.write(  
                        "<servlet>\n"+  
                        "    <servlet-name>"+annotation.name()+"</servlet-name>\n"+  
                        "    <servlet-class>"+elm.getQualifiedName()+"</servlet-class>\n"+  
                        "</servlet>\n");  
                    pw.write(  
                        "<servlet-mapping>\n"+  
                        "    <servlet-name>"+annotation.name()+"</servlet-name>\n";  
                    if (annotation.urlPatterns().length>0)  
                        pw.write("<url-pattern>"+annotation.urlPatterns()[0]+"</url-pattern>\n");  
                    pw.write("</servlet-mapping>\n");  
                }  
                pw.write("</web-app> \n");  
                pw.close();  
            }  
            catch (IOException ioe)  
            {  
                messenger.printMessage(Kind.ERROR, ioe.getLocalizedMessage());  
            }  
        }  
        return true;  
    }  
}
```

Ainsi, en déclarant ce processeur dans le fichier `META-INF/services/javafx.annotation.processing.Processor` et en utilisant l'archive lors d'une compilation, le fichier `web.xml` est automatiquement généré.

Il y a une petite difficulté avec cette démarche récursive. En effet, comment compiler la classe du processeur si ce dernier n'existe pas encore ? Le compilateur, découvrant la présence du service, cherche à utiliser le processeur pour lui-même. Il faut alors ajouter, lors de la première compilation, le paramètre `-proc:none`. Ensuite, la compilation peut s'effectuer normalement. Pour éviter ces subtilités, il est préférable d'utiliser deux projets. L'un s'occupe de compiler le processeur et possède ce paramètre, l'autre applique le processeur en important l'archive et n'a pas besoin de paramètre.

Pour voir les cycles lors de la compilation, ajoutez le paramètre `-XprintRounds` à la ligne de commande. Il est également possible d'indiquer explicitement le processeur à utiliser. Ajoutez le paramètre `-processor` pour cela. Un petit `javac -help` révélera toutes les options intéressantes.

5 GÉNÉRATION D'UNE CLASSE

Cette technologie est séduisante pour exploiter au mieux les annotations, mais elle présente certaines limites. Il n'est pas possible de modifier une classe en train d'être compilée, lire le contenu d'une méthode ou accéder aux classes internes. Les seules possibilités sont l'ajout de classes ou de ressources. Pour modifier le comportement d'une classe, il faut alors utiliser le pattern « Décorateur ». Ce dernier consiste à proposer une classe de même interface et déléguer les traitements vers l'original. Pour que l'application construise la classe décorée, il faut utiliser un factory.

Déclarons une annotation permettant à une interface de générer automatiquement une implémentation du pattern décorateur.

```
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
```

```
@Target(ElementType.TYPE)
public @interface Decorator
{ }
```

Cette annotation n'est possible que sur une interface. Puis, déclarons une interface annotée.

```
@Decorator
public interface MonInterface
{
    void mamethode(int a) throws RuntimeException;
    String uneautremethode();
}
```

Nous souhaitons produire une classe implémentant cette interface, mais dont toutes les méthodes délèguent leurs traitements à une autre instance.

```
abstract class MonInterfaceDecorator implements MonInterface
{
    protected MonInterface _decorated;

    protected MonInterfaceDecorator(MonInterface decorated)
    { _decorated=decorated; }

    @Override
    public void mamethode(int a) throws RuntimeException
    {
        _decorated.mamethode(a);
    }

    @Override
    public String uneautremethode()
    {
        return _decorated.uneautremethode();
    }
}
```

Ainsi, pour décorer le comportement d'une seule méthode, il suffit d'hériter de la classe générée et de surcharger la méthode.

```
public class MonImplementationDecoree extends MonInterfaceDecorator
{

    public MonImplementationDecoree()
    {
        super(new MonImplementation());
    }

    @Override
    public void mamethode(int a) throws RuntimeException
    {
        // Before...
        super.mamethode(a);
        // After...
    }
}
```

```
}  
}
```

Un petit factory qui vérifie la présence d'une classe avec le suffixe `Decoree`, permet d'adapter l'application suivant la présence ou non des classes générées.

```
public static MonInterface factory()  
{  
    try  
    {  
        Class<?> cl=Class.forName(MonImplementation.class.getName()+"Decoree");  
        return (MonInterface)cl.newInstance();  
    }  
    catch (NoClassDefFoundError e)  
    {  
        return new MonImplementation(); // Par défaut  
    }  
    catch (Exception e)  
    {  
        return null;  
    }  
}
```

Pour obtenir cela, nous devons rédiger un processeur qui capture les annotations `@Decorator` pour générer la classe correspondante et le déclarer dans le répertoire `META-INF/services`.

```
@SupportedAnnotationTypes("processor.Decorator")  
@SupportedSourceVersion(SourceVersion.RELEASE_6)  
public class DecoratorProcessor extends AbstractProcessor  
{  
    public boolean process(final Set<? extends TypeElement> annotations,  
                           final RoundEnvironment renv)  
    {  
        final File file = processingEnv.getFile();  
        final Messenger messenger = processingEnv.getMessager();  
  
        if (!renv.processingOver())  
        {  
            try  
            {  
                Set<TypeElement> interfaces=  
                    ElementFilter.typesIn(renv.getElementsAnnotatedWith(Decorator.class));  
                for (TypeElement interf:interfaces)  
                {  
                    final PackageElement pack=(PackageElement)interf.getEnclosingElement();  
                    final Name qualifiedName=interf.getQualifiedName();  
                    final String qualifiedImpl=qualifiedName+"Decoree";  
                    final String impl=qualifiedImpl.substring(qualifiedImpl.lastIndexOf('.')+1);  
  
                    messenger.printMessage(Kind.NOTE, "Generate "+qualifiedImpl);  
                    JavaFileObject jfo=file.createSourceFile(qualifiedImpl);  
                    PrintWriter pw=new PrintWriter(jfo.openWriter());  
                    pw.println("package "+pack.getSimpleName()+";");  
                    pw.println("@javax.annotation.Generated({})");  
                    pw.println("abstract class "+impl+" implements "+qualifiedName);  
                    pw.println("{");  
                    pw.println("protected "+qualifiedName+" _decorated;");  
                    pw.println("protected "+impl+"("+qualifiedName+" decorated)");  
                    pw.println("{ _decorated=decorated; }");  
  
                    for (ExecutableElement m:ElementFilter.methodsIn(interf.getEnclosedElements()))  
                    {  
                        pw.println("@Override");  
                        pw.print("public "+m.getReturnType()+" ");  
                        pw.print(m.getSimpleName()+"(");  
                        int cnt=0;  
                        for (VariableElement p:m.getParameters())  
                        {  
                            pw.print(p.asType()+" "+p);  
                            ++cnt;  
                            if (cnt<m.getParameters().size())  
                                pw.print(',');  
                        }  
                        pw.print(")");  
                    }  
                }  
            }  
        }  
    }  
}
```

```

    if (m.getThrownTypes().size()>0)
    {
        pw.print(" throws ");
        cnt=0;
        for (TypeMirror e:m.getThrownTypes())
        {
            pw.print(e);
            if (++cnt<m.getThrownTypes().size())
                pw.print(',');
        }
        pw.println();
        pw.println("{}");
        if (!m.getReturnType().toString().equals("void"))
            pw.print("return ");
        pw.print("_decorated."+
            m.getSimpleName()+"");
        cnt=0;
        for (VariableElement p:m.getParameters())
        {
            pw.print(p);
            ++cnt;
            if (cnt<m.getParameters().size())
                pw.print(',');
        }
        pw.println(");");
        pw.println("}");
    }

    pw.println("}");
    pw.close();
}
}
catch (IOException ioe)
{
    messenger.printMessage(Kind.ERROR, ioe.getLocalizedMessage());
    ioe.printStackTrace();
}
}
return true;
}
}

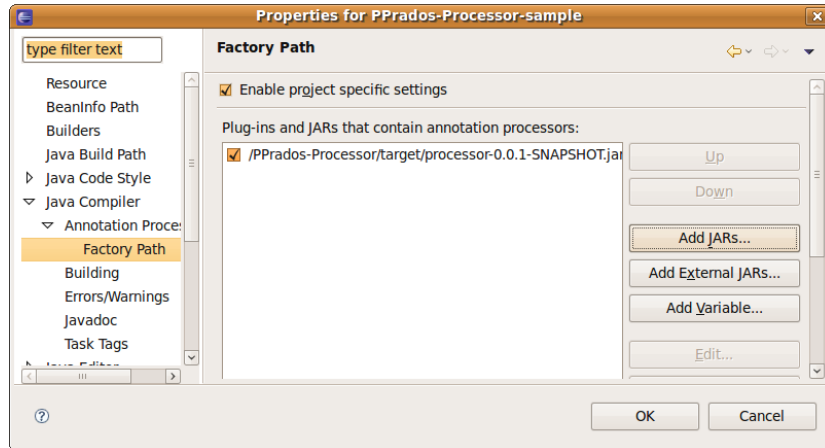
```

Si l'interface évolue, en ajoutant une nouvelle méthode par exemple, la classe décorée correspondante est immédiatement mise à jour. Ce ne serait pas le cas sans la génération automatique de classes.

Cette approche peut être utilisée pour ajouter de nouvelles interfaces à une classe, enrichir le code de pre- post conditions, ajouter le traitement de transactions ou vérifier les privilèges, etc. Elle est plus efficace que l'utilisation d'Auto-Proxy (proxy généré dynamiquement par la JVM pour répondre à des interfaces) car le code est compilé. Il n'y a pas d'utilisation de l'inspection. Mais elle oblige à utiliser un Factory pour construire les instances.

6 ECLIPSE

Eclipse 3.4 intègre les processeursⁱⁱⁱ, mais avec quelques limitations. Pour utiliser un processeur présent dans une archive, il faut l'indiquer dans le paramétrage du projet sous Properties/Java compiler/Annotation processing/Factory path. Les erreurs produites par le processeurs apparaissent alors comme des erreurs dans le source, avec une intégration parfaite dans Eclipse. Cela permet d'ajouter des processeurs permettant certaines validations dans les règles de codage.



Cette intégration dans Eclipse présente quelques faiblesses, dû à l'architecture même de l'environnement. En effet, une compilation peut intervenir à chaque caractère saisi. Si le processeur effectue trop de travail, l'expérience utilisateur est dégradée. Il faut éviter, pour les processeurs utilisés par Eclipse :

- D'itérer sur tous les types ou tous les fichiers ;
- D'utiliser des API exigeant d'autres compilations (`getPackage()` ou `getTypeDeclaration()`)
- D'effectuer des traitements longs lors de la compilation. Reportez les vers la phase de réconciliation ;
- D'utiliser dans des annotations des noms de classes plutôt que les classes elle-même (`@Ref(Toto.class)`). Préférez les types « dur » aux chaînes de caractères ;
- Évitez d'avoir plusieurs sources qui génèrent une seule cible ;
- Évitez les cycles dans la production de classes ;
- N'utilisez pas le type générique pour gérer les annotations (`@SupportedAnnotationTypes("*")`) ;
- Déclarez le processeur dans une archive différente et privée

7 AUTRES SOLUTIONS

D'autres approches permettent une prise en compte plus fine des annotations. Par exemple, le framework de programmation par aspect AspectJ permet l'injection de code, de méthode ou d'attribut et la modification de la structure d'une classe. Par contre, ce dernier n'est pas capable de générer des ressources comme des fichiers de paramètres. Une combinaison des deux approches est à prendre en compte pour intégrer toute les dimensions des annotations dans un programme Java.

Un prochain article évoquera l'utilisation d'AspectJ pour enrichir un programme Java.

Philippe Prados
 articles@philippe.prados.name
 Architecte chez Atos Origin

i <http://www.fusionsoft-online.com/articles-java-annotations.php>

ii <http://google.com/codesearch/p?hl=fr#h0n4vxU5-X4/glassfish/appserv-commons/src/java/com/sun/enterprise/deployment/annotation/introspection/ClassFile.java>

iii <http://www.eclipse.org/jdt/apt/introToAPT.html>