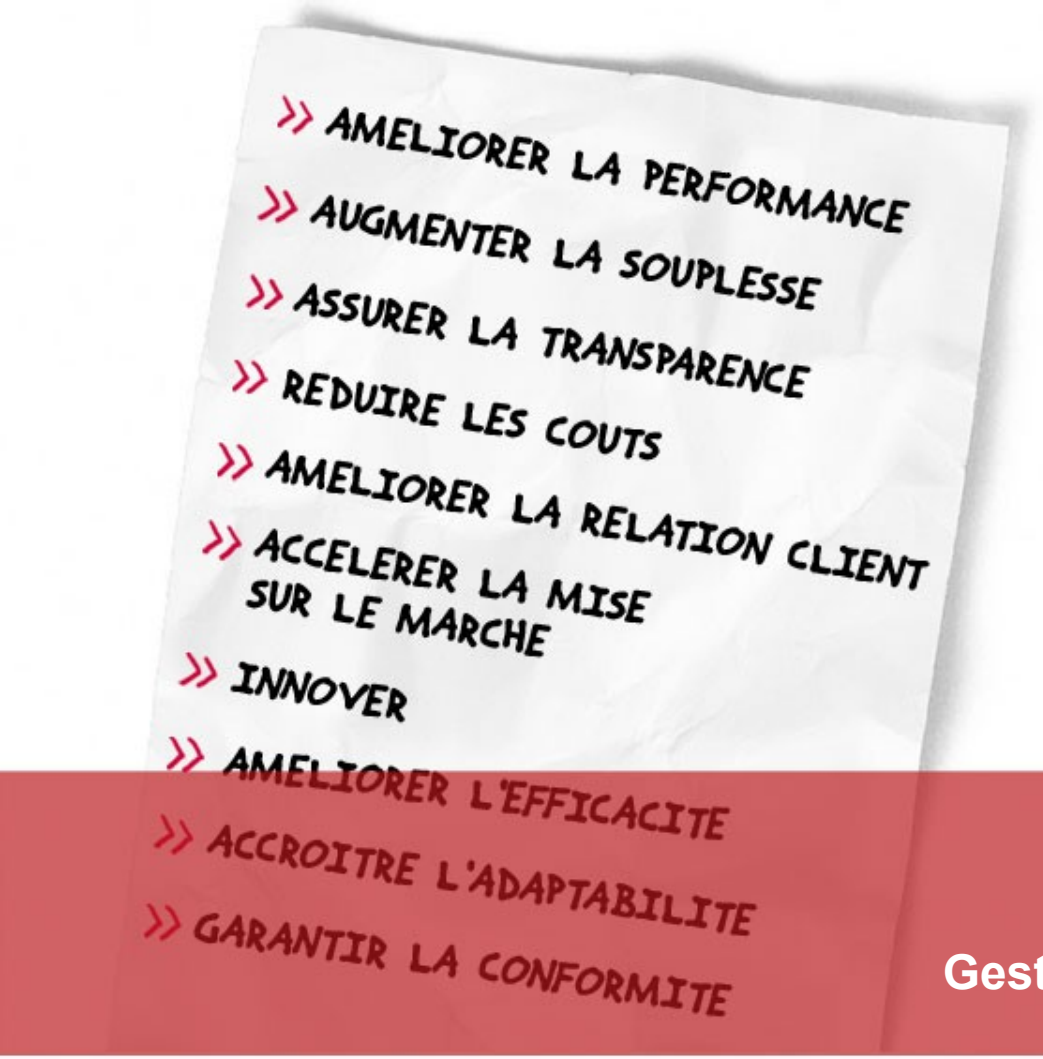


Atos Origin™

CONSULTING > SOLUTIONS > OUTSOURCING

JSR269

Gestion des annotations à la compilation

Antonio Goncalves

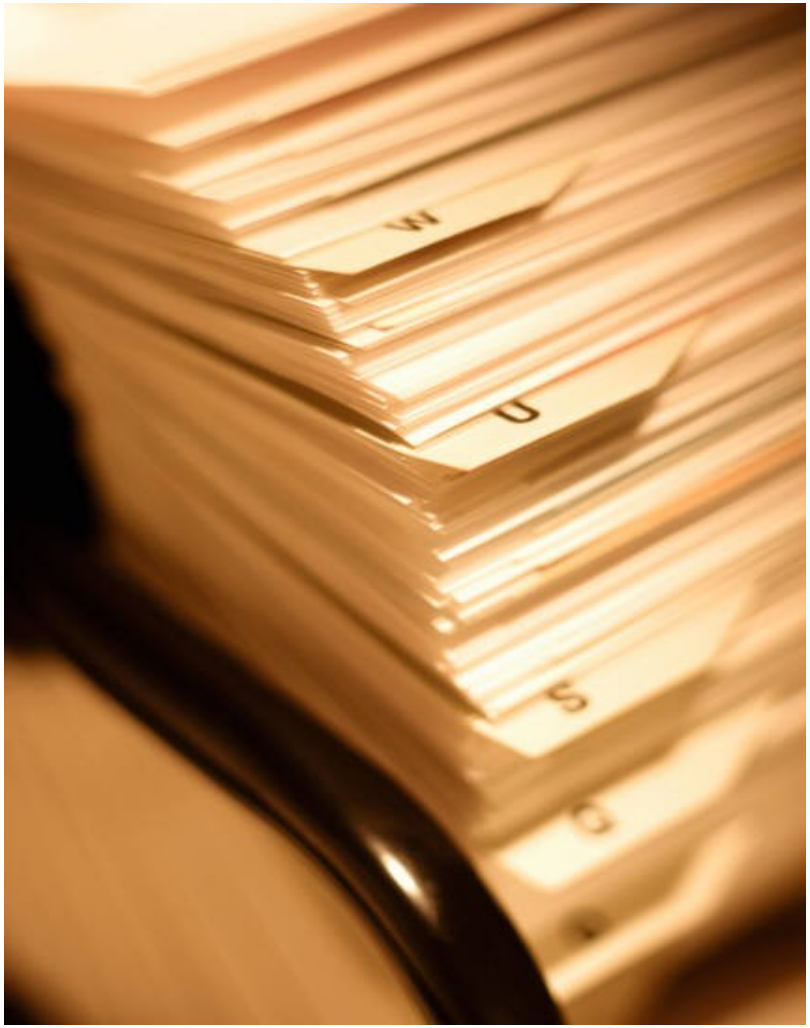
JUG – Juin 2009

Curriculum Vitae

- Philippe Prados
- Architecte Java, Consultant

Sommaire

- Rappel sur les annotations
- Les annotations lors de la compilation
- Génération d'une ressource
- Génération d'une classe
- Utilisation avec Eclipse
- Synthèse



Les annotations

- Informations complémentaires aux éléments structurants du langage
  - Classe, Interface, attribut, méthode, paramètre, variable locale
- Informations statiques (calculable à la compilation) et potentiellement riches (grappe d'objets)
- Déclaration d'une annotation à partir d'une interface
- Existence de meta-annotations (annotations d'annotations)

Exemple

```
// Déclaration d'une annotation
public @interface MonAnnotation
{
  int id();
  Class cl();
  String value() default "hello";
}

// Exemple de valorisation
@MonAnnotation("Bonjour")
@MonAnnotation(id=123,cl=String.class,value="hey")


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

Meta-Annotations

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

Utilisation au runtime

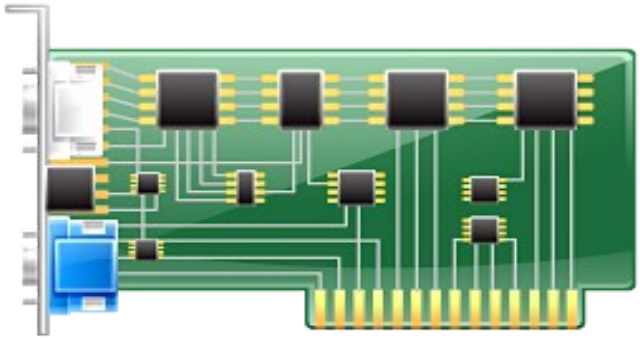

```
public interface CouleurPreferee
{
  String getCouleur();
}

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
{
}
```

JSR269 - @Retention(RetentionPolicy.SOURCE)

- Un framework de JDK6+ permettant, en autre, de gérer les annotations visibles uniquement dans les sources
- Des annotations absentes de l'introspection et de l'exécution
- Framework permettant d'intervenir lors de la compilation des classes annotées
- Pour Geek

Un « Processeur »

- Une classe annotée, indiquant les annotations et la version du JDK des classes gérées
- Invoqué par le compilateur après l'analyse syntaxique des classes
- Permet d'introspecter les classes en cours de compilation (API spécifique)
- Un processeur est une classe présente dans une archive, déclarée dans un fichier spécial :
  `META-INF/services/javax.annotation.processing.Processor`
- Le fichier possède une ligne de texte en UTF-8 avec le nom de la classe du processeur



AbstractProcessor – Generation de fichier

```
@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 Filer filer = processingEnv.getFiler();
    final Messager messager = processingEnv.getMessager();
    if (!env.processingOver())
    { try
      {
        FileObject jfo=filer.createResource(StandardLocation.CLASS_OUTPUT, "",
                                            "web.xml",(Element)null);
        messager.printMessage(Kind.NOTE, "Writing web.xml");
        PrintWriter pw = new PrintWriter(jfo.openOutputStream());
        pw.write(...
        pw.close();
      }
      catch (IOException ioe)
      {
        messager.printMessage(Kind.ERROR, ioe.getLocalizedMessage());
      }
    }
    return true;
  }
}
```

Abstract Processor – Génération de classe

» Objectif : générer un wrapper ad-hoc à chaque évolution d'une interface

```
// L'annotation Decorator
@Target(ElementType.TYPE)
public @interface Decorator
{ }

// L'utilisation
@Decorator
public interface MonInterface
{
  void maMethode(int a) throws RuntimeException;
  String uneAutreMethode();
}

// Exemple de wrapper à générer
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();
  }
}
```

Utilisation d'un @Decorator

```
public class MonImplementationDecoree
extends MonInterfaceDecorator
{
  public MonImplementationDecoree()
  {
    super(new MonImplementation());
  }

  @Override
  public void maMethode(int a) throws RuntimeException
  {
    // Before...
    super.mamethode(a);
    // After...
  }
  // Pas d'autres méthodes à déclarer,
  // même en cas d'évolution de l'interface
}
```

Génération d'un fichier source

```
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);
     messager.printMessage(Kind.NOTE, "Generate "+qualifiedImpl);
     JavaFileObject jfo=filer.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("{");
     …
     for (ExecutableElement m:ElementFilter.methodsIn(interf.getEnclosedElements()))
     {
       ...
```

» La classe est ensuite compilée et peut faire partie d'un autre cycle de génération !

Cas d'usage

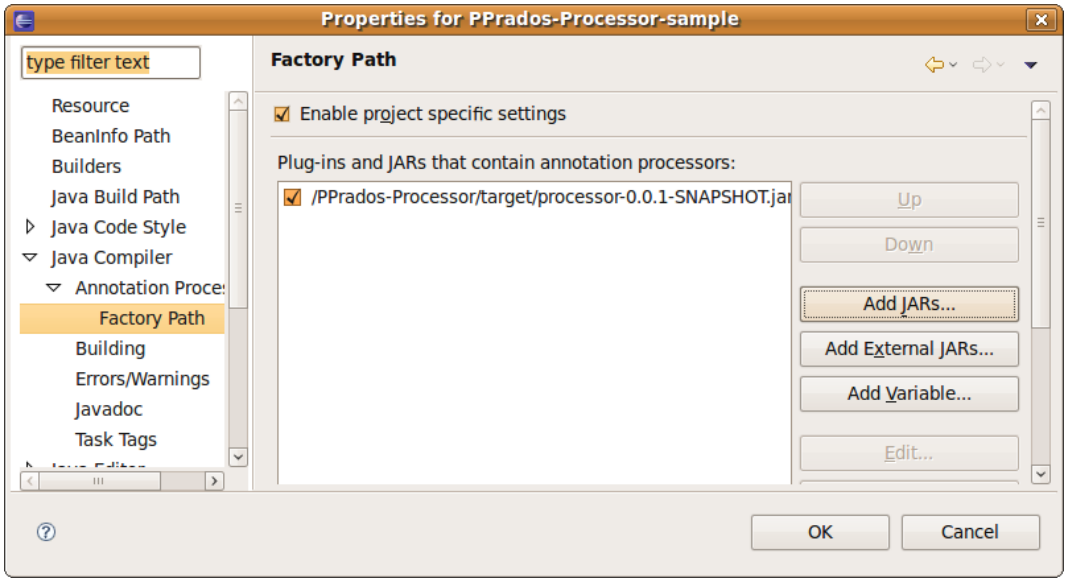
- Cette approche peut être utilisée pour ajouter de nouvelles interfaces à une classe, enrichir le code de pre- post conditions, ajouter le traitement de transactions, vérifier les privilèges, etc.
- Elle est plus efficace que l'utilisation d'Auto-Proxy (proxy généré dynamiquement par la JVM pour implémenter des interfaces) car le code est compilé.
- Il n'y a pas d'utilisation de l'introspection à l'exécution.

Attention lors de la compilation du Processeur

- Comme le fichier `META-INF/services/javax.annotation.processing.Processor` est présent, le compilateur cherche à l'appliquer lors de sa propre compilation !
- La première génération est donc difficile. Les suivantes utilisent la précédente.
- Solution :
  - Deux projets dont l'un pour le processeur uniquement
  - Paramètre `-proc:none` pour interdire l'utilisation de processeur lors de la compilation de ce projet

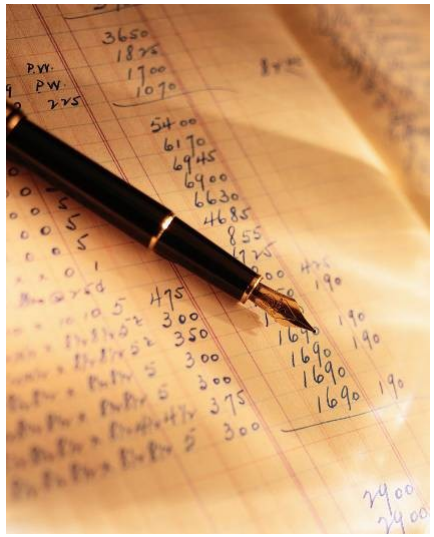
Intégration à Eclipse

- Eclipse propose une intégration « compatible » mais pas identique à l'utilisation d'un simple compilateur Java.
- Chaque sauvegarde d'un fichier peut entrainer la génération d'autres classes et la compilation de tous les impacts
- Menu Properties/Java compiler/Annotation processing/Factory path



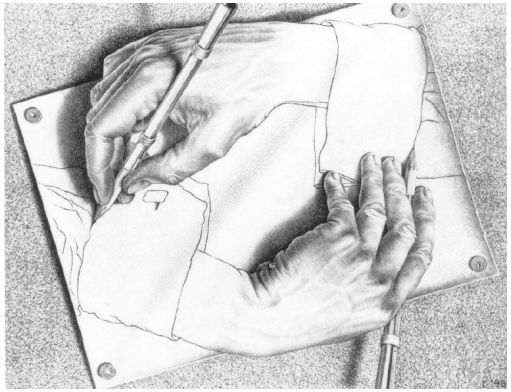
Bonne pratique avec Eclipse

- 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 ;
  - D'utiliser dans des annotations des noms de classes plutôt que les classes elle-même (@Ref(Toto.class)).Préférez les types « dur » aux chaines 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("*")) ;



Autre approches

- Pour une prise en compte plus fine des annotations.
  - La programmation par aspect via AspectJ
    - permet l'injection de code, de méthode ou d'attribut et la modification de la structure d'une classe.
    - Mais, 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 programma Java.
    - JSR269 pour générer les ressources, des classes annotées ou des aspects
    - AspectJ pour injecter du code



Sponsors


Xebia


zenika
ARCHITECTURE INFORMATIQUE


OBJET
DIRECT
homsys


NOVEDIA
VISION


SFEIR

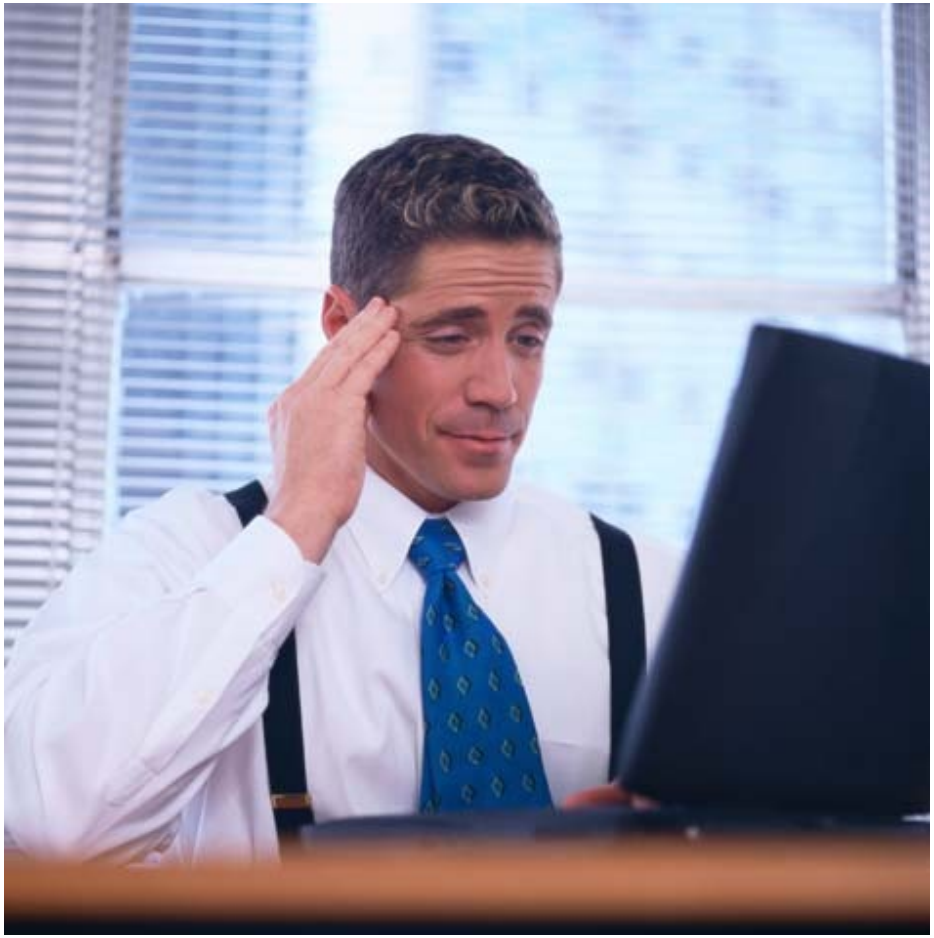

OXiane


Fast Connect


valtech


BK Consulting

Questions ?



Résumé

- Sommaire
- Les annotations
- Exemple
- Meta-Annotations
- Utilisation au runtime
- JSR268 - @Retention(RetentionPolicy.SOURCE)
- Un « Processeur »
- AbstractProcessor – Generation fichier
- Abstract Processor – Génération de classe
- Utilisation d'un @Decorator
- Génération d'un fichier source
- Cas d'usage
- Attention lors de la compilation du Processeur
- Intégration à Eclipse
- Bonne pratique avec Eclipse
- Autre approches
- Questions ?

Atos Origin™

CONSULTING > SOLUTIONS > OUTSOURCING

Pour plus d'informations, contacter :

Philippe PRADOS
+33 (0)6 70 03 89 60
philippe.prados@atosorigin.com

Atos Origin
18 avenue d'Alsace
92926, Paris la Défense Cedex
www.atosorigin.fr

ADVANCE YOUR BUSINESS »