APT in a nutshell

@gdigugli

@dbaeli

Speakers

@dbaeli

- Chien de berger Agile pour

- Développeur de plus de 30 ans
- Usine logicielles & DevOps
- Qualité Logicielle

@gdigugli

- Développeur java depuis 1999
- Architecte pour

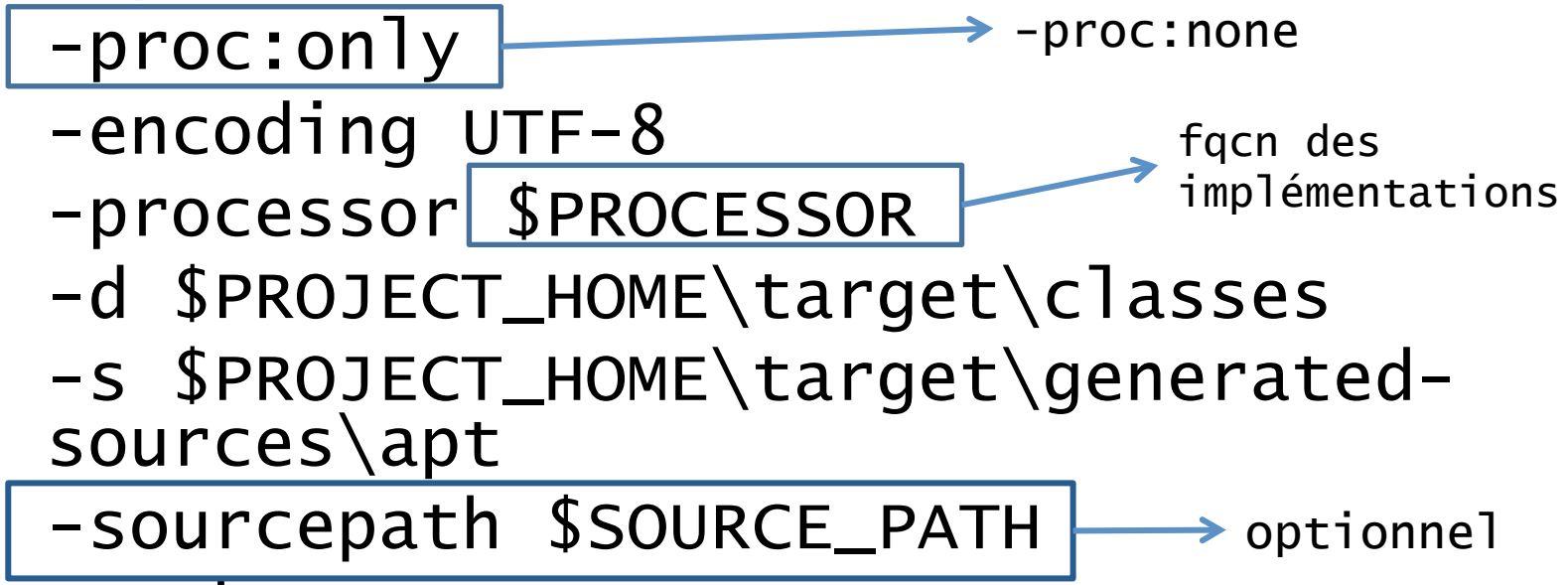
- ILOG - IBM
  - ✓ librairie graphique 2D
  - ✓ moteur de règles
- Prima-Solutions
  - ✓ plate-forme de services pour J2EE
  - ✓ code génération de modèle métier

APT : qu'est-ce que c'est?

- Annotation Processing Tool
- Souvenez vous la commande CPP et les #define
- Les processeurs s'intercalent entre la lecture des fichiers sources et le générateur de bytecode
- Approche de plugins basée sur java.util.ServiceLoader (META-INF/services)
- Pas de framework de templating pour générer des fichiers code source java

Appeler APT depuis la ligne de commande javac

```
javac
  -cp $CLASSPATH
  -proc:only
  -encoding UTF-8
  -processor $PROCESSOR
  -d $PROJECT_HOME\target\classes
  -s $PROJECT_HOME\target\generated-
  sources\apt
  -sourcepath $SOURCE_PATH
  -verbose
  $FILES
```



Appeler APT depuis maven

- Le plugin maven-compiler avec des options passées « à la main »
- Le plugin org.bsc.maven:maven-processor-plugin (google code)

```
- <execution>
    <id>generate-i18n-source</id>
  - <goals>
      <goal>process</goal>
    </goals>
    <phase>generate-sources</phase>
  - <configuration>
      <compilerArguments>-encoding UTF-8</compilerArguments>
      <outputDirectory>${project.build.directory}/generated-sources/apt</outputDirectory>
    - <processors>
        <processor>org.ez18n.apt.processor.SiteBundleProcessor</processor>
        <processor>org.ez18n.apt.processor.SiteBundlePropertiesProcessor</processor>
        <processor>org.ez18n.apt.processor.CSVReportProcessor</processor>
      </processors>
    </configuration>
  </execution>
```

L'API javax.annotation.processing

Interface Processor et classe AbstractProcessor

Method Summary	
Iterable<? extends Completion>	getCompletions(Element element, AnnotationMirror annotation, ExecutableElement member, String userText) Returns to the tool infrastructure an iterable of suggested completions to an annotation.
Set<String>	getSupportedAnnotationTypes() Returns the names of the annotation types supported by this processor.
Set<String>	getSupportedOptions() Returns the options recognized by this processor.
SourceVersion	getSupportedSourceVersion() Returns the latest source version supported by this annotation processor.
void	init(ProcessingEnvironment processingEnv) Initializes the processor with the processing environment.
boolean	process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) Processes a set of annotation types on type elements originating from the prior round and returns whether or not these annotations are claimed by this processor.

Exemple de processor

- @SupportedAnnotationTypes pour déclarer l'annotation cible

```
@SupportedAnnotationTypes(value = "org.ez18n.apt.LabelBundle")
@SupportedSourceVersion(RELEASE_6)
public final class CSVReportProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
```

- Créer un FileObject pour écrire du contenu

```
try {
    final FileObject file = processingEnv.getFiler().createResource(CLASS_OUTPUT, "", "i18n_report.csv");
    final Writer writer = file.openWriter();



    writer.close();
} catch (FilerException e) {
    return false;
} catch (IOException e) {
    processingEnv.getMessager().printMessage(Kind.ERROR, e.getMessage());
}
```

Meta modèle d'un fichier source java

Subinterfaces of Element in javax.lang.model.element	
interface	ExecutableElement Represents a method, constructor, or initializer (static or instance) of a class or interface, including annotation type elements.
interface	PackageElement Represents a package program element.
interface	TypeElement Represents a class or interface program element.
interface	TypeParameterElement Represents a formal type parameter of a generic class, interface, method, or constructor element.
interface	VariableElement Represents a field, `enum` constant, method or constructor parameter, local variable, or exception parameter.

Quelques sous type d'Element que l'on caste en fonction de Element.getKind()

Méta modèle d'un fichier source

Enum Constant Summary

ANNOTATION_TYPE	An annotation type.
CLASS	A class not described by a more specific kind (like ENUM).
CONSTRUCTOR	A constructor.
ENUM	An enum type.
ENUM_CONSTANT	An enum constant.
EXCEPTION_PARAMETER	A parameter of an exception handler.
FIELD	A field not described by a
INSTANCE_INIT	An instance initializer.
INTERFACE	An interface not describe
LOCAL_VARIABLE	A local variable.
METHOD	A method.
OTHER	An implementation-reser
PACKAGE	A package.
PARAMETER	A parameter of a method or constructor.
STATIC_INIT	A static initializer.
TYPE_PARAMETER	A type parameter.

```
for (Element element : roundEnv.getElementsAnnotatedWith(LabelBundle.class)) {
    if (element.getKind() != ElementKind.INTERFACE)
        continue;
    final TypeElement bundleType = (TypeElement) element;
    for (Element enclosedElement : bundleType.getEnclosedElements()) {
        if (enclosedElement.getKind() != ElementKind.METHOD)
            continue;
        final ExecutableElement method = (ExecutableElement) enclosedElement;
        Label labelAnnotation = enclosedElement.getAnnotation(Label.class);
```

Comparaison avec java.lang.reflect

Java.lang.reflect	Javax.annotation.processing
java.lang.Class	TypeElement
Constructor	ExecutableElement
Field, Parameter	VariableElement
Method	ExecutableElement
java.lang.Package	PackageElement

- NO Class.newInstance()
- NO instanceOf, NO isAssignable()
- NO getConstructor, getMethod, …
- Impossible de tester l'arbre d'héritage lorsqu'on navigue dans le source d'une classe

Ca sert à quoi ?

- Injecter des patterns répétitifs et complexes
- Générer des factories, des singletons
- Générer des délégations avec du code de management
  - Assertions
  - Sonde JMX
  - Transaction
  - Gestion des data sources
- Générer des rapports sur du code
  - Tables de références
  - Requêtes SQL embarquées dans le code java

Pattern avec injection – je fais un framework

- Annoter des beans ou des interfaces
  - Générer les implémentations ou les proxy
  - Injecter les implémentations
  - Générer les descripteurs pour le framework d'injection ou utiliser un classpath scanneur
- Le code client ne doit pas avoir de références sur le code généré
- Possibilité de mixer le code généré avec des annotation interprétées à runtime
  - ➢ Utiliser les conventions de nommage des classes cibles et la réflection

Analyse et transformation de code vers des fichiers plats

- Générer des fichiers pour le tableurs
  - Cartographie des @Deprecated dans une base de code volumineuse
  - Données métiers pour de la documentation : les constantes de toutes les énumérations d'un modèle métier
- Générer des fichiers properties
  - Configuration de l'application pour le staging (développement/recette/production)

DSL avec des annotations

- Ajouter des annotations qui paramètrent un pattern complexe
  - Par exemple: le pattern pour des MBeans en ajoutant des méthodes de moyenne ou de dérivée en fonction du temps sur les sondes primitives
- Le code client référence directement le code généré
  - Le code généré ne peut pas être utilisé dans le même module
  - requière une modularisation soigneuse

No limit …

- Tentation de générer trop de patterns
- Un plugin APT est difficile à maintenir
- Les tests unitaires sont complexes à écrire
- Chronophage à debugger
- Parfois plus de complexité dans un processeur APT que de maintenir un pattern à la main
- Peu de références ou de support
- Outillage inexistant pour le templating du code à générer
  - Penser à utiliser freemarker ou velocity

Compilation : une ou deux passes ?

- Une passe
  - Le plugin APT s'exécute dans la même exécution que la compilation des sources java 'statiques'
  - Le code source apparait directement sous forme de bytecode (.class)
  - Difficile à debugger en cas de soucis
- Deux passes
  - On exécute javac en mode proc:only
  - Puis une deuxième fois en mode proc:none avec les sources générées dans le sourcepath
  - Le debug devient possible sur le code généré

Templating

- A votre guise !
- Respecter l'indentation classique java si vous travaillez en 'deux passes'
- Un petit moteur de template tient dans une classe
  - Moteur de macro de apache ant

```
package ${package.name};

import javax.annotation.Generated;
import com.google.gwt.i18n.client.LocalizableResource.Generate;
import com.google.gwt.i18n.client.Messages;

@Generated(value = "${process.class}",  date = "${process.date}")
@Generate(format = "com.google.gwt.i18n.rebind.format.PropertiesFormat")
public interface ${target.class.name} extends Messages, ${source.class.name} {
${methods.code}
}
```

```
final Map<String, String> conf = new HashMap<String, String>();
conf.put("process.class", getClass().getName());
conf.put("process.date", DateFormat.getDateTimeInstance(SHORT, SHORT).format(new Date()));
conf.put("source.class.name.camel", toCamelCase(bundleType));
conf.put("target.class.name", getTargetSimpleName(bundleType));
conf.put("source.class.name", bundleType.getSimpleName().toString());
```

APT dans mon IDE

- La plupart des IDE sont configurés avec javac en mode proc:none
  - Les IDE sont souvent configurable pour activer le mode de compilation ‘une passe’
  - Aucun support pour le mode ‘deux passes’

- En mode ‘deux passes’ : penser à configurer le répertoire de sources générées dans les projets de l’IDE

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>add-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>target/generated-sources/apt</source>
          <source>target/generated-sources/apt-test</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```