

Introduction au développement d'un processeur d'annotations en Java

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

1 Sample project	1
2 Qu'est-ce qu'un processeur d'annotations ?	1
3 Pourquoi utiliser les processeurs d'annotations ?	2
4 Le cycle de vie d'un processeur d'annotations	2
5 Le framework APT (Annotation Processing Tool)	2
6 Création d'une annotation personnalisée .	2
7 Implémentation d'un processeur simple . .	2
8 Génération de code de base	3
8.1 Utilisation des éléments de langage Java	3
8.2 Génération de code complexe	3
9 Cas d'utilisation concrets	3
10 Bonnes pratiques et pièges à éviter	3

1 Sample project

<https://github.com/ebro/sample-annotationprocessor>

2 Qu'est-ce qu'un processeur d'annotations ?

- **Définition simple:** Un processeur d'annotations est un outil qui analyse le code source à la recherche d'annotations spécifiques.

- **Fonctionnement:** Il intercepte le processus de compilation pour exécuter du code personnalisé en fonction des annotations trouvées.
- **Analogie:** C'est comme un « traducteur » qui transforme les annotations en instructions compréhensibles par la machine.
- **Exemple:** L'annotation `@Override` indique que une méthode redéfinit une méthode héritée. Le compilateur utilise cette information pour vérifier la cohérence. Qu'est-ce qu'un processeur d'annotations ?
- **Définition simple:** Un processeur d'annotations est un outil qui analyse le code source à la recherche d'annotations spécifiques.
- **Fonctionnement:** Il intercepte le processus de compilation pour exécuter du code personnalisé en fonction des annotations trouvées.
- **Analogie:** C'est comme un « traducteur » qui transforme les annotations en instructions compréhensibles par la machine.
- **Exemple:** L'annotation `@Override` indique que une méthode redéfinit une méthode héritée. Le compilateur utilise cette information pour vérifier la cohérence.

3 Pourquoi utiliser les processeurs d'annotations ?

- **Génération de code:** Automatisation de tâches répétitives (getters, setters, builders, etc.).
- **Validation de données:** Vérification de contraintes à la compilation (non-nullité, plages de valeurs, etc.).
- **Configuration:** Paramétrage de composants à partir d'annotations.
- **Documentation:** Génération automatique de documentation à partir d'annotations.
- **Intégration avec d'autres outils:** Liaison avec des outils de build, de test, etc.

4 Le cycle de vie d'un processeur d'annotations

1. **Détection des annotations:** Le compilateur identifie les éléments annotés.
2. **Invocation du processeur:** Le processeur est appelé pour chaque élément annoté.
3. **Analyse des annotations:** Le processeur extrait les informations contenues dans les annotations.
4. **Génération de code (optionnel):** Le processeur peut générer de nouvelles classes, méthodes ou fichiers.
5. **Intégration dans le résultat de la compilation:** Le code généré est intégré au résultat final.

5 Le framework APT (Annotation Processing Tool)

- **Rôle:** Fournit l'infrastructure de base pour développer des processeurs d'annotations.
- **Interfaces clés:**
 - **Processor:** Point d'entrée du processeur.
 - **RoundEnvironment:** Contient les éléments annotés lors d'un round de traitement.
 - **AnnotationMirror:** Représente une annotation.

- **Cycle de vie:**

- Initialisation
- Traitement des annotations
- Génération de code
- Nettoyage

6 Création d'une annotation personnalisée

- **Syntaxe:**

```
///  
import java.lang.annotation.Target;  
import  
java.lang.annotation.ElementType;  
import java.lang.annotation.RetentionPolicy;  
import  
java.lang.annotation.Retention;
```

```
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.SOURCE)  
public @interface Log {  
    String value() default "";  
}
```

- **Éléments:**

- **@Target:** Indique où l'annotation peut être utilisée (méthodes, classes, etc.).
- **@Retention:** Détermine la durée de vie de l'annotation.
- **value:** Attribut de l'annotation.

7 Implémentation d'un processeur simple

```
public class LogProcessor implements  
Processor {  
    // ...  
    @Override  
    public boolean process(Set<  
extends TypeElement> annotations,  
RoundEnvironment roundEnv) {  
        for (Element element : roundEnv.getElementsAnnotate
```

```

{
    // Traitement de l'élément
    annoté (cf. ci-dessous)
}
return true;
}
}

```

8 Génération de code de base

```

// Création d'un fichier source Java
JavaFileObject sourceFile = processingEnv.getSourceFile("GeneratedLogger");
try (Writer writer =
sourceFile.openWriter()) {
    writer.write("public class
GeneratedLogger {\n");
    // ...
    writer.write("}");
}

```

8.1 Utilisation des éléments de langage Java

- **Types génériques:**
 - Créer des processeurs qui fonctionnent avec des types génériques pour une plus grande flexibilité.
 - Exemple : Un processeur qui génère des méthodes `equals()` et `hashCode()` pour n'importe quelle classe.
- **Annotations:**
 - Utiliser des méta-annotations pour créer des hiérarchies d'annotations.
 - Combiner plusieurs annotations pour exprimer des contraintes complexes.
 - Exemple : Créer une annotation `@Validated` qui combine des annotations de validation comme `@NotNull`, `@Size`, etc.
- **Lambdas:**
 - Utiliser des lambdas pour écrire du code plus concis et fonctionnel.

8.2 Génération de code complexe

- **Création de structures de données:**
 - Générer des classes, des interfaces, des énumérations, etc.
 - Créer des hiérarchies de classes complexes.
 - Exemple : Générer un DAO (Data Access Object)
- **Arbres abstraits syntaxiques (AST):**
 - Manipuler le code source à un niveau plus bas.
 - Transformer le code en fonction de règles spécifiques.
- **Templates de code:**
 - Utiliser des templates pour générer du code.

9 Cas d'utilisation concrets

- **Validation de données:** Vérifier la cohérence des données avant leur utilisation.
- **Génération de code pour des frameworks:** ORM, REST, etc.
- **Configuration de projets:** Charger des propriétés à partir d'annotations.
- **Documentation:** Générer des rapports, des diagrammes, etc.

10 Bonnes pratiques et pièges à éviter

- **Modularité:** Séparer les différentes responsabilités du processeur.
- **Testabilité:** Écrire des tests unitaires pour vérifier le comportement du processeur.
- **Documentation:** Documenter clairement le processeur et ses fonctionnalités.
- **Éviter les régressions infinies:** Assurez-vous que le processeur ne génère pas de code qui déclenche à nouveau le traitement.