Introduction à Spoon

HAI913I – Evolution & Restructuration

Bachar Rima

Université de Montpellier

19 octobre 2021

Sommaire

Introduction

Definition (Spoon)

Une librairie Java *open-source* pour :

- L'analyse et la transformation de code source Java ;
- L'analyse de code source Java décompilé (à partir de bytecode) ;
- La transpilation (e.g., `Java` $\rightarrow$ `JavaScript`) ;
- ...

Installation (Maven Plugin)

```
<dependencies>
    <dependency>
        <groupId>fr.inria.gforge.spoon</groupId>
        <artifactId>spoon-core</artifactId>
        <version>8.0.0</version>
    </dependency>
</dependencies>
```

Workflow
(1)

1. Créer un **launcher**.
2. Configurer le **parseur** du launcher : *e.g., chemin du code source, chemin de la JRE, chemin du code cible, chemin du code cible bytecode, l'auto-importation, les commentaires, ...*
3. Définir des **processeurs** (*i.e., des visiteurs d'AST de noeuds spécifiques*) qui peuvent utiliser des **mécanismes Spoon** divers afin d'extraire des propriétés du **modèle Spoon** (*un AST*) : *e.g., filtres, queries, scanners, itérateurs, paths, patterns, templates, et générateurs*.
4. Ajouter les processeurs au launcher.
5. Construire le modèle en utilisant le launcher.
6. Lancer le launcher afin d'appliquer les processeurs sur le modèle construit (et éventuellement générer du code cible transformé).

Workflow
(2)

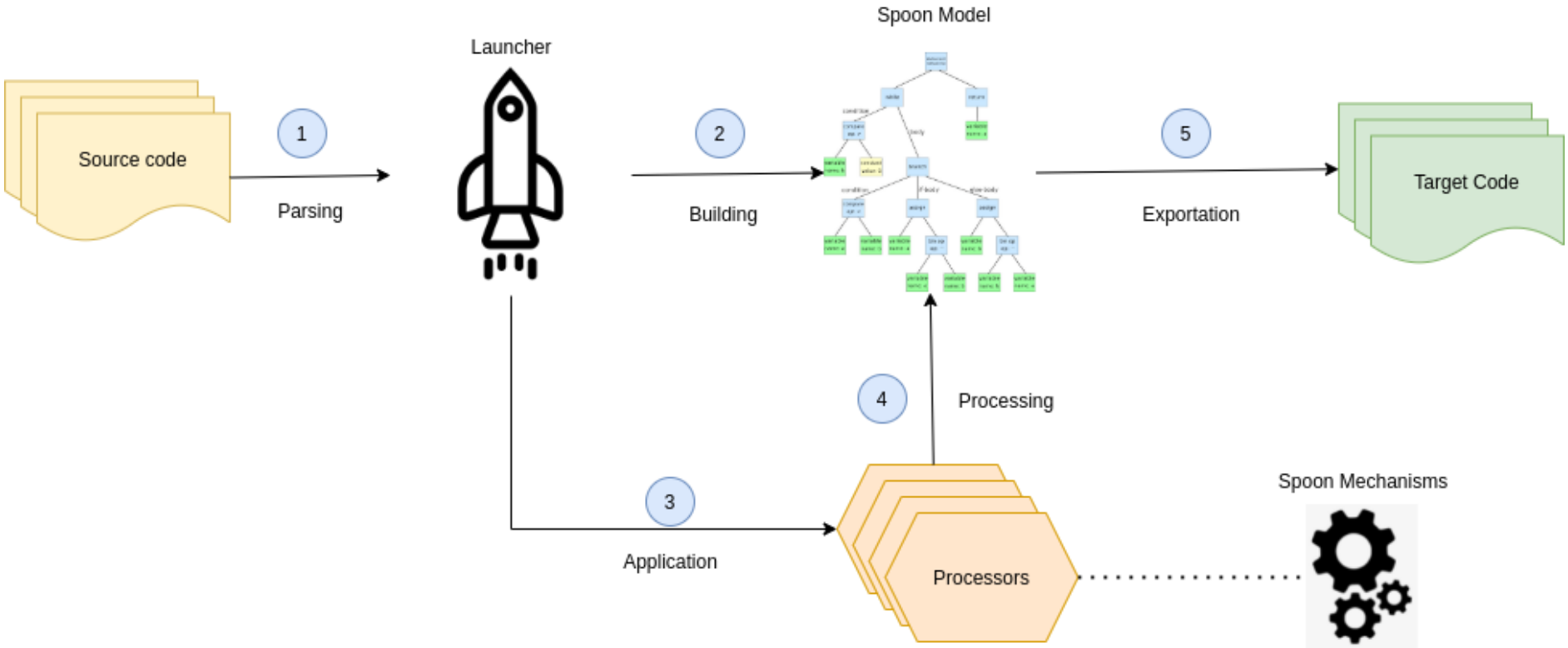


Figure – Workflow typique d'un projet Spoon

Le Métamodèle de Spoon

Idée

- Spoon fournit un métamodèle Java à grain fin permettant d'accéder à n'importe quel élément en lecture/écriture.
- Tous les éléments du métamodèle sont modélisés par des interfaces dont les noms commencent par `Ct` (*Compile-time*) dans une hiérarchie d'héritage multiple.
- Les viewpoints du métamodèle :

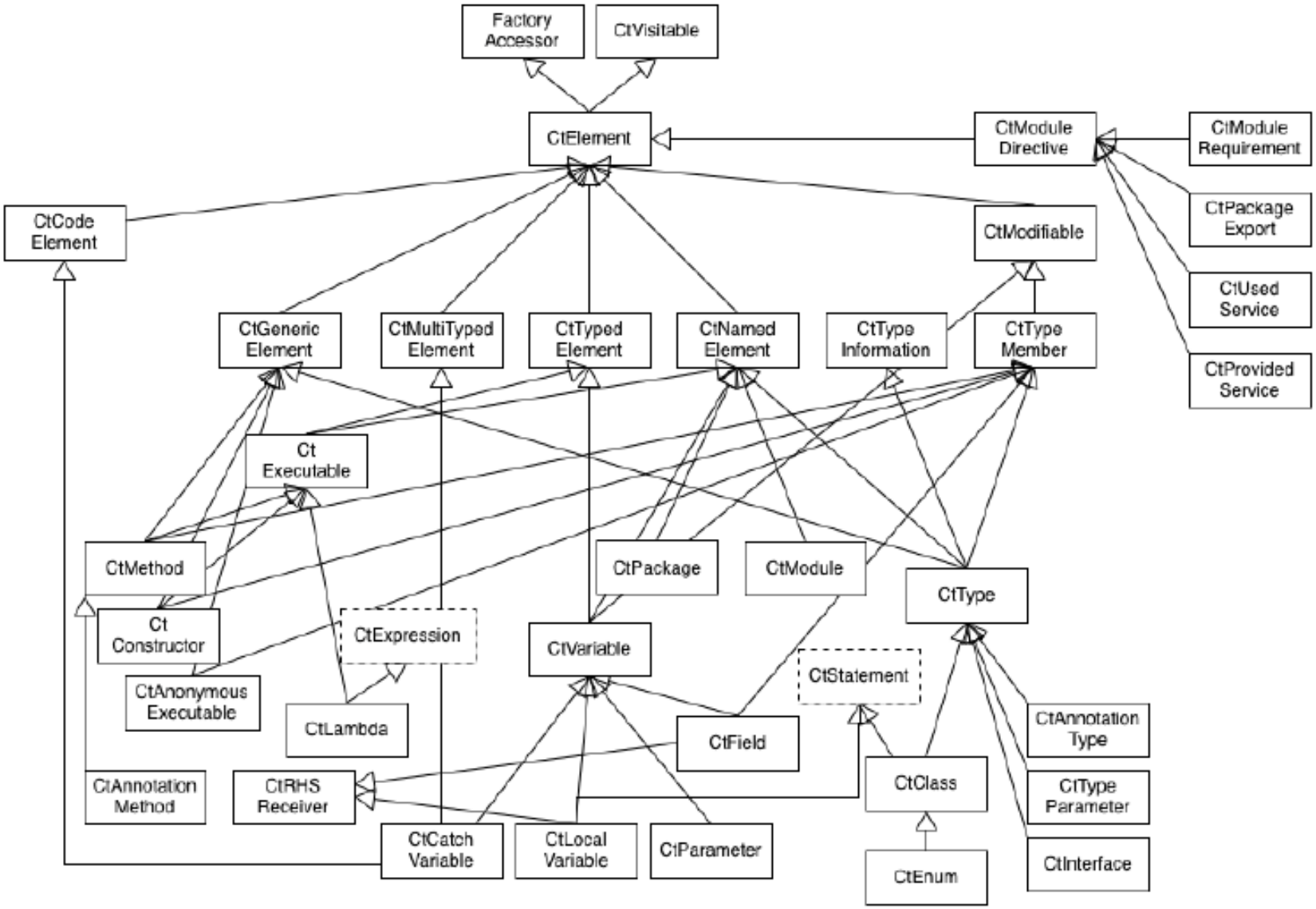
  éléments structurels : les éléments de déclarations d'un programme (*e.g. classes*, *interfaces*, *énumérations*, *variables*, *méthodes*, *variables*, ...)

  éléments exécutables : les éléments exécutables de Java (*e.g. les corps de méthodes/constructeurs*, *invocations de méthodes/constructeurs*, *les statements*, *les expressions*, ...)

  éléments de référence : des éléments désignant des références de type.

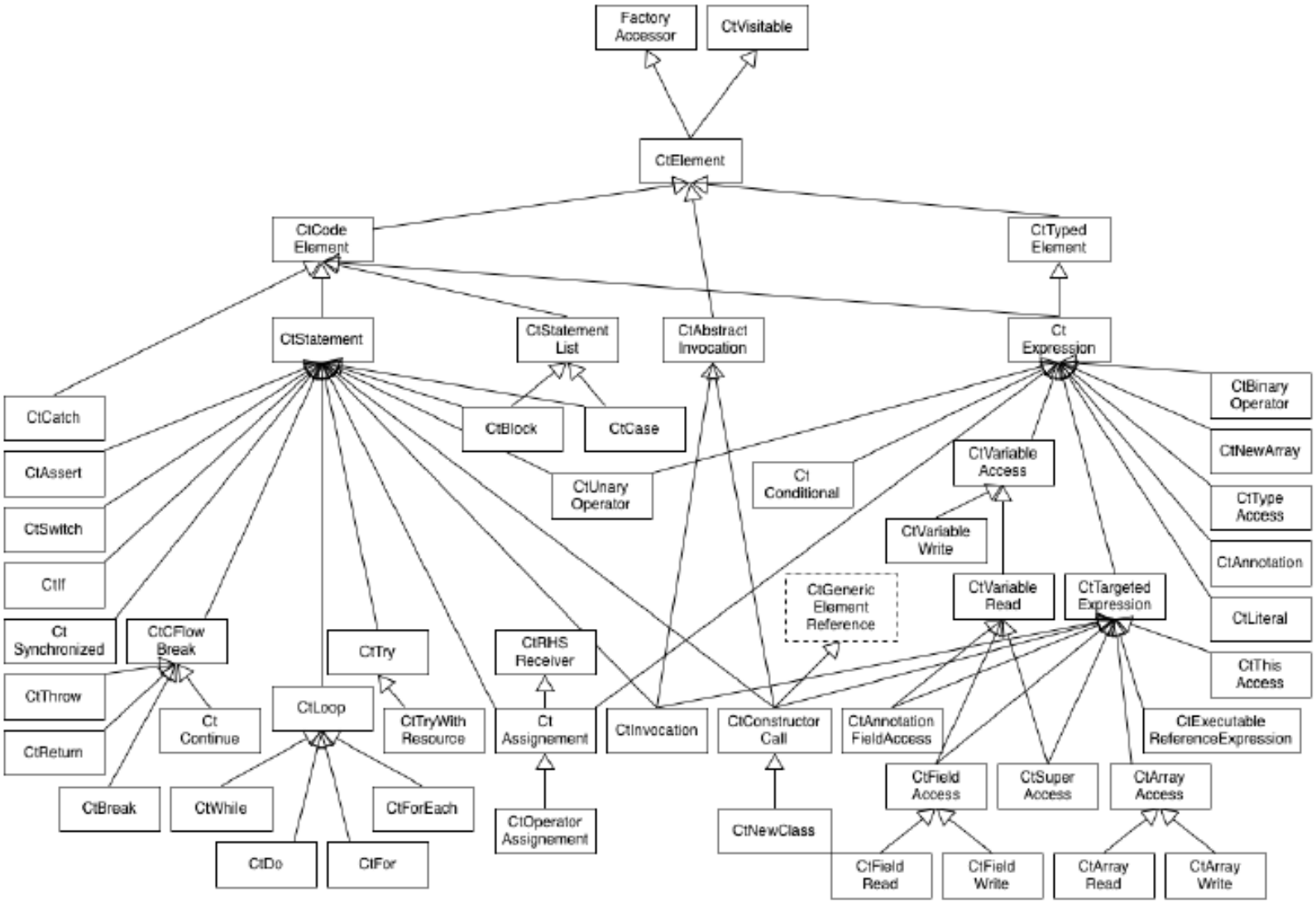
Le Métamodèle de Spoon

Viewpoint Structurel



Le Métamodèle de Spoon

Viewpoint Executable



Le Métamodèle de Spoon

Viewpoint Références (1)

définition : référencer des éléments qui ne sont pas forcément réifiés au sein du métamodèle, (*e.g., éléments de bibliothèques tierces*).

motivation : flexibilité lors de la construction/modification du modèle du programme en cours d'analyse.

exemple : référencer un objet de type **String** ne désigne pas le modèle compilable de `String.java`.

référence d'une cible : `CtType#getType()`.

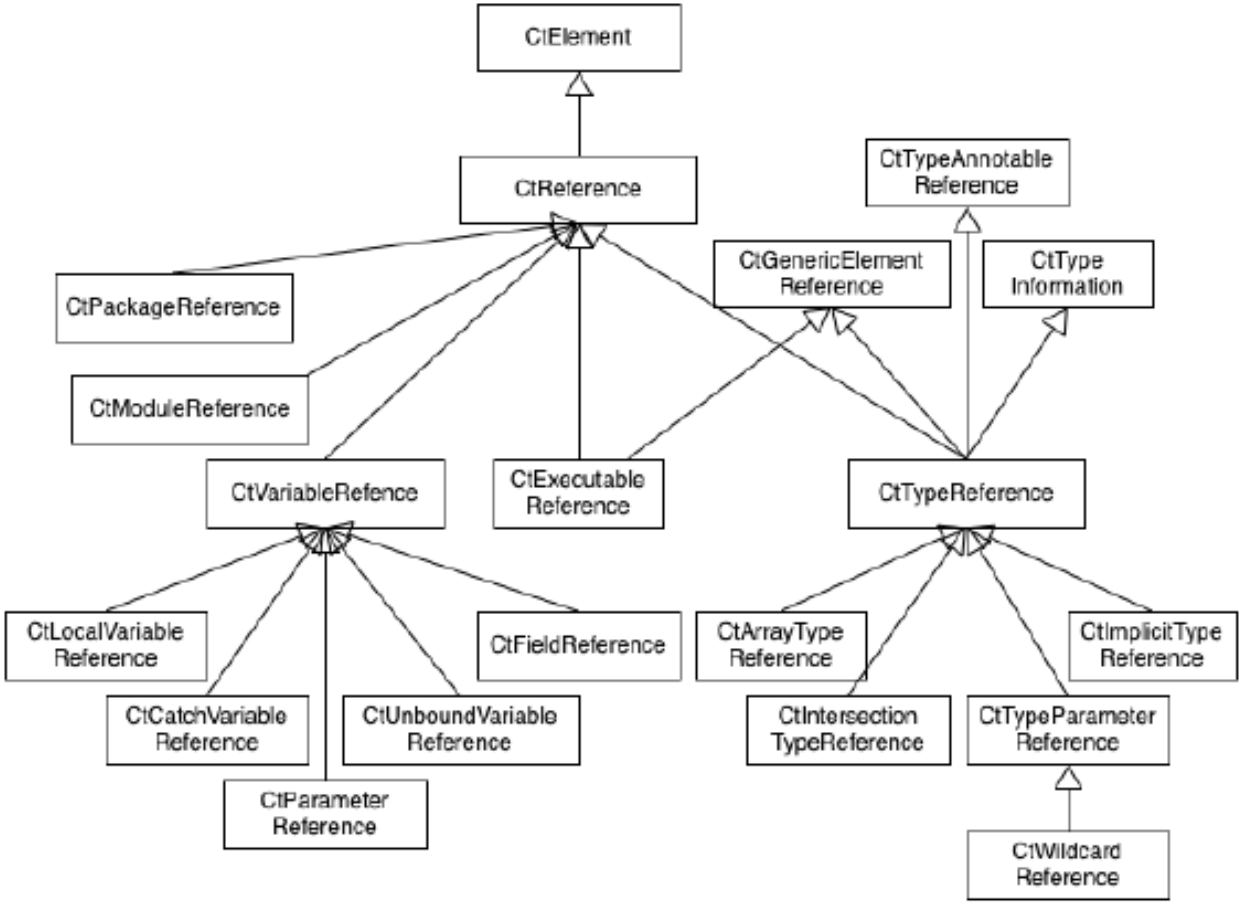
type ciblé d'une référence : `CtTypeReference#getTypeDeclaration()`.

mécanisme : référencement lors de la construction du modèle et ne ciblant que les éléments dont le code source est fourni en entrée de Spoon.

propriété : résolution de références faible puisque les cibles référencées ne doivent pas nécessairement exister au préalable.

Le Métamodèle de Spoon

Viewpoint Références (2)



Factories

motivation : lors de la conception et l'implémentation des transformations, on aura besoin de créer des implémentations pour les interfaces des éléments désirés, les initialiser, puis les ajouter au modèle construit.

mécanisme : Spoon offre une hiérarchie d'usines (Factories), où chaque usine est destinée à la création de nœuds spécifiques du modèle, et des méthodes d'usinage génériques.

exemples :

- `Class()` : fournit l'accès à l'usine `ClassFactory` spécialisée pour l'usinage des classes.
- `Code()` : fournit l'accès à l'usine `CodeFactory` spécialisée pour l'usinage des éléments exécutables.
- `Method()` : fournit l'accès à l'usine `MethodFactory` spécialisée pour l'usinage des méthodes.
- `createClass()` : une méthode générique permettant de créer un nœud vide désignant une classe.
- `createField()` : une méthode générique permettant de créer un nœud vide désignant un champs.

Getters/Setters Standards

Grâce à la réflexion, Spoon permet de récupérer/modifier différents noeuds du modèle en employant des getters/setters appropriés pour chaque type de noeud, et en utilisant des critères de recherche/modification différents :

- `CtClass#getConstructors()` : récupérer les constructeurs d'une classe.
- `CtType#getFields()` : récupérer les champs d'un type (*classe/interface*).
- `CtType#getMethods()` : récupérer les méthodes d'un type.
- `CtType#getField(String name)` : récupérer un champs nommé d'un type.

Filters

Aperçu

- **principe** : récupérer des noeuds satisfaisant des prédicats bien définis.
- **implémentations** :
  1. **natives** fournies par Spoon : *e.g., filtres de types* (`TypeFilter(Class<T> typeClass)`) *et d'annotations* (`AnnotationFilter(Class<? extends java.lang.Annotation> typeAnnotation)`).
  2. **personnalisables** : classes étendant `AbstractFilter<E extends CtElement>` (*superclasse abstraite de tous les filtres Spoon*) et implémentant la méthode `boolean matches(E element)`.

Filtres

Exemples

```
// collecting all assignments of a method body
list1 = methodBody.getElements(new TypeFilter(CtAssignment.class)
    );

// collecting all deprecated classes
list2 = rootPackage.getElements(new AnnotationFilter(Deprecated.
    class));

// a custom filter to select all public fields
list3 = rootPackage.filterChildren(new AbstractFilter<CtField>(
    CtField.class){
    @Override
    public boolean matches(CtField field){
      return field.getModifiers().contains(ModifierKind.PUBLIC);
    }
  }).list();
```

Les Queries

Aperçu

définition : un mécanisme de filtrage plus sophistiqué que les filtres classiques.

syntaxe : similaire à celles des streams Java.

propriétés :
- enchaînables ;
- réutilisables sur différents noeuds ;
- rédigeables par le biais d'expressions lambdas.

évaluation : différentes manières d'évaluation, mais la plus courante est celle retournant la liste des résultats de toutes les queries d'une chaîne via la méthode `CtQuery#list()`.

Les Queries

Exemples (1)

```
// collecting all class names
list = myPackage
  .map((CtClass c) -> c.getSimpleName())
  .list();

// collecting all deprecated classes
list2 = rootPackage
  .filterChildren(new AnnotationFilter(Deprecated.class))
  .list();

// creating a custom filter to select all public fields using
    Java 8 lambdas
list3 = rootPackage
  .filterChildren((CtField field) -> field.getModifiers()
  .contains(ModifierKind.PUBLIC))
  .list();
```

Les Queries

Exemples (2)

```
// a query which processes non -deprecated methods of deprecated
    classes
list4 = rootPackage
  .filterChildren((CtClass cls) ->
    cls.getAnnotation(Deprecated.class) != null)
  .map((CtClass cls) -> cls.getMethods())
  .map((CtMethod <?> method) ->
    method.getAnnotation(Deprecated.class) == null)
  .list();

// reusing a query
CtQuery q = Factory
  .createQuery()
  .map((CtClass cls) -> c.getSimpleName());
String cls1Name = q.setInput(Class1).list().get(0);
String cls2Name = q.setInput(Class2).list().get(0);
```

Les Queries

Exemples (3)

```
// prints each deprecated element
rootPackage
  .filterChildren(new AnnotationFilter(Deprecated.class))
  .forEach((CtElement e) -> System.out.println(e));

// returns the first deprecated element
CtElement firstDeprecated = rootPackage
  .filterChildren(new AnnotationFilter(Deprecated.class))
  .first();
```

Les Processeurs

Aperçu

motivation : définir des méthodes d'analyse et de transformation de code source, employant les différents outils d'interaction avec le modèle Spoon.

défintion : un processeur est une classe encapsulant ces différentes méthodes.

principe : tous les processeurs héritent de la classe de base de processeurs `AbstractProcessor<E extends CtElement>`, et permettent de traiter et d'analyser individuellement des types de noeuds spécifiques du modèle Spoon.

design pattern : `Visitor` appliqué aux éléments du modèle Spoon, où chaque élément définit une implémentation de la méthode `accept()` en vue d'être visité par un processeur

Les Processeurs

Processus d'usage Standard

1. Définir un processeur étendant `AbstractProcessor<E extends CtElement>` traitant un noeud de type spécifique du modèle ;
2. Eventuellement définir un prédicat de sélection des noeuds à traiter via la méthode `boolean isToBeProcessed(E candidate)` ;
3. Définir le traitement des éléments sélectionnés dans la méthode `void process(E element)` ;
4. Eventuellement arrêter le processus de traitement explicitement n'importe où dans le code du processeur défini, en invoquant la méthode `public void interrupt()`.

Les Processeurs

Exemples (1)

```
// Exemple d'un processeur de commentaires Spoon
public class CtCommentProcessor extends AbstractProcessor<
    CtComment> {
  @Override
  public boolean isToBeProcessed(CtComment candidate){
    // process only javadoc comments
    return candidate.getCommentType() == CtComment.CommentType.
        JAVADOC;
  }

  @Override
  public void process(CtComment comment){
    // process the comment
  }
}
```

Les Processeurs

Exemples (2)

```
// Exemple d'un processeur de clauses catch vides
public class CatchProcessor extends AbstractProcessor<CtCatch> {
  /* attributes */
  // empty catch clauses
  List <CtCatch> emptyCatchs = new ArrayList<>();

  @Override
  public boolean isToBeProcessed(CtCatch candidate){
    // process only empty catch clauses
    return candidate.getBody().getStatements().isEmpty();
  }
```

Les Processeurs

Exemples (3)

```
  @Override
  public void process(CtCatch element){
    getFactory()
    .getEnvironment()
    .report(this, Level.WARN, "empty catch clause"
      + element.getPosition().toString());

    emptyCatchs.add(element);
  }
}
```

Les Launchers

Aperçu

définition : un launcher pour la construction du modèle Spoon d'un code source, son traitement, son affichage et sa compilation, en utilisant le builder natif d'Eclipse JDT.

mécanismes :

1. spécifier l'ensemble des processeurs à appliquer sur des fichiers code source en entrée.
2. traiter directement un code source String d'une classe en entrée, via la méthode `Launcher.parseClass(String code)`.

Il existe aussi des launchers spécialisés :

- `IncrementalLauncher` :effectuer des builds incrémentaux en utilisant un cache.
- `MavenLauncher` : effectuer un build à partir d'un fichier de dépendances d'un projet Maven `pom.xml`.
- `JarLauncher` : construire un modèle de code source à partir d'un fichier `.jar` en utilisant un décompilateur pour décompiler le bytecode du jar.

Les Launchers

Exemples (1)

```
public class App {
  public static void main(String[] args) {
  // Example 1 : methods of a class
  CtClass l = Launcher.parseClass("
    class A {
      void m() { System.out.println(\"Hello, World!\"); }
    }");
  Set methods = l.getAllMethods();
   for(Object o: methods.toArray())
      System.out.println(o.toString());
    }
  }
```

Les Launchers

Exemples (2)

```
/* OUTPUT */
// public native final Class<?> getClass() {}
// public final void wait(long arg0, int arg1) throws
    InterruptedException {}
// public native final void wait(long arg0) throws
    InterruptedException {}
// public final void wait() throws InterruptedException {}
// public native final void notifyAll() {}
// public boolean equals(Object arg0) {}
// private static native void registerNatives() {}
// public native final void notify() {}
// void m() { System.out.println("Hello, World!"); }
// public native int hashCode() {}
// protected void finalize() throws Throwable {}
// public String toString() {}
// protected native Object clone() throws
    CloneNotSupportedException {}
```

Les Launchers

Exemples (3)

```
String[] args = {
  "-i", "src/main/java/spoon/test",
  "-o", "target/spooned"
};

Launcher launcher = new Launcher();
CommentProcessor commentP = new CommentProcessor();
CatchProcessor catchP = new CatchProcessor();
launcher.addProcessor(commentP);
launcher.addProcessor(catchP);

launcher.setArgs(args);
launcher.run()
```

Autres notions et outils

Spoon dispose d'autres outils pertinents lors de l'interaction avec le modèle d'un code source :

- `Scanners` : une mécanisme simple de visiter un noeud et ses noeuds enfants via un processus de scan (*e.g., un scan de recherche profonde sur le model, assurant que tous les noeuds enfants sont visités une fois*).
- `Iterators` : une mécanisme simple pour itérer à travers les noeuds enfants d'un noeud parent (*e.g., itérateur DFS, itérateur BFS*).
- `Paths` : l'interface `CtPath` définit un chemin vers un `CtElement` dans le métamodèle (*similairement à XPath avec les documents XML*).
- ...

Plus d'informations : consulter le booklet EclipseJDTAndSpoonBookletFichier.pdf sur Moodle.

Sources

- Spoon - Source Code Analysis and Transformation for Java : https://spoon.gforge.inria.fr/
- OW2con'18 Spoon : open source library to analyze, rewrite, transform, transpile Java source code : https://www.youtube.com/watch?v=ZZzdVTIu-OY