M2 Informatique AIGLE

HMIN306

Évolution et Restructuration

Rapport de TP

(TP 2)

Bachar Rima,
Amandine Paillard

17 janvier 2020

Table des matières

Chapitre 1

TP2 - Analyse statique et dynamique

1.1 Parties 1 et 2 - AST d'Eclipse JDT

1.1.1 Le modèle Java (*Java Model*) du JDT

Tout projet Java peut être représenté en interne par un modèle léger et tolérant aux pannes, désignant l'arbre enraciné par le dossier du projet contenant la totalité de ses éléments.

Le modèle obtenu n'est pas si riche et verbeux qu'un modèle représenté par l'AST du JDT, mais offre quand même un ensemble de traitements de base, tel que la visualisation de la vue `Package Explorer` d'Eclipse.

L'ensemble des concepts définissant les différents éléments du modèle Java sont localisés dans le plugin `org.eclipse.jdt.core`.

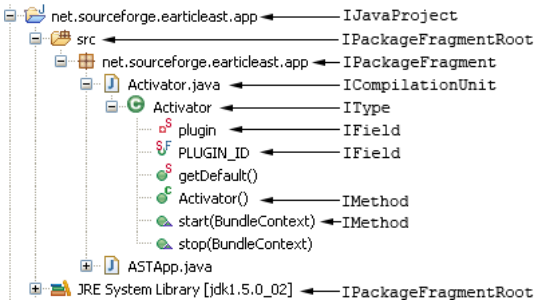

FIGURE 1.1 – Un exemple d'un projet visualisé dans `Package Explorer` grâce au modèle Java du JDT

```
1   // getting the root workspace
2  IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
3
4  // getting the project "someJavaProject" from the root workspace
5  IProject project = root.getProject("someJavaProject");
6
7  // opening the java project
8  project.open(null /*IProgressMonitor*/);
9
10 // getting the java project handle
11 IJavaProject javaProject = JavaCore.create(project);
12
```

```java
// getting a type in the java project
IType lwType = javaProject.findType("some.package.somewhere.Type");

// getting the compilation unit corresponding to the type
ICompilationUnit lwCompilationUnit = lwType.getCompilationUnit();
```

Listing 1.1 – Exemple de récupération d'une unité de compilation désignant un type dans le projet

1.1.2 L'AST du JDT

L'AST du JDT fournit une API permettant de modifier, créer, lire et supprimer du code source indirectement en manipulant ses nœuds, en traitant en entrée des unités de compilation désignant du code source (i.e. `ICompilationUnit` du modèle Java). Il s'agit d'un outil de base utilisés par plusieurs fonctionnalités d'Eclipse IDE (*refactoring, quick fix, quick assist,* ...).

Le flux de travail lors de l'utilisation de l'AST peut être résumé de la manière suivante :

1. fournir du code source en entrée sous forme d'un fichier Java ou un tableau de caractères (`char[]`) ;
2. parser le code source en utilisant une instance de `ASTParser` pour obtenir un AST et éventuellement des informations calculées supplémentaires (i.e *bindings*) sur ses nœuds ;
3. manipuler l'AST directement ou indirectement à travers une instance de `ASTRewrite` ;
4. appliquer les changements de l'AST au code source d'origine via l'interface *wrapper* du code source `IDocument`.

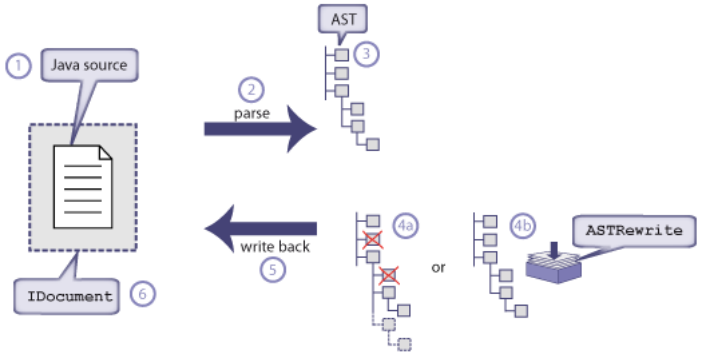

FIGURE 1.2 – Le flux de travail avec l'AST du JDT

Quelques classes de base

`ASTNode` : superclasse de tous les nœuds de l'AST.

`ASTParser` : classe définissant un parseur de code source et fournissant un AST.

`TypeDeclaration` : un nœud désignant la déclaration d'un type (*i.e.* une classe, une interface, une énumération).

`MethodDeclaration` : un nœud désignant la déclaration d'une méthode.

`VariableDeclaration` : un nœud désignant la déclaration d'une variable.

`SimpleName` : un nœud désignant n'importe quel `String` ne désignant pas un mot-clé Java, `true`, `false`, ou `null`.

`ASTVisitor` : la superclasse de toutes les classes implémentant le patron de conception *Visitor*, afin de visiter les nœuds de l'AST.

Flux de travail avec les visiteurs des nœuds de l'AST

preVisit(node) : visiter le nœud générique de l'AST (*i.e.* une instance de la classe `ASTNode`) avant de visiter son type spécifique.

visit(concreteNode) : visiter le nœud spécifique de l'AST (*i.e.* une instance d'une sous-classe d'`ASTNode`).

récursion : visiter les nœuds enfants du nœud visité, si la méthode `visit()` retourne `true`.

endVisit(concreteNode) : terminer la visite du nœud spécifique de l'AST.

postVisit(node) : terminer la visite du nœud générique de l'AST.

```
1   public class MethodDeclarationVisitor extends ASTVisitor {
2
3    private ArrayList<MethodDeclaration> methods = new ArrayList<>();
4
5    @Override
6    public boolean visit(MethodDeclaration node) {
7       methods.add(node);
8       return super.visit(node);
9    }
10
11   public ArrayList<MethodDeclaration> getMethods(){return methods;}
12  }
```

Listing 1.2 – Exemple d'un visiteur collectant les déclarations des méthodes d'une classe

Les propriétés structurelles d'un nœud de l'AST

Les propriétés structurelles d'un nœud de l'AST sont des métadonnées sur le nœud accessibles via des descripteurs, instances de la classe abstraite `StructuralPropertyDescriptor` ou de l'une de ses spécialisations `SimplePropertyDescriptor`, `ChildPropertyDescriptor` ou `ChildListPropertyDescriptor`, selon la valeur de la propriété.

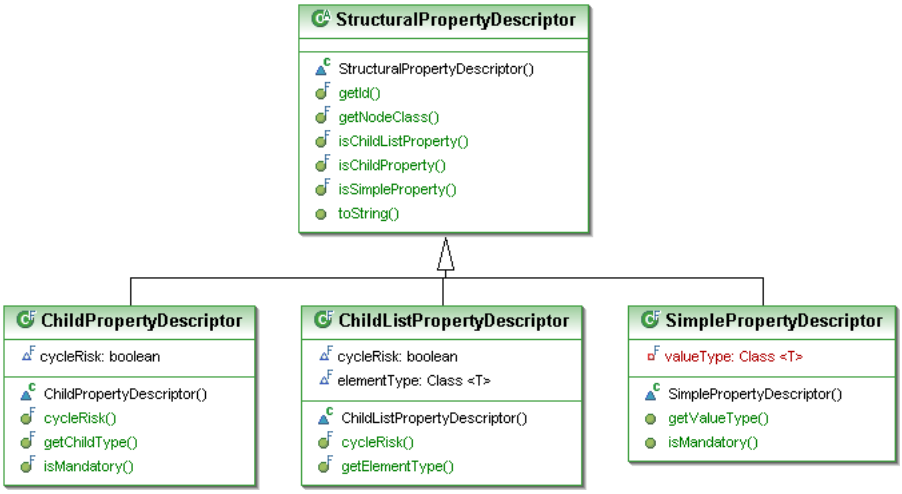

FIGURE 1.3 – Hiérarchie des descripteurs des propriétés structurelles d'un nœud de l'AST

Récupérer des informations sur les nœuds de l'AST

On peut récupérer des informations sur les nœuds de l'AST :

1. en accédant à des méthodes dédiées pour les nœuds spécifiques : *e.g.* les méthodes `getName()` et `thrownExceptions()` pour récupérer le nom et la liste des exceptions levées, respectivement, d'un nœud désignant une déclaration d'une méthode (*i.e.* une instance de la classe `MethodDeclaration`).
2. en utilisant des méthodes génériques d'accès aux propriétés structurelles des nœuds : *e.g.* les méthodes `getStructuralProperty(propertyDescriptor)` et `structuralPropertiesForType()` retournant, respectivement, la valeur d'une propriété structurelle pointée par le descripteur « `propertyDescriptor` » et la liste des propriétés structurelles du nœud générique de l'AST courant.

FIGURE 1.4 – Exemple des propriétés structurelles d'une déclaration de méthode

Les *bindings* d'un nœud de l'AST

Des informations sur certains nœuds de l'AST peuvent être calculées via le mécanisme des liaisons (*bindings*). Il s'agit d'heuristiques exécutées lors de la construction de l'AST via le parseur pour calculer des informations diverses.

En pratique, certaines classes définissant des nœuds spécifiques de l'AST disposent au moins de la méthode `resolveBinding()` retournant le *binding* du type du nœud. D'autres classes disposent davantage de méthodes de résolution de liaisons diverses, telles que la classe `MethodInvocation` désignant l'invocation d'une méthode et disposant de la méthode `resolveMethodBinding()` retournant le *binding* de la déclaration de la méthode invoquée.

```
1    int i = 7;
2    System.out.println("Hello!");
3    int x = i * 2;
```

Listing 1.3 – Exemple d'un bout de code sur lequel la résolution des liaisons est activée

Remarque.
Le service de résolution des *bindings* doit être explicitement activé lors de la création du parseur avant de parser le code source, via l'invocation de la méthode du parseur `setResolveBindings(true)`.

Modification des nœuds de l'AST

Parfois nous aurons besoin d'introduire des changements au code source analysé à travers l'AST. Pour ce faire, soit on effectue des modifications directement sur l'AST, soit on délègue la gestion des modificationsw à une instance de `ASTRewrite`.

FIGURE 1.5 – Exemple du *binding* de la référence de la variable i à la ligne 3 du listing 1.3

```
1  // creating an instance of rewrite on the CompilationUnit "unit"'s AST
2  ASTRewrite rewrite = ASTRewrite.create(unit.getAST());
3
4  VariableDeclarationStatement statement =
       createNewVariableDeclarationStatement(
5  manager, ast); // creating a new variable declaration statement
6
7  // getting the first reference index of the block in which to add the
       statement
8  int firstReferenceIndex = getFirstReferenceIndex(manager, block);
9
10 // recover the list of statements of the block to rewrite
11 ListRewrite statementsListRewrite = rewrite.getListRewrite(
12 block, Block.STATEMENTS_PROPERTY);
13
14 // inserting the statement into the list of statements at the 1st reference
       index
15 statementsListRewrite.insertAt(statement, firstReferenceIndex, null);
```

Listing 1.4 – Exemple d'ajout d'un nœud à un AST via ASTRewrite

```
1  ASTRewrite rewrite = ASTRewrite.create(unit.getAST());
2  // renaming the name of the method invocation
3  rewrite.set(methodInvocation, MethodInvocation.NAME_PROPERTY, newName, null);
4
5  // same effect using a different method
6  rewrite.replace(methodInvocation.getName(), newName, null);
```

Listing 1.5 – Exemple de modification d'un nœud de l'AST via ASTRewrite

```
1  // enable modification recording at the root of the AST
2  unit.recordModifications();
3  // ...
4  VariableDeclarationStatement statement =
       createNewVariableDeclarationStatement(
5  manager, ast); // creating a new variable declaration statement
6
7  // getting the first reference index of the block in which to add the
       statement
8  int firstReferenceIndex = getFirstReferenceIndex(manager, block);
9
```

```
// adding the statement at the beginning of the block
block.statements().add(firstReferenceIndex, statement);
```

Listing 1.6 – Exemple de modification directe d'un nœud de l'AST

Répercuter les modifications des nœuds de l'AST au code source

Les changements effectués sur l'AST sont répercutés au code source d'origine par le biais de l'interface *wrapper* `IDocument` et ses implémentations.

```
  /*
* document: the source code file parsed by ASTParser
* options: source code formatter options, (null for default options)
*/

// invoked on the ASTRewrite instance if used
TextEdit rewriteAST(IDocument document, Map options);

// invoked on the CompilationUnit if the AST is directly modified
TextEdit rewrite(IDocument document, Map options);
```

Listing 1.7 – méthodes de répercussion des modifications de l'AST au code source

```
  // get a file buffer manager
ITextFileBufferManager bufferManager =
    FileBuffers.getTextFileBufferManager();

// get the path of the source file "unit" (CompilationUnit)
IPath path = unit.getJavaElement().getPath();

try{
  /* connect a path of a file buffer manager
  * after this call, the document of the file described by "path"
  * can be obtained and modified
  */
  bufferManager.connect(path, null);

  // retrieve the text file buffer
  ITextFileBuffer textFileBuffer = bufferManager.getTextFileBuffer(path);

  // ask the buffer for a working copy of the document
  IDocument document = textFileBuffer.getDocument();

  /*rewrite changes of AST into the document*/

  // commit changes to the underlying file
  textFileBuffer.commit(null /*ProgressMonitor*/, false /*Overwrite*/);
}

catch (Exception e){
  /*handle exception*/
}

finally {
  /* disconnect the file buffer manager from the path
  * after this call, the document of the file described by "path"
  * should no longer be modified
  */
  bufferManager.disconnect(path, null);
}
```

Listing 1.8 – Exemple générique de répercussion des modifications d'un AST au code source

```
  CompilationUnit astRoot = ...; // current compilation unit

// creating an instance of rewrite on the CompilationUnit "astRoot"'s AST
ASTRewrite rewrite = ASTRewrite.create(astRoot.getAST());

// retrieving the body of the first method of the first class in the unit
Block block = ((TypeDeclaration) astRoot.types().get(0))
  .getMethods()[0].getBody();

// retrieve the list of statements of the body
ListRewrite listRewrite = rewrite.getListRewrite(block,
    Block.STATEMENTS_PROPERTY);

// creating a comment
Statement placeHolder = rewrite.createStringPlaceholder("//mycomment",
ASTNode.EMPTY_STATEMENT);

// inserting the comment prior to the method's body
listRewrite.insertFirst(placeHolder, null);

// retrieving the text-based modifications introduced to the AST of "astRoot"
TextEdit textEdits = rewrite.rewriteAST(document, null);

// committing the changes to the document
textEdits.apply(document);
```

Listing 1.9 – Exemple d'ajout d'un commentaire au début du corps d'une méthode

1.1.3 Les processeurs utilisant l'AST du JDT

Dans le cadre de ce TP, nous avons repris le code de la classe utilisant l'`ASTParser` qui nous a été fourni et nous l'avons restructuré pour avoir une version orientée-objet dans la classe `Parser`. `Parser` est utilisée par tous les processeurs cherchant à visiter les nœuds d'un AST pour effectuer des tâches d'analyse spécifiques. En effet, nous avons défini un processeur de base `BaseProcessor` utilisant une instance de `Parser` et factorisant sa configuration. Ce processeur est spécialisé par tous les processeurs spécifiques.

Pour la première partie du TP2, nous avons défini le processeur `InfoProcessor` définissant des méthodes permettant de visiter les nœuds des classes, de leurs attributs, de leurs méthodes et des méthodes invoquées dans ces dernières et d'extraire les informations requises (*cf.* Figure ??).

Pour la deuxième partie du TP2, nous avons défini le processeur `StatsProcessor` définissant des méthodes permettant d'extraire les statistiques requises sur les éléments d'un projet (cf. Figure 1.7).

Les deux processeurs sont testés sur un projet contenant des *sample codes* illustrant différents patrons de conception (`Composite`, `Visitor`, `Singleton`, ...).

```
Class: composite.Song
Superclass: SongComponent
Attributes:
  private String songName
  private String bandName
  private int releaseYear
Methods:
  @Override public String Song::getSongName()
  @Override public String Song::getBandName()
  @Override public int Song::getReleaseYear()
  @Override public void Song::displaySongCompone

Class: composite.DiscJockey
Superclass: N/A
Attributes:
  private SongComponent songList
Methods:
  public SongComponent DiscJockey::getSongList()
  public void DiscJockey::displaySongList()

Class: composite.SongListGenerator
Superclass: N/A
Attributes:
Methods:
  public static void SongListGenerator::main(Str

Class: composite.SongComponent
```

(a) Aperçu des résultats orientés-classes obtenus par `InfoProcessor`

```
Class: composite.Song
Superclass: SongComponent
Methods:
  @Override public String Song::getSongName()
  @Override public String Song::getBandName()
  @Override public int Song::getReleaseYear()
  @Override public void Song::displaySongComponentInfo() invokes
    StringBuilder::append(this.getSongName(): String)
    Song::getSongName()
    StringBuilder::append(" ": String)
    StringBuilder::append(this.getBandName(): String)
    Song::getBandName()
    StringBuilder::append(" in ": String)
    StringBuilder::append(this.getReleaseYear(): int)
    Song::getReleaseYear()
    System.out::println(buf.toString())
    StringBuilder::toString()

Class: composite.DiscJockey
Superclass: N/A
Methods:
  public SongComponent DiscJockey::getSongList()
  public void DiscJockey::displaySongList() invokes:
    songList::displaySongComponentInfo()
...
```

(b) Aperçu des résultats orientés-méthodes obtenus par `InfoProcessor`

```
Nombre de packages : 7
Nombre de classes : 36
Nombre de méthodes : 115
Lignes de code : 1510
Moyenne méthodes/classe : 3.1944444444444446
Moyenne attributs/classe :1.0833333333333333
Moyenne lignes/méthode :13.130434782608695
nombre maximal de paramètres par rapport à toutes les méthodes : 3

10% classes avec plus grand nombre de méthodes :
  state.ATMMachine
  visitor.TaxVisitor
  factory.EnemyShip

10% classes avec plus grand nombre d'attributs :
  state.ATMMachine
  observer.StockObserver
  singleton.Singleton

10% classes avec plus grand nombre de méthodes et d'attributs :
  state.ATMMachine

classes avec plus que 3 méthodes :
  composite.Song
  composite.SongComponent
  composite.SongGroup
  singleton.Singleton
  visitor.TaxVisitor
  ...

10% des méthodes qui possèdent le plus grand nombre de statements par classe :
adapter.EnemyRobot : smashwithHands
observer.GrabStocks : main
state.HasCard : ejectCard
visitor.TaxHolidayVisitor : TaxHolidayVisitor
composite.Song : displaySongComponentInfo
factory.RocketEnemyShip : RocketEnemyShip
...
```

FIGURE 1.7 – Aperçu des résultats obtenus par `StatsProcessor`

1.1.4 Le graphe d'appel statique

Pour construire le graphe d'appel statique d'un projet, nous avons défini tout d'abord un graphe d'appel de base `AbstractCallGraph` définissant l'interface commune à tous les graphes d'appel.

Le graphe d'appel statique est modélisé dans la classe `StaticCallGraph`, héritant l'interface définie par `AbstractCallGraph` et utilisant une instance de `BaseProcessor` pour définir les traitements nécessaires pour la construction du graphe d'appel statique en visitant les nœuds de l'**AST** obtenus par le parser de `BaseProcessor`. Pour ce faire, nous avons adopté une approche récursive consistant à construire le graphe d'appel statique de chaque unité de compilation et de réaliser leur union pour obtenir le graphe d'appel statique de tout le projet. Chaque appel est associé à un nombre désignant le nombre de fois que cet appel est réalisé statiquement dans le code du projet (*cf.* Figure 1.8).

```
factory.EnemyShipTesting::doStuffEnemy:
  ---> enemy::followHeroShip (1 fois)
  ---> enemy::enemyShipShoots (1 fois)
  ---> enemy::displayEnemyShip (1 fois)
state.HasCard::requestCash:
  ---> System.out::println (1 fois)
adapter.EnemyTank::driveForward:
  ---> System.out::println (1 fois)
  ---> generator::nextInt (1 fois)
state.CorrectPin::requestCash:
  ---> this.context::getCashInMachine (3 fois)
  ---> this.context::setATMState (1 fois)
  ---> this.context::setCashInMachine (1 fois)
  ---> System.out::println (2 fois)
  ---> this::ejectCard (1 fois)
  ---> this.context::getNoCashState (1 fois)
observer.StockGrabber::register:
  ---> observers::add (1 fois)
observer.StockObserver::update:
  ---> observer.StockObserver::displayPrices (1 fois)
visitor.Necessity::accept:
  ---> visitor::visit (1 fois)
visitor.TaxVisitor::visit:
  ---> visitor.TaxVisitor::computeTax (3 fois)
singleton.GetTheTiles::run:
  ---> instance::getLettersList (1 fois)
  ---> instance::getTiles (1 fois)
  ---> System::identityHashCode (1 fois)
  ---> Singleton::getInstance (1 fois)
  ---> System.out::println (4 fois)
```

FIGURE 1.8 – Aperçu des résultats obtenus par `StatsProcessor`

Les graphe d'appel statique est testé sur le même projet précédent.

1.2 Partie 3 - Étude de l'outil Spoon

1.2.1 Introduction

`Spoon` est une librairie *open-source* permettant :

1. l'analyse et la transformation de code source Java ;
2. l'analyse de bytecode Java après sa décompilation ;
3. la transpilation (*e.g.* `Java` → `JavaScript`)
4. ...

1.2.2 Installation en tant qu'un *plugin* Maven

Créer un nouveau projet Maven sous Eclipse. Afin d'utiliser `Spoon`, il faut ajouter les lignes suivantes dans le fichier de gestion des dépendances Maven `pom.xml` :

```
<dependencies>
```

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

1.2.3 Le métamodèle de Spoon

`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. Ils peuvent être répartis en trois catégories :

éléments structurels : les éléments de déclarations d'un programme (*e.g. classes, interfaces, énumérations, variables, méthodes, variables, ...*) (*cf.* Figure 1.9).

éléments exécutables : les éléments exécutables de Java (*e.g.* les corps de méthodes/constructeurs, invocations de méthodes/constructeurs, les *statements*, les expressions, ...) (*cf.* Figure 1.10).

éléments de référence : des éléments désignant des références de type (*cf.* Figure 1.11).

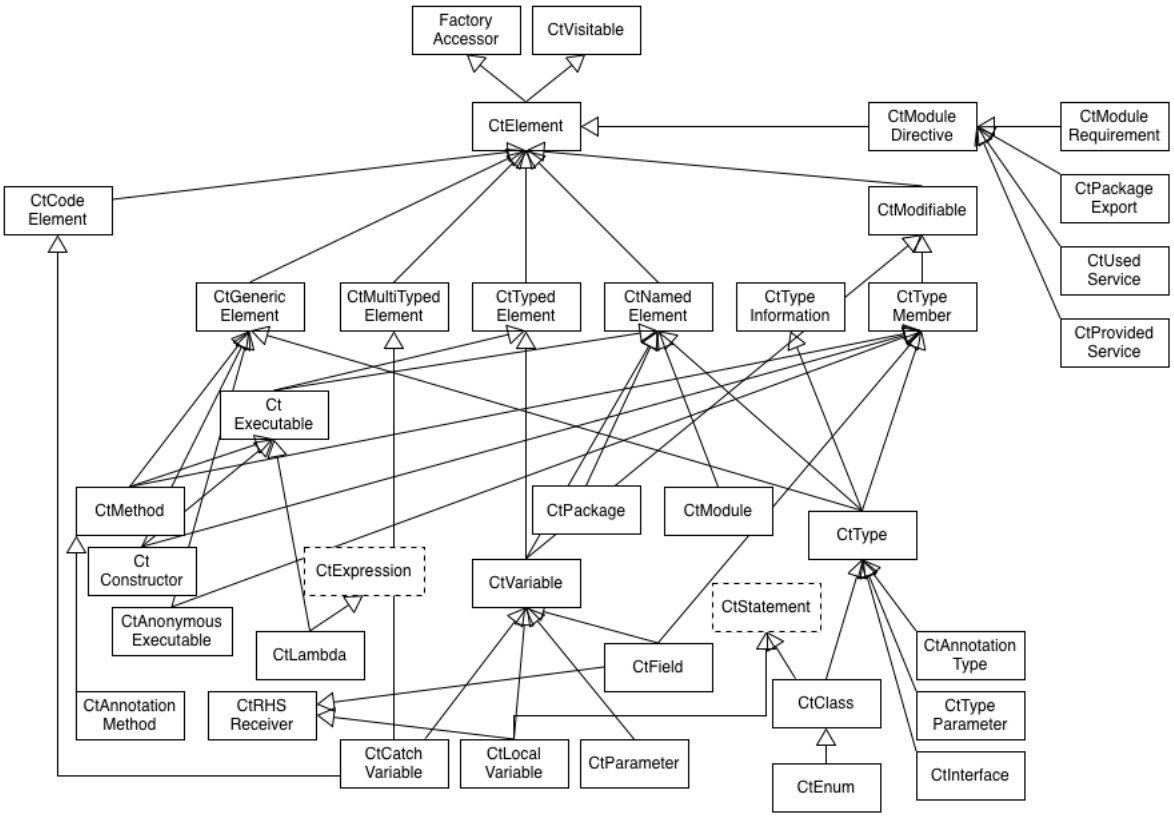

FIGURE 1.9 – Éléments structurels du métamodèle `Spoon`

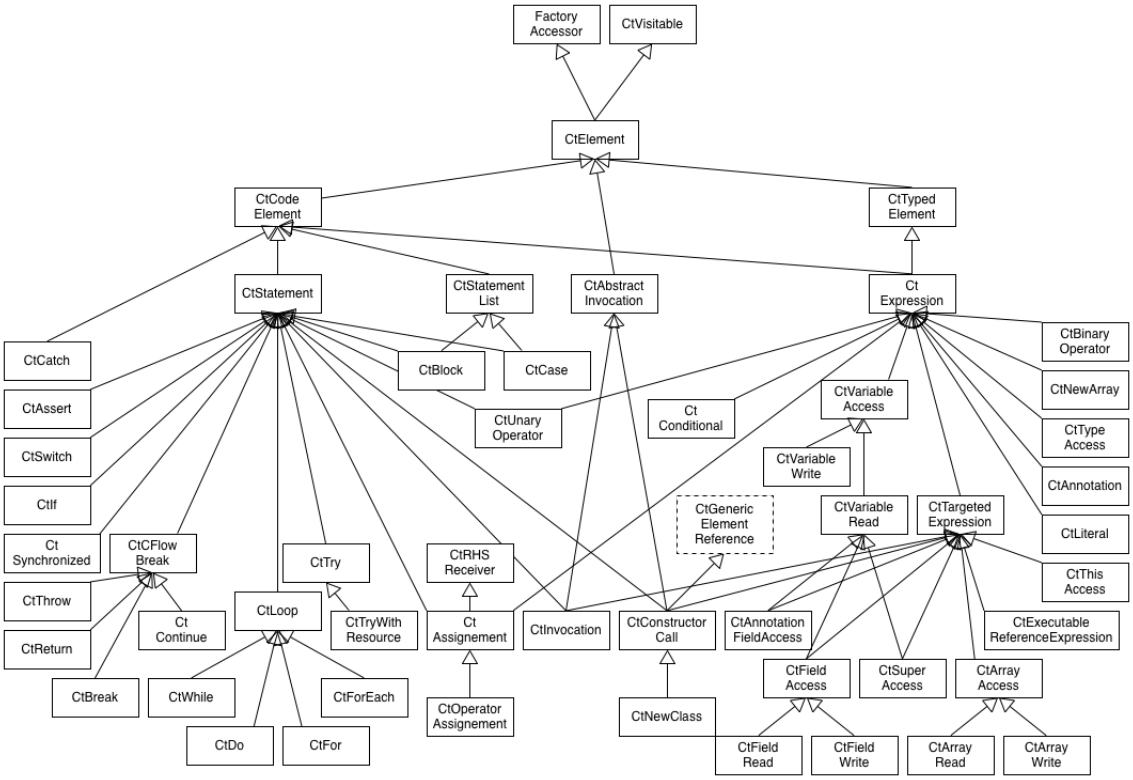

FIGURE 1.10 – Éléments exécutables du métamodèle `Spoon`

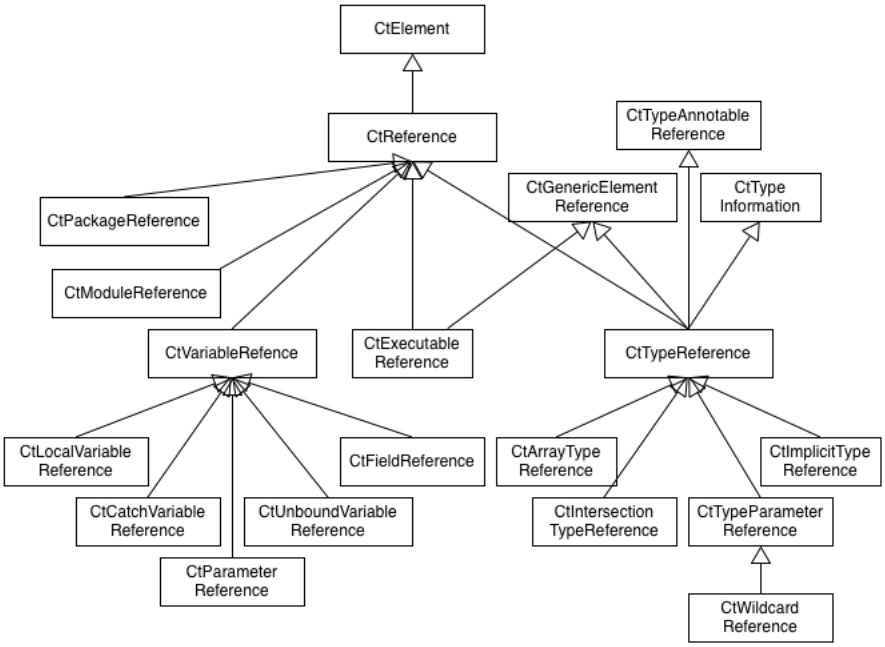

FIGURE 1.11 – Éléments de référence du métamodèle `Spoon`

1.2.4 Les références

Le mécanisme de référencement de type désigne le référencement d'éléments qui ne sont pas forcément réifiés au sein du métamodèle, comme ils peuvent appartenir à des bibliothèques tierces. Ceci permet une flexibilité lors de la construction/modification du modèle du programme en cours d'analyse.

Par exemple, référencer un objet de type `String` ne désigne pas le modèle compilable de `String.java`, étant donné que le code source de `String.java` ne fait pas partie (*en général*) du programme en cours d'analyse.

Pour récupérer la référence d'un type cible et le type ciblé d'une référence, on utilise, respectivement, les méthodes `CtType#getType()` et `CtTypeReference#getTypeDeclaration()`.

La résolution des références se fait lors de la construction du modèle et ne cible que les éléments dont le code source est fourni en entrée de `Spoon`. Cette résolution est faible puisque les cibles des références ne doivent pas nécessairement exister au préalable.

1.2.5 Le processus standard d'utilisation de Spoon

1. construire le modèle `Spoon` du projet à analyser ;
2. analyser et effectuer des *queries* sur les parties pertinentes du projet ;
3. transformer le code source qui doit être transformé ;
4. fournir un code source transformé du projet.

1.2.6 Afficher le modèle Spoon d'un code source Java

```
java -cp <spoon-jar> spoon.Launcher -i <class>.java --gui --noclasspath
```

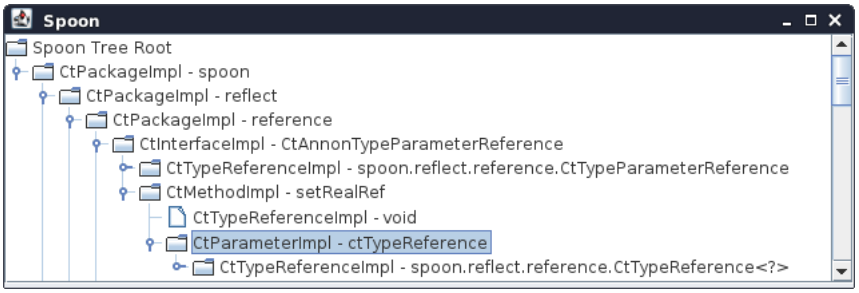

FIGURE 1.12 – Exemple d'un modèle Spoon d'une unité de code Java

1.2.7 Les *factories*

Lors de la conception et l'implémentation des transformations, nous aurons besoin de créer des implémentations des interfaces d'éléments fournies, les initialiser et les ajouter au modèle construit. Pour ce faire, `Spoon` offre une hiérarchie d'usines (***Factories***) où chaque usine est destinée à la création de nœuds spécifiques du modèle `Spoon`.

L'interface d'usine de base `Factory` fournit des points d'accès aux usines spécialisées telles que :

Class() : fournit l'accès à l'usine `ClassFactory` spécialisée pour l'usinage des classes.

Constructor() : fournit l'accès à l'usine `ConstructorFactory` spécialisée pour l'usinage des constructeurs.

Field() : fournit l'accès à l'usine `FieldFactory` spécialisée pour l'usinage des attributs.

Method() : fournit l'accès à l'usine `MethodFactory` spécialisée pour l'usinage des méthodes.

Code() : fournit l'accès à l'usine `CodeFactory` spécialisée pour l'usinage des éléments exécutables.

Type() : fournit l'accès à l'usine `TypeFactory` spécialisée pour l'usinage des types.

En outre, elle fournit des méthodes d'usinage génériques telles que :

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 attribut.

createMethod() : une méthode générique permettant de créer un nœud vide désignant une méthode.

1.2.8 Les *getters/setters* standards

En utilisant la réflexion, `Spoon` permet de récupérer/modifier différents nœuds du modèle en employant des *getters/setters* appropriés pour chaque type de nœud, utilisant des critères de recherche/modification différents.

Par exemple, pour récupérer les constructeurs, les méthodes et les attributs d'une classe, nous utiliserons, respectivement, `CtClass#getConstructors()`, `CtType#getMethods()` et `CtType#getFields()`.

D'autre part, si nous souhaitons récupérer un attribut par son nom, nous utiliserons `CtType#getField(String name)`, par exemple.

1.2.9 Les filtres

Le principe des filtres est de récupérer des nœuds satisfaisant des prédicats bien définis. Certaines implémentations de filtres sont fournies par `Spoon`, telles que les filtres de types (`TypeFilter(Class<T> typeClass)`) et d'annotations (`AnnotationFilter(Class<? extends java.lang.Annotation> typeAnnotation)`). Toutefois, une implémentation personnalisable peut être fournie par l'utilisateur par l'extension de la classe `AbstractFilter<E extends CtElement>`, superclasse abstraite de tous les filtres `Spoon`. Cette classe implémente l'interface `Filter<E extends CtElement>` et il suffit de fournir une implémentation de sa seule méthode, `boolean matches(E element)`.

Exemple

```
  // 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();
```

Listing 1.10 – Exemple d'utilisation de filtres Spoon

1.2.10 Les *queries*

À partir de `Spoon` 5.5, on a introduit les *queries*, désignant un mécanisme de filtrage plus sophistiqué que les filtres classiques. En particulier, elles peuvent être enchaînées, réutilisées sur différents nœuds et rédigées par le biais de lambdas.

Lors de l'enchaînement des *queries* :

— si le résultat d'une *query* est non nul, il sera passé en entrée de la *query* suivante dans la chaîne.

— si le résultat d'une *query* est itérable, chacun de ses éléments constituera une entrée différente à la *query* suivante.

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

Exemple

```
  // 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();

// 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);

// 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();
```

Listing 1.11 – Exemple d'utilisation des queries Spoon

Remarque.
Spoon dispose d'autres outils pertinents lors de l'interaction avec le modèle d'un code source, mais qui ne seront pas abordés dans ce tutoriel étant au delà de la portée du TP, tels que les *scanners*, les *iterateurs*, les *paths*, les *roles*, les *templates* et les *patterns*.

1.2.11 Les processeurs

Pour définir des méthodes d'analyse et de transformation de code source, employant les différents outils d'interaction avec le modèle `Spoon` vu précédemment, `Spoon` propose la notion de **processeur**, une classe encapsulant ces différents traitements.

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 nœuds spécifiques du modèle `Spoon`.

En interne, le traitement des éléments est effectué par le biais du *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 visiteur.

Processus standard d'utilisation d'un processeur Spoon

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

Exemples

```
1  public class CtCommentProcessor extends AbstractProcessor<CtComment> {
2
3    @Override
4    public boolean isToBeProcessed(CtComment candidate){
5      // process only javadoc comments
6      return candidate.getCommentType() == CtComment.CommentType.JAVADOC;
7    }
8
9    @Override
10   public void process(CtComment comment){
11     // process the comment
12   }
13 }
```

Listing 1.12 – Exemple d'un processeur de commentaires Spoon

```
1    public class CatchProcessor extends AbstractProcessor<CtCatch> {
2    /*attributes*/
3    // empty catch clauses
4    List<CtCatch> emptyCatchs = new ArrayList<>();
5
6    @Override
7    public boolean isToBeProcessed(CtCatch candidate){
8      // process only empty catch clauses
9      return candidate.getBody().getStatements().isEmpty();
10   }
11
12   @Override
13   public void process(CtCatch element){
14     getFactory()
15     .getEnvironment()
```

```
16      .report(this, Level.WARN, "empty catch clause"
17        + element.getPosition().toString());
18
19      emptyCatchs.add(element);
20    }
21  }
```

Listing 1.13 – Exemple d'un processeur de clauses catch vides

1.2.12 Les *launchers*

La classe `Launcher` modélise un *launcher* CLI pour la construction du modèle `Spoon` d'un code source, ainsi que son traitement, affichage et sa compilation, en utilisant le *builder* natif d'Eclipse JDT.

Le *launcher* permet de spécifier l'ensemble des processeurs à appliquer sur des fichiers de code source en entrée, comme il peut être utilisé pour traiter directement un `String` en entrée, désignant le code source d'une classe, en invoquant la méthode statique `Launcher.parseClass(String code)`.

Par ailleurs, des*launchers* dédiés plus spécifiques peuvent être utilisés :

`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*.

Exemples

```
1   public class App {
2     public static void main(String[] args) {
3         // Exemple 1 : Les méthodes d'une classe
4       CtClass l = Launcher.parseClass("
5           class A {
6               void m() { System.out.println("Hello, World!"); }
7            }
8       ");
9       Set methods = l.getAllMethods();
10      for(Object o: methods.toArray())
11         System.out.println(o.toString());
12    }
13  }
```

Listing 1.14 – Exemple de construction d'un modèle Spoon d'une classe en directe

```
1   */Méthodes de la classe App et celles héritées d'Object*/
2   ============================================================
3   public native final Class<?> getClass() {}
4   public final void wait(long arg0, int arg1) throws InterruptedException {}
5   public native final void wait(long arg0) throws InterruptedException {}
6   public final void wait() throws InterruptedException {}
7   public native final void notifyAll() {}
8   public boolean equals(Object arg0) {}
9   private static native void registerNatives() {}
10  public native final void notify() {}
11  void m() {
12      System.out.println("Hello, World!");
13  }
14  public native int hashCode() {}
```

```
protected void finalize() throws Throwable {}
public String toString() {}
protected native Object clone() throws CloneNotSupportedException {}
```

Listing 1.15 – Résultat du listing 1.14

```
  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();
```

Listing 1.16 – Exemple d'utilisation d'un Launcher pour appliquer plusieurs processeurs à un projet

1.2.13 Instrumentation de code et traces d'appels

Dans le cadre de ce TP, nous avons construit un projet Maven dépendant de `spoon-core 8.0.0.jar` pour effectuer des tâches d'analyse dynamique sur des codes source. Au sein du projet, nous avons défini un processeur Spoon `ToStringGenerator` pour traiter les classes d'un projet quelconque de la manière suivante :

1. filtrer les attributs ayant des *getters* et leurs *getters* ;
2. instrumenter les méthodes faisant appel à ces *getters* ;
3. si la classe ne déclare pas une implémentation de la méthode `toString()`, celle-ci sera générée automatiquement en utilisant les *getters* et les attributs filtrés.

L'instrumentation des méthodes est *offline* et est effectuée par un *logger* personnalisé composant une instance statique du *logger* de l'**API Java** pour le *logging* (*i.e.* `java.util.logging.Logger`). En outre, les traces d'appel sont formatées par :

`java.util.logging.SimpleFormatter` : un formateur de texte simple, prédéfini par l'**API** du *logging* pour les messages à *logger* ;

`HTMLFormatter` : un formateur personnalisé permettant de formater du texte dans des pages **HTML**.

Afin de tester notre processeur, nous avons défini un projet de test contenant les classes suivantes (*cf.* Figure 1.13) :

Empty : une classe vide ne définissant pas d'attributs propres, mais définissant une méthode `display()` qui affiche un message indiquant que la classe est vide ;

Closed : une classe ayant des attributs propres mais ne définissant aucun *getter* dessus. La classe définit une méthode `callClosedAttrK()` d'accès à l'attribut `closedAttrK` ($K \in \mathbb{N}^*$) sans passer par un *getter*, et une méthode `callAllClosedAttr()` invoquant les méthodes précédentes pour tous les attributs ;

AlreadyHasToString : une classe ayant des attributs propres, avec leurs *getters* correspondant, mais ayant déjà sa propre implémentation de la méthode `toString()`. La classe

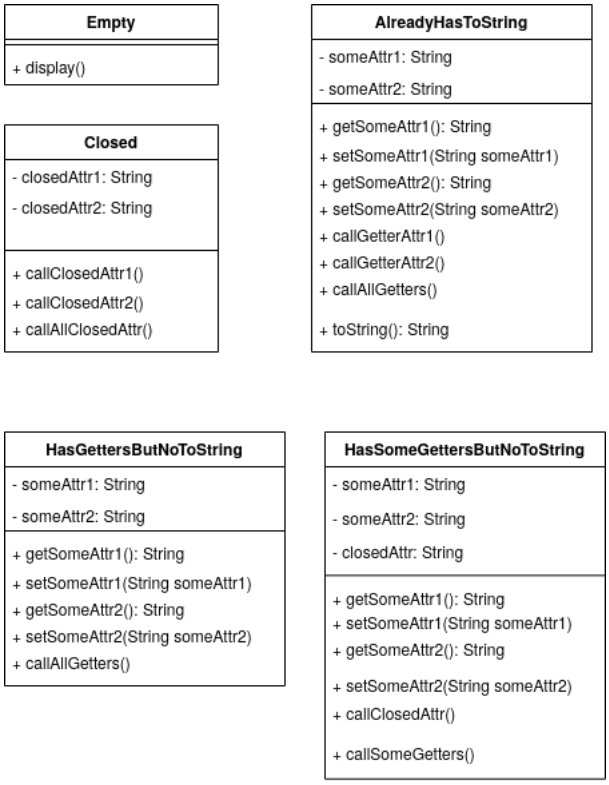

FIGURE 1.13 – Classes de test de `ToStringGenerator`

définit une méthode `callGetterAttrK()` pour accéder à un attribut `someAttrK` ($K \in \mathbb{N}^*$) via son *getter* correspondant, et une méthode `callAllGetters()` invoquant les *getters* de tous les attributs ;

HasSomeGettersButNoToString : une classe ayant des attributs propres n'ayant pas tous de *getters* correspondant, et n'ayant pas une implémentation de `toString()`. La classe définit la méthode `callClosedAttr()` pour accéder à son attribut n'ayant pas un *getter* et la méthode `callSomeGetters()` pour accéder aux attributs ayant leurs propres *getters* via leur *getters*.

HasGettersButNoToString : une classe ayant des attributs propres ayant tous de *getters* correspondant, mais n'ayant pas une implémentation de `toString()`. La classe définit la méthode `callAllGetters()` pour accéder à tous les attributs via leurs *getters*.

Main : la classe définissant les différents scénarios d'exécution en manipulant des instances des classes dessus.

Selon la définition de `ToStringGenerator()`, seules les classes `HasSomeGettersButNoToString` et `HasGettersButNoToString` auront une implémentation de `toString()` générée à partir de leurs attributs ayant des *getters* (*cf.* Listing 1.17).

```
1   // generated toString() for class
        to_string_generator.data.HasGettersButNoToString:
2  /* Automatically generated by Spoon */
3  @java.lang.Override
4  public java.lang.String toString() {
5      java.lang.StringBuffer buf = new java.lang.StringBuffer();
6      buf.append("someAttr1:\n");
7      buf.append("==========\n");
8      buf.append(getSomeAttr1()+"\n\n");
9
10     buf.append("someAttr2:\n");
11     buf.append("==========\n");
12     buf.append(getSomeAttr2()+"\n\n");
13
```

```
14      return buf.toString();
15  }
16  // generated toString() for class
        to_string_generator.data.HasSomeGettersButNoToString:
17  /* Automatically generated by Spoon */
18  @java.lang.Override
19  public java.lang.String toString() {
20      java.lang.StringBuffer buf = new java.lang.StringBuffer();
21      buf.append("someAttr1:\n");
22      buf.append("==========\n");
23      buf.append(getSomeAttr1()+"\n\n");
24
25      buf.append("someAttr2:\n");
26      buf.append("==========\n");
27      buf.append(getSomeAttr2()+"\n\n");
28
29      return buf.toString();
30  }
```

Listing 1.17 – implémentations de toString() générées automatiquement pour les classes HasGettersButNoToString et HasSomeGettersButNoToString

Par ailleurs, toutes les méthodes invoquant des *getters* seront instrumentées par l'insertion d'instructions *sensors* consistant à *logger* l'invocation par `SpoonLogger` dans un fichier texte et un fichier **HTML** dans le dossier `log/` associé à chaque projet instrumenté par le processeur.

Un exemple d'une méthode qui sera instrumentée par le processeur est la méthode `HasGettersButNoToString::ca` (*cf.* Listing 1.18)

```
1       public void callAllGetters() {
2           java.lang.System.out.println("called all attributes with getters:");
3           my_spoon.logger.SpoonLogger.info(
4           "to_string_generator.data.HasGettersButNoToString::getSomeAttr1()
                invoked from
                to_string_generator.data.HasGettersButNoToString::callAllGetters()");
5           java.lang.System.out.println("calling someAttr1 with getter:" +
                getSomeAttr1());
6           my_spoon.logger.SpoonLogger.info(
7           "to_string_generator.data.HasGettersButNoToString::getSomeAttr2()
                invoked from
                to_string_generator.data.HasGettersButNoToString::callAllGetters()");
8           java.lang.System.out.println("calling someAttr2 with getter:" +
                getSomeAttr2());
9       }
```

Listing 1.18 – instrumentation de la méthode HasGettersButNoToString::callAllGetters

Enfin, les traces d'exécution obtenues par le *logging* du projet de test peuvent être visualisées dans la figure 1.14.

```
Jan 16, 2020 1:48:06 PM my_spoon.logger.SpoonLogger info
INFO: to_string_generator.data.HasGettersButNoToString::getSomeAttr1() invoked from
to_string_generator.data.HasGettersButNoToString::callAllGetters()
Jan 16, 2020 1:48:06 PM my_spoon.logger.SpoonLogger info
INFO: to_string_generator.data.HasGettersButNoToString::getSomeAttr2() invoked from
to_string_generator.data.HasGettersButNoToString::callAllGetters()
Jan 16, 2020 1:48:06 PM my_spoon.logger.SpoonLogger info
INFO: to_string_generator.data.HasSomeGettersButNoToString::getSomeAttr1() invoked from
to_string_generator.data.HasSomeGettersButNoToString::callSomeGetters()
Jan 16, 2020 1:48:06 PM my_spoon.logger.SpoonLogger info
INFO: to_string_generator.data.HasSomeGettersButNoToString::getSomeAttr2() invoked from
to_string_generator.data.HasSomeGettersButNoToString::callSomeGetters()
```

(a) Log textuel par SpoonLogger

Thu Jan 16 13:48:06 CET 2020 /n /n /n /n

LogLevel	Time	LogMessage
INFO	Jan 16,2020 13:48	to_string_generator.data.HasGettersButNoToString::getSomeAttr1() invoked from to_string_generator.data.HasGettersButNoToString::callAllGetters()
INFO	Jan 16,2020 13:48	to_string_generator.data.HasGettersButNoToString::getSomeAttr2() invoked from to_string_generator.data.HasGettersButNoToString::callAllGetters()
INFO	Jan 16,2020 13:48	to_string_generator.data.HasSomeGettersButNoToString::getSomeAttr1() invoked from to_string_generator.data.HasSomeGettersButNoToString::callSomeGetters()
INFO	Jan 16,2020 13:48	to_string_generator.data.HasSomeGettersButNoToString::getSomeAttr2() invoked from to_string_generator.data.HasSomeGettersButNoToString::callSomeGetters()

(b) Log HTML par SpoonLogger

FIGURE 1.14 – Logs par SpoonLogger pour les classes traitées par ToStringGenerator

1.2.14 Le graphe d'appel dynamique

Pour construire le graphe d'appel dynamique d'un projet, nous avons ciblé les classes contenant une méthode `main()` définissant des exécutions particulières du code du projet. En particulier, l'ajout des invocations au graphe doit se faire à la volée par le biais d'instructions *sensors* instrumentant la méthode `main()` d'une classe et toutes les méthodes invoquées dedans (*si leurs déclarations sont disponibles*).

Le graphe d'appel dynamique est modélisé dans la classe `DynamicCallGraph` qui reprend l'interface définie par le graphe d'appel de base `AbstractCallGraph` définissant l'interface commune à tous les graphes d'appel et utilisée dans la première partie du TP2. Toutefois, nous l'avons réadaptée pour que ses membres soient statiques. Cette réadaptation découle du fait que l'insertion des instructions *sensors* est plus facile et moins verbeuse avec `Spoon` depuis un contexte statique que depuis un contexte d'instance. Ainsi il suffit d'insérer les instructions *sensors* dans `main()` et dans les méthodes invoquées dedans et de terminer l'instrumentation par l'insertion d'une instruction d'affichage du graphe d'appel dynamique obtenu dans la méthode `main()` instrumentée. L'instrumentation des méthodes est *offline* et est effectuée par le biais d'un processeur `Spoon` `DynamicCallGraphProcessor` utilisant des filtres et des méthodes d'accès aux noeuds de l'**AST** obtenu pour un projet.

Afin de tester notre processeur, nous avons défini un projet de test contenant les classes suivantes (*cf.* Figure 1.15) :

Person : une classe définissant une interface de base d'une personne. Une personne peut posséder zéro/plusieurs maison(s) si elle a 21 ans ou plus.

Address : une classe définissant une interface de base d'une adresse.

House : une classe définissant une interface de base d'une maison. Une maison a exactement une adresse.

Main : la classe définissant les différents scénarios d'exécution en manipulant des instances des classes dessus.

Pour créer plusieurs scénarios d'exécution, nous utilisons `java.util.Random` pour générer des nombres aléatoires désignant les âges des personnes, ce qui entraînera des modifications sur

leur capacité de posséder une maison et par conséquent sur le chemin d'exécution choisi dans la méthode `main()` du projet. Les méthodes des classes sont implémentées de manière à pouvoir tester plusieurs localisations différentes des méthodes invoquées, dont l'insertion des instructions *sensors* dépendera.

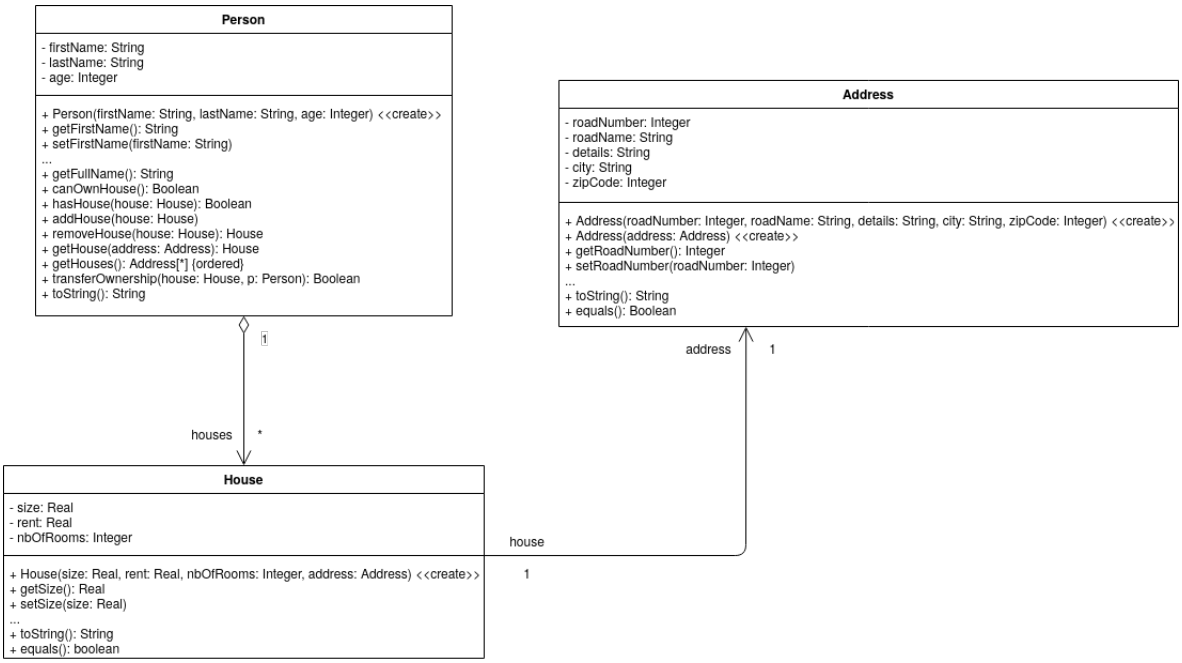

FIGURE 1.15 – Classes de test de `DynamicCallGraphProcessor`

Ainsi, toutes les méthodes invoquées dans `main()` et les méthodes invoquées dedans seront instrumentées par l'insertion d'instructions *sensors* consistant à ajouter leurs invocations dans la liste des invocations de `DynamicCallGraph`.

Un exemple d'une méthode qui sera instrumentée par le processeur est la méthode `Person::addHouse` (*cf.* Listing 1.19)

```
1    public void addHouse(dynamic_callgraph.data.House house) {
2        my_spoon.callgraph.DynamicCallGraph.addInvocation(
3        "dynamic_callgraph.data.Person::addHouse",
             "dynamic_callgraph.data.Person::canOwnHouse");
4        my_spoon.callgraph.DynamicCallGraph.addInvocation(
5        "dynamic_callgraph.data.Person::addHouse",
             "dynamic_callgraph.data.Person::hasHouse");
6        if (canOwnHouse() && (!hasHouse(house))) {
7            my_spoon.callgraph.DynamicCallGraph.addInvocation(
8            "dynamic_callgraph.data.Person::addHouse",
                 "java.util.List::add");
9            this.houses.add(house);
10       }
11   }
```

Listing 1.19 – instrumentation de la méthode Person::addHouse

Enfin, un aperçu du graphe d'appel dynamique obtenu du projet de test peut être visualisé dans la figure 1.16.

Remarque.
Le code source du TP2 est disponible sur https://github.com/anonbnr/hmin306_TP/tree/master/tp2/src.

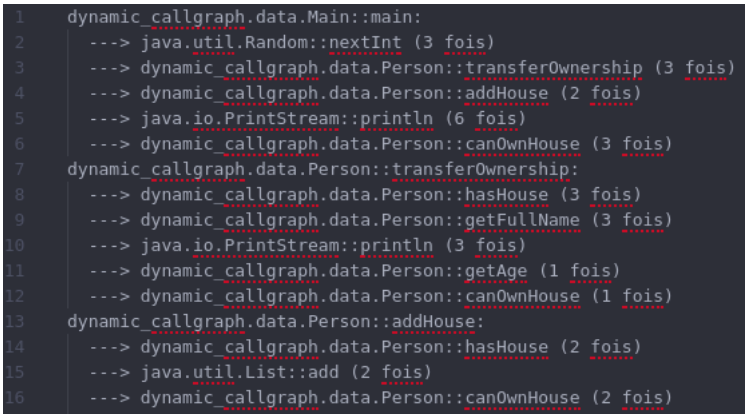

FIGURE 1.16 – Aperçu des invocations du graphe d'appel dynamique obtenu du projet de test