Annexes

Table des matières

I Description de modèles et des adaptateurs 4

1 Description de modèles 4

2 Description des adaptateurs 14

3 Opérateurs de composition 16

3.1 Description des opérateurs de composition 16
3.2 Les introductions 17
3.2.1 Introduction d'une assertion 17
3.2.2 Concrétisation d'une méthode 18
3.2.3 Insertion d'un paramètre dans une signature 19
3.2.4 Introduction d'un invariant de classe 20
3.2.5 Introduction d'une méthode dans une classe 21
3.2.6 Introduction d'un attribut dans une classe 22
3.2.7 Introduction d'une superclasse dans une classe 23
3.2.8 Introduction d'une sousclasse dans une classe 24
3.2.9 Introduction d'une classe dans un paquetage 25
3.2.10 Introduction d'un paquetage dans un paquetage 26
3.2.11 Introduction d'une association dans un paquetage 27
3.3 Les fusions 28
3.3.1 Fusion d'une méthode concrète dans une méthode abstraite 29
3.3.2 Fusion d'une méthode concrète dans une méthode concrète 30
3.3.3 Fusion par simple ajout de constituants. 33
3.3.4 Fusion personnalisée de deux classes 35
3.3.5 Fusion d'associations. 37
3.4 Modifications 39
3.4.1 Renommage 39
3.4.2 Changement du typage 40
3.4.3 Modification de la visibilité d'un élément 41
3.5 Interceptions 42
3.5.1 Interception avant une méthode 43
3.5.2 Interception après une méthode 44
3.5.3 Interception autour d'une méthode 45
3.5.4 Interception sur levée d'exception 46

4 L'expression des cibles 47

5 Langage de composition 48

Table des figures

1 Hiérarchie du métamodèle 4
2 Métamodèle des préoccupations 5
3 Paquetage 6
4 Association 6
5 Extrémité d'association 7
6 Classe 7
7 Assertion 8
8 Invariant de classe 8
9 Méthode 9
10 Corps de méthode 9
11 Assertion de méthode 10
12 Precondition 10
13 Postcondition 10
14 Attribut 10
15 Association 11
16 Signature 11
17 Signature de méthode 11
18 Signature d'attribut 12
19 Modificateurs de visibilité 12
20 Type 12
21 Paramêtre 13
22 Appel de méthode 13
23 Métamodèle de composition 14
24 Diagramme de la classe Adapter 15
25 Héritage des adaptateurs 15
26 Diagramme de la classe Adaptation 16
27 Hiérarchie des adaptations pour les modèles 16
28 L'introduction 17
29 Introduction d'une assertion 17
30 Concrétisation d'une méthode 18
31 Introduction d'un nouveau paramètre 19
32 Introduction d'un invariant de classe. 20
33 Introduction d'une méthode dans une classe 21
34 Introduction d'un attribut dans une classe 22
35 Introduction d'une superclasse dans une classe 23
36 Introduction d'une sous-classe dans une classe 24
37 Introduction d'une classe dans un paquetage. 25
38 Introduction d'un paquetage dans un paquetage. 26
39 Introduction d'une association dans un paquetage. 27
40 La fusion 28
41 Fusion d'une méthode concrete dans une méthode abstraite . . . 29
42 Fusion d'une méthode concrète dans une méthode concrète . . . 30
43 Fusion par simple ajout de constituants. 33
44 Fusion personnalisée de deux classes 35

45 Fusion d'associations. 37
46 La modification 39
47 Renommage 39
48 Changement du typage 40
49 Modification de la visibilité d'un élément 41
50 Classe Interception 42
51 Interception avant une méthode 43
52 Interception après une méthode 44
53 Interception autour d'une méthode 45
54 Interception autour d'une méthode 46

Première partie

Description de modèles et des adaptateurs

1 Description de modèles

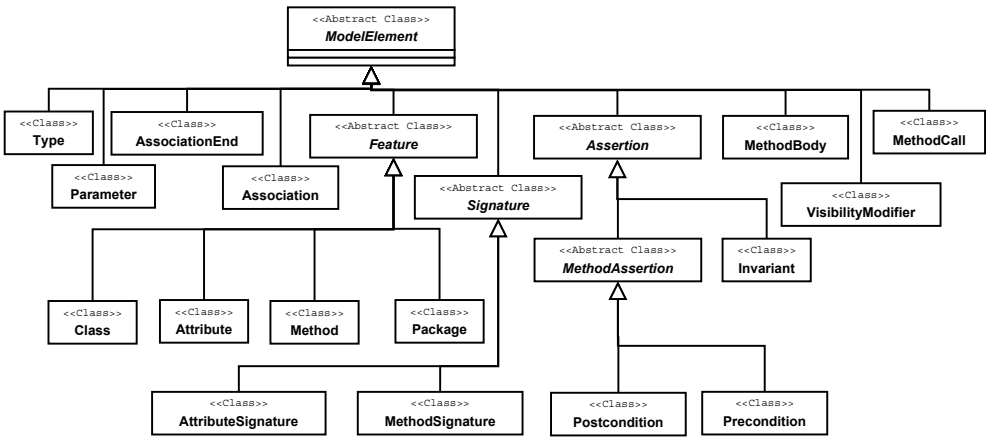

Fig. 1 – Hiérarchie du métamodèle

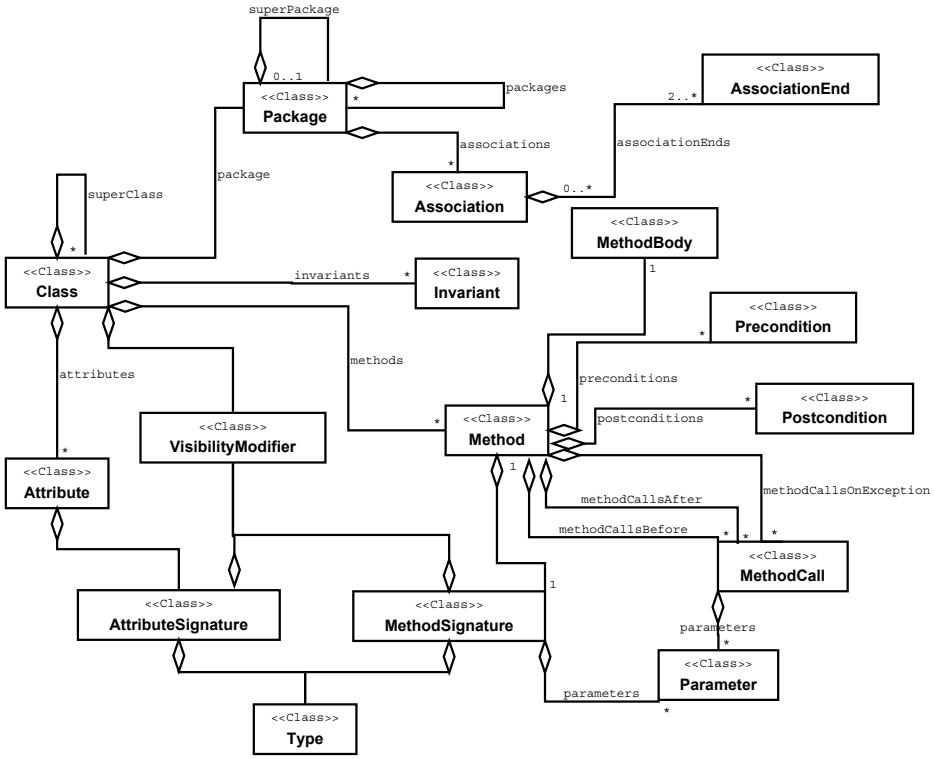

FIG. 2 – Métamodèle des préoccupations

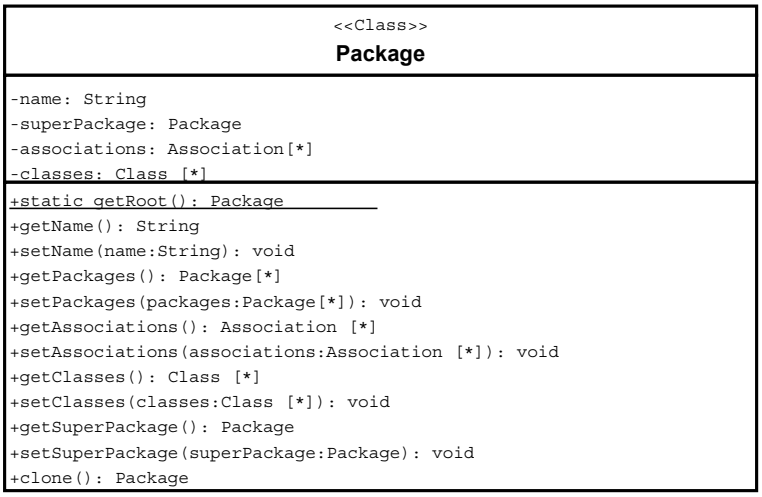

FIG. 3 – Paquetage

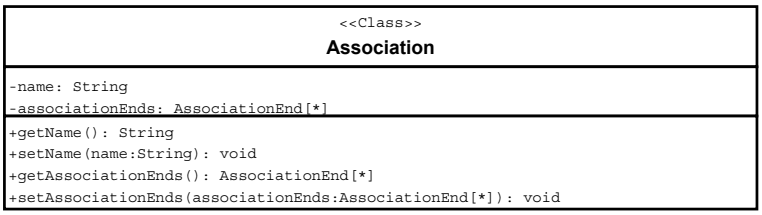

FIG. 4 – Association

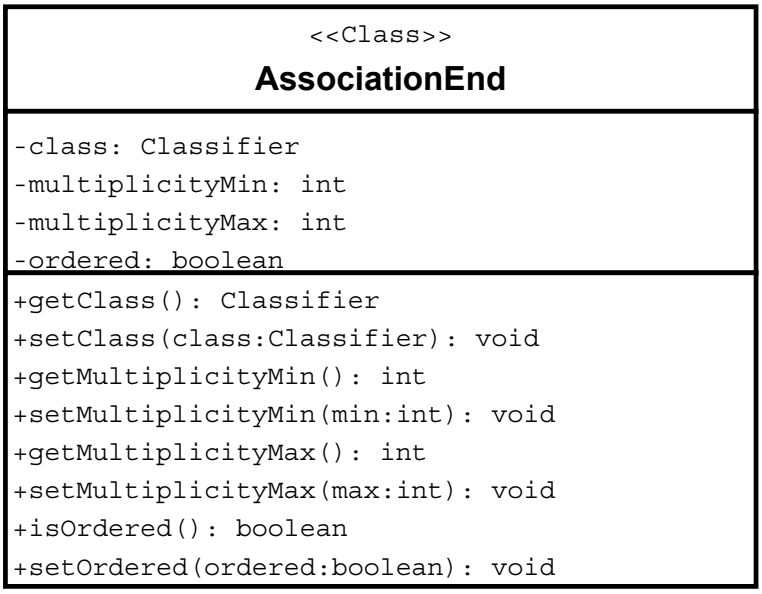

FIG. 5 – Extrémité d'association

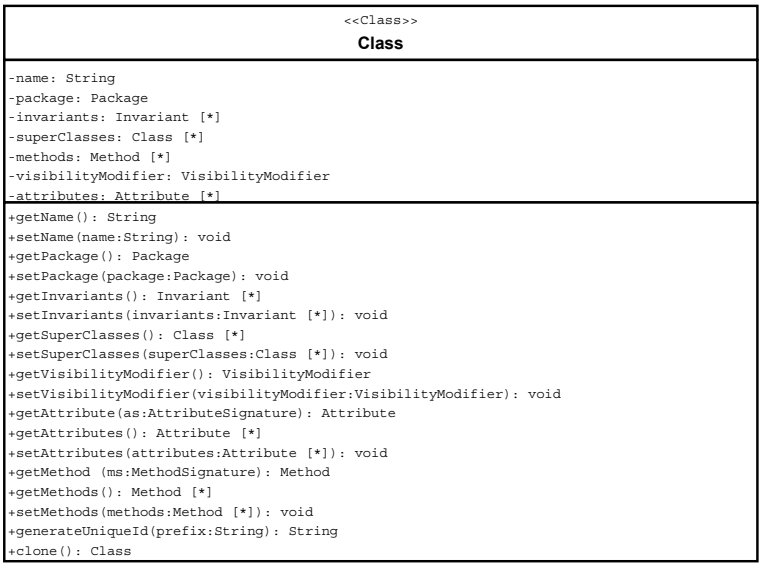

FIG. 6 – Classe

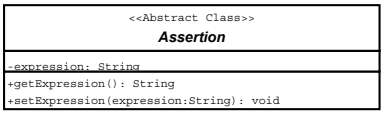

FIG. 7 – Assertion

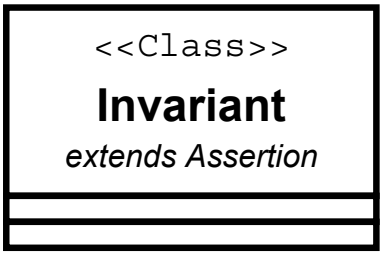

FIG. 8 – Invariant de classe

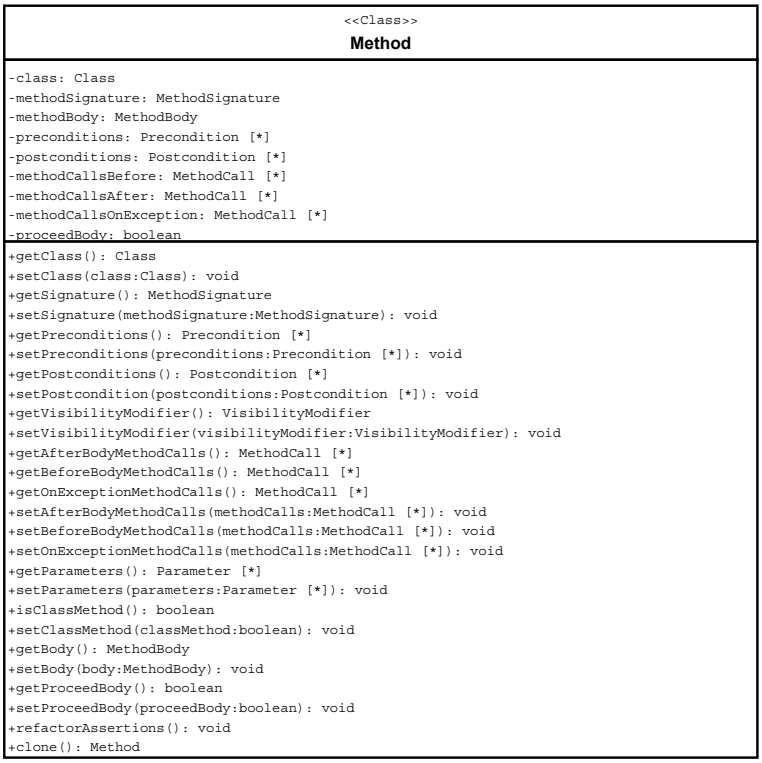

FIG. 9 – Méthode

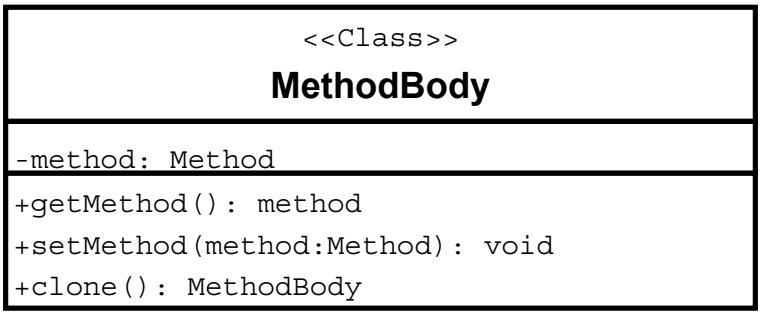

FIG. 10 – Corps de méthode

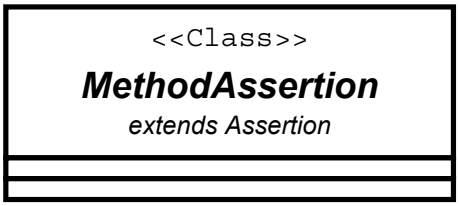

FIG. 11 – Assertion de méthode

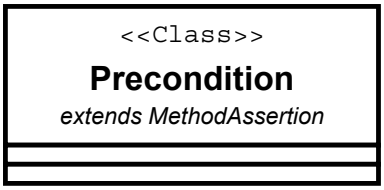

FIG. 12 – Precondition

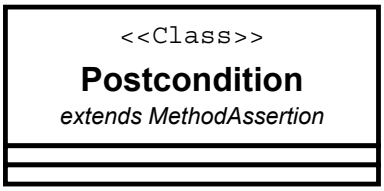

FIG. 13 – Postcondition

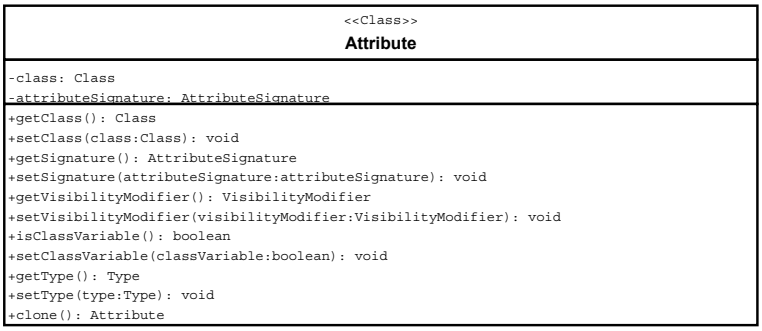

FIG. 14 – Attribut

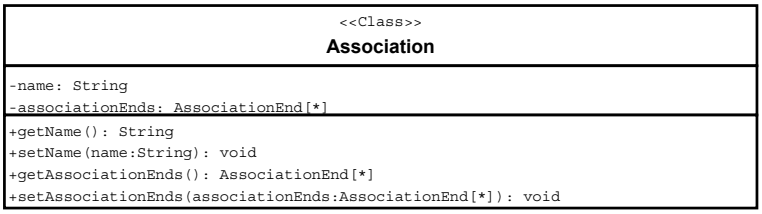

FIG. 15 – Association

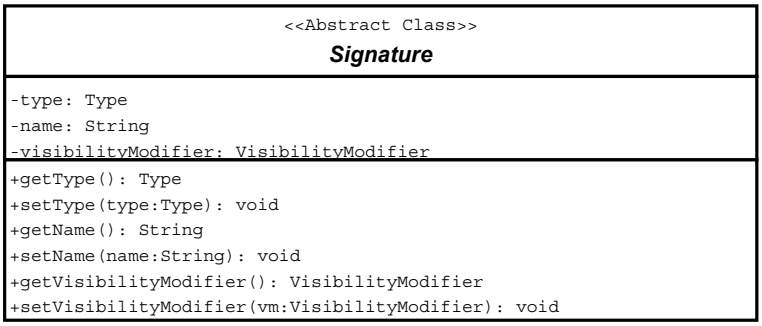

FIG. 16 – Signature

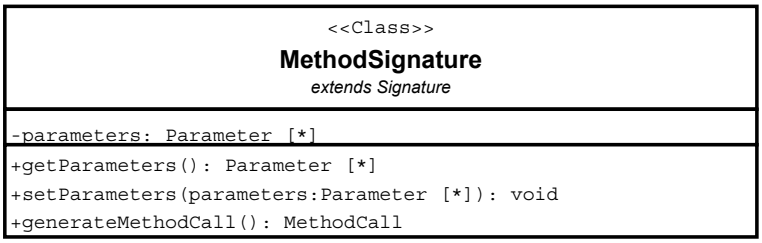

FIG. 17 – Signature de méthode

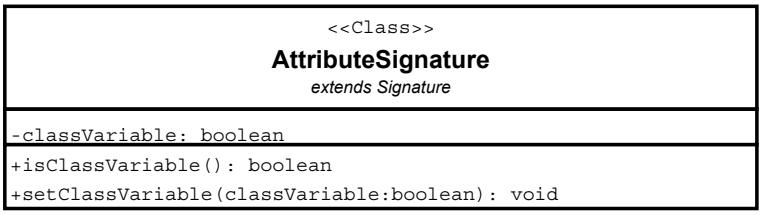

FIG. 18 – Signature d'attribut

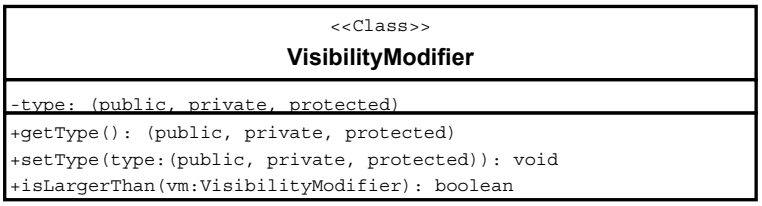

FIG. 19 – Modificateurs de visibilité

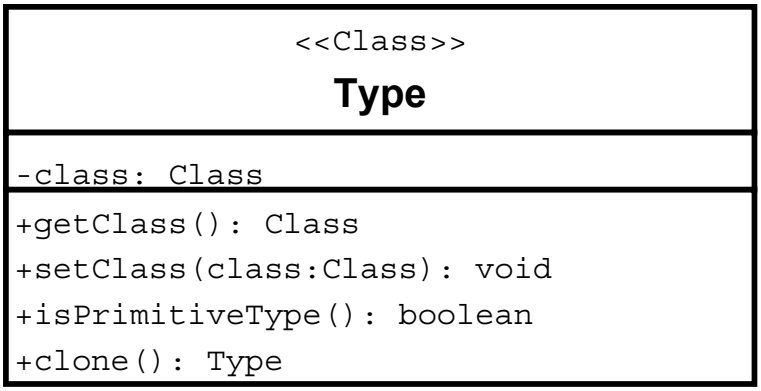

FIG. 20 – Type

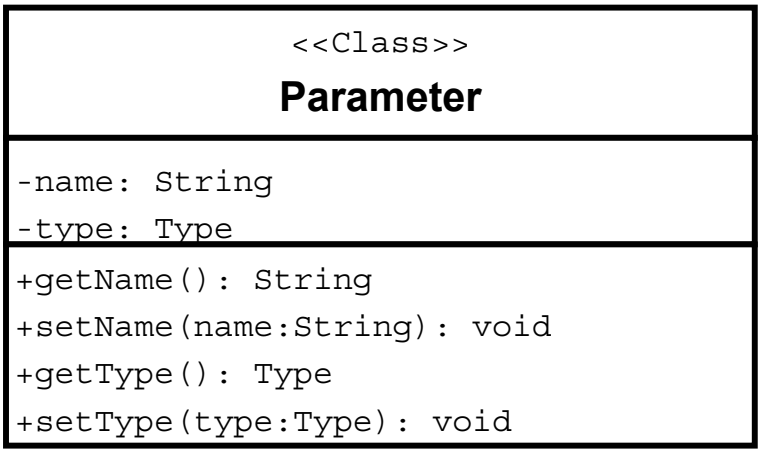

FIG. 21 – Paramêtre

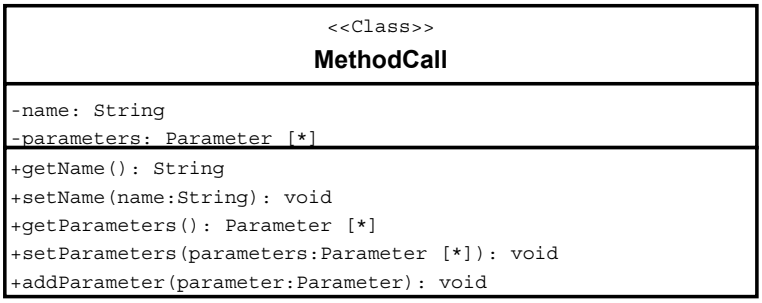

FIG. 22 – Appel de méthode

2 Description des adaptateurs

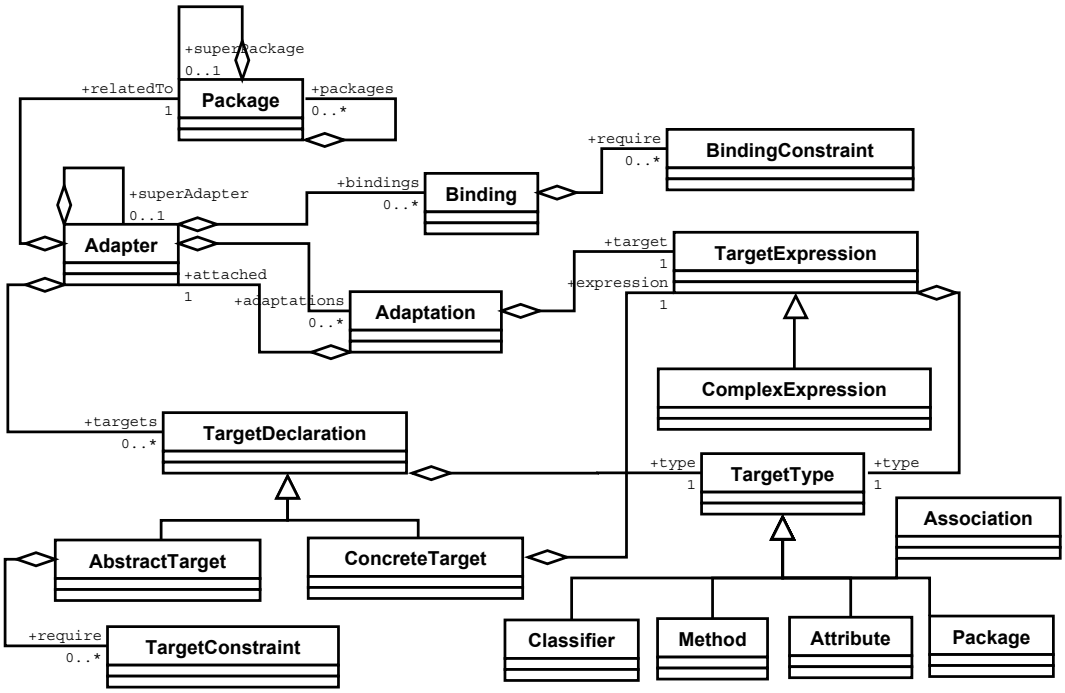

FIG. 23 – Métamodèle de composition

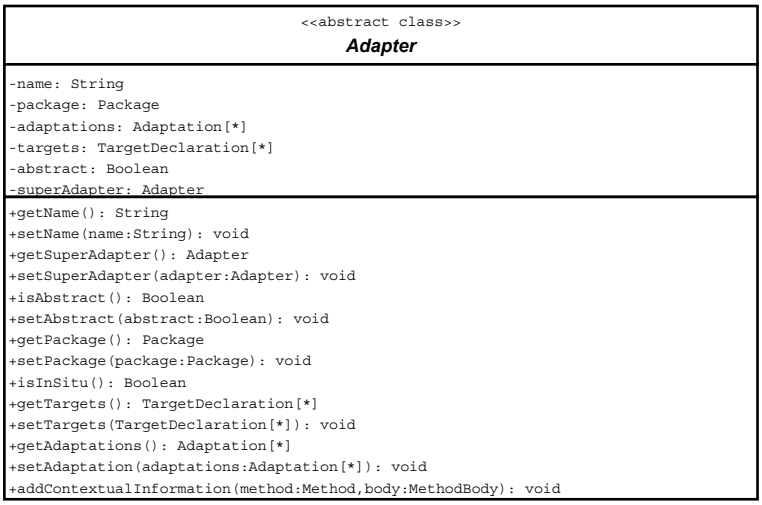

FIG. 24 – Diagramme de la classe Adapter

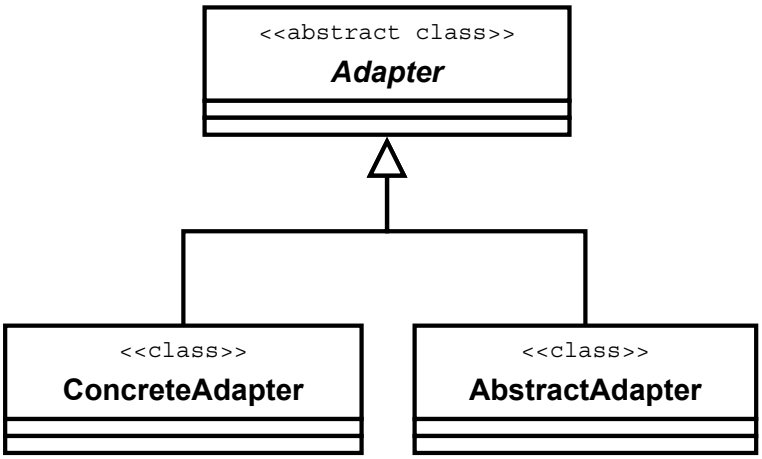

FIG. 25 – Héritage des adaptateurs

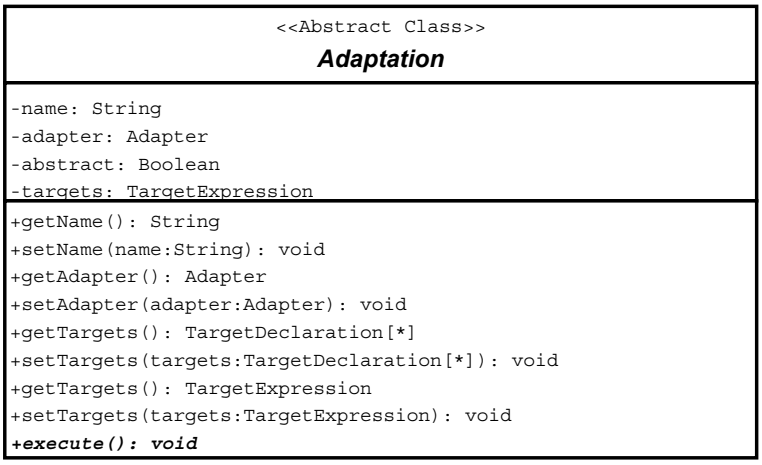

FIG. 26 – Diagramme de la classe Adaptation

3 Opérateurs de composition

3.1 Description des opérateurs de composition

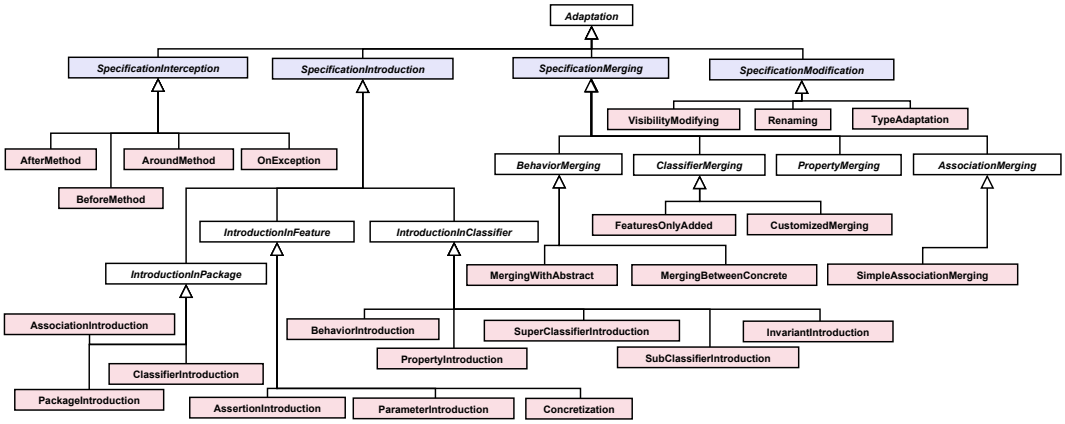

FIG. 27 – Hiérarchie des adaptations pour les modèles

Les adaptations sont décomposées en quatre sous-parties formant les grandes familles d'adaptations : les introductions, les fusions, les modifications et les interceptions.

3.2 Les introductions

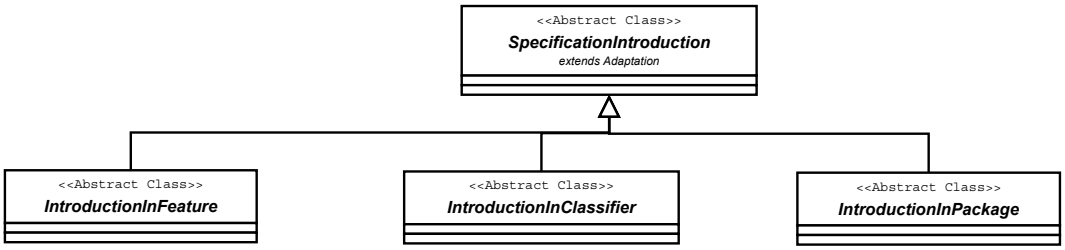

FIG. 28 – L'introduction

L'introduction s'effectue à différents niveaux. Il est par exemple possible d'introduire des assertions ou des nouveaux paramètres dans une méthode, ou bien des méthodes, attributs ou invariants dans une classe, ou bien encore des classes, des associations, ou des paquetages dans un paquetage.

3.2.1 Introduction d'une assertion

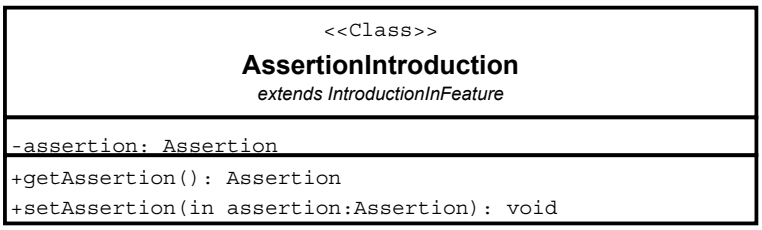

FIG. 29 – Introduction d'une assertion

Rôle : On veut ajouter une assertion (pré- ou post-condition) à une méthode d'une classe.

Description de la sémantique :

- Paramètres de l'adaptation
  - assertion : Assertion, assertion à introduire. Afin de pas risquer de créer des effets de side on introduira une copie de l'assertion.
  - targets : Collection de Method, méthodes cibles de l'introduction.
- Contraintes associées
  - @post : forAll target : Method in targets {target.contains(assertion.clone()) ;}
    nous utiliserons cette écriture pour décrire que toutes les méthodes (ou autres) contiennent un clone d'une entité nommée.
  - @post : les assertions ne sont pas redondantes.

– Description en pseudo-code

Algorithm 1 Introduction d'une assertion

```
Procedure execute () {
    forall m : Method in target.getMethods () {

        m.getAssertions().append(assertion.clone()) ;
        m.refactorAssertions () ; // remove redundants assertions
}
}
```

refactorAssertions permet de refactoriser les assertions, en retirant par exemple les redondances au niveau des assertions, ou encore en détectant un conflit entre deux assertions.

Exemple d'utilisation : Une utilisation possible est d'ajouter une précondition à une méthode afin de mieux la contraindre.

3.2.2 Concrétisation d'une méthode

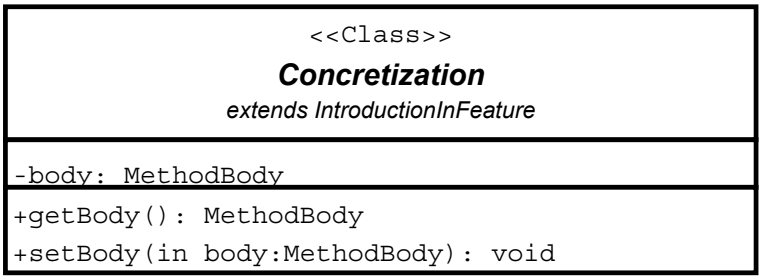

FIG. 30 – Concrétisation d'une méthode

Rôle : On veut concrétiser (ie donner le corps à) une méthode abstraite.

Description de la sémantique :
- Paramètres de l'adaptation
  - body : MethodBody, corps de la méthode. Ce ne sera pas ce corps qui sera directement affecté aux méthodes mais toujours une copie afin de ne pas créer d'effet de bord sur d'autres méthodes si ce corps venait à être modifié.
  - targets : Collection de méthodes auxquelles il faut donner un corps.
- Contraintes associées

- @pre : forAll target : Method in targets {target.isAbstract() = true}
- @pre : le type de retour du corps de la méthode doit être covariant avec celui de la signature.
- @pre : le corps de la méthode ne doit pas créer de conflit avec la signature de la méthode (nom et type des variables).
- @post : forAll target : Method in targets {target.getBody().equals(body)}
- @post : forAll target : Method in targets {target.isAbstract() = false}

– Description en pseudo-code

Algorithm 2 Concrétisation d'une méthode.

```
Procedure execute () {
    forall m : Method in target.getMethods () {
        m.setBody(body.clone()) ;
        m.setAbstract(false) ;
}
}
```

Exemple d'utilisation : On souhaite donner un corps à une méthode abstraite m.

3.2.3 Insertion d'un paramètre dans une signature

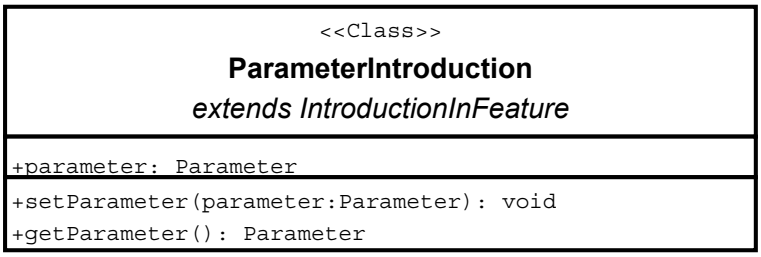

FIG. 31 – Introduction d'un nouveau paramètre

Rôle. On veut ajouter un paramètre à la signature d'une méthode existante d'un modèle.

Description de la sémantique

- Paramètres de l'adaptation
  - parameter : Parameter, paramètre à introduire. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie du paramètre et non le paramètre lui-même.

  - targets : Collection de Method, méthodes dans lesquelles il faut introduire le paramètre.
- Contraintes associées
  - @pre : forAll target : Method in targets { forall p : Parameter in target.getSignature().getParameters() {! p.getName.equals(parameter.getName()) ;}}
  - @post : forAll target : Method in targets { target.getSignature().getParameters().size()@post = target.getSignature().getParameters().size()@pre + 1 ;}
  - @post : forAll target : Method in targets { target.getSignature().getParameters().lastElement().equals(p
- Description en pseudo-code

Algorithm 3 Introduction d'un paramètre

```
Procedure execute () {
    forall m : Method in target.getMethods() {

        Parameter newParameter = parameter.clone() ;
        m.getSignature().getParameters().append(newParameter) ;
    }
}
```

Exemple d'utilisation. Une utilisation possible c'est d'adapter une méthode m1 d'un modèle M1 pour qu'elle devienne conforme à la méthode m2 d'un modèle M2.

3.2.4 Introduction d'un invariant de classe

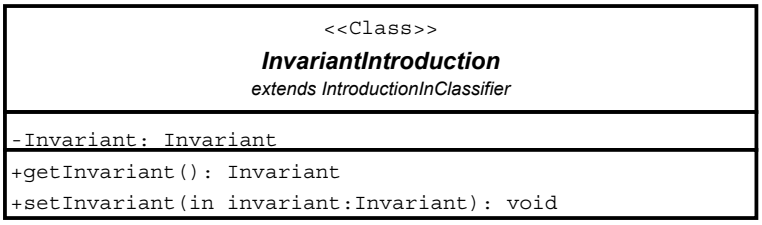

FIG. 32 – Introduction d'un invariant de classe.

Rôle : On veut introduire un invariant de classe dans une ou plusieurs classes d'un modèle.

Description de la sémantique
- Paramètres de l'adaptation
  - invariant : Invariant, l'invariant à introduire. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de l'invariant et non l'invariant lui-même.

  - targets : Collection de Class, classes cibles.
- Contraintes associées
  - @pre : L'invariant porte sur un ou plusieurs attributs des classes cibles.
  - @post : forall c : Class in targets {c.contains(invariant.clone()) ;}.
  - @post : Les invariants ne sont pas redondants.
- Description en pseudo-code

Algorithm 4 Introduction d'un invariant.

```
Procedure execute () {
    forall c : Class in target.getClasses () {

        c.getInvariants().append (invariant.clone()) ;
        // avoid side effects
        c.refactorInvariants() ; // remove redundants invariants.
}
}
```

Exemple d'utilisation : On veut rajouter une invariant portant sur un attribut a d'une classe c.

3.2.5 Introduction d'une méthode dans une classe

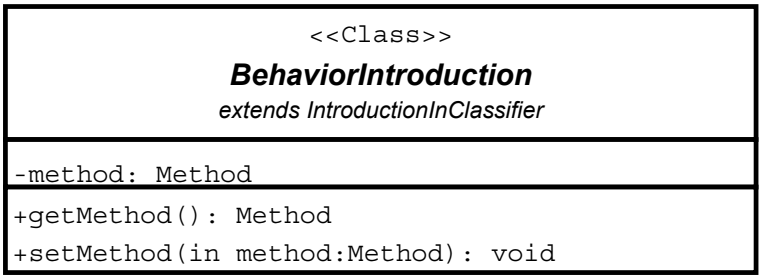

FIG. 33 – Introduction d'une méthode dans une classe

Rôle : On veut introduire une méthode dans une ou plusieurs classes d'un modèle.

Description de la sémantique
- Paramètres de l'adaptation
  - method : Method, méthode à introduire. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de la méthode et non la méthode elle-même.

  - targets : Collection de Class, classes cibles.
- Contraintes associées
  - @pre : forall target : Class in targets {forall m : Method in target.getMethods() {! m.getSignature().equals(method.getSignature()) ;}}
  - @post : forall c : Class in targets {c.contains(method.clone()) ;}
- Description en pseudo-code

Algorithm 5 Introduction de méthode dans une classe.

```
Procedure execute() {
    forall c : Class in target.getClasses() {

        // To avoid side effects we copy a clone of the method
        Method newMethod = method.clone() ;
        newMethod.setClass (c) ;
        c.getMethods().append(newMethod) ;
}
}
```

Exemple d'utilisation : Ajout d'une méthode m d'une classe c dans une classe c1.

3.2.6 Introduction d'un attribut dans une classe

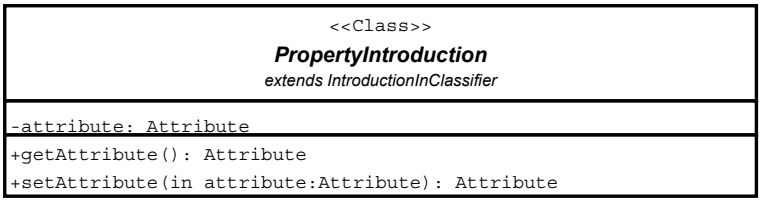

FIG. 34 – Introduction d'un attribut dans une classe

Rôle : On veut introduire un attribut dans une ou plusieurs classes d'un modèle.

Description de la sémantique
- Paramètres de l'adaptation
  - attribute : Attribute, attribut à introduire. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de l'attribut et non l'attribut lui-même.
  - targets : Collection de Class, classes cibles.
- Contraintes associées

- @pre : forall target : Class in targets {forall a : Attribute in target.getAttributes() {! a.getName().equals(attribute.getName()) ;}}
- @post : forall target : Class in targets {target.contains(attribute.clone()) ;}

– Description en pseudo-code

Algorithm 6 Introduction d'un attribut dans une classe

```
Procedure execute () {
    forall c : Class in target.getClasses() {

        Attribute newAttribute = attribute.clone() ;
        c.getAttributes().append(newAttribute) ;
        newAttribute.setClass(c) ;
}
}
```

Exemple d'utilisation : Introduction d'un attribut a dans une classe c.

3.2.7 Introduction d'une superclasse dans une classe

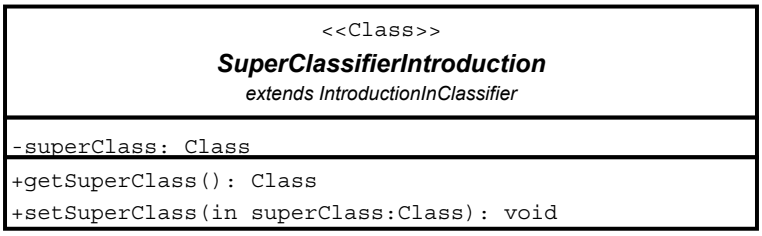

FIG. 35 – Introduction d'une superclasse dans une classe

Rôle : On veut introduire une superclasse dans une ou plusieurs classes d'un modèle.

Description de la sémantique

– Paramètres de l'adaptation
  - superclass : Class, qui qui deviendra superclasse des cibles. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de la classe et non la classe elle-même.
  - targets : Collection de Class, classes cibles.

– Contraintes associées
  - @pre : la classe cible ne doit pas descendre de superclass ou en être parente.

  - @post : forall c : Class in targets {c.getSuperClasses().contains(superclass) ;}
- Description en pseudo-code

Algorithm 7 Introduction d'une superclase dans une classe.

```
Procedure execute() {
    forall c : Class in target.getClasses() {
        c.getSuperClasses().append(superclass) ;
}
}
```

Exemple d'utilisation : Ajout d'une superclasse s dans une classe c.

3.2.8 Introduction d'une sousclasse dans une classe

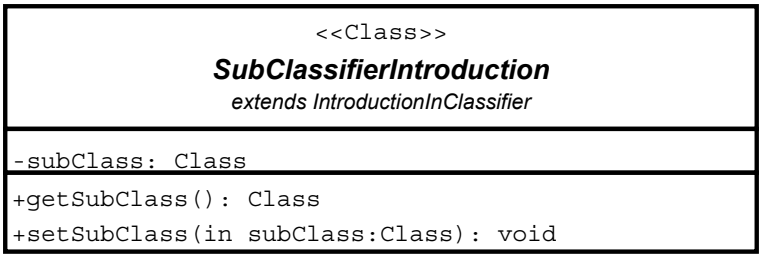

FIG. 36 – Introduction d'une sous-classe dans une classe

Rôle : On veut définir une classe comme sous-classe de classes d'un modèle.

Description de la sémantique
- Paramètres de l'adaptation
  - subclass : Class, classe qui deviendra sous-classe des cibles. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de la classe et non la classe elle-même.
  - targets : Collection de Class, classes cibles.
- Contraintes associées
  - @pre : la classe cible ne doit pas descendre de subclass ou en être parente.
  - @post : forall target : Class in targets {subClass.getSuperClasses().contains(target.clone()) ;}
- Description en pseudo-code

Algorithm 8 Introduction d'une sous-classe d'une classe.

```
Procedure execute() {

    forall target : TargetDeclaration in getTargets() {


        forall c : Class in target.getClasses() {

            // clone to avoid side effects
            subClassifier.getSuperClasses().append(c.clone()) ;
    }
}
```

Exemple d'utilisation : Définition d'une sous-classe s d'une classe c.

3.2.9 Introduction d'une classe dans un paquetage

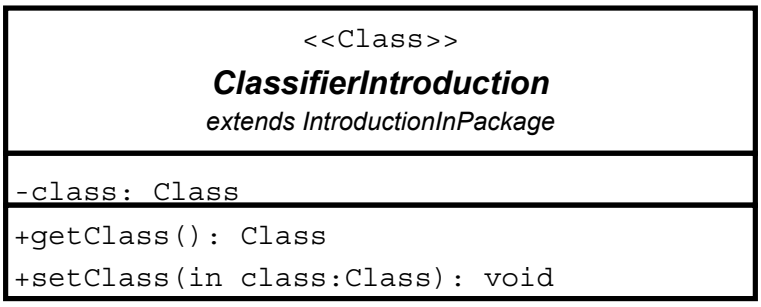

FIG. 37 – Introduction d'une classe dans un paquetage.

Rôle : On veut introduire une classe c dans un paquetage p.

Description de la sémantique

- Paramètres de l'adaptation
  - class : Class, classe à introduire. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de la classe et non la classe elle-même.
  - targets : Collection de Package, paquetages cibles.
- Contraintes associées
  - @pre : forall target : Package in targets {forall c : Class in target.getClasses() {! c.getName().equals(class.getName()) ;}}
  - @post : forall target : Package in targets {target.contains(class.clone()) ;}

– Description en pseudo-code

Algorithm 9 Introduction d'une classe dans un paquetage.

```
Procedure execute() {
    forall p : Package in target.getPackages() {
        Class classClone = class.clone() ;
        // clone to avoid side effects
        p.getClasses().append(classClone) ;
        classClone.setPackage (p) ;
}
}
```

Exemple d'utilisation : Ajouter une classe dans un paquetage donné.

3.2.10 Introduction d'un paquetage dans un paquetage

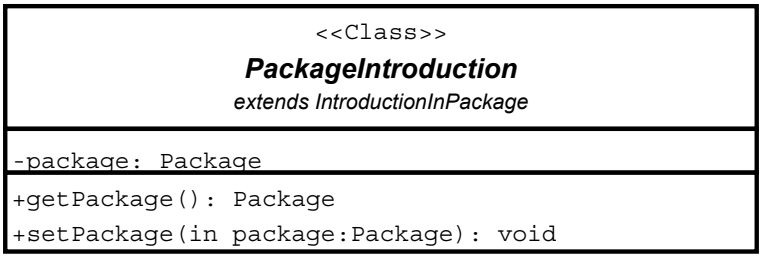

FIG. 38 – Introduction d'un paquetage dans un paquetage.

Rôle : On veut introduire un paquetage p1 dans un paquetage p0.

Description de la sémantique

– Paramètres de l'adaptation
  – package : Package, paquetage à introduire. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie du paquetage et non le paquetage lui-même.
  – targets : Collection de Package, les paquetages cibles.
– Contraintes associées
  – @pre : forall target : Package in targets {forall p : Package in target.getPackages() {! p.getName().equals(package.getName()) ;}}
  – @post : forall target : Package in targets {target.contains(package.clone()) ;}
– Description en pseudo-code

Algorithm 10 Introduction d'un paquetage dans un paquetage.

```
Procedure execute() {
    forall p : Package in target.getPackages() {
        Package packageClone = package.clone() ;
        // clone to avoid side effects
        p.getPackages().append(packageClone) ;
        packageClone.setPackage (p) ;
}
}
```

Exemple d'utilisation : Introduire un paquetage afin que celui-ci devienne sous-paquetage du paquetage courant.

3.2.11 Introduction d'une association dans un paquetage

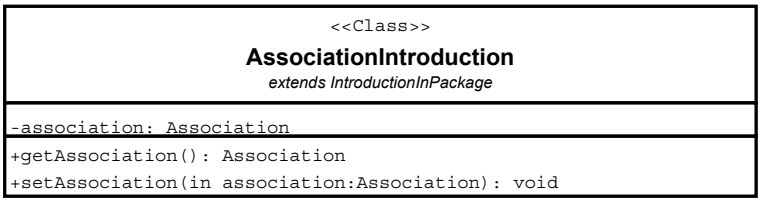

FIG. 39 – Introduction d'une association dans un paquetage.

Rôle : On veut introduire une association de classes dans un paquetage p.

Description de la sémantique

- Paramètres de l'adaptation
  - association : Association, association à introduire. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de l'association et non le paquetage lui-même.
  - targets : Collection de Package, les paquetages cibles.
- Contraintes associées
  - @pre : forall target : Package in targets {forall p : Package in target.getPackages() {forall asso : Association in p.getAssociations() {! asso.getName().equals(association.getName()) ;}}
  - @post : forall target : Package in targets {target.contains(association.clone()) ;}
- Description en pseudo-code

Algorithm 11 Introduction d'une association dans un paquetage.

```
Procedure execute() {
    forall p : Package in targets.getPackages() {

        Association associationClone = association.clone() ;
        // clone to avoid side effects
        p.getAssociations().append(associationClone) ;
        associationClone.setPackage (p) ;
}
}
```

Exemple d'utilisation : Introduire une association entre des classes dans un paquetage.

3.3 Les fusions

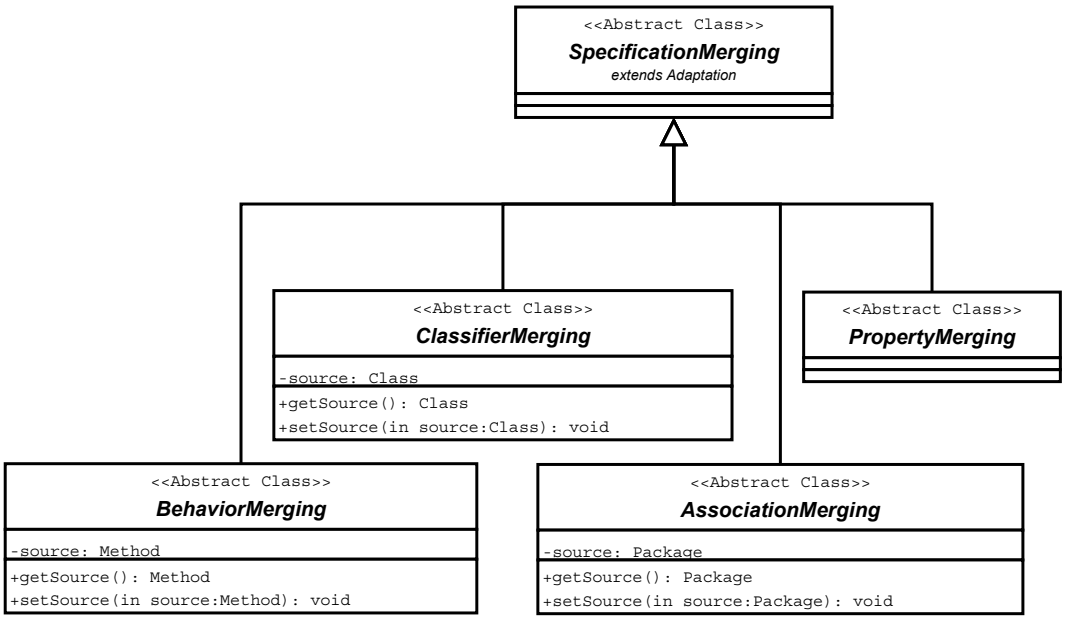

FIG. 40 – La fusion

3.3.1 Fusion d'une méthode concrète dans une méthode abstraite

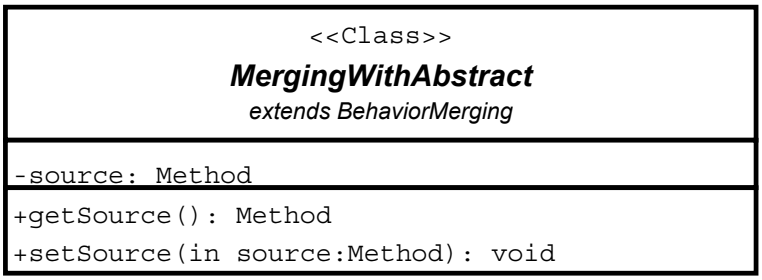

FIG. 41 – Fusion d'une méthode concrete dans une méthode abstraite

Rôle : On veut fusionner les deux méthodes, l'une étant abstraite et l'autre étant concrète. La fusion donne à la méthode abstraite le corps de la méthode concrète et lui retire son caractère abstrait.

Description de la sémantique

- Paramètres de l'adaptation
  - source : Method, méthode concrète dont on veut fusionner le corps. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de la méthode et non la méthode elle-même.
  - targets : Collection de Method, méthodes abstraites dans lesquelles on veut fusionner le corps de source.
- Contraintes associées
  - @pre : forall m : Method in targets {m.getMethodSignature().equals(source.getMethodSignature()) ;}
  - @pre : forall m : Method in targets {m.isAbstract() = true ; }
  - @post : forall m : Method in targets {m.isAbstract() = false ;}
  - @post : forall m : Method in targets {m.getBody().equals(source.getBody())) ;}
- Description en pseudo-code

Algorithm 12 Fusion du corps d'une méthode concrète dans une méthode abstraite.

```
Procedure execute () {
    forall m : Method in targets.getMethods () {
        // clone to avoid side effects
        m.setBody(source.getBody().clone()) ;
        m.setAbstract(false) ;
}
}
```

Exemple d'utilisation : On a une méthode abstraite qui est ensuite concrétisée dans plusieurs classes filles. On souhaite donner comme corps à la méthode abstraite le corps de l'une de ses méthodes redéfinies.

3.3.2 Fusion d'une méthode concrète dans une méthode concrète

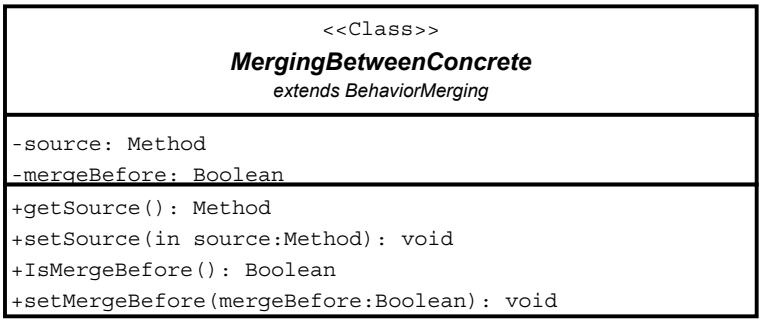

FIG. 42 – Fusion d'une méthode concrète dans une méthode concrète

Rôle : On veut fusionner les deux méthodes, toutes deux concrètes. La deuxième méthode recevra la fusion. Il est possible de paramétrer lequel des deux corps de méthodes sera exécuté en premier. Le type de retour sera toujours celui de la méthode cible, recevant la fusion. La visibilité de la méthode fusionnée sera la visibilité de méthode ayant la plus grande visibilité entre les deux méthodes à fusionner. Enfin, si les signatures des deux méthodes à fusionner diffèrent, les paramètres des deux méthodes seront concaténés, pour l'instant sans liaison entre ces paramètres.

Description de la sémantique

- Paramètres de l'adaptation
  - source : Method, méthode concrète. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de la méthode et non la méthode elle-même.
  - targets : Collection de Method, méthodes concrètes dans lesquelles on veut fusionner le corps de la méthode source.
  - mergeBefore : Boolean, si positionné à vrai le corps de source sera exécuté avant le corps de la méthode cible, sinon après.
- Contraintes associées
  - @pre : forall m : Method in targets {m.isAbstract() = false ; }
  - @post : forall m : Method in targets {if isMergeBefore() { m.getBeforeBodyMethodCalls().lastElement() equals(source.generateMethodCall()) ; } else { m.getAfterBodyMethodCalls().firstElement().equals(source.generateMethodCall()) ;}}
  - @post : forall m : Method in targets {m.getSignature().getParameters().size()@post =

m.getSignature().getParameters().size()@pre +
source.getSignature().getParameters().size() ;}

– Description en pseudo-code

Algorithm 13 Fusion du corps d'une méthode concrète dans une autre méthode concrète.

```
Procedure execute () {
    forall m : Method in targets.getMethods () {
        // first, clone m
        Method oldMethod = m.clone() ;
        // Then rename and mask old method
        new Renaming(m).execute() ; // auto rename
        new VisibilityModifying (m,"private").execute() ;
        // change the visibility of the method if necessary
        VisibilityModifier vmsource = source.getVisibilityModifier() ;
        if (vmsource.isLargerThan(m.getVisibilityModifier()){

            new VisibilityModifying (m, source.getVisibilityModifier()).execute() ;
        }
        // create new method call
        MethodCall mcall = new MethodCall() ;
        mcall.setMethodName(oldMethod.getName()) ;
        // introduce copies of source parameters into m
        forall p : Parameter in source.getSignature().getParameter() {

            Parameter newParam = p.clone() ;
            Parameter binded = getParameterBindedWith (p) ;
            if (binded == null) { // no parameter is binded with p

              if (source.getSignature().containsParameterNamed(p.getName())

                // rename the parameter if necessary
                newParam.setName (m.getName() + newParam.getName()) ;
              m.getSignature().getParameters().append(newParam) ;
              // add the parameter to the method call
              mcall.getParameters().append(newParam) ;
            }
            else { // p is binded with binded

              // add binded to the method call
              mcall.getParameters().append(binded) ;
            }
        }
        // Introduce the method call before or after the body
        if (isMergeBefore ()) {

            // introduce at the last position of method calls before
            m.getBeforeBodyMethodCalls().append(mCall) ;
        } else {
        // introduce at the first position of the method calls after
```

Exemple d'utilisation : On a une méthode qui initalise une collection et une autre qui affiche son contenu, on veut les fusionner pour ne faire plus qu'une méthode.

3.3.3 Fusion par simple ajout de constituants.

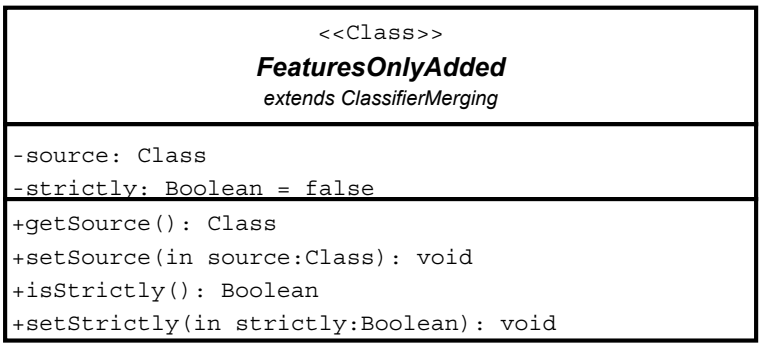

FIG. 43 – Fusion par simple ajout de constituants.

Rôle : On veut ajouter dans les classes définies par target tous les constituants (invariants, attributs, méthodes) de la classe source. Un paramètre strictly permet de spécifier le comportement de cette fusion s'il est impossible d'ajouter un élément de la classe source dans la classe target (conflit de nom par exemple).

Description de la sémantique

- Paramètres de l'adaptation
  - source : Class, classe contenant les constituants à ajouter. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie des constituants et non les constituants eux-mêmes.
  - target : Collection de classes dans lesquelles on veut ajouter les constituants.
  - strictly : Boolean, si positionné à vrai une erreur sera signalée s'il est impossible d'ajouter un élément donné.
- Contraintes associées
  - @pre : forall c : Class in targets {c.isInterface() == false; }
  - @post : forall c : Class in targets {forall f : Features in sources.getFeatures() {c.contains(f.clone() ;}}
- Description en pseudo-code

Algorithm 14 Ajout des constituants d'une classe dans d'autres classes.

```
Procedure execute () {
    forall c : Class in targets.getClasses() {
        // add invariants
        forAll i : Invariant in source.getInvariants() {

            if (c.contains(i) && isStrictly())


              error"invariants already existing";
            else

              c.getInvariants().append(i.clone()) ;
        }
        // add attributes
        forAll a : Attribute in source.getAttributes() {

            if (c.contains(a) && isStrictly())


              error"attribute already existing";
            else

              c.getAttributes().append(a.clone()) ;
        }
        // add methods
        forAll m : Method in source.getMethods() {

            if (c.contains(m) && isStrictly())


              error"method already existing";
            else

              c.getMethods().append(m.clone()) ;
        }
}
}
```

Exemple d'utilisation : Une utilisation possible serait par exemple l'aplatissement d'une hiérarchie de classe afin qu'une seule classe contienne tous les éléments de ses classes parentes.

3.3.4 Fusion personnalisée de deux classes

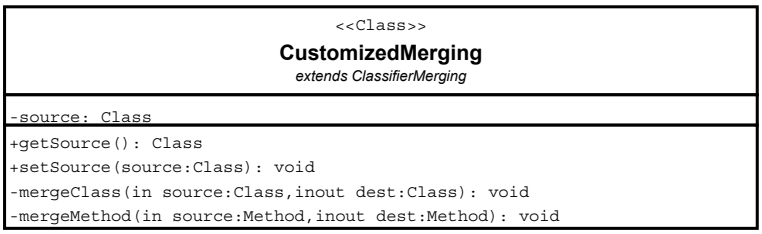

FIG. 44 – Fusion personnalisée de deux classes

Rôle On veut fusionner plusieurs classes d'un modèle dans une classe d'un modèle.

Description de la sémantique

- Paramètres de l'adaptation
  - source : Class, classe source de la fusion.
  - targets : Collection de Class, classes cibles de la fusion.
- Contraintes associées
- Description en pseudo-code

Algorithm 15 Fusion de classes

```
Procedure execute(){

    forall classtarget : Class in targets.getClasses () {

        mergeClass (source, classtarget) ;
    }
}
Procedure mergeClass (source, dest : Class) {
    // Merge on inheritance hierarchy
    forall c : Class in source.getSuperClasses() {

        if (c not in dest.getSuperClasses())
            dest.getSuperClasses().append(c) ;
    }
    // add invariants not existing in dest
    forAll i : Invariant in source.getInvariants() {

        if ( ! dest.contains(i))

            c.getInvariants().append(i.clone()) ;
    }
    // Methods' merge
    forall m : Method in source.getMethods () {

        Method bindedMethod = getMethodBindedWith(m) ;
        if (bindedMethod != null) {
            mergeMethods (m, bindedMethod) ;
        }
        else {

            Method oldMethod = dest.getMethod (m.getSignature ()) ;
            if (oldMethod != null)
            // if method already exists in dest, merge source in dest

              mergeMethods (m, oldMethod) ;
            else { // the method doesn't exist and is not binded, simply add it

              Method mDuplicata = m.clone() ;
              mDuplicata.setClass(dest) ;
              dest.getMethods().append(mDuplicata) ;
            }
        }
    }
    // Attribute's merge
    forall a : Attribute in source.getAttributes() {


        Attribute oldAttribute = dest.getAttribute(a.getSignature()) ;
```

Exemple d'utilisation Une utilisation possible est par exemple de fusionner tous les éléments d'une classe C1, décrivant une vue d'un procédé, d'un modèle M1 dans une classe C2, décrivant une seconde vue du même procédé, d'un modèle M2, afin que C2 contiennent tous les élèments de C1, formant une vue enrichie.

3.3.5 Fusion d'associations.

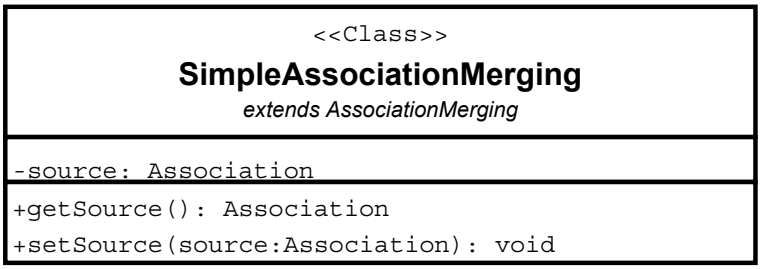

FIG. 45 – Fusion d'associations.

Rôle : On veut fusionner deux associations. Le nom de la nouvelle association sera un nouveau nom, unique et elle recevra les extrémités de l'associations source. Si une extrémité de la source est déjà présente dans l'association cible alors les multiplicités de cette extrémités seront éventuellement étendues.

Description de la sémantique

- Paramètres de l'adaptation
  - source : Association, association qource de la fusion.
  - target : Collection d'Association, les associations qui recevront la fusion.
- Contraintes associées
  - @pre :
  - @post : Toutes les associations de targets sont enrichies à partir du contenu de l'association source.
- Description en pseudo-code

Algorithm 16 Fusion d'associations.

```
Procedure execute () {

    forall a : Association in targets.getAssociations() {


        forAll end : AssociationEnd in source.getAssociationEnds() {

            boolean found = false ;
            forall ae : AssociationEnd in a.getAssociationEnds() {


              // if an association end pointing to the same class
              // is found, merge both of two.
              if (ae.getClass().equals(end.getClass())){

                mergeEnd (end, ae) ;
                found = true
              }
            }
            // if no end found, simply add it.
            if ( ! found)

              a.getAssociationEnds().append(end.clone()) ;
        }
    }
}
Procedure mergeEnd (source, target : AssociationEnd) {


    // if one of the two ends is ordered, maintain the order
    target.setOrdered(source.isOrdered()) ;
    // set the multiplicityMin to the min of the multiplicityMin of the two ends
    target.setMultiplicityMin(min (source.getMutiplicityMin(), target.getMultiplicityMi
    // set the multiplicityMax to the max of the multiplicityMax of the two ends
    // or -1 for +inf
    if (target.getMultiplicityMax()== -1 || source.getMultiplicityMax() == -1)
        target.setMultiplicityMax(-1) ;
    else {

        target.setMultiplicityMax(max (source.getMutiplicityMax(), target.getMultiplici
    }
}
```

Exemple d'utilisation : Une utilisation possible serait par exemple la spécialisation d'associations, ce qui est possible en fusionnant deux associations.

3.4 Modifications

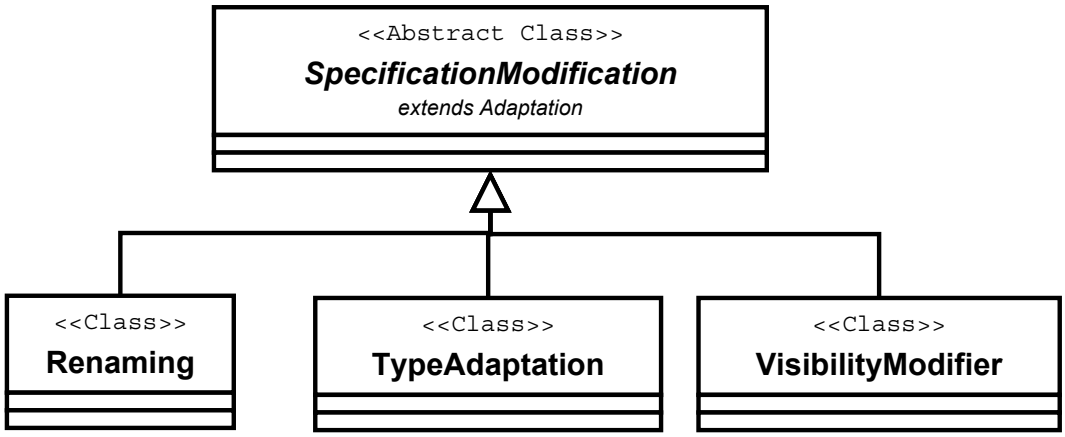

FIG. 46 – La modification

On peut aussi avoir besoin de modifier le nom d'une méthode, le type d'un attribut ou sa visibilité. Cette catégorie d'adaption répond à cette demande.

3.4.1 Renommage

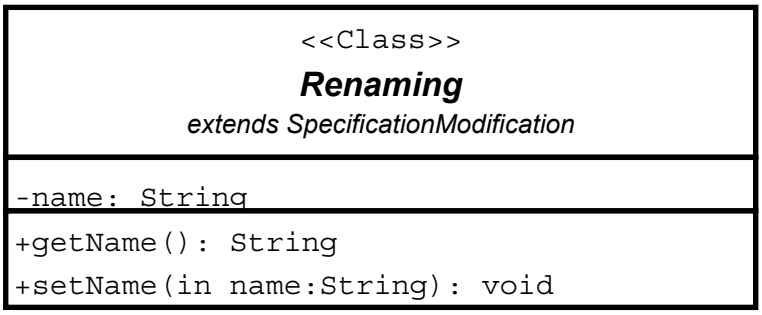

FIG. 47 – Renommage

Rôle : On veut renommer l'élément.

Description de la sémantique

- Paramètres de l'adaptation
  - targets : Collection de Feature, éléments à renommer.
  - name : String, nouveau nom.
- Contraintes associées
  - @pre : forall f : Features in targets {(f.isClass() || f.isAttribute() || f.isMethod()) = true;}
  - @pre : le nom choisi ne doit pas créer de conflit.
  - @post : forall f : Features in targets {f.getName().equals(name);}
- Description en pseudo-code

Algorithm 17 Renommage des éléments.

```
Procedure execute () {
    forall f : Feature in targets.getFeatures () {

        f.setName(name);
    }
}
```

Exemple d'utilisation : On veut renommer toutes les classes d'un paquetage en rajoutant un prefixe devant.

3.4.2 Changement du typage

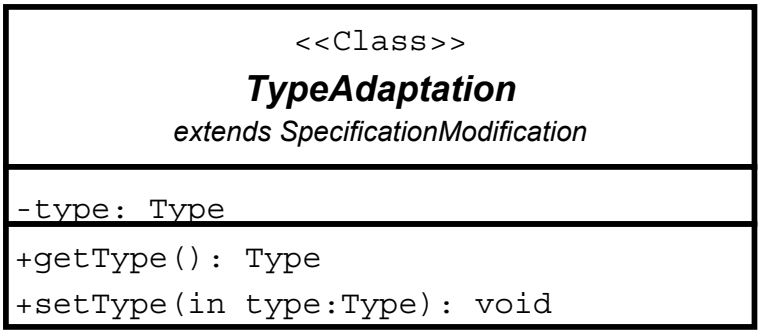

FIG. 48 – Changement du typage

Rôle : On veut changer le type d'une méthode ou d'un attribut.

Description de la sémantique

- Paramètres de l'adaptation

  - targets : Collection de Feature, éléments dont on veut changer le type.
  - type : Type, nouveau type. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie du type et non le type lui-même.
- Contraintes associées
  - @pre : forall f : Features in targets {(f.isAttribute() || f.isMethod()) = true ;}
  - @post : forall f : Features in targets {f.getType().equals(type) ;}
- Description en pseudo-code

Algorithm 18 Changement de type des éléments.

```
Procedure execute () {
    forall f : Feature in targets.getFeatures () {


        // to avoid side effects we always affect a
        // copy of the type
        f.setType(type.clone()) ;
    }
}
```

Exemple d'utilisation : On veut changer le type de retour d'une méthode par exemple en le spécifiant d'avantage.

3.4.3 Modification de la visibilité d'un élément

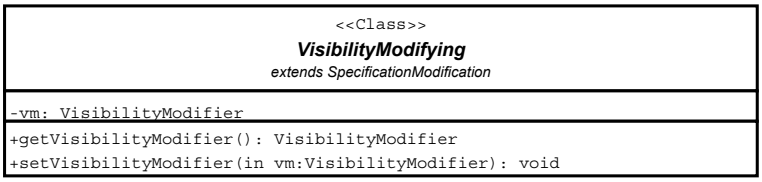

FIG. 49 – Modification de la visibilité d'un élément

Rôle : On veut changer la visibilité d'une classe, d'une méthode ou d'un attribut.

Description de la sémantique

- Paramètres de l'adaptation
  - targets : Collection de Features, éléments dont on veut changer la visibilité.

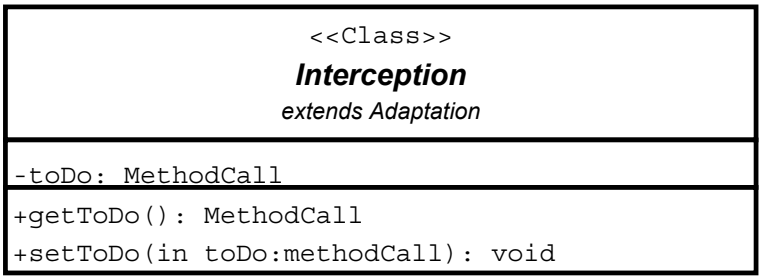

FIG. 50 – Classe Interception

- vm : VisibilityModifier, nouvelle visibilité parmi public, private, protected. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de la visibilité et non la visibilité elle-même.

– Contraintes associées
  – @pre : forall f : Features in targets {(f.isClass() || f.isAttribute() || f.isMethod()) = true;}
  – @post : forall f : Feature in targets {f.getVisibilityModifier().equals(vm);}
– Description en pseudo-code

Algorithm 19 Modification de la visibilité d'un élément.

```
Procedure execute () {
    forall f : Feature in targets.getFeatures () {


        // to avoid side effects we always affect a
        // copy of the visibility modifier
        f.setVisibilityModifier(vm.clone());
    }
}
```

Exemple d'utilisation : On veut masquer une méthode en la rendant privée alors qu'elle était publique.

3.5 Interceptions

Une interception ajoute un comportement à la méthode, à effectuer soit avant le corps de la méthode (BeforeMethod), soit après le corps de la méthode (AfterMethod), soit à la place du corps de la méthode (AroundMethod), soit à la levée d'une exception (OnException). Ce comportement ajouté se représente sous forme d'appel à des méthodes (MethodCall) et la classe Method est pourvue

de listes de MethodCall permettant d'ordonner les méthodes à appeller avant le corps de la méthode, après, ou encore à la levée d'exceptions.

3.5.1 Interception avant une méthode

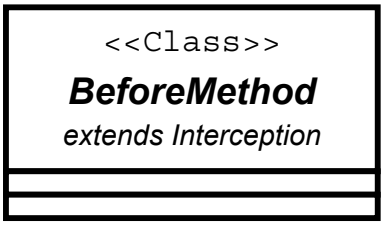

FIG. 51 – Interception avant une méthode

Rôle : On veut changer le comportement d'une méthode, en ajoutant un appel de méthode à exécuter avant le corps de la méthode.

Description de la sémantique

- Paramètres de l'adaptation
  - targets : Collection de Method, méthodes à intercepter.
  - toDo : MethodCall, appel à la méthode a exécuter avant le corps de la méthode. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de l'appel et non l'appel lui-même.
- Contraintes associées
  - @pre : l'appel à la méthode désigné par toDo doit être valide et conforme à la signature de la méthode.
  - @post : forall target : Method in targets {target.getBeforeBodyMethodCalls().firstElement().equals(toD
  - @post : forall target : Method in targets {target.getBeforeBodyMethodCalls().size()@post == target.getBeforeBodyMethodCalls().size()@pre +1 ;}
- Description en pseudo-code

Algorithm 20 Interception avant une méthode.

```
Procedure execute () {
    forall m : Method in targets.getMethods () {

        // add a copy of the method call in first position of the list
        // of method calls to call before proceeding the method body.
        target.getBeforeBodyMethodCalls().insert(toDo.clone(),0) ;
    }
}
```

Exemple d'utilisation : On veut par exemple tracer tous les accès à une méthode particulière.

3.5.2 Interception après une méthode

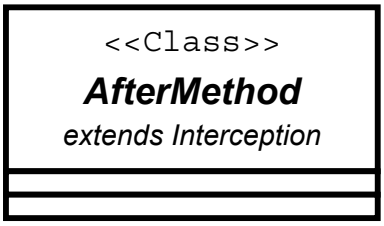

FIG. 52 – Interception après une méthode

Rôle : On veut changer le comportement d'une méthode, en ajoutant un appel de méthode à exécuter après de la méthode.

Description de la sémantique

- Paramètres de l'adaptation
  - targets : Collection de Method, méthodes à intercepter.
  - toDo : MethodCall, appel à la méthode a exécuter après le corps de la méthode.Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de l'appel et non l'appel lui-même.
- Contraintes associées
  - @pre : l'appel à la méthode désigné par toDo doit être valide et conforme à la signature de la méthode.
  - @post : forall target : Method in targets {target.getBeforeBodyMethodCalls(). lastElement().equals(toDo) ; }
  - @post : forall target : Method in targets {target.getBeforeBodyMethodCalls().size()@post = target.getBeforeBodyMethodCalls().size()@pre +1 ;}
- Description en pseudo-code

Algorithm 21 Interception après une méthode.

```
Procedure execute () {
    forall m : Method in target.getMethods () {

        // add a copy of the method call in last position of the list
        // of method calls to call after proceeding the method body.
        target.getAfterBodyMethodCalls().append(toDo.clone()) ;
    }
}
```

Exemple d'utilisation : On veut par exemple tracer tous les accès à une méthode particulière.

3.5.3 Interception autour d'une méthode

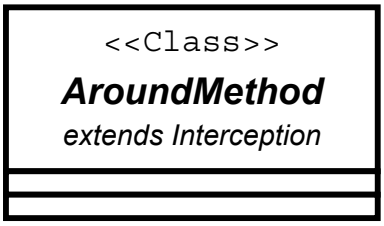

FIG. 53 – Interception autour d'une méthode

Rôle : On veut changer le comportement d'une méthode, en exécutant une méthode donnée à la place du corps actuel de la méthode.

Description de la sémantique

- Paramètres de l'adaptation
  - targets : Collection de Method, méthodes à intercepter.
  - toDo : MethodCall, appel complet de la méthode à appeller pour remplacer le corps de la méthode. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de l'appel et non l'appel lui-même.
- Contraintes associées
  - @pre : l'appel à la méthode désigné par toDo doit être valide et conforme à la signature de la méthode.
  - @post : forall target : Method in targets {target.getBeforeBodyMethodCalls().lastElement().equals(toD
  - @post : forall target : Method in targets {target.getBeforeBodyMethodCalls().size()@post = target.getBeforeBodyMethodCalls().size()@pre +1 ;}
  - @post : forall target : Method in targets {target.getProceedBody() = false ;}
- Description en pseudo-code

Algorithm 22 Interception autour d'une méthode.

```
Procedure execute () {
    forall m : Method in targets.getMethods () {

        // add a copy of the method call instead of the method body at the
        // last position of the list of method calls to call before the
        // method body and set target.proceedBody to false .
        target.getAfterBodyMethodCalls().append(toDo.clone()) ;
        target.setProceedBody(false) ; // the old body must not be performed
    }
}
```

Exemple d'utilisation : On veut par exemple annuler toutes les modifications d'une variable en modifiant les corps des modificateurs de cette variable.

3.5.4 Interception sur levée d'exception

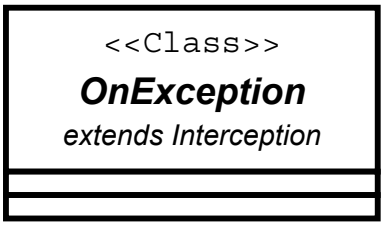

FIG. 54 – Interception autour d'une méthode

Rôle : On veut changer le comportement d'une méthode, en indiquant le comportement lors d'une levée d'exception.

Description de la sémantique

- Paramètres de l'adaptation
  - targets : Collection de méthodes à intercepter.
  - toDo : MethodCall, appel complet de la méthode à appeller en cas de levée d'exception. Afin d'éviter tout risque d'effet de bord ultérieur nous affecterons une copie de l'appel et non l'appel lui-même.
- Contraintes associées
  - @pre : l'appel à la méthode désigné par toDo doit être valide et conforme à la signature de la méthode.
  - @post : forall target : Method in targets {target.getOnExceptionMethodCalls().lastElement().equals(toI

- @post : forall target : Method in targets
  {target.getOnExceptionMethodCalls().size()@post ==
  target.getOnExceptionMethodMethodCalls().size()@pre +1 ;}

– Description en pseudo-code

Algorithm 23 Interception sur levée d'une exception dans une méthode.

```
Procedure execute () {
    forall m : Method in targets.getMethods () {

        // add a copy of the method call in last position of the list
        // of method calls to call when the method body throws an exception.
        target.getOnExceptionMethodCalls().append(toDo.clone()) ;
    }
}
```

Exemple d'utilisation : On veut par exemple tracer tous les exceptions levées dans une méthode particulière.

4 L'expression des cibles

Expression simple de cible Une méthode peut simplement être exprimée avec son nom complet et ses paramètres :

"p.c.m (int n)"

Un attribut peut être identifié par son nom complet.

"p.c.a"

Un paquetage enfin sera désigné par son nom complet, les sous-paquetages étant séparés par des '.'.

Une association, unique dans un espace de nommage, ne sera désigné que par son nom.

Cette notation permet d'expliciter clairement une cible unique et connue.

Il est aussi permi de définir plusieurs cibles avec une seule TargetExpression. Auquel cas il suffira de séparer la définition des cibles par une virgule.

Expression complexe de cibles Le metacaractère '*' remplace n'importe quelle chaîne de caractère jusqu'à un espace, une parenthèse ou fin de ligne.

'?' ne remplace qu'un seul caractère, non parenthèse, espace ou fin de ligne.

[a-z] exprime un caractère contenu dans l'intervalle indiqué.

[a-z]#n avec n entier exprime une chaîne constituée de n caractère successifs exactement contenus dans l'intervalle indiqué.

[a-z]#* indique un nombre indéfini de caractères, 0 étant le nombre minimum.

[a-z]#+ indique un nombre indéfini de caractères, 1 étant le nombre minimum.

{x} indique que le motif entre accolades est facultatif.

les caractères '[', ']', '{', '}', '#' et '\' sont exprimés en les faisant précéder d'un '\'.

Par exemple pour indiquer toutes les classes contenues dans un paquetage "accessoires" et ses sous paquetages et commençant par la lettre 'C' on notera :

« *accessoires.{*.}C[a-zA-Z0-9]#** »

ou bien pour récupérer tous les accesseurs et modificateurs publics (*get* et *set*) de toutes les classes d'un paquetage « accessoires » on pourra écrire :

« public * accessoires.[A-Z][a-zA-Z0-9]#*. ?et*(*#*) »

5 Langage de composition

Ce langage dédié permet soit de spécifier un protocole de composition, soit de définir une composition.

Une préoccupation sera toujours définie avec les déclarations de :

1) La préoccupation, en la nommant.

2) Des compositions mises en uvre (s'il s'agit d'une composition).

3) De l'adaptateur, déclarant lui-même :

a. Ses cibles

b. Ses contraintes de cibles

c. Ses correspondances

d. Ses contraintes de correspondance

e. Ses adaptations

1) La ligne 01 (figures 1 et 2) permet de déclarer une préoccupation (concern), en la nommant.

concern concern_ name

Exemple :

concern example.arithmetical_ expr_ display

2) La ligne 02 de la figure 2 commençant par compose permet d'indiquer qu'il s'agit d'une composition. Les deux préoccupations à composer sont indiquées sous la forme suivante :

compose concern1 with concern2

Exemple :

compose example.arithmetical_ exp_ evaluation with example.arithmetical_ exp_ description

A noter que le caractère invasif (*in situ*) ou *ex situ* se définit aussi à ce niveau là. Si la préoccupation cible, concern2 donc, porte le même nom que la préoccupation courante, alors la composition à lieu *in situ*, sinon elle a lieu *ex situ*.

3) Vient ensuite la déclaration et la définition de l'adaptateur par le mot-clé *adapter*.

Si l'adaptateur est réutilisable (*protocole de composition*) alors il sera déclaré abstrait (figure 1 ligne 02).

S'il s'agit de la concrétisation d'une composition, l'adaptateur sera concret (figure 2 ligne 03) et le mot-clé *extends* permettra de définir quel est le super-adaptateur. Le super-adaptateur est unique car les adaptateurs ne supportent que l'héritage simple.

Une accolade ouvrante suivant le nom de l'adaptateur permettra d'indiquer la portée de l'adaptateur. La syntaxe est donc :

abstract adapter adapterName {

| adapter adapterName extends superAdapterName {

Exemples :

abstract adapter DisplayProtocolAdapter {

adapter DescriptionAndEvaluation extends EvaluationProtocolAdapter {

3.a) Un adaptateur peut contenir un ensemble de cibles (*target*), pouvant être abstraites ou concrètes (lignes 04, 05 et 07 24 et 04 à 08 25). Ces cibles peuvent spécifier des classes (*Class*), ou des méthodes (*Method*).

Une déclaration de cible suivra deux syntaxes différentes suivant si la cible est abstraite ou si elle est concrète, redéfinissant une cible abstraite.

Une cible abstraite commence par *abstract* puis le type (*Class, Method, Attribute, Association* ou *Package*) suivi du mot-clé *target*. Suit une description (facultative mais recommandée) du *rôle* de la cible, entre guillemets puis ' :' et le nom de la cible.

abstract Package|Class|Method|Attribute|Association target "target role" : targetName

Une cible concrète commence par le mot-clé *target*, suivi du nom de la cible concrétisée (donc présente dans l'adaptateur abstrait étant étendu), puis "=' puis la valeur de la cible.

target classTargetName = class_path

| target methodTargetName = method_signature

| target attributeTargetName = attribute_name

| target packageTargetName = package_name

Par exemple :

abstract Class target "classes used as a Leaf" : leafClass

abstract Method target "method verifying if a litteral is valid" : isValid

target numericClass = example.arithmetical_exp_description.NumericLitteral

target isValid = public boolean isValid () in example.arithmetical_exp_desc.NumericLitteral

3.b) Des contraintes peuvent être associées aux cibles abstraites de façons suivantes :

require C1 inherits from C0

require assertion As (not) valid with M

require invariant I (not) valid with C

require C (not) contains (abstract) method M

require C (not) contains attribute A

require P (not) contains (abstract) class C

require P (not) contains interface C

Exemples :

require leafClass inherits from nodeClass
require leafClass not contains method public void isALeaf()
require leafClass contains attribute value

Pour une méthode, si on explicite la cible sans utiliser de *target*, il faut donner sa signature exacte à cause de la surcharge, pour un attribut son simple nom suffit.

3.c) Les correspondances sont aussi définies au niveau de l'adaptateur. Elles portent sur un type (Attribute, Method, Class, Association) et peuvent aussi être déclarées abstraites. Elles portent un nom, et il est possible de leur attribuer un rôle, dans le but de faciliter leur réutilisation.

Voici un exemple de déclaration de correspondance :

abstract Attribute binding " attribute(s) matching in Camera Class with other ones in another Class " : cameraAttributeBinding {

La correspondance a ensuite besoin des masques à appliquer sur les modèles source et cible.

Le masque à appliquer sur le modèle source commence par la déclaration *source-model {* suivi des valeurs transmises pour chaque propriété de l'élément, puis de la même chose pour le modèle cible (*target-model*). On peut spécifier une valeur pour la visibilité de l'élément, son nom, son type, sa classe, son paquetage ou ses paramètres. Ces valeurs peuvent être soit fixées, soit libres. Si elles sont libres on peut encore indiquer si on recherche l'égalité dans la correspondance sur cette partie d'élément, ou si la valeur prise n'importe pas. Dans l'exemple ci-dessous ces différentes possibilités sont présentées :

abstract Attribute binding " attribute(s) matching in Camera Class with other ones in another Class " : cameraAttributeBinding {
source-model {
visibility : =
*class-element : **
*type : **
name : "description"
class : "Camera"
package : "example.business_subject"
parameters : ""
}
target-model {
visibility : =
*class-element : **
*type : **
name : "desc"
class : "AppareilPhoto"
package : "example.technical_subject"
parameters : ""
}

Le caractère * permet de laisser totalement libre la valeur sur cette propriété de l'élément.

Le caractère = indique que l'égalité sur la propriété de l'élément impactée est nécessaire, quelque soit la valeur.

Enfin les autres valeurs entre guillemets sont des valeurs fixes.

Dans cet exemple, les attributs *description* de la classe *example.business_subject.Camera* et *desc* de *example.technical_subject.AppareilPhoto*, ayant la même visibilité sont mis en correspondance.

3.d) Comme avec les target, les correspondaces peuvent être contraintes. Ces contraintes portent sur la satisfaction (ou non) d'une correspondance et sur la multiplicité de cette dernière. Les contraintes sont exprimées de la manière suivante :

require binding_name constraint_type

avec *constraint_type* pouvant prendre comme valeurs :

1to1

nto1

1ton

isVerified

isNotVerified

3.e) Sont ensuite définies les adaptations, elles aussi, concrètes ou abstraites. Une adaptation possède un nom unique dans l'adaptateur et utilise un des opérateurs de composition (voir syntaxe des opérateurs plus bas). Seul un adaptateur abstrait peut contenir des adaptations abstraites, et un adaptateur concret héritant d'un adaptateur abstrait doit redéfinir toutes ses adaptations abstraites.

La déclaration d'une adaptation commence par le mot-clé adaptation, suivi de son nom puis de son rôle entre guillemet, facultatif mais recommandé (surtout pour les adaptateurs réutilisables). Les ' :' en fin de ligne indiquent le début de l'adaptation.

adaptation becomeNode "Modify class to make it a node" :

extend class nodeClass with ASTNode

abstract adaptation NodeUpdate "Concretize method displayType in Node class" :

concretize method public void displayType() in nodeClass

Enfin l'adaptateur se termine en fermant l'accolade ouverte lors de sa déclaration.

Voici la syntaxe de l'intégralité des opérateurs actuellement disponibles :

add descendant C1, C2, ... to C0

add features from C1, C2, ... to C3, C4,

add parent C0 to C1, C2, ...

concretize method M in C (with assertion As1, As2,)

extends class C1, C2, with C0

intercept after M1, M2, ... in C (with assertion As1, As2,)

intercept around M1, M2, ... in C (with assertion As1, As2,)

intercept before M1, M2, ... in C (with assertion As1, As2,)

intercept onException M1, M2, ... in C (with assertion As1, As2,)

introduce attribute A1, A2, ... into C

introduce method M1, M2, ... into C

introduce assertion As1, As2, ... into M
introduce invariant I1, I2, ... into C
introduce parameter Pr1, Pr2, ... into M
introduce class C1, C2, into Pa
introduce association Asso1, Asso2, into Pa
introduce package Pa1, Pa2, ... into Pa
merge abstract method M in C with M1
merge concrete method M in C with M1
merge association Asso1 with Asso2
Où :
C représente une classe,
M, une méthode
Pr, un paramètre
Pa, un paquetage
A, un attribut
As, une assertion
T un type
I, un invariant
Asso, une association

Toutes les cibles peuvent être explicitées directement, soit utiliser via une variable *target*.

Ce qui est entre parenthèses peut ne pas être spécifié dans le cadre d'une adaptation abstraite, mais devra l'être dans le cadre d'une adaptation concrétisée.

Et voici maintenant deux exemples de préoccupations complètement décrites. La première (24) décrit une préoccupation réutilisable avec son protocole de composition. Le second (25) concrétise une composition de deux préoccupations.

Algorithm 24 Le protocole de composition associé à l'affichage d'un arbre.

```
01 concern example.arithmetical_expr_display
02 abstract adapter DisplayProtocolAdapter {
04 abstract Class target "class(es) being used as a tree node " : nodeClass
05 abstract Class target " class(es) used as a leaf " : leafClass
06 require leafClass inherits from nodeClass
07 abstract Class target " class(es) used as a composite " : compositeClass
08 require compositeClass inherits from nodeClass
09
10 adaptation becomeNode "Modify class to make it a node " :
11 extend class nodeClass with ASTNode
12 adaptation becomeLeaf "Modify class to make it a leaf " :
13 extend class leafClass with ASTLeaf
14 adaptation becomeComposite "Modify class to make it a composite " :
15 extend class compositeClass with ASTComposite
16
18 abstract adaptation NodeUpdate "Concretize method displayType in Node
class " :
19 concretize method public void displayType() in nodeClass
20 abstract adaptation LeafUpdate "Concretize method displayValue in Leaf
class " :
21 concretize method public void displayValue() in leafClass
22 abstract adaptation CompositeUpdate "Concretize method children in Com-
posite class " :
23 concretize method public ASTNode[] children() in compositeClass
24 }
```

Algorithm 25 La composition de la description et de l'évaluation des expressions arithmétiques.

```
01 concern example.arithmetical_exp_composition
02 compose example.arithmetical_exp_evaluation with example.arithmetical_exp_description
03 adapter DescriptionAndEvaluation extends EvaluationProtocolAdapter {
04 target expressionClass = example.arithmetical_exp_description.Expression
05 target numericClass = example.arithmetical_exp_description.NumericLitteral
06 target binaryOpClass = example.arithmetical_exp_description.BinaryExpression
07 target unaryOpClass = example.arithmetical_exp_description.UnaryExpression
07 target unaryOpClass = example.arithmetical_exp_description.UnaryExpression
08
09 adaptation numericUpdate :
10 concretize method public Integer value() in numericClass with
11 post result = numericClass.getValue()
12 adaptation binaryDescendant :
13 add descendant Plus, Minus to binaryOpClass
14 adaptation unaryDescendant :
15 add descendant UnaryMinus to unaryOpClass
16 }
```