Université de Mons-Hainaut
Faculté des Sciences
Institut informatique

Etude et automatisation de refactorings pour le langage Delphi

Mémoire réalisé par Dahmane Sélim
dans le service de génie logiciel

Année académique 2004-2005

Directeur :	**Co-directeur :**	**Rapporteurs :**
Mens Tom	Real Jean-Christophe	Bruyère Véronique
		Delgrange Olivier

Remerciements

Pour commencer, je voudrais remercier plusieurs personnes qui m'ont permis de mener ce mémoire à bien :

Je voudrais remercier Mr Mens pour m'avoir proposé le sujet de ce mémoire, pour m'avoir permis de le réaliser dans son service, pour avoir été disponible durant toute l'année et pour ses relectures et remarques.

Je tiens aussi à remercier Mr Real pour avoir proposé conjointement le sujet avec Mr Mens et pour m'avoir fourni l'outil Raincode© et sa documentation. Merci pour son aide dans la compréhension de cet outil et pour ses remarques.

Mes remerciements vont également à Mme Bruyère pour ses conseils sur la rédaction de documents volumineux.

Je remercie aussi toutes les personnes de mon entourage pour leur soutien moral et leur aide dans la relecture de ce document.

Finalement, afin de n'oublier personne, merci à tous ceux et celles qui ont participé de près ou de loin à ce travail.

Table des matières

Table des figures

Introduction

Dans cette section, les motivations du mémoire vont être décrites. Ensuite, l'agencement des chapitres sera expliqué et pour chacun d'eux, un résumé de ce qu'ils sont supposés présenter sera donné. Finalement, la majeure contribution du mémoire sera présentée.

Motivations

Le mémoire se déroule avec la collaboration du service de recherche et développement de la société IDlink[1].

IDLink est une société spécialisée dans le domaine de l'identification automatique. Elle a pour optique de réaliser et développer des solutions informatiques permettant aux entreprises clientes d'avoir une meilleure traçabilité de leurs stocks, personnel et machines. Une partie importante de ces solutions sont réalisées en *Delphi*.

Par ailleurs, il est souvent nécessaire de restructurer le code des applications volumineuses pour maintenir la qualité à un niveau acceptable. Une technique populaire qui vise à améliorer la structure du code orienté objets est le *refactoring*.

Dans le domaine du service de génie logiciel qui effectue des recherches dans l'évolution des logiciels et dans les outils supportant cette évolution, les refactorings sont une activité très étudiée.

L'automatisation de certains refactorings s'avère très utile et a déjà largement été réalisée pour plusieurs langages orientés objets.

Pour les implémenter, de l'information provenant du code source est nécessaire. Cette information peut, par exemple, provenir d'un parser du langage pour lequel on implémente ces refactorings.

Certains outils implémentent déjà des refactorings pour le Delphi. Cependant, tous ces outils sont commerciaux vu que l'accès au parser de Borland© n'est pas libre. IDlink fournissant un outil, nommé Raincode, permettant, entre autre, de

[1] http ://www.idlink.be

parser le Delphi, le mémoire consiste donc à automatiser certains refactorings pour ce langage.

Les chapitres

Chapitre 1 Les refactorings étant une technique visant à restructurer du code orienté objets, les notions de base de ce paradigme sont décrites : classe, objet, encapsulation, méthode, message, héritage, polymorphisme, constructeur, destructeur et visibilité.

Chapitre 2 Le partie orientée objets de Delphi est présentée en illustrant les concepts définis dans le chapitre 1.

Chapitre 3 Les refactorings et les activités les concernant sont introduits globalement. En d'autres termes, ce qu'ils sont, à quoi ils servent, comment les utiliser, etc.

Chapitre 4 L'objectif de ce chapitre est d'introduire certaines techniques de compilation qui permettent la transformation du code source en une information plus utilisable. Ceci est une première étape nécessaire pour automatiser les refactorings.

Chapitre 5 Dans ce chapitre, les deux refactorings automatisés dans le cadre du mémoire vont être décrits de manière générale.

Chapitre 6 Ce chapitre présente des généralités sur Raincode, explique les algorithmes du Rename Method et Remove Parameter en méta-langage et illustre l'implémentation de ces derniers.

Chapitre 7 Finalement, une conclusion et un ensemble de perspectives termineront le mémoire.

Contribution

La majeure contribution du mémoire, hormis qu'une étude générale des refactorings soit faite au chapitre 3, est de montrer que l'automatisation de refactorings pour le Delphi est faisable et surtout, ce qui n'est pas le cas pour les outils payants existants, de détailler ce processus pour 2 refactorings : Rename Method et Remove Parameter.

Chapitre 1

Paradigme orienté objets

Ce chapitre a pour objectif d'énoncer les principes de base du paradigme orienté objets. [1] et [2] sont des ouvrages détaillés sur le sujet.

1.1 Procédural et orienté objets

Les programmes écrits en langage procédural sont constitués d'un ensemble de procédures manipulant des données, chacune de ces procédures ayant une tâche propre à réaliser. Ces programmes sont donc une succession d'appels de procédures (chacune pouvant en appeler d'autres dans son corps) avec une procédure particulière étant la procédure principale ou programme principal (par exemple, la fonction `main()` en C) qui constitue le point de départ et d'arrivée de chaque programme.

Les données qui sont utilisées peuvent être déclarées dans un conteneur [1] et dans ce cas, toutes les procédures appartenant à ce conteneur peuvent les manipuler. Elles peuvent, par ailleurs, être déclarées dans une procédure et ensuite passées de procédure en procédure (en utilisant divers modes de passage : par référence, par valeur, etc.).

L'approche du paradigme orienté objets consiste à organiser la solution du problème que le programme doit résoudre autour d'un ensemble d'objets. L'exécution du programme est donc une création (ou instanciation) de plusieurs objets qui vont collaborer ensemble pour réaliser une tâche donnée. Le mécanisme permettant aux objets de collaborer entre eux est l'envoi de messages.

[1] Par exemple, en `C`, c'est le fichier qui les contient ; en Pascal c'est l'unité qui les contient

1.2 Objet

Chaque objet a un état et un comportement. Son état est stocké dans des variables d'instance. Tous les objets sont distincts c'est-à-dire que 2 objets ayant les mêmes valeurs stockées dans leurs variables demeurent distincts. On appelle ceci le principe d'identité. Le comportement qu'un objet peut adopter est décrit par les messages qu'il peut recevoir.

Dans la figure suivante, les 2 cadres représentent 2 objets de type `Chien` possédant 2 variables d'instance (`nom` et `race`) et pouvant recevoir 3 messages différents (`aboyer()`, `manger()` et `dormir()`).

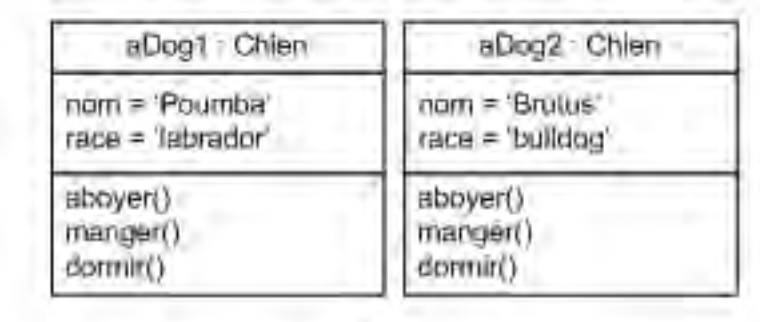

FIG. 1.1 – Représentation de deux objets instanciés

1.3 Classe

Une classe définit la structure d'une famille d'objets. Elle permet d'instancier un ou plusieurs objets du type de la classe [2].

On peut y déclarer des variables d'instance définissant la structure de l'état des objets de la famille et des méthodes définissant le comportement des objets de cette famille.

Voici un exemple d'une classe `Chien` (en pseudo-code) permettant de définir des objets de type `Chien` :

[2] Chaque nouvelle classe correspond à un nouveau type.

```
Début classe Chien
    Début variables d'instance
        race : Chaîne de caractères
        nom : Chaîne de caractères
    Fin variables d'instance
    Début méthodes
        dormir()
        Début dormir
            WriteOnScreen("ZZZzzzZZZzzz...")
        Fin dormir
        manger()
        Début manger
            WriteOnScreen("Miam miam")
        Fin manger
        aboyer()
        Début aboyer
            WriteOnScreen("Wouaf Wouaf")
        Fin aboyer
    Fin méthodes
Fin classe Chien
```

L'état des objets de type `Chien` instanciés grâce à cette classe est composé d'un nom et d'une race et le comportement qu'ils peuvent adopter est `aboyer()`, `manger()` ou `dormir()`.

Donc, de par la nature des classes, un programme orienté objets sera défini par un ensemble de classes, ce qui va permettre une grande modularité. On dit que la classe est l'unité d'abstraction du programme.

Aussi, il faut savoir que plusieurs langages implémentent le concept de classe en représentant les différentes instances de la classe par des zones mémoires différentes qui stockent les valeurs des variables d'instance. Par contre, une seule copie des méthodes est conservée en mémoire, de sorte que chaque instance pointe vers cette copie. Ceci permet de ne pas occuper inutilement la mémoire.

1.4 Méthodes et messages

On parle de message quand on veut indiquer qu'un objet doit adopter un comportement particulier. Par exemple, on pourrait envoyer le message `aboyer()` à `aDog1` pour que cet objet adopte le comportement défini par la méthode `aboyer()`.

On parle de méthode quand on veut définir le comportement que l'objet en question doit adopter à la réception du message. Le comportement de `aDog1` à la réception du message `aboyer()` sera d'écrire "Wouaf Wouaf" à l'écran.

Donc, de par la nature des objets, on peut voir que l'état est *encapsulé* dans des variables d'instances et le comportement *encapsulé* dans des méthodes contrairement aux langages procéduraux.

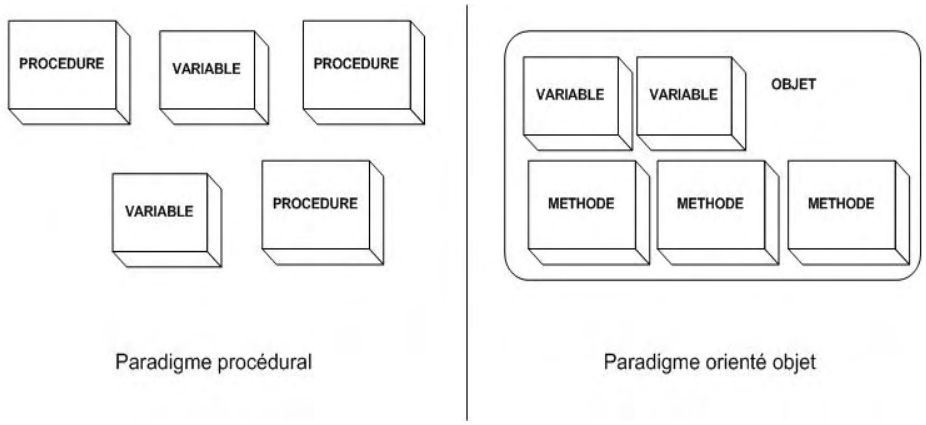

FIG. 1.2 – Différence d'encapsulation entre le paradigme procédural et le paradigme orienté objets

L'encapsulation est un principe clé du paradigme orienté objets.

1.5 Héritage

L'héritage est un mécanisme permettant de définir une relation binaire "est une sorte de" entre 2 classes. Elle se représente graphiquement de la manière suivante.

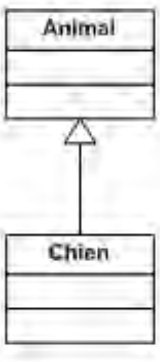

FIG. 1.3 – Représentation de la relation d'héritage entre deux classes

On dit que la classe Chien hérite de la classe Animal et on appelle Chien *sous-classe* de Animal et Animal *superclasse* de Chien. On se rend compte que la relation d'héritage établie, ici, est logique : un chien "est une sorte" d'animal.

Cette relation permet donc de former des hiérarchies de classes tel que l'exemple présenté dans la figure 1.4.

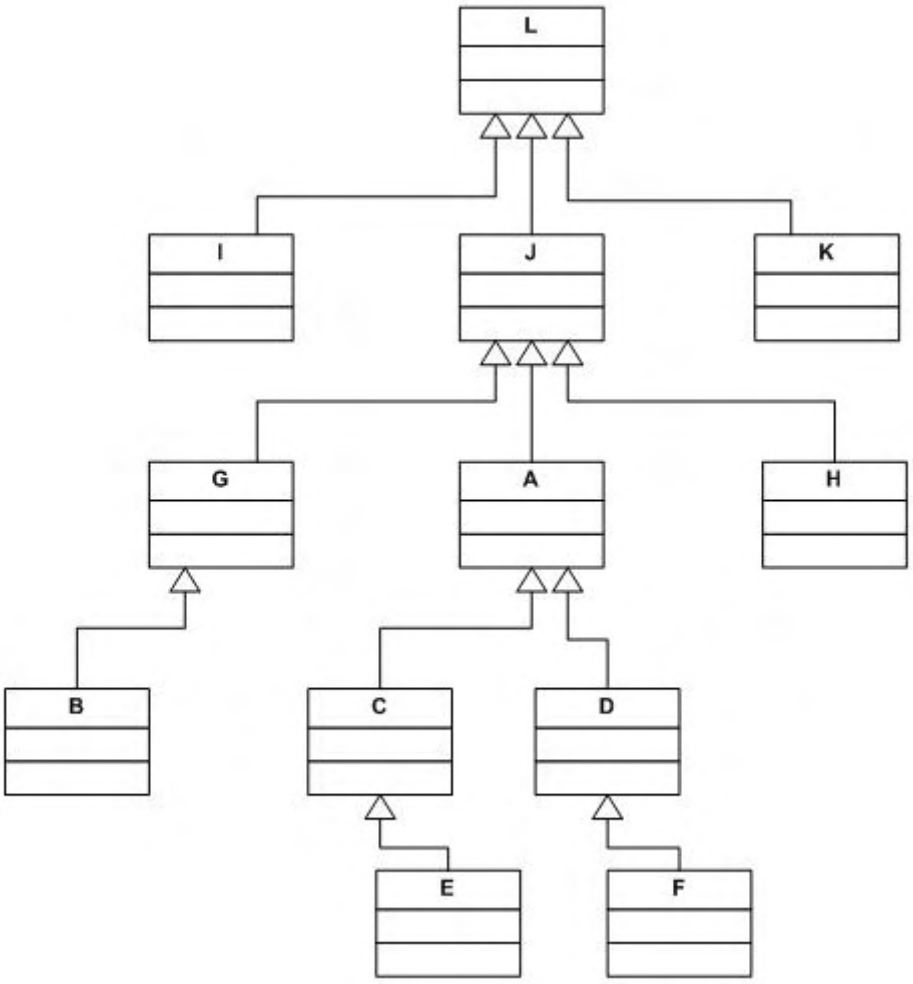

FIG. 1.4 – Exemple d'hiérarchie de classes

On définit une classe comme étant une ancêtre d'une autre si elle se trouve sur le chemin allant de la superclasse de cette dernière à la racine de l'arbre d'héritage. Similairement, on définit une classe comme étant une descendante d'une autre si elle se trouve dans le sous-arbre d'héritage dont la racine est la classe dont elle descend.

Dans la figure 1.4, on peut dire, par exemple, que les classes C, D, E et F sont descendantes de A et L et J sont ses ancêtres. Autre exemple, les classes A B C D E F G H sont descendantes de J, L étant son ancêtre.

Quand une sous-classe hérite d'une superclasse, elle hérite, en fait, des variables et des méthodes de la superclasse. On dit qu'une sous-classe *raffine* la définition de sa superclasse quand elle définit de nouvelles variables et méthodes en plus de celles héritées.

Le mécanisme d'héritage est très intéressant car il permet la réutilisation de l'état et du comportement d'un objet.

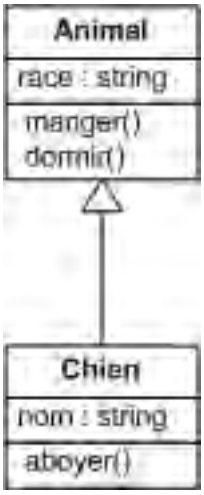

FIG. 1.5 – Illustration de l'héritage

Dans l'exemple de la figure 1.5, la classe `Chien` hérite de la variable `race` et des méthodes `manger()` et `dormir()` de la classe `Animal`. On pourra donc envoyer les messages `dormir()` ou `manger()` à une instance de type `Chien`, ce qui permet la réutilisation du code des méthodes définies dans la classe `Animal`.

1.6 Polymorphisme

Le mot de polymorphisme signifie "qui peut prendre plusieurs formes". Ce concept octroie la possibilité de définir des méthodes de même nom dans l'application mais comportant des paramètres différents.

Dans la plupart des langages orientés objets actuels, ce concept est implémenté en distinguant deux cas :

- Le polymorphisme paramétrique (également appelé surcharge ou overloading)

- Le polymorphisme d'héritage (également appelé redéfinition ou overriding)

1.6.1 Polymorphisme paramétrique

Le polymorphisme paramétrique représente la possibilité de définir plusieurs méthodes de même nom dans une même classe mais possédant des paramètres formels différents (en nombre et/ou en type). Le polymorphisme paramétrique rend ainsi possible le choix automatique de la bonne méthode à adopter en fonction des données passées en paramètre.

On pourrait, par exemple, surcharger la méthode `dormir()` dans l'exemple de la figure 1.5 de la classe `Animal` en ajoutant un nouvelle méthode
`dormir(tempsDeRepos : Entier)`.

La signature d'une méthode est constituée de son nom et de la liste des types de ses paramètres. C'est donc la signature qui permet de déterminer quelle méthode sera exécutée lorsque l'on envoie un message à un objet.

Le polymorphisme paramétrique permet ainsi de définir des opérateurs dont l'utilisation sera différente selon le type des paramètres qui lui sont passés. Il est donc possible par exemple de surcharger l'opérateur `+` et de lui faire réaliser des actions différentes selon qu'il s'agisse d'une opération entre deux entiers (addition) ou entre deux chaînes de caractères (concaténation).

1.6.2 Polymorphisme d'héritage

La possibilité de redéfinir une méthode dans une classe héritant d'une classe de base[3] s'appelle la redéfinition. Redéfinir une méthode dans une sous-classe signifie déclarer une méthode de même signature qu'une autre définie dans une des ancêtres de cette sous-classe.

Une variable du type de la classe de base pourra référencer, dans le temps, différents objets de types différents (du moment que ces types soient descendants du type de base). Mais, à tout moment, il est possible d'envoyer le même message à l'objet référencé par la variable sans se soucier de son type intrinsèque : il s'agit du polymorphisme d'héritage. Ceci permet de faire abstraction des détails des classes spécialisées d'une hiérarchie, en les masquant par une interface[4] commune (qui est celle de la classe de base).

Pour illustrer ceci, imaginons que nous définissions une classe `Aigle` héritant de la classe `Animal` (figure 1.6). La méthode `manger()`, définie dans la classe `Animal` et redéfinie dans chacune des sous-classes (en l'occurrence, `Chien` et `Aigle`) de manière adéquate, permettra, grâce au polymorphisme d'héritage, à chaque animal de

[3] Classe de base est un synonyme de superclasse.

[4] Interface signifiant ici : tous les messages qu'une classe peut recevoir.

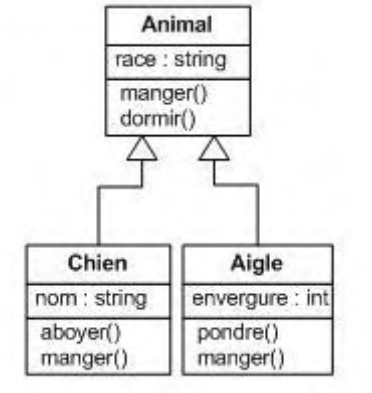

FIG. 1.6 – Illustration du polymorphisme d'héritage

manger de manière appropriée en fonction du type de l'objet référencé au moment de l'appel. Cela permettra notamment d'envoyer le message `manger()` sur un objet référencé par une variable de type `Animal` sans avoir à se préoccuper du type d'animal référencé à ce moment.

Pour terminer, il reste à dire que le polymorphisme d'héritage est bien en concordance avec la définition du concept de polymorphisme. Bien que le nom et les paramètres formels de la méthode qui redéfinit soient identiques à ceux de la méthode redéfinie, le paramètre qui diffère, ici, est l'objet sur lequel la méthode est appelée.

1.7 Constructeur et destructeur

Parmi les différentes méthodes d'un objet se distinguent deux types de méthodes bien particulières et remplissant un rôle précis dans sa gestion : les constructeurs et les destructeurs.

1.7.1 Constructeur

Comme son nom l'indique, le constructeur sert à construire l'objet en mémoire. Un constructeur va donc se charger de mettre en place les données, de les lier avec les méthodes et de créer le diagramme d'héritage de l'objet, autrement dit de mettre en place toutes les liaisons entre les ancêtres et les descendants. C'est une méthode un peu différente des autres car elle ne s'applique pas à un objet mais à une classe. On dit que les constructeurs sont des méthodes de classe et non des méthodes d'instance.

Un objet peut ne pas avoir de constructeur explicite. Dans ce cas, un constructeur générique lui sera commis d'office. Comme pour toute méthode, un constructeur

peut être surchargé, et donc effectuer diverses actions en plus de la construction même de l'objet. On utilise ainsi généralement les constructeurs pour initialiser les champs de l'objet. A différentes initialisations peuvent donc correspondre différents constructeurs.

1.7.2 Destructeur

Le destructeur est l'opposé du constructeur : il se charge de détruire l'instance de l'objet. La mémoire allouée pour le diagramme d'héritage est libérée. Il peut servir à éliminer de la mémoire le code correspondant aux méthodes d'un type d'objet si il ne réside plus aucune instance de cet objet en mémoire.

Tout comme pour les constructeurs, un objet peut ne pas avoir de destructeur. Donc, là aussi, un destructeur générique lui sera commis d'office. Un objet peut posséder plusieurs destructeurs. Leur rôle principal reste identique, mais la destruction de certaines variables internes peut différer d'un destructeur à l'autre.

1.8 Visibilité

Afin de pouvoir garantir la protection des données, il convient de pouvoir masquer certaines données et méthodes internes les gérant, et de pouvoir laisser visibles certaines autres devant servir à la gestion publique de l'objet.

La notion de visibilité varie d'un langage orienté objets à l'autre. C'est pourquoi seulement celle qui est spécifique à Delphi sera présentée dans le prochain chapitre.

Chapitre 2

Présentation de Delphi

Tout d'abord, il faut savoir que parler de Delphi en tant que langage est un abus. Le langage auquel on fait allusion en parlant de Delphi est le Pascal objet. Delphi est en fait un framework au dessus du Pascal objet.
Le Pascal objet n'est pas orienté objets pur. C'est un langage hybride de l'orienté objets et du procédural[1].
Pour ne pas alourdir ce chapitre, on supposera que le lecteur a une connaissance en Pascal et qu'il sait :

- Déclarer et utiliser des variables, constantes, procédures et fonctions.
- Utiliser les différents opérateurs du langage.
- Utiliser les structures de contrôles répétitives : For ... Do ..., Repeat ... Until ..., While ... Do ...
- Utiliser les structures de contrôles alternatives : If ... Then ... Else, Case ... Of ... End
- Déclarer et utiliser des types, records, pointeurs et tableaux.

Et donc, seul le côté orienté objets de Delphi sera présenté. Les diverses notions O.O.[2] définies dans le chapitre précédent seront reprises et illustrées ici.

Au niveau des références, [3] constitue une lecture intéressante au sujet de Delphi et [4] au sujet du Pascal.

2.1 Agencement d'une application Delphi

Chaque application Delphi est constituée d'un programme principal défini dans un fichier `.dpr` (**D**elphi **Pr**oject) et d'un ensemble d'unités chacune définie dans un fichier `.pas` (**Pas**cal). Le programme principal d'où l'application démarre et se

[1] Ceci parce qu'on peut réaliser des applications totalement procédurales comme on peut réaliser des applications totalement orientée objets.

[2] O.O. pour Orienté Objets

termine a l'allure suivante :

```
Program nom de programme;
Uses unités utilisées;
Const déclaration de constantes;
Type déclaration de types;
Function déclaration de fonctions;
Procedure déclaration de procédures;
Var déclaration de variables;
BEGIN
...
Instructions
...
END.
```

Où les déclarations Const, Type, Function, Procedure et Var peuvent être placées dans n'importe quel ordre logique[3] et la suite d'instructions définies entre BEGIN et END est l'implémentation du programme.

Les unités sont, quant à elles, définies de la manière suivante :

```
Unit nom de l'unité;
Interface
...
Implementation
...
BEGIN
...
END.
```

Dans la partie Interface, on trouve des déclarations de clauses Uses, de constantes, de types, de variables, procédures et fonctions.

Dans la partie Implementation, on retrouve ces mêmes déclarations et aussi l'implémentation des procédures ou fonctions déclarées dans la partie Interface ou la partie Implementation. Les déclarations qui se trouvent dans la partie Interface seront utilisables par d'autres programmes ou unités, celles qui se trouvent dans la partie Implémentation servent à l'usage interne de l'unité.

Le code situé entre BEGIN et END est le code d'initialisation de l'unité qui sera exécuté une et une seule fois.

[3]Par ordre logique, on entend l'utilisation d'une variable, constante, type, procédure ou fonction après sa déclaration

Lors de la compilation d'une application Delphi, un diagramme d'utilisation des unités est maintenu à jour. A chaque fois que le compilateur rencontre le mot clé **Uses** suivi d'un nom d'unité, il vérifie si cette dernière n'existe pas déjà dans le diagramme et l'ajoute dans ce cas. Ensuite, quand le diagramme est construit, le code d'initialisation de chacune des unités rencontrées est exécuté en fonction de sa position dans le diagramme.

Nous allons examiner un exemple illustratif plutôt que de donner une spécification technique précise de la manière avec laquelle ce diagramme est construit.

Voici un programme et deux unités.

```
Program UnitTransitivity;
Uses Unit1,Unit2;
BEGIN
   WriteLn('Prog Running');
   ReadLn;
END.
```

```
Unit Unit1;
Interface
Uses Unit2;
Implementation
BEGIN
   WriteLn('Unit1 Running');
END.
```

```
Unit Unit2;
Interface
Implementation
BEGIN
   WriteLn('Unit2 Running');
END.
```

Lorsque l'application est exécutée, l'ordre dans lequel les chaînes de caractères sont affichées est le suivant :

Unit2 Running
Unit1 Running
Prog Running

Donc, le compilateur a exécuté les codes d'initialisation des unités selon la numérotation des noeuds du diagramme d'utilisation représenté sur la figure 2.1.

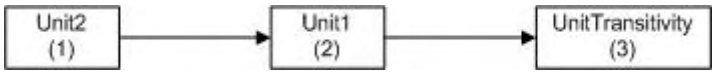

FIG. 2.1 – Diagramme d'utilisation d'unités

2.2 Classe

Chaque nouvelle classe déclarée permet l'instanciation d'un nouveau type d'objet. La déclaration d'une classe se fait dans un bloc type.

```
Type
   MyClass = class
   end ;
```

Cette déclaration permet donc de définir des objets de type MyClass.

2.3 Variables d'instance et méthodes

Les variables d'instance sont déclarées à l'intérieur de la classe comme une suite de variables normales sans le mot clé **var**.

```
Type
   MyClass = class
      champs1 : Integer
      champs2 : string
      champs3 : Boolean
   end ;
```

Les méthodes peuvent avoir ou ne pas avoir de valeur de retour. On utilise le mot clé **Function** dans le premier cas, **Procedure** dans le second.

```
Type
   MyClass = class
      champs1 : Integer
      champs2 : string
      champs3 : Boolean
      Function myFunc(param1 : Interger, param2 : Boolean) : Boolean ;
      Procedure myProc(param1 : Boolean) ;
   end ;
```

L'implémentation des méthodes ne se fait pas lors de la déclaration de la classe comme dans d'autres langages O.O. tels que Java ou C#. Si la classe est déclarée dans une unité, on devra placer le code des méthodes dans le bloc Implementation. Si elle est, par contre, déclarée dans le programme principal, on placera le code après la déclaration de la classe.

```
Unit myUnit;
Interface
Type
   MyClass = class
      Function myFunc(param1 : Interger, param2 : Boolean) : Boolean;
      Procedure myProc(param1 : Boolean);
   end;
Implementation
Function MyClass.myFunc(param1 : Interger, param2 : Boolean) : Boolean;
Begin
code de la méthode myFunc de la classe MyClass
End;
Procedure MyClass.myProc(param1 : Boolean);
Begin
code de la méthode myProc de la classe MyClass
End;
```

```
Program myProg;
Type
   MyClass = class
      Function myFunc(param1 : Interger, param2 : Boolean) : Boolean;
      Procedure myProc(param1 : Boolean);
   end;
Function MyClass.myFunc(param1 : Interger, param2 : Boolean) : Boolean;
Begin
code de la méthode myFunc de la classe MyClass
End;
Procedure MyClass.myProc(param1 : Boolean);
Begin
code de la méthode myProc de la classe MyClass
End;
```

2.4 Constructeurs et destructeurs

Les constructeurs et destructeurs se déclarent et s'implémentent comme des méthodes normales mais utilisent respectivement les mots clés Constructor et Destructor en lieu et place de Procedure ou Function.

```
Unit myUnit ;
Interface
...
Type
   MyClass = class
      ...
      Constructor Create(param1 : Interger, param2 : Boolean) ;
      Destructor Destroy(param1 : Interger) ;
   end ;
Implementation
...
Constructor MyClass.Create(param1 : Interger, param2 : Boolean)
Begin
code du constructeur Create de la classe MyClass
End ;
Destructor MyClass.Destroy(param1 : Boolean)
Begin
code du destructeur Destroy de la classe MyClass
End ;
```

2.5 Instanciation et envoi de message

La définition des classes comme décrit ci-dessus permet d'instancier des objets, de leur envoyer des messages et de les détruire.

```
...
Var
   a : MyClass ;
Begin
a := MyClass.Create(2,true) ;
a.myProc(true) ;
a.myFunc(4,false) ;
a.Destroy(true) ;
...
End.
```

Ces types d'instructions se trouvent toujours entre Begin et End. Dans cet exemple, la première instruction appelle le constructeur de la classe MyClass permettant l'instanciation d'un objet du type MyClass. Cet objet sera référencé par la variable a. Les 2 instructions suivantes envoient chacune un message à l'objet référencé par a. La dernière détruit l'objet.

2.6 Héritage

La définition d'une sous-classe se fait en spécifiant, après le mot **class**, sa superclasse entre parenthèses.

```
Type
  MySubClass = class(MyClass)
    ...
  end;
```

MySubClass héritera donc de toutes les variables et méthodes de MyClass. Lorsqu'aucune superclasse n'est spécifiée lors de la déclaration d'une classe, la classe générique TObject est imposée comme ancêtre. TObject est une classe fournie par le système. Elle permet d'avoir un type commun à toutes les classes définies dans l'application.

Ce type d'héritage est aussi appelé l'héritage simple. Delphi ne permet pas l'héritage multiple comme en C++.

2.7 Visibilité

A chacun des membres d'une classe, c'est-à-dire variables, méthodes, constructeurs et destructeurs est associée une visibilité. Si rien n'est spécifié lors de la déclaration du membre, celui-ci est considéré comme public. Autrement, il faut déclarer le membre dans le bloc de visibilité adéquat.

```
Type
  MyClass = class
    Public
      déclarations publiques
    Private
      déclarations privées
    Protected
      déclarations protégées
  end;
```

Les membres dits publics sont accessibles depuis tous les modules : le programme et les unités. On peut considérer que les éléments publics n'ont pas de restriction particulière.

La visibilité privée restreint la portée d'un membre au module où il est déclarée. Ainsi, si un objet est déclaré dans une unité avec un champ privé, alors ce champ ne pourra être accédé qu'à l'intérieur même de l'unité. Cette visibilité est à bien considérer. En effet, si une classe descendante doit pouvoir accéder à une variable ou une méthode privée, alors cette classe doit nécessairement être déclarée dans le même module que son ancêtre.

La visibilité protégée est semblable à la visibilité privée sauf que toute variable ou méthode protégée est accessible par toute classe descendante, quel que soit le module où cette descendante se situe.

2.8 Polymorphisme et liaison de méthodes

Pour le polymorphisme paramétrique, il suffit juste de déclarer 2 méthodes avec le même nom, des paramètres différents et la directive **overload**. La méthode adéquate sera utilisée en fonction des paramètres présents lors de l'appel.

```
Type
  MyClass = class
    Function myFunc(param1 : Integer, param2 : Boolean) : Boolean ;overload ;
    Function myFunc(param1 : Integer, param2 : Double) : Boolean ;overload ;
  end ;
```

Avant d'illustrer le polymorphisme d'héritage, il convient d'introduire la notion de liaison statique ou dynamique de méthode. Par défaut, les méthodes sont liées statiquement. Ceci voulant dire que le choix de la méthode utilisée lors de l'envoi d'un message à un objet ne dépend pas de son type mais du type de la variable le référençant. Supposons la définition suivante.

```
Type
   MyClass = class
      Procedure myProc;
   end;
   MySubClass = class(MyClass)
      Procedure myProc;
   end;
...
Procedure MyClass.myProc;
Begin
   WriteLn('Classe mère');
End;
Procedure MySubClass.myProc;
Begin
   WriteLn('Classe fille');
End;
...
```

Lors de l'envoi du message myProc sur un objet de type MySubClass référencé par une variable de type MyClass, c'est la méthode de la classe mère qui est appelée.

```
...
Var
   a : MyClass
...
a := MySubClass.Create;
a.myProc;
...
```

La dernière instruction affichant la chaîne "Classe mère" alors que l'objet référencé est un objet de la sous-classe. Ceci n'est certainement pas le comportement du polymorphisme d'héritage.

Il faut que les méthodes soient liées dynamiquement pour obtenir le polymorphisme d'héritage c'est-à-dire que la méthode utilisée lors de l'envoi d'un message à un objet ne dépend que du type de l'objet référencé par la variable. Ceci se fait grâce à la directive **virtual** pour la méthode de la classe mère et la directive **override** pour la classe fille. Ici, la chaîne 'Classe fille' sera affichée lors de l'exécution de la même série d'instructions.

```
Type
   MyClass = class
      Procedure myProc ;virtual ;
   end ;
   MySubClass = class(MyClass)
      Procedure myProc ;overrride ;
   end ;
...
Procedure MyClass.myProc ;
Begin
   WriteLn('Classe mère') ;
End ;
Procedure MySubClass.myProc ;
Begin
   WriteLn('Classe fille') ;
End ;
...
```

Enfin, il faut savoir que la liaison statique a un avantage au niveau de la rapidité d'exécution car les méthodes sont liées lors de la compilation et non lors de l'exécution. La liaison dynamique, quant à elle, est plus lente mais permet le polymorphisme d'héritage, concept clé du paradigme orienté objets.

Chapitre 3

Les refactorings

Ce chapitre a pour but d'introduire au lecteur les refactorings et certaines des activités les concernant. Une étude détaillée de toutes ces activités peut être trouvée dans [13].

3.1 Evolution du logiciel

La gestion de projets [5] est une activité très utilisée dans le développement d'application industrielle. Elle a pour objectif de fournir des moyens et techniques permettant de terminer le projet en respectant les contraintes de budget, temps et ressources. Une des techniques majeures utilisée dans la gestion de projets logiciels est la division du développement du logiciel en une suite de tâches qui sont : la structuration du problème et des exigences du client, le design de l'application, l'implémentation, les tests et l'évolution du logiciel. On appelle ceci le *processus du logiciel* (*software process*).

Parmi ces différentes phases, celle qui est, dans la plupart des projets actuels, la plus critique est la phase d'évolution. Elle comprend plusieurs activités : l'ajout de nouvelles fonctionnalités, la correction de bugs, l'adaptation de l'application à un nouvel environnement et l'amélioration des attributs de qualité (i.e performance, robustesse, etc.).

M.M. Lehman et L.A. Belady [6], [7] ont travaillé dans ce domaine pendant plusieurs décennies. Ils ont observé un ensemble de lois concernant l'évolution du logiciel dont certaines sont encore d'actualité. Voici deux lois intéressantes.

La première loi, appelée *Loi des changements continus* est énoncée comme suit (1980) :

> *A program that is used and that, as an implementation of its specification, reflects some other reality undergoes continuing change or becomes progressively less useful. The change or decay process continues until it is*

judged more cost effective to replace the program with a recreated version.

La première phrase, encore d'actualité, affirme que les programmes doivent évoluer sans quoi ils deviennent de moins en moins utilisés. Donc, les logiciels ont un cycle de vie qui leur est propre et sans évolution, ce cycle de vie est raccourci. Cependant, la seconde phrase dit que les changements apportés au logiciel pour le faire évoluer sont assimilés à la détérioration du logiciel et ne seront plus réalisés dès qu'il sera jugé plus rentable de remplacer l'application avec une version nouvelle. Cette deuxième phrase est liée à la deuxième loi.

La 2ème loi, *Loi de la complexité croissante*, s'énonce comme suit (1980) :

> *As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it.*

Donc, plus un programme évolue, plus sa structure se détériore et plus sa complexité augmente. On comprend, maintenant, pourquoi l'évolution du logiciel est une phase critique. En effet, elle doit comprendre des activités pour maintenir la structure du code acceptable et compréhensible pour les programmeurs qui seront amenés à faire évoluer l'application. Autrement, il sera très difficile à ces programmeurs d'ajouter de nouvelles fonctionnalités et donc de respecter les contraintes de temps et budget.

Ces activités de restructuration sont connues sous le nom de *restructuring* [8] et dans le cas de l'orienté objets *refactoring* [9], [10].

3.2 Définitions

E.J. Chikofsky & J.H. Cross ont défini le terme *restructuring*, dans un travail de standardisation des termes utilisés dans le domaine [11], de la manière suivante (1990) :

> *Restructuring is the transformation from one representation to another, while preserving the subject system's external behavior (functionality and semantics). A restructuring transformation is often one in appearence, such as altering code to improve its structure in the traditional sense of structured design.* [...] *Restructuring is often used as a form of preventive maintenance.*

Restructurer son code n'est pas quelque chose de nouveau. Cette technique est pratiquée depuis l'existence de la programmation. Elle est typiquement réalisée manuellement, ou avec l'aide d'outils comme un editeur de texte avec la fonctionnalité

search and replace.

La version orientée objets du terme précédent, *refactoring*, a été défini par Opdyke [9] comme suit :

> *the process of changing an object-oriented software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure.*

M. Fowler, dans son livre [10] :

> *Refactoring : a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.*
>
> *Refactor : to restructure software by applying a series of refactorings without changing its observable behaviour.*

L'idée est, ici, de réorganiser les classes, variables et méthodes de l'application sans en modifier le comportement pour faciliter les extensions futures.

La première apparition des refactorings est donc due à W. Opdyke. Il a défini dans sa thèse 26 refactorings de bas niveau et 3 de plus haut niveau. M. Fowler a, par après, repris certains de ces refactorings et créé un catalogue de 72 refactorings dans son livre. Très récemment, J. Kerievsky a écrit un catalogue de 27 refactorings [19] qui permettent de restructurer du code en introduisant des *Design Patterns* [20].

3.3 Exemple

Voici un exemple illustrant l'utilisation de deux refactorings. Il fait intervenir les refactorings Add Class et Pull Up Method. Add Class permet d'ajouter une nouvelle classe à l'application tandis que Pull Up Method déplace des méthodes de sous-classes avec des signatures identiques vers une superclasse.

Supposons une ville virtuelle dans laquelle des voitures et des bus peuvent se déplacer. On peut associer à cette ville un graphe qui représente les divers chemins que ces bus et voitures peuvent emprunter. Les classes Car et Bus contiennent chacune une méthode calculateNextNode() qui permet de calculer le noeud suivant vers lequel le véhicule doit se déplacer. Cette méthode de calcul est identique pour les voitures et les bus.

Il est préférable d'éviter ce code dupliqué car si l'on modifie l'algorithme de calcul à un endroit, il faudra le modifier de l'autre coté. De plus, le design de l'application oblige la duplication du code de cette méthode à chaque fois que l'on rajoutera de nouveaux véhicules utilisant cette méthode de calcul.

On peut éviter ceci en ajoutant une classe Vehicule comme superclasse de Car et Bus par le refactoring Add Class et ensuite, appliquer Pull Up Method sur calculateNextNode().

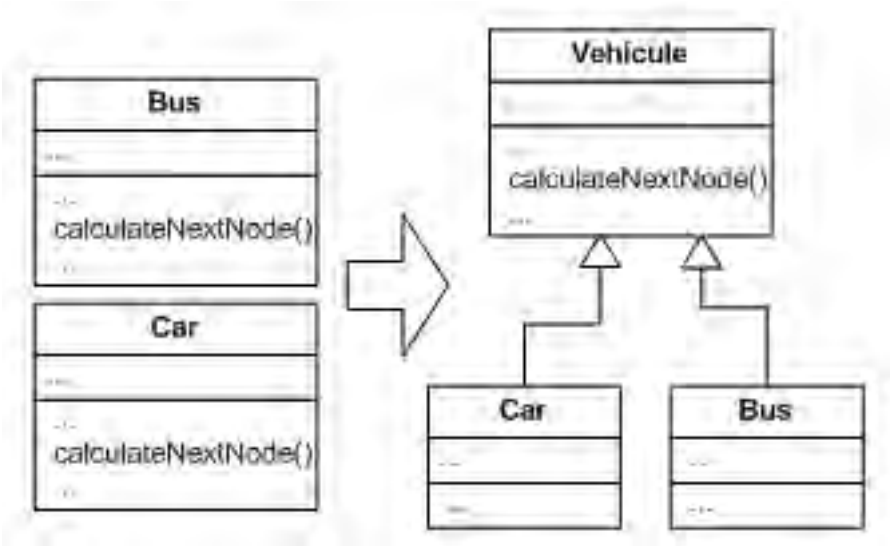

FIG. 3.1 – Application de Add Class et ensuite Pull Up Method sur calculateNextNode()

Il est trivial de voir dans cet exemple que l'application des refactorings conserve le comportement grâce à l'héritage.

3.4 Conservation du comportement

Par définition, un refactoring ne devrait pas modifier le comportement du programme. Malheureusement, une définition précise de ce qu'est le comportement est rarement donnée dans la littérature.

Ceux qui ont étudié les refactorings dans l'optique de les appliquer manuellement donnent une définition de la conservation du comportement très forte et supposent que le raisonnement humain associé à des techniques informelles permettront de converver le comportement.

Ceux qui les ont étudiés dans le but de les implémenter proposent des définitions moins fortes mais utilisent des techniques pour s'assurer qu'on ne va pas introduire de bugs.

3.4.1 Optique manuelle

Fowler [10] et Kerievsky [19] utilisent, comme technique informelle, une discipline de tests rigoureuse (*Unit Testing*). Si après l'application du refactoring, on n'observe pas d'échec lors de l'exécution des tests sur le programme transformé, alors il y a de grandes chances pour que le refactoring conserve le comportement. Malheureusement, certains refactorings vont faire échouer certains tests parce que ces derniers portent sur la structure du programme qui va être modifiée par ces refactorings [23].

3.4.2 Optique implémentationnelle

La première définition de la conservation du comportement a été donnée par Opdyke [9] et stipule que, pour les mêmes valeurs d'input, l'ensemble des outputs résultants doit être le même avant et après l'application d'un refactoring.

Il a utilisé, pour cela, le formalisme des préconditions qui lui permet de s'assurer que les refactorings définis dans sa thèse ne s'appliquent pas dans des cas où l'on est sûr que le comportement n'est pas conservé. Il a donc défini un ensemble de préconditions pour chaque refactoring et si ces dernières sont satisfaites, alors le refactoring peut être appliqué.

Imposer la préservation des outputs en fonction des inputs n'est, parfois, pas une contrainte assez forte dans certains domaines d'application car ceci ne tient pas compte d'autres critères de comportement importants. En fonction du domaine d'application :

- Pour les logiciels avec des contraintes de temps réel, un aspect essentiel du comportement est le temps d'exécution de certaines opérations. Dans ce cas, l'application du refactoring doit aussi préserver toutes les propriétés temporelles.
- Pour les logiciels embarqués, les contraintes de mémoire et de consommation d'énergie sont aussi un aspect important du comportement qui doit être préservé par le refactoring.
- Pour les logiciels critiques, la fiabilité est un aspect important à prendre en compte. En effet, la synchronisation entre processus ou threads doit être conservée dans ce type d'application.

Ces domaines d'application spécifiques sortent du cadre de ce mémoire et nous nous tiendrons donc à la définition donnée ci-dessus.

Une autre approche que celle des préconditions est de prouver formellement que le refactoring préserve la sémantique complète du programme. Pour un langage avec une sémantique simple et définie formellement comme Prolog, on peut montrer que certains refactorings conservent cette sémantique [24].

Pour des langages plus complexes, comme C++, Delphi, C#, Java, etc. où il

est très difficile de définir une sémantique formelle, ce n'est pas envisageable [1]. Dans ces langages de dernière génération, il faut imposer, en plus, des restrictions sur le type d'expressions auxquelles les refactorings s'appliquent. En effet, un refactoring rajoutant une variable dans un programme réalisant des calculs avec les adresses mémoires ne conservera pas le comportement.

3.5 Effets des refactorings sur la qualité

La définition de refactoring fait intervenir la notion d'amélioration de la structure du programme.

On peut, lorsque l'on applique un refactoring, essayer d'évaluer cette amélioration de structure en mesurant les *attributs internes de qualité* du programme. On entend, par attributs internes de qualité : la taille d'une entité, la complexité d'une entité, le couplage entre entités, etc.

Il est possible de mesurer ces attributs grâce à différentes *métriques logicielles*. On pourrait, par exemple, prendre le nombre de lignes de code comme métrique de taille du programme, le nombre de méthodes comme métrique de taille d'une classe, le nombre d'instructions comme métrique de taille d'une méthode, etc.

Une métrique intéressante pour évaluer la complexité d'une méthode est la complexité cyclomatique [21]. L'étude précise de cette métrique ainsi que de méthodes de calcul peuvent être trouvées dans l'article référencé. Mais un exemple illustratif a, quand même, été réalisé en annexe pour en donner une intuition.

Maintenant, une fois ces attributs internes de qualité mesurés, des études [25, Chap 9] ont montré que ceux-ci peuvent avoir un impact sur les *attributs externes de qualité* qui sont plus des termes qu'un humain utilisera plus pour exprimer la qualité d'un logiciel. On retrouve parmi ces attributs externes la robustesse, la performance, la facilité d'évolution et de maintenance, la réutilisabilité des composants, la portabilité, etc.

Bien sûr, l'évaluation de la qualité est quelque chose de subjectif et diffère d'un individu à l'autre. Mais, à chaque application de refactoring, la mesure des attributs internes pourra aider chacun à se faire une opinion sur l'évolution des attributs externes et donc de la qualité du logiciel.

[1] On peut faire, ici, une analogie aux compilateurs de ces langages. En effet, ceux-ci sont utilisés mais ils n'ont pas été prouvés formellement corrects.

3.6 "The Double W : Where and When"

Savoir ce que sont les refactorings est une chose mais il faut aussi répondre aux questions "Quand appliquer un refactoring ?" et "Où appliquer un refactoring dans le code ?" pour pouvoir en faire usage.

Pour la première question, M. Fowler a donné une série de règles qui peuvent aider à sentir le moment auquel on devrait appliquer des refactorings. D'après lui, des périodes destinées aux refactorings ne devraient pas être programmées périodiquement mais on devrait les utiliser :

- Lors de l'ajout de nouvelles fonctionnalités : si le code est peu clair et rend difficile l'ajout de nouvelles fonctionnalités, alors il faut essayer de modifier le design du code pour rendre l'ajout de la fonctionnalité plus facile.
- En présence de bugs dans l'application : la présence de bugs peut provenir du fait que le code est mal compris. Retravailler le design peut amener de la clarté et donc aider à trouver un bug.
- Lors de réunions de *code review* de l'équipe de développement : ce type de réunion permet de transmettre les connaissances de chaque programmeur au sujet des divers composants de l'application. Parfois, le code de certains composants peut être mal compris par certaines personnes et donc une modification de la structure du code peut les aider à mieux cerner ce que chaque composant est censé réaliser.

Une technique très utilisée pour identifier les endroits du code sujets aux refactorings est la détection de *bad smells* [10, Chapter 3]. Ce sont des structures dans le code qui entraînent un appauvrissement de la qualité du logiciel. Un exemple de bad smell est la duplication de code (cf. calculateNextNode()).

Une méthode de détection automatique de bad smells a été développée par Simon et al. [18] qui utilisent des *métriques logicielles* pour atteindre leur but. Tourwé et Mens [12] spécifient la détection de ces bads smells en utilisant des prédicats logiques du premier ordre. Tous proposent une série de refactorings applicables en fonction des bad smells trouvés.

D'autres techniques ont été développées mais elles ne seront pas citées ici car ce n'est pas le but de cette section d'en donner une liste exhaustive.

3.7 Outils supportant les refactorings

Pour les lecteurs intéressés par les outils qui existent actuellement, voici un tableau qui, pour chacun des langages orientés objets repris ci-dessous, donne un ensemble d'outils avec un support pour les refactorings.

Il faut signaler que cette liste est non-exhaustive mais permet de montrer que le support des refactorings est une activité déjà réalisée dans le monde industriel.

Langage	**Nom de l'outil**	**Site Web**	**Libre**
Java	Xrefactory	www.xref-tech.com/xrefactory-java/main.html	N
	RefactorIT	www.refactorit.com	N
	jFacor	www.instantiations.com/jfactor/	N
	IntelliJ IDEA	www.intellij.com/idea/	N
	Eclipse	www.eclipse.org	O
C++	CppRefactory	cpptool.sourceforge.net/cgi-bin/moin.cgi	O
	Xrefactory	www.xref-tech.com	N
	Ref++	www.ideat-solutions.com/refpp/	N
Smalltalk	Refactoring Browser	st-www.cs.uiuc.edu/users/brant/Refactory/	O
C#	ReSharper	www.jetbrains.com/resharper/	N
	Refactor ! Pro	www.devexpress.com/Products/NET/Refactor/	N
	C# Refactory	www.xtreme-simplicity.net/csharprefactory.html	N
Delphi	Modelmaker Tool	www.modelmakertools.com	N
	Castalia	www.delphi-expert.com	N

Chapitre 4

Notions de compilation

L'automatisation d'un refactoring passe par 3 phases. Une fois son input récupéré, il faut :

Récupérer de l'information du code source Le but de cette phase est de transformer la chaîne de caractères représentant le code source en un format plus utilisable.

Analyser cette information Le refactoring ne peut être appliqué que si les préconditions garantissant qu'il n'introduira pas de bug sont satisfaites.

Appliquer En fonction du résultat de la phase précédente, le refactoring est appliqué ou un message d'erreur est renvoyé à l'utilisateur.

La première phase de ce processus est typiquement réalisée à l'aide des techniques de compilation. La compilation est un sujet qui a déjà été très largement étudié dans la littérature. Un ouvrage majeur dans le domaine a été écrit par A.V. Aho, R. Sethi et J.D. Ullman [22].

Donc, cette section aura comme objectif d'introduire certaines des techniques d'analyse de code source pour permettre à tout lecteur de se faire une idée de la manière avec laquelle on peut transformer le code source en un format plus utilisable.

4.1 Introduction

L'analyse du code source est généralement composée de 3 phases :

Analyse lexicale. Le but est, ici, de transformer la chaîne de caractères représentant le code source en une suite de mots qui sont valides pour le langage. On appelle ces mots les *lexèmes* (*tokens*) du langage.

Analyse syntaxique. L'analyse syntaxique a pour objectif de vérifier que cette suite de lexèmes est bien conforme à la *grammaire* du langage. On représente

le fait que la chaîne respecte la grammaire par la construction d'un *arbre de dérivation* (*parse tree*).

Analyse sémantique. Durant cette phase, on s'occupe d'annoter l'arbre précédent avec des *attributs sémantiques*.

Aussi, dans ce chapitre, on va supposer que ces trois phases vont s'exécuter séquentiellement. Dans le cas de Delphi, ceci est réalisable mais il faut toutefois noter que pour certains langages comme le C, il est impossible d'exécuter les analyses de manière séquentielle. Ceci étant dû à la manière avec laquelle la grammaire est construite.

Une vue d'ensemble peut être observée sur la figure 4.1. Il faut noter que 2 activités sont réalisées pendant ces 3 phases : la gestion de la table des symboles et la gestion des erreurs.

A chaque fois qu'un identificateur est rencontré dans le code source, il sera enregistré dans la table des symboles. Donc, la table des symboles est une structure de données qui permet de stocker les identificateurs du programme et plusieurs champs les concernant. Ces champs fournissent des informations diverses comme le type de l'identificateur, sa portée, etc. L'information de chacun des identificateurs va s'enrichir de phase en phase et c'est pourquoi cette activité est liée avec les trois.

De même, une gestion des erreurs devra s'effectuer dans chacune des phases. Au niveau de l'analyse lexicale, on essaiera de détecter les mots qui ne sont pas des lexèmes du langage. Dans la seconde phase, les suites de lexèmes qui ne respectent pas la grammaire. Pendant l'analyse sémantique, les structures syntaxiques correctes mais qui n'ont pas de signification dans le contexte dans lequel elles sont utilisées. Par exemple, lorsque l'opérateur d'addition a pour opérandes une procédure et un tableau d'éléments.

4.2 Analyse lexicale

Comme dit précédemment, l'analyse lexicale a pour objectif de reconnaître les lexèmes du langage.

Plus précisément, pendant cette phase, on va s'occuper de :

- Supprimer tous les caractères d'espacement (retours chariot, tabulations, blancs) et commentaires.
- Identifier chaque lexème et stocker son type et sa valeur
- Stocker les identificateurs dans la table des symboles
- Détecter les erreurs lexicales

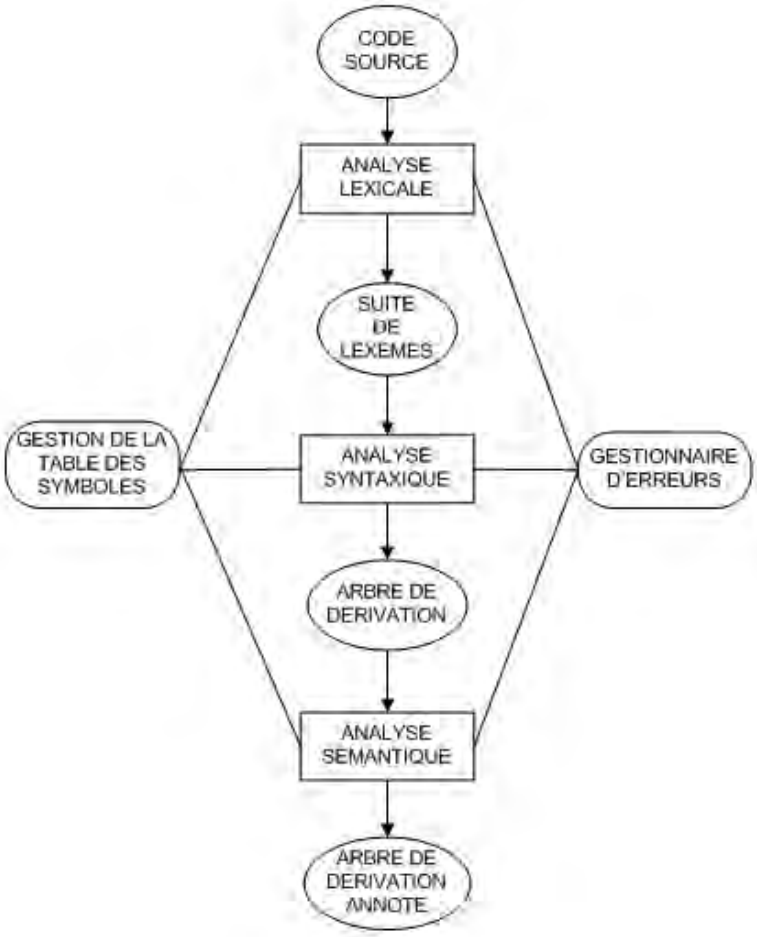

FIG. 4.1 – Vue d'ensemble des phases d'analyse de code source

4.2.1 Spécification des lexèmes

Commençons par deux définitions :

Définitions
Un *alphabet* Σ est un ensemble de symboles. Exemple :$\{0, 1\}$.
Un *mot* sur un alphabet est une suite, éventuellement vide, de symboles appartenant à cet alphabet. Exemple de mots sur $\{0, 1\}$: 1011, 0, 1, ϵ (le mot vide).

Pour permettre de spécifier de manière concise les lexèmes du langage, une notation spéciale va être utilisée : les *expressions régulières*. Elles permettent de carac-

tériser un ensemble particulier de mots définis sur un alphabet.

Définition
1. L'expression régulière ϵ décrit l'ensemble contenant le mot vide $\{\epsilon\}$
2. Si a est un symbole de Σ, alors l'expression régulière a représente $\{a\}$
3. Si e_1 et e_2 sont deux expressions régulières décrivant respectivement les ensembles de mots L_1 et L_2, alors
$e_1|e_2$ décrit $L = L_1 \cup L_2$
$e_1.e_2$ décrit $L = \{\ uv \mid u \in L_1\ et\ v \in L_2\}$ (la concaténation d'un mot de L_1 et d'un mot de L_2)
e_1^* décrit $L = \{\ u_1u_2...u_n|\ n \in \mathbb{N},\ n \geq 0\ et\ u_i \in L_1\ \forall i\ \}$ (zéro ou plusieurs concaténations de mots de L_1)

Exemple

Si l'on veut caractériser les mots (sur un alphabet alphanumérique) qui sont des *suites de longueur supérieure ou égale à 1 composées de chiffres et/ou de lettres commençant par une lettre* par une expression régulière e. Celle-ci sera définie comme suit

$$e = e_1.(e_1|e_2)^*$$

Où $e_1 = a|b|...|z|A|B|...|Z$
et $e_2 = 0|1|...|9$

4.2.2 Reconnaissance des lexèmes

Pour reconnaître les lexèmes définis par une expression régulière, on utilise des diagrammes de transition.

Définitions

Un *diagramme de transition* est un graphe orienté dont les sommets sont appelés états et dont les arcs sont appelés transitions. Les transitions sont étiquetées par des symboles de l'alphabet. Aussi, le diagramme comporte des états particuliers : un état initial et un ou plusieurs états finaux. Un diagramme de transition décrit, en fait, l'algorithme qui sera utilisé pour reconnaître un certain type de lexème.

Plus précisément, voici comment ces diagrammes sont utilisés pour reconnaître un lexème.

Initialisation.
On suppose que l'on se trouve à un endroit particulier dans la suite de symboles composant le code source et sur l'état initial du diagramme de transition.

Itération.
Supposons que l'on se trouve sur un état e_1 à une certaine étape de l'algorithme, on ne pourra emprunter une transition t partant de e_1 que si le prochain symbole à lire dans le code source est celui qui est présent sur t. Si l'on trouve une telle transition, on avance d'une position dans la suite de symbole et on change d'état en empruntant la transition t. Si pas, on arrête de parcourir le diagramme de transition. Si l'on tombe sur un état final lorsque l'on s'arrête, alors c'est que l'on vient de reconnaître un lexème.

Exemple

Considérons l'expression régulière définie dans la section précédente, pour rappel :

$$e = e_1.(e_1|e_2)^* \text{ où } e_1 = a|b|...|z|A|B|...|Z \text{ et } e_2 = 0|1|...|9.$$

Un diagramme de transition permettant de décrire l'algorithme que l'on utiliserait pour reconnaître les lexèmes décrits par cette expression est représenté en figure 4.2.

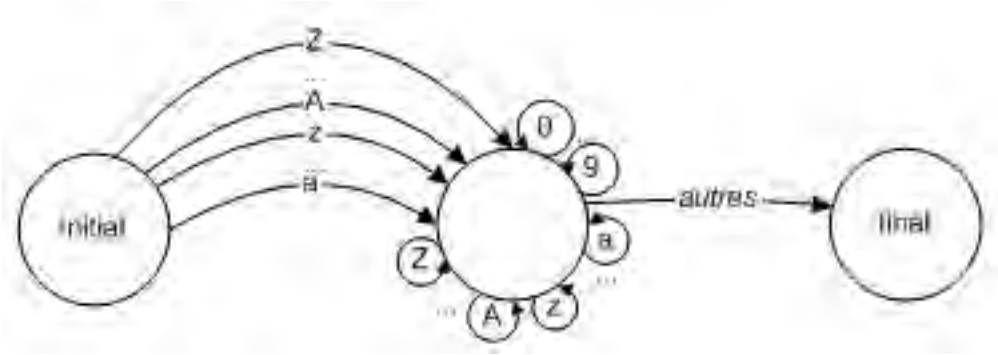

FIG. 4.2 – Diagramme de transition pour l'expression rationnelle e

L'étiquette *autres* présente sur la dernière transition représente le fait que l'on lise un caractère autre que ceux décrits sur le diagramme.

Remarques

- Ces diagrammes de transitions peuvent être vus comme des automates puisqu'ils possèdent des états, des transitions, des états initiaux, finaux et étiquettes.
- Ces automates peuvent être déterministes ou non déterministes. Un automate est déterministe si, en chacun de ses états, il n'existe pas deux transitions

sortantes avec la même étiquette. Il faut donc que chacun des automates définis pour reconnaître les lexèmes soit déterministe s'il on veut pouvoir les transformer en un algorithme.
- Il existe une méthode algorithmique de construction d'un automate fini non déterministe à partir d'une expression régulière. Elle est très bien expliquée dans la section 7 du chapitre 3 de [22] et n'a donc pas été redétaillée ici.
- Il existe un algorithme de transformation d'un automate fini non déterministe en un automate fini déterministe. Il est communément appelé *Subset Construction Algorithm*. Lui aussi est décrit dans ce même livre dans la section 6 du chapitre 3.

4.2.3 Fonctionnement général

Voici comment un analyseur lexical fonctionne de manière générale lorsque les diagrammes de transition pour les divers lexèmes du langage ont été construits.

Initialisation.
On se positionne au début du code source.

Itération.
On va tester les différents diagrammes les uns après les autres.
Si l'un d'entre eux fonctionne, on ajoute le nouveau lexème reconnu par ce diagramme à la liste des lexèmes.
Sinon, on se trouve alors en présence d'une erreur, on arrête l'analyseur lexical et on signale l'erreur.
Si la fin du code source est atteinte, l'analyse lexicale a été réalisée avec succès.

Remarque

Lorsque l'on ajoute un lexème à la liste des lexèmes reconnus, on ajoute en fait son type et sa valeur. Par exemple, pour un identificateur, son type sera représenté par '**ID**' et sa valeur sera un pointeur vers l'entrée de la table des symboles correspondant à cet identificateur, pour un nombre, son type sera '**NBR**' et sa valeur la valeur du nombre.

4.3 Analyse syntaxique

Dans cette phase, on va vérifier que la suite de lexèmes respecte bien la grammaire du langage ou, en d'autres termes, la structure syntaxique du langage. Ceci est souvent représenté par le fait que l'on puisse construire un arbre de dérivation.

4.3.1 Grammaire

La plupart des langages de programmation ont une structure syntaxique récursive qui peut être définie par une grammaire.

Définition

Pour définir une grammaire, nous avons besoin de définir 2 ensembles particuliers :

- L'ensemble des symboles *terminaux*, noté T, qui n'est composé que des lexèmes qui ont été définis pour le langage (i.e identificateurs, mots clés, caractères de séparation, etc.). De manière générale, cet ensemble contient les mots qui permettront de former des phrases [1] du langage.
- L'ensemble des symboles de variables, noté V. Les symboles de variables sont des symboles abstraits représentant chacun un ensemble de phrases particulières. Ces dernières vont servir à définir l'ensemble des phrases du langage.

Une grammaire consiste donc en un ensemble de règles, appelées aussi *productions*, de la forme

$$X \rightarrow \alpha$$

$$\text{Où } X \in V \text{ et } \alpha \in (T|V)^*$$

On représente les productions qui ont un second membre vide par

$$X \rightarrow \epsilon$$

De plus, la grammaire contient un symbole de variable particulier appelé *axiome*. L'axiome est le symbole de variable initial permettant de générer toutes les phrases du langage.

Exemple

Voici une petite grammaire définissant la syntaxe d'expressions arithmétiques :

[1] Une code source n'est qu'une phrase particulière d'un langage.

$$
\begin{aligned}
E &\rightarrow EOE \\
E &\rightarrow (E) \\
E &\rightarrow -E \\
E &\rightarrow \mathbf{ID} \\
E &\rightarrow \mathbf{NBR} \\
O &\rightarrow + \\
O &\rightarrow - \\
O &\rightarrow * \\
O &\rightarrow /
\end{aligned}
$$

Cette grammaire contient 2 symboles de variables O et E et 8 symboles terminaux '(', ')', '**ID**', '**NBR**', '+', '-', '*', '/'. L'axiome est le symbole E.

4.3.2 Dérivation

La dérivation est le processus par lequel on va transformer l'axiome du langage en la liste de lexèmes représentant le code source.

L'idée principale est, ici, que lors d'une dérivation d'une chaîne arbitraire composée de symboles terminaux et de symboles de variables, on va remplacer un des symboles de variable par le membre de droite d'une production ayant pour membre de gauche ce symbole de variable.

Définition

Plus précisément, nous dirons que $\alpha A \beta$ se dérive en $\alpha\gamma\beta$, noté $\alpha A\beta \Rightarrow \alpha\gamma\beta$, si
- $A \rightarrow \gamma$ est une production et
- $\alpha, \beta \in (T|V)^*$

Exemple

En considérant la grammaire définie dans la section précédente, voici les dérivations qu'il faudrait effectuer pour obtenir la phrase -(**ID** + **ID**) :

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(EOE) \Rightarrow -(\mathbf{ID}OE) \Rightarrow -(\mathbf{ID} + E) \Rightarrow -(\mathbf{ID} + \mathbf{ID})$$

4.3.3 Arbre de dérivation

Un arbre de dérivation peut être vu comme la représentation graphique des dérivations de l'axiome pour obtenir une phrase particulière du langage.

Définition

Soit une grammaire G d'axiome S sur les ensembles T et V. Un arbre de dérivation est un arbre ayant les propriétés suivantes :

- La racine est étiquetée par S.
- Les feuilles sont étiquetées par un lexème ou par ϵ.
- Les noeuds intérieurs sont étiquetés par un symbole de variable.
- Si A est un symbole de variable étiquetant un noeud intérieur et que $X_1, X_2, ..., X_n$ (symboles terminaux et/ou symboles de variables) sont, de gauche à droite, les étiquettes des fils de ce noeud, alors $A \rightarrow X_1X_2...X_n$ est une production de G. Si il existe une production $A \rightarrow \epsilon$, alors un noeud étiqueté par A peut n'avoir qu'un seul noeud fils étiqueté par ϵ.

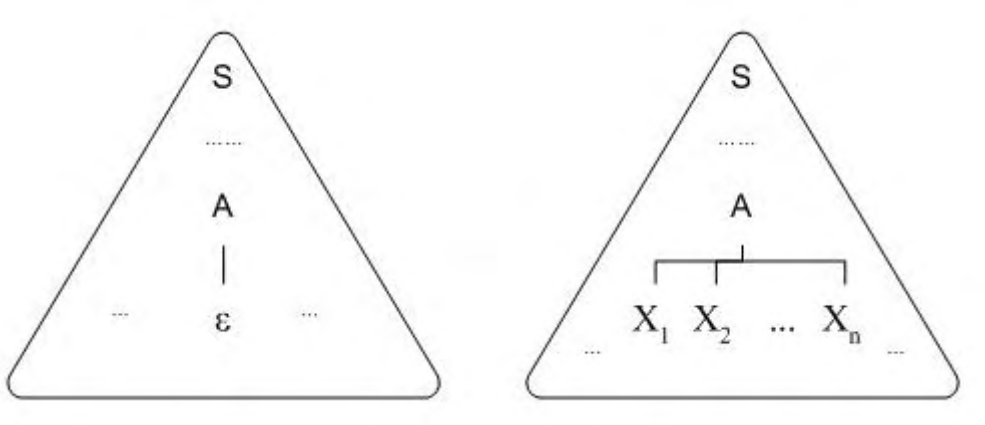

FIG. 4.3 – Représentation de la dérivation d'un symbole de variable dans un arbre de dérivation.

Exemple

La figure 4.4 représente un arbre de dérivation correspondant aux dérivations effectuées pour obtenir l'expression -(**ID**+**ID**).

4.3.4 Fonctionnement général

Le but d'un algorithme d'analyse syntaxique est donc de construire un arbre de dérivation en choisissant les bonnes productions pour que les lexèmes constituant le code source soient contenus dans les feuilles de l'arbre.

Il existe un algorithme d'analyse syntaxique développé par Earley [26] qui fonctionne pour n'importe quelle grammaire. Cette méthode a le défaut d'avoir une complexité en $\mathcal{O}(n^3)$ où n est le nombre de lexèmes.

De ce fait, d'autres méthodes ont été développées. Elles permettent d'effectuer une analyse syntaxique en $\mathcal{O}(n)$ mais pour certaines grammaires seulement. Il existe deux grandes voies : l'analyse syntaxique *descendante* et l'analyse syntaxique *ascen-*

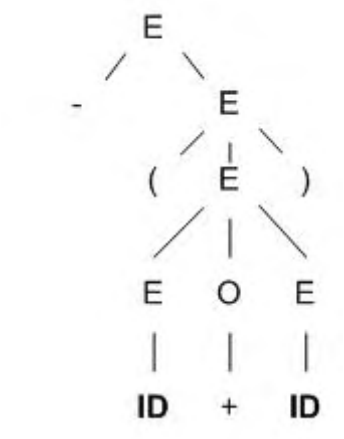

FIG. 4.4 – Arbre de dérivation obtenus pour l'expression -(**ID**+**ID**)

dante. Pour des raisons de concision, seule la méthode d'analyse syntaxique descendante sera expliquée. Elle a été choisie, aussi, parce qu'elle est très naturelle.

Analyse syntaxique descendante

Ayant en input une suite de lexèmes $w \in T^*$, cette méthode va essayer, si possible, de construire un arbre de dérivation en respectant la grammaire. La construction de cet arbre va se réaliser de haut en bas en lisant les lexèmes un par un de gauche à droite.

L'idée principale de l'algorithme est l'utilisation d'une table bidimensionnelle $M(X, a)$ qui va permettre à l'algorithme de savoir quelle est la bonne production à utiliser lorsque le noeud qui en cours de traitement est étiqueté par un non terminal X et que le lexème à lire est a.

Algorithme

L'algorithme utilise une variable ip qui va référencer le lexème qui est en cours de traitement.

Initialisation.
Le premier noeud à traiter est l'axiome et ip référence le premier lexème du code source.

Itération.
Soit X l'étiquette du noeud en cours de traitement et a le lexème référencé par ip.

- Si X est un symbole terminal :
  Si $X = a$ alors mettre à jour ip vers le prochain lexème de la liste et le noeud en cours de traitement.
  Sinon erreur syntaxique

- Si X est un symbole de variable :
  Consulter la table des règles permises M avec X et a comme paramètres.
  Si une règle $X \rightarrow X_1X_2...X_n$ est proposée, appliquer cette règle et mettre à jour le noeud en cours de traitement.
  Sinon si une règle $X \rightarrow \epsilon$ est proposée, mettre à jour le noeud en cours de traitement.
  Sinon erreur syntaxique.

L'algorithme s'arrête lorsque l'on rencontre une erreur syntaxique ou lorsque l'on atteint la fin de la liste de lexèmes.

L'illustration de l'arbre qui est en construction est représentée sur la figure 4.5. Les hachures représentent les lexèmes déjà traités.

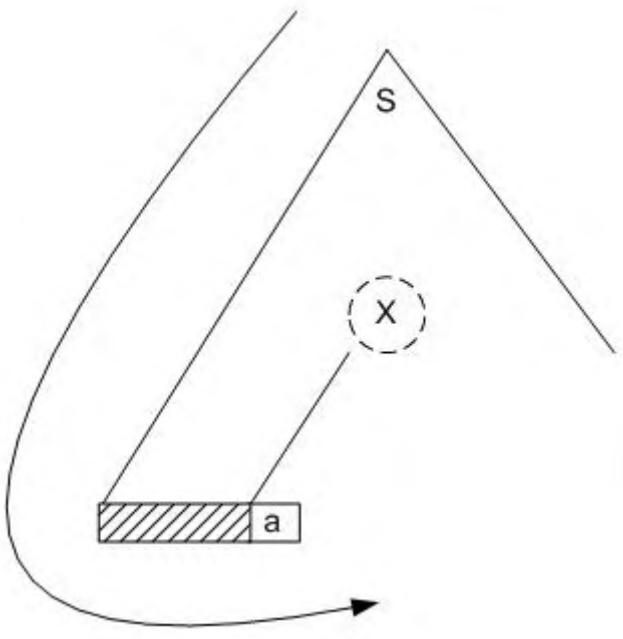

FIG. 4.5 – Représentation de l'arbre en construction lors du traitement d'un noeud X et d'un lexème a.

Remarques

- Depuis le début de cette section, nous parlons de la "construction d'un arbre de dérivation" mais dans l'algorithme précédent, il n'y a aucune instruction qui effectue cette tâche.
  Pour réaliser ceci, on pourrait rajouter, lors de l'initialisation,
  `create_parse_tree(`X`)` qui initialiserait l'arbre avec l'axiome. Aussi, lorsque la table $M(X, a)$ renvoie une production permettant de dériver le non terminal X, on ajouterait un appel du type `add_subnodes_to(`$X, X_1, X_2, \ldots, X_n$`)` qui créerait les noeuds fils de X.
  Habituellement, les compilateurs ne construisent pas directement cet arbre de dérivation mais une version condensée de celui-ci. Cependant, l'outil qui va être utilisé pour implémenter les refactorings crée explicitement cet arbre de dérivation, c'est la raison pour laquelle ces instructions sont quand même introduites ici.
- Il existe deux versions de cet algorithme utilisant des techniques différentes pour permettre de savoir comment mettre à jour le noeud en cours de traitement. La première utilise la récursivité et la seconde utilise une pile. Elles sont toutes les deux décrites dans la section 4 du chapitre 4 de [22].
- Une méthode permettant de construire la table $M(X, a)$ est expliquée de manière détaillée dans cette même section et elle ne sera donc pas rappelée ici.

Exemple

Pour illustrer ce qui vient d'être expliqué, nous allons changer de grammaire car celle définie auparavant n'est pas appropriée pour l'analyse syntaxique descendante [2].

$$
\begin{aligned}
&E \rightarrow TE' \\
&E' \rightarrow +TE' \mid \epsilon \\
&T \rightarrow FT' \\
&T' \rightarrow *FT' \mid \epsilon \\
&F \rightarrow (E) \mid \textbf{ID}
\end{aligned}
$$

Il faut noter que $A \rightarrow B|C$ est équivalent à $A \rightarrow B$ et $A \rightarrow C$.

[2] La raison pour laquelle cette grammaire ne convient pas se détermine lors de la construction de la table $M(X, a)$.

La table M pour cette grammaire est montrée ci dessous.

Symbole de variable	Symbole terminal				
	ID	$+$	$*$	$($	$)$
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$
F	$F \to$ **ID**			$F \to (E)$	

Pour l'expression $\mathbf{ID} + \mathbf{ID} * \mathbf{ID}$, voici les quelques premières itérations de l'algorithme :

Au départ, le noeud à traiter est E et ip référence **ID**

FIG. 4.6 – Initialisation

E est un symbole de variable et $M(E, \mathbf{ID})$ renvoie la production $E \to TE'$.

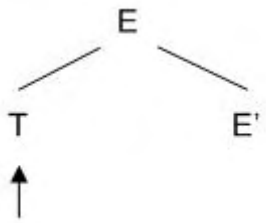

FIG. 4.7 – Application de $E \to TE'$

L'itération suivante est représentée sur la figure 4.8. La table $M(T, \mathbf{ID})$ renvoie $T \rightarrow FT'$.

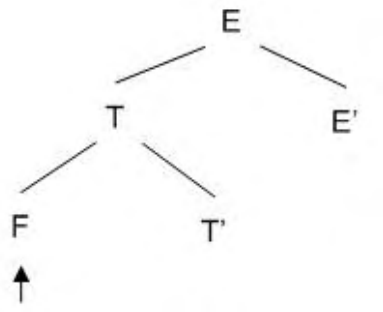

FIG. 4.8 – Application de $T \rightarrow FT'$

Ensuite, sur la figure 4.9, F se dérive **ID**.

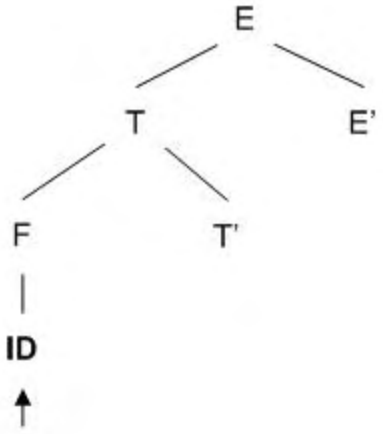

FIG. 4.9 – Application de $F \rightarrow \mathbf{ID}$

Lors de l'itération suivante (figure 4.10), **ID** qui était le noeud en cours de traitement était un symbole terminal et correspondait à *ip*. Le lexème en cours de traitement est maintenant + et le noeud en cours de traitement T'.

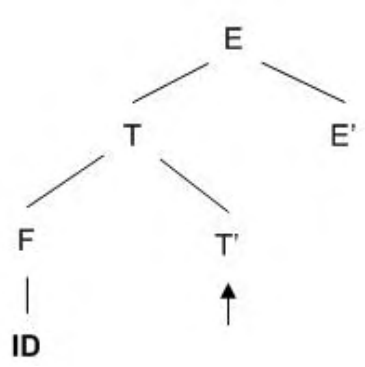

FIG. 4.10 – Mise à jour du noeud en cours de traitement

L'algorithme continuera comme ceci jusqu'à ce que le dernier identificateur soit lu vu que l'expression **ID** + **ID** ∗ **ID** respecte la grammaire et donc n'induira pas d'erreur.

4.4 Analyse sémantique

Durant cette phase, on va s'occuper d'ajouter de l'information à l'arbre de dérivation en annotant les noeuds avec des *attributs sémantiques*.

On distingue deux types d'attributs : les attributs synthétisés qui sont calculés en fonction des attributs des fils d'un noeud et les attributs hérités qui sont, eux, calculés en fonction des attributs du père et des frères d'un noeud. Ils permettent, en autre, de calculer : la valeur des expressions arithmétiques, le type des expressions, la notation postfixe des expressions (utilisée lors de la génération de code) et tout autre attribut utile à la compréhension du langage.

Le calcul de ces attributs va se faire grâce à des règles sémantiques qui vont être ajoutées à la grammaire. On appellera l'ensemble *grammaire attribuée*.

4.4.1 Grammaire attribuée

Définition

Dans une grammaire attribuée, chaque production $A \rightarrow \alpha$ où $A \in V$ et $\alpha \in (T|V)^*$ est associée à un ensemble de règles sémantiques de la forme $b := f(c_1, c_2, ..., c_k)$ où f est une fonction et soit

- b est un attribut synthétisé de A et $c_1, c_2, ..., c_k$ sont des attributs des symboles terminaux ou non terminaux de α, ou
- b est un attribut hérité d'un des symboles de α et $c_1, c_2, ..., c_k$ sont des attributs de A et/ou des symboles de α.

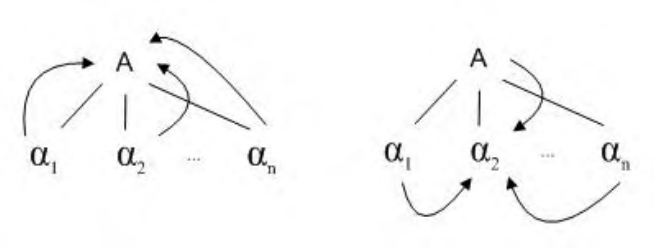

FIG. 4.11 – Représentation schématique, à gauche, du calcul d'un attribut synthétisé de A et d'un attribut hérité de α_2 à droite

Exemples

Pour une production $E \rightarrow E + E$ permettant d'additionner deux opérandes, on pourrait calculer un attribut synthétisé val contenant le résultat :

Règle syntaxique	Règle sémantique
$E_1 \rightarrow E_2 + E_3$	$E_1.val := E_2.val + E_3.val$

Remarque : la notation pointée $A.b$ exprime le fait que l'on utilise l'attribut b du non terminal A.

Lorsque des identificateurs sont déclarés, on pourrait utiliser un attribut synthétisé type pour récupérer le type et un attribut hérité in pour répandre ce type à la liste des identificateurs :

Règle syntaxique	Règles sémantiques
$D \rightarrow T : L$	$L.in := T.type$
$T \rightarrow$ **int**	$T.type := integer$
$L \rightarrow L_1,$ **id**	$L_1.in := L.in$ `add_type(id,`$L.in$`)`
$L \rightarrow$ **id**	`add_type(id,`$L.in$`)`

`add_type(id,`$L.in$`)` est une routine permettant d'ajouter un type à un identificateur dans la table des symboles.

Une illustration de l'utilisation des attributs type et in est représentée sur la figure 4.12.

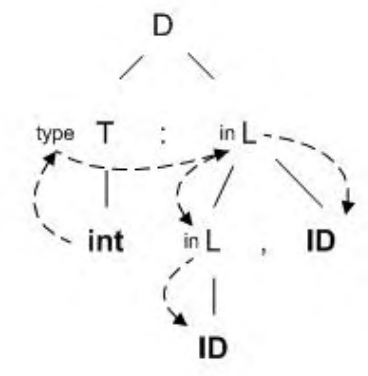

FIG. 4.12 – Exemple d'utilisation d'un attribut synthétisé et d'un attribut hérité

4.4.2 Fonctionnement général

Disposant des règles syntaxiques et sémantiques, il reste à implémenter effectivement le calcul des attributs.

Il existe plusieurs types d'implémentations :

- Des implémentations utilisant des grammaires S-attribuées. Ces méthodes ne permettent l'évaluation que des attributs synthétisés mais elles peuvent s'effectuer en même temps que l'analyse syntaxique.
- Des implémentations utilisant des grammaires L-attribuées. Elles permettent l'évaluation des attributs synthétisés et une partie des attributs hérités. Elles ont aussi l'avantage de s'effectuer en même temps que l'analyse syntaxique.
- Une méthode générale fonctionnant pour tous types d'attributs mais ne pouvant s'effectuer qu'après l'analyse syntaxique.

La dernière des implémentations sera présentée ci-après. Les méthodes utilisant des grammaires S ou L-attribuées sont intensivement étudiées dans les sections 3, 4, 5 et 6 du chapitre 5 de [22].

La méthode générale, parfois appelée méthode du graphe de dépendance, fait intervenir 3 étapes pour réaliser l'évaluation des règles sémantiques :

1. On associe, dans un premier temps, un graphe de dépendance à l'arbre de dérivation. Ce graphe va montrer les relations de dépendance entre les attributs de la grammaire.
2. Ensuite, on effectue un tri topologique de ce graphe. Ce tri va permettre de savoir l'ordre dans lequel les règles sémantiques doivent être évaluées.
3. Enfin, il ne reste plus qu'à effectivement évaluer ces règles sémantiques.

Graphe de dépendance

Si une règle sémantique calculant un attribut b a besoin de la valeur d'un attribut c, on dit que b dépend de c.

Un graphe de dépendance est un graphe orienté qui possède un noeud pour chaque attribut sémantique et une arrête du noeud c au noeud b si l'attribut b dépend de c.

Voici l'algorithme créant un graphe de dépendance associé à un arbre de dérivation :

Pour chaque noeud n dans l'arbre **Faire**
  Pour chaque attribut a du symbole étiquetant n **Faire**
    Construire un noeud dans le graphe de dépendance pour a
Pour chaque noeud n dans l'arbre **Faire**
  Pour chaque règle sémantique $b := f(c_1, c_2, ..., c_k)$
      associée à la production utilisée en n **Faire**
    Pour i allant de 1 à k **Faire**
      construire une arrête du noeud pour c_i au noeud pour b

Tri topologique

Un tri topologique d'un graphe orienté acyclique est une numérotation des noeuds du graphe telle que les arrêtes vont toujours d'un noeud de numéro plus petit vers un noeud de numéro plus grand. Donc, si une arrête va d'un noeud n à un noeud m, le numéro de $n <$ le numéro de m.

Après cette étape, l'ordre d'évaluation des règles sémantiques est connu et donc l'évaluation sémantique peut s'effectuer.

Exemple

Nous allons illustrer ceci avec un exemple. La grammaire définie dans la section traitant l'analyse syntaxique va être réutilisée. Des règles sémantiques ont été rajoutées pour permettre d'évaluer la valeur des expressions arithmétiques définies par cette grammaire.

Règle syntaxique	Règles sémantiques
$E \rightarrow TE'$	$E.val := E'.val$ $E'.h := T.val$
$E' \rightarrow +TE'_1$	$E'_1.h := T.val + E'.h$
$E' \rightarrow \epsilon$	$E'.val := E'.h$
$T \rightarrow FT'$	$T.val := T'.val$ $T'.h := F.val$
$T' \rightarrow *FT'_1$	$T'_1.h := F.val * T'.h$
$T' \rightarrow \epsilon$	$T'.val := T'.h$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \mathbf{NBR}$	$F.val := \mathbf{NBR}.val$

Cette grammaire attribuée fait intervenir : un attribut synthétisé val qui va, pour chaque noeud, contenir la valeur d'une expression arithmétique et un attribut hérité h, pour les noeuds E' et T', permettant de répandre la valeur de sous-expressions dans l'arbre.

Voici l'arbre de dérivation correspondant à l'expression $2 + 3 * 4$:

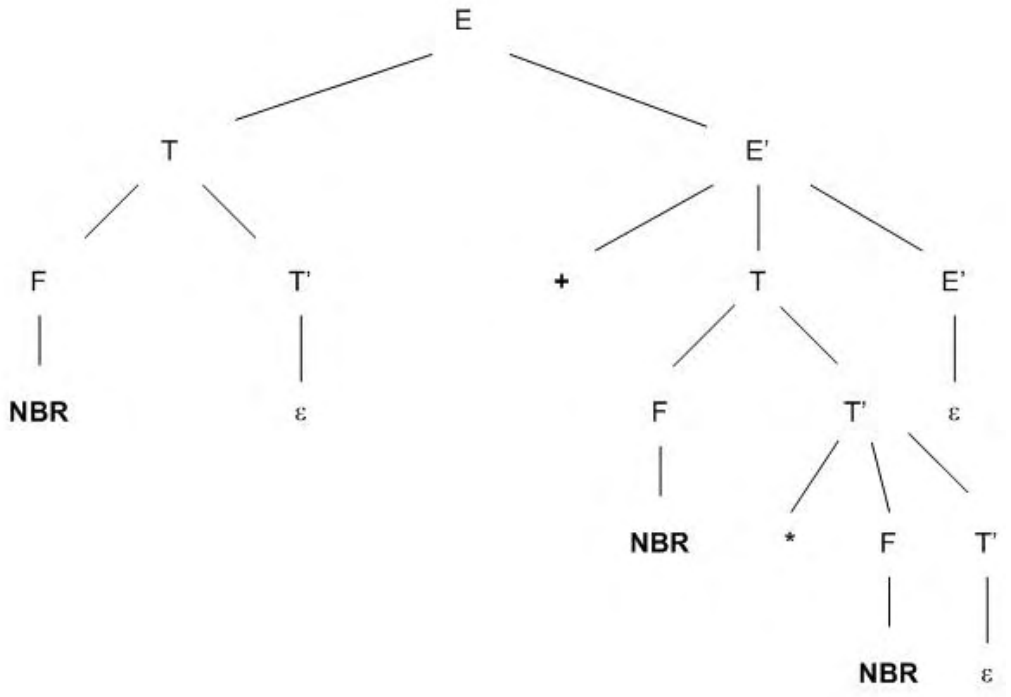

FIG. 4.13 – Arbre de dérivation obtenu pour $2 + 3 * 4$

Ensuite, on crée le graphe de dépendance associé. Les cercles colorés sont les noeuds du graphe. Pour des raisons de clarté, les noeuds correspondant à l'attribut val sont en rouge et ceux correspondant à l'attribut h sont en vert.

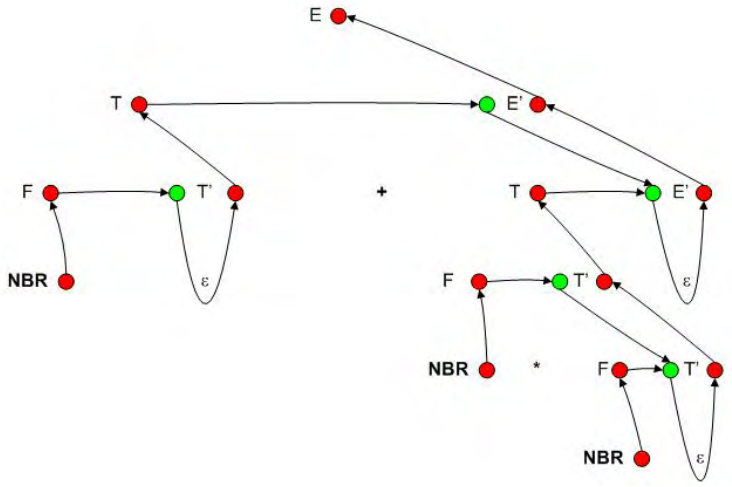

FIG. 4.14 – Graphe de dépendance associé à l'arbre de dérivation

Ce graphe ne contenant pas de cycle, on peut effectuer un tri topologique.

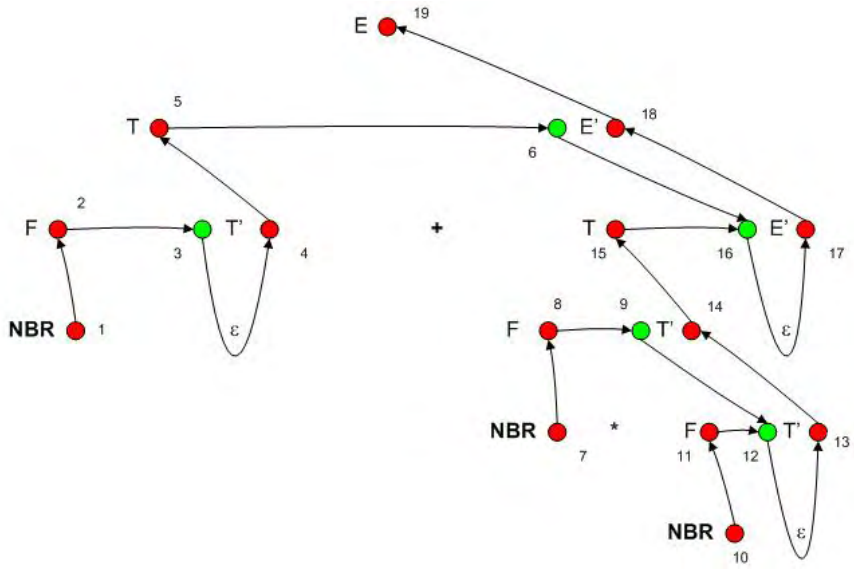

FIG. 4.15 – Tri topologique du graphe de dépendance

Enfin, grâce au tri topologique, on peut effectivement évaluer la valeur de l'expression arithmétique.

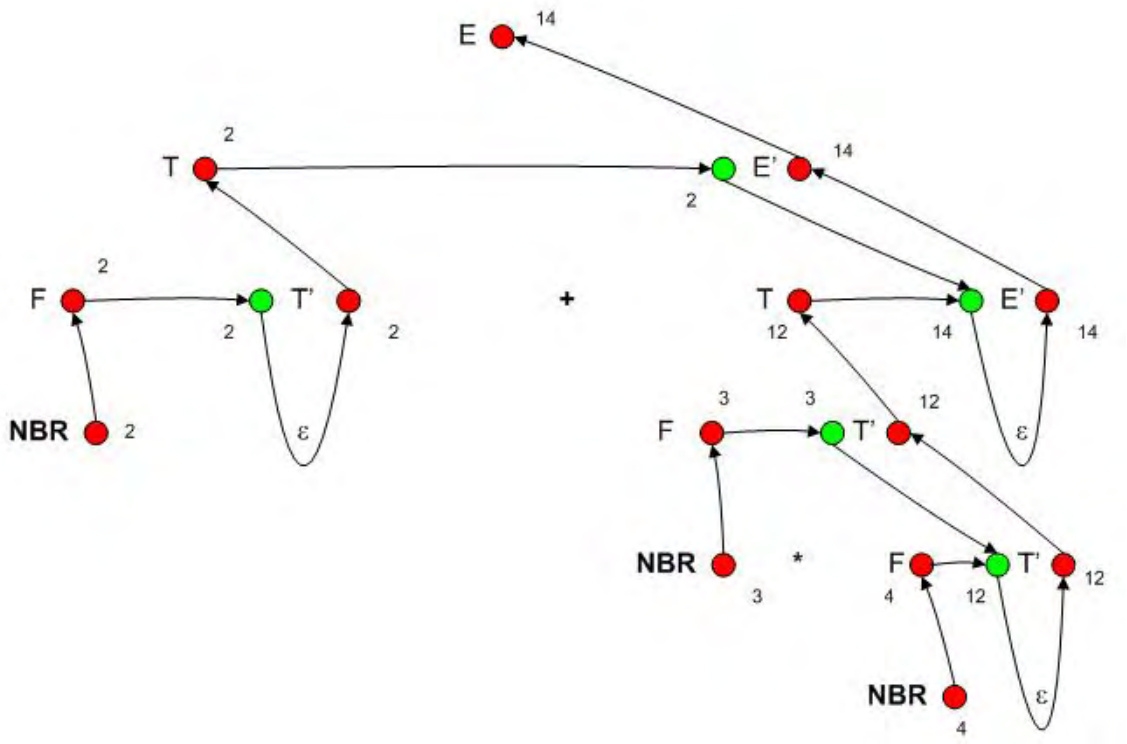

FIG. 4.16 – Calcul effectif de la valeur de l'expression arithmétique $2 + 3 * 4$

Chapitre 5

Rename Method et Remove Parameter

Ce chapitre a pour objectif d'expliquer les deux refactorings qui ont été implémentés dans le cadre du mémoire.

Aussi, leurs préconditions et une discussion de ces dernières vont être présentées. Les préconditions qui ont été utilisées viennent de la thèse de S.Tichelaar [15] (voir aussi [14]) qui a étudié les refactorings en détail assez récemment. Il les a non seulement étudiés en détail mais aussi de manière indépendante du langage pour lequel on les utilise. Donc, seules les préconditions pertinentes pour le Delphi ont été reprises de cette thèse.

Enfin, le Rename Method et Remove Parameter ont été choisis car ils sont représentatifs de certains problèmes que l'on peut rencontrer lorsque l'on veut automatiser des refactorings. Des exemples de tels problèmes sont : la recherche de méthodes dans une hiérarchie, la comparaison de signatures de méthodes , la recherche d'appels de méthodes particulières, etc.

5.1 Rename Method(Method m, String newname)

Ce refactoring renomme la méthode m et toutes les méthodes définies dans la même hiérarchie et liées par le polymorphisme d'héritage [1]. Tous les appels à ces méthodes doivent être modifiés pour que ceux-ci réfèrent le nouveau nom donné.

Dans l'exemple de la figue 5.1, la méthode `getNN()` de `B` est renommée en `nextNode()`. Les méthodes des classes `A`, `C`, `D`, `E`, `F` sont aussi renommées pour

[1] Quand une méthode est définie dans une classe de base et que cette méthode est redéfinie dans les sous-classes de la classe de base, on dit parfois que toutes ces méthodes sont liées par le polymorphisme d'héritage. Ceci n'est bien sûr pas à confondre avec la liaison statique ou dynamique d'une méthode qui n'a rien à voir avec ceci.

qu'il n'y ait pas d'incohérence au point de vue de la redéfinition. Aussi, les appels à ces méthodes doivent être modifiés en conséquence (ceci étant représenté, dans l'exemple, par une classe `X` utilisant la méthode `getNN()` de `B` dans l'une de ses méthodes).

Le fait que l'on renomme aussi les méthodes liées avec m par le polymorphisme d'héritage des classes de la même hiérarchie est nécessaire pour que le comportement soit conservé. De plus, si l'on ne renomme pas les appels à ces méthodes, il est clair qu'il n'y aura pas de conservation du comportement du programme initial.

5.1.1 Préconditions

1. Toutes les sous-classes de la plus haute superclasse (nommons cette superclasse *hs*) contenant une méthode redéfinie par m ainsi que *hs* elle-même ne doivent pas déjà contenir une méthode avec une signature composée de *newname* et de la même liste de paramètres que m.
2. *newname* doit être valide pour pouvoir être attribué comme nom de méthode.

5.1.2 Discussion des préconditions

La première précondition empêche de remplacer, involontairement, une méthode qui serait héritée par *hs*. Par exemple, si `A` héritait d'une méthode `nextNode()` avant le Rename Method, le comportement du programme ne serait pas conservé car la méthode héritée ne serait plus utilisée lorsque l'on enverrait un message `nextNode()` à un objet de type `A` et serait remplacée par la méthode renommée après l'application du refactoring.

Elle empêche aussi une possible double définition de cette méthode dans *hs*. Par exemple, si `A` contenait déjà une méthode `nextNode()`, le programme ne compilerait même plus.

Enfin, elle empêche des doubles définitions et des remplacements involontaires des méthodes des sous-classes de *hs*. Par exemple, si `C` contenait une méthode `nextNode()` avant l'application du refactoring, on constaterait une double définition dans la classe `C`, un remplacement involontaire de cette méthode `nextNode()` déjà présente dans `C` par la méthode renommée dans la classe `F` [2] et un remplacement involontaire de la méthode renommée de la classe `A` par cette méthode déjà présente dans `C`. Des problèmes semblables sont rencontrés si l'on prend les classes `B`, `D`, `E` ou `F` au lieu de `C` dans cet exemple.

Pour la deuxième précondition, il est clair que si le nouveau nom n'est pas valide pour le langage, le programme transformé ne compilera plus.

[2] Si un objet de type `F` invoquait la méthode `nextNode` de `C` avant le refactoring, il invoquerait la méthode `nextNode` de `F` après le refactoring.

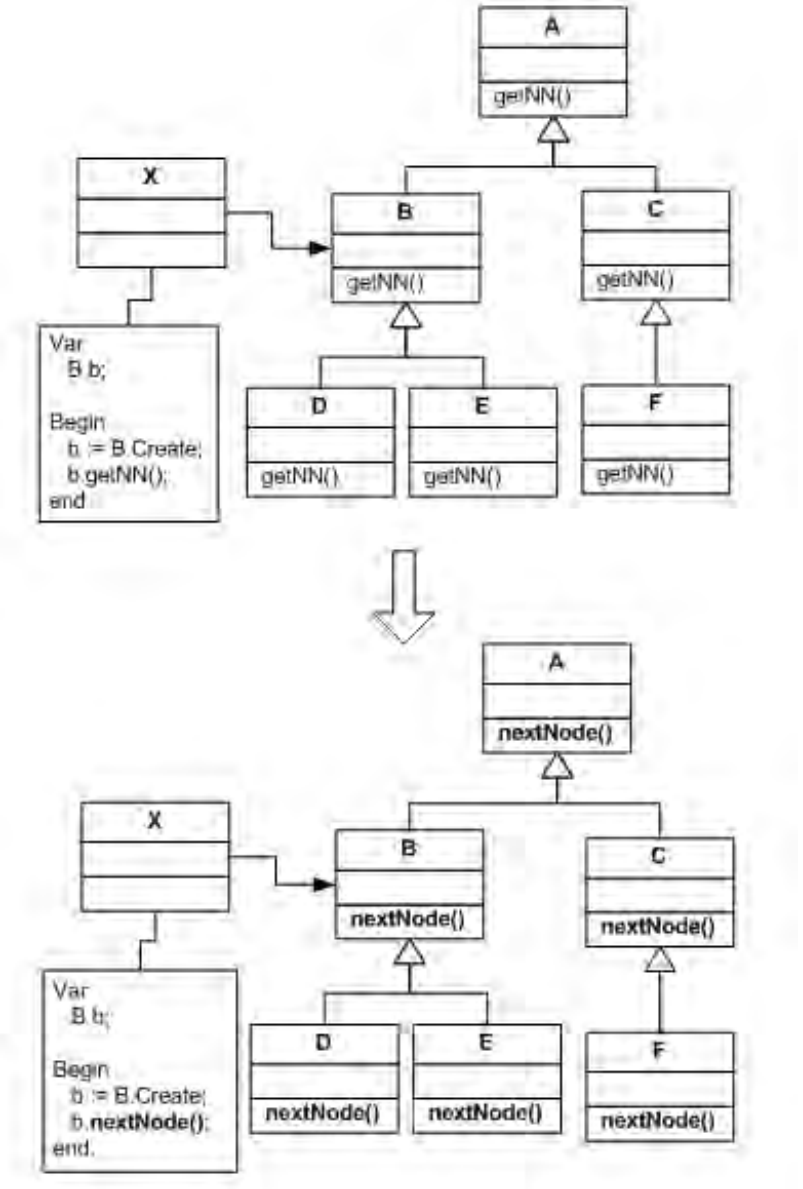

FIG. 5.1 – Application du Rename Method sur `getNN()` de la classe `B`.

5.2 Remove Parameter(Parameter p)

Ce refactoring retire le paramètre p des arguments de sa méthode. Toutes les définitions de cette méthode dans d'autres classes de la même hiérarchie seront modifiées et le paramètre correspondant en position *pos* sera supprimé, *pos* étant

la position de p dans la liste de paramètres de sa méthode. Ceci pour garantir une cohérence dans la redéfinition. Aussi, dans tous les appels à ces méthodes, on va supprimer l'argument en position *pos*.

Dans l'exemple de la figure 5.2, le paramètre `o` est retiré de la méthode `nextNode (o : TObject)` située dans la classe `B`. Les méthodes des classes `A`, `C`, `D`, `E`, `F` sont aussi modifiées pour qu'il n'y ait pas d'incohérence au point de vue de la redéfinition. Aussi, on supprime l'argument actuel `anObject` d'un appel à la méthode `nextNode` de `B` (et il faut faire de même pour tout autre appel à une des méthodes modifiées par le refactoring).

Ce refactoring a des similitudes avec le refactoring Rename Method. Dans les deux cas, la signature d'une méthode change et donc la recherche des méthodes à modifier et des appels à ces dernières est la même.

5.2.1 Préconditions

1. p et les paramètres correspondants ne doivent pas être utilisés dans aucune des méthodes qui sont susceptibles d'être modifiées.
2. Toutes les sous-classes de la plus haute superclasse (nommons cette superclasse hs) contenant une méthode redéfinie par la méthode du paramètre p (appelons la m) ainsi que hs elle-même ne doivent pas déjà contenir une méthode avec une signature correspondant à m lorsqu'on lui enlève p.

5.2.2 Discussion des préconditions

La première précondition empêche de retirer des paramètres qui seraient utilisés dans les corps de leur méthode. Ceci créerait des problèmes de références et le programme ne compilerait plus.

Pire, il se pourrait qu'il existe une variable d'instance avec le même nom que ce paramètre. Ceci impliquerait, après l'application du refactoring, l'utilisation de la variable d'instance au lieu du paramètre qui viendrait d'être retiré.

La seconde précondition est semblable à la précondition 1 du refactoring Rename Method. Ceci parce que, dans les deux cas, il y a modification de la signature des méthodes impliquées dans ces refactorings et ceci pourrait créer des conflits avec des méthodes qui existeraient déjà.

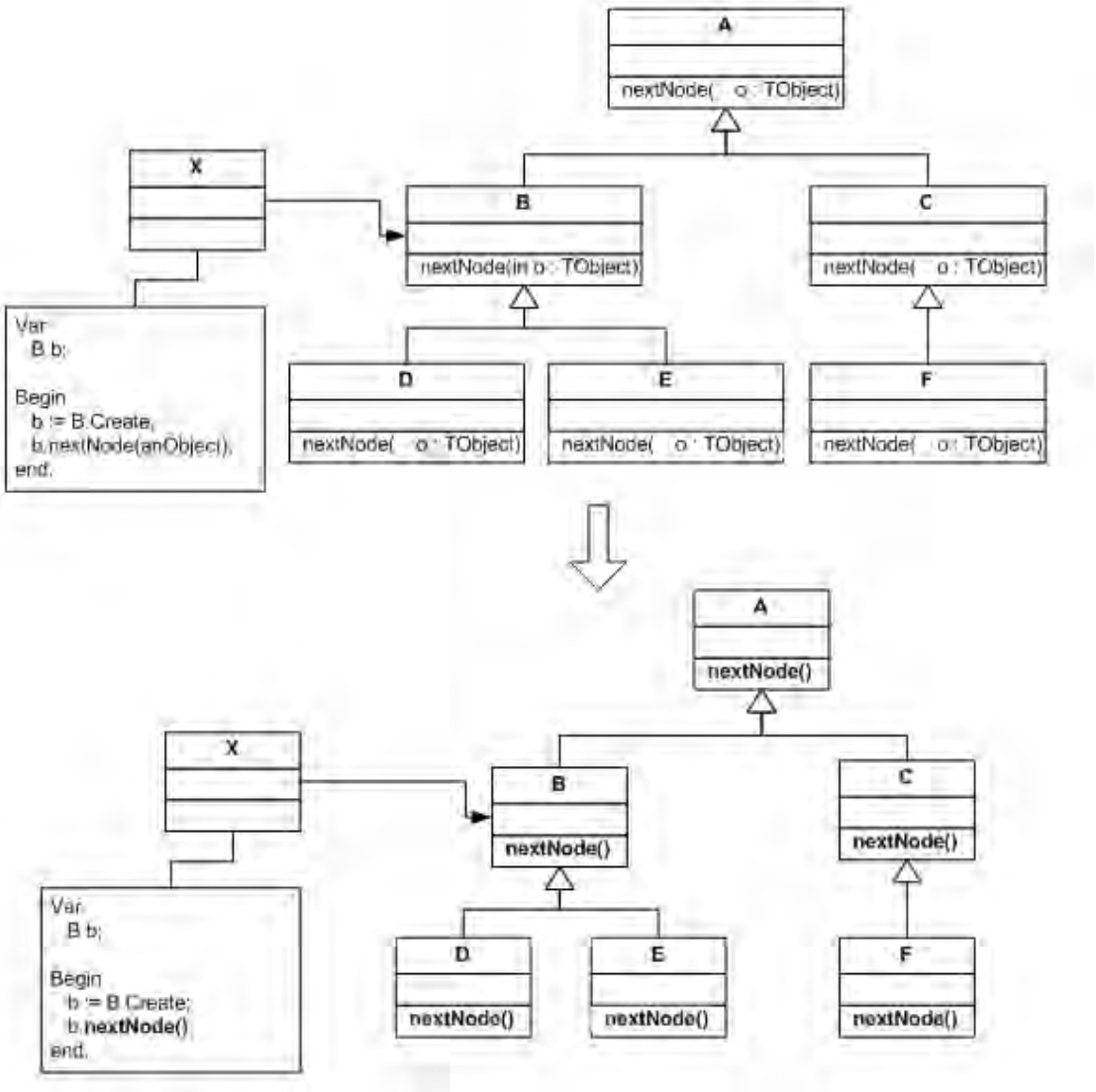

FIG. 5.2 – Application du Remove Parameter sur le paramètre `o` de la méthode `nextNode()` de la classe `B`.

Chapitre 6

Implémentation

Ce chapitre a pour but de décrire l'implémentation des deux refactorings mentionnés au chapitre précédent.

Dans un premier temps, l'outil, nommé Raincode© , qui a été utilisé va être présenté de manière générale. Ensuite, les algorithmes définis pour implémenter ces deux refactorings vont être expliqués. Par après, quelques remarques sur le code seront effectuées. Enfin, des exemples d'utilisation du Rename Method et Remove Parameter seront montrés.

6.1 Généralités sur Raincode

Raincode[1] est un outil qui permet d'effectuer différents types d'actions sur du code source représenté sous la forme d'un arbre de dérivation annoté.

On retrouve parmi ces actions : l'insertion, la suppression ou le remplacement de parties de code, le calcul de métriques mesurant certaines propriétés du code ou encore l'identification de parties de code non-conformes à certaines normes, etc.

Un langage de script est mis à disposition pour effectuer ces actions. Ce langage de script a une syntaxe très proche du Pascal et possède les structures des langages procéduraux habituelles. Il permet :

- La déclaration et l'utilisation de variables et constantes. Le langage étant typé dynamiquement, aucune spécification de type ne doit être réalisée.
- L'utilisation d'opérateurs arithmétiques usuels et d'opérateurs spécifiques.
- La déclaration et l'utilisation de procédures [2].
- L'utilisation de structures itératives et conditionnelles.

Aussi, il est possible de manipuler deux structures de données différentes :

- Les listes qui sont aussi indexables.

[1] http ://www.raincode.com/

[2] Le mécanisme de récursion est utilisable.

– Les tables associatives qui sont composées d'une série de paires nom-valeur (c.f. `struct` en C).

Voici, maintenant, comment se déroule le processus d'application d'un script sur une source Delphi.

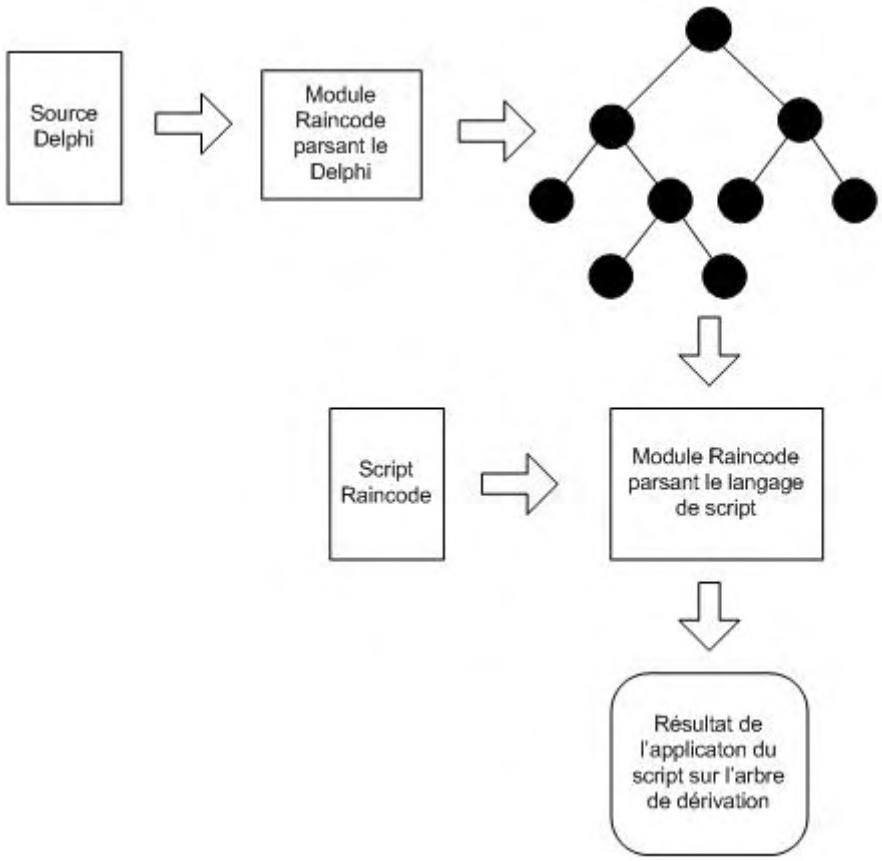

FIG. 6.1 – Processus d'application d'un script sur une source.

Pour illustrer ce qui vient d'être dit, voici un exemple démonstratif de l'utilisation de cet outil.

Considérons l'unité Delphi suivante. Elle contient 1 procédure seule nommée `hello` et 2 classes (`class1` et `class2`) contenant chacune 1 méthode.

```
unit test;

interface

type
    class1 = class
     procedure m1();
    end;

    class2 = class
     procedure m2();
    end;

    procedure hello();

implementation

    procedure class1.m1();
    begin
     WriteLn('method m1() running...');
    end;

    procedure class2.m2();
    begin
     WriteLn('method m2() running...');
    end;

    procedure hello();
    begin
     WriteLn('procedure hello() running...');
    end;

begin
end.
```

Considérons, maintenant, le script de la figure 6.2. Il est constitué de 2 procédures : `INIT` et `PrintInfo`.

Lorsque ce script va être lancé sur un arbre de syntaxe, la procédure `INIT` va d'abord être exécutée.

La procédure `INIT` comporte 4 exemples illustrant le langage de script.

```
PROCEDURE INIT;

VAR
        l;
BEGIN

    OUT.WriteLn(' ----------------------------------------');-- Exemple 1
    IF ROOT IS Unit THEN
         OUT.WriteLn(' Unit');
    ELSE
         OUT.WriteLn(' Program');
    END;

    OUT.WriteLn(' ----------------------------------------');-- Exemple 2
    OUT.Write(' Number of classes:');
    l := ROOT.SubNodes | X IS ClassType;
    OUT.WriteLn(#l);

    OUT.WriteLn(' ----------------------------------------');-- Exemple 3
    OUT.WriteLn(' Procedures declarations:');
    l := ROOT.SubNodes | X IS ProcedureHeading;
    FOR e IN l DO
         PrintInfo(e);
    END;

    OUT.WriteLn(' ----------------------------------------');-- Exemple 4
    OUT.Write(' Number of Methods:');
    l := ROOT.SubNodes | X IS ProcedureHeading
                       AND #(X.Ancestors |[Y] Y IS ClassType)>0;
    OUT.WriteLn(#l);

END INIT;

PROCEDURE PrintInfo(node)
BEGIN
        OUT.WriteLn(' -Non Terminal position: line start=', node.LineNr
                    ,' col start=' ,node.ColNr, ' line end=',node.EndLineNr
                    ,' col end=' ,node.EndColNr);

        FOR ln IN PATCH.NtImage(node) DO
                OUT.WriteLn(' ',ln);
        END;
END;
```

FIG. 6.2 – Exemple de script Raincode© .

Dans l'exemple 1, le type noeud se trouvant à la racine de l'arbre de dérivation est testé grâce à l'opérateur `IS`. Dans le cas d'une unité, "Unit" est affiché sinon "Program" est affiché.

Dans le second exemple, le nombre de classes contenues dans la source est calculé. Pour cela, on récupère l'ensemble des noeuds de l'arbre de dérivation sous la forme d'une liste grâce à l'attribut sémantique `SubNodes` et cette liste va être filtrée grâce à l'opérateur `|`. Celui-ci va créer une liste contenant les éléments de la liste située à gauche qui satisfont la condition booléenne située à la droite de l'opérateur (La variable prédéfinie `X` va référencer successivement les noeuds de `ROOT.SubNodes`).

Enfin, la longueur de la liste est affichée grâce à l'opérateur `#`.

Dans l'exemple 3, on récupère tous les noeuds qui sont des déclarations de procédures ou méthodes sans type de retour et pour chacun de ces noeuds, on appelle la routine `PrintInfo`. Cette routine va afficher la position du noeud reçu en paramètre grâce aux attributs sémantiques `LineNr`, `ColNr`, `EndLineNr`, `EndColNr`. Elle va ensuite, grâce à la routine `NtImage` du module `PATCH`[3], afficher la chaîne de caractères du code source que le noeud reçu en paramètre représente.

Dans le dernier exemple, on va calculer le nombre de méthodes[4] contenues dans la source. Donc, on récupère les noeuds de type `ProcedureHeading` mais on vérifie aussi pour chacun d'eux que l'on retrouve un noeud de type `ClassType` dans leurs ancêtres vu qu'une méthode est déclarée *dans* une classe. Ici, `[Y]` permet de spécifier le nom de la variable qui va être utilisée dans la deuxième condition de filtrage.

On obtient le résultat suivant lorsqu'on exécute ce script sur la source Delphi précédente.

```
----------------------------------------
Unit
----------------------------------------
Number of classes:2
----------------------------------------
Procedures declarations:
-Non Terminal position: line start=7 col start=5 line end=7 col end=18
procedure m1()
-Non Terminal position: line start=11 col start=5 line end=11 col end=18
procedure m2()
-Non Terminal position: line start=14 col start=4 line end=14 col end=20
procedure hello()
-Non Terminal position: line start=18 col start=3 line end=18 col end=23
procedure class1.m1()
-Non Terminal position: line start=23 col start=3 line end=23 col end=23
procedure class2.m2()
-Non Terminal position: line start=28 col start=3 line end=28 col end=19
procedure hello()
----------------------------------------
Number of Methods:2
```

6.2 Algorithmes

Dans cette section, les algorithmes des deux refactorings implémentés vont être présentés en méta-langage. Il n'aurait pas été approprié de décrire le code de script directement car ceci aurait introduit beaucoup de détails.

[3]Ce module fournit des procédures qui permettent de modifier le code source sur lequel un script est lancé.

[4]Pour cet exemple, on suppose que les méthodes n'ont pas de type de retour.

Mais, avant de décrire ces algorithmes, une section discutant des éléments que les refactorings manipulent va être introduite.

6.2.1 Domaines des refactorings

Les refactorings et leurs préconditions manipulent des éléments du langage et ceux-ci possèdent des attributs. Pour faciliter et clarifier l'écriture des algorithmes, ces derniers vont être expliqués en terme de ces éléments. Voici la liste des éléments et attributs les concernant qui seront utilisés dans les algorithmes (les éléments sont précédés par un • et les attributs par un ◇) :

- *Program* représente le programme principal de l'application Delphi :
  - ◇ *Units* renvoie la liste des unités utilisées dans l'application.
  - ◇ *Classes* renvoie la liste des classes contenues dans ce programme.
  - ◇ *AloneFuncs* renvoie une liste d'éléments *Function* qui sont les procédures ou fonctions seules déclarées dans ce programme principal.
  - ◇ *FileOwner* renvoie le nom de fichier auquel ce programme appartient.
  - ◇ *InitBlock* renvoie l'élément *Block* représentant l'implémentation du programme.
- *Unit* représente une des unités définies dans l'application :
  - ◇ *Program* renvoie l'élément *Program* utilisant cette unité dans cette application.
  - ◇ *Classes* renvoie la liste des classes contenues dans cette unité.
  - ◇ *AloneFuncs* renvoie une liste d'éléments *Function* qui sont les procédures ou fonctions seules déclarées dans l'unité.
  - ◇ *FileOwner* renvoie le nom de fichier auquel cette unité appartient.
  - ◇ *InitBlock* renvoie l'élément *Block* représentant le code d'initialisation de l'unité.
- *Class* représente une classe :
  - ◇ *MethodsList* renvoie la liste des méthodes (élément *Method*) qui sont définies dans la classe.
  - ◇ *Superclass* renvoie la superclasse si elle existe, void sinon.
  - ◇ *Owner* renvoie le conteneur (*Program* ou *Unit*) de la classe.
- *Method* représente la déclaration une méthode :
  - ◇ *Name* renvoie le nom de la méthode.
  - ◇ *Params* renvoie la liste des paramètres de la méthode (élément *Parameter*).
  - ◇ *Implementation* renvoie un élément *Implementation* qui correspond à l'implémentation de la méthode.
  - ◇ *Owner* renvoie l'élément *Class* contenant la méthode.
  - ◇ *Binding* renvoie le type de liaison de la méthode : virtual, override ou void (void lorsque la méthode est liée statiquement).
  - ◇ *Visibility* renvoie la visibilité de la méthode : public, private ou protected.

- *Implementation* représente l'implémentation d'une méthode ou d'une procédure ou fonction seule [5] :
  - ◇ *Name* renvoie le nom de la méthode.
  - ◇ *Params* renvoie la liste des paramètres de la méthode.
  - ◇ *ImplBlock* renvoie le bloc d'instructions implémentant la méthode.
- *Function* représente une procédure ou fonction seule [6] :
  - ◇ *Name* renvoie le nom de la procédure ou fonction.
  - ◇ *Params* renvoie la liste des paramètres de la procédure ou fonction.
  - ◇ *Implementation* renvoie un élément *Implementation* qui représente l'implémentation de la procédure ou fonction.
- *Parameter* représente un paramètre reçu par une méthode, procédure ou fonction :
  - ◇ Name renvoie le nom du paramètre.
  - ◇ Owner renvoie une référence vers la méthode, procédure ou fonction à laquelle ce paramètre appartient.
- *Block* représente un bloc d'instructions :
  - ◇ *MethodCalls* renvoie la liste des appels de méthodes (élément *MethodCall*) présents dans le bloc d'instructions.
  - ◇ *VarUsed* renvoie la liste de variables (élément *Variable*) utilisées (en lecture ou écriture) dans le bloc d'instructions.
- *MethodCall* représente un appel de méthode :
  - ◇ *Name* renvoie le nom de la méthode appelée.
  - ◇ *Args* renvoie la liste des arguments présents dans l'appel à la méthode.
  - ◇ *Ref* renvoie une référence vers un élément *Method* représentant la déclaration de cette dernière.
- *Variable* représente une variable de l'application :
  - ◇ *Name* renvoie le nom de la variable.
  - ◇ *ParamRef* renvoie une référence vers un élément *Parameter* si cette variable a été reçue en paramètre par la méthode, procédure ou fonction, void sinon.

Donc, pour résumer, une application Delphi est composée d'un programme principal et d'un ensemble d'unités. Chacun de ces éléments contient un ensemble de classes contenant des méthodes et un ensemble de procédures ou fonctions seules. Ces méthodes, procédures et fonctions connaissent les éléments représentant leurs implémentations. Les implémentations sont constituées d'un bloc d'instructions.

De manière technique, chaque élément sera, dans le cadre de ce mémoire, représenté par un type de noeud particulier de l'arbre de dérivation. Par exemple, *Class*

[5] Pour rappel, en Delphi, le code d'implémentation des méthodes, procédures ou fonctions n'est pas défini lorsqu'elles sont déclarées (voir page 22).

[6] On aurait pu, ici, faire une distinction entre les procédures et fonctions mais il n'était pas utile de créer 2 éléments exactement identiques hormis leur nom.

sera représenté par un noeud déclarant une classe (i.e. `ClassType`), *Method* par un noeud déclarant une méthode (i.e. `ProcedureHeading` ou `FunctionHeading`), etc. Les attributs de ces éléments seront, quant à eux, représentés par des attributs sémantiques attachés à ces noeuds[7]. Aussi, la notation $a.b$ sera utilisée pour représenter le fait que l'on utilise l'attribut b de l'élément a.

Il faut, cependant, souligner le fait qu'il existe d'autres approches qui visent à représenter ces éléments et attributs autrement que par des symboles de variables et attributs sémantiques. S.Tichelaar [14] [15] récupère des informations de l'arbre de dérivation annoté pour créer un méta-modèle (nommé FAMIX) qui est la représentation du programme sous la forme d'un schéma entités-relations. Ceci lui permet de travailler indépendamment du langage.

Une autre approche est celle de Mens et al. [16] [17] qui transforment, grâce à un plugin pour l'outil Fujaba[8], l'arbre de dérivation en un graphe spécifique possédant divers types de noeuds pour les divers éléments à traiter. Cette représentation sous forme de graphe leur permet de spécifier les refactorings de manière formelle et d'analyser leurs propriétés.

6.2.2 Rename Method (Method m, String newname)

De manière globale, l'algorithme du Rename Method sera constitué de 2 grandes étapes :

- Vérifier les préconditions.
- Si les préconditions sont satisfaites, récupérer la liste des méthodes à renommer et de tous les appels à ces méthodes et les renommer.
  Sinon, afficher un message d'erreur.

Vérifier les préconditions

La précondition 1 (page 58) nécessite dans un premier temps de connaître la méthode redéfinie par m de la plus haute superclasse de la classe contenant m, appelons la *m'*. Donc, si m est `virtual` ou liée statiquement, on se trouve déjà sur cette plus haute méthode (*m'* vaut m) sinon, dans le cas `override`, une procédure `GetRootVirtualMethod(Method m)` retournant cette méthode doit être utilisée avec m comme argument :

```
GetRootVirtualMethod(Method m)
  SuperC := Method.Owner.Superclass;
  Tant Que SuperC <> void Faire
```

[7] Ceci n'est pas tout à fait ce qui a été réalisé dans l'implémentation. Ceci est explicité dans la section suivante.

[8] http ://www.fujaba.de/

```
  Pour tout m1 de Superclass.MethodList
   Si m1.Binding = virtual Et IsSignatureMatching(m,m1) Alors
    Retourner m1 ;
   FinSi
  FinPour
  SuperC := SuperC.Superclass ;
 FinTantQue
 Retourner void ;
```

Donc, l'idée est de remonter de classe en classe dans l'hiérarchie jusqu'à ce que l'on découvre une méthode avec la même signature que m et qui soit `virtual`.

`IsSignatureMatching(m1,m2)` est une procédure renvoyant vrai si `m1` et `m2` ont le même nom et la même liste de paramètres, faux sinon.

Ensuite, il faut vérifier que :

1. la classe c contenant m' ne contienne pas déjà une méthode avec une signature composée de *newname* et de la même liste de paramètres que m'
2. c n'hérite pas d'une méthode ayant une signature composée de *newname* et de la même liste de paramètres que m'
3. les sous-classes de c ne contiennent pas de méthode ayant une signature composée de *newname* et de la même liste de paramètres que m'

Une procédure `CheckNameConflict(Method m, String newname)` va réaliser cette vérification. Elle renvoie faux si on détecte un conflit vrai sinon. On lui donne m' et *newname* comme arguments :

```
CheckNameConflict(Method m, String newname)
 Pour tout m1 de m.Owner.MethodsList
  Si m1 <> m Alors
   Si IsNewNameConflicting (m1, m, newname) Alors
    Retourner faux ;
   FinSi
  FinSi
 FinPour

 Pour tout m1 de InheritedMethod(m.Owner)
  Si IsNewNameConflicting (m1, m, newname) Alors
   Retourner faux ;
  FinSi
 FinPour
```

```
Pour tout s de SubClassesThatCanInherit(m)
 Pour tout m1 de s.MethodsList
  Si IsNewNameConflicting (m1, m, newname) Alors
   Retourner faux ;
  FinSi
 FinPour
FinPour

Retourner vrai ;
```

En supposant que `InheritedMethod` renvoie les méthodes héritées depuis la racine de l'hiérarchie par la classe donnée en argument et que `SubClassesThatCanInherit` retourne les sous-classes qui peuvent hériter de la méthode passée en argument, cette procédure remplit bien son rôle.

`IsNewNameConflicting(m1,m2,newname)` est une procédure renvoyant vrai si `m1` et `m2` ont la même liste de paramètres et que `m1` a comme nom `newname`, faux sinon.

Détaillons, maintenant, `InheritedMethod` :

```
InheritedMethod(Class c)
 Si c.Superclass = void Alors Res := {} ;
 Sinon Res := InheritedMethod(c.Superclass) ;
       Pour tout m dans c.Superclass.MethodsList
        Si(m.Visibility <> private
           Ou m.Owner.Owner.FileOwner = c.Owner.FileOwner)Alors
         Res := Res ∪ {m} ;
        FinSi
       FinPour
 FinSinon
 Retourner Res ;
```

Cette procédure est récursive. Le cas de base est celui d'une classe n'ayant pas de superclasse. Dans ce cas, la classe n'hérite d'aucune méthode.

Dans le cas général, les méthodes héritées d'une classe seront les méthodes héritées par sa superclasse et les méthodes de sa superclasse qui ne sont pas de visibilité privée ou étant dans le même fichier [9].

Il reste, enfin, à détailler `SubClassesThatCanInherit`.

[9] Pour rappel, la visibilé en Delphi est décrite dans le chapitre 2.

```
SubClassesThatCanInherit(Method m)
  Res := {} ;
  Si m.Visibility = private Alors
   Pour tout s dans SubclassesOf(m.Owner)
    Si(s.Owner.FileOwner = m.Owner.Owner.FileOwner)Alors
     Res := Res ∪ {s} ;
    FinSi
   FinPour
  Sinon Res := SubclassesOf(m.Owner) ; FinSinon
  Retourner Res ;
```

Donc, si `m` n'est pas privée, toutes les sous-classes peuvent en hériter. Dans le cas où `m` est privée, seules les sous-classes contenues dans le même fichier que la classe contenant `m` peuvent en hériter.

Mais, pour que ceci fonctionne, il faut que `SubclassesOf` retourne toutes les sous-classes d'une classe.

```
SubClassesOf(Class c)
  Res := {} ;
  S := DirectSubClassesOf(c) ;
  Tant que S ≠ {} Faire
   x := RemoveFirst(S) ;
   Res := Res ∪ {x} ;
   S := S ∪ DirectSubClassesOf(x) ;
  FinTantQue
  Retourner Res ;
```

Dans cet algorithme, `RemoveFirst` permet de retirer le premier élément de la liste passée en argument et de retourner ce premier élément. `S` permet de connaître les sous-classes qui sont en cours de traitement. A chaque itération, on retire la première sous-classe de `S`, on l'ajoute au résultat et on ajoute ses sous-classes à `S` ; `DirectSubClassesOf` renvoyant les classes filles de la classe passée en argument.

```
DirectSubClassesOf(Class c)
  Res := {} ;
  Pour tout c1 dans GetAllClasses()
   Si(c1.Superclass = c)Alors
    Res := Res ∪ {c1} ;
   FinSi
  FinPour
  Retourner Res ;
```

`GetAllClasses()` renvoie la liste de toutes les classes de l'application : celles du programme principal et celles des unités.

Ceci termine les explications de la première précondition (page 58). Pour la précondition 2, il suffit de définir une procédure `CheckName(String s)` qui va vérifier si `s` est lexicalement conforme à la manière de définir un nom de méthode en Delphi. Dans l'implémentation, ceci a été réalisé avec la procédure `Match(s,e)` du module `LEX` de Raincode qui permet de vérifier que la chaîne `s` est conforme à l'expression régulière `e`.

Récupérer les méthodes à renommer et tous les appels à ces méthodes et les renommer

Le but est, ici, de récupérer les éléments *Method* qui doivent être renommés et les appels à ces méthodes. On connaît déjà la méthode la plus haute dans l'hiérarchie qui est redéfinie par m (pour rappel, m'). Elle a été obtenue grâce à `GetRootVirtualMethod`. Il ne reste plus qu'à obtenir les méthodes `override` qui la redéfinissent et qui se situent dans les sous-classes de la classe contenant m'. La procédure `GetOverrider` a été définie à cet effet :

```
GetOverrider(Method m)
  Res := {};
  Pour tout s dans SubClassesThatCanInherit(m)
   Pour tout m1 dans s.MethodsList
    Si IsSignatureMatching(m,m1)
        And m1.Binding = overrride Alors
      Res := Res ∪ {m1};
    FinSi
   FinPour
  FinPour
  Retourner Res;
```

Cet algorithme utilise les procédures `SubClassesThatCanInherit` et `IsSignatureMatching` définies précédemment pour récupérer les méthodes redéfinissant m'.

On possède, maintenant, les méthodes à renommer (m' et celles qui la redéfinissent). Pour faciliter les notations, supposons qu'elles soient contenues dans une liste `MethodsToRename`. Il ne reste, donc, plus qu'à récupérer les appels à ces méthodes.

Pour ce faire, une procédure `GetMethodsCalls` va être utilisée :

```
GetMethodsCalls(List MethodsToRename)
  Res := {};
  Pour tout mc dans GetAllMethodCalls()
   Si Contains(mc.Ref, MethodsToRename) Alors
    Res := Res ∪ {mc};
   FinSi
  FinPour
  Retourner Res;
```

La procédure `Contains(e,l)` renvoie vrai si l'élément `e` est contenu dans la liste `l`.

Donc, seuls les appels de méthodes qui concernent celles qui sont passées dans la liste en paramètre seront renvoyés. Il reste à détailler `GetAllMethodCalls` qui va renvoyer l'ensemble des appels de méthodes de l'application [10].

```
GetAllMethodCalls()
  P := GetTheApplicationProgram();
  MethodsCalls := P.InitBlock.MethodCalls;
  Pour tout f dans P.AloneFuncs
   MethodsCalls := MethodsCalls ∪ f.Implementation.ImplBlock.MethodCalls
  FinPour
  Pour tout u dans P.Units
   MethodsCalls := MethodsCalls ∪ u.InitBlock.MethodCalls
   Pour tout f dans u.AloneFuncs
    MethodsCalls := MethodsCalls ∪ f.Implementation.ImplBlock.MethodsCalls
   FinPour
  FinPour
  Pour tout c dans GetAllClasses()
   Pour tout m dans c.MethodsList
    MethodsCalls := MethodsCalls ∪ m.Implementation.ImplBlock.MethodsCalls
   FinPour
  FinPour
```

Donc, on va récupérer les éléments *MethodCall* des divers blocs d'instructions de l'application. Ces blocs sont soit le corps du programme principal, soit le corps du bloc d'initialisation des unités, soit le corps des procédures ou fonctions seules ou soit le corps des méthodes des classes de l'application.

[10] On doit considérer, ici, *tous* les appels de méthodes car les méthodes à renommer sont susceptibles d'être appelées de n'importe où dans l'application (si non privées).

Ici, `GetTheApplicationProgram` renvoie l'élément représentant le programme principal de l'application.

Maintenant que l'on connaît les méthodes à renommer et les appels à ces méthodes, il ne reste plus qu'à modifier l'attribut *Name* des éléments de types *Method* et *Implementation* contenus dans `MethodsToRename` et des éléments de type *MethodCall* récupérés grâce à `GetMethodsCalls`.

6.2.3 Remove Parameter(Parameter p)

Le schéma global est le même que pour le Rename Method.

Vérifier les préconditions

La précondition 1 (page 60) du Remove Parameter demande dans un premier temps de retrouver toutes les méthodes qui vont être concernées par ce refactoring.

On va d'abord utiliser la procédure `GetRootVirtualMethod` avec `p.Owner` comme paramètre. Cet appel nous donne la méthode *m'* la plus haute dans l'hiérarchie et redéfinie par *p.Owner*. Ensuite, on récupère les méthodes redéfinissant *m'* grâce à `GetOverrider`. Comme pour le Rename Method, donnons un nom à cet ensemble de méthodes (*m'* et les méthodes qui la redéfinissent) : `MethodsToModify`.

Maintenant, il faut vérifier que les blocs implémentant ces méthodes ne contiennent pas de variables qui réfèrencent le paramètre en position *pos* de la liste de paramètres de chacune de ces méthodes, *pos* étant la position de p dans la liste de paramètres de sa méthode (i.e. `p.Owner`).

Définissons une procédure `CheckParamNotUsed` qui va effectuer cette vérification. Elle renvoie faux si on détecte l'utilisation de p ou d'un des paramètres correspondants, vrai sinon.

```
CheckParamNotUsed(List MethodsToModify, Integer pos)
  Pour tout m dans MethodsToModify
    Pour tout v dans m.Implementation.ImplBlock.VarUsed
      Si IsAtPos(v.Ref, m.Implementation.Params, pos) Alors
        Retourner faux ;
      FinSi
    FinPour
  FinPour
  Retourner vrai ;
```

`IsAtPos(e,l,pos)` retourne vrai si l'élément `e` se situe en position `pos` dans la liste `l`, faux sinon.

La précondition 2 est très semblable à la précondition 1 du Rename Method. La recherche des méthodes qui peuvent créer des conflits est la même que dans `CheckNameConflict` mais, au lieu d'utiliser la procédure `IsNewNameConflicting(m1, m2, s)`, on utilise une procédure `IsParamRemovedConflicting(m1, m2, pos)` renvoyant vrai si `m1` et `m2` ont la même signature lorsque l'on retire le paramètre en position `pos` de la liste des paramètres de `m2`, faux sinon.

Récuperer les méthodes auxquelles on doit retirer un paramètre et le retirer

On possède déjà l'ensemble des méthodes à modifier, elle sont contenues dans `MethodsToModify`. Ensuite, on récupère les appels à ces méthodes avec la procédure définie précédemment `GetMethodsCalls` avec `MethodsToModify` comme argument.

Finalement, on doit retirer les paramètres en position *pos* des listes de paramètres des méthodes contenues dans `MethodsToModify` et de leur implémentation (éléments de type *Implementation*). Il faut faire de même avec les arguments situés en position *pos* des appels de méthodes récupérés par `GetMethodsCalls`.

6.3 Le code

Le code implémentant ces deux refactorings s'inspire des algorithmes définis ci-dessus. Il n'aurait pas été approprié de décrire ce code de script tellement la grammaire Delphi est volumineuse et ceci aurait impliqué beaucoup de détails.

C'est la raison pour laquelle les grandes idées ont été expliquées en méta-langage en utilisant une représentation abstraite éléments-attributs pour les symboles de variables et attributs sémantiques.

Le code est composé de 3 fichiers : Rename-beta0.7.rcs qui contient les procédures spécifiques au Rename Method, RemoveParameter-beta0.4.rcs contenant celles spécifique au Remove Parameter et DelphiUtil.rcs contenant des procédures communes aux deux refactorings.

Une copie du code est fournie dans les documents annexes. Pour pouvoir le comprendre complètement, il faut utiliser la grammaire du module Raincode parsant le Delphi. Elle est aussi fournie dans les documents annexes.

Comme dit précédemment, tous les attributs décrits dans la section précédente ne sont pas représentés par des attributs sémantiques dans le code. Dans le cas où l'information n'est pas disponible sur le noeud directement, des routines permettant de rechercher cette information ont été écrites. Par exemple, `GetSuperClass` pour

l'attribut *Superclass* de l'élément *Class*, `GetOwningClass` pour l'attribut *Owner* de l'élément *Method*, etc.

6.3.1 Inputs

Dans les explications des refactorings et algorithmes, on a vu que certains des inputs pouvaient être des éléments : le Rename Method reçoit un élément *Method* et le Remove Parameter un élément *Parameter*.

Cependant, le code a été pensé pour que les refactorings puissent être aisément intégrés dans un environnement de développement. L'idée, pour les utiliser, serait :

- Dans le cas du Rename Method, que l'utilisateur positionne le curseur de la souris sur la méthode qu'il veut renommer et spécifie un nouveau nom de méthode.
- Dans le cas du Remove Parameter, que l'utilisateur positionne le curseur sur le paramètre qu'il veut retirer.

Donc, pour réaliser ceci, les scripts reçoivent la ligne du curseur et la colonne du curseur en lieu et place des éléments Method et Parameter.

Dans les deux cas, des procédures ont été écrites pour vérifier que la position du curseur est réellement sur une méthode ou sur un paramètre de méthode.

6.3.2 Hypothèses de travail

Les deux refactorings implémentés ne fonctionnent pas avec tous les types de sources Delphi. On ne peut appliquer ces refactorings :

- sur des fichiers `.dpr`. Ceci est dû au fait que le module Raincode pour le Delphi, appelé DelphiRc, n'effectue pas d'analyse sémantique sur ce type de source. Or, les attributs sémantiques sont nécessaires pour pouvoir exécuter l'un des deux refactorings.

- sur plusieurs fichiers. Le script du Rename Method ne fonctionnera pas, par exemple, si une des méthodes à renommer dans l'hiérarchie se trouve dans une classe qui est dans un fichier différent du fichier contenant la méthode que l'on voulait renommer au départ.
  Dans la grammaire Delphi utilisée par DelphiRc, le symbole de variable `Ident` représente un identificateur qui peut être le nom d'une classe, d'une méthode, d'une variable, etc. Celui-ci possède un attribut sémantique `Ref` qui pointe vers le symbole de variable représentant la déclaration de l'objet que cet `Ident` représente.
  Dans la version actuelle de DelphiRc, cet attribut `Ref` n'est pas calculé pour les noeuds `Ident` qui réfèrent à un objet (méthode, variable, paramètre) déclaré

dans un autre fichier. Cet attribut est nécessaire car on doit pouvoir connaître les déclarations des méthodes impliquées dans le refactoring.
Il faut aussi remarquer que cette limitation fait tomber les problèmes de visibilité puisque toutes les classes de l'application Delphi sur laquelle on veut appliquer un script doivent être contenues dans le même fichier.

- sur des méthodes surchargées. Ceci signifie, dans le cas du Remove Parameter, qu'on ne peut supprimer un paramètre d'une méthode surchargée.
  L'attribut `Ref` des noeuds `Ident` n'est pas calculé entièrement pour les méthodes surchargées. Ceci parce DelphiRc n'effectue pas de synthèse de type et donc, il n'est pas possible de connaître les déclarations de méthodes surchargées en utilisant `Ref` puisqu'elles se différencient par les types de leurs paramètres.
  Le fait que l'on ait pas accès aux types des variables et expressions implique, par ailleurs, que les préconditions n'ont pas pu être implémentées en utilisant la signature d'une méthode comme elle a été définie précédemment.
  En effet, au lieu de comparer les signatures de deux méthodes en utilisant leur nom et les types de leurs paramètres, on les compare en utilisant leur nom et les nombres de paramètres de chacune d'elle. Ceci est plus restrictif et empêche certains cas qui sont valides de se produire.
  Par exemple, supposons qu'une classe contienne une méthode `m(a : Integer ; b :Integer)` et qu'une de ses sous-classe contienne une méthode `n (a : Boolean ; b :String)`. Actuellement, le script du Rename Method refusera de renommer `n` en `m` alors que ce cas est normalement valide.

- sur des méthodes avec des paramètres initialisés. Ceci simplifie l'implémentation des préconditions. Une méthode avec des paramètres initialisés est une méthode qui, lorsqu'on l'invoque, n'est pas obligée de recevoir autant d'arguments qu'elle n'a de paramètres mais seulement des arguments pour ses paramètres non initialisés. Par exemple, `m(a : Integer = 5 ; b : Boolean)` peut être invoquée avec `o.m(true)` si `o` est un objet de la classe définissant `m`.

6.3.3 Transformations

L'implémentation des transformations au niveau du Rename Method a été réalisée, lorsque l'on a récupéré tous les noeuds `ProcedureHeading` ou `FunctionHeading` représentant les éléments *Method* et *Implementation* et les noeuds `FunctionCall` représentant les éléments *MethodCall*, en utilisant la routine `ReplaceNt(n,s)` du module `PATCH` sur le noeud `Ident` représentant le nom de ces méthodes ou appels à celles-ci. Cette routine permet de remplacer la chaîne de caractères que représente le noeud `n` par la chaîne de caractères `s`.

Au niveau du Remove Parameter, une procédure calculant les positions des paramètres des méthodes concernées et des arguments des appels à ces méthodes avec leur séparateur respectif (';' et ',') et leur spécification de type a été définie. Une fois ces positions obtenues, la routine `Delete(line, col, endl, endc)` du module `PATCH` est utilisée. Plusieurs cas sont envisagés : si l'argument est seul, on ne retire pas de séparateur ; si l'argument est dernier, on retire le séparateur qui le précède ; sinon on retire l'argument et le séparateur le suivant. Aussi, il a fallu gérer les cas des listes de paramètres de même type car, en Delphi, `m( a, b : Integer)` est équivalent à `m( a : Integer ; b : Integer)`.

6.4 Exemples d'utilisation

Dans cette section, des exemples illustrant les scripts implémentés vont être présentés. Dans tous ces exemples l'unité Delphi de la figure 6.3, contenue dans `Exemple.pas`, va être utilisée.

Elle est constituée de 4 classes. `B` hérite de `A` et `C` et `D` héritent de `B`. La méthode `meth` définie dans `B` est redéfinie dans `C` et `D`. Toutes les méthodes sont vides sauf les méthodes `meth` de `B` et `C`. La méthode `meth` de cette dernière effectue un appel à la méthode qu'elle redéfinit de `B` grâce au mot clé `inherited` (similaire à `super` en Java). Un objet de type `C` est instancié dans le bloc initialisant l'unité et on lui envoie le message `meth` avec les arguments `1`, `2+4` et `3`.

Cette exemple n'effectue pas vraiment de tâche intéressante mais va permettre d'illustrer les 2 refactorings implémentés.

6.4.1 Rename Method

Nous allons, dans un premier temps, lancer le script sans paramètre[11]. La ligne de commande suivante (tapée dans une console) :
`> Delphirc.exe :Script=Rename-beta0.7.rcs Exemple.pas --` donne le résulat montré sur la figure 6.4.

Lançons de nouveau ce script avec les paramètre `1 1 a`. La position $(1, 1)$ ne représente pas un nom de méthode. Le résultat est visible sur la figure 6.5.

Illustrons maintenant la précondition 2 vérifiant que le nom de méthode fourni est correct. Les paramètres suivants sont fournis au script `9 15 ?-0- ?`. La position $(9, 15)$ représente la méthode `meth` déclarée dans la classe `B`. On peut visualiser le résultat sur la figure 6.6.

[11] Les paramètres fournis au script doivent se situer apres les `--`.

```
unit Exemple;
interface
type
   A = class
    procedure a(i : Integer; j: Integer; k:Integer); end;

   B = class(A)
    procedure b(i : Integer; j: Integer; k:Integer);
    procedure meth (i : Integer; j: Integer; k:Integer);virtual; end;

   C = class(B)
    procedure meth (i,j: Integer; k:Integer):override; end;

   D = class(B)
    procedure d(i : Integer; j: Integer; k:Integer);
    procedure meth (i : Integer; j: Integer; k:Integer):override; end;
implementation
  procedure A.a(i : Integer; j: Integer; k:Integer); begin  end;
  procedure B.meth(i : Integer; j: Integer; k:Integer);
  begin k := k + 1; end;
  procedure B.b(i : Integer; j: Integer; k:Integer); begin  end;
  procedure C.meth(i : Integer; j: Integer; k:Integer);
  begin inherited meth(1,2,k) end;
  procedure D.d(i : Integer; j: Integer; k:Integer); begin  end;
  procedure D.meth(i : Integer; j: Integer; k:Integer); begin  end;
var
  cc : C;
begin
  cc := C.Create;
  cc.meth(1,2+4,3);
end.
```

FIG. 6.3 – Unité Delphi contenue dans `Exemple.pas`.

```
Usage: DelphiRc.exe Rename-beta0.7.rcs source.pas -- x y newName
where source.pas is the source file you want to use RenameMethod on,
(x,y) the mouse cursor location in source.pas and newName the name
you want to give to the method pointed by the mouse.
```

FIG. 6.4 – Informations d'utilisation du Rename Method.

```
Rename Method refactoring (beta 0.7 version):

Checking position ...
The current cursor position is not on a method! (code B)
```

FIG. 6.5 – Véfirication de la position.

```
Rename Method refactoring (beta 0.7 version):

Checking position ...
The current cursor position IS on a method (code G)
Checking preconditions ...
The new name: ?-0-? is not a valid ident name
regular expression:(_|[a-zA-Z])((|[0-9a-zA-Z])|_)*
Preconditions Checks Not Passed!
```

FIG. 6.6 – Illustration de la préconditions 1.

Les 3 prochains exemples illustrent la précondition 1 du Rename Method. Essayons d'abord de renommer la méthode `meth` de `B` en `a`. Comme expliqué dans le chapitre 5, ceci n'est pas une application valide car un objet de type `B` (ou d'un type descendant) sur lequel on invoque la méthode `a` de `A` avant le refactoring, invoquerait la méthode `a` de `B` après le refactoring. Le résultat est représenté sur la figure 6.7.

```
Rename Method refactoring (beta 0.7 version):

Checking position ...
The current cursor position IS on a method (code G)
Checking preconditions ...
Check Failed: there is already an inherited method with the same signature
Method:
ProcedureHeading 5 5 5 51
procedure a(i : Integer; j: Integer; k:Integer)

In class:
TypeDecl 4 5 4 56
A = class        procedure a(i : Integer; j: Integer; k:Integer); end

Preconditions Checks Not Passed!
```

FIG. 6.7 – Illustration d'un conflit avec une méthode héritée.

Ensuite, renommons cette même méthode en `b`. Ceci n'est pas valide non plus car deux méthodes avec des signatures identiques seraient contenues dans la classe `B`. Le résultat est visible sur la figure 6.8.

```
Rename Method refactoring (beta 0.7 version):

Checking position ...
The current cursor position IS on a method (code G)
Checking preconditions ...
Check Failed : There is already a method with the same signature in the class

Method:
ProcedureHeading 8 8 5 52
procedure b(i : Integer; j: Integer; k :Integer)

In class:
TypeDecl 7 9 4 68
B = class(A)     procedure b(i : Integer; j: Integer; k :Integer);     procedur
e meth (i : Integer; j: Integer; k:Integer);virtual; end

Preconditions Checks Not Passed!
```

FIG. 6.8 – Illustration d'un conflit avec une méthode de la même classe.

Pour terminer l'illustration de cette précondition, envisageons le cas plus subtil représenté sur la figure 6.9. Renommons la méthode `meth` de `C` en `d`. Ceci n'est pas non plus autorisé car des problèmes de remplacement involontaire avec `B` [12] et de double définition dans `D` se présenteraient.

```
Rename Method refactoring (beta 0.7 version):

Checking position ...
The current cursor position IS on a method (code G)
Checking preconditions ...
Check Failed: there is already a method with the same signature in a sub class

Method:
ProcedureHeading 15 15 5 51
procedure d(i : Integer; j: Integer; k:Integer)

In class:
TypeDecl 14 16 4 69
D = class(B)     procedure d(i : Integer; j: Integer; k:Integer);     procedure
meth (i : Integer; j: Integer; k:Integer);override; end

Preconditions Checks Not Passed!
```

FIG. 6.9 – Illustration d'un conflit avec une méthode de la sous-classe `D`.

Cet exemple termine l'illustration des préconditions. Illustrons, maintenant, le cas où le Rename Method renomme effectivement une méthode. Pour cela, nous renommerons la méthode `meth` de `B` en `MyNewName`. Les informations concernant les modifications sont affichées dans la console sur le figure 6.10. Le fichier transformé est comparé sur la figure 6.11 avec le fichier original à l'aide l'outil CsDiff[13]. Les

[12] Si un objet de type `D` invoquait la méthode `meth` de `B` avant le refactoring, cet objet invoquerait la méthode `d` de `D` après.

[13] http ://www.componentsoftware.com/Products/CSDiff/

caractères rouges barrés sont ceux qui ont disparus dans le fichier transformé. Les caractères bleus sont ceux qui n'existaient pas dans le fichier original.

```
Rename Method refactoring (beta 0.7 version):

Checking position ...
The current cursor position IS on a method (code C)
Checking preconditions ...
Preconditions succeeded

Begin the rename application

Here is the location of the references to the method(s)
and method calls to be updated:

ProcedureHeading 9 9 6 64
procedure meth (i : Integer; j: Integer; k:Integer);virtual

ProcedureHeading 12 12 6 54
procedure meth (i,j: Integer; k:Integer);override

ProcedureHeading 16 16 6 65
procedure meth (i : Integer; j: Integer; k:Integer);override

ProcedureHeading 19 19 4 55
procedure B.meth(i : Integer; j: Integer; k:Integer)

ProcedureHeading 22 22 4 55
procedure C.meth(i : Integer; j: Integer; k:Integer)

ProcedureHeading 25 25 4 55
procedure D.meth(i : Integer; j: Integer; k:Integer)

FunctionCall 23 23 20 30
meth(1,2,k)

FunctionCall 30 30 4 19
cc.meth(1,2+4,3)
```

FIG. 6.10 – Informations concernant l'application du Rename Method.

On voit clairement que les 3 méthodes impliquées dans le Rename Method ont été renommées, ainsi que leur implémentation et les appels à ces méthodes.

6.4.2 Remove Parameter

Illustrons la précondition 1 du Remove Parameter. Pour cela, nous allons exécuter le script de ce refactoring avec la ligne commande suivante :
`>Delphirc.exe :Script=RemoveParameter-beta0.4.rcs Exemple.pas -- 9 46`
La position $(9, 46)$ représente la position du paramètre `k` de la méthode `meth` de `B`. Le résultat est présenté sur la figure 6.12.

Le paramètre `k` étant utilisé dans les méthodes `meth` de `B` et `C`. L'application du refactoring est refusée.

```
unit Exemple;
interface
type
   A = class
    procedure a(i : Integer; j: Integer; k:Integer); end;

   B = class(A)
    procedure b(i : Integer; j: Integer; k:Integer);
    procedure methMyNewName (i : Integer; j: Integer; k:Integer);virtual; end;

   C = class(B)
    procedure methMyNewName (i,j: Integer; k:Integer);override; end;

   D = class(B)
    procedure d(i : Integer; j: Integer; k:Integer);
    procedure methMyNewName (i : Integer; j: Integer; k:Integer);override; end;
implementation
  procedure A.a(i : Integer; j: Integer; k:Integer); begin  end;
  procedure B.methMyNewName(i : Integer; j: Integer; k:Integer);
  begin k := k + 1; end;
  procedure B.b(i : Integer; j: Integer; k:Integer); begin  end;
  procedure C.methMyNewName(i : Integer; j: Integer; k:Integer);
  begin inherited methMyNewName(i,2,k) end;
  procedure D.d(i : Integer; j: Integer; k:Integer); begin  end;
  procedure D.methMyNewName(i : Integer; j: Integer; k:Integer); begin  end;
var
  cc : C;
begin
  cc := C.Create;
  cc.methMyNewName(1,2+4,3);
end.
```

FIG. 6.11 – Comparaison du fichier d'origine et du fichier transformé.

La précondition 2 du Remove Parameter étant similaire à la précondition 1 du Rename Method, nous allons tout de suite illustrer l'application du Remove Parameter sur le paramètre `j` de la méthode `meth` de `B`. L'illustration est présente sur la figure 6.13.

On voit clairement que les paramètres des méthodes impliquées et les arguments des appels à ces méthodes sont supprimés avec leur séparateur adéquat et leur spécification de type si nécessaire.

```
Remove Parameter refactoring (beta 0.4 version):

Checking position ...
The current position IS on a method parameter. (code 1)
Checking preconditions ...
Parameter used:
Ident 20 20 9 9
k

In:
RegularBlock 20 20 3 23
begin k := k * 1; end

Parameter used:
Ident 20 20 14 14
k

In:
RegularBlock 20 20 3 23
begin k := k * 1; end

Parameter used:
Ident 23 23 28 28
k

In:
RegularBlock 23 23 3 33
begin inherited meth(1,2,k) end

Preconditions Checks Not Passed!
```

FIG. 6.12 – Illustration de la précondition 1 du Remove parameter.

```
unit Exemple;
interface
type
   A = class
    procedure a(i : Integer; j: Integer; k:Integer); end;

   B = class(A)
    procedure b(i : Integer; j: Integer; k:Integer);
    procedure meth (i : Integer; j: Integer; k:Integer);virtual; end;

   C = class(B)
    procedure meth (i,j: Integer; k:Integer);override; end;

   D = class(B)
    procedure d(i : Integer; j: Integer; k:Integer);
    procedure meth (i : Integer; j: Integer; k:Integer);override; end;
implementation
  procedure A.a(i : Integer; j: Integer; k:Integer); begin  end;
  procedure B.meth(i : Integer; j: Integer; k:Integer);
  begin k := k + 1; end;
  procedure B.b(i : Integer; j: Integer; k:Integer); begin  end;
  procedure C.meth(i : Integer; j: Integer; k:Integer);
  begin inherited meth(i,2,k) end;
  procedure D.d(i : Integer; j: Integer; k:Integer); begin  end;
  procedure D.meth(i : Integer; j: Integer; k:Integer); begin  end;
var
  cc : C;
begin
  cc := C.Create;
  cc.meth(1,2+4,3);
end.
```

FIG. 6.13 – Illustration de l'application du Remove parameter sur `j`.

Chapitre 7

Conclusion

Outre le chapitre 1 résumant le paradigme orienté objets, le chapitre 2 introduisant la partie orientée objets de Delphi et le chapitre 3 donnant une vue générale des refactorings, nous avons vu un processus d'automatisation de ces derniers.

Au chapitre 4, nous avons vu comment transformer le code source, grâce aux techniques de compilation, en une structure utilisable pour pouvoir appliquer un refactoring.

Au chapitre 5, deux refactorings ont été étudiés en détails ainsi que leurs préconditions.

Finalement, l'explication de l'implémentation de ces deux refactorings pour le Delphi a été réalisée au chapitre 6, en décrivant notamment les algorithmes de ces refactorings en méta-langage, en expliquant l'outil utilisé, nommé Raincode© et en donnant des informations sur le code source des scripts. Ce chapitre se termine en illustrant l'utilisation de ces refactorings.

J'aimerais mentionner, par ailleurs, ce que le mémoire m'a apporté personnellement :

- Je me suis perfectionné de manière non négligeable dans le paradigme orienté objets.
- J'ai appris le langage Delphi sans jamais avoir vraiment réalisé d'application importante. Je me suis aussi amélioré dans la connaissance des structures syntaxiques de langages importants comme le Delphi.
- J'ai appris l'utilisation de l'outil Raincode et de son langage de script et de manière plus générale, la philosophie de ce type d'outil.
- J'ai abordé un domaine grandissant du reengineering : les refactorings.

Enfin, concernant les perspectives, le mémoire offre plusieurs opportunités :

- Pour enrichir l'information disponible sur l'arbre de dérivation fourni par DelphiRc, il pourrait être intéressant de réaliser une analyse sémantique complète.

Une synthèse de type pourrait être envisagée ainsi que le calcul complet de l'attribut `Ref` des noeuds `Ident`.

- Possédant plus d'informations, il serait intéressant d'améliorer les refactorings pour qu'ils puissent fonctionner avec plusieurs fichiers, des méthodes surchargées, des fichiers `.dpr`. On pourrait, aussi, relaxer les préconditions pour qu'elles n'empêchent pas des cas valides de se produire (voir section 6.3.2).
- Il serait intéressant, par ailleurs, d'implémenter plusieurs autres refactorings pour les intégrer tous dans un outil de développement.
- Une autre opportunité serait d'étudier l'implémentation de refactorings génériques [27] fonctionnant pour tout langage orienté objets. Le fait que le langage de script soit typé dynamiquement pourrait permettre de définir des procédures génériques réalisant ceci.
- Enfin, une dernière opportunité pourrait être de calculer les attributs internes de qualité (voir section 3.5) lors de l'application de refactorings et d'étudier leurs évolutions. Ceci permettrait d'aider à évaluer l'amélioration de la qualité de la structure du code lorsqu'un refactoring est appliqué.

Annexe A

Complexité cyclomatique

Cet exemple est destiné à illustrer la complexité cyclomatique introduite au chapitre 3.

La complexité cyclomatique $V(G)$ est définie comme le nombre de chemins linéairement indépendants dans le graphe de flux d'exécution G d'une méthode. C'est aussi une borne inférieure sur le nombre de tests à effectuer pour s'assurer que toutes les instructions ont été exécutées.

Pour la calculer, une fois le graphe de flux construit, $V(G) = e - n + 2$ où e est le nombre d'arrêtes et n le nombre de noeuds. Un autre moyen est de compter le nombre de *régions* dans le graphe de flux d'exécution.

Voici un exemple de code en méta-langage.

```
(1) Tant que a Faire
(2)     Si b Alors
(3)         <Bloc d'instructions S1>
(4)     Sinon
(5)         <Bloc d'instructions S2>
(6)     Fin Si
(7) Fin Tant que
```

On peut lui associer un graphe de flux (représenté, ici, par la figure A.1).

Un graphe de flux d'exécution est un graphe orienté dont :

- Chaque noeud représente une série d'instructions.
- Chaque arrête représente le branchement possible d'une série d'instructions vers une autre.
- Certaines arrêtes, quand plusieurs branchements sont possibles, sont étiquetées par une condition qui induit le changement de flux d'exécution.

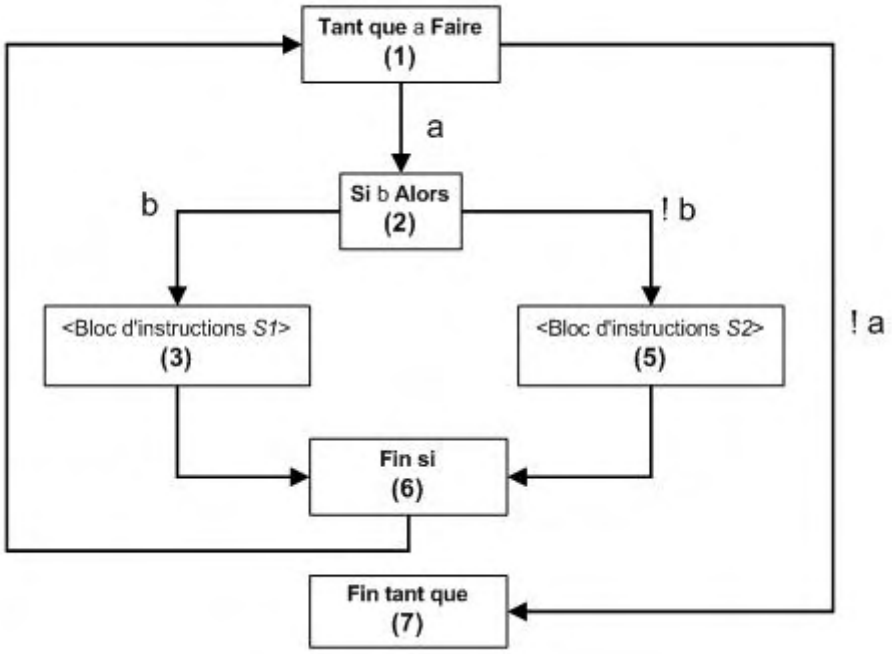

FIG. A.1 – Exemple de graphe de flux d'exécution

Ce type de graphe possède un noeud initial et un noeud final, un noeud initial n'ayant pas d'arrête entrante et un noeud final n'ayant pas d'arrête sortante.

Dans l'exemple présenté ci dessus, la complexité cyclomatique vaut 3 puisque le graphe de flux comporte 7 arrêtes et 6 noeuds.

Par ailleurs, on observe 3 régions sur la figure A.2.

Enfin, on entend par chemin, une séquence de noeuds démarrant du noeud initial et se terminant par le noeud final. Le fait que ces chemins doivent être linéairement indépendants est à comprendre au sens algébrique du terme. En fait, ils forment une base des chemins possibles dans le graphe et on peut, à partir de ces chemins, en générer n'importe quel autre. Les 3 chemins linéairement indépendants que l'on pourrait choisir, ici, sont [17], [123617] et [125617]. On peut générer [1236125617] par la combinaison linéaire suivante [123617] + [125617] − [17] .

Cette métrique nous donne donc un indice sur la complexité du code. En effet, plus la complexité cyclomatique est importante et plus le code sera complexe au niveau du flux d'exécution.

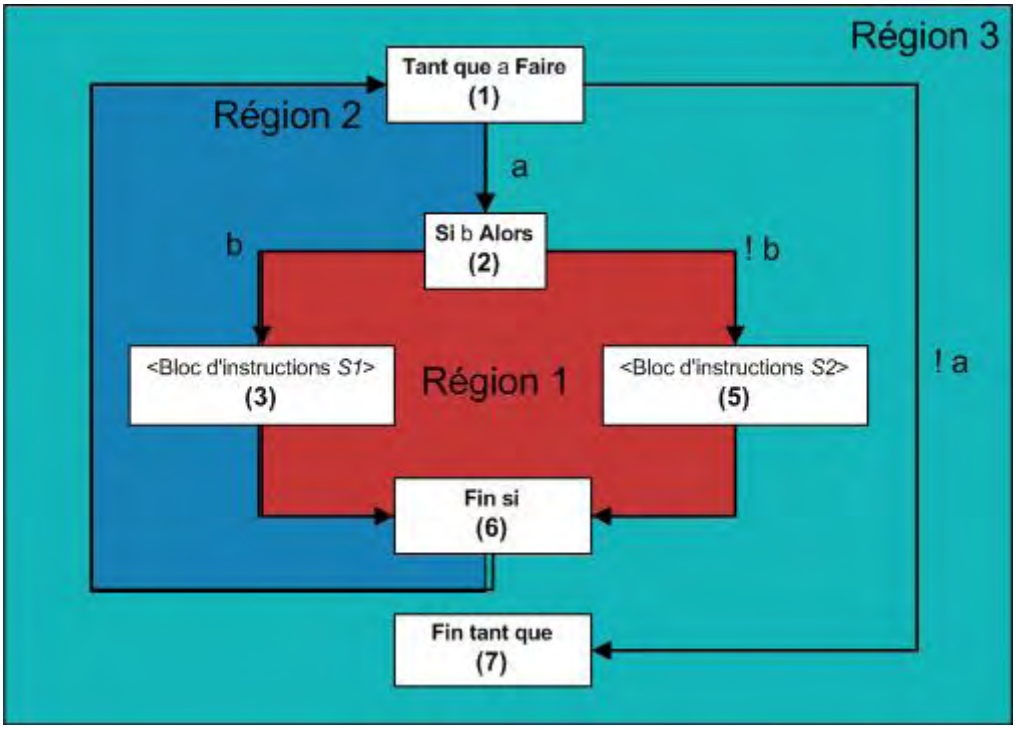

FIG. A.2 – Représentation des régions dans le graphe de flux

Bibliographie

[1] *L'orienté objet - Cours et exercices en UML, Java, C# et C++* de H. Bersini et I. Wellesz
Edition eyrolles, 2002. ISBN : 2-212-11108-8

[2] *Introduction à la programmation objet en Java : Cours et exercices* de J. Brandeau
Edition Dunod, 1999. ISBN : 2-100-04106-1

[3] *Delphi in a Nutshell* de R. Lischner
Edition O'Reilly, 2000. ISBN : 1-565-92659-5

[4] *Programmer en Turbo Pascal 7* de C. Delannoy
Edition Eyrolles, 2002. ISBN : 2-212-08986-4

[5] *Software Project Management* de B. Hugues, M. Cotterell
McGraw Hill, 2002. ISBN : 0-077-09834-X

[6] *Program Evolution : Processes of Software Change* de M.M. Lehman, L.A. Belady
Academic Press Professional, 1985. ISBN : 0-124-42440-6

[7] *On understanding Laws, Evolution, and Conversation in the Large-Program Life Cycle*
M.M. Lehman
The Journal of System and Software, vol. 1, p. 213-221, 1980.

[8] *An instroduction to Software Restructuring*
R.S. Arnold 1986.
Tutorial on Software Restructuring

[9] *Refactoring object-oriented frameworks*
W. Opdyke
PhD dissertation, University of Illinois at Urbana-Champaign, 1992.

[10] *Refactoring : Improving the Design of Existing Code* de M. Fowler
Addison-Wesley, 1999. ISBN : 0-201-48567-2

[11] *Reverse Engeneering and Design Recovery : A Taxonomy*
E.J. Chikofsky & J.H. Cross
IEEE Software, vol. 7, p.13-17

[12] *Identifying Refactoring Opportunities Using Logic Meta Programming*
T. Mens & T. Tourwé
Proc. European Conf. Software Maintenance and Reengineering, p. 91-100, 2003.

[13] *A Survey of Software Refactoring*
T. Mens & T. Tourwé
IEEE Transactions on software engineering, vol. 30, p.126-139, 2004.

[14] *A Meta-model for Language-Independent Refactoring*
S. Tichelaar, S. Ducasse, S. Demeyer et O. Nierstrasz

[15] *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*
S. Tichelaar
PhD dissertation, université de Bern, 2001.

[16] *Formalising Refactorings with Graph Transformations*
T. Mens, N. Van Eetvelde, D. Janssens, S. Demeyer
Journal of Software Maintenance and Evolution, p.1001-1025, 2004.

[17] *Formalising Behaviour Preserving Program Transformations*
T. Mens, S. Demeyer, D. Janssens

[18] *Metrics Based Refactoring*
F. Simon, F.Steinbrückner and C. Lewerentz
Proc. European Conf. Software Maintenance and Reengineering, p. 30-38,2001.

[19] *Refactoring To Patterns* de J. Kerievsky
Addison-Wesley, 2004. ISBN : 0-321-21335-1

[20] *Design Patterns : Elements of Reusable Object-Oriented Software* de E.gamma, R.Helm, R.Johnson, J.Vlissides
Addison-Wesley, 1995. ISBN : 0-201-63361-2

[21] *A Complexity Measure*
T. J. McCabe
IEEE Transactions Software Engineering, Vol. SE-2, N° 4, 1976.

[22] *Compilers : Principles, Techniques and Tools* de A.V. Aho, R. Sethi et J.D. Ullman
Addison-Wesley, 1988. ISBN : 0-201-10088-6

[23] *"Test First World"*
J. U. Pipka
Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering, 2002.

[24] *Semantics Preserving Transformation Rules for Prolog*
M. Proietti et A. Pettorossi
Proc. Symp. Partial Evaluation and Semantics-Based Program Evaluation, vol. 26, no. 9, p. 274-284, 1991.

[25] *Software Metrics : A Rigorous and Pratical Aproach, Revised* de N.E. Fenton et S.L. Pfleeger
Course Technology, 2ème édition (1998). ISBN : 0-534-95425-1

[26] *An efficient context-free parsing algorithm* de J.Earley
Comm. *ACM* 13 :2, p. 94-102.

[27] *Towards Generic Refactoring*
Ralf Lämmel