REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

N° d'ordre : .....

UNIVERSITE DE M'SILA

FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

Département d'Informatique

MEMOIRE de fin d'étude

Présenté pour l'obtention du diplôme de MASTER

Domaine : Mathématiques et Informatique
Filière : Informatique
Spécialité : Systèmes d'Informations Avancés

Par : MEHOUES Ahmed

SUJET

Un compilateur de diagramme de classes UML pour la génération des cas de test basant sur ANTLR

Soutenu publiquement le : 19 / 06 /2014 devant le jury composé de :

Mr BENAHCENE Rachid	**Université de M'sila**	**Président**
Mr. MOKHTARI Rabah	**Université de M'sila**	**Rapporteur**
Mr BOUBAKIRE Mohamed	**Université de M'sila**	**Examinateur**

Promotion : 2013 /2014

Remerciements

Je remercie Allah, tout puissant, de nous avoir donné la force pour survivre, ainsi que l'audace pour dépasser toutes les difficultés.

J'aimerais remercier vivement mon encadreur MEKHTARI Rabah, leurs conseils et surtout leurs sens pédagogiques m'ont permis de trouver la force de mener à bout ce mémoire.

Je remercie pour la gentillesse et spontanéité avec lesquelles vous avez bien voulu dirigé ce travail.

J'adresse également tous mes remerciements à tous les membres de jury qui ont accepté de juger mon travail.

Mes remerciements aussi à tous mes enseignants de département informatique de l'université de M'sila.

Je remercie s'adressent aussi à mes aimes et mes collègues de l'université de M'sila en particulier la promotion de deuxième années master informatique spécialité Systèmes d'Informations Avancés.

J'exprime ma profonde reconnaissance à toutes les personnes qui ont contribué de près ou de loin pour le bon déroulement de ce travail.

Encore merci à tous.

Dédicaces

A mes parents,

A ma famille,

A mes amis,

A mes collègues…

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

Introduction générale

La compilation est l'un des domaines les plus riches et les plus passionnants de l'informatique. Pour autant c'est aussi l'un des plus difficiles à aborder de par la diversité et la complexité des problématiques qu'il englobe et la quantité de notions théoriques qu'il fait intervenir .

ANTLR (ANother Tool for Language Recognition), est un environnement de développement des compilateurs et de traducteurs qui génère des analyseurs lexicaux et des analyseurs syntaxiques descendants, utilisant des descriptions grammaticales écrites en Java, C#, Python ou C++. Il possède des primitives de constructions d'arbres syntaxiques abstrait (AST).

L'émergence de l'approche objet a été considérée comme la solution ultime à la crise du logiciel dans les années 1970. La technologie objet, grâce à sa maturité a permis de mieux appréhender le développement d'applications complexes et d'améliorer leur qualité à différents points de vue.

Le test est pour la vérification si le système répond aux spécifications et mettre en évidence les défauts.

Des nombreux travaux ont été faite sur l'utilisation des diagrammes UML pour la génération de cas de test, par exemple on présente une approche permettant de générer des cas de test à partir du diagramme UML (Proposal of a Method to Support Testing for Java Programs with UML)[11] ,cette approche utilise les diagrammes de classe ayant une syntaxe, semblable à celle du langage Java, pour générer des cas de test .

La génération de cas de test à partir des diagrammes de classes reste un domaine peu exploité.

Le présent mémoire a pour objectif d'utiliser une nouvelle approche basée sur les informations contenues dans les diagrammes de classes UML(classes, attributs, méthodes, association,…) et générateur le ANTLR (parcoure l'arbre AST), Elle permet la génération de cas de test.

En utilisant Antlr, nous voulons générer et implémenter un compilateur de diagramme de classe pour la génération automatique des cas de test. Notre compilateur sert à vérifier :

- l'absence de définition d'une classe ;
- les attributs ;
- méthodes ; et

- les relations entre les classes.

Nous avons organisé notre mémoire en quatre chapitres :

Le Première chapitre présente le domaine des compilateurs et ces principes et les différentes étapes de compilation.

Le deuxième chapitre présente les notions et concepts de base de diagramme de classes. Ainsi que les types des tests.

Le troisième chapitre présente le générateur ANTLR et ces déférentes caractéristiques.

Le quatrième chapitre sert à présente la grammaire de diagramme de classes, et les étapes et l'architecture de l'application

Une conclusion qui clôtura notre travail de recherche en citant les objectifs, les résultats et les difficultés rencontrées.

Chapitre I

Introduction aux compilateurs

I.1 INTRODUCTION

Un langage informatique structuré est en fait une notation permettant de décrire des expressions et des textes à l'intention des humains et des ordinateurs. Le monde que nous connaissons s'appuie sur des langages de différents types (Java, XML, HTML, UML…) pour différents objectifs. Souvent, on aura besoin d'aller d'un langage à un autre pour transformer, translater, ou traduire un texte d'une structure à une autre. La réalisation automatique d'une telle translation est appelé une compilation. Ainsi, l'outil logiciel qui exécute cette action est le compilateur.

Dans ce chapitre, nous décrivons les différentes formes de compilateur, puis nous esquissons la structure typique d'un compilateur et présentons les tendances actuelles des définitions des langages de programmation et de l'architecture des ordinateurs qui façonnent les compilateurs.

I.2 Définition d'un compilateur

Définition

Un compilateur est un programme qui lit un texte, rédigé dans un langage de programmation, le langage source, qui le traduit en un programme équivalent rédigé dans un autre langage, le langage cible [1].

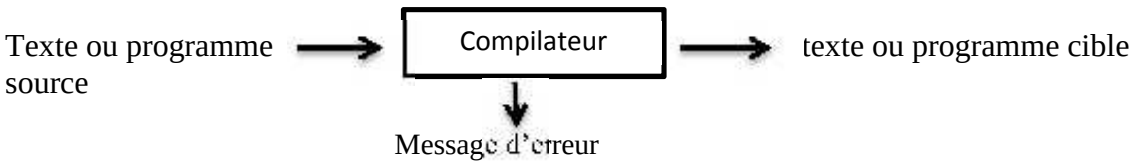


Figure I.1 Schéma d'un compilateur [1].

I.3 Structure d'un compilateur

La compilation se décompose en deux phases

Une phase d'analyse : qui va reconnaître les variables, les instructions, les opérateurs et élaborer la structure syntaxique du programme ainsi que certaines propriétés sémantiques [1].

Une phase synthèse : construit le programme cible désiré, à partir de la représentation intermédiaire et des informations de la table de symbole.

La phase d'analyse est fréquemment dénommée phase frontale du compilateur, la synthèse étant la phase finale [1].

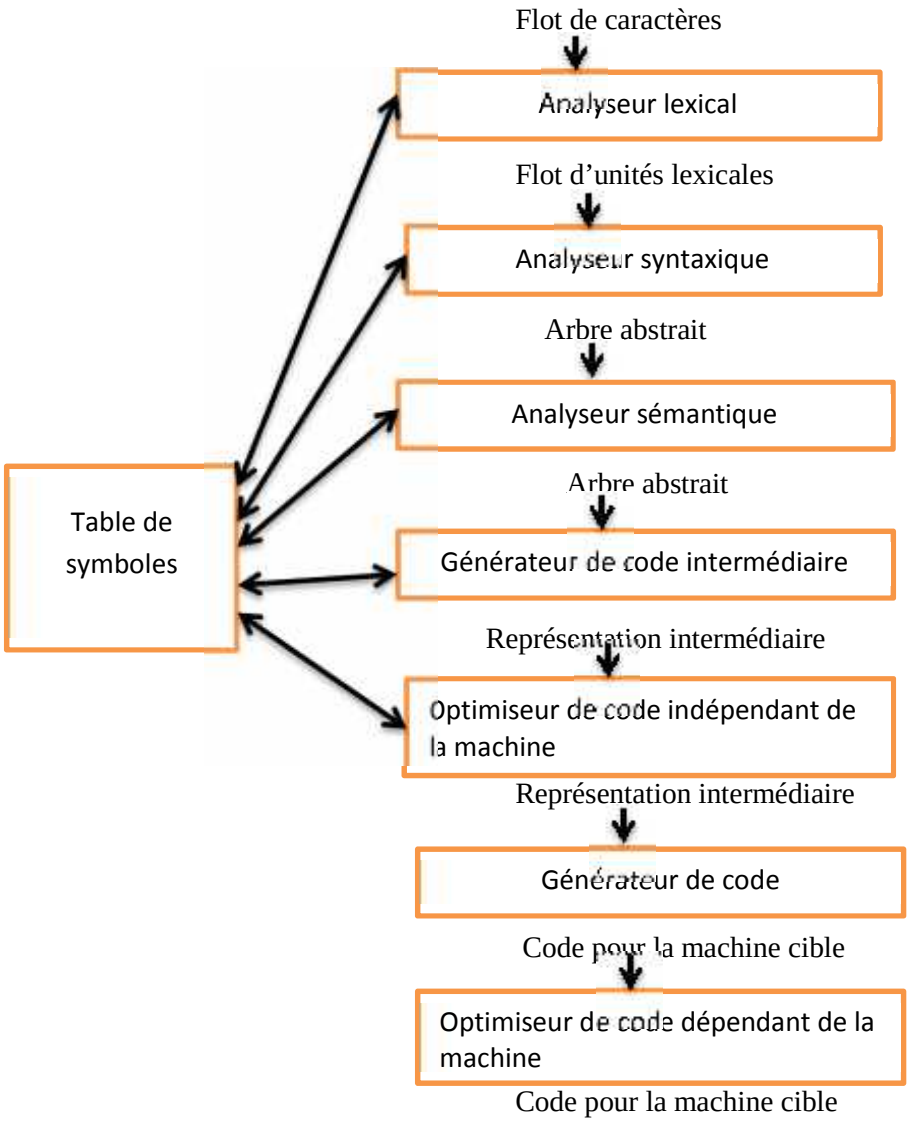


Figure I.2 Phases d'un compilateur [1].

I.3.1 Analyse lexical

La première phase d'un compilateur est appelée analyse lexical, l'analyseur lexical lit le texte source, il est susceptible d'effectuer d'autres tâches que l'indentification des léxèmes.

L'une de ces tâches est l'élimination des commentaires et des blancs (espace, tabulation, fins de ligne, et éventuellement d'autres caractères utilisés pour séparer les unités lexicales dans l'entrée).une autre tâche est dédiée à la corrélation des messages d'erreurs émis par le compilateur, dans certains compilateur, l'analyseur lexical construit une copie de la source avec les messages d'erreurs insérés aux positions adéquates [1].

I.3.1.1Unité lexical, motifs et lexèmes

Une unité lexicale est une suite de caractères qui a une signification collective.

Un motif est une description de la forme que les lexèmes d'une unité lexicale donnée peuvent prendre.

Un lexème est une séquence de caractères dans le programme source qui est reconnue par le motif d'une unité lexical et est identifiée par l'analyseur lexical comme une instance de cette unité lexicale [1].

Exemple :

Position = initiale + vitesse * 60

1. Position est un lexème auquel on fait correspondre l'unité lexical « (id,1) » id est l'identificateur et 1 référence l'entrée de la table de symboles associée à cette position
2. Le symbole d'affectation « = » est un lexème auquel correspondre l'unité lexical(=)
3. « initiale » est un lexème auquel correspondre l'unité lexical (id,2), 2 référence l'entrée de la table de symboles associée à initiale
4. « + » est un lexème auquel correspondre l'unité lexical (+)
5. vitesse = est un lexème auquel correspondre l'unité lexical (id,3),3 référence l'entrée de la table de symboles associée à vitesse
6. « * » est un lexème auquel correspondre l'unité lexical(*)
7. « 60 » est un lexème auquel correspondre l'unité lexical (60)

(id,1) (=) (id,2) (+) (id,3) (*) (60)

I.3.2 Analyse syntaxique

Un Analyseur syntaxique prend en entrée des unités lexicales issues d'un analyseur lexical et traite les noms des unités lexicales comme des symboles terminaux d'une grammaire non contextuelle.

L'analyseur syntaxique construit alors un arbre d'analyse pour la séquence d'unités lexicales, l'arbre d'analyse peut être construit soit abstracto (en parcourant les étapes de dérivation correspondantes) [1].

I.3.2.1 Grammaire non contextuelles

une grammaire est un ensemble de symboles terminaux (les entrées), un autre ensemble de non terminaux (les symbole représentant des constructions syntaxiques) et un ensemble de production, chacune indiquant un moyen selon lequel des chaines représentant par certains autres non terminaux [3].

Une production est constituée d'une partie gauche (le nom terminaux à remplace) et d'une partie droite (la chaine de symboles grammaticaux qui sert à remplacer la partie gauche).

I.3.2.1.1 Les symboles terminaux [1]

- **a.** Les lettres minuscules du début de 'alphabet, telles que a,b et c
- **b.** Les symboles d'opérateurs tels que +,*, etc
- **c.** Les signes de ponctuation tels que les parenthèses, la virgule, etc.
- **d.** Les chiffres 0,1,…….9.
- **e.** Les chaine en caractères gras telles que **id** ou **si**, chacune d'entre elles représentant un symbole terminal unique.

I.3.2.1.2 Les symboles non terminaux [1]

- **a.** Les lettres majuscules du début de l'alphabet, telles que A, B et C.
- **b.** La lettre S, qui quand elle est utilisée, est généralement le symbole de départ.
- **c.** Les mots en minuscules et en italique tels que *expr* ou *instr*.
- **d.** Pour parler de constructions de langages de programmation, on pourra utiliser des lettres majuscules pour représenter des non terminaux dénotant ces constructions.

I.3.2.1.3 Dérivations

Le processus qui part du non terminaux de départ d'une grammaire et qui remplace successivement la partie gauche de l'une de ses productions par sa partie droite est appelé une dérivation, si c'est toujours le non terminaux le plus gauche (respectivement le plus à droite) qui remplace, on parle de dérivation gauche (respectivement droite) [1].

I.3.2.1.4 Arbre d'analyse

Un arbre d'analyse est une représentation graphique d'une dérivation, dans laquelle il y a un nœud pour chaque non terminal qui apparaît dans la dérivation.

Les feuilles sont étiquetées par des terminaux et les nœuds intérieurs par des non-terminaux et la racine est étiquetée par le symbole de départ de la grammaire, et les fils d'un nœud interne étiqueté par N correspondent aux membres d'un des choix de N, dans l'ordre et les terminaux étiquetant les feuilles correspondent à la suite de lexèmes d'entrée, dans l'ordre [1]

Exemple l'arbre d'analyse de – (**id** + **id)**

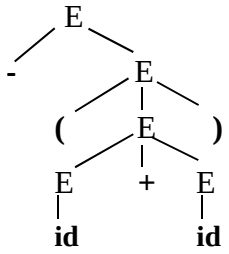


Figure I.3 arbre d'analyse.

I.3.2.1.5 Ambiguïté

Une grammaire ambiguïté est une grammaire qui produit plusieurs dérivations gauches ou plusieurs dérivations droite pour une même phrase [6].

Exemple : id + id * id

Figure I.4 Deux arbre d'analyse pour id + id * i [1].

I.3.2.2 Analyse syntaxique descendante

L'analyse syntaxique descendante comme la construction d'un arbre d'analyse pour la chaine d'entrée de la racine en créant les nœuds de l'arbre d'analyse en suivant son ordre préfixe (parcoure en profondeur) [6].

De façon équivalente, l'analyse syntaxique descendante peut être vue comme la recherche d'une dérivation gauche pour une chaine d'entrée.

I.3.2.3Analyse syntaxique par descente récursive

Un programme d'analyse syntaxique par descente est constitué d'ensemble de procédures, une pour chaque non terminal. La procédure regarde son entrée et décidé quelle production applique à son non terminal.

Les terminaux de la partie droite de la production sont identifiés des appels à leur procédure.

Le retour en arrière, dans les cas où la mauvaise a été choisie, est une possibilité [1].

I.3.2.4 Analyse syntaxique ascendante

Une analyse syntaxique ascendante correspond à la construction d'un arbre d'analyse pour une chaine d'entrée en partant des feuilles(bas) et en remontant jusqu'à la racine (haut) [1].

Exemple id * id

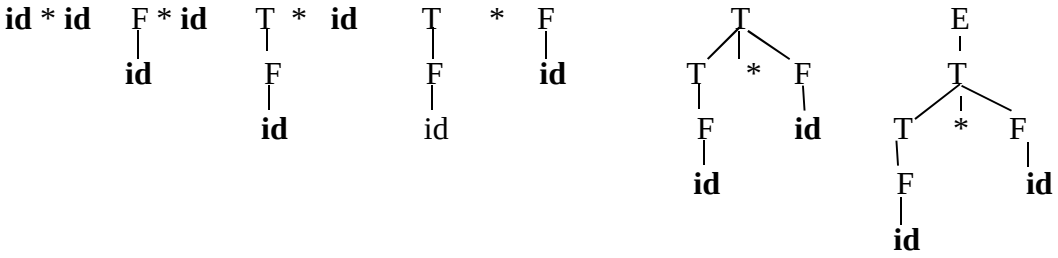


Figure I.5 Analyse ascendante de id * id [1].

I.3.2.5 Réductions

On peut voir l'analyse ascendante comme un processus de réduction d'une chaine vers le symbole de départ de la grammaire. A chaque étape de réduction une sous chaine particulière, reconnue par la partie droite d'une production, est remplacée par le non terminal de la partie gauche de cette production.

Une réduction est l'inverse d'une étape de dérivation.

Le but de l'analyse syntaxique ascendante est donc de construire une dérivation à rebours [1].

I.3.3 Analyse sémantique

L'analyse sémantique utilise l'arbre abstrait et les informations de table des symboles pour vérifier que le programme source est sémantiquement correct vis-à-vis de la définition du langage [1].

Il collecte également des informations sur les types et les conserve, dans l'arbre abstrait ou dans la table de symbole, pour la phase de production du code intermédiaire.

Le contrôle de type est une partie importante de l'analyse sémantique, le compilateur vérifie que les opérandes de chaque opérateur sont valides.

Par exemple, de nombreux langages de programmation requièrent que l'indice d'un tableau soit un entier, le compilateur doit donc signaler une erreur si un tableau est indicé par un nombre en virgule flottante [1].

La spécification du langage peut permettre certaines conversions de type appelées coercitions

Prenons par exemple le cas d'un opérateur arithmétique binaire pouvant être appliqué soit à deux entiers, soit à deux nombre en virgule flottante [1]. Si l'opérateur est appliqué à un nombre flottante et un entier, le compilateur convertie l'entier en un nombre en virgule flottante si cette coercition est prévue par le langage.

I.3.4 Production de code intermédiaire

Au cours de la traduction d'un programme source vers du code cible, le compilateur peut construire plusieurs représentations intermédiaire, de diverses formes. Les arbres abstraits sont une de ces formes de représentation intermédiaire, fréquemment utilisée pour l'analyse syntaxique et sémantique.

A l'issue des analyses syntaxique et sémantique du programme source, de nombreux compilateurs produisent explicitement une représentation intermédiaire de bas niveau (proche de la machine), que l'on peut considérer comme un programme destiné à une machine abstraite.

Cette représentation doit répondre à deux impératifs : être facile à produire et être facile à traduire vers la machine cible [1].

I.3.4.1 Code à trois adresses

Dans le code à trois adresses toute instruction comporte au plus un opérateur dans sa partie droite, on ne peut donc pas y utiliser d'expressions arithmétique complexes.

Une expression du langage source telle que x + y * z doit être traduite en une suite d'instruction à trois adresse, comme la suivante :

t1 = y * z

t2 = x + t1

Où t1 et t2 sont des noms temporaires inventés par le compilateur.

Le code à trois adresses est une représentation linéaire d'un arbre abstrait.

Le code à trois adresse repose sur deux concepts : les adresse et les instructions, en termes de programmation par objet, ces concepts correspondent à des classes et les diverses sortes d'adresses et instructions correspondent à des sous classes appropriées [1].

I.3.5 Optimisation de code

La phase d'optimisation de code indépendante de la machine tente d'améliorer le code intermédiaire afin qu'il en résulte un meilleur code cible. En générale « meilleur » signifie plus rapide, mais on peut viser d'autres objectifs, comme un code final plus court ou consommant peu d'énergie [1].

En optimisant le code, le compilateur commence par réaliser un travail sur le code intermédiaire en [1]:

- Diminuant la taille du code (travail sur les boucles : réduction du nombre de variables et des nombres d'opérations) ;
- Ordonnancement des instructions en fonction des dépendances ;
- Utilisant au mieux les registres du processeur ;
- Déroulant des boucles pour profiter des unités vectorielles des processeurs ;

I.3.6 Production de code

La phase finale de modèle de compilateur est la génération de code, celui-ci prend en entrée la représentation intermédiaire produite par la partie frontale du compilateur, ainsi que les informations pertinentes de la table de symboles et produire un programme sémantiquement équivalent.

Le programme cible doit préserver la signification du programme source et d'être de très bonne qualité, c'est-à-dire utiliser au mieux les ressources de la machine cible.

En outre, le générateur de code lui-même doit s'exécuter rapidement.

Le générateur de code a trois tâches principales : la sélection des instructions, l'allocation et l'assignation de registres, et l'ordonnancement des instructions [1].

- La sélection des instructions choisit les instructions appropriées de la machine cible pour réaliser les instructions de la représentation intermédiaire.
- l'allocation et l'assignation de registres décident quelles valeurs on va garder dans des registres.
- L'ordonnancement des instructions choisit l'ordre dans lequel sera planifiée l'évaluation des instructions.

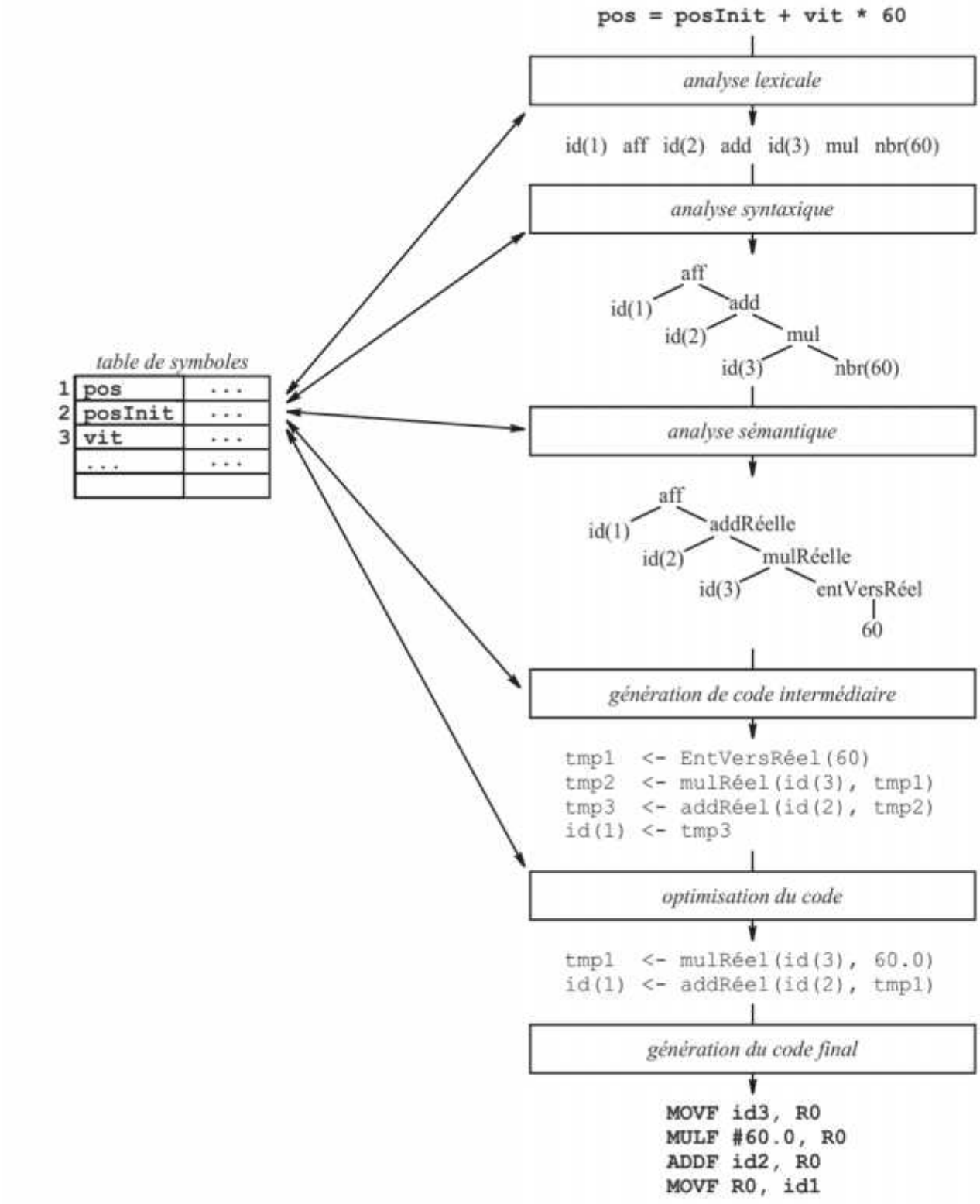


Figure I.6 les phases logiques de la compilation d'une instruction [6].

I.4 Les différents outils de la compilation

Le rôle d'un tel outil est de générer à partir d'une définition de grammaire, un parser qui effectue une analyse durant le parsing.

Dans le cas d'un fichier texte, le paraser acquiert son contenu comme un élément, puis se base sur la grammaire définie pour le décomposer en sous-éléments.

La séquence de ces sous-éléments est analysée sur la base des définitions syntaxiques propres au langage et permit de contrôler sa conformité.

I.4.1 Générateur Lex & Yacc

Le couple Lex (LEXical parser) & Yacc (Yet Another Compiler-Compiler) est le premier essai concluant pour réaliser un méta compilateur utilisable en pratique [4].

- Au niveau lexical:
  - ✓ la génération est complète
  - ✓ les performances sont quasi optimum
  - ✓ le format rigide et la syntaxe ont mal vieilli
  - ✓ les messages d'erreurs sont pauvres ou inexistants
- Au niveau syntaxique
  - ✓ la génération se fait à partir de grammaires sous-classes des LR(1)
  - ✓ les performances obtenues sont optimum
  - ✓ la gestion des erreurs est problématique
- Au niveau sémantique
  - ✓ la traduction est synchronisée par la syntaxe car elle est fondée sur un seul attribut synthétisé**.**

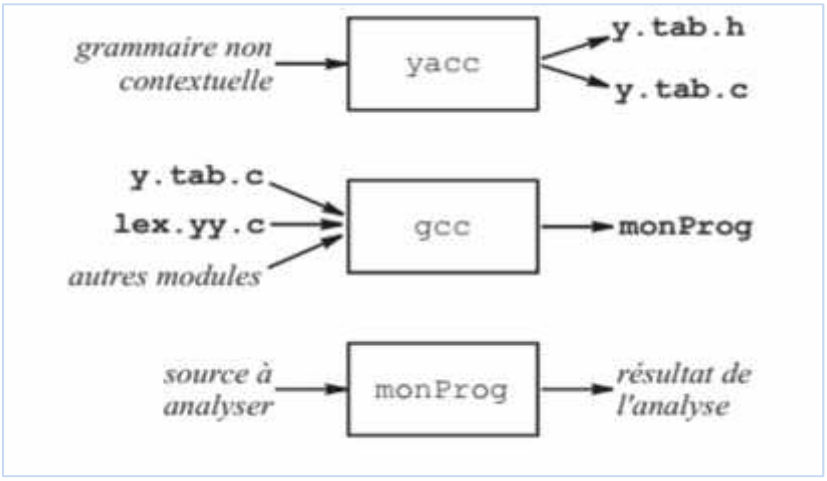


Figure I.7 Utilisation courante de yacc [4].

I.4.2 Générateur JavaCC

javacc est un générateur d'analyseurs syntaxiques descendants développé et maintenu dans les laboratoires Sun MicroSystems. Javacc supporte les grammaires attribuées, autrement dit, du code à exécuter par l'analyseur syntaxique qui doit être généré pour cette grammaire peut y être enchâssé pour effectuer diverses tâches dont la création d'arbres syntaxiques par exemple. Javacc par défaut ne supporte pas la création d'arbres syntaxiques. L'outil JJTree s'occupe de la génération automatique d'arbres syntaxiques avec Javacc [7].

Une grammaire javacc est un fichier dont l'extension est «-jj », et dont la composition est un mélange de définition au format BNF **(Backus–Naur Form)** et de code java, qu'on retrouvera dans le parser généré [7].

I.4.3 Générateur ANTLR

ANTLR (AnotherTool for LanguageRecognigtion) est un générateur d'analyseurs. C'est un outil qui propose un framework pour construire des compilateurs ou des traducteurs à partir de descriptions grammaticales qui peuvent éventuellement contenir des instructions écrites en Java, C++, C#, etc. Un générateur d'analyseurs est un outil qui lit une grammaire en entrée et la convertit en un programme qui peut reconnaître un texte et le traiter suivant les règles de cette grammaire. ANTLR a été développé par Terrence Parr dans le langage Java, mais il peut générer des analyseurs dans un code écrit dans l'un de ses nombreux langages de programmation cibles (Java, Ruby, Python, C, C++, C#). Dans le cadre de ce exemple, nous choisirons Java comme langage cible [8]

En effet, un avantage d'ANTLR sur Javacc est sa capacité de générer des parser dans d'autres langages que Java. Bien que Java soit multiplateforme, il peut être nécessaire dans certains cas d'obtenir un parser dans un langage spécifique pour l'introduire dans un programme existant [10].

I.5 Conclusion :

Dans ce chapitre, nous avons présenté des déférentes étapes de compilation ainsi que les technique de compilation, et les outils de générer les compilateurs, une description détaillées de digramme de classe sera présente dans le deuxième chapitre.

Chapitre II

Diagramme de classes et test

II.1Introduction

Les diagrammes de classes UML représentent un ensemble de classes, d'interfaces et des liens, ainsi que leurs relations. Ce sont les diagrammes les plus fréquents dans la modélisation des systèmes à objets. Ils présentent la vue de conception statique d'un système. La création des tests peut être manuelle ou automatique avec des déférents types de test.

II.2 Les concepts de base :

II.2 .1 Classes et objets

Une classe est la description d'un ensemble d'objet ayant une sémantique. Des attributs, des méthodes, et des relations en commun. Une classe composée d'un nom, d'attributs et d'opérations. Le nom de la classe doit évoquer le concept décrit par la classe. Il commence par une majuscule [2].

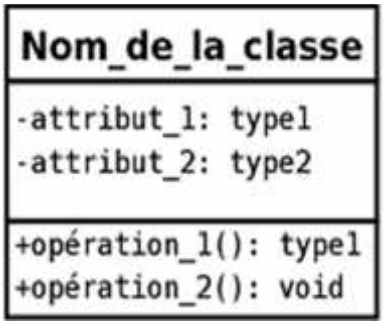


Figure II.1 Représentation UML d'une classe.

Un objet est une instance d'une class

II.2 .2 Les attributs et les opérations

Les attributs définissent des informations qu'une classe ou un objet doi t connaître. Ils représentent les données encapsulées dans les objets de cette classe. Chacune de ces informations est définie par un nom, un type de données, une visibilité et peut être initialisé. Le nom de l'attribut doit être unique dans la classe.

Une opération est une fonction applicable aux objets d'une classe. Une opération permet de décrire le comportement d'un objet. Une méthode est l'implémentation d'une opération. [1]

Chaque méthode est désignée soit seulement par son nom, sa liste de paramètres et son type de résultat. La signature d'une méthode correspond au nom de la méthode et la liste des paramètres en entrée.

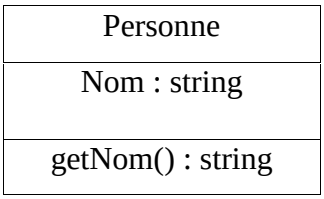


Figure II.2 Méthode getNom() [12].

II.2 .2.1 Visibilité des attributs et opérations

Chaque attribut ou opération d'une classe peut être de type public, protégé, privé ou paquetage. Les symboles + (public), # (protégé), - (privé) et ~ (paquetage) sont indiqués devant chaque attribut ou opération pour signifier le type de visibilité autorisé pour les autres classes [5].

Les droits associés à chaque niveau de confidentialité sont [5]:

• Public (+) Attribut ou opération visible par tous.

• Protégé (#) Attribut ou opération visible seulement à l'intérieur de la classe et pour toutes les sous-classes de la classe.

• Privé (-) Attribut ou opération seulement visible à l'intérieur de la classe.

• Paquetage (~) Attribut ou opération ou classe seulement visible à l'intérieur du paquetage où se trouve la classe.

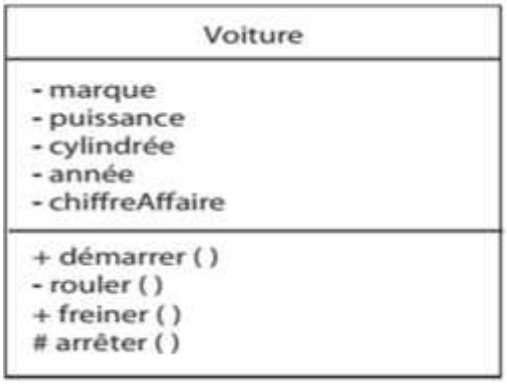


Figure II.3 Exemple de visibilité des méthodes [5].

II.2 .3 Association, multiplicité et navigabilité

II.2 .3.1Association

Un lien est une connexion physique ou conceptuelle entre les instances de classes donc entre objets [5]. Une association décrit un groupe de liens ayant une même structure et une même sémantique. Un lien est une instance d'une association.

Chaque association peut être identifiée par son nom.Une association entre classes représente les liens qui existent entre les instances de ces classes [5].

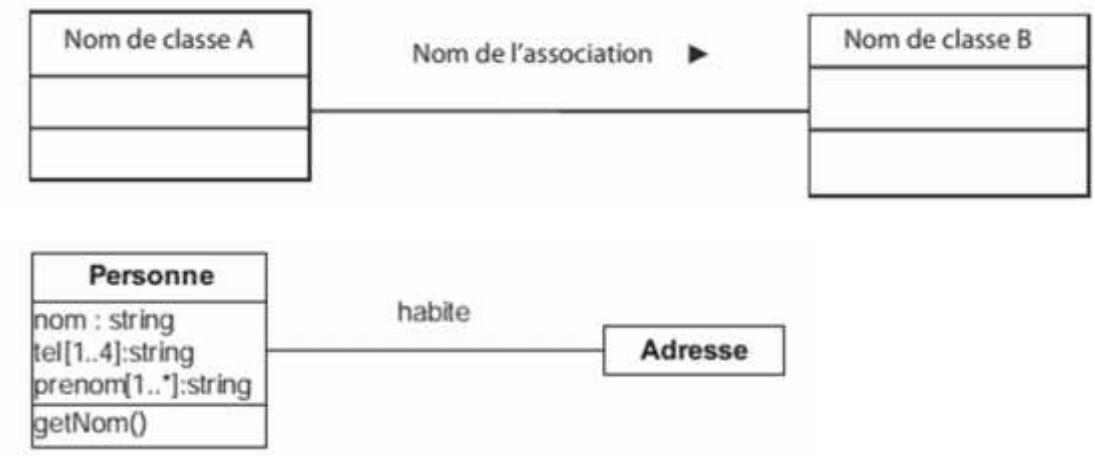


Figure II.4 Exemple d'association [12].

La figure II .4 présente l'association nommée habite, qui associe les classes Personne et Adresse. Cette association signifie que les objets instances de la classe Personne et les objets instances de la classe Adresse peuvent être reliés. En d'autres termes, cela signifie que des personnes habitent à des adresses [12].

II.2.3.1.1 Rôle d'association

Le rôle tenu par une classe vis-à-vis d'une association peut être précisé sur l'association [5].

Figure II.5 Exemple de rôles d'une association [5]

II.2.3.1.2 Associations réflexives :

Une classe peut également être associée avec lui-même, en utilisant une association réflexive. La figure II.6 montre comment une classe de personne pourrait être liée à elle-même. Quand une classe est associée à elle-même, cela ne veut pas dire que par exemple une classe est liée à elle-même, mais une instance de la classe est liée à une autre instance de la classe [2].

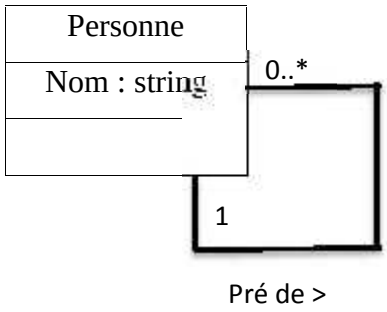


Figure II.6 : Exemple d'une association réflexive [2].

II.2.3.1.3 Multiplicité

La multiplicité indique un domaine de valeurs pour préciser le nombre d'instance d'une classe vis-à-vis d'une autre classe pour une association donnée. La multiplicité peut aussi être utilisée pour d'autres usages comme par exemple un attribut multi-valué. Le domaine de valeurs est décrit selon plusieurs formes [5] :

Tableau II.1 : Les multiplicités

Indicateur	Signification
0..1	zéro ou un
1	Une seule
0..*	zéro ou plus
1..*	Un ou plusieurs
n	Seul n (où n> 1)
0..n	de zéro à n (où n> 1)
1..n	un à n (où n> 1)
m .. n	de m à n (n, m entiers naturels ; n>m)

Dans le cas de multiplicité d'associations, il faut indiquer les valeurs minimale et maximale d'instances d'une classe vis-à-vis d'une instance d'une autre classe [5].

À une instance de A correspond 0 ou 1 instance de B.
À une instance de B correspond0 à nombre non déterminé d'instances de A.

À une instance de A correspond1 à un nombre non déterminé d'instances de B.
À une instance de B correspond 2 à 10 instances de A.

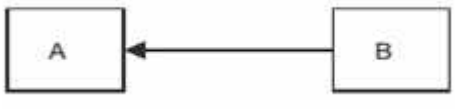
À une instance de A correspond 2 à 4 instances de B.
À une instance de B correspond 1 ou 3 instances de A.

Figure II.7 Exemple de multiplicités [5].

II.2.3.1.4 Navigabilité

La navigabilité indique si l'association fonctionne de manière unidirectionnelle ou bidirectionnelle, elle est matérialisée par une ou deux extrémités fléchées.

Navigabilité unidirectionnelle de A vers B. Pas de navigabilité de B vers A



Navigabilité unidirectionnelle de B vers A. Navigabilité de A vers B

Navigabilité bidirectionnelle entre A et B Habituellement représentée sans flèche

Figure II.8 Représentation de la navigabilité d'association [5]

Par défaut, on admet qu'une navigabilité non définie correspond à une navigabilité implicite.

Figure II.9 Exemple de navigabilité d'une association [5].

Une personne sont associées ses copies d'examen mais l'inverse n'est pas possible (retrouver directement l'auteur de la copie d'examen, notamment avant la correction de la copie).

II.2.3.2 Association de dimension supérieure à 2 et classe-association

Une association de dimension supérieure à 2 se représente en utilisant un losange permettant de relier toutes les classes concernées.
Une classe-association permet de décrire soit des attributs soit des opérations propres à l'association. Cette classe-association est elle-même reliée par un trait en pointillé au losange de connexion. Une classe-association peut être reliée à d'autres classes d'un diagramme de classes [5].

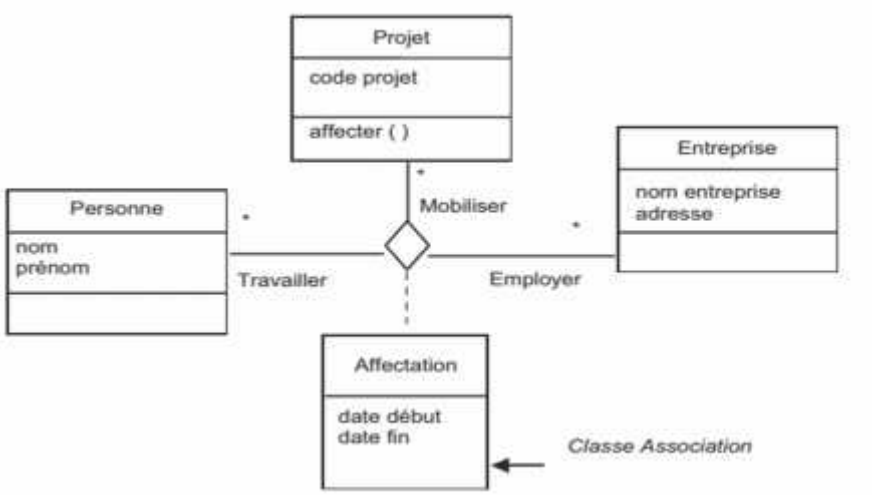


Figure II.10 Exemple d'une association de dimension 3 et d'une classe-association [5].

Un exemple d'une association de dimension 3 comprenant une classe-association « Affectation » est donné à l'exemple. La classe-association Affectation permet de décrire les attributs propres à l'association de dimension 3 représentée.

II.2.4 Agrégation

L'agrégation est une association qui permet de représenter un lien de type « ensemble » comprenant des « éléments ». Il s'agit d'une relation entre une classe représentant le niveau « ensemble » et 1 à n classes de niveau « éléments ». L'agrégation représente un lien structurel entre une classe et une ou plusieurs autres classes [5].

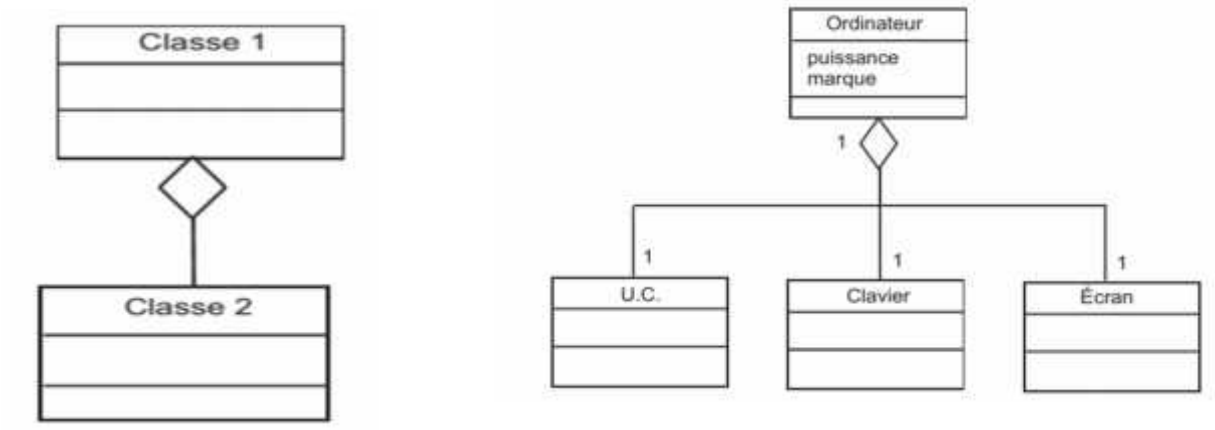


Figure II.11 Exemple d'agrégation [5].

Dans cet exemple, nous avons modélisé le fait qu'un ordinateur comprend une UC, un clavier et un écran.

II.2.5 Composition

La composition est une relation d'agrégation dans laquelle il existe une contrainte de durée de vie entre la classe « composant » et la ou les classes « composé ». Autrement dit la suppression de la classe « composé » implique la suppression de la ou des classes « composant ».

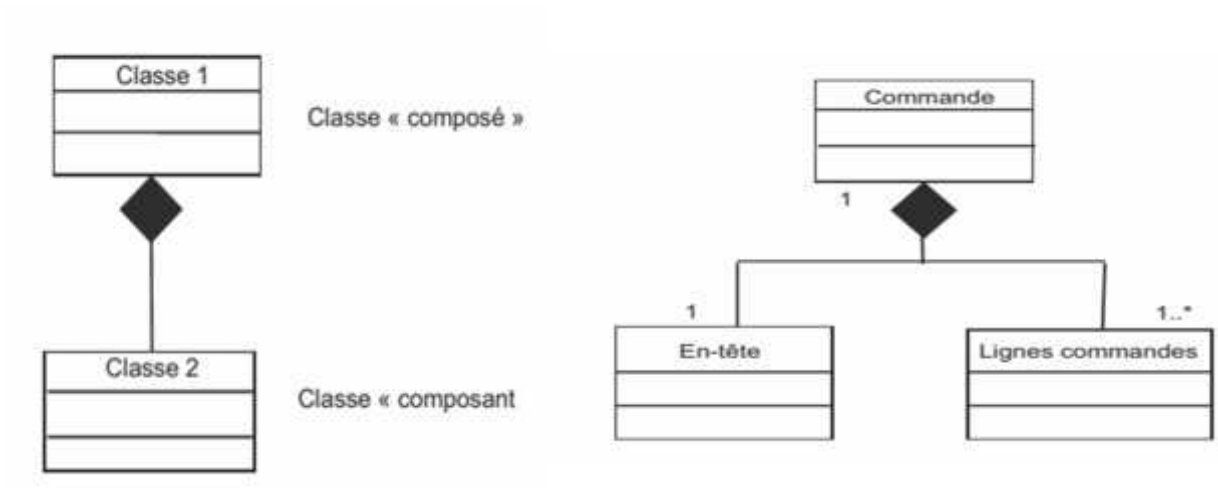


Figure II.12 Exemple d'une relation de composition [5].

II.2.6 Qualification

La qualification d'une relation entre deux classes permet de préciser la sémantique de l'association et de qualifier de manière restrictive les liens entre les instances. Seules les instances possédant l'attribut indiqué dans la qualification sont concernées par l'association. Cet attribut ne fait pas partie de l'association [5].

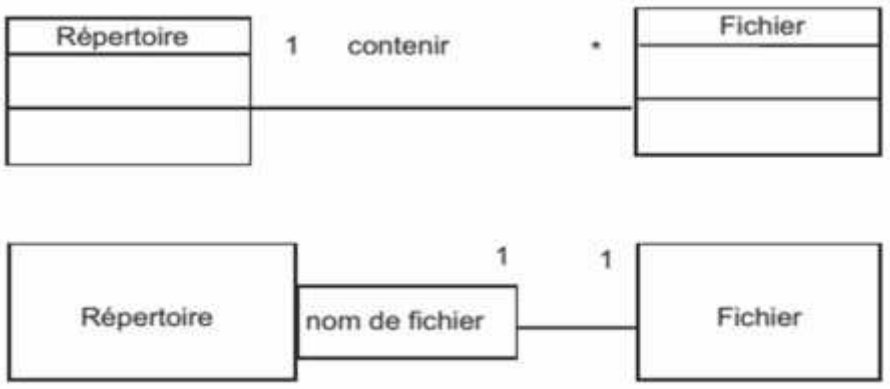


Figure II .13 Exemple d'association qualifiée [5].

Soit la relation entre les répertoires et les fichiers appartenant à ces répertoires. À un répertoire est associé 0 à n fichiers. Si l'on veut restreindre cette association pour ne considérer qu'un fichier associé à son répertoire, la relation qualifiée est alors utilisée pour cela.

II.2.7Dépendance

La dépendance entre deux classes permet de représenter l'existence d'un lien sémantique. Une classe B est en dépendance de la classe A si des éléments de la classe A sont nécessaires pour construire la classe B. La relation de dépendance se représente par une flèche en pointillé entre deux classes.

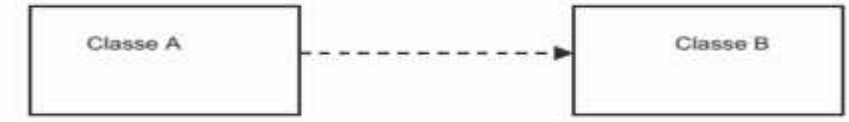


Figure II.14 Représentation d'un lien de dépendance [5].

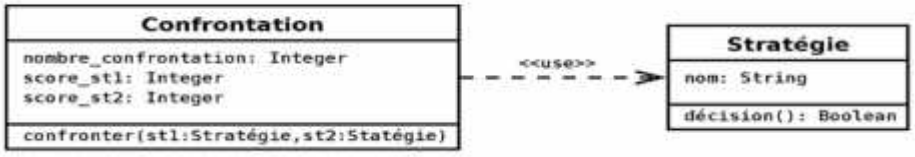


Figure II.15 Exemple de relation de dépendance [2]

L' exemple montre que la classe Confrontation utilise la classe Stratégie car la classe Confrontation possède une méthode confrontée dont deux paramètres sont du type Stratégie. Si la classe Stratégie, notamment son interface, change, alors des modifications devront également être apportées à la classe Confrontation.

II.2.8 Interface

Le rôle d'une interface est de regrouper un ensemble d'opérations assurant un service cohérent offert par un classeur et une classe en particulier.

Une classe d'interface permet de décrire la vue externe d'une classe. La classe d'interface, identifiée par un nom, comporte la liste des opérations accessibles par les autres classes. Le compartiment des attributs ne fait pas partie de la description d'une interface. L'interface peut être aussi matérialisée plus globalement par un petit cercle associé à la classe source. La classe utilisatrice de l'interface est reliée au symbole de l'interface par une flèche en pointillé. La classe d'interface est une spécification et non une classe réelle. Une classe d'interface peut s'assimiler à une classe abstraite [2]

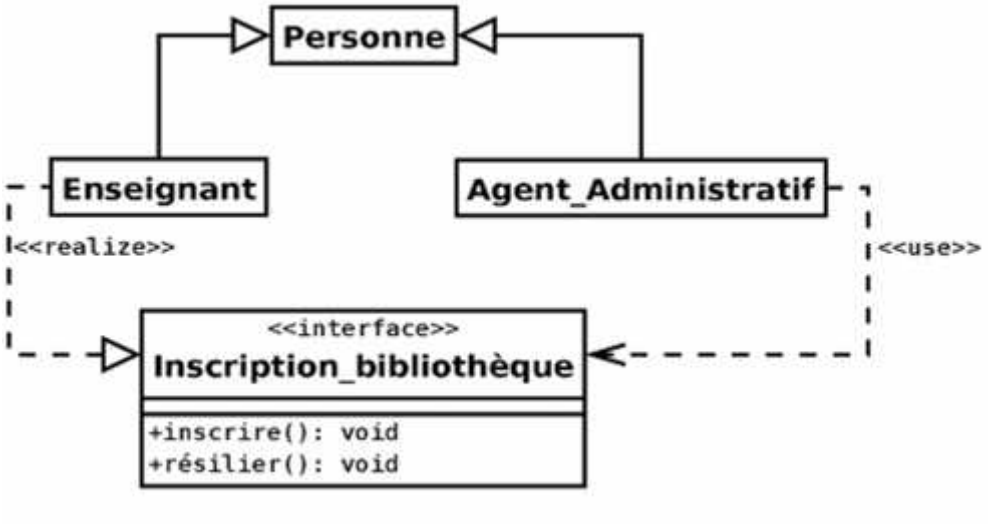


Figure II .16 Exemple de diagramme mettant en œuvre une interface [2].

II.2.9 Généralisation et spécialisation et l'héritage simple

La généralisation est la relation entre une classe et deux autres classes ou plus partageant un sous-ensemble commun d'attributs et/ou d'opérations. La classe qui est affinée s'appelle super-classe, les classes affinées s'appellent sous-classes. L'opération qui consiste à créer une super-classe à partir de classes s'appelle la généralisation. Inversement la spécialisation consiste à créer des sous classes à partir d'une classe[5].

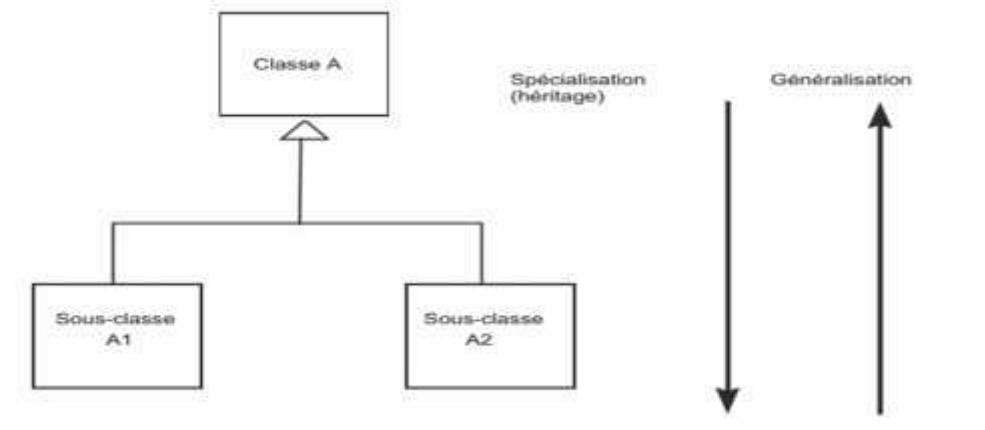


Figure II .17 Formalisme de la relation de généralisation [5].

• la sous-classe A1 hérite de A, c'est une spécialisation de A.
• la sous-classe A2 hérite de A, c'est une spécialisation de A.

L'héritage permet à une sous-classe de disposer des attributs et opérations de la classe dont elle dépend. Un discriminant peut être utilisé pour exploiter le critère de spécialisation entre une classe et ses sous-classes. Le discriminant est simplement indiqué sur le schéma, puisque les valeurs prises par ce discriminant correspondent à chaque sous-classe.

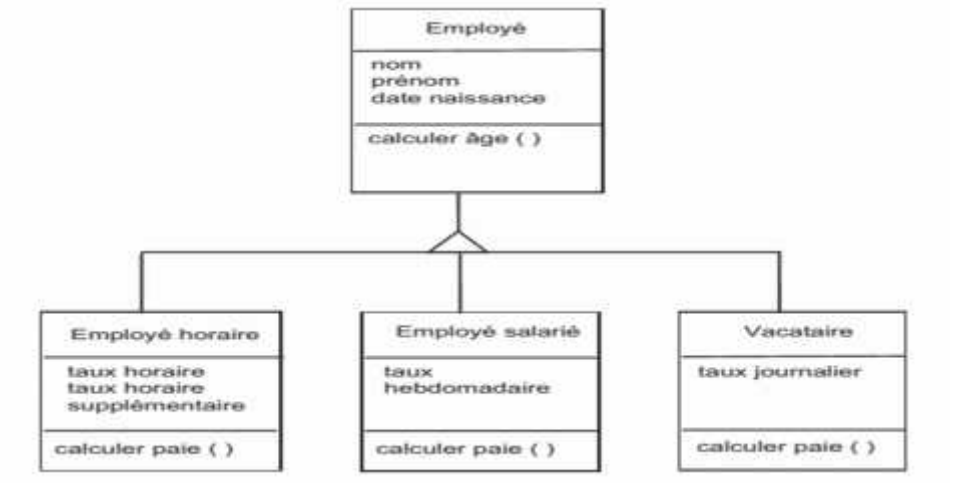


Figure II.18 Exemple de relation de spécialisation [5].

Dans cet exemple, les attributs nom, prénom et date de naissance et l'opération « calculer âge» de« Employé » sont hérités par les trois sous-classes : Employé horaire, Employé salarié, Vacataire.

II.2.9.1 L'héritage multiple

Dans certains cas, il est nécessaire de faire hériter une même classe de deux classes « parentes » distinctes. Ce cas correspond à un héritage multiple [5].

Dans exemple ci-dessous exprime d'héritage multiple où la classe « Véhicule amphibie » hérite des classes « Véhicule terrestre » et « Véhicule marin ».

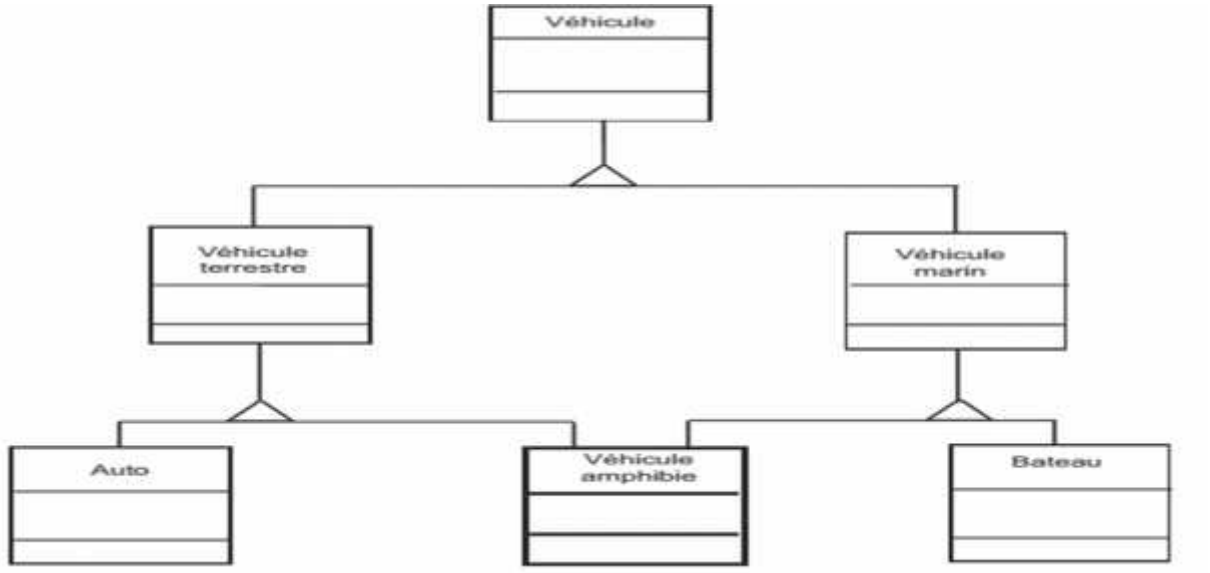


Figure II.19 Exemple de relation d'héritage multiple [5]

II.2.10 Classe abstraite et opération abstraite

Une classe abstraite est une classe pour laquelle il n'est pas possible de créer d'instances directement [5]. Une opération abstraite d'une classe A est une opération ne possédant pas d'implémentation dans A mais qui doit obligatoirement être implémentée dans les sous - classes de A [5].

Toute classe contenant au moins une opération abstraite est abstraite [5].

II.3 le test

Définition : Le test est l'exécution ou l'évaluation d'un système ou des composants d'un système par des moyens manuels ou automatisés afin de vérifier qu'il répond à ses spécification ou pour identifier les différences entre les résultats obtenus et les résultats attendus(spécifiés préalablement dans la conception) [14].

II.4 Test statique et test dynamique

Ces deux catégories de test sont présentes dans le processus de Vérification & Validation. À l'intérieur de ce processus.

✓ Test statique

Le test statique se résume à une analyse, sans exécution, d'une présentation (modèle, programme, etc.). Les techniques de vérification utilisées lors du test statique implique l'analyse de la présentation qui peut correspondre par exemple à un code source du programme pour détecter des erreurs [15].

✓ Test dynamique

Le test dynamique consiste à exécuter un programme afin d'analyser son comportement [15]. Tout comme le test statique, le test dynamique introduit un ensemble de techniques telles que [15].

- Les tests unitaires : Ils ont pour but de vérifier le bon fonctionnement d'un module indépendamment du reste du programme. Cette phase doit être particulièrement soignée car elle permet de d´détecter au plus tôt des erreurs, limitant ainsi leur propagation. Les tests unitaires vont pour cela tester la couverture du code, c'est-`a-dire l'ensemble des instructions à tester.
- Les tests d'intégrations : Une fois chaque module logiciel testé par des tests unitaires, les tests d'intégrations permettent de s'assurer que les différents modules fonctionnent bien ensemble
- Les tests de validation : Le test de validation permet de vérifier si toutes les exigences client d´écrites dans le document de spécification d'un logiciel, écrit à partir de la spécification des besoins, sont respectées.

Tout d'abord, les composants sont testés en premier, ce qui correspond au test unitaire. Ensuite, les tests d'intégration consistent à tester les interactions entre ces composants lors de leur intégration. Et enfin les tests de validation portent sur le test global des modules intégrés correspondant au test du système [15].

II.5 Approches structurelles

Les approches structurelles ou basées implémentation sont souvent assimilées au test de la boîte blanche à cause de leur vision portée sur le code. Ces approches sont fondées sur les structures internes du programme sous test principalement les graphes de contrôle.Elles sont surtout utilisées durant le processus de test unitaire [16]. L'utilisation des tests basés sur les approches structurelles a pour but de garantir que les instructions (conditions, chemins, etc.) implémentées au niveau du code sont exécutées tel que prévu.

Pour ce type d'approches. On considère à la fois le code dans chacune des méthodes et les interactions entre les méthodes de la classe. Chaque méthode peut être testée à partir d'un ensemble d'entrées. Les techniques de test utilisées par les approches structurelles sont basées sur la couverture des instructions, des chemins. Le test de valeurs spéciales, de partitions d'ensembles et d'évaluation symbolique. Dans le domaine des approches structurelles, nous pouvons citer des techniques de test comme le test d'instructions, le test de branches. Le test de chemin et le test de conditions [15].

II.5.1 Le test d'instructions

Le test d'instructions consiste à exécuter, au moins une fois, le plus grand nombre d'instructions dans le programme.

II.5.2 Le test de branches

Le test de branches, contrairement au test d'instructions. Consiste à exécuter au moins une fois le plus grand nombre de branches du programme. Ce test peut être utilisé pour vérifier les appels entre les différents composants du logiciel. L'exécution de ce type de test prend en compte les possibilités de traverser au moins une fois les branches du programme, de vérifier si les branches parcourues lors elle-même test sont compatibles et de vérifier les différentes combinaisons d'exécution des branches.

II.5.3 Le test de chemins

Le test de chemins consiste à exécuter un nombre maximum de chemins sur le graphe de contrôle, ce qui donne la possibilité d'effectuer un contrôle dynamique des enchaînements. Il permet d'exécuter tous les chemins ou des chemins choisis à travers un système [16].

II.5.4 Le test de conditions

Ces tests constituent une Version améliorée des tests de branches et sont complémentaires des tests de chemins. Le test de conditions permet de tester les conditions logiques contenues dans un programme. Une condition simple est une variable booléenne (variable ayant comme valeur vrai ou faux) ou une expression relationnelle. Si, dans un programme, les décisions sont une conséquence de calculs d'expression booléenne, il faut alors créer un ensemble de cas de test pour pouvoir fournir les différentes valeurs en entrée de la condition, et ce dans le but de satisfaire toutes les combinaisons possibles [15].

II.6 Approches fonctionnelles

Les approches fonctionnelles ou basées sur la spécification, quant à elles, sont assimilées au test de la boîte noire en ce sens qu'elles transforment les entrées en sorties basées sur les spécifications propres du logiciel en ignorant la structure interne de l'objet testé [16]. Les tests générés à partir de cette approche offrent plusieurs avantages. Tout d'abord, il est bon de préciser que les spécifications. Établies lors de l'analyse, décrivent les fonctions que le logiciel est supposé offrir. Aussi, le processus menant à la production de tests basés sur les spécifications permet au développeur de mieux cerner les spécifications du logiciel et par la même occasion de déceler des erreurs ou ambiguïtés dans les spécifications. Un autre avantage de l'utilisation de cette approche est son indépendance vis-à-vis de l'implémentation des spécifications. Il existe plusieurs techniques de base dérivées des approches fonctionnelles.

II.6.1 Le test par classes d'équivalences

Cette technique de test, consiste à identifier parmi l'ensemble des données possibles en entrée. D'une part, des classes d'équivalence, et ce, en regroupant les données d'entrée ayant des caractéristiques ou propriétés communes et, d'autre part, il vérifie le comportement du composant vis-à-vis des éléments d'une même classe d'équivalence (le but est de faire en sorte

que le composant sous test se comporte de la même manière pour une même classe d'équivalence). Cette technique a pour objectif de contrôler les données en sortie dans le but de déterminer si celles-ci correspondent aux différentes classes d'équivalence définies, d'une part et de s'assurer de l'adéquation du composant par rapport aux spécifications, d'autre part [15].

II.6.2 Le test aux limites

Cette technique de test consiste à choisir des données égales ou proches des bornes des classes d'équivalence.

II.6.3 Le test basé sur les besoins

Le test basé sur les besoins est un test permettant de démontrer que le système implémente proprement les spécifications [15]. Il s'agit d'un test de validation. En effet, la majorité des erreurs détectées dans un système proviennent de l'implémentation mal effectuée des besoins [17].

Outre ces techniques de base, il existe d'autres approches basées sur l'utilisation des méthodes semi-formelles et formelles.

II.7 Méthodes Formelles

Les méthodes semi-formelles, par la diversité de leurs outils et par leurs outils dans le développement, sont très utilisées. Elles permettent de modéliser différents aspects d'un système. Parmi ces méthodes, nous pouvons citer le langage UML largement utilisé dans le développement de systèmes orientés objet. Le langage UML permet, en effet, de produire plusieurs modèles tels que le modèle objet, les diagrammes d'états, les diagrammes de collaboration, les diagrammes d'interactions, etc. Ces différents modèles, présentés de manière graphique, constituent, en fait, de véritables sources d'informations en ce qui concerne les spécifications du logiciel et pouvant être utilisées pour supporter le processus de test [18].

Cependant, les méthodes semi-formelles posent certains problèmes à la mise en place de tests dérivés des spécifications. Ces problèmes se résument au manque, parfois, de rigueur dans la sémantique. En effet, les modèles obtenus à partir des méthodes semi-formelles peuvent être ambiguës et peuvent poser le problème. Par la même occasion, de compréhension et

d'interprétation pour établir un processus de génération de test. Pour pallier ce manque de précision et aussi pour éviter les problèmes d'interprétation, nous avons les méthodes formelles.

Les méthodes formelles utilisées dans le développement de systèmes informatiques sont des techniques basées sur les mathématiques pour décrire les propriétés de ces systèmes [16]. De telles méthodes fournissent un cadre à l'intérieur duquel on peut spécifier, développer et vérifier des systèmes informatiques d'une façon systématique. Elles reposent sur l'utilisation de la logique mathématique et offrent un langage éliminant l'ambiguïté et un formalisme de preuve permettant de garantir la validité des énoncés logiques. Elles permettent au développeur d'avoir une meilleure compréhension, d'obtenir des spécifications concises et peuvent servir de base à la vérification formelle et au développement de test par l'intermédiaire de l'utilisation des langages de spécification formels. Ces langages sont composés de trois composants importants qui sont [21]:

- une syntaxe définissant la notation spécifique avec laquelle la spécification est représentée
- une sémantique permettant de définir l'univers des objets qui seront utilisés pour décrire le système
- un ensemble de relations définissant les règles qui indiquent quels objets satisfont les besoins.

Ces différents langages de spécification formels se regroupent sous deux catégories

- Les langages de spécification basés modèle
- Les langages algébriques de spécification

II.7.1 Les langages de spécification basés modèle

Les langages de spécification basés modèle utilisent les mathématiques pour modéliser explicitement l'état du système. Ces langages expriment leurs fonctionnalités par le changement d'état en se basant sur le modèle d'état [16].

II.7.2 Les langages algébriques de spécification

Les langages algébriques de spécification décrivent le logiciel en établissant des instructions formelles, appelées axiomes, sur les relations entre les opérations et les fonctions intervenant sur ces opérations[16].

Par ailleurs, malgré leur faiblesse, les méthodes semi-formelles sont très utiles dans le processus de génération de test à partir des spécifications. Elles sont également largement utilisées relativement aux méthodes formelles dans un processus de conversion des informations véhiculées par un langage semi-formelles vers un langage formel. Il est donc avantageux de combiner les méthodes formelles et les méthodes semi-formelles dans un même et unique but étant celui de la génération de test cl partir des spécifications [20].

II.8 Test à partir de modèles

Dans le test orienté objet, il existe principalement deux familles d'approches. D'un côté, nous avons les approches structurelles et de l'autre les approches fonctionnelles [16].Le test à partir de modèles (en anglais Model-Based Testing (MBT)), sur lequel nous nous basons pour nos recherches, repose sur les principes du test fonctionnel. Le modèle de test est la représentation externe formelle ou semi-formelle du système testé. Ce modèle est créé dans le but de s'en servir pour la génération de tests [19].

Le sous-ensemble d'UML que nous utilisons est basé sur trois diagrammes : diagrammes de classes (permettant de modéliser des points de contrôle et d'observation du système sous test), diagrammes d'objets et des diagrammes d'états/transitions. Ces diagrammes sont enrichis avec le langage OCL (Object Constraint Language) permettant de modéliser les comportements dynamiques du SUT (System Under Test) [19].

II.8.1 Les tests à partir diagrammes de classes

Les diagrammes de classes représentent la vue statique du modèle. Ils décrivent les objets abstraits du système et leurs dépendances. Les dépendances sont représentées par des associations binaires ou réflexives entre les classes.

Il est à noter que l'héritage d'objets n'est pas pris en compte. Les attributs de classe permettent de décrire la structure d'un objet. Ils sont caractérisés par une valeur par défaut et

un type (les types supportés sont les entiers, les booléens et les énumérés – une énumération étant une classe composée que de littéraux).

Enfin, les opérations modélisent les actions effectuées par l'objet. Elles peuvent être définies avec ou sans paramètre d'entrée ou de sortie. Comme pour les attributs, elles peuvent avoir les types : entier, booléen ou énuméré [19].

II.8.2 les cas de test

Un cas de test est un rapport contenant l'ensemble de questions concernant les classes, les méthodes, les attributs, les relations, l'héritage, et les cardinalités, d'un diagramme de classes.

Le test sert à répondre à cet ensemble de questions pour vérifier la conformité du code implémenté avec son diagramme de classes.

Par exemple l'auteurs Tetsuro Katayama et Yusuke Yabuya [11] présenté un outil dans lequel on peut générer des cas de test, l'exemple est présente dans la figure ci-dessous

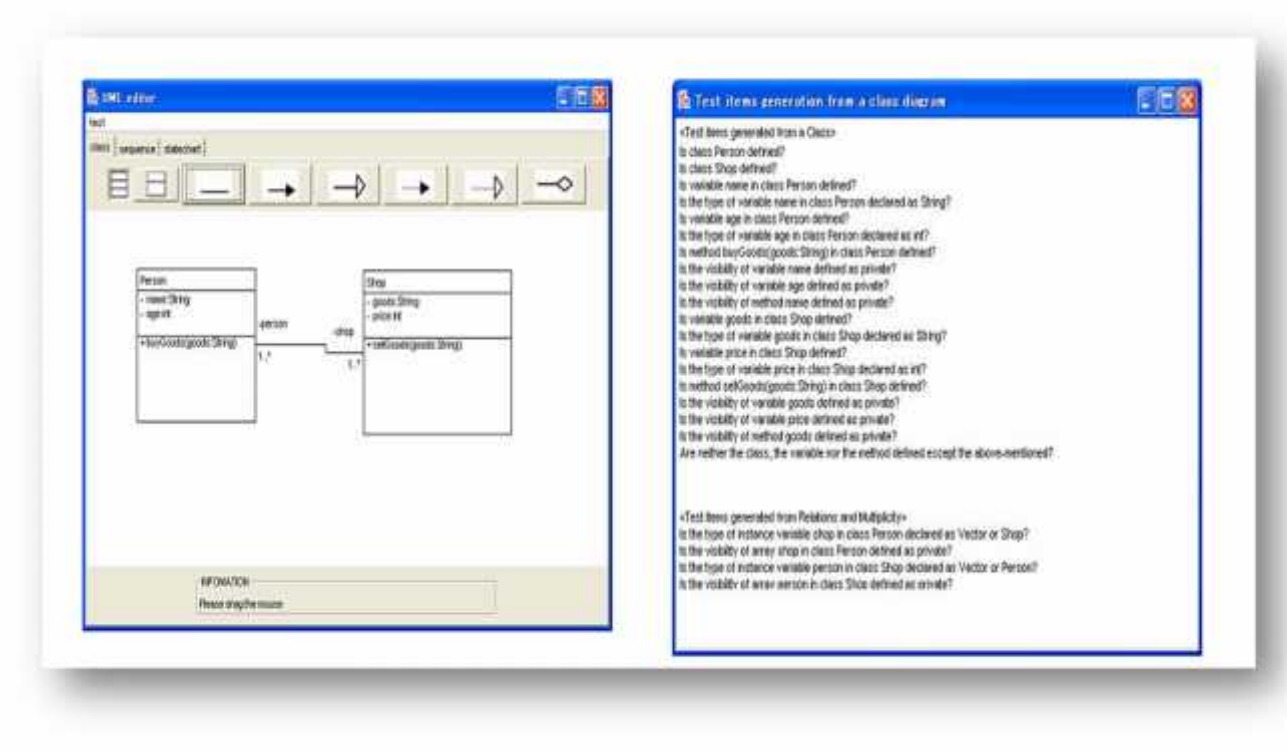


Figure II.20 Généré le test de diagramme de classes [11].

II.9 Conclusion

Dans ce chapitre nous avons fait une description détaillée des diagrammes classe, et les principaux éléments de ce dernier qui sont les classes et leurs relations: association, généralisation, et héritage ainsi que les déférent types de test.l'outil ANTLR sera présente dans le prochain chapitre.

Chapitre III

Le générateur ANTLR

III.1 Introduction

Pour réaliser les analyses lexicales et syntaxiques, il faut spécifier une grammaire (Lexer, Parser) qui décrit la façon de regrouper le flux de caractères en un flux de lexème (tokens) puis D'analyser le flux de tokens sortant. Il suffit de créer un fichier d'extension .g4 définissant Les unités lexicales et syntaxiques de la grammaire. Dans sa génération en langage Java, ANTLR créée un ensemble de classes permettant l'utilisation de l'arbre créé par le parser.

III.2 ANTLR (ANother Tool for Language Recognition)

Est un environnement de développement de compilateurs et de traducteurs qui génère des analyseurs lexicaux et des analyseurs syntaxiques descendants. ANTLR a été principalement développé par Terrence Parr. ANTLR peut être classé dans la catégorie des environnements bout en bout, c'est-à-dire un environnement qui permet de développer le compilateur au complet en se servant uniquement de l'outil. Toutefois, il est possible avec ANTLR de laisser à l'utilisateur la possibilité de compléter le code généré grâce à du code que ce dernier a écrit. Ce code peut être écrit avec les langages de programmation C++, Java, C# et Python [9].

III.2.1 Evolution d'ANTLR

L'avantage de la version 3 d'ANTLR est la construction automatique d'un arbre syntaxique abstrait (AST) où ne figurent que des tokens. Il faut alors écrire une grammaire d'arbre qui permet d'exploiter cet arbre. Un inconvénient est qu'il faut insérer dans cette grammaire des actions dans un langage cible, ce qui dénature la grammaire. De plus, ces actions doivent tenir compte de la représentation interne par AntLR de cette structure d'arbre [9].

Dans la version 4 d'ANTLR, le parti pris d'AntLR est de plus construire d'arbre syntaxique abstrait mais de se contenter de l'arbre syntaxique concret, c'est à dire de l'arbre de dérivation. ANTLR permet cependant de générer des classes implémentant un visiteur d'arbre. Les actions permettant d'exploiter la structure d'arbre sont ainsi externalisées et la grammaire reste intacte. Un visiteur d'arbre externe permet beaucoup plus facilement d'implémenter des possibilités de visite comme le fait de ne pas visiter une branche ou de visiter plusieurs fois une branche [9].

Pour réaliser les analyses lexicale et syntaxique avec ANTLR, il faut [10] :

1. Créer un fichier d'extension .g4 définissant les unités lexicales et syntaxiques de la grammaire. ANTLR génèrera lui-même par la suite les classes correspondant au lexer et au parser.
2. Utiliser le compilateur Ant pour générer les analyseurs spécifiés dans la grammaire.
3. Définir un programme principal dans le langage cible qui utilisera les analyseurs (lexer et le parser) générés pour effectuer le travail souhaité.

III.2.2 Fichiers générés

Une fois la compilation réussie, ANTLR génère un ensemble de fichiers. Il s'agit notamment [9] :

- d'une classe pour le parser ;
- d'une classe pour le lexer ;
- d'une ou plusieurs classes pour les tokens générés XXXTokensType.java.

III.2.3 Avantages

ANTLR possède un certain nombre d'avantages [8].

- Il possède des primitives de constructions d'arbres syntaxiques abstrait(AST). Insérées dans une grammaire, ces primitives permettent à l'analyseur de construire une représentation intermédiaire d'un programme source sous forme d'un arbre AST.
- Il permet d'écrire des grammaires d'arbres permettant de générer des analyseurs d'arbres fonctionnant sur le même principe que les analyseurs syntaxique. Un arbre AST est transformé en une structure linéaire de nœuds et est donnée en entrée à un analyseur d'arbre de manière analogue à la structure linéaire de tokens donnée en entrée à un analyseur syntaxique.
- Un analyseur ANTLR peut être associé à un groupe de patrons de chaînes. On peut augmenter les règles d'une grammaire ANTLR par des primitives permettant à l'analyseur généré d'invoquer ces patrons de chaînes et ainsi de produire un code résultat.

III.2.4 Nouvelles caractéristiques d'ANTLR

Bien qu'ANTLR soit un analyseur descendant, les apports théoriques récents permettent de s'affranchir des inconvénients majeurs de ces analyseurs [8]:

- Il est possible d'écrire des grammaires non factorisées à gauche.
- Il est possible d'écrire des règles directement récursives à gauche.
- Il est possible d'écrire des grammaires non ambigües en ordonnant les alternatives d'une règle.

III.3 Écriture de la grammaire

III.3.1Les multiplicités

Dans le corps d'une règle, des formalismes sont utilisés pour caractériser la multiplicité d'une expression dans une construction. Ces formalismes sont [13]:

- **(expression) *** : **zéro ou plus.** Ceci signifie que l'expression entre parenthèses peut être rencontrée zéro ou plusieurs fois dans la construction à reconnaître ;
- **(expression) +** : **un ou plus**. Ceci signifie que l'expression entre parenthèses peut être rencontrée une ou plusieurs fois dans la construction à reconnaître ;
- **(expression) ?** : **zéro ou un**. Ceci signifie que l'expression entre parenthèses peut être rencontrée une fois ou ne pas du tout être rencontrée dans la construction à reconnaître.
- Exemple de règle avec multiplicité

Nombre_decimal:

('0' .. '9')+(',' ('0'..'9')+)?

III.3.2 Les alternatives

Lorsque plusieurs constructions se rapportent au même lexème, au lieu de définir plusieurs règles, il est judicieux de les regrouper dans une seule règle à plusieurs alternatives. Les alternatives sont séparées les unes des autres par un "|" dans le corps de la règle [13].

```
nom-regle[arguments] returns [valeur_de_retour]
```

```
options{ // les options liées uniquement à cette règle }

  : alternative 1

  | alternative 2

  | …
  | alternative n
  ;
```

Exemple de règle : reconnaissance des espaces dans un flux de caractères

```
White_Space
    : ' '
    | '\t'
    | '\r' '\n'
    | '\n'
  ;
```

III.3.3 Les options

On remarquer dans la structure d'une grammaire qu'il figure plusieurs endroits dans lesquels on peut définir des options. En effet, les options permettent de donner certaines fonctionnalités à la grammaire. Elles peuvent concerner toute la grammaire, juste une classe ou bien être spécifique à une règle [13].

Dans la section Options d'une grammaire ANTLR, on peut spécifier une série de clé/valeur qui modifie la façon dont ANTLR génère du code. Ces options affectent globalement tous les éléments contenus dans la grammaire, sauf si on les annule dans une règle. Ce paragraphe décrit toutes les options disponibles [13].

La section des options doit venir après l'en-tête de la grammaire et doit avoir la forme suivante :

```
options {
nom1 = valeur1;
nom2 = valeur2;
```

```
...
}
```

Les noms des options sont toujours des identificateurs, mais les valeurs peuvent être des identificateurs, des chaînes de littéraux entre des guillemets simples, des nombres entiers, ou le littéral spécial (l'astérisque souvent utilisable uniquement avec l'option k). Les valeurs sont toutes des littéraux et, par conséquent, ne peuvent pas être des noms d'option. Pour les chaînes littérales formées d'un seul mot comme « Java », vous pouvez l'écrire tout simplement, comme indiqué ci-après [13] :

```
options {language=Java;}
```

La liste qui suit résume les options ANTLR au niveau de la grammaire.

III.3.3.1 Language

Précisez le langage cible dans lequel ANTLR devra générer les analyseurs.

ANTLR utilise le CLASSPATH pour trouver le répertoire *org/antlr/* codegen/ *templates/Java* ; dans ce cas, c'est Java le langage cible, et c'est lui par défaut[13].

III.3.3.2 Output

Générer des modèles de sortie, un modèle ou des arbres AST. Cette option est disponible uniquement pour les grammaires qui contiennent des analyseurs syntaxiques et un analyseur d'arbres. La valeur par défaut est de ne rien générer [13].

III.3.3.3 Les actions

ANTLR permet d'ajouter aux règles des instructions écrites dans le langage cible. Ces instructions sont appelées actions [13].

Une action est définie entre accolades et peut être ajoutée :

- **au début de la règle** Ainsi l'action sera exécutée lorsque l'analyseur rencontrera cette règle ;
- **dans le corps de la règle** : l'action sera exécutée pendant le traitement de la règle.

Exemple d'actions associées à une règle

```
ENTIER {System.out.println ("un entier est rencontré"); }//
action au début
         : (‘0’…’9’)+ {System.out.println ("traitement de
l'entier"); }
```

Une action peut être constituée d'instructions simples ou d'une suite d'instructions plus ou moins complexes portant aussi bien sur des opérations mathématiques que sur des accès à des fichiers ou à une base de données. Cependant, pour la lisibilité et l'efficacité de la grammaire, il faut éviter d'ajouter des actions complexes ou trop longues. Il vaudrait mieux définir les traitements par des procédures dans un autre fichier ou paquetage et appeler ces procédures comme actions dans les règles [13].

Exemple de grammaire de l’expression mathématique

```
grammarExpr;

prog:stat+;

stat: expr NEWLINE

| ID'='expr NEWLINE

| NEWLINE ;

expr:

expr ('*'|'/') expr

| expr ('+'|'-') expr

| INT

| ID

| '('expr')' ;

ID : [a-zA-Z]+ ;

INT : [0-9]+ ;
```

```
NEWLINE:'\r'?'\n';

WS : [ \t]+ -> skip ;
```

Il faut exécuter ANTLR sur le fichier Expr.g4 qui contient la grammaire. Le nom du fichier doit être identique au nom de la grammaire [9].

Grammaires sont constituées d'un ensemble de règles et tokens des règles pour la structure syntaxique comme

`stat` et `expr` ainsi que des règles pour symboles de vocabulaire (tokens), tels que les identifiants et les nombres entiers

identifiants : `ID : [a-zA-Z]+ ;`

nombres entiers : `INT : [0-9]+ ;`

Règles commençant par une lettre minuscule comprennent les règles de l'analyseur. Règles commençant par une lettre majuscule comprennent les règles lexicales (token)

On séparer les alternatives d'une règle avec l'opérateur |, et nous pouvons symboles de groupe avec entre parenthèses dans les paragraphes. Par exemple, le paragraphe ('*' | '/') correspond soit un symbole de multiplication ou un symbole de division.
Dans sa génération en langage Java, ANTLR créée un ensemble de classes permettant l'utilisation de l'arbre créé par le parser. Ainsi à partir d'une grammaire appelée Expr, ANTLR créée [9]:
1. le Lexer : ExprLexer.java

2. le Parser : ExprParser.java

3. les Listes de tokens:Expr.tokens ;ExprLexer.tokens

4. les Listeners : ExprListener.java (interface)

ExprBaseListener.java (classe implémentant à vide l'interface)

Cette grammaire permet d'analyser des expressions. La priorité des opérateurs et l'associativité à gauche sont préservées grâce à l'ordre des alternatives dans la règle expr.

L'utilisation des compléments tels que sum est expliquée plus loin. A partir de la grammaire d'expression précédente ANTLR généré la structure d'arbre suivante à partir de l'expression [9]:

193

a = 5

b = 6

a+b*2

(1+2)*3

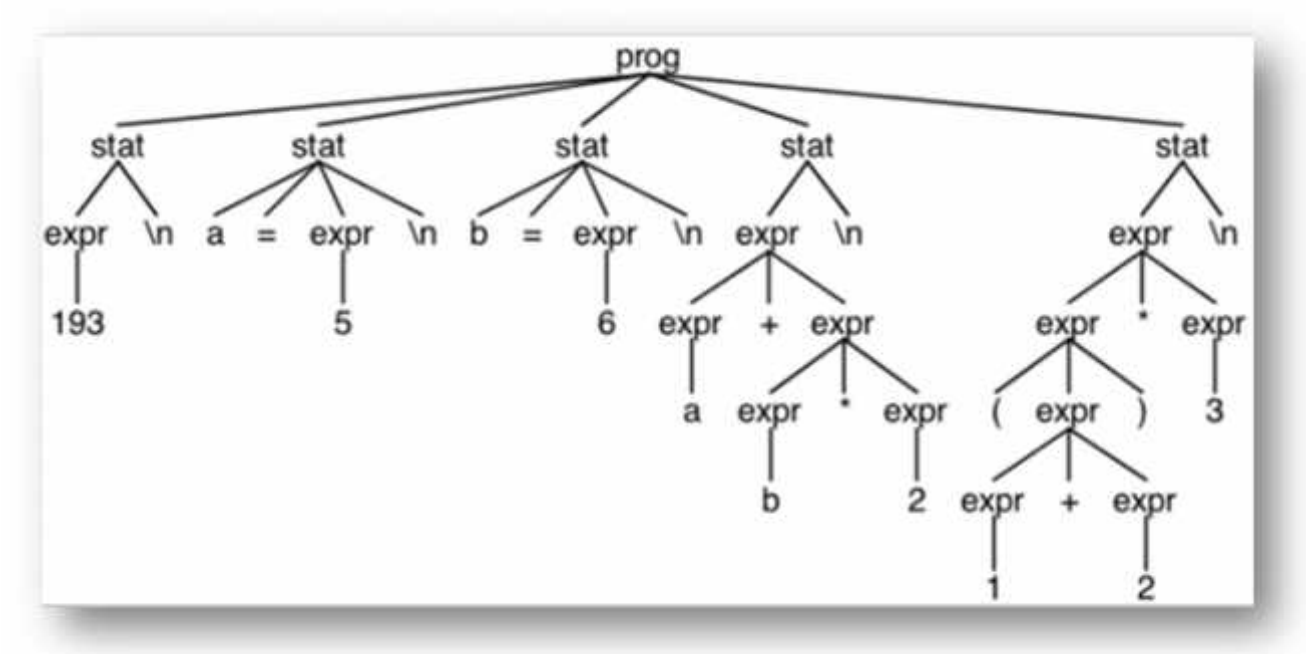


Figure III.1 Arbre de dérivation [9].

III.4 CONCLUSION

Nous avons présenté dans ce chapitre l'outil ANTLR son évolution et ses avantages par rapport aux autres générateurs d'analyseurs lexicaux et syntaxique.

Chapitre IV

Un compilateur des diagrammes de classe

IV.1 Une Grammaire pour diagramme de classe

Pour réaliser les analyseurs lexicale et syntaxique, il faut spécifier une grammaire (Lexer,Parser) qui décrit la façon de regrouper le flux de caractères en un flux de tokens puis d'analyser le flux de tokens sortant. Il suffit de créer un fichier d'extension .g4 définissant les unités lexicales et syntaxiques de la grammaire [9].

Il faut exécuter ANTLR sur le fichier Expr.g4 qui contient la grammaire. Le nom du fichier doit être identique au nom de la grammaire. Avec la commande java accessible il suffit de créer la commande suivante dans un fichier batch (version Windows) pour obtenir l'ensemble des fichiers

java -jar antlr-4.0-complete.jar CD.g4 [9]

IV.1.1 Les fichiers générés

CDParser.java : Ce fichier contient la définition de classe de l'analyseur spécifique à grammaire CD qui reconnaît notre syntaxe du langage de diagramme de class

CDLexer.java : ANTLR extrait automatiquement un analyseur et lexer séparée spécification de notre grammaire.

CD.tokens : ANTLR attribue un numéro de type de token pour chaque token nous définissons et stocke ces valeurs dans ce fichier

CDListener.java : est l'interface qui décrit les rappels que nous pouvons mettre en œuvre.

CDBaseListener.java : est un ensemble de mises en œuvre par défaut vides.

Figure IV.1 Les fichiers générer par ANTLR pour la grammaire CD

IV.1.2 Grammaire CD (grammaire diagramme de classe)

```
grammar CD;

options {

  language = Java;

}

CASDIGRAM :'CD';

CLAS      :'Class';

LACCOL         :'{';

RACCOL         :'}';

EXTEND          :'extends';

VISIB      :'+'|'-'|'#' ;

DP          : ':' ;

LP          : '(' ;

RP          :  ')' ;

cdiagram: CASDIGRAM cdname LACCOL clas*  association* RACCOL;

clas : CLAS cname LACCOL ( attribut  ';' )* ( methode';')* ( EXTEND cname ';' )?  RACCOL;

attribut : VISIB attriname DP typeattri ;

attriname : ID ;

methode: VISIB methname LP parametre RP DP typeattri;

parametre : (   |attrimeth (',' attrimeth )*)?;

attrimeth : attriname DP typeattri ;

association : 'Association' assoname LACCOL 'type=' typeasso ';' from cname ';'role* to cname';' role* RACCOL ;
```

```
role :'role=' ID ';' cardinalite ;

cardinalite :cardin ',' cardin ;

methname : ID ;

from :'from';

to : 'to' ;

cardin: ('0..1'|'0..*'| NonZeroDigit| '1..' NonZeroDigit|)+ ;

cname :ID;

cdname :ID;

assoname :ID ;

typeasso: 'agre'|'compo'|'standar' ;

ID  : ('a'..'z'|'A'..'Z'|'_')+('0'..'9')*;

WS  : (' '|'\t'|'\n'|'\r')+ ->skip;

typeattri : ('int' |'string' 'fleat'|'void'|'char'|'boolean');

NonZeroDigit :[1-9];
```

IV.1.2.1 Lexer (L'analyseur lexical)

Le lexer correspond à l'analyseur lexical. Les tokens sont définis dans cette partie. Ils permettront de reconnaître les mots-clés de la grammaire, comme par exemple `CASDIGRAM, CLAS`

On identifie donc les mots-clés de la grammaire. Les symboles sont identifiés selon la suite d'instructions suivante :

```
CASDIGRAM :'CD';

CLAS      :'Class';

LACCO     :'{';

RACCOL    :'}';
```

```
EXTEND    :'extends';

VISIB     :'+'|'-'|'#' ;

DP         : ':' ;

LP         : '(' ;
```

IV.1.2.2 Parser (L'analyseur syntaxique)

On définit dans le parser les règles qui permettront de construire l'AST Pour chaque règle, ANTLR crée une racine d'arborescence. Pour chaque token présent dans une règle, il crée un nœud (Tree) et l'ajoute comme fils du nœud courant. Par défaut les nœuds forment une liste et AntLR attache les nœuds de la liste à un nœud racine de type nil.

Exemple de règle

```
clas : CLAS cname LACCOL ( attribut  ';' )* ( methode';')* ( EXTEND cname ';' )?  RACCOL
```

Pour utiliser les classes créées, il faut créer une classe contant une méthode main telle que :

```
public class CDCompiler {

    public static void main(String[] args) throws Exception{

        String inputFile="C:/ah/exp.class";

        //create a CharStream that reads from standard input

        InputStream is= new FileInputStream(inputFile);

        ANTLRInputStream input=new ANTLRInputStream(is);

        //create a lexer that feeds off of inputCharStream

        CDLexer lexer=new CDLexer(input);

        //create a buffer of tokens pulled from the lexer

        CommonTokenStream tokens=new CommonTokenStream(lexer);

        //create a parser that feeds off the tokens buffer
```

```
        CDParser sap=new CDParser(tokens);
        try{
            ParseTree tree=sap.cdiagram();
            ParseTreeWalker walker=new ParseTreeWalker();
            ToTestCases toTestCase=new ToTestCases();
            walker.walk(toTestCase, tree);
        }catch(Exception e){
            System.out.println("probleme");
        }
    }
}
```

IV.1.3.Présentations des différentes méthodes

Pour chaque règle, ANTLR crée une racine d'arborescence. Pour chaque *token* présent dans une règle, il crée un nœud (Tree) et l'ajoute comme fils du nœud courant. Par défaut les nœuds forment une liste et ANTLR les attache à un nœud racine de type nil.

La classe CDListener est une classe qu'on doit implémenter pour visiter l'arbre (tree) syntaxique déjà créé. Elle doit hériter de la classe ParseTreeListener générée par ANTLR.

IV.1.3.1.Méthode enterClas

```
@Override
public void enterClas(CDParser.ClasContext ctx) {
    super.enterClas(ctx); //To change body of generated methods, choose Tools | Templates.
    c_name=ctx.getChild(1).getText();
    test_case+=" is class "+ c_name+" defined?\n";
}
```

Figure IV.2 Méthode enterClas.

Cette méthode nous permet d'extraire le nom d'une classe

IV.1.3.2. Méthode enterAttribut

```
@Override
public void enterAttribut(CDParser.AttributContext ctx) {
    super.enterAttribut(ctx); //To change body of generated methods, choose Tools | Templates.
    visibility=ctx.getChild(0).getText();
    attriname=ctx.getChild(1).getText();
    typeatt=ctx.getChild(3).getText();
    test_case+="is variable "+ attriname +" in class " +c_name+" defined ?\n";
    test_case+="is the visibility of variable "+ c_name +" defined as " +visibility +" ?\n";
    test_case+="is the type of variable "+ attriname +" in class " + c_name + " declared as "+typeatt+ "?\n"
}
```

Figure IV.3 Méthode enterAttribut

La méthode précédente fait la vérification des attributs : ses visibilités, ses noms, et ses types.

IV.1.3.3.Méthode enterMethode

```
public void enterMethode(CDParser.MethodeContext ctx) {
    super.enterMethode(ctx); //To change body of generated methods, choose Tools | Templates.
    meth_name=ctx.getChild(1).getText();

    test_case+="is methode "+meth_name+" of the "+c_name+" class defined?\n";
    test_case+= "is the visibility of methode "+meth_name+" in class "+ c_name+" defined "+visibility+" ?\n";

}
```

Figure IV.4 Méthode enterMethode

La méthode précédant nous donne le nom de la méthode de la classe ainsi que sa visibilité.

IV.1.3.4.Méthode enterAssociation :

```
@Override
public void enterAssociation(CDParser.AssociationContext ctx) {
    super.enterAssociation(ctx); //To change body of generated methods, choose Tools | Templates.
ass_name=ctx.getChild(1).getText();
type_asso=ctx.getChild(4).getText();
test_case+="is association "+ass_name+" defined ?\n";
test_case+="is type of association "+ass_name+" is "+type_asso+" ?\n" ;
}
```

Figure IV.5 Méthode enterAssociation

Cette méthode nous donne le nom de l'association et son type.

IV.1.3.5.Méthode enterAttrimeth

```
@Override
public void enterAttrimeth(CDParser.AttrimethContext ctx) {
    super.enterAttrimeth(ctx); //To change body of generated methods, choose Tools |
     meth_name=ctx.getParent().getText();
    attmeh=ctx.getChild(0).getText();
    typreturne=ctx.getChild(2).getText();
    test_methode+=" parmetre of methode "+attmeh+" defined as "+ meth_name +"\n";
    test_methode+="the type returne of methode "+attmeh+" is "+typreturne+"\n";
}
```

Figure IV.6 Méthode enterAttrimeth

La méthode présente ci-dessus nous donne les attributs (les paramètres) de la méthode et le type de retour.

IV.1.4 présentation de l'interface principale :

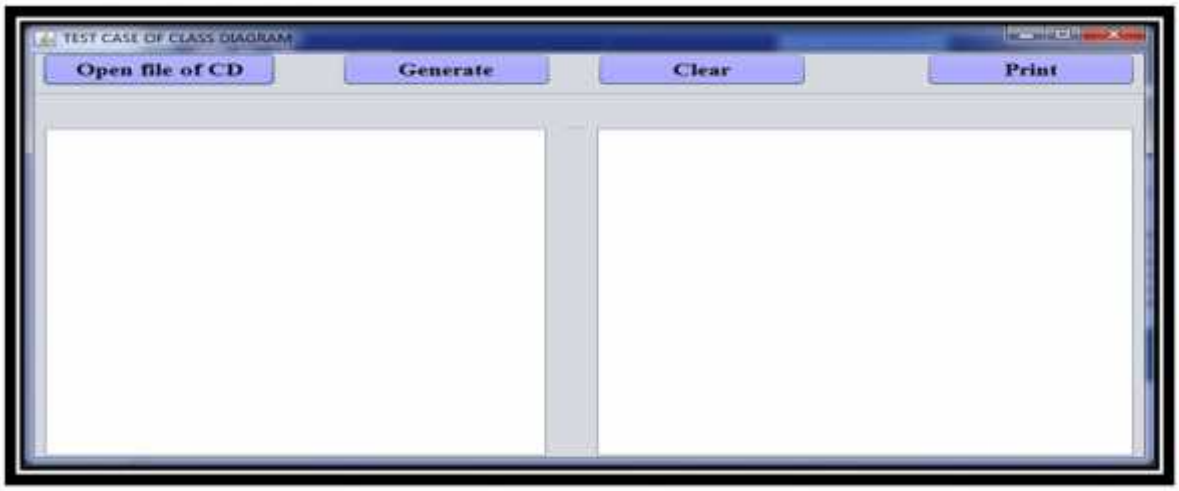


Figure IV.7 l'interface principale

L'interface principale contient les boutons suivants :

- Open file of CD : il permet d'ouvrir le diagramme de classes sous forme de texte déjà compilé.
- Generate : pour générer des cas de test à partir le fichier ouvrier avec le bouton Open file of CD
- Clear : efface le contient de fichier générer.
- Print : pour imprimer les cas de test.

IV.1.4.1 Ouvrir le fichier de diagramme de classes

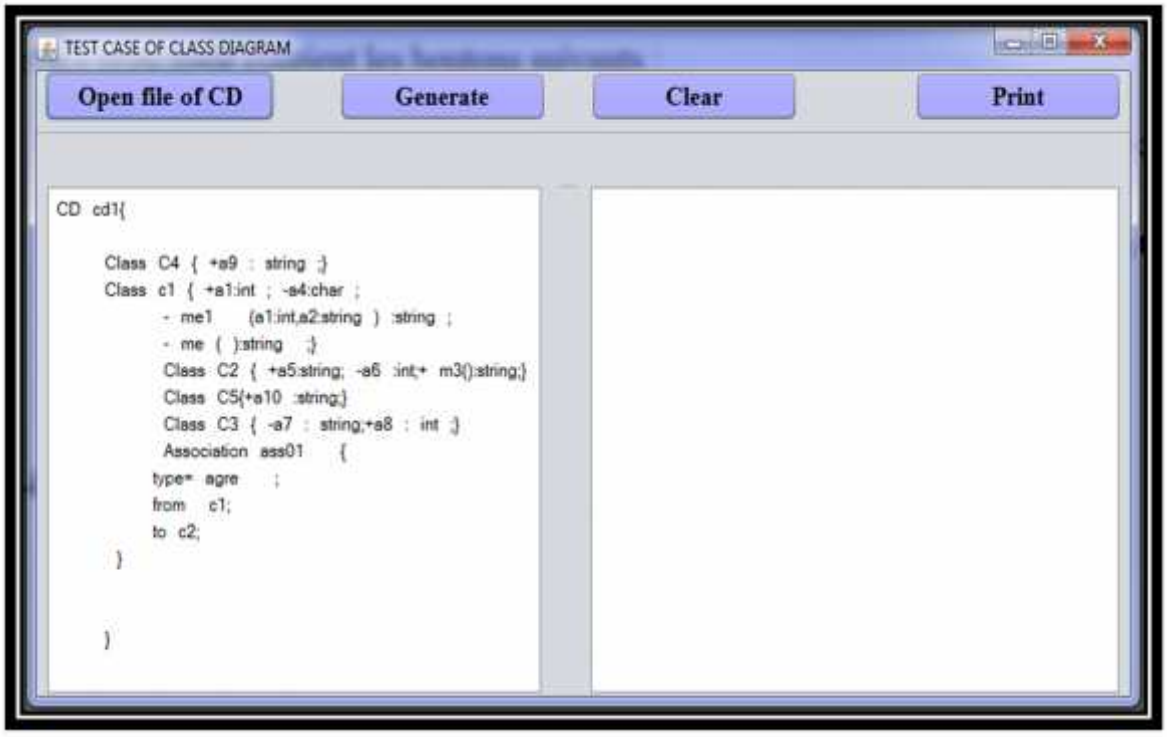


Figure IV.8 Ouvrir fichier de CD

IV.1.4.2 Générer les cas de test

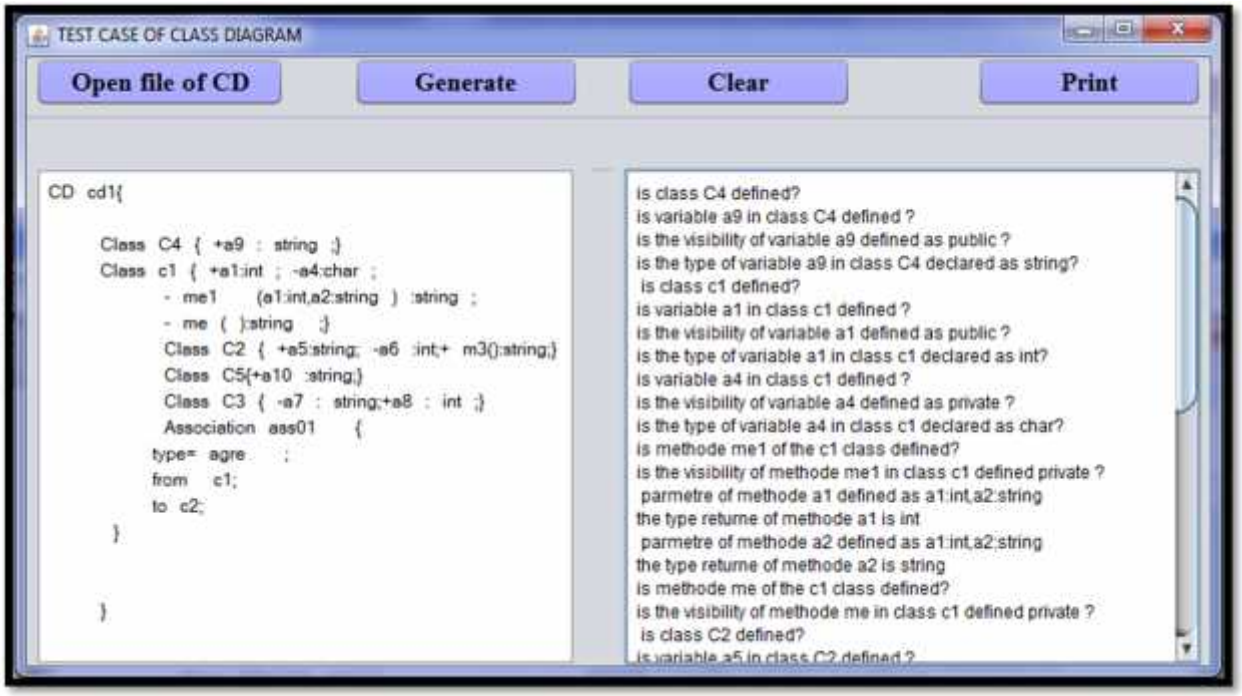


Figure IV.9 Génération les cas de test

Conclusion générale

Conclusion Générale

Ce mémoire de fin d'études présente la réalisation d'un générateur de cas de test à partir d'un diagramme de classe. Nous avons proposé une approche pour la réalisation d'un compilateur de diagramme de classe pour générer des cas de test, basée sur le générateur ANTLR.

Cette approche permet de compiler le diagramme de classe à partir d'une grammaire fondé selon les règles d'ANTLR.

Cette grammaire nous donne un choix vaste dans l'écriture de code source, et l'extraction des données se fait par l'arbre AST, la dite arbre est généré par ANTLR

Pendant l'extraction des données on fait le parcours de l'arbre et générer les cas de tests, par exemple (vérification de la définition des classes, des méthodes, etc…).

Finalement, cette nouvelle approche permet d'une part de compiler le diagramme de classes qui déjà crée dans la grammaire et d'autre part générer les cas de test, pour la vérification et la validation de diagramme.

Perspectives :

La compilation de diagramme de classe pour la génération des cas de tests basé sur ANTLR est un sujet intéressant dans la recherche, pour la continuité et contribution dans ce domaine nous recommandons :

Utiliser une nouvelle approche permet de transformer le diagramme de classes d'une forme graphique vert notre application et étendre notre approche pour aller jusqu'à la vérification de la sémantique de diagramme de classe.

Bibliographie

[1] Alfred Aho ,Monica Lam,Ravi Sethi, Jeffrey Ullman , 2007 Compilateurs : principes, techniques et outils - 2e édition, Editeur : Pearson,923 pages.

[2] Benoît Charroux,Aomar Osmani,Yann Thierry-Mieg ,2009, UML2 Pratique de la modélisation ,Edition 2 , paris,Pearson Education France ,256 page

[3] Dick henri, e.bal,Ceriel j.h.jacobs ,Koen g.langendoen ,2002, compilateurs Cours et exercices corrigés, paris,Dunod, 774 pages .

[4] John R. Levine ,Tony Mason ,Doug Brown,1992, lex & yacc, 2nd Edition, O'Reilly Media,388pages

[5] Joseph Gabay,David Gabay ,2008 , UML 2 Analyse et conception Mise en œuvre guidée avec études de cas, paris,dunod, 240 pages

[6] Henri Garreta, 2001, Techniques et outils pour la compilation, 67 pages.

[7] MicroSystem, S. 2000. JJTree home page https://javacc.dev.java.net/doc/JJTree.html.

[8] Parr, T. J. et Quong, R W. 1995. ANTLR: A predicated- LL(k) parser generator, Software Practice and Experience, page. 789- 810.

[9] Terence Parr. , 2013,The Definitive ANTLR 4 Reference Second Edition. Pragmatic Bookshelf, 328 page.

[10] Terence Parr,2004, ANTLR Reference Manual,152pages

[11] Tetsuro Katayama Yusuke Yabuya ,2005, “Proposal of a Method to Support Testing for Java Programs with UML”, Proceedings of the 12th Asia-Pacific Software Engineering Conference , IEEE

[12] Xavier Blanc ,Isobelle Moumier ,2006,UML 2 pour les développeurs : Cours avec exercices corrigés , Editeur : Eyrolles,202 pages

[13] Yves MOUAFO,2013, Tutoriel sur la mise en place d'une grammaire avec ANTLR, http://yvesmouafo.developpez.com/tutoriels/java/grammaire-antlr/

[14] Jane Huffman Hayes, 1994, "Testing Object-Oriented Programming Systems (OOPS): A Fault Based Approach", The Proceeding of the International Symposium on Object Oriented Methodology and System (lSOOMS).

[15] Ian Sommerville, 2004,"Software Engineering. Seventh Edition", Pearson Education Limited, England.

[16] Johannes Ryser, Stefan Berner, Matiin Glinz, 1998, "On the State of the Art in Requirements-Based Validation and of Software ", Technical Report 98-12, Institute fur Informatics, University of Zurich.

[17] Richard Bender, 2003,"Introduction to Requirements-based Testing", Borland Conference, BorCon, San Jose, California.

[**18] Aynur Abdurazik, Jeff Offutt**,2000, "Using UML Collaboration Diagrams for Static Checking And Test Generation", The Third International Conference on the Unified Modeling Language , York, UK.

[19] Elizabeta FOURNERET, 2012, thèse "Génération de tests à partir de modèles UML/OCL pour les systèmes critiques évolutifs, Université de Franche-Comté, France

[20] Paul Amman, Paul E. Black, 2000,"Test Generation and Recognition with Formal Methods". The First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland.

[21] Roger S. Pressman, 2004,"Software Engineering A Practitioner's Approach", The McGraw-hill Companies, Inc, Sixth Edition,

Abstract

We have realize a UML class diagram compiler based on ANTLR for generating test cases automatically.

We have used ANTLR to generate our lexer and parser, in java language, from a grammar proposed for UML class diagram.

We have used the set of generated java classes in a separated tool to demonstrate the utility of our approach.

Key word: complier, ANTRL, grammar, class diagram, Test case,

.

Résumé

Nous avons réalisé dans ce travail un compilateur de diagramme de classes UML basant sur l'ANTLR pour la génération automatique des cas de test.

Nous avons utilisé ANTLR pour générer le parser et le lexer en java, à partir d'une grammaire de diagramme de classes.

ANTLR généré une ensemble de classes java qu'on a utilisé pour développer un outil séparé, afin de montrer l'utilité de notre approche.

Mots clé : compilateur, ANTLR, grammaire, diagramme de classes, cas de test.

ملخص:

نامل من خلال هذا العمل بإنجاز مترجم لمخططات الصنفيات UML اعتمادا على اداة ANTLR من اجل التوليد الالي لحالات الاختبار.

تعتمد هذه الترجمة على أداة ANTLR من اجل توليد التحليل اللفظي و المفردات و التحليل القواعدي الصرفي لإنشاء حالات الاختبار انطلاقا من قواعد مخططات التصنيف.

ANTLR يولد مجموعة منclasses java التي تم استخدامها من اجل تطوير هذه الأداة المنفصلة, وشرح مدى نجاعة و فعالية هذه الطريقة.

كلمات مفتاحية: ANTLR, قواعد النحو, تحليل النحوي, تحليل المفردات,مترجم , مخططات الصنفياتUML