



DOSSIER INTERPRÉTEUR DU LANGAGE LIR

PROJET PROPOSÉ PAR FRÉDÉRIQUE BARRIOS

Nicolas CAMINADE, Sylvan COURTIOL,
Pierre DEBAS, Heïa DEXTER,
Lucàs VABRE

Sommaire

Sommaire	4
I Plan projet (annexe)	5
II Spécifications détaillées	6
Introduction	7
1 Première itération	8
1.1 Commande	9
1.2 Commande debut	10
1.3 Commande fin	10
1.4 Commande defs	11
1.5 Commande affiche	11
1.6 Commande affiche avec une expression	11
1.7 Commande var pour une chaîne de caractères	12
1.8 Commande var pour un entier	12
1.9 Expression concaténation sur chaîne de caractères	13
1.10 Expression logique	13
1.11 Expression arithmétique	14
2 Deuxième itération	15
2.1 Commande efface	16
2.2 Commande lance	17
2.3 Commande stop	17
2.4 Etiquette	17
2.5 Instruction	18
2.6 Instruction vaen	18
2.7 Commande lance à partir d'une étiquette	19
2.8 Instruction procédure	19
2.9 Instruction retour	20
2.10 Commande liste	20
2.11 Instruction	21

3	Troisième itération	22
3.1	Commande sauve	23
3.2	Commande charge	24
3.3	Instruction si... vaen	24
III	Conception	26
1	Itération 1	27
1.1	Paquetage interpreteurlir.donnees.litteraux	27
1.2	Paquetage interpreteurlir.donnees	28
1.3	Paquetage interpreteurlir.expressions	28
1.4	Paquetage interpreteurlir.motscles	29
1.5	Paquetage interpreteurlir	30
1.6	Illustration avec des diagrammes d'objets	31
2	Itération 2	32
2.1	Diagrammes d'objets	32
2.2	Paquetage interpreteurlir.donnees(.litteraux)	33
2.3	Paquetage interpreteurlir.expressions	33
2.4	Paquetage interpreteurlir.programmes	34
2.5	Paquetage interpreteurlir.motscles	35
2.6	Paquetage interpreteurlir.motscles.instructions	36
2.7	Paquetage interpreteurlir	37
3	Itération 3	38
3.1	Diagrammes d'objets	38
3.2	Paquetage interpreteurlir.donnees(.litteraux)	39
3.3	Paquetage interpreteurlir.expressions	40
3.4	Diagramme de classes général	41
4	Projet final	42
4.1	Paquetage interpreteurlir.donnees(.litteraux)	42
4.2	Paquetage interpreteurlir.expressions	43
4.3	Paquetage interpreteurlir.programmes	44
4.4	Paquetage interpreteurlir.motscles	45
4.5	Paquetage interpreteurlir.motscles.instructions	46
4.6	Paquetage interpreteurlir	47
IV	Codage	
	(annexe)	48
V	Tests	49
	Démarche globale	50

5 Tests du paquetage interpreteurlir.donnees.litteraux	51
5.1 Litteral	51
5.2 Chaîne	51
5.3 Entier	51
5.4 Booleen	51
6 Tests du paquetage interpreteurlir.donnees	52
6.1 Identificateur	52
6.2 IdentificateurChaîne et IdentificateurEntier	52
6.3 Variable	52
7 Tests du paquetage interpreteurlir.expressions	53
7.1 Expression	53
7.2 ExpressionChaîne	53
7.3 ExpressionEntier	53
7.4 ExpressionBoolenne	53
8 Tests du paquetage interpreteurlir	54
8.1 InterpreteurException et ExecuteurException	54
8.2 Contexte	54
8.3 Analyseur	54
9 Tests du paquetage interpreteurlir.programmes	55
9.1 Etiquette	55
9.2 Programme	55
9.3 Les programmes de tests	55
10 Tests du paquetage interpreteurlir.motscles	56
10.1 Commande	56
10.2 EssaiCommande	56
10.3 CommandeCharge	56
10.4 CommandeDebut	57
10.5 CommandeDefs et CommandeFin	57
10.6 CommandeEfface, CommandeLance et CommandeListe	57
10.7 CommandeSauve	57
11 Tests du paquetage interpreteurlir.motscles.instructions	58
11.1 Instruction	58
11.2 InstructionAffiche, InstructionEntre et InstructionSi(Vaen)	58
11.3 InstructionProcedure, InstructionRetour, InstructionStop et InstructionVaen	58
11.4 InstructionVar	58

VI Conclusion	59
12 Conception et implémentation	60
12.1 Le livrable	60
12.2 Conception	60
13 Organisation du groupe	61
13.1 Travail en binôme	61
13.2 Répartition de la charge de travail	61
13.3 Communication	62
14 Conclusion générale	63
VII Manuel utilisateur (annexe)	64

Première partie

Plan projet (annexe)

Deuxième partie

Spécifications détaillées

Introduction

Le projet interpréteur LIR a été réalisé selon un modèle de cycle de vie itératif. Dans ce document des spécifications détaillées du projet seront présentées les fonctionnalités ajoutées à l'interpréteur au cours de chaque itération avec les récits d'utilisation des fonctionnalités ajoutées ou des éléments nécessaires au bon fonctionnement de l'interpréteur LIR.

Chapitre 1

Première itération

Contenu de la première itération

Dans cette première itération, l'objectif est d'avoir un prototype de l'interpréteur avec des fonctionnalités de base. Ces premières fonctionnalités sont les commandes et instructions suivantes :

- Commande **debut** qui efface toutes les lignes de programme mémorisées ainsi que tous les identificateurs mémorisés.
- Commande **fin** qui quitte l'interpréteur.
- Commande **defs** qui affiche le contexte de l'interpréteur, i.e. affiche la liste des identificateurs définis durant la session avec leur valeur.
- Instruction **affiche** qui évalue la valeur de l'expression et l'affiche sur la sortie texte courante ou alors provoque un saut de ligne sur la sortie texte courante.
- Instruction **var** qui affecte la valeur de l'expression à la variable nommée par l'identificateur.

Récits d'utilisation proposés lors de l'itération 1

1.1 Commande

Récit d'utilisation

Titre : Exécution d'une commande

Récit : Exécution d'une commande

En tant que : programmeur avec l'interpréteur LIR

Je souhaite : exécuter une commande

Afin de : obtenir le résultat de cette commande ou une confirmation de son exécution

Critères d'acceptation

À partir du fait : l'interpréteur affiche un invite

Alors : j'entre une ligne de commande

Enfin : j'obtiens le résultat de cette commande ou un retour m'informant du bon déroulé de l'exécution de la commande ou de son échec.

1.2 Commande debut

Récit d'utilisation

Titre : debut

Récit : Réinitialiser l'environnement de l'interpréteur LIR

En tant que : programmeur

Je souhaite : vider l'intégralité du contexte d'exécution

Afin de : obtenir un environnement de travail vierge

Critères d'acceptation

À partir de : d'une session de l'interpréteur LIR

Alors : j'entre la commande `debut`

Enfin : L'interpréteur efface toutes les lignes de programme mémorisées ainsi que tous les identificateurs mémorisés

1.3 Commande fin

Récit d'utilisation

Titre : Commande fin

Récit : Quitter l'interpréteur

En tant que : programmeur avec l'interpréteur LIR

Je souhaite : quitter l'interpréteur LIR et avoir un message m'informant de la fermeture de session

Afin de : fermer la session courante de l'interpréteur LIR

Critères d'acceptation

À partir de : une session de l'interpréteur LIR ouverte

Alors : je souhaite quitter l'interpréteur et fermer la session courante en exécutant la commande `fin`

Enfin : le processus courant de l'interpréteur LIR s'arrête

1.4 Commande defs

Récit d'utilisation

Titre : Affichages du contexte courant (commande defs)

Récit : Affichages du contexte courant (commande defs)

En tant que : programmeur avec l'interpréteur LIR

Je souhaite : voir toutes les variables définies dans la session courante (identificateur et valeur)

Afin de : connaître le contexte actuel de la session courante de l'interpréteur

Critères d'acceptation

À partir du fait : des variables sont définies dans la session courante de l'interpréteur

Alors : je souhaite connaître le contexte actuel en exécutant la commande defs

Enfin : l'interpréteur affiche chaque variable ligne par ligne avec son identificateur et sa valeur

1.5 Commande affiche

Récit d'utilisation

Titre : Faire un saut de ligne avec la commande affiche

Récit : Provoquer le saut de ligne sur la sortie de texte courante

En tant que : Programmeur

Je souhaite : que l'interpréteur LIR saute une ligne sur la sortie de texte courante

Afin de : Provoquer un saut de ligne sur cette sortie

Critères d'acceptation

À partir du fait : que j'ai une sortie de texte courante

Alors : j'entre la commande affiche

Enfin : l'interpréteur saute une ligne sur la sortie de texte courante

1.6 Commande affiche avec une expression

Récit d'utilisation

Titre : Commande affiche (expression)

Récit : Afficher le contenu d'une expression sur la console de l'interpréteur

En tant que : Programmeur

Je souhaite : que l'interpréteur LIR évalue et affiche le contenu de l'expression que l'on lui donne

Afin de : d'afficher le résultat de l'expression en argument

Critères d'acceptation

À partir de : l'interpréteur affichant un invite

Alors : j'entre la commande affiche et écrit l'expression dont je veux le résultat affiché

Enfin : l'interpréteur affiche le résultat de l'expression

1.7 Commande var pour une chaîne de caractères

Récit d'utilisation

Titre : Commande var (Chaîne de caractères)

Récit : Initialiser une chaîne de caractère dans variable / Changer sa valeur

En tant que : Programmeur

Je souhaite : que l'interpréteur LIR stock une chaîne dans une variable

Afin de : pouvoir récupérer/manipuler cette chaîne plus tard dans le programme

Critères d'acceptation

À partir du fait : que j'ai la possibilité de saisir une ligne de commande

Alors : je tape la commande var et met une chaîne de caractère entre double guillemets comme valeur : var <nomVariable>=<chaîne>

Enfin : l'interpréteur enregistre dans la variable spécifié la chaîne de caractère voulue et renvoie la variable suivie de sa valeur (en tant que feed-back)

1.8 Commande var pour un entier

Récit d'utilisation

Titre : Commande var (Entier)

Récit : Initialiser un entier dans variable / Changer sa valeur

En tant que : Programmeur

Je souhaite : que l'interpréteur LIR stock un entier dans une variable

Afin de : pouvoir récupérer/manipuler cet entier plus tard dans le programme

Critères d'acceptation

À partir du fait : que j'ai la possibilité de saisir une ligne de commande

Alors : je tape la commande var et met un entier comme valeur : `|var<nomVariable>=<entier>`
|

Enfin : l'interpréteur enregistre dans la variable spécifié l'entier voulu et renvoie la variable suivie de sa valeur (en tant que feed-back)

1.9 Expression concaténation sur chaîne de caractères

Récit d'utilisation

Titre : Opérateur + sur les chaînes de caractères

Récit : Concaténation de chaînes

En tant que : Programmeur

Je souhaite : accoler deux chaînes l'une à la suite de l'autre

Afin de : créer des messages dépendant du contexte d'exécution sur la sortie standard.
Représenter une valeur entière par son écriture chiffrée en base 10.

Critères d'acceptation

À partir de : deux chaînes de caractères ou une chaîne et un entier, en tant qu'identificateurs déclarés ou expressions littérales.

Alors : En utilisant une expression de type `var nouvelleChaine = opeGauche + opeDroite`, j'obtiens la concaténation de deux chaînes.

Enfin : L'identificateur `nouvelleChaine` contient la chaîne constituée des deux primordiales concaténées. L'interpréteur confirme en affichant la nouvelle valeur ou m'informe d'une erreur. L'opération peut être récursive mais n'est pas commutative. Une concaténation s'effectue toujours par la droite.

1.10 Expression logique

Récit d'utilisation

Titre : Expression logique dans un branchement conditionnel

Récit : Opérations relationnelles sur deux entiers ou sur deux chaînes de caractères

En tant que : Programmeur

Je souhaite : que l'Interpréteur LIR compare deux entiers avec une relation d'ordre ou d'équivalence

Afin que : d'exécuter ou non une branche du code avec l'instruction `si <expression> va en`

Critères d'acceptation

À partir de : d'une ligne de programme à mémoriser et d'identificateurs auxquels une valeur aura été affectée préalablement ou de constantes littérales de même type.

Alors : j'entre une expression composée de deux opérandes de même type et l'interpréteur évalue l'expression.

Les opérandes peuvent être :

- deux constantes littérales
- deux identificateurs
- une constante littérale et un identificateur

Enfin : si l'expression (condition dans l'instruction) est vraie alors l'exécution continuera à partir du numéro de ligne spécifié par l'étiquette, sinon l'exécution continuera en séquence.

1.11 Expression arithmétique

Récit d'utilisation

Titre : Expression arithmétique

Récit : Calculer à l'aide d'expression arithmétique

En tant que : Programmeur

Je souhaite : que l'Interpréteur LIR effectue une opération arithmétique courante (addition, soustraction, multiplication, quotient ou reste d'une division entière)

Afin que : j'en exploite ou vois le résultat

Critères d'acceptation

À partir de : d'une ligne de l'interpréteur ou d'une ligne de programme à mémoriser et d'identificateurs auxquels une valeur aura été affectée préalablement ou de constantes littérales numériques.

Alors : j'entre une expression composée de deux opérandes de type entier signé et d'un opérateur.

Les opérandes peuvent être :

- deux constantes littérales
- deux identificateurs
- une constante littérale et un identificateur

Enfin : j'obtiens le résultat de l'opération ou un message d'erreur m'informant que l'opération est impossible pour les identificateurs ou constantes littérales saisies.

Chapitre 2

Deuxième itération

Contenu de la deuxième itération

Dans cette deuxième itération, l'objectif est d'ajouter des fonctionnalités permettant l'écriture de programmes simple en LIR, à savoir :

- Commande **efface** qui efface toutes les lignes de programme dont le numéro d'étiquette est dans la plage comprise entre `<etiquette_debut>` et `<etiquette_fin>`.
- Commande **lance** qui démarre l'exécution d'un programme à partir de son plus petit numéro d'étiquette ou du numéro d'étiquette indiqué par l'utilisateur.
- Commande **liste** qui affiche toutes les lignes de programme mémorisées dans l'ordre croissant des numéros de ligne.
- Instruction **stop** qui arrête l'exécution du programme.
- Instruction **vaen** qui continue l'exécution à partir du numéro spécifié par étiquette.
- Instruction **procedure** qui transfère l'exécution du programme au numéro d'étiquette spécifié et qui reprendra en séquence lorsque la procédure sera terminée.
- Instruction **retour** qui, rencontrée après un appel de procédure, provoque un retour à l'instruction qui suit son appel.

Récits d'utilisation proposés lors de l'itération 2

2.1 Commande efface

Récit d'utilisation

Titre : Commande efface

Récit : Utilisation de la commande efface

En tant que : Programmeur

Je souhaite : Supprimer une ou plusieurs lignes d'un programme

Afin de : Effacer les instructions d'un bloc de code

Critères d'acceptation

À partir de : une ou plusieurs lignes de programme mémorisé et leur étiquettes

Alors : on tape la commande : efface <etiquette_debut> : <etiquette_fin>

Enfin : l'interpréteur efface les lignes de programme dont le numéro d'étiquette est compris dans la plage, comprise entre `etiquette_debut` et `etiquette_fin`

2.2 Commande lance

Récit d'utilisation

Titre : Commande lance sans argument

Récit : Exécuter le programme à partir de l'étiquette la plus petite

En tant que : Programmeur avec l'interpréteur LIR

Je souhaite : Exécuter le programme chargé avec la commande lance

Afin de : obtenir le comportement du programme chargé pour atteindre son objectif

Critères d'acceptation

À partir de : lignes d'instructions chargé dans la session courante de l'interpréteur LIR

Alors : lorsque j'entre la commande lance sans arguments et la valide le programme s'exécute à partir de l'étiquette la plus petite

Enfin : le contexte de l'interpréteur contient le contexte final du programme exécuté

2.3 Commande stop

Récit d'utilisation

Titre : Commande stop

Récit : Utilisation de la commande stop

En tant que : Programmeur

Je souhaite : Arrêter un programme

Afin de : terminer son execution

Critères d'acceptation

À partir du fait : Qu'un programme comporte au moins une instruction

Alors : on tape la commande : <etiquette> stop

Enfin : À son exécution, le programme s'arrête lorsqu'il a atteint l'étiquette indiquée. Puis l'interpréteur affiche de nouveau un invite.

2.4 Etiquette

Récit d'utilisation

Titre : Étiquettes

Récit : Ordonner les lignes d'un programme avec les étiquettes

En tant que : Programmeur avec l'interpréteur LIR

Je souhaite : ajouter des instruction au programmes dans un ordre précis

Afin de : que les instructions puissent être exécutées dans le bon ordre

Critères d'acceptation

À partir de : l'interpréteur LIR et des instructions définies

Alors : lorsque j'entre une instruction précédée d'une étiquette alors celle-ci est enregistrée avec son étiquette pour pouvoir être exécutée plus tard.

Enfin : lorsque le programme est lancé alors les instructions s'exécutent l'ordre des étiquettes.

2.5 Instruction

Récit d'utilisation

Titre : Instructions

Récit : Consulter et modifier le contexte d'exécution

En tant que : programmeur

Je souhaite : faire réaliser des actions par l'interpréteur

Afin de : déclarer des variables, des fonctions, effectuer des sauts conditionnels, des itérations, connaître et manipuler le contexte d'un programme.

Critères d'acceptation

À partir de : ligne de commande ou programme

Alors : J'entre une instruction pour effectuer une action précise

Enfin : Le contexte est modifié en fonction de cette instruction. L'interpréteur m'informe en cas d'erreur de syntaxe

2.6 Instruction vaen

Récit d'utilisation

Titre : Instruction `vaen`

Récit : Sauts inconditionnels

En tant que : programmeur

Je souhaite : effectuer un saut vers une ligne spécifique d'un programme.

Afin de : Créer des branchements ou des itérations dans mes programmes.

Critères d'acceptation

À partir de : la saisie d'un programme

Alors : j'entre la commande **vaen** suivie du numéro de la ligne où je veux effectuer le saut.

Enfin : lors de l'exécution de l'instruction, le programme ignorera les lignes suivantes et sautera directement à la ligne indiquée.

2.7 Commande lance à partir d'une étiquette

Récit d'utilisation

Titre : Commande lance <Étiquette>

Récit : Exécuter le programme à partir de l'étiquette argument

En tant que : Programmeur avec l'interpréteur LIR

Je souhaite : Exécuter le programme chargé avec la commande lance <étiquette>

Afin de : obtenir le comportement et objectif du programme chargé

Critères d'acceptation

À partir de : lignes d'instructions chargé dans la session courante de l'interpréteur LIR

Alors : lorsque j'entre la commande lance sans arguments et la valide le programme s'exécute à partir de l'étiquette passé en argument

Enfin : le contexte de l'interpréteur contient le contexte final du programme exécuté à partir de l'étiquette spécifiée

2.8 Instruction procédure

Récit d'utilisation

Titre : Procédure

Récit : Ordonner à l'interpréteur à exécuter des lignes de code à partir de l'étiquette de l'instruction.

En tant que : Programmeur

Je souhaite : transférer l'exécution au numéro d'étiquette spécifié.

Afin de : exécuter le programme puis reprendre en séquence une fois le procédure terminée.

Critères d'acceptation

À partir de : Plusieurs lignes de code et d'identificateurs déclarés, dont la portée est globale.

Alors : En utilisant l'instruction `procedure <etiquette>`

Enfin : Alors l'interpréteur va chercher la ligne qui a pour identificateur celui référencé en étiquette et va l'exécuter jusqu'à la fin de la séquence.

2.9 Instruction retour

Récit d'utilisation

Titre : retour

Récit : Ordonner a l'interpréteur de retourner à la suite de l'instruction qui suit son appel.

En tant que : Programmeur

Je souhaite : retourner à la suite de la ligne de code qui a précédé l'appel de procédure.

Afin de : d'exécuter le programme qui allait s'exécuter si l'appel de procédure n'avait pas été fais.

Critères d'acceptation

À partir de : Plusieurs lignes de code et a la suite d'une instruction procédure.

Alors : En utilisant l'instruction `retour`

Enfin : Alors l'interpréteur va chercher la ligne qui suivait l'instruction procédure et va l'exécuter jusqu'à la fin de la séquence.

2.10 Commande liste

Récit d'utilisation

Titre : Commande liste

Récit : Utilisation de la commande liste avec argument

En tant que : Programmeur

Je souhaite : que l'Interpréteur LIR affiche toutes les lignes de programme mémorisées dans l'ordre croissant des numéros de ligne dans un intervalle donné.

Afin que : je visualise uniquement les lignes de cet intervalle dans l'ordre croissant.

Critères d'acceptation

À partir de : aucune ou plusieurs lignes de programme à mémoriser et de leurs étiquettes et d'un intervalle d'entier passé en argument

Alors : j'entre la commande `liste <etiquette_debut>:<etiquette_fin>`

Enfin : l'interpréteur affiche toutes les lignes de programme mémorisées, s'il y en a, dans l'ordre croissant de leur étiquette et dont les étiquettes sont situées dans cet intervalle donné.

2.11 Instruction

Récit d'utilisation

Titre : Commande liste

Récit : Utilisation de la commande liste sans argument

En tant que : Programmeur

Je souhaite : que l'Interpréteur LIR affiche toutes les lignes de programme mémorisées dans l'ordre croissant des numéros de ligne.

Afin que : je visualise ces lignes dans leur ordre d'exécution

Critères d'acceptation

À partir de : d'aucune ou plusieurs lignes de programme à mémoriser et de leurs étiquettes

Alors : j'entre la commande `liste`

Enfin : l'interpréteur affiche toutes les lignes de programme mémorisées, s'il y en a, dans l'ordre croissant de leur étiquette.

Chapitre 3

Troisième itération

Contenu de la première itération

Dans cette troisième itération, l'objectif est de couvrir toutes les fonctionnalités attendues. Ces dernières concernent la lecture et l'écriture de fichier et l'ajout d'une structure de contrôle à l'interpréteur :

- Commande **sauve** qui sauvegarde les lignes de programme dans le fichier texte indiqué en argument.
- Commande **charge** qui charge dans le contexte les lignes de programme sauvegardées dans le fichier texte indiqué en argument.
- Instruction **si... vaen** : si la condition est vraie alors l'exécution continuera à partir du numéro de ligne spécifié par l'étiquette, sinon l'exécution continuera en séquence.

Récits d'utilisation proposés lors de l'itération 3

3.1 Commande sauve

Récit d'utilisation

Titre : Commande sauve

Récit : Sauvegarde d'un programme dans un fichier

En tant que : Programmeur dans l'interpréteur LIR

Je souhaite : sauvegarder un programme LIR dans un fichier

Afin de : Pourvoir reprendre mon travail où je m'étais arrêté

Critères d'acceptation

À partir du fait : Qu'un programme (avec des étiquettes) ai été saisi

Alors : lorsque j'entre la commande sauve avec en argument le chemin du fichier (dans lequel on souhaite sauvegarder le travail) sauve <cheminFichier>

Enfin : les lignes de codes tapées dans l'interpréteur s'enregistrent dans le fichier passé en argument de la commande pour pouvoir être rechargées plus tard par l'interpréteur LIR avec la commande charge <cheminFichier>

3.2 Commande charge

Récit d'utilisation

Titre : Commande charge

Récit : Chargement d'un programme à partir d'un fichier

En tant que : Programmeur avec l'interpréteur LIR

Je souhaite : charger un programme LIR préalablement enregistré dans un fichier

Afin de : je puisse réutiliser un programme LIR sans repartir de zéro.

Critères d'acceptation

À partir du fait : un fichier contenant un programme LIR sur mon ordinateur

Alors : lorsque j'entre la commande charge avec en argument le chemin de ce fichier

Enfin : les lignes de codes enregistrées dans le fichier sont chargée dans le programme pour pouvoir être exécutées et/ou modifiées par l'interpréteur LIR

3.3 Instruction si... vaen

Récit d'utilisation

Titre : Instruction Si...vaen

Récit : Sauts conditionnels

En tant que : programmeur

Je souhaite : effectuer un saut vers une ligne spécifique d'un programme si la condition est remplie.

Afin de : Créer des branchements ou des itérations dans mes programmes.

Critères d'acceptation

À partir de : la saisie d'un programme

Alors : j'entre la commande **si** suivie de la condition a remplir **vaen** suivie du numéro de la ligne où je veux effectuer le saut.

Enfin : lors de l'exécution de l'instruction, le programme ignorera les lignes suivantes et sautera directement à la ligne indiquée si il valide la condition imposée.

Troisième partie

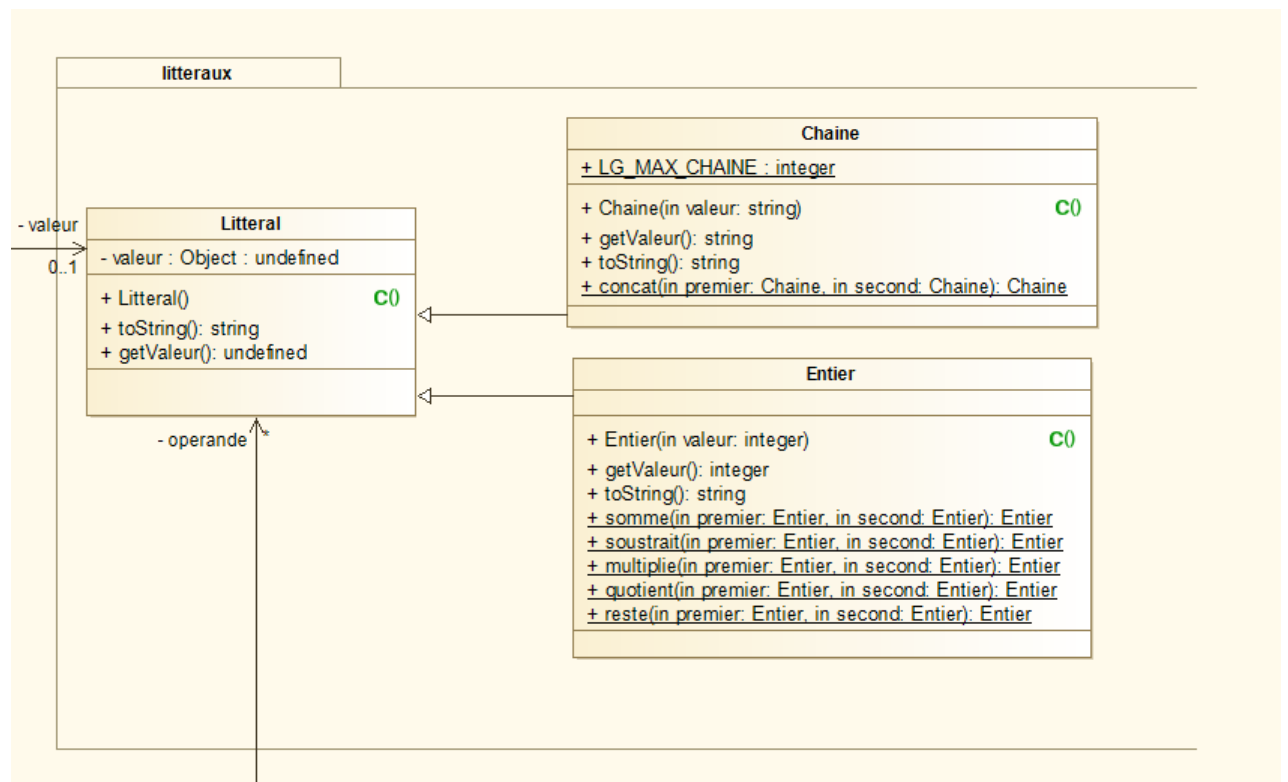
Conception

Chapitre 1

Itération 1

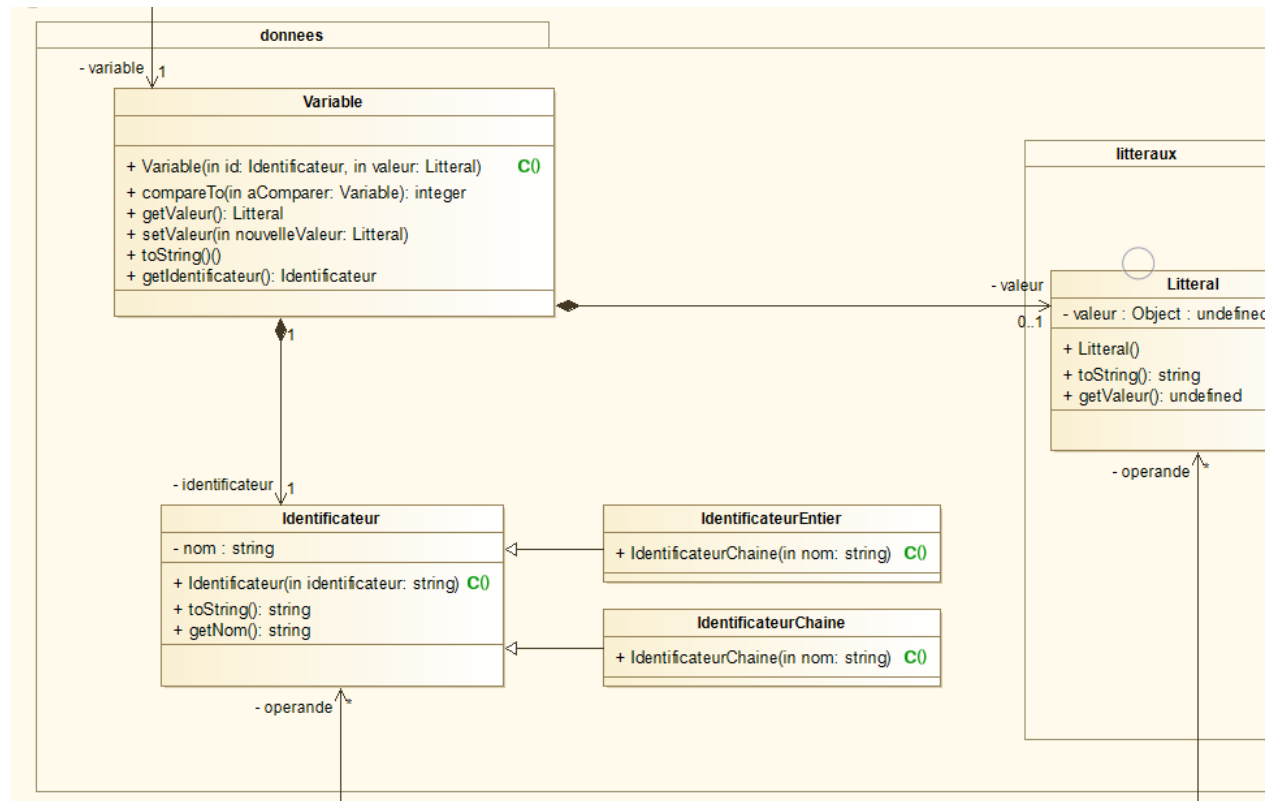
L'objectif de l'itération 1 était un prototype qui devait premièrement pouvoir se lancer et s'éteindre. De plus le prototype devait pouvoir gérer (mémorisation, affectation) des données de type chaînes. Les commandes debut, defs, fin et l'instruction var ont donc été ajoutés afin d'obtenir ces fonctionnalités.

1.1 Paquetage interpreteurlir.donnees.litteraux



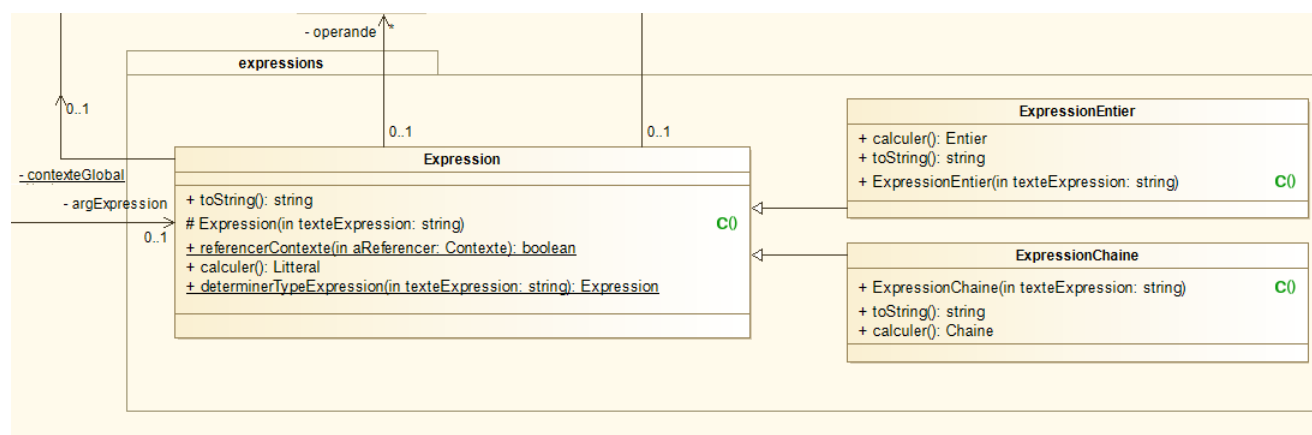
Le choix de conception des littéraux a été une classe parente `Literal` qui permet d'englober tous les types de données du programme. La classe `Entier` a été détaillée dans la conception cependant elle n'a pas été codée à cette itération pour se concentrer sur les chaînes. Les littéraux sont immuables pour permettre leur passage sans problème.

1.2 Paquetage interpreteurlir.donnees



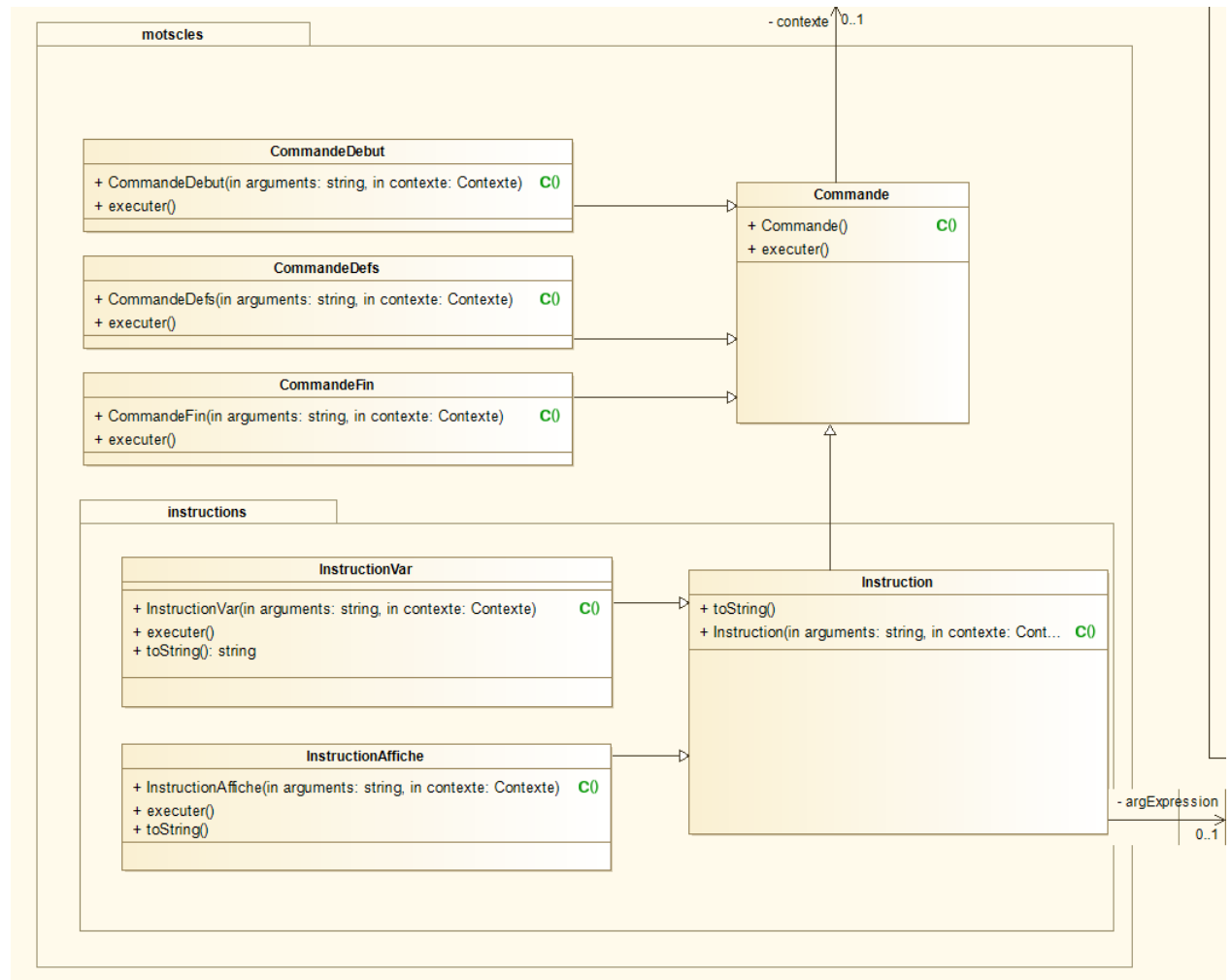
Pour les données une classe variable a été choisie composée d'un littéral et d'un identificateur. L'identificateur a comme classes dérivées les deux types affectables du projet soit les entiers et les chaînes.

1.3 Paquetage interpreteurlir.expressions



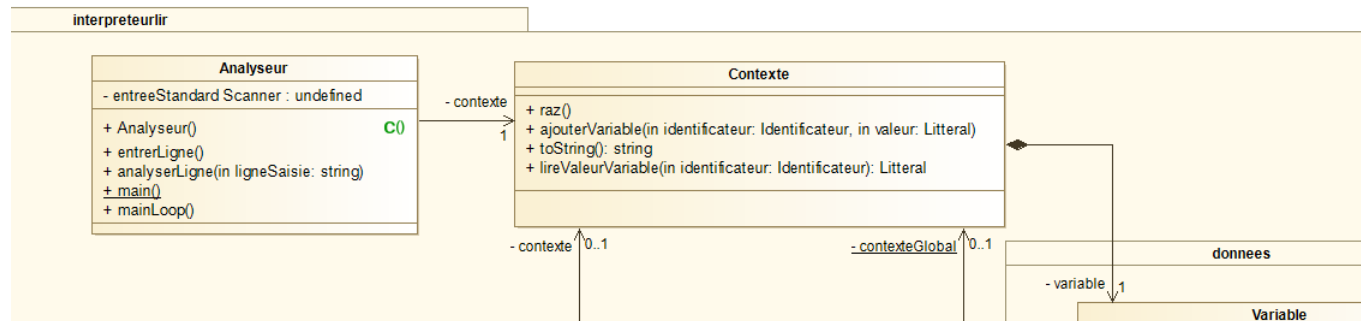
Comme pour le reste de notre conception les expressions sont typées et sont une spécialisation d'une classe Expression générale regroupant les comportements communs. Une méthode de classe d'Expression permet de créer le bon type d'expression.

1.4 Paquetage interpreteurlir.motscles



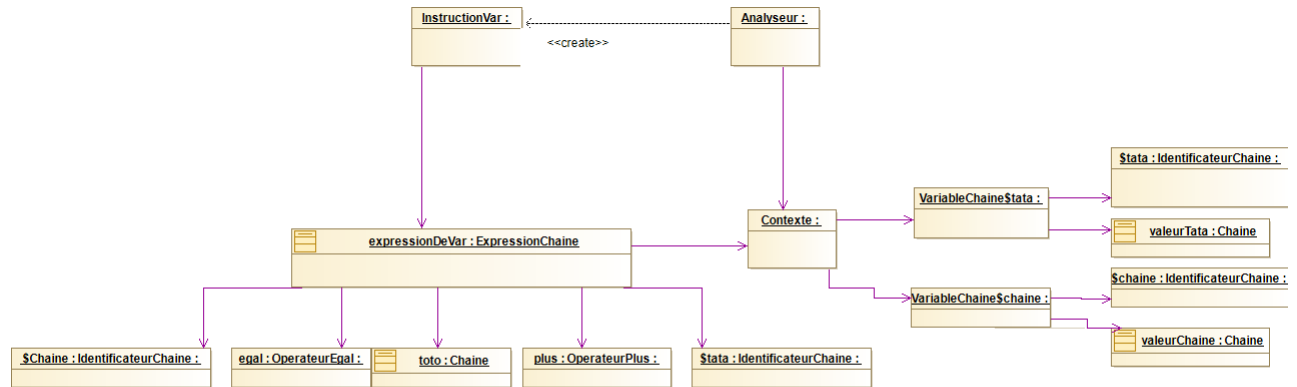
La conception de l'itération 1 contient ce qui devait être fait lors de cette itération à quelques détails près comme la classe `InstructionAffiche` qui n'a pas été codée car non nécessaire aux fonctionnalités choisies. L'itération 1 voulait permettre de manier des chaînes il fallait donc que les commandes connaissent le contexte contenant les variables. La solution choisie a été une attribut d'instance dans `Commande` initialiser à la construction de la commande par passage de la référence du contexte global par le constructeur. Une instance de commande correspond à un objet ayant toutes les informations nécessaire pour être exécuté (String arguments dans le constructeur). Les commandes et instructions fonctionnent en 2 temps, la construction qui valide les arguments et créer les éléments nécessaires à l'exécution puis l'exécution qui est la réalisation du comportement de la commande.

1.5 Paquetage interpreteurlir

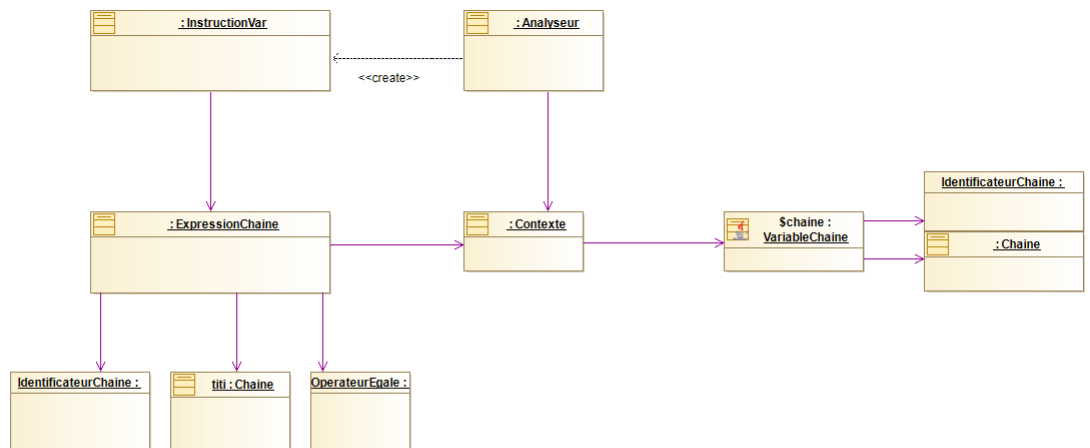


Le contexte regroupe l'entièreté des variables définies dans la session courante. Une variable n'est accessible que par l'intermédiaire du contexte grâce à l'identificateur qui sert de clé. L'Analyseur est la classe qui permet le fonctionnement de tout. Une mainLoop permet de demander en continue une ligne à l'utilisateur puis celle-ci est analysée, à partir du mot clé une commande/instruction est créée en passant le reste de la ligne en argument. L'analyse des arguments se fait au niveau le plus interne possible (Analyseur analyse le mot clé, la commande les arguments qui construit ensuite les éléments dont elle a besoin qui s'occupe eux-mêmes de vérifier leur validité à la construction). Si une erreur dans la ligne à interpréter est détectée alors une InterpreterException est levée et se propage jusqu'à l'analyseur qui affiche l'erreur.

1.6 Illustration avec des diagrammes d'objets



var \$chaine = "toto" + \$tata



var \$chaine = "titi"

Voici des diagrammes qui ont été faits pendant la réflexion de cette conception. Ils permettent d'illustrer le fait qu'une instruction créer les éléments dont elle a besoin. Seul changement dans la conception par rapport à ces diagrammes : les opérateurs sont gérés en interne des instructions (il n'y a pas de classe `Operateur`).

Chapitre 2

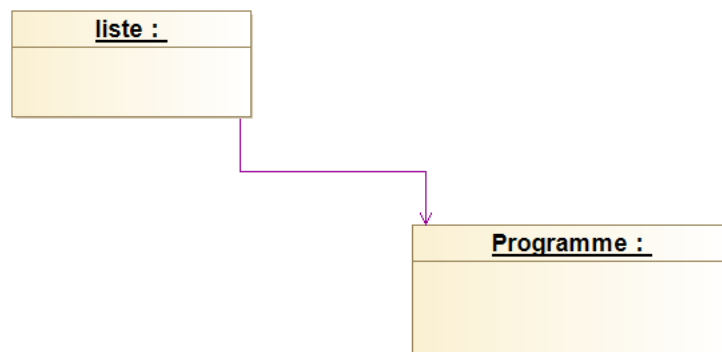
Itération 2

L'itération 2 avait pour objectif d'ajouter le type entier. Puis il fallait pouvoir faire un programme, c'est-à-dire des instructions ordonnées avec des étiquettes exécutables plus tard. Pour compléter les objectifs de cette itération certaines commandes et instructions ont été réalisées (efface, liste, lance/affiche, entre, vaen, procedure, stop, retour).

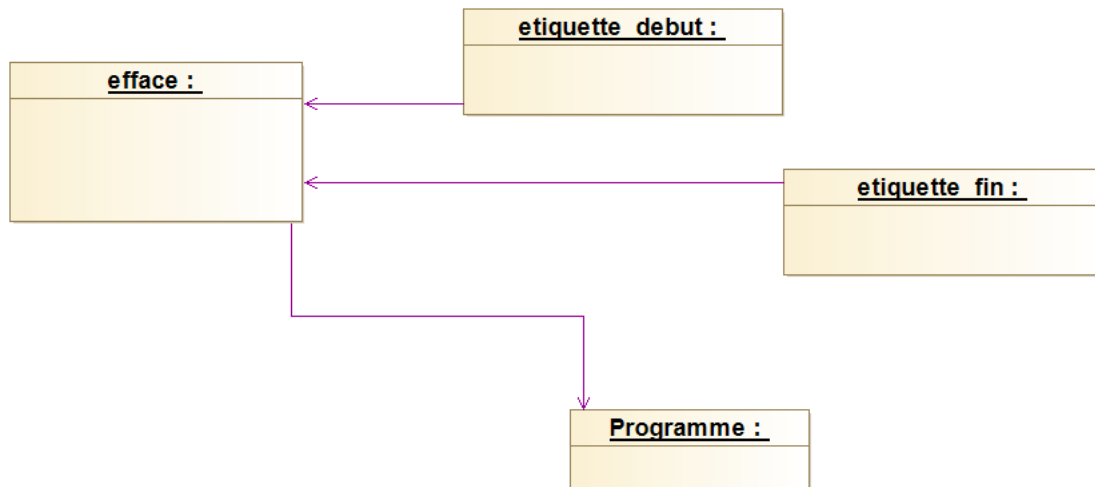
2.1 Diagrammes d'objets

Comme conseillé par notre tuteur, nous avons commencé la conception de l'itération 2 par des diagrammes d'objets. Ci-dessous quelques exemples.

liste



Le premier montre que la commande liste fait appel au programme (contenant les lignes de codes constituant un programmes) pour exécuter son comportement.



La commande efface connaît donc les deux étiquettes qui définissent son comportement spécifique d'instance. Pour son exécution elle doit connaître le programme global de la session courante de l'interpréteur LIR.

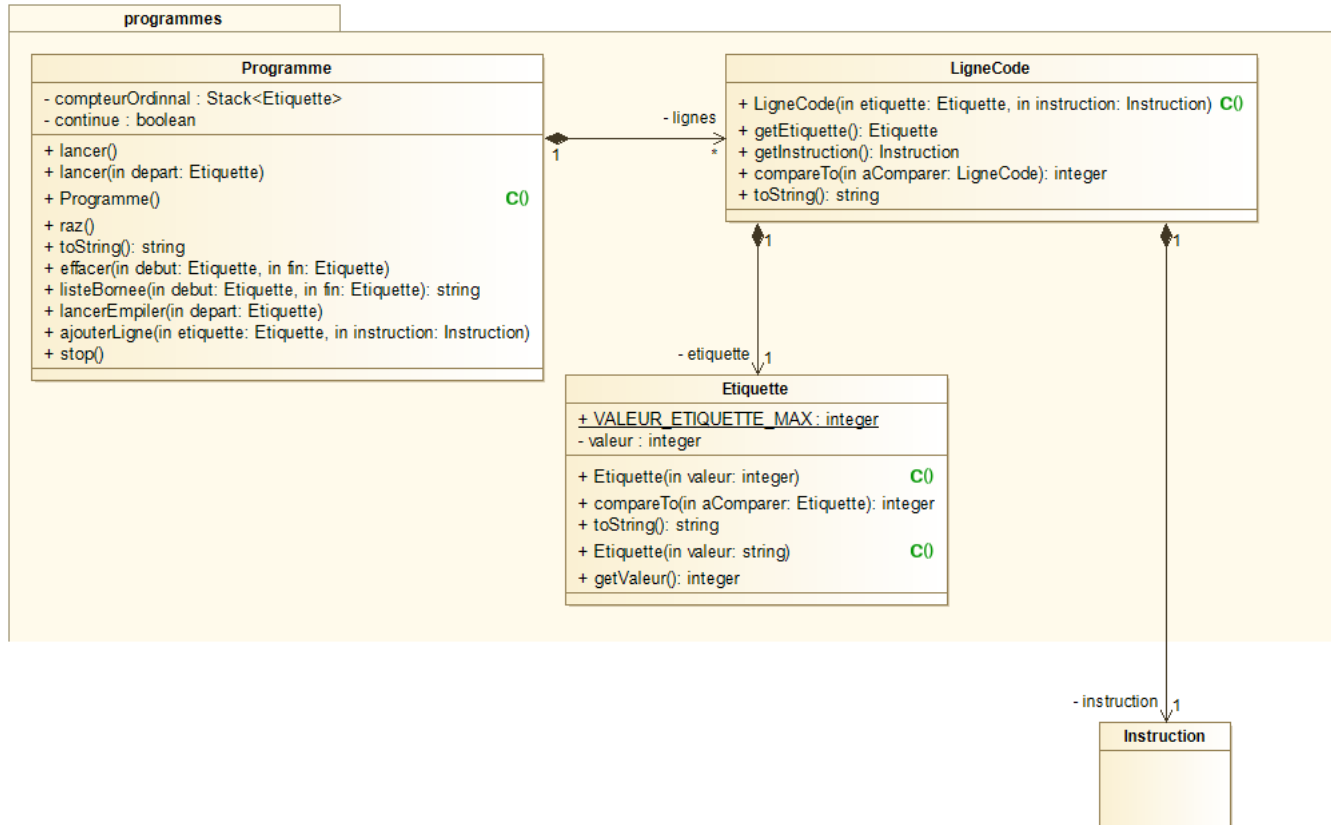
2.2 Paquetage `interpreteurlir.donnees(.litteraux)`

Les paquets `donnees` et `litteraux` n'ont que très peu changé en conception mais les classes liées aux entiers ont été codées pendant cette itération.

2.3 Paquetage `interpreteurlir.expressions`

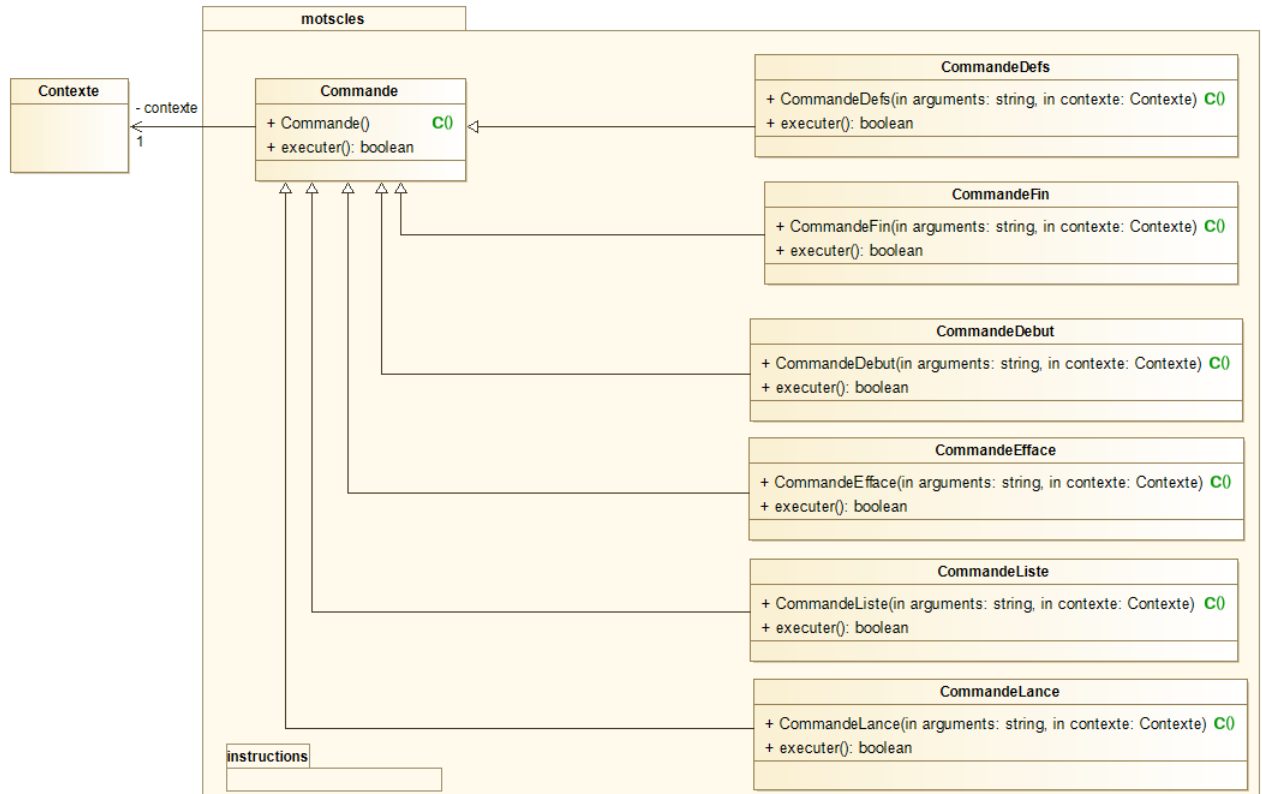
Comme pour les données, pas de changement de conception mais programmation de `ExpressionEntier`.

2.4 Paquetage interpreteurlir.programmes



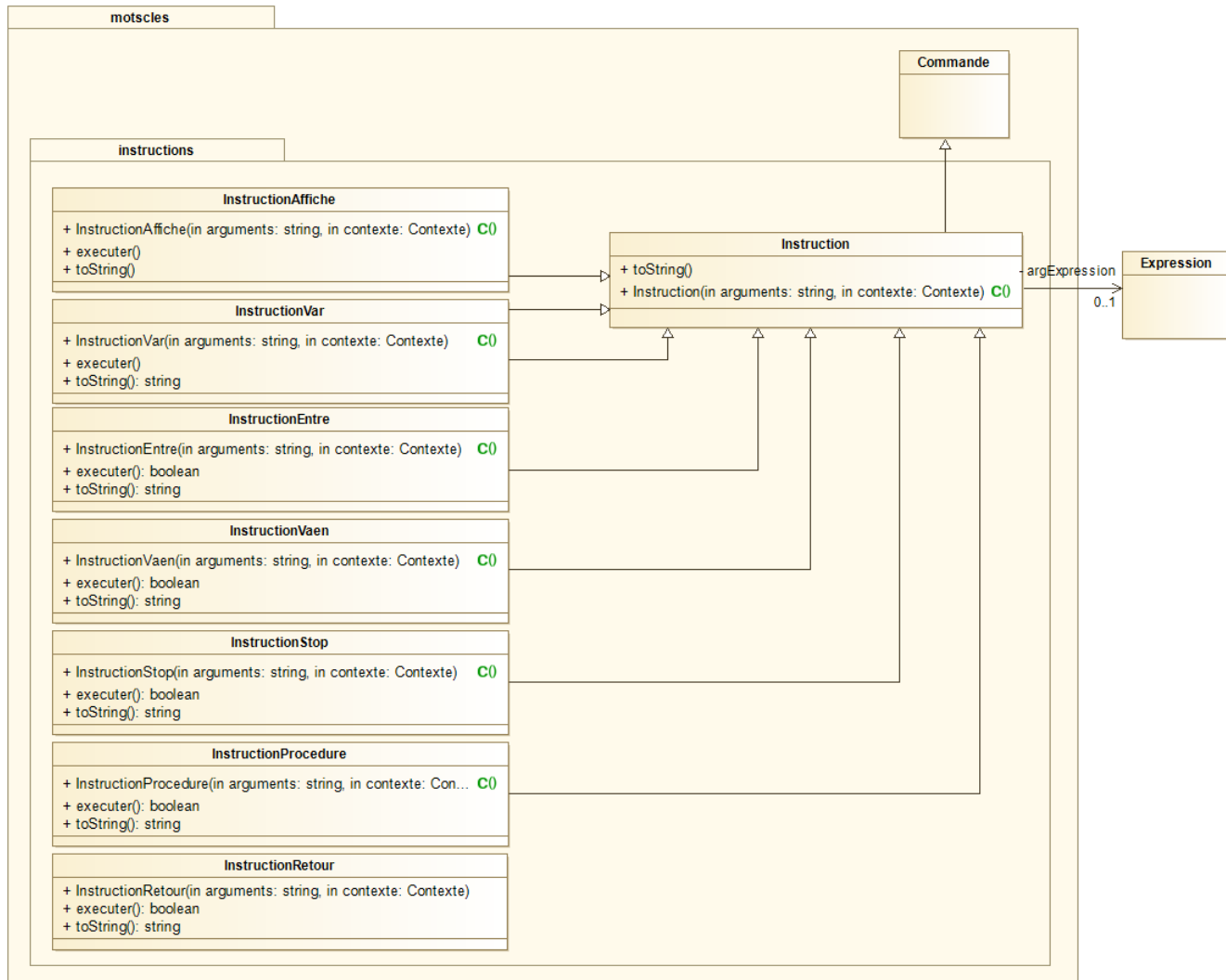
Premièrement la classe étiquette permet d'ordonner les lignes de codes. Le Programme contient des méthodes pour tous les comportement qu'il doit réaliser ce qui permet de les intégrer en interne ce qui rend leur usage plus simple pour les commandes et instructions. Seul la méthode `vaen` est absente de la conception car nous nous sommes rendu compte qu'elle était nécessaire pendant la programmation. Autre changement, le programme doit enregistrer les lignes de codes. La conception montre une classe **LigneCode** prévue à cet effet cependant sur le conseil de notre tuteur nous avons utilisé une `TreeMap<Etiquette, Instruction>` ce qui a rendu **LigneCode** obsolète. La classe avait été programmée et testée mais nous l'avons supprimée car `TreeMap` était une meilleure solution.

2.5 Paquetage interpreteurlir.motscles



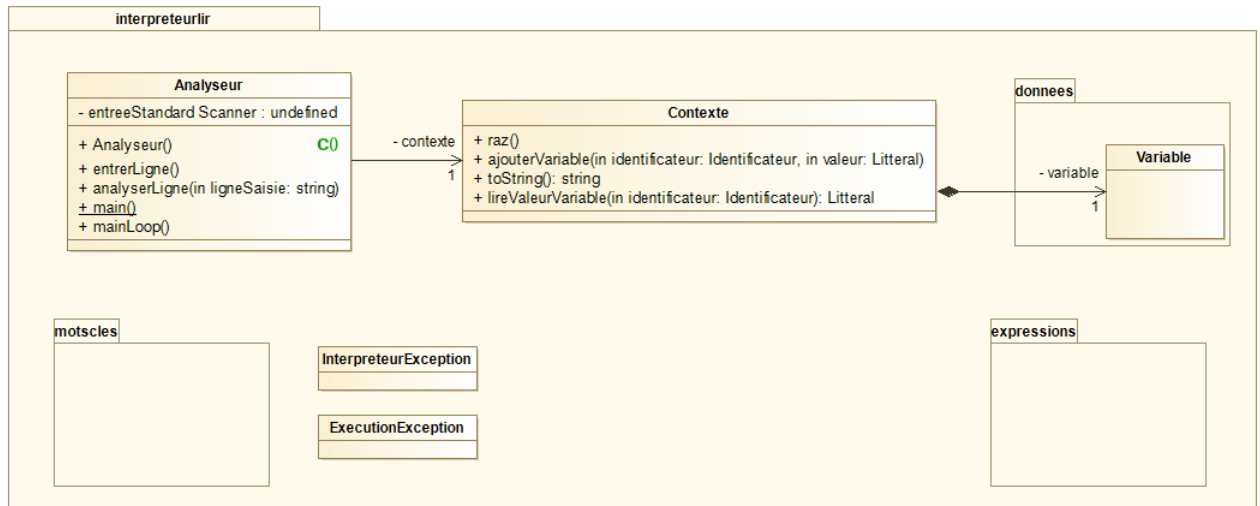
Les commandes à ajouter à cette itération ont été ajoutée à la conception en suivant le même principe de la dualité construction/exécution. Seul changement notable (non montré dans le diagramme car décidé pendant la programmation), l'ajout du programme nécessite que les commandes connaissent celui-ci. Après une longue réflexion nous avons choisis de le déclaré comme attribut `protected` dans la classe `Commande` et de le référencer au lancement de l'interpréteur sans savoir si c'était un bon choix ou non.

2.6 Paquetage interpreteurlir.motscles.instructions



Aucun changement notable, seulement ajout des nouvelles instructions.

2.7 Paquetage interpreteurlir



Ajout de l'exception `ExecutionException` lancée pour une erreur à l'exécution comme une division par 0 (contrairement à l'`InterpreterException` qui est lancée à la construction). Elle également affichée par l'`Analyseur`.

Chapitre 3

Itération 3

L'itération 3 a ajoutée les expressions booléennes avec l'instruction si vaen. Et les commandes permettent d'enregistrer et charger un programme LIR dans l'interpréteur (commande charge et sauve).

3.1 Diagrammes d'objets

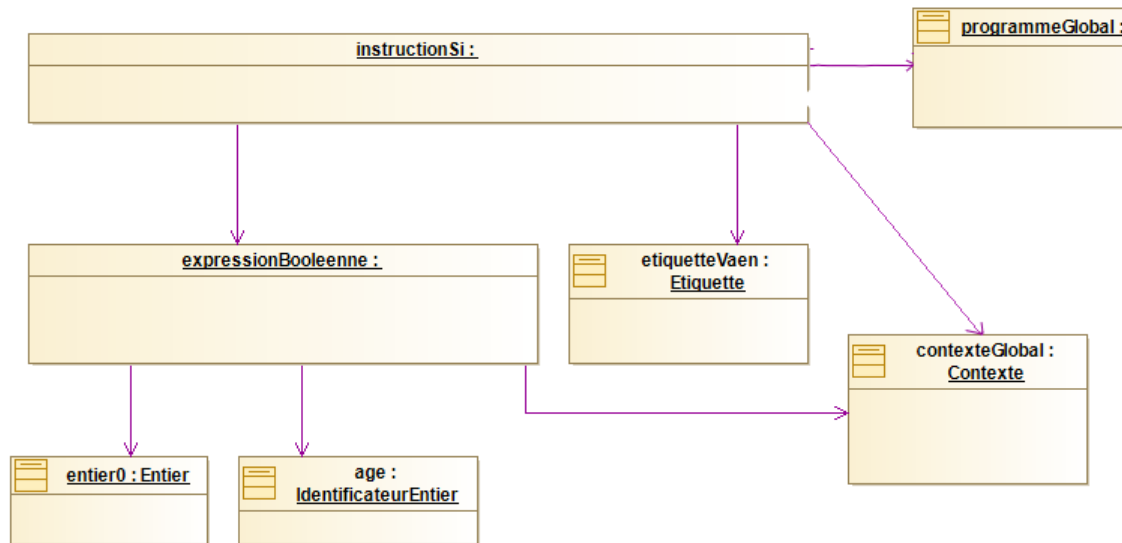


sauve monProgramme.lir



charge monProgrammeExemple.lir

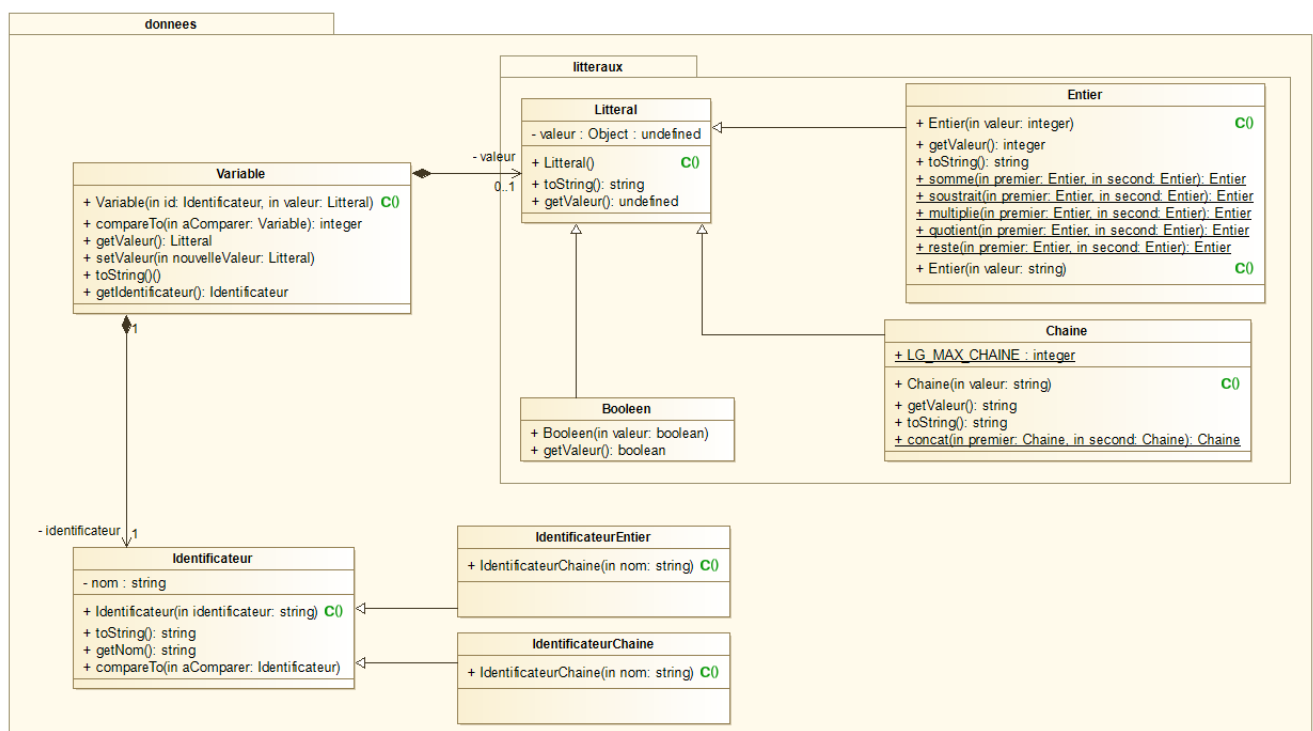
Les commandes sauve et charge sont liées au programme pour pouvoir charger ou récupérer des lignes de codes. Ces commandes connaissent une chaînes de texte correspondant au chemin du fichier.



si age < 0 vaen 100

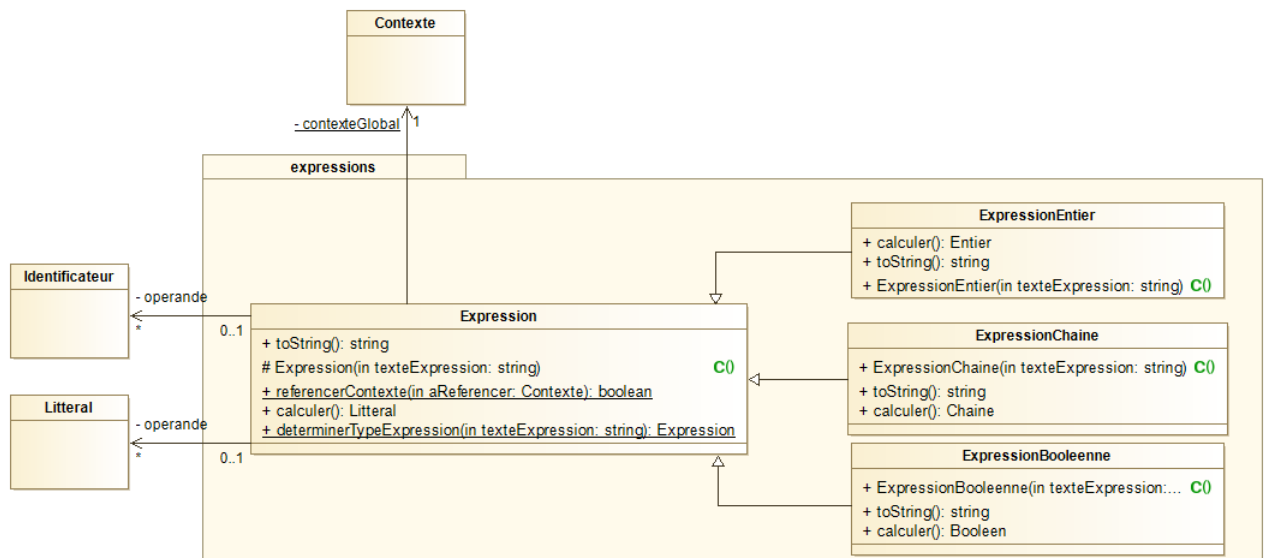
L'instruction si a besoin pour fonctionner d'une ExpressionBooleenne et de connaître le contexte pour chercher les valeurs des variables à comparer. Elle doit connaître l'étiquette où aller si la condition est vraie et donc du programme pour appeler la méthode du programme vaen.

3.2 Paquetage interpreteurlr.donnees(.litteraux)



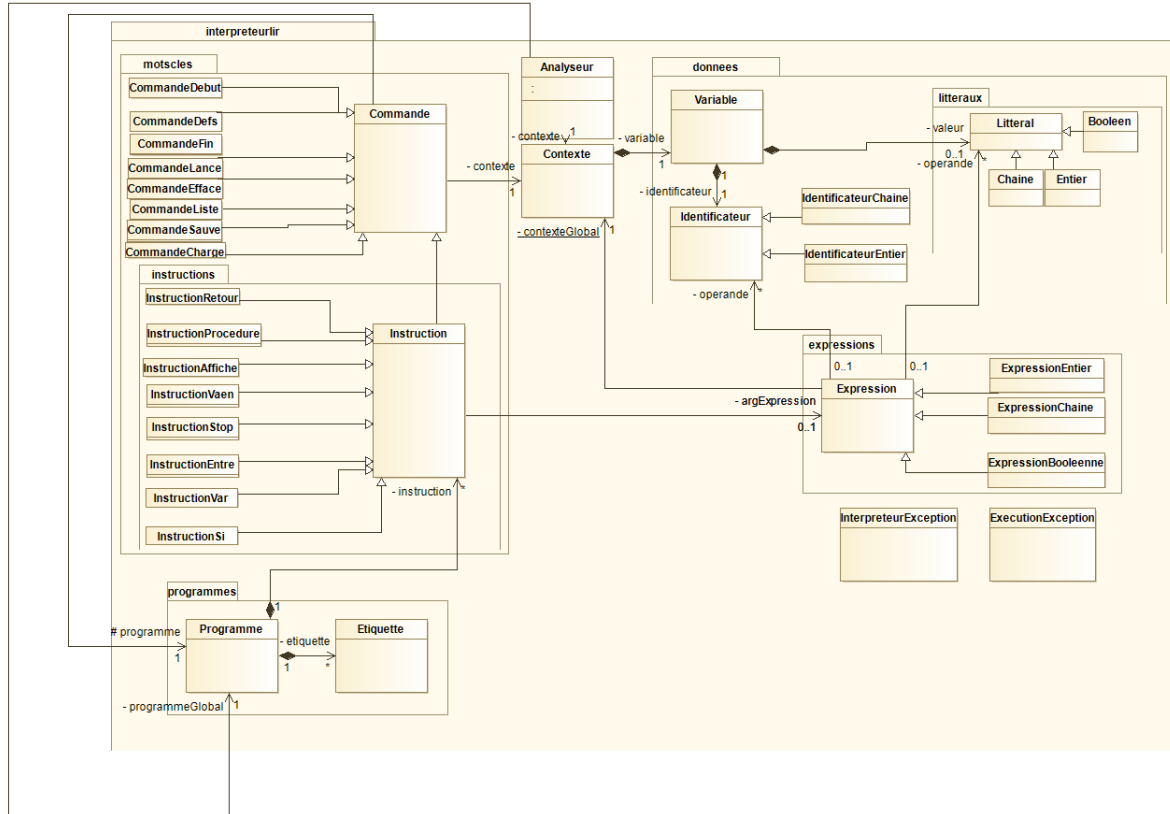
Le type booléen hérite de Litteral pour garder la logique de Litteral pouvant référencer chaque type de valeur du programme.

3.3 Paquetage interpreteurlir.expressions



L'expression booléenne ne s'obtient pas avec la méthode `déterminerExpression` car celle-ci est utilisée que par si vaen qui utilise que ce type d'expression. Le constructeur d'`ExpressionBooléenne` est donc utilisé directement.

3.4 Diagramme de classes général



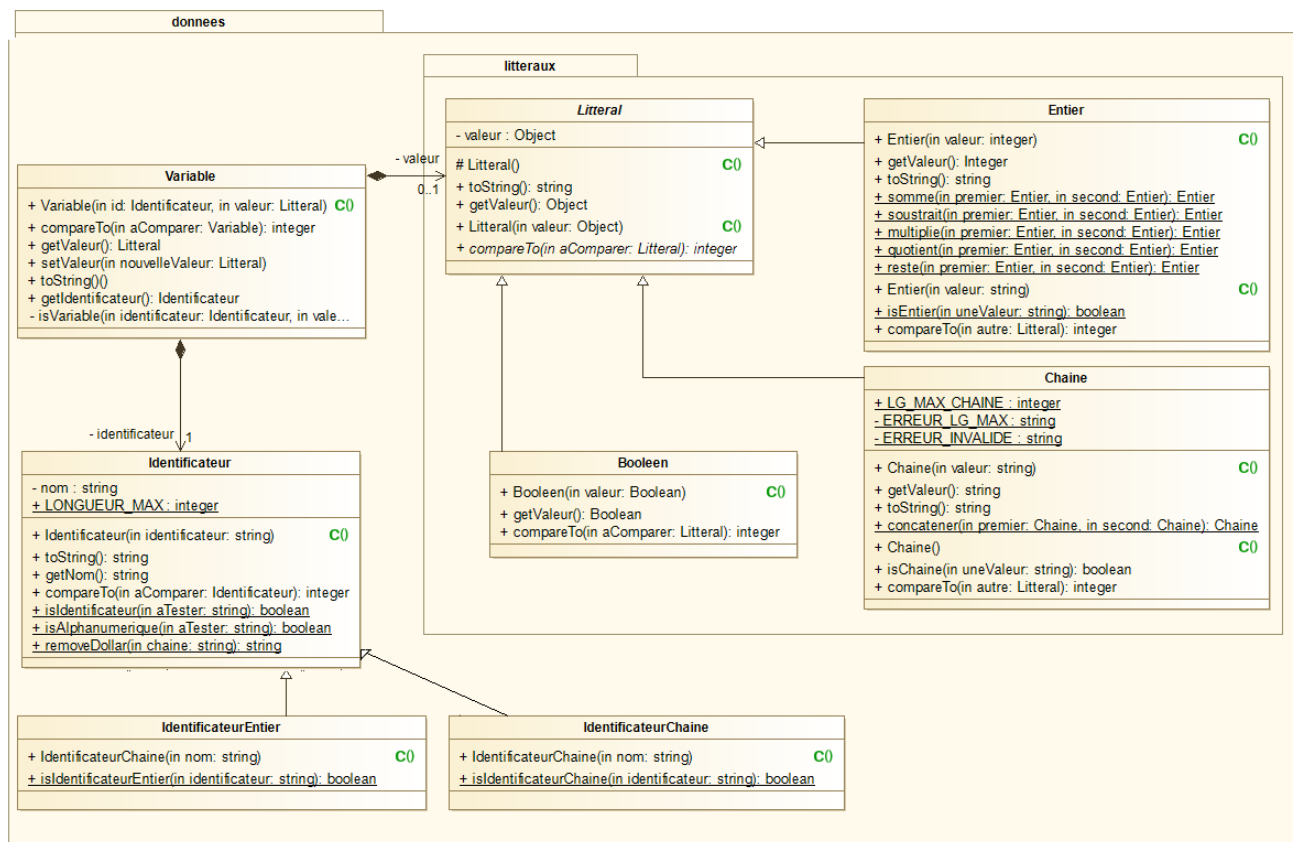
Les commandes sauve et charge ont été ajoutés à la conception mais sont similaires aux autres commandes. Pareil pour l'instruction si vaen. Ce diagramme général permet de voir l'ensemble de la conception pour ce qui est des associations et généralisation des classes.

Chapitre 4

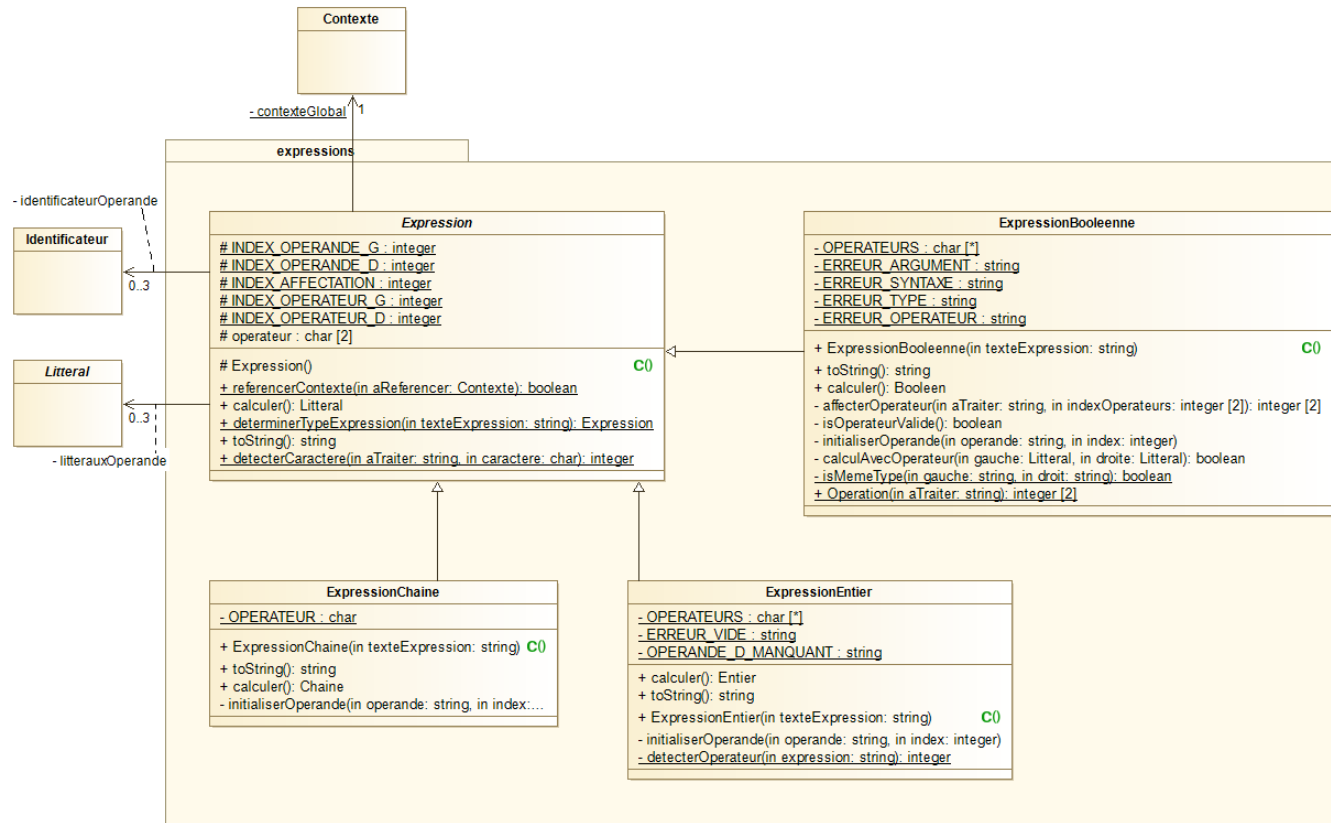
Projet final

Ce chapitre contient les diagrammes de classes représentant le logiciel codé. Les diagrammes de l'itération trois ont été complétés à partir du code pour ajouter les détails d'intégration dans les diagrammes (méthodes privées par exemple).

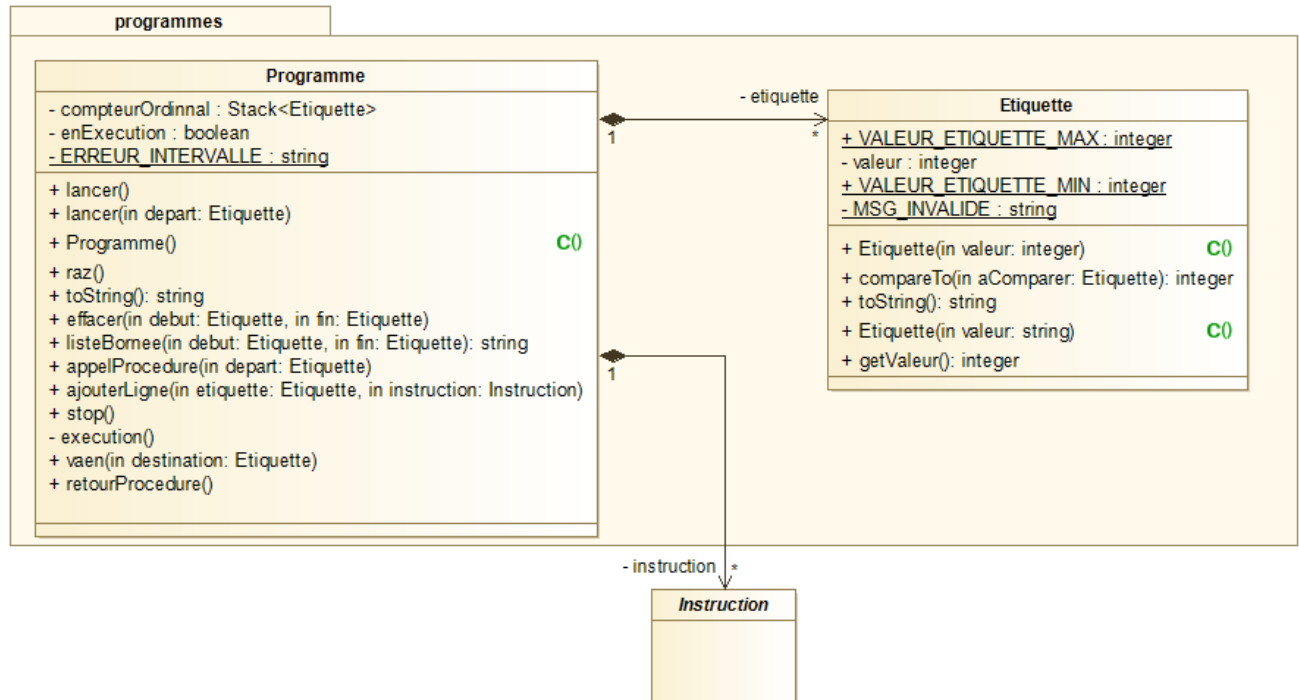
4.1 Paquetage `interpreteurlr.donnees(.litteraux)`



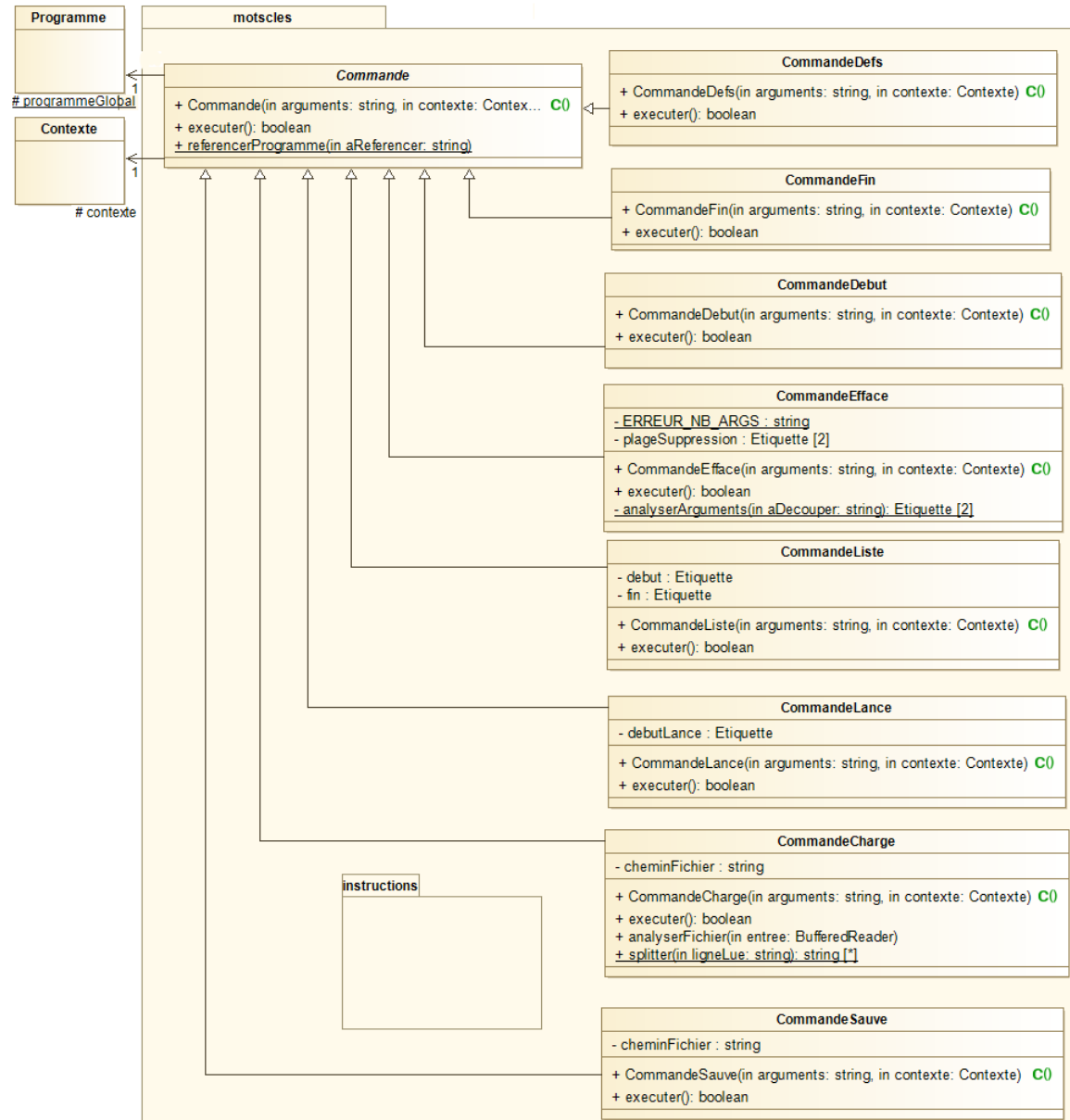
4.2 Paquetage interpreteurlir.expressions



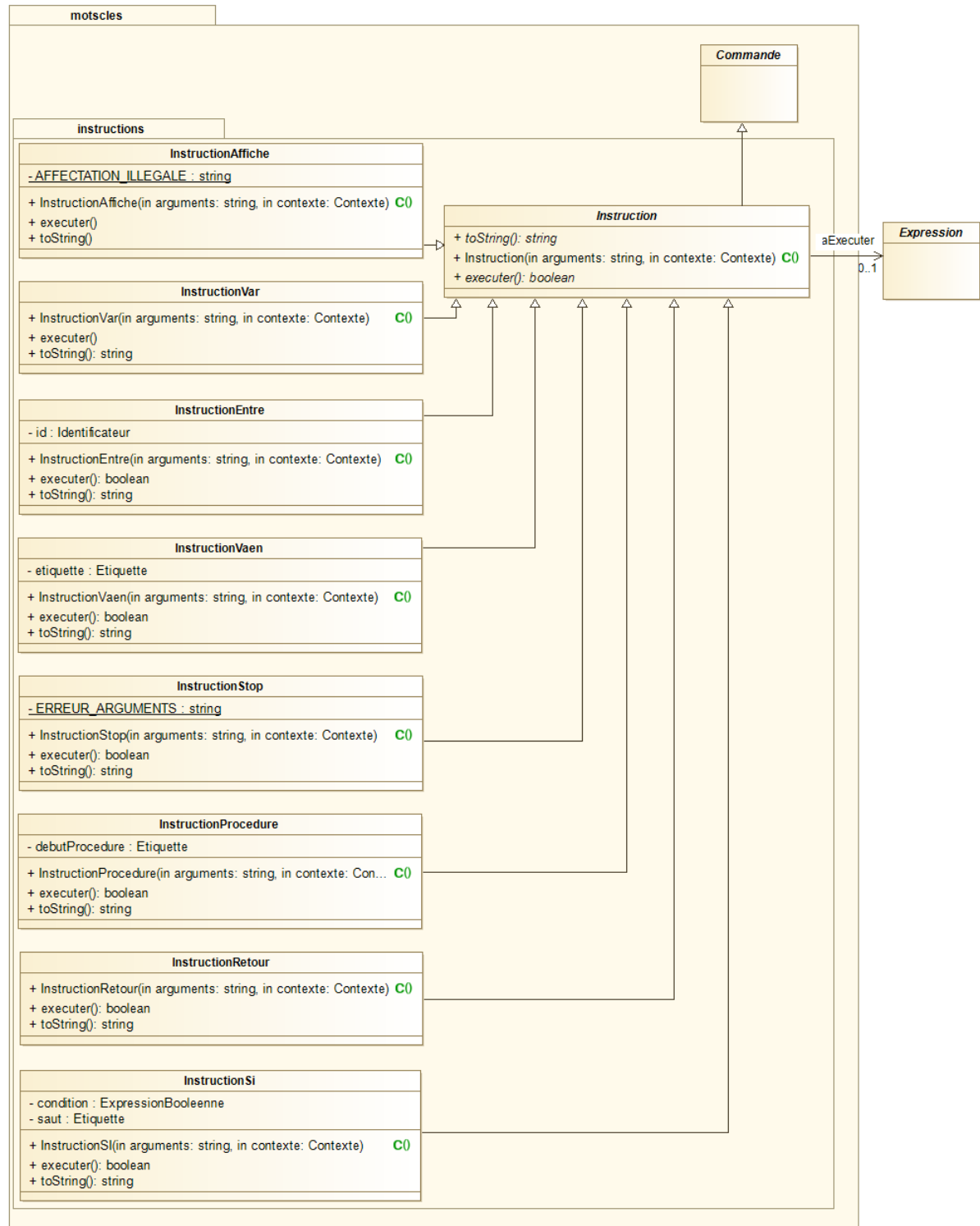
4.3 Paquetage interpreteurlir.programmes



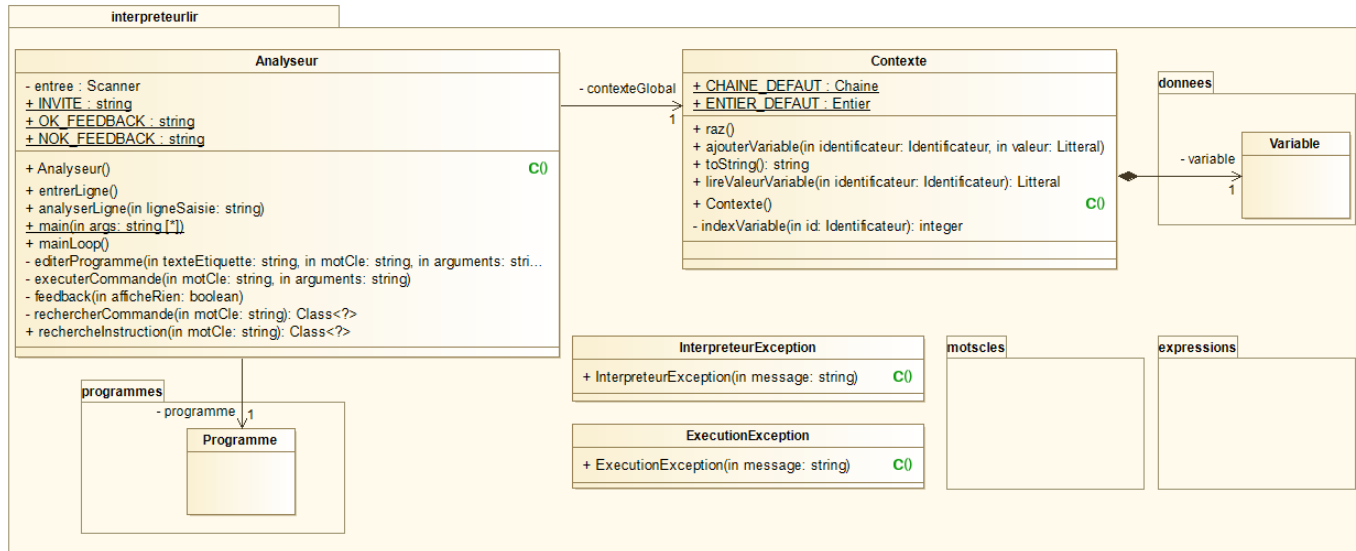
4.4 Paquetage interpreteurlir.motscles



4.5 Paquetage interpreteurlir.motscles.instructions



4.6 Paquetage interpreteurlir



Quatrième partie

Codage (annexe)

Cinquième partie

Tests

Démarche globale

Afin de développer l'Interpréteur LIR selon un modèle de cycle de vie itératif, nous avons privilégié la méthode du TDD, Test Driven Development ou développement dirigé par les tests en français.

Ainsi, la majorité des composants de ce logiciel ont été développés selon cette méthode à l'aide des outils de tests écrits lors des TD de Programmation Orientée Objet du semestre 2. Par conséquent, nous n'avons pas utilisé le framework de test JUnit.

Chapitre 5

Tests du paquetage interpreteurlir.donnees.litteraux

5.1 Litteral

Lors des itérations 1, la classe Litteral a été développée comme une classe non-abstrait, en effet nous n'avions pas encore abordé cette notion en cours. Ainsi cette classe a été développée en TDD et nous avons par conséquent effectué les tests unitaires de cette classe.

Cependant à la fin de l'itération 3, dans une optique d'amélioration des codes sources, nous avons passé cette classe en abstrait, ainsi les tests unitaires menés n'avaient plus lieu d'être et ont donc été tout de même conservés en commentaire.

5.2 Chaîne

Les jeux de tests de la classe Chaîne prennent en compte les cas de chaînes vides, la taille maximale des chaînes, leur syntaxe (avec le contenu de la chaîne entre "). Aussi l'opération de concaténation a été testée. Tous les tests menés ont été concluants.

5.3 Entier

La classe Entier est très proche de la classe Integer existant déjà dans le JDK, ainsi son développement a été rapide. À l'instar des tests menés pour la classe Chaîne, tous les tests de la classe Entier notamment des opérations arithmétiques ont été concluants.

Notons, que pour les opérations arithmétiques telles que la division et le reste de la division, le cas particulier de la division par zéro a été testé à part.

5.4 Booleen

La classe Booleen n'a posé aucun problème particulier.

Chapitre 6

Tests du paquetage interpreteurlir.donnees

6.1 Identificateur

La classe Identificateur a été développée en TDD lors de l'itération 1 cependant elle a été passée en abstract lors de l'itération 3, comme pour la classe Litteral, les tests unitaires menés lors de l'itération 1 n'avaient plus lieu d'être et ont été conservés en commentaire. La méthode d'instance compareTo() testée avant le passage de la classe en abstrait et vaut pour les identificateurs d'entier et de chaîne.

6.2 IdentificateurChaine et IdentificateurEntier

Lors des tests unitaires des deux classes, la syntaxe des identificateurs a été testées. Les tests ont été concluants.

6.3 Variable

La classe Variable a été développée lors de l'itération 1 et a donc été testée avec les identificateurs d'entier et de chaîne et seulement avec des valeurs de type Chaine, en effet, la classe Entier ne faisait pas partie de la conception de l'itération 1.

Chapitre 7

Tests du paquetage interpreteurlir.expressions

7.1 Expression

Cette classe Expression a été passée en abstract lors de l'itération 3 cependant les tests des méthodes statiques restent pertinents et ont donc été conservés.

7.2 ExpressionChaine

Lors du développement de la classe ExpressionChaine, l'ambiguïté des symboles des opérateurs ("+" et "=") a posé problème. En effet, il fallait déterminer l'emplacement de l'opérateur en ignorant ces symboles s'ils sont contenu dans les constantes littérales. Au fil du développement, les tests ont été concluants.

7.3 ExpressionEntier

Il n'y a rien à signaler sur le développement de ExpressionEntier.

7.4 ExpressionBoolenne

Le développement a été scindé en deux. Un premier groupe a commencer à écrire les interfaces et les tests cependant il a rencontré des difficultés à coder le constructeur, ont été repris par la suite par un autre groupe. Cependant la méthode calculer() n'a pas posé de problème lors du développement.

Chapitre 8

Tests du paquetage interpreteurlir

8.1 InterpreterException et ExecuteurException

Ces deux exceptions sont héritées de RuntimeException et n'ajoute aucun comportement supplémentaire. Par conséquent, leurs tests n'étaient pas déterminants pour la suite du développement de l'interpréteur.

8.2 Contexte

Le développement de la classe Contexte s'est déroulé sans difficultés. Aussi les tests ont été concluants.

8.3 Analyseur

L'Analyseur n'a pas de tests unitaires car tous les tests ont été menés lors de l'intégration. Tests d'intégration effectués en deux parties, la première lors de l'itération 1 avec le test de la prise en charge des commandes et d'instructions exécutées directement, la seconde lors de l'itération 2 pour l'édition de programme.

Chapitre 9

Tests du paquetage interpreteurlir.programmes

9.1 Etiquette

Aucune difficulté n'a été rencontrée lors du développement de la classe Etiquette, aussi les tests ont été concluants.

9.2 Programme

Lors de l'implémentation de la classe Programme, certaines méthodes ne pouvaient être testées directement car il manquait encore des instructions permettant de le faire. Lors de l'implémentation de ces instructions, les tests de celles-ci ont permis de tester également les méthodes de programmes. Ainsi certains tests de Programme n'ont pu être menés que lors de l'intégration avec les instructions ou commandes.

9.3 Les programmes de tests

Lors de l'itération 2, alors que les commandes sauve et charge n'étaient pas encore implémentées, un composant permettait de charger un programme complet au lancement de l'interpréteur pour les démonstrations.

Lors de l'itération 3, après l'implémentation de la commande charge, quatre fichiers contenant un programme écrit en langage LIR ont été écrits pour les démonstrations et tests finaux de l'interpréteur. Il s'agit de l'exemple de programme proposé dans le cahier des charges et des programmes EtatCivil, Median3Entiers et Factorielle.

Chapitre 10

Tests du paquetage interpreteurlir.motscles

10.1 Commande

Le développement de la classe `Commande` a été mené lors de l'itération 1 en TDD, en effet la classe a été passée en classe abstraite lors de l'itération 3. Les tests mené ont été conservés en commentaire.

10.2 EssaiCommande

Lors de l'itération 1, avant l'implémentation de classe `Analyseur`, pour l'intégration des premières commandes `debut`, `defs` et `fin` nous avons utilisé une classe `EssaiCommande`. Une fois la classe `Analyseur` implémentée, l'intégration des autres commandes a été par la suite testée avec.

10.3 CommandeCharge

Le développement de charge n'a pas été simple, en effet, son implémentation a soulevé un problème de conception. La commande charge doit faire appel à l'`Analyseur` cependant celui-ci n'a pas été conçu de façon assez générale pour prendre en compte ce cas de figure. Au vu des délais à tenir, nous avons choisi la solution qui nous paraissait la plus viable. Celle-ci impliquait de recréer des parties de la classe `Analyseur` au sein même de la classe `CommandeCharge`.

Les tests de charge ont encore une particularité, en effet, ils dépendent de la machine sur laquelle les tests sont effectués, il faut donc adapter les tests à la machine utilisée. Cela consiste à avoir sur la machine utilisée des chemins d'accès et des fichiers coïncidant avec ceux des tests.

10.4 CommandeDebut

La commande debut a évolué entre l'itération 1 et 2 avec l'ajout de la remise à zéro du programme en plus de celle du contexte.

10.5 CommandeDefs et CommandeFin

Le développement de ces commandes n'a pas posé de problème, aussi leurs tests ont été concluants.

10.6 CommandeEfface, CommandeLance et CommandeListe

Le développement de ces trois commandes n'a pas posé de problème particulier. Leurs tests ont permis de tester par la même occasion les méthodes de la classe Programme.

10.7 CommandeSauve

Le développement de ces commandes n'a pas posé de problème. À l'instar de charge, la commande sauve nécessite que les chemins pour les tests soient accessibles sur la machine utilisée.

Chapitre 11

Tests du paquetage `interpreteurlir.motscles.instructions`

11.1 Instruction

Le développement de la classe `Instruction` a été mené lors de l'itération 1 en TDD, en effet la classe a été passée en classe abstraite lors de l'itération 3. Les tests mené ont été conservés en commentaire.

11.2 `InstructionAffiche`, `InstructionEntre` et `InstructionSi(Vaen)`

Le développement de ces instructions n'a pas posé de problème, aussi leurs tests ont été concluants.

11.3 `InstructionProcedure`, `InstructionRetour`, `InstructionStop` et `InstructionVaen`

Le développement de ces quatre instructions n'a pas posé de problème particulier. Leurs tests ont permis de tester par la même occasion les méthodes de la classe `Programme`.

11.4 `InstructionVar`

Le développement de l'instruction `var` a posé un léger problème avec la nécessité que l'expression suivant le mot clé `var` contienne obligatoirement une affectation. Hormis ce souci, le développement de cette instruction n'a pas posé outre problème.

Sixième partie

Conclusion

Chapitre 12

Conception et implémentation

12.1 Le livrable

À l'issue de ce projet, nous avons pu implémenter toutes les fonctionnalités de l'interpréteur LIR telles qu'elles étaient exposées dans le cahier des charges. Notre version de l'interpréteur fonctionne comme attendu par la MOA, bien que la gestion des erreurs et les messages affichés à l'écran auraient gagné à être plus précis et que certaines parties du code mériteraient une optimisation.

12.2 Conception

Ce besoin d'optimisation découle de difficultés rencontrées lors de la conception des classes. Ces difficultés s'expliquent notamment par notre manque d'expérience. Si nous devions refaire ce projet, il est clair que certains des choix que nous avons faits ne seraient pas réitérés.

Commencer directement par générer des diagrammes d'objets en lieu de diagrammes de classes nous aurait certainement permis de gagner quelques heures de travail au moment de la conception initiale. Nous avons cependant fait mieux pour intégrer des notions apprises au cours du projet, comme par exemple le passage en abstraction de certaines superclasses.

Chapitre 13

Organisation du groupe

13.1 Travail en binôme

Lors de chaque itération, nous avons autant que possible privilégié le travail en binôme, en fonction des disponibilités de chacun. Cette modalité nous a permis de nous assurer que tout le monde participait activement au développement et se sentait intégré et valorisé au sein du groupe.

Nous avons aussi fait en sorte de mettre en place une rotation des binômes afin que chaque membre du groupe puisse travailler avec tout le monde. Nous avons ainsi pu nous confronter à d'autres de travailler, partager nos savoirs et nos expériences personnels et assurer une forte cohésion au sein du groupe.

13.2 Répartition de la charge de travail

Malheureusement, le travail en binôme n'est forcément garant d'une répartition efficace de la charge de travail. Cela pose en effet des contraintes cumulatives ; lorsque un membre du groupe a terminé sa tâche, si la suivante nécessite un travail à deux, ce membre devait parfois attendre que son binôme se libère. Il est arrivé qu'un des deux membres d'un binôme prenne du retard sur sa tâche. Cela a occasionnellement posé un frein sur cette modalité de travail.

Devant travailler le weekend, nous nous sommes également heurtés aux aléas des disponibilités personnelles de chacun. Nous avons donc dû composer avec des contraintes familiales, universitaires (devoirs à rendre, révisions,...) ou personnelles. Ces difficultés seraient mitigées dans un contexte professionnel avec des horaires de travail définis dans un contrat.

Nous regrettons aussi de ne pas avoir mis en place un roulement dans les responsabilités (chef de projet, secrétaire, gestionnaire de configuration). Nous avons préféré nous concentrer sur le code.

13.3 Communication

Tout au long du projet, nous avons mis l'accent sur la communication, afin de toujours avoir un aperçu de l'avancée de notre travail. L'utilisation d'un serveur *Discord* dédié au projet a été un outil primordial. En effet, cet outil nous a permis de travailler en binôme en visioconférence, d'organiser des réunions MOE en distanciel.

L'utilisation de « salons » thématiques de conversation a aussi ouvert la possibilité de s'entraider lorsqu'une difficulté se présentait, faire circuler les informations, organiser les réunions, ou plus simplement discuter (moments de convivialité). Grâce à la synchronisation avec le dépôt Github, chaque membre recevait en temps réel les notifications sur l'évolution du projet.

Nous pensons que la communication a été un atout de taille dans la conduite de ce projet. En effet, elle nous a permis de surmonter au mieux les difficultés qui se sont présentées au cours de la conception et du développement de l'interpréteur.

Chapitre 14

Conclusion générale

Nous avons vécu ce projet comme une expérience enrichissante que nous considérons dans l'ensemble comme une réussite. Nous avons pu acquérir et consolider des compétences précieuses au travail d'équipe. Nous tâcherons au cours des prochains projets tutorés de réinvestir nos succès et apprendre de nos échecs.

Septième partie

Manuel utilisateur
(annexe)