



RODEZ

# CONCEPTION INTERPRÉTEUR DU LANGAGE LIR

---

PROJET PROPOSÉ PAR FRÉDÉRIQUE BARRIOS

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

# Sommaire

<b>III</b>	<b>Conception</b>	<b>2</b>
<b>1</b>	<b>Itération 1</b>	<b>3</b>
1.1	Paquetage interpreteurlir.donnees.litteraux . . . . .	3
1.2	Paquetage interpreteurlir.donnees . . . . .	4
1.3	Paquetage interpreteurlir.expressions . . . . .	4
1.4	Paquetage interpreteurlir.motscles . . . . .	5
1.5	Paquetage interpreteurlir . . . . .	6
1.6	Illustration avec des diagrammes d'objets . . . . .	7
<b>2</b>	<b>Itération 2</b>	<b>8</b>
2.1	Diagrammes d'objets . . . . .	8
2.2	Paquetage interpreteurlir.donnees(.litteraux) . . . . .	9
2.3	Paquetage interpreteurlir.expressions . . . . .	9
2.4	Paquetage interpreteurlir.programmes . . . . .	10
2.5	Paquetage interpreteurlir.motscles . . . . .	11
2.6	Paquetage interpreteurlir.motscles.instructions . . . . .	12
2.7	Paquetage interpreteurlir . . . . .	13
<b>3</b>	<b>Itération 3</b>	<b>14</b>
3.1	Diagrammes d'objets . . . . .	14
3.2	Paquetage interpreteurlir.donnees(.litteraux) . . . . .	15
3.3	Paquetage interpreteurlir.expressions . . . . .	16
3.4	Diagramme de classes général . . . . .	16
<b>4</b>	<b>Projet final</b>	<b>18</b>
4.1	Paquetage interpreteurlir.donnees(.litteraux) . . . . .	18
4.2	Paquetage interpreteurlir.expressions . . . . .	19
4.3	Paquetage interpreteurlir.programmes . . . . .	19
4.4	Paquetage interpreteurlir.motscles . . . . .	20
4.5	Paquetage interpreteurlir.motscles.instructions . . . . .	21
4.6	Paquetage interpreteurlir . . . . .	22

# **Troisième partie**

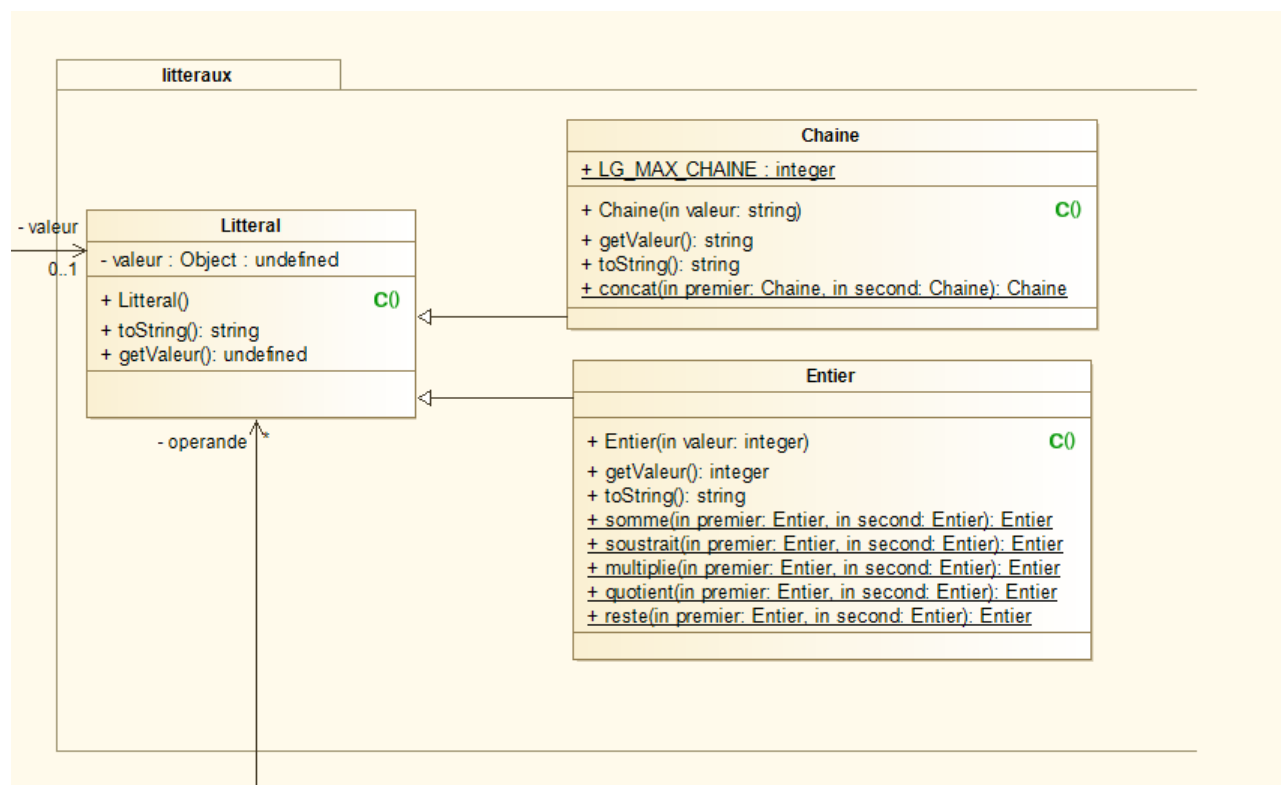
## **Conception**

# 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 chaines. Les commandes debut, defs, fin et l'instruction var ont donc été ajoutés afin d'obtenir ces fonctionnalités.

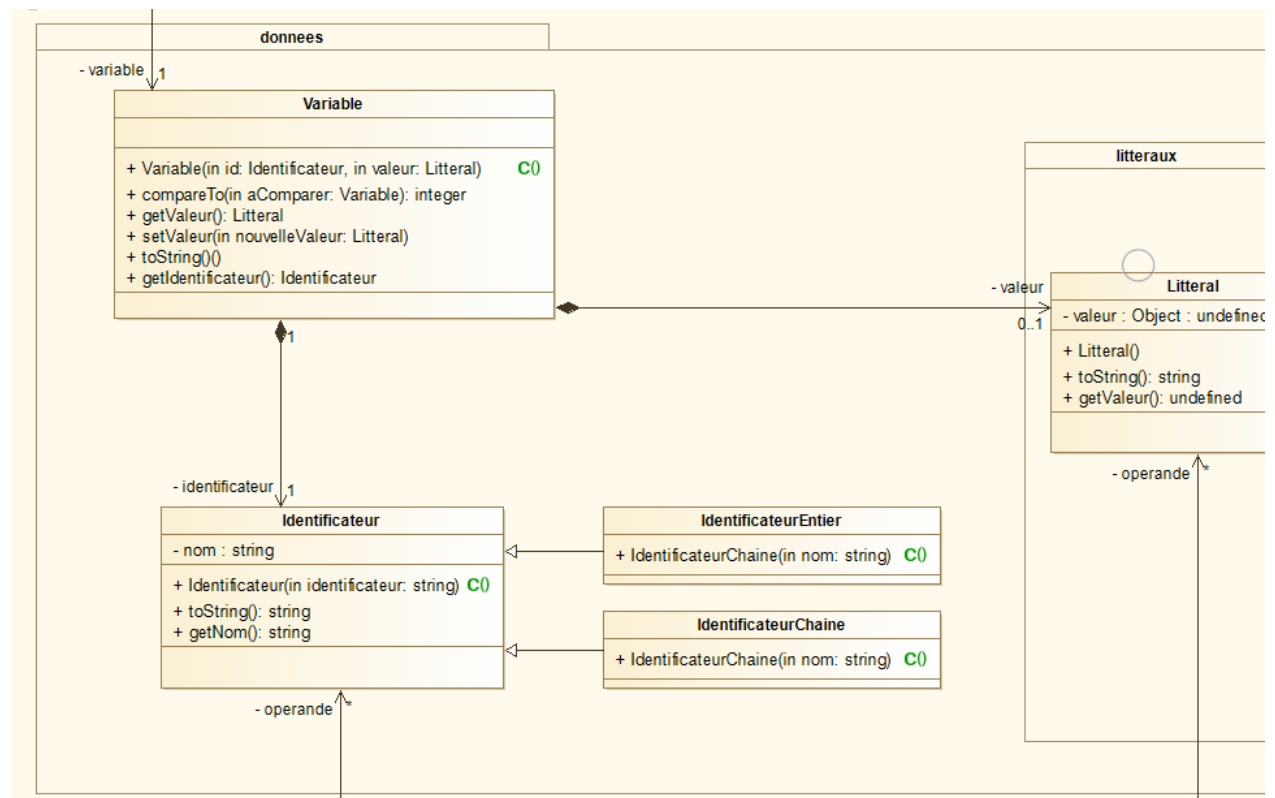
### 1.1 Paquetage interpreteurlir.donnees.litteraux



Le choix de conception des littéraux a été une classe parente Litteral 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

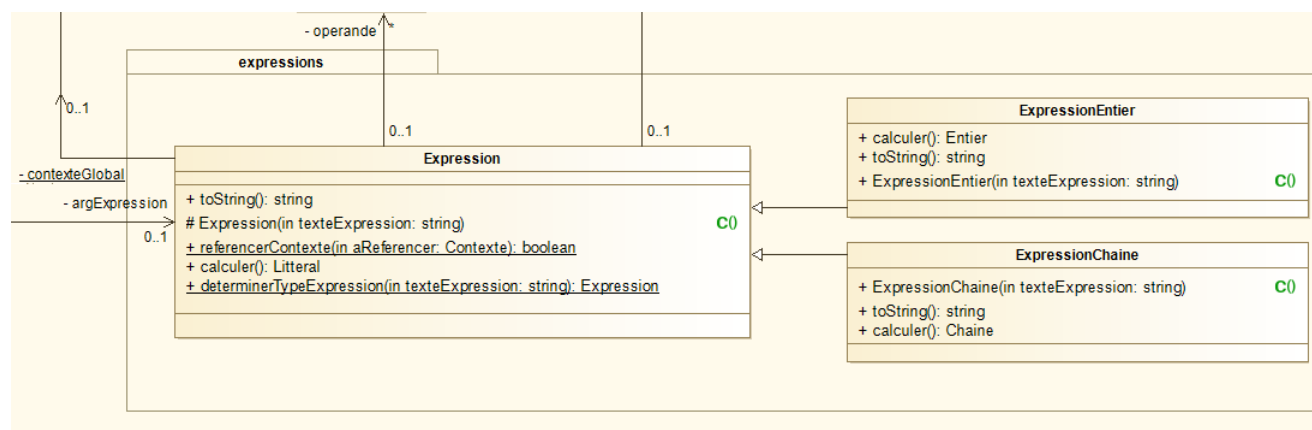
été codée à cette itération pour se concentrer sur les chaînes. Les littéraux sont immuables pour permettre leur passage sans problème.

## 1.2 Paquetage interpreteurlir.donnees



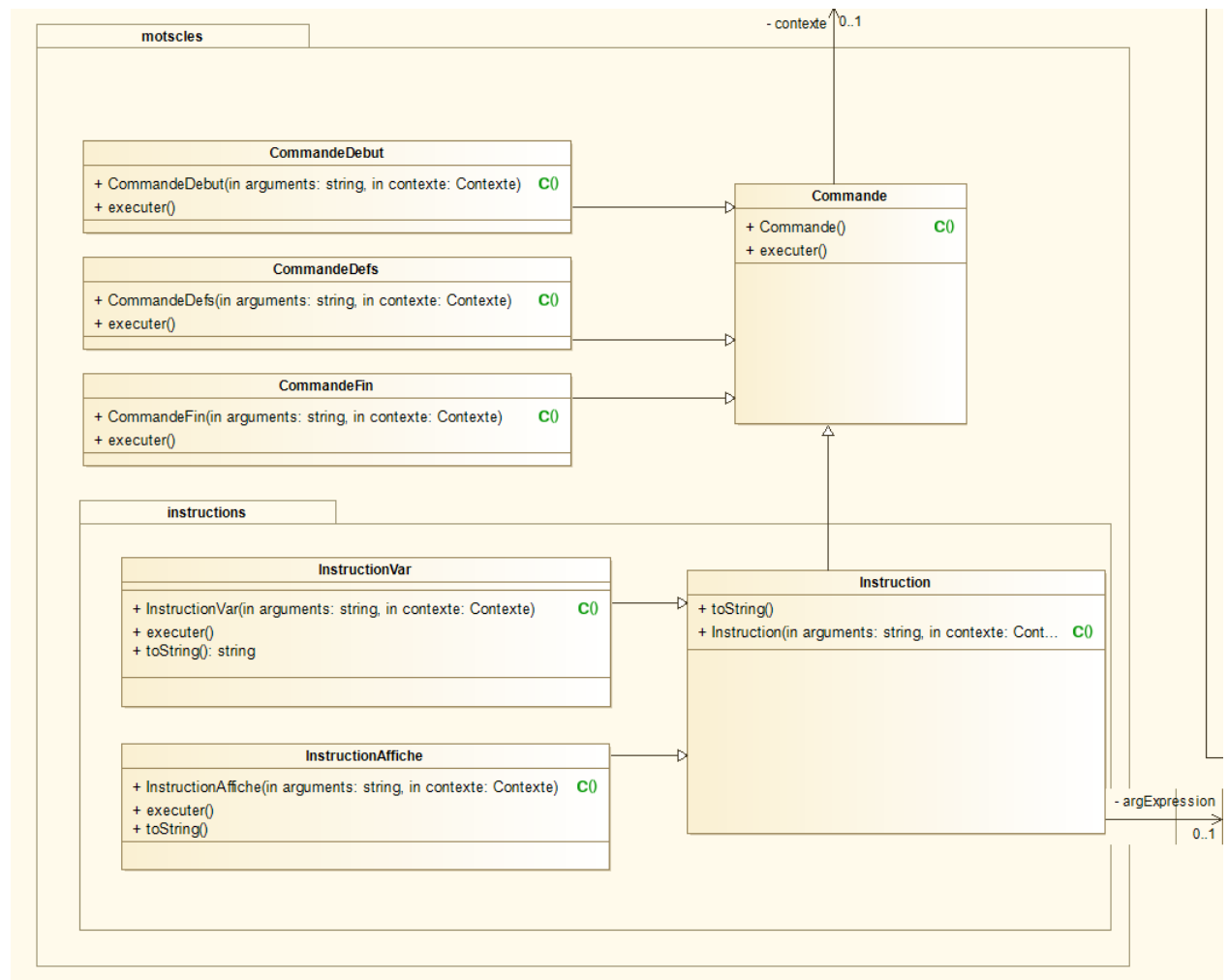
Pour les données une classe variable a été choisie composée d'un littéral et d'un identificateur. L'identificateur a comme classes dérivées les deux types affectables du projet soit les entiers et les chaînes.

## 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.

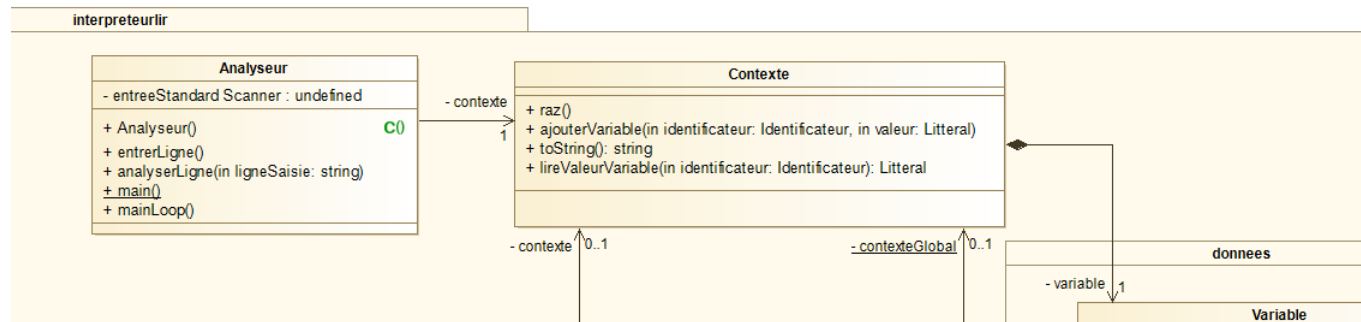
## 1.4 Paquetage interpreteurlir.motscles



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 un 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 nécessaires pour être exécuté (String arguments dans le constructeur). Les commandes et instructions fonctionnent en 2 temps, la construction qui valide les arguments

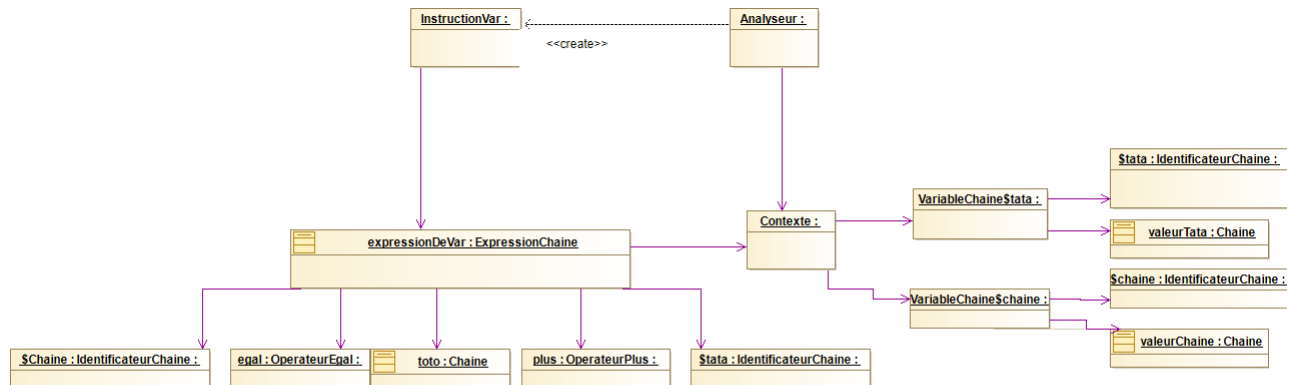
et créer les éléments nécessaires à l'exécution puis l'exécution qui est la réalisation du comportement de la commande.

## 1.5 Paquetage interpreteurlir

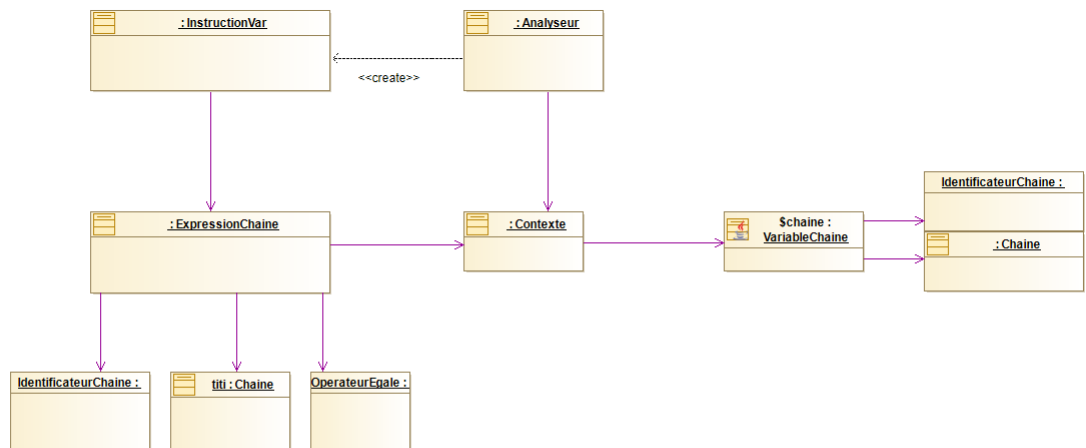


Le contexte regroupe l'entièreté des variables définies dans la session courante. Une variable n'est accessible que par l'intermédiaire du contexte grâce à l'identificateur qui sert de clé. L'Analyseur est la classe qui permet le fonctionnement de tout. Une `mainLoop` permet de demander en continue une 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 `InterpreteurException` 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

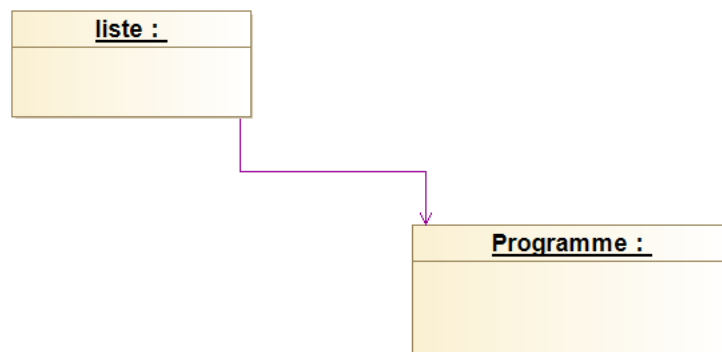
## Itération 2

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).

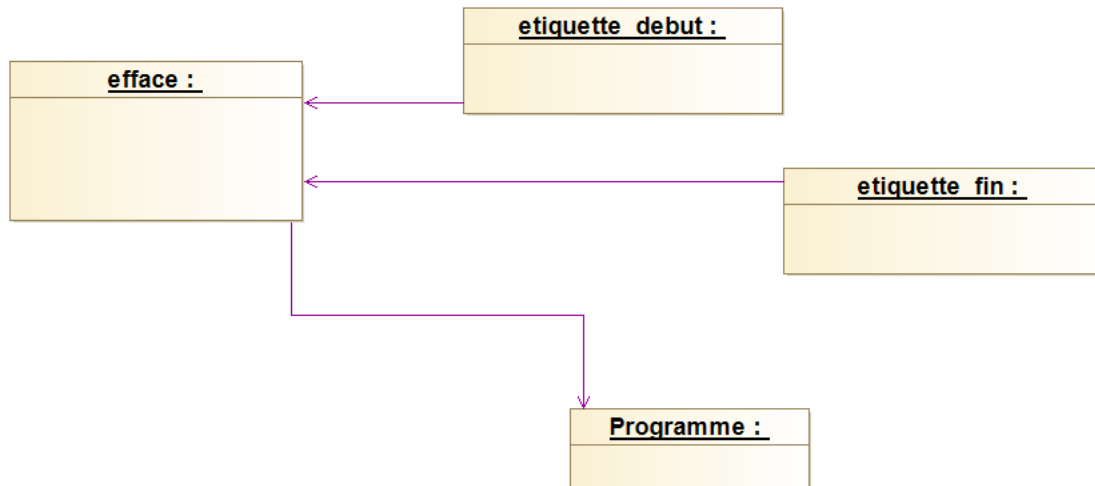
### 2.1 Diagrammes d'objets

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.

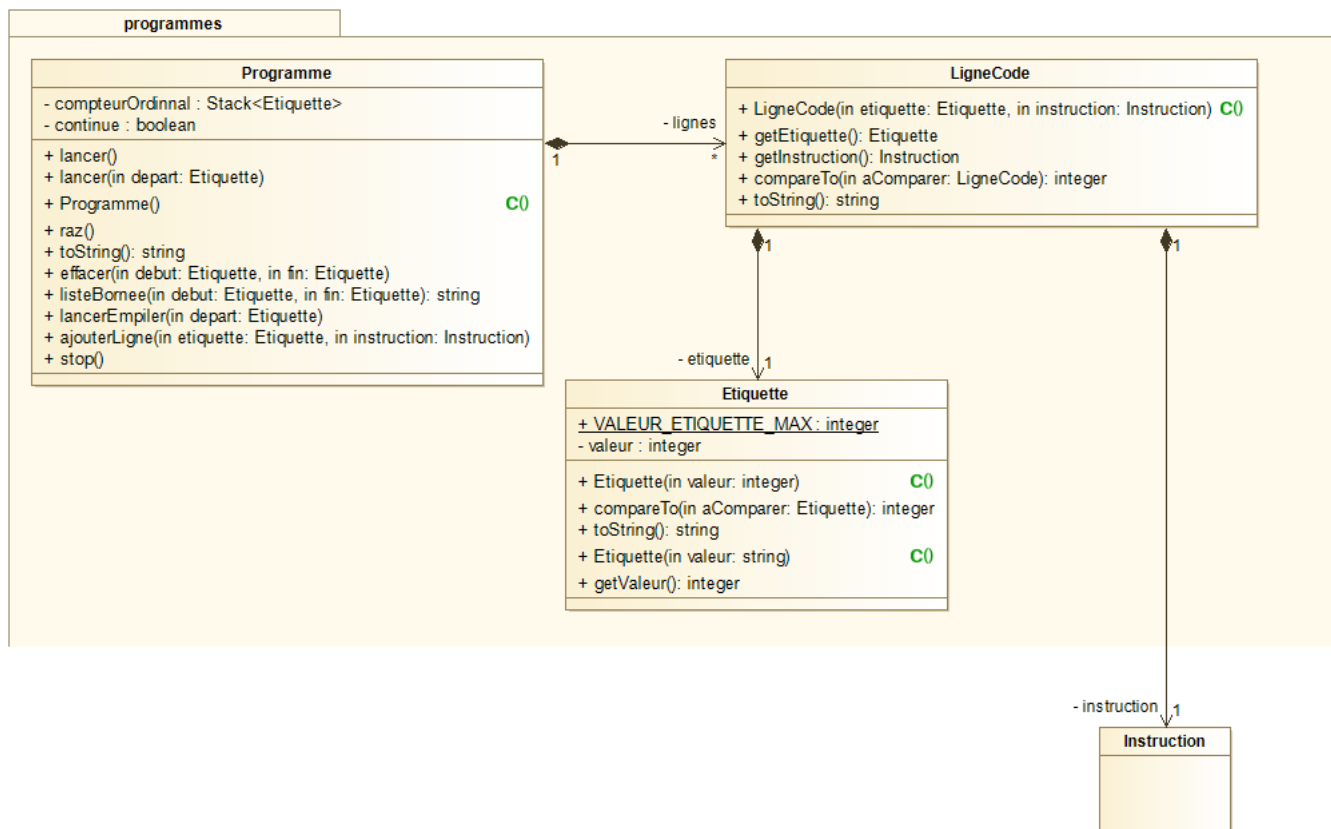
## 2.2 Paquetage `interpreteurlir.donnees(.litteraux)`

Les paquetages `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`

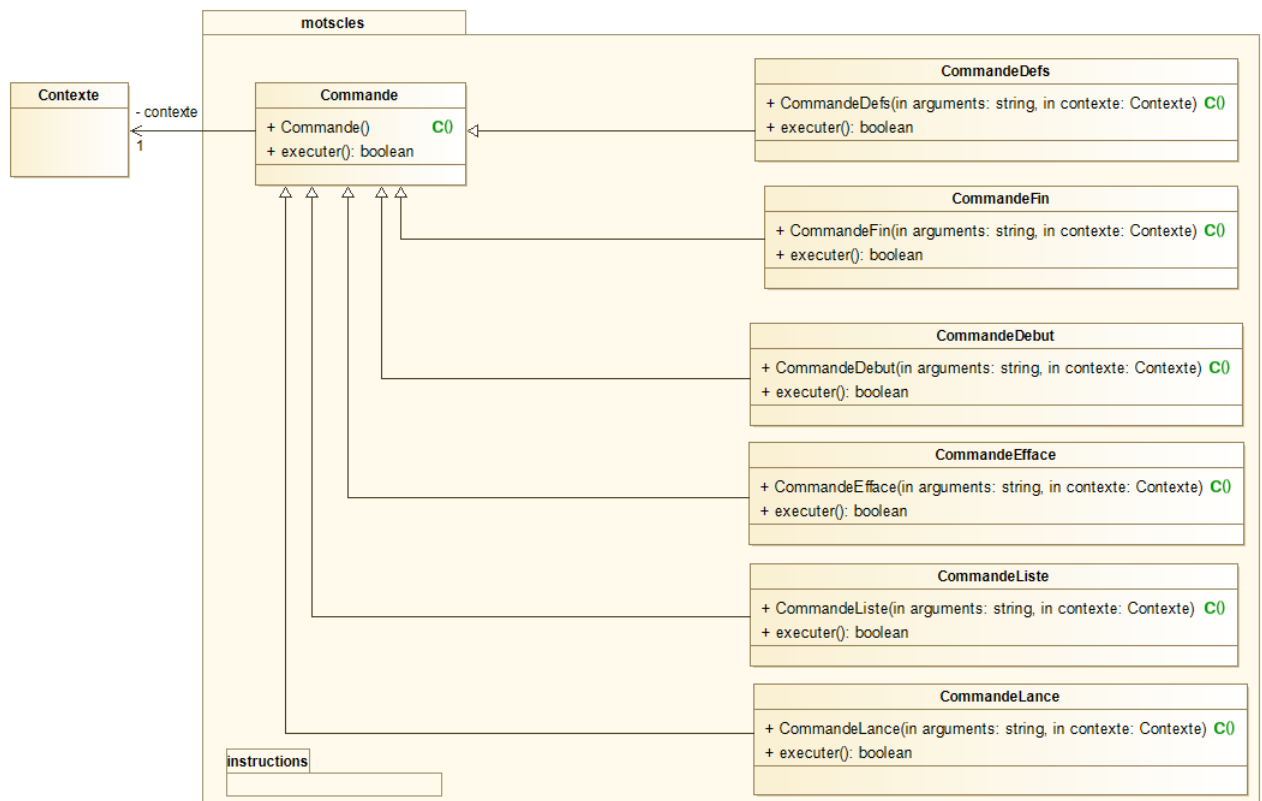
Comme pour les données, pas de changement de conception mais programmation de `ExpressionEntier`.

## 2.4 Paquetage interpreteurlir.programmes



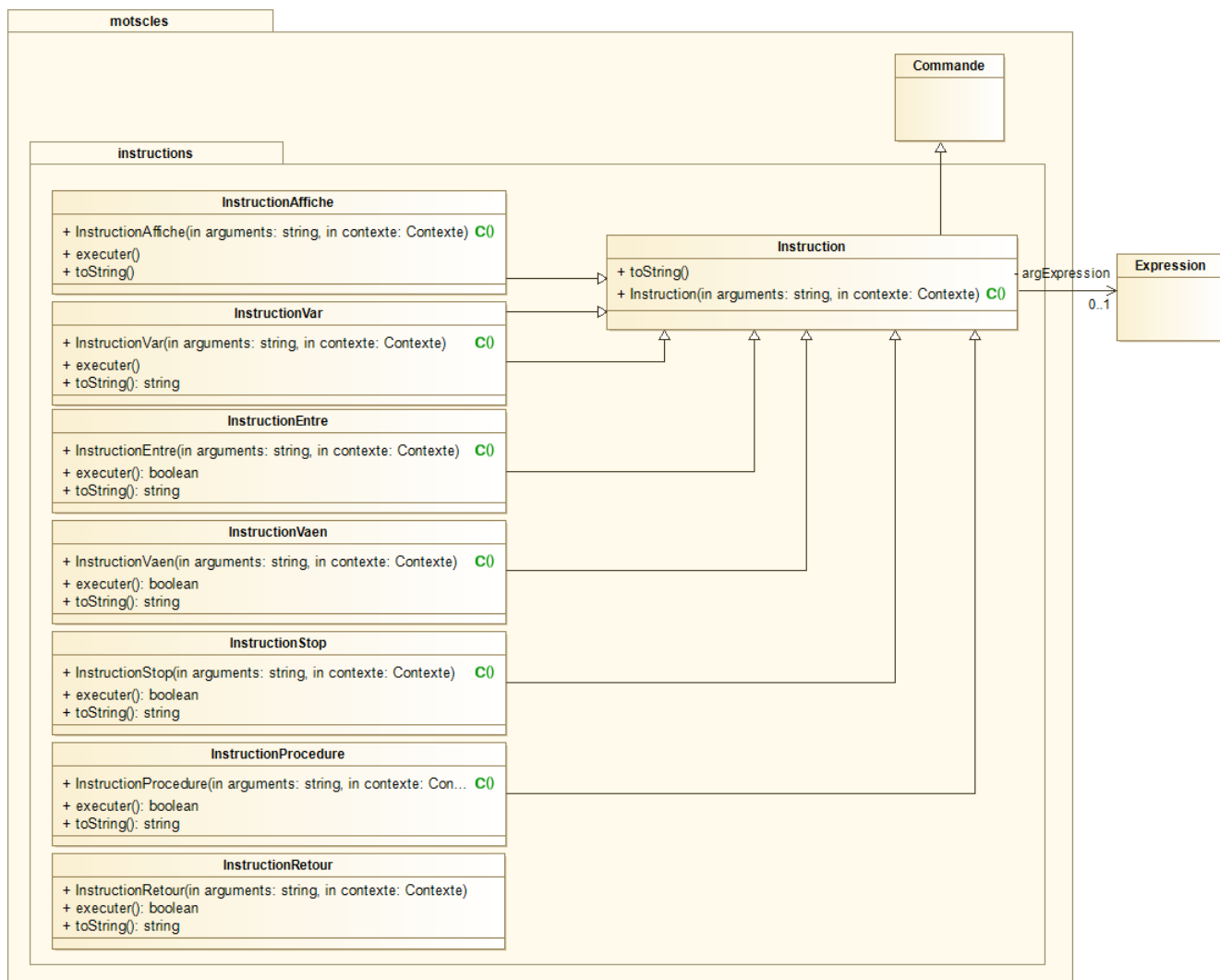
Premièrement la classe étiquette permet d'ordonner les lignes de codes. Le Programme contient des méthodes pour tous les comportement qu'il doit réaliser ce qui permet de les intégrer en interne ce qui rend leur usage plus simple pour les commandes et instructions. Seul la méthode vaen est absente de 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



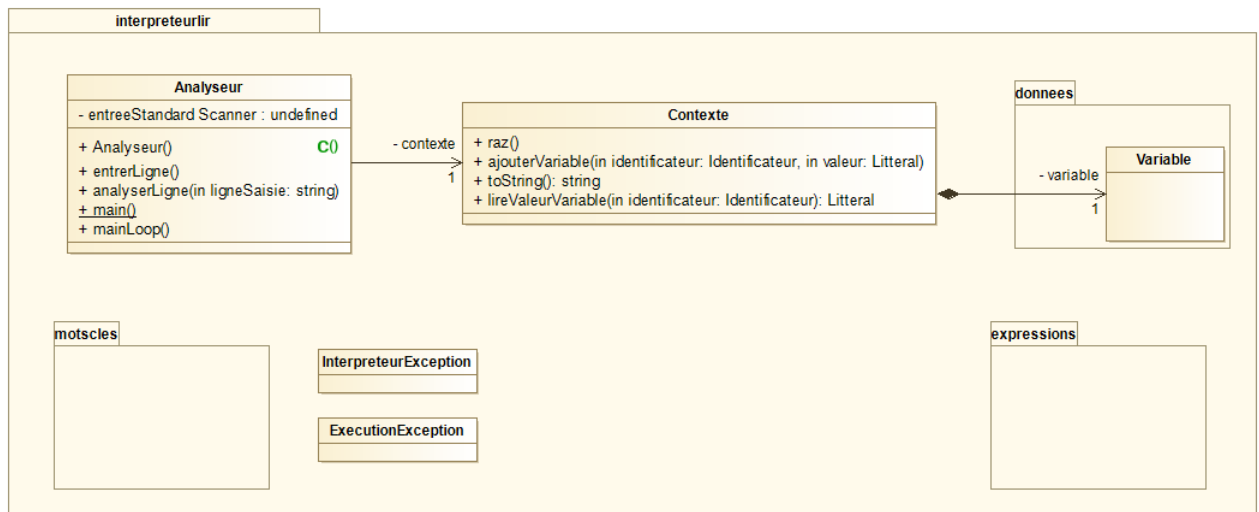
Les commandes à ajouter à cette itération ont été ajoutée à la conception en suivant le même principe de la dualité construction/exécution. Seul changement notable (non montré dans le diagramme car décidé pendant la programmation), l'ajout du programme nécessite que les commandes connaissent celui-ci. Après une longue réflexion nous avons choisis de le déclarer 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



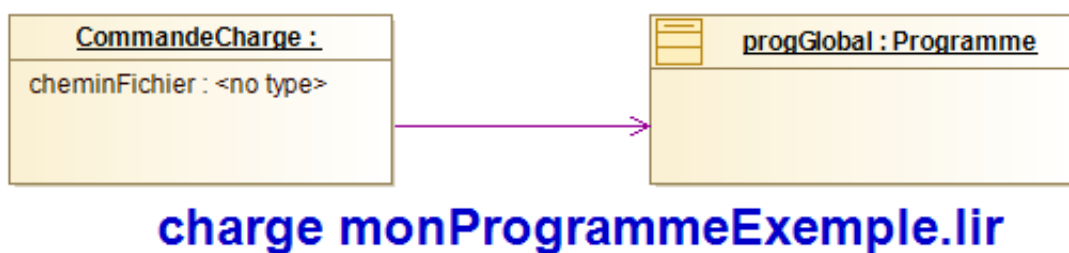
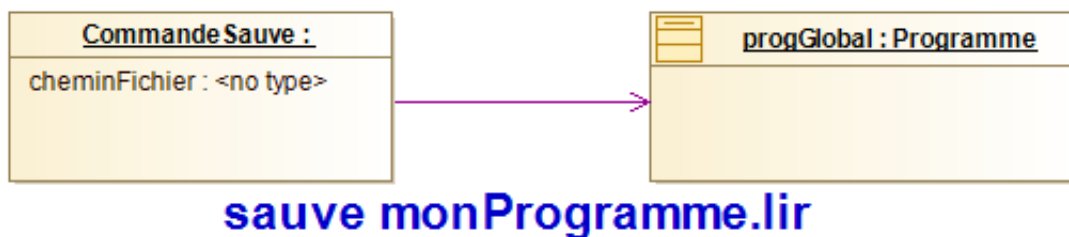
Ajout de l'exception `ExecutionException` lancée pour une erreur à l'exécution comme une division par 0 (contrairement à l'`InterpreterException` qui est lancée à la construction). Elle également affichée par l'`Analyseur`.

# Chapitre 3

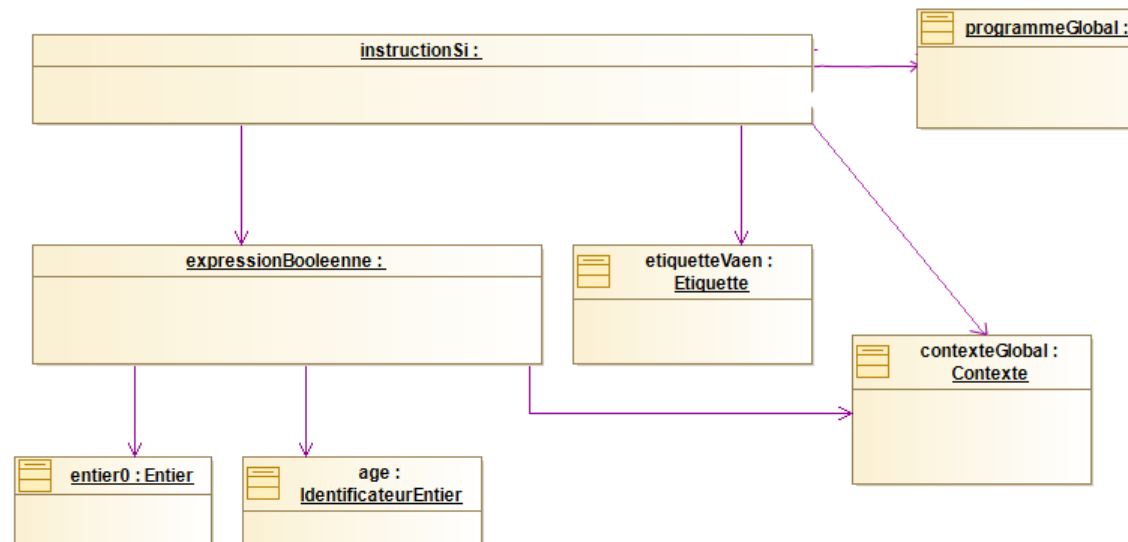
## Itération 3

L'itération 3 à 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



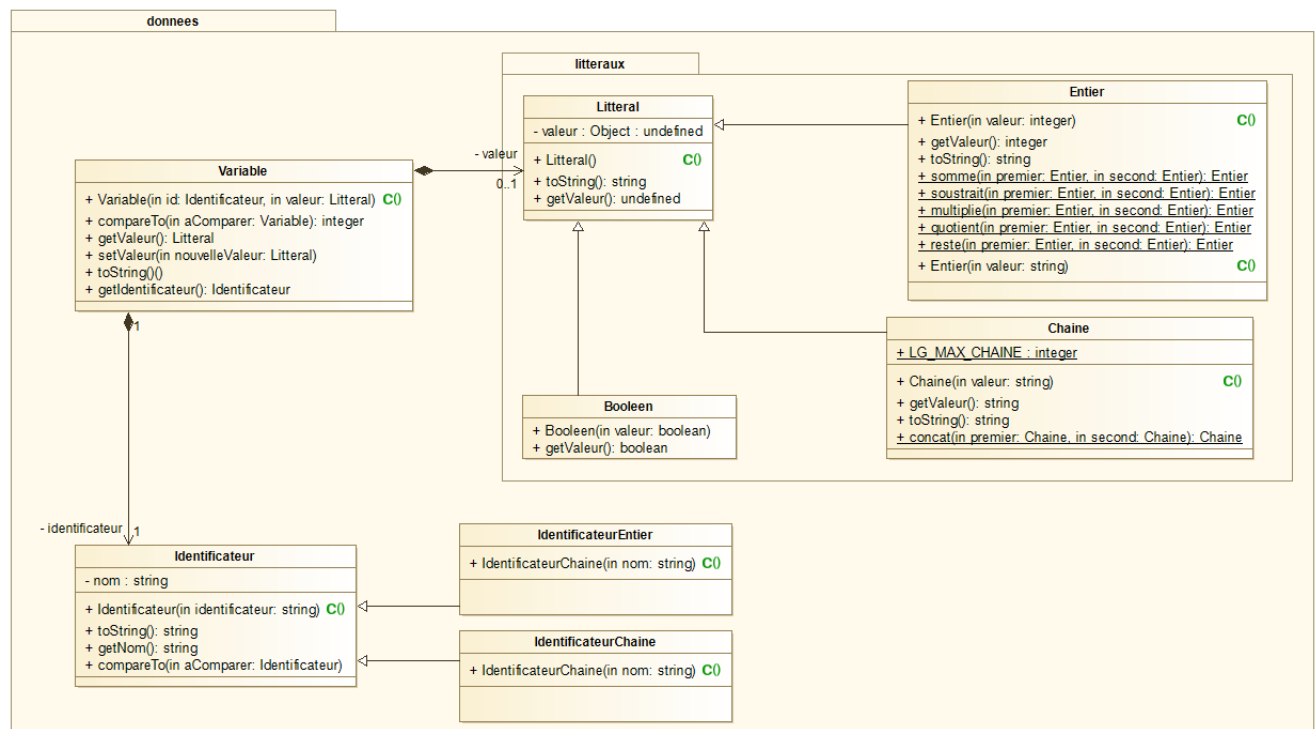
Les commandes sauve et charge sont liées au programme pour pouvoir charger ou récupérer des lignes de codes. Ces commandes connaissent une chaînes de texte correspondant au chemin du fichier.



**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.

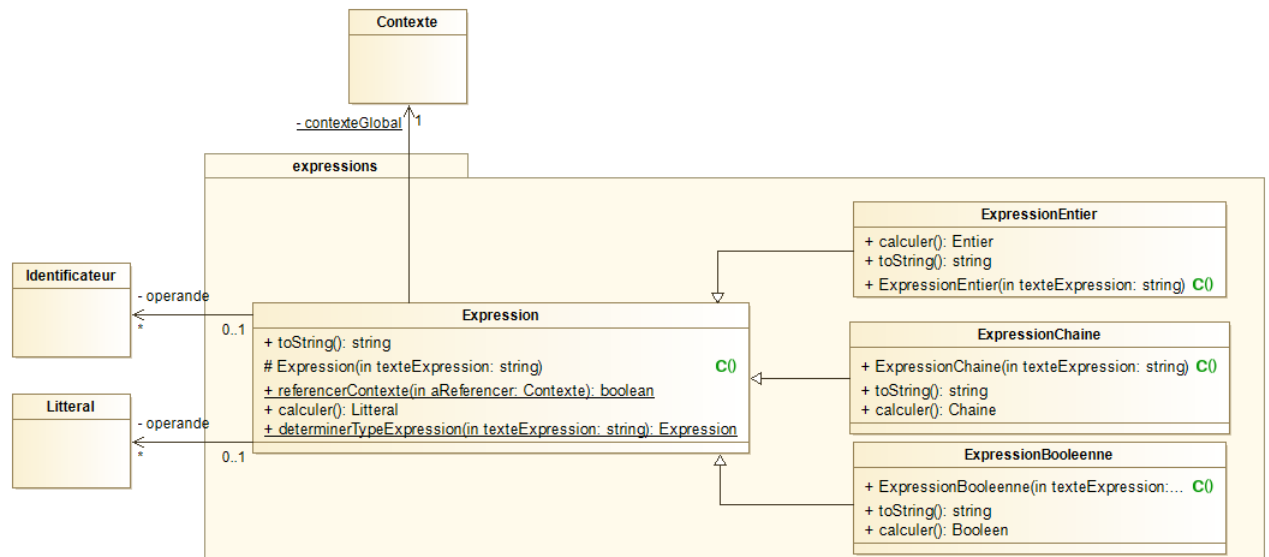
## 3.2 Paquetage interpreteurlir.donnees(.litteraux)





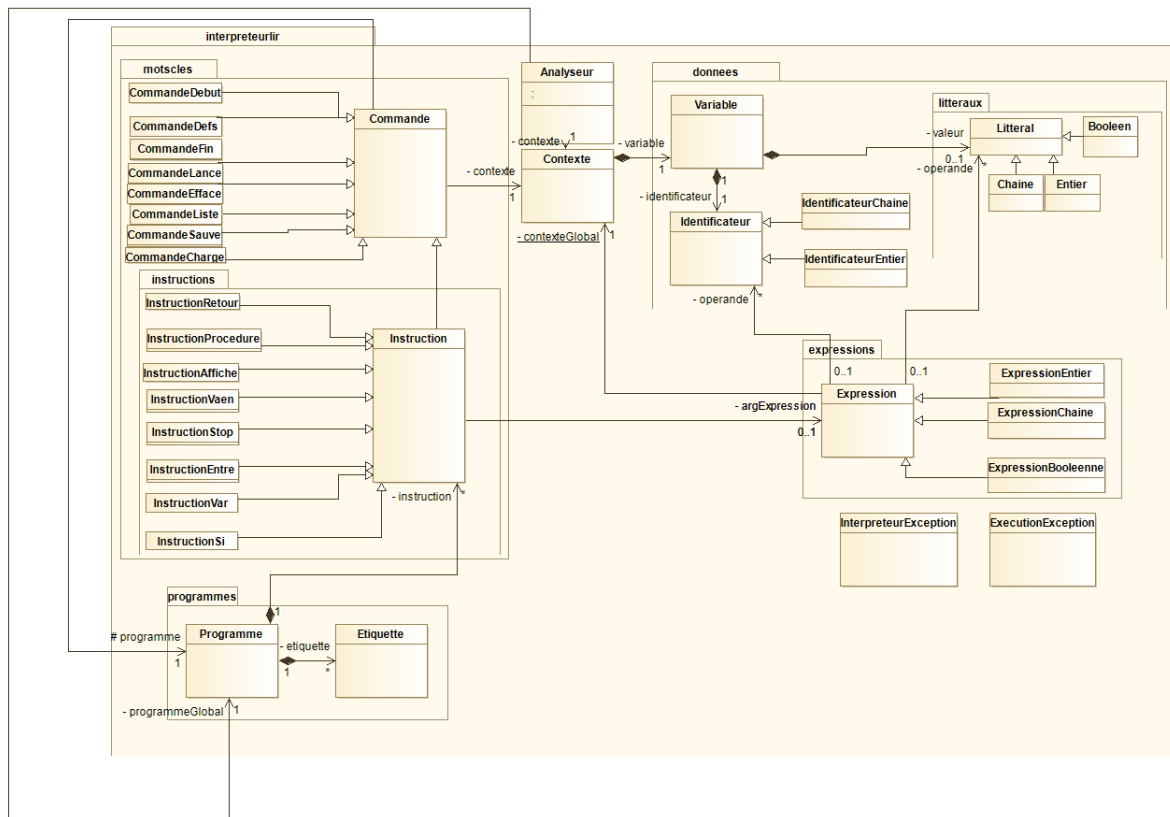
Le type booléen hérite de Litteral pour garder la logique de Litteral pouvant référencer chaque type de valeur du programme.

### 3.3 Paquetage interpreteurlir.expressions



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

### 3.4 Diagramme de classes général



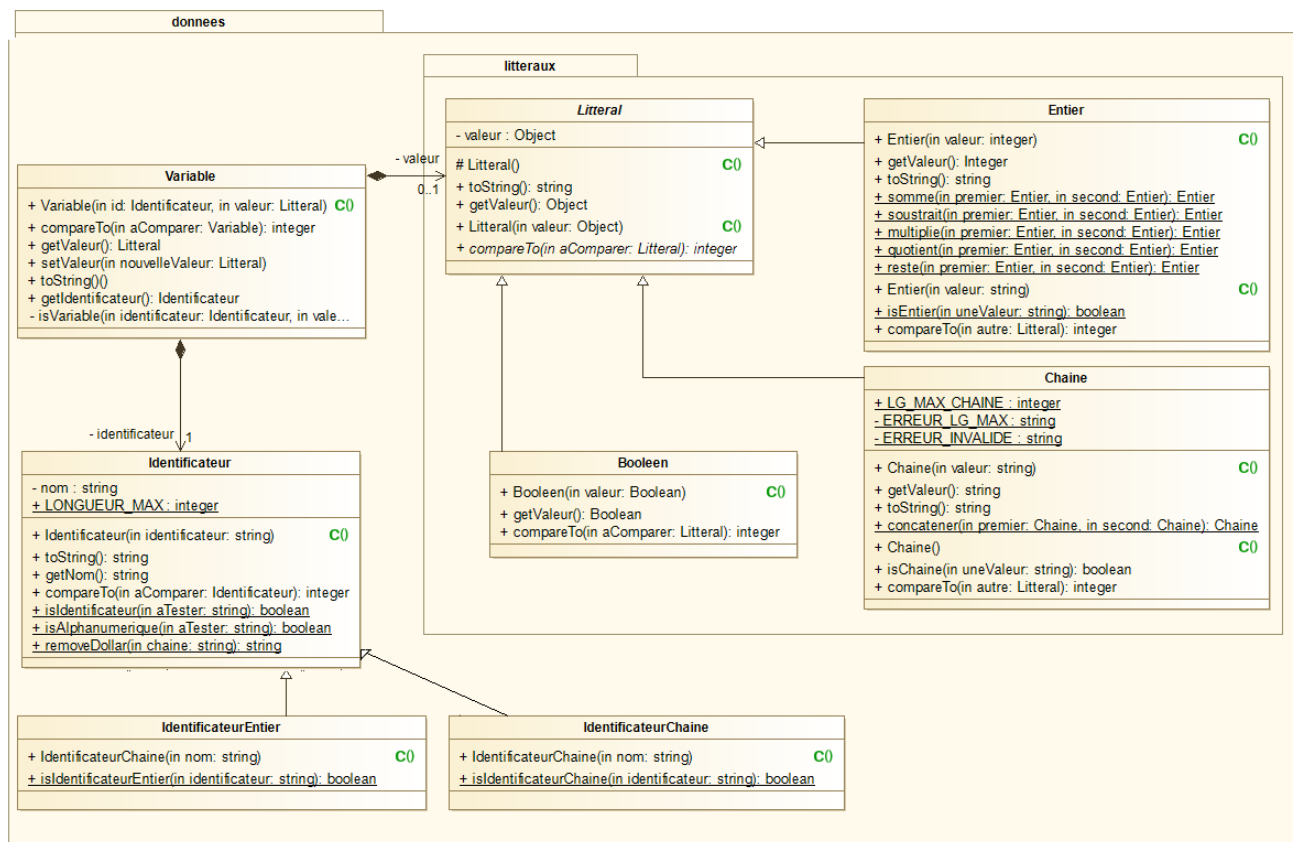
Les commandes sauve et charge ont été ajoutés à la conception mais sont similaires aux autres commandes. Pareil pour l'instruction si vaen. Ce diagramme général permet de voir l'ensemble de la conception pour ce qui est des associations et généralisation des classes.

# Chapitre 4

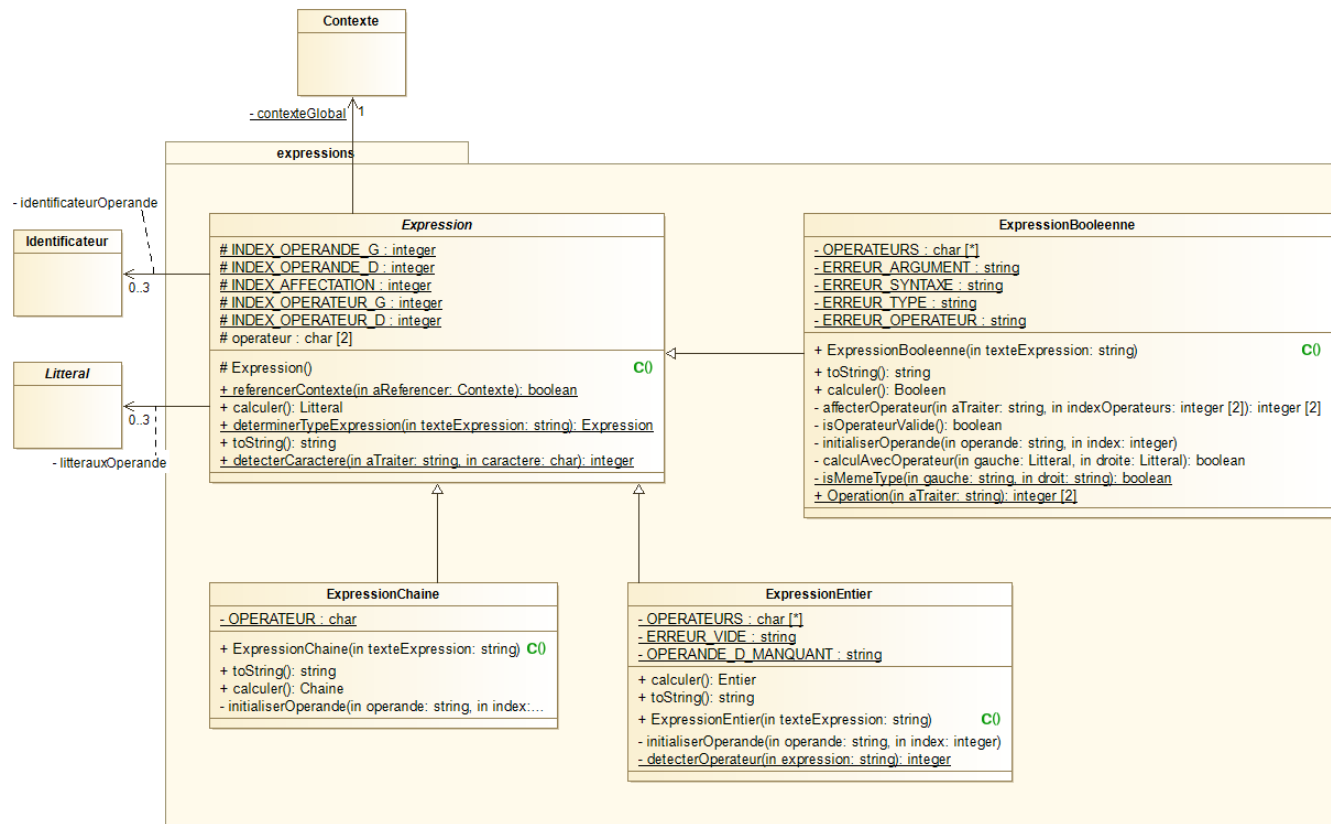
## Projet final

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 le diagrammes (méthodes privées par exemple).

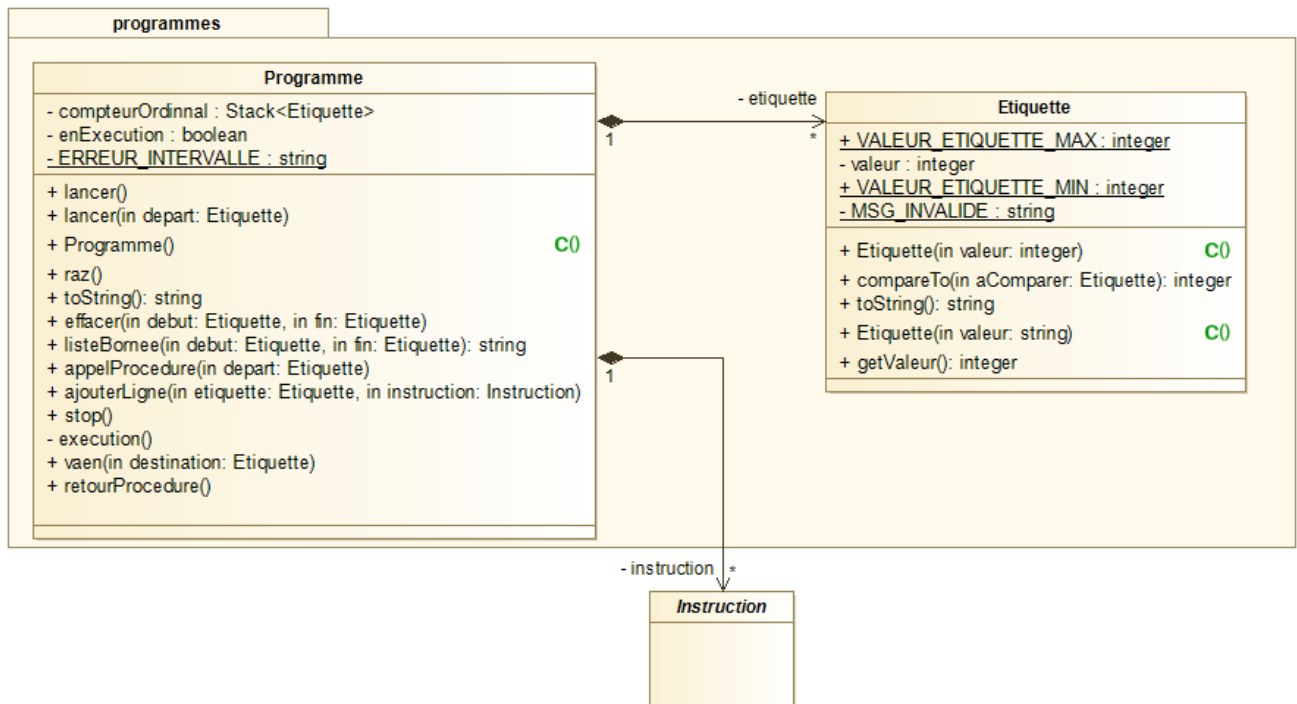
### 4.1 Paquetage `interpreteurlr.donnees(.litteraux)`



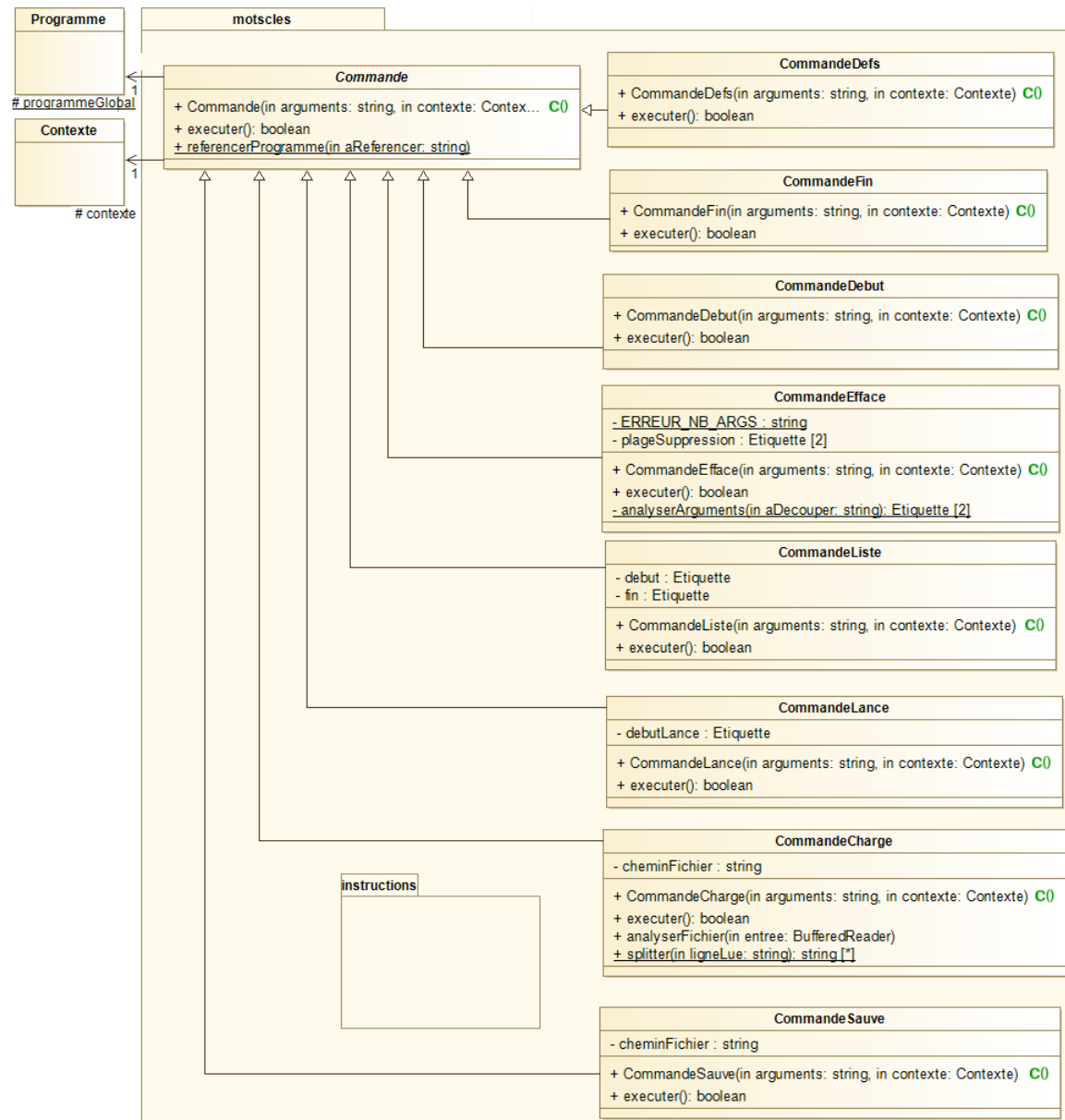
## 4.2 Paquetage interpreteurlir.expressions



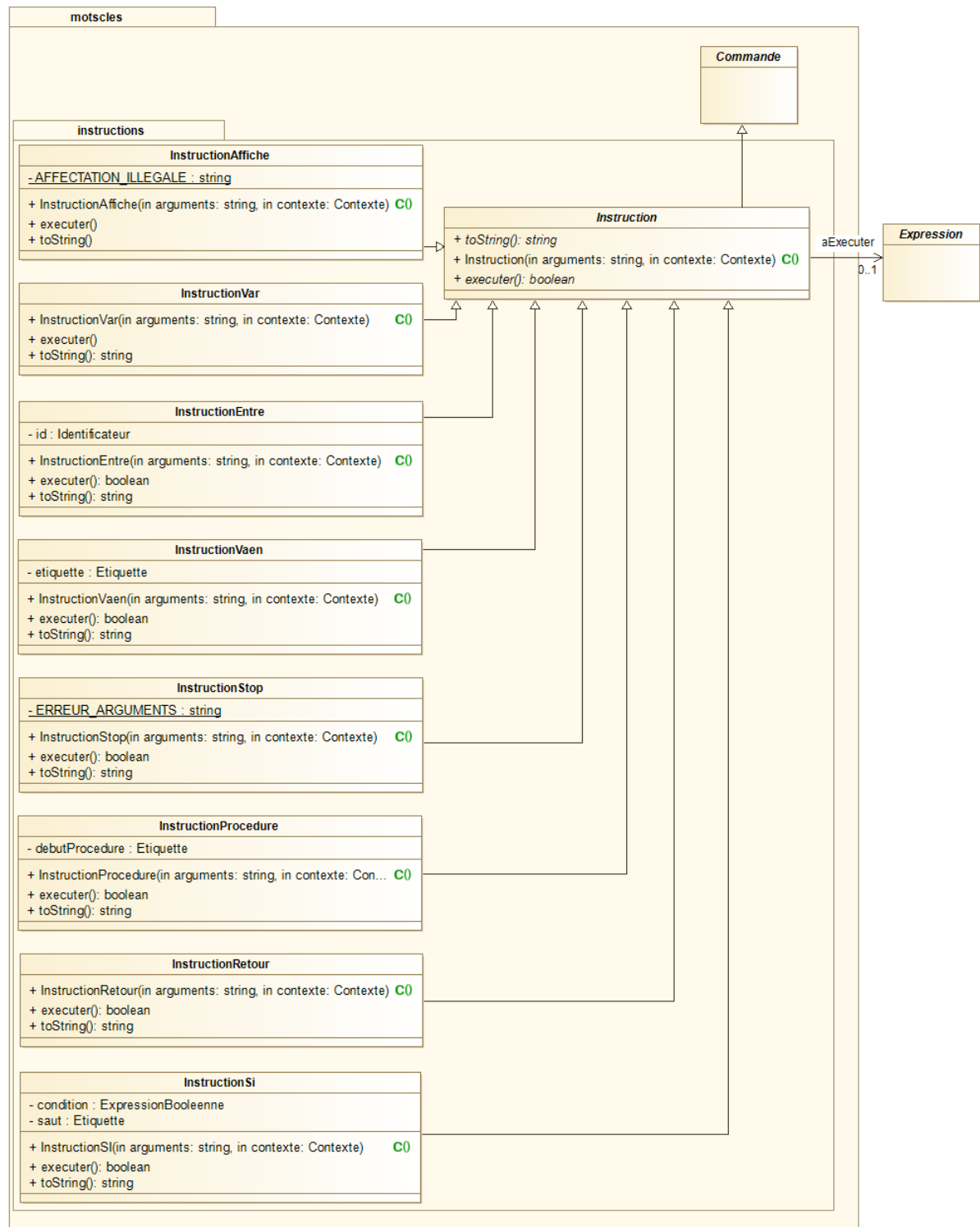
## 4.3 Paquetage interpreteur.lir.programmes



## 4.4 Paquetage interpreteurlir.motscles



## 4.5 Paquetage interpreteurlir.motscles.instructions



## 4.6 Paquetage interpreteurlir

