Projet Génie Logiciel

Gabriel Soria	**Meriem Lyna Safar Remali**	**Julian Gomez**
Camille Kasprzak	**Charles Breton**	**Arthur Lenne**
Alex Soubeyrand	**Maël Veyrat**	

Documentation de conception

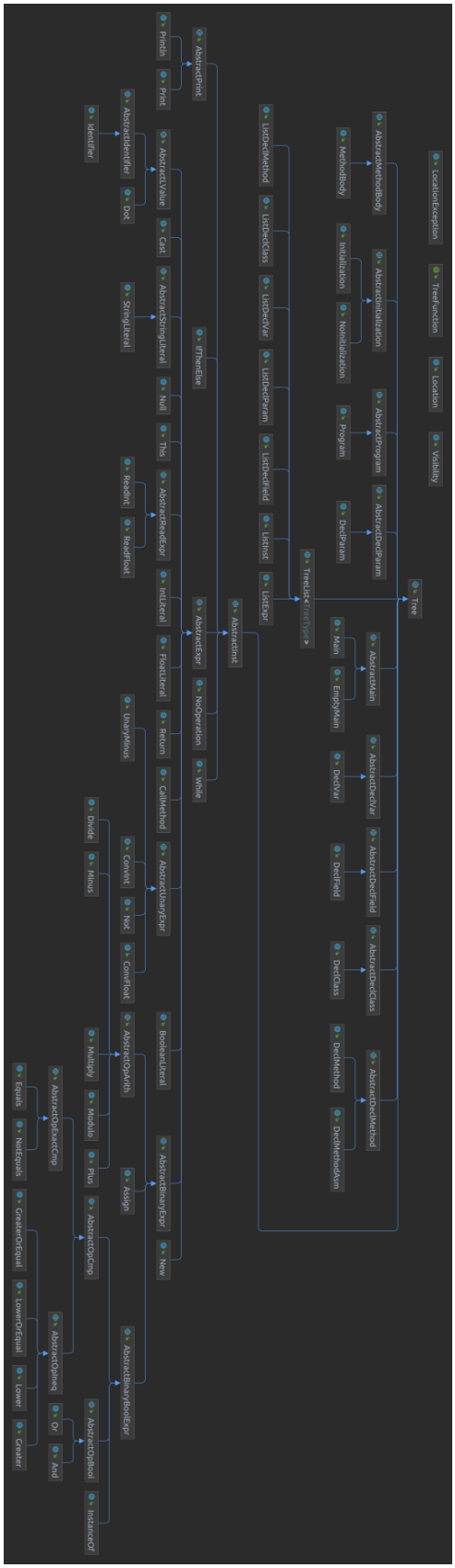

Figure 1: Diagramme de classes.

Dans cette partie de la documentation, nous présentons les modifications complexes apportées à l'architecture de base du compilateur. Ces ajouts peuvent être des fonctions, des classes ou des implémentations jugées importantes à la compréhension globale du projet.

Fonctions

Fonctions codeExp

```
protected void codeExp(DecacCompiler compiler, int registreNb);
```

Nous avons créé des méthodes codeExp pour gérer l'allocation des registres pour différentes opérations telles que ADD, SUB, MUL, etc. Ces méthodes permettent à chaque instruction parent de réinitialiser le contexte des registres utilisés (réinitialisés à 2), ce qui permet de les réutiliser à nouveau et de connaître ceux qui sont déjà utilisés, évitant ainsi d'écraser accidentellement une valeur dont nous avions besoin.

Dans le cas des opérations arithméthiques, la signature suivante est utilisée :

```
protected void codeExp(DecacCompiler compiler, int registreNb, boolean value, Label e)
```

Elle contient deux informations supplémentaires.

`value` correspond à la valeur attendue par la condition, ce qui permet d'enchaîner les `!(<expr>)`.

`e` est le label auquel sauter si le résultat ne correspond pas à `value`.

Fonctions codeGenCond

La fonction codeGenCond sert à générer le code d'une condition.

```
protected void codeGenCond(DecacCompiler compiler, boolean value, Label e)
```

Elle prend en paramètres le compilateur, un booléen (pour la gestion de la négation) et un label (représentant le label du ‹ else ›).

Classes

AbstractBinaryBoolExpr.java

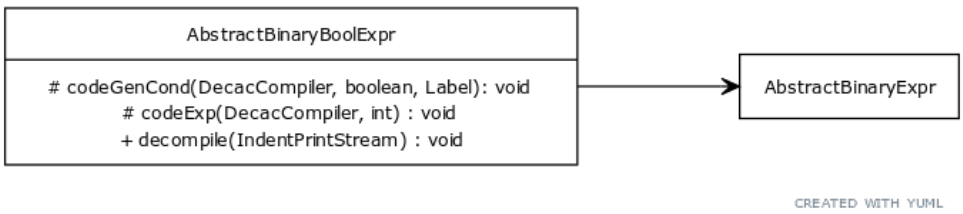

Figure 2: Diagramme UML simplifié de la classe AbstractBinaryBoolExpr.

Cette classe sert à regrouper les fonctions de comparaison et d'opérations booléennes. Elle est utilisée pour déclarer codeExp et codeGenCond une seule fois, étant donné que chacun de ces cas est généralisable.

Cependant, il existe d'autres cas spécifiques tels que les assigns de booléens, les boolean literals et les opérations « not », qui ne sont pas généralisables en raison de leur position dans l'architecture. Cette classe hérite de la classe AbstractBinaryExpr.

GestionnaireErreur.java

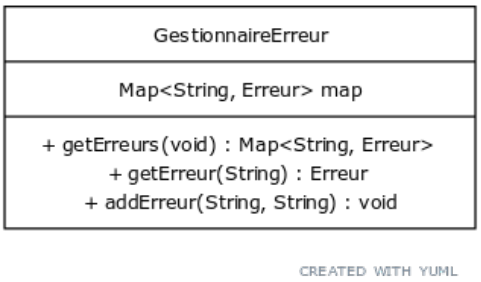

Figure 3: Diagramme UML de la classe GestionnaireErreur.

La classe `GestionnaireErreur.java`, comme son nom l'indique, permet de gérer les erreurs dans notre projet. Cette classe implémente une `Map` contenant en clé le nom de l'erreur et en valeur le message d'erreur retourné par le compilateur.

Elle est utilisée lors de la génération de code. Seules les erreurs qui ont été utilisées par les instructions figureront dans le code assembleur, les autres ne seront pas générées afin d'optimiser la taille du fichier assembleur.

Lors de la génération de code, chaque instruction peut faire appel au label de l'erreur (qui a été généralisé) en y accédant grâce à cette classe présente dans le compilateur.

ListDeclField.java, AbstractDeclField.java & DeclField.java

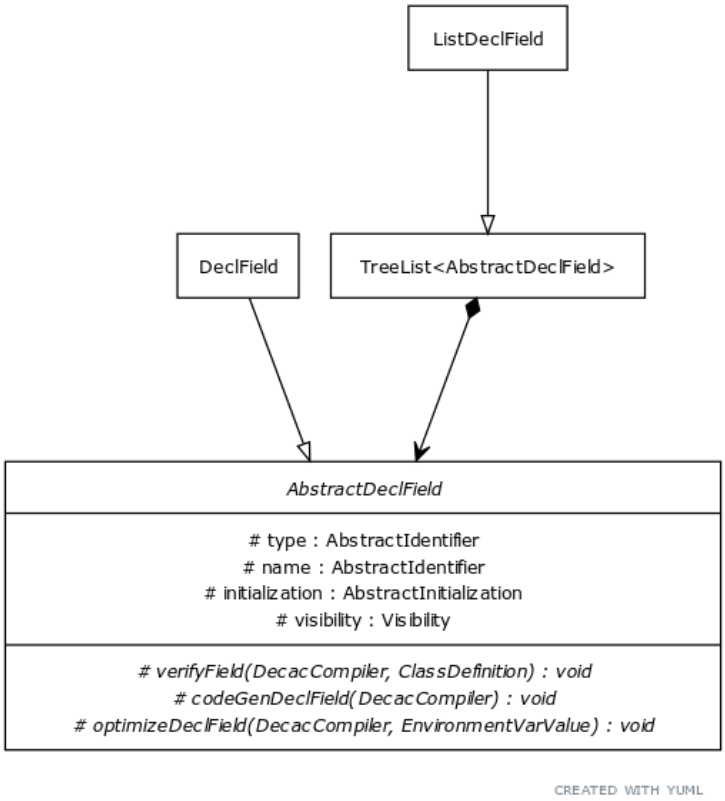

Figure 4: Diagramme UML simpliste des classes permettants la déclaration d'attributs.

Les trois classes ci-dessus permettent la vérification et la génération de code associé à la déclaration d'attributs de classes.

ListDeclMethod.java, AbstractDeclMethod.java & DeclMethod.java

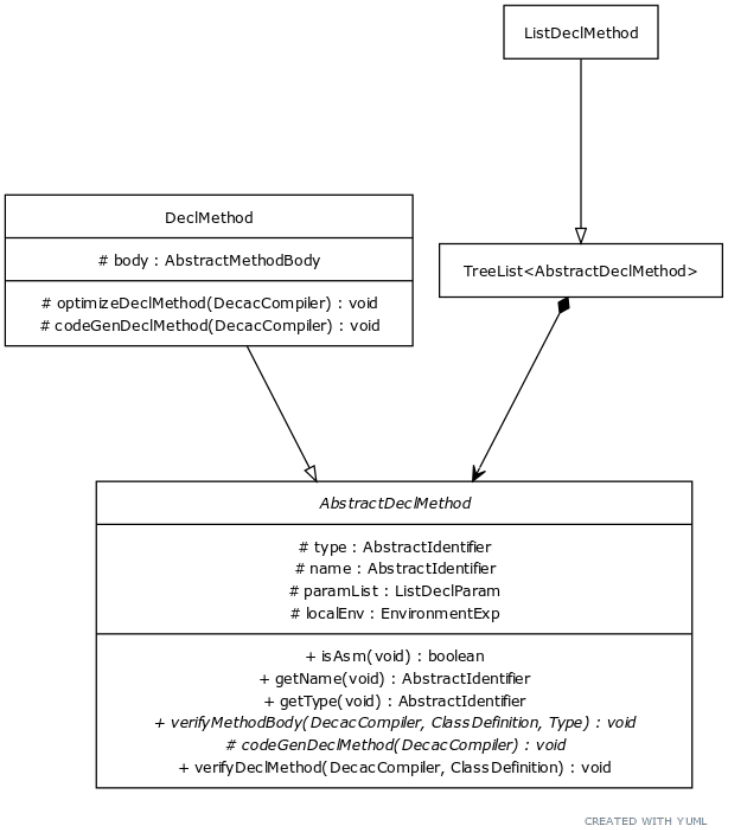

Figure 5: Diagramme UML simpliste des classes permettants la déclaration de méthodes.

Les trois classes ci-dessus permettent la vérification et la génération de code associé à la création de nouvelles méthodes. La méthode `isAsm()` de la classe abstraite `AbstractDeclMethod` permet de différencier `asm` des autres méthodes car elles nécessitent un traitement différent. La méthode `verifyMethodBody` appelle la méthode de vérification du corps de la méthode, cette classe est détaillée plus bas.

DeclMethodAsm.java

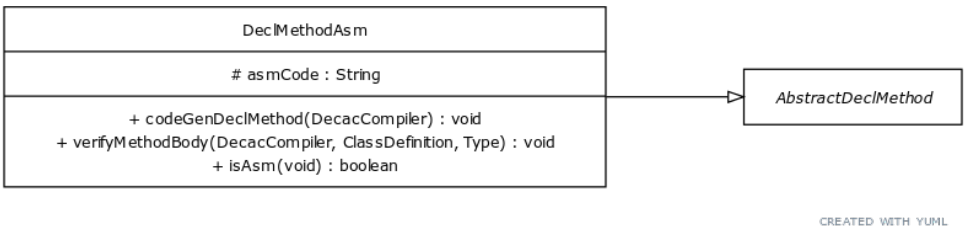

Figure 6: Diagramme UML simpliste des classes permettants la déclaration de méthodes asm.

La classe DeclMethodAsm permet de gérer l'utilisation de la méthode `asm`. Sa vérification ne contient aucun code, c'est à la charge de l'utilisateur d'écrire un programme asm respectant les conventions. De même, la génération de code copie le code de l'utilisateur directement dans le ficher généré, en ajoutant le label de la méthode et en vérifiant à la fin si la fonction conteanait bien un `return`.

AbstractMethodBody.java & MethodBody.java

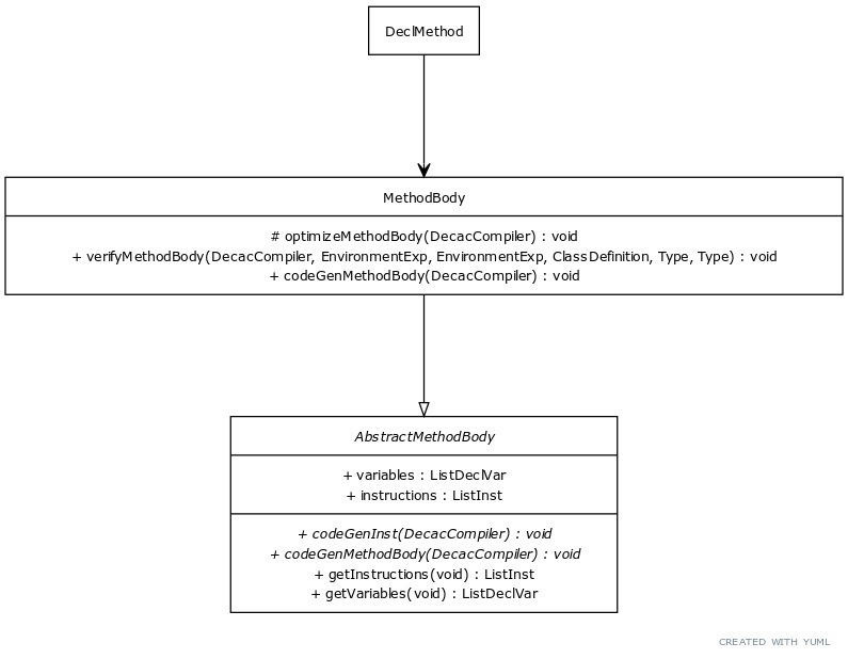

Figure 7: Diagramme UML simpliste des classes associées au corps des méthodes.

Les deux classes ci-dessus permettent la vérification et la génération du code pour le corps des méthodes. `verifyMethodBody` appelle à son tour les méthodes de vérifications de ses paramètres grâce aux deux environnements de variables passées en paramètres correspondants à l'environnement des variables déjà connues ainsi que celui de ses paramètres. Cette dernière appelle également la méthode de vérification des instructions qui la composent.

ListDeclParam.java, AbstractDeclParam.java & DeclParam.java

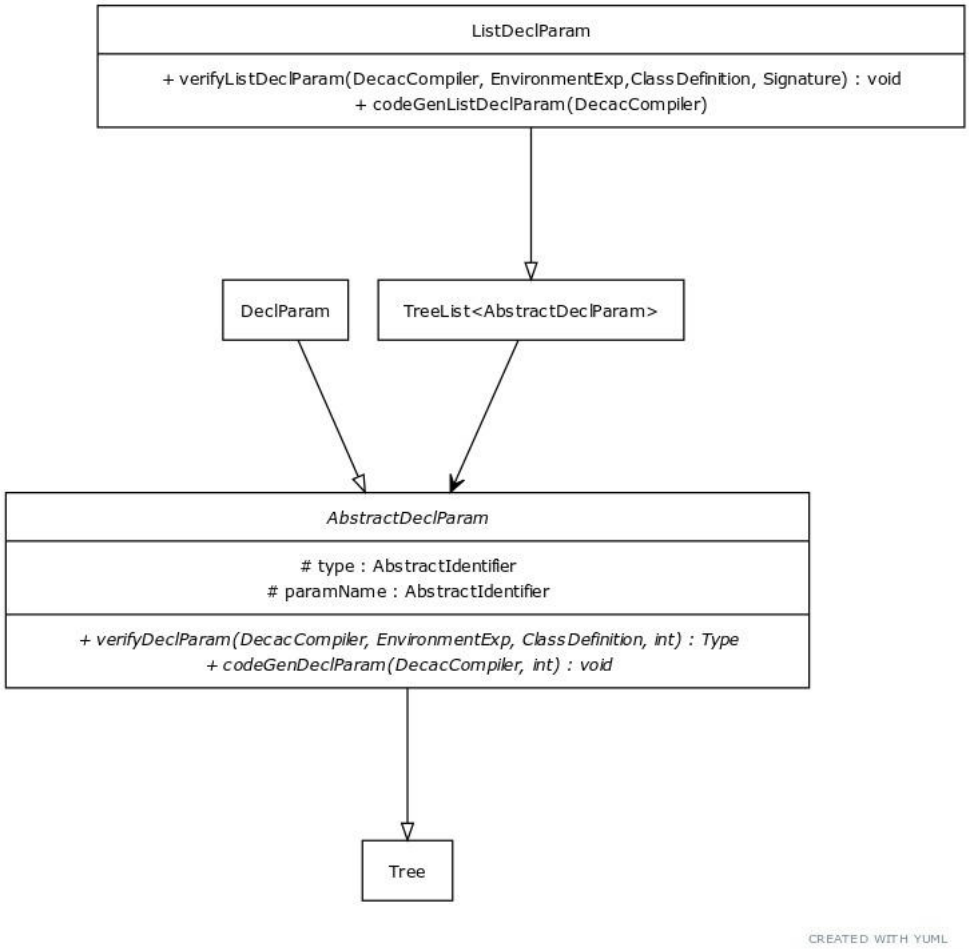

Figure 8: Diagramme UML simpliste des classes pour la déclaration de paramètres.

Les classes ci-dessus permettent la vérification et la génération de code associé aux paramètres de méthodes. La vérification de paramètre vérifie le type, le nom, et déclare le paramètre dans l'environnement local. Le type retourné est le type du paramètre.

CallMethod.java

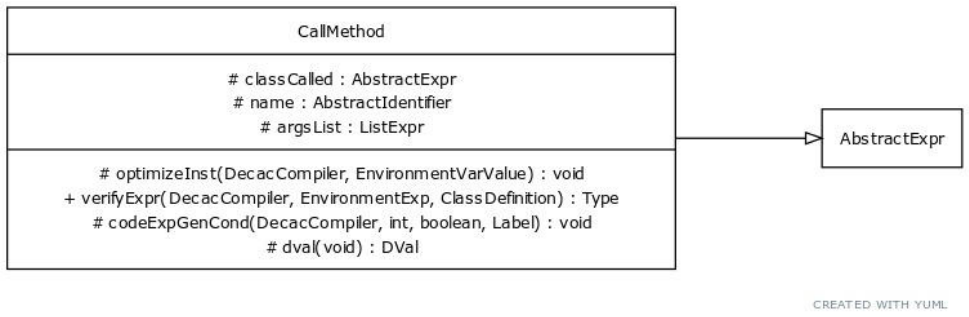

Figure 9: Diagramme UML simpliste de la classe CallMethod.

CallMethod sert à la vérification et à la génération du code associé à l'appel de méthode dans le code Deca. La génération de code gère le passage des paramètres à la fonction ainsi que l'ajout et la suppression dans SP, et stocke le résultat de la fonction dans le registre donné lors de l'appel aux fonctions de génération de code asm.

Return.java

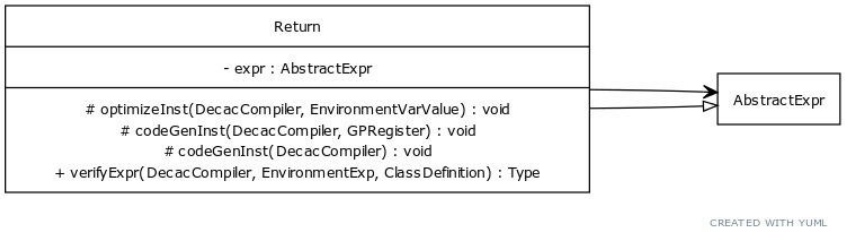

Figure 10: Diagramme UML simpliste de la classe Return.

La classe `Return` sert à la vérification et à la génération de code associé au retour de méthodes. Elle vérifie notamment que le retour n'est pas de type `void` et qu'il correspond au type de retour déclaré lors de la déclaration de la méthode.

This.java

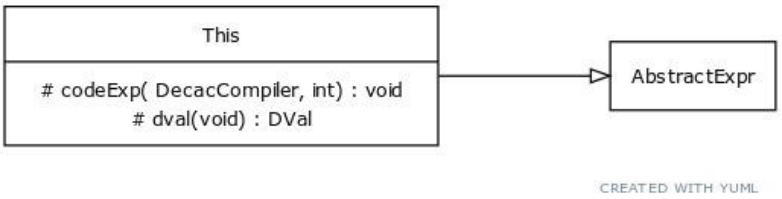

Figure 11: Diagramme UML simpliste de la classe This.

La classe `This` permet de vérifier et générer le code associé au `this` dans le code Deca. La vérification sert à vérifier que l'on appelle bien `this` dans une classe et permet de récupérer la classe appelée. La génération de code stocke la référence de la classe associée dans le registre donné.

New.java

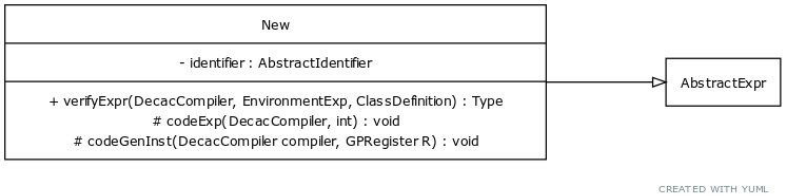

Figure 12: Diagramme UML simpliste de la classe New.

La classe `New` permet de vérifier et de générer le code associé à la création d'une nouvelle instance d'un objet, elle permet de vérifier que l'identifier appelé est bien une classe et gère également l'appel au constructeur de la classe instanciée.

Null.java

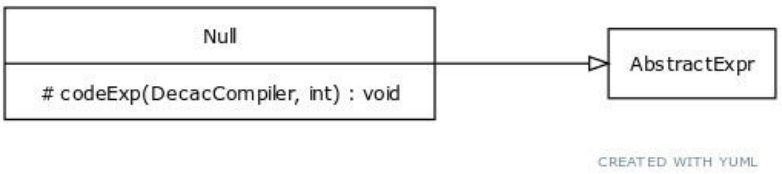

Figure 13: Diagramme UML simpliste de la classe Null.

La classe `Null` permet de gérer la valeur `null`, ce qui est la valeur par défaut des objets lorsqu'ils sont utilisés en attributs de classe.

Dot.java

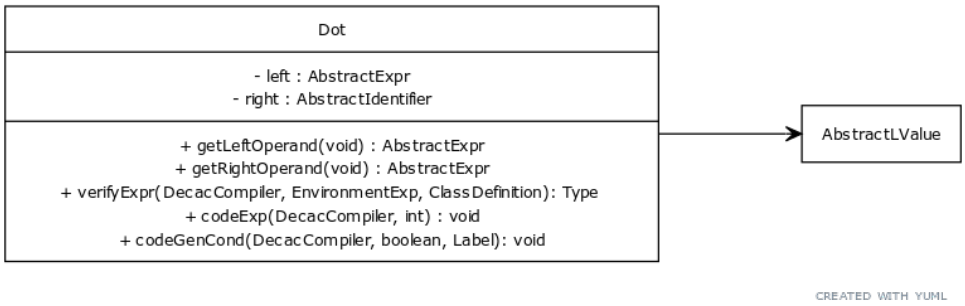

Figure 14: Diagramme UML simpliste de la classe MethodBody.

La classe `Dot` permet de gérer les accès aux attributs ou fonctions d'une classe via son identifier. Elle vérifie que l'appel est valide en regardant la visibilité du champs appelé, le déréférencement nul, etc.